TALLINN UNIVERSITY OF TECHNOLOGY
SCHOOL OF ENGINEERING
Department of Electrical Power Engineering and Mechatronics

ESTABLISHING UNITY TO DIGITAL TWIN HARDWARE COMMUNICATION THROUGH ROS FOR ENHANCED INTERACTIVITY AND REAL-TIME SIMULATION

ROS-I KAUDU SIDE LOOMINE UNITY JA DIGITAALSE KAKSIKU RIISTVARA VAHEL TÄIUSTATUD INTERAKTIIVSUSE JA REAALAJA SIMULATSIOONI TAGAMISEKS

MASTER THESIS

Student: Diana Belolipetskaja

Student code: 212098MAHM

Supervisor: Hadi Ashraf Raja, Researcher

Co-supervisor: Anton Rassõlkin, Tenured Associate Professor

Tallinn, 2024

(*On the reverse side of title page*)

AUTHOR'S DECLARATION

Hereby I declare, that I have written this thesis independently.
No academic degree has been applied for based on this material. All works, major viewpoints and data of the other authors used in this thesis have been referenced.

"....." ..... 202…..

Author: .....
/signature /

Thesis is in accordance with terms and requirements

"....." ..... 202….

Supervisor: ….....
/signature/

Accepted for defence

"....."..... 202… .

Chairman of theses defence commission: .....
/name and signature/

Non-exclusive Licence for Publication and Reproduction of Graduation Thesis[1]

I, Diana Belolipetskaja hereby

1. grant Tallinn University of Technology (TalTech) a non-exclusive license for my thesis "Establishing Unity to Digital Twin Hardware communication through ROS for enhanced interactivity and Real-Time Simulation",

supervised by Hadi Ashraf Raja and co-supervised by Anton Rassõlkin,

1.1 reproduced for the purposes of preservation and electronic publication, incl. to be entered in the digital collection of TalTech library until expiry of the term of copyright;

1.2 published via the web of TalTech, incl. to be entered in the digital collection of TalTech library until expiry of the term of copyright.

1.3 I am aware that the author also retains the rights specified in clause 1 of this license.

2. I confirm that granting the non-exclusive license does not infringe third persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

_____ (*date)*

[1] *The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.*

ABSTRACT

Author: Diana Belolipetskaja *Type of the work:* Master Thesis

Title: Establishing Unity to Digital Twin Hardware communication through ROS for enhanced interactivity and Real-Time Simulation

Date: 13.05.2024 72 *pages (the number of thesis pages including appendices)*

University: Tallinn University of Technology

School: School of Engineering

Department: Department of Electrical Power Engineering and Mechatronics

Supervisor(s) of the thesis: Hadi Ashraf Raja, Researcher; Anton Rassõlkin, Tenured Associate Professor

Consultant(s):

Abstract:

Digital twin technology in the automotive industry is a rapidly expanding trend that offers precise, cost-effective simulation and testing environments, as well as advanced platforms for predictive maintenance. Autonomous vehicles stand out as a unique case due to their extensive sensor arrays and the vast amounts of data they generate, enabling the creation of highly detailed models and environments for testing and analysis. For managing this extensive data, a robust and reliable connection is crucial for accurate analysis and maintenance prediction.

This thesis aims to establish a seamless connection between digital twin hardware — specifically, motors that simulate road load for electric vehicle propulsion drive systems— and visualization software with a user interface. Real-time connectivity is a primary requirement for the digital twin project, so this work also carefully selects the appropriate microcontroller to ensure the fastest possible data transmission. Selection of the right microcontroller is key to achieving accurate, efficient data analysis and reliable predictive maintenance. To select the proper microcontroller number of tests were conducted to compare the performance of each microcontroller.

Keywords: digital twin, middleware, ROS, connection, microcontroller, Unity, visualization.

