Tampere University

Arttu Ruusiala

VEHICLE AUTOMATION SOFTWARE DEVELOPMENT USING SOFTWARE-ONLY SIMULATION

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Prof. Hannu-Matti Järvinen
Dr. Juhana Helovuo
October 2022

ABSTRACT

Arttu Ruusiala: Vehicle automation software development using software-only simulation
Master of Science Thesis
Tampere University
Master's Programme in Information Technology
October 2022

Automatic driving and driver assistance systems are gaining attraction in the automotive industry. Their development is not an easy task and requires enormous amounts of testing and validation. However, conducting all testing with a real car is expensive and inefficient. A possible solution to streamline testing is simulation, especially software-only simulation. In software-only simulation, everything is simulated using just software. It does not require any specialized hardware making it cheaper and easier to establish and scale up the number of testing environments.

The goal of this thesis was to study how a software-only simulation environment could be built using readily available open-source components. A simulator environment based on an open-source driving simulator, CARLA, was built, and an example application was developed and integrated into it using Robot Operating System 2 (ROS2). The example application, Carlabot, supports manual driving with a gamepad and utilizes a LiDAR sensor to implement a simple collision avoider, which slows down or stops the car if something is detected in front of the car.

The process of setting up a CARLA simulator environment using predefined assets, such as vehicle and world model, proved to be straightforward, and integrating a simple example application was fairly uncomplicated. However, using the environment for real product development would require customizing at least the assets.

Software-only simulation brings benefits to the software development of automatic vehicles. It allows testing on a scale that is not viable using just real hardware, and it enables using test automation already in integration testing. Software-only simulation supports agile software development, where testing begins early, already during the development.

Keywords: software-only simulation, CARLA, ROS2, automatic driving, simulation

TIIVISTELMÄ

Arttu Ruusiala: Ajoneuvoautomaation ohjelmistokehitys käyttäen ohjelmistopohjaista simulaatiota
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Lokakuu 2022

Automaattinen ajaminen ja erilaiset kuljettajaa avustavat järjestelmät herättävät nyt suurta kiinnostusta autoteollisuudessa. Niiden kehittäminen ei kuitenkaan ole täysin vaivatonta, ja kehitykseen liittyy paljon testausta ja validointia. Oikean auton käyttäminen testaukseen on kallista ja tehotonta. Simulaatiota, ja erityisesti ohjelmistopohjaista, eli kokonaan ohjelmistolla toteutettua simulaatiota, voidaan käyttää testauksen tehostamiseen. Kun simulaatio on ohjelmistopohjaista, voidaan testaukseen käyttää erikoislaitteiston sijaan yleiskäyttöistä laitteistoa, ja testaus voidaan siten toteuttaa halvemmalla ja testausympäristön monistaminen on helpompaa.

Tämän työn tavoitteena oli tutkia, miten ohjelmistopohjainen simulaatioympäristö toteutetaan käyttäen saatavilla olevia avoimen lähdekoodin komponentteja. Työssä kehitettiin myös esimerkkisovellus, Carlabot, joka integroitiin simulaatioympäristöön käyttäen Robot Operating System 2 -väliohjelmistoa. Varsinaisen simulaatioympäristön pohjana käytettiin avoimen lähdekoodin ajosimulaattori CARLAa. Carlabot-esimerkkisovellus tukee ohjausta manuaalisesti peliohjaimen avulla, sekä toteuttaa yksinkertaisen LiDAR-anturiin perustuvan törmäyksenestojärjestelmän, joka havaitsee kohteet ajoneuvon edessä, ja hidastaa tai pysäyttää ajoneuvon tarvittaessa.

CARLA-simulaattorin pystyttäminen käyttäen sen mukana tulevia malleja, kuten maailma- tai ajoneuvomalleja, oli suoraviivaista, ja esimerkkisovelluksen integroiminen CARLAan ei tuottanut suuria haasteita. Työssä tehty esimerkkisovellus ja ympäristö olivat kuitenkin varsin yksinkertaisia, ja ympäristön käyttäminen oikeaan tuotekehitykseen vaatisi vähintään simulaattorin mallien muokkaamista.

Ohjelmistopohjaisen simulaation käytöstä automaattisten ajoneuvojen ohjelmistokehitykseen on merkittäviä hyötyjä. Se mahdollistaa testauksen huomattavasti suuremmassa mittakaavassa kuin pelkällä oikealla laitteistolla on mahdollista. Simulaatioympäristön käyttäminen testiautomaatiossa mahdollistaa automaattisen integraatiotestauksen. Ohjelmistopohjaisen simulaation käyttö tukee ketterää ohjelmistokehitystä, jossa testaus alkaa aikaisessa vaiheessa, jopa kehityksen ollessa vielä kesken.

Avainsanat: ohjelmistopohjainen simulaatio, CARLA, ROS2, automaattinen ajaminen, simulaatio

PREFACE

This thesis was done as a part of my work at Atostek. The thesis was supervised by Doctor Juhana Helovuo from Atostek and Professor Hannu-Matti Järvinen from the Faculty of Information Technology and Communication Sciences, Tampere University. I am grateful for their continuous support and insightful comments throughout the thesis project.

Haluan kiittää vanhempiani ja sisaruksiani, jotka ovat väsymättä tukeneet ja kannustaneet minua läpi koulu-urani, ja erityisesti isääni, jonka jalanjäljissä olen päätynyt ohjelmistoalalle. Kiitän ystäviäni sekä työkavereitani Atostekilla, varsinkin vertaistuesta heitä, jotka samaan aikaan kanssani painivat omien diplomitöidensä parissa. Lopuksi erityiskiitokset Nooralle jatkuvasta henkisestä tuesta ja satunnaisesta tiellä olosta.

Tampere, 7th October 2022

Arttu Ruusiala

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

ADS	Automated Driving System
API	Application programming interface
CAN	Controller Area Network
DARPA	Defense Advanced Research Projects Agency
DCPS	Data-Centric Publish-Subscribe model
DDS	Data Distribution Service
DDSI	Data Distribution Service Interoperability Wire Protocol
ECU	Electronic Control Unit
GDS	Global Data Space
GNSS	Global Navigation Sattelite System
GPU	Graphics Processing Unit
HIL	Hardware-In-the-Loop
IDL	Interface Definition Language
IMU	Inertial Measurement Unit
IoT	Internet of Things
IoV	Internet of Vehicles
ISO	International Organization for Standardization
LiDAR	Light Detection And Ranging
OMG	Object Management Group
QoS	Quality of Service
ROS2	Robot operating system 2
RTPS	Real-Time Publish-Subscribe
SIL	Sofware-In-the-Loop
UE4	Unreal Engine 4
UML	Unified Modeling Language
V2I	Vehicle-to-Infrastructure
V2P	Vehicle-to-Pedestrian

V2V	Vehicle-to-Vehicle
V2X	Vehicle-to-Everything
XML	eXtensible Markup Language

1. INTRODUCTION

Automatic driving has been a growing trend in the automotive industry in the last decade. American electric car manufacturer Tesla has been pioneering automatic driving and many traditional car manufacturers are also following the trend. However, automatic driving is not a trivial challenge to solve. It is safety-critical, and requires heaps of testing to verify the functionality. Automatic driving utilizes various sensors perceiving the environment such as LiDARs or cameras. Many studies of automatic driving suggest that sufficient testing and verification requires up to billions (1 000 000 000) of driven testing kilometers [1][2]. That is a gruesome task to achieve using only real vehicles. Consequently, simulation appears to be a solution for this.

Simulation has been an integral part of car development for a long time. Traditionally, it has been utilized in vehicle dynamics, and when modeling the physical properties of a vehicle. For example, engineers have modeled the vehicle body behavior during a crash or effects of vibrations on the durability of parts in the engine. However, to develop automatic driving or driver assistance functions, different kind of simulation is needed. While the accurate simulation of vehicle dynamics is still valuable, the importance of simulating the perception of the surrounding environment is rising. The simulation should be able to maintain a model of the surrounding world and produce realistic sensor data for the automatic driving system. Ideally, the system under test should not be able to differentiate the simulated world from a real one. While realistic sensor simulation is important, it is crucial that the physics simulation and the vehicle-world interaction is realistic. This includes for example accurate simulation of wheels and traction. The field of mobile robotics simulation tools is growing quickly and there are multiple open-source tools for simulation including Gazebo [3] and CARLA [4] as well as proprietary solutions such as Cognata [5]. In CARLA and Cognata the focus of simulation is on automatic driving and realistic sensor simulation, while Gazebo focuses more on robotics and accurate physics simulation.

The objective of this thesis is to study how an automatic driving development environment could be established using software-only simulation and readily available open-source components, and to build a demonstrator of such system that could be used as a reference when planning to use software-only simulation in the future. The CARLA simulator [4] is chosen as the basis of this setup, and it is accompanied by the Robot operating system 2 (ROS2), providing a means for communication between the components [6].

The study is conducted using a case study, where a small proof-of-concept application is developed, and integrated into the setup. This study aims to analyze and evaluate the benefits of using this kind of simulation environment in automatic vehicle control software development. The analysis is based on qualitative information gathered from personal observations during the development and setup of the example case, as well as from related literature. The used literature sources were journal articles, where the same or similar tools were utilized for software-only simulation, and the documentation provided with the tools. The quantitative research method was chosen as it was suitable for a case study where the results are solely observational and numerically unmeasurable.

Chapter two describes the background of vehicle software development and how simulation is a central part of it. The following third chapter is introducing the simulation tools that are used, after which the fourth chapter presents the demonstrator project of this study, and gives an overview of the setup. Chapter five presents the results and the analysis, and finally in chapter six there are conclusions and ideas for future steps.

2. BACKGROUND

This chapter covers in Section 2.1 general knowledge and background of vehicle software development, focusing on automatic driving. Section 2.2 introduces simulation, which is often used when developing and testing automatic driving systems. Finally, Section 2.3 discusses sensors used in automatic vehicles.

2.1 Vehicle software development

A modern car, or any moving vehicle, contains a lot of electronics and software. Almost everything is controlled by a computer. Therefore, vehicle development, which has traditionally been mechanical engineering, nowadays involves more and more software engineering. From motor control systems to driver assistance functions, and from safety systems to passenger entertainment systems, everything is connected and running some kind of computer software.

This section starts with the division of onboard and offboard vehicle software in Subsection 2.1.1. It is followed by short description of the history of automatic driving. Safety is an important part of vehicle software development. There are standards and regulations that aim to enforce safety. These are briefly discussed in Section 2.1.3. Software verification typically involves a combination of simulation testing and testing using real hardware and vehicle. Section 2.2 describes the simulation aspect of automatic vehicle software development in more detail.

2.1.1 Onboard and offboard software

Vehicle software can be divided into onboard and offboard software, where onboard covers the software running on a hardware in the vehicle itself, and offboard the other systems the vehicle might be connected to. Offboard software could be, for example, an online navigation service, to which the vehicle is connected, or a diagnostics tool used during maintenance. Onboard software in cars traditionally spread across multiple electronic control units (ECU) all having specified purposes. They are connected using e.g. the controller area network (CAN). An example of a traditional such system is an electronic braking control system, which is a safety and assistance system for cars. It has electrical sensors to control and adjust the brake hydraulics to maintain the traction to the road, and to prevent the wheels from locking while braking. [7]

Vehicles connected to external services, and other vehicles form a subcategory of Internet of Things (IoT), Internet of Vehicles (IoV). Another term, often heard when vehicle communications are discussed, is Vehicle-to-Everything (V2X) and its various derivatives, such as Vehicle-to-Vehicle (V2V), Vehicle-to-Infrastructure (V2I), and Vehicle-to-Pedestrian (V2P). [8] They stand for different ways the vehicle can communicate and interact with the environment. For example, V2V communication could be automatic cars exchanging information about their planned actions, or V2I could be a road info system warning cars about an accident or congestion further up the road.

