

# Truck Platooning: Modelling and Implementation with Concurrent and Distributed Architecture

Akhilesh Kakkayamkode  
Department of Computer Science  
Fachhochschule Dortmund  
Dortmund, Germany  
akhilesh.kakkayamkode001@stud.fh-dortmund.de

Angel Mary  
Department of Computer Science  
Fachhochschule Dortmund  
Dortmund, Germany  
angel.mary014@stud.fh-dortmund.de

Arjun Veeramony  
Department of Computer Science  
Fachhochschule Dortmund  
Dortmund, Germany  
arjun.veeramony001@stud.fh-dortmund.de

**Abstract**— Truck platooning, an innovative transportation concept, involves a string of trucks driving closely together with minimal time gaps. The lead truck is operated by a driver, while the follower trucks operate under a master-slave control system. The main objective is to reduce fuel consumption by connecting the trucks through a client-server network and distributing the leader's trajectory. The close proximity of the trucks in the platoon formation optimizes aerodynamics, resulting in more efficient airflow and reduced overall energy consumption. Furthermore, truck platooning enhances road safety and reduces congestion and labor costs due to the absence of human intervention in the following vehicles. This paper aims to contribute to the understanding and advancement of truck platooning by exploring various aspects, including underlying principles, system architecture, and communication protocols. Through this comprehensive investigation, we aim to advance the understanding and implementation of truck platooning, paving the way for its widespread adoption and integration into contemporary transportation systems.

**Keywords**—*Platooning, leading truck, following truck, architecture*

## I. INTRODUCTION

Revolutionizing the conventional landscape of freight transportation, truck platooning seamlessly integrates wireless technology and automated driving systems, forming a cohesive convoy led by a single driver in the Leading Truck. This cutting-edge paradigm aims to optimize operations by reducing the dependence on individual drivers in each truck. At its essence, our methodology employs a Master-Slave model, where the Leading Truck assumes the role of the master, orchestrating the movements of the follower trucks in a synchronized manner. This decentralized architecture facilitates effective communication and task allocation among the trucks, ultimately enhancing the overall performance of the platoon. To ensure robust communication, we employ TCP (Transmission Control Protocol) as the foundation for data exchange between the trucks, ensuring reliability and stability.

Moreover, our project places a substantial emphasis on concurrent programming with OpenMP, allowing for parallel task execution and improving the responsiveness and efficiency of the truck platooning system. Additionally, this paper delves into GPU implementation using OpenCL to adjust speeds for following trucks, unlocking the computational power of graphics processing units. This comprehensive exploration contributes to advancing the understanding and implementation of truck platooning, ushering in a new era of efficiency and innovation in freight transportation.

## II. MOTIVATION

The primary aim of this project is to revolutionize transportation systems through the implementation of cutting-edge technology in truck platooning. By employing a master-slave configuration, seamless connectivity among trucks is established, enabling simultaneous task execution and communication. In the event of communication anomalies, immediate measures are taken to rectify them, and if unsuccessful, the platoon comes to a halt immediately, prioritizing safety. Continuous monitoring of distances between trucks ensures the integrity of the platoon, while adaptive behaviors respond dynamically to environmental changes. Furthermore, advanced obstacle detection systems are implemented to mitigate collision risks, emphasizing safety and efficiency.

## III. SKETCH OF APPROACH

### A. UML Model for Truck Platooning- Behavior diagrams

Figures Fig 1 to Fig 5 & (please refer Appendix A1) represents the behavioural structure for the design of the truck platooning. Here only two trucks were considered one leading truck and one following truck. Desired number of following trucks can be connected as per the design in real time system and all of these will be the replica of first following truck. As it will be explained in the following sections, a TCP/IP socket communication was implemented for the system, as well as a Master-Slave architecture, where the leading truck has driver and it takes most of the computation and following truck will follow it. Following truck also send real-time or emergency information back and lead truck will process data accordingly therefore threading method is used to implemented it.

### 1) Activity Diagram

Fig 1 shows the activity diagram of leading and following truck. From the initial point each truck in a platoon needs a way of transmitting signals electronically to all following vehicles i.e. communication establishment. System is designed to always pass a communication signal between trucks in every 0.1s to ensure proper connectivity.

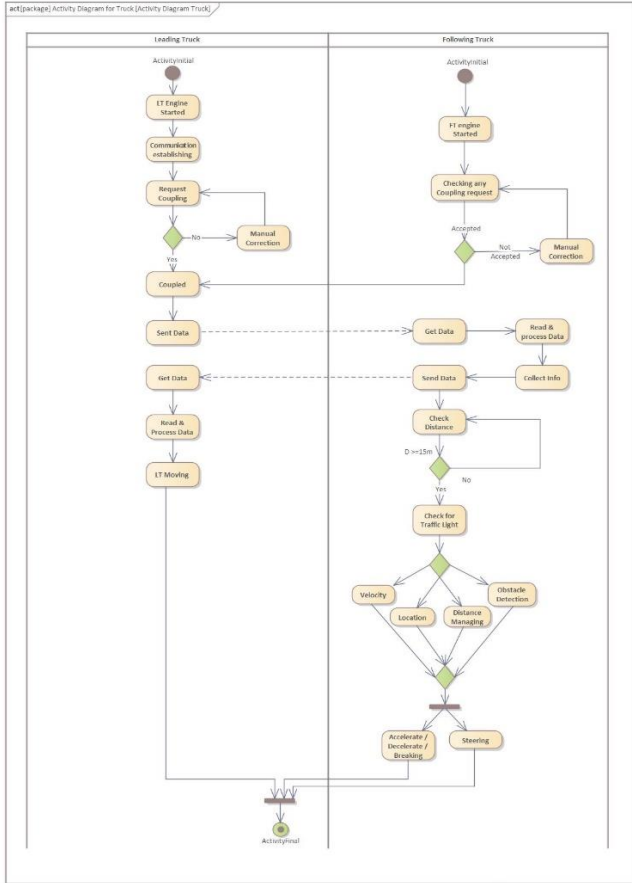


Fig 1. Activity diagram of Trucks

Only via this mechanism, it is possible to reduce the following distance significantly and get coupled to each other. In coupling process lead truck sends coupling broadcast message and following truck checks the ID match and accept it. Once coupled, sensors on lead truck collect real time data and transmit it to following truck. Following truck receives data, read, process and execute it in the system. The data include mainly current location, current velocity, steering angle, and if any emergency alerts. Each following truck sends its status information, including velocity, acceleration, and other relevant parameters, to the leading truck. The leading truck collects and processes this information to make decisions regarding its own behavior and to understand the state of each following truck in the platoon. Fig A1 (Appendix) will depict the communication of trucks.