LÕPUTÖÖ LÜHIKOKKUVÕTE

Autor: Diana Belolipetskaja *Lõputöö liik:* Magistritöö

Töö pealkiri: ROS-i kaudu side loomine Unity ja digitaalse kaksiku riistvara vahel täiustatud interaktiivsuse ja reaalaja simulatsiooni tagamiseks

Kuupäev: 13.05.2024 72 *lk (lõputöö lehekülgede arv koos lisadega)*

Ülikool: Tallinna Tehnikaülikool

Teaduskond: Inseneriteaduskond

Instituut: Elektroenergeetika ja mehhatroonika instituut

Töö juhendaja(d): Hadi Ashraf Raja, teadur; Anton Rassõlkin, kaasprofessor tenuuris

Töö konsultant (konsultandid):

Sisu kirjeldus:

Digitaalse kaksiku tehnoloogia autotööstuses on kiiresti arenev trend, mis pakub täpseid, kulutõhusaid simulatsiooni- ja testimiskeskkondi ning täiustatud platvorme ennustavaks hoolduseks. Autonoomsed sõidukid paistavad silma ainulaadse juhtumina oma ulatuslike andurimassiivide ja genereeritavate tohutute andmehulkade tõttu, mis võimaldavad luua väga detaailseid mudeleid ja testimise ning analüüsimise keskkondi. Nende ulatuslike andmete haldamiseks on täpse analüüsi ja hoolduse prognoosimise jaoks üliolulline tugev ja usaldusväärne side mudelite vahel.

Selle lõputöö eesmärk on luua sujuv ühendus digitaalse kaksiku riistvara – täpsemalt mootorid, mis simuleerivad teekoormust elektrisõidukite veoajamisüsteemide jaoks – ja visualiseerimistarkvara vahel. Digitaalse kaksiku projekti üks peamistest nõuetest on reaalaja ühendus ja andmeedastus, seega valitakse selle töö käigus hoolikalt ka sobiv mikrokontroller, et tagada võimalikult kiire andmeedastus. Õige mikrokontrolleri valik on täpse, tõhusa andmeanalüüsi ja usaldusväärse prognoositava hoolduse saavutamise võtmeks. Õige mikrokontrolleri valimiseks viidi läbi mitmeid teste, et võrrelda iga mikrokontrolleri jõudlust.

Märksõnad: digitaal kaksik, vahevara, ROS, side, mikrokontroller, Unity, visualisatsioon.

Department of Electrical Power Engineering and Mechatronics

THESIS TASK

Student: Diana Belolipetskaja, 212098MAHM

Study programme, main speciality: MAHM Mechatronics

Supervisor(s): Hadi Ashraf Raja, Researcher; Anton Rassõlkin, Tenured Associate Professor

Thesis topic:

(in English) Establishing Unity to Digital Twin Hardware communication through ROS for enhanced interactivity and Real-Time Simulation

(in Estonian) ROS-i kaudu side loomine Unity ja digitaalse kaksiku riistvara vahel täiustatud interaktiivsuse ja reaalaja simulatsiooni tagamiseks

Thesis main objectives:

1. Test different types of microcontrollers and choose the most suitable one
2. Configure the ROS environment on the PC
3. Create a connection between ROS and microcontroller to control road load simulating motor
4. Create a connection between ROS and the virtual environment
5. Test the final solution with a full connection between the virtual environment and hardware

Thesis task and time schedule:

No	Task description	Deadline
1.	Literature review	24.12.2023
2.	Creating a connection between the road load simulating motors and ROS	20.01.2024
3.	Creating a connection between Unity and ROS	24.02.2024

4.	Testing different microcontrollers and analysing the results	20.03.2024
5.	Creating the full connection between the virtual environment and hardware	04.04.2024
6.	Writing the results of the work	13.05.2024

Language: English **Deadline for submission of thesis:** "13" May 2024

Student: …….. ..... "....."..... 20…..a
/signature /

Supervisor: …….. ..... "....."..... 20…..a
/signature/

Consultant: …….. ..... "....."..... 20…..a
/signature/

Head of study programme:

…….....…….... "....."..... 20…..a
/signature/

CONTENTS

PREFACE

This master's thesis describes the development of a connection between digital twin hardware and visualization software, and the selection of appropriate components for this task.

I would like to express my gratitude to Hadi Ashraf Raja and Anton Rassõlkin for their mentoring throughout the entire process, and to the members of the Mechatronics and Autonomous Systems Research Group for their support and assistance during this work. Additionally, I want to thank my friends and family for their encouragement and support.

