



CHALMERS
UNIVERSITY OF TECHNOLOGY



Multi-user Driving Simulation

Master's Thesis in Master Programme of Communication Engineering

Jiahui Liu and Yanni Xie

MASTER'S THESIS 2018:EX098

Multi-user Driving Simulation

Jiahui Liu and Yanni Xie



Department of Electrical Engineering
Communication Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Multi-user Driving Simulation
Jiahui Liu and Yanni Xie

© Jiahui Liu and Yanni Xie, 2018.

Supervisor: Emil Knabe, Volvo Car Corporation
Examiner: Tommy Svensson, Chalmers University of Technology
Supervisor: Hao Guo, Chalmers University of Technology

Master's Thesis 2018:EX098
Department of Electrical Engineering
Master Programme of Communication Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone: Jiahui Liu +46(0)790398287 Yanni Xie +46(0)762838667

Typeset in L^AT_EX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2018

Abstract

Functionalities of a supportive and autonomous car need to be tested in the context of traffic scenarios. However, due to time and space limitations, real scenario testings are usually unpractical for design and refine purpose. In this case, virtual testing including software simulation is a pretty good alternative. This thesis aims to implement a real-time cross-platform multi-user driving simulation under a specific scenario for the convenience of testers.

In this thesis, we firstly give an introduction on several important concepts about driving simulation as well as the simulation platforms that we used. The driving simulation is mostly conducted in Unity which is a user-friendly game engine with high flexibility on development and good visualization effects on rendering. The scope of the thesis is also proposed in details in the first section. Then we present the implementation process of the simulation. The focus of the simulation is on the coordination and communication among distributed vehicle objects on different entities. A modified client-server network architecture is applied and a networking algorithm is developed to simulate the coordination. To build communication links over the network, we propose a protocol denoted as the SimS protocol which is based on User Datagram Protocol (UDP) in the simulation. In addition, some smoothing strategies to improve the visualization performance of the simulation, e.g., interpolation and extrapolation, are introduced and implemented as well. In the third section of the thesis, we present the simulation results, performing some analysis together with doing comparisons with regard to different settings of simulation parameters. The simulation is firstly conducted within Unity and then extended to cross-platform, namely the integration between Unity and VIRET Virtual Test Drive (VIRET VTD). Finally, we conclude the work and propose some potential future work on network time protocol (NTP) to further improve the simulation performance.

Keywords: Driving simulation, Unity, Real time, Networking, Client-server model, UDP, SimS protocol, Extrapolation, Cut-in scenario.

Acknowledgements

We would like to take this opportunity to express our sincere gratitude and appreciation to Emil Knabe, Tommy Svensson and Hao Guo.

Our deep gratitude goes to our supervisor at Volvo Cars, Emil Knabe who has provided guidance and support as a driving force throughout this master thesis. You arranged the thesis following meeting each week, providing us great patience and help. We spent very enjoyable time with you and got so many inspirations from you.

We would like to express our appreciation to our examiner Prof. Tommy Svensson and our supervisor PhD candidate Hao Guo at Chalmers University of Technology, for your great help and support throughout this thesis work. Your inputs and suggestions are of great value to this thesis.

A special thanks goes to all the professors and classmates in MPCOM. We had a memorable experience throughout the two-year master study at Chalmers. We are thankful to our coworkers at Volvo Cars as well for your showing an interest in this thesis project and the results.

Jiahui Liu and Yanni Xie, Gothenburg, September 2018

Contents

List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Background	1
1.1.1 Multi-user Driving Test	1
1.1.2 Simulation Platform	1
1.1.3 Integrated Development Environment	2
1.1.4 Multi-platform Integration	3
1.1.5 Simulation Scenarios	3
1.2 Aims and Scope	4
1.2.1 Aims	4
1.2.2 Scope	5
1.3 Methodology	5
1.4 Outline	6
2 Implementation	7
2.1 Network Configuration	7
2.1.1 Network Architecture and Synchronization	7
2.1.2 Communication Protocols on Transport Layer	9
2.1.3 Network Algorithm	11
2.2 Communication Protocol (SimS Protocol)	16
2.2.1 Vehicle Mono-behaviour	17
2.2.2 Runtime Data Bus (RDB)	17
2.2.3 Package Structure	17
2.2.4 Port Specification	19
2.2.5 Serialization and Parsing	20
2.3 Smoothing Strategy	21
2.3.1 Extrapolation	21
2.3.2 Interpolation	22
2.3.3 Smooth Camera Follow	23
3 Results and Analysis	25
3.1 Simulation Results	25
3.1.1 Simulation within Unity	25
3.1.2 Simulation on Multiple Platforms	27

3.2	Performance Comparison	29
3.2.1	Effects of Extrapolations	29
3.2.2	WiFi and Ethernet	29
4	Closure	33
4.1	Conclusions	33
4.2	Future Work	33
	Bibliography	35

List of Figures

1.1	The User Interface of Unity Editor	2
1.2	HIL dSPACE Simulator	3
1.3	Pilot System Setup	4
2.1	Illustration of Two Network Architectures	8
	(a) Peer-to-peer Networking	8
	(b) Client-server Model	8
2.2	The Open Systems Interconnection (OSI) Model	10
2.3	Illustration of the Network within Unity	11
2.4	Execution Order of Event Functions in Unity Scripts [3]	12
2.5	Illustration for the Ring Buffers on a Client's Side	15
2.6	The Network Work-flow	16
2.7	Port specification	19
2.8	Illustration for Extrapolation	22
2.9	Illustration for Interpolation	22
3.1	Illustration for Multi-user Driving Simulation within Unity	26
3.2	Vehicle A's Trajectories on the Two Clients	26
3.3	Illustration of Extended Cut-in Scenario	27
3.4	The synchronization among different simulation platforms	28
3.5	Illustration of Extrapolation Effects	29
3.6	Illustration of the Rate of Delayed or Lost Package under WiFi and Ethernet Connections	30

List of Tables

1.1	Comparison between Unity and Unreal Engine	2
1.2	The set up of the system	5
2.1	Comparison between Peer-to-peer Networking and Client-server Model	9
2.2	Comparison between UDP and TCP	11
2.3	Package Structure	18
3.1	Comparison on Average Behaviour between WiFi and Ethernet	30

1

Introduction

1.1 Background

1.1.1 Multi-user Driving Test

Functionalities of supportive and autonomous cars need to be tested in the context of traffic scenarios. In perspective of software test, this kind of simulation is typically done by modeling one human-controlled vehicle denoted as ego vehicle, and a number of predefined virtual road users denoted as dummy vehicles [1, 2]. However, there is a growing need for including two or more ego vehicles in the same scenario. One case is a head-to-head exploratory simulation test where two testers sit next to each other manipulating their own ego-vehicle object via different simulators. In this way they can communicate easily on tuning specific maneuvers, which brings benefits to function tests, feature explorations and system demonstrations.

The real-time multi-user driving simulation presented above is much similar to the concept of multi-player computer games. In a simulation session of multiple ego vehicles, the distributed simulators are connected either into a Local Area Network (LAN) when there is a physically short range among them or via the Internet for a long distance case. These two cases are similar to the design of multi-player LAN games and online games, respectively. Hence, it's promising to investigate real-time interaction solutions in game industry and build new solutions to the multi-user driving simulation in automotive industry.

1.1.2 Simulation Platform

To begin with, we need to select a simulation platform. From the perspective of visualization, a game engine provides a decent user interface in 3D animation. Moreover, a game engine provides both high level and low level application programming interfaces (APIs), making it quite flexible to develop. Therefore, a game engine is very appropriate to be a driving simulation platform. The most two popular ones nowadays are Unity [3] and Unreal [4] Engine. A comparison of them is shown in Table 1.1.

1. Introduction

Game Engine	Programming Languages	Graphical Capabilities	Hardware Demands	Cross-platform
Unity	C#, JavaScript	Strong	Lower	Yes
Unreal	C++, Blueprints	More Powerful	Higher	Yes

Table 1.1: Comparison between Unity and Unreal Engine

As is shown above, Unity uses C# and JavaScript as programming language while Unreal Engine uses Blueprints Visual Scripting for novice programmers and C++ for further developments[5, 6]. In terms of graphic visualization, Unreal Engine has a more graphically intensive shading model than Unity's standard shader, which means Unreal has better graphics potential and is more artist friendly. And Unreal Engine relies more on hardware due to its editor, C++ compiling, and material building, which means Unity is faster and cleaner. Both of them are cross-platform game engine supporting multiple platforms such as Windows, Linux and PlayStation, but Unity makes the development process for mobile games a lot easier than Unreal. In addition, Unity provides a wide assets store and many tutorials for personal development and its editor is more user friendly. Considering the case of use and our focus on function realization rather than graphics, we choose Unity for the development in our project. The user interface of Unity editor is shown in Figure 1.1