The focus of this thesis is on onboard software used for automatic or autonomous driving and driver's assistance. The distinction between automatic and autonomy is the need for some background control system, hence the ability to operate autonomously. For example, an automatic car could drive a predefined route, but if an unexpected obstacle appeared in the middle of its route, it would stop, and not be able to recover before the obstacle disappeared, or it received help from an outside system or a human. However, an autonomous car could try to adapt to the unexpected obstacle by, for example, modifying its route. The terms are often used as synonyms for each other, but technically, autonomy is much more difficult and rare. This thesis refers to them both from now on as *automated driving systems* (ADS).

Onboard ADS software can be divided into four categories: positioning, perception, planning, and control. The positioning system is localizing and keeps track of the current position of the vehicle. The perception system is utilizing sensors to extract information about the surrounding environment. The planning systems consist of path or route planning, behavior planning, meaning making the decisions and reacting to the surrounding environment, and motion planning, which is planning the actual motion actions, such as acceleration or steering. The control system is taking the actions from motion planning and transforming them into commands that can be issued to lower-level control units. [8]

Offboard software in the automatic driving context could be, for example, an upper-level control system for fleet management or a data collection service fetching information from onboard software. It could be a service that keeps an up-to-date high-definition map of the world, using data gathered from vehicles and shares it with an onboard navigation system running on all vehicles connected to the service. Because the world is constantly changing, the navigation system would then send updates to the map when it notices discrepancies between the map and the world it is perceiving with its sensors.

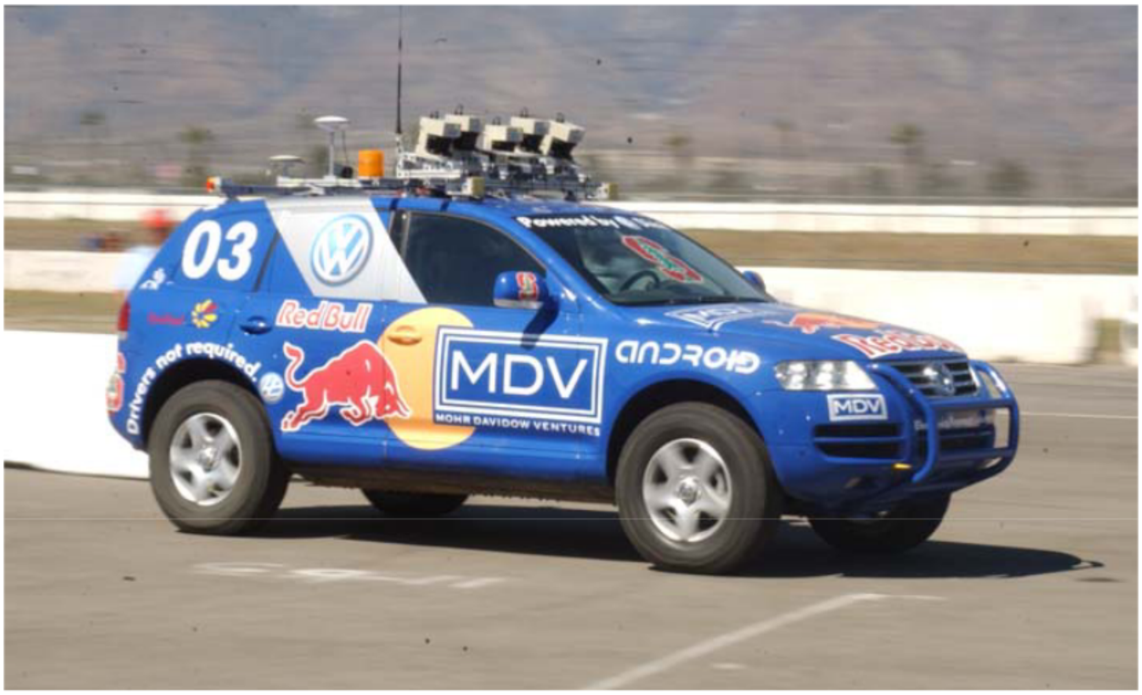

Figure 2.1. *The winner of 2005 DARPA grand challenge, Stanford Racing Team [9]*

2.1.2 History of automatic driving

Automated driving systems (ADS) are onboard software responsible for either controlling the vehicle or assisting a human driver. In a sense, early driver assistance systems such as the anti-lock braking system which date as far as the ’70s, could be counted as ADS, but the breakthrough in automated driving systems happened in the early 2000s. Especially, the DARPA (Defense Advanced Research Projects Agency) challenges were playing an important role in getting more traction for ADS development.

The United States Department of Defense organized the first DARPA grand challenge in 2004, which started a series of competitions pushing the automated driving technologies forward. The first challenge was held in the Mojave Desert, United States, and the goal was to drive a 240 km off-road course without human intervention. None of the contestants were able to finish the race, and no winner was declared. All the contestants had either suffered mechanical problems, withdrawn or disqualified from the race, or got stuck in the difficult terrain. The challenge was scheduled again for the next year 2005. At that time five vehicles completed the over 200 km route, and the era of automatic driving was kicked off. Figure 2.1 displays the car of the 2005 winning team, the Stanford Racing Team. [9]

The capabilities of ADS can vary a lot, and for that reason, ADS are often categorized based on the level of autonomy achieved. SAE International, formerly named as Society of Automotive Engineers, is an organization developing and maintaining standards for various industries, including the automotive industry. They have defined in their standard

Table 2.1. *Table describing SAE levels of driving automation [10]*

SAE level	Name	Narrative definition	Execution of Steering and Acceleration/ Deceleration	Monitoring of Driving Environment	Fallback Performance of Dynamic Driving Task	System Capability (Driving Modes)
Human driver monitors the driving environment						
0	No Automation	The full-time performance by the human driver of all aspects of the dynamic driving task, even when enhanced by warning or intervention systems.	Human driver	Human driver	Human driver	n/a
1	Driver Assistance	The driving mode-specific execution by a driver assistance system of either steering or acceleration/deceleration using information about the driving environment and with the expectation that the human driver perform all remaining aspects of the dynamic driving task.	Human driver and system	Human driver	Human driver	Some driving modes
2	Partial Automation	The driving mode-specific execution by one or more driver assistance systems of both steering and acceleration/ deceleration using information about the driving environment and with the expectation that the human driver perform all remaining aspects of the dynamic driving task.	System	Human driver	Human driver	Some driving modes
Automated driving system (“system”) monitors the driving environment						
3	Conditional Automation	The driving mode-specific performance by an automated driving system of all aspects of the dynamic driving task with the expectation that the human driver will respond appropriately to a request to intervene.	System	System	Human driver	Some driving modes
4	High Automation	The driving mode-specific performance by an automated driving system of all aspects of the dynamic driving task, even if a human driver does not respond appropriately to a request to intervene.	System	System	System	Some driving modes
5	Full Automation	The full-time performance by an automated driving system of all aspects of the dynamic driving task under all roadway and environmental conditions that can be managed by a human driver.	System	System	System	All driving modes

J3016 six levels of driving automation for on-road vehicles. The levels start with 0, where there is no automation, but the system may give a warning or momentarily intervene in the driving, to level 5 where no human is required at all. Table 2.1 describes the levels in detail. [10] Most of the current ADS fall into level 1 or 2. For example, an adaptive cruise control is a level 1 driver assistance system, and Tesla admits that its full self-driving is only at level 2 [11]. However, many car manufacturers, such as Toyota [12], Stellantis [13], and Polestar [14], have ambitious plans for level 3 or higher automation. The simulation environment studied in this work could be used to develop and test a system from any SAE level. The example application built in this work categorizes in level 1 driver assistance.

2.1.3 Regulations and standards

There are standards and regulations to guide the development of moving vehicles. The ISO 26262, titled *Road Vehicles – Functional safety* [15], is the most important standard for the automotive industry. It is an adaptation of IEC 61508: *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems* for road vehicles. The goal of ISO 26262 is to ensure that functional safety is achieved during the whole lifecycle of a vehicle: development, production, operation, service, and decommissioning. It applies to all safety-related systems in road vehicles, whether they were electrical, electronic, or software components. [15]

Traditionally, the standards have been focused on functional safety. However, cybersecurity-related standards have emerged in recent years, such as ISO 21434: *Road Vehicles – Cybersecurity Engineering* [16], and SAE J3061: *Cybersecurity Guidebook for Cyber-Physical Vehicle Systems* [17]. [18] The standards offer references and requirements, which help to achieve safety and security.

The V-model, in Figure 2.2, is a development process, which is often followed in the automotive industry. It suits well for safety-critical requirements and verification-driven development of vehicle software. [18] For each design phase in the development, there is a corresponding testing phase, which validates and verifies the system is fulfilling the requirements. Testing in a simulated environment is an important part of the validation and testing.

2.2 Simulation

The level of simulation can vary based on how it is used, and what is needed from the simulation, from a light-weight software-only simulation to a sophisticated *Hardware-In-the-Loop* (HIL) simulation. *Software-In-the-Loop* (SIL) testing is often the most lightweight simulation environment.

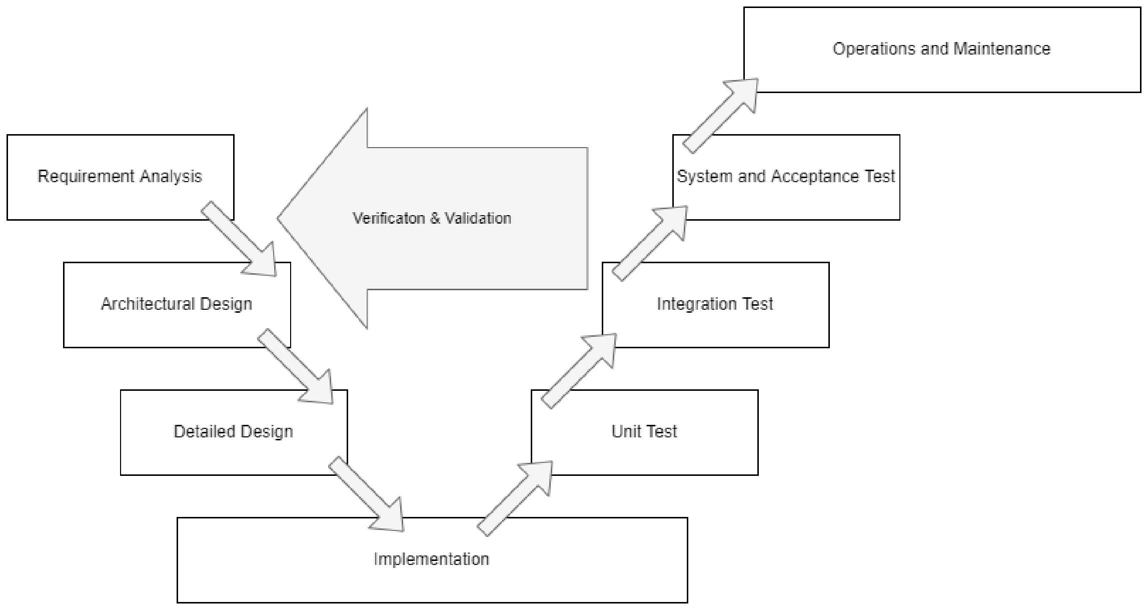

Figure 2.2. *V-model development process, adapted from [18].*

The idea of SIL is that the simulation environment does not require any specific hardware to run, and everything in the simulation is implemented using software. Possible hardware requirements are satisfied by emulating the hardware. Interfaces to other devices, such as sensors, are simulated in the software. Platform hardware emulation is utilized if the tested software needs to be running on a specific hardware. The vehicle’s lower-level control units, such as steering or braking controllers, are simulated as separate software components that implement the necessary functionality of the interface. They are acting as if they were actual independent hardware components, when communicating with the software under test. A more inexpensive SIL environment could be utilized already during the software and vehicle hardware development. It could be light enough for each developer to have their own simulation environment for testing. This allows quick development and testing cycles and enables agile software development.

Hardware-In-the-Loop, in contrast, relies on using many of the same hardware components as the real vehicle. This means that the software under test is running on the same computer hardware as in a real vehicle, and communicating with real other hardware components of the vehicle. Possible inputs for those other components are then simulated in software, instead of simulating the whole vehicle. High-end HIL simulation can be used as a last testing and verification milestone before testing with a vehicle in the real world. It is usually more expensive and complicated to build and maintain than the SIL environment.

