ANNEX

1. **CONTRIBUTION PERCENTAGE**

   Five members 20% each.

2. **SOURCE CODE**

   a. **Server code:**

DPS_Platooning_System.cpp:

```
#include "MessageHandler.h"
#include "TransmissionHandler.h"
#include "SFollowingTruckInfo.h"
#include "SMessageFeedBack.h"


#include <iostream>


int main()
{
    TransmissionHandler transHandler;
    transHandler.NetworkConfiguration();
    transHandler.ThreadInitialization();


    MessageHandler msgHandler;
    msgHandler.ThreadInitialization();


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

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

        Sleep(500);

        SMessageFeedBack feedback = msgHandler.getHandlingFeedBack();

        transHandler.sendMsgOnce(feedback);
    }
}
```

PlatooninManager.cpp:

```
#include "PlatooningManager.h"

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

PlatooningManager::PlatooningManager()
{
}

PlatooningManager::~PlatooningManager()
```

```
{

}


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;

}
```

```
bool PlatooningManager::removeTruck(int id)
{
    lock_guard<mutex> lg(_mtx);
    auto iter = m_trucksMap.find(id);
    if (iter != m_trucksMap.end())
    {
        m_trucksMap.erase(iter);
        return true;
    }
    return false;
}

int PlatooningManager::getNumOfFollowingTruck()
{
    _mtx.lock();
    int num = m_trucksMap.size();
    _mtx.unlock();
    return num;
}

void PlatooningManager::updateSequenceNo(int startSequence)
{
    _mtx.lock();
    for (auto iter = m_trucksMap.begin(); iter != m_trucksMap.end(); iter++)
    {
```

```
        int curSequence = iter->second->getFollowingSequence();
        if (curSequence > startSequence)
        {
            iter->second->setFollowingSequence(curSequence - 1);
        }
    }
    _mtx.unlock();
}

list<SOCKADDR_IN*> PlatooningManager::getCommunicationAddr()
{
    list<SOCKADDR_IN*> addrs;
    _mtx.lock();
    for (auto iter = m_trucksMap.begin(); iter != m_trucksMap.end(); iter++)
    {
        SOCKADDR_IN addr = iter->second->getCommunicationAddr();
        addrs.push_back(&addr);
    }
    _mtx.unlock();
    return addrs;
}
```

TransmissionHandler.cpp:

```
#include "TransmissionHandler.h"
#include <thread>
```

```
#define PORT_UDP 11500

#define PI 3.14


TransmissionHandler::TransmissionHandler()

{


}


TransmissionHandler::~TransmissionHandler()

{

closesocket(sockSrv);

WSACleanup();

}


void TransmissionHandler::NetworkConfiguration()

{

WORD wVersionRequested;

WSADATA wsaData;

int err;


wVersionRequested = MAKEWORD(2, 2);


err = WSAStartup(wVersionRequested, &wsaData);

if (err != 0)
```

```
{
cout << "Socket Lib Configuration Failed ! " << endl;
}

if (LOBYTE(wsaData.wVersion) != 2 ||
HIBYTE(wsaData.wVersion) != 2)
{
WSACleanup();
return;
}

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!";
}
```

```
}

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

void TransmissionHandler::sendMsg()
{
Sleep(4000);
int len = sizeof(SOCKADDR), t = 0;
SMessageFeedBack feedback;
feedback.m_FeedbackType = FeedBackType::CONTROL_FEEDBACK;
while (1)
{
 if (UDPAddressList.empty())
 continue;
 //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);
}
Sleep(2000);
t++;
}
}

void TransmissionHandler::sendMsgOnce(const SMessageFeedBack& msg)
{
int len = sizeof(SOCKADDR);

memcpy(sendBuf, &msg, sizeof(SMessageFeedBack));

sendto(sockSrv, sendBuf, 100, 0, (SOCKADDR*)&addrClient, len);
}

void TransmissionHandler::sendMsgThreadExecution()
{
int len = sizeof(SOCKADDR);
while (1) {
unique_lock <std::mutex> lck(_mtx);
while (!_wakeUp)
```

```
_cv.wait(lck);

memcpy(sendBuf, &truckInfo, sizeof(SFollowingTruckInfo));

sendto(sockSrv, sendBuf, 100, 0, (SOCKADDR*)&addrSrv, len);

_wakeUp = false;
}
}

SOCKADDR_IN TransmissionHandler::clientAddr()
{
return addrClient;
}

SFollowingTruckInfo TransmissionHandler::recvMsg()
{
int len = sizeof(SOCKADDR);
SFollowingTruckInfo msg;
recvfrom(sockSrv, recvBuf, 100, 0, (SOCKADDR*)&addrClient, &len);
memcpy(&msg, recvBuf, sizeof(SFollowingTruckInfo));
if (msg.m_request == RequestType::JOIN_REQUEST)
UDPAddressList.push_back(addrClient);
else if (msg.m_request == RequestType::LEAVE_REQUEST)
{
```

```
auto iter = find(UDPAddressList.begin(), UDPAddressList.end(), addrClient);
if (iter != UDPAddressList.end())
UDPAddressList.erase(iter);
}

return msg;
}
```