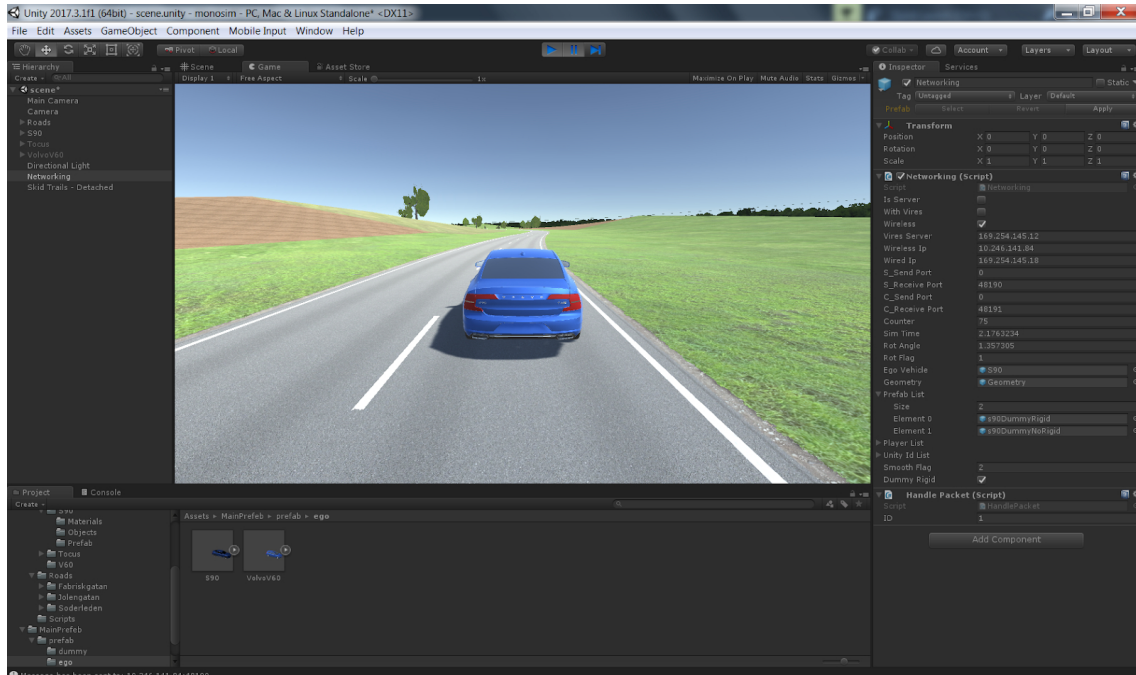


Figure 1.1: The User Interface of Unity Editor

1.1.3 Integrated Development Environment

Visual Studio is an integrated development environment (IDE) [7]. One can use Visual Studio to write code with the language of C++, C#, Visual Basic, JavaScript,

TypeScript, Python, and more[7, 8]. Visual Studio is used in our project to support C# code as Unity utilizes C# scripts to describe the behaviour of objects.

1.1.4 Multi-platform Integration

Besides game engines, there are many other driving simulation platforms. For example, Hardware-in-the-loop (HIL) dSPACE simulator which is shown in Figure 1.2, as a real-time simulator including processors and input/output cards, together with VIRES Virtual Test Drive (VIRES VTD) desktop application for visualization is widely used as a simulation platform in the automotive industry. It would be powerful putting a bunch of virtual vehicle objects from different driving simulation platforms into the same network and making them communicate with each other. To realize this kind of integration including real-time communications and coordinations, a common protocol defining the communication regulations and specifying rules on data exchange, named as Simulation Scenarios (SimS) protocol in our case, needs to be designed.



Figure 1.2: HIL dSPACE Simulator

1.1.5 Simulation Scenarios

Scenario is a repeatable and executable setup of scene, traffic and various kind of deterministic actions. In our case, scenarios are written in Extensible Markup Language format and can be edited in any text editor. Several driving scenarios are quite representative and essential to the test of autonomous driving functions and will be used in our project, such as the cut-in scenario and the highway-merge scenario.

The static objects included in these simulation scenarios, such as the road network and the traffic events for dummy vehicles, will be defined by OpenSCENARIO files.

1.2 Aims and Scope

1.2.1 Aims

In a nutshell, the aim of this project is to build up a pilot driving simulation system where multiple ego and dummy vehicles join together in a common simulation session such that some functions in autonomous driving could be tested in a easier way. The illustration of the whole architecture is shown in Figure 1.3.

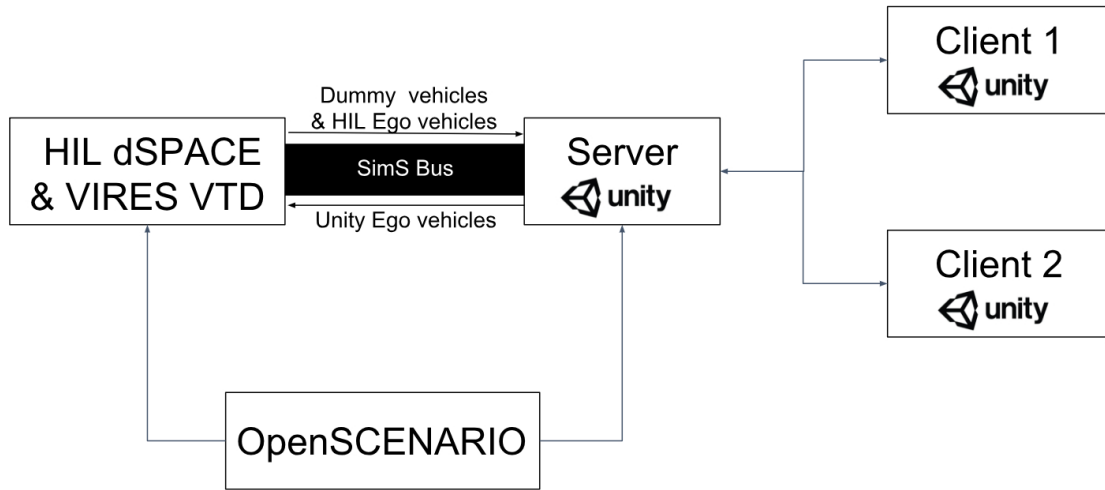


Figure 1.3: Pilot System Setup

As is shown in Figure 1.3, the system is mainly composed of two parts. The first part is to build up the network of distributed simulators which only simulates ego vehicles in Unity applications. There are many choices that need to be done about network setups including communication range of the network, concurrent users, network architecture and transport layer protocols. Some of them are limited by the scope below. The second part is to realize the integration between Unity and the other simulation platforms, which in our case is HIL dSPACE simulator together with VIRES VTD application that simulates some other ego vehicles and dummy vehicles. The SimS protocol used by the whole system needs to be defined. Meanwhile, the simulation scenario of this pilot system is controlled by OpenSCENARIO files.

The aims of this project are described in details as below:

- Discover the challenges in coordination and interactions between distributed simulators in real-time system. Investigate the corresponding solutions to the networking of multiple players in video game industry. Find benefits and limitations of different strategies with regard to latency and robustness under various network setup cases.
- Explore the differences on requirements between our automotive simulation session and pure multi-player games. Design the networking of multiple simulated ego vehicles within standalone Unity applications. Evaluate the performance by measuring the latency.
- Define and realize the common protocol denoted as SimS protocol that is used by the whole pilot driving simulation system. The protocol includes message type, package structure, package payload, the way to communicate and so on.

1.2.2 Scope

The choices on network setups have a direct effect on the networking solution to real-time system. The range of the choices is limited in this part. There are four combinations in total with regard to the range of the network and concurrent users. The former is a reflection of the system tolerance on delay while the latter affects communication capacity and the choice on network architecture. The category in details is shown in Table 1.2.

Network	Concurrent users	
	2	>2
Local Area Network (short range)		✓
the Internet (long range)		

Table 1.2: The set up of the system

Among the four cases, although the one with two concurrent users connecting to LAN might be the most common setup in the sense of the practical manipulation environment, we'll mainly focus on the one where more than two concurrent users are connected into LAN because it would be more flexible and could be easily scaled down. However, the other three cases are under discussion to some extent in this thesis.

1.3 Methodology

The thesis is carried out by two students. We cooperate and discuss with each other all the time. We work in parallel in most of the phases using different methods so that each of us could have a comprehensive understanding on the topic and a better

way together with a backup plan could always be found.

The whole process of this thesis are mainly divided into three phases, namely the strategy study phase, the implementation phase, and the phase of test and further improvements. These phases are conducted in a serial sequence while there could be overlaps between the phases.

There are some differences between building a console app using visual studio and building a Unity app using visual studio. Most of the functions in the thesis work are firstly built in console apps. When they are tested to work properly, they are added to Unity development.

1.4 Outline

In Chapter 2 we present the whole implementation part of the thesis work in depth. Strategies, algorithms and key methods to realize a function are included. As indicated in the aims and scope part, the implementation starts from networking within Unity then expands to Unity adaptation to the common protocol. In Chapter 3 we show the simulation results and perform some analysis with regard to these results. In Chapter 4 we draw a conclusion of the thesis topic and describe the related future work for further improvements.

2

Implementation

2.1 Network Configuration

This section defines the network behaviour of multiple users within Unity, dealing with communication and state synchronization issues between multiple terminals.

Subsection 2.1.1 lists advantages and limitations of two kinds of network architecture and explains the reason to choose client-server model as the proper one for this project. Some modifications based on the typical client-server model in this project are mentioned as well. Subsection 2.1.2 presents the selection of the proper communication protocol for the network on transport layer. Subsection 2.1.3 explains several techniques and the whole algorithm applied in building up this network. The techniques include asynchronous methods, ring buffers, non-authoritative server, etc.

2.1.1 Network Architecture and Synchronization

The simulation is ultimately run on different computers and manipulated by multiple end users, which means it is a distributed application. When it comes to a distributed system, synchronization on the states between multiple machines becomes vital. The way to conduct synchronization depends on network architecture which refers to how the computers are organized and how the workload is partitioned in distributed computing. Two basic architectures that are widely used are peer-to-peer (P2P) networking [9] and client-server model [10]. The illustration of them is shown in Figure 2.1.

For peer-to-peer networking, as shown in Figure 2.1 (a), there is no central entity and peers exchange information among each other in a fully connected mesh topology. Each peer controls its own state and works as a mealy machine where the output state is determined by the current state and the current inputs. Ensured the same initial state and inputs each time, all the peers will end up with the same status, which is the key to synchronize.