For example, Ford Motor Company has utilized HIL simulation in the development of a SAE level 2 ADS system. They used Simulink [19] and CarSim [20] as the base of the hardware simulation. [21] Simulink is a framework for developing, testing, and simulating using model-based design. It is a popular framework for setting up a HIL environment.

CarSim is a simulation tool, which is specialized in the simulation of the mechanical and physical properties of a car. These two along with Gazebo [3] are simulation tools made specifically for HIL or SIL testing. However, there are other options as well.

Video games could be utilized for driving simulation. Many of them have stunningly realistic visuals and quite advanced physics simulations. For example, there are studies where Grand Theft Auto V [Rockstar Games, 2014] is used as a simulation environment for testing automated driving systems [22][23]. However, using video games poses certain challenges. Video games do not usually support detailed benchmarking and customization. The control over the simulated world is limited. It can be difficult to have a deterministic environment. Data collection from a video game might be difficult and require modifications to the game, which might not be possible due to the closed-source nature of video games.

However, using a video game engine to create bespoke driving simulation from the ground up is a way to benefit from technologies created for video games. A video game engine is a framework providing tools for needed in making video games, such as real-time rendering, animation, and physics simulation. Originally they were used just to create video games, but nowadays, they have become more common also in other fields, that can benefit from photorealistic rendering, such as filming, architecture, and automotive industry. [24] Unity [25] and Unreal Engine [26] are both game engines that are widely used as the core of a simulation environment. CARLA [4] and Airsim [27] are open-source examples of simulators based on a game engine.

2.3 Sensors

Sensors and perception are an important part of vehicle software. They provide the ADS means to localize and position themselves, as well as react and adapt to the changing environment. Sensors can be classified into two categories: vehicle dynamics sensors and environmental sensors. Vehicle dynamics sensors are observing and measuring the state of the vehicle itself. Such sensors are, for example, wheel encoder, tire pressure sensor, and inertial measurement unit (IMU). Environmental sensors, such as LiDAR, camera, radar, or Global Navigation Satellite System (GNSS) antenna, are perceiving the surrounding environment. For autonomous driving, LiDAR is one of the most important sensors, along with the GNSS.

Subsection 2.3.1 briefly discusses the vehicle dynamics sensors. From the environmental sensors, LiDAR is introduced in Subsection 2.3.2, as it is the most important, and the only sensor used, in the demonstrator of this work.

2.3.1 Vehicle dynamics sensors

Vehicle dynamics sensors are producing information about the physical state of the vehicle itself. Many of these sensors are present even in modern cars without any ADS capabilities. A wheel encoder is an electromechanical sensor that measures the angular position or movement of a shaft or an axle. They are used for odometry measurements. [2] Figure 2.3 shows anti-lock braking system (ABS), which is an example of a wheel encoder in a passenger car.

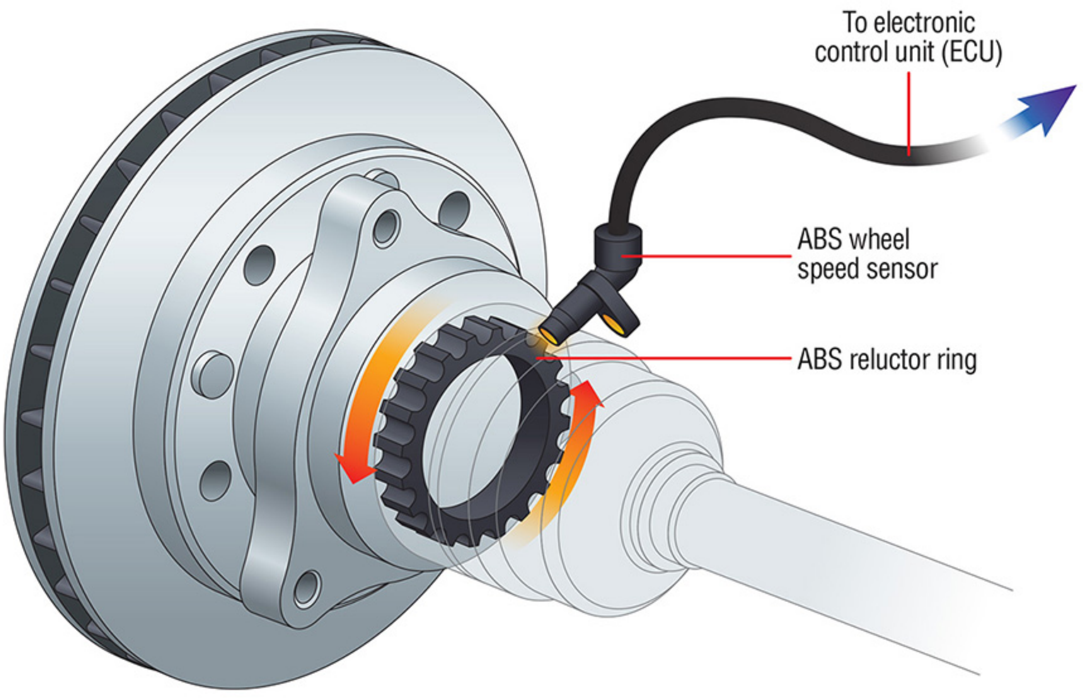

Figure 2.3. *Anti-lock braking system (ABS). [28]*

IMU is a sensor that is a combination of accelerometers, gyroscopes, and sometimes magnetometers. It measures the acceleration and angular velocity of the vehicle, and when combined with the GNSS, it can provide accurate positioning results. [8]

2.3.2 LiDAR sensor

LiDAR is an acronym for light detection and ranging. It is an application of laser ranging, where transmitting laser radiation and measuring the reflected radiation is used to calculate the distance to objects. Examples of other applications are laser rangefinders or scanners. In laser ranging, the measuring instrument emits a beam of laser radiation towards the measured object and then proceeds to measure the reflection. There are two methods to calculate the range of the object. The first, time of flight is based on sending short laser pulses and then measuring accurately the time it takes for the pulse to be reflected back from an object. The second, solid state ranging is based on the amplitude (or intensity) modulated continuous laser beam and the range is calculated using the phase difference between transmitted and received beam. [29]

LiDARs typically use the pulsed time of flight ranging and rotating beams to make millions of measurements of a surrounding environment. Measurements produce a point cloud, which is a 3D map of the environment as seen by the LiDAR. Figure 2.5 displays an example of a visualized point cloud. The main properties of LiDAR sensors are the number of beams, the vertical and the horizontal resolution (Figure 2.4), the range, the rotation frequency, and the points generated per second. A LiDAR can emit one or more laser beams for each measurement cycle. The beams are usually vertically stacked and the angles between the beams are determining the vertical resolution of the LiDAR. The horizontal resolution is the angle that the sensor is rotated between each measurement. The range describes how far the sensor can detect objects. Rotation frequency tells the frequency at which the measurements are taken. [29]

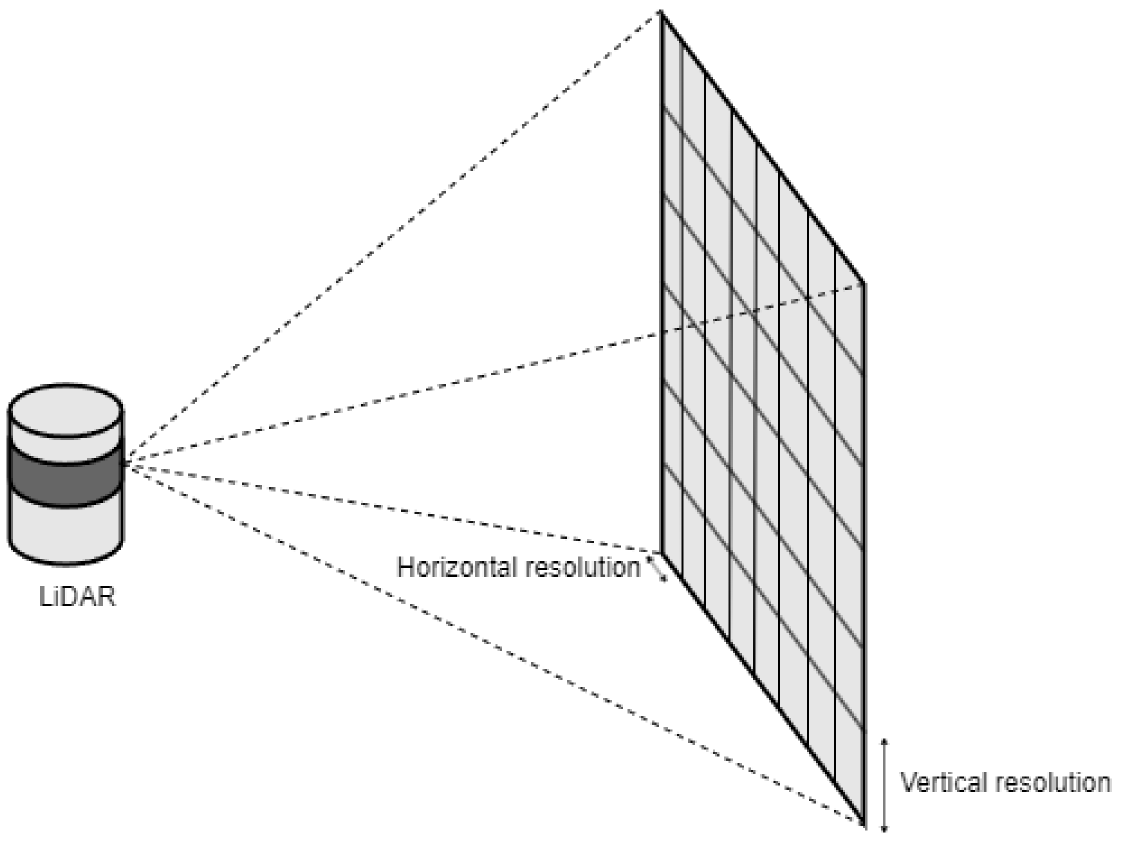

Figure 2.4. *Illustration of horizontal and vertical resolution of LiDAR.*

LiDARs can be simulated using ray tracing. Ray tracing is a method for simulating the behavior of light by casting a ray of light, either from an observation point or from a light source. It is then checked and calculated where the light ray would hit the objects in the scene. [30] When the properties of a LiDAR are known, a virtual sensor can be constructed. Rays are cast from the simulated LiDAR origin, and the distance to the

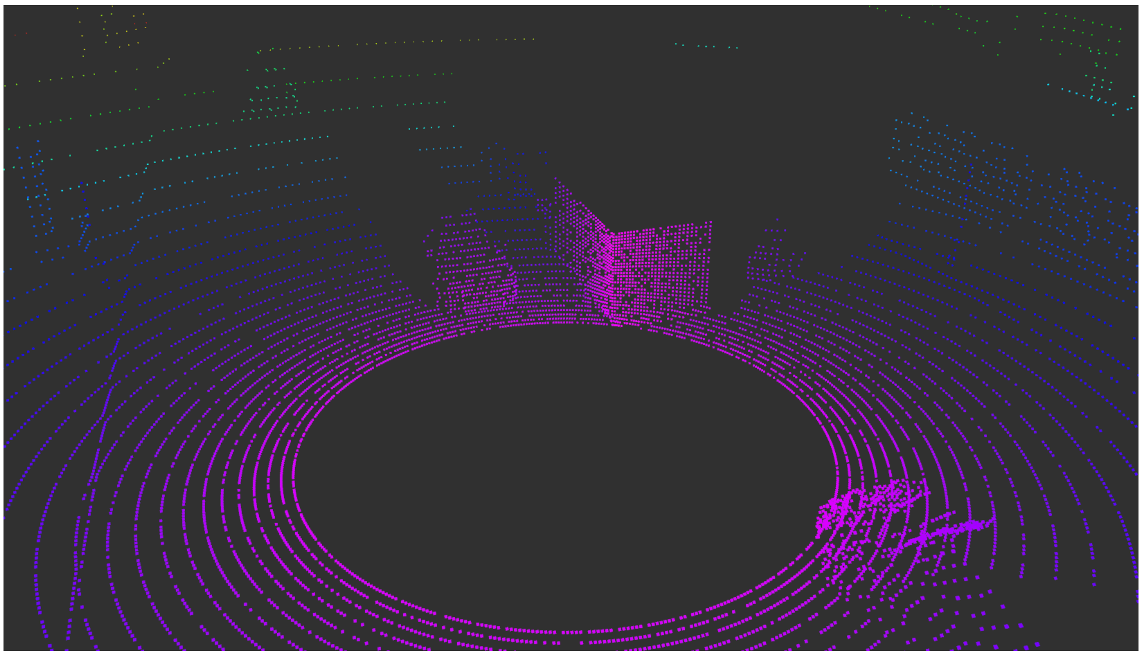

Figure 2.5. *LiDAR generated point cloud data visualized in RViz tool.*

intersection point of the ray and objects in the simulation environment is calculated. These rays are calculated with the frequency and angles similar to how a real sensor would send laser beams. Calculated distances can be used to produce a point cloud. Using just ray tracing produces a too ideal result. It does not produce any noise and disturbances that are present in real-world measurements. For that reason, drop-off of measurements and addition of noise are introduced into the produced data in simulation. These can be based on, for example, the distance the ray has to travel, or completely at random.

