A Simulation Platform for Connected Autonomous Vehicles Incorporating Physical and Communication Simulators

by

Yuhao Chen
B.Eng., Northwest University, 2023

A Project Report Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

© Yuhao Chen, 2024
University of Victoria

All rights reserved. This project report may not be reproduced in whole or in part,
by
photocopying or other means, without the permission of the author.

A Simulation Platform for Connected Autonomous Vehicles Incorporating Physical and Communication Simulators

by

Yuhao Chen
B.Eng., Northwest University, 2023

Supervisory Committee

Dr. Lin Cai, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Kin Fun Li, Departmental Member
(Department of Electrical and Computer Engineering)

ABSTRACT

This project report provides a holistic record of the development of a connected autonomous vehicle simulation framework incorporating a physics simulator and a communication simulator. The development of this tool aims to help researchers in vehicle communication protocols to evaluate the simulated performance of their solutions in the physical world. By using this tool, communication researchers can observe the impact of their communication protocols on the actual connected autonomous vehicle operation process without the need to delve into the underlying logic of vehicle kinematic simulation. They only need to configure simple parameters and deploy their own protocols on the communication simulator and see the effect. This project report will start by introducing the components and operating principles of the entire system, and then demonstrate its usage through a simple simulation example.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Tables vi

List of Figures vii

Acknowledgements viii

Dedication ix

1 Introduction 1

2 The Problem to be Solved 4

3 Joint Simulation Platform 7

3.1 Overview 7

3.2 Robot Operating System 2 9

3.2.1 Node and Message 10

3.2.2 Topic and Service 10

3.3 F1Tenth Gym 12

3.3.1 Command-Observation Model 12

3.3.2 Single-Track Kinematic Model 13

3.3.3 Maps 15

3.4 RViz2 16

3.5 Gym Bridge 18

3.6 Car Control 19

3.6.1 Convention 19
3.6.2 Autonomous Driving Algorithms 21
3.7 Auxiliary Moudules 25
3.7.1 Waypoint Recorder and Displayer 25
3.7.2 Measurement Module 25
3.8 Communication Simulation Design 27
3.9 Communication in Docker and ROS2 28
3.10 NS3 and Tap Bridge 30
3.11 Environment Configuration 32
3.11.1 ROS2 33
3.11.2 NS3 33
3.11.3 Docker Containers 33
3.11.4 Host Machine 34
3.12 Process and Automation 34
3.12.1 Preparation Phase 35
3.12.2 Defining Driving Rules 35
3.12.3 NS3 Script Writing 35
3.12.4 Deployment Phase 35
3.12.5 Running the Simulation 35
3.12.6 Improvement 36
3.12.7 Automation Scripts 36

4 Experiments **37**

5 Conclusions **42**

Bibliography **43**

List of Tables

Table 4.1 Simulation results. 41

List of Figures

Figure 1.1 Structure of the project report. 3

Figure 3.1 System overview. 8
Figure 3.2 ROS2 architecture [1]. 10
Figure 3.3 ROS2 topic mechanism. 11
Figure 3.4 ROS2 service mechanism. 11
Figure 3.5 F1Tenth Gym command-observation model. 12
Figure 3.6 Demonstration of single-track kinematic model. 14
Figure 3.7 Occupancy grid. 15
Figure 3.8 UI of RViz2. 16
Figure 3.9 Gym bridge. 18
Figure 3.10 High-speed motion trajectory using small L. 22
Figure 3.11 High-speed motion trajectory using large L. 23
Figure 3.12 Geometric relationship in the follow the arc. 24
Figure 3.13 Calculation of time to collision. 26
Figure 3.14 Communication simulation topology. 27
Figure 3.15 Connection structure between container and node. 31

Figure 4.1 Experiment map. 38
Figure 4.2 NS3 configuration, channel attributes can be selected flexibly. . 38
Figure 4.3 Visualization of the physical environment. 39
Figure 4.4 Kinematic performance with channel delay of 10ms. 39
Figure 4.5 Kinematic performances with channel delay of 100ms, 200ms and 300ms. 40

ACKNOWLEDGEMENTS

I would like to thank:

my supervisor, Dr. Lin Cai, for mentoring and supporting me in this wonderful learning period.

Lecture instructors, for teaching diversiform knowledge to let me extend my horizon in the electrical and computer field.

Labmates and classmates, for helping me quickly integrate into this new environment, and providing me with a lot of valuable information and suggestions.

My family and friends, for their long-term support.

Life's experiences are like a journey; what matters is not the destination, but the scenery and insights along the way.

Hua Yu

DEDICATION

This project report is dedicated to researchers in vehicular communication. Hope this project report and the tools I developed can be of some help to researchers in vehicular communication.

Chapter 1

Introduction

In recent years, connected and autonomous vehicles (CAV) have rapidly entered our everyday lives. Autonomous driving and vehicular communication are becoming increasingly important. In 2023, Cruise and Waymo launched their autonomous taxi services in San Francisco, California, and in the first half of 2024, Baidu's Apollo Go service was introduced in multiple cities across China, sparking widespread public discussion.

To realize intelligent transportation in the real world, vehicle hardware, autonomous driving algorithms, and vehicular communication capabilities all play crucial roles. How to better leverage communication systems to assist autonomous driving algorithms in making decisions has become a key research interest for many scholars. Research on autonomous driving algorithms and vehicular communication has been highly active, with many researchers in both the control and communication fields attempting to explore this topic. However, there are very few laboratories capable of supporting large-scale experiments on integrated autonomous driving and communication. This is due to the high requirements for software and hardware resources, the high cost of experimental equipment, and the extensive space and human resources needed for such experiments.

In the context of limited resources, computer simulation is a commonly used solution. However, there are few open-source CAV simulators to choose from. These simulation platforms either excel in physical simulation, being able to simulate various physical sensor observations and complex physical effects, but lack highly customizable communication simulation, such as CARLA and PanoSim, or they incorporate professional communication simulators but are not specifically designed for autonomous vehicles in their physical simulation aspects, such as CoCo Games [2, 3].

In our lab, which focuses on communication research, using these existing solutions often requires extensive modifications and customized development.

Therefore, rather than undertaking extensive modification work, it is preferable to develop a moderately complex physical-communication joint simulation platform specifically designed for CAV communication researchers. This simulation platform needs to handle both the physical simulation of the vehicle's kinematic state and the simulation of inter-vehicle network communication. The physical simulation involves modeling the vehicle's pose and motion trajectory during various conditions such as acceleration, deceleration, braking, turning, and collisions. Additionally, the platform should allow for flexible integration of different channels and protocols for network communication through communication simulation, enabling vehicular communication researchers to implement their own modules. This would facilitate the observation of the entire process of network data packets being transmitted between vehicles and delivered to upper-layer applications. Also, it must be simple yet precise in physical simulation, focusing on macro vehicle physics without delving into physical details to ensure simulation efficiency, while also offering a high degree of professionalism and customization in autonomous driving algorithms and communication simulation, meeting the needs of vehicular communication researchers for communication module design and experimentation.

This project will address the specific challenges faced by our lab and introduce the design principles of each module of this new simulator, concluding with a demonstration of a simple use case. The structure of this project report will proceed according to the logic illustrated in Figure 1.1.

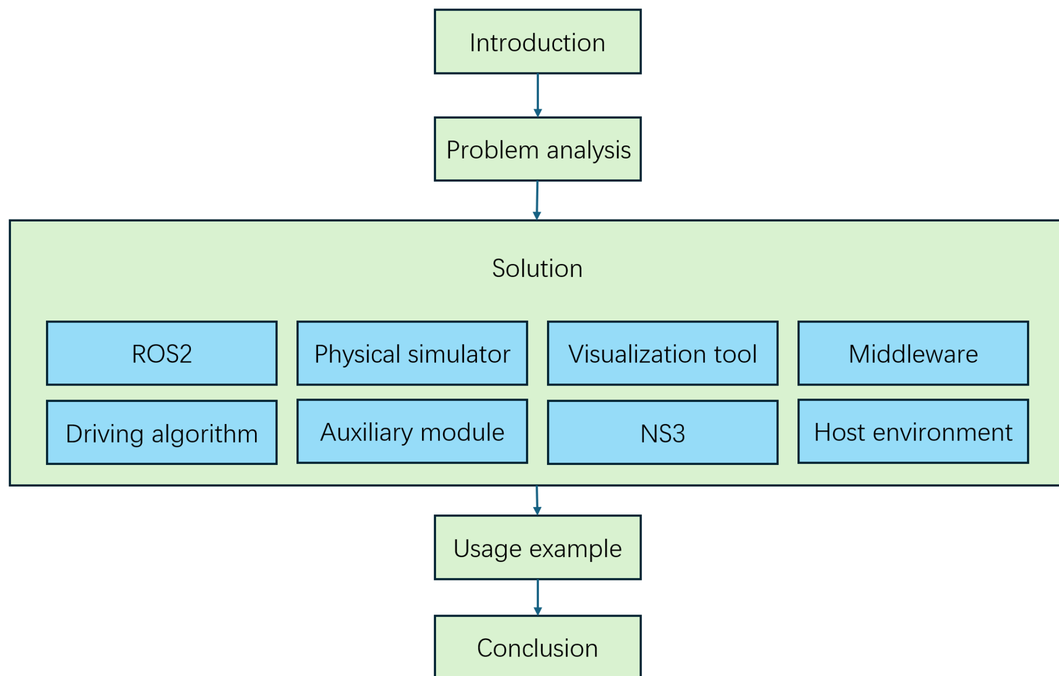


Figure 1.1: Structure of the project report.

Chapter 2

The Problem to be Solved

The development of this tool set originated from the real-world needs of researchers in the laboratory who conduct vehicle communication research for autonomous driving. In the process of developing vehicle communication protocols, they often face issues such as unclear design standards and difficulties in quantifying theoretical standards. For example, the question of how many milliseconds of delay requirement is suitable for the transmission of vehicle collision avoidance and emergency braking messages often involves the following problems:

- Are delay requirements for different vehicles different?
- How will the delay requirement change under different road friction coefficients or other road settings?
- How much time does the vehicle need to react after receiving the emergency braking signal?
- Do different driving control strategies have different delay requirements?

Due to the complexity of these issues, previous researchers have not been able to establish clear standards. Although transportation departments in various countries have conducted a series of studies on the safety standards for autonomous driving vehicles, the work of translating these safety standards into communication requirements is still very arduous and complex. Moreover, due to the complexity of the real world and the diversity of communication standards, this translation also lacks wide applicability.

However, these problems must be solved in the real engineering field. If theoretical derivation is difficult to achieve, we can use experimental verification methods. For research laboratories studying vehicle communication, their resources are usually limited, whether in terms of economic resources, human resources or site resources, and it is difficult to support the testing tasks of multiple vehicles in a large number of different scenarios.

