Simulating and modeling the Truck Platooning System following principles of distributed and parallel architecture

Nhat Lam Nguyen
Fachhochschule Dortmund
Master Embedded System Enginnering
Dortmund, Germany
nhat.nguyen003@stud.fh-dortmund.de

Nhat Quang Nguyen
Fachhochschule Dortmund
Master Embedded System Enginnering
Dortmund, Germany
nhat.nguyen001@stud.fh-dortmund.de

Shirin Babaeikouros
Fachhochschule Dortmund
Dortmund, Germany
Master Embedded System Enginnering
shirin.babaeikouros001@stud.fh-dortmund.de

Hadis Mohammadi
Fachhochschule Dortmund
Dortmund, Germany
Master Embedded System Enginnering
hadis.mohammadi005@stud.fh-dortmund.de

***Abstract*—This paper presents a comprehensive study on the development of a truck platooning system in the context of distributed and parallel systems. The objective of this research is to address the challenges in the transportation industry by leveraging advanced networking and computing technologies. The main outcomes of this study include the design and evaluation of the system architecture, algorithms, and protocols for efficient communication and coordination among platooning trucks. The implementation of the system demonstrates its feasibility and effectiveness in improving fuel efficiency, reducing traffic congestion, and enhancing road safety. The results of the evaluation highlight the potential of the truck platooning system to revolutionize the logistics industry. This work provides valuable insights and serves as a foundation for future research in the field of distributed and parallel systems for intelligent transportation systems.**

1. Introduction [Lam]

The transportation industry plays a critical role in global economies, and the need for efficient and sustainable logistics solutions has become increasingly important. Truck platooning, a concept where multiple trucks travel closely together in a coordinated manner, has emerged as a promising solution to improve fuel efficiency, reduce traffic congestion, and enhance road safety. In this context, this paper presents a comprehensive study on the development of a truck platooning system, leveraging the principles of distributed and parallel systems.

The primary motivation behind this project is to address the challenges faced by the transportation industry, such as the increasing demand for goods transportation, environmental concerns, and the need for optimized resource utilization. By developing a truck platooning system, we aim to harness the power of advanced networking and computing technologies to achieve these goals.

To achieve an efficient and reliable truck platooning system, various aspects need to be considered, including the system architecture, communication protocols, coordination algorithms, and vehicle communication. This paper focuses on providing a comprehensive overview of these key components, from the high-level network architecture down to the algorithm and protocol level.

By developing a truck platooning system, we aim to demonstrate its feasibility and effectiveness in real-world scenarios. This research serves as a foundation for future advancements in the field of distributed and parallel systems for intelligent transportation systems.

2. Concept [Lam]

The truck platooning system presented in this paper utilizes a socket protocol for communication and coordination among the platooning trucks. This section provides a detailed overview of the system, focusing on the network architecture, node architecture, and the algorithm employed for control.

1) Network Architecture: The network architecture of the truck platooning system is designed around the utilization of socket protocol for vehicle-to-vehicle (V2V) communication. This protocol enables real-time data exchange between the trucks in the platoon. Each truck acts as a socket client, sending and receiving information to and from other trucks in the platoon. The network architecture ensures reliable and low-latency communication, facilitating coordination and synchronization among the platooning trucks.
2) Node Architecture: The node architecture encompasses the components and functionalities of each

truck within the platooning system. Each truck is equipped with sensors, actuators, and computing resources necessary for autonomous driving and communication. The node architecture includes the integration of socket communication modules, perception systems for environment sensing, and a PID controller for control actions. The PID controller algorithm is responsible for adjusting the speed and position of each truck based on the feedback received from the sensors and the communication with other trucks in the platoon.

3) Control Algorithm: The control algorithm employed in the truck platooning system is the PID (Proportional-Integral-Derivative) controller. The PID controller uses feedback from sensors, such as distance sensors and speed sensors, to dynamically adjust the speed and position of each truck in the platoon. The PID controller calculates the appropriate control signals based on the error between the desired position and speed and the actual position and speed of the truck. By continuously adjusting these control signals, the PID controller ensures that each truck maintains a safe and optimal distance while traveling in a platoon.

By utilizing the socket protocol for communication and the PID controller algorithm for control, the truck platooning system aims to achieve improved fuel efficiency, reduced traffic congestion, and enhanced road safety. The next section will present the evaluation of the implemented system to assess its performance and effectiveness in real-world scenarios.

3. Implementation

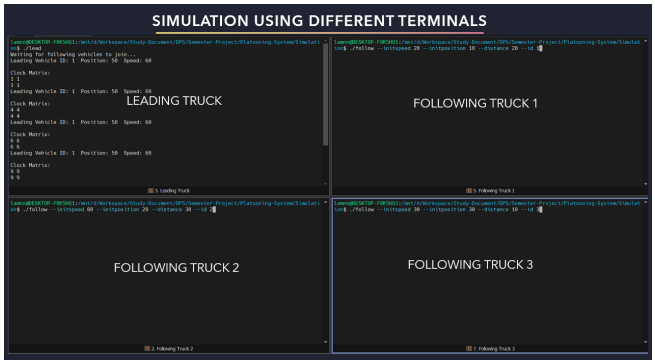

Figure 1. Simulation program running on different terminals

3.1. Requirement [Lam]

The truck platooning system has specific requirements and objectives aimed at improving fuel efficiency, reducing traffic congestion, and enhancing road safety. To achieve fuel efficiency improvement, the system should enable trucks to form and maintain platoons with a minimum distance between them, optimize fuel consumption through efficient platooning strategies and control algorithms, and evaluate the resulting fuel efficiency improvements. In terms of traffic congestion reduction, the system should provide real-time traffic information to trucks, facilitate efficient coordination and communication within platoons, and assess the impact of the system on reducing traffic congestion. For road safety enhancement, the system should have mechanisms for detecting and responding to obstacles or hazards on the road, allowing trucks to exit platoons safely when necessary, and ensuring secure and reliable communication with the Traffic Management System to prevent unauthorized access or interference. The evaluation of the truck platooning system should involve analyzing metrics and conducting simulations, experiments, or data analysis to assess its performance, effectiveness, and impact on fuel efficiency, traffic congestion, and road safety.

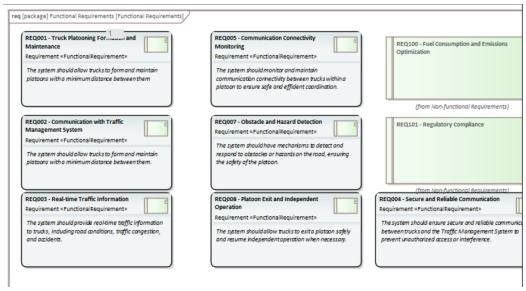

Figure 2. Requirement Diagram

3.2. Main Use Cases [Lam]

Before proceeding on to more specifics, requirements should be first defined. Determining system capabilities, formulating functional and non-functional needs, and comprehending the perspectives of several stakeholders are all necessary steps in the analysis of requirements for emergency vehicle handling in smart traffic systems. The following key requirements need to be listed and analyzed:

3.3. Comparison Communication Protocol for Truck Platooning [Shirin]

In comparison to DSRC and C-V2X, socket communication brings unique advantages to the table, particularly in the context of running simulations on virtual environments for truck platooning systems. Socket communication is a versatile and widely-used method for communication between devices or processes over a network. It operates on the transport layer of the OSI model and can be implemented using various protocols such as TCP (Transmission Control Protocol) or UDP (User Datagram Protocol). One notable advantage of socket communication is its flexibility in terms of deployment and integration. Unlike DSRC and C-V2X, which are specific communication protocols designed for vehicular communication, sockets can be implemented in

various programming languages and environments. This versatility allows developers to choose the language that best suits their simulation environment, making it easier to integrate with existing simulation frameworks and tools.

Moreover, socket communication can be tailored to specific simulation requirements, enabling seamless communication between virtual vehicles within the platooning system. The ability to customize communication parameters, such as latency and reliability, makes it a powerful tool for simulating different scenarios and testing the robustness of the platooning algorithms.

• **Versatility and Flexibility**: Socket communication is programming language agnostic: Unlike DSRC and C-V2X, which are specific communication protocols designed for vehicular communication, socket communication is a general-purpose method that can be implemented in various programming languages. This versatility allows developers to choose the language that best suits their simulation environment, making it easier to integrate with existing simulation frameworks and tools.

• **Ease of Integration**: Sockets can be easily integrated into simulation frameworks: Since sockets are a widely-used and well-supported communication method, they can be seamlessly integrated into simulation frameworks for truck platooning systems. This ease of integration facilitates the development of realistic and dynamic virtual environments for testing and refining platooning algorithms.

• **Customization for Simulation Requirements**: Adjustable parameters for simulation scenarios: Socket communication provides the flexibility to customize communication parameters such as latency, packet loss, and reliability. This is particularly beneficial for running simulations that mimic various real-world scenarios, allowing researchers and developers to test the robustness of platooning algorithms under different conditions.

• **Independence from Dedicated Infrastructure**: No need for specialized vehicular communication hardware: Unlike DSRC and C-V2X, which rely on specific communication technologies and protocols, sockets can operate over standard network protocols like TCP/IP. This means that simulations using socket communication can be conducted in virtual environments without the need for specialized vehicular communication hardware. This independence streamlines the setup process for simulations.

• **Scalability and Network Simulation**: Suitable for large-scale deployments: Socket communication is well-suited for simulating large-scale platooning scenarios. It can handle communication between a higher number of vehicles within the virtual environment, making it an excellent choice for testing the scalability of platooning algorithms. Additionally, the ability to run simulations on standard network infrastructure enables the exploration of scenarios involving a significant number of vehicles.

• **Cost-Effective Simulation Setup**: Eliminates the need for dedicated communication infrastructure: DSRC and C-V2X might require dedicated communication hardware, which can be expensive to set up for simulation purposes. Socket communication, being software-based and leveraging existing network infrastructure, offers a more cost-effective solution for running simulations.

In summary, socket communication's versatility, ease of integration, customization options, independence from dedicated hardware, scalability, and cost-effectiveness make it a compelling choice for running simulations on virtual environments for truck platooning systems. These factors contribute to the effectiveness of socket communication in creating realistic and dynamic scenarios for testing and optimizing platooning algorithms.

3.4. V2V Communication using Socket Protocol [Lam]

V2V (Vehicle-to-Vehicle) communication is a critical component in facilitating efficient coordination and information exchange among vehicles within a truck platooning system. One widely adopted protocol for V2V communication is the Socket protocol. By utilizing Socket programming, vehicles can establish a reliable and secure communication channel to facilitate the seamless exchange of data and messages.

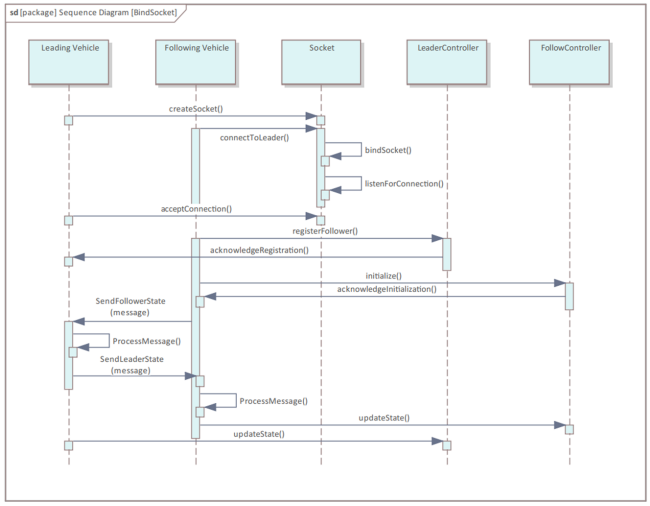

Figure 3. V2V communication using the Socket protocol

To implement V2V communication using the Socket protocol, the following steps are typically undertaken in the truck platooning system:

Firstly, each vehicle participating in the platooning system creates a Socket object, which serves as a communication endpoint. The Socket object is instantiated by specifying the appropriate address family

Secondly, the Socket object of each vehicle is bound to a specific IP address and port number, effectively establishing its unique identity for communication purposes. This IP address and port number can be assigned manually or obtained dynamically.

Next, vehicles initiate a connection to establish a communication link between them. In the case of TCP-based communication, the initiating vehicle, acting as the client,

employs the Socket's connect() method to connect to the receiving vehicle, functioning as the server, specified by its IP address and port number.

Once the connection is established, vehicles can utilize the Socket's send() and receive() methods to exchange data and messages. Prior to transmission, data is typically serialized into a suitable format, such as JSON, and then transmitted over the Socket connection. Upon reception, the receiving vehicle deserializes the received data, enabling further processing and analysis.

Finally, when the communication session is deemed complete or vehicles are no longer part of the platoon, the Socket connection is closed using the Socket's close() method, ensuring the release of system resources.

It is of paramount importance to incorporate proper error handling, security measures (e.g., encryption), and data validation techniques to uphold the reliability and integrity of V2V communication utilizing the Socket protocol.

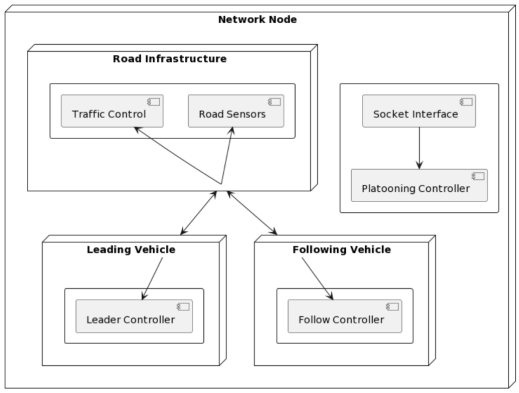

Figure 4. Network Node

By implementing V2V communication using the Socket protocol, vehicles within the platooning system can effectively exchange vital information, including platoon formation parameters, real-time traffic updates, and safety-related messages. Consequently, this fosters enhanced efficiency, safety, and coordination within the truck platooning system.

3.5. Communication Failure Handling [Lam]

The "Communication Failure Handling" state machine diagram provides a comprehensive framework for managing communication failures in a system. The process starts in the initial state, where the system is in normal communication mode. In the event of a communication failure, the system transitions to the "Attempt to Reestablish Communication" state. In this state, the system makes efforts to restore the communication link with the primary channel. If the communication is successfully restored, the system transitions to the "Inform Management System" state. Here, the system notifies the management system about the successful reestablishment of communication, ensuring that all relevant parties are informed. However, if the communication cannot be restored, the system enters the "Safety Mode"

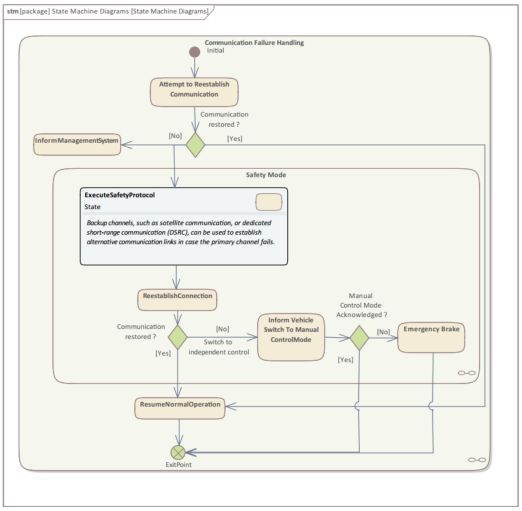

Figure 5. State Machine Diagram for Communication Failure Handling

state. In this state, the system activates a predefined safety protocol to ensure the system operates in a safe manner. The safety mode ensures that the system can handle critical operations and maintain essential functionalities despite the communication failure. The system remains in the safety mode until the communication is restored. Once the communication is reestablished, the system transitions to the "Reestablish Connection" state. In this state, the system verifies if the connection has been successfully reestablished. If the connection is restored, the system returns to normal operation, resuming regular communication. However, if the connection cannot be reestablished, the system switches to the "Switch to Independent Control" state. In this state, the system transitions to independent control mode, relying on backup channels such as satellite communication or dedicated short-range communication (DSRC). The system also informs the vehicle to switch to manual control mode, ensuring that the vehicle is prepared for the change in control. The system then waits for the acknowledgment of the manual control mode. If the acknowledgment is not received, an emergency brake is triggered to ensure the safety of the vehicle and its occupants. On the other hand, if the acknowledgment is received, the system resumes normal operation and reaches the exit point, indicating the completion of the communication failure handling process and the resumption of regular system functioning.

