Aalto University

MASTER'S THESIS 2019

Performance evaluation of a machine learning environment for intelligent transportation systems

Anton Debner

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.
Otaniemi, 27 May 2019

Supervisor:	Senior university lecturer Vesa Hirvisalo
Advisor:	M.Sc. (Tech.) Jussi Hanhirova

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Author
Anton Debner

Title	Performance evaluation of a machine learning environment for intelligent transportation systems
School	School of Science
Master's programme	Computer, Communication and Information Sciences
Major	Computer Science
Code	SCI3042
Supervisor	Senior university lecturer Vesa Hirvisalo
Advisor	M.Sc. (Tech.) Jussi Hanhirova
Level	Master's thesis
Date	27 May 2019
Pages	61
Language	English

Abstract

While automotive manufacturers are already implementing Autonomous Driving (AD) features in their latest commercial vehicles, fully automated vehicles are still not a reality. In addition to AD, recent developments in mobile networks enables the possibility of Vehicle-to-Infrastructure (V2I) and Vehicle-to-Vehicle (V2V) communication. Vehicle-to-Everything (V2X) communication, or vehicular Internet of Things (IoT), can provide solutions that improve the safety and efficiency of traffic. Both AD and vehicular IoT need improvements to the surrounding infrastructure and vehicular hardware and software. The upcoming 5G network not only reduces latency, but improves availability and massively increases the amount of supported simultaneous connections, making vehicular IoT a possibility.

Developing software for AD and vehicular IoT is difficult, especially because testing the software with real vehicles can be hazardous and expensive. The use of virtual environments makes it possible to safely test the behavior of autonomous vehicles. These virtual 3D environments include physics simulation and photorealistic graphics. Real vehicular hardware can be combined with these simulators. The vehicle driving software can control the virtual vehicle and observe the environment through virtual sensors, such as cameras and radars.

In this thesis we investigate the performance of such simulators. The issue with existing open-source simulators is their insufficient performance for real-time simulation of multiple vehicles. When the simulation is combined with real vehicular hardware and edge computing services, it is important that the simulated environment resembles reality as closely as possible. As driving in traffic is very latency sensitive, the simulator should always be running in real-time. We select the most suitable traffic simulator for testing these multi-vehicle driving scenarios. We plan and implement a system for distributing the computational load over multiple computers, in order to improve the performance and scalability.

Our results show that our implementation allows scaling the simulation by increasing the amount of computing nodes, and therefore increasing the number of simultaneously simulated autonomous vehicles. For future work, we suggest researching how the distributed computing solution affects latency in comparison to a real-world testing environment. We also suggest the implementation of an automated load-balancing system for automatically scaling the simulation to multiple computation nodes based on demand.

Keywords autonomous driving, machine learning, performance, game engine

Aalto-yliopisto

Tiivistelmä

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tekijä
Anton Debner

Työn nimi Robottiautojen tutkimukseen tarkoitetun virtuaalisen koneoppimisympäristön suorituskyvyn evaluointi							
Korkeakoulu	Perustieteiden korkeakoulu						
Maisteriohjelma	Computer, Communication and Information Sciences						
Pääaine	Computer Science					**Koodi**	SCI3042
Valvoja	Vanhempi yliopistonlehtori Vesa Hirvisalo						
Ohjaaja	DI Jussi Hanhirova						
Työn laji	Diplomityö	**Päiväys**	27.05.2019	**Sivuja**	61	**Kieli**	englanti

Tiivistelmä

Vaikka uusimmista automalleista löytyy jo itsestään ajavien autojen ominaisuuksia, robottiautot vaativat vielä runsaasti kehitystä ennen kuin ne kykenevät ajamaan liikenteessä täysin itsenäisesti. Robottiautojen ohella ajoneuvojen ja infrastruktuurin välinen (V2X) kommunikaatio ja tuleva 5G mobiiliverkkoteknologia sekä mobiiliverkkojen tukiasemien yhteyteen sijoitettavat laskentapilvet mahdollistavat liikenteen turvallisuuden ja sujuvuuden parantamisen. Tätä V2X kommunikaatiota voidaan esimerkiksi hyödyntää varoittamalla ajoneuvoja nurkan takaa tulevista pyöräilijöistä, jalankulkijoista ja huonoista tieolosuhteista.

Robottiautojen ja V2X kommunikaation hyödyntämistä on hankala tutkia oikeassa liikenteessä. Fyysisten autojen ja tieverkostoa ympäröivän infrastruktuurin rakentaminen on kallista, lisäksi virhetilanteista johtuvat onnettomuudet voivat aiheuttaa henkilö- ja tavaravahinkoja. Yksi ratkaisu on virtuaalisten testausympäristöjen käyttö. Tällaiset simulaattorit kykenevät mallintamaan ajoneuvojen käyttäytymistä reaaliaikaisen fysiikkamoottorin avulla ja tuottamaan valokuvamaista grafiikkaa simulaatioympäristöstä. Robottiauton ohjelmisto voi hallita simuloidun auton käyttäytymistä ja havainnoida simuloitua ympäristöä virtuaalisten kameroiden ja tutkien avulla.

Tässä diplomityössä tutkitaan liikennesimulaattorien suorituskykyä. Avoimen lähdekoodin simulaattorien ongelmana on niiden huono skaalautuvuus, eikä niiden suorituskyky riitä simuloimaan useita autoja reaaliajassa. Tässä diplomityössä tehdään lyhyt katsaus olemassa oleviin simulaattoreihin, joiden joukosta valitaan parhaiten yllämainittujen ongelmien tutkimiseen soveltuva simulaattori. Simulaattorin suorituskyvyn ja skaalautuvuuden parantamiseksi suunnitellaan järjestelmä, joka hajauttaa simulaattorin työkuorman useammalle laskentapisteelle. Kyseinen järjestelmä toteutetaan ja sen toimivuutta testataan mittaamalla.

Mittaustulokset osoittavat, että hajautettu laskenta parantaa simulaattorin suorituskykyä ja että reaaliaikaisesti simuloitujen autojen lukumäärää voidaan kasvattaa lisäämällä laskentapisteiden lukumäärää. Jatkotutkimukseksi ehdotetaan tutkimaan simulaation hajauttamisen vaikutusta viiveisiin, ja kuinka simulaattorin aiheuttamat ylimääräiset viiveet suhtautuvat tosielämän viiveisiin. Lisäksi suositellaan automaattisen kuormituksentasaajan toteuttamista, jonka avulla simulaatiota voidaan automaattisesti hajauttaa useille laskentapisteille tarvittavan laskentakapasiteetin mukaisesti.

Avainsanat robottiauto, simulaatio, koneoppiminen, suorituskyky, pelimoottori

Contents

1. Introduction

In 2017, 5G Automotive Association published a white paper [1] describing three example use cases for utilizing vehicle-to-anything (V2X) communication and Multi-access Edge Computing (MEC) for connected Autonomous Driving (AD). These use cases are 1) real-time situational awareness of road and traffic conditions combined with high definition maps. 2) Utilizing vehicular cameras from other vehicles to enable the host vehicle "see through" obstacles while passing them. 3) Using the infrastructure and other traffic users for Vulnerable Road User (VRU) discovery. VRU discovery can be used to detect and warn the driver about nearby VRUs, such as pedestrians and cyclists.

These use cases introduce strict requirements for the infrastructure. Both, communication links and the computation services must have very low latency, high availability and high reliability. In addition, the real-time requirement and the huge amount of data used by a single vehicle place requirements for the computational power and storage capacity. A single autonomous vehicle is estimated to generate as much as 4 000 GB of data every day by 2020 [29] in addition to data downloaded through V2X. As it is unrealistic for the local vehicular hardware to be able to store and process all of the data, cloud-based computation solutions are needed. Additionally, to satisfy the low-latency requirement, these computation platforms should be placed as close to the users as possible. In other words, it is necessary to use Edge Computing services that are placed near the mobile broadband provider access points. [1]

To be able to research the use cases of connected AD, the infrastructure and the vehicles must support this technology. While 5G [2] is being integrated in to the infrastructure in the near future, widespread vehicular Internet-of-Things (IoT) is still further away from reality. Even if the required technology was already tested and available, it would take time until a significant portion of the vehicles in traffic were capable of V2X communication. This slows down the research, development and implementation of connected driving.

Developing software for AD and vehicular IoT is difficult, especially because testing the software with real vehicles can be hazardous and expensive. One solution to this is to use virtual environments. The use of virtual environments makes it possible to safely test the behavior of autonomous vehicles. These virtual 3D environments include physics simulation and photorealistic graphics. Real vehicular hardware can be combined with these simulators. The vehicle driving software can control the virtual vehicle and observe the environment through virtual sensors, such as cameras and radars.

Virtual environments can also be used for training Machine Learning (ML) models. Both AD and connected AD benefit from the recent developments of ML and computing hardware. One key application of ML for AD is in perceiving the surrounding environment. While the perception of environment, *e.g.* lane markings, can be done using classical methods [33, 4], the use of pure ML models or a mix of ML and classical methods can bring better results [32, 56]. What is common with all image recognition and object detection ML models, is that they typically have to be trained with large datasets of annotated images. While there are several publicly available datasets collected from real world [13, 7, 16, 41, 62, 24, 37], virtual environments can also be used to generate datasets for training and testing ML models. Virtual KITTI [25] and Synthia [51] are existing examples of such datasets, and that ML models trained on virtual datasets can achieve decent results with real world data [10].

1.1 Research problem

The issue with existing open-source simulators is their insufficient performance for real-time simulation of multiple vehicles. Simulators are struggling with simultaneously rendering the simulation environment from multiple virtual cameras. To further highlight the issue, the hardware of latest partially AD-capable commercial vehicles can contain as many as eight on-board cameras, in addition to other sensors. Simulating even one such vehicle in real-time can be computationally too demanding. This is especially problematic when researching V2X applications, as vehicle-to-vehicle communication requires the presence of multiple, if not dozens or hundreds of vehicles.

While it is possible to simultaneously simulate more vehicles by running the simulation slower than real-time, we believe that being able to run the simulation in real-time is crucial. When the simulation is combined with real vehicular hardware and edge computing services, it is important that the simulated environment resembles reality as closely as possible. Automated driving is very latency sensitive and running the simulation slower than real-time can hide issues caused by latencies in the non-simulated parts of the researching environment. Additionally, increased performance is also beneficial for non-real-time applications, such as generating ML training datasets.

The goal of this thesis is to find a suitable existing open-source simulator and improve its performance to enable the real-time simulation of multiple vehicles.

1.2 Contributions

In this thesis, we compared several traffic simulators suitable for researching AD and vehicular IoT. From these simulators, we chose one named CARLA. Out of all considered simulators, CARLA appeared to have the highest amount and quality of relevant functionality and ready-to-use assets.

We improved the performance of CARLA simulator by improving its scalability. We planned and implemented a system for distributing the computational load over multiple computers. Our results show that our implementation allows scaling the simulation by increasing the amount of computing nodes, and therefore the number of simultaneously simulated autonomous vehicles.

In addition to the implementation and analysis of distributed simulation presented in this thesis, the author of this thesis contributed to the following three related research papers.

Research paper I: "A machine learning environment for evaluating autonomous driving software"

The author worked together with another research assistant on creating and measuring a system, that is a combination of CARLA and machine learning software TensorFlow. This paper [27] is partially based on these measurements.

Research paper II: "AI Accelerator Latencies in Hybrid Vehicular Simulation"
The author worked together with another research assistant on creating and measuring a system, that is a combination of CARLA and machine learning software TensorFlow. This paper [28] is based on these measurements.

Research paper III: "Scalability of a Machine Learning Environment for Autonomous Driving Research"
The author of this thesis worked as the first author on this paper [15]. This paper demonstrates the performance benefits of the scaling solution implemented in this thesis. The author made these performance measurements and the scaling solution.

1.3 Structure

The structure of this thesis is as follows. In the background chapter (chapter 2) we go over the necessary background information around the topic. We will then present the tools we have used in chapter 3 and select one of the available open source simulators in section 3.4. We discuss the structure of this simulator in chapter 4, followed by the technical details of our plan for distributed computing in chapters 5, 6 and 7. Finally, we present performance measurements of our implementation (chapter 8), followed by discussion (chapter 9) and a conclusion (chapter 10).

2. Background

2.1 Autonomous vehicles

The idea of autonomous vehicles has been around for a long time, but they have been in serious development since the advancements in vehicular machine vision in 1980s [17]. In the 21st century, two challenges held by DARPA showed [39] that AVs are capable of navigating in a desert road without any human intervention. In the first challenge (2004), the best teams were able to make only 5 % of the total trip. In the second challenge (2005), some participants were able to fully finish the race without any human intervention. Since these challenges, the development focus shifted to maneuvering in urban environments with other vehicles. For example, in 2011 the Grand Cooperative Driving Challenge [31] focused on cooperative driving in intersections.

The development of the most advanced features was enabled by the recent advances in machine learning. While features, such as forward collision warning systems and automated emergency braking features are possible without machine learning models, navigating in traffic among human drivers is an extremely complex issue. Machine learning can help with solving issues that are intangible and very hard to thoroughly define.

In the USA, a crash causation survey [57] came to the conclusion that 94 % of the surveyed traffic accidents between years 2005 to 2007 are estimated to be caused by driver error. From these cases, 41 % are attributed to recognition error and 33 % to decision error. Based on this, it is reasonable to claim that one of the main motivations for developing autonomous vehicles is to increase safety. In fact, European Union has set a long-term goal of reducing yearly traffic-related fatalities to zero by 2050 [38]. This long-term goal is split into concrete short-term goals that are periodically reviewed. The main mechanism for reducing fatalities is by increasing the autonomous capabilities of all vehicles and moving towards fully autonomous traffic. This includes both encouraging the research of new technologies and making current technology mandatory in all new vehicles.

For 2020s the European Union goals include situational automated driving in motorway and at low speed in cities. Current driver assistive systems, such as reversing camera/sensor, lane assist, intelligent speed assistance and automated emergency braking system will be made mandatory in all new vehicles. By 2030 25% of short trips in cities should be covered by shared automated vehicles. Another interesting detail is that by 2022 all new vehicles should be connected to the internet and that most of them should be capable of communicating with nearby vehicles and infrastructure. This means that vehicular IoT is closer to becoming a reality and V2X applications could potentially be utilized in all vehicles manufactured after 2022.

