Truck Platooning

Farhad Gulizada
Matric. No:7216770
Masters in Embedded Systems Engineering
Fachhochschule Dortmund
Dortmund, Germany
Contribution Percentage: 20

Negin Amiri
Matric. No:7216441
Masters in Embedded Systems Engineering
Fachhochschule Dortmund
Dortmund, Germany
Contribution Percentage: 20

Zhao Tao
Matric. No:7216774
Masters in Embedded Systems Engineering
Fachhochschule Dortmund
Dortmund, Germany
Contribution Percentage: 20

Raghuveer Rajesh Dani
Matric. No:7216427
Masters in Embedded Systems Engineering
Fachhochschule Dortmund
Dortmund, Germany
Contribution Percentage: 20

Zaur Gurbanli
Matric. No:7216615
Masters in Embedded Systems Engineering
Fachhochschule Dortmund
Dortmund, Germany
Contribution Percentage: 20

***Abstract* — The increasing number of vehicles on our roads has led to a pressing need for efficient traffic management. With the surge in autonomous cars, it has become even more important to effectively integrate autonomous vehicles into urban environments. This project is specifically focused on platooning trucks, which is a grouping of trucks consisting of a lead vehicle and one or more followers that are connected through advanced technology, enabling autonomous movement. As a result, the follower vehicles can maintain a safe distance from the lead vehicle and move independently without direct interaction from the driver. This innovative platooning system is controlled solely by the driver of the lead truck, who has full control over the platoon's movement. Overall, this technology provides a safer and more efficient means of transportation for the goods and products we rely on every day.**

Index Terms*—Platoon, Autonomous, Socket Programming *(NEGIN)

I. Introduction

This project presents a software-based approach to platooning a line of trucks. The first truck is a leader with a driver and other trucks do not have a driver and just follow the leader truck via a communication system and exchange signals. The proposed system integrates vehicle communication and sensing technologies to provide a safe and efficient semi-autonomous platoon in the environment.

The software-based solution employs a system that takes into consideration various factors such as the communication protocol, types of signals that need to be transferred, and how the trucks are going to be networked.

The communication protocol used by the platooning system is crucial to ensure reliable and efficient communication between the leader and follower trucks. The protocol should be able to handle large amounts of data in real time and maintain a stable connection between the trucks even in areas with adverse geographic conditions. The signals exchanged between the trucks should be accurate and precise, allowing the follower trucks to respond promptly to the movements of the leader truck.

To achieve a safe and efficient platooning system, the software-based approach also integrates various sensing technologies, such as GPS and Ultrasonic Radar. Ultrasonic radar acquires the distance between the two trucks and this distance data is used to maintain the gap between the trucks.

Overall, a software-based platooning system has the potential to improve road safety, reduce fuel consumption, and increase the efficiency of freight transportation. As the technology continues to evolve, platooning will likely become more widespread, providing a solution to the challenges faced by the trucking industry.

II. Literature Review

The concept of platooning in the transportation industry has gained a lot of attention due to its potential to improve road safety, reduce fuel consumption, and increase road capacity. The ability of trucks to follow each other closely, known as platooning, has been demonstrated to be a promising solution to address these challenges. A platoon consists of a group of vehicles that follow each other at close distances, with the lead vehicle being driven by a human driver and the following vehicles being semi-autonomous or

fully autonomous. In this project, we have considered the following trucks to be fully autonomous i.e. driverless.

To achieve safe and efficient platooning, communication and sensing technologies play a crucial role. In recent years, there have been significant developments in vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communication technologies that allow vehicles to communicate with each other and the surrounding environment. These technologies can be used to provide real-time data about traffic conditions, road hazards, and weather conditions to enable the platoon to operate more safely and efficiently.

Moreover, sensing technologies such as LiDAR and radar can be used to detect the distance and speed of the vehicles in the platoon, ensuring that each vehicle maintains a safe distance from the vehicle in front of it. The proposed software-based solution in this project takes into consideration various factors, including the communication protocol, types of signals that need to be transferred, and how the trucks will be networked, to provide a safe and efficient platoon. We have proposed a solution using Ultrasonic Sensor for distance measurement.

Several research studies have been conducted on platooning, and the results have shown promising benefits, including reduced fuel consumption, improved traffic flow, and increased safety. However, there are also some challenges associated with platooning, such as the need for accurate and reliable communication and sensing technologies and the potential risks associated with cyber-attacks.

The integration of communication and sensing technologies will enable safe and efficient platooning, which has the potential to improve road safety, reduce fuel consumption, and increase road capacity. Further research is needed to address the challenges associated with platooning and to optimize the platooning algorithms for different road and traffic conditions.

A. Distributed Computing Systems

Any system, which contains some computing components that are working as a unified platform for a purpose, is assumed as a distribution system. As a distributed system functioning system structure of the truck platoon is a master as the leading truck and other components are slaves as following trucks functioning based on the serves of the master.

The proposed system utilized sensing elements along with the velocity transmitted from the master to estimate the desired distance with the help of the CACC to be able to reach trucks and afterwards to keep stabilized distance for safety reasons. Firstly, the following truck should accept the velocity and based on the coming speed the function of every slave truck is to calculate to determine the required value of the desired distance to be able to keep the predetermined safe distance range. And if there is any gap between the following trucks due to the reason when a truck leaves the truck platoon or just there is any increased value of the speed transmitted to the following trucks with the help of the calculated outputs of the first step the following trucks apply to try to adjust parameters as long as the desired distance is not achieved.