3.6. Obstacle Detection Handling [Lam]

The "Obstacle Handling" state machine diagram outlines a detailed process for handling obstacles in a truck platooning system. The process begins in the initial state, where the system is ready to detect obstacles. Once an obstacle is detected, the system transitions to the "Inform Road Management" state. In this state, the system communicates the obstacle information to the road management system, providing crucial details to ensure coordinated action. From

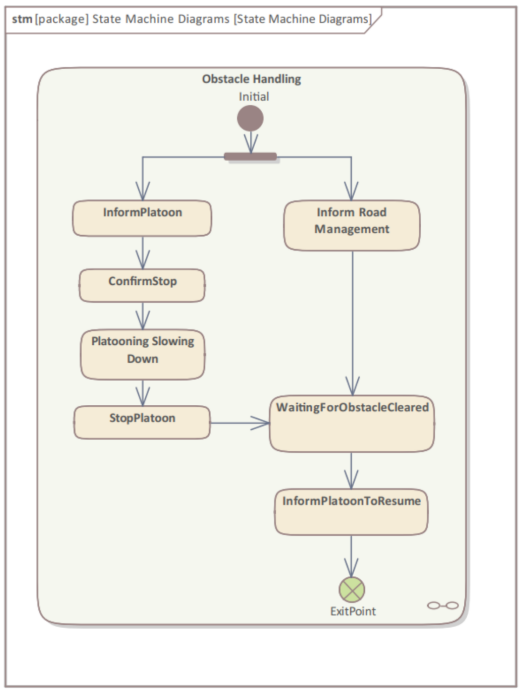

Figure 6. State Machine Diagram for Obstacle Detection Handling

there, the system moves to the ”Inform Platoon” state, where the platoon members are notified about the obstacle. This information enables the platoon to prepare for the necessary actions. Next, the system enters the ”Confirm Stop” state, where the platoon members collectively decide whether to stop based on the obstacle information. If the decision is to stop, the system transitions to the ”Platooning Slowing Down” state, gradually reducing the speed of the platoon to ensure a smooth and synchronized slowdown. Once the platoon comes to a complete stop, it enters the ”Stop Platoon” state, ensuring that all vehicles in the platoon have fully halted. The system then waits in the ”Waiting for Obstacle Cleared” state until the obstacle is cleared. Once the obstacle is no longer present, the system transitions to the ”Inform Platoon to Resume” state, instructing the platoon members to resume normal operation. Finally, the process reaches the exit point, indicating the successful handling of the obstacle and the resumption of platooning operations.

3.7. Safe Distance Maintaining [Quang]

3.7.1. Adaptive Cruise Control. In truck platooning formation, a group of trucks travels closely together, led by a lead truck. One of the biggest requirement is that the trucks maintain a safe distance between each travel and travel in a harmonized speed. Therefore, Adaptive Cruise Control plays a crucial role in truck platooning systems Adaptive Cruise Control, often referred to as ACC, permits the driver to hand the demands of acceleration and deceleration of the vehicle over to various algorithms deployed in the vehicle. The ACC system operates in the following two modes:

• **Speed control**: In speed control mode, the ego car maintains a driver-set speed. To achieve this, the ACC system utilizes a sensor, such as radar, to measure the distance to the preceding vehicle in the same lane (lead car) and the relative velocity of the lead car. If the distance to the lead car is equal to or greater than the safe distance, the ACC system remains in speed control mode. The control goal in this mode is to track the driver-set velocity and ensure a smooth and consistent travel speed.

• **Spacing control**: In spacing control mode, the ego car focuses on maintaining a safe distance from the lead car. The ACC system continuously monitors the real-time radar measurements to determine if the lead car is closer than the safe distance threshold. If the lead car's distance is smaller than the safe distance, the ACC system switches to spacing control mode. The control goal in this mode is to adjust the acceleration and deceleration of the ego car to maintain the safe distance from the lead car.[1]

The ACC system enables each truck in the platoon to maintain a safe and consistent distance from the lead truck. By utilizing radar sensors, the ACC system continuously monitors the distance and relative speed between the trucks. It automatically adjusts the acceleration and deceleration of each truck to ensure the desired spacing is maintained. By facilitating synchronized movement, ACC contributes to improved safety, efficiency, and overall performance in long-haul transportation.

3.7.2. Possible Controller for the ACC system. After doing some research there are three possible Controller that have been in used in Adaptive Cruise Control System. First one is the Proportional-Integral-Derivative Controller or PID for short. Second is the Model Predictive Control, often know as MPC. And lastly is the Fuzzy Logic Controller:

• **Proportional-Integral-Derivative (PID)**: Control algorithm that adjusts based on the error (deviation) between the desired state and the actual state. The PID controller calculates control actions by considering the proportional, integral, and derivative terms of the error signal.[2]

• **Model Predictive Control (MPC)**: Control technique that uses a dynamic model of the system to predict its future behavior. It formulates an optimization problem by considering system constraints and objectives and solves it repeatedly to generate optimal control actions.[3]

• **Fuzzy Logic Controller**: Based on fuzzy logic, which allows for the representation of imprecise or uncertain information. Use linguistic variables and rules to make control decisions. These controllers can handle complex and nonlinear relationships between inputs and outputs, making them suitable for ACC systems operating in diverse driving conditions.[4][5]

3.7.3. Advantages and disadvantages of the different Controller. PID control offers several advantages and limitations. On the positive side, its simplicity and wide usage make it a popular choice in control systems. PID controllers are relatively easy to implement and tune, with well-established methods available. They can provide real-time

response, allowing for quick adjustments to changes in the process variable. This makes them suitable for applications that require fast and dynamic control. However, PID control has limitations. It lacks inherent adaptation to changing conditions and disturbances, relying on manually set gains. This can limit its performance in systems with highly dynamic or unpredictable behavior. Additionally, PID controllers may require frequent tuning to maintain optimal performance, as changes in the system or operating conditions can degrade their effectiveness.

With the Model Predictive Control there is the advantage of utilizing a dynamic model for predicting system behavior and optimizing control actions. It considers factors like constraints and objectives while demonstrating adaptability to changing conditions. However, MPC requires complex implementation and tuning, and it can be computationally demanding. Furthermore, the prediction computation introduces latency in the control loop could make it less suitable for real-time applications.

Fuzzy Logic Controller offers advantages in handling uncertainty and imprecision, intuitive representation, adaptability to changes, and capturing non-linear relationships. It effectively deals with uncertain information and provides a human-readable control logic. FLC is flexible and can be adjusted to changing system conditions. Despite its advantages, Fuzzy Logic Controller also has some disadvantages. It may have high computational complexity and challenges in interpretability. Additionally, its optimization capability is limited compared to dedicated optimization algorithms

3.7.4. Apllying PID in implementation. When implementing a control system for a concept planning simulation and a small implementation, we chose the PID Controller as it offers simplicity and ease of implementation.

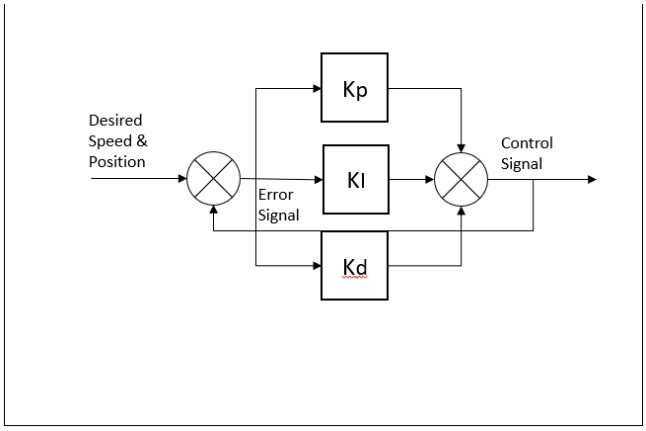

Figure 7. PID Controller

The PID controller calculates the control input based on the error between the desired setpoint and the current system output. The control input is determined using three components: the proportional term (P), the integral term (I), and the derivative term (D). The formula for the PID controller is as follows:

*Output = Kp * Error + Ki * Integral(Error) + Kd * Derivative(Error)*

where:

• **Control Input**: The output control signal or action to be applied to the system.

• **Kp**: Proportional gain, a constant that determines the influence of the proportional term.

• **Ki**: Integral gain, a constant that determines the influence of the integral term.

• **K**: Derivative gain, a constant that determines the influence of the derivative term.

• **Error**: The difference between the desired setpoint and the current system output.

• **Integral(Error)**: The accumulated sum of the error over time.

• **Derivative(Error)**: The rate of change of the error over time.

From the mathematical equation of the PID Controller. We derived a Constrain Block Diagram as well as a Parametric Diagram.

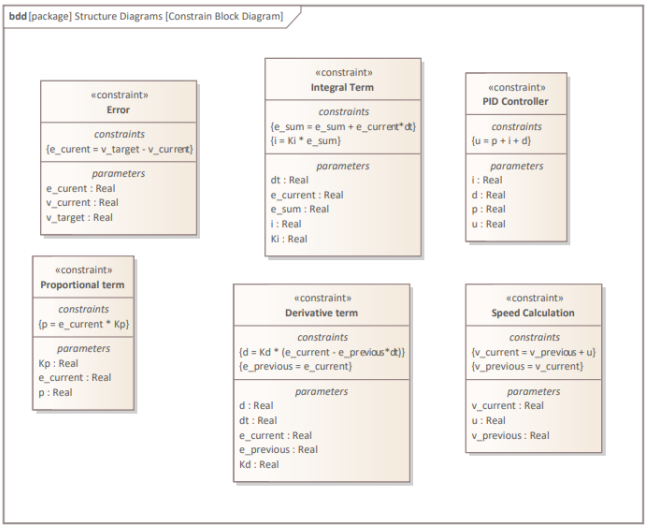

Figure 8. Constrain Block Diagram

For a better understanding of interaction between the PID Controller, we use an activity diagram to represent the process flow in a distance-maintaining system.

• **Send Speed and Position Data**: This action involves the leading truck transmitting the current speed and position data to the platoon system. It ensures that the most up-to-date information is shared with the platoon system.

• **Receive & Send Leader Information**: This action encompasses both receiving and sending leader truck's information. It involves receiving information from the leading truck about its relative position and speed and also passing these information to the following trucks, potentially for coordination or synchronization purposes.

• **Receive Leader Information**: This action focuses on the following trucks receiving information about the leader's position and speed from the platoon system. It

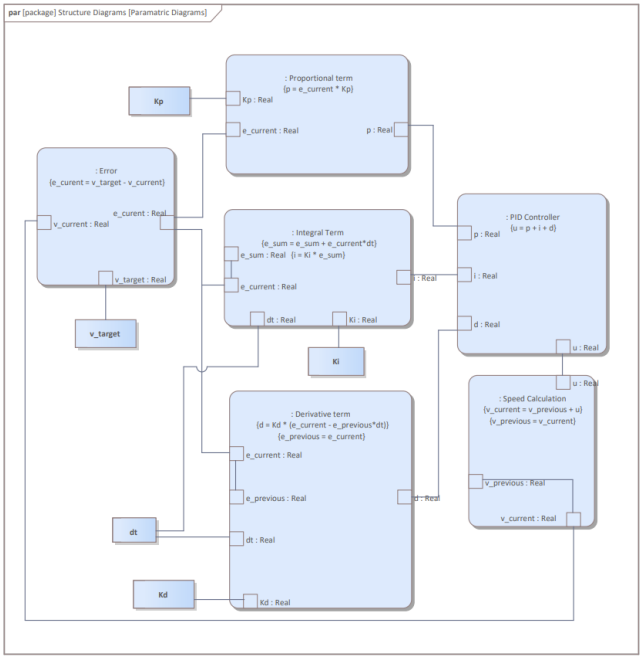

Figure 9. Parametric Diagram

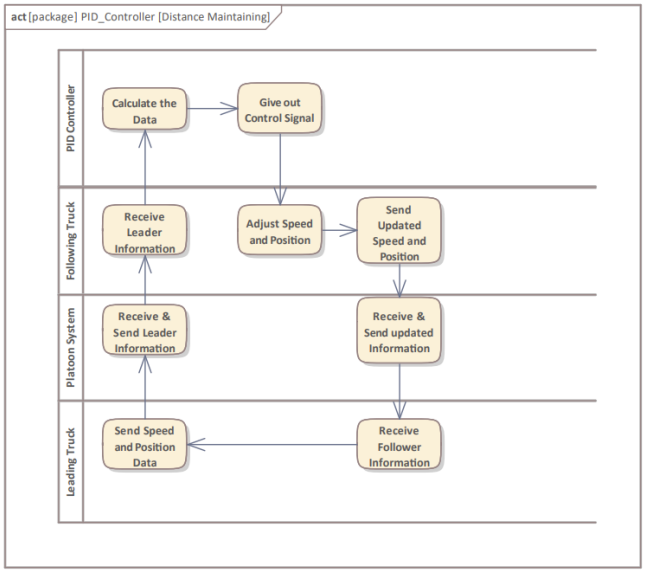

Figure 10. Activity Diagram for Distance Maintaining

allows the following truck to calculate and maintain the desired distance.

• **Calculate the Data**: In this action, the PID performs calculations using the received information. It involves computations related to the error between the desired distance and the current distance.

• **Give out Control Signal**: This action involves providing the calculated control signal as output. It represents the instruction or command generated by the PID control to adjust the speed or position of the following truck.

• **Adjust Speed and Position**: This action pertains to modifying the speed and position of the following trucks based on the control signal. It ensures that the following trucks maintain the desired distance from the leading truck.

• **Send Updated Speed and Position**: After adjusting the speed and position, this action involves transmitting the updated data to the platooning system. It ensures that the adjusted information is shared with relevant entities.

• **Receive & Send updated Information**: This action involves both receiving and sending updated information. It includes the reception of updated data from the following trucks and the subsequent transmission of relevant information to the leading truck.

• **Receive Follower Information**: This action focuses on the leading truck receiving specific information related to the following truck‘ position and speed.

3.7.5. Apllying PID in implementation. To enhance the mathematic calculation performance of the system, for our simulation, we will task the PID calculation to the GPU. Traditionally, GPUs were designed for rendering graphics in video games and other visual applications. However, their architecture is highly parallel and well-suited for performing large-scale computations. Therefore, GPUs can deliver significant computational power compared to traditional CPUs. This can be advantageous for control systems that require complex calculations or operate on large datasets. Following is some of the advantages that GPUs’ calculation might bring to the current system:

• Real-Time Responsiveness

• Handling Multiple Truck

• Optimization and Tuning

• Computational Efficiency

• Sensor Data Processing

Especially in our simulation, because a C++ based simulation will be programmed and launched, CUDA will be used to program the PID Controller. CUDA stands for Compute Unified Device Architecture. It is a parallel computing platform and application programming interface (API) model created by NVIDIA. The primary goal of CUDA is to provide a programming model that allows developers to write code that can execute in parallel on NVIDIA GPUs. It includes a set of extensions to the C and C++ programming languages, which allows developers to write code known as ”CUDA kernels” that can run on the GPU. To be able to program with CUDA, the system must first meets the hardware and software requirement. These requirements can be found in the CUDA documentation provided by NVIDA. For the simulation, firstly, we define a CUDA kernel function for PID Calculation. Then we use CUDA memory management functions to allocate, free device memory and transfer the data between the main process and GPU memory. To compile the code, we use the nvcc-compiler from CUDA.

