A Framework for Distributed Simulation of Intelligent Transportation Systems

Amrish Arunachalam Kulasekaran

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (FNS)

Supervisor: Professor Vinny Cahill

Septemper 2020

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Amrish Arunachalam Kulasekaran

September 14, 2020

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Amrish Arunachalam Kulasekaran

September 14, 2020

Acknowledgments

I would like to thank Professor Vinny Cahill for his support and guidance throughout this dissertation. The regular meetings, consistent advice helped me to stay on the right course.

I would also like to thank the IT team of the School of Computer Science and Statistics for their help to set up the remote server for this dissertation.

Finally, I would like to say a heartfelt thanks to my family and friends, especially my mom and dad, for backing me every step of the way. I am eternally grateful for every opportunity you have afforded me in life.

Amrish Arunachalam Kulasekaran

University of Dublin, Trinity College
Septemper 2020

A Framework for Distributed Simulation of Intelligent Transportation Systems

Amrish Arunachalam Kulasekaran, Master of Science in Computer Science

University of Dublin, Trinity College, 2020

Supervisor: Professor Vinny Cahill

Around the world, lots of countries and people face economical, social and environmental impacts due to traffic congestion. The main reason for the congestion is the increase in traffic demand and over traffic flow in a road network. An effective and efficient solution against congestion is to utilize the road infrastructure to its full capacity. Intelligent Transportation helps (ITS) us to address congestion by continuously monitoring and regulating the traffic flow. It achieves this by establishing cooperation and coordination among the vehicles. These systems must be tested and evaluated in a realistic simulation environment before real-world deployment.

The ITS simulation framework current present is purely based on vehicular network simulation, and they lack to address distributed and real-time characteristics of ITS. This dissertation introduces a framework consisting of only Carla simulator to perform distributed ITS simulation. The feasibility of the proposed approach is evaluated by simulating customised ITS application created by extending the designed framework.

Contents

List of Figures

Chapter 1

Introduction

The dissertation presents a simulation framework for Intelligent Transportation System (ITS). The novelty of the work is the framework distributes the computation of the actor's (vehicle and roadside infrastructure)mobility and behaviour models (i.e., the actors involved in the simulation are distributed over the network) and establishes communication via a socket. Carla [5], an autonomous driving simulator, serves as a centralized simulation environment, and the actors are spawned in the simulation as clients using the Application Programming Interface (API) provided by the framework. Carla simulates the vehicle's movement based on the vehicle control provided by its mobility model. The distributed environment created by the framework looks similar to figure 1.1.

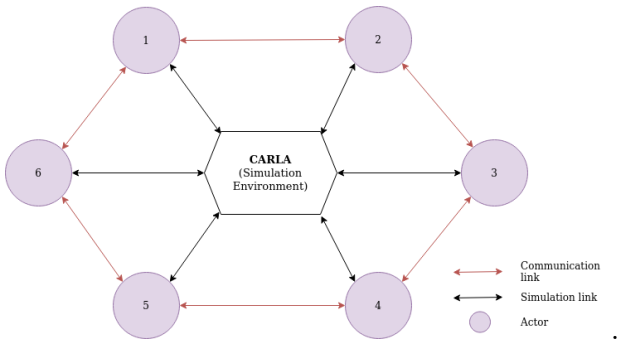


Figure 1.1: Simulation Environment

1.1 Motivation

Transportation is essential for our economy and society. A country's economic, social, and political life depends upon an effective transport system. Transport is a cornerstone of European integration and crucial for the free flow of citizens, services, and goods to be served. It is also a significant contributor to the economy, accounting for about 664 billion Euros in Gross Value Added in 2017 (i.e. 5% of the total EU's GVA) and employed over 11 million people [6].

According to the "Statistical pocketbook 2019" of European Commission [6], 50.1% of total freight transportation were made via roadways. Nearly 73% of passengers made their journey through road and among them, almost 90% of them made the road trip by car. These statistics help us to understand the importance of roadways. The increase in traffic volume creates demand and eventually leads to traffic congestion across the EU. Due to traffic congestion, a driver spends approximately 28 hours in traffic every year. Additionally, it also creates environmental and economical impacts.

Traffic congestion usually occurs when traffic flow exceeds the road capacity. Many factors can cause traffic congestion, but infrastructure stands at the top. A simple traditional solution to reduce congestion is expanding the infrastructure. However, this is not an effective or economically reasonable solution because one cannot keep on extending as demand increases. Preferably, utilizing the infrastructure capacity to the fullest is more efficient and rational.

ITS helps us to address congestion by continuously monitoring and regulating the traffic flow. For instance, consider a junction with traffic signals that have fixed time to switch signal. They are only effective for the ideal traffic flow, when the traffic flow decreases the driver has to wait till green and the traffic signal creates congestion as the traffic flow increases. On the other hand, ITS can help the traffic signals dynamically adjust the switching time depending on the traffic flow.

ITS components are installed in the vehicles and along the roadside to collect traffic data, guide and control the flow. Real-world traffic scenarios are complex, highly dynamic. Because of these reasons, it must be tested and evaluated before real-world deployment. Else, they may lead to a serious problem instead of avoiding congestion.

Including a simulator in the development cycle of ITS will help us to evaluate the system in all possible scenarios by testing it in the simulation environment mirroring real-world ITS. Using simulators to test and evaluate ITS is cost-effective and risk-free compared to build a real-world test site.

1.2 Project Overview

1.2.1 Intelligent Transportation System

A set of different applications that use state-of-the-art technologies to monitor and regulate the traffic flow are known as ITS. Throughout the world, many countries have designed and developed ITS, and they have unique visions and goals to improve infrastructure applying ITS. Despite counties having different perspective toward ITS, they share the common goal to improve the current transportation infrastructure. This dissertation follows the standards and architecture of ITS proposed by the European Union [7].

1.2.2 Approach

To simulate the ITS, we need a road network, traffic infrastructure, mobility model to control vehicle behaviour, traffic demand to generate traffic flow and network model to create a communication infrastructure.

Traffic simulators like SUMO [8], PVT-Vissim [9], etc. can create road infrastructure and simulate traffic flow based on demand and vehicle behaviour, but they do not support communication between vehicles. On the other hand, network simulators like ns3 [10], OMNET++ [11], etc. can create network infrastructure and simulated realistic communication, but they do not support mobility. Therefore, traffic simulators and network simulator should be coupled through a TCP link to create an ITS simulation environment.

The above-mentioned simulators have to be extended for developing a customised TMS like Cooperative Adaptive Cruise Control (CACC) [12], platooning, slot-based driving [13], etc. For example, PLEXE [14] implemented a platooning application by extending the Viens[3] framework (Viens is a vehicular network simulation framework created by

coupling SUMO and OMNET++). The framework to simulate the ITS environment for developing TMS will be similar to the figure.

This framework performs space-continuous and time-discrete simulation (i.e. it updates the simulation after every time step). After each time step, the traffic simulator computes the behaviour of all vehicles based on the network simulation results to update the vehicle states and generate movement trace. Then, it sends the movement trace to network simulator. The network simulator updates the position of all nodes based on the received movement trace, simulates the communication and sends the result back to the traffic simulator. The sequence diagram in the figure illustrates this process. The approach lacks implementing the distributive nature of ITS and can only perform time-discrete simulation, but in real scenario time is continuous.

This dissertation introduces a framework that uses only Carla, autonomous driving simulator for simulating ITS. The framework is created by extending the Client API of Carla. Based on the traffic demand, vehicles are added as Carla client using the framework. The distributed network formed by the clients creates the communication infrastructure.

In the proposed approach, peer to peer socket communication is used to exchange messages instead of simulating it. Each client node is responsible for its behaviour computation based on the messages it receives. This framework is capable of performing a time-continuous simulation with the help of Carla Server's asynchronous mode. Thus, it can be used to create a more realistic ITS simulation.

1.2.3 Research Aims

The research intends to design and develop a simulation framework for ITS, that creates a realistic and distributed simulation environment. The main motive is to help developers to concentrate on developing and evaluating traffic management system (TMS) models. Apart from the above, the specific objectives of this dissertation is to evaluate the feasibility of the proposed approach by creating customised scenarios using the designed framework

1.2.4 Project Scope

The dissertation focuses on designing a approach to developing a simulation framework using only an autonomous driving simulator to create an environment closely resembling real-world ITS. Especially focuses on simulating and evaluating an actor's behaviour while developing new traffic management systems in a distributed environment.

The framework distributes computation of the actor's behaviour among several nodes in a network and exchanges information between them through socket communication. This dissertation is not concerned about implementing the network infrastructure or underlying protocols involved in vehicular communication.

1.2.5 Benefits of this Research

The benefit of this research is to improve the testing and evaluation processes involved while developing TMS and ITS.

1.2.6 Road Map

The rest of this dissertation is divided into four parts, starting with chapter 2 that provides the necessary background information for this work and the state-of-the-art simulation framework for testing ITS. Chapter 3 is about the design of the framework and decisions that lead to the development of the proposed approach. The following chapter explains the architecture of the framework and APIs provided for creating custom scenarios using the framework. Chapter 5 will discuss the evaluation of the framework. Finally, the dissertation concludes with an outline stating the pros and cons of this work and future work to enhance the framework.

Chapter 2

Background

This chapter discusses the required background information needed for a better understanding of the proposed work. In the first section explains the overall architecture of ITS and their components. The second section discusses the current state-of-the-art techniques used to perform ITS simulation. At last, an overview of the tools used to build the framework is given.

2.1 Intelligent Transportation System

The Intelligent Transportation Systems (ITS) comprises of different applications that improve the safety, security and efficiency of transportation and provide a better driving experience. ITS achieves this by gathering necessary data from the sensors or devices placed in the infrastructure or vehicles. Initially, ITS only provided intelligence to vehicles and roadside infrastructure, it was not able to share information between them and make them cooperate to regulate the traffic. The European Commission introduced the Cooperative Intelligent Transportation System as a new category of ITS [15]. In which drivers, vehicles, passengers or road operators can directly interact with each other or with the surrounding infrastructure and figure 2.1 illustrates a typical CITS environment. CITS takes advantage of the communication and cooperation between different actors and effectively regulate the traffic flow by exchanging information about traffic jams, congestion, accidents.

Figure 2.1: C-ITS Environment [1]

2.1.1 Architecture

Intelligent Transportation System consists of four sub-systems or stations [7], that helps transportation infrastructure to gain intelligence and real-world information . All the Sub-Systems are made up of ITS Station (ITS-S), it may contain a single ITS-S or network of ITS-S.