LIST OF ABBREVIATIONS AND SYMBOLS

3D	3 Dimensional
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter
DT	Digital Twin
EV	Electrical Vehicle
IoT	Internet of Things
LCM	Lightweight Communications and Marshalling
MCU	Microcontroller Unit
OS	Operating System
PC	Personal Computer
PDS	Propulsion Drive System
PLA	Polylactic Acid
PWM	Pulse Width Modulation
ROS	Robot Operating System
SLAM	Simultaneous Localization and Mapping
SOC	State of Charge
SOH	State of Health
TB	Test Bench
UAV	Unmanned Aerial Vehicle
UI	User Interface
USB	Universal Serial Bus
vSLAM	Visual Simultaneous Localization and Mapping

YARP Yet Another Reverse Proxy

1 INTRODUCTION

In the dynamic landscape of modern industrial and technological advancements, the convergence of Digital Twins (DT), simulation environments, and user interfaces (UI) has emerged, revolutionizing the way people conceptualize and interact with complex systems. As people all around the world adopt the exciting possibilities of Industry 4.0 and the Internet of Things (IoT), combining real-world things with their digital versions is extremely important for moving technology forward. This mix of physical hardware and its digital representations changes the whole life cycle of the product, technology or service and it helps speed up new ideas, accelerate prototyping, make things work better, predict when maintenance is needed, and make the resource consumption more environmentally friendly. It's a big deal for many different industries, like making things, healthcare, energy, and transportation.

In the exciting future of digital technology, a significant challenge lies in ensuring seamless communication between real-world objects, their digital representations (incl. dedicated models), and the control mechanisms, enabling the transmission of accurate data at desired speeds to facilitate thorough data analysis. This is particularly crucial in applications where human life might be at risk, and even a few milliseconds of delay can prove fatal, such as in autonomous vehicles. This underscores the importance of robust communication systems capable of delivering data swiftly and accurately. Moreover, proper data analysis could lead to better power consumption and optimization, further highlighting the significance of effective communication protocols. This thesis is dedicated to addressing communication challenges by focusing on the development of reliable communication between physical objects and their models in the virtual environment, with the ultimate aim of enhancing safety and efficiency in such sort of applications.

This thesis work is a part of the research project, which is focused on developing a DT for the Propulsion drive of Autonomous Electric Vehicle (EV), that is supported by the Estonian Research Council under grant PSG453. The thesis aims to create reliable communication between the real physical motors, which serve as loading drive systems of a called demonstrator and visualization environment Unity by implementing Robot Operating System (ROS) middleware. For this, selecting of the appropriate microcontroller unit (MCU) is crucial to ensure the transmission and reception of data at the required frequency. Moreover, to be able to send and receive digital and/or analog signals to the personal computer (PC), the implementation of an MCU is necessary, adding to the complexity and depth of the project.

2 LITERATURE REVIEW

2.1 Digital twins

At the intersection of academia and industry, DTs have merged as a game-changing idea that is revolutionizing the way we design, monitor, and optimize complex systems. The main function of the DTs is to give a detailed, functional, physical, and comprehensive description of the system as a whole that can operate simultaneously with real systems. DTs are virtual models that realistically replicate the characteristics, actions, and behaviours of the physical part's entities. Stated otherwise, virtual models and real-world entities have comparable appearances and behaviours – that is, they are twins [1].

DT has found widespread implementation across various fields, changing the way industries operate and innovate. In manufacturing, DTs enable virtual prototyping, optimize production processes, and facilitate predictive maintenance, leading to increased efficiency and reduced downtime [1]. In agriculture, the DTs with their real-time virtual representation and union of data, modeling, and what-if simulation create a solution to overcome limitations in automation and decision-making support in diverse agricultural fields [2]. In healthcare, DTs are utilized for personalized medicine, patient monitoring, and surgical simulations, allowing for better diagnosis and treatment planning [3]. Urban planning benefits from DTs by optimizing city infrastructure, enhancing traffic management, and simulating environmental impacts for sustainable development [4]. With their versatility and transformative potential, DTs continue to redefine how we conceptualize, design, and manage systems across diverse domains, promising a future of innovation and efficiency. Their vast potential can be implemented across various fields, as shown in Figure 2.1, including the automotive industry.