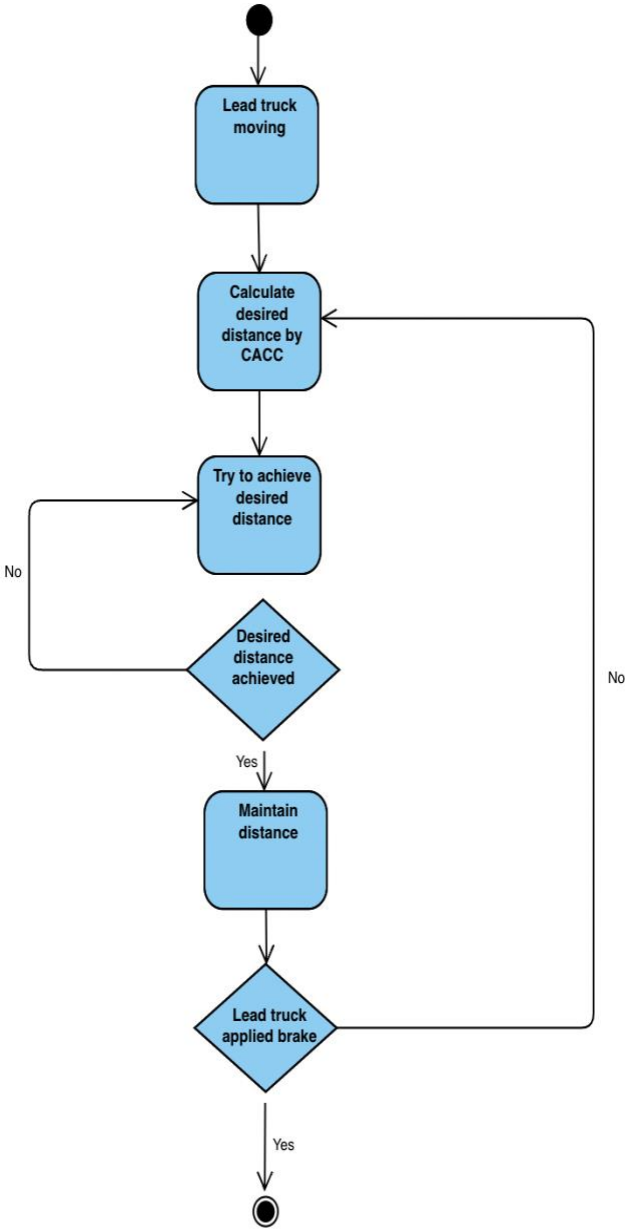

Figure 1. Truck platoon activity diagram

Afterwards, when the distance is achieved the purpose is to maintain the distance same over the running period of the truck platoon before the system has detected any kind of failure in the communication which means if there is any total failure in the communication despite the failure whether encountered in the UDP protocol or TCP protocol the system is considered to be un-servable and therefore the system functioning is stopped. Therefore, since total failure causes the system to stop functioning then the trucks stop operating as the slave part of the master-slave architecture and the system shuts down. Also, another case for the trucks to stop

functioning is when the destination is reached and there is not any need for the communication and the total system to keep running.

B. Networking of System

As far as the system is managed and controlled by the leader truck, and any signal is transferred to and from it, the considered network for the system is master-slave as below.

Since the leading truck is the master in the network of the truck platoon the main load of the connection is upon the leading truck and it is responsible for the prevention of message loss and connection stability while its resources are dedicated to GPU as well. Besides, the slaves are connected to the master based on the UDP protocol to eliminate generally the latency happening because of the number of the several following trucks and also the TCP protocol if there is any failure in case of the UDP connection.

At present, there are several widely used communication protocol for Master Slave communication mode such as TCP, UDP, MQTT. Every protocol has different advantages and feature as well as drawback. To find a appropriate communication protocol for platooning system, relaible, efficiency, real-time and resource consumption are the main factors that developer should take into account.According to various application scenario,different weight value is supposed to bestowed to different factor,then developer is able to find a optimal communication protocol for system. Therefor,t he feature of these communication protocols are presented below.

TCP is a connection-oriented, reliable, byte stream-based transport layer communication protocol.Due to three-way handshake to establish connection, the advantage of TCP is that it is more reliable and stable and when the data is being transmitted, there are confirmation, window, retransmission, and congestion. However, there are still some drawbacks that it is relatively slow, low efficiency, high system resource usage. Generally, TCP is applied in occasions that require reliability and transmission integrity e.g. file transmission, mail transmission.

UDP is a connectionless transport layer protocol that provides transaction-oriented 、 simple and unreliable information transfer services. The advantage is that it has high efficiency, high throughput and good real-time performance, is slightly more secure than TCP. On the other hand, the drawback is that it is unreliable and unstable since there is no connection establishment step and the size of transporting data is limited. Generally, It is applied in the occasions that emphasizing transmission real-time and efficient performance rather than transmission integrity e.g. audio and Multimedia Applications.

MQTT is a message protocol based on the publish/subscribe paradigm.Compared with others,advantage is that it is reliable and lightweight, simple, and easy to implement. It has low protocol resource usage,low power consumption and tolerance for unstable networks.But,MQTT MQTT does not support peer-to-peer communication.Usually, It is applied in the occasions that has limited computing power and low-bandwidth, unreliable networks e.g. remote control.

In face, the platooning system require a hard real time property very much, so Tcp will be removed from the alternatives. The platooning system,in some occasion,need P2P communication supporting, so MQTT is not appropriate. Overall, UDP is a optimal choice considering the different weight value of these communication's advantage.