MessageHandler.cpp:

```
#include "MessageHandler.h"

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

void MessageHandler::OnMessage(const SFollowingTruckInfo& truckInfo, const SOCKADDR_IN& addr)
{
m_truckInfo = truckInfo;
m_addrClient = addr;
if (!m_wakeUp)
{
m_wakeUp = true;
```

```
m_cv.notify_one();
}
}

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:
cout << "\n";
cout << "Received the leave request from Truck <" << m_truckInfo.m_truckID << ">" << endl;
feedbackMsg.m_FeedbackType = FeedBackType::LEAVE_FEEDBACK;
if (LeavePermissionCheck())
{
cout << "                    Permit to Leave the platoon!" << endl;
cout << "                    ...Permit to leave the platoon!" << endl;
feedbackMsg.m_LeaveAllowed = Permission::ALLOWED;
int curSequence = m_truckInfo.m_followingSequenceNo;
PlatooningManager::getInstance()->updateSequenceNo(curSequence);
```

```
}
else
m_feedbak.m_LeaveAllowed = Permission::REFUSED;
break;
default:
break;
}
m_feedbak = feedbackMsg;

m_wakeUp = false;
}
}

bool MessageHandler::JoinPermissionCheck()
{
return true;
}

bool MessageHandler::LeavePermissionCheck()
{
return true;
}

SMessageFeedBack MessageHandler::getHandlingFeedBack()
{
```

```
return m_feedbak;
}
```

b. **GPU code:**

CUDAFunction.cu:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#define MINIMUN_SPACING 3
#define MAX_SIZE 10

__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;

}

//coordinate[2]:        coordinate of leading truck.two elements(x,y)
//distance[size]:       distance with front truck sorted by following sequence
//relativePos[size][2]: relative distance based on leading truck
//size:                 number of following truck
//distanceCheck:        check distance is reasonanle or not.true:reasonable false:not reasonable
extern "C" void PositionCalculation(float* coordinate, float* distance, float relativePos[MAX_SIZE][2],
bool *distanceCheck, int size)
{
float* dev_coordinate;
float* dev_dis;
float dev_rPos[MAX_SIZE][2];
bool* dev_check;

//allocate GPU memory
```

```
cudaMalloc((void**)&dev_coordinate, 2 * sizeof(float));
cudaMalloc((void**)&dev_dis, size * sizeof(float));
cudaMalloc((void**)&dev_rPos, 2 * size * sizeof(float));
cudaMalloc((void**)&dev_check, size * sizeof(bool));

//Copy input arry from host memory to GPU buffers.
cudaMemcpy(dev_coordinate, coordinate, 2 * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(dev_dis, coordinate, size * sizeof(float), cudaMemcpyHostToDevice);