Although this kind of network seems to be easy to implement because of the identical role of each peer, there could be several limitations. Firstly, it could actually be very hard to make it stable because it's exceptionally difficult to ensure that the simulation is completely deterministic [11]. (A process is deterministic means that the next state could be predicted based on the data in the present state.) Only

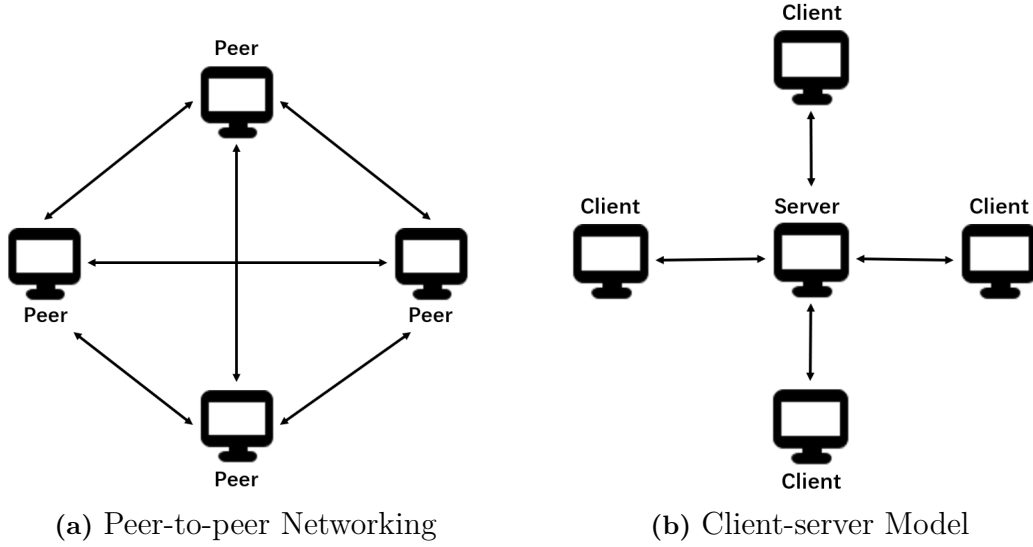


Figure 2.1: Illustration of Two Network Architectures

tiny difference, such as the difference between peers on computation, will result in complete out-of-synchronization over time. Secondly, in order to synchronize the states, a peer within the network needs to wait until all the users' commands for that turn are received before simulating that turn. This means that each user has the latency that equals to the most lagged user within a turn. Thirdly, it's hard for a user to join a simulation in the middle of a process because it's very hard to decide the initial state of the inserted user in practice.

Being different from peer-to-peer networking, there is a central entity (the authoritative server) that controls the whole status in client-server model, as is shown in Figure 2.1 (b). Except for the server, all the other entities are called clients and they don't exchange information with each other. According to the star topology, every client connected to the server constantly sends requests to the server and receives updates from the server, locally creating a representation of the simulation state. In this way, it is only the server that deals with all the user inputs and conducts the computation, thus guaranteeing the state synchronization by sending identical data to all the clients within each update.

It's obvious that the client-server model could solve some of the limitations mentioned above in the peer-to-peer part. It is not necessary to ensure the simulation being completely deterministic. The latency is shortened much if the server sits in a data center with a high-performance backbone connection. The server could always provide the most recent state of the simulation so that it's easier for a new client to join in the middle. However, client-server model has some limitations with regard to the other aspects as well, such as the required bandwidth. Bandwidth plays an important role in communication. The problem is that the server needs to propagate every detail needed to be synchronized to all the clients within each update. In contrast, a peer only needs to send its own user input to another, thus

saving bandwidth a lot.

The overall comparison between these two architectures is shown in Table 2.1. Note that the disadvantages of peer-to-peer networking can be enlarged with the increase of concurrent users. Peers are easier to be out of synchronization [12]. In game industry, this architecture is only applied for turn-based (round-based) games because those cases are not affected that much by latency. Considering our driving simulation is in real time which means it's sensitive to latency, and more than two concurrent users in the simulation as mentioned in the thesis scope, client-server model is more suitable for this simulation. Thankfully Unity supports client-server model.

Network Architecture	Stable Synchronization	Latency	Join in the Middle	Bandwidth Requirement
Peer-to-peer	Harder	Higher	Harder	Lower
Client-server	Easier	Lower	Easier	Higher

Table 2.1: Comparison between Peer-to-peer Networking and Client-server Model

Note that we don't use a typical client-server model, instead modifying it in some ways. The server in the typical client-server model is totally authoritative, which means clients are nothing but a way for the user inputs to be sampled and for the objects to be locally rendered. In this case clients have to wait for the server's response and the round-trip-time would lead to a delay between user inputs and corresponding changes on the screen, which should be improved in our real-time driving simulation. An inspiration from game industry is that a client could simulate its next movement locally while waiting for the instructions from the server and then correct the result when it isn't consistent with that from server's update. However, in contrast to multi-player game, there is no cheating behaviour in our multi-user driving simulation. A client has the authority to decide its own states and don't require the correction from the server. Therefore, a better way to balance state synchronization and visualization delay here is that a client directly reports its current status to the server after local calculation rather than sends requests or commands to the server and lets the server make the decision. Then the server is responsible for multicasting the message to the other clients as soon as it receives a message.

2.1.2 Communication Protocols on Transport Layer

When it comes to data exchange among the server and a bunch of clients, communication protocol shall be the focus. Communication protocol usually defines rules that govern the communication system, including syntax, semantics, synchronization of communication and possible error recovery methods. A classic conceptual model for a communication system is the Open Systems Interconnection (OSI) model [13]. It characterizes and standardizes the communication functions for the purpose of

2. Implementation

systems interconnection. The functions are mapped into the layers, and each layer solves a distinct class of problems relating to, for instance: application-, transport-, internet- and network interface-functions[13, 14]. A communication system could be divided into seven layers as indicated in Figure 2.2. One layer provides services to the layer above it and requests services from the layer below it.

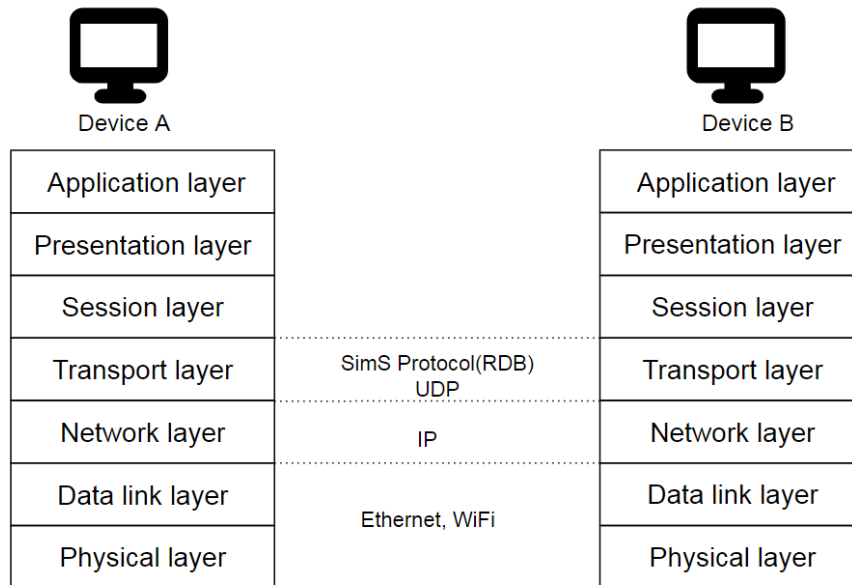


Figure 2.2: The Open Systems Interconnection (OSI) Model

To transmit a message, a protocol has to be selected in each layer. Among the seven layers, the physical layer is the most basic one that is responsible for the transmission and reception of unstructured raw data between a device and a physical transmission medium. Two protocols here that are commonly used in laboratories are Ethernet and WiFi. A comparison between their effects to the final results will be shown in subsection 3.2.2. With regard to the layers above, we select protocols from the Internet protocol suite. A foundation protocol chosen on the network layer is the Internet Protocol (IP) and our project doesn't focus on that too much. Instead our focus is on the transport layer.

The transport layer provides end-to-end communication services for applications[15, 16]. The User Datagram Protocol (UDP) is the basic transport layer protocol providing an unreliable datagram service while the Transmission Control Protocol (TCP) provides flow-control, connection establishment, and reliable transmission of data. The comparison of these two protocols is clearly shown in Table 2.2.

Protocol	Data Format	Data in Order	Re-transmission	Reliable	Real-time
UDP	Datagram	No	No	No	Yes
TCP	Segment	Yes	Yes	Yes	No

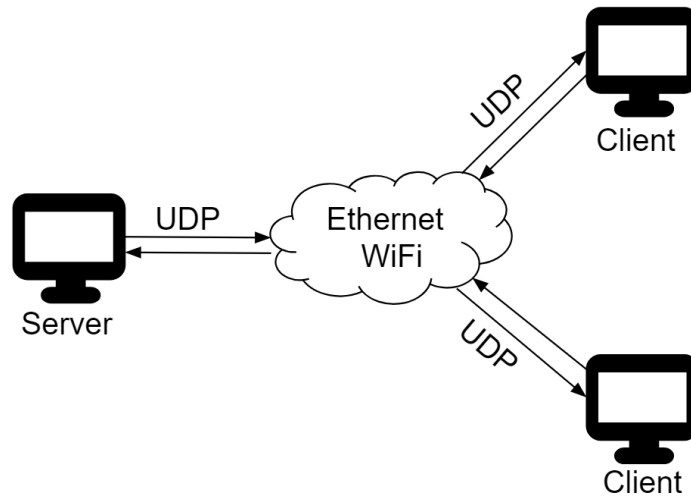
Table 2.2: Comparison between UDP and TCP

TCP provides reliable transmission service by detecting the lost data and retransmitting it. It could also guarantee a correct order of data by adding a sequence number to identify each byte of data. In contrast, UDP doesn't have a retransmission mechanism because it provides a connectionless datagram service that emphasizes reduced latency over reliability, which means that UDP is more suitable for real-time applications. Considering the goal for the project is to carry out the multi-user driving simulation in real time, UDP is selected to be the base of our transport layer protocol. Its defects on data reliability could be compensated by some techniques mentioned in the section 2.3.

With regard to the application layer, real-time transport protocol (RTP) whose implementations are mostly built on UDP seems to be a good choice. However, RTP is typically used to deliver multimedia data, including audio and video data while the information we actually need to transmit in this project is requests and responds. A simulation frame is calculated according to the inputs in each update and then rendered rather than being delivered from the server side directly. Hence, SimS protocol will be developed based on UDP and used instead of RTP.

2.1.3 Network Algorithm

Based on the above part, the network setups within Unity platform could be described as Figure 2.3. In this part, the networking workflow is explained in details.

**Figure 2.3:** Illustration of the Network within Unity

In Unity, except for the physical game objects which include ego vehicles, road networks and cameras in our project to be rendered in the scene, an empty game object i.e., game object without physical shape or characteristics, being responsible for managing networking shall be created in the hierarchy of Unity editor. Moreover, the corresponding script shall be attached to this object as its component in the game object inspector. In this way, the empty game object could act as a network manager, controlling the network behaviour using the attached script. Therefore, how Unity scripts work should be figured out firstly.