Society of Automotive Engineers (SAE) defines [52] six levels for automated vehicles. Levels from 0 to 2 include driver assistive technology, such as collision and lane departure warning systems. Levels 3 and 4 are reserved for highly automated vehicles, which can mostly operate without human intervention. Level 5 refers to full automation, which should be capable of handling all roadway and environmental conditions that are managed by a human driver. Level 5 no longer requires a human driver, pedals or even a steering wheel.

While current commercial vehicles can be equipped with all the sensors necessary for full level 5 autonomous driving, their software and processing power is lacking behind.

Vehicle manufacturer Tesla claims on its website [40] that all its vehicles "have the hardware needed for full self-driving capability at a safety level substantially greater than that of a human driver". This hardware includes 8 cameras providing 360 degrees of visibility up to 250 meters in range. In addition to this, it has a forward-facing radar and twelve ultrasonic sensors. However, an article from 2017 [36] claims that Tesla's autopilot software version 8.1 utilizes only two cameras out of the eight available. Also, Tesla's director of AI said on Q3 2018 that the vehicles are not able to use their latest neural network models due to computational constraints of the current hardware, but they plan to improve this in their upcoming hardware iteration [54].

Unlike Tesla, a Google's sister company Waymo appears to be using LiDARs (Light Detection and Ranging) in addition to cameras. Waymos vehicles are equipped with "three different types of LIDAR sensors, five radar sensors, and eight cameras" [46].

Autonomous vehicle control can be divided in route planning, decision making and short-term motion-planning and control. While the first depends on maps and can be computed offline, the rest are dependent on the real-time perception of the environment. Vehicle motion planning and control is discussed in depth in [47]. Perception is mainly based on machine vision, which has made huge advancements in recent years due to the progress in machine learning and deep neural networks [53, 56]. Machine vision can be used to recognizes and classify objects from the sensor data, *e.g.*, from LiDAR and cameras. This can then be used for Simultaneous Localization and Mapping (SLAM) to more accurately track the position of the vehicle and to create a virtual real-time map of the surrounding environment [66, 58, 30].

2.2 Game engines

In this section we will go over the basics of game engines in a depth that is required for understanding the methods used in this thesis.

Simulating autonomous vehicles requires the ability to simulate the physical dynamics of a vehicle and the ability to produce photorealistic images for the sensor feed. We would also like the ability to easily generate different types of traffic scenarios and to control the vehicles by either human or computer input.

In practice, this means that a subset of the real world, or a imaginary world, is mathematically modeled. These models can then be used to numerically calculate the next discrete simulation state, based on the previous state and elapsed time (delta time) since the previous state. [26]

The simulation is a soft real-time system, in a sense that missing a deadline is not catastrophic. These deadlines include updating physics above a desired minimum frequency to keep the physics stable. While missing a dozen physics update deadlines might be completely unnoticeable, constantly missing the deadlines may cause odd behavior. However, the simulation can usually resume normal operation if the physics update frequency rises back above the minimum. [26]

Game engines are frameworks for creating games. It can sometimes be difficult to distinguish a game from the game engine, as the component of the game engine might be built to be specifically suitable for one particular game. Gregory [26] suggests in his book "Game Engine Architecture" that the term "game engine" should be reserved for software that is extensible and suitable for many different games. In other words, we can use a game engine as a framework for the autonomous vehicle testing environment.

2.2.1 Modern game engines

Modern game engines typically comprise of separate modules for rendering, physics, audio, animations, AI and online multiplayer.

The main component of a game engine is typically the "game loop" that iterates over repeatedly. On each iteration, each module is allowed to update its state. While a modern game engine most likely runs these modules on separate threads, it usually makes sense to tie their update frequencies together. [26] For example, it does not make sense to render the game at a higher frequency than what the game state is updated. If the physics or game logic states have not changed, the subsequent renders would be identical to each other.

On the other hand, not all modules have to update at the same frequency. For example, it might be enough to update the AI only once every couple of seconds to avoid unnecessarily repeating identical calculations.

2.2.2 Game editor

Usually game engines include some sort of a graphical editor to make game level design easier. The editor allows the user to view the 2D or 3D scenery from a virtual camera, that can be placed in any angle or position. This view typically enables the manipulation of the scenery, for example by positioning new game objects in the scene.

The editor also enables easily editing the properties of each game object. These properties include position, orientation, mass, 3D model and scripts.

2.2.3 Physics simulation

For any physics-based game or simulation, one of the main parts of a game engine is the physics simulation engine. The physics engine is responsible for simulating physical systems, such as rigid body dynamics. This is usually done by numerically integrating mathematical models that are based on physical laws, such as Newtons second law of motion [67][26].

For example, the position of a rigid body under the effect of gravity could be computed as follows:

$$p_{n+1} = p_n + v_n * \triangle t$$

$$v_{n+1} = v_n + g$$

Where p_n is the position and v_n is the velocity at state n. The $\triangle$t is the elapsed time since the previous state and g is an approximation of the gravitational acceleration (typically 9.81 m/s). In a 3D environment, the position and velocity of an object can be stored as a three dimensional vector.

A typical physics step in a game engine iterates over every game object and updates its position based on its previous state and the current forces acting upon it[26]. In addition, each game object is checked for collisions with other game objects. Collisions can be resolved by, for example, directly adjusting their positions or by adding opposing forces to the overlapping objects until there is no overlap in subsequent iterations.

Collisions occur when the collider meshes (circles, spheres or more complex triangle meshes) of two separate game objects overlap. While a naive implementation of checking each object for collisions against every other object in the scene would be an $O(n^2)$ operation, there are many techniques for optimizing this process. Still, many game engines can be seen to slow to a crawl if there are lot of complex objects overlapping each other at the same time.

By default, modern game engines such as Unreal Engine 4 will only calculate the physics state s_{n+1}, if the state s_{n-1} has already been rendered [22]. It also cannot render the s_{n+1} if it has not yet been fully calculated. This means that computationally expensive physics step can restrict the rendering speed and vice versa.

To further increase the stability of physics calculations, a single physics step might be divided in smaller substeps. Each substep advances the state with a fraction of the total $\triangle$t. However, only the physics state from the final substep will be rendered. [22]

2.2.4 Game time versus real time

In this thesis, we are especially interested in real-time simulations. Therefore, it is important to define the difference between real time and game time.

Real time is the progression of time in the real world. The game time can be tied to the real time. This happens when on each physics update the $\triangle$t is equivalent to the real time that has passed since the previous physics update.

However, nothing stops the game engine from progressing the physics state by $time\_scale$*$\triangle$t, where $time\_scale$ is an arbitrary constant. If $time\_scale$ is less than 1, the game time will progress slower than real time. Likewise, if $time\_scale$ is larger than 1, the game time will progress faster than real time.

The physics engine also often has a configurable maximum for $\triangle$t, which might affect the progression of game time in some cases. As the physics simulation is numerically integrated, it can become unstable with too large values for $\triangle$t. For example, if an object with 1 meter radius is moving at 10 m/s and the $\triangle$t is 1 second, the object would move 10 meters on each physics step. As a result, it could incorrectly move through a 8 meter thick wall without the game engine ever detecting a collision. By restricting the $\triangle$t to a more reasonable 1/30 seconds, the collision would occur as expected. However, if the physics step takes longer than the maximum allowed $\triangle$t, the game will no longer progress in real time.

2.2.5 Non-deterministic physics simulation

As a result of the real-time requirement and floating point precision errors, game engines are typically not deterministic. Running the exact same physics simulation multiple times might not always end up with identical results.

Instead of advancing physics state by a fixed delta-time of, *e.g.*, 1/60 seconds per update, the frequency varies on the currently available resources of the computer and the complexity of the physics calculations. After one second, instead of having calculated 60 fixed-length updates, there might be any amount of updates in range of, *e.g.*, 1/10 to 1/200 seconds with the total sum of delta-time being one second. As the delta-time varies randomly, the floating point precision errors will begin to pile up. This is usually not an issue in video games, but it might become an issue in scientific simulations that are expected to be fully repeatable.

Some game engines can be configured to be deterministic by forcing a fixed update frequency. However, missing deadlines with a fixed frequency will cause the simulation to slow down in comparison to real-time. The simulation would always advance by 1/60 seconds, even if more time had elapsed since last update.

2.2.6 Rendering

Rendering is the process of generating images from 2D or 3D models. In its simplest form, it can be performed by shooting a ray from each pixel of a virtual camera. The resulting color of a pixel is retrieved from the first object that the ray collides with. As a single image consists of millions of pixels, rendering is usually done with the GPU. This is because each pixel requires very similar calculations and GPU is ideal for executing massive amounts of similar calculations in parallel. [26]

In order to make the rendering appear realistic, we have to take the physical properties of the light and materials of the reflective surfaces in to account. An object reflects light differently when viewed from different angles, depending on the direction of incoming light. This can be calculated using the Bidirectional Reflectance Distribution Function (BRDF) [42]. Modern game engines aim to use the concept of Physically Based Rendering (PBR) [49], which uses BRDF as one of the key principles. In other words, PBR is the attempt to model the flow of light as realistically as possible. Some effects that affect real-life cameras, such as lens flare and depth-of-field can be added as a post processing effect to the rendered image. It is important to note that even when using PBR techniques, the rendered image depends on the quality of the 3D-models, their materials and textures.

As games run in real-time, game engines have to balance between the quality and speed of the rendering pipeline. This means that PBR techniques are often high-performing approximations of the underlying physical model [26, 49]. In order to make games visually pleasing to look at, they typically render between 30 to 144 frames per second. In context of real-time vehicular simulation, the frame rate should be high enough to match the update rate of real-life sensors.

3. Tools

3.1 Unreal Engine 4

Unreal Engine 4 is the fourth major version of the game engine made by Epic Games. It is publicly available, free to use and its source code can be viewed and modified freely. However, it is not open source and its users might have to pay royalties based on the revenue made by using Unreal Engine. [22]

Epic Games uses Unreal Engine for its own popular multiplayer games, such as Unreal Tournament series [64] and Fortnite [19]. As a result, the game engine has been modified to include networking features suitable for hosting at least 100 concurrent players and 40 000 replicated simulation actors in real-time on a single server instance. [65] [21]

Unreal Engine is one of the few publicly available game engines that includes state-of-the-art technology. As an example of such technology, in 2018 NVIDIA and Epic Games showcased new real-time raytracing technology, RTX, being used in Unreal Engine [14]. Additionally, Unreal Engine supports NVIDIA GameWorks [44], which is a SDK for various advanced visual effects, physics simulation and rendering techniques. Unreal Engine is also constantly receiving updates and new features from Epic Games. It also has a very active community and its users can make pull requests to its git repository or release their modifications as additional plugins. [22]

Unreal Engine uses NVIDIAs PhysX [45] for the physics simulation [23] and supports the most popular graphics APIs, such as DirectX, OpenGL and Vulkan. It also supports multiple platforms, such as Windows, macOS, Linux, Android, iOS, most recent game consoles and virtual reality platforms. [22]

Unreal Engine source code consists mostly of C++, but it also offers a high-level visual scripting system called *Blueprint*. [22]

3.2 Unreal Engine networking system

This section, and its subsections, are mostly based on the official Unreal Engine networking documentation in [21].

Unreal Engine 4 includes a system for synchronizing the game state between the game server and game client instances in multiplayer online games. In this thesis, we will use this system for distributing the rendering load.

The system is based on an authoritative client-server model, where the authoritative server is responsible for making all meaningful actions. While the clients are running the game locally, all actions are sent to the server. The server then responds to these actions and sends the updated state to the clients.

The system can run in four different modes:

- Standalone mode runs both the server and the client locally on the same game instance. This is used for singleplayer games and the server will not accept connections from other clients.
- Dedicated Server is running the server without any human-interaction capabilities, such as sounds, graphics or input. It can be used for situations that require a high-performing server.
- Listen Server is running both the server and the client locally, but it is also accepting incoming connections from other clients. In other words, listen server can be used by one of the players to host a multiplayer session.
- Client mode will not run any server-side logic and must be connected to an external server.

Basic principles

While the system is capable of replicating the state from the server to the clients, the replication logic must be defined by the developer. Actors and their variables are not replicated unless specifically requested. This is partly to save bandwidth. It makes no sense to spend bandwidth and cpu cycles on transferring data that can be cheaply calculated locally or loaded from disk. On the other hand, clients do not necessarily need to know everything that the server is doing.

Replicated properties can be either defined at low-level c++ code by declaring functions and variables with special *UPROPERTY* keywords, or with the high-level Blueprint visual scripting language.

All properties that are marked as replicated, will be replicated to the clients when the value is changed. Again, it does not make sense to waste bandwidth replicating property values that have not been changed.

These properties can be modified locally by the clients, but the changes will not apply to other game instances. When the client receives an updated value for this property from the server, any local changes will be overwritten.

There are three different types of replicated function calls

- *Client* functions are executed at a specific client, when called by the server.
- *NetMulticast* functions are client functions that are executed by all clients.
- *Server* functions are executed at the server, when called by a client.

There are however restrictions on how the functions can be called. For a client to be able to call server functions, the client must have ownership over the actor-object that implements the function.

Optimizations

Depending on the available bandwidth and processing power, it might not be possible to update every object on every update iteration. This is why the networking system can be configured to prioritize certain actions. Each actor can be given a priority, which affects the update frequency. Actors with high priority will be updated more often than others.

Actors can also be defined as not-relevant for certain clients. This also helps to save bandwidth, as it might not be necessary for every client to be aware of all replicated objects.

It is also possible to define a custom replication driver, which allows to fine-tune how the replication works.

TCP versus UDP

The networking system uses UDP for transferring data between the server and the clients. This makes sense, because in multiplayer games it is typically vital to keep the latency low at the cost of dropped packages. It is important to make the player feel that the player's character responds to input immediately. Additionally, there is usually no point in resending dropped packages as they are no longer relevant. This can be compared to streaming a video: There is no point in receiving and processing video frame *n*, if we have already displayed frame *n+1*.

3.3 Unity

Alongside Unreal Engine, Unity [60] is another popular cross-platform modern game engine. Unity is used in some of the simulators that were looked at during this thesis.