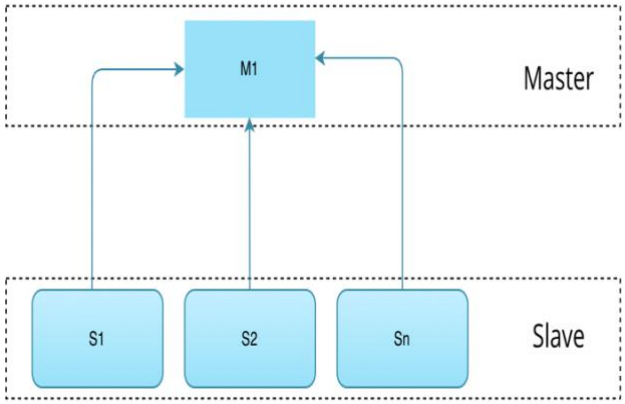

Figure 2. System Network (master-slave)

C. Parallel Architectures

Three concurrency libraries that are often used in programming are Pthreads, OpenMP, and C++14 threads.

A library called Pthreads (POSIX threads) is used in Unix-like operating systems to create and manage threads. It offers a low-level thread API and enables precise control over thread management, including the ability to create, join, and synchronize threads. Pthreads, on the other hand, demand manual thread management and might be more prone to mistakes. The parallel computing high-level library OpenMP (Open Multi-Processing) is available for C and C++. It specifies parallel regions via compiler directives, and the library is in charge of thread generation and synchronization. For parallelizing loops and other computations that can be broken down into separate processes, OpenMP is frequently utilized. It is easier to use than Pthreads and frequently improves performance with little code modification. A threading library included in the C++ standard library is called C++14 threads. With features like thread-safe memory management and support for move-only types, it offers a high-level API for setting up and controlling threads. In terms of control over thread management, C++14 threads are comparable to Pthreads, but they offer a more up-to-date, object-oriented interface.

Therefore, OpenMP offers a high-level interface for parallel computing, Pthreads offers low-level control over

thread management, and C++14 threads offer a cutting-edge, object-oriented interface for C++ thread management. The requirements for the application, the degree of thread management control required, and the programming language being used greatly influence the library selection. By taking all the aforementioned information into account, it is more suitable to use a C++ thread for implementing the GPU algorithm in the truck platooning project.

C. CACC

Cooperative Adaptive Cruise Control (CACC) is a vehicle automation technique that improves the performance of conventional Adaptive Cruise Control (ACC) systems through vehicle-to-vehicle communication. The CACC system enables vehicles to communicate with one another and move in a platoon formation, which can free up more road space and boost capacity while also enhancing fuel economy.

III. MODELING

A. Network

1. Network structure between master and slaves

The network of the truck platoon is of the main parts of the provision of control over autonomous trucks which is based on the master slave. And it should be designed to be effective and independent of the number of trucks in the platoon. So, it requires the master to be programmed so that the connection is not down in case of an increasing number of requests and in parallel the information required for the GPU to process is available. To provide scalability of the system master part of the master-slave architecture utilizes the threads whose number is 3. These threads are dedicated to messaging receiving and sending and an additional thread for GPU. And the slave side contains 2 threads that are designed for handling the message and after receiving it and the other one to send feedback to the master. On the other hand, to provide less load in the connection UDP protocol is made used from the protocol point of view which is fast in comparison to TCP protocol. However, in terms of ensuring the data reaches the required address, the UDP protocol is not suitable. That's why the UDP protocol is based on the feedback coming from the following trucks to notify the master so that the data loss prevention function of the network system is assured. And the feedback is waited by the master, but if it is not delivered the master tries to reconnect to the slave 3 times in general so that data is delivered again from the leading truck. However, despite the applied feedback technique, the UDP connection cannot be relied on to perform its function properly. Therefore, in some cases the TCP connection is needed to be utilized when the feedback sent does not reach the leading truck, especially for the trucks with which the reconnection attempt failed in case of the UDP connection. Despite the TCP connection being less failure-prone, it is appropriate to be used when defined attempt times exceed in the UDP connection, but if the connection failure happens again even in the TCP connection the network of the truck platoon is down and the whole system is stopped to prevent any accident before it happens. When a communication failure happened, the following trucks can detect communication failure by checking whether there is a new message from the leading truck. If there is no new message coming within 10s, the client system interprets this as a communication failure and then the following truck brakes and comes to a stop.

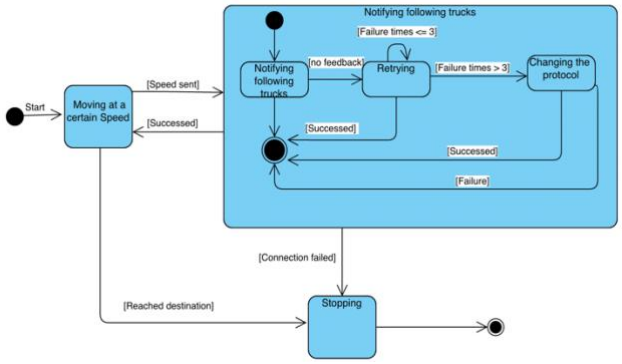

Figure 3. Platoon workflow state diagram

2. Joining/leaving trucks

As mentioned before network system mostly is based on the UDP protocol to provide fast connection along with the feedback coming from the slaves to provide data loss safety and additionally the TCP protocol to address the problems encountered when the UDP protocol fails. However, since the fact the UDP protocol is fast and is utilized to eliminate the latency factor in the connection of several trucks when a truck or fewer trucks are required to be added or removed from the line, the process involves using the UDP protocol again. And when a truck is about to join, firstly the join request is sent to the leading truck. be removed from the list of the trucks or it will stay in the line.

Based on the coming response handled by the master the new coming truck is allowed to join the line and added to the list of the present trucks of the truck platoon. Otherwise, it is rejected to join. Similarly, when a truck sends leaving signal to the master based on the coming response the truck is granted to leave and the registration of the truck will be deleted.