In Unity scripting, there are a number of event functions that get executed in a predetermined order as a script executes [17]. This execution order is described in Figure 2.4. A customized function could be written inside a specific event function so that it will get called by Unity at a specific stage in run-time.

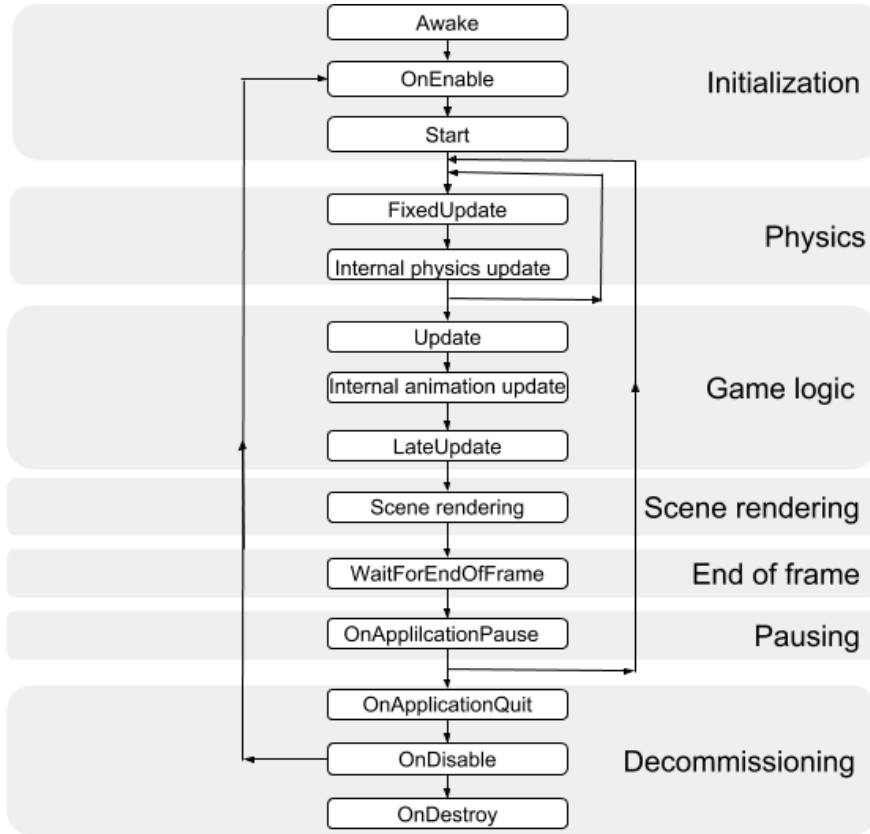


Figure 2.4: Execution Order of Event Functions in Unity Scripts [3]

To ensure multiple users staying in synchronization, we take the principle that the server and all of the clients execute the same code from the same scripts on the same game objects at the same time. To control the flow of logic, for example to assign a specific function only to the server or client, the bool value *isServer* could be set up and checked. As is shown in Figure 2.4, to control the initialization stage in run-time, code could be written inside the *Awake* function. This function is always called for the active game objects during start up before the first frame update and is only

called once. Another important task is to keep track of the simulation parameters in run-time, including logic and interactions, animations, camera positions, etc. The common way is to write this part of code inside the *Update* function which is the main workhorse function of frame updates. This function is called once per frame, allowing one to monitor inputs and other events regularly and take appropriate action. Moreover, there are other event functions in Unity we can use for updating, such as *FixedUpdate*. The difference will be discussed in the following chapter.

A basic task for a network in the initial stage is to settle up its configuration, which means to figure out the connecting users for a point within the network. A contact list indicating the IP addresses of the communication destinations could be applied for either the server or a client. According to the illustration in Figure 2.3, the only communication destination for a client is the server IP while the server's contact list should include all the IP endpoints that have talked to it so far. Therefore, the contact list of a client shall be fixed in the *Awake* function while the list of the server could be updated at any time in run-time.

When it comes to the network communication which is based on UDP as mentioned above, two main operations are sending and receiving messages. Actually there are several methods to implement these two process for *UdpClient* class in C#. However, some of them are blocking method which means another task is blocked and unable to execute until the first task has finished. In this case, an instance can't execute sending and receiving at the same time. One solution could be applying multithreading to ensure the concurrent tasks. By this way an end-point could output messages whenever it needs to without waiting until some data come in. Meanwhile, there is another solution to simplify the problem called asynchronous method. Asynchronous method allows the code to start an operation in a way that does not hold up the entire thread of execution. The framework calls you back when it's done [18]. Related events will be triggered in the callback function when being indicated that the response is available.

We choose to apply the asynchronous method for convenience of resolving the concurrent process here. Take the receiving process as an example. Firstly a *UdpClient* instance calls its asynchronous method *BeginReceive*. *BeginReceive* method will start a receiving operation that does not block the main thread. When there is something to receive, *BeginReceive* method invokes another callback method *OnReceive* and pass its asynchronous receive result in object format to *OnReceive* as inputs. A byte array containing the datagram data will be accessible then by applying the corresponding asynchronous method *EndReceive*. Subsequent events will be triggered according to the received information and the *UdpClient* instant needs to start the receiving process again. This loop will be executed from the very beginning (in *Awake* function) till the end of the simulation application. The pseudo-code about this part is shown as Algorithm 1.

Next, we discuss the flow of dual-way network communications within Unity. Note that there are two kinds of objects (vehicles) being rendered on a client's side. The

Algorithm 1 Asynchronous Receive Process

```
1: procedure BEGINRECEIVE(Callback OnReceive, Object Null)
2: end procedure
3: procedure ONRECEIVE(AsyncResult ar)
4:   ByteArray buffer  $\leftarrow$  EndReceive(AsyncResult ar, IPEndPoint remoteEP)
5:   Do some judgments and trigger some events
6:   BeginReceive(Callback OnReceive, Object Null)
7: end procedure
```

ego-vehicle controlled by the client itself is called a local vehicle and can be handled directly according to the instant inputs from keyboards. This part is discussed in the mono-behaviour part in Section 2.2. In contrast to the local vehicle, the other ego-vehicles controlled by the other clients are called remote vehicles. To control the movements of the remote vehicles, information is updated by the packets sent from the server.

To render the simulation, on one hand, a client is responsible for sending continuous status report of its local vehicle to the server. The status includes information about the ego vehicle's position, orientation, etc. Details about the format of the message are discussed in Section 2.2. This kind of message should be sent once per frame for refreshing the simulated graph. Therefore this sending process is done in the *Update* function. On the other hand, once the server gets an update message from a client, it is responsible for delivering this message to all the other clients to keep the corresponding vehicle being rendered correctly on all the clients. This update information should be sent out from the server immediately once it's received for the purpose of reducing latency.

When a client receives messages from the server which contain the status information about the remote vehicles, a key point is to apply a ring buffer for storing the current message and processing messages smoothly. The storage of the current message is realized as below: The ring buffer has a maximum size and works as a message queue. A newly received message is added to the queue while a message that has already been processed is deleted from the queue. If there are several remote vehicles being rendered on a client simultaneously, several message queues should be created and handled in parallel because we need to assign the updated state information to the corresponding remote vehicle for simulating. It means these message queues could be distinguished by the vehicle ID and there is an ID list being responsible for the registration of the active vehicle IDs. The illustration about the ring buffer is shown in Figure 2.5.

Then comes the part of processing messages in the queue. Before a message is added to a message queue, there is a pre-operation: Parse the message from a byte array to the object format which could be called object packet below. To check the available (unprocessed) packets stored in the buffer, a loop over all the existing message queues could be applied. A client shall deal with the available packets as

many as possible in a frame. This process occurs within *Update* function as well. If there is no packet to deal with in a frame because of network latency or lost datagram caused by UDP, extrapolation method would be applied. This function is discussed in Section 2.3. Otherwise just do the object synchronization, rendering the corresponding remote vehicle object according to the status data contained in the packet. Note that if the vehicle ID is identified as new, it means a new client has just joined the network. To simulate this new remote vehicle, firstly we should create the vehicle object in correct state.

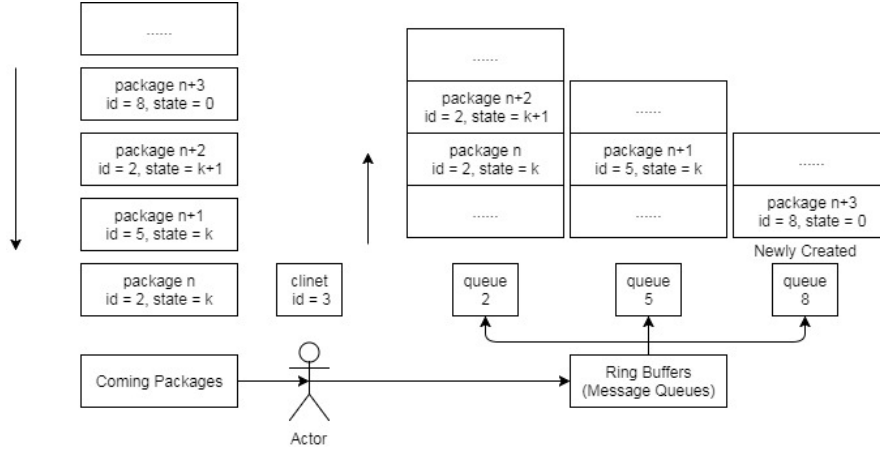


Figure 2.5: Illustration for the Ring Buffers on a Client's Side

From the above we could see that the server and clients do the same operations, sending and receiving messages in the dual-way communication, but the happening time and the triggered events are different to some extents. Although the code is the same for all the instances, different execution parts could be distinguished by the bool value *isServer*. With regard to the happening time of sending messages, clients send message each frame in the *Update* function while the server forwards the message as soon as it receives in the *OnReceive* callback. With regard to the events triggered by the received information, the cases are shown as below:

1. The *UDPCClient* instance plays a role as the server:

It should firstly check the remote IP-endpoint that sends the message to find if it is a new client joining the network. If it's true, the server needs to update its contact list by adding the new IP-endpoint. Otherwise the server could just keep its current contact list. Then the server should multicast its received data i.e., the byte array, to all the clients listed in its contact list except for the sender. Note that the sender does not need the data again from the server because it has the authority to update the status of its local vehicle. If it accepts the data from the server again, there would be a latency gap which could lead to its newly rendered local vehicle jumping forth and back.