3. SIMULATION TOOLS

This chapter gives an introduction to simulation tools that were used in this work. Section 3.1 describes the CARLA simulator used for world and sensor simulation. Robot Operating System 2 (ROS2) is presented in Section 3.2 followed by Data Distribution Service (DDS) in Section 3.3.

3.1 CARLA

CARLA is an open-source driving simulator focused on supporting the development of automatic driving systems. Its main use context is in the automotive industry and automatic driving of passenger cars, but it can be customized for other fields of automatic driving due to its open-source nature [31]. Using custom maps and vehicle models, it can be used, for example, in the development of automatic heavy working machines. CARLA is developed and maintained by the Computer Vision Center of Universitat Autonoma de Barcelona, and it has some well-known companies as sponsors such as Intel and Toyota Research Institute. CARLA has been utilized in both SIL [32] and HIL [33] testing and is gaining popularity in autonomous driving research. Its key features are realistic world simulation, focusing on urban environments, including traffic and pedestrian simulation, and sensor simulation. [4]

The following subsections will describe the main aspects of CARLA. The first Subsection 3.1.1 describes the architecture of CARLA. It is followed by Subsection 3.1.2 describing the vehicles, Subsection 3.1.3 the sensors, and Subsection 3.1.4 the world in CARLA.

3.1.1 Architecture of CARLA

CARLA follows the client-server architecture. The core of the server and world simulation is a game engine, Unreal Engine 4 (UE4). It provides a 3D world with physics simulation and tools to create game logic for controlling the vehicles and other objects in the world. A CARLA plugin on top of the game engine provides the functionality needed for the server to communicate with clients and control the simulation state. The main responsibilities of the server are updating the world-state and its actors, computing the physics and rendering the simulated sensor data. In CARLA, objects in the simulated world that can

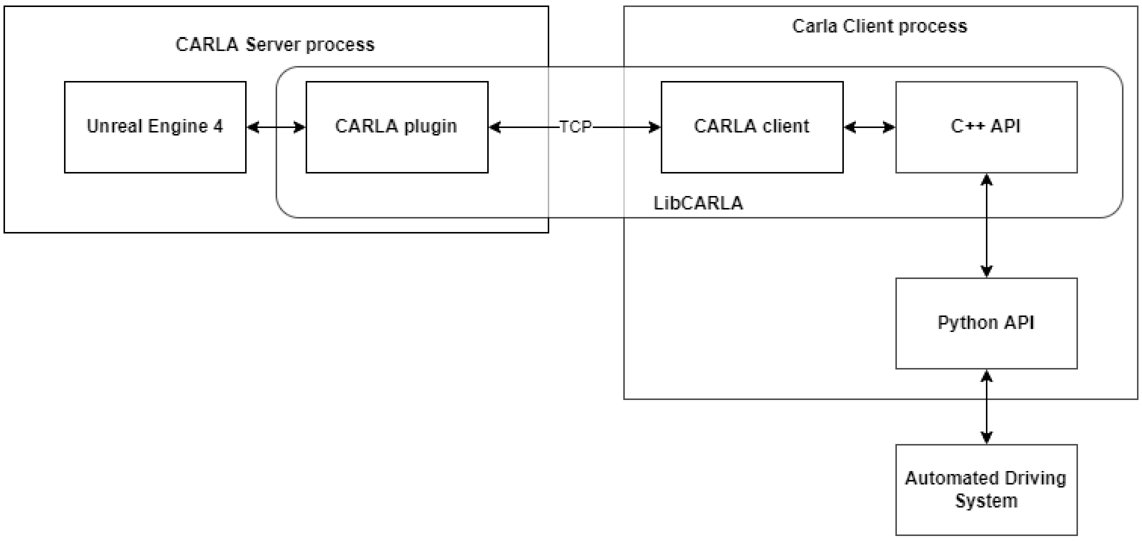

Figure 3.1. *Architecture of CARLA server and client.*

do actions and interact with other objects are called *actors*. Actors can be, for example, pedestrians, vehicles, sensors attached to the vehicles, or traffic lights and signs. Vehicles controlled outside of the simulation i.e. controlled by vehicle control software running against the simulator, are called *ego vehicles*. [34]

The client-side controls the logic of actors and manipulates the world conditions such as weather, or time of day. The server and the clients do not need to be running on the same machine. The connection is established using TCP/IP protocol. Both the server and the client utilize LibCARLA for establishing the connection between the two. CARLA provides multiple ways for clients to connect to the server. CARLA client application programming interface (API) is available for Python and C++. For Python, there is a client library that is packed with a CARLA installation package, whereas the C++ library needs to be built from the source code. Figure 3.1 presents the architecture of CARLA with ADS connected to it using the Python API. In addition, there is support for Robot Operating system (ROS), both original and ROS2, using the CARLA ROS bridge. It is using the Python API to translate the communication to ROS topics and services. CARLA ROS bridge supports most of the same functionalities as the Python API, but some functions are missing. [34]

When the CARLA server is started, it starts up using the default configurations and one of the default maps. The server can be configured to some degree using command line arguments when starting it. For example, to set the network ports it is using for client connections. However, the majority of configuration is done using clients. Also, a freshly started CARLA server does not have any actors. They need to be spawned from the client side. CARLA installation comes with a set of Python scripts. There are both examples of how to use the Python API, and utility scripts to ease the configuration, and manipulate the simulation world state. [34]

Actors are defined in CARLA with *blueprints*. A CARLA blueprint is, in a way, an extended

Figure 3.2. *A screenshot of Tesla Model 3 in CARLA.*

class definition. It is a combination of things, such as functionality and code, a 3D model, and several attributes, of which some are configurable. CARLA blueprints allow spawning new instances of actors to the simulation. There are five different types of actors: sensors, spectators, traffic signs and lights, vehicles, and walkers. [34]

3.1.2 Vehicles

Vehicles are actors that consist of a 3D model and a list of attributes that describe the vehicle. Some attributes can be modified after spawning while others are only describing the properties of a vehicle. Version 0.9.13 has around 35 pre-defined vehicles including passenger cars, bikes, and trucks. Vehicles include both imaginary vehicles and vehicle models from real manufacturers, such as Tesla Model 3 shown in Figure 3.2. From the actor's point of view, ego vehicle and common traffic vehicles are the same. The difference is only which entity is controlling the driving. Ego vehicles are controlled with control messages through CARLA client API, while traffic vehicles are controlled by one of the traffic simulators provided by CARLA. [34]

3.1.3 Sensors

Sensors in CARLA are actors, which retrieve information from the surrounding simulation environment, and pass it on to the client. Sensor actors should be attached to a parent actor, usually to a vehicle. CARLA comes with an implementation of a variety of sensors such as RGB cameras, depth cameras, LiDARs, radars, and IMUs. These sensors can produce data continuously, but there are also sensors that produce only on certain events. Such a sensor is, for example, a collision sensor that provides data only if a collision

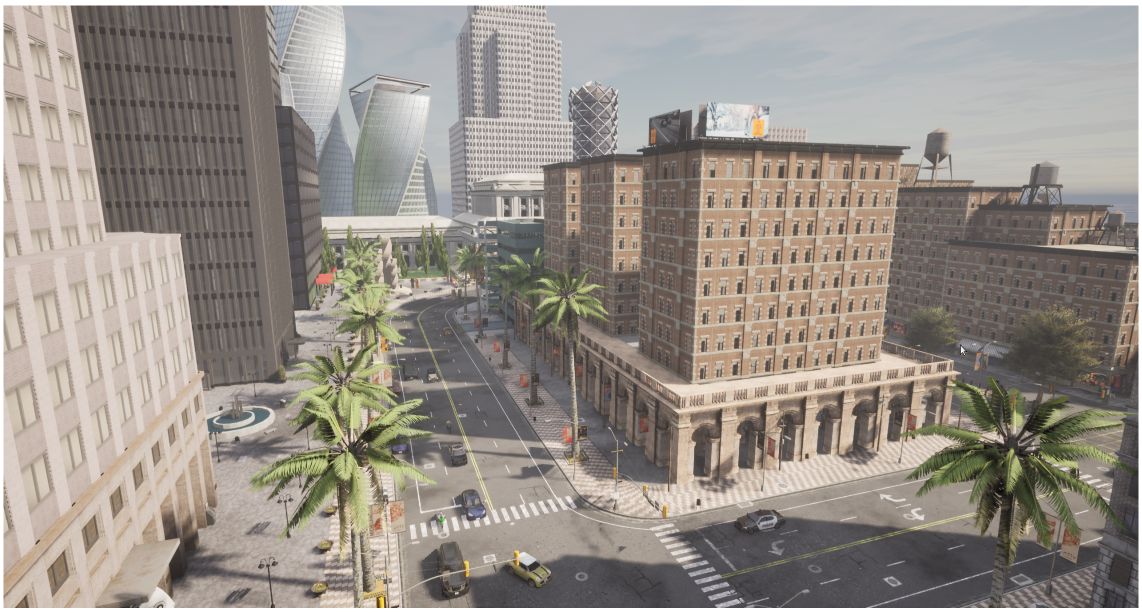

Figure 3.3. *A scene from Town10HD map in CARLA.*

happens.

Sensors are spawned from the client-side, but for most sensors, the computation for their measurements is done on the server-side. The server uses LibCARLA to serialize the sensor data and send it to the client using TCP. The client deserializes it and depending on the API used, makes it available for the connected automated driving system.

3.1.4 World

CARLA version 0.9.13 comes with 8 different pre-defined maps. They vary from a simple small-town layout to a highway loop and city environments. Figure 3.3 displays a scene from an urban map Town10HD. A map consists of a 3D model of the world and its road definition. Road definition is based on OpenDRIVE standard [35] by ASAM (Association of Standardization of Automation and Measuring Systems). OpenDRIVE defines roads, lanes, junctions, and road network-related objects, such as road signs or traffic lights. New and existing maps can be created and modified using Unreal Engine Editor. In addition, generating maps from the OPENDrive definition is supported.

World simulation support changing environmental conditions, such as time of day or weather. Weather options include controlling cloudiness, rain intensity, wind intensity, azimuth, and angle of the sun, fog, and light interaction with small particles in the air such as pollution or dust. Snow and ice, which are quite important in Finland, are missing. Additionally, most of the weather effects only affect camera sensors making, for example, LiDARs work too ideally.