It is being developed by Unity Technologies and it was first released in 2005. Similarly to Unreal Engine, it has a very active community and it is being constantly updated with new features.

Unlike Unreal Engine, Unity is fully based in C# and the user-made extensions and game logic are also written in C#. While Unity does not contain a Blueprint-like visual scripting tool, C# can be seen as more user-friendly than C++, as it is a higher-level language with automatic memory management. Unity also uses NVIDIA PhysX for physics simulations.

Unity has publicly released its source code, but only as a read-only license. Modifications are not allowed.

3.4 Simulation environments

We chose to focus on CARLA [18] in this thesis, due to the amount of relevant features CARLA offers, the fast-paced ongoing development and the active community around it.

In this section, we will go over the main features of CARLA and briefly compare it to alternative autonomous vehicle simulation software.

3.4.1 CARLA

CARLA [18] is an open-source platform for autonomous driving research. It was first released in 2017. It is under constant development, and several new releases were made while writing this thesis. CARLA is built on top of Unreal Engine, which means that it is capable of producing high quality graphics and realistic physics simulation. It has a very active community on both GitHub and instant messaging platforms, which complements the partially incomplete documentation.

Its features include pedestrians and vehicles equipped with naive AI, that is capable of navigation and following basic traffic rules. CARLA also provides free-to-use digital assets, such as several vehicle models, buildings and a variety of urban decals such as trees, trash bins, benches and bus stops. It also includes basic traffic control objects, such as functional traffic lights, speed-limit signs and stop-signs. As a result, CARLA is capable of modeling a lively traffic in an urban city out-of-the-box, without any need for configuration. CARLA also supports several different weather conditions, such as light or heavy rain and different times of day. It also attempts to model glossy reflections from the surface of wet road during rain.

While CARLA is meant for researching autonomous vehicles, the naive AI is not meant to represent any realistic autonomous vehicle software. Instead of simulating computationally expensive sensors, the AI directly utilizes the physics state retrieved from Unreal Engine. As a result, the AI handles normal traffic scenarios with perfect knowledge of the surrounding world and a very low computational load. A decent computer can handle dozens, if not hundreds, of simultaneous pedestrians and vehicles with decent performance.

The naive AI can be replaced by communicating with the CARLA client API over TCP. This API can be used to send control input to the simulation actors, to receive information about the simulation state and to receive data from freely placeable sensors. The API is a mix of Python and C++, allowing the easy use of the massive amount of tools available for Python, such as Tensorflow and its higher level APIs. On the other hand, C++ can be used when maximum performance is important.

Currently CARLA includes three visual sensors: an RGB camera, a rotating LiDAR (Light Detection and Ranging) and a sensor capable of detecting objects in front of it. These sensors are meant to imitate sensors that can be utilized in real, physical autonomous vehicles. It is important that these sensors function as closely to their real-life counterparts as possible, as this enables the testing of real autonomous vehicle software in simulated environment. Ideally autonomous vehicle AI should function identically in real and simulated environment. In addition to these real-life sensors, CARLA provides tools for detecting collisions and lane invasions. Lane invasion is a scenario, where a vehicle drifts to the wrong lane on a road. CARLA also offers access to the physical state of each actor, including its position, velocity and orientation in the 3D environment. While the AI should not utilize any information that cannot realistically be acquired in real-life, this additional information is very useful for validating the actions of the AI. It can be used to automatically detect misbehavior and it can be used to provide training data for machine learning models.

3.4.2 Other simulation environments

In addition to CARLA, other simulation environments were also considered. There are so many simulators related to the research of autonomous vehicles, that only some are discussed here. The general impression is that there is a lot of on-going development in the area of these simulators. Even during writing this thesis, some simulators received major updates and at least one transitioned from closed-source to open-source. All of these vary in offered features, which were more or less relevant to our interests. The largest differences come from the choice of the game engine, the amount of features and the amount of premade assets offered. For example, some simulators were rejected simply because they use out-dated game engines that are unable to provide realistic graphics. A majority also lacked the support for simulating and controlling multiple vehicles simultaneously.

Airsim
Microsoft's Airsim [55] is yet another open-source simulator for autonomous vehicles. It is mainly developed for Unreal Engine, but it also has an experimental release for Unity. It supports multiple simultaneous vehicles, different weather effects and lightning depending on time of day. As the name suggests, it also includes accurate aerodynamic simulation for aerial vehicles in addition to ground vehicles.

It is very similar to CARLA in terms of features, and it could have been a good alternative for the purposes of this thesis.

BeamNG.Research
From all other simulators introduced in this section, BeamNG.Research [5] focuses most on in-depth physics calculations. These calculations include the temperature of each component in the engine, the transmission of power to the ground through a detailed drivetrain, and even the deformation of rubber in tires as forces are applied. It also features soft body physics, meaning that collisions can dynamically bend, break and tear

the hull and components of the vehicles. The parts are not only visually deformed, but they also affect the physics simulation in mostly realistic manner. BeamNG.Research uses their own physics algorithms on top of Torque3D [59] game engine.

While BeamNG.Research outperforms other simulators in physics simulation, the computational power required to calculate these physics is significantly higher. This will likely be an issue with real-time simulation of multiple autonomous vehicles, which is why it is not the best candidate for our goal of a collaborative driving research platform.

Deepdrive

Similarly to CARLA, Deepdrive [50] is an autonomous vehicle research platform developed on top of Unreal Engine. Key difference to CARLA is the optimized rendering pipeline, which can be significantly faster than what CARLA offers. This is because the sensor rendering pipeline uses shared memory, instead of TCP, for transferring the sensor data to the vehicle AI. On the other hand, using shared memory introduces a limit to the overall amount of data the sensors can send on each game loop iteration.

While more efficient rendering is a desired property, Deepdrive lacks in other features, such as pedestrians, traffic lights, non-autonomous vehicles and prebuilt urban cities. In other words, it does not offer as complete traffic simulation as CARLA.

On the other hand, it could be worthwhile to try to integrate the rendering pipeline from Deepdrive to CARLA. This would require significant changes to the client-server communication of CARLA and is outside of the scope of this thesis.

The rest of the simulators

NVIDIA DRIVE Constellation [43] is a simulator by NVIDIA, that is only available to NVIDIAs partners. It is advertised as being able to produce high-quality graphics and being compatible with on-board vehicular hardware provided by NVIDIA. As it is not publicly available, it was not suitable for use in this thesis. It appears to be also running on Unreal Engine, but there is no clear information about it.

Baidu Apollo (Not really a simulator, more like a vehicle AI platform that can be connected to real or simulated vehicles)

Gazebo [34] is a multipurpose robotic simulator, which has also plugins for autonomous vehicle, pedestrian and city simulation. However, the graphics engine seems to be out-dated in comparison to Unreal Engine -based simulators.

Webots [63] is another multipurpose simulator, that offers a variety of tools and assets for autonomous vehicles. Its first open-source release was published during the making of this thesis and was therefore not considered as a viable option.

4. Structure of CARLA

In this chapter we go over the architecture of CARLA, in order to set a basis for the changes required to distribute the computational load over multiple computers.

4.1 High-level architecture

As illustrated on Figure 4.1, the CARLA architecture can be seen as two separate components: the CARLA client and the CARLA server. The CARLA client can be used to control the actors inside the simulation and to receive data from the sensors, such as on-board cameras connected to the vehicles. The client is connected to the CARLA server over TCP. The server is a plugin for Unreal Engine that coordinates the simulation inside the Unreal Engine simulation instance. The server reacts to control inputs received from the client, while the Unreal Engine is responsible for calculating physics and rendering graphics. As of CARLA 0.9.0, multiple clients can be connected to a single server simultaneously and a single client can control multiple actors.

While one CARLA server supports multiple client connections coming

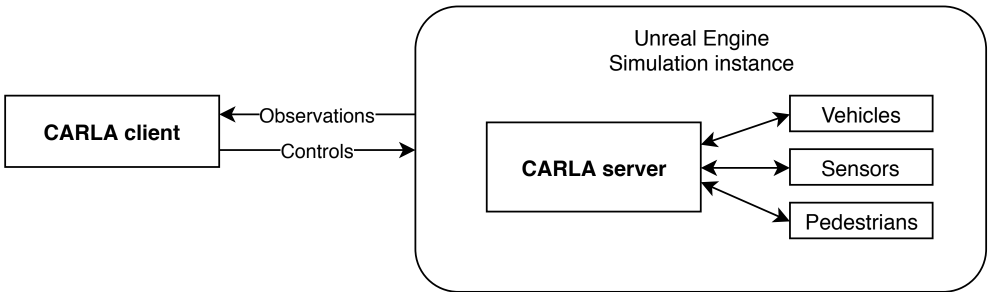

Figure 4.1. Original high-level architecture of CARLA 0.9. CARLA server is a plugin for Unreal Engine that contains all the logic and tools for running the traffic simulation. CARLA client can control the simulation actors and receive live sensor data from them over TCP. Multiple clients can communicate with the server simultaneously.

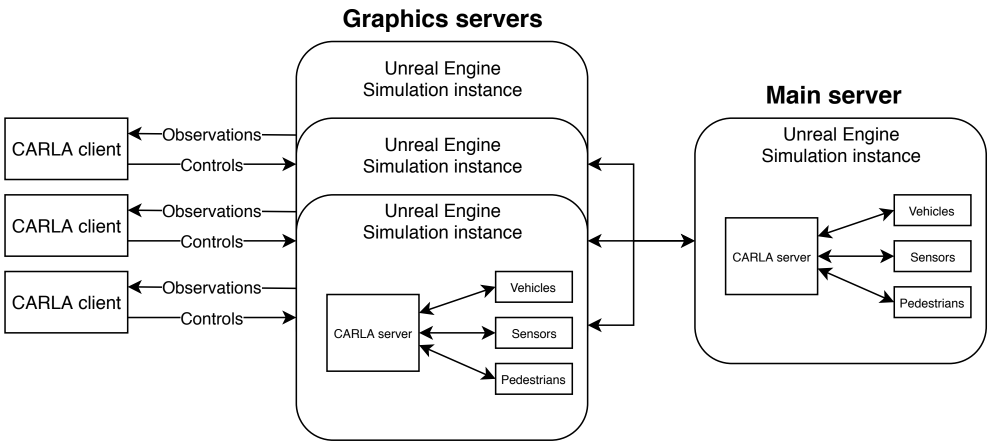

Figure 4.2. Improved architecture of CARLA for distributing the computational load. The main server holds the ground truth of the simulation state. This state is synchronized to any amount of client servers existing on separate computers. Computationally intensive sensors, such as cameras, are dedicated to a single server at a time, allowing the computational load to be divided among all servers. The CARLA clients can communicate with any server, including the main server.

from multiple computers, the simulation itself can only be run on a single computer. Therefore, the performance of the simulation is limited by the hardware resources of a single computer. The goal is to remove this restriction by implementing a system for distributing the rendering load over multiple computers, as illustrated in Figure 4.2. From this figure, we can see that the idea is to have a scalable amount of servers for graphics rendering and one main server for calculating the simulation state. The simulation state will always be synchronized between the main server and the rendering servers. In other words, every actor in the simulation world should ideally have the same position and velocity on each of the servers at every point in time.

4.2 Low-level architecture

On the UML diagram in Figure 4.3 we can see the main components of CARLA, including a system for spawning Actors, a system for storing references to such Actors, a system for tagging these Actors for custom rendering, a system for sending information about these actors (pos, vel, acc) and the actual server which manages the RPC and sensor data streaming connections.

CarlaEpisode contains information about the current episode and methods for initializing a new episode. In this context, episode means an arbitrary simulation scenario with a specific simulation map, weather and simulation actors. A new episode begins when the simulation is reset to

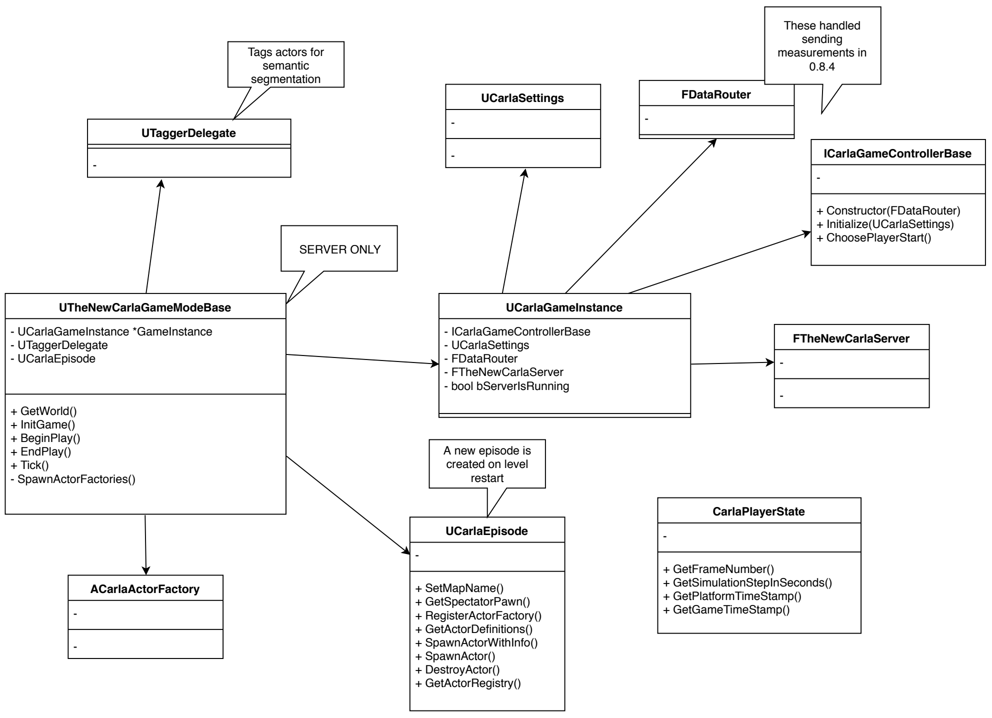

Figure 4.3. A sketch UML diagram of the most important components of CARLA used during the initialization. As CARLA 0.9.0 introduced a new communication protocol between the server and the client, this diagram contains a some amount of deprecated classes that are no longer in use, but have to be accounted for.

the initial state.