2. The *UDPCClient* instance plays a role as a client:

It should firstly parse the received byte array to the format of independent object packets. Then register the vehicle ID and add the packet to the corresponding message queue i.e., the ring buffer, according to the ID.

Overall, the illustration for the whole network work-flow could be seen in Figure 2.6.

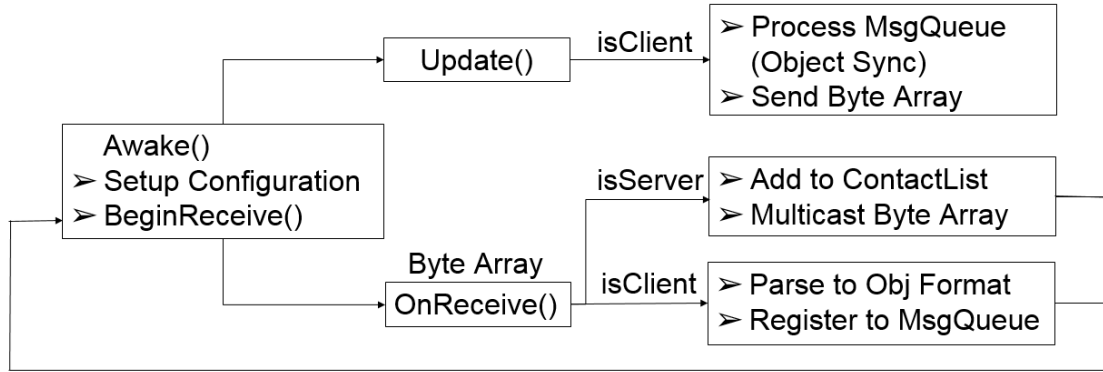


Figure 2.6: The Network Work-flow

2.2 Communication Protocol (SimS Protocol)

To communicate not only within Unity applications, but in a wider range together with the other platforms, such as HIL dSPACE simulator and VIREs VTD desktop application, a communication protocol specifying the communication rules needs to be defined. As indicated in Subsection 1.1.4, the common protocol is denoted as SimS protocol. It should regulate the exchanged message format and specify the ports for data exchange as well. With regard to the message format, we need to note that programming languages applied in different platforms could be different. For example, Unity uses C# as programming language while VIREs uses C++. They both come with built-in serialization functionality for entire object. But this functionality is not common. It varies according to the corresponding programming language, thus limiting its applicability. Therefore the exchanged message format should be defined on the level of primitive types in binary.

To achieve the integration, the logic is that the central communication is only among servers and it is the server that distributes the external messages to the clients. In terms of Unity, the external messages are received on Unity server and then external objects are rendered on all Unity clients. To be specific, when Unity server identifies the coming external messages as effective according to SimS protocol, it will instantly multicast the messages to the Unity clients that are currently on its contact list. Meanwhile, Unity server should have added the external server to its contact list so that it could send Unity internal messages to the external server, VIREs server in this case, as well.

2.2.1 Vehicle Mono-behaviour

The communication contents used for realizing coordination among distributed ego-vehicle objects highly depend on the vehicle's mono-behaviour contents. Therefore, we should have a good understanding on vehicle mono-behaviour.

In general, vehicle movements should follow physical laws, such as applying gravity. So a rigidbody component is added to the ego-vehicle object. The vehicle simulation animation follows vehicle dynamics and is basically reflected on a vehicle's position, rotation and velocity. A vehicle's rotation includes both body rotation and wheel rotations. We apply the arrow keys on the keyboard as inputs to control the status of a vehicle object while the vehicle's reaction corresponding to the inputs is defined in player controlling script.

Besides the vehicles, another important object would be the camera which records the visualization view of the user. In most cases, the camera in a standalone Unity application looks at the center of the local ego-vehicle and follows its movement. In this view, related remote vehicles and road networks within a range are included as well. There could be another case where the user wants to have a bird view to visualize all the objects included in the simulation session. In this case, the camera could be attached to a fixed position and adjust the graph resolution by scrolling the mouse.

2.2.2 Runtime Data Bus (RDB)

This thesis project is in collaboration with the Simulation Scenarios project which includes development on VIRES part. A protocol called Runtime Data Bus (RDB) developed by VIRES VTD has already been developed [19]. RDB distributes runtime data about objects, vehicle states etc. to any data consumer periodically. The interface could be configured to run via TCP, UDP, shared memory or loopback port [19]. Therefore, RDB is a good reference for SimS protocol and SimS inherits most characteristics of the RDB protocol.

We then explain the SimS protocol in details. The regulations on message formatting are mentioned in Subsection 2.2.3. The regulations on communication ports used within this network are mentioned in Subsection 2.2.4. Although SimS protocol is an interface among different driving simulation platforms, which means it could be used among Unity server and other servers, it is used among different standalone Unity applications i.e., Unity server and Unity clients, as well in this project for simplicity.

2.2.3 Package Structure

A message is conveyed in the unit of a package. A typical package is composed of headers and payloads. A header refers to supplemental data which is placed at the

beginning of a block of data. It is vital that there must be a clear and unambiguous specification or format in header composition, being beneficial to parse process. The data following the header is denoted as payload which represents the actual intended message.

To describe the status of a vehicle object completely, multiple types of contents need to be conveyed inside one package. Therefore, our package should contain several data blocks and each block describing one type of content is composed of a block header and its corresponding payload. This kind of block header is denoted as the entry header. Note that there is a type of special entry header defining the boundary of a frame, followed by no payload. Typically, a message vector for a complete simulation frame should be enclosed by *START_OF_FRAME* and *END_OF_FRAME* [19]. Besides entry headers, we also need another kind of header, denoted as message header, in the beginning of our package to mark the package as the desired one, distinguishing it from the other packages. In general, headers as management structures are introduced for the purpose of parsing the actual elements easily. They mainly contain information about the header size and the following data size. In addition, message header needs to enclose some extra information regarding to the whole simulation, such as the frame number and the simulation time.

General Category	Specific Category	Length (bytes)
MSG_HDR	MSG_HDR	24
MSG_ENTRY_HDR	START_OF_FRAME (Id = 1)	16
MSG_ENTRY_HDR	OBJECT_STATE (Id = 9)	16
	EntryOfObjectState	208
MSG_ENTRY_HDR	WHEEL_INFO (Id = 14)	16
	EntryOfWheel1	44
	EntryOfWheel2	44
	EntryOfWheel3	44
	EntryOfWheel4	44
MSG_ENTRY_HDR	END_OF_FRAME (Id = 2)	16

Table 2.3: Package Structure

The details of the package structure are listed in Table 2.3. We could see there are several kinds of entry headers which could be identified by block IDs. The ID indicates the type of following contents. As mentioned in the section of vehicle mono-behaviour, vehicle body status and wheel status are the focus. Hence, we allocate one block of object state to describe the former and one block of wheel information to describe the latter. Object state mainly contains information about the user ID, vehicle type, vehicle geometry, absolute coordinates in the virtual world, orientation of vehicle body. Extra information could be included as an extended part to improve the rendering performance with regard to accuracy, such as vehicle velocity, acceleration and accumulated travel distance. With regard to the wheel information, each of the four wheels (we assume that the vehicle is not a truck) makes an equal contribution and they are distinguished by the wheel ID. The wheel

information mainly includes the user ID, wheel ID, tire radius, spring compression, steering angle and rotation angle.

The length of each part is shown in the third column in Table 2.3. According to that, the total length of a basic package which only includes the information of one ego-vehicle is 472 bytes. However, the SimS protocol could be flexible. The status of two or more vehicles could be transmitted in one package as well. In that case, all payloads are extended to contain more information from different vehicles while all the headers are kept the same. The length of a package may vary then depending on the amount of vehicles included in the package.

2.2.4 Port Specification

Besides the package structure, SimS protocol regulates the ports used in the communication among multiple users as well. Note that the port we discuss here is at the software level. It is capable of identifying a specific process. When being binded to an IP address and a transport layer protocol (UDP in this case), a port conveys the origin or destination network address of a message.

SimS protocol defines that a host uses one port for sending process and another one for receiving process, and the two port numbers used on all clients' side are the same. However, these two port indicators for the two processes should be opposite on the server's side. For example, if we specify 48190 as the port number dealing with outgoing messages on the clients' side, it plays as the port number dealing with incoming messages on the server's side. In an ideal case, two port numbers are enough for the whole simulation. The specification is illustrated in Figure 2.7. Default port numbers are 48190 and 48191. Note that the default port numbers could be changed, but it is important to keep the rules mentioned above. All consumers need to adapt accordingly then.

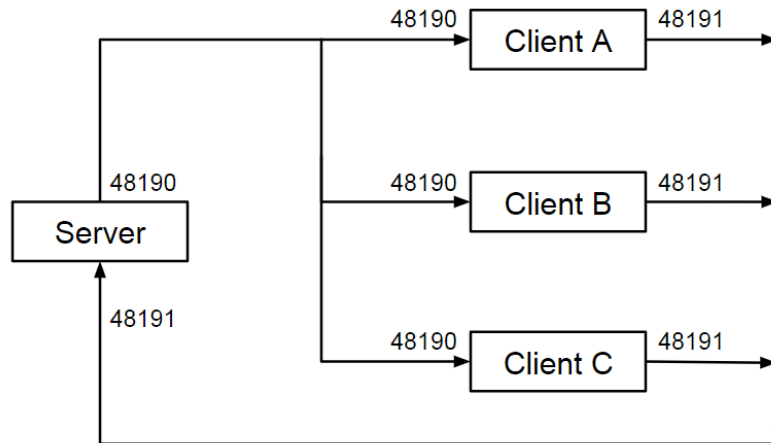


Figure 2.7: Port specification

There could be another case where a server application and a client application are run together on one host. However, a port could only deal with one process at one time. If we still utilize only two ports on this host, each port number would be used for two processes concurrently, leading to an overload. Therefore, two more port numbers need to be utilized in this case.