ITS Station Reference Architecture

The ITS Station is the base component upon which a sub-system is built and its reference architecture helps to understand the functionality of ITS Station. Its architecture follows the principles of the OSI model[16] for the layered communication protocol, and it is extended to include different ITS applications. In the figure 2.2,

- Access layer is responsible for the functionality of Physical and Data Link layers
- Network and Transport layer is responsible for the functionality of Network and Transport layers
- Facilities layer is responsible for the functionality of Session, Presentation and Application layers

- Application layer
- Management layer is responsible for managing the communication with the ITS Station
- Security layer is responsible for providing security services to ITS Station and it is also considered as a part of management entity.

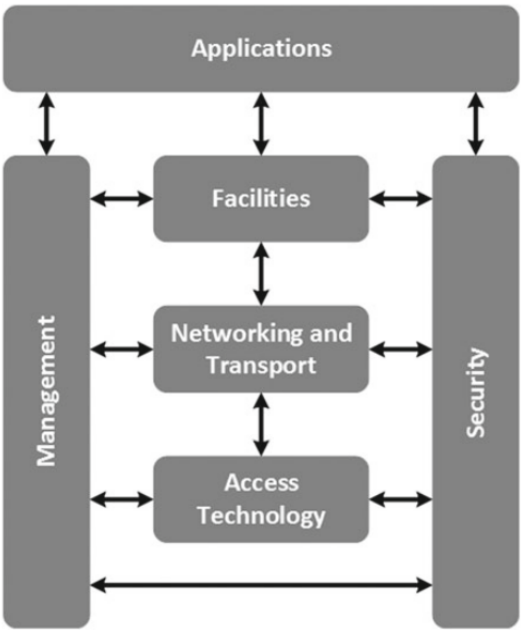


Figure 2.2: ITS Station Reference Architecture

Functional Components of ITS Station

Based on their functionality, ITS Station can be divided into four sub-clauses [7]

- **Host :** It helps to run the ITS application
- **Gateway :** It helps to connect a ITS Station with the external propriety network. Facilities layer helps to connect different protocols.
- **Router :** It helps to connect one ITS Station with another. Network and transport layer helps to connect different ITS-S.
- **Border Router :** It is similar to router but it can also connect ITS-S with external network.

ITS Sub-Systems

- **Personal Sub-Systems/Station** help us to access ITS applications or platform through smartphones, Human Machine Interface or similar devices. They can be used as a user interface for crowdsourcing, getting feedbacks or as an interface for other sub-systems or station. Its internal network contains only ITS-S host.
- **Roadside Sub-Systems/Station** are placed along the roadside to collect real-time traffic (e.g. traffic flow, volume, etc.) and environment (e.g. temperature, wind speed, humidity, etc.) information, acts as a gateway or a router for other ITS Sub-Systems and controls equipments like a traffic signal, electronic sign-boards, etc to provide assistance to the drivers. Its internal network contains ITS-S host, gateway, router and border router.
- **Vehicle Sub-Systems or Station** are installed in the vehicles and connected to their proprietary vehicular network (e.g. CAN bus). They accumulate information about the vehicle, their surrounding and drivers trip details. It can be used to control the vehicle during an emergency situation. They provide the collected information to the driver and other Sub-Systems. Its internal network contains ITS-S host, gateway and router.
- **Central Sub-Systems or Station** are used to monitor the other Sub-System.

2.1.2 Vehicular Communication

As mentioned above, vehicular communication plays a vital role in ITS to establish co-operativeness and sharing information. It is implemented using communication technology based on WAVE/IEEE 802.11p [17]. Depending on the Subsystem a vehicle communicates with the modes are divided into three:

- **Vehicle-to-Vehicle (V2V):** In this mode it communicates with another vehicle.
- **Vehicle-to-Roadside Infrastructure (V2R):** In this mode it communicates with the roadside infrastructure.
- **Vehicle-to-Everything (V2X):** In this mode it can communicate with device in the traffic infrastructure.

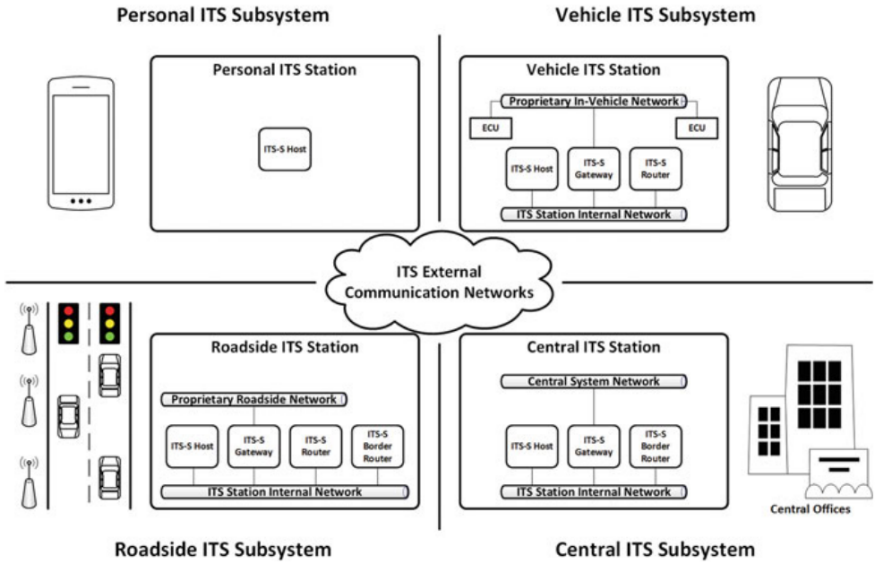


Figure 2.3: European ITS communication sub-systems

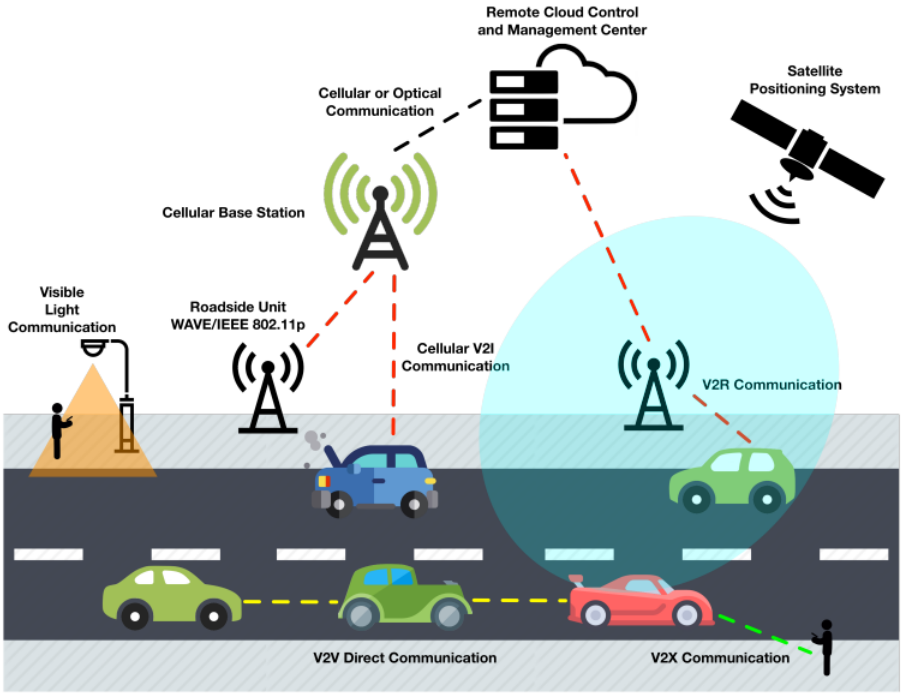


Figure 2.4: Vehicular Communication Modes [2]

2.1.3 Cooperative Awareness Message (CAM)

Cooperative Awareness Message (CAM) [18] is a part of the basic cooperative services that are available in all ITS Stations. CAM is similar to a beacon that is exchanged between an ITS-S and its neighbours to make them aware of their surrounding environment and to enable cooperativness between them. CAM is generated and transmitted periodically but the frequecy is determined by the generating station.

The content provided by a CAM may vary depending on ITS-S type by all the messages contains the reference position, status and type of the station generating the message. CAM is only transmitted to the stations that are within the communication range of the orginating station and they are not forwarded to other stations once received.

2.1.4 Decentralized Environmental Notification Message (DENM)

Decentralized Environmental Notification Message (DENM) [1] is a part of basic cooperative service to support the Road Hazard Warning (RHW) application and available in all the ITS stations. DENM is used to notify and warn ITS Stations and users about events that are hazardous to the traffic flow (i.e. vehicle breakdown, accident, lane status, etc..). ITS application triggers and generates DENM upon the occurrence of an event. DENM transmission is repeated and lasts until the triggered event is resolved. A DENM broadcast is terminated once the predefined time limit set by CA service expires or by the ITS application that triggered the DENM.

The DENM contains information about the event that triggered it along with the type and location of the event. DENM are broadcasted through vehicle-to-vehicle (V2V) and vehicle-to-infrastructure communication. Once received, a station forwards the DENM to other stations and only shows the relevant notifications to the driver.

2.2 Current State of the Art in ITS Simulation

Veins is a popular vehicular network simulation framework researchers use for ITS simulation and evaluation. In this paper [19], they created an C-ITS applications by extending it to evaluate the application's Quality of Service (QoS). An overview

of Viens and its simulation technique is briefly described in the next section before discussing the paper.

2.2.1 Veins

Viens is a simulation framework that performs vehicular communication simulation. It uses OMNET++, discrete-event network simulator and SUMO, discrete-time traffic simulator and binds them using Traffic Control Interface (TraCI) a protocol based on TCP socket. Veins simulate both OMNET++ and SUMO parallelly, and TraCI creates a bidirectional-coupling and make the simulators interact with each other.

SUMO simulate traffic based on the demand and generates movement trace and sends it to OMNET++. Then, the network simulator updates all the nodes based on the movement trace simulates communication and sends back the simulation result to SUMO.

Viens implemented vehicle and roadside ITS subsystem as network entities to perform vehicular communication. Each entity contains three components and they are extended to create new ITS application, the components are

- **Network Interface Card (NIC):** It implements protocols and stack required vehicular communication
- **Mobility:** It is responsible for the mobility of the network entity, it configures the position based on the movement trace for a vehicle or a static position for RSU
- **Application:** It implements application, that generates and shares message based traffic information to perform a basic simulation.

Bidirectionally Coupled Simulation

Viens extended both the simulators and added a communication model to exchange and buffer commands and messages (e.g., simulation results, movement trace).