From the halt position following truck starts moving when the distance to the lead truck/ preceding truck exceeds 15m and no obstacle condition detects. Then it increases the acceleration to catch up with the lead one and follows up to the destination.

### 2) State Machine diagram of Leading Truck.

Fig 2 denotes the State Machine diagram of leading truck. In this, the driver is responsible for the behaviour, so its state machine diagram is quite simple. All trucks have the same basic “moving” states: stop, acceleration, constant velocity, turn and deceleration which all should be communicated properly. The leading truck is often equipped with advanced sensors, such as radar and cameras. These sensors help the truck gather real-time data about the road conditions, traffic, and other obstacles. The collected data is then shared with the following trucks, allowing them to adjust their behaviour accordingly. The basic principle comes as follows: the behaviour of the leading truck, meaning direction, velocity, position, and lane, are saved in a list at every few positions, received data from other trucks is also processed and while executing it, lead truck simultaneously transmit it to other trucks. This way, the behaviour at that specific position is known. The leading truck initiates manoeuvres such as lane changes, exits, turn and merges. The following trucks adapt their movements based on the actions of the leader, ensuring a coordinated transition.

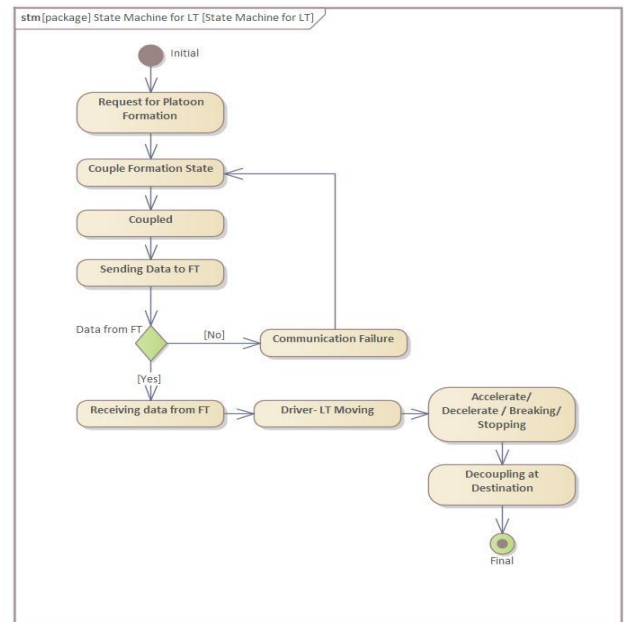


Fig 2. State Machine Diagram of Leading Truck

### 3) State Machine diagram of Following Truck

Fig 3 depicts the State Machine diagram of following truck. In Following truck, after coupling it starts receiving and transmitting data to lead truck as well as to other following truck. The communication and data exchange among trucks are critical for maintaining synchronization and responding to changes in the convoy's dynamics. That way, each truck in the platoon gets the most updated status of the truck ahead. If any point data misses, truck will go to communication failure state. In the normal working, it will go through different states like Acceleration (velocity of following truck, constant velocity, deceleration and Halt.

The logic works as follows:

- if the velocity of following truck is less than the velocity of leading truck ( $FT\_V < LT\_V$ ), the truck will go to an acceleration state;

- if the velocity of following truck is nearly same the velocity of leading truck ( $FT\_V = LT\_V$ ), the truck will go to a constant velocity state;
- if the velocity of following truck is greater than the velocity of leading truck ( $FT\_V > LT\_V$ ), the truck will go to a deceleration state;
- if the velocity of leading truck is zero, the truck will go to halt state.

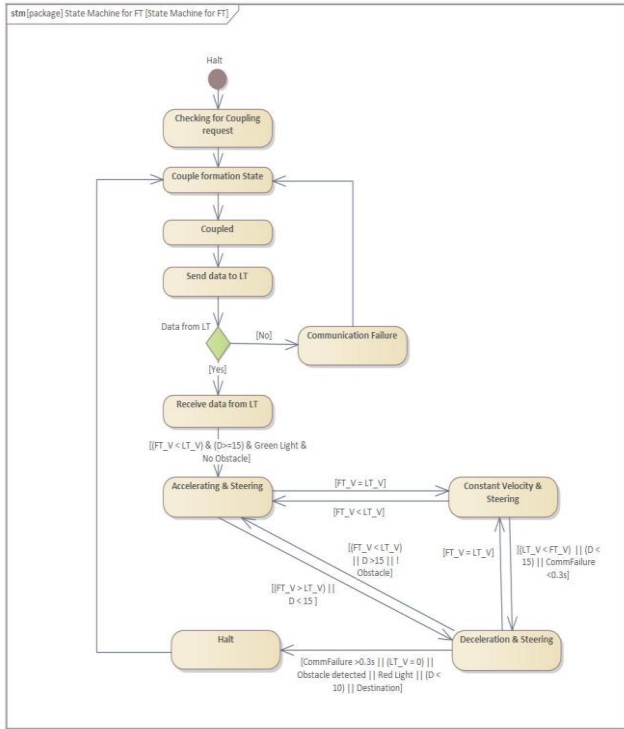


Fig 3. State Machine Diagram of Following Truck

The described states showcase how the following truck continuously adjusts its speed to match the leading truck, ensuring a coordinated and safe platooning experience. This level of automation and coordination is facilitated by advanced technologies such as V2V communication, adaptive cruise control, and sensor systems that enable trucks to operate closely together in a platoon.