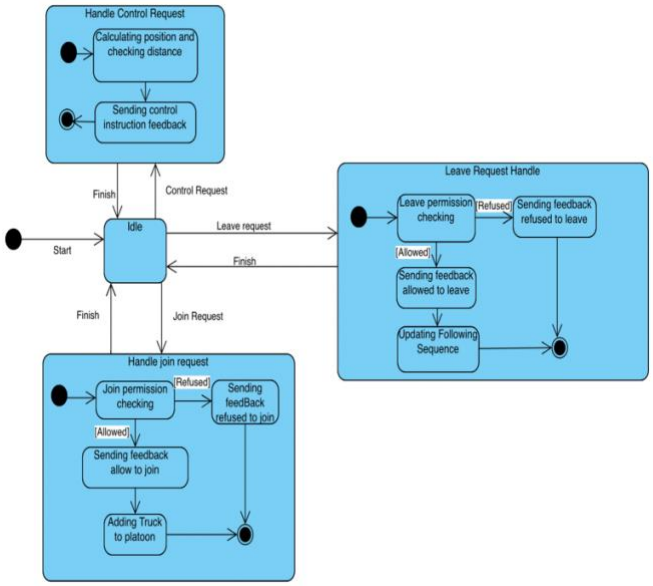

Figure 4. Joining and leaving truck state diagram

3. CACC based connection

Since the truck platoon is a system of a leading truck with a driver and the following trucks to provide synchronization and eliminate any potential risks certain piece of info about the leading truck is sent to the following trucks which contain the speed of the leading truck to be transmitted. And based on the speed the following trucks can adjust speed based on the CACC technique to increase the capability of avoiding crashes or any accidents.

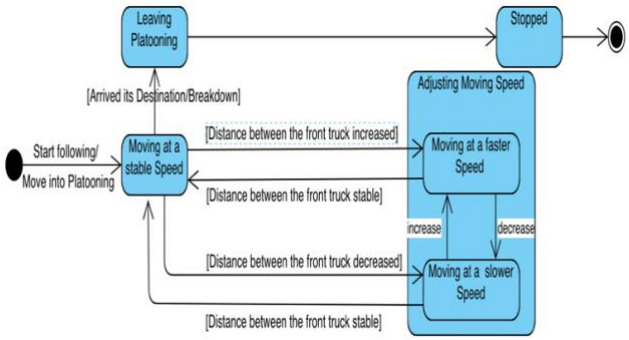

Figure 5. Following the truck moving state diagram

As seen from the figure above this enables every slave truck to stabilize the speed if it moves from a fast speed to a slower speed or vice versa so that the truck at the end comes to a stabilized speed ending up leaving the line when the targeted distance is achieved or when there is a communication failure when the UDP connection fails after 3 attempts and the TCP protocol does.

B. CACC

1. Working Principle of CACC

The concepts of adaptive cruise control, a cutting-edge safety feature that enables vehicles to keep a secure distance from the vehicle in front of them, are the foundation of cooperative adaptive cruise control. By allowing vehicles to interact with one another and coordinate their motions, CACC expands on this technology and produces a platoon of closely spaced and precisely synced vehicles.

Keeping an accurate and steady location in the platoon is achieved through the utilization of both hardware and software components in CACC systems. These include devices for establishing a connection between vehicles, several sensors, and sophisticated algorithms for controlling vehicles. In the platoon, using wireless connection among vehicles enables data exchange which comes from the sensors for detecting the velocity and location of vehicles in front.

Used control algorithms for CACC systems are meant to help cars react swiftly and seamlessly to environmental changes like shifting traffic patterns or deteriorating road conditions. CACC systems can keep a stable flow of traffic, minimize the need for harsh deceleration and speeding, and increase overall traffic safety by coordinating the actions of the cars in the platoon.

2. Implementation of CACC

Both hardware and software elements must be used in conjunction to implement CACC systems. Although the software components commonly consist of control algorithms and some protocols for creating communication that allows the vehicles to interact and synchronize their motions, the hardware components frequently comprise communication devices and sensors.

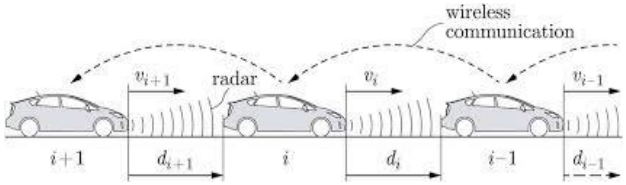

Figure 6. A platoon of vehicles equipped with CACC [1]

A radar system that uses ultrasonic waves to determine the location of the vehicle in front is one of the sensors used in CACC systems. UDP and TCP network protocols are used in CACC systems to allow cars to communicate with one another and exchange data regarding their position, speed, and acceleration.

Vehicles can react swiftly and flawlessly to environmental changes thanks to the control algorithms utilized in CACC systems. To allow vehicles to remain in a secure and constant position in the platoon, these algorithms often combine feedback control with predictive control. For

calculating the desired distance between 2 trucks, the following spacing policy is used,

$$d_{r,i}(t) = r_i + h * v_i(t)$$

where $d_{r,i}$ is the desired distance between vehicle i and $i-1$, h is the time headway in seconds, and r_i is the standstill distance.