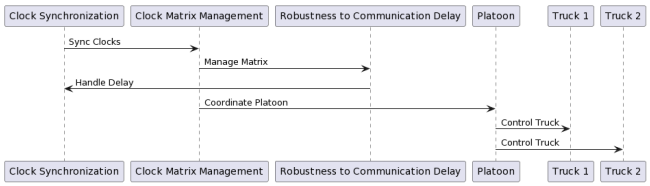

Figure 11. SEQUENCE DIAGRAM: CLOCK MATRIX PATTERN

3.8. Clock Matrix for Time Synchronization [Hadis]

Incident Management aims to detect and manage incidents, such as accidents or road hazards, in real-time to minimize their impact on traffic flow. The system should have the capability to promptly detect incidents, notify relevant authorities, and coordinate with autonomous vehicles to divert traffic and implement appropriate traffic control measures. The requirements include rapid incident detection, efficient incident reporting, effective communication with emergency services, and dynamic rerouting of autonomous vehicles. Challenges involve accurate incident detection, reliable communication during emergencies, and coordination with various stakeholders.

We use sequence diagram to illustrates the order and flow of interactions between various components or entities in a system over time. In the context provided, the sequence diagram focuses on depicting the interactions within a platoon system, highlighting specific aspects such as clock synchronization, clock matrix management, communication delays, and overall coordination and control within the platoon.

- **Clock Synchronization:** The diagram captures how different components in the platoon synchronize their clocks. Clock synchronization is crucial for ensuring that all devices or nodes in the system share a common understanding of time. The sequence of messages or actions related to clock synchronization is visually represented, emphasizing the steps taken to align the clocks across the platoon.
- **Management of the Clock Matrix:** The Clock Matrix is a method used for time synchronization in distributed systems. The diagram outlines the flow of interactions involved in managing the Clock Matrix. This may include messages exchanged between components to update and maintain the matrix, ensuring that the synchronization process is effective and accurate.
- **Handling Communication Delays:** Communication delays are inherent in distributed systems, where different components may experience varying levels of latency in message transmission. The sequence diagram illustrates how the platoon system addresses and manages these communication delays. It may depict actions taken to mitigate delays and ensure that the synchronization process remains reliable despite the challenges posed by network latencies.
- **Coordination and Control Within the Platoon:** The diagram visualizes how components within the platoon coordinate and control their actions. This could involve messages exchanged to convey instructions, updates, or acknowledgments among the platoon members. The sequence of interactions emphasizes the organized flow of information and actions that contribute to the effective functioning of the entire platoon.
- **Arrows Indicating Message Flow:** Arrows on the diagram serve as visual indicators of the direction of message flow between different participants or components in the platoon system. These arrows help in understanding the order and dependencies of interactions, illustrating how information or commands move between the various entities within the system.

3.8.1. Clock synchronization disadvantages. Effective time synchronization is essential for the seamless operation of distributed systems, ensuring that different components can function cohesively with a shared understanding of time. Among the various methods employed, the Clock Matrix has gained popularity for its promise of improved accuracy and resilience. However, it's crucial to delve into the drawbacks associated with the Clock Matrix to gain a comprehensive understanding of its limitations.

When multiple devices or components work together, it's crucial that they all agree on what time it is. The Clock Matrix is a method that helps them do that more accurately, but it's not perfect. This discussion will explore the challenges it faces, like becoming more complicated as more devices are involved, being sensitive to delays in communication, struggling to handle a growing number of devices, and facing potential security risks. Understanding these downsides is key to making informed decisions about using the Clock Matrix for time synchronization in distributed systems.

- **Complex Implementation:** Clock synchronization systems in platooning involve intricate implementations, requiring precise coordination and communication between multiple vehicles. The complexity arises from the need to establish and maintain synchronization across various components, such as onboard clocks, communication modules, and sensors. Achieving seamless integration and ensuring consistent synchronization can be challenging, potentially leading to increased development and maintenance costs.
- **Vulnerability To Communication Issues:**Platooning systems heavily rely on communication between vehicles to maintain synchronized movement. However, this dependence on communication introduces a vulnerability. Any disruptions or delays in communication channels, such as signal interference, network congestion, or loss of connectivity, can impact the accuracy

of time synchronization. In platooning scenarios, where precise coordination is crucial for safe and efficient operations, communication issues may lead to misalignments among vehicles, potentially compromising safety.

- **Cybersecurity Risks:** Clock synchronization systems are susceptible to cybersecurity threats, as they often involve the exchange of time-related data over networks. Malicious actors could attempt to manipulate or inject false timestamp information, leading to inaccuracies in the synchronized clocks. Cyber attacks on the clock synchronization infrastructure may compromise the integrity of platooning operations, posing risks to the safety and reliability of the entire system. Ensuring robust cybersecurity measures is crucial to mitigate these potential threats.
- **Network Dependency:** Clock synchronization in platooning systems relies on network connectivity to exchange timing information among vehicles. The dependency on a network introduces a risk of disruptions due to network failures, coverage gaps, or other issues. If a vehicle loses connection to the network or experiences delays in receiving synchronization updates, it may lead to desynchronization among platooning vehicles. Redundancy measures and failsafes need to be implemented to address network-related challenges and maintain the integrity of the synchronization process.

3.8.2. Improving the clock synchronization. Addressing the challenges associated with clock synchronization in platooning systems requires a combination of technological solutions, robust protocols, and contingency measures. Here are potential solutions for each of the identified disadvantages:

-*Complex Implementation:* we need to implement standardized protocols and interfaces to streamline the integration process. Develop calibration tools and guidelines that simplify the synchronization setup for different vehicle models.

-*Vulnerability to Communication Issues:*Introduce redundancy in communication channels to mitigate the impact of disruptions. Implement error-checking mechanisms and protocols that can detect and correct communication errors. Develop algorithms that predict and compensate for communication delays to maintain synchronization. Consider integrating vehicle-to-everything (V2X) communication technologies that enhance reliability and reduce the likelihood of interference.

-*Cybersecurity Risks:*It is essential that we employ robust encryption and authentication mechanisms to secure communication channels and timestamps. Regularly update software and firmware to patch vulnerabilities and address emerging cybersecurity threats. Implement intrusion detection systems and anomaly detection algorithms to identify and respond to malicious activities. Conduct regular cybersecurity audits and collaborate with experts to stay ahead of potential threats. -*Network Dependency:*It's necessary for us to design platooning systems with adaptive

communication protocols that can handle varying network conditions. Integrate local communication capabilities to allow vehicles to synchronize even in the absence of a stable network connection. Implement decentralized synchronization algorithms that reduce dependence on a central network server. Develop failover mechanisms that allow vehicles to operate safely in the event of network failures, relying on onboard sensor data for short periods.

4. Evaluation [Shirin]

Testing platooning systems involves assessing various aspects of the technology to ensure its safety, efficiency, and reliability. Platooning refers to a group of vehicles moving closely together, autonomously, to improve traffic flow and fuel efficiency. It's crucial to adopt a comprehensive and iterative testing approach that includes a combination of simulation, controlled environment testing, and real-world scenarios to ensure the platooning system's safety and effectiveness. Collaboration with regulatory bodies, industry partners, and experts in autonomous vehicle technology is also essential throughout the testing process.

The effectiveness of safety features, including collision avoidance systems, emergency braking, and Lane-Keeping Assistance is scrutinized. Testing protocols for assessing the platooning system's response to sudden changes in traffic conditions or obstacles are discussed:

4.1. Collision Avoidance Systems

a. Laboratory Testing - Hardware-in-the-Loop (HIL) Simulation:. HIL simulation provides a controlled environment to rigorously evaluate the collision avoidance system's performance. Researchers can design scenarios where unexpected obstacles or abrupt changes in traffic conditions occur within the platoon. The simulation involves integrating actual collision avoidance system hardware into the virtual environment, allowing for real-time interaction between the system and simulated platoon dynamics.

Benefits:

- Dynamic Scenarios: Simulating dynamic platooning scenarios with sudden obstacles ensures a thorough assessment of the collision avoidance system's responsiveness.
- Hardware Integration: Testing with actual hardware components enhances the realism of the simulation, providing insights into the system's real-world behavior.

b. Virtual Testing:. Virtual testing extends the evaluation to a broader spectrum of scenarios by leveraging simulation software. Researchers can design virtual environments with diverse and challenging conditions, such as multiple obstacles or rapid changes in traffic dynamics. This method allows for scalability and cost-effectiveness in assessing

the collision avoidance system's capabilities under various platooning configurations.

c. Real-world Testing:. Real-world testing involves taking the collision avoidance system into controlled environments to validate its performance in actual platooning scenarios. Researchers gradually introduce complexity, starting with basic scenarios and progressing to more intricate situations involving multiple interacting vehicles. Real-world testing provides insights into the system's adaptability and effectiveness in dynamic environments.

Benefits:

- Real-world Dynamics: Testing in a controlled, yet realistic, environment ensures that the collision avoidance system performs as expected in actual platooning conditions.
- Scenario Complexity: Gradually increasing the complexity of scenarios allows for a thorough evaluation of the system's robustness.

4.2. Emergency Braking

a. Hardware-in-the-Loop (HIL) Simulation:. Similar to collision avoidance testing, HIL simulation for emergency braking focuses on integrating actual hardware components into a simulated platooning environment. Researchers create scenarios that demand emergency braking responses, such as sudden deceleration of lead trucks or obstacles entering the platoon's path. This method allows for the real-time evaluation of the emergency braking system's performance in dynamic platooning conditions.

Benefits:

- Realistic Platoon Dynamics: Simulating emergency braking within the platoon's dynamics ensures accurate representation of real-world scenarios.
- Hardware Integration: Using actual emergency braking hardware enhances the fidelity of the simulation.

b. Virtual Testing:. Virtual testing for emergency braking extends the evaluation to a broader set of scenarios using simulation software. Researchers can create virtual environments with varying road conditions, traffic dynamics, and unexpected events. This method allows for scalable and efficient testing of the emergency braking system's capabilities in diverse platooning scenarios.

c. Sensor Fusion and Perception Testing:. Sensor fusion and perception testing focus on assessing the accuracy and reliability of sensors crucial for emergency braking. Researchers validate the sensor fusion capabilities, ensuring precise data collection and effective decision-making in emergency situations within the platoon.

Benefits:

- Accurate Data Collection: Ensuring sensors provide precise information is critical for the effectiveness of emergency braking responses.
- Multi-Sensor Integration: Validating the integration of various sensors, such as radar and lidar, enhances the system's overall perception capabilities.

4.3. Lane-Keeping Assistance (LKA)

a. Hardware-in-the-Loop (HIL) Simulation:. HIL simulation for Lane-Keeping Assistance involves integrating actual hardware components into a simulated platooning environment. Scenarios include intentional and unintentional lane departures within the platoon, evaluating the LKA system's ability to maintain lane discipline.

Benefits:

- Realistic Platoon Dynamics: Simulating within the platoon's dynamics ensures accurate representation of real-world scenarios.
- Hardware Integration: The use of actual LKA hardware enhances the fidelity of the simulation.

b. Virtual Testing:. Virtual testing for Lane-Keeping Assistance involves creating virtual environments with diverse scenarios using simulation software. This allows researchers to test the system's capabilities in various conditions, such as different road geometries and traffic dynamics.

Benefits:

- Cost-Effective Iterations: The virtual environment allows for rapid iterations through different scenarios without the logistical challenges and expenses associated with real-world testing. Researchers can fine-tune algorithms and parameters efficiently, ensuring optimal LKA system performance for truck platooning.
- Algorithmic Optimization: Virtual testing provides a platform to evaluate and optimize the LKA system's algorithms under various conditions.
- Efficient Evaluation of Edge Cases: The virtual environment enables the creation of edge cases and rare scenarios that may be difficult to replicate in real-world testing.

c. Sensor Calibration Tests:. Sensor calibration tests are critical for optimizing sensor parameters to ensure the precise and consistent reception of data for accurate lane detection. This phase involves adjusting parameters such as field of view and sensitivity to achieve optimal performance.

Benefits:

- Enhanced Precision: Calibrated sensors minimize the likelihood of false positives or negatives in lane detection.
- Adaptability to Environmental Changes: Adaptive calibration ensures consistent performance across changing conditions.
- Optimal System Integration: Integration with vehicle dynamics enhances the overall effectiveness of the LKA system.

The benefits of each testing methodology, along with specific use cases, contribute to a robust evaluation of platooning system safety and effectiveness. This iterative testing approach, spanning simulation, controlled environments, and real-world scenarios, ensures the reliability and adaptability of platooning systems in diverse conditions. It sets the foundation for the successful implementation of autonomous truck platooning, addressing the challenges and complexities inherent in real-world driving scenarios.

5. Summary and Outlook [Quang]

Truck platooning is a technology that enables a group of trucks to travel closely together in a formation, with a lead truck dictating the speed and direction for the rest of the platoon. This innovative system offers numerous benefits, such as increased fuel efficiency, reduced traffic congestion, improved safety, and lower carbon emissions. This research project provides a comprehensive overview of a truck platooning system. To summarize the whole system, the State Machine Diagrams illustrates the different states and transitions involved in the platooning process. Moreover, there is the Package Diagram to depict the components and their relationships within the system and the Block Definition Diagrams to showcase the key blocks and their interactions.

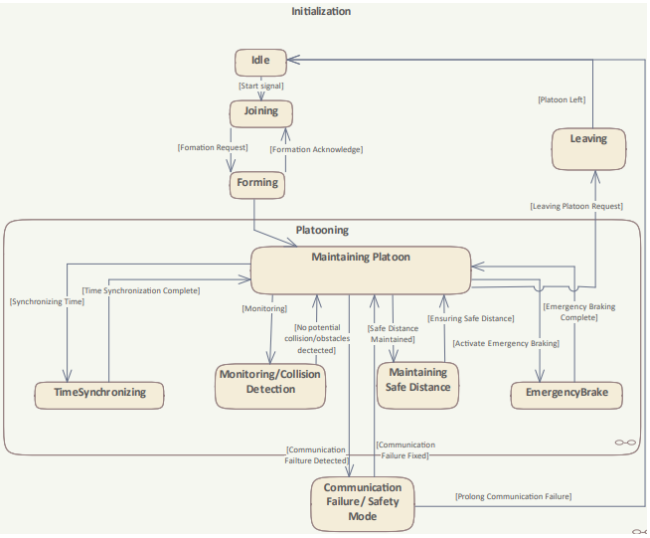

Figure 12. State Machine for Truck Platooning System

The future of truck platooning systems appears promising, as the technology continues to evolve and attract significant interest from the transportation industry. The continued advancements in sensors, connectivity, and automation will enhance the capabilities of truck platooning systems. The ongoing transition towards electric and autonomous trucks will significantly impact the future of truck platooning. Electric trucks offer reduced emissions and lower operating costs, while autonomous technology can enhance the safety

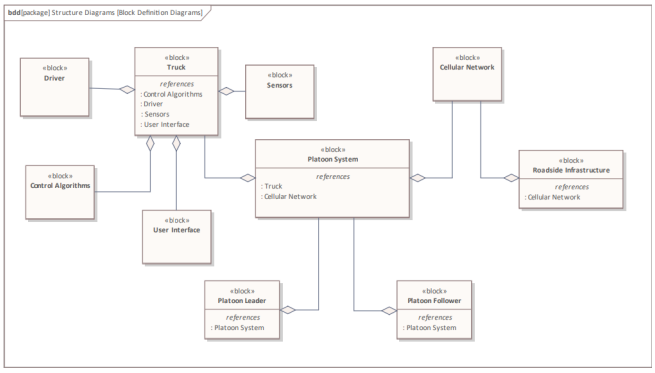

Figure 13. Truck Platooning System's Block Definition Diagram

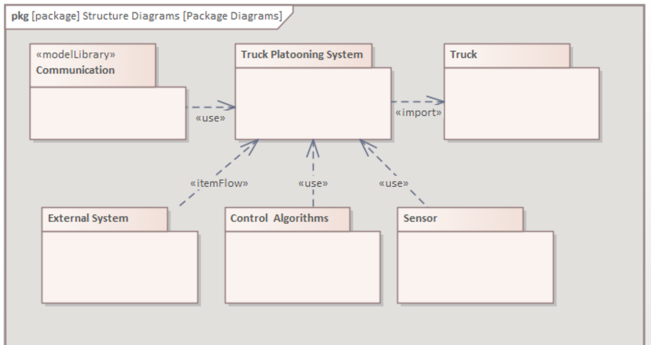

Figure 14. Truck Platooning System's Package Diagram

and efficiency of platooning operations. Integrating these technologies with truck platooning systems will be a key focus in the coming years.