#### 4) Activity Diagram of Distance Management.

Fig 4 depicts the activity diagram of distance management. As inputs it requires the position and the velocity of the truck ahead, which are sent by the leading truck, as well as its own position and velocity. Here the optimum distance considered between each truck is 15m.

The logic works as follows:

- if the distance between the current truck and the truck ahead is greater than 15, the truck will go to an acceleration state;
- if the distance between the trucks is nearly 15m it is considered that the trucks have an acceptable distance. Therefore, the truck will maintain its current velocity;
- if the distance between the trucks is lesser than 15, the truck will deaccelerate;
- if the distance between the truck or any vehicle forwards is lesser than 10 or any obstacle detected or the traffic light is red, the truck will immediately stop.

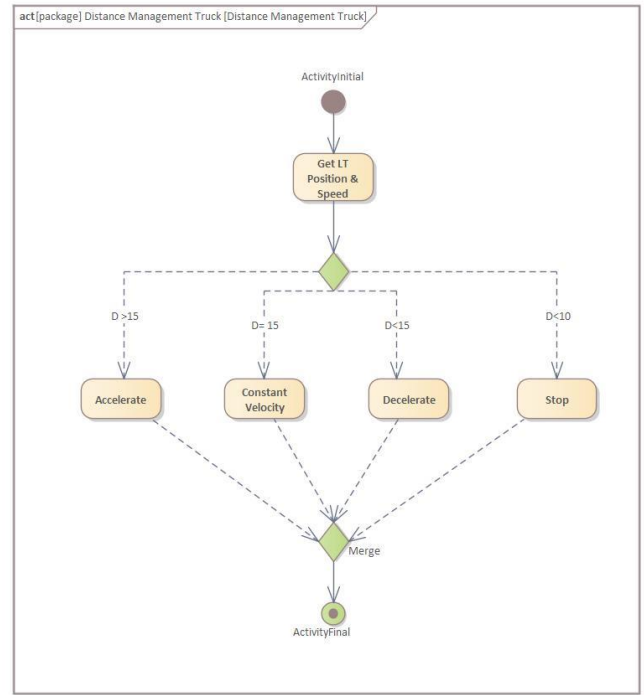


Fig 4. Activity diagram of distance management

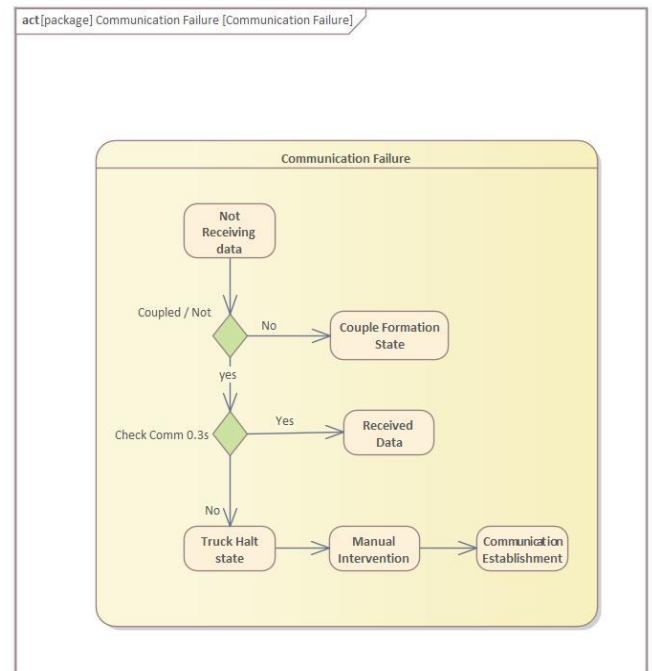


Fig 5. Activity diagram of Communication failure

#### 5) Activity Diagram of Communication Failure.

Fig 5 depicts the activity diagram of communication failure state. The several causes of Communication Failure include technical issues like faults or malfunctions in the communication hardware or software of the trucks or environmental Interference like as electromagnetic interference, radio signal obstruction or due to network congestion may disrupt the wireless communication between the trucks. It will lead to loss of synchronization. If the system enters communication failure state, it first check whether trucks are still coupled or not, if not it system will go to couple formation state, if yes then system will provide a 0.3s

time to regain connection and truck will be stopped if it is not able establish connection within the given time. Then system seek for manual intervention to take over the truck.

#### IV. THE ARCHITECTURE CONCEPT

The utilization of distributed architecture is instrumental in attaining the predefined objectives, as it involves seamless coordination among various components dispersed across diverse platforms. The truck platooning system operates on the foundational principles of distributed architecture, wherein each truck is equipped with its dedicated information processing system.

##### A. Architectural Patterns

In the initial stages of our project, we contemplated two parallel programming approaches: the Node-only model and the Master-Slave model. The critical factor guiding our decision to choose the Master-Slave model for the truck platooning scenario lies in the need for centralized coordination. Given the hierarchical nature of truck platooning, where a leading truck directs the movements of followers, the Master-Slave architecture facilitates efficient command transmission and ensures synchronized execution, making it a suitable choice.

##### B. Communication Protocol

For our communication protocol, we opted for TCP over other alternatives due to its reliability and connection-oriented nature. In the dynamic and safety-critical environment of truck platooning, maintaining a stable and dependable connection is paramount. TCP's ability to establish and sustain connections reduces the risk of data loss or errors during the exchange of commands and data between the leading and following trucks, aligning perfectly with the real-time demands of truck platooning operations.