CARLA provides four means to simulate traffic and run specific scenarios: Traffic Manager, Scenario Runner, Scenic, and SUMO. Traffic Manager is the simplest tool. It is a module within CARLA that controls the vehicles from the client-side. It is simply enabled by setting the autopilot attribute on for a vehicle actor. The Traffic Manager also controls pedestrians. Vehicles controlled by Traffic Manager are driving and making random turns at intersections, while obeying traffic lights and speed limits, and avoiding collisions. Although it is possible to configure some vehicles to ignore the traffic rules to test more authentic traffic behavior. The Traffic Manager is running as a client, but keeps track of the simulation state, and a registry of all vehicles. It utilizes them to calculate a driving path and the reactions to traffic lights, or potential collisions. The calculated plan is then translated to driving commands which are sent to the CARLA server. The calculations for all Traffic Manager Controlled actors are done simultaneously, and the commands are sent to the server as a batch. [34]

The other three (Scenario Runner, Scenic, and SUMO) are not included in the main CARLA package, but are available separately. Scenario Runner allows for defining and running complete and complex scenarios. Scenario Runner comes with a set of example scenarios. The scenarios are defined either with a Python interface, or using the OpenSCENARIO standard [36]. OpenSCENARIO is a standard defining a file format for describing complex and dynamic driving and traffic simulator scenarios. It is maintained by ASAM. Scenario Runner has support for running scenarios with distinct metrics for evaluating the performance of the AD systems being tested.

Scenic [37] is alternative for defining and running scenarios. It is a domain-specific probabilistic programming language for modeling environments for robots and autonomous cars. It is a separate project from CARLA, and supports multiple other simulators.

SUMO is the fourth option for traffic simulation. It is an open-source project focused on microscopic traffic simulation. Microscopic traffic simulation means that each vehicle and its dynamics are simulated individually, whereas in macroscopic only traffic flow and its density is simulated. [38] SUMO can be connected with CARLA to control vehicles in the simulation similarly to with Traffic Simulator.

3.2 Robot operating system 2 (ROS2)

Robot Operating System 2 (ROS2) is an open-source middleware for developing robotics applications. It is maintained by Open Source Robotics Foundation (OSRF). ROS2 provides a topic-based publisher/subscriber transport layer for communication and useful libraries and tools for robot development. Its predecessor Robot Operating System (ROS) was lacking support for real-time performance and was only available for certain operating systems. Therefore, ROS2 was introduced in 2014 to improve on these aspects [6] and in 2017 the first distribution of ROS2 was released. ROS2 utilizes Data Distribution Service (DDS) for its inter-process communication. [39]

The ROS2 communication network is referred to as the ROS2 graph. It consists of nodes each responsible for a single purpose. This spreads the functionality of the application to single-purpose modules, which communicate with each other using topics, services, actions, and parameters. Topics are used for nodes to send messages to each other. Each topic has a name, and message type that defines the structure of the data. Nodes can subscribe to topics to receive these messages, or publish their messages in the topics. Topics are meant for continuous data streams such as sensor data or state information. They are the main communication channel in ROS2 and the only one used in the scope of this thesis.

Services similarly to topics have a name but instead of publishing and subscribing, one node is acting as a server and others as clients. A client node requests from the server node, and the server node returns some response. Services are meant for remote procedure calls, such as querying the state of a node, or running some calculation. Figure 3.4 shows an example of nodes communicating via topic and service.

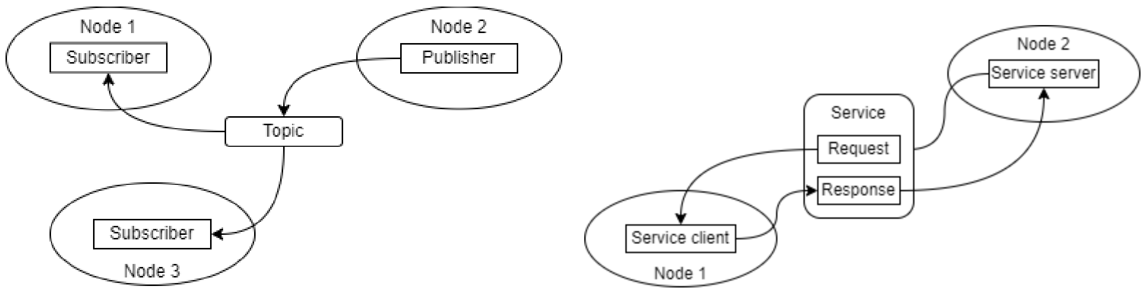

Figure 3.4. *Examples ROS2 nodes communicating with a topic and a service.*

Actions are similar to services, but for long-running tasks. They are built using services and topics. The action client node requests some goal from the action server node, using a goal service. The action server will then execute the action towards the goal, while publishing its progression in the feedback topic. Finally, when the goal is achieved, the action server sends the result to the client using the result service. Figure 3.5 shows an example of nodes communicating via action. [40]

ROS2 includes tools such as RViz, rqt, and roslaunch. RViz is a 3D visualizer that can visualize the robot itself, as well as the environment it is working in, and the data its sensors are producing. It works by subscribing to relevant topics and then drawing a visualization of the published data. Rqt provides a software framework for various GUI tools for ROS2. It is highly customizable via plugins. Roslaunch is a tool that can be used to automate and configure the launching of ROS2 nodes. [40]

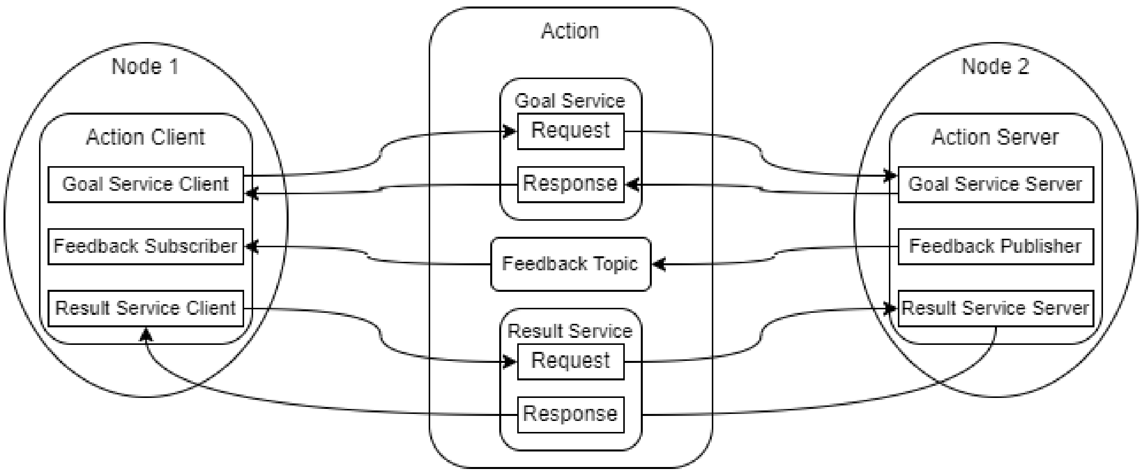

Figure 3.5. *Example of ROS2 nodes communicating with an action.*

3.3 Data Distribution Service (DDS)

Data Distribution Service (DDS) specifies a standard for networking middleware that implements the publisher-subscriber pattern for real-time systems. Object Management Group (OMG) is a non-profit technology standard consortium, which manages the definition of the standard, but the implementations for it are provided by different vendors. For example, CARLA ROS bridge is using eProsima's implementation of the DDS, and the Flexbot framework, used in the demonstrator in this work, relies on RustDDS [41], which is an open-source DDS implementation for Rust programming language, maintained by Atostek Oy. DDS standard consists of two parts: DDS API standard and Data Distribution Service Interoperability Wire Protocol (DDSI). The DDSI specifies the Real-time Publish-Subscribe (RTPS) wire-level protocol and it provides wire interoperability across different implementations while the DDS standard specifies the API that allows different vendors to make their implementations. [42]

DDS is widely used for applications that depend on distributing high volumes of data with real-time constraints. It is used, for example, in applications in automated financial trading, defense, aviation, and supervisory control and data acquisition systems. DDS aims to answer three key challenges: real-time, dependability, and high performance, which make DDS viable for these fields. First, it operates in real-time, meaning the information is delivered consistently at right time and in the right place. Real-time does not necessarily mean that the data is distributed with low latency, but that it may not miss the deadlines set for it. The second challenge addressed is dependability. It means that the system integrity is kept, and data is available and reliable even if hardware or software failures occur. The third challenge is the high performance, which means that the DDS can distribute very high volumes of data with low latencies while also succeeding in the two aforementioned challenges. [42]

The core of the DDS API standard is a data-centric publish-subscribe model (DCPS).

The idea is that the information is in a state that is shared between all nodes. That state is called Global Data Space (GDS). Information in the GDS is defined by topics. The concept of topics is similar to ROS2 topics. They have a name, a data type, and a collection of Quality of Service (QoS) policies. Topic names need to be unique within a GDS. Topic data types are structural, and can be defined with a variety of syntaxes, such as Interface Definition Language (IDL), eXtensible Markup Language (XML), Unified Modeling Language (UML), and annotated Java. [42]

The nodes in DDS are called participants and consist of publishers and subscribers, which contain *DataWriters* and *DataReaders* for specific topics. They are abstractions for reading and writing to a topic, and utilize QoS to match which readers can subscribe to which writers in a topic. The matching for DDS subscriptions is made against the name and the type of the topic along with QoS policies offered by DataWriters and requested by DataReaders. To match a reader to a writer requested policies should be less demanding than the ones offered. [42]

The QoS can define, for example, the availability of the data to the participants with *Durability* policy. The *Volatile* level of Durability does not maintain the published data for participants joining the topic late, while *Transient local* and *Transient* make sure that the data is stored for late joiners. Another example is the *Reliability* policy, which dictates if lost data is guaranteed to be re-transmitted with *Reliable* policy, or if the lost data is not retransmitted with *Best effort* policy. [42]

4. DEMONSTRATOR

This chapter introduces the demonstrator simulation environment in Section 4.1, Flexbot framework in Section 4.2, and a proof of concept application, *Carlabot*, in Section 4.3.

4.1 Environment setup

The goal was to set up a demonstrative development and testing environment using the CARLA simulator. To test this environment in practice, some software to integrate into the simulation is needed. Therefore, a small vehicle control application, called *Carlabot*, was developed using the Flexbot framework. The communication between the simulation and the application is established using ROS2. Visualization of the sensor data utilizes the RViz visualization tool which is a part of ROS2. The simulation environment and the application, as described in Figure 4.1, are running the same computer, although that is not necessary. The example application represents SAE level 1 [10] automation system. It is a driver assistance system providing simplistic collision avoidance, and therefore it requires a human driver. Manual driving is implemented using a gamepad.

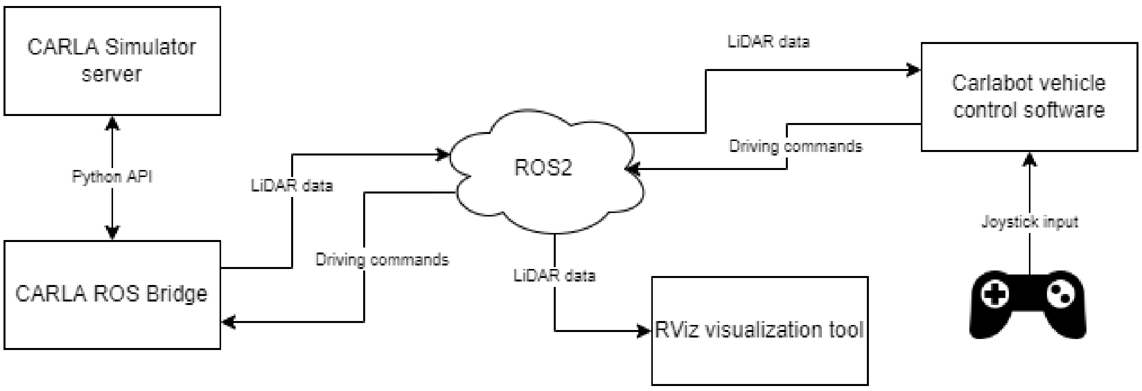

Figure 4.1. *Architecture of the demonstrator environment.*

The vehicle model used in the simulation was Tesla model 3. It was chosen as it is an electrical car without a gearbox, which simplifies the controlling, as gear changes are not required. The world model is not significant for this work, so Town10HD_Opt was chosen, since it was one of the more detailed and visually rich maps of the pre-defined maps. A virtual LiDAR sensor is attached to the roof of the car. Additionally, a virtual camera sensor follows the car from behind. The images are not used in the example application, but they are visualized in the RViz visualization for a human driver to see better what is happening in the simulation. An example view of RViz visualization is shown in the Figure 4.2.