In conclusion, truck platooning systems have the potential to revolutionize the transportation industry by improving efficiency, safety, and sustainability. While there are still hurdles to overcome, continued advancements in technology, supportive regulations, infrastructure development, industry collaboration, and the integration of electric and autonomous technologies will pave the way for the widespread adoption of truck platooning systems in the near future.

6. Limitation[Hadis]

The development and implementation of truck platooning systems hold the promise of revolutionizing the logistics and transportation industry, offering potential benefits such as enhanced fuel efficiency, improved traffic flow, and reduced emissions. However, like any transformative technology, platooning systems come with their own set of challenges and limitations. In this exploration, we dive into the multifaceted limitations associated with truck platooning systems.

6.1. Human Interaction

*Mixed Traffic Challenges:*The integration of platooning vehicles into mixed traffic scenarios presents challenges

in coordinating with human-driven vehicles. Algorithms and communication protocols must be developed to enable seamless integration, considering diverse driving styles and unpredictable behaviors of human drivers. Clear signaling mechanisms, such as specialized lights or displays on platooning trucks, help convey their intentions to human drivers, reducing confusion. Infrastructure adjustments, like dedicated lanes, can facilitate smoother interactions between autonomous and human-driven vehicles, fostering harmonious coexistence on the road.

Emergency Scenarios Handling Unpredictable Situations: In emergency scenarios or sudden changes in road conditions, autonomous trucks must respond coherently and predictably. Coordinating responses with human drivers in these situations requires robust communication strategies to avoid confusion and enhance overall road safety.

6.2. Environmental Factors

Adverse Weather Impact: Harsh weather conditions such as heavy rain, snow, fog, or storms can significantly impact the performance of sensors and communication systems essential for platooning. Reduced visibility and slippery roads may challenge the system's ability to accurately detect obstacles, maintain proper spacing, and respond swiftly to changing conditions. *Variability in Road Surfaces:* Platooning trucks must adapt to diverse road surfaces, including uneven or poorly maintained roads. Ensuring stability and safety across different road conditions, such as potholes, construction zones, or unpaved stretches, requires robust sensor fusion and control algorithms.

7. Affidavit

We (Nhat Lam Nguyen, Nhat Quang Nguyen, Shirin Babaeikouros, Hadis Mohammadi) herewith declare that we have composed the present paper and work our self 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.

References

[1] MathWork. Adaptive Cruise Control System Using Model Predictive Control [Online]. Available: https://de.mathworks.com/help/mpc/ug/adaptive-cruise-control-using-model-predictive-controller.html#d126e40997

[2] Artisan Controls Corporation. PID What is PID and how does it work? Auto tuning PID with the 5400 Controller [Online]. Available: http://www.artisancontrols.com/UtilityImages/ProductSupportPdf

[3] University of Stutgart Institute for Systems Theory and Automatic Control. Model Predictive Control [Online]. Available: https://www.ist.uni-stuttgart.de/research/group-of-frank-allgoewer/model-predictive-control/

[4] T. Kuruvilla. (2023, April 28) "Developing Adaptive cruise control using Fuzzy Logic,".MathWorks Blogs [Online]. Available: https://blogs.mathworks.com/student-lounge/2023/04/28/adaptive-cruise-control-using-fuzzy-logic/

[5] [5] K. Alomari, R. C. Mendoza, S. Sundermann, D. Goehring, R. Rojas. "Fuzzy Logic-based Adaptive Cruise Control for Autonomous Model Car,". in Conf. ROBOVIS 2020 - International Conference on Robotics, Computer Vision and Intelligent Systems, Portugal, 2020

[6] J. B. Kenney, "Dedicated Short-Range Communications (DSRC) Standards in the United States," Proceedings of the IEEE, vol. 99, no. 7, pp. 1162-1182, 2011.

[7] Author: Ali Balador, Alessandro Bazzi, Unai Hernandez-Jayo, Idoia de la Iglesia, Hossein Ahmadvand Title: A survey on vehicular communication for cooperative truck platooning application Published by: Available online 28 February 2022 DOI: doi.org/10.1016/j.vehcom.2022.100460

[8] B. Zarebavani, F. Jafarinejad, M. Hashemi, S. Salehkaleybar, "cuPC: CUDA-based Parallel PC Algorithm for Causal Structure Learning on GPU", IEEE Transactions on Parallel and Distributed Systems (TPDS).

[9] Ozbay, K., Yang, H., & Holguin-Veras, J. (2019). Optimal Formation Patterns of Trucks in Platoons for Safe and Efficient Freight Transportation. Transportation Research Record, 2673(8), 343-355.

[10] Li, L., & Peng, H. (2018). A review on the coordination of heavy-duty truck platooning. Transportation Research Part C: Emerging Technologies, 89, 276-298.