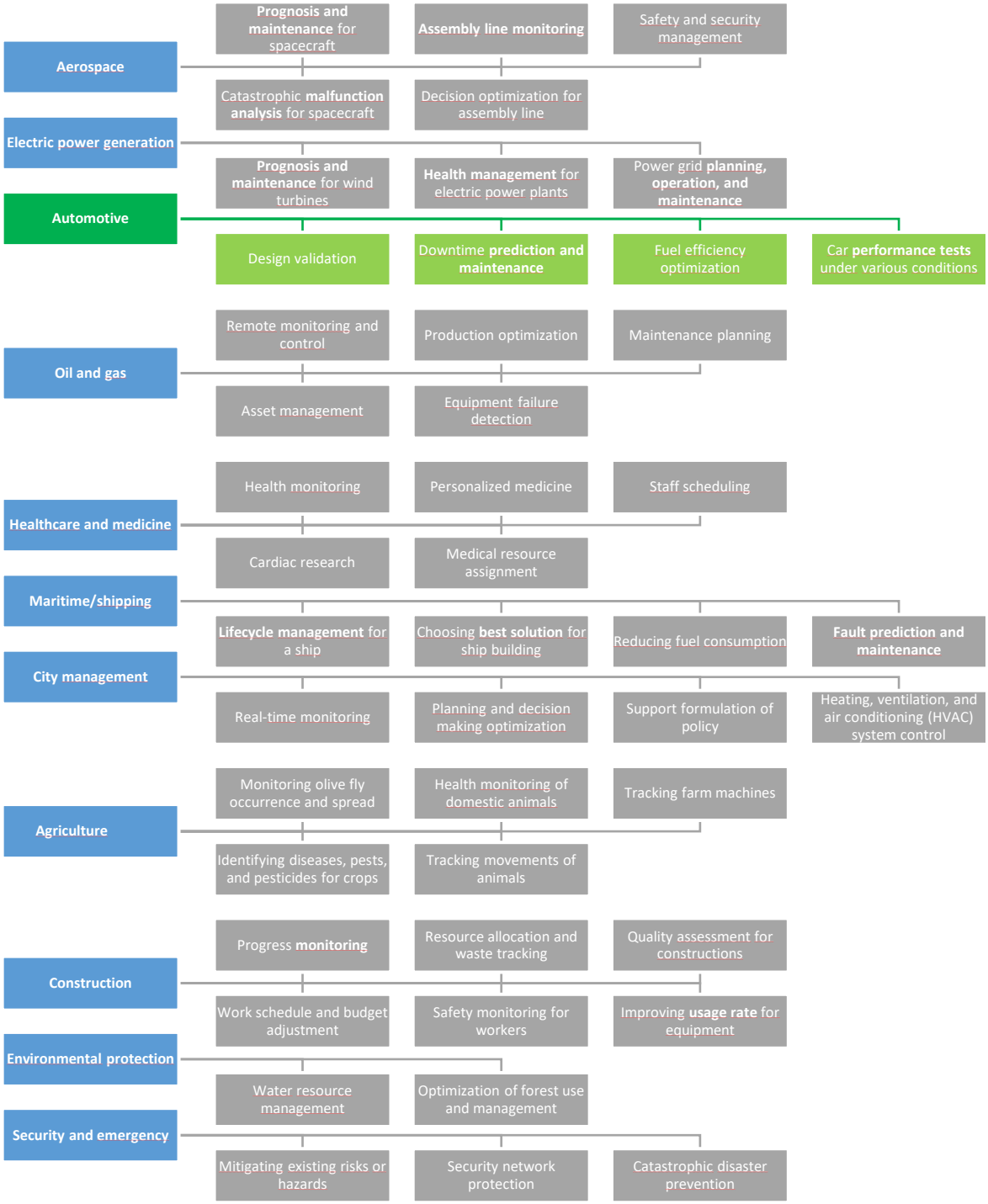


Figure 2.1 Analysis of DT potential implementation across different industry fields

2.1.1 Application of digital twin technology within the automotive industry

Among growing concerns about climate change, the automotive industry faces increasing pressure to transition towards sustainable transportation solutions. With transportation accounting for 18% of global carbon dioxide emissions as of 2019, there's an urgent need to adopt measures to reduce this number [5]. DT technology presents a promising avenue for achieving this transition by enabling the development of more energy-efficient and environmentally friendly vehicles. By creating virtual replicas of cars and their components, DTs facilitate advanced simulations and optimizations that lead to reduced fuel consumption, lower emissions, and improved overall sustainability in the automotive sector.