Therefore, a demand has arisen for a simulation tool that can realistically simulate such testing. Vehicle communication researchers hope to have a simulation tool that can accurately and reliably simulate various aspects of vehicle communication technologies, and through this tool, observe the specific impact of different driving strategies and communication protocols on the behavior of autonomous driving vehicles, so as to evaluate whether the design can meet the communication requirements of the real-world autonomous driving.

Looking at various simulation tools, we can easily find professional physical simulation software and communication simulation software. To meet the above needs, a currently feasible approach is to first simulate the performance of communication protocols through a communication simulator, and then test whether such performance can support the needs of autonomous driving applications in the physical simulation software. However, the efficiency and accuracy of this approach are undoubtedly very low.

Therefore, to optimize this testing process, autonomous driving vehicle communication researchers now need a joint simulator that has the performance of a professional vehicle driving simulator, while also taking into account the simulation accuracy of the communication network part. This is the problem that this project aims to solve.

This idea is not difficult to design theoretically, but it is rarely implemented in reality. Analyzing this problem, it can be divided into five sub-problem:

1. How to perform physical simulation, the selection of the vehicle driving physical simulator.

2. How to perform autonomous control, the selection of the autonomous driving function platform and algorithms.

3. How to perform communication simulation, the selection of the network communication simulator.

4. How to connect the above modules and make them work collaboratively?

5. How to visualize the results and make accurate measurements of the simulation to build quantifiable standards?

This project first investigate these problems, conduct systematic design and implementation for the corresponding modules, and finally assemble a complete physical-communication simulation platform that serves to facilitate the research on connected autonomous vehicles. This will allow communication researchers to conveniently deploy their own communication protocols, observe their impact on the actual autonomous driving behavior of vehicles, without the need to understand the underlying principles of the autonomous driving system and algorithms, thereby accelerating the research process and saving the cost.

Chapter 3

Joint Simulation Platform

3.1 Overview

As introduced in the previous chapter, first, we need to make a selection on the physical simulator. The F1Tenth project is a platform built by researchers from the University of Pennsylvania School of Engineering for autonomous driving vehicle racing competitions. It provides a complete set of supporting facilities, including a physical simulation software F1Tenth Gym, a development board adapted to the ROS2 operating system, and a series of hardware to help autonomous driving enthusiasts explore the principles and implementation of autonomous driving algorithms and deploy them on F1Tenth racing cars for testing [4].

Therefore, with the help of this powerful platform, we can modify it according to the real needs. In the physical simulation module, we directly use the Gym simulator provided by F1Tenth, which has the advantages of being completely open source, with moderate complexity, fully compatible with ROS2 and developed based on ROS2 and RViz2. For the control module, we can directly use ROS2 to write control algorithms to interact with F1Tenth Gym.

At the communication simulation level, we use NS3, which is one of the most popular open-source networking simulator. This can allow networking researchers to directly deploy their developed communication modules into the network environment without the need for a large amount of code transfer and adaptation work.

Secondly, we need to connect these two series of tools. ROS2 uses eProsima Fast-DDS to implement, which uses the publisher-subscriber model. In simple terms, it transmits data over the network. Fortunately, NS3 is a network simulator, and we

can use the Linux tap device to send the real data packets from ROS2 into NS3 for processing. The latency of tap device can be as low as nanoseconds and it is used by many high-precision measurement tools. Therefore, it can ensure the rapid delivery of data packets [5, 6]. For multi-cars simulation, docker containers are used to play the part of different kinds of agents.

Finally, we need to develop a series of auxiliary functions, such as the measurement function and control function of the simulator.

Figure 3.1 shows the overall system design architecture.

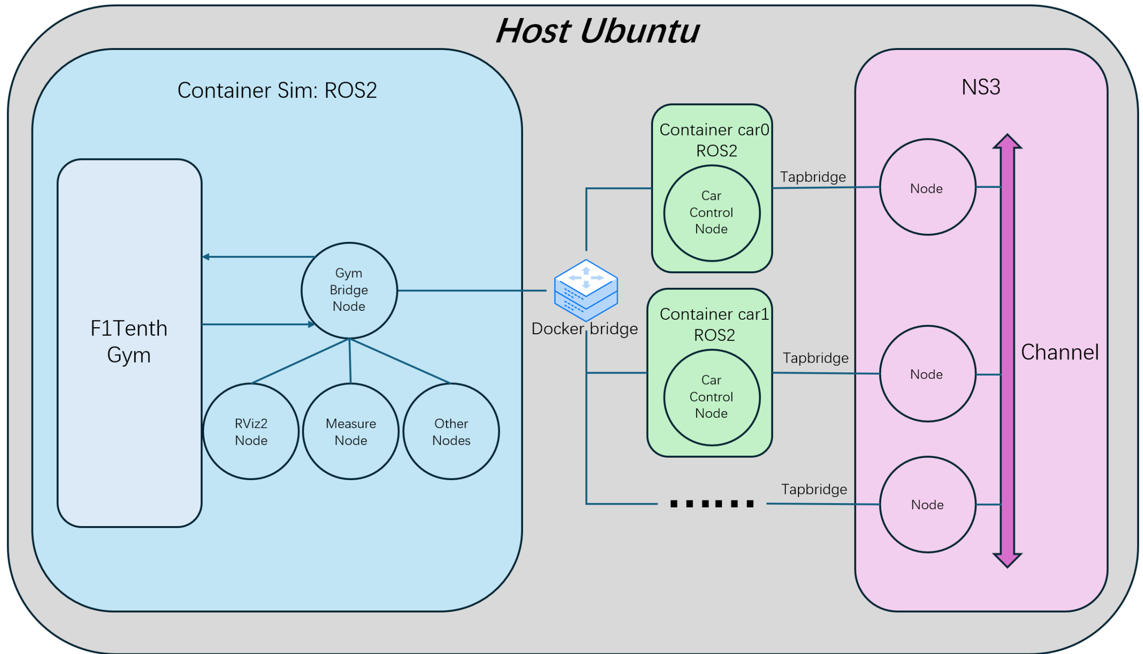


Figure 3.1: System overview.

Among them, the F1Tenth Gym simulator and visualization nodes run in a container named Sim. Each vehicle control node runs in a separate container, which is to simulate the independent operating decision environment of each vehicle in a real situation. NS3 runs on the host Ubuntu, which manages the communication channels between vehicles.

Taking the emergency braking process as an example, the nodes in the Sim container play the role of the physical world, receiving the control commands from the car control nodes and passing them to the F1Tenth Gym. The F1Tenth Gym executes the control commands, updates its internal physical simulation state, and transmits the updated results and new observation information (such as lidar information) back,

which are displayed by RViz2 for visualization, and the measure node records the execution time, the distance between vehicles, and other information. Finally, the observation information is distributed back to the car control nodes.

If a vehicle decides to send others an emergency braking signal, the data packet carrying the command will be sent into the NS3 node through the tapbridge on the other side, flow through the channel in NS3, and be sent to the container of another vehicle. After the other vehicle receives it, it will immediately perform emergency braking. Thus, a simulation of emergency braking between two vehicles is completed.

ROS2 is implemented based on FastDDS, and the underlying protocol of FastDDS is either TCP or UDP. This means that the information exchange between ROS2 nodes and between different devices running ROS2 is directly carried out through network packets. When ROS2 generates and sends a new data packet, it broadcasts the packet to all possible receivers, including the virtual network interface connected to the tap device. The tapbridge in NS3 listens to these tap devices in real-time, and once a new data packet arrives, it forwards it as a packet generated by the NS3 node into the NS3 channel.

Next, I will introduce the principles and implementation details of each subsystem.

3.2 Robot Operating System 2

ROS2, short for Robot Operating System 2, is an open-source robot control software that aims to organize and coordinate the various sensors, processors, and everything else the robot needs to run [7]. Although it is called an "Operating System", it does not directly handle the tasks of processor scheduling, memory management, file systems, and other traditional operating system functions. Instead, it focuses on communication and decision-making capabilities.

As shown in Figure 3.2, ROS2 needs to run on a traditional operating system. It utilizes the Fast DDS, Cyclone DDS, and Connext DDS protocols to transmit data, allowing the various components to communicate and collaborate with each other. On top of this, it provides the ROS Client Library API to help developers build their own ROS2 node programs. It supports three mainstream programming languages: C++, Python, and Java [1].

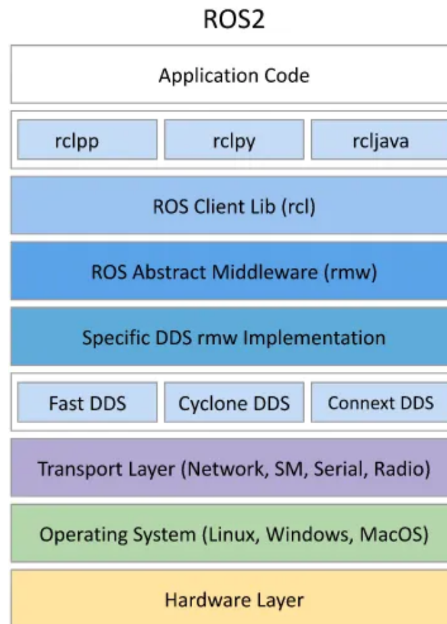


Figure 3.2: ROS2 architecture [1].

3.2.1 Node and Message

A node is the smallest unit to handle information in ROS2. ROS2 treats all functionalities as node behaviors. A node can receive and transmit messages. Similar to a function, given some inputs, it produces corresponding outputs. Correspondingly, a message is the basic information unit, passing through all the ROS2 systems. They are sent and received by different kinds of nodes. Messages are also the top abstraction of data, like Lego bricks - it can contain basic data types and be packaged into other messages. Using this feature, developers can create highly customized messages.

The entire ROS2 system is composed of countless nodes operating in parallel, exchanging messages, and producing corresponding actions to control the robot's behavior.

3.2.2 Topic and Service

The message mechanism allows nodes to communicate freely with each other, and a node can choose to exchange information with any other node. However, freedom is not necessarily the best approach. Imagine if each node had to manually manage which nodes to communicate with, the development work would become exceptionally cumbersome and prone to bugs. Therefore, the two important basic concepts of

service and topic were introduced.

Topic is a broadcasting mechanism. A node can choose to subscribe to a topic to receive information under that topic, or act as a broadcast station for that topic, publishing information to it. Subscribing and publishing to topics are both free actions. Adding this intermediate layer of topics in node communication can greatly facilitate the development work of the nodes.