8. APPENDIX

8.1. Diagrams

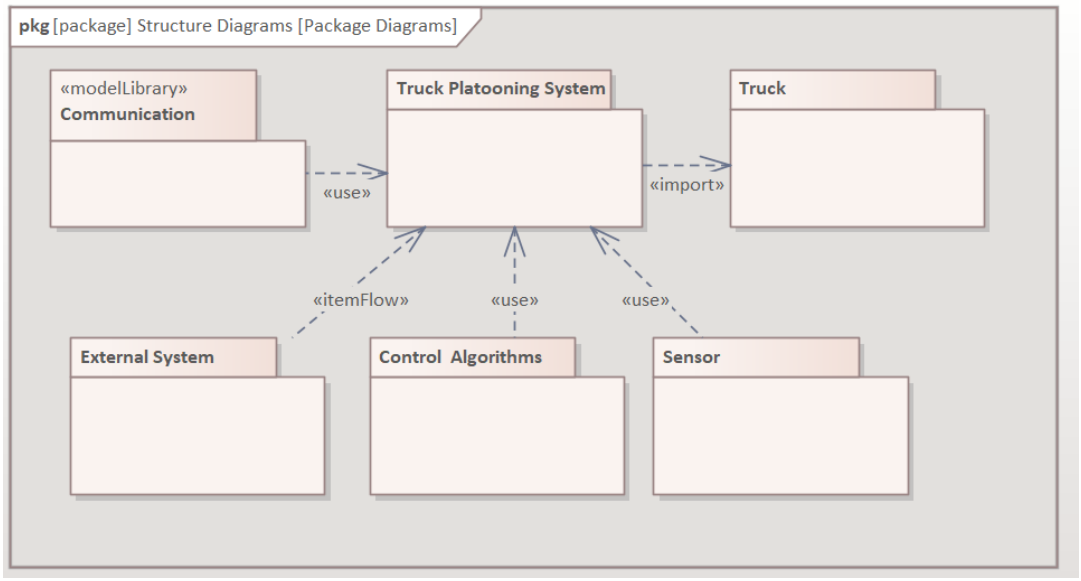

Figure 15. Package Diagram

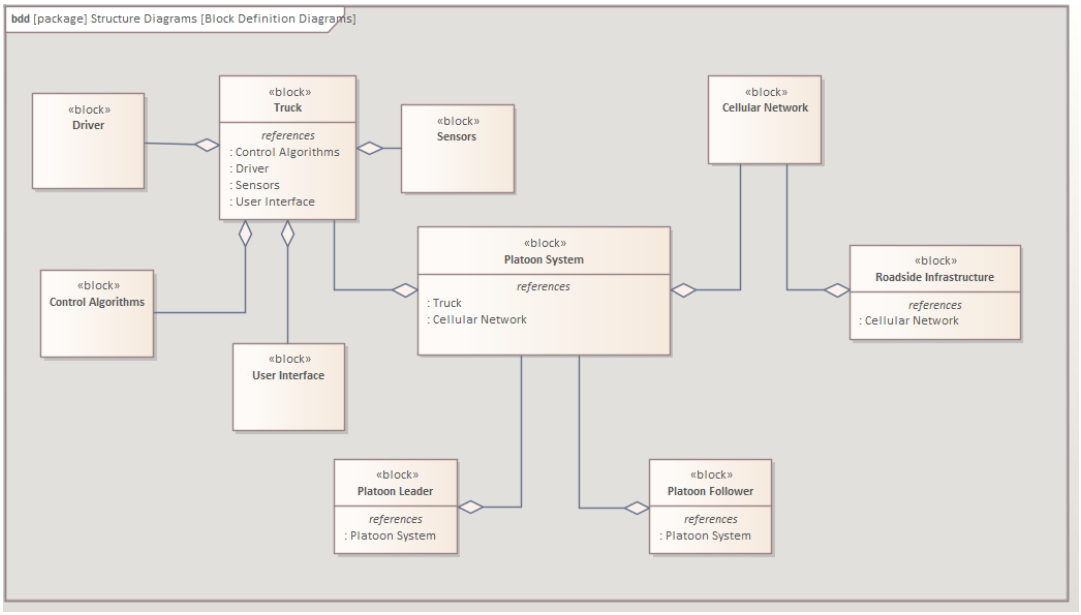

Figure 16. Block Diagram

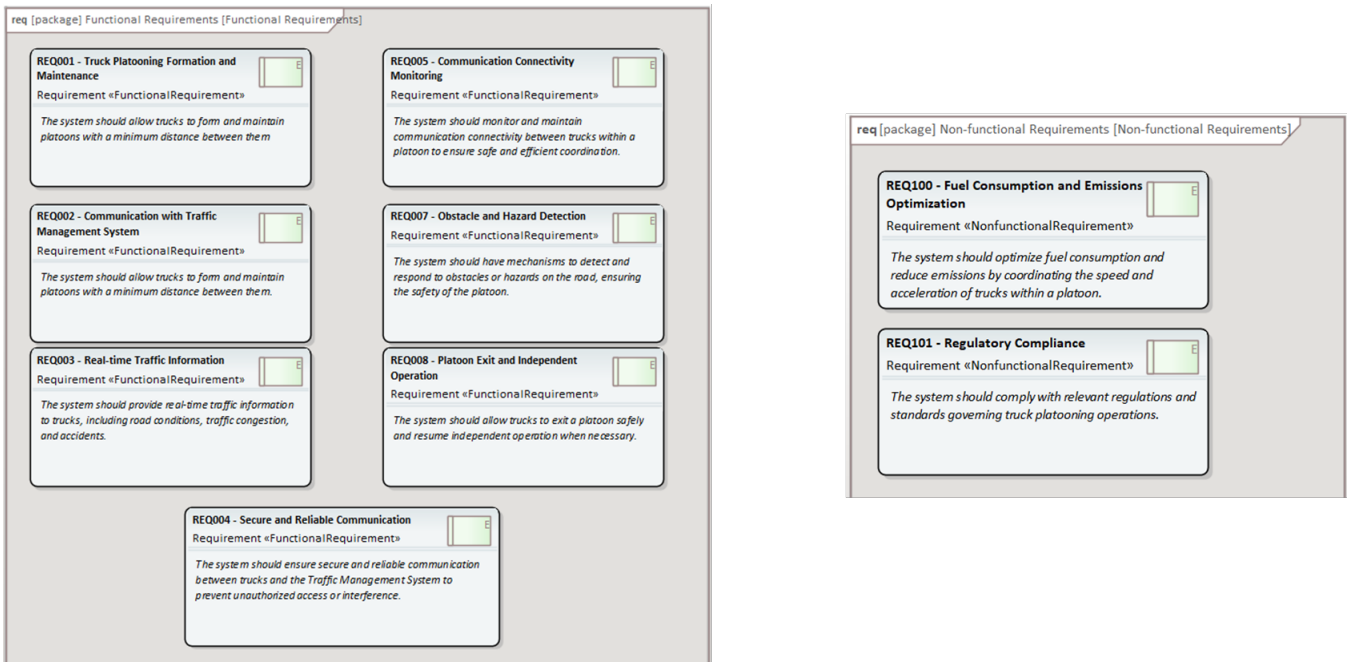

Figure 17. Requirement Diagrams

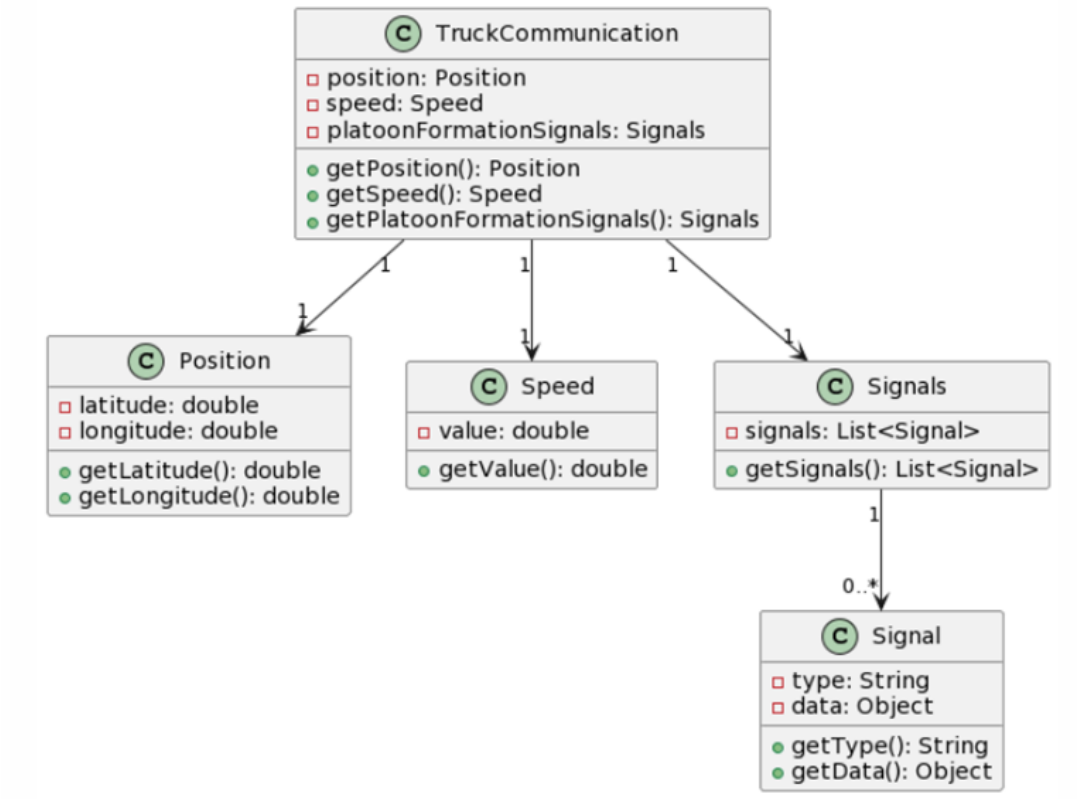

Figure 18. Identification of required data for truck communication

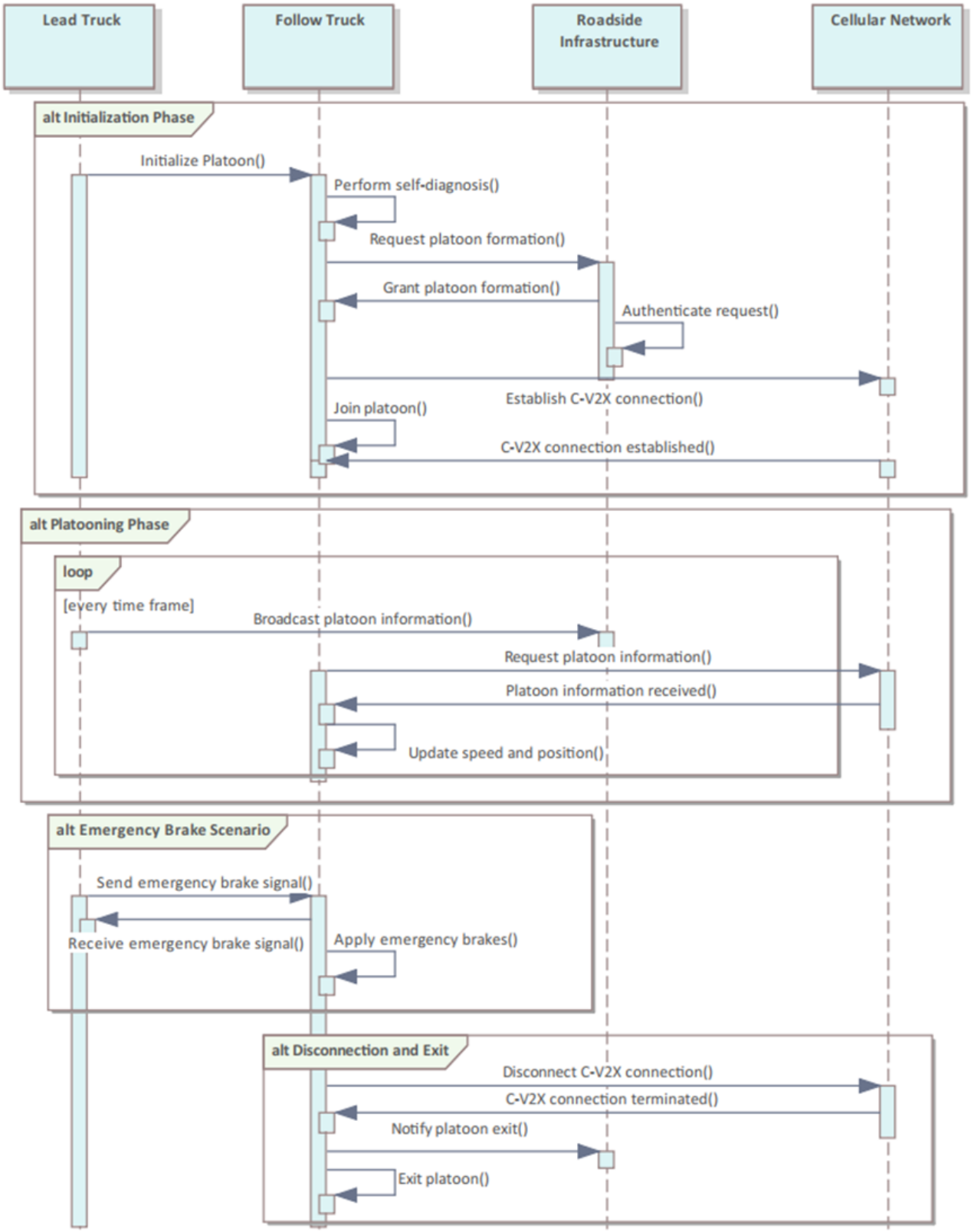

Figure 19. System communication specification using sequence diagram model

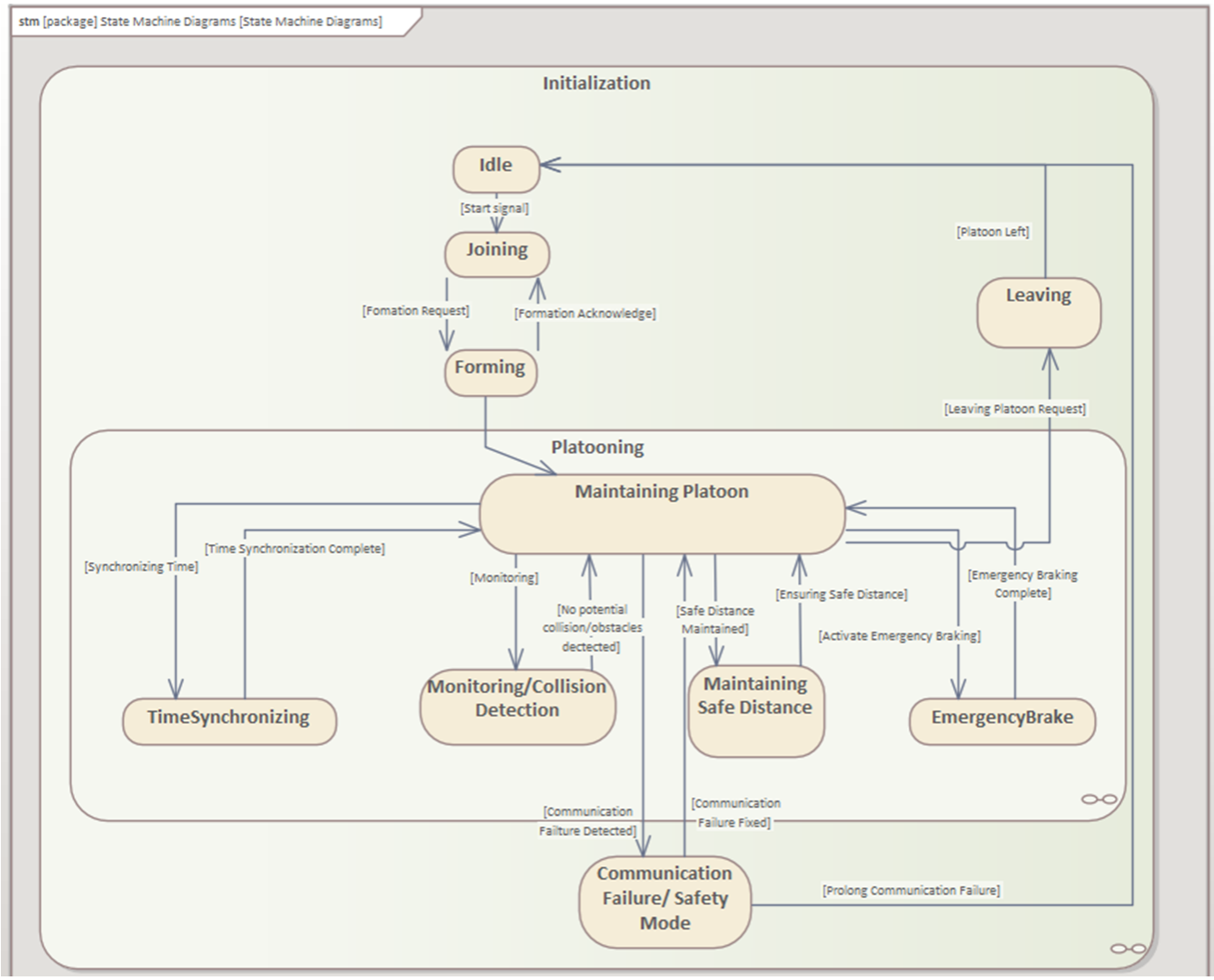

Figure 20. System operation states specified with a state machine diagram model

```
            followersPosition[followerMessage.senderId] = followerMessage.position;
            followersSpeed[followerMessage.senderId] = followerMessage.speed;

            _obstacleDetected = followerMessage.obstacleDetected;
            // Check if obstacle is detected by the follower
            if (_obstacleDetected)
            {
                std::cout << "Obstacle Detected by the follower" << std::endl;
                while(speed>0)
                {
                    speed--; // Set leader's speed to 0 if obstacle is detected by the follower
                    std::this_thread::sleep_for(std::chrono::milliseconds(10));
                }
            }

        }
    }

    void LeadingVehicle::printState()
    {
        std::cout << "Leading Vehicle ID: " << id;
        std::cout << "  Position: " << position;
        std::cout << "  Speed: " << speed << std::endl;

        #pragma omp parallel for
        for (int iD : followerId)
        {
            if (_obstacleDetected)
            {
                std::cout << "Follower ID " << iD << " is facing an obstacle" << std::endl;
            }
            std::cout << "Connected Follower ID: " << iD;
            std::cout << "  Position: " << followersPosition[iD];
            std::cout << "  Speed: " << followersSpeed[iD] << std::endl;
        }
        std::cout << std::endl;

        printClockMatrix();
    }

    int main()
    {
        LeadingVehicle leadingVehicle(1, 50, 60.0);
        leadingVehicle.startServer();

        while (true)
        {
            #pragma omp parallel for
            // Increment the clock matrix by 1
            leadingVehicle.incrementClockMatrix(1);
            std::this_thread::sleep_for(std::chrono::milliseconds(100));

            // Perform other tasks in the main function, if any
        }

        leadingVehicle.stopServer();
        return 0;
    }
```