CarlaServer implements the server-side of the API used for communicating between the simulation and the CARLA clients. It is responsible for opening and listening to a given TCP port and for streaming sensor data to the CARLA clients through another port.

ActorRegistry implements a searchable data-structure for storing the reference and metadata of all CARLA-related simulation actors. It maps an ID-pointer pair for each actor, where the ID is an integer. This enables the CARLA clients to refer to the simulation actors by their unique IDs. This also enables easy listing and deletion of all existing simulation actors.

ActorFactory allows spawning new simulation actors to the simulation by their name at runtime. It dynamically creates a database of all possible simulation actors, including their names and alternative options, such as different colors of a vehicle. All actors spawned through the ActorFactory are registered to the ActorRegistry.

4.3 Execution timeline

Each game loop iteration (aka. "tick" or "update") consists of three distinct parts: rendering, game logic and physics. As game logic and physics

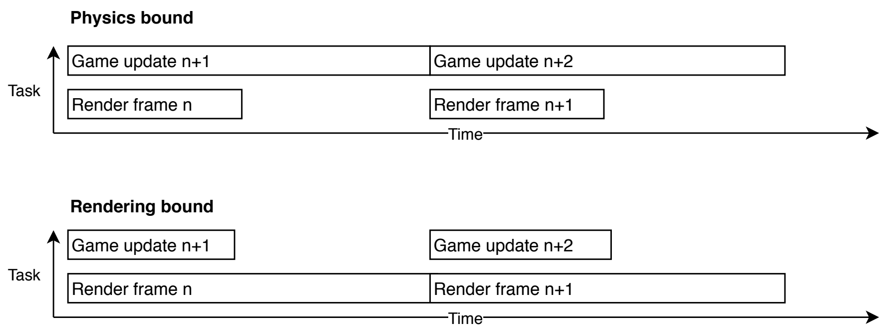

Figure 4.4. This figure shows how the performance of the simulation is always limited by either rendering or game state update. The game update consists of physics simulation and executing all game logic code, including the CARLA server. Rendering is always performed for frame n-1, while the game engine is computing the state n. This is because a game state can only be rendered after it is fully computed.

depend on each other, they are often executed sequentially. This is of course a slight simplification, as Unreal Engine offers the ability to define whether a specific game logic code is executed before, during or after physics calculations [20]. However, the point is that game update and rendering can be seen as two separate "blocks" that are done in parallel.

The performance of the simulation depends on both of these blocks, as the next update will only begin after both blocks have been fully completed. Figure 4.4 demonstrates both of these possibilities. Usually the physics simulation takes up most of the time in the game update step, as physics simulation of multiple vehicles is computationally expensive.

In this thesis, our goal is to improve the performance in those cases where the simulation is rendering bound. This is possible, because the rendering load depends on multiple independent sensors. As the sensors do not affect each other, they can be independently rendered on separate rendering nodes. Distributing the game logic or physics calculations would be a completely different problem, because the physics state of each object can affect every other object in the simulation.

5. Plan for distributed simulation

Three different approaches for distributing the rendering were considered during this thesis. Options 1 and 2 utilize the features of the Unreal Engine for synchronizing the simulation state between the different simulation instances. Option 3 would be to implement a completely new system for synchronizing the state.

Option 2 was chosen for the ease of use and the minor impact on the client API.

5.1 Option 1: Two client connections

Option 1 would be to keep most of the CARLA server functionality only on the main server and to use the rendering server simulation instances only for retrieving sensor data, as illustrated in Figure 5.1. This option requires the least development effort, because most of the features are already working on the main server. We would only need to ensure that the sensors work correctly on the client simulation instances and that the simulation state is correctly synchronized using the Unreal Engine replication system.

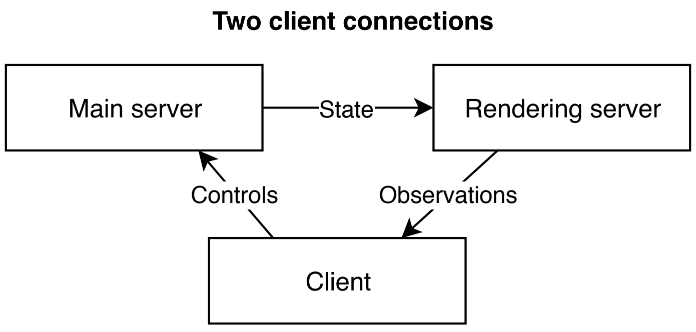

Figure 5.1. Option 1: Client connects to both the main server and the rendering server. Controls are sent to the main server, while sensor feed is retrieved from the rendering server.

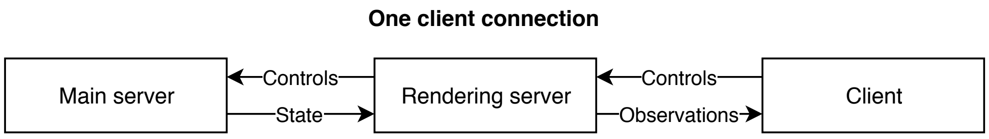

Figure 5.2. Option 2: Client connects only to the rendering server. The rendering server redirects the controls to the main server.

The downside of this approach is that the CARLA client would then need to manage two separate connections, one for the main server and one for the rendering server. This would either need drastic changes to all existing clients, or major modifications to the client API to enable automatic connection management. In either case, the client must be aware of the IP addresses of both the main server and the rendering server.

5.2 Option 2: One client connection

Option 2 reduces the number of TCP connections from client to one, as illustrated in Figure 5.2. The simulation state is synchronized using the Unreal Engine replication system. The sensor output is retrieved straight from the rendering server, but the controls are relayed to the main server through the rendering server.

This simplifies the system from the user's perspective, as clients only need to manage one TCP connection at a time. The client also does not even need to know whether it has connected to the main server or one of the rendering servers, as the communication is handled the same in both cases. This approach is also closer to the original architecture of CARLA, making it backwards compatible with all setups that only utilize a single server.

The disadvantage is that all controls have to pass through the rendering server, instead of being directly sent to the main server. This introduces additional latency. However, the latency may be reduced if the Unreal Engine can extrapolate the next simulation state from the controls, before it receives the updated state from the main server.

5.3 Option 3: Custom system

The third option is to create a custom synchronization system without using the existing features of Unreal Engine. The CARLA client-server

API already offers features for getting and setting the state of every actor in the simulation. Therefore, the simplest implementation would be to build a specialized client software, which retrieves the state from the main server and then sends it to the rendering servers. This would offer an easily malleable solution, as the client can be modified without recompiling the CARLA server. It would also be easier to modify the synchronization at runtime, depending on the needs of the client.

However, latency-wise it would be better to implement this functionality straight in the CARLA server. This way the data would flow directly between two servers, without needing the client in the middle.

The disadvantage is that sending the full simulation state on every frame would be computationally expensive and would consume needless amounts of network bandwidth. Therefore, this approach probably requires a significant amount of optimization to be better than the Unreal Engine replication system.

The advantage of this approach is full control over the synchronization process. For example, it could be implemented using TCP instead of UDP, solving the danger of dropping packages. It is also possible to design the optimizations in such way that best benefit the use-case of CARLA. For example, by only synchronizing actors that are nearby some user-defined points-of-interest, such as certain vehicles or intersections.

It is also good to notice that Unreal Engine supports creating a custom replication pipeline, however, this is outside of the scope of this thesis.

6. Connecting multiple simulation instances

The Unreal Engine contains a feature for easily connecting separate instances together. The server can be launched by giving it the name of the desired simulation scene file and a special *?listen* command as follows:

```
CARLA.sh level-name?listen
```

The server is now running as a *listen server*, meaning that it can accept connections from other instances while also acting as a client itself. [In contrast to a dedicated server, which is strictly a server without any graphical elements].

The clients can then be connected to the server by giving the IP of the server. During the connection process, the clients download the simulation scene from the server.

```
CARLA.sh main-server-IP:port
```

It would also be easy to implement a new command in the CARLA client API for connecting the client instances to a server at a specific IP address during runtime. This would enable the user to switch the server at runtime, without restarting the entire simulation software.

6.1 Running a unique CARLA server on each simulation instance

Connecting the simulation instances together with the default Unreal Engine implementation is not enough. While the vehicles created and moved on the main server can also be seen on the rendering servers, there is no way to control anything on the rendering servers.

This is because the Unreal Engine networking capabilities are designed for an authoritative server-client model, which means that only the authoritative main server is authorized to control the simulation/game. This

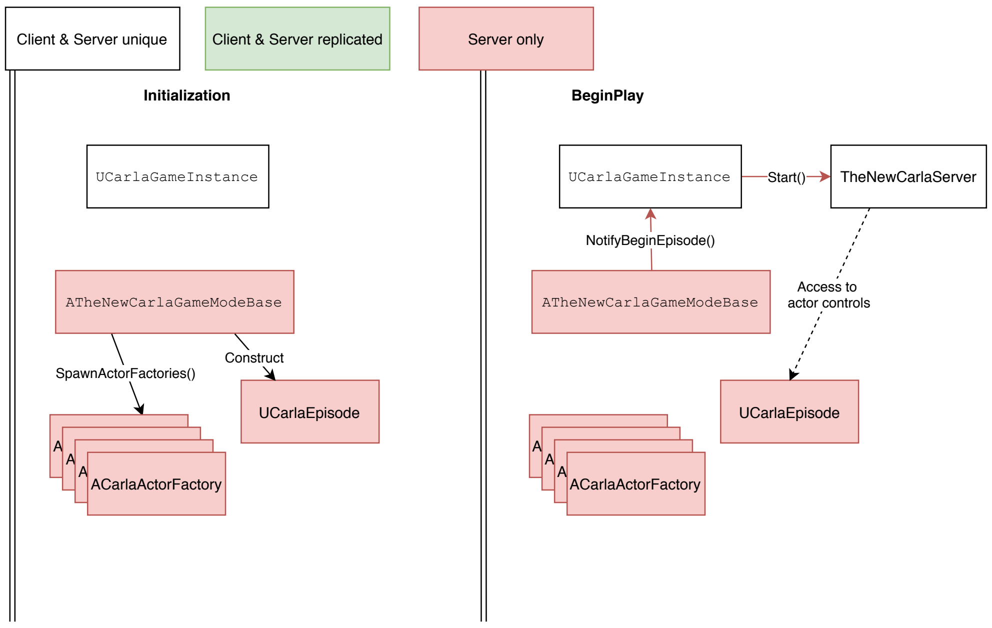

Figure 6.1. CARLA server initialization sequence in 0.9.0. The initialization can be divided in 4 separate steps: starting up the Unreal engine process, loading the simulation scene, calling InitGame event, calling BeginPlay event. Only the last two are shown in this figure. Object instances colored with red appear only on the main simulation instance. Arrows between objects illustrate function calls. This graph shows that the client simulation instances are missing almost all of CARLA server functionality.

is ideal for cheat-prevention in online multiplayer games, as it prevents malicious clients from affecting the game in arbitrary ways. In addition to cheat-prevention, there is no reason to waste bandwidth on synchronizing, or initializing, objects that the clients cannot affect in any way. [21] For example, the clients do not need to know the logic for determining when a game session ends, as only the main server has the authority to end it.

Figure 6.1 illustrates the objects, and their respective method calls, that are only initialized on the main server. It can be seen that key components of the CARLA server rely on such objects, which means that the rendering servers are not initializing the CARLA server plugin correctly. As a result, CARLA clients cannot communicate with the rendering servers.

From the Figure 6.1 it can be seen that most of the server initialization is done by the class *ATheNewCarlaGameModeBase*, which is a subclass of *AGameModeBase*. Without going too much in to details, this class is responsible for initializing other key components, such as the *UCarlaEpisode* and *ACarlaActorFactory*. It is also indirectly responsible for making the *ATheNewCarlaServer* listen for incoming RPCs on a specific TCP port, that is only known after initializing the CarlaSettings object.

As per Unreal Engine networking architecture, *AGameModeBase* is only initialized on the main server. However, a closely related class *AGameStateBase* is designed to transfer information about the current game state

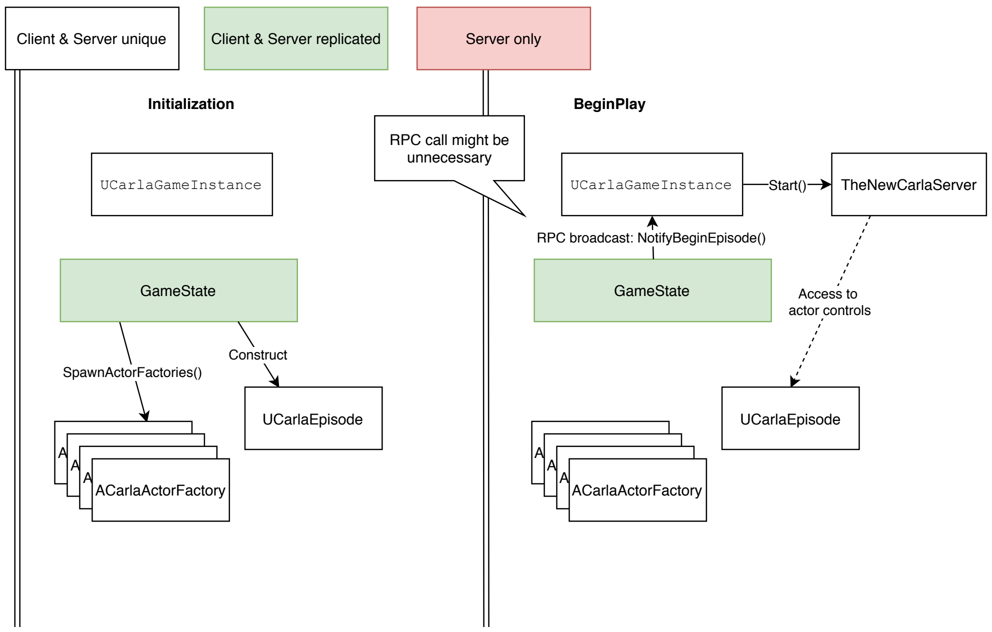

Figure 6.2. This graph shows the improved initialization sequence of the CARLA server. Key initialization procedures are moved from the server-only class (GameMode) to the shared class (GameState) (illustrated as green) that is first created on the main server, and then replicated on the client simulation instances.

to all game instances. This class is replicated, meaning that it will be instantiated on all servers. By migrating the CARLA server initialization procedures from the *AGameModeBase* to a subclass of *AGameStateBase*.