IV. IMPLEMENTATION

A. *Server*

1. Parallelism Realization

For the Parallelism part, based on C++ 14 std::thread, one main thread, and two working threads are used in this system. One working thread is set to send messages to the following trucks as Figure 5 shows, another one is used to handle messages received from the following trucks as Figure 6 shows.

```
void TransmissionHandler::ThreadInitialization()
{
    /*thread Configuration*/
    thread sendThread(&TransmissionHandler::sendMsg, this);
    sendThread.detach();
}
```

Figure 7. Server Sending Message Thread Initialization

```
void MessageHandler::ThreadInitialization()
{
    thread msgHandleThread(&MessageHandler::msgHandleExecution, this);
    msgHandleThread.detach();
}
```

Figure 8. Server Handling Message Thread Initialization

The workflow is that the main thread takes responsibility to receive a message from the following trucks then parses the message to a readable data structure and delivers the data structure to a message handler that can simultaneously handle the message as Figure 7 shows. Meantime the *send message* thread always sends a message including relevant information about the leading truck with a modifiable period. Following the communication realization part will do a detailed description.

```
while (1)
{
    SFollowingTruckInfo msg = transHandler.recvMsg();

    msgHandler.OnMessage(msg, transHandler.clientAddr());

    Sleep(500);

    SMessageFeedBack feedback = msgHandler.getHandlingFeedBack();

    transHandler.sendMsgOnce(feedback);
}
```

Figure 9. Sever Main Thread

2. Critical Section

To realize a flexible and manageable shared resource interface, a Meyer's Singleton is used in the sever project as Figure 8 shows. If the control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for the completion of the initialization. This ensures that concurrent threads must be initialized when they acquire static local variables, so it is thread-safe.

```
PlatooningManager* PlatooningManager::getInstance()
{
    static PlatooningManager ins;
    return &ins;
}
```

Figure 10. Realization of Meyer's Singleton

The singleton PlatonningManager with the intention to manage a shared resource storing the information of each following truck defined a set of behaviours if modification, read, add and remove as Figure 9 shows, so these behaviours are critical sections.

```
public:
    bool addTruck(int id, Truck* truck);

    Truck* getTruck(int id);

    bool removeTruck(int id);

    int getNumOfFollowingTruck();

    void updateSequenceNo(int startSequence);

    list<SOCKADDR_IN*> getCommunicationAddr();

private:
    map<int, Truck*> m_trucksMap;
    mutex _mtx;
```

Figure 11. Critical Section Declaration

To avoid some unexpected consequences and make sure being thread-safe, synchronous access to the shared resource must be guaranteed, the std::mutex class being a synchronization primitive from the C++ concurrency support library is used to protect shared data from being simultaneously accessed by multiple threads as Figure 9 shows. Here used two methods to give mutex and release the mutex. One is to use ***mutex.lock()*** to lock the mutex, blocks if the mutex is not available and mutex and then use ***mutex.unlock()*** to unlock the mutex. However, because ***mutex.unlock()*** must be executed before the function exit otherwise dead-lock will happen, this way can not handle the function with an uncertain exit point such as the ***getTruck(int id)*** function has two exit points shown in Figure 10. Therefore, another method is used as an alternative. To be specific, ***std::lock_guard*** is a mutex wrapper that provides a convenient RAII-style mechanism for owning a mutex for the duration of a scoped block. When a lock_guard object is created, it attempts to take ownership of the mutex it is given. When control leaves the scope in which the lock_guard object was created, the lock_guard is destructed and the mutex is released.

```
bool PlatooningManager::addTruck(int id, Truck* truck)
{
    if (truck)
    {
        _mtx.lock();
        m_trucksMap.insert(pair<int, Truck*>(id, truck));
        _mtx.unlock();
        return true;
    }
    return false;
}

Truck* PlatooningManager::getTruck(int id)
{
    lock_guard<mutex> lg(_mtx);
    auto iter = m_trucksMap.find(id);
    if (iter != m_trucksMap.end())
        return iter->second;

    return nullptr;
}
```

Figure 12. Critical Section Declaration

3. Communication Realization

Real-time performance is critical for Platooning System, so UDP protocol is the best choice to apply in communication.

Network configuration based on windows2 TCP/IP standard is presented in Figure 11. There are some configuration steps i.e. loading the socket library, and creating a socket binding socket port. Address information including IP address and port is set into ***SOCKADDR_IN*** and then is bound by sever UDPSocket. After successful binding, multiple client UDPSockets with the same IP address and port can send messages to the server.

```
printf("Sever is operating!\n\n");


sockSrv = socket(AF_INET, SOCK_DGRAM, 0);

SOCKADDR_IN addrSrv;
addrSrv.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
addrSrv.sin_family = AF_INET;
addrSrv.sin_port = htons(PORT_UDP);

if (bind(sockSrv, (SOCKADDR*)&addrSrv, sizeof(SOCKADDR)) == SOCKET_ERROR)
{
    cout << "udp binding failed!";
}
```

Figure 13. UDP Server Configuration

As Figure 12 shows, this is the receive message function of the server. Throughout calling ***recvfrom*** function, the server can receive network messages from every following truck. The raw content is stored in the ***recvBuff*** being a char array and this raw content will be parsed to a readable data structure.

```
recvfrom(sockSrv, recvBuf, 100, 0, (SOCKADDR*)&addrClient, &len);
memcpy(&msg, recvBuf, sizeof(SFollowingTruckInfo));
```

Figure 14. UDP Server receive function

As Figure 13 shows, this is the send message function of the server. Firstly, the message structure should be transferred to the corresponding **char*** type stored in the ***sendBuff***, then through calling ***sendTo*** function, the server can send a network message to a specified following truck.

```
memcpy(sendBuf, &feedback, sizeof(SMessageFeedBack));
sendto(sockSrv, sendBuf, 100, 0, (SOCKADDR*)&addr, len);
```

Figure 15. UDP Server send function

4. MessageHandle Realization

As mentioned before, the message is pushed to MessageHandler class using a separate thread to handle it. In the function body, the ***std::unique_lock*** from the C++ concurrency support library is a general-purpose mutex ownership wrapper allowing deferred locking, time-constrained attempts at locking, recursive locking, transfer of lock ownership, and use with condition variables used to occupy and release the mutex. Then a ***condition_variable*** is put into a while loop with Boolean ***wake_Up*** as a condition. The initial value of ***wake_Up*** is false, so the current thread will get into the state of waiting for waking up. Once ***wake_Up*** is set to true and ***notify_one*** function is called, the current thread will be waked up to execute the task once.