Since OMNET++ is a discrete-event simulator, it should be triggered to perform the simulation. The framework schedules a trigger at a regular interval to update the node movement and perform the simulation. It follows the same trigger approach for SUMO,

as it is a discrete-time simulator the approach suits well. The communication module buffers any messages received by the simulators until the next trigger.

At each time step, OMNET++ sends the simulation result as commands and trigger to SUMO, then SUMO performs traffic simulation. After the simulation, SUMO sends the generated movement trace back to OMNET++ and waits for the next trigger. This way OMNET++ updates the position of all the nodes based on the movement trace received from SUMO and performs simulation until the next trigger. During the simulation, OMNET++ makes its nodes interact with inter-vehicle communication and reassign their attributes (e.g., speed, route). In this process, SUMO computes the behaviour of each vehicle in the traffic based on the attributes (e.g., speed, route) of the corresponding nodes. Figures 2.5 and 2.6 illustrates the entire process.

2.2.2 Extending Veins for ITS Application Simulation

Veins is extended to implement multiple ITS application to evaluate the QoS of the message broadcasting model. The applications implemented are:

- Hazard warning (HW), an event-driven road Safety application
- Dynamic speed limit (DSL), a traffic management application

First, they create the Traffic Manager Control (TMC) application entity class, it is inherited from BaseWAVEApplLayer class. Then they override the custom application component of Veins RSU with TMC. Now, Veins RSU is customised to send HW and DSL messages. Likewise, the application component of the vehicle entity is overridden with CarApps class. Now, Venis is fully extended and ready for the ITS simulation

2.3 Tools

This section discusses the various tools used by the framework to perform the simulation. The framework uses Carla Server as a simulation engine and CarlaViz to visualize the entire simulation in a webpage.

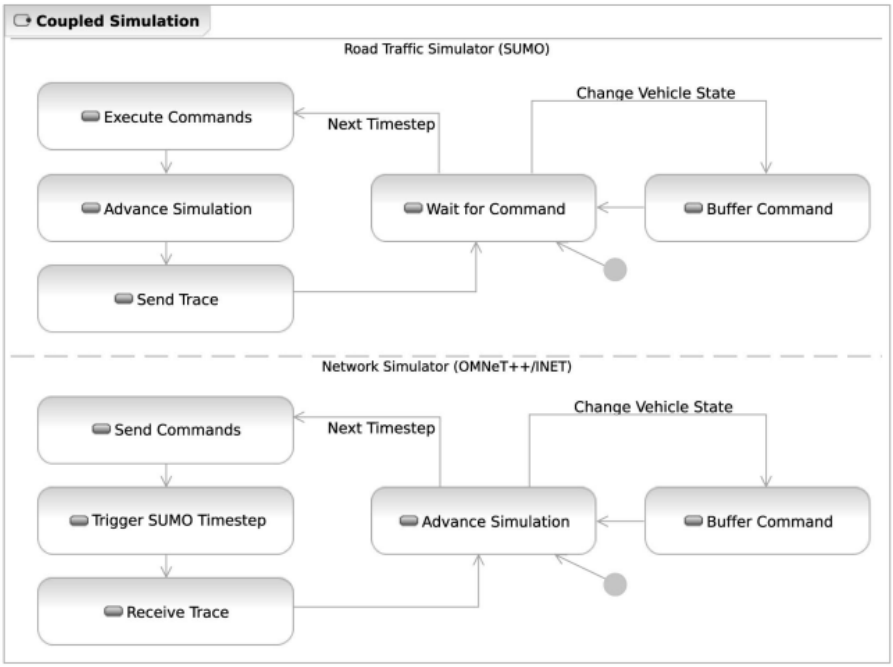


Figure 2.5: Flow Diagram of the simulation [3]

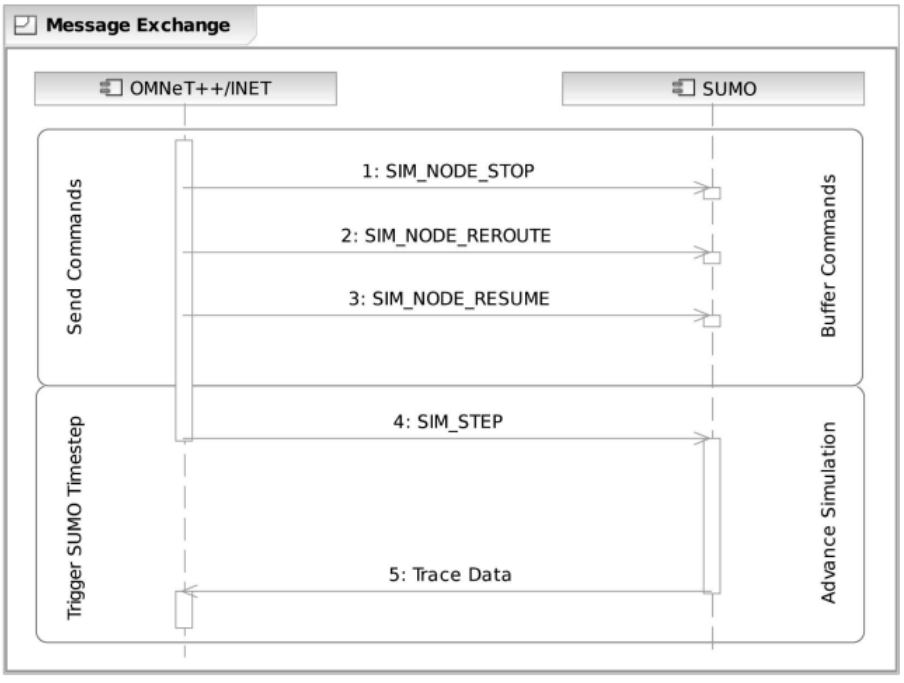


Figure 2.6: Sequence diagram of messages exchanged between simulators [3]

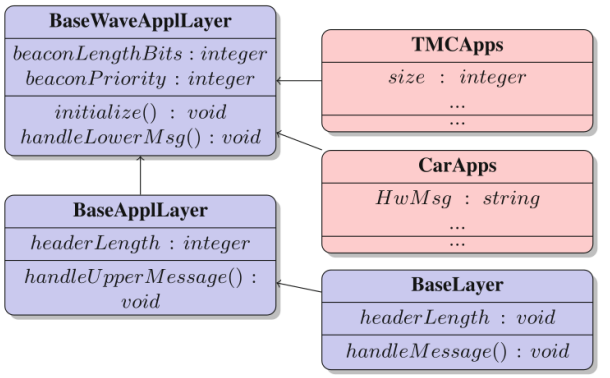


Figure 2.7: class of TMC and CarApps [3]

2.3.1 CARLA

"CARLA is an open-source autonomous driving simulator" [4]. It provides a set of API to perform autonomous driving simulation and build as a tool to help researchers and companies to "democratize autonomous driving RD" [4]. It used the Unreal Engine to compute physics and perform simulation and follows the OpenDRIVE standard to design road network and related settings. The simulation performed by the Unreal Engine is controlled by its API provided by Carla in Python and C++. Apart from autonomous driving RD, this is a great tool to simulate mobility.

Architecture

Carla is built based on a client-server architecture pattern. The server is coupled with the Unreal Engine, and it holds sole responsibility for simulation (i.e., physic computation, sensor and scene rendering are performed). The server is divided into two components based on the elements they control in the simulation. The figure 2.8 illustrates the architecture of Carla.

- **World Server:** This component is responsible for creating the simulation environment. This includes the building, road network, actors (vehicles, sensors and other static properties) etc. It is used to initiates sessions to perform the simulation. Once, initiated it sets the simulation world where actors can be placed.

- **Agent Server:** This component is responsible for the simulating actors. It provides two threads, Control Thread (It receives commands for simulation from a client) and Measurements Thread (It streams the live sensor data from the simulation).

The Carla Server only creates the platform and provides the infrastructure for performing the simulation. The behaviour of all the element is controlled by the Carla Client. A user can gain complete control over an actor, starting from creating it and controlling their behaviour throughout their lifetime.

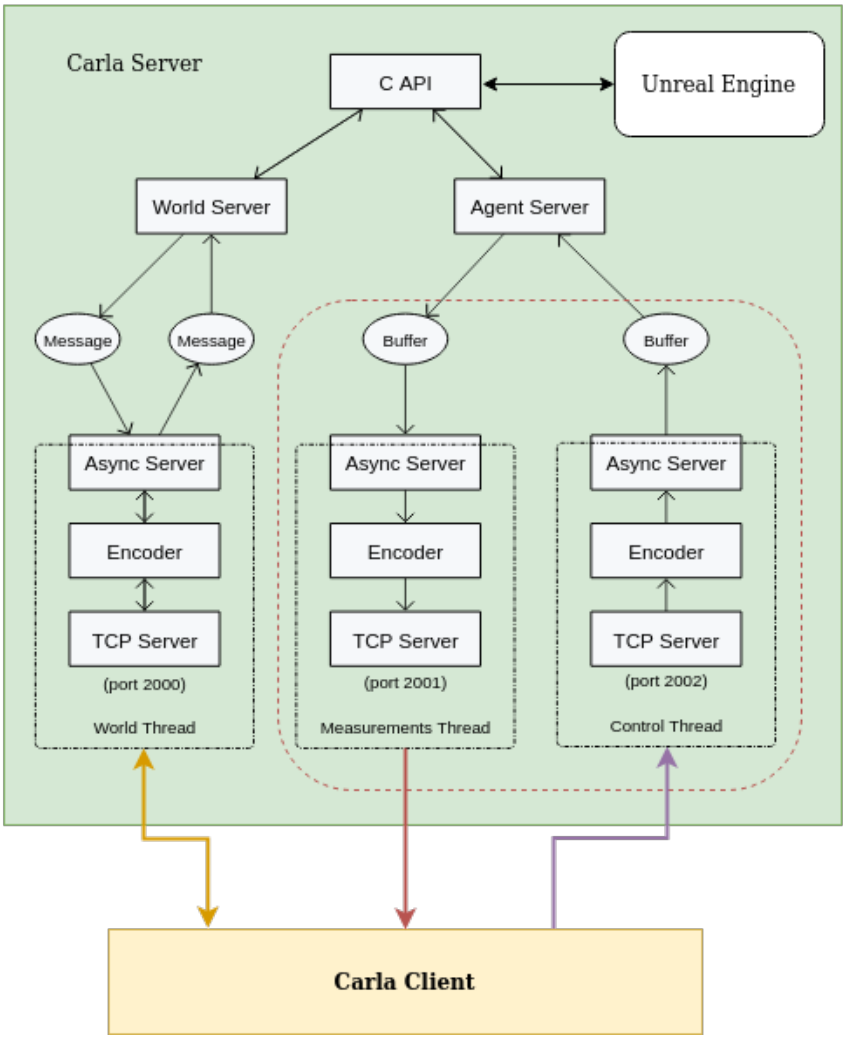


Figure 2.8: Carla Architecture [4]

Client-server synchrony

This section explains about different modes of communication between a client and the server. The server runs in Asynchronous mode by default. In this mode, it simulates as fast as possible without does not wait for any client. On the other hand, in Synchronous mode, the server waits for the client. It proceeds to simulate only if the client gives the signal to do it.