2.2.5 Serialization and Parsing

When packetizing data according to the package structure defined in Section 2.2.2, there is a need to write the data to a binary stream or byte array. When a package is detected and received, the actual data, such as object state and wheel information, needs to be extracted from the binary stream and used to update the status of the corresponding remote vehicle. In this section, we have a discussion on the serialization and parsing process.

It is convenient that C# language provides some useful classes for the translation between primitive types and binary streams or byte arrays, such as *BinaryWriter*, *BinaryReader* and *BitConverter*. Here we take the first two classes as an example.

As indicated in Table 2.3, a message is composed of several basic categories. Note that we do not write a package in the unit of primitive types e.g., Char, Single, Double, Int, String, etc. Instead, we define some structs, that correspond to the basic categories firstly and write the package in the unit of these structs. The conversion between a struct which is a managed type and a block of binary data which is an unmanaged type could be completed by *Marshal* class. This way, one struct could be reused several times, thus forming the binary data efficiently. Moreover, a *BinaryWriter* instance could then serialize the binary data to a stream buffer with an expandable capacity created by a *MemoryStream* instance.

Similarly, we parse the message in the unit of structs as well. A *BinaryReader* instance is used to read the received binary stream stored in a non-resizable *MemoryStream* instance block by block. A block of binary stream corresponds to a basic category. The translation from a block of binary data to a specific struct could be completed by the *Marshal* class as well. There is another method to split the whole binary stream to blocks. We could use *Buffer.BlockCopy* method to transfer a segment of data from the source to a temporary buffer and then parse it. However, in this way we need an offset pointer to find the right beginning position of each block. On the contrary, *BinaryReader* class does not need an offset pointer because it has the advantage of reading the binary stream continuously, finding out the beginning position of each block automatically.

In addition, as mentioned in Subsection 2.2.3, we could transmit multiple vehicles' status in one package as an extension since SimS protocol is flexible. To parse the extended package in this case, we could reform it to several basic packages firstly. Different information segments from the same vehicle could be stored together to a basic package according to the user ID.

2.3 Smoothing Strategy

As indicated in Subsection 2.1.3, there are two kinds of vehicle objects being rendered on a client's side. A client updates the status of its local vehicle right after sensing the request sampled from the keyboard. In this way any input could be rendered instantly so that the local vehicle moves smoothly on the screen. In contrast to this, a client could only update the status of its remote vehicles based on the received messages from the server. Note that there is a trade-off between reliability and latency for network transport layer protocol. We chose to use UDP to guarantee the real-time characteristic of the multi-user driving simulation. The obvious drawback would be no reaction on the lost or delayed messages. So a client may have no package to deal with for a remote vehicle object in an update period, which will lead to the vehicle teleporting from one point to another [21]. In this section we will introduce some local prediction strategies to mitigate this phenomenon for a better visualization performance of the simulation.

2.3.1 Extrapolation

A strategy would be to use extrapolation which means to move the remote vehicle object locally based on its current state. Extrapolation strategy assumes that the object dose not change its current route for a while, so it predicts the movements according to the object's current direction and velocity [21]. For instance, the new position calculation follows

$$P_1 = P_0 + V_0 * \Delta t, \quad (2.1)$$

where P_0 , P_1 represents the predicted position and the latest position respectively. V_0 represents the latest velocity and Δt is the time of last frame. Parameters such as latest positions and velocities for all the remote vehicle objects are stored in lists. The lists are updated each frame so that the newest parameters could be accessed anytime. If the latency is not too high, the extrapolation would accurately reproduce the object's expected movement until a new update message arrives, resulting in a smooth movement pattern [21].

The illustration of extrapolation is shown in Figure 2.8. Say client A has received packages $k - 2$, $k - 1$ consecutively. These packages are used for updating the status of remote vehicle B. Client A will receive package $k + 1$ in the future, but package k is lost when being transmitted. In the current frame n which is being rendered, client A will do extrapolation for vehicle B to make up for the contents included in package k .

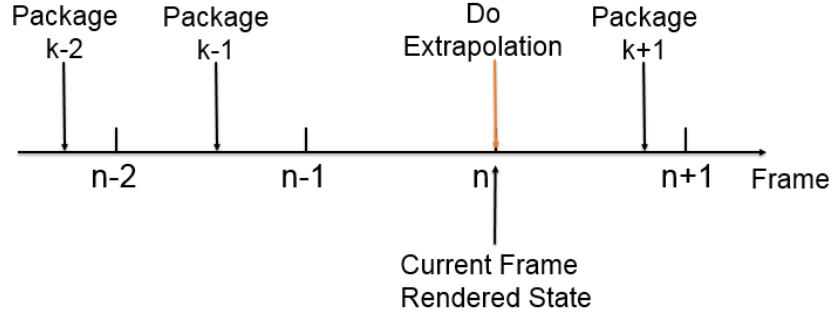


Figure 2.8: Illustration for Extrapolation

2.3.2 Interpolation

As mentioned in the subsection above, extrapolation has the limitation of assuming the object not changing its current route in a short time. Nevertheless, there might be exemptions where one find it hard to do predictions. For instance, the route of an object may change sharply, thus being non-deterministic. In this case, we do not simulate the remote vehicles forward in time. Instead, we choose to render the local vehicle in the present time while rendering the remote vehicles in the past times. The past time could refer to one or two frames before. To render the object in past times, we just need the update messages that have already been received, thus making no external predictions.

Similarly, the messages that have been received could be non-consecutive as well because of the UDP characteristic of no acknowledgment, retransmission, or timeout. For instance, client A wants to render the status of remote vehicle B in the past time. Say the current frame is n and the past time refers to two frames before, i.e., $n - 2$. Client A has received package $k - 3$ and $k - 1$, but it needs the lost package $k - 2$ for rendering the update. In this case, we could apply the strategy of interpolation which means to construct new data point within the range of known data points. In the example above, the known data points could be position points and are parsed from package $k - 3$ and $k - 1$. As a result, vehicle B will smoothly move between those points. The illustration of interpolation is shown in figure 2.9.

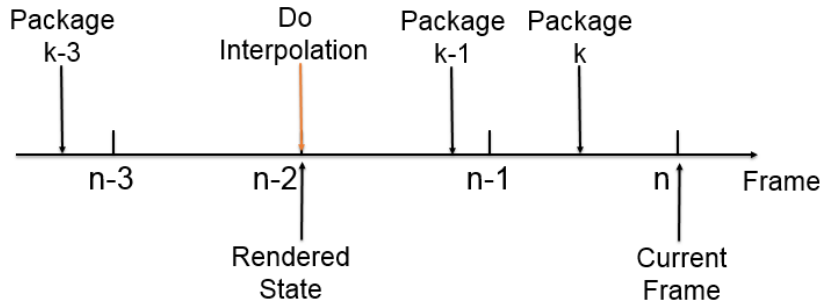


Figure 2.9: Illustration for Interpolation

Interpolating points could be done either in a linear way or a non-linear way. Having decided the initial position and the target position, the interpolated position depends on the parameter t which is clamped to the range $[0, 1]$ in the linear method. For the non-linear method, the goal is to achieve an effect of some spring-damper-like function. Parameter t in formula 2.1 is replaced by some other information here, such as the current velocity and the expected time to reach the target. Usually, the linear method is enough to move a vehicle object smoothly. The most common use for the non-linear method would be smoothing a follow camera, which is explained in the next subsection.

2.3.3 Smooth Camera Follow

As mentioned in Subsection 2.2.1, it is the camera object that captures and displays the view to each user. The most common case is that it follows its local ego-vehicle object around the play field. Therefore, the camera should frequently adjust its position according to the following object. To make the visualization better, the camera should move from the current position towards the target position smoothly. Therefore, the non-linear interpolation method could be applied here to gradually change the camera's position towards a desired goal over time.

3

Results and Analysis

In this chapter, we present the result of the multi-user driving simulation. On one hand, the simulation itself can not be presented in a static way. A live demo and some video clips are shown in the final presentation. We only enclose some screenshots in the report. On the other hand, results related to the data collection of the simulation are included. Some analysis and comparison in terms of the simulation performance according to these results are shown as well.

3.1 Simulation Results

The multi-user driving simulation is firstly conducted within Unity desktop applications and then adapted to the integration among multiple driving simulation platforms which are Unity applications and HIL dSPACE simulator together with VIRES VTD desktop application in our case. Hence, the simulation results are divided to these two parts.

3.1.1 Simulation within Unity

The simulation within Unity is conducted in two stages. The first stage is to ensure the network we built using client-server model actually works. It means that some simple messages, such as strings and the system time, could be sent from a client and received on another one successfully. This kind of simple message does not need to be parsed. The second stage is to make sure the contents included in a package are serialized and parsed correctly. The communication within the network is based on UDP and SimS protocol.

The start up of the simulation within Unity is shown in Figure 3.1. The simulated vehicle model is Volvo *S90* and the simulated road is Jolengatan in Gothenburg. One server and two clients are included in the test session. The server together with one client runs on the right laptop and the other client runs on the left laptop. Note that the server application is the small window on the right laptop and no ego-vehicle object is controlled by the server. As a result, two ego-vehicle objects should be seen from each part's view. The car in blue represents the local vehicle and the car in red represents the remote vehicle on each side. We could see clearly that the scenes and relative position of the vehicles on three applications are synchronized, which means the simulation is successful.