Figure 21. COMMUNICATION FAILURE

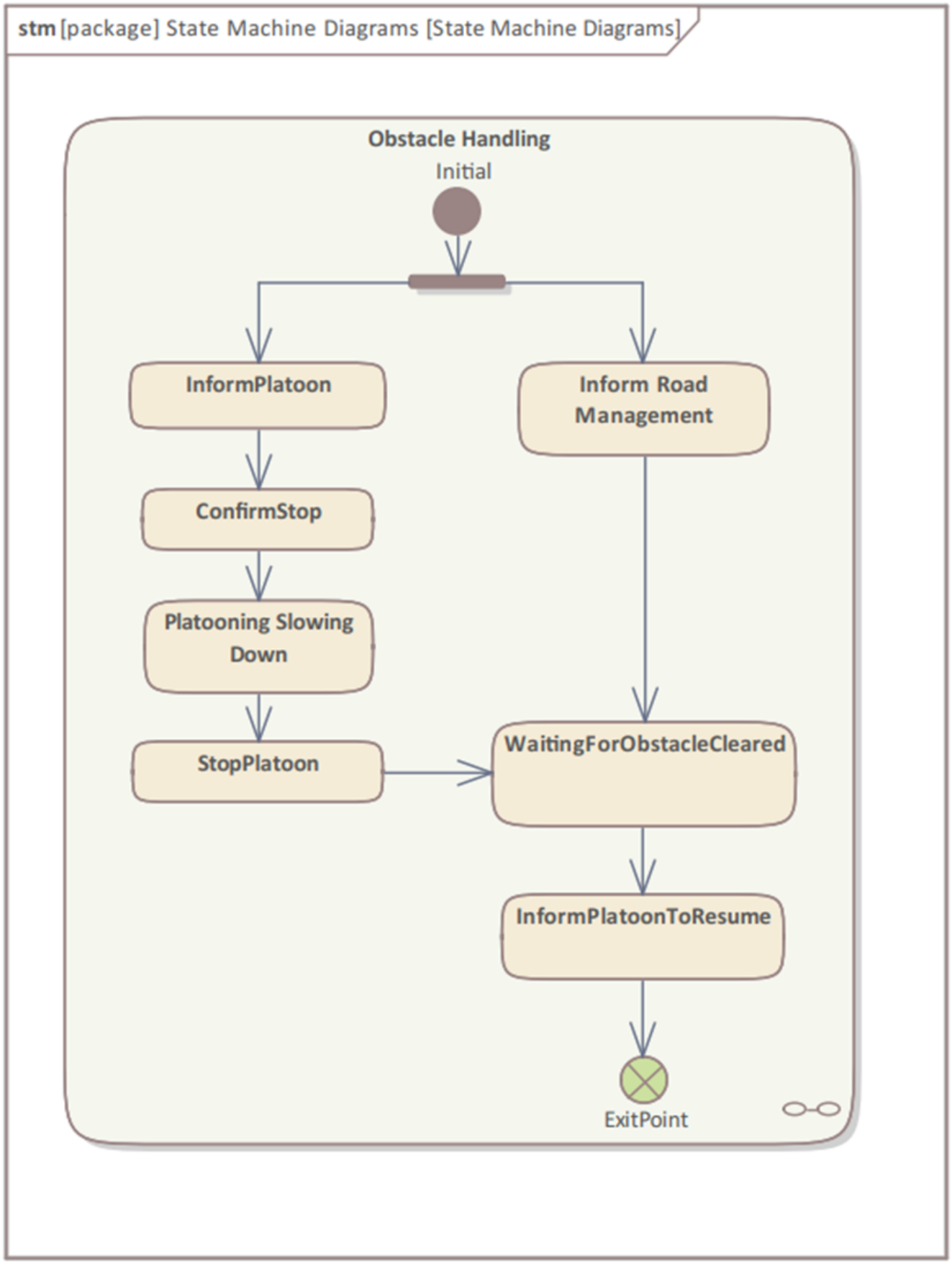

Figure 22. OBSTACLE DETECTION

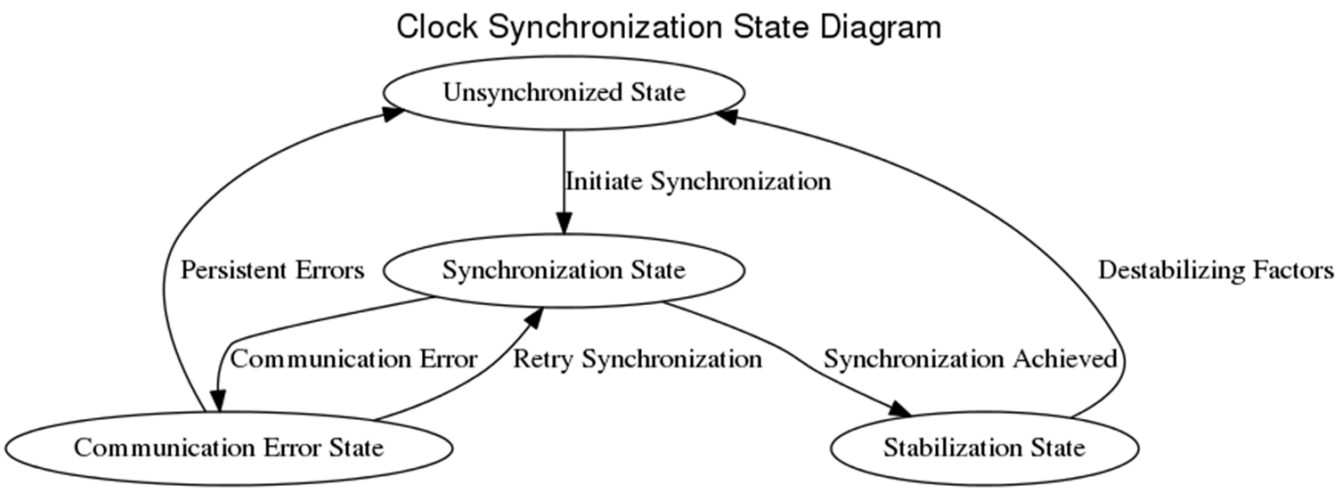

Figure 23. Clock synchronization State Diagram

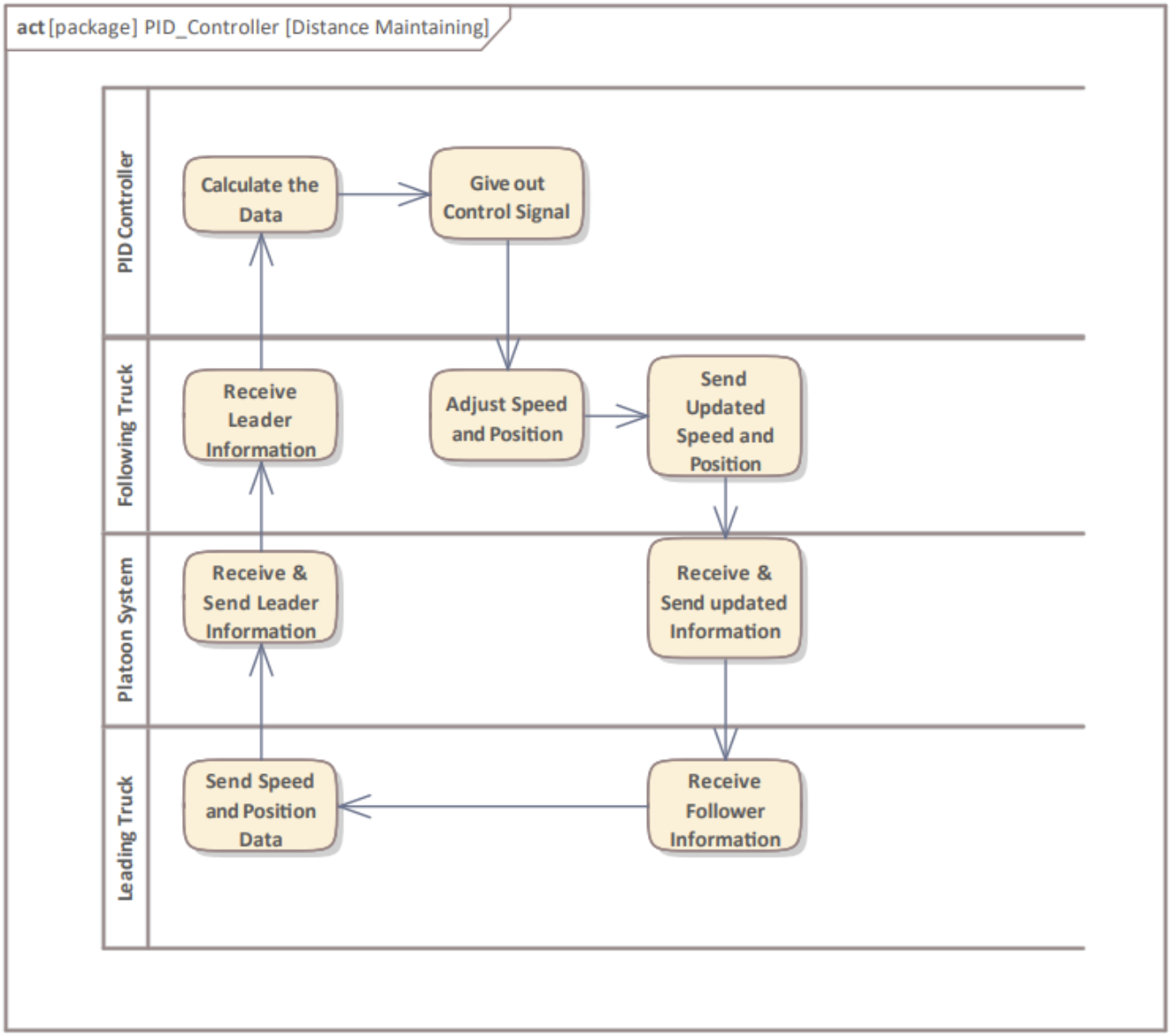

Figure 24. Activity Diagram for the Safe Distance Maintaining

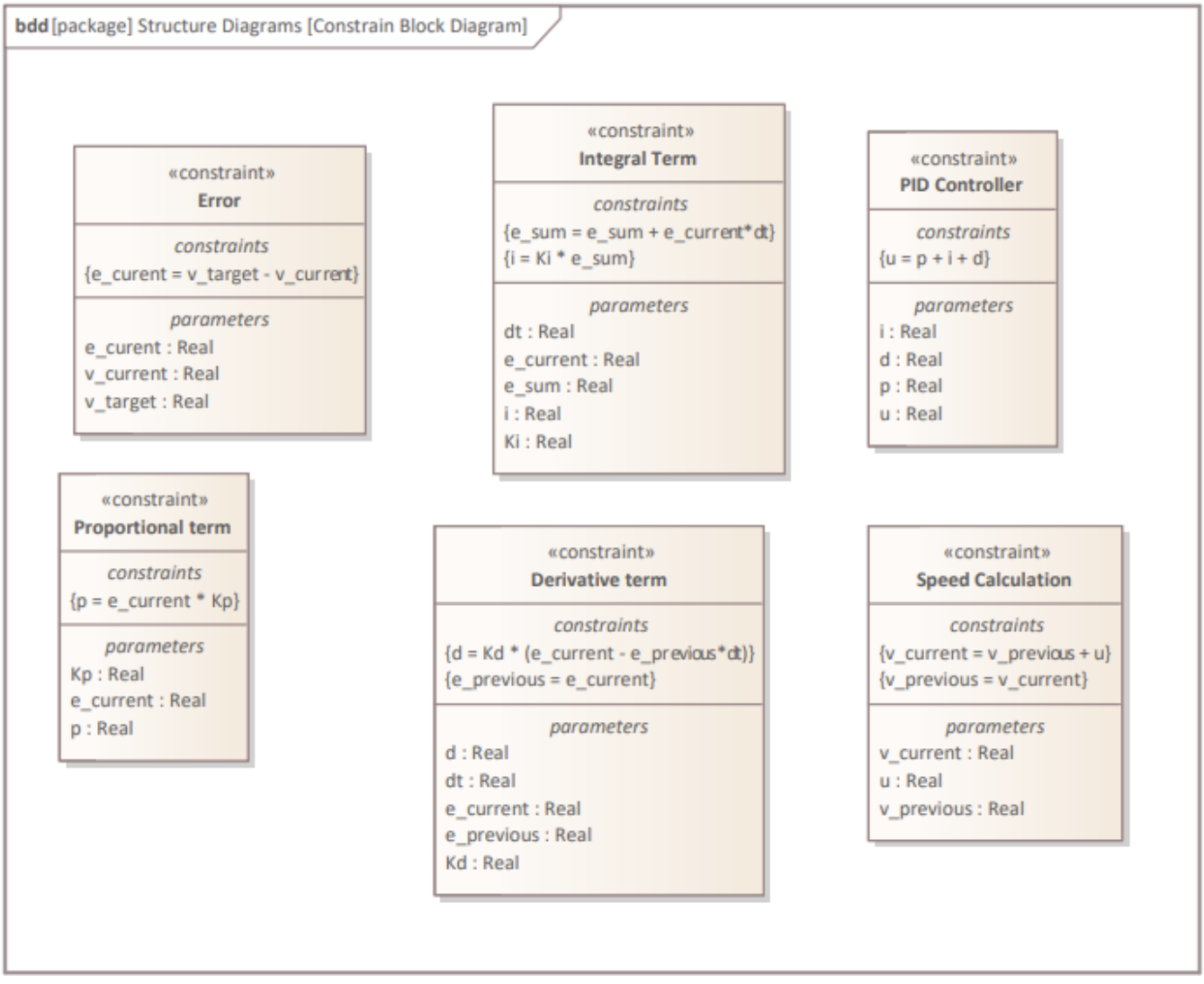

Figure 25. Constrain Blocks Diagram

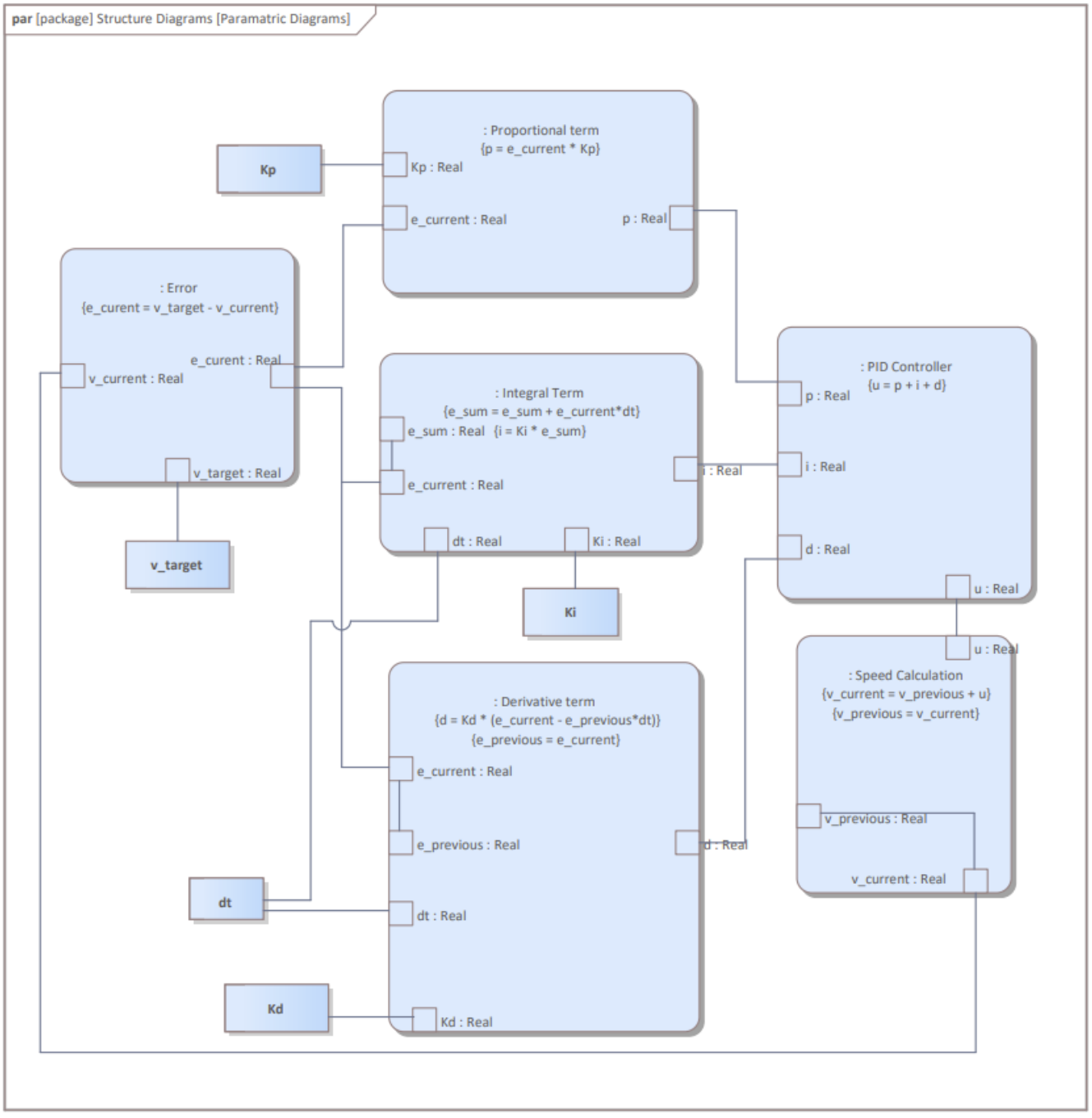

Figure 26. Parametric Diagram

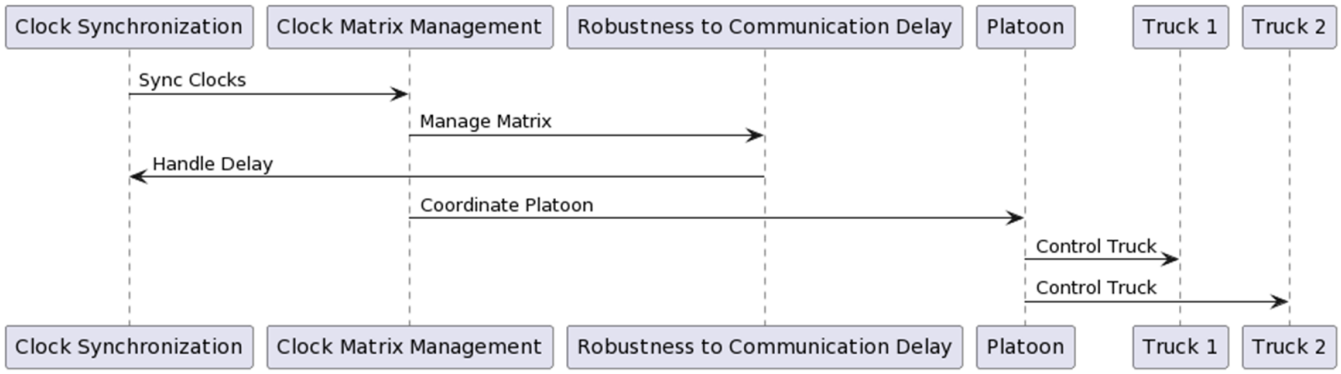

Figure 27. Sequence diagram: clock matrix pattern

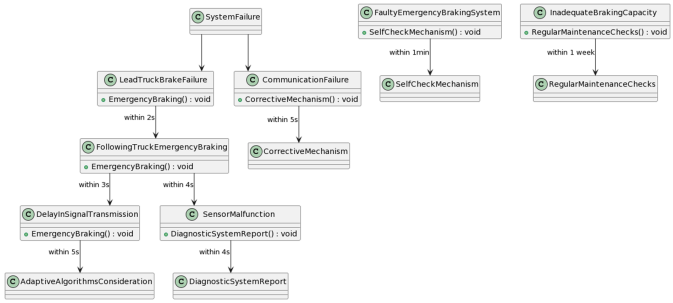

Figure 28. Testing and Validation

Model	Purpose	Target Architecture	Level of Parallelism	Programming Approach
MPI	Distributed memory systems, message passing	Multiple computers/nodes	Process-level	Explicit message passing between nodes
OpenMP	Shared memory systems, multi-core CPUs	Single machine	Thread-level	Directive-based parallelization
CUDA	GPU computing, data-parallel computations	GPUs	Thread-level	Extension to C/C++ for GPU programming