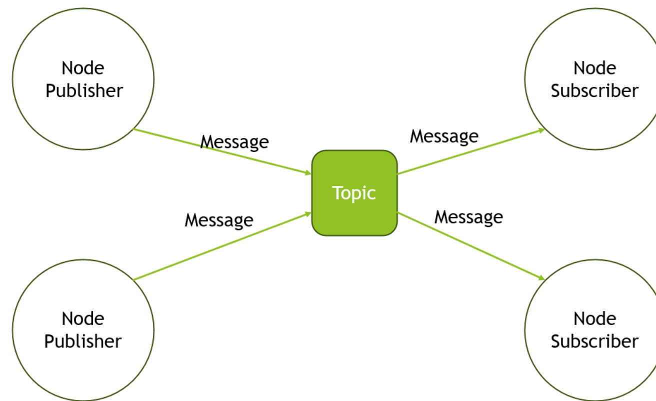


Figure 3.3: ROS2 topic mechanism.

Service is based on the client-server design principle, and it is very effective in scenarios where there is no need to constantly send and receive information. Whenever a node needs another node, it will first send a request to the service server of the corresponding node. The corresponding node will then receive the request and return the message that the client needs.

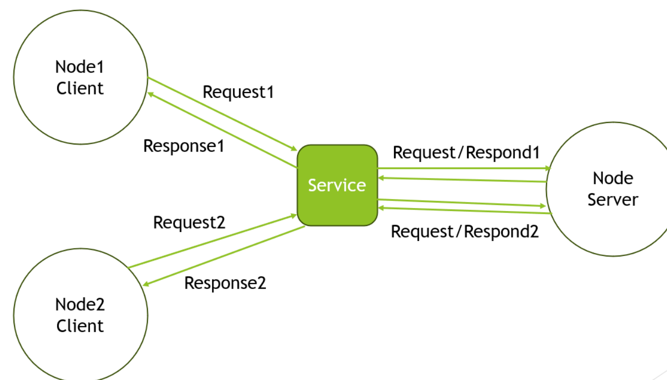


Figure 3.4: ROS2 service mechanism.

Through ROS2's nodes, messages, topics, and services, the entire ROS2 subsystem can be built.

3.3 F1Tenth Gym

F1Tenth Gym is a tool developed by the F1Tenth organization to simulate the motion of F1Tenth vehicles [8]. While it has the capability of physical simulation, it has a simpler structure and is easier to use compared to other physical simulators like Gazebo.

Specifically, physical simulators like Gazebo and Unity have a finer simulation granularity [9, 10]. For example, they can simulate the forces and displacements on each structural component of a vehicle when it is subjected to a collision. This is certainly an excellent feature, but it may not be desirable for researchers studying vehicle-to-vehicle communication. The latter often do not care about the internal physical state of the vehicle and very small physical effects. Therefore, using tools like Gazebo and Unity often requires a lot of preliminary work, such as vehicle modeling, node attribute setting and binding, and map creation, etc.

When vehicle-to-vehicle communication researchers focus on the external behavior of the vehicle and its interaction with the environment, they are often only concerned with the vehicle's appearance, motion state, and the simplest external environment. Therefore, F1Tenth Gym can well match this requirement.

3.3.1 Command-Observation Model

The F1Tenth Gym physical simulator is based on a command-observation model, as shown in Figure 3.5. During the simulation, for the outside world, F1Tenth Gym only provides two interfaces: command input and observation output, and the internal state of the simulator is a black box. This encapsulated design pattern simplifies the complexity of external development, allowing developers to not worry about the internal physical state.

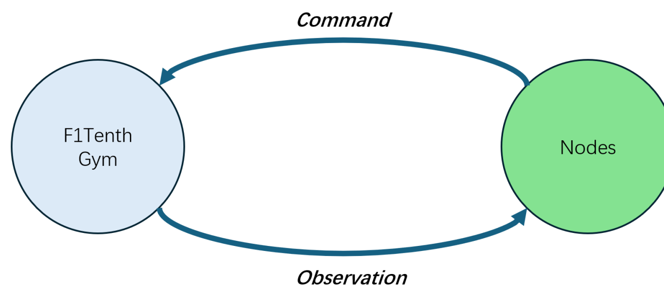


Figure 3.5: F1Tenth Gym command-observation model.

The input commands can be divided into three categories: initialization commands, reset commands, and control commands. The initialization command defines the initial state of the physical world through a series of parameters, such as:

- Surface friction coefficient.
- Vehicle shape.
- Cornering stiffness coefficient.
- Distance from center of gravity to axle.
- Height of center of gravity.
- Total mass of the vehicle.
- Moment of inertial of the entire vehicle about the z axis.
- Steering angle constraint.
- Steering velocity constraint.
- Switching velocity constraint.
- Longitudinal acceleration, velocity constraint.

The reset command is used to reset the vehicle's state, i.e., to re-set the vehicle's position, orientation, velocity, acceleration, etc. The control command is responsible for transmitting the control instructions output by the autonomous driving algorithm, i.e., the expected velocity and steering angle.

After each command is input, F1Tenth Gym will return a series of information, which is called observation. Observation contains information describing the environment and vehicle state, including the vehicle's position, direction, speed, and sensor (such as lidar) observations for the current step.

3.3.2 Single-Track Kinematic Model

Although this project does not directly implement the core of the physical simulator, it is still necessary to understand the general principle of its internal simulation of vehicle motion, so that researchers studying vehicle-to-vehicle communication can

understand the physical simulation of vehicles , e.g., based on what kind of model, in order to be able to modify it according to their own needs.

The single-track model allows a physically plausible description of the driving behavior of vehicles without major modeling and parameterization effort [11]. It is a simple and effective model for estimating vehicle motion. Its usage can be illustrated by Figure 3.6.

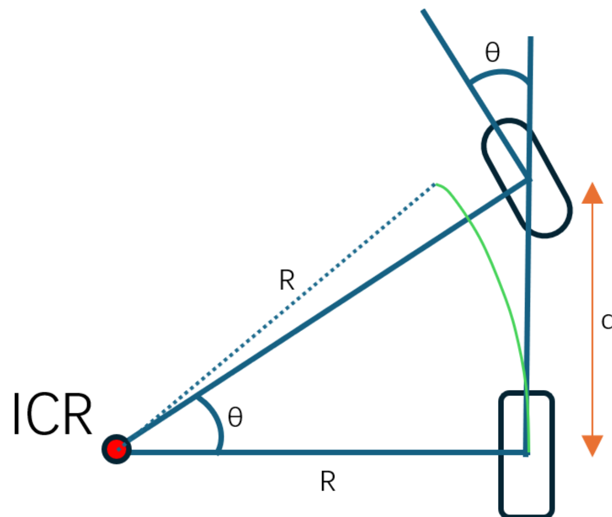


Figure 3.6: Demonstration of single-track kinematic model.

$$R = \frac{d}{\tan\theta} \tag{3.1}$$

We can imagine the driving vehicle as a bicycle, with a wheelbase of d between the front and rear wheels. When the front wheel is turned at an angle of θ to the left, according to the single-track kinematic model, the rear wheel will move along the green arc with a radius of R centered at the instant center of rotation (ICR). The radius R can be calculated using Equation (3.1), and then we can calculate where the vehicle will move to and what its pose will be in the next time step, given the current vehicle motion state.

3.3.3 Maps

In ROS2, maps are represented using an occupancy grid, the F1Tenth Gym simulator also uses the same approach.

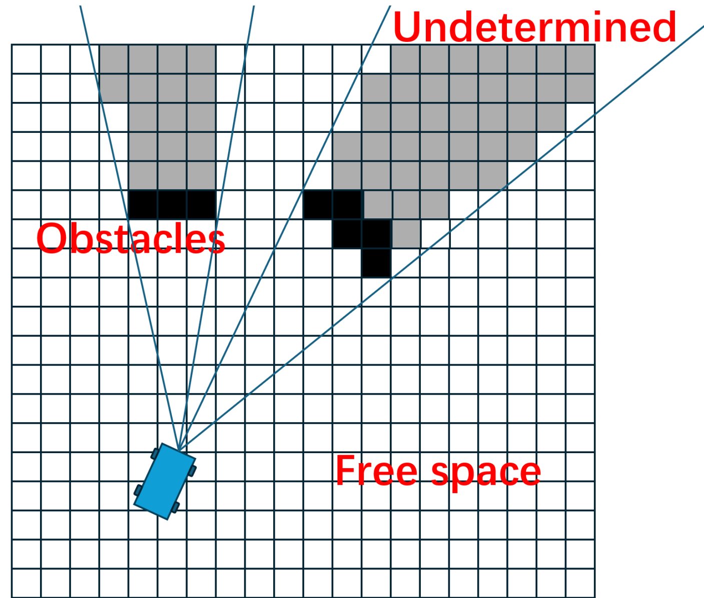


Figure 3.7: Occupancy grid.

The occupancy grid is a type of grid map that represents occupancy. As the Figure 3.7 shows, an occupancy grid is composed of a grid of cells, similar to pixels in a photo. Each cell is represented by an 8-bit integer value between 0 and 100. This value represents the degree of confidence that the cell is occupied by an obstacle. 0 represents free space, and 100 represents 100% certainty that the cell is occupied. Similar to maps, the occupancy grid also has the concept of scale. The resolution represents the length of each grid cell in the actual physical space.

The F1Tenth Gym receives an 8-bit grayscale image as input and reads the corresponding map description YAML file to create the map environment of the physical world. The YAML file contains the map description information, including the resolution, map origin location, occupancy threshold, etc. The F1Tenth Gym will place obstacles in the corresponding positions in the simulation space according to the settings.

Therefore, when creating the map, it is necessary to pre-draw the corresponding format and specification images in software like Photoshop.

3.4 RViz2

Visualization is an important part of the physical simulation process, as it allows researchers studying vehicular communication to understand the behavior of vehicles in the physical environment. The RViz2 provided by ROS2 can meet this requirement. If F1Tenth Gym is considered as the backend of the physical simulator, then RViz2 can be understood as the frontend.

RViz2 is a powerful visualization tool that runs as a ROS2 node [7]. RViz2 constantly listens to all ROS2 topics and can visualize various types of messages, such as maps, markers, laser scans, robot descriptions, etc. After receiving a message on a subscribed topic, RViz will draw these contents based on certain rules.

Figure 3.8: UI of RViz2.

In this joint simulation environment, RViz2 is used to visualize the physical world. The following key visualization message types are utilized:

1. Occupancy Grid: This includes map information, such as an occupancy grid map and its associated metadata (e.g., resolution, origin, size, scale). These messages are published to the /map topic and listened to by RViz2 for display.