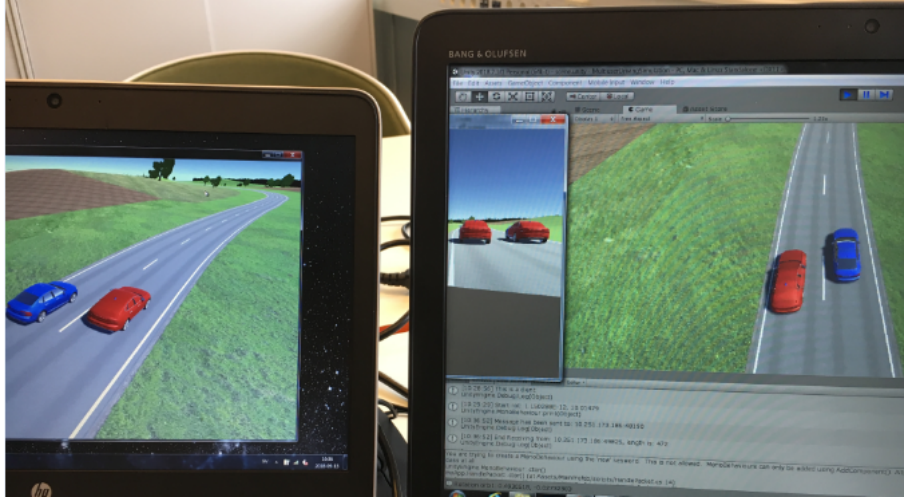


Figure 3.1: Illustration for Multi-user Driving Simulation within Unity

In the process of the simulation, a client continually renders the movements of its local vehicle according to the keyboard inputs and transmits the message of its local vehicle's newest status in the meantime. The update message should be parsed correctly on the other client's side and used for rendering to keep the synchronization of vehicles' status. We would like to show the comparison of the parsed information and the transmitted information here. We take the position information as an example. Say vehicle A is the local vehicle of client A and the remote vehicle of client B. Figure 3.2 shows the trajectories of vehicle A rendered on the two clients. In Unity world, y axis points upward; z axis points forward; and x axis points to the right direction. We ignore the trajectories on y dimension and only present them on the z-x plane. Ideally, the two trajectories should be exactly the same if the serialization and parsing processes are done in a correct way.

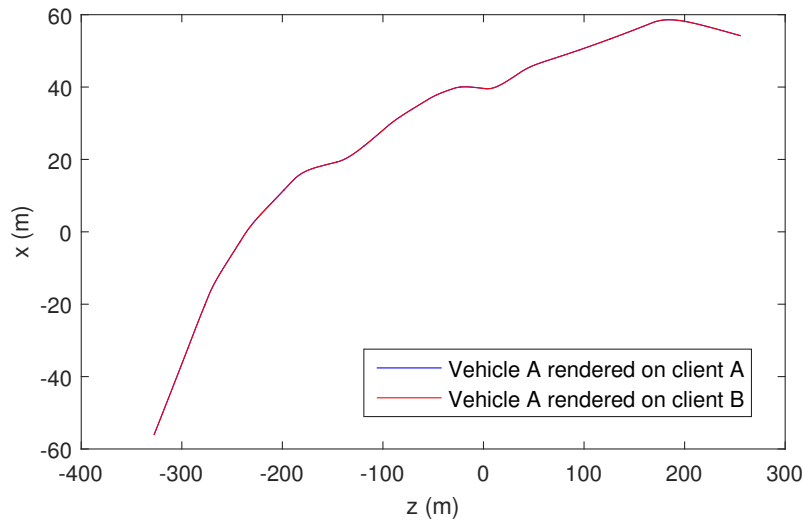


Figure 3.2: Vehicle A's Trajectories on the Two Clients

According to Figure 3.2, we see these two trajectories completely overlap with each other, which means the serialization and parsing processes are implemented correctly.

3.1.2 Simulation on Multiple Platforms

In this section, the multi-user driving simulation is done cross-platform. Here we drive the vehicles under a real-world scenario not only to make the simulation more real but to test the functionalities of autonomous car. The illustration of the chosen scenario called "extended cut-in scenario" is shown in Figure 3.3.

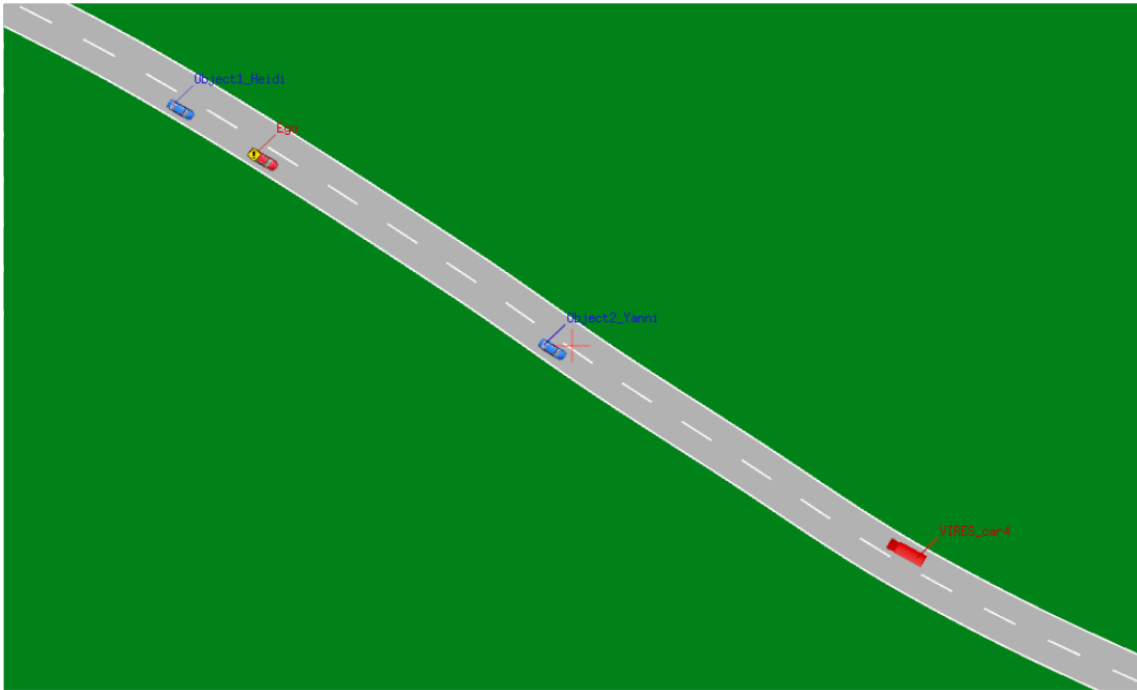


Figure 3.3: Illustration of Extended Cut-in Scenario

According to Figure 3.3, four vehicles in total are included in the set-up of this scenario. The vehicle called *Ego* is controlled by HIL dSPACE simulator and rendered in VIRES VTD application, serving as an autonomous car equipped with the adaptive cruise control (ACC) function which aims to maintain a safe distance from the vehicles ahead by automatically adjusting the ego-vehicle's speed [20]. The *VIREScar4* vehicle is a dummy vehicle generated by VIRES server, serving as a trigger for the scenario. These two vehicles in red are the external ones for Unity platform. The other two vehicles in blue called *Object1_Heidi* and *Object2_Yanni* are Unity internal vehicles coming from two Unity clients respectively. The initial relative positions of these four vehicles are shown as Figure 3.3.

At the beginning of this scenario, *Object1_Heidi* follows *Ego* smoothly and *Ego* follows *Object2_Yanni* on the same lane. At this time, *Ego* locks at *Object2_Yanni*

as the target for the ACC function. Meanwhile, the dummy vehicle *VIRES_car4* comes from the opposite direction on the adjacent lane. When *VIRES_car4* passes by *Ego*, the two Unity internal vehicles are triggered to speed up instantly. *Object2_Yanni* is supposed to go far away finally and *Object1_Heidi* overtakes *Ego*, called a cut-in. During this process, *Ego* should re-lock at *Object1_Heidi* as the target so that it could automatically speed down to maintain a safe distance when *Object1_Heidi* brakes.

As indicated in the above sections, it is the VIRES server that talks to the Unity server directly. Besides the management on the internal messages, they both are responsible for receiving the external messages and multicasting them to the internal clients as well. A screenshot of the video clip recording the cross-platform multi-user driving simulation under the extended cut-in scenario is shown in the Figure 3.4. The screenshot catches the moment when *Object1_Heidi* was going to make a cut-in. *VIRES_car4* had already passed by all the other three vehicles at this time, so it was no longer in the view of the others. The laptop with label 1 of the graph runs as the VIRES server. The one with label 2 runs as a Unity client while running the Unity server in the background and the one with label 3 runs as the other Unity client. Being the same as the last subsection, the vehicle in blue represents the local vehicle and the vehicle in red represents the remote vehicle on each side. Note that the view in the laptop with label 3 is some kind of like a bird view. We could see clearly that the scenes and relative positions of all vehicles on three screens are synchronized, which means the simulation is successful.



Figure 3.4: The synchronization among different simulation platforms

3.2 Performance Comparison

3.2.1 Effects of Extrapolations

As mentioned in Subsection 2.3.1, the extrapolation strategy could be applied to smoothen the movements of vehicle objects and get a better visualization effect. We do not apply the interpolation strategy because we consider the movements of the vehicle objects in this project predictable. The effect of extrapolation could be quantified as shown in Figure 3.5. Say vehicle A is a remote vehicle on client B. The two subplots illustrate the simulated trajectories of vehicle A along x axis and z axis respectively with time going on. The red curves represent the case of using extrapolation while the blue curves represent the case of not using extrapolation.

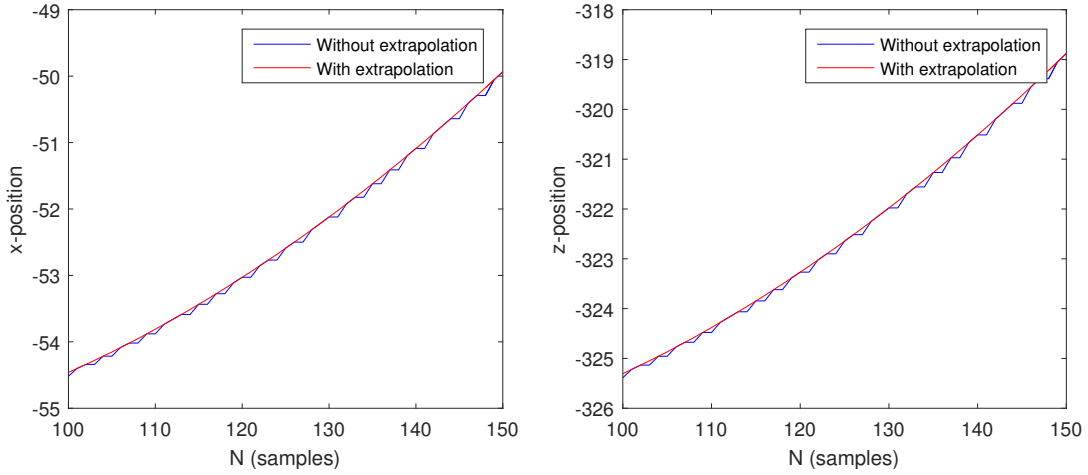


Figure 3.5: Illustration of Extrapolation Effects

Comparing the two curves in each subplot of Figure 3.5, we could see that the blue curves have fluctuations due to the fact that some packages are delayed or lost when being transmitted via UDP. Without extrapolation leads to the remote vehicle object staying at the last position in these frames. In contrast, the red curves are much smoother because extrapolation could compensate the drawback of UDP on data reliability.