Almeaibed et al. [6] highlights the importance of addressing safety and security concerns in autonomous vehicles. In light of the ongoing digital revolution, the adoption of intelligent, data-driven systems holds promise for enhancing safety, efficiency, and security across transportation processes. At the core of this transformative change lies the concept of DTs, which allows sophisticated simulations and continuous real-time monitoring. Through a case study on a vehicle follower model, the article demonstrates the effectiveness of DTs in automating decision-making processes and reducing risks associated with cyber-attacks and accidents. Emphasizing the need for robust data transmission, reception, and processing, the article calls for heightened attention to privacy, safety, and security considerations in the development of autonomous vehicles.

In [7] Eaty and Bagade discussed the application of DT technology in the field of electrical vehicle battery management, aiming to improve predictive maintenance and prolong battery lifespan. With the increasing number of electrical vehicles worldwide, battery state monitoring become a significant concern, encompassing various challenges such as safety and range estimation. The article presents a DT framework for electrical vehicle battery management, where the State of Health (SOH) is predicted in the cloud while the State of Charge (SOC) is estimated on the vehicle. This framework employs a continuous learning method for SOH prediction and utilizes the Kalman filter for SOC estimation and experimental results show promising accuracy in predicting SOH.

DT technology is also being implemented in the driver assistant field. Liao et al. [8] introduce a DT framework for connected vehicles, integrating physical and cyber layers with various modules. Specifically, an advanced driver assistance system utilizing vehicle-to-cloud communication is developed. Through vehicle-to-cloud, data is uploaded from on-board devices to a server, which then creates a virtual world,

processes the data, and sends it back to vehicles. A case study on cooperative ramp merging demonstrates the framework's benefits for mobility and environmental sustainability, despite communication delays and packet losses.

The potential applications of DTs technology in the automotive industry are vast. As depicted in Figure 2.2, various facets of automotive operations can benefit from DTs implementation. From autonomous navigation to manufacturing processes, predictive maintenance, and even customer service, DTs offer a range of solutions to enhance efficiency, productivity, and innovation throughout the automotive value chain. By providing virtual representations of physical vehicles and systems, DTs enable detailed simulations, real-time monitoring, and data-driven decision-making.