2. Robot Model: This contains the model information of the vehicle. It is important to note that the robot model only includes the geometric information and

color, and does not include the physical properties. RViz2 is a visualization tool and does not perform physical simulation. The robot model is defined in a file with the .xacro extension, which follows the XML standard. The vehicle defined in the .xacro file starts with the three-dimensional geometric structure of the base link, and then connects other 3D geometric structures through a series of joints and links. For example, one could define a cuboid as the abstract representation of the vehicle's body, and then create four cylindrical abstractions for the wheels and connect them to the body using links. This simple modeling approach, although losing the details of the internal structure of the physical object, greatly simplifies the workload of vehicle communication researchers, as they can quickly define an arbitrary number of vehicles using just a hundred lines of XML, providing convenient and easy extensibility.

3. Laser Scan: This includes the lidar detection information of the vehicle. RViz2 subscribes to this message and renders it, allowing researchers to quickly inspect the "vision" of the vehicle and simplify the debugging process.

4. Marker: Marker is an important visualization message in RViz2, which allows the publishing ROS2 node to act as a paintbrush and specify points, lines, spheres, and other geometric shapes to be drawn in RViz2. Markers are very useful in visualizing the vehicle's trajectory and algorithm debugging, as researchers can manually specify various detection and planning information to be drawn in RViz2.

5. Other Information: RViz2 supports many other visualization message types, such as camera information to depict the scene seen by the on-board cameras. Researchers can highly customize the types of visualization information according to their needs.

3.5 Gym Bridge

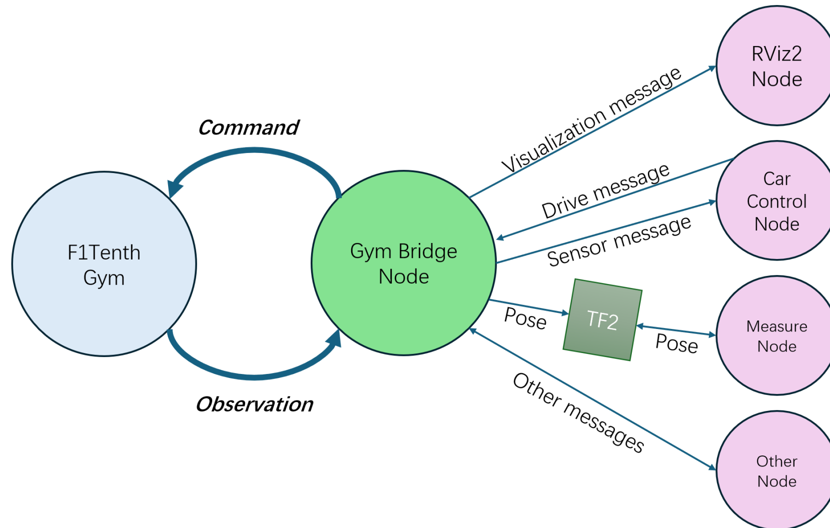


Figure 3.9: Gym bridge.

As a bridge connecting the control nodes, physics simulator, RViz2, and various auxiliary nodes, the gym bridge plays a crucial role. Its functionality can be represented by the diagram in Figure 3.9.

The gym bridge is responsible for handling the interactions with the backend of the gym, collecting vehicle control information, passing it to the F1Tenth Gym for processing, and then distributing the resulting observations to various nodes. For example, it packages the vehicle pose observations returned from the F1Tenth Gym into ROS2 messages and sends them to the RViz2 node for visualization. It also packages the sensor information returned from the F1Tenth Gym and passes it back to the car control node.

The term “TF2” refers to the ROS2 module specifically designed for maintaining and calculating rigid body transformations. ROS2 can automatically maintain the transformation relationships between the various coordinate frames. TF2 also adopts a broadcast-and-query mechanism, where the control node of a physical module can broadcast its current pose to the ROS2 TF2 space. ROS2 will record this pose, and if other modules query the pose information based on the coordinate frame of this physical module, ROS2 can automatically perform the calculations and return the

correct value. More details on the use of TF2 will be provided in the subsequent sections.

3.6 Car Control

Researchers in vehicular communication may not need to engage with the F1Tenth Gym backend and Gym bridge operations. However, they will frequently encounter the content of this section, as it is essential for defining the autonomous driving control behavior of the vehicle. To facilitate usage by researchers, four autonomous driving control algorithms have been pre-written. Therefore, this section will focus on the relevant aspects of the vehicle control module.

3.6.1 Convention

Some conventions need to be established to facilitate researchers in developing vehicle control nodes. First, it is essential to clarify the structure of a ROS2 node. A ROS2 node generally consists of a class that inherits from the Node superclass. The Node superclass provides a series of methods to implement common functionalities, such as publishing or subscribing to topics, creating or reading node parameters, binding callback functions, console output, and more. The specific functionality class (like the vehicle control node that researchers will develop) primarily focuses on implementing the control algorithms.

The first convention pertains to namespaces. In ROS2, each node has its own namespace, which serves to distinguish between nodes, variables, and other elements that may share the same name. In this joint simulation environment, which supports the simulation of multiple vehicles simultaneously, there will inevitably be situations where several vehicles run the same node. Without namespace differentiation, it would be challenging for other nodes and the developers themselves to identify which vehicle has sent a particular message. In this simulator, the namespace for each vehicle is uniformly represented as “carx”, where “x” is a non-negative integer starting from 0. The simulator by default creates corresponding namespaces starting from 0. For instance, if there are three vehicles in the environment, the simulator will default to creating the namespaces car0, car1, and car2.

Next is the convention regarding topic names, and topic names must start with slash. Some common topic names are outlined below:

- /sim_control (std_msgs/String): Simulation control topic, all nodes subscribe to this topic to act in phase.
- /car0/drive (ackermann_msgs/AckermannDriveStamped): Drive command topic subscribed by gym bridge for car0's control, will be transmitted into F1Tenth Gym backend.
- /car1/odom (nav_msgs/Odometry): Odometry information of car1 published by gym bridge.
- /car2/scan (sensor_msgs/LaserScan): Scan information of car2 published by gym bridge.
- /car1/goalpoint_marker (visualization_msgs/Marker): Current goalpoint of car1, subscribed by RViz2 for visualization purpose.

It is important to note that topics for different vehicles must be distinguished by adding the corresponding vehicle's namespace as a prefix. For instance, each vehicle requires a separate topic for controlling that specific vehicle, such as the /drive topic. In ROS2, the namespace is automatically prefixed with a slash. Therefore, if the namespace is specified as "car1" when launching the node, the final namespace obtained by the node will be "/car1". By appending the namespace to the topic name, a topic tailored for a specific vehicle can be constructed.

Lastly, there is the convention for frame names. The simulator environment contains numerous frames, such as the map frame, car frame, etc., each representing a three-dimensional coordinate system. When developers utilize TF2, they need to specify the frame name they are querying. Here are some common frame names:

- car1/base_link: Chassis frame of car1.
- car0/laser: Laser frame of car0.
- map: Global map frame.

In contrast to topic names, frame names are not allowed to begin with a slash. This distinction helps differentiate whether a name is a topic name or a frame name. Since ROS2 automatically adds a slash prefix to namespaces, it is important to manually remove this prefix when constructing frame names associated with each vehicle.

3.6.2 Autonomous Driving Algorithms

The selection of autonomous driving algorithms is diverse, and this simulator integrates four preset algorithms: wall following, gap following, pure pursuit, and rapidly exploring random tree. Researchers in vehicular communication can easily invoke these pre-written algorithms, as well as modify or write their own autonomous driving rules as needed. Due to space limitations, this section will focus on introducing the most practical and extensible algorithm for vehicular communication scenario simulations: pure pursuit.

Pure pursuit is a highly effective map-based autonomous driving algorithm [12]. Its core concept involves a series of predefined waypoints that the vehicle follows based on its positional information. The vehicle selects the most suitable waypoint as the goal point and then calculates the necessary steering angle and speed, based on its current position, to direct the vehicle toward the goal point.

This requirement can be broken down into the following subproblems: locating the vehicle's position (x_{car}, y_{car}) corresponding to the map frame, determining the optimal waypoint from a given series of waypoints $(x_{goal1}, y_{goal1}), (x_{goal2}, y_{goal2}), (x_{goal3}, y_{goal3})$, and so on at the current moment, and calculating the vehicle's steering angle θ after selecting the optimal waypoint.

Localization is the first issue that needs to be addressed. In the real world, localization is typically achieved using methods such as GNSS, UWB, or particle filters. However, in the simulator, localization becomes significantly simpler because F1Tenth Gym can accurately return the vehicle's position and orientation information. Thus, in the simulator, the vehicle's position can be packaged as odometry information by the gym bridge and sent back to each vehicle. The position information obtained in this manner is absolutely accurate, eliminating the impact of localization errors on subsequent communication effects observations.

The waypoints to be tracked are a series of predefined points which will be loaded into the pure pursuit algorithm for optimal local waypoint selection and tracking. The optimal local waypoint is usually set as a point within a certain distance ahead of the vehicle. This distance is commonly referred to the look-ahead distance in autonomous driving algorithms, usually denoted by L. However, in practice, it is impossible to ensure that a local trackpoint always lies on the forward semicircle with radius L. Therefore, the semicircle needs to be slightly expanded by a width of $2 \times a$ to a semicircular ring with an inner radius of $L - a$ and an outer radius of $L + a$.

Inside the semicircular ring, the nearest waypoint is selected as the best local tracking waypoint.

The choice of L requires careful consideration. As illustrated in Figure 3.10, if L is set too small, the vehicle will use a larger steering angle to correct its course when it deviates from the predefined route, leading to highly unstable trajectories at high speeds, with the vehicle swaying left and right in an S-shape along the intended route. However, this strategy can more accurately track the predefined driving route during low-speed driving.

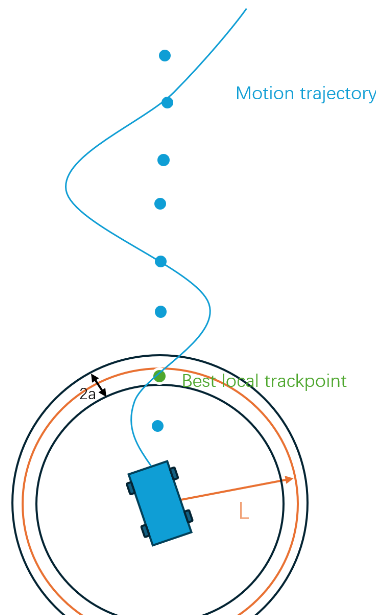


Figure 3.10: High-speed motion trajectory using small L.

Conversely, if L is set too large, as shown in Figure 3.11, the vehicle will not rush to return to the intended trajectory when it deviates but will instead make lower-amplitude adjustments using a larger correction space. This will result in temporary deviations from the intended route, but the trajectory will become smoother. This approach is particularly effective at high speeds. The same phenomenon applies to human driving. For example, when changing lanes on a highway, a longer distance is typically required compared to city streets, as sharply turning the steering wheel at high speeds can lead to highly unstable driving conditions and increase the risk of accidents.