Figure 29. Comparing parallel programming models

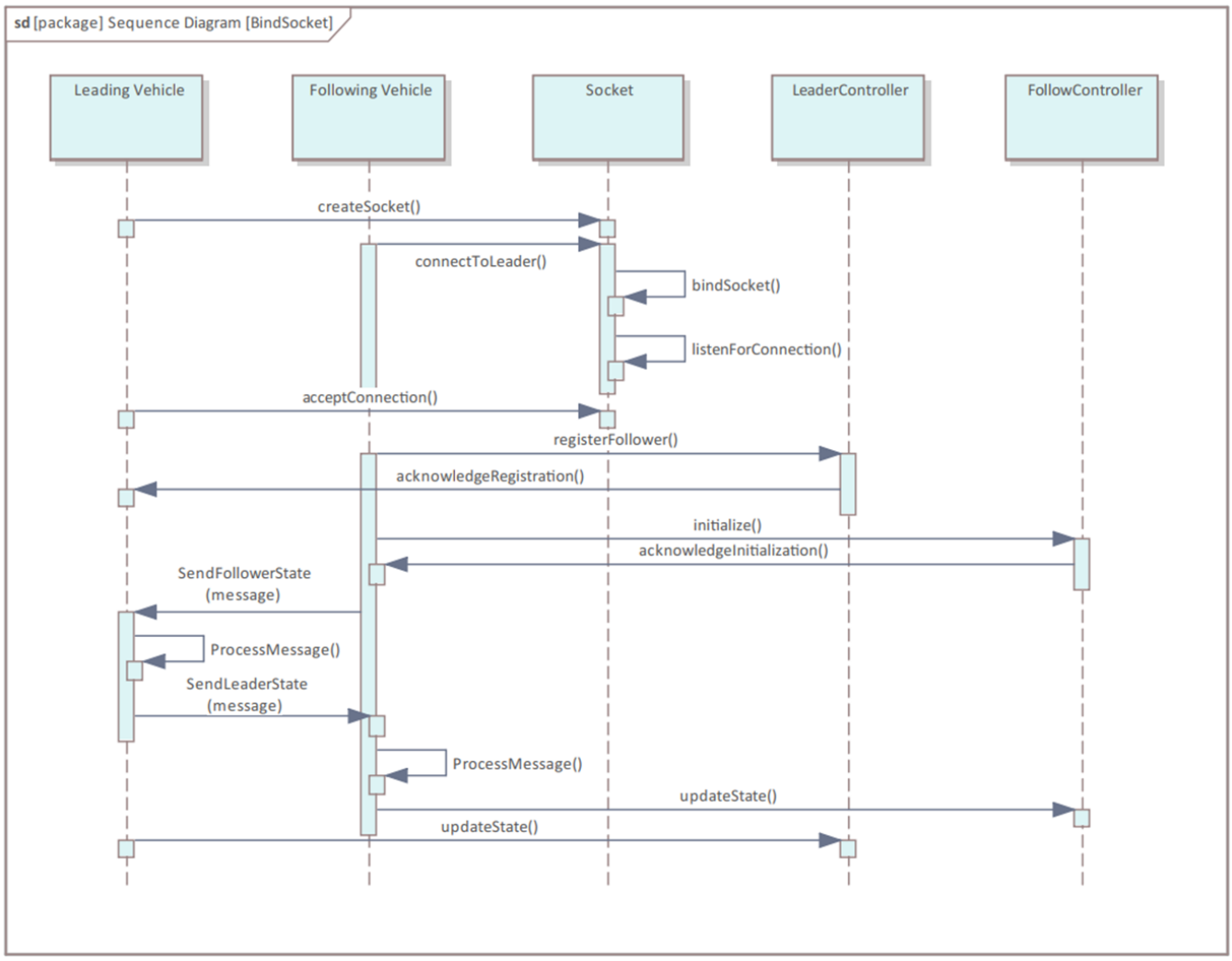

Figure 30. Simulation Sequence Diagram

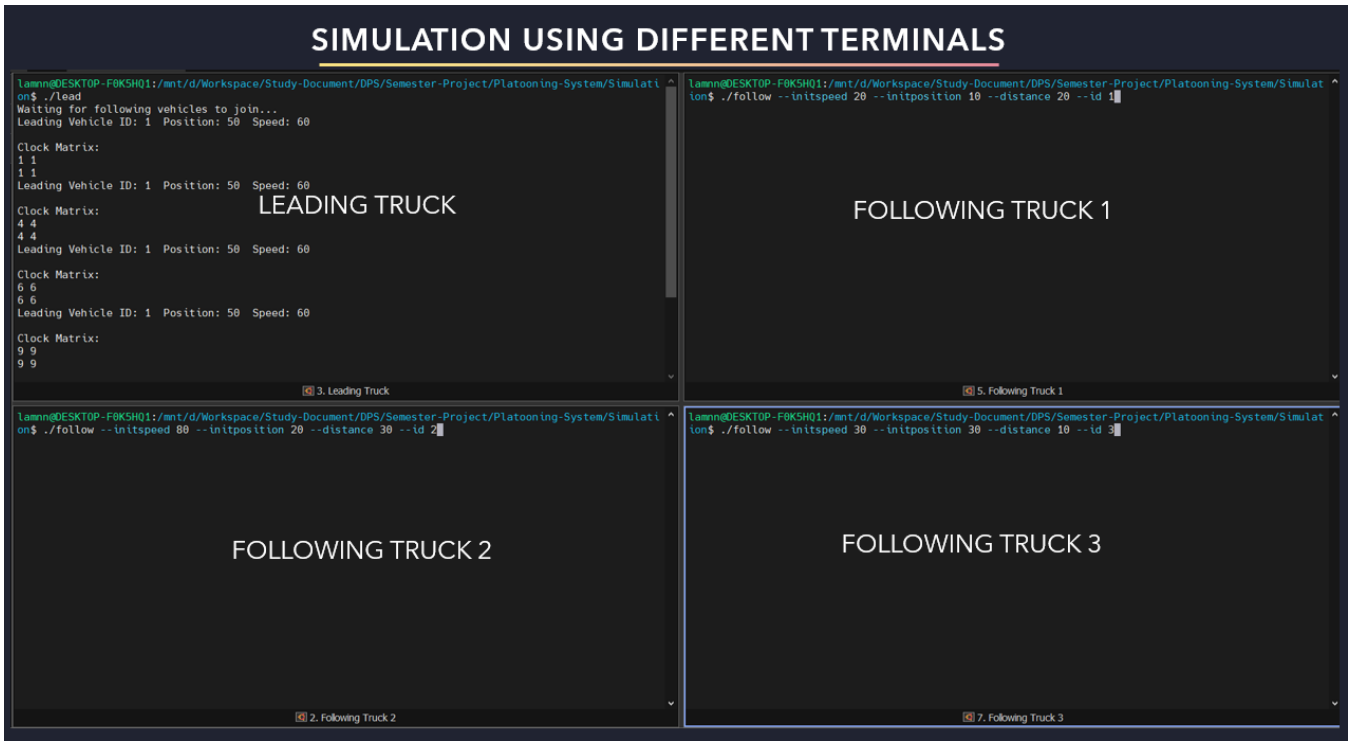

Figure 31. Simulation program running on different terminals

8.2. Code

```cpp
/**
 * @file follow.h
 *
 * @brief This is the header file for simulating the following vehicles.
 *
 * This header file contains the class declaration for the FollowingVehicle class
 * and other related structures and functions necessary for simulating a following vehicle.
 * The FollowingVehicle class represents a vehicle that follows a leading vehicle in a convoy.
 * It provides functionalities for connecting to the leader, sending and receiving messages,
 * and controlling its position and speed based on the received information.
 *
 * @ingroup DPS - FHDO
 *
 * @author [Lam Nguyen Nhat]
 *
 */

#ifndef FOLLOW_H
#define FOLLOW_H

#include <iostream>
#include <cstring>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <termios.h>
#include <unistd.h>
#include <sys/select.h>
#include <thread>
#include <omp.h>
#include <vector>

const int STDIN = 0;

const int PORT = 8080; // Port number for communication

struct Message
{
    // Define the message structure for communication
    int senderId;
    double position;
    double speed;
    bool isConnected;
    bool obstacleDetected;
    std::vector<std::vector<int>> clockMatrix;
};

class FollowingVehicle
{
private:
    int id;
    double position;
```

Figure 32. Source code for following vehicle header file

```
47
48  class FollowingVehicle
49  {
50  private:
51      int id;
52      double position;
53      double speed;
54      int serverSocket;
55      double targetDistance; // Predefined distance between vehicles
56      double Kp;             // Proportional gain of the PID controller
57      double Ki;             // Integral gain of the PID controller
58      double Kd;             // Derivative gain of the PID controller
59      double integralError;  // Integral error for the PID controller
60      double previousError;  // Previous error for the PID controller
61      bool isConnected; // Indicates if the vehicle is connected to the leader
62      bool _obstacleDetected; // Indicates if the vehicle is detecting an obstacle
63      std::vector<std::vector<int>> clockMatrix;
64
65  public:
66      /**
67       * @brief Constructor for the FollowingVehicle class.
68       *
69       * @param id The ID of the following vehicle.
70       * @param initialSpeed The initial speed of the following vehicle.
71       * @param initialPosition The initial position of the following vehicle.
72       * @param targetDistance The predefined distance between vehicles.
73       * @param Kp The proportional gain of the PID controller.
74       * @param Ki The integral gain of the PID controller.
75       * @param Kd The derivative gain of the PID controller.
76       */
77
78      FollowingVehicle(int id, double initialSpeed, double initialPosition, double targetDistance, double Kp, double Ki, double Kd);
79
80      /**
81       * @brief Getter for the ID of the following vehicle.
82       *
83       * @return The ID of the following vehicle.
84       */
85      int getId() const;
86
87      /**
88       * @brief Getter for the position of the following vehicle.
89       *
90       * @return The position of the following vehicle.
91       */
92      double getPosition() const;
93
94      /**
```

Figure 33. Source code for following vehicle header file

```
 99      double getSpeed() const;
100
101      /**
102       * @brief Setter for the position of the following vehicle.
103       *
104       * @param newPosition The new position of the following vehicle.
105       */
106      void setPosition(double newPosition);
107
108      /**
109       * @brief Setter for the speed of the following vehicle.
110       *
111       * @param newSpeed The new speed of the following vehicle.
112       */
113      void setSpeed(double newSpeed);
114
115      /**
116       * @brief Connects the following vehicle to the leader vehicle.
117       *
118       * @param ipAddress The IP address of the leading vehicle.
119       */
120      void connectToLeader(const std::string &ipAddress);
121
122      /**
123       * @brief Sends a follower message to the leader vehicle.
124       */
125      void sendFollowerMessage();
126
127      /**
128       * @brief Sends follower messages continuously at a specified interval.
129       *
130       * @param interval The interval between sending follower messages.
131       */
132      void sendFollowerMessagesContinuously(int interval);
133
134      /**
135       * @brief Receives the state information from the leader vehicle.
136       */
137      void receiveStateFromLeader();
138
139      /**
140       * @brief Receives messages from the leader vehicle continuously.
141       */
142      void receiveMessageFromLeader();
143
144      /**
145       * @brief Prints the state information of the following vehicle.
146       */
147      void printState();
148
149      /**
150       * @brief Stops the server socket connection.
151       */
152      void stopServer();
153
154      void printClockMatrix();
155  };
156
```

Figure 34. Source code for following vehicle header file

```
152         void stopServer();
153
154         void printClockMatrix();
155     };
156
157     /**
158      * @brief Sets the terminal in non-canonical mode and disables input buffering.
159      */
160     void setNonCanonicalMode();
161
162     /**
163      * @brief Restores the terminal settings.
164      */
165     void restoreTerminalSettings();
166
167     /**
168      * @brief Checks if there is keyboard input available.
169      *
170      * @return True if there is keyboard input available, false otherwise.
171      */
172     bool isKeyPressed();
173
174     /**
175      * @brief Parses the command-line arguments.
176      *
177      * @param argc The number of command-line arguments.
178      * @param argv The array of command-line arguments.
179      * @param id Reference to the ID of the following vehicle.
180      * @param initialSpeed Reference to the initial speed of the following vehicle.
181      * @param initialPosition Reference to the initial position of the following vehicle.
182      * @param targetDistance Reference to the target distance between vehicles.
183      * @param Kp Reference to the proportional gain of the PID controller.
184      * @param Ki Reference to the integral gain of the PID controller.
185      * @param Kd Reference to the derivative gain of the PID controller.
186      */
187     void parseCommandLineArguments(int argc, char *argv[], int &id, double &initialSpeed, double &initialPosition, double &targetDistance, double &Kp, double &Ki, double &Kd);
188
189     #endif // FOLLOW_H
```

Figure 35. Source code for following vehicle header file

```cpp
#include "follow.h"

FollowingVehicle::FollowingVehicle(int id, double initialSpeed, double initialPosition, double targetDistance, double Kp, double Ki, double Kd)
    : id(id), position(initialPosition), speed(initialSpeed), serverSocket(0), targetDistance(targetDistance),
      Kp(Kp), Ki(Ki), Kd(Kd), integralError(0.0), previousError(0.0)
      {
        clockMatrix = std::vector<std::vector<int>>(2, std::vector<int>(2, 0));
      }

int FollowingVehicle::getId() const
{
    return id;
}

double FollowingVehicle::getPosition() const
{
    return position;
}

double FollowingVehicle::getSpeed() const
{
    return speed;
}

void FollowingVehicle::setPosition(double newPosition)
{
    position = newPosition;
}

void FollowingVehicle::setSpeed(double newSpeed)
{
    speed = newSpeed;
}

void FollowingVehicle::connectToLeader(const std::string &ipAddress)
{
    // Create client socket and connect to the leading vehicle
    serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket < 0)
    {
        std::cerr << "Error creating client socket" << std::endl;
        return;
    }

    struct sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(PORT);

    if (inet_pton(AF_INET, ipAddress.c_str(), &(serverAddress.sin_addr)) <= 0)
    {
        std::cerr << "Invalid address/ Address not supported" << std::endl;
        return;
    }

    if (connect(serverSocket, (struct sockaddr *)&serverAddress, sizeof(serverAddress)) < 0)
    {
        std::cerr << "Connection failed" << std::endl;
        isConnected = false; // Set isConnected flag to false in case of connection failure
        return;
    }
    isConnected = true;
    std::cout << "Connected to the leading vehicle!" << std::endl;
}
```

Figure 36. Source code for following vehicle header file

```
64
65  void FollowingVehicle::sendFollowerMessage()
66  {
67      // Send follower message only if connected to the leader
68      if (isConnected)
69          {
70          // Send follower message
71          Message followerMessage;
72          followerMessage.senderId = id;
73          followerMessage.position = position;
74          followerMessage.speed = speed;
75          followerMessage.isConnected = true;
76          followerMessage.obstacleDetected = _obstacleDetected;
77
78          if (send(serverSocket, &followerMessage, sizeof(followerMessage), 0) < 0)
79          {
80              std::cerr << "Error sending follower message" << std::endl;
81          }
82      }
83      else
84      {
85          std::cerr << "Communication failure: Not connected to the leader" << std::endl;
86      }
87  }
88
89  void FollowingVehicle::sendFollowerMessagesContinuously(int interval)
90  {
91      while (true)
92      {
93          // Parallelize the function using OpenMP
94          #pragma omp parallel
95          {
96              // Each thread will execute a separate instance of the function
97              sendFollowerMessage();
98          }
99          std::this_thread::sleep_for(std::chrono::milliseconds(interval));
100     }
101 }
102
103 void FollowingVehicle::receiveStateFromLeader()
104 {
105     Message message;
106     double targetSpeed = 0;
107
108     if (!isConnected)
109     {
110         std::cerr << "Communication failure: Not connected to the leader" << std::endl;
111         return;
112     }
113
114     if (recv(serverSocket, &message, sizeof(message), 0) < 0)
115     {
116         std::cerr << "Error receiving state from leader" << std::endl;
117     }
118     else
119     {
120         clockMatrix = message.clockMatrix;
121         //  Check if obstacle is detected
122         if (_obstacleDetected)
123         {
124             std::cout << "Obstacle detected by the leader. Setting leader's speed to 0." << std::endl;
125             targetSpeed = 0.0; // Set leader's speed to 0 if obstacle is detected
126             _obstacleDetected = true;
```

Figure 37. Source code for following vehicle

```
    else
    {
        clockMatrix = message.clockMatrix;
        //  Check if obstacle is detected
        if (_obstacleDetected)
        {
            std::cout << "Obstacle detected by the leader. Setting leader's speed to 0." << std::endl;
            targetSpeed = 0.0; // Set leader's speed to 0 if obstacle is detected
            _obstacleDetected = true;
        }
        targetSpeed = message.speed;
        // Use the received speed as the targetSpeed

        // PID controller for velocity control
        double error = targetSpeed - speed;
        integralError += error;
        double derivativeError = error - previousError;
        double controlSignal = Kp * error + Ki * integralError + Kd * derivativeError;
        previousError = error;

        // Update speed
        speed += controlSignal;

        // Maintain distance between vehicles
        double distanceError = targetDistance - (message.position - position);
        double positionControlSignal = Kp * distanceError;
        position -= positionControlSignal;
    }
}

void FollowingVehicle::receiveMessageFromLeader()
{
    while (true)
    {
        if (isKeyPressed())
        {
            char c;
            std::cin.get(c);
            if (c == 'o')
            {
                std::cout << "Obstacle detected. Stopping follower vehicle." << std::endl;
                _obstacleDetected = true;
            }
        }

        // Parallelize the loop using OpenMP
        #pragma omp parallel for
        for (int i = 0; i < 100; i++)
        {
            receiveStateFromLeader();
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(100)); // Add a delay between receiving messages
    }
}

void FollowingVehicle::printClockMatrix()
{
    std::cout << "Clock Matrix:" << std::endl;
    for (const auto& row : clockMatrix)
    {
        for (const auto& element : row)
        {
```

Figure 38. Source code for following vehicle

```
174  void FollowingVehicle::printClockMatrix()
175  {
176      std::cout << "Clock Matrix:" << std::endl;
177      for (const auto& row : clockMatrix)
178      {
179          for (const auto& element : row)
180          {
181              std::cout << element << " ";
182          }
183          std::cout << std::endl;
184      }
185  }
186
187  void FollowingVehicle::printState()
188  {
189      std::cout << "Following Vehicle ID: " << id;
190      std::cout << "  Position: " << position;
191      std::cout << "  Speed: " << speed << std::endl;
192  }
193
194  void FollowingVehicle::stopServer()
195  {
196      close(serverSocket);
197  }
198
199  // Function to set the terminal in non-canonical mode and disable input buffering
200  void setNonCanonicalMode()
201  {
202      struct termios t;
203      tcgetattr(STDIN_FILENO, &t);
204      t.c_lflag &= ~(ICANON | ECHO);
205      tcsetattr(STDIN_FILENO, TCSANOW, &t);
206  }
207
208  // Function to restore the terminal settings
209  void restoreTerminalSettings()
210  {
211      struct termios t;
212      tcgetattr(STDIN_FILENO, &t);
213      t.c_lflag |= ICANON | ECHO;
214      tcsetattr(STDIN_FILENO, TCSANOW, &t);
215  }
216
217  // Function to check if there is keyboard input available
218  bool isKeyPressed()
219  {
220      fd_set readSet;
221      FD_ZERO(&readSet);
222      FD_SET(STDIN, &readSet);
223
224      struct timeval timeout;
225      timeout.tv_sec = 0;
226      timeout.tv_usec = 0;
227
228      int selectResult = select(STDIN + 1, &readSet, NULL, NULL, &timeout);
229      if (selectResult == -1)
230      {
231          std::cerr << "Error in select" << std::endl;
232          return false;
233      }
234
```