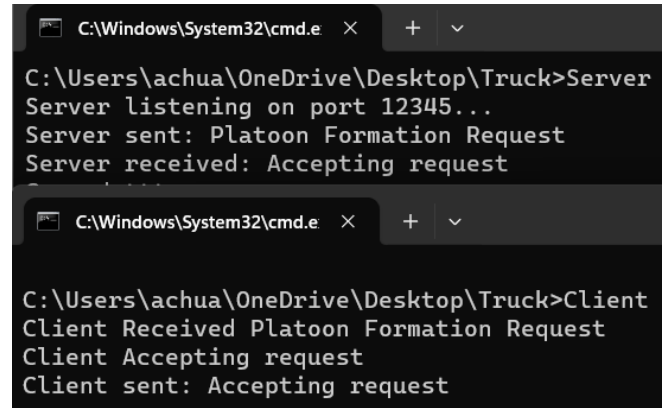
#### V. IMPLEMENTATION

We established a simple TCP/IP socket system communication and implemented a Master-Slave structure in the system, where the primary truck handles most computations, so threading processes is mainly implemented on it. The number of threads is directly proportional with the number of trucks in the convoy, with each thread managing one truck's data. All trucks adhere to the same basic movement states such as stopping, accelerating, constant velocity, turning, and decelerating.

However, their paths to these states differ. The driver controls the leading truck's actions, while the others operate differently. The concept is simple: the leading truck's actions, like direction and speed, are recorded for each move and immediately shared with the following trucks. This ensures they mimic the lead truck's behavior, checking for obstacles and maintaining a safe distance at the same time. In our project, the input behavior of the lead truck, which is basically controlled by the input from the driver is simulated by reading the Strings line by line from the text file. So, the above mentioned implementation of the truck platoon system is done in different phases.

##### 1) Server and Client TCP/IP Socket Communication

In this phase we have initially created 2 nodes, and established TCP/IP communication between them. Here, one node will act as server and other as the client. Initially the Server/ lead truck is created and is bind with a specific IP address and port. The client/follower truck establishes the connection to the server by connecting to the same IP address and port number used by server. Followed by the server sending a "Platoon Formation Request" to the client, and the client accepting it. The Output is shown in Fig 6.



```
C:\Windows\System32\cmd.e  X  +  v
C:\Users\achua\OneDrive\Desktop\Truck>Server
Server listening on port 12345...
Server sent: Platoon Formation Request
Server received: Accepting request

C:\Windows\System32\cmd.e  X  +  v
C:\Users\achua\OneDrive\Desktop\Truck>Client
Client Received Platoon Formation Request
Client Accepting request
Client sent: Accepting request
```

Fig. 6 Server and Client TCP Communication

##### 2) Configuring the lead truck

In the second phase of the project, we focused on setting up the leading truck. This truck is responsible for implementing the behavior outlined in a text file. It reads the file line by line, moving accordingly based on the instructions given. When it reaches the final destination indicated by the "Stop" message, the truck stops and stays put. Additionally, the leading truck records its every behavior sequentially so that it can be communicated to its follower trucks. Corresponding output is shown in Fig. 7.

##### 3) Configuring the following trucks

In the next phase of our project, we focused on implementing the following truck to travel alongside the leading truck. This involved replicating the behavior of the leading truck by sending messages from the leading truck to the following truck, with the leading truck acting as a server and the following truck as a client. We designed the system to support multiple following trucks, each managed by a separate thread. The output is provided in Fig. 8.

To handle multiple trucks simultaneously, we modified the configuration of the leading truck to accommodate different threads, with each thread responsible for calculating parameters and managing communication for its respective following truck. Additionally, one thread was dedicated to the activity of the leading truck, such as reading the text file and controlling the truck's movement.

```
C:\Windows\System32\cmd.e  X  +  v
Parent Truck Accelerating:
Accelerating. Speed: 55
Accelerating. Speed: 60
Accelerating. Speed: 65
Accelerating. Speed: 70
Accelerating. Speed: 75
Parent moving in Constant Velocity
Speed: 75
Speed: 75
Speed: 75
Parent moving in Constant Velocity
Speed: 75
Speed: 75
Speed: 75
Turning
Updating Turning angles and Location data in Server
Parent Truck Breaking/decelerating:
Speed: 70
Speed: 65

C:\Windows\System32\cmd.e  X  +  v
Speed: 30
Speed: 25
Parent Truck Breaking/decelerating:
Speed: 20
Speed: 15
Speed: 10
Speed: 5
Speed: 0
Journey Ended
```

Fig. 7 Configuration for Leading truck

#### 4) Distance Management and Obstacle Detection

In the case of obstacle detection and distance management, the leading truck, under driver control, manages these aspects. The following truck, on the other hand, periodically checks for obstacles and if safe distance is maintained between it and the preceding truck. In the event of an obstacle, it halts/ adjusts its speed to maintain a safe distance between the truck and obstacle until the obstacle clears. And if the set distance range between the truck's changes, the follower trucks adjust its speed accordingly until the optimal distance between the trucks are maintained. The output for distance management and obstacle detection is shown in Fig. 9.

```

Checking for Obstacle:
Distance Check:
Checking for Obstacle:
Obstacle detected:
Reducing Speed:
Speed: 48
Speed: 46
Obstacle gone:
Increasing to speed to maintain optimal distance:
Child Truck Accelerating:
Accelerating. Speed: 55
Accelerating. Speed: 60
Accelerating. Speed: 65
Obstacle gone:
Increasing to speed to maintain optimal distance:
Distance Check:
Distance not optimal:
Reducing/Increasing Speed to maintain optimal distance:
adjusting speed
adjusting speed
Optimal Distance between trucks are maintained.
Checking for Obstacle:
Obstacle detected:
Reducing Speed:
Speed: 48
Speed: 46
Obstacle gone:

```

Fig. 9 Obstacle Detection and Distance Management

### 5) GPU Implementation

In our GPU implementation, we focused on calculating the adjusted velocities of the following trucks within the platoon and the corresponding output is shown in Fig. 10.

```
C:\Windows\System32\cmd.e  X  +  v

C:\Users\achua\OneDrive\Desktop\Truck>Client
Client Received Platoon Formation Request
Client Accepting request
Client sent: Accepting request

-----
Checking for Obstacle:
Child Truck Accelerating:
Accelerating. Speed: 5
Accelerating. Speed: 10
Accelerating. Speed: 15
Accelerating. Speed: 20
Accelerating. Speed: 25

-----
Checking for Obstacle:
Distance Check:
Checking for Obstacle:
Child Truck Accelerating:
Accelerating. Speed: 30
Accelerating. Speed: 35
```

Fig. 8 Configuration for Following truck

```

Microsoft Visual Studio Debu...
Truck Platooning Simulation
Kernel loading done
defer kernel execution
After building
After kernel execution
Truck Velocities:
30.00 35.00 28.00 32.00
Adjusted Speeds:
30.32 34.62 28.33 31.92
C:\Users\wakash\Downloads\ConsoleApplication1\vsdbg\Debug\ConsoleApplication1.exe (process 336) exited with code 0.
Press any key to close this window . . .

```

Fig. 10 Open CL implementation for adjusting speeds

## VI. SUMMARY AND OUTLOOK

The truck platooning project aims to modernize transportation systems by incorporating advanced technology into truck convoys. Through a master-slave setup, the project ensures smooth communication and task execution among trucks, with a strong focus on safety. Adaptive behaviors enable trucks to adjust to different environments, while advanced obstacle detection systems reduce collision risks, prioritizing efficiency and safety. Implementation involved setting up TCP/IP socket communication and threading processes, with the leading truck directing movements based on instructions from a text file. Follower trucks replicate the lead truck's actions through server-client communication, managed by separate threads.

Additionally, integrating sensors and cameras for image detection to the system is crucial to enhance the system's capabilities, and implementing deadlines and scheduling for tasks is essential for efficient operations. This last one can be solved by working with pthread instead of OpenMP.

#### ACKNOWLEDGEMENT

We express our heartfelt gratitude to Prof. Dr. Stefan Henkler for his unwavering support and guidance throughout the entire course of this project.

#### REFERENCES

- [1] Truck platooning application - S. Ellwanger, E. Wohlfarth, 2017 IEEE Intelligent Vehicles Symposium (IV).
- [2] “An OpenMP Application Programming Interface, Examples”, Version 4.5.0, 11.2016.
- [3] Zhang L, Chen F, Ma X, Pan X. Fuel economy in truck platooning: a literature overview and directions for future research. Journal of Advanced Transportation. 2020 Jan 3;2020.



# APPENDIX A Behavioral Diagram for the Platooning System

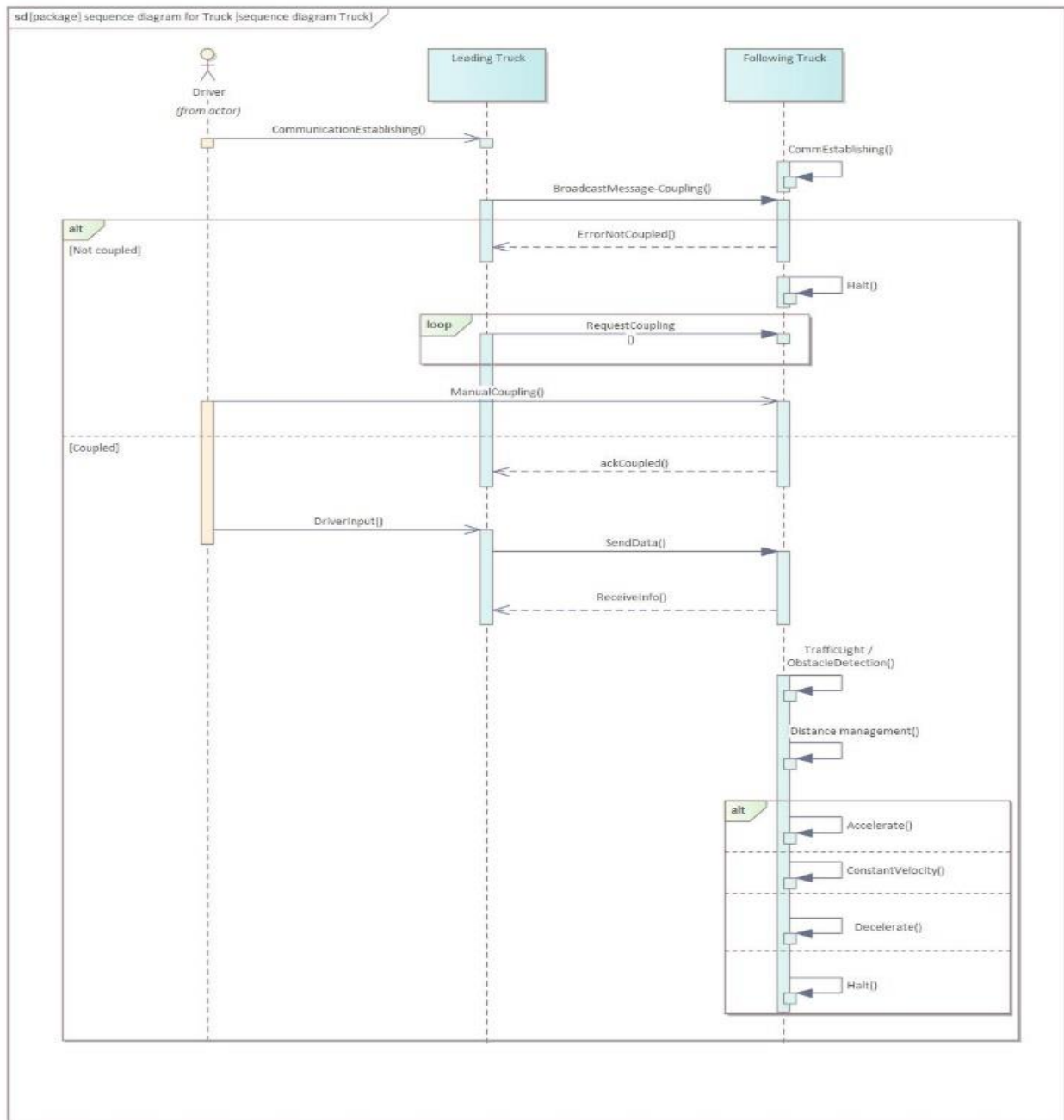


Fig A1. Sequence Diagram of the Platooning system

## APPENDIX B

### Code Implementation

#### A. TCP Communication

##### 1. Server.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <WinSock2.h>
#include <omp.h>
#include <list>

#pragma comment(lib, "ws2_32.lib")