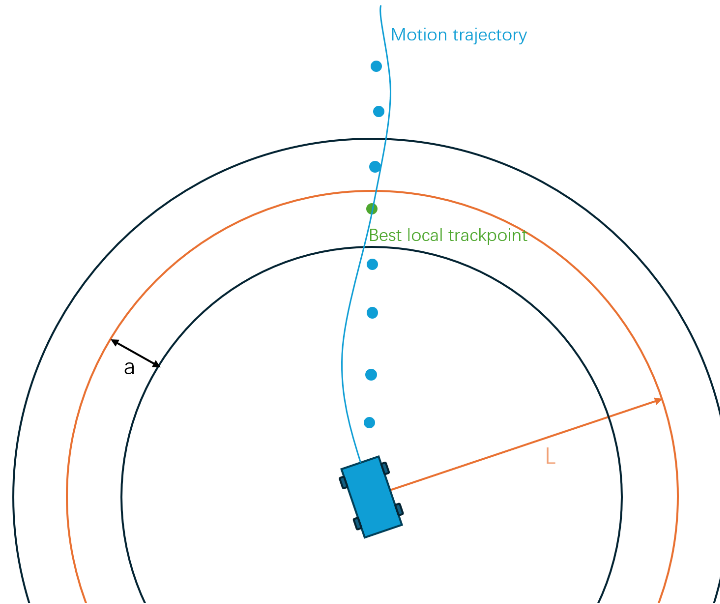


Figure 3.11: High-speed motion trajectory using large L.

Once the optimal local tracking waypoint is determined, the next step is to calculate the required steering angle based on the vehicle's current position. In the pure pursuit phase, the concept employed is known as "follow the arc", which suggests that the vehicle's trajectory will follow an arc. One end of this arc is at the vehicle's base link, while the other end is the target point that needs to be tracked. By leveraging the single-track kinematic model, the center of the circle corresponding to the arc is referred to as the instant center of rotation. Intuitively, if the arc's curvature is large, the vehicle will require a greater steering angle; conversely, if the curvature is smaller, a smaller steering angle will be needed. Therefore, calculating the steering angle can be reduced to determining the curvature of the circle that the trajectory arc lies on. Since curvature is the inverse of the radius, the problem further reduces to finding the radius of the circle on which the trajectory arc lies. To simplify the problem, the base link frame of the vehicle is used as the coordinate system, making the vehicle's position the origin, and the coordinates of the optimal local track point (x, y). The geometric relationship depicted in Figure 3.12 can then be established.

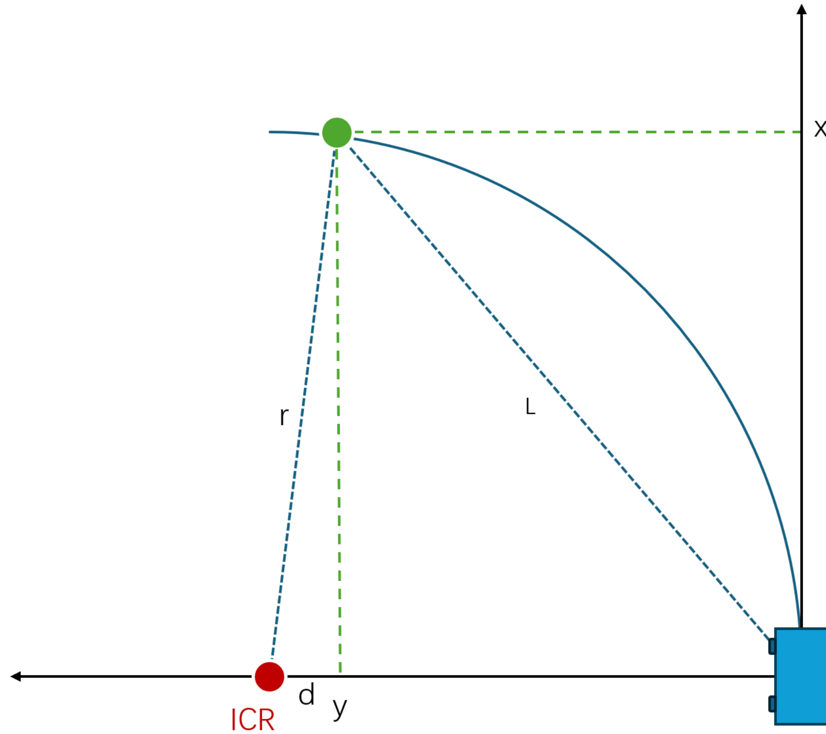


Figure 3.12: Geometric relationship in the follow the arc.

Here, L represents the look-ahead distance, as we always select the local tracking waypoint that is approximately at the look-ahead distance. The calculation for the curvature γ is given by equations below.

$$y + d = r \tag{3.2}$$

$$d^2 + r^2 = x^2 \tag{3.3}$$

$$x^2 + y^2 = L^2 \tag{3.4}$$

$$r = \frac{L^2}{2 \times y} \tag{3.5}$$

$$\gamma = \frac{1}{r} = \frac{2 \times y}{L^2} \tag{3.6}$$

Once the curvature γ is obtained, the mapping between curvature and steering angle can be adjusted using the P control in the PID controller.

Since the global tracking waypoints pre-recorded by the developer and the real-

time position and orientation information obtained by the vehicle are both based on the map frame, the calculation method mentioned earlier, which is based on the base link frame, cannot be directly applied. This is where rigid body transformation comes into play, and it is precisely where the TF2 module proves useful. Whenever pure pursuit is required, TF2 can be queried to obtain the coordinates of the local tracking waypoint in the map frame, transformed into the current base link frame coordinates. This allows the aforementioned calculation method to be applied directly.

3.7 Auxiliary Moudules

Some auxiliary modules play a crucial role in the simulator. In this section, several of the more commonly used ones will be introduced.

3.7.1 Waypoint Recorder and Displayer

The waypoint recorder and displayer modules handle the task of recording and visualization. They are responsible for recording the vehicle's trajectory and constructing Marker messages to be displayed in RViz2. Before conducting joint simulations, the vehicle's driving path must be defined in advance. This path can be obtained through two main methods: the first is by manually driving the vehicle in the simulator and recording real-time driving parameters, such as position, speed, and orientation. The second method is by manually defining the vehicle's driving parameters. For simulations that require high precision, the method of predefining the vehicle's driving parameters is typically used. For example, in vehicle collision tests, the position, speed, and orientation of each waypoint on the collision path can be predefined. Once the simulation begins, the vehicle will follow this information to proceed. Since autonomous driving demands high performance, this information is stored in a CSV file for the autonomous driving algorithm to read. Compared to other formats, CSV offers better read and write performance.

3.7.2 Measurement Module

The measurement module is responsible for recording various types of information throughout the entire simulation process. It continuously measures and records each vehicle's position, speed, inter-vehicle distance, time to collision, collision occurrences,

and other relevant data. This is extremely useful for vehicular communication researchers to closely analyze the simulation process after its completion. The module constantly listens to the information broadcast by the gym bridge, calculates the aforementioned parameters internally, and writes them into a log file (also in CSV format).

It is worth mentioning the information regarding time to collision (TTC). This represents the estimated time until a collision occurs between two vehicles. It is calculated by dividing the distance between the two vehicles by the sum of their speeds, similar to the classic "meeting point" problem from elementary mathematics. However, since vehicles are not always moving directly towards each other, the velocity component of one vehicle in the direction of the other must be used for the calculation. As odometry information only includes the vehicle's longitudinal velocity, the velocity in the corresponding direction must be obtained by multiplying the velocity by the cosine of the angle in that direction. Determining this angle involves several steps of rigid body transformation calculations. Therefore, the measurement module maintains a transformation matrix specifically designed to store the transformation relationships between the frames of different vehicles. This matrix is continuously updated in real-time and is referenced when calculating the time to collision.

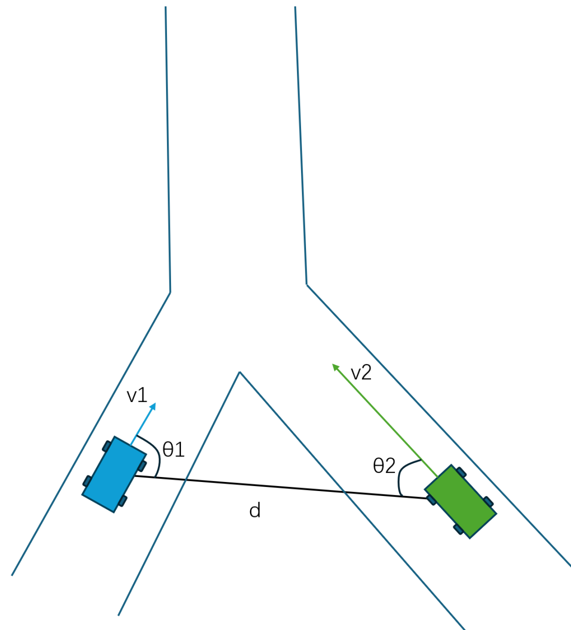


Figure 3.13: Calculation of time to collision.

$$TTC = \frac{d}{v_1 \times \cos\theta_1 + v_2 \times \cos\theta_2} \tag{3.7}$$

3.8 Communication Simulation Design

The following sections will introduce the communication simulation portion. The basic design concept of the communication simulation involves abstracting the physical simulator and each vehicle as communication nodes and placing these nodes within the environment to be simulated in order to observe their communication performance. Figure 3.14 illustrates the topology used to implement this functionality.

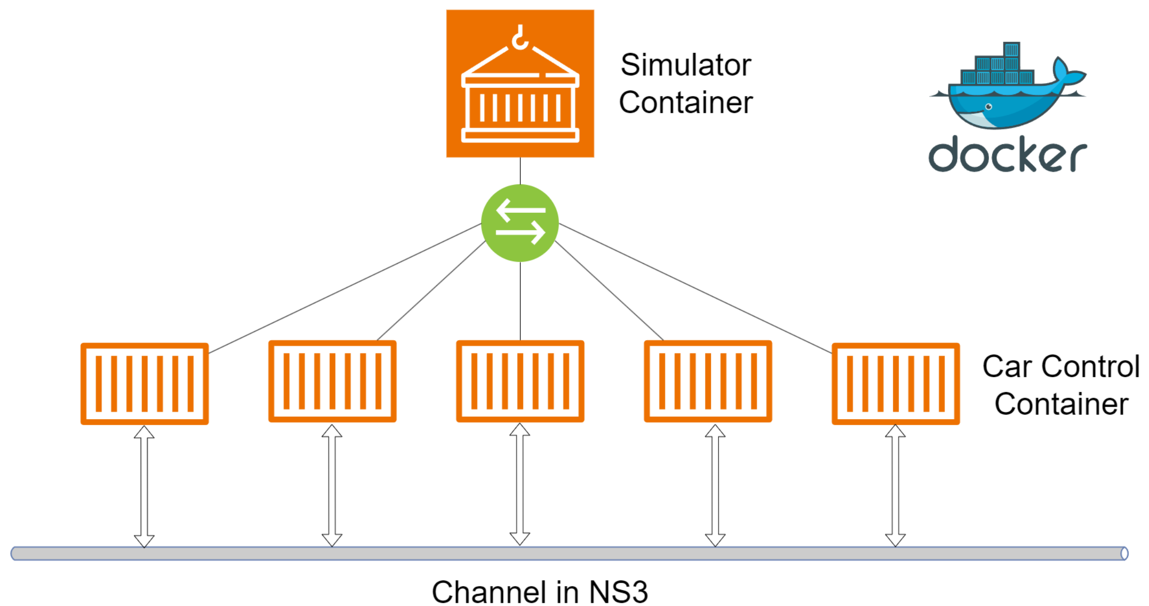


Figure 3.14: Communication simulation topology.