//call kernel function
positionCalculateExecution << <1, size >> > (dev_coordinate, dev_dis, dev_rPos, dev_check);

//Copy output arry from GPU device to Host
cudaMemcpy(relativePos, dev_rPos, size * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(distanceCheck, dev_check, size * sizeof(float), cudaMemcpyDeviceToHost);

//Free device memory
cudaFree(dev_coordinate);
cudaFree(dev_dis);
cudaFree(dev_rPos);
cudaFree(dev_check);
}
```

c. Client code

TruckControlCode.cpp:

```
#include <iostream>
#include <Winsock2.h>
#include <stdio.h>
#include <Ws2tcpip.h>
#include <thread>
#pragma comment(lib, "ws2_32.lib")
#include "SFollowingTruckInfo.h"
#include "SMessageFeedBack.h"
#include "TransmissionHandler.h"
#include "MessageHandler.h"

using namespace std;

SMessageFeedBack msgFeedback;

bool wantToLeave = false;
bool wantToJoin = true;

SFollowingTruckInfo truckInfo;

int main()
{
TransmissionHandler tansHandler;
tansHandler.Configuration();
MessageHandler msgHandler;
clock_t curTime, tempTime, sendTime;
int timeOut;

cout << "Set Truck Info:\n"
<< "Truck ID(int):";
cin >> truckInfo.m_truckID;
cout << "Current Velocity(km/h):";
cin >> truckInfo.m_velocity;
cout << "Current Distance(m):";
cin >> truckInfo.m_distance;
cout << "Following Time(s):";
cin >> timeOut;
msgHandler.setCurDistance(truckInfo.m_distance);
msgHandler.setCurVelocity(truckInfo.m_velocity);
```

```
curTime = clock();
tempTime = clock();
sendTime = clock();

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);
}


if (wantToLeave)
{
wantToLeave = false;
tempTime = clock();
truckInfo.m_request = 2;
truckInfo.m_followingSequenceNo = msgHandler.getFollowingSequence();
cout << "Send leave request to leading truck...\n";
tansHandler.sendMsg(truckInfo);
}

}

}
```

TransmissionHandler.cpp:

```
#include "TransmissionHandler.h"
#include <thread>

TransmissionHandler::TransmissionHandler()
{
}

TransmissionHandler::~TransmissionHandler()
{
closesocket(sockSrv);
WSACleanup();
}

void TransmissionHandler::Configuration()
{
WORD wVersionRequested;
WSADATA wsaData;
int err;

wVersionRequested = MAKEWORD(2, 2);

err = WSAStartup(wVersionRequested, &wsaData);
if (err != 0)
{
cout << "Socket Lib Configuration Failed ! " << endl;
}

if (LOBYTE(wsaData.wVersion) != 2 ||
HIBYTE(wsaData.wVersion) != 2)
{
WSACleanup();
}

printf("Client is operating!\n\n");

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);


/*thread Configuration*/
```

```
thread sendThread(&TransmissionHandler::sendMsgThreadExecution, this);
sendThread.detach();
}

void TransmissionHandler::sendMsg(const SFollowingTruckInfo& msg)
{
truckInfo = msg;
//wake up thread
if (!_wakeUp)
{
_wakeUp = true;
_cv.notify_one();
}
}

void TransmissionHandler::sendMsgThreadExecution()
{
int len = sizeof(SOCKADDR);
while (1) {
unique_lock <std::mutex> lck(_mtx);
while (!_wakeUp)
_cv.wait(lck);

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

_wakeUp = false;
}
}

SMessageFeedBack TransmissionHandler::recvMsg()
{
int len = sizeof(SOCKADDR);
SMessageFeedBack msgFeedback;
recvfrom(sockSrv, recvBuf, 100, 0, (SOCKADDR*)&addrClient, &len);
memcpy(&msgFeedback, recvBuf, sizeof(SMessageFeedBack));

return msgFeedback;
}
```