The client can also control the simulation time-step. By default, the server does not follow a fixed time gap, it is called "Variable time-step". The time gap depends on the computation and rendering of physic. The simulation time-step can be fixed by a client. Based on this the server can perform a time-continuous and time-discrete simulation. The figure 2.10 shows all the configuration modes provided by Carla.

	Fixed time-step	Variable time-step
Synchronous mode	Client is in total control over the simulation and its information.	Risk of non reliable simulations.
Asynchronous mode	Good time references for information. Server runs as fast as possible.	Non easily repeatable simulations.

.

Figure 2.9: Configuration Modes Provided by Carla [4]

2.3.2 CARLAVIZ

Carlaviz is a visualization plugin developed for Carla to view the simulation environment through a web browser. It streams the world created by the server along with the actors (i.e. Vehicles and pedestrians) living in them and updates their status on the air. Along with this, it streams data from the sensors and visualizes using tables and graphs. Added to this, Carlaviz also allows users to draw texts, points and lines on the visualization. The figure is the sample screenshot of the visualization.

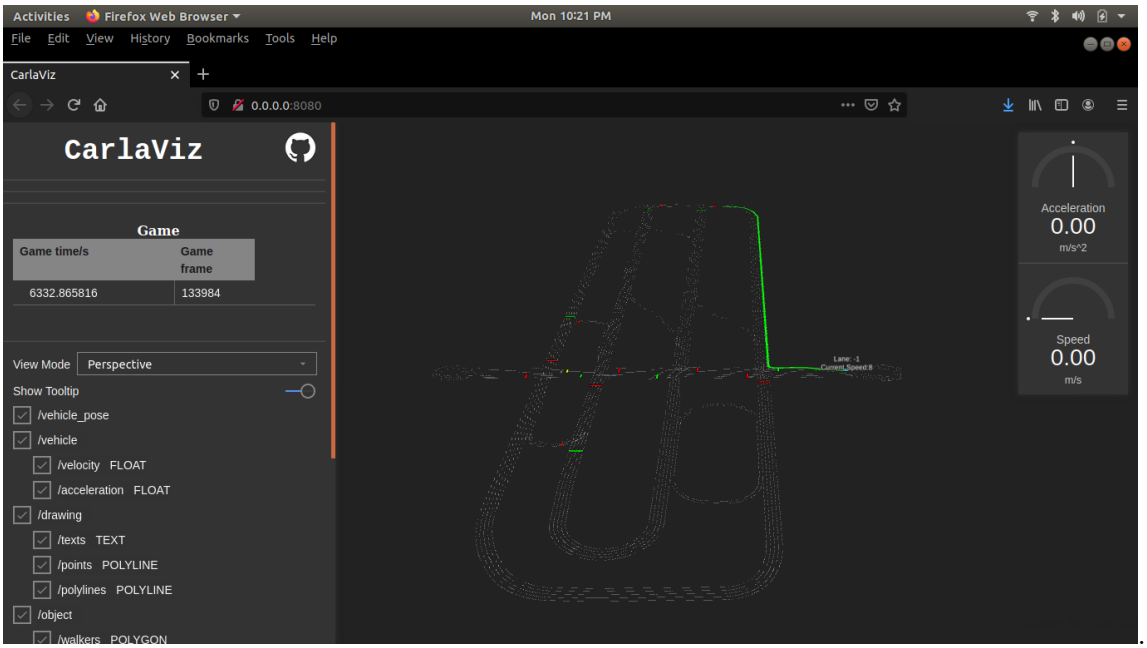


Figure 2.10: CarlaVIZ Visualisation Screenshot

Chapter 3

Design

This chapter discusses the design of the proposed approach in detail. The first section, explains the limitations in the state-of-the-art approach, and a set of requirements is extracted from that. The second section describes the overall design of the framework. The last part reviews the design.

3.1 Design Decisions

This section formulates a set of requirements for the proposed design. It is done by reviewing the state-of-the-art explained by the previous chapter and finding the limitations.

3.1.1 Limitations of previous work

In the state-of-the-art section, the ITS application is simulated by extending a framework designed to simulate a vehicular network. The ITS simulation is about simulating and evaluating the coordinative and cooperative behaviour of actors. The message sharing is needed only to establish this behaviour. The approach performs a realistic simulation from the network perspective but lacks to implement the distributive and real-time characteristics of ITS. These characteristics have a great impact on the behaviour of actors. This is followed by the limitations of this approach.

centralised behaviour computation

The traffic simulator of Veins is responsible for the computation of the behaviour of the vehicles in the simulation. Once the traffic simulation is triggered, it starts the computation loop. For each iteration, the behaviour of a vehicle is decided based on the attribute of the corresponding node in the network simulator. The traffic simulator updates the simulation after the loop is complete.

In reality, all the vehicles are independent and responsible for computing their behaviour based on the message it receives. It changes the trajectory as soon as it receives the message.

The traffic simulator does not update the vehicle behaviour as soon as it receives the message, and this is because of the centralised computation.

Message receiving and behaviour computation process are divided

Aforementioned, the vehicle changes behaviour based on the message it receives. It involves two processes to achieve this one receiving the message and computing the behaviour.

In the state-of-the-art approach, the processes are divided between simulator. The network simulator receives or broadcasts a message. The traffic simulator computes the behaviour.

Time-discrete simulation

The state-of-the approach uses a scheduled timer to trigger the network simulator, and then the network simulator triggers the traffic simulator. The trigger is introduced to establish synchronization among the simulator. The approach makes the entire simulation time-discrete, but in real scenario time is continuous.

Apart from the above-mentioned limitations, one more issue is the developers have to understand both the simulators to extend the framework for creating an ITS application. This increases the complexity of the approach.

3.1.2 Requirements

Based on the limitations discusses in the above section, the requiremens set for the proposed design are:

- The computation of the actors should be distributed to implement the distributed characteristic of ITS
- The actor should be able to compute its behaviour and exchange messages from the same node.
- The complexity involved should be reduced

3.2 Proposed Design

The framework is developed by extending the Carla python API for creating an vehicle and Roadside ITS Subsystem. Using the Carla multi-client support developers can spawn multiple ego vehicles to generate traffic and roadside unit to the simulation server. Before explaining how it performs ITS simulation. Let us discuss the motive behind using Carla,the design of vehicle and Roadside ITS Subsystem and communication component in the framework.

3.2.1 Why Carla

The motive behind choosing Carla are:

- Carla's robust and scalable client-server architecture, this allows us to implement the distributive nature of the ITS.
- Carla Client's domination over the computation of the behaviour of the actor it spawned, the server simulates the actors and updates their states based on control commands sent by the client.
- Carla can perform both time-continuous and time-discrete simulation based on its client-server synchrony mode.
- Carla's web plugin CarlaViz helps us to visualize the entire simulation through live streams. This helps to visually evaluate the whole simulation.

3.2.2 Design of ITS Subsystem

Design of Vehicle

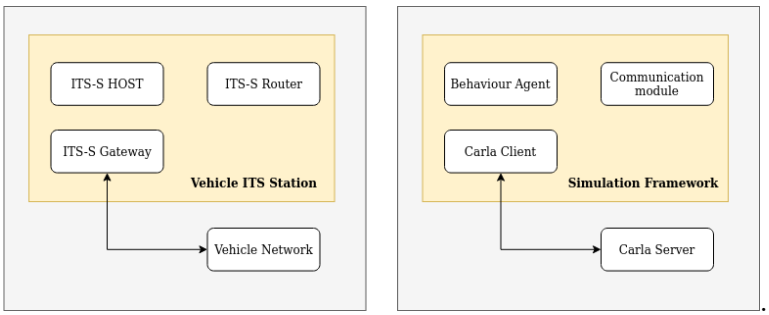


Figure 3.1: Design of Vehicle ITS Subsystem

Aforementioned, a vehicle ITS-S installed in vehicles as On Board Unit (OBU). ITS-S host runs the application to control the OBU, it uses IST-S gateway to gather vehicle information for the proprietary vehicular network, and ITS-S router shares it with other Subsystems. Likewise, the framework uses Carla client to collect the ego vehicle information and a communication module shares it with other Carla Client. Additionally, a behaviour Agent controls the behaviour of the ego vehicle. The framework provides a waypoint navigation based driving agent to support the ego vehicle mobility as a sub-component of behaviour agent.

Design of Roadside Infrastructure

Roadside infrastructure is standalone devices placed along the road as Road Side Unit (RSU), to control traffic. ITS-S host runs the application to control the RSU. It uses ITS-S router and ITS-S gateway to communicate with other Subsystems and traffic infrastructure (e.g., traffic signal, digital display boards). Similarly, the framework uses the communication module to communicate with the ego vehicle and control module to control the RSU.

3.2.3 Design of Communication Component

Aforementioned, peer to peer socket communication is performed to share messages between actors. Each actor spawned in the simulation has an attribute called “role_name”,

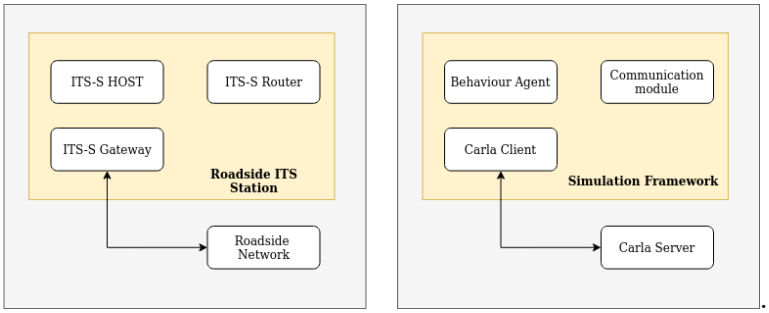


Figure 3.2: Design of Roadside ITS Subsystem

and the framework assigns this attribute with a string combining the Internet Protocol (IP) address and hosts of the actor's socket server (e.g., "role_name" = "ip:host").