// Declaration of the communicator function
void communicator(int valueToSend, SOCKET clientSocket);
int commcheck(int valueToSend, SOCKET clientSocket);
class FIFOList {
public:
    void appendValue(int value) {
        myList.push_back(value);
    }

    int fetchFirstValue() {
        if (!myList.empty()) {
            int lastValue = myList.front();
            myList.pop_front(); // Remove the fetched value
            return lastValue;
        } else {
            std::cerr << "The list is empty.\n";
            return -1; // Return a sentinel value to indicate an empty list
        }
    }

    bool isEmpty() {
        return myList.empty();
    }
private:
    std::list<int> myList;
};

int main()
{
    WSADATA wsaData; //--> supports socket pro on windows envi
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        std::cerr << "Failed to initialize Winsock.\n";
        return 1;
    }

    SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);/**
    if (serverSocket == INVALID_SOCKET)
    {
        std::cerr << "Failed to create server socket.\n";
        WSACleanup();
        return 1;
    }

    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY; //Assigning the address of my own machine
    serverAddress.sin_port = htons(12345); //--> specifying network byte order
```



```

    if (bind(serverSocket, reinterpret_cast<sockaddr *>(&serverAddress),
sizeof(serverAddress)) == SOCKET_ERROR)
    {
        std::cerr << "Failed to bind the server socket.\n";
        closesocket(serverSocket);
        WSACleanup();
        return 1;
    }

    if (listen(serverSocket, SOMAXCONN) == SOCKET_ERROR)
    {
        std::cerr << "Error while listening for connections.\n";
        closesocket(serverSocket);
        WSACleanup();
        return 1;
    }
    std::cout << "Server listening on port 12345...\n";

    SOCKET clientSocket = accept(serverSocket, nullptr, nullptr);
    if (clientSocket == INVALID_SOCKET)
    {
        std::cerr << "Failed to accept the client connection.\n";
        closesocket(serverSocket);
        WSACleanup();
        return 1;
    }

    int ttt=0;
    FIFOList myFIFOList;

    const char *message = "Platoon Formation Request";
    int plr = 0;

    send(clientSocket, reinterpret_cast<const char *>(&plr), sizeof(int), 0);
    std::cout << "Server sent: " << message << "\n";

    char buffer[256];
    int bytesRead = recv(clientSocket, buffer, sizeof(buffer) - 1, 0);
    if (bytesRead == SOCKET_ERROR)
    {
        std::cerr << "Error while receiving data from the Client.\n";
    }
    else
    {
        buffer[bytesRead] = '\0';
        std::cout << "Server received: " << buffer << "\n";
        if (strcmp(buffer, "Accepting request") == 0)
        {
            std::cout << "Canged ttt\n";
            ttt=1;
        }
    }

    // Opening Sample file
    std::ifstream inputFile("sample.txt");
    if (!inputFile.is_open())
    {
        std::cerr << "Failed to open the file.\n";
        closesocket(clientSocket);
        closesocket(serverSocket);
        WSACleanup();
        return 1;
    }

    // Initialize speed
    int speed = 0;

    // Initialize flag to control communication in Thread 1
    int lst=1;

```

```

// Parallel region with two threads
#pragma omp parallel num_threads(2)
{
    #pragma omp sections
    {

// Section 1: Read file (handled by Thread 1)
#pragma omp section
    {
        std::string line;
        while (std::getline(inputFile, line))
        {
            if (line == "Accelerate")
            {
                std::cout << "Parent Truck Accelerating:\n";
                int t = 1;
                //communicator(t, clientSocket);
                myFIFOList.appendValue(1);
                for (int i = 0; i < 5; ++i)
                {
                    speed += 5;
                    std::cout << "Accelerating. Speed: " << speed << "\n";
                }
            }
            else if (line == "Break")
            {
                std::cout << "Parent Truck Breaking/deaccelerating:\n";
                myFIFOList.appendValue(3);
                for (int i = 0; i < 5; ++i)
                {
                    speed -= 5;
                    std::cout << "Speed: " << speed << "\n";
                }
            }
            else if (line == "Turn")
            {
                myFIFOList.appendValue(6);
                std::cout<<"Turning\n";
                std::cout<<"Updating Turning angles and Location data in Server\n";

            }
            else if (line == "Stop")
            {
                myFIFOList.appendValue(4);
                std::cout<<"Journey Ended\n";
                closesocket(serverSocket);
                break;
            }
            else
            {
                std::cout << "Parent moving in Constant Velocity\n";
                myFIFOList.appendValue(2);
                for (int i = 0; i < 3; ++i)
                {
                    std::cout << "Speed: " << speed << "\n";
                }
            }
            //Distance Check(5);
            myFIFOList.appendValue(5);
        }
    }

// Section 0: Communication (handled by Thread 0)
#pragma omp section
    {
        while (!myFIFOList.isEmpty())
        {
            if(myFIFOList.isEmpty())
            {

```

```

        std::cout<<"List is empty";
        lst=0;
    }

    int firstValue = myFIFOList.fetchFirstValue();
    communicator(firstValue, clientSocket);
}