Figure 39. Source code for following vehicle

```cpp
/**
 * @file lead.h
 *
 * @brief This is the header file for simulating the leading vehicle.
 *
 * This header file contains the class declaration for the LeadingVehicle class
 * and other related structures and functions necessary for simulating a leading vehicle.
 * The LeadingVehicle class represents the vehicle that leads a convoy of following vehicles.
 * It provides functionalities for starting and stopping a server, accepting and handling
 * connections from followers, and sending state information to the followers.
 *
 * @ingroup DPS - FHDO
 *
 * @author [Lam Nguyen Nhat]
 *
 */

#ifndef LEAD_H
#define LEAD_H

#include <iostream>
#include <vector>
#include <map>
#include <cstring>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <thread>
#include <algorithm>
#include <omp.h>
#include <vector>

const int PORT = 8080;

struct Message
{
    int senderId;
    double position;
    double speed;
    bool isConnected;
    bool obstacleDetected;
    std::vector<std::vector<int>> clockMatrix;
};

class LeadingVehicle
{
private:
    int id;
    double position;
    double speed;
    std::vector<int> followers;
    int serverSocket;
    std::vector<int> followerId;
    std::map<int, int> followersPosition;
    std::map<int, int> followersSpeed;
    bool _obstacleDetected;
    std::vector<std::vector<int>> clockMatrix;

public:
    /**
     * @brief Constructor for the LeadingVehicle class.
     *
     * @param id The ID of the leading vehicle.
```

Figure 40. Source code for following vehicle

```
58
59  public:
60      /**
61       * @brief Constructor for the LeadingVehicle class.
62       *
63       * @param id The ID of the leading vehicle.
64       * @param initialPosition The initial position of the leading vehicle.
65       * @param initialSpeed The initial speed of the leading vehicle.
66       */
67      LeadingVehicle(int id, double initialPosition, double initialSpeed);
68
69      /**
70       * @brief Getter for the ID of the leading vehicle.
71       *
72       * @return The ID of the leading vehicle.
73       */
74      int getId() const;
75
76      /**
77       * @brief Getter for the position of the leading vehicle.
78       *
79       * @return The position of the leading vehicle.
80       */
81      double getPosition() const;
82
83      /**
84       * @brief Getter for the speed of the leading vehicle.
85       *
86       * @return The speed of the leading vehicle.
87       */
88      double getSpeed() const;
89
90      /**
91       * @brief Starts the server for the leading vehicle.
92       */
93      void startServer();
94
95      /**
96       * @brief Stops the server for the leading vehicle.
97       */
98      void stopServer();
99      void incrementClockMatrix(int value);
100     void printClockMatrix();
101
102 private:
103     void createServerSocket();
104     void acceptFollowers();
105     void addFollower(int followerSocket);
106     void sendMessageToFollowers();
107     void sendStateToFollowers();
108     void handleFollowerConnection(int clientSocket);
109     void printState();
110 };
111
112 #endif // LEAD_H
```

Figure 41. Source code for following vehicle

```
// Use the received speed as the targetSpeed
double targetSpeed = message.speed;
double leaderPosition = message.position;

// Allocate device memory for speed and position
double* deviceSpeed;
double* devicePosition;
cudaMalloc((void**)&deviceSpeed, sizeof(double));
cudaMalloc((void**)&devicePosition, sizeof(double));

// Copy speed and position data to device memory
cudaMemcpy(deviceSpeed, &speed, sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(devicePosition, &position, sizeof(double), cudaMemcpyHostToDevice);

// Call the CUDA kernel for speed and distance calculation
calculateSpeedAndDistance<<<1, 1>>>(deviceSpeed, devicePosition, leaderPosition, targetSpeed, targetDistance, Kp, Ki, Kd);

// Copy the results back to the host
cudaMemcpy(&speed, deviceSpeed, sizeof(double), cudaMemcpyDeviceToHost);
cudaMemcpy(&position, devicePosition, sizeof(double), cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(deviceSpeed);
cudaFree(devicePosition);
```

Figure 42. CUDA memory allocation for PID Calculation

```
__global__ void calculateSpeedAndDistance(double* deviceSpeed, double* devicePosition, double leaderPos
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    // Calculate speed
    double error = targetSpeed - deviceSpeed[index];
    double integralError = 0.0; // Assume integralError and previousError are stored in device memory
    double previousError = 0.0;
    double derivativeError = error - previousError;
    double controlSignal = Kp * error + Ki * integralError + Kd * derivativeError;
    deviceSpeed[index] += controlSignal;

    // Calculate distance
    double distanceError = targetDistance - (leaderPosition - devicePosition[index]);
    double positionControlSignal = Kp * distanceError;
    devicePosition[index] -= positionControlSignal;
}
```

Figure 43. CUDA kernel function for PID Calculation

```cpp
/**
 * @file lead.h
 *
 * @brief This is the header file for simulating the leading vehicle.
 *
 * This header file contains the class declaration for the LeadingVehicle class
 * and other related structures and functions necessary for simulating a leading vehicle.
 * The LeadingVehicle class represents the vehicle that leads a convoy of following vehicles.
 * It provides functionalities for starting and stopping a server, accepting and handling
 * connections from followers, and sending state information to the followers.
 *
 * @ingroup DPS - FHDO
 *
 * @author [Lam Nguyen Nhat]
 *
 */

#ifndef LEAD_H
#define LEAD_H

#include <iostream>
#include <vector>
#include <map>
#include <cstring>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <thread>
#include <algorithm>
#include <omp.h>
#include <vector>

const int PORT = 8080;

struct Message
{
    int senderId;
    double position;
    double speed;
    bool isConnected;
    bool obstacleDetected;
    std::vector<std::vector<int>> clockMatrix;
};

class LeadingVehicle
{
private:
    int id;
    double position;
    double speed;
    std::vector<int> followers;
    int serverSocket;
    std::vector<int> followerId;
    std::map<int, int> followersPosition;
    std::map<int, int> followersSpeed;
    bool _obstacleDetected;
    std::vector<std::vector<int>> clockMatrix;

public:
    /**
     * @brief Constructor for the LeadingVehicle class.
     *
     * @param id The ID of the leading vehicle.
```

Figure 44. Source code for leading vehicle

```
58
59  public:
60      /**
61       * @brief Constructor for the LeadingVehicle class.
62       *
63       * @param id The ID of the leading vehicle.
64       * @param initialPosition The initial position of the leading vehicle.
65       * @param initialSpeed The initial speed of the leading vehicle.
66       */
67      LeadingVehicle(int id, double initialPosition, double initialSpeed);
68
69      /**
70       * @brief Getter for the ID of the leading vehicle.
71       *
72       * @return The ID of the leading vehicle.
73       */
74      int getId() const;
75
76      /**
77       * @brief Getter for the position of the leading vehicle.
78       *
79       * @return The position of the leading vehicle.
80       */
81      double getPosition() const;
82
83      /**
84       * @brief Getter for the speed of the leading vehicle.
85       *
86       * @return The speed of the leading vehicle.
87       */
88      double getSpeed() const;
89
90      /**
91       * @brief Starts the server for the leading vehicle.
92       */
93      void startServer();
94
95      /**
96       * @brief Stops the server for the leading vehicle.
97       */
98      void stopServer();
99      void incrementClockMatrix(int value);
100     void printClockMatrix();
101
102 private:
103     void createServerSocket();
104     void acceptFollowers();
105     void addFollower(int followerSocket);
106     void sendMessageToFollowers();
107     void sendStateToFollowers();
108     void handleFollowerConnection(int clientSocket);
109     void printState();
110 };
111
112 #endif // LEAD_H
```

Figure 45. Source code for leading vehicle

```cpp
#include "lead.h"

LeadingVehicle::LeadingVehicle(int id, double initialPosition, double initialSpeed)
    : id(id), position(initialPosition), speed(initialSpeed), serverSocket(0)
    {
        // Initialize the clock matrix with zeros
        clockMatrix = std::vector<std::vector<int>>(2, std::vector<int>(2, 0));
    }

int LeadingVehicle::getId() const
{
    return id;
}

double LeadingVehicle::getPosition() const
{
    return position;
}

double LeadingVehicle::getSpeed() const
{
    return speed;
}

void LeadingVehicle::incrementClockMatrix(int value)
{
    // Increment each element in the clock matrix by the given value
    for (auto& row : clockMatrix)
    {
        for (auto& element : row)
        {
            element += value;
        }
    }
}

void LeadingVehicle::printClockMatrix()
{
    std::cout << "Clock Matrix:" << std::endl;
    for (const auto& row : clockMatrix)
    {
        for (const auto& element : row)
        {
            std::cout << element << " ";
        }
        std::cout << std::endl;
    }
}

void LeadingVehicle::startServer()
{
    createServerSocket();

    std::cout << "Waiting for following vehicles to join..." << std::endl;

    std::thread followerThread(&LeadingVehicle::acceptFollowers, this);
    followerThread.detach();

    std::thread sendThread(&LeadingVehicle::sendMessageToFollowers, this);
    sendThread.detach();
}

void LeadingVehicle::stopServer()
```

Figure 46. Source code for leading vehicle

```
63  void LeadingVehicle::stopServer()
64  {
65      close(serverSocket);
66  }
67
68  void LeadingVehicle::createServerSocket()
69  {
70      serverSocket = socket(AF_INET, SOCK_STREAM, 0);
71      if (serverSocket < 0)
72      {
73          std::cerr << "Error creating server socket" << std::endl;
74          exit(EXIT_FAILURE);
75      }
76
77      struct sockaddr_in serverAddress;
78      serverAddress.sin_family = AF_INET;
79      serverAddress.sin_addr.s_addr = INADDR_ANY;
80      serverAddress.sin_port = htons(PORT);
81
82      if (bind(serverSocket, (struct sockaddr *)&serverAddress, sizeof(serverAddress)) < 0)
83      {
84          std::cerr << "Error binding server socket" << std::endl;
85          exit(EXIT_FAILURE);
86      }
87
88      if (listen(serverSocket, 1) < 0)
89      {
90          std::cerr << "Error listening on server socket" << std::endl;
91          exit(EXIT_FAILURE);
92      }
93  }
94
95  void LeadingVehicle::acceptFollowers()
96  {
97      while (true)
98      {
99          int clientSocket = accept(serverSocket, nullptr, nullptr);
100         if (clientSocket < 0)
101         {
102             std::cerr << "Error accepting client connection" << std::endl;
103             continue;
104         }
105         addFollower(clientSocket);
106         std::thread followerThread(&LeadingVehicle::handleFollowerConnection, this, clientSocket);
107         followerThread.detach();
108     }
109 }
110
111 void LeadingVehicle::addFollower(int followerSocket)
112 {
113     followers.push_back(followerSocket);
114 }
115
116 void LeadingVehicle::sendMessageToFollowers()
117 {
118     while (true)
119     {
120         sendStateToFollowers();
```

Figure 47. Source code for leading vehicle

```
116  void LeadingVehicle::sendMessageToFollowers()
117  {
118      while (true)
119      {
120          sendStateToFollowers();
121          printState();
122          std::this_thread::sleep_for(std::chrono::milliseconds(300));
123      }
124  }
125
126  void LeadingVehicle::sendStateToFollowers()
127  {
128      Message message;
129      message.senderId = id;
130      message.position = position;
131      message.speed = speed;
132      // message.clockMatrix = clockMatrix;
133
134      // Parallelize the loop using OpenMP
135      #pragma omp parallel for
136      for (int followerSocket : followers)
137      {
138          if (send(followerSocket, &message, sizeof(message), 0) < 0)
139          {
140              std::cerr << "Error sending state to follower" << std::endl;
141          }
142      }
143  }
144  void LeadingVehicle::handleFollowerConnection(int clientSocket)
145  {
146      while (true)
147      {
148          Message followerMessage;
149          int bytesRead = recv(clientSocket, &followerMessage, sizeof(followerMessage), 0);
150          if (bytesRead <= 0)
151          {
152              auto it = std::find(followers.begin(), followers.end(), clientSocket);
153              if (it != followers.end())
154              {
155                  followers.erase(it);
156                  std::cout << "Follower ID " << followerMessage.senderId << " is disconnected" << std::endl;
157
158                  // Remove the disconnected vehicle Id from the vector
159                  auto it2 = std::remove(followerId.begin(), followerId.end(), followerMessage.senderId);
160                  followerId.erase(it2, followerId.end());
161              }
162              break;
163          }
164
165          if (std::find(followerId.begin(), followerId.end(), followerMessage.senderId) == followerId.end())
166          {
167              followerId.push_back(followerMessage.senderId);
168          }
169
170          followersPosition[followerMessage.senderId] = followerMessage.position;
171          followersSpeed[followerMessage.senderId] = followerMessage.speed;
172
173          _obstacleDetected = followerMessage.obstacleDetected;
174          // Check if obstacle is detected by the follower
```

Figure 48. Source code for leading vehicle

```
            followersPosition[followerMessage.senderId] = followerMessage.position;
            followersSpeed[followerMessage.senderId] = followerMessage.speed;

            _obstacleDetected = followerMessage.obstacleDetected;
            // Check if obstacle is detected by the follower
            if (_obstacleDetected)
            {
                std::cout << "Obstacle Detected by the follower" << std::endl;
                while(speed>0)
                {
                    speed--; // Set leader's speed to 0 if obstacle is detected by the follower
                    std::this_thread::sleep_for(std::chrono::milliseconds(10));
                }
            }

        }
    }

    void LeadingVehicle::printState()
    {
        std::cout << "Leading Vehicle ID: " << id;
        std::cout << "  Position: " << position;
        std::cout << "  Speed: " << speed << std::endl;

        #pragma omp parallel for
        for (int iD : followerId)
        {
            if (_obstacleDetected)
            {
                std::cout << "Follower ID " << iD << " is facing an obstacle" << std::endl;
            }
            std::cout << "Connected Follower ID: " << iD;
            std::cout << "  Position: " << followersPosition[iD];
            std::cout << "  Speed: " << followersSpeed[iD] << std::endl;
        }
        std::cout << std::endl;

        printClockMatrix();
    }

    int main()
    {
        LeadingVehicle leadingVehicle(1, 50, 60.0);
        leadingVehicle.startServer();

        while (true)
        {
            #pragma omp parallel for
            // Increment the clock matrix by 1
            leadingVehicle.incrementClockMatrix(1);
            std::this_thread::sleep_for(std::chrono::milliseconds(100));

            // Perform other tasks in the main function, if any
        }

        leadingVehicle.stopServer();
        return 0;
    }
```

Figure 49. Source code for leading vehicle

8.3. Git Overview

8.3.1. Lines of code. [h]

Table 1. LINE COUNTS OF CODES

File	Line Count
`follow.h - header file for following vehicle`	189
`follow.cpp - source file for following vehicle`	310
`lead.h - header file for leading vehicle`	112
`lead.cpp - source file for leading vehicle`	227
`follow.cu - cuda file for following vehicle`	55

8.3.2. Number of submits per person.

1) Nhat Lam Nguyen: 838 lines (simulation program)
2) Nhat Quang Nguyen: 55 lines (adding CUDA kernel on simulation program)
3) Hadis Mohammadi:150 (implementing clock synchronization on the code)

8.3.3. Number of commits per person.

1) Nhat Lam Nguyen: 10 commits - (simulation program + EA diagram implementation)
2) Nhat Quang Nguyen: 5 commits - (simulation program with CUDA + PID Controller for Distance Maintaining + Constrain Blocks and Parametric Diagram)
3) Hadis Mohammadi: 4 commits - (clock synchronization implemented by clock matrix - Diagram implementation)

8.3.4. Structure. There are three main folders inside the repository:

- Diagrams: Contains Enterprise Architect diagrams implementation
- Simulation: Contains sources code for simulation program

8.3.5. Contribution by each member.

- Nhat Lam Nguyen:
  - Contribute: 40%
  - Work estimate (in hours): 30 hours
  - Tasks: Design the architecture of the project, implement Enterprise Architecture Diagrams, communication network, analysis and implement simulation program.
- Nhat Quang Nguyen:
  - Contribute: 20%
  - Work estimate (in hours): 12 hours
  - Tasks: Code PID Controller with CUDA in simulation program, Controllers for Distance Maintaining, Constrain Blocks and Parametric Diagram.
- Hadis Mohammadi
  - Contribute: 20%
  - Work estimate (in hours): 12 hours
  - Tasks: analysis clock matrix, implement clock matrix using matrix pattern
- Shirin Babaeikouros
  - Contribute: 20%
  - Work estimate (in hours): 12 hours
  - Tasks: Testing Approaches, comparison between socket communication, DSRC, and C-V2X as an evaluation of their respective advantages