The communication module considers the simulation world created by Carla as a live-map. Before broadcasting a message, it retrieves all the actors "role_name" attribute from the world and sends the message to the actor's socket server.

To broadcast a message only within a filter is created. The role of the filter is to retrieve the "role_name" of the actors within a mentioned range. The filtering is done based on the actor's location in the world. For instance, consider an RSU that has a communication radius of 200 meters and it needs to broadcast a message. The communication module uses the filter to get the "role_name" of actors within a 200-meter radius of the RSU and broadcasts the message only to those actors. The figure 3.3 below is the code snippet of the broadcasting function and "isInRange" is the filter function.

3.2.4 ITS Simulation Performed by the Framework

The entire design has three main components, the Carla server that takes care of the simulation, the Client that controls the behaviour of actors and the CarlaViz that visualizes the entire thing.

For instance, consider an RSU is broadcasting a lane closed message and three vehicles moving towards it. The RSU broadcasts the lane change message throughout its life-time. The vehicles spawned at a random location will be moving in its planned path using waypoint navigation until they enter the communication range of RSU.

```
communicationRange = Container().GetState().CommunicationRange
orgin = Container().GetActor()
world = Container().GetWorld()
_list = world.get_actors().filter("*vehicle*")
_list.extend (world.get_actors.filter('passtatic.prop.streetsign'))

orginLocation = orgin.get_location()
orginId = orgin.id

def isInRange(v) :  orginLocation.distance(v.get_location()) <= communicationRange

for vehicle in _list:
    if vehicle.id is not orginId or isInRange(vehicle):
        array = vehicle.attributes.get('role_name').split(':')
        ip = array[1]
        port = array[2]
        self.__loop.run_until_complete(self.Broadcast(data,ip,port))
```

Figure 3.3: Broadcast function code snippet

Once a vehicle enters the communication range, it receives the message and changes its path based on the lane id. The sequence diagram in the figure illustrates the message broadcasted by the RSU is only received by the vehicle only when it enters the communication range. Apart from this, the figure shows that the vehicle and RSU behaviour computation is carried out separately by their respective clients, and they coordinate by sending messages.

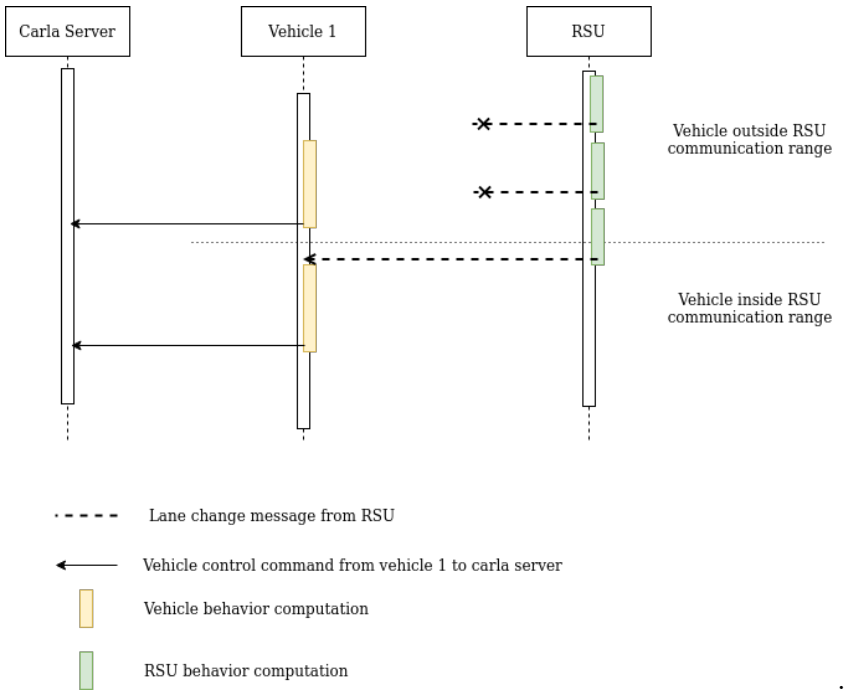


Figure 3.4: Sequence diagram: Simulation Performed by the Framework

Chapter 4

Implementation

First part of this chapter . Starting with its component architecture, followed by a flow diagram and at last, it tells how to extend this framework by adding message handlers and behaviour agent for both vehicle and roadside infrastructure for creating new ITS application.

4.1 Framework

It is a ITS simulation framework written in Python to spawn vehicle and roadside ITS subsystem in Carla simulator, It takes care of creating and maintaining the life-time of the actors in the simulation and provides extensions to customise the behaviour of the actors and messages to interact with other actors.

4.1.1 Configuring the Framework

A set of key-value pairs configures the framework and the spawned actor, and they are:

- **CarlaHost:** This is a required property to provide the IP address of Carla server.
- **CarlaPort:** This is a required property to provide the port of Carla server.
- **Port:** This is a required property to provide the port of actors socket server.

- **Type:** This is a required property, and based on this the framework spawns a vehicle or RSU in the Carla server. It should be either Vehicle or RSU.
- **CommunicationRange:** This is a required property, and based on this the communication of the actor is set.
- **ActorSpawning:** This is an optional property, and based on the spawn point of the actor is determined. The actor is spawned at a random location if the value is not available.
- **ActorDestination:** This is an optional property. Based on this the vehicle destination is decided. The destination is selected at random if the value is not available.

4.1.2 Component Architecture

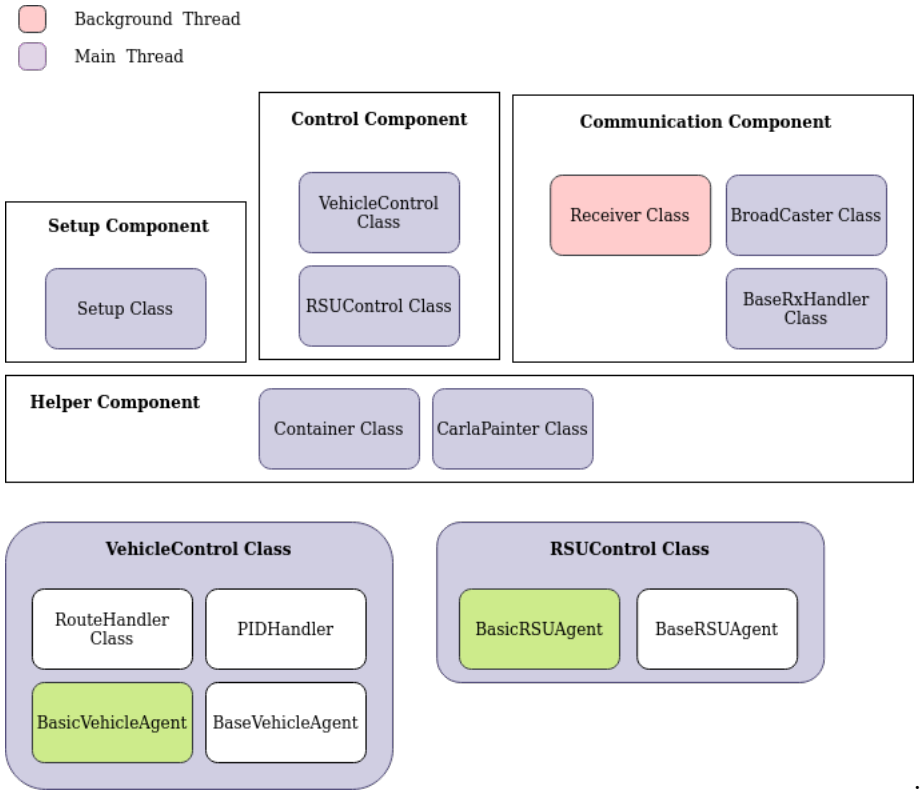


Figure 4.1: Component Architecture

The framework has four components are they are divided based on functionality. The figure 4.1 is the component architecture of the framework and the entire class diagram is available in the appendix.

Setup Component

The responsibility of this component is to set up the entire framework to spawn the actor in the Carla server. It has only one class called "Setup" that acts as the entry point for the framework. The code snippet in the figure 4.2 shows how to initialise the framework.The responsibility of "Setup class" is to create Carla Client instance, initialise the states and global variables.

```
if __name__ == "__main__":
    try:
        frameworkConfigurationFile = "RSU-3Config.json"
        Setup(frameworkConfigurationFile)
    except KeyboardInterrupt:
        print("close!!")
```

Figure 4.2: Code Snippet to Initialise The Framework

Control Component

This component is responsible for creating, spawning and maintaining the lifetime and controlling the actor. This component separate control class for RSU and vehicle called "VehicleControl" and "RSUControl". The "VehicleControl Class" has a driver agent that acts as the mobility model for the vehicle and the agent is based on waypoint navigation.

Communication Component

This component is responsible for broadcasting and receiving messages. It plays a vital role in establishing the coordination and cooperation between the actors. At this stage, the component only supports one-hop communication.

The “Receiver class” receivers the message and pushes it to their respective message handlers based on its type. This class is responsible for initing and maintaining the lifetime of the socket server.

The “BroadCaster Class” broadcasts the message to other actors within the assigned communication range. It provides V2V, V2R and V2X communication modes and based on the mode it filters the actors and sends the message to them.

Helper Component

This component provides support functioning of the framework. It has two classes, they are:

- **Container Class:** It is a singleton class that holds the states of the actor, simulation world and map throughout the life-time of the actor. Additionally, it holds the metadata of “Broadcaster class” and message handler classes. It also holds a key-value dictionary property to store the handled messages, and it can be retrieved by other components when needed.
- **CarlaPainter Class:** It is used to create a client instance for CarlaViz and draw additional features to the visualization. It allows actors to draw text, point and lines.

4.2 Extending the framework

The framework provides extensions for certain sub-component to customize it. The decorator design pattern [20] is used to add the customized component to the framework with affecting its behaviour. This way, users can easily customize by just creating a class and adding the decorator at the top of the class name. The rest of the explains how to add message handlers to the communication module and customise the behaviour of the vehicle and RSU.

4.2.1 Vehicle Behaviour

The behaviour of the vehicle is controlled by the driver agent of the framework. To customise, it the “BasicVehicleAgent class” should be overridden using the “@drivin-

gAgent" decorator.

First, a behaviour class is created inheriting from "BaseVehicleAgent class". The behaviour logic is written inside the "RunStep" abstract function. After creating, the decorator is added at the top. The decorator adds the customised agent to the framework to control the vehicle behaviour.