The simulation environment was set up on a desktop PC running Ubuntu 20.04 Linux. The setup does not have very demanding requirements for the hardware, although, to utilize CARLA's high-quality image rendering, a separate graphics processing unit (GPU) is recommended. The processor was Intel i5-10600K and GPU NVIDIA RTX 3090, but lower-end components would probably have been sufficient. For manual driving, Sony Dualshock 3 gamepad was used. Ubuntu 20.04 Linux has USB drivers for Dualshock 3, and the gamepad is accessible as a generic joystick device.

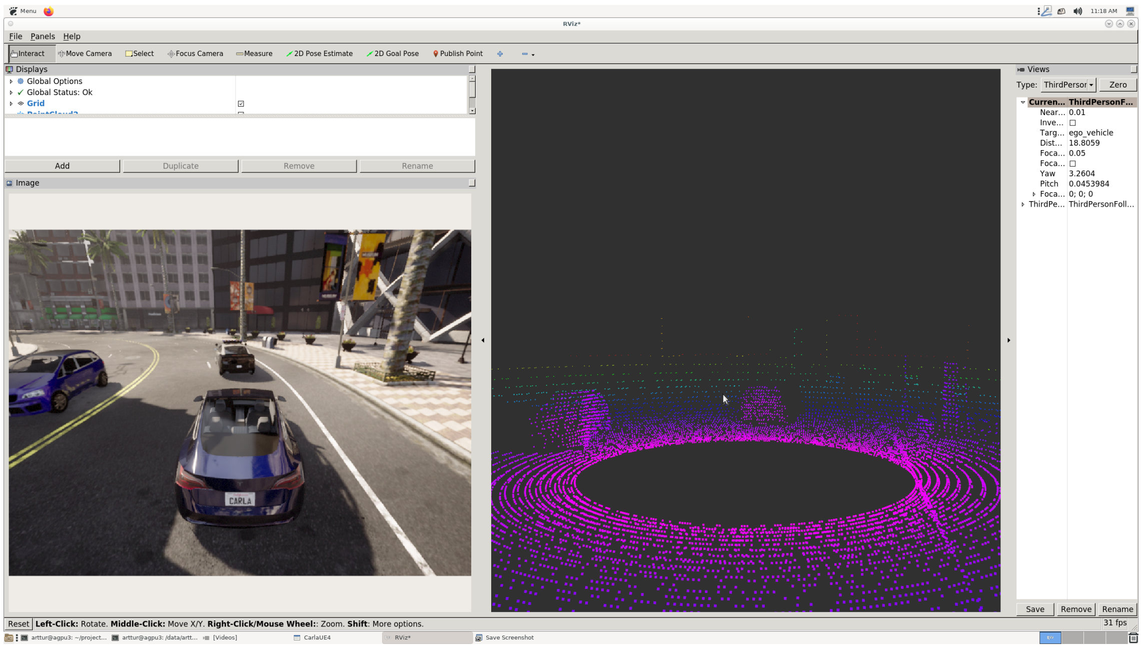

Figure 4.2. *A screenshot of RViz visualization, where the virtual camera feed is shown on the left, and the virtual LiDAR pointcloud view in on the right.*

CARLA provides install packages for both Linux and Windows. Alternatively, one can build CARLA from the source codes as it is an open-source project. Using a ready-made package is likely enough for most users as it allows customization of the environments and vehicles, and integrating your control software into it, to control the vehicle in simulation. However, building from sources allows customizing CARLA more by, for example, adding new or modifying existing sensors.

Setting up the simulation environment requires some manual steps, although those could be automated with a script. Spawning actors can be partly done with the CARLA ROS bridge. Spawning the ego vehicle and the actors, such as sensors related to it, can be done using the CARLA ROS bridge, but generating general traffic and pedestrians are handled easier through the Python API with a script.

There is an option to utilize containerization with CARLA. A Docker image including the

CARLA server is available in the Docker Hub. This enables running the CARLA server without having to install the dependencies or running multiple instances of the server. [34] In this work, containers were not used, but they were recognized as something that could be used in the future if this kind of environment setup would be taken into wider use. Packing the server and related scripts for setting up the environment in a Docker container could significantly simplify the setup process and make the environment more easily attributable. This could be used, for example, in automated testing, or distributing the same testing environment to every developer without the inconvenience of installing dependencies and doing manual configurations.

4.2 Flexbot framework

Flexbot is a software framework developed at Atostek Oy [43]. It provides a platform for creating microservice architecture based onboard robot control software. Control software is split into nodes that each have a specific task. The system structure specification describing the nodes of the application, and the connections between them, is written in Haskell. From that specification, a code generator generates the node interfaces and data structures needed to implement the nodes in Rust language. This allows the developer to focus on the actual functionality and implementation of the nodes. Flexbot framework provides logging of the communication between nodes. Flexbot code generator also produces a visual graph of the nodes and how they are connected in graphml format, which can be viewed and modified with graph editor tools, such as yEd. [44][45]

Flexbot is flexible when mapping nodes to threads or processes. Each node can run on its own thread within a single process or the nodes can be spread across multiple processes. They can even be defined external when the Flexbot application is communicating with non-Flexbot systems. The node mapping can be easily modified independently of the actual implementation of the nodes by changing the specification and running the code generation again. The nodes have typed inputs and outputs, and the defined communication channels between them can use various methods depending on how the nodes are mapped to processes. Communication between nodes within the same process can be generated in native Rust communication. For inter-process communication (IPC), for example, ROS2 can be used. [44][45]

4.3 Example Application

A proof-of-concept *Carlabot* example application was made using the Flexbot framework. Carlabot implements a simple LiDAR-based collision avoider and manual driving using a gamepad. Figure 4.3 presents the nodes of Carlabot and how they communicate with each other and the CARLA ROS Bridge. Carlabot consists of three nodes: *joystickDriver*,

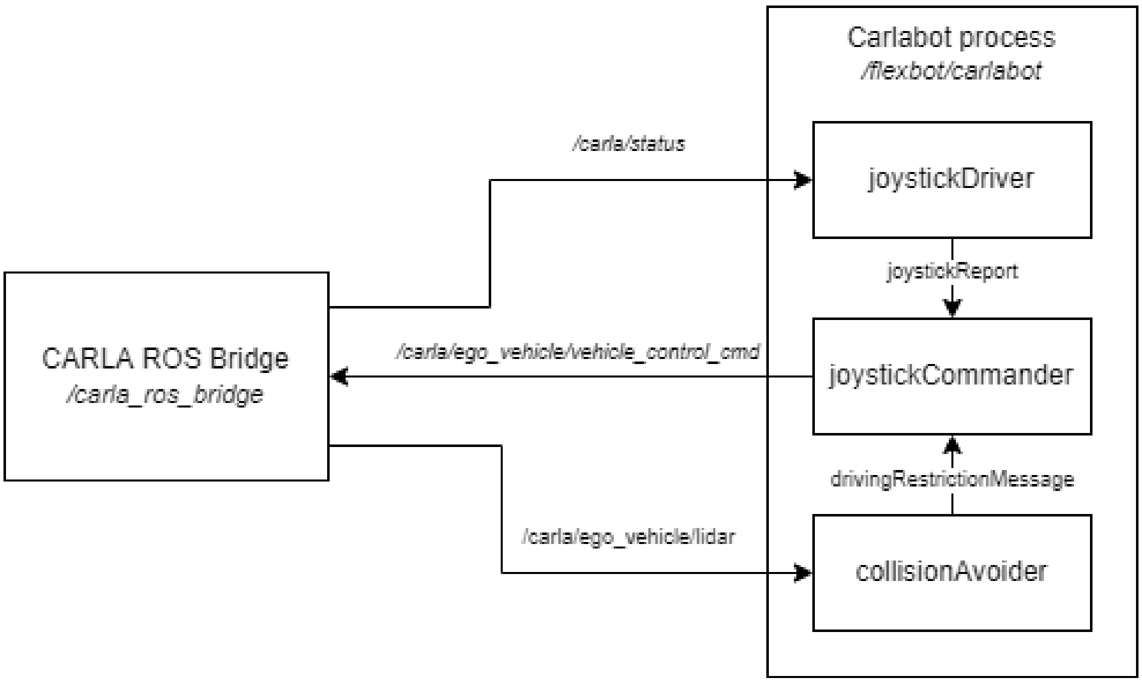

Figure 4.3. *Graph of ROS2 communication between the example application and CARLA ROS Bridge*

joystickCommander, and *collisionAvoider*.

JoystickDriver is responsible for reading the joystick state, i.e. the buttons pressed and analog stick positions, from the generic Linux joystick device. The reading of the joystick state is done whenever CARLA ROS Bridge publishes to `/carla/status` topic, and the simulation time is ticked. CARLA server is running in asynchronous mode without a fixed timestep meaning it is running with the quickest tick rate possible with the hardware. The joystick state is read every tick to ensure that a driving command is issued every time the simulation state is updated. JoystickDriver sends a JoystickReport message to the joystickCommander node, which combines the joystick commands with driving restrictions coming from collisionAvoider.

CollisionAvoider node subscribes to `/carla/ego_vehicle/lidar` topic to receive the LiDAR data from the simulation. It processes the LiDAR data and decides to either limit the speed or stop the vehicle completely. CollisionAvoider sends drivingRestriction-Message to the joystickCommander node. JoystickCommander utilizes the information from the other two nodes to produce vehicle control commands, which it publishes to `/carla/ego_vehicle/vehicle_control_cmd` topic.

The collisionAvoider receives the point cloud and checks if there are any points in the defined action areas illustrated in Figure 4.4. CollisionAvoider is interested in an area spanning 1.5 meters to both sides of the LiDAR origin, and in front of it. Points closer than 5.9 meters in front of the LiDAR cause stopping, and points closer than 10.0 meters invoke slowdown. Slowdown applies half of the maximum braking power to the driving