}

}

// Close the sockets and clean up
closesocket(clientSocket);
closesocket(serverSocket);
WSACleanup();

return 0;
}

// Definition of the communicator function
void communicator(int valueToSend, SOCKET clientSocket)
{
    // Send the received value to the client
    send(clientSocket, reinterpret_cast<const char *>(&valueToSend), sizeof(int), 0);
}

int commcheck(int valueToSend, SOCKET clientSocket)
{
    // Send the received value to the client
    int x=100;
    send(clientSocket, reinterpret_cast<const char *>(&x), sizeof(int), 0);
    int receivedNumber;
    rcv(clientSocket, reinterpret_cast<char *>(&receivedNumber), sizeof(int), 0);
    return receivedNumber;
}

```

## 2. Client.cpp

```

#include <iostream>
#include <WinSock2.h>
#include <random>

#pragma comment(lib, "ws2_32.lib")
int getRandomNumber();

int main() {
    // Initialize Winsock
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        std::cerr << "Failed to initialize Winsock.\n";
        return 1;
    }

    // Create a socket
    SOCKET clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (clientSocket == INVALID_SOCKET) {
        std::cerr << "Failed to create client socket.\n";
        WSACleanup();
        return 1;
    }

    // Connect to the server
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = inet_addr("127.0.0.1"); // Assuming server is running on
the same machine

```

```

serverAddress.sin_port = htons(12345);

if (connect(clientSocket, reinterpret_cast<sockaddr*>(&serverAddress),
sizeof(serverAddress)) == SOCKET_ERROR) {
    std::cerr << "Failed to connect to the server.\n";
    closesocket(clientSocket);
    WSACleanup();
    return 1;
}

// Initialize speed
int speed = 0;

int bytesRead;
int receivedNumber;
while(bytesRead = recv(clientSocket, reinterpret_cast<char*>(&receivedNumber),
sizeof(int), 0)){

    if (bytesRead > 0) {
        if(receivedNumber==0)
        {

            std::cout << "Client Received Platoon Formation Request" << "\n";
            std::cout << "Client Accepting request" << "\n";
            const char* response = "Accepting request";
            send(clientSocket, response, strlen(response), 0);
            std::cout << "Client sent: " << response << "\n";
            std::cout << "-----" <<
"\n";

        }

        if(receivedNumber==1)
        {
            std::cout << "Child Truck Accelerating:\n";
            for (int i = 0; i < 5; ++i) {
                // Increase speed by 5 for each iteration
                speed += 5;
                std::cout << "Accelerating. Speed: " << speed << "\n";
            }
            std::cout << "-----"
" << "\n";
        }

        if(receivedNumber==3)
        {
            std::cout << "Child Truck Breaking/deaccelerating:\n";
            for (int i = 0; i < 5; ++i) {
                // Decrease speed by 5 for each iteration
                speed -= 5;
                std::cout << "Speed: " << speed << "\n";
            }
            std::cout << "-----"
" << "\n";
        }

        if(receivedNumber==2)
        {
            std::cout << "Child Truck moving in Constant Velocity\n";
            for (int i = 0; i < 5; ++i) {
                std::cout << "Speed: " << speed << "\n";
            }
            std::cout << "-----" <<
"\n";
        }

        if(receivedNumber==4)
        {
            std::cout << "Journey Ended.....\n";
            break;
        }

        if(receivedNumber==6)
        {

```

```

std::cout << "-----\n";
" << "\n";
std::cout << "Receiving Turning Info(Turning angle & Location data)\n";
std::cout << "Turning on the exact location\n";
std::cout << "-----\n";
" << "\n";
    }
    if(receivedNumber==5)
    {
        int DC=getRandomNumber();
        std::cout<< DC<<"Distance Check:\n";
        if(DC==1)
        {
            std::cout<<"Distance not optimal:\nReducing/Increasing Speed to maintain
optimal distance:\n";
            for (int i = 0; i < 2; ++i) {
                std::cout << "adjusting speed" <<"\n";
            }
            std::cout<<"Optimal Distance between trucks are maintained.\n";
        }
        else{
            //std::cout<<"The distance between trucks is optimal.... "<<DC<<"\n";
        }
    }
}
int Obs=getRandomNumber();
std::cout<< "Checking for Obstacle:\n";
if(Obs==1)
{
    std::cout<<"Obstacle detected:\nReducing Speed:\n";
    for (int i = 0; i < 2; ++i) {
        // Decrease speed by 5 for each iteration
        speed -= 2;
        std::cout << "Speed: " << speed << "\n";
    }
    std::cout<<"Obstacle gone:\nIncreasing to speed to maintain optimal
distance:\n";
    for (int i = 0; i < 2; ++i) {
        // Decrease speed by 5 for each iteration
        speed += 2;
        //std::cout << "Speed: " << speed << "\n";
    }
}
else{
    //std::cout<<"No obstacle detected: "<<Obs<<"\n";
}

}
// Close the socket and clean up
closesocket(clientSocket);
WSACleanup();

return 0;
}
int getRandomNumber() {
    // Generate a random number between 1 and 3
    return std::rand() % 3 + 1;
}

```

## B. GPU Implementation

### 1. Host Code – adj\_speed.cpp

```
// adj_speed.cpp
#define CL_USE_DEPRECATED_OPENCL_2_0_APIS
#pragma comment(lib, "OpenCL.lib")
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include "C:/Program Files (x86)/AMD APP SDK/3.0/include/CL/cl.h"

#define MAX_SOURCE_SIZE 8192

int main(void) {
    printf("Truck Platooning Simulation\n");

    // Create the four input
    const int LIST_SIZE = 4;

    // Truck information arrays
    float* velocities = (float*)malloc(sizeof(float) * LIST_SIZE);
    float* adjusted_speeds = (float*)malloc(sizeof(float) * LIST_SIZE);
    float communication_gain = 0.1f;

    velocities[0] = 30.0f;
    velocities[1] = 35.0f;
    velocities[2] = 28.0f;
    velocities[3] = 32.0f;