In the execution body of the thread task, there is a switch case gramma used to handle different types of request messages. So the variable request type is the objective of a switch. There are totally three request types control request, join request, and leave request meaning three different cases.

```
void MessageHandler::msgHandleExecution()
{
    int len = sizeof(SOCKADDR);
    while (1) {
        unique_lock <std::mutex> lck(m_mtx);
        while (!m_wakeUp)
            m_cv.wait(lck);

        SMessageFeedBack feedbackMsg;
        float velocity = (float)m_truckInfo.m_velocity / 100;
        switch (m_truckInfo.m_request)
        {
            case RequestType::CONTROL_REQUEST:
                feedbackMsg.m_FeedbackType = FeedBackType::CONTROL_FEEDBACK;
                cout << "\n";
                cout << "Received the control request from Truck <" << m_truckInfo.m_truckID << "> " << endl;
                cout << "       Current Velocity is : " << velocity <<"km/h"<<endl;

                break;
            case RequestType::JOIN_REQUEST:
                cout << "\n";
                cout << "Received the join request from Truck <" << m_truckInfo.m_truckID << ">" << endl;
                feedbackMsg.m_FeedbackType = FeedBackType::JOIN_FEEDBACK;
                if (JoinPermissionCheck())
                {
                    cout << "Permit to Join the platoon!" << endl;
                    feedbackMsg.m_JoinAllowed = Permission::ALLOWED;
                    int squence = PlatooningManager::getInstance()->getNumOfFollowingTruck() + 1;
                    Truck* newTruck = new Truck(m_truckInfo.m_truckID, m_addrClient, squence);
                    PlatooningManager::getInstance()->addTruck(m_truckInfo.m_truckID, newTruck);
                    cout << "Successfully join to the platoon!" << endl;
                }
                else
                    feedbackMsg.m_JoinAllowed = Permission::REFUSED;
                break;
            case RequestType::LEAVE_REQUEST:
```

Figure 16. Sever Message Handler Execution

5. GPU Realization of Part of the algorithm

To improve the calculating speed, parts of the algorithm are moved to the GPU device. The position calculation of each following truck and distance check is the parts moved to GPU. On the leading truck, there is a GPS to acquire the current position of the leading truck i.e. coordinate of the leading truck then according to the distance with the front truck sent from the following trucks and the following sequence, the position of each following truck can be calculated. The distance between the front truck sent from every following truck should be inspected. If the distance to

too close, the leading truck will send control feedback to the following truck to speed down to increase the distance.

As Figure 15 presents, this is the cuda kernel function to achieve the calculation in parallelism by GPU threads. There are some mandatory steps to be ready for calling kernel functions such that allocating GPU memory, and copying input array from host memory to GPU buffers. Then it is feasible to call the kernel function. After the kernel function finishes, copy the output array of the kernel function from the GPU device to the Host. Finally, device memory should be free.

```
__global__ void positionCalculateExecution(float* coordinate, float* distance,
                                           float relativePos[MAX_SIZE][2], bool* distanceCheck)
{
    const int tid = threadIdx.x;

    //calculate relative distance
    if (tid == 0)
        relativePos[tid][0] = distance[0];
    else
    {
        for (int i = tid; i >= 0; i--)
        {
            relativePos[tid][0] = distance[tid];
        }
    }
    // check reasonability of distance
    if (distance[tid] > MINIMUN_SPACING)
        distanceCheck[tid] = true;
    else
        distanceCheck[tid] = false;
}
```

Figure 17. CUDA kernel function

B. Client

1. Parallelism Realization

For the Parallelism part, based on C++ 14 std::thread, one main thread, and one working thread is used in this system. The working thread is set to send messages to the leading truck as Figure 16 shows.

```
/*thread Configuration*/
thread sendThread(&TransmissionHandler::sendMsgThreadExecution, this);
sendThread.detach();
```

Figure 18. Client Sending Message Thread Initialization

```
while (1)
{
    curTime = clock();
    if ((curTime - tempTime) * 1000 / CLOCKS_PER_SEC > timeOut * 1000)
        wantToLeave = true;

    if (wantToJoin)
    {
        wantToJoin = false;
        truckInfo.m_request = 1;
        truckInfo.m_velocity = msgHandler.getCurVelocity();
        tansHandler.sendMsg(truckInfo);
    }

    msgFeedback = tansHandler.recvMsg();
    int end = msgHandler.Handle(msgFeedback, wantToJoin);
    if (end == 1)
        break;

    if ((curTime - sendTime) * 1000 / CLOCKS_PER_SEC > 5000)//send per 2s
    {
        sendTime = clock();
        truckInfo.m_request = 0;
        truckInfo.m_velocity = 100 * msgHandler.getCurVelocity();
        truckInfo.m_distance = 100 * msgHandler.getCurDistance();
        tansHandler.sendMsg(truckInfo);
    }
```

Figure 19. Client Main thread

The workflow is that the main thread takes responsibility to receive a message from the leading truck then parses the message to a readable data structure and delivers the data structure to a message handler that can handle the message as Figure 17 shows. Meantime the *send message* thread can be waked up to send the message including the relevant request of the following truck. Following the communication realization part will do a detailed description.