This keeps the support for changing simulation levels and restarting the current simulation level during runtime, as the GameState will always be destroyed and recreated at the beginning of a new level. Recreating is important to remove all references to any old actors of the previous level. However, it is unnecessary to recreate the connection between the Python clients and the CARLA server. To keep the CARLA server connections open between simulation episodes, the CARLA server should still be instantiated from GameInstance. The GameInstance is created only once, before even loading a simulation scene.

While the GameState is now replicated from the server to the clients, the constructor of the GameState is still executed locally on every simulation instance and the variables are, by default, not replicated. Events, such as *InitGame* and *BeginPlay* are called only on the server, but we can make the clients run the replicated counterpart *OnRep_ReplicatedHasBegunPlay*, which is called on the clients after the server has run *BeginPlay*.

With the above changes, all of the CARLA server components are now created on every simulation instance and the CARLA clients successfully can establish communication with any of the simulation instances.

7. Reimplementing main functionality

While the simulation instances are now synchronized and capable of receiving connections from the CARLA clients, they are still not fully functional. In this section, we focus on enabling the normal functionality of CARLA.

7.1 Function categories

The CARLA API commands can be divided in to three different categories: client-executable, main-server only and main-server only with a return value.

Client-executable commands should work as is. Main-server commands that do not return anything must be redirected from the rendering server to the main server. Commands that have a meaningful return value need additional steps for fetching the return value asynchronously.

The following list is incomplete, but it gives a good idea on what types of functions exist in each category.

1. **Executed at main server (with return value)**
    - Spawn actor
    - Get most recent vehicle control input
2. **Executed at main server (no return value)**
    - Destroy actor
    - Control vehicle
    - Control traffic light
    - Control pedestrian
3. **Executed at client server**
    - Get metadata (i.e. Map name, ping, blueprint library, ...)

- Get all actors
- Get actor state: location, speed, traffic light state, ...
- Sensor: is listening, stop (Sensors are client-only in our implementation)

4. **Executed at client and/or Main server (Depends on desired outcome)**

   - CollisionEvent
   - LaneInvasionEvent
   - Visual debugging tools (*e.g.* Draw a 3d-line inside the simulation environment)

The fourth category is not implemented during this thesis, as they are extra features that are not required to run the simulation. They can be executed both on the client and Main server, depending on what the user desires to achieve with them. For example, visual debugging tools are probably desired to be used on a specific server that is currently visible for the viewer, regardless of whether it is the client or the Main server. One idea for implementing these functions is to always execute them on all servers, as they are not expected to have any meaningful return value.

7.2 Client functions and sensors

Client-executable functions are the easiest to implement. Like the name implies, these functions can be fully executed at the client simulation instances. They do not need to send or receive any data from the main server. In practice, the CARLA client can already call client-executable functions normally, and the server can successfully execute them without any modifications.

Client-executable functions are also special in a sense that they cannot affect other simulation instances. This can be used as an advantage for anything that should only affect one simulation instance. In our case, we can simplify the distribution of rendering by creating all sensors through client functions. If the sensors exist only on one server, they cannot affect the computational load of other servers.

While isolating sensors to their own simulation instances is enough for this thesis, in some situations it might be beneficial to expose them to other servers as well. For example, a client controlling a vehicle might momentarily want to access the live feed provided by a camera on a smart intersection. The sensors could also be created on all servers, while making sure that they will perform the computationally expensive rendering only when requested by one or more CARLA client. Another viable option would be to just expose the location, orientation and parameters of the sensors to other servers.

Client-executable functions can also return any values synchronously, unlike the main-server functions as discussed below.

7.3 Main-server functions (without return value)

All functions that affect the simulation state must be executed at the main server. If they are executed on the client server, their changes cannot affect the other servers. Therefore, these functions have to be redirected to the main server from the client servers.

7.3.1 Redirecting to main server

Unreal Engine supports RPC between separate Unreal Engine instances depending on two conditions: the client simulation instances have to make RPCs from an actor instance that is both replicated to the main server and owned by the client. The actor must exist on the client server, because the client will initiate the RPC by calling the method from that actor instance. Likewise, the actor must exist on the main server, as the RPC will be executed with a reference to that particular actor instance. Both servers also have to agree that the client is the owner of the actor.

In practice, the simplest option is to make all RPCs through the client's own PlayerController instance. This is because every client instance will always have at least one PlayerController that is replicated between the simulation client and the simulation server, and the client has the ownership. The PlayerController is a native class of Unreal Engine that is used to control actors, and the clients must have ownership over its own controller to be able to control the player's actor.

For example, let's take a look on how the *DestroyActor* function can be implemented. We created a new *CarlaPlayerController* class for redirecting all commands to the main server. In the *CarlaPlayerController* we declare a new function with a specialized *UFUNCTION* syntax as follows.

```
UCLASS()
class CARLA_API ACarlaPlayerController : public APlayerController
{
  ... // Other declarations are not shown
  UFUNCTION(Server, Reliable, WithValidation)
  void DestroyActor(uint32 ActorId);
};
```

The *Server* flag in the *UFUNCTION* means that the method will be executed on the server. It can be called from the server, or the client that has ownership of the object instance, but it will always execute on the server. *Reliable* means that Unreal Engine should guarantee that the RPC is always executed. For example, if the network packet containing the call is dropped, Unreal Engine will resend the packet until it goes through. *WithValidation* specifies that there is an implementation of *DestroyActor_Validate* function, which is used to prevent cheating and otherwise invalid calls. While we do not need the validation flag/function, Unreal Engine will not compile without it. Therefore, in our use case we can define the function as always returning *true*. The implementation of the actual *DestroyActor* method also needs to be named with an additional "*_Implementation*" suffix.

The method in *CarlaPlayerController* can be implemented as follows:

```
void CarlaPlayerController::DestroyActor_Implementation(uint32 ActorId);
{
  Episode->DestroyActor(ActorId);
}
void CarlaPlayerController::DestroyActor_Validate(uint32 ActorId);
{
  return true;
}
```

And the corresponding method in CarlaEpisode can be simplified as follows:

```
void CarlaEpisode::DestroyActor(uint32 ActorId);
{
  if (GetWorld()->IsServer()) {
    auto ActorView = GetActorRegistry().Find(ActorId);
    DestroyActor(ActorView.GetActor());
  }
  else {
    // Redirect to main server
    PlayerController->DestroyActor(ActorId);
  }
}
```

Both the server and the clients have their independent instances of *CarlaEpisode* and replicated counterparts of a *CarlaPlayerController* instance. We utilize this fact to redirect the command from the client *CarlaEpisode* to the main server *CarlaEpisode*. If the *DestroyActor* is called on the main server simulation instance, it can be executed normally. Otherwise, it is redirected from the client to the main server through the *CarlaPlayerController* class.

7.3.2 Vehicle controls

A non-obvious limitation of Unreal Engine is that a vehicle actor cannot be controlled without an instance of a Controller. Directly adjusting the throttle value of a vehicle has no effect until it is assigned an controller.

There are two types of controllers: AIController and PlayerController. PlayerControllers act as an interface between the player and the actor, therefore the PlayerController exist directly on the client simulation/game instance. PlayerControllers have full replication support and handle the communication between client servers and the main server automatically. AIControllers are autonomous controllers that designed to run on the server only, as typically there would be no reason to expose AI logic on the clients in a multiplayer game.

We decided to implement the vehicle controls using AIControllers on the main server, because there were issues with assigning each client simulation instance with multiple properly replicated PlayerControllers. As the AIControllers only exist on the main server, we have to redirect all vehicle control commands from the rendering servers to the main simulation instance. This is implemented using RPCs in similar fashion as the DestroyActor command described above.

Even if we were able to assign multiple replicated PlayerControllers to each client simulation, this would limit the way the actors can be controlled. Each actor can only have one assigned controller at a time. If these controllers existed on the clients, these actors could only be controlled from one simulation instance at a time. While this might not feel like a meaningful limitation, there are application where it can be useful that a single vehicle can be controlled from multiple instances. For example, smart intersections could be implemented by having dedicated clients to momentarily control vehicles entering that particular intersection. It is also better to implement autopilot straight on the main server to reduce latency and to avoid switching between multiple controllers.

The drawback of this implementation is that we cannot utilize the optimization features of the replicated PlayerControllers. Unreal Engine has controller classes, such as the WheeledVehicleController, which have features created specifically for replicated vehicle movement. According to the Unreal Engine documentation, these controllers are capable of extrapolating physics on the client servers before receiving updated physics state from the main server. This extrapolation feature would help to reduce the latency as perceived by the vehicles on the client server. For example, the vehicle can appear to start turning immediately, instead of continuing forward until the next state update is received.

7.4 Main-server functions (with return value)

Main-server executable functions are slightly more complicated to implement, if they have to return a value to the client server. The implementation of RPC in Unreal Engine does not allow return values, as it is impossible to know when and if there will be a response. Even if a response was guaranteed within a few milliseconds (the round-trip-time between the two servers), it would be counter-productive to stall the game loop while waiting.

Figure 7.1 illustrates two possible solutions for retrieving return values, as well as the original non-distributed implementation. The commands can either be directly sent from the client to the main server or the return value can be fetched later as a separate command.

The first option appears to be easier to implement. However, a direct connection to the main server would require a second outgoing TCP connection from the CARLA client. Having to manage connections between multiple servers and the CARLA client goes against our design principles, as discussed in 5. In addition to this, we would still not know when the

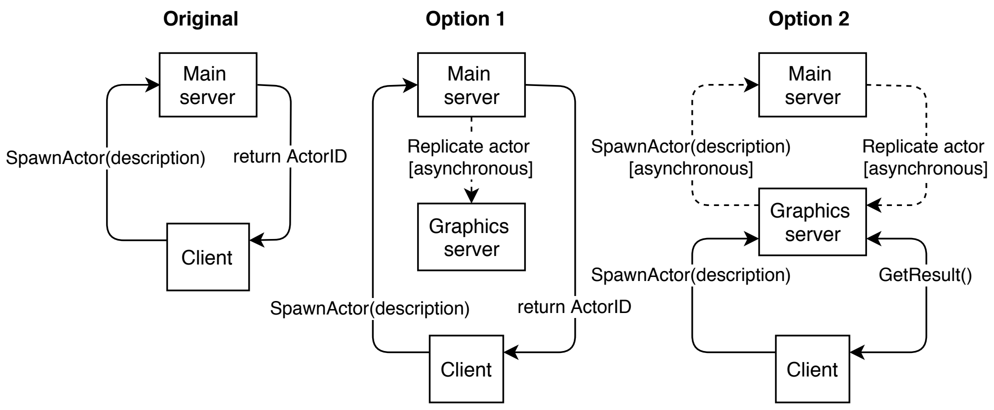

Figure 7.1. Different options for returning values of main-server executable functions. SpawnActor is used as an example of such function. Left option is the original implementation, with just one server. Option one uses a direct TCP connection from the CARLA client to the main server. The value is returned immediately, but the effect of the call is asynchronously transferred to the rendering server. Option two asynchronously redirects the call from the rendering server to the main server as an RPC. Due to the asynchronicity, the return value has to be later fetched with a separate function call.

new simulation state will be replicated to the rendering server. Even if we have a reference to the new actor as a result of the *SpawnActor*-command, we cannot know when the rendering server will be aware of this new actor. Therefore, any client-executable commands that require this new actor reference can fail.

While the second option is more suitable for our design principles, it cannot directly return a value to the CARLA client. While the rendering server redirects the command to the main server, it cannot know beforehand if this command will succeed and what it will return. Therefore, we need a system for fetching the result afterwards. This system is discussed in the next section.

7.5 Dealing with asynchronous calls

For dealing with the asynchronous nature of the main-server functions, we need a system for fetching the results afterwards.

We cannot know when the call reaches the main server. We need to be able to deal with dropped packages and failed calls. We also need to take in to account the fact that multiple CARLA clients can be simultaneously making calls through one or more rendering servers.

We can assign a unique ID to each main-server call that requires a return value.

Then we can define a *Status* enumeration and three new functions to the CARLA API:

- Enum Status : "Pending", "Failed", "Success"
- Status GetStatus(uint32 ID) : Returns the status of a method call
- String GetError(uint32 ID) : Returns the error string of a failed call as given by the main server
- Int GetResult(uint32 ID) : Returns the result of a successful call as given by the main server

Now we can store the results of all calls to the replicated CarlaPlayer-Controller class instance, that is also responsible for redirecting the calls. Initially, when the RPC is made from the client server, the status of the call is stored as "*Pending*" in a suitable data structure paired with the ID.

After the main server receives and executes the call, it can modify this data structure that exists in the replicated CarlaPlayerController. If the variable containing this data structure is declared as *Replicated* in the source code, its value will be replicated to the client simulation instance. This replication will only happen from the main server to the client server. Any modifications made by the client instance will be discarded, but this does not matter as at this point the status would no longer be "*Pending*", but "*Failed*" or "*Success*" instead.

Now the CARLA client can call main-server executable functions normally, but it has to periodically poll the rendering server until a result is received.

7.5.1 Additional modifications

Some methods need additional modifications in order to properly replicate their effects from the main server back to the rendering servers. SpawnActor is one of such commands.

As the name suggests, SpawnActor creates (spawns) a new actor in the simulation. SpawnActor takes an ActorDescription as its parameter. The ActorDescription is used to describe the desired properties of the actor. These properties include actor type, such as the model and color of a Vehicle, or a Sensor and its parameters. In addition to SpawnActor creating a new actor in the main server, it is also assigned a unique ID. This ID is stored in the queryable *ActorRegistry*, which the CARLA client uses to refer to these actors.

While the new actor is replicated to the client simulation instances, it will not have an ID registered in the client instance of the *ActorRegistry* by default. Therefore, we need to make some adjustments to the code. We can add a replicated ID property to each actor and add a short client-only code to the constructor, which registers this newly constructed actor to the registry. The code can be made client-only with a simple if-statement, that checks if the code is being executed on the main server.