Docker containers [13] are utilized to containerize the functionality of each node, as docker containers allow for convenient and rapid batch deployment while offering robust capabilities. The simulator container plays the role of the physical world, where any interaction between the vehicles and the physical world, such as acceleration, braking, and steering, should not experience any delay. Therefore, the simulator container is directly connected to each vehicle control container through a switch. The vehicle control container, as the name suggests, hosts the node responsible for controlling each vehicle's behavior. It receives information from the simulator container,

such as sensor data and localization information, and then uses the autonomous driving algorithm deployed within the container to decide on the appropriate control commands, which are then sent back to the simulator container.

Inter-vehicle communication is a key focus of the simulation. Each vehicle control container is connected to a channel created by Network Simulator 3 (NS3) to simulate the communication environment. The simplest example shown here involves connecting all vehicles to the same bus. On this bus, vehicular communication researchers can manually set various parameters, such as channel delay, packet loss rate, link speed, and other attributes.

3.9 Communication in Docker and ROS2

By default, docker automatically configures network devices for each container upon creation. Any newly created container, unless manually specified, will be connected to a virtual bridge device that is configured during docker installation, typically named "docker0". The virtual bridge device functions similarly to a switch, operating at the data link layer, allowing any network interface card connected to this virtual bridge to communicate directly with each other. This setup enables natural communication between the simulator container and the car control containers without the need for manual configuration. While the automatically configured network is convenient, it lacks customization options. Therefore, in addition to the default bridge mode, docker also offers "host", "container", and "none" modes.

Docker uses Linux's namespaces technology for resource isolation, such as PID namespace for process isolation, mount namespace for filesystem isolation, and network namespace for network isolation. A network namespace provides an isolated network environment, including network interfaces, routing, and iptables rules, all of which are isolated from other network namespaces. A docker container is typically allocated its own independent network namespace.

If the host mode is used when starting a container, the container does not get its own independent network namespace but instead shares the network namespace with the host machine. The container does not create its own network interface card or configure its own IP, but instead uses the host's IP and ports. The container mode specifies that the newly created container shares a network namespace with an existing container rather than with the host machine. The newly created container does not create its own network interface card or configure its own IP, but instead

shares the IP address and port range with the specified container. Despite sharing the network, the two containers remain isolated in other aspects, such as the filesystem and process list. The processes in the two containers can communicate through the loop back network interface device. In none mode, the docker container has its own network namespace but does not have any network configuration. This means that the docker container has no network interface card, IP address, or routing information. Developers must manually add network interfaces and configure IP addresses for the docker container.

ROS2 is developed based on the Data Distribution Service (DDS) and Real-time Publish-Subscribe (RTPS) middleware [14]. DDS is a decentralized distributed communication standard, while RTPS is a protocol that provides services in compliance with the DDS standard. DDS is information-centric and resembles a broadcast model where all communication entities share an abstract data bus, allowing any entity to publish or read messages on this bus. However, DDS goes a step further by incorporating multiple parallel communication pathways, enabling each entity to focus only on messages of interest while automatically ignoring irrelevant ones. Compared to traditional server-client models and broker-based models, DDS reduces system complexity, especially in intricate distributed systems like autonomous driving systems. These systems involve numerous complex logics. If applications had to manually manage communication entities as in traditional communication application development, the development workload would become immense as the system evolves, with increasing in-vehicle nodes and vehicle counts. This would likely lead to bugs and reduced system robustness. However, with a unified data bus, even when new communication roles are added, the communication model remains simple.

RTPS enables best-effort publish-subscribe communication through unreliable transports like UDP. In RTPS, communication entities are classified as either publishers or subscribers. Publishers can publish any message to the global data space, while subscribers can receive messages from any topics they have subscribed to. This functionality closely aligns with the services provided by ROS2. The RTPS protocol is responsible for core services such as dynamic discovery and message delivery.

The communication scope of RTPS follows these characteristics: nodes within the same domain ID and subnet can freely communicate with each other. For example, any nodes running on hosts within the 192.168.1.0/24 subnet, as long as they are configured with the same domain ID, can discover and communicate with each other, whereas hosts across subnets cannot. However, there is an exception: if a host has

two network interfaces that belong to different subnets, nodes within both subnets can communicate with the nodes on that host. This feature is crucial as it serves as the foundation for implementing joint simulation functionalities.

3.10 NS3 and Tap Bridge

Network Simulator 3 is a specialized communication network simulator capable of simulating the entire process of network data packet generation and transmission across links [15]. To achieve high-quality and convenient network communication simulations, NS3 abstracts all network communication objects into five categories:

- Node: A node is an abstraction of a network node, typically representing a physical device in the real world, such as a computer or router. Researchers can add the necessary functions to the node according to their requirements.
- Application: Application represents the actual users of the network—the user applications. Researchers can create a series of applications to simulate the generation, transmission, and reception of data packets. Applications run within nodes.
- Channel: Channel is an abstraction that represents the physical medium of communication and the communication rules applied within it. For example, a CSMA channel will simulate a bus topology and apply carrier sensing within it. Nodes can be connected to the channel.
- Net device: Net device is an abstraction of the network peripherals in a computer such as a network card. Each net device has corresponding hardware attributes, such as a MAC address, and an appropriate driver. A node can have multiple net devices installed.
- Topology Helpers: These are used to facilitate the rapid construction of network topology. In network configuration, researchers often need to manually add net devices to nodes, connect them to channels, assign IP addresses, MAC addresses, and so on. To save researchers from having to perform these repetitive tasks each time, NS3 uses topology helpers to manage these operations centrally. This allows researchers to complete the entire process with just a few lines of code.

When simulating vehicular communication in NS3, vehicles can be abstracted as nodes, with autonomous driving applications on the vehicles, such as ROS2, serving as applications. However, NS3 cannot directly deploy ROS2 to act as an application and generate corresponding data packets. Previous researchers in vehicular communication typically had to model the behavior of vehicular communication applications first, and then define the behavior of abstract applications in NS3, manually simulating the information sent and received by vehicles. Thus, a communication bridge is needed to connect ROS2 and NS3.

Fortunately, NS3 provides the tap bridge module. By leveraging the tap device in Linux, it forwards the actual data packets generated by Linux applications to the tap device, which are then read into NS3. In this simulation setup, each vehicle's autonomous driving program runs in an independent ROS2 environment, with each ROS2 environment running within its own docker container. Therefore, at the abstraction level in NS3, each docker container represents a node, and the network card within the container needs to be connected to the net device of the NS3 node via the tap device, which then pushes packets into the channel for simulation. The structure diagram illustrating the connection between containers and nodes in NS3 is shown in Figure 3.15.

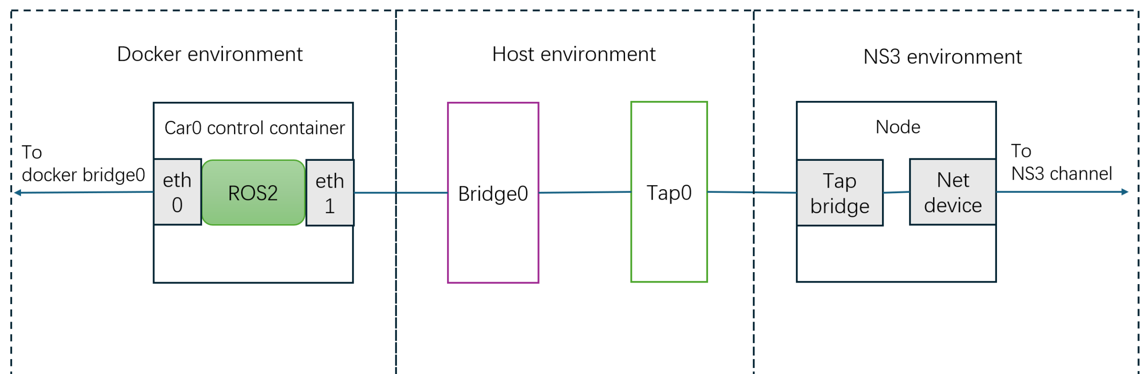


Figure 3.15: Connection structure between container and node.

First, the corresponding network device needs to be created in the docker container. As previously mentioned, by default, docker automatically creates a network device for the container, named eth0, which is connected to the default docker bridge, docker0, responsible for communication between containers. Additionally, another network device, eth1, needs to be created within the container to interface with NS3. Tap bridge has two entries: one connects to the tap device on the host machine

where NS3 is running, and the other connects to the net device in the NS3 node. The network device eth1 within the container cannot directly communicate with the tap device on the host machine, so a Linux bridge must be constructed to bridge them. As a result, the traffic generated within the docker container (data packets sent by ROS2) will be sent via eth1 and a virtual eth link to the bridge on the host machine, then transferred to NS3 via the tap device and tap bridge, and ultimately delivered to another container.

To measure the impact of various intermediate components on communication latency, I wrote two ROS2 nodes. The talker node publishes a ROS2 message containing a timestamp of when it was sent, and the listener node calculates the time difference upon receiving the message. At the ROS2 application layer, the observed delay fluctuations are only at the nanosecond level, far below 1 ms. The measurements were performed on a computer with an Intel® Core™ i7-13700HX processor running at a clock frequency of 3.7GHz, and the memory read-write speed is 4800MT/s.

ROS2 also serves as the synchronization benchmark in this simulation platform. A key feature of the platform is the use of real autonomous driving applications to make driving decisions and send/receive communication data packets. Therefore, both the physical simulator and the communication simulator aim to synchronize with ROS2. ROS2 consists of multiple nodes, each with a different frequency depending on its function, and it automatically controls synchronization between these nodes. For F1Tenth Gym and NS3, as long as their simulation frequency is higher than the frequency at which ROS2 handles physical control and communication messages, it ensures that communication between ROS2 and F1Tenth Gym/NS3 is not bottlenecked by low simulation frequency, which could otherwise hinder simulation accuracy.