2. Communication Realization

Network configuration based on the windows2 TCP/IP standard is presented in Figure 18. There are some configuration steps i.e. loading the socket library, and creating a socket. Address information including IP address and port is set into ***SOCKADDR_IN***. After that, the following truck using UDPSocket with the predefined ***SOCKADDR_IN*** can send a message to the server.

```
sockSrv = socket(AF_INET, SOCK_DGRAM, 0);


inet_pton(AF_INET, "192.168.178.48", &addrSrv.sin_addr.S_un.S_addr);
addrSrv.sin_family = AF_INET;
addrSrv.sin_port = htons(11500);
```

Figure 20. UDP Client Configuration

```
recvfrom(sockSrv, recvBuf, 100, 0, (SOCKADDR*)&addrClient, &len);
memcpy(&msgFeedback, recvBuf, sizeof(SMessageFeedBack));
```

Figure 21. UDP Client Receive Function

As Figure 19 shows, this is the receive message function of the client. Throughout the calling ***recvfrom*** function, the server can receive network messages from the leading truck. The raw message is stored in the ***recvBuff*** being a char array and this raw message will be parsed to a readable data structure.

```
memcpy(sendBuf, &truckInfo, sizeof(SFollowingTruckInfo));
sendto(sockSrv, sendBuf, 100, 0, (SOCKADDR*)&addrSrv, len);
```

Figure 22. UDP Client Send Configuration

As Figure 20 shows, this is the send message function of the client. Firstly, the message structure should be transferred to the corresponding **char*** type stored in the ***sendBuff***, then through calling ***sendTo*** function, the following truck can send a network message to the leading truck.

3. Message Handle Realization

Considering limited resources and loading performance in the real-time hardware system, in the following trucks, the message handler shares the same thread with receive message function.

In the execution body of the thread task, there is a switch case gramma used to handle different types of feedback messages. So the variable feedback type is the objective of the switch. There are totally three feedback types i.e. control feedback, join feedback, and leave feedback meaning three

different cases. If the feedback type is control feedback, using the data regarding the velocity of the leading truck and current distance with the front truck, the system is about to calculate the desired distance with the front truck based on the CACC algorithm, then controls the truck moving velocity to approach the desired distance.

```
int MessageHandler::Handle(const SMessageFeedBack& msgFeedback, bool & wantToJoin)
{
    switch (msgFeedback.m_FeedbackType)
    {
    case FeedBackType::CONTROL_FEEDBACK:
        desiredVelovity = (float)msgFeedback.m_velocity / 100;
        if (desiredVelovity == 0)
            return 0;
        cout << "Velocity of Leading Truck:" << desiredVelovity << " km/h" << endl;
        cout << "Velocity of Following Truck:" << curVelocoty << " km/h" << endl;
        if (curVelocoty != 0)
            desiredDistance = MINIMUN_SPACING + desiredVelovity * (curDistance / curVelocoty);
        speedControl();
        cout << "Current Distance from the front truck: " << desiredDistance << "m" << endl << endl;
        break;
    case FeedBackType::JOIN_FEEDBACK:
        if (msgFeedback.m_JoinAllowed == Permission::ALLOWED)
        {
            cout << "Recieved Feedback! Seccessful Join!" << endl;
            followingSequence = msgFeedback.m_sequenceNo;
        }
        break;
    case FeedBackType::LEAVE_FEEDBACK:
        if (msgFeedback.m_LeaveAllowed == Permission::ALLOWED)
        {
            cout << "Recieved Feedback! Allowed to Leave!" << endl;
            cout << "Successfully leave!\nWhether Join agin(Yes/No):";
            string str;
            cin >> str;
```

Figure 23. Client Message Handler Execution

V. SIMULATION

Simulation of a system is completed based on the terminal, realized communication mechanism of one server to multiple clients.

As Figure 22 shows, it is a simulation of the velocity of the leading truck that is used as virtual real-time data to reveal the reaction of the following truck and algorithm output. A sin function-based velocity generator is exploited to mock a piece of leading truck velocity and then the mock data is sent to all the following trucks by UDP socket.

```
//leading truck simulation
float leadingVelocity = 40 + 5 * sin(0.02 * PI * t);
feedback.m_velocity = leadingVelocity * 100;

for (auto iter = UDPAddressList.begin(); iter != UDPAddressList.end(); iter++)
{
    SOCKADDR_IN addr = *iter;
    memcpy(sendBuf, &feedback, sizeof(SMessageFeedBack));
    sendto(sockSrv, sendBuf, 100, 0, (SOCKADDR*)&addr, len);
}
```

Figure 24. Simulation of leading truck velocity(based on sin function)

On sever end, sever take responsibility to receive all request from each following truck and then handle the message, after which sever will send feedback message back to each following truck. The whole process is revealed on the simulation terminal, as Figure 12 shows. First of all, when the server system starts, it configures the network and is ready to receive a message from each following truck in the platoon with printing information "server is operating!".