8. Measurements

In this chapter we go over measurements designed to prove that the computational load is distributed to multiple nodes and that the performance is increased.

Briefly, we ran an experiment where we periodically increase the simulation load by increasing the amount of AD vehicles in the simulation. We repeated this experiment with varying amount of rendering computers.

8.1 Experiment setup

The experiment was made by running sets of measurements with one to four CARLA server instances running on separate computers, connected to each other through LAN. Each computer is running the exact same build of our modified version of CARLA, on top of Ubuntu 16.04 and the latest drivers. Details of the machines are described in the table 8.1.

A custom Python client was written for performing the experiment. The client is designed to simultaneously connect to multiple CARLA servers to avoid issues with synchronizing measurements between multiple client processes. The client is capable of collecting performance statistics from

	Main server	Rendering server 2	Rendering server 3	Rendering server 4
CPU	Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz	Intel(R) Xeon(R) CPU E31230 @ 3.20GHz	Intel(R) Core(TM) i7 CPU 970 @ 3.20GHz	Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz
GPU	GeForce GTX 1080 TI 12 GB	GeForce GTX 1050 TI 4 GB	GeForce GTX 1050 TI 4 GB	GeForce GTX 680 2 GB
Memory	32 GB	16 GB	12 GB	18 GB

Figure 8.1. This table shows the hardware specifications for each server used in the experiments.

all servers individually. These statistics include total number of vehicles in the simulation, time spent calculating each simulation frame and total images received from the simulation server.

The experiments are designed to be run in a set of episodes, where each episode consists of a specific amount of frames. For each episode, the computational load is incrementally increased. For example, if we ran an experiment with 10 episodes, where each episode is 2000 frames long. For each episode, we increase the computational load by creating one additional vehicle and camera for each server. This would result in a performance measurement with the amount of vehicles ranging from 4 to 40 in discrete steps of four vehicles.

To imitate a realistic simulation scenario, each vehicle is equipped with one camera and is moving around the simulation scene with the default autopilot enabled. We run most of the experiments with a relatively multiple resolutions, starting from 180x120, which should already be enough for some image classification CNN models. The default autopilot is enabled, in order to create a more realistic simulation scenario with some load on the physics engine, the vehicle position replication system and to avoid any potential optimization that Unreal Engine might use for a stationary camera.

At the start of a new episode, we add new vehicles to the simulation. Before we start logging data, we allow the system to settle for several seconds in order to avoid any initial variance interfering with the measurements. During the episode, each CARLA server sends an RPC to the client after each simulation step. Each RPC is logged, which means that we can keep count of the total amount of frames received from each server. The episode will run until we have received a predetermined amount of frames from the main simulation server. From the duration of the episode and the amount of frames received from each server, we can calculate statistics such as average received frames per second. Additionally, some of the statistics, such as the time spent calculating a single simulation frame, are calculated on the CARLA server, as provided by the CARLA client API. This is useful, as it allows us to ignore any additional delay added from transferring the RPC over from the CARLA server to our client and any additional overhead caused by the client. Our client should not affect the results in any way.

In order to speed up the experiments, the simulation will not be reset between episodes. This can have some effect on the results, as some loca-

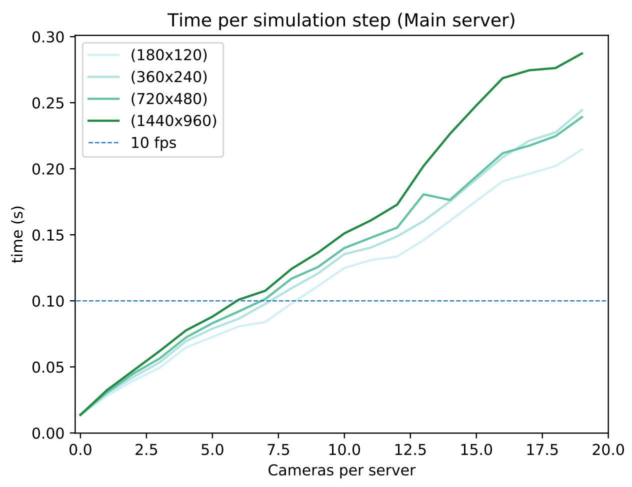

Figure 8.2. This graph shows the average computation time of a single simulation step, when the experiment is run on the main server without sharing the simulation state to other servers. The measurements were done separately for four different resolutions. One simulation step consists of rendering one image from all vehicular cameras.

tions in the simulation scene can be more graphically intensive. However, the simulation is restarted when the number of rendering computers is changed.

All of the experiments are run using the default weather and quality settings of CARLA 0.9.1 in the latest version of the urban simulation scene named "Town01", which was the default scene of CARLA during the start of this thesis. Each simulation instance also has a graphical window with the size 360x240, which causes a small additional load on the GPUs. As the size of the window is relatively small and it stays constant, it should not significantly affect our measurements. The window size however could be reduced even further or potentially even disabled altogether.

8.2 Results

We start off with the baseline results achieved by running the experiment on the main server in isolation. The simulation state is not shared to the other servers. In Figure 8.2, it can be seen how the resolution and the number of cameras (one camera per vehicle) affects the update frequency of the simulation. Interestingly, the number of cameras have a far more significant effect on the performance than the resolution. Even as resolution 1440x960 has ~63 times more pixels than 180x120, it only slows down the simulation step by only roughly 40 %. However, the slowdown is almost linear to the number of cameras. Each added camera adds an average of 11 ms to 14 ms to the computation time, depending on the

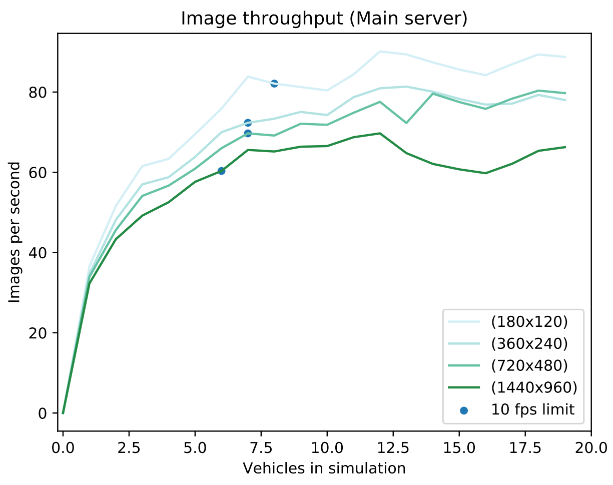

Figure 8.3. This graph shows the image throughput achieved by running the experiment on the main server, without sharing the simulation state to other servers. Image throughput is the sum of images received from all cameras. Dots on the lines represent the last experiment that achieved more than 10 fps on average.

resolution. It is likely that resolution has a low impact on the performance because these experiments were run on a relatively powerful GPU (GTX 1080 TI). A lower-end GPU might show larger difference between different resolutions, but this was not explicitly measured during this thesis.

Figure 8.3 shows the total image throughput achieved by running the simulation on the main server. Total image throughput is the sum of images produced per second, or the simulation update frequency multiplied by the number of cameras. Each camera produces one image on each simulation update. While the graph shows some odd behavior towards higher numbers of vehicles, it can be seen that the maximum throughput is slightly over 80 images per second with the lowest resolution, and slightly below 70 for the highest resolution. The plots would likely behave more neatly if this experiment was repeated multiple times. With low amount of vehicles the throughput is low because there is not enough rendering load. This is because each camera is rendered only once per simulation step. Each simulation step also includes additional computations, including the physics step. With only one vehicle in the scene, these additional computation steps are acting as a bottleneck as the GPU spends most of the time waiting for new rendering commands.

Both figures (8.2 and 8.3) highlight the limit of 10 simulation steps per second. This is the minimum frame rate that the CARLA documentation suggests [9] to be used in CARLA version 0.9. Lower frame rates might result CARLA in no longer running in real-time, as the maximum physics update delta time is limited to keep the physics stable. As a coincidence,

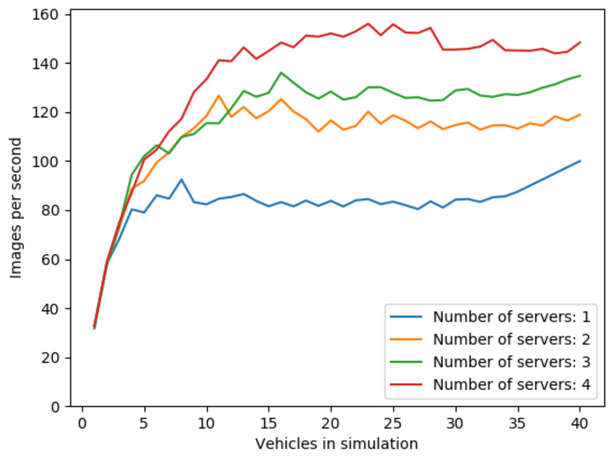

Figure 8.4. This graph shows how the total image throughput scales when new servers are added to the setup.

this minimum is quite close to the camera update frequency seen in other literature [39, 12, 11, 8]. In Figure 8.2, this limit is shown as a dotted line. In Figure 8.3, the dots on the lines represent the last experiment that achieved more than 10 fps on average. With this in mind, the main server is capable of simultaneously simulating 6 to 8 vehicles with a throughput of 60 to 80 images per second depending on the resolution.

Figure 8.4 shows the total image throughput achieved when the computational load is distributed to additional servers. This experiment was run four times with increasing number of servers: The first line represents the performance on the main server, while the fourth line utilizes all four servers. The image throughput shown on y-axis is the sum of images received from all servers per second. If server A was to produce 10 images per second and server B 40 images per second, the throughput shown on this graph would be 50 images per second.

X-axis represents the number of vehicles, and thus the number of cameras, that are simultaneously present in the simulation. Each camera produces one image on each simulation step. As no load balancing is done, the vehicles are evenly distributed across the different servers. The computationally most powerful server has the same workload as the weakest server.

By looking at this graph, we can see that the throughput clearly increases as the number of servers is increased. The lowest throughput of 80 fps is achieved by running the simulation only the main server and the highest throughput of 150 fps is achieved by running the simulation on all servers. This means that the implementation of distributed rendering is working.

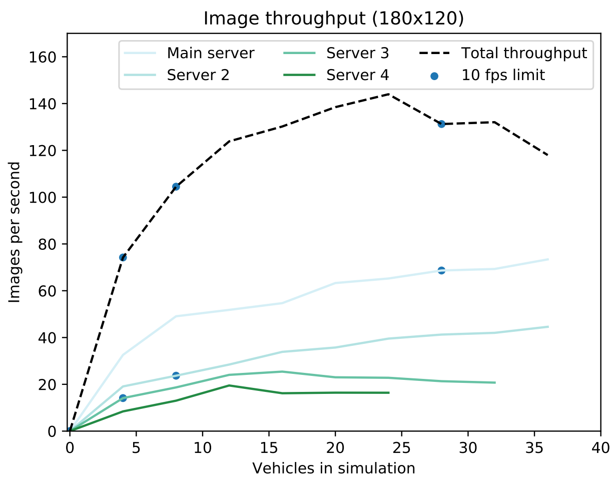

Figure 8.5. This graph shows the total image throughput with all four servers and 180x120 resolution.

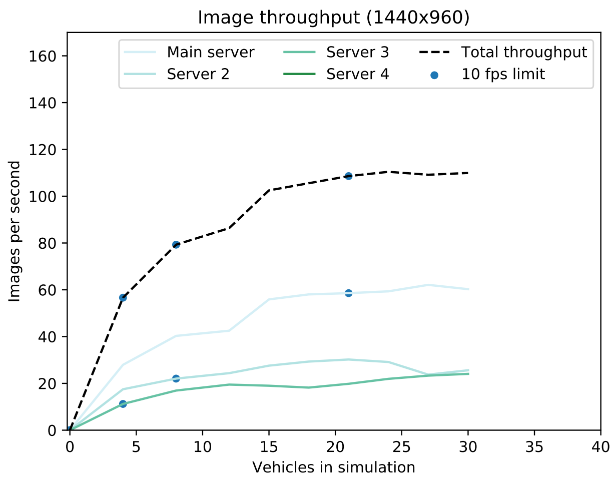

Figure 8.6. This graph shows the total image throughput for all four servers with 1440x960 resolution.

However, we can also notice that the throughput is not multiplied by four, even if the number of servers is increased from one to four. This is most likely due to significantly weaker hardware on the other servers, but there might also be some additional overhead caused by the Unreal Engine's replication system. It could be interesting to measure this on identical hardwares.

We can also see that each setup becomes saturated at different number of vehicles. With one server, the maximum throughput is achieved around 6-7 vehicles. Similarly, two, three and four server setups achieve their maximums around 10, 15 and 20 vehicles respectively.

Figure 8.5 gives us a better insight on the relative performance differences for each server. Each line represents the image throughput attained from that particular server. Similarly to the main server figures discussed

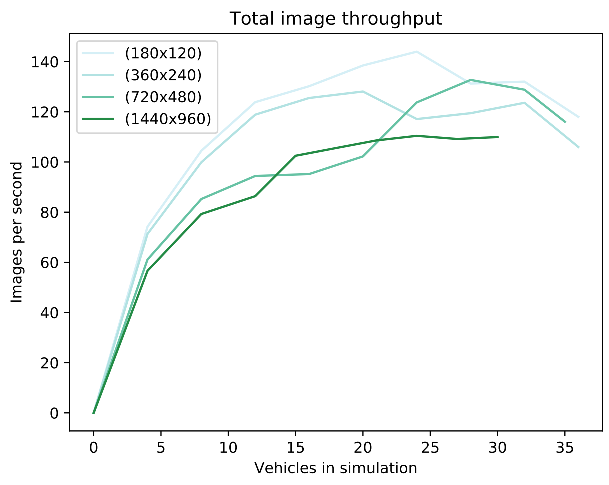

Figure 8.7. This graph shows the total image throughput for all four servers with measured resolutions.

above, the last experiments with over 10 fps are highlighted on the lines. While hard to interpret from this graph, the servers 1 (Main Server), 2 and 3 reach this limit at 7, 2 and 1 cameras respectively. Server 4 is barely under 10 fps even with only one camera. The reason for plots of servers 3 and 4 being cut short is that they become practically non-responsive with higher number of vehicles. While they are still producing some images, their average computation time for a single simulation step increases to the range of 1-2 seconds. Filtering out these non-responsive servers also affects the total throughput shown on the graph.