3.2.2 WiFi and Ethernet

The reactions of the system should be in real time, which means the case with a lower latency is a better choice. As mentioned in Subsection 2.1.2, the simulation could be conducted via the connection of either WiFi or Ethernet in laboratories under LAN. We need to make a comparison between these two connections to find a better choice. The average latency could be reflected on the rate of delayed or lost package. A comparison on this rate under two types of connections is presented in Figure 3.6. The tests on each case are conducted 20 times randomly during several

working days at different laboratory places and 2000 - 4000 packages are transmitted each time so that we can collect a large amount of data and get the average behaviour. Although the specific collected data presented below is not repeatable, the conclusion according to the average performance is generic.

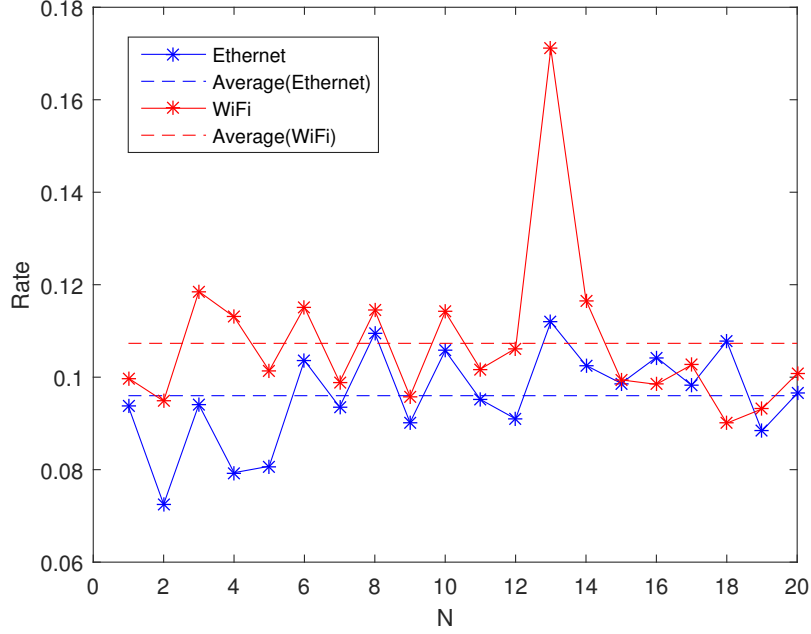


Figure 3.6: Illustration of the Rate of Delayed or Lost Package under WiFi and Ethernet Connections

As shown in Figure 3.6, the red line represents the case of using WiFi connection and the blue line represents the case of using Ethernet connection. The rate is calculated according to the equation below.

$$Error\ rate = \frac{Number\ of\ delayed\ or\ lost\ packages}{Number\ of\ total\ transmitted\ packages} \quad (3.1)$$

The average behaviour could be reflected on the dashed lines in Figure 3.6 and Table 3.1 which records the summation of 20 times' tests.

Connections	Total Transmitted Packages	Total Delayed or Lost Packages	Rate
Ethernet	59386	5701	9.60 %
WiFi	65532	7033	10.73 %

Table 3.1: Comparison on Average Behaviour between WiFi and Ethernet

From Figure 3.6, we could see that the rate of delayed or lost package in the simulation under Ethernet connection is lower than that under WiFi connection most

of the time. According to the average behaviour, we find that Ethernet is a better choice with low latency for the simulation.

4

Closure

4.1 Conclusions

In the thesis project, we find that Unity as a game engine is proper to develop driving simulations because it is capable of rendering 3D scenes with a high quality and is highly flexible for developers providing both high level and low level APIs.

Some ideas of the multi-user driving are inspired from game industry. However the differences between them are presented as well, such as the authority of the server and the communication platforms. Therefore we choose to implement the network from scratch and use low level APIs rather than high level APIs provided by Unity on communication.

To achieve real-time communication over the network, we find it proper to transmit messages via UDP under a modified client-server framework. Asynchronous methods shall be applied to process multiple tasks concurrently and ring buffers could make the simulation more stable. Under laboratory environment, we find that Ethernet connection is a better choice than WiFi in terms of reducing latency.

SimS protocol regulates communication rules as well as message formatting. Some improvements could be done upon the headers to reduce communication bandwidth if large amount of vehicle objects are included in a test session.

To achieve a good visualization effect on object synchronization, some smoothing strategies have been applied depending on the possibility of making correct predictions. However there were still some twitches in the simulation sometimes. Further improvements could be on time synchronization which are mentioned in the next subsection.

4.2 Future Work

As mentioned in the conclusions, there are still some twitches in the driving simulation sometimes. It could be caused by hardware issues regarding to operating speed. It could also be caused by the inconsistency between camera following and the rendering on remote vehicle's transform. More exploration on the event functions such as *Update* and *FixedUpdate* provided by Unity should be made in this

case. Another possibility would be the unavoidable latency on the communication. To reduce this kind of effect, network time protocol (NTP) for time synchronization should be applied. It aims to limit the time difference among multiple simulators within a few milliseconds.

Another aspect is that the open simulation interface (OSI) could be applied in the future. It aims for easier compatibility of virtual testing for automated driving functions.

Bibliography

- [1] Wang, Jianqiang, Wu, Jian, et al., "Driving safety field theory modeling and its application in pre-collision warning system", *Transportation research part C: emerging technologies*, Vol. 72, pp. 306–324, Nov. 2016.
- [2] Eichberger, Arno, et al., "A situation based method to adapt the vehicle restraint system in frontal crashes to the accident scenario", *Proceedings of the 21st ESV Conference*, June 2009.
- [3] Okita A, "Learning C programming with Unity 3D", AK Peters/CRC Press, Aug. 2014.
- [4] Sanders A, "An Introduction to Unreal Engine 4", AK Peters/CRC Press, pp. 1-14, Oct. 2016.
- [5] Pachoulakis, Ioannis and Pontikakis, Georgios, "Combining Features of the Unreal and Unity Game Engines to Hone Development Skills", *arXiv preprint arXiv:1511.03640*, Nov. 2015.
- [6] Mayden, A., "Unreal Engine 4 vs. Unity: Which Game Engine Is Best for You", *Preuzeto*, Vol. 5, no. 7, pp. 2017, Oct. 2014.
- [7] Randolph, Nick, et al., "Professional visual studio 2010", John Wiley Sons, July 2010.
- [8] Microsoft, "Get started with Visual Studio 2017", *visualstudio.com*, [Online]. Available: <https://tutorials.visualstudio.com/vs-get-started/intro>. [Accessed: Jan.2018].
- [9] Buford, John and Yu, Heather and Lua, Eng Keong, "P2P networking and applications", Morgan Kaufmann, Mar. 2009.
- [10] Berson, Alex, "Client/server architecture", No. IEEE-802, McGraw-Hill, 1992.
- [11] Fiedler, Glenn, "What every programmer needs to know about game networking." *Glenn Fiedler's Game Development Articles and Tutorials*, Vol. 24, 2010. [Online]. Available: <https://gafferongames.com/post/what-every-programmer-needs-to-know-about-game-networking/>. [Accessed: Mar. 5, 2018].
- [12] Posey, Brien, "Understanding the Differences between Client/server and Peer-to-peer network", 2007. [Online]. Available: <http://www.techrepublic.com/article/understanding-the-differences-between-clientserver-and-peer-to-peer-networks/1055415>. [Accessed: Jan. 12, 2018].
- [13] Zimmermann, Hubert, "OSI reference model–The ISO model of architecture for open systems interconnection", *IEEE Transactions on communications*, Vol. 28, no. 4, pp. 425-432, Apr. 1980.
- [14] Mackay, Steve, Edwin Wright, and John Park, "Practical Data Communications for Instrumentation and Control", Elsevier, pp. 199-204, Jun. 2003.

- [15] Braden, Robert, "Requirements for Internet hosts-communication layers", No. RFC 1122. pp. 8-10, 1989.
- [16] Raychaudhuri, Dipankar, et al., "Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols." Wireless Communications and Networking Conference, 2005 IEEE. Vol. 3, pp. 1664-1669, IEEE, 2005.
- [17] Unity Technologies, "Execution Order of Event Functions". [Online]. Available: <https://docs.unity3d.com/Manual/ExecutionOrder.html>. [Accessed: Feb. 18, 2018].
- [18] Programmr, "Difference between asynchronous and non blocking", Jun. 2016. [Online]. Available: <http://www.programmr.com/blogs/difference-between-asynchronous-and-non-blocking>. [Accessed: Mar. 17, 2018].
- [19] Vires Simulationstechnologie GmbH, "Virtual Test Drive". [Online]. Available: <http://vires.com/products.html>. [Accessed: Mar. 5, 2018].
- [20] Winner, Hermann, and Michael Schopper, "Adaptive cruise control", Handbook of Driver Assistance Systems: Basic Information, Components and Systems for Active Safety and Comfort, pp. 1-44, Aug. 2014.
- [21] Bevilacqua, F., "Building a Peer-to-Peer Multi-player Networked Game" Aug. 2013. [Online]. Available: <https://gamedevelopment.tutsplus.com/tutorials/building-a-peer-to-peer-multiplayer-networked-game-gamedev-10074>. [Accessed: Jan. 12, 2018].