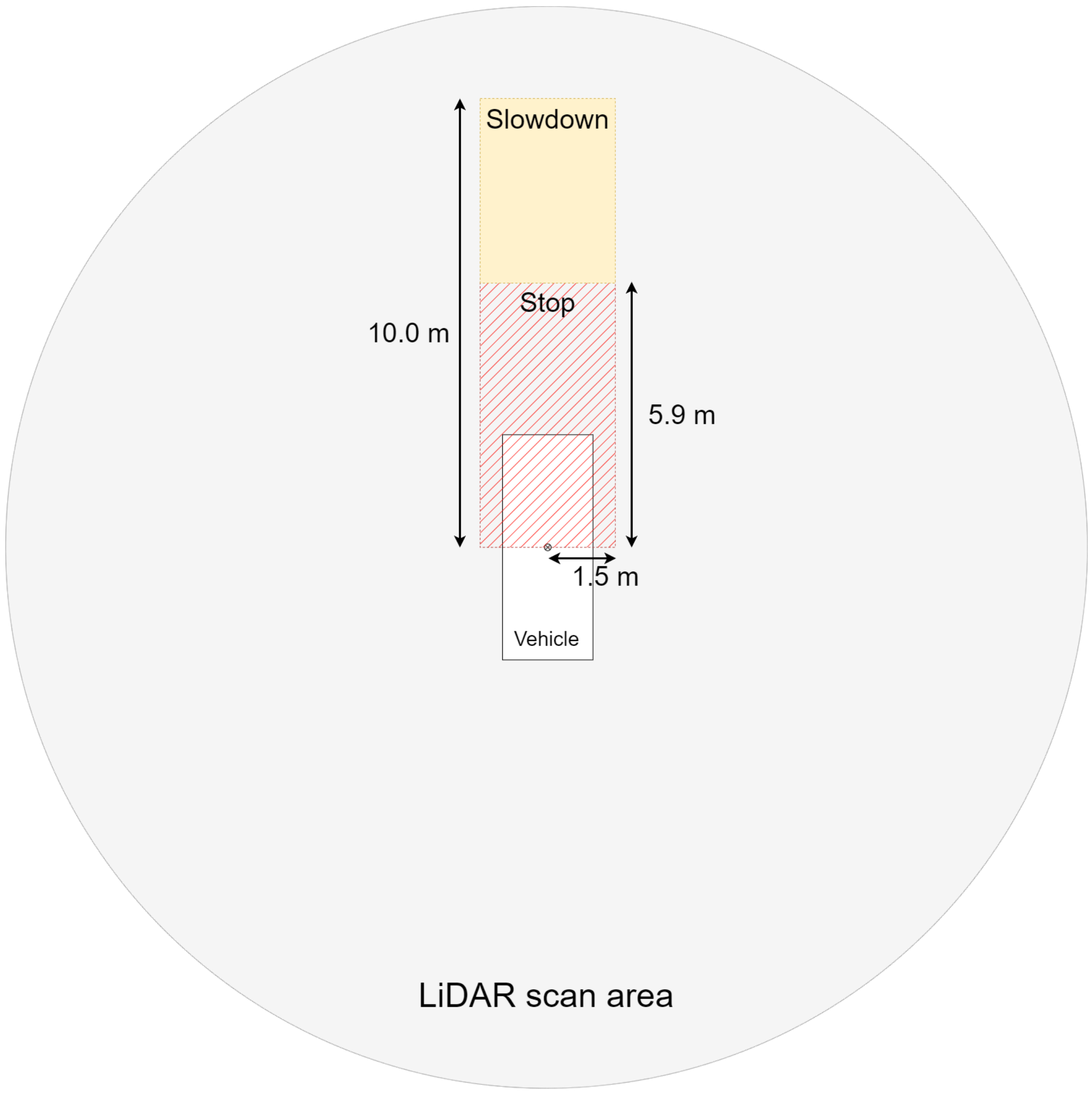

Figure 4.4. *CollisionAvoider issues a slowdown or a stop if LiDAR detects something in front of the car.*

command but does not prevent driving forwards completely. It is possible to bypass the restrictions from collisionAvoider by either pressing the bypass button on the controller or switching the drive gear to reverse.

Sony Dualshock 3 gamepad is used for manual driving (Figure 4.5). The left analog stick is used to control the throttle and braking: up to accelerate, and down to brake. The right analog stick is for steering. The cross button engages the handbrake. The square button bypasses the collision avoidance as long as it is pressed, and the triangle button puts the car into reverse gear as long as it is pressed.

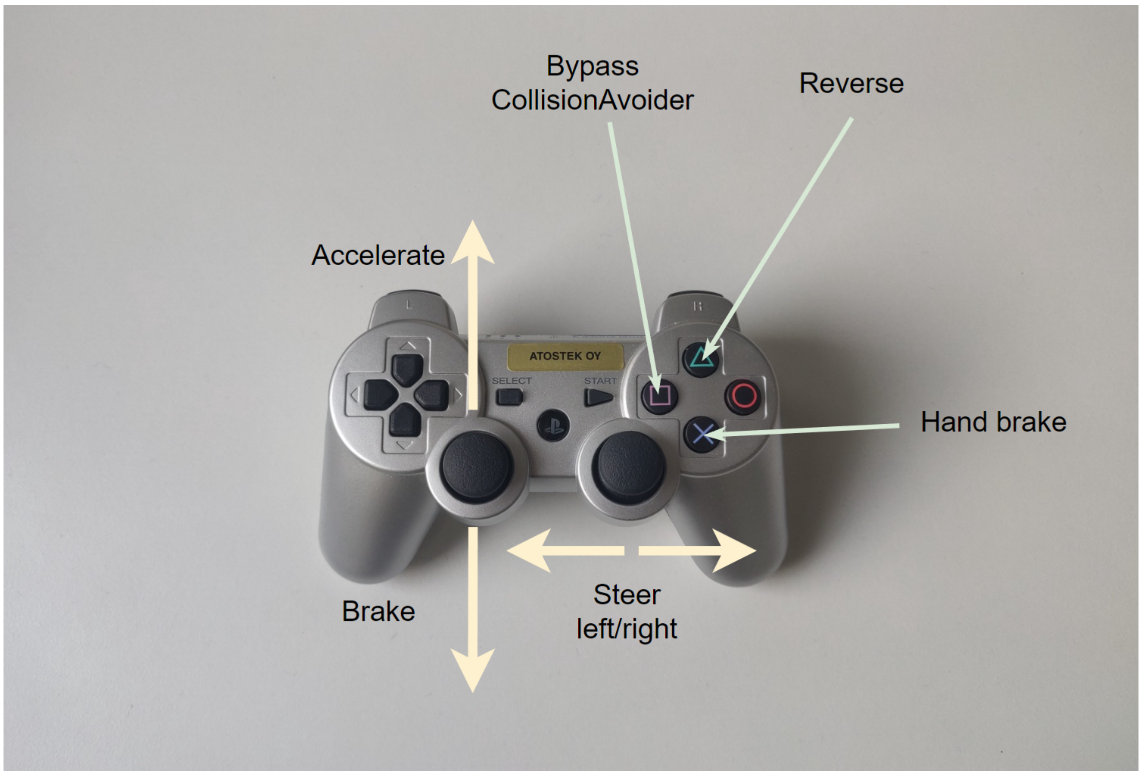

Figure 4.5. *Sony Dualshock 3 gamepad used for manual driving.*

5. RESULTS

Chapter 5 presents the results and the analysis. First, Section 5.1 describes the benefits of using software-only simulation, then in Section 5.2 challenges are outlined.

5.1 Benefits

Having a software-only simulation environment available when doing automatic driving development gives huge benefits. Developing a system, such as the example application developed in this work, that utilizes point cloud data from LiDAR sensors is nearly impossible without some kind of simulation or playback of real LiDAR data. While capturing real data and replaying it to test software might give realistic test scenarios, there are some problems with it. First, it is not interactive, and therefore, any reactions the ADS has to the data do not reflect in the replayed data. For example, when replaying point cloud data where collision with an obstacle would be imminent, and the ADS should avoid this by either braking or steering the vehicle away from the obstacle, the decision to act can be done from replayed data, but how the scenario would continue depends on the decision and replayed data cannot reflect that. With simulation, the data is always produced based on the scenario, and thus allows the testing of what happens after the decision is made, and ADS has reacted to the data.

Another problem with replaying data is that the amount of sensor data could be very large. One LiDAR scan can contain millions of points, and a LiDAR can scan tens of times per second. Without compression, it becomes very consuming to save that much data. Simulation, however, can produce a vast amount of data as needed without the need for saving huge chunks of data on a disk. Many simulation environments, including CARLA, support determinism, which enables reproducibility comparable to replaying saved data.

The key benefit of software-only simulation, when compared to HIL simulation, is the cost. While it probably loses in realism, it very likely wins in costs when setting up an environment, or scaling up the number of simulation instances.

Software-only testing enables testing very early in the development cycle. Simulation, in general, does not involve the same safety risks as real-world testing. Collisions and accidents in simulation do not break the vehicle or put anyone in danger. Therefore, a simulation environment can be used to test software that is unfinished or known to be broken. Each developer can have their own simulation environment they are using to test their code and make small iterative changes between the tests.

Building a simulator setup using dedicated hardware requires knowledge of the hardware components and how to configure and install them. Acquiring the hardware might be difficult and time-consuming. In 2022, the global pandemic and economic instability have increased the lead times of many hardware components. Given the component shortage, problems with global logistics are not going to disappear any time soon, so relying more on general multipurpose hardware and implementing the specialization in software is likely going to be a growing trend. Software-only simulation is doing exactly that.

The great scalability of software-only testing enables driving more test kilometers than what would be realistically possible with a physical test vehicle. Many machine learning based systems need to be trained with test data. Gathering test data from the real world is a very time-consuming task. First, it involves driving around and capturing the data. Then, someone needs to label the captured data. For example, when training a camera-based system that detects and recognizes objects it sees, one needs to capture images and then label the interesting objects in the images before they can be used to train the system. Software-only simulation makes this process easier. Capturing data from the simulation is easy, and does not involve nearly as much manual labor as gathering data from the real world does. Simulation tools such as CARLA know where and what each object is, and therefore it is straightforward for them to automatically label the data semantically as it is produced.

Using a simulated environment grants possibilities to automate the testing further. Integrating simulation into an automated testing pipeline allows executing integration tests already in the automated testing phase. In addition to just unit testing, SIL or HIL in a pipeline allows tests that have all the software components present and integrated. Automated testing could even utilize an algorithm to generate test scenarios automatically and evaluate the performance on a much larger scale than manually would be possible [46].

Automatic driving, especially if people are involved in the operating environment, is often a constraint with safety requirements stemming from standards and regulations. To fulfill those requirements, comprehensive validation testing is required. While final validation should be done using a real vehicle, simulation can be used to complement it. For example, some scenarios might be too difficult or dangerous to test with a real vehicle.

5.2 Challenges

The first challenge of software-only simulation is how to integrate the ADS being tested into the simulation, and at the same time have it compatible with a real vehicle. There is a need for underlying drivers that communicate either directly with the actuators controlling the vehicle, or some lower-level control system that does the controlling/driving of the vehicle. The simulation has to fulfill the same interface as the drivers or lower-level control system. It should be the same from the ADS perspective to communicate with real hardware and the simulation. It is worthwhile to consider both simulation and hardware integration when designing the software architecture and interfaces of an ADS system.

In a way, during this work, the challenge of integration was realized. There were challenges in establishing the ROS2 communication channel between the CARLA ROS Bridge and the Carlabot application. The problem stemmed from a lack of prior usage of ROS2 communication with the Flexbot framework. Flexbot framework is quite new, and it had not been used to this extent before to communicate with non-Flexbot ROS2 nodes. Some troubleshooting was needed and improvements to the framework were made to achieve stable communication.

The next challenges relate to the properties of a simulation environment. Simulation probably works too ideally. The real world is always more complex than what can be achieved with simulation. Therefore, there might be factors that make it more difficult for ADS to work in the real environment. Software that works perfectly in a simulation could fail miserably when being introduced to the real world. Being too dependent on simulation is not ideal, but combining it with real vehicle testing is beneficial. Simulation can help smooth out the biggest and most obvious flaws in the software before entering the physical world, which is often more time-consuming and expensive [33].

Setting up a meaningful simulation environment is not a trivial task. From the experience of this work, the initial setup of CARLA is quite quick and does not take too much work to get some level of simulation world running. CARLA is packed with a sufficient selection of maps, vehicles, and sensors. However, to gain the best benefit from simulation, and to use it in real product development, one probably needs to tailor the environment quite a lot. For example, matching the vehicle model to one that is available for real-world testing might need some modeling work. The world environments that come prepacked with CARLA are representing urban environments that might lack the distinctive local features that might be important for ADS development. For example, developing ADS for Finnish roads would probably require modifying and adding features that are present in Finland but might be missing or different in California, often used as a base model. Such features might include, for example, traffic signs or road markings.

If the configuration of the simulation environment requires a lot of manual work, setting up multiple instances of the environment takes up time and requires repeating manual configuration tasks. The risk of making mistakes increase, and it is not guaranteed that the environments are the same. After setting up an environment manually duplicating it might not yield the same result. Using scripts and configuration files helps in this, but might not completely solve it. Containerization, meaning using a virtualized isolated environment containing all the dependencies and configurations, is a potential solution for this challenge. For example, Docker or Podman container including the whole simulation environment would make the setup of an environment more effortless.

Carla has a weather system that affects the quality of data coming from simulated sensors. Unfortunately, however, the weather system only applies sensors that produce data by rendering such as cameras. The rain and fog effects are visible in the produced images, but not in data from ray casting based sensors such as LiDARS. In that sense, LiDAR data is a bit too ideal. There are mechanics to add noise and control the attenuation and loss of rays, but those mechanisms are probability-based. Luckily, however, due to CARLA being open-source, it would be possible to improve the sensor simulation and add such features.