3.11 Environment Configuration

After completing the theoretical design of the architecture and process described above, the corresponding simulation environment needs to be configured on ROS2, NS3, docker containers, and the host machine.

3.11.1 ROS2

For ROS2 running in docker containers, it can autonomously send and receive data through all network interfaces available within the containers. The only requirement is to configure the same domain ID for ROS2 across all containers and use appropriate topic names for publishing or receiving messages.

3.11.2 NS3

The configuration of NS3 should be tailored to the specific simulation requirements. The process follows the general principles of NS3 simulation script writing, which include creating the required number of nodes, building the topology, defining the channel type and setting channel properties, installing net devices on the corresponding nodes, and finally connecting the host machine's tap devices via the tap bridge. The network simulation is highly flexible, vehicular communication researchers can construct some virtual application nodes alongside the nodes using real ROS2 to increase the complexity of the environment, thereby observing the behavior of vehicles during the network or physical simulation process.

3.11.3 Docker Containers

Configuring network devices in docker containers requires certain tricks, as docker does not provide a direct interface for operating the container's network namespace. Therefore, developers need to manually locate the network namespace corresponding to the docker container for configuration. The Linux network namespace is directly associated with the PID. By using the docker inspect command along with the container name, the PID of the specified container can be found. Once the PID is obtained, the ip netns exec command can be used to directly configure the network devices in the docker container.

The first step is to configure the new network interface, adding eth1 to the docker container and assigning it an IP address and MAC address. It is important to ensure that the subnet of eth1's IP address does not match the automatically assigned docker network address. This distinction is necessary to differentiate whether the data packets are transmitted through the docker bridge or the NS3 environment (tap device). As mentioned earlier, the interaction between vehicles and the physical world should not be influenced by the network environment, meaning communication between the

vehicle control container and the physical simulator container should occur directly via the docker bridge, bypassing NS3. In contrast, data packets for communication between vehicle control containers should be transmitted through NS3. This raises another issue: how to prevent data packets that should be transmitted through NS3 from reaching another vehicle control container via the docker bridge? This can be achieved by utilizing the Linux iptables firewall.

In this simulator design, eth0 of the vehicle control container is responsible for communication with the simulator container, while eth1 handles communication with the NS3 environment. Since eth0 and eth1 reside in different subnets, setting up iptables rules on each vehicle control node to drop all packets from the eth0 subnet except those originating from the physical simulator container effectively prevents vehicle control nodes from "cheating" and achieving latency-free transmission via the docker bridge. This ensures that all data packets received by one vehicle control container from another vehicle control container are transmitted through NS3.

3.11.4 Host Machine

The environment setup on the host machine involves creating the corresponding number of Linux bridge devices and tap devices, as well as creating the necessary virtual eth pairs to connect eth1 within the container to the bridge on the host machine.

In this simulator, the names of the bridge and tap devices are br-x and tap-x, respectively, where x represents the vehicle control container's number. This number corresponds to the identifier in ROS2, starting from 0 for the first vehicle, and incrementing by one for each additional vehicle.

3.12 Process and Automation

In summary, the operational principles of the various modules within the physical simulation and communication simulation have been described in detail. However, the steps and details involved are numerous and complex. To facilitate ease of use for researchers in vehicular communication, this simulator also provides a series of shell automation scripts for quickly creating the simulation environment. This section introduces the general usage process of the autonomous driving physical-communication joint simulator and explains the functions of each automation script.

3.12.1 Preparation Phase

Vehicular communication researchers must first determine the scenario to be simulated, including the required physical and network environments. This involves creating the corresponding physical maps (occupancy grid), setting environment parameters (such as ground friction coefficient), considering the number of vehicles and the attributes each vehicle should possess (such as speed, acceleration, maximum steering angle), designing communication protocols, network topology, link and node attributes, and so on.

3.12.2 Defining Driving Rules

Researchers need to either write their own or choose an appropriate autonomous driving algorithm from those that have already been built into the simulator. If selecting the pure pursuit algorithm for precise simulation, it is necessary to define the waypoints that the vehicle needs to track during its movement, along with associated speed and direction information.

3.12.3 NS3 Script Writing

Implement designed network environment in NS3. At a minimum, nodes corresponding to the number of vehicles should be created, and additional virtual nodes can be introduced to make the channel congested and challenging, thus testing whether the designed communication protocol can ensure the safety of autonomous driving.

3.12.4 Deployment Phase

The physical simulator and autonomous driving algorithms are deployed into the corresponding docker containers. The appropriate network interfaces, bridges, tap devices, veth pairs, and iptables rules are configured.

3.12.5 Running the Simulation

The physical simulator and NS3 are launched, and the autonomous driving algorithms are executed to observe the behavior of the vehicles in the physical world. After the simulation ends, the simulation logs are reviewed for analysis.

3.12.6 Improvement

Based on the simulation results, improvements can be made to the autonomous driving algorithms, communication protocol designs, and other aspects.

3.12.7 Automation Scripts

The automation scripts can greatly simplify the configuration tasks during the deployment phase. The provided scripts include:

- docke_run: used for batch launching of simulator containers and vehicle control containers.
- build_all: used for building all ROS2 nodes within the containers in a single operation.
- network_setup: used for batch configuration of the network environment on docker containers and the host machine.
- launch_all_control_nodes: used for batch launching of all vehicle control nodes.
- teardown: used to delete all newly created docker containers and network configurations on the host machine, restoring the host machine's network to its original state.

Chapter 4

Experiments

In this chapter, an example of a vehicle collision simulation will be presented to demonstrate the impact of channel delay on emergency braking in autonomous driving. The intention here is to present a use case example for this simulation platform, allowing users of the platform to modify any setting based on this example and conduct their own experiments.

Consider the following scenario: two vehicles, car 0 and car 1, are approaching each other at a speed of 10 meters per second. They must stop under certain conditions to avoid a collision. Unfortunately, due to software version incompatibility or a positioning system failure, car 1 is unaware that another vehicle, car 0, is rapidly approaching. Fortunately, car 1 is equipped with an emergency communication module that will immediately execute an emergency braking task upon receiving an emergency brake signal. Car 0, being more advanced, can communicate in real-time with the infrastructure, obtaining information about the position, speed, and direction of travel for both itself and car 1. Additionally, car 0 is equipped with a collision avoidance module based on time to collision, which will send an emergency brake signal to the other vehicle when it detects an imminent collision. However, there is a delay in information transmission between car 0 and car 1, leaving the fate of the two vehicles in the hands of the communication protocol. A good protocol should prioritize the timely transmission of emergency messages with minimal delay.

The researchers want to determine whether, under these conditions, if a CSMA channel is used and car 0 transmits an emergency brake signal when the time to collision is less than 0.7 second, what is the maximum channel delay that can prevent a collision between the two vehicles?

To investigate this, the researchers constructed the following map. The two vehi-

cles are placed at opposite ends of the map, with several global track points positioned between them, each maintaining a speed of 10 m/s.

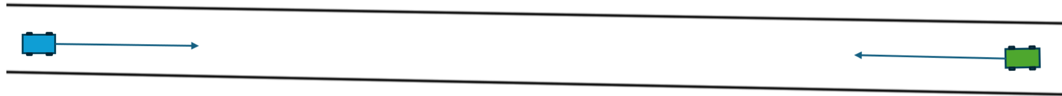


Figure 4.1: Experiment map.

Next, configure car 0 as the sender of the emergency brake signal and modify the pure pursuit node so that both vehicles immediately execute the braking task upon receiving the emergency brake message. Additionally, car 0 is responsible for detecting the time to collision and sending the emergency brake message accordingly.

In NS3, create a CSMA channel and set different channel delays. Then configure the host environment according to the established process.

```
NodeContainer nodes;
nodes.Create(n_car);

CsmaHelper csma;
// csma.SetChannelAttribute("DataRate", DataRateValue(DataRate("1000Mbps")));
csma.SetChannelAttribute("Delay", TimeValue(MilliSeconds(50)));

NetDeviceContainer devices = csma.Install(nodes);

TapBridgeHelper tapBridge;
tapBridge.SetAttribute("Mode", StringValue("UseBridge"));

for (int i = 0; i < n_car; i++)
{
    std::string device_name = "tap-" + std::to_string(i);
    tapBridge.SetAttribute("DeviceName", StringValue(device_name));
    tapBridge.Install(nodes.Get(i), devices.Get(i));
}

NS_LOG_UNCOND("Channel is ready, start your ROS2 simulation!");

Simulator::Stop(Hours(3.));
Simulator::Run();
Simulator::Destroy();
```

Figure 4.2: NS3 configuration, channel attributes can be selected flexibly.

Once setting is ready, start the simulation.