4.2.2 Roadside Infrastructure Behaviour

The framework does not provide any behaviour agent for RSU. The behaviour agent class is added to the framework using "@rsuAgent" decorator.

First, a behaviour class is created inheriting from "BaseRsuAgent class". The behaviour logic is written inside the "RunStep" abstract function. After creating, the decorator is added at the top. The decorator adds the customised agent to the framework to control the RSU behaviour.

4.2.3 Adding Message Handler

The behaviour of actors depends on the message they receive. The framework has a collection of handlers to receive, process and stores the messages in the container class and later it can be retrieved by the agent class to compute logic based on the message. Any new message handlers can be added to the framework using "@addRxHandler('MessageType')" decorator, where MessageType is the type of the message the class handles. (e.g., "@addRxHandler('DENM')" for handling DENM message).

First, a handler class is created inheriting from "BaseRxHandler class". The handling logic is written inside the "Main" abstract function. After creating, the decorator is added at the top. The decorator adds it to the handler collection in the communication module.

Chapter 5

Evaluation

This chapter discusses the process of testing the framework and evaluating the same. The scope of the evaluative is to ensure the feasibility of the proposed approach and measure the effort involved in extending the framework to create new ITS application and simulate it. For testing two scenarios simulation and each scenario is based on a different application. An overview of the testing infrastructure is discussed before explaining the scenarios.

5.1 Test Setup

For testing, a distributed network is built. The Carla server runs in a machine located at the Trinity College Dublin Linux lab at South Leinster Street. To run the Client (i.e. to spawn the actors) an AWS Linux EC2 instance and Lenovo laptop were used and a separate Linux EC2 instance to run CarlaViz server.

In both, the scenario, three vehicles and three roadside unit were spawned from two client nodes (i.e., two RSU and one vehicle form EC2 instance and two vehicles and one RSU from the laptop). The figure illustrates the entire setup.

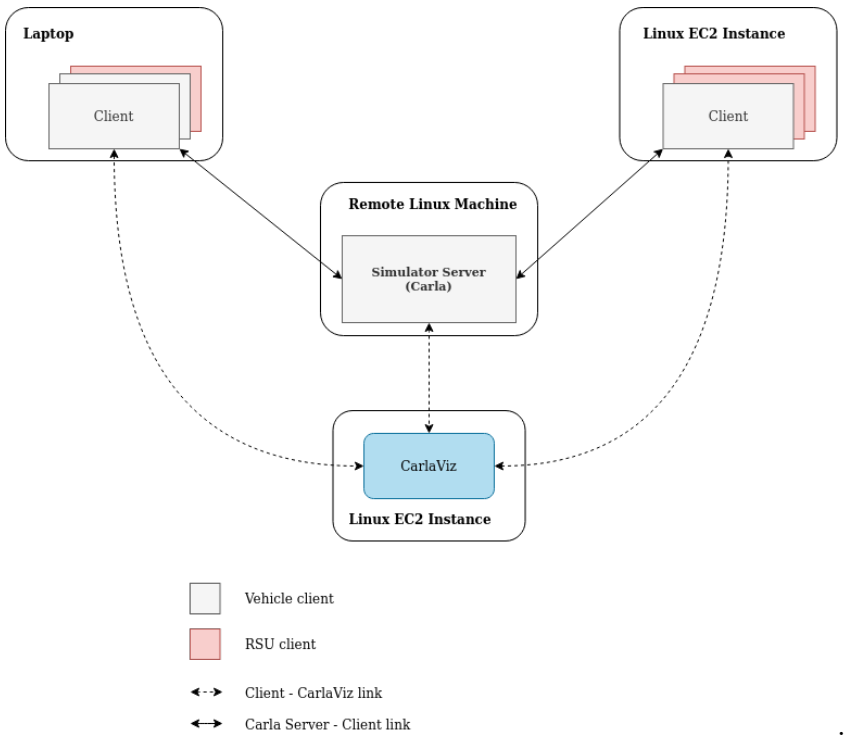


Figure 5.1: Test Setup

5.2 Scenario

5.2.1 CAM Exchange

The motive for simulating this scenario is to test the performance of the communication module to establish vehicular communication between vehicle and RSU. In this scenario, the CAM message is exchanged between all the actors. Once received, the actor displays the message using the CarlaViz "draw_text" function. The content for the CAM message used in this scenario is a string combining the actor's "role_name" and "id" attribute. The code snippets of this scenario is available in appendix.

Implementation

- To create a custom behaviour for RSU to broadcast its "role_name" and "id" attribute as CAM and to display the received message using CarlaViz, "CamRSU" behaviour class is created by inheriting "BaseRsuAgent" class and then override

the “BaseBehaviour” class with “CamRSU” by adding the decorator “@rsuAgent”.

- To make the vehicles receive the CAM, “CaMsgHandler” class is created by inheriting “BaseRxHandler” class and added to the framework using “@addRxHandler”.
- To create a custom behaviour for Vehicles to broadcast its “role_name” and “id” attribute as CAM and to display the received message using CarlaViz, “CamVehicle” behaviour class is created by inheriting “BaseVehicleAgent” class and then override the “BaseBehaviour” class with “CamVehicle” by adding the decorator “@rsuAgent”.

5.2.2 DENM Exchange

The motive of this scenario is to create a sample ITS safety application that sends hazard warnings. It sends lane status and speed limit messages. Two RSU broadcast the lane status message and the third broadcast the speed limit message. In this scenario, vehicles behave according to the DENM it receives from the RSU. The path and speed of the vehicle is visualized using CarlaViz to evaluate the behaviour. The code snippets of this scenario is available in appendix.

Implementation

- To create a custom behaviour for RSU to broadcast lanchange and speed limit messages as DENM, “DenmRSU” behaviour class is created by inheriting “BaseRsuAgent” class and then override the “BaseBehaviour” class with “DenRSU” by adding the decorator “@rsuAgent”. As three RSU broadcast different message all have different behaviour and three classes are created.
- To make the vehicles receive the DENM, “DenMsgHandler” class is created by inheriting “BaseRxHandler” class and added to the framework using “@addRxHandler”.
- To create a custom behaviour for a vehicle to receive DENM and act accordingly and visualize its path and speed using CarlaViz, “DenmVehicle” behaviour class

is created by inheriting “BaseVehicleAgent” class and then override the “Base-Behaviour” class with “DenmVehicle” by adding the decorator “@rsuAgent”.

5.3 Discussion

Since the framework does not have an evaluation component, visualization provided by the CarlaViz is used to evaluate both the scenarios.

The screenshots of DENM scenarios in the appendix shows the message exchange and cooperation between actors. The basic aspect of ITS is to improve the safety and driving experience by establishing cooperation among the traffic. Thus based on the screenshots of the scenario visualization and the test setup made, we can conclude that the proposed method is feasible to perform a distributed ITS simulation.

The decorator design pattern followed in the framework helps it to extend it without altering the flow and behaviour of the. The developer has to create handlers and behaviour classes according to their need. This approach allows developers to create an ITS application with a little knowledge about the framework. Thus we can conclude, the framework can be easily extended.

Chapter 6

Conclusion

6.1 Conclusion and Limitations

This dissertation presents a distributive approach for ITS simulation and implements it by designing a framework extended from the Carla Client API. It is designed based on the limitations of the current state-of-the-art ITS simulation approach.

Chapter 3 reveals that the state-of-the-art approach is designed based on the vehicular network simulation and lacks implementing distributive and real-time characteristics of the ITS. Based on the limitations, a set of requirements are created for the proposed approach. In the later part of the chapter, the design ideas of the proposed approach and compares it through a discussion.

The framework developed based on the proposed design is evaluated in Chapter 5. The scope of the evaluation is to check the feasibility of the approach and flexibility provided by the framework to extend it and simulate ITS applications. The results both the scenarios are favourable to the approach indicating the motive behind designing the scenario is achieved simulating using the framework. In the end, it became evident that the approach is indeed feasible.

Though, the framework performed a distributed simulation of an extended ITS application. It has certain limitations, they are:

- The distributed network to perform the simulation should be created manually. This including installing and running the framework in all client nodes, Carla simulator in the server node and CarlaViz in a client node.
- The communication module can only perform single-hop communication. This is a big limitation while simulating safety application, but a workaround approach can be created by adding message handlers with certain protocols.
- The performance of nodes can affect the performance of the simulation. A workaround approach is to make use of the synchronous mode of Carla server.

6.2 Future work

The designed framework was able to perform the distributed ITS simulation and implemented the distributed and real-time characteristics of ITS. The framework is designed as a proof of concept for the approach and there are many this to be considered to improve the framework and the import things are mentioned below

- Adding an evaluation component. The current design only provides the visualization using CarlaViz to evaluate the behaviour. Addition of performance metrics for both actors and ITS environment will improve the evaluation process.
- Creating an interactive dashboard that controls the simulation environment and visualize evaluation matrices.
- The communication component only supports single-hop communication. It should be improved to support various protocols.

Bibliography

[1] ETSI, "Intelligent transport systems (its);vehicular communications; basic set of applications; part 3: Specifications of decentralized environmental notification basic service," *EN 302 637-3 V1.2.1 (2014-09)*.

[2] F. Arena and G. Pau, "An overview of vehicular communications," *Future Internet*, vol. 11, p. 27, 01 2019.

[3] C. Sommer, R. German, and F. Dressler, "Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis," *IEEE Transactions on Mobile Computing (TMC)*, vol. 10, pp. 3–15, January 2011.

[4] Carla, "Carla simulator documentation," *https://carla.readthedocs.io/en/latest/*.

[5] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," *arXiv preprint arXiv:1711.03938*, 2017.

[6] "Statistical pocketbook 2019," *Mobility and Transport - European Commission*, May 2020.

[7] ETSI, "Intelligent transport systems (its); communications architecture," *ETSI EN 302 665 V1.1.1 (2010-09)*.

[8] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner, "Microscopic traffic simulation using sumo," in *The 21st IEEE International Conference on Intelligent Transportation Systems*, IEEE, 2018.

[9] N. E. Lownes and R. B. Machemehl, "Vissim: A multi-parameter sensitivity anal-

ysis," in *Proceedings of the 38th Conference on Winter Simulation*, WSC '06, p. 1406–1413, Winter Simulation Conference, 2006.

[10] G. F. Riley and T. R. Henderson, *The ns-3 Network Simulator*, pp. 15–34. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[11] A. Varga, *OMNeT++*, pp. 35–59. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[12] 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 *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pp. 260–265, 2011.