Secondly, when there is a new join request message coming, sever will check to join permission and print information of allow or refusion and then send feedback message and add the information of this new truck to memory. When there is a control request message coming, sever will print the current velocity of the following truck, calculate the position of the following truck and check whether the following truck's current distance from the front truck is reasonable after that server will send a feedback message back to following the truck. When there is a leave request coming, sever will check to leave permission and print information of allow or refusion and then send feedback message and remove the information of this truck from memory.

```
D:\VS2019File\DPS_Code_Prj\DPS_Code_Prj\DPS_Platooning_System\x64\Debug\DPS_Platooning_System.exe
                Current Velocity is : 44.99km/h

Received the control request from Truck <1002>
                Current Velocity is : 44.96km/h

Received the control request from Truck <1001>
                Current Velocity is : 44.91km/h

Received the control request from Truck <1003>
                Current Velocity is : 44.75km/h

Received the control request from Truck <1004>
                Current Velocity is : 44.75km/h

Received the control request from Truck <1002>
                Current Velocity is : 44.65km/h

Received the control request from Truck <1001>
                Current Velocity is : 44.52km/h

Received the control request from Truck <1003>
                Current Velocity is : 44.22km/h

Received the control request from Truck <1004>
                Current Velocity is : 44.22km/h

Received the leave request from Truck <1002>
                Permit to Leave the platoon!
                ...Permit to leave the platoon!
```

Figure 25. Sever End Simulation Terminal

On the client end, multiple clients can be communicated simultaneously with the server. For each following truck, the client takes responsibility to receive network messages from the leading truck and parsing network messages to readable feedback data structure, after which the following truck is going to handle the feedback. According to different feedback types, there is a different reaction. To be more detailed. As Figure 24 presents, join feedback is coming, and the following truck will join the platoon or keep waiting and requesting. When control feedback is coming, the following truck will use the data velocity of the leading truck from feedback information and data current distance with the front truck from the sensor to calculate the desired distance with the front truck based on the CACC algorithm. According to the output of CACC i.e. desired distance with the front truck, the velocity of the following truck is about to be modified to approach the desired distance with the front truck. The real-time velocity of the leading truck and following truck is revealed as well as the real-time distance with the front truck is revealed on the terminals.

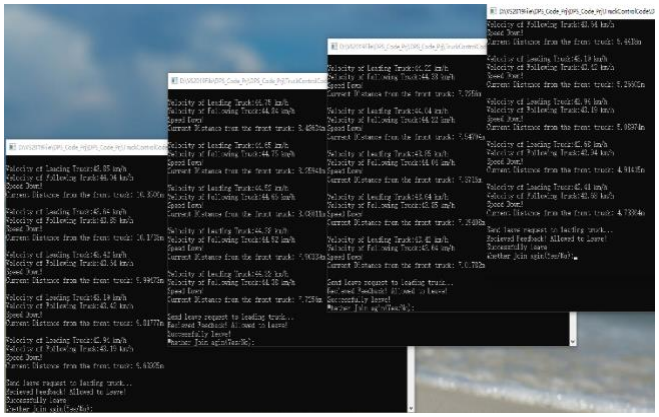

Figure 26. Client Ends Simulation Terminal

VI. CONCLUSION

In conclusion, the project implements a multi-threaded system for platooning trucks, using networking protocols along with **C++14 std::thread** and synchronization primitives such as **std::mutex** and **std::lock_gaurd** to ensure thread-safety and avoid race conditions. The proposed system includes a Meyer's Singleton to manage shared resources, with a critical section protected by mutexes to guarantee synchronous access. The server receives messages from each truck and handles them accordingly, while the client receives feedback from the server and adjusts the following truck's velocity to maintain a safe distance from the leading truck. The system utilizes the **CACC algorithm** to keep a safe and self-adjusting distance varied with the velocity of the leading truck. Overall, the system provides a flexible and manageable interface for communication and control of platooning trucks.

ACKNOWLEDGEMENT

We would like to take this opportunity to express our sincere gratitude and appreciation to Professor Dr. Henkler for his guidance and support throughout our project. His invaluable insights, constructive feedback and encouragement have been instrumental in shaping our project and enabling us to achieve our goals.

We are deeply grateful to Professor Dr. Henkler for his dedication and commitment to ensuring the success of this project. His expertise and willingness to share his knowledge and experience have been a constant source of inspiration for us. We have learned a lot from him and his guidance has been an immense help in our academic and professional journey.

REFERENCES

[1] Ploeg, J., Semsar-Kazerooni, E., Lijster, G., Wouw, van de, N., & Nijmeijer, H. (2015). Graceful degradation of cooperative adaptive cruise control. IEEE Transactions on Intelligent Transportation Systems, 16(1), 488-497. https://doi.org/10.1109/TITS.2014.2349498

[2] G. Naus, R. Vugts, J. Ploeg, R. van de Molengraft, and M. Steinbuch, "Cooperative adaptive cruise control, design and experiments," in *Pro- ceedings of the 2010 American control conference*. IEEE, 2010, pp. 6145–6150.

[3] R. Vugts, "String-stable CACC design and experimental validation," *Diss. Technische Universiteit Eindhoven*, 2010.

[4] T. Stanger and L. del Re, "A model predictive cooperative adaptive cruise control approach," in *Proc. IEEE American Control Conference (ACC)*, 2013, pp. 1374–1379.

[5] R. Rajamani and C. Zhu, "Semi-autonomous adaptive cruise control systems," *IEEE Trans. Veh. Technol.*, vol. 51, no. 5, pp. 1186–1192, Sep. 2002.

[6] J. Ploeg, B. T. M. Scheepers, E. van Nunen, N. van de Wouw, and H. Nijmeijer, "Design and experimental evaluation of cooperative adaptive cruise control," in *Proc. 14th Int. IEEE Conf. Intell. Transp. Syst.*, Oct. 5–7, 2011, pp. 260–265.

[7] Z. Wang, G. Wu, P. Hao, K. Boriboonsomsin, and M. J. Barth, "Developing a platoon-wide eco-cooperative adaptive cruise control (CACC) system," in *Proc. IEEE Intell. Veh. Symposium*, Jun. 2017.