Figure 8.6 shows the same experiment with the highest resolution (1440x960). It shows fairly similar results to the lowest resolution, with the exception of Server 4 not being able to produce any images. This can be due to both, the significantly weaker GPU and CPU in comparison to other servers.

Figure 8.7 shows the total throughputs from all four resolutions, when using all four servers. This graph reveals an issue with the experiments. For example, the resolution 720x480 experiences a slight drop in performance around 12-22 vehicles, and rises above lower resolutions between 25-34 vehicles. It is highly likely that this is simply random noise that could be reduced by running more repetitions of the experiments.

Figures 8.10 and 8.11 demonstrate the latency between the measuring client and the CARLA simulation instances. The latency is measured by checking how long it takes to receive a response to an RPC ping from the CARLA server. The highest latency for the lowest resolution is slightly below 1 ms, and for the highest resolution slightly over 2,5 ms. With the

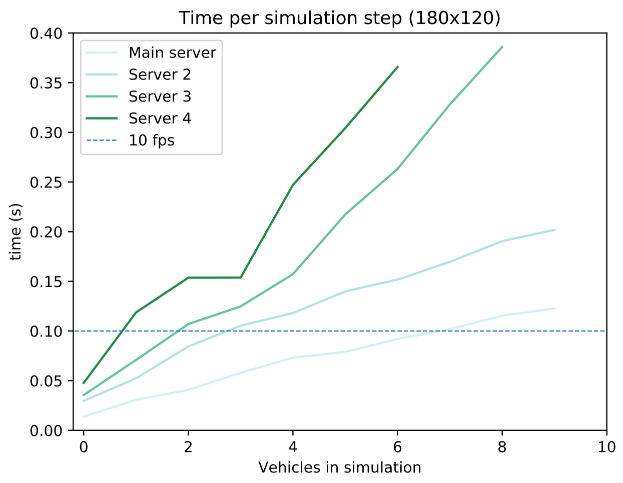

Figure 8.8. This graph shows the average computation time of a single simulation step for all four servers with 180x120 resolution.

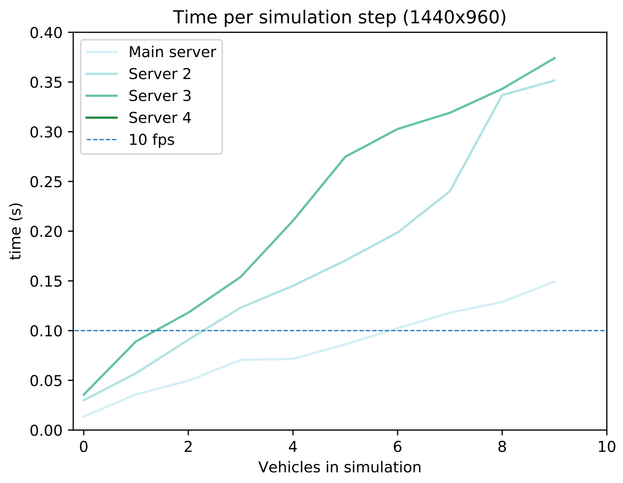

Figure 8.9. This graph shows the average computation time of a single simulation step for all four servers with 1440x960 resolution.

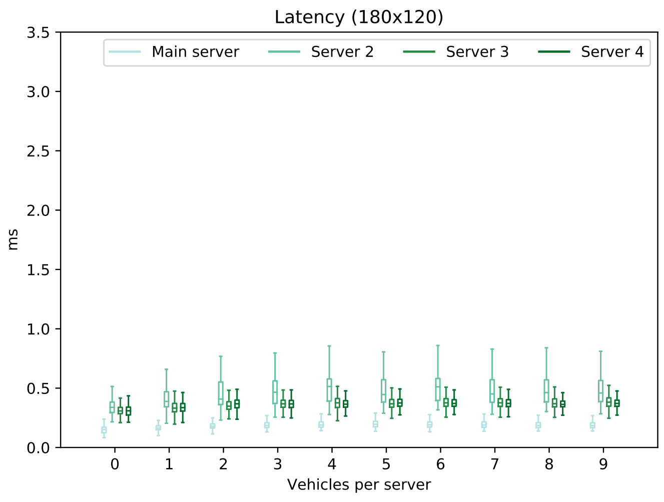

Figure 8.10. This graph shows the latency between the CARLA Python client and the CARLA simulation instances. The latency is measured on the Python client by measuring how long it takes for a CARLA server to respond.

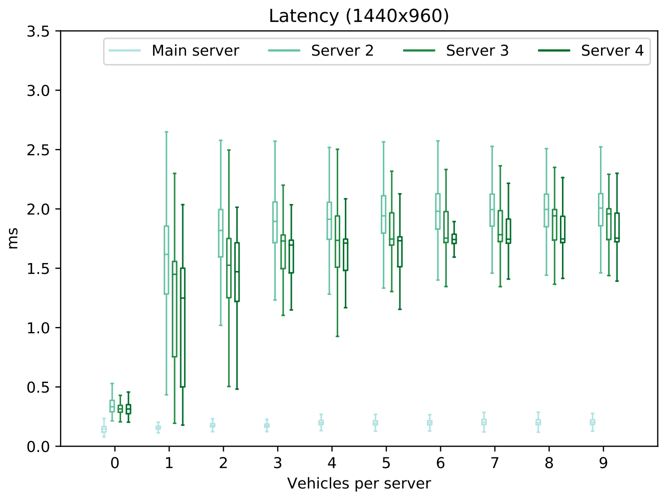

Figure 8.11. This graph shows the latency between the CARLA Python client and the CARLA simulation instances. The latency is measured on the Python client by measuring how long it takes for a CARLA server to respond.

highest resolution the mean variance increases slightly when the number of camera sensors is increased. This same effect is not visible with the lowest resolution. However, in both cases the main server has a very low latency of around 0,2 ms regardless of the amount of camera sensors. This is most likely due to the combination of the main server having a very strong GPU and running on the same computer as the measuring client. There is no additional latency from communicating over LAN for the main server. It can also be seen that the strongest of the rendering servers, Server 2, has slightly more latency on average. The reason for this is not known, but it might be that a faster server uses more time rendering the images and therefore has less time to respond to the RPCs. There is also significantly more variance with all rendering servers when larger resolution is used. The variance might result from the fact that the ping RPC can arrive to the server at different times of the simulation step. In any case, a latency of below 3 ms is very reasonable in comparison to the total time the servers spend in the simulation step.

9. Discussion

From the results in the previous chapter can be seen that our implementation can perform significantly better than a single computer setup. Our setup of four heterogeneous servers offers almost twice the rendering output of a single server.

As previously discussed, we can assume that a reasonable minimum update frequency for the sensors is 10 images per second. We can also assume that autonomous vehicles can have up to eight cameras, which we know is the case with Tesla's current vehicles. The results show that our reasonably powerful single-server setup can support up to eight vehicles with one camera, or just one eight-camera vehicle. While using all of our four servers, we can get a total throughput of almost 160 images per second. This would be enough for either 16 single-camera or two eight-camera vehicles. However, our rendering servers were barely able to produce 10 frames per second with a single camera. While one could think that the rendering performance is mostly dependent on the GPU, it is clearly not the only limiting factor. Servers 2 and 3 have significant differences in performance, even as they have the same GPU (GTX 1050 TI). It might be that the performance of the GPU is more relevant when using even higher resolutions than 1440x960. High resolution cameras are important when detecting obstacles over longer distances [61], and therefore when driving at higher speeds.

While in our case the performance does not scale linearly to the number of servers, we have also shown that this is probably due to the drastic hardware differences between our testing servers. It is reasonable to assume that with four servers, that are as powerful as our main server, we could support almost four times more vehicles than a single server setup. It is however unknown how far the system can be scaled. It is not hard to imagine a use case for simulating very large cities with hundreds of cars communicating through V2X, but at some point the Main Server or the Unreal Engine will likely become the bottleneck. The maximum scalability is at least dependent on the network bandwidth and CPU demand of the Unreal Engine networking module and the CPU demand of the physics step. While it is possible to always add more rendering servers, the Main Server still must be capable of synchronizing the simulation state to all these server in addition to running the physics simulation. While games like Fortnite have shown that Unreal Engine can support at least 100 simultaneous players, they rarely have that many vehicle-like physically complex actors. It is reasonable to expect that physically complex actors require more resources to accurately synchronize across servers, as they consist of multiple individual physical parts that are connected together through various joints (*e.g.* vehicle wheel suspension system). In contrast, three 3D-vectors can be enough to fully describe the position, velocity and orientation of a simple actor, such as a player character in Fortnite. On the other hand, Fortnite must synchronize hundreds of player-usable resources (*e.g.* guns, ammunition and destructible buildings), gunshots and buildings created by the players.

It would also be possible to reduce the CPU load of the rendering servers by disabling all client-side physics simulation. While the physics are disabled, the vehicles would still continue to receive state updates from the main server, meaning that they would not completely become out of sync. However, as the main server has the sole authority on the simulation state, it must be capable of calculating physics for all vehicles in real-time even after these optimizations. Likewise, the update frequency of the Main Server could be optimized by removing all unnecessary operations from the Main Server. For example, the Main Server could be implemented as a dedicated server without any graphical window or rendering load.

The use of heterogeneous servers and different client demands opens up a need for load balancing. It would be helpful to automatically monitor the available resources on each server and use that information to assign new clients to the least strained server. Likewise, the computationally most demanding clients could be directed to the most powerful servers. This could be implemented using a container management platform, such as Kubernetes [35], in addition to creating a ready-to-use container containing the CARLA server.

As discussed in previous chapters, we believe that the simulation should always run in real-time. This is because controlling an AV and V2X communication are both very latency sensitive tasks [1, 2]. As a reminder, in our vision only the vehicular sensor data is simulated. We could also extend the simulation to account for, *e.g.*, latencies, bandwidth and availability limitations caused by the position and number of 5G and WiFi access points in relation to the locations of the vehicles and the number of concurrent users. However, the important point is that we can use real vehicular hardware and real fog computing services with the simulator. This allows us to verify that real hardware is able to cope with the latency- and computational speed requirements of V2X applications. This also means, that in order to properly test the V2X applications, the latencies caused by the simulation should be as close to a non-simulated environment as possible. The latency-aspect of vehicular fog computing is discussed in more detail in [68], along with the possibility of using the vehicles themselves as part of the fog computing network.

Figures 8.10 and 8.11 show that the latency between the CARLA client and the rendering nodes is low (< 3 ms). While this is promising, the RTT between the client and the servers is only a small portion of the total latency. For example, the distributed nature of the simulation means that there is additional latency introduced when synchronizing the simulation state between the Main Server and the rendering nodes.

In the simulation framework, there are three connection links that can introduce latency: simulation instance synchronization, sensor output and control input. In a single-server setup, the sensor output and control input latencies are dependent on the RTT (Round-Trip Time) between the vehicle AI and the CARLA server. If the client and the server exist on the same computer or the same local network, this latency can be assumed to be insignificant as shown in Figures 8.10 and 8.11. The update frequency of the simulation adds to the latency, as the physics are only updated once per game loop iteration. For example, with a relatively high update frequency of 60 fps, it would take at maximum 16.7 ms and on average 8.3 ms to start processing a control command. With 60 fps, the latencies are relatively low. However, with 10 fps, the maximum latency rises to 100 ms. Distributing the simulation over multiple computers further increases the latency. Each control command must now first travel from the vehicle AI to the client server, and then to the Main Server. If both the Main server and the rendering server are running at 10 fps, it could take as long as 200 ms for the control command to reach the Main Server. We have also discussed the latencies between the simulation and the vehicle control client in [28] and [15].

In addition to these latencies, the effect of the Unreal Engine's networking module is unclear as it was not measured in this thesis. The state synchronization can be done in many ways and the Unreal Engine offers possibilities for optimizations and customizations. As discussed in previous chapters, the networking module uses UDP for synchronization and dynamically determines when to update the state of each simulation actor. In other words, some packets might be dropped and some actors might not always be synchronized. It is possible that the simulation states have slight differences between separate rendering nodes. Desynchronized simulation states could result in falsely detecting a flaw in the vehicle AI, even if the AI only did a wrong action because of the latency. In the worst case scenario, AV_a might momentarily perceive AV_b vehicle running red lights, when in reality AV_b has already come to a full stop. In this case the rendering node might also report a collision between two vehicles, when there was none. There is more discussion about state synchronization in online game engines in, for example, [48] and [3]. There is also a more extensive survey of latency reduction techniques from a wider perspective in [6], which also touches the topic of online multiplayer games.

Measuring the amount of desynchronization between two servers can prove to be quite difficult. One way could be to log the simulation state and the timestamp on every frame on all servers. The simulation states can then be compared for any differences at every point in time. However, this might require accurately synchronizing the clocks between the two servers. Another approach could be to display the sensor output from two server side-by-side monitors. These monitors could then be recorded with a high-speed camera and then visually inspected for any differences. Similarly, timestamps could be used to measure the average time it takes to synchronize the simulation state between the Main Server and a rendering node. A unique identifier could also be added to control commands in order to track a particular command and to measure how long it takes for it to reach the Main Server.

It is also important to remember that simulated sensors do not completely match their real-world counterparts. For example, vibrations, movement and different lighting conditions might be hard to model. This is also the case with different weather conditions. A machine learning model trained on simulated rain might perform completely different with real rain. However, [51] shows that at least semantic segmentation of real images can be performed well with a machine learning model that is trained from simulated datasets. The same paper shows that a combination of virtual images and real images can provide greater accuracy than real images alone. Another example of transferring knowledge from simulated to real environment is show in [10].

10. Conclusion

Developing software for autonomous vehicles and connected driving is difficult, because testing the software with real vehicles can be hazardous and expensive. Machine learning models used in AVs [61] also have to be trained on huge datasets, that have to be collected and often manually annotated. The use of virtual environments makes it possible to safely test the behavior over thousands of driving hours in simulated traffic. Virtual environments make it possible to hand-craft difficult corner cases, such as traffic accidents, that rarely happen in real traffic, and to even create automatically annotated training data from these situations [51]. Virtual environments enable the research and testing of V2X communication applications, *e.g.* Vulnerable Road-user Discovery [1], before physically integrating the required communication links [2] and edge computing services to the road infrastructure.