[13] D. Marinescu, J. Čurn, M. Bouroche, and V. Cahill, "On-ramp traffic merging using cooperative intelligent vehicles: A slot-based approach," in *2012 15th International IEEE Conference on Intelligent Transportation Systems*, pp. 900–906, 2012.

[14] M. Segata, S. Joerer, B. Bloessl, C. Sommer, F. Dressler, and R. Lo Cigno, "PLEXE: A Platooning Extension for Veins," in *6th IEEE Vehicular Networking Conference (VNC 2014)*, (Paderborn, Germany), pp. 53–60, IEEE, 12 2014.

[15] M. Alam, J. Ferreira, and J. Fonseca, "Introduction to intelligent transportation systems," *Intelligent Transportation Systems Studies in Systems, Decision and Control*, p. 1–17, 2016.

[16] "Information technology - open systems interconnection - basic reference model: The basic model," *ISO/IEC 7498-1*, 1994.

[17] D. Jiang and L. Delgrossi, "Ieee 802.11p: Towards an international standard for wireless access in vehicular environments," in *VTC Spring 2008 - IEEE Vehicular Technology Conference*, pp. 2036–2040, 2008.

[18] ETSI, "Intelligent transport systems (its);vehicular communications; basic set of applications; part 2: Specification of cooperative awareness basic service," *EN 302 637-2 V1.3.1 (2014-09)*.

[19] M. Karoui, M. Kassab, H. Aniss, and M. Berbineau, "Enhance veins simulator

for realistic evaluation scenarios," in *International Workshop on Communication Technologies for Vehicles*, pp. 128–140, Springer, 2017.

[20] R. Guru, "Decorator design pattern," *https://refactoring.guru/design-patterns/decorator*.

Appendix 1

This appendix contains the class diagram of the framework. It is divided into three parts:

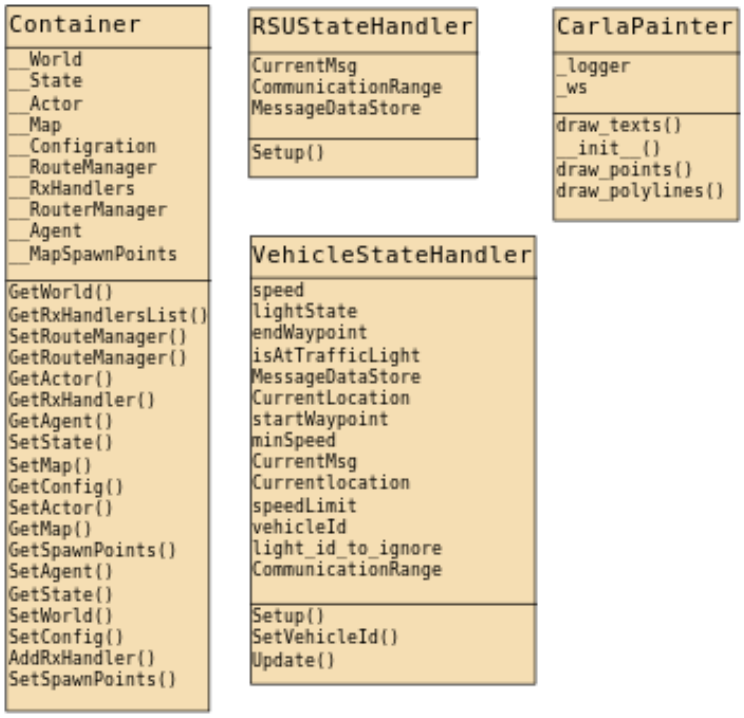


Figure 1: Class Diagram of Helper Component

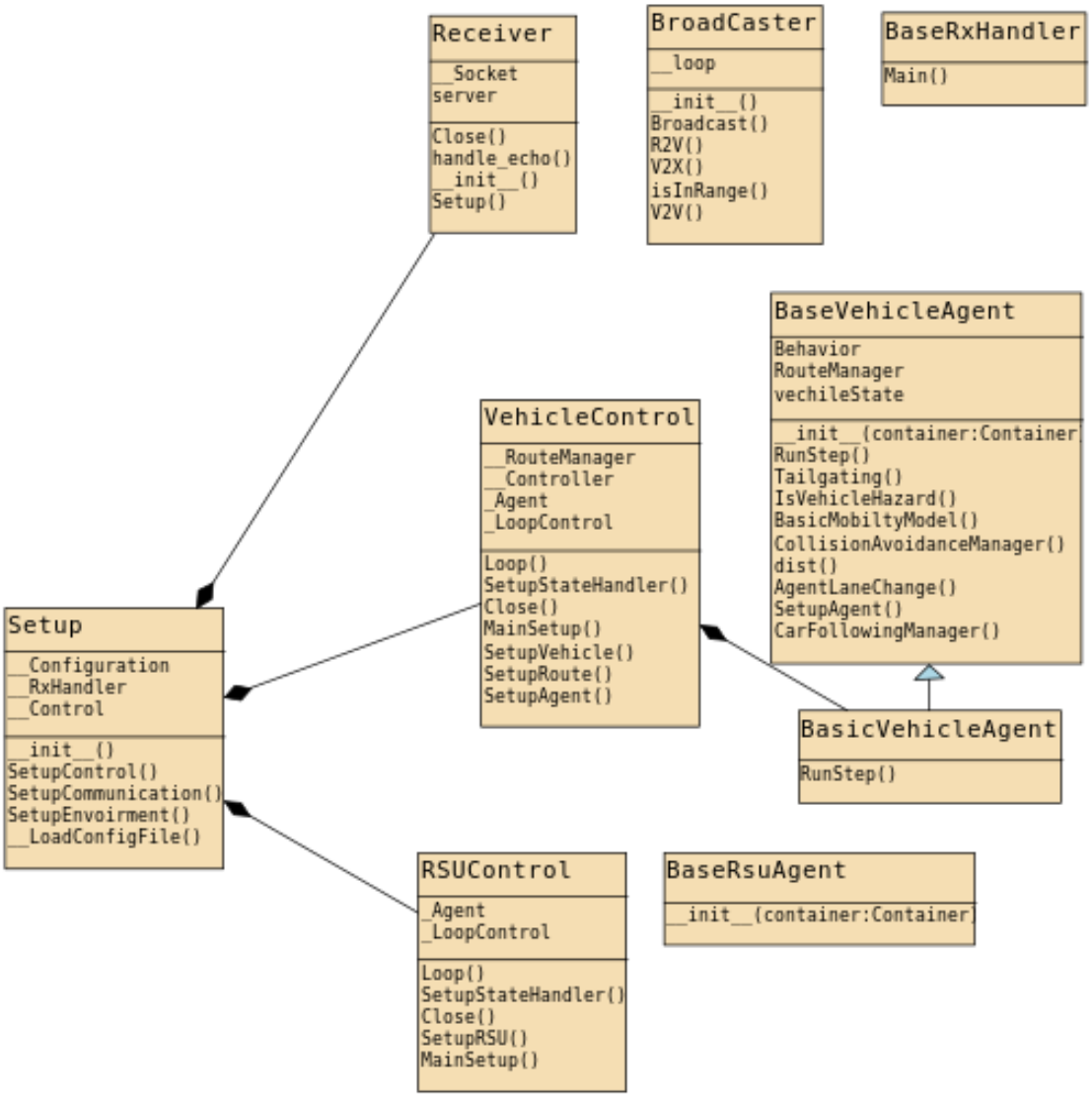


Figure 2: Class Diagram of Setup, Control and Communication Components

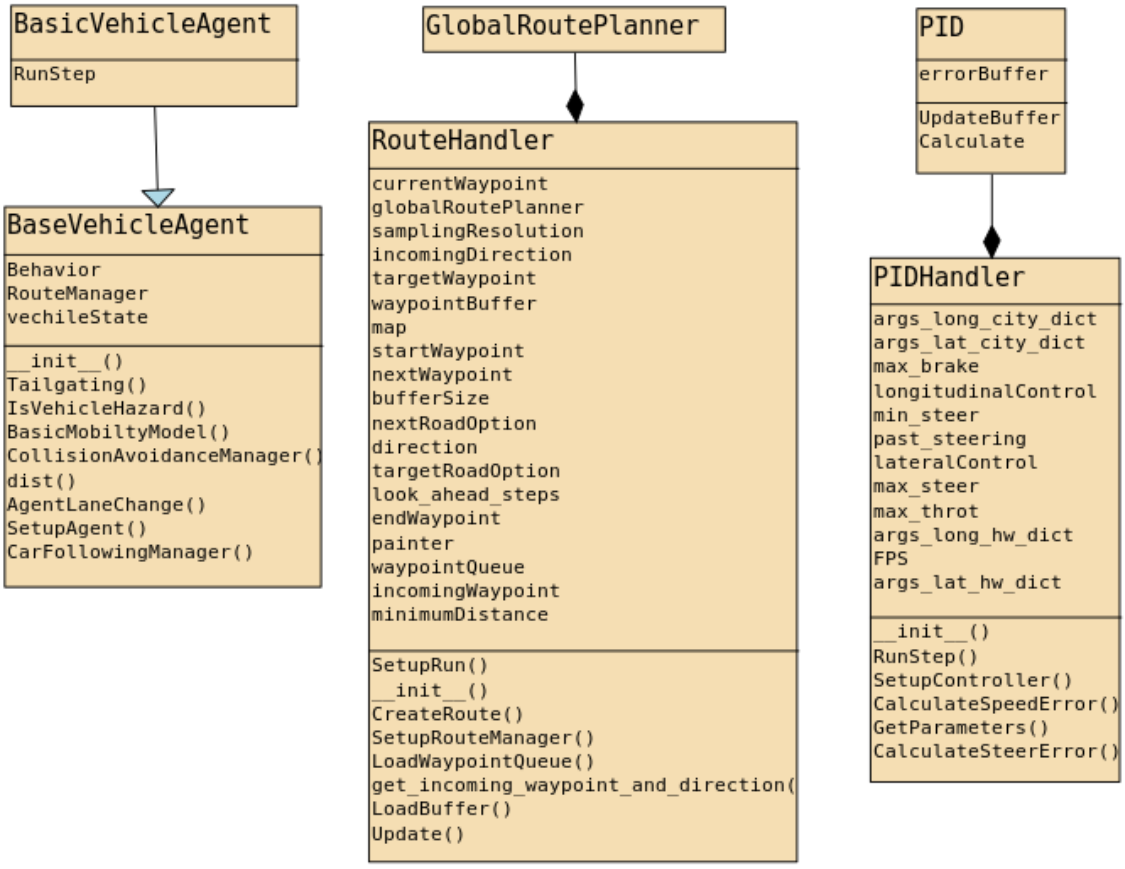


Figure 3: Class Diagram of Vehicle Control Class

Appendix 2

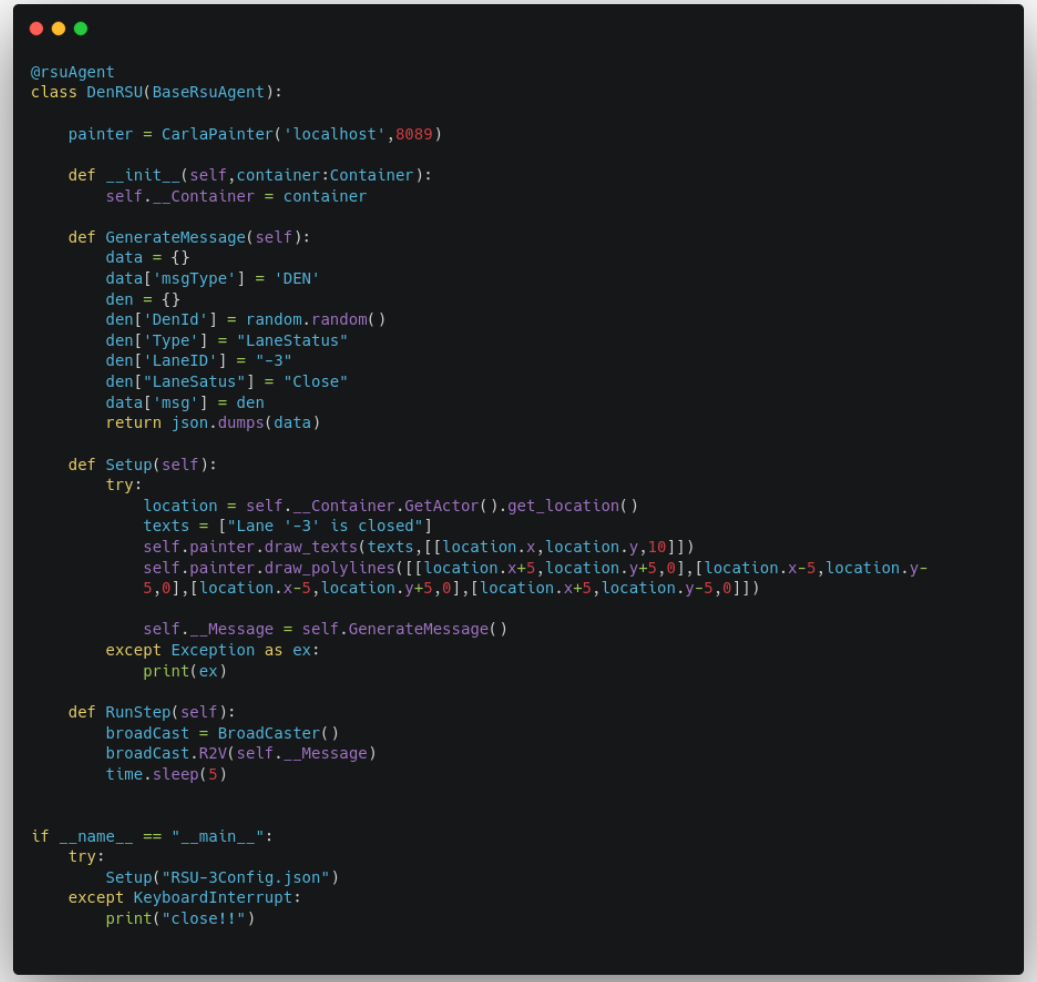
This appendix contains code snippets of the two scenarios implemented by extending the designed framework

```
### DEN Message handler ------------

@addRxHandler('DEN')
class DenHandler(BaseRxHandler):

    def __init__(self,container:Container):
        self.__Container = container

    def Main(self,data):

        message = data['msg']
        denId = message['DenId']
        # ["DEN"]
        messageDict : dict = self.__Container.GetState().CurrentMsg["DEN"]

        #Checking if the Den message exists
        if denId in messageDict.keys():
            #if yes skip
            return

        #if no add to the dict
        message['Status'] = "Pending"
        messageDict[denId] = message

        # update the state
        self.__Container.GetState().CurrentMsg['DEN'] = messageDict
```

Figure 4: Code Snippet of DEN Message Handler

```python
@drivingAgent
class DENAgent(BaseVehicleAgent):

    DenSpeedLimit = None
    painter = CarlaPainter('localhost',8089)

    def __init__(self,container:Container):
        self.__Container = container

    def CheckForLaneChangeDen(self,location):
        message : dict = self.__Container.GetState().CurrentMsg['DEN']

        for _, value in message.items():
            #lane change message
            if value['Type'] == "LaneStatus":
                cwp = self.RouteManager.currentWaypoint
                claneId = cwp.lane_id

                if value['Status'] is not 'Sucess' and str(claneId) == value['LaneID']:

                    vehicle_list = self.__Container.GetWorld().get_actors().filter("*vehicle*")
                    def distace(v): return v.get_location().distance(location)
                    vehicle_list = [v for v in vehicle_list if distace(v) < 45 and v.id !=
self.vechileState.vehicleId]
                    status = self.AgentLaneChange(location,cwp,vehicle_list)
                    value['Status'] = 'Sucess' if status else 'Failed'

                    return True

            elif value['Type'] == "SpeedLimit":
                print(value)

                if value['Status'] is not 'Sucess':
                    if int(value['Speed']) == 0:
                        self.DenSpeedLimit = None
                    else:
                        self.DenSpeedLimit = int(value['Speed'])
            else:
                pass
        return False

    def RunStep(self):

        egoVehicle = self.__Container.GetActor()
        ego_vehicle_loc = egoVehicle.get_location()

        #display lane id and speed
        texts = [f"Lane: {self.RouteManager.currentWaypoint.lane_id}\n Current Speed:
{int(get_speed(egoVehicle))} "]
        self.painter.draw_texts(texts,[[ego_vehicle_loc.x,ego_vehicle_loc.y,10]])

        targetSpeed, behaviour = self.BasicMobiltyModel()

        if behaviour is "normal" and self.CheckForLaneChangeDen(ego_vehicle_loc):
                targetSpeed = min(self.Behavior.max_speed, self.vechileState.speedLimit - 5)

        if self.DenSpeedLimit is not None:
            targetSpeed = self.DenSpeedLimit

        return targetSpeed
```

Figure 5: Code Snippet of DEN Vehicle Behaviour



```python
@rsuAgent
class DenRSU(BaseRsuAgent):

    painter = CarlaPainter('localhost',8089)

    def __init__(self,container:Container):
        self.__Container = container

    def GenerateMessage(self):
        data = {}
        data['msgType'] = 'DEN'
        den = {}
        den['DenId'] = random.random()
        den['Type'] = "LaneStatus"
        den['LaneID'] = "-3"
        den["LaneSatus"] = "Close"
        data['msg'] = den
        return json.dumps(data)

    def Setup(self):
        try:
            location = self.__Container.GetActor().get_location()
            texts = ["Lane '-3' is closed"]
            self.painter.draw_texts(texts,[[location.x,location.y,10]])
            self.painter.draw_polylines([[location.x+5,location.y+5,0],[location.x-5,location.y-
            5,0],[location.x-5,location.y+5,0],[location.x+5,location.y-5,0]])

            self.__Message = self.GenerateMessage()
        except Exception as ex:
            print(ex)

    def RunStep(self):
        broadCast = BroadCaster()
        broadCast.R2V(self.__Message)
        time.sleep(5)


if __name__ == "__main__":
    try:
        Setup("RSU-3Config.json")
    except KeyboardInterrupt:
        print("close!!")
```

Figure 6: Code Snippet of DEN RSU Behaviour

Appendix 3

This appendix contains screen shot of the DENM exchange scenario. In this simulation the vehicle initially moves in lane "-7" and when it comes in the communication range of RSU, it receives the lane closes message and it initiates lane change and moves to lane "-6" and at last when it comes near last RSU, it receives speed limit warning and reduces the speed

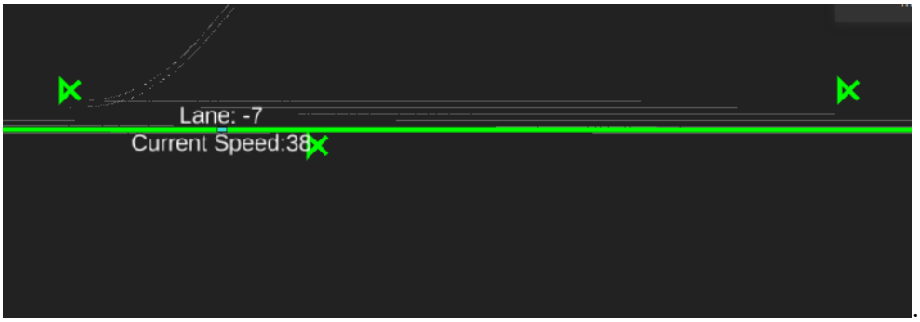


Figure 7: Scene 1: initially moves in lane '-7'

Figure 8: Scene 2: initiates lane change

Figure 9: Scene 3: Changes the lane

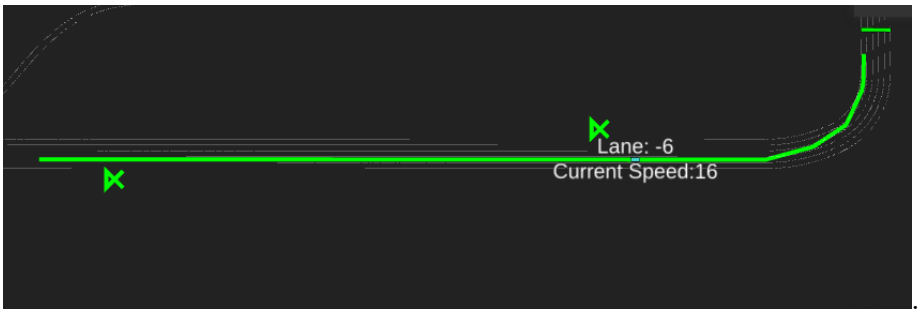


Figure 10: Scene 4: Reduces the speed

Appendix 3

- **Project git repository :** https://github.com/amrishAK/ITS-Simulation-Framework
- **Installing Carla :** https://carla.readthedocs.io/en/latest/start_quickstart/
- **Installing CarlaViz:** https://carla.readthedocs.io/en/latest/plugins_carlaviz/