    // Load the platooning kernel source code into the array source_str
    FILE* fp;
    char* source_str;
    size_t source_size;

    if (fopen_s(&fp, "adj_speed_kernel.cl", "r") != 0) {
        fprintf(stderr, "Failed to load kernel.\n");
        exit(1);
    }

    source_str = (char*)malloc(MAX_SOURCE_SIZE);
    source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
    fclose(fp);
    printf("Kernel loading done\n");

    // Get platform and device information
    cl_device_id device_id = NULL;
    cl_uint ret_num_devices;
    cl_uint ret_num_platforms;

    cl_int ret = clGetPlatformIDs(0, NULL, &ret_num_platforms);
    cl_platform_id* platforms = NULL;
    platforms = (cl_platform_id*)malloc(ret_num_platforms * sizeof(cl_platform_id));

    ret = clGetPlatformIDs(ret_num_platforms, platforms, NULL);

    ret = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, 1,
        &device_id, &ret_num_devices);

    // Create an OpenCL context
    cl_context context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);

    // Create a command queue
    cl_command_queue command_queue = clCreateCommandQueue(context, device_id, 0, &ret);

    // Create memory buffers on the device for each variable
    cl_mem velocities_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
        LIST_SIZE * sizeof(float), NULL, &ret);
```

```

cl_mem adjusted_speeds_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    LIST_SIZE * sizeof(float), NULL, &ret);

// Copy the truck velocities to their respective memory buffer
ret = clEnqueueWriteBuffer(command_queue, velocities_mem_obj, CL_TRUE, 0,
    LIST_SIZE * sizeof(float), velocities, 0, NULL, NULL);

printf("Before kernel execution\n");

// Create a program from the kernel source code
cl_program program = clCreateProgramWithSource(context, 1,
    (const char*)&source_str, (const size_t*)&source_size, &ret);

// Build the program
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

printf("After building\n");

// Create the OpenCL kernel
cl_kernel kernel = clCreateKernel(program, "platooning_communication", &ret);

// Set the arguments of the kernel
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&velocities_mem_obj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&adjusted_speeds_mem_obj);
ret = clSetKernelArg(kernel, 2, sizeof(float), (void*)&communication_gain);

// Execute the OpenCL kernel
size_t global_item_size = LIST_SIZE;
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
    &global_item_size, NULL, 0, NULL, NULL);

printf("After kernel execution\n");

// Read the memory buffer (adjusted speeds) on the device to the local variable
ret = clEnqueueReadBuffer(command_queue, adjusted_speeds_mem_obj, CL_TRUE, 0,
    LIST_SIZE * sizeof(float), adjusted_speeds, 0, NULL, NULL);

// Display the output to the screen
printf("Truck Velocities:\n");
for (int i = 0; i < LIST_SIZE; i++)
    printf("%.2f\t", velocities[i]);

printf("\nAdjusted Speeds:\n");
for (int i = 0; i < LIST_SIZE; i++)
    printf("%.2f\t", adjusted_speeds[i]);

// Clean up
ret = clFlush(command_queue);
ret = clFinish(command_queue);
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(velocities_mem_obj);
ret = clReleaseMemObject(adjusted_speeds_mem_obj);
ret = clReleaseCommandQueue(command_queue);
ret = clReleaseContext(context);
free(velocities);
free(adjusted_speeds);

return 0;
}

```



## 2. Kernel Code – adj\_speed\_kernel.cl

```
// adj_speed_kernel.cl

__kernel void platooning_communication(__global const float* velocities, __global float*
adjusted_speeds, float communication_gain) {
    int i = get_global_id(0);

    float current_velocity = velocities[i];

    // Simulate communication: adjust speed based on the average velocity of the platoon
    float sum_velocities = current_velocity;
    for (int j = 0; j < get_global_size(0); j++) {
        if (j != i) {
            sum_velocities += velocities[j];
        }
    }

    float average_velocity = sum_velocities / get_global_size(0);

    // Adjust the speed based on communication
    adjusted_speeds[i] = current_velocity + communication_gain * (average_velocity -
current_velocity);
}
```

## APPENDIX C

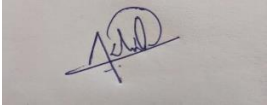
### Git Overview

- A. Lines of Code:  
Server.cpp – 238 lines  
Client.cpp – 145 lines  
Adj\_speed.cpp – 128 lines  
Adj\_speed\_kernel.cl – 20 lines
- B. Number of Submits per Person:  
Akhilesh Kakkayamkode – 15  
Arjun Veeramony – 3  
Angel Mary – 4
- C. Structure:  
There are mainly four folders.
1. Code: It includes two separate folders for source codes for Socket Programming and GPU Implementation.
  2. UML Diagrams: It contains all UML behavioural diagrams created during the project period.
  3. Presentation: Includes the ppt for presenting the topic.
  4. Final Report: A separate folder was created for adding the final report.
- D. Contribution by Each Member:
- Akhilesh Kakkayamkode:
- Code – GPU Implementation complete code
  - Report – Abstract, Introduction, Architecture Concept, GPU Implementation, Appendix B, Appendix C
- Arjun Veeramony:
- Code – Socket Programming complete code
  - Report – Motivation, Implementation, Conclusion
- Angel Mary:
- UML Diagrams – State Machine Diagrams for Leading and Following trucks, Activity Diagram for Truck platooning System, Sequence Diagram, Activity diagram for Communication failure scenario, Activity Diagram for Distance Management
  - Report – Sketch of Approach, Appendix A

## AFFIDAVIT

We Akhilesh Kakkayamkode, Angel Mary and Arjun Veeramony herewith declare that we have composed the present paper and work ourself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The paper and work in the same or similar form has not been submitted to any examination body and has not been published. This paper was not yet, even in part, used in another examination or as a course performance.

Dortmund  
31.01.2024



Akhilesh Kakkayamkode



Angel Mary



Arjun Veeramony