Another caveat that could be improved by customizing the CARLA is the physics simulation. The physics in CARLA are provided by Unreal Engine 4. The video game physics it provides might not be realistic enough if physics is an important part of the testing. That also could be improved to some degree by modifying the simulator. However, CARLA does not aim to have perfectly realistic physics simulation, but instead focuses on delivering high-quality visual sensor simulation. If advanced physics simulation is essential, some other simulation tool might be more suitable for the task.

6. CONCLUSION

The goal of this thesis was to study software-only simulation for vehicle software development and analyze the advantages it provides over using a hardware-in-the-loop simulation or a prototype vehicle. To accomplish that, a demonstrator environment, and an example ADS application, called Carlabot, was developed. The environment was based on an open-source driving simulator CARLA, and the communication between the simulator and the example application was established using the Robot Operating System 2 (ROS2) middleware. CARLA simulator was accompanied by CARLA ROS Bridge which allows the simulator to communicate with ROS-based applications.

The example application, Carlabot, implements manual driving using a gamepad and a simple collision avoider, which utilizes point cloud data from a LiDAR sensor. The collision avoider slows or completely stops the driving if obstacles are detected in front of the car.

Carlabot utilizes the Flexbot software framework developed at Atostek Oy. Flexbot is a platform for developing node-based onboard robot control software. Flexbot provides tools to generate code for the node interfaces and communication channels automatically using a system structure specification. Flexbot supports multiple types of communication channels between the nodes, such as native Rust for communication within a process, or ROS2 for inter-process communication.

During the work, it was identified that assembling a software-only simulation environment using open-source components is not complicated when using predefined models and configurations. CARLA is documented fairly well and comes with an example application to get something interactive and moving in the simulation quickly. Establishing the communication with the Carlabot application using ROS2 was more time-consuming, and required some troubleshooting and debugging, as the Flexbot framework had not been used with ROS2 on this scale.

Using software-only simulation alone for testing ADS is not enough, but combining it with other testing environments brings considerable benefits to automatic vehicle development. It can be used as a first level of testing early on in the development cycle. Software-only simulation can empower the developer to test the software already during the development, and make small iterative changes between the tests. Including it in automatic testing pipelines allows automated integration testing. Overall, software-only simulation enables testing on a much larger scale and is cheaper than what is possible only using a real vehicle, or hardware-in-the-loop simulation.

In the future, one could further tailor the simulation environment to correspond better to a specific use case. In this work, the example application was a toy example of how the CARLA simulator could be used, and how to integrate to it. The world, the vehicle model, and the sensors used could be modified from the pre-defined ones to something more realistic. An interesting next step with the CARLA simulator would be to create and customize a simulation environment for an existing ADS.

Additionally, the modified CARLA server along with the ROS Bridge could be packed into a containerized environment using, for example, Docker or Podman. Containerization makes setting up the environment easier, and thus supports the efforts of using this kind of simulator environment in an automated pipeline or distributing it to multiple developers.

REFERENCES

[1] Li, R. and Zhai, R. Estimation and Analysis of Minimum Traveling Distance in Self-driving Vehicle to Prove Their Safety on Road Test. *Journal of Physics: Conference Series* 1168 (Feb. 2019).

[2] Yurtsever, E., Lambert, J., Carballo, A. and Takeda, K. A Survey of Autonomous Driving: *Common Practices and Emerging Technologies*. *IEEE Access* 8 (2020), pp. 58443–58469.

[3] Koenig, N. and Howard, A. Design and use paradigms for gazebo, an open-source multi-robot simulator. *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. IEEE, 2004, pp. 2149–2154.

[4] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A. and Koltun, V. CARLA: An Open Urban Driving Simulator. *Proceedings of the 1st Annual Conference on Robot Learning* (2017), pp. 1–16.

[5] *Cognata Autonomous and ADAS Simulation. Cognata*. URL: `https://www.cognata.com/simulation/` (visited on 03/03/2022).

[6] Thomas, D., Woodall, W. and Fernandez, E. Next-generation ROS: Building on DDS. *ROSCon Chicago 2014*. Open Robotics, Sept. 2014.

[7] Schaeuffele, J. and Zurawka, T. *Automotive Software Engineering, Second Edition*. SAE International, 2016.

[8] Gao, C., Wang, G., Shi, W., Wang, Z. and Chen, Y. Autonomous Driving Security: State of the Art and Challenges. *IEEE Internet of Things Journal* 9.10 (May 2022), pp. 7572–7595.

[9] Buehler, M., Iagnemma, K. and Singh, S. The 2005 DARPA Grand Challenge. The Great Robot Race. *Springer Tracts in Advanced Robotics* (2007), p. 552.

[10] *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles (SAE J3016). SAE International*. Apr. 2021. URL: `https://www.sae.org/standards/content/j3016_202104/` (visited on 03/22/2022).

[11] Stumph, R. Tesla Admits Current ‘Full Self-Driving Beta’ Will Always Be a Level 2 System: Emails. *The Drive* (Mar. 9, 2021). URL: `https://www.thedrive.com/tech/39647/tesla-admits-current-full-self-driving-beta-will-always-be-a-level-2-system-emails` (visited on 07/11/2022).

[12] *Toyota to Offer Rides in SAE Level-4 Automated Vehicles on Public Roads in Japan Next Summer. Toyota Research Institute Inc.* Oct. 24, 2019. URL: `https://global.`

toyota/en/newsroom/corporate/30344967.html?_ga=2.143289329.554576385.1657549111-450261683.1657549111 (visited on 07/11/2022).

[13] *Stellantis Shares Results of L3Pilot Automated Driving Project. Stellantis*. Oct. 13, 2021. URL: https://www.stellantis.com/en/news/press-releases/2021/october/stellantis-shares-results-of-l3pilot-automated-driving-project (visited on 07/11/2022).

[14] Ramey, J. Polestar 3 with Level 3 Autonomous Tech on the Way. *Autoweek* (Jan. 10, 2022). URL: https://www.autoweek.com/news/green-cars/a38737805/polestar-3-level-3-autonomous-ride-pilot/ (visited on 07/11/2022).

[15] *ISO 26262-1:2018 Road vehicles — Functional safety — Part 1: Vocabulary. International Organization for Standardization*. Dec. 2018. URL: https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en (visited on 07/15/2022).

[16] *ISO/SAE DIS 21434 – road vehicles — cybersecurity engineering. International Organization for Standardization, SAE International*. Aug. 2020. URL: https://www.iso.org/obp/ui/#iso:std:iso-sae:21434:ed-1:v1:en (visited on 07/21/2022).

[17] *SAE J3061: Cybersecurity Guidebook for Cyber-Physical Vehicle Systems. SAE International*. Dec. 2021. URL: https://www.sae.org/standards/content/j3061_202112 (visited on 07/21/2022).

[18] Oka, D. K. *Building Secure Cars: Assuring the Automotive Software Development Lifecycle*. John Wiley & Sons, Inc., 2021.

[19] *Simulink. MathWorks*. URL: https://se.mathworks.com/products/simulink.html (visited on 07/21/2022).

[20] *CarSim. Mechanical Simulation Corporation*. URL: https://www.carsim.com/ (visited on 07/21/2022).

[21] Joshi, A. *Automotive Applications of Hardware-in-the-Loop (HIL) Simulation*. SAE International, Aug. 2019.

[22] Yun, H. and Park, D. Virtualization of Self-Driving Algorithms by Interoperating Embedded Controllers on a Game Engine for a Digital Twining Autonomous Vehicle. *Electronics* 10.17 (Aug. 2021), p. 1..14.

[23] Martinez, M., Sitawarin, C., Finch, K., Meincke, L., Yablonski, A. and Kornhauser, A. *Beyond Grand Theft Auto V for Training, Testing and Enhancing Deep Learning in Self Driving Cars*. Dec. 2017. (Visited on 07/06/2022).

[24] Gregory, J. *Game Engine Architecture*. Third Edition. CRC Press, Taylor & Francis Group, 2018.

[25] *Unity. Unity Technologies*. URL: https://unity.com/ (visited on 06/06/2022).

[26] *Unreal Engine. Epic Games*. URL: https://www.unrealengine.com/en-US (visited on 06/06/2022).

[27] Shah, S., Dey, D., Lovett, C. and Kapoor, A. AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. *Field and Service Robotics*. July 18, 2017.

[28] *ABS Sensors. Apec Automotive*. Sept. 29, 2021. URL: `https://apecautomotive.co.uk/techmate-guides/abs-sensors/` (visited on 08/18/2022).

[29] Toth, C. K. and Shan, J. *Topographic Laser Ranging and Scanning: Principles and Processing*. Second Edition. CRC Press, 2018.

[30] Akenine-Möller, T., Haines, E., Hoffman, N., Pesce, A., Iwanicki, M. and Hillaire, S. *Real-Time Rendering*. Fourth Edition. CRC Press, Taylor & Francis Group, 2018.

[31] Hofbauer, M., Kuhn, C. B., Petrovic, G. and Steinbach, E. TELECARLA: An Open Source Extension of the CARLA Simulator for Teleoperated Driving Research Using Off-the-Shelf Components. *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, Oct. 2020, pp. 335–340.

[32] Stević, S., Krunić, M., Dragojević, M. and Kaprocki, N. Development of ADAS perception applications in ROS and "Software-In-the-Loop" validation with CARLA simulator. *Telfor Journal* 12.1 (2020), pp. 40–45.

[33] Brogle, C., Zhang, C., Lim, K. L. and Braunl, T. Hardware-in-the-Loop Autonomous Driving Simulation Without Real-Time Constraints. *IEEE Transactions on Intelligent Vehicles* 4.3 (Sept. 2019), pp. 375–384.

[34] *CARLA Documentation. 0.9.12*. URL: `https://carla.readthedocs.io/en/0.9.12/` (visited on 05/22/2022).

[35] *ASAM OpenDRIVE Standard*. Aug. 3, 2021. URL: `https://www.asam.net/standards/detail/opendrive/` (visited on 06/10/2022).

[36] *ASAM OpenSCENARIO Standard*. 2022. URL: `https://www.asam.net/standards/detail/openscenario/` (visited on 06/09/2022).

[37] Fremont, D. J., Kim, E., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A. L. and Seshia, S. A. Scenic: a language for scenario specification and data generation. *Machine Learning* (Feb. 2, 2022).

[38] Lopez, P. A., Behrisch, M., Bieker-Walz, L., Erdmann, J., Flötteröd, Y.-P., Hilbrich, R., Lücken, L., Rummel, J., Wagner, P. and Wießner, E. Microscopic Traffic Simulation using SUMO. *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, Nov. 7, 2018.

[39] Maruyama, Y., Kato, S. and Azumi, T. Exploring the performance of ROS2. *Proceedings of the 13th ACM SIGBED International Conference on Embedded Software (EMSOFT)*. 2016, pp. 1–10.

[40] *ROS 2 (Robot Operating System) Documentation. Open Robotics*. URL: `https://docs.ros.org/en/foxy/index.html` (visited on 03/03/2022).

[41] *RustDDS. Atostek Oy*. URL: `https://github.com/jhelovuo/RustDDS` (visited on 05/30/2022).

[42] Corsaro, A. and C., D. The Data Distribution Service – The Communication Middleware Fabric for Scalable and Extensible Systems-of-Systems. *System of Systems*. InTech, Mar. 2, 2012.

[43] *Autonomous Machines and Vehicles. Atostek Oy*. URL: `https://atostek.com/en/services/autonomous-machines-and-vehicles/` (visited on 10/07/2022).

[44] Daubaris, P., Helovuo, J., Makitalo, N. and Mikkonen, T. On ROS 2 Software Development Challenges. *Manuscript submitted for publication* (2022).

[45] Helovuo, J. Native Rust components for ROS2. *ROSCon Kyoto 2022 (to appear)*. Oct. 2022.

[46] Straub, J. Automated testing of a self-driving vehicle system. *2017 IEEE AUTOTESTCON*. IEEE, Sept. 2017, pp. 1–6.