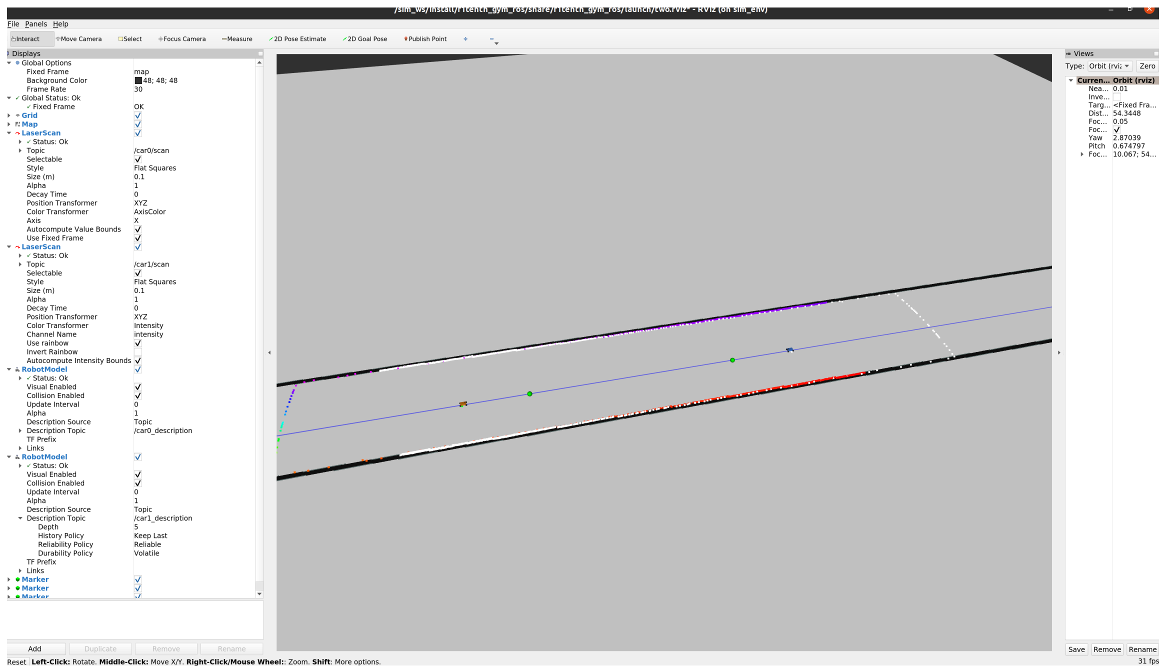
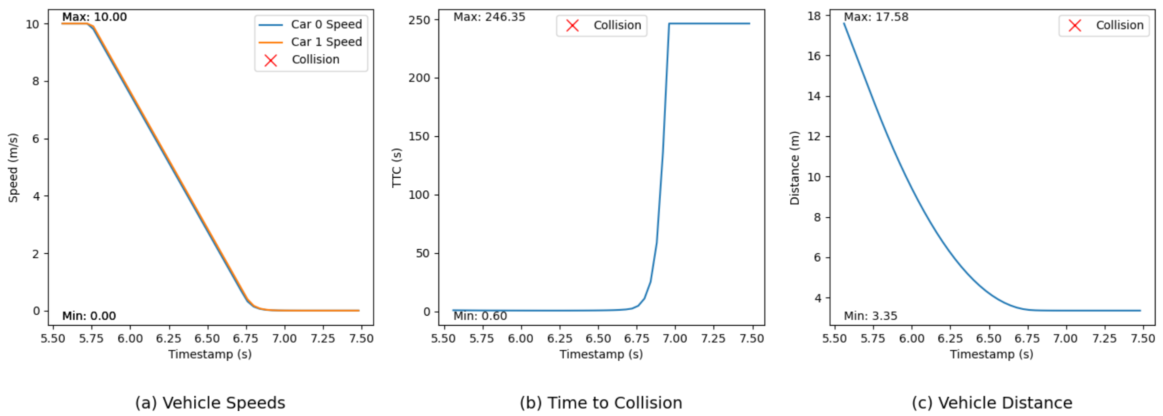


Figure 4.3: Visualization of the physical environment.

By reviewing the log, the accurate kinestate of both vehicles can be observed.



(a) Vehicle Speeds

(b) Time to Collision

(c) Vehicle Distance

Figure 4.4: Kinematic performance with channel delay of 10ms.

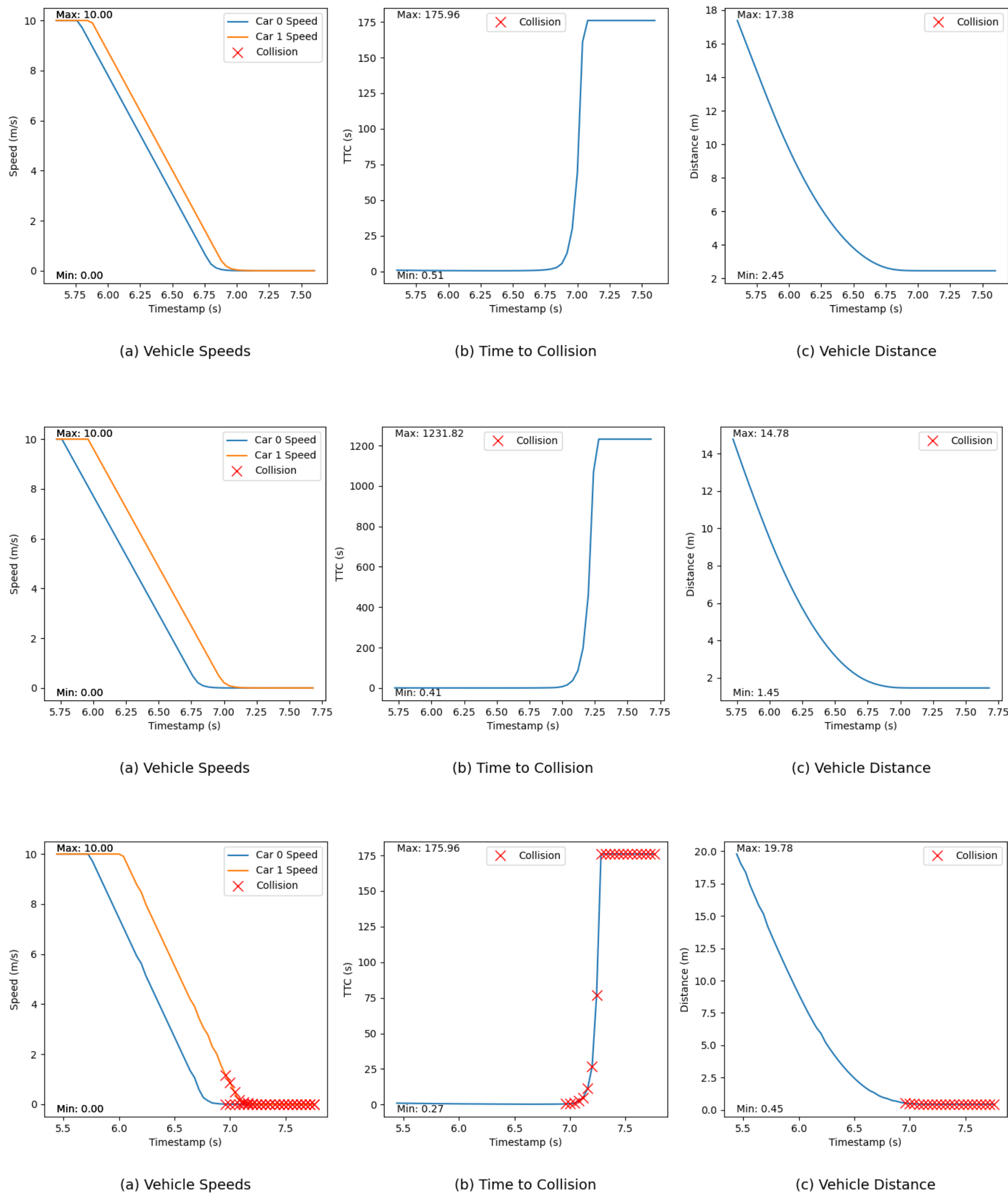


Figure 4.5: Kinematic performances with channel delay of 100ms, 200ms and 300ms.

In Figures 4.4 and 4.5, column (a) records the real-time longitudinal speeds of car0 and car1 measured by the simulator. Column (b) records the instant time to collision (TTC) between car0 and car1, and column (c) records the distance between the two cars.

In the measurements, the distance between the two vehicles is calculated based on the distance between the base links of the two vehicle models. Specifically, it is the distance between the origins of the vehicle base link frames. This origin point serves as a reference for locating other model components, and most models use the physical center point as the origin of the base link frame. The calculation of TTC follows Equation (3.7).

Channel Delay	**Minimum TTC**	**Minimum Distance**	**Collision**
10ms	0.6s	3.35m	No
100ms	0.51s	2.45m	No
200ms	0.41s	1.45m	No
300ms	0.27s	0.45m	Yes

Table 4.1: Simulation results.

Some key experimental results are recorded in Table 4.1. It is evident that as the channel delay increases, both the minimum TTC and the minimum distance between the vehicles steadily decrease. This indicates that car 1 brakes later and, as a result, the distance between the two vehicles after they come to a complete stop becomes shorter. When the channel delay reaches 300ms, a collision finally occurs. At this point, the closest recorded distance between the two cars is 0.45 meters. The simulator detects a collision between the two cars through overlap detection of the vehicle models, the collision can also be observed in the simulator at the same time.

Vehicular communication researchers can flexibly adjust various parameters in the experiment according to their specific needs to simulate the desired scenario and observe the resulting outcomes.

Chapter 5

Conclusions

In summary, this project has designed and implemented a connected autonomous vehicles physical-communication simulation platform incorporating F1Tenth Gym, ROS2 and NS3. It balances highly customizable physical simulation, a well-visualized environment, and high-precision communication simulation. This framework allows vehicular communication researchers to deploy their protocols directly in their familiar NS3 environment, thereby observing or measuring the direct impact of their protocols on the operation of CAVs in the physical world, providing strong evidence of their protocol's reliability.

Furthermore, if researchers purchase the F1Tenth hardware platform and communication modules, they can even deploy the vehicle control nodes on real vehicles, transforming the simulation from a computer-based one into a real-world physical experiment, significantly enhancing the credibility of their experiments.

In the future, interested researchers and developers can further develop more diverse and practical functionalities based on this framework. I will make my effort to maintain this simulation tool and apply the valuable knowledge gained from this project to future engineering practices.

Bibliography

[1] Jegathesan Shanmugam. ROS2 from the ground up, 2022. https://medium.com/@nullbyte.in/ros2-from-the-ground-up-part-1-an-introduction-to-the-robot-operating-system-4c2065c5e032.

[2] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[3] Malintha Fernando, Ransalu Senanayake, and Martin Swany. CoCo Games: Graphical game-theoretic swarm control for communication-aware coverage. *IEEE Robotics and Automation Letters*, 7(3):5966–5973, 2022.

[4] D. Karthik M. O'Kelly, H. Zheng and R. Mangharam. F1tenth: An open-source evaluation environment for continuous control and reinforcement learning.

[5] Koushik Bhimavarapu, Zhibo Pang, Ognjen Dobrijevic, and Pawel Wiatr. Unobtrusive, accurate, and live measurements of network latency and reliability for time-critical internet of things. *IEEE Internet of Things Magazine*, 5(3):38–43, 2022.

[6] I-Chun Chao, Shinn-Yan Lin, Kang B. Lee, Fred Proctor, Chien-Chung Shen, and Fan-Ren Chang. Precise latency measurement of unidirectional-data-flow network equipment. In *2014 IEEE International Frequency Control Symposium (FCS)*, pages 1–3, 2014.

[7] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[8] Matthew O'Kelly, Hongrui Zheng, Dhruv Karthik, and Rahul Mangharam. F1tenth: An open-source evaluation environment for continuous control and reinforcement learning. In *NeurIPS 2019 Competition and Demonstration Track*, pages 77–89. PMLR, 2020.

[9] GAZEBO. Gazebo, 2024. https://gazebosim.org/.

[10] Unity. Unity, 2024. https://unity.com/.

[11] Dieter Schramm, Manfred Hiller, and Roberto Bardini. *Single Track Models*, pages 225–257. 07 2018.

[12] R. Craig Conlter. Implementation of the pure pursuit path tracking algorithm. 1992.

[13] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[14] G. Pardo-Castellote. OMG data-distribution service: architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, pages 200–206, 2003.

[15] George F. Riley and Thomas R. Henderson. *The ns-3 Network Simulator*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.