The research of AVs and connected driving in simulated environments is limited by the performance of such simulators, as these simulators can typically only utilize the resources of a single computer at a time. The real-time simulation of even one or two AVs can be computationally too expensive, as modern AVs might contain as many as eight on-board cameras. While the number of simulated vehicles could be increased by dropping the real-time requirement, we believe that running the simulation in real-time is essential. As driving a vehicle is a very latency-sensitive operation, the latencies in a simulated environment should be as close to reality as possible. If the rest of the hardware is operating at an unrealistically high speed in comparison to the simulation, it might hide issues caused by having too high latencies. It is more practical to run the simulation in real-time, than to match the slower simulation by artificially slowing down the computational speed of the AV hardware, the edge and cloud servers and the communication links between all of these.

In this thesis we investigated the possibility to increase the simulation performance by distributing the computational load over multiple computers. We selected CARLA as the most suitable simulator for researching AVs and V2X out-of-the-box. We discussed the architecture of CARLA and different solutions for distributing the computational load. We implemented and measured one of these solutions.

While the results are promising, further work is required to test how far the system can be scaled. As our aim was to allow the simultaneous simulation of multiple AVs in real-time, it is crucial to verify that the distributed nature of our solution does not introduce too much additional latency. It would also be beneficial to automate the deployment of the simulation instances in a fog or cloud computing platform and to automatically balance the computational load between the servers.

It is also important to remember that simulated sensors do not completely match their real-world counterparts. For example, vibrations, movement and different lighting conditions might be hard to model. This is also the case with different weather conditions. A machine learning model trained on simulated rain might perform completely different with real rain. Modeling loss of traction on icy and wet surfaces is yet another problem, along with simulating the visual look of snow and ice. However, even if the vehicles and their sensors behave slightly different in the simulation than in the real world, the connected driving use cases are mostly unaffected. We believe that simulations are useful for testing applications of V2X communication.

Bibliography

[1] 5GAA. *5G Automotive Association: Toward fully connected vehicles: Edge computing for advanced automotive communications*. 2017. URL: http://5gaa.org/wp-content/uploads/2017/12/5GAA_T-170219-whitepaper-EdgeComputing_5GAA.pdf.

[2] 5GIA. *5G Infrastructure Association: Vision White Paper, February 2015*. 2015. URL: http://5g-ppp.eu/wpcontent/uploads/2015/02/5G-Vision-Brochure-v1.pdf.

[3] Sudhir Aggarwal, Hemant Banavar, Amit Khandelwal, Sarit Mukherjee, and Sampath Rangarajan. "Accuracy in dead-reckoning based distributed multi-player games". In: *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. ACM. 2004, pp. 161–165.

[4] Mohamed Aly. "Real time detection of lane markers in urban streets". In: *2008 IEEE Intelligent Vehicles Symposium*. IEEE. 2008, pp. 7–12.

[5] BeamNG GmbH. *BeamNG.research*. Version 1.3.0.0. URL: https://www.beamng.gmbh/research (visited on 2019-02-22).

[6] Bob Briscoe et al. "Reducing internet latency: A survey of techniques and their merits". In: *IEEE Communications Surveys & Tutorials* 18.3 (2014), pp. 2149–2196.

[7] Gabriel J. Brostow, Jamie Shotton, Julien Fauqueur, and Roberto Cipolla. "Segmentation and Recognition Using Structure from Motion Point Clouds". In: *ECCV (1)*. 2008, pp. 44–57.

[8] Claudio Caraffi, Tomáš Vojíř, Jiří Trefny, Jan Šochman, and Jiří Matas. "A system for real-time detection and tracking of vehicles from a single car-mounted camera". In: *2012 15th international IEEE conference on intelligent transportation systems*. IEEE. 2012, pp. 975–982.

[9] *CARLA documentation*. URL: `https://carla.readthedocs.io/en/latest/` (visited on 2019-03-02).

[10] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. "Deepdriving: Learning affordance for direct perception in autonomous driving". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 2722–2730.

[11] Hyunggi Cho, Paul E Rybski, Aharon Bar-Hillel, and Wende Zhang. "Real-time pedestrian detection with deformable part models". In: *2012 IEEE Intelligent Vehicles Symposium*. IEEE. 2012, pp. 1035–1042.

[12] Hyunggi Cho, Young-Woo Seo, BVK Vijaya Kumar, and Ragunathan Raj Rajkumar. "A multi-sensor fusion system for moving object detection and tracking in urban driving environments". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2014, pp. 1836–1843.

[13] Marius Cordts et al. "The cityscapes dataset for semantic urban scene understanding". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 3213–3223.

[14] Dana Cowley. *Porsche, NVIDIA, and Epic Games reveal 'The Speed of Light' for Porsche 911 Speedster concept*. URL: `https://www.unrealengine.com/en-US/blog/porsche-nvidia-and-epic-games-reveal-the-speed-of-light-for-porsche-911-speedster-concept` (visited on 2019-02-22).

[15] Anton Debner, Matias Hyyppä, Jussi Hanhirova, and Vesa Hirvisalo. "Scalability of a Machine Learning Environment for Autonomous Driving Research". In: *The Proceedings of IEEE International Conference on Industrial Informatics (INDIN)*. To appear 2019.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[17] Ernst D Dickmanns. "The development of machine vision for road vehicles in the last decade". In: *Intelligent Vehicle Symposium, 2002. IEEE*. Vol. 1. IEEE. 2002, pp. 268–281.

[18] 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*. 2017.

[19] Epic Games. *Fortnite*. URL: `https://www.epicgames.com/fortnite/en-US/home` (visited on 2019-02-22).

[20] Epic Games. *Unreal Engine documentation*. URL: `https://www.unrealengine.com` (visited on 2019-02-22).

[21] Epic Games. *Unreal Engine Networking documentation*. URL: `https://docs.unrealengine.com/en-US/Engine/Networking` (visited on 2019-02-22).

[22] Epic Games. *Unreal Engine official website*. URL: `https://docs.unrealengine.com` (visited on 2019-02-22).

[23] Epic Games. *Unreal Engine Physics documentation*. URL: `https://docs.unrealengine.com/en-us/Engine/Physics` (visited on 2019-02-22).

[24] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. “The pascal visual object classes (voc) challenge”. In: *International journal of computer vision* 88.2 (2010), pp. 303–338.

[25] Adrien Gaidon, Qiao Wang, Yohann Cabon, and Eleonora Vig. “Virtual worlds as proxy for multi-object tracking analysis”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 4340–4349.

[26] Jason Gregory. *Game Engine Architecture, Second Edition*. 2nd. Natick, MA, USA: A. K. Peters, Ltd., 2014. ISBN: 1466560010, 9781466560017.

[27] Jussi Hanhirova, Anton Debner, Matias Hyyppä, and Vesa Hirvisalo. “A machine learning environment for evaluating autonomous driving software”. In: *The Proceedings of the Embedded World Conference (EWC)*. 2019.

[28] Jussi Hanhirova, Anton Debner, Matias Hyyppä, and Vesa Hirvisalo. “AI Accelerator Latencies in Hybrid Vehicular Simulation”. In: *The workshop on Emerging Deep Learning Accelerators (EDLA, in conjunction with HiPEAC)*. 2019.

[29] *Intel (2017): "Data is the new oil in the future of automated driving"*. 2017. URL: `https://newsroom.intel.com/editorials/krzanich-the-future-of-automated-driving/`.

[30] Rong Kang, Jieqi Shi, Xueming Li, Yang Liu, and Xiao Liu. "DF-SLAM: A Deep-Learning Enhanced Visual SLAM System based on Deep Local Features". In: *arXiv preprint arXiv:1901.07223* (2019).

[31] Roozbeh Kianfar et al. "Design and experimental validation of a cooperative driving system in the grand cooperative driving challenge". In: *IEEE transactions on intelligent transportation systems* 13.3 (2012), pp. 994–1007.

[32] Jihun Kim and Minho Lee. "Robust lane detection based on convolutional neural network and random sample consensus". In: *International Conference on Neural Information Processing*. Springer. 2014, pp. 454–461.

[33] Zu Kim. "Realtime lane tracking of curved local road". In: *2006 IEEE Intelligent Transportation Systems Conference*. IEEE. 2006, pp. 1149–1155.

[34] Nathan Koenig and Andrew Howard. "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sendai, Japan, 2004-09, pp. 2149–2154.

[35] Kubernetes. URL: `https://www.kubernetes.io` (visited on 2019-02-22).

[36] Fred Lambert. *Tesla's Autopilot 2.0 is now using 2 out of 8 cameras with the new update*. URL: `https://electrek.co/2017/03/30/tesla-autopilot-2-0-camera-8-1-update/` (visited on 2019-02-22).

[37] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer. 2014, pp. 740–755.

[38] Directorate-General for Mobility and Transport (European Commission). *Preparatory work for an EU road safety strategy 2020-2030*. 2018. DOI: `10.2832/15318`. URL: `https://doi.org/10.5281/zenodo.1248998`.

[39] Michael Montemerlo, Sebastian Thrun, Hendrik Dahlkamp, David Stavens, and Sven Strohband. "Winning the DARPA Grand Challenge with an AI robot". In: *AAAI*. 2006, pp. 982–987.

[40] Tesla Motors. *Tesla autopilot*. URL: `https://www.tesla.com/presskit/autopilot` (visited on 2019-02-22).

[41] Gerhard Neuhold, Tobias Ollmann, Samuel Rota Bulo, and Peter Kontschieder. “The mapillary vistas dataset for semantic understanding of street scenes”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 4990–4999.

[42] Fred E Nicodemus. “Directional reflectance and emissivity of an opaque surface”. In: *Applied optics* 4.7 (1965), pp. 767–775.

[43] NVIDIA. *DRIVE Constellation*. URL: `https://www.nvidia.com/en-us/self-driving-cars/drive-constellation/` (visited on 2019-02-22).

[44] NVIDIA. *GameWorks and UE4*. URL: `https://developer.nvidia.com/nvidia-gameworks-and-ue4` (visited on 2019-02-22).

[45] NVIDIA. *PhysX SDK*. URL: `https://developer.nvidia.com/physx-sdk` (visited on 2019-02-22).

[46] Sean O’Kane. *How Tesla and Waymo are tackling a major problem for self-driving cars: data*. URL: `https://www.theverge.com/transportation/2018/4/19/17204044/tesla-waymo-self-driving-car-data-simulation` (visited on 2019-02-23).

[47] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli. “A survey of motion planning and control techniques for self-driving urban vehicles”. In: *IEEE Transactions on intelligent vehicles* 1.1 (2016), pp. 33–55.

[48] Wladimir Palant, Carsten Griwodz, and Pål Halvorsen. “Evaluating dead reckoning variations with a multi-player game simulator”. In: *Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video*. ACM. 2006, p. 4.

[49] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[50] Craig Quiter and Maik Ernst. *deepdrive/deepdrive: 2.0*. 2018-03. DOI: `10.5281/zenodo.1248998`. URL: `https://doi.org/10.5281/zenodo.1248998`.

[51] German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio M Lopez. “The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 3234–3243.

[52] SAE. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. DOI: 10.4271/j3016_201806. URL: https://doi.org/10.4271/j3016_201806.

[53] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural networks* 61 (2015), pp. 85–117.

[54] SeekingAlpha. *Tesla Q3 2018 Conference Call Transcript*. URL: https://seekingalpha.com/article/4214138-tesla-inc-tsla-ceo-elon-musk-q3-2018-results-earnings-call-transcript?part=single (visited on 2019-02-22).

[55] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. “AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles”. In: *Field and Service Robotics*. 2017. eprint: arXiv:1705.05065. URL: https://arxiv.org/abs/1705.05065.

[56] Mennatullah Siam, Sara Elkerdawy, Martin Jagersand, and Senthil Yogamani. “Deep semantic segmentation for automated driving: Taxonomy, roadmap and challenges”. In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2017, pp. 1–8.

[57] Santokh Singh. *Critical reasons for crashes investigated in the national motor vehicle crash causation survey*. Tech. rep. 2015.

[58] Takafumi Taketomi, Hideaki Uchiyama, and Sei Ikeda. “Visual SLAM algorithms: A survey from 2010 to 2016”. In: *IPSJ Transactions on Computer Vision and Applications* 9.1 (2017), p. 16.

[59] *Torque3D*. URL: http://torque3d.org/ (visited on 2019-02-22).

[60] *Unity3D*. URL: https://unity3d.com/ (visited on 2019-02-22).

[61] Jessica Van Brummelen, Marie O’Brien, Dominique Gruyer, and Homayoun Najjaran. “Autonomous vehicle perception: The technology of today and tomorrow”. In: *Transportation research part C: emerging technologies* 89 (2018), pp. 384–406.

[62] Shenlong Wang et al. “Torontocity: Seeing the world with a million eyes”. In: *arXiv preprint arXiv:1612.00423* (2016).

[63] *Webots*. URL: https://cyberbotics.com/ (visited on 2019-02-22).

[64] Wikipedia. *Unreal (video game series) — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Unreal%20(video%20game%20series)&oldid=881305745. [Online; accessed 21-March-2019]. 2019.

[65] Wikipedia. *Unreal Engine*. URL: https://en.wikipedia.org/wiki/Unreal_Engine (visited on 2019-02-22).

[66] Ryan W Wolcott and Ryan M Eustice. “Visual localization within lidar maps for automated urban driving”. In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2014, pp. 176–183.

[67] H.D. Young, R.A. Freedman, and A.L. Ford. *University Physics with Modern Physics*. Pearson Education, 2012. ISBN: 9780321850027. URL: https://books.google.fi/books?id=5fgsAAAAQBAJ.

[68] Chao Zhu, Jin Tao, Giancarlo Pastor, Yu Xiao, Yusheng Ji, Quan Zhou, Yong Li, and Antti Ylä-Jääski. “Folo: Latency and Quality Optimized Task Allocation in Vehicular Fog Computing”. In: *IEEE Internet of Things Journal* (2018).