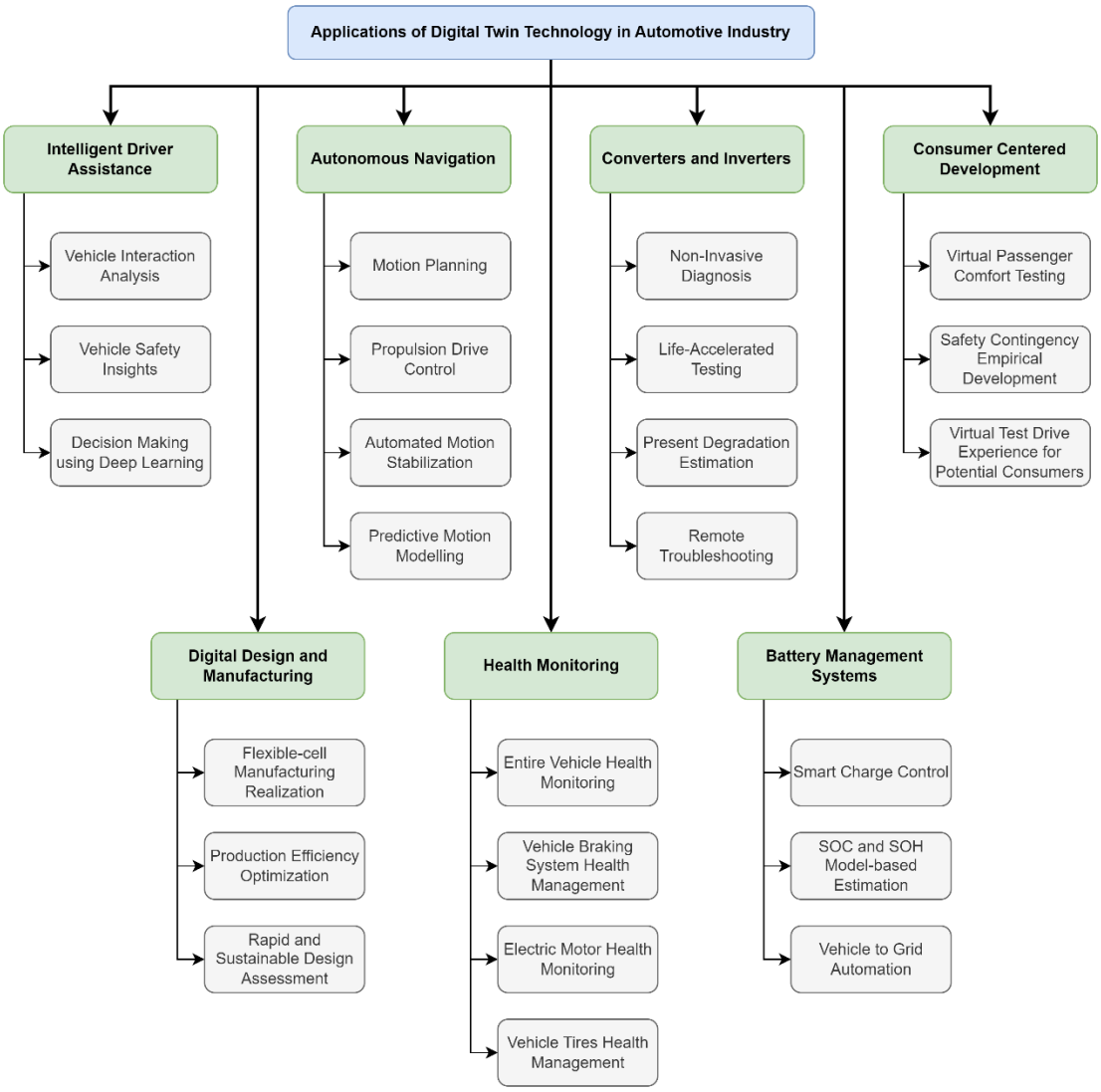


Figure 2.2 Possible applications of DT technology in automotive industry[5]

By synthesizing insights and data from various DT application fields, the automotive industry can adopt a comprehensive, interconnected strategy for design, production, navigation, and customer satisfaction. This integrated approach ultimately enhances efficiency, safety, and innovation across the entire value chain.

For instance, non-invasive diagnosis and life-accelerated testing in the converters and inverters field can significantly improve health monitoring, particularly in evaluating a vehicle's overall health. Real-time degradation estimation provides crucial data for monitoring individual components like braking systems and electric motors, enabling timely maintenance and improved safety.

In another example, integrating autonomous navigation with battery management optimizes motion planning and predictive modeling. By including battery data, such as SOC and SOH, autonomous vehicle navigation can adjust to battery constraints, extending the driving range and maximizing energy efficiency. Smart charge control can also influence propulsion drive control, enabling vehicles to make more energy-conscious decisions based on their battery status.

These strategies can help refine vehicle manufacturing and usage, reducing the consumption of materials and resources while promoting a more sustainable and efficient automotive industry.

2.1.2 Digital twin for propulsion drive of autonomous electric vehicle

DT for propulsion drive of autonomous EV is a project with the goal to develop and create an unsupervised prognosis and control platform for EV propulsion drive system (PDS) performance estimation and monitoring. This goal includes developing physical models of different energy systems components, development and implementation of virtual sensors for DT concept and additionally develop an artificial intelligence-based system that can use virtual sensors. The research platform is based on the ISEAUTO self-driving shuttle [9].

The DT of this project consists of the real physical entity, virtual entity, service entity, data and connections between all the parts. The general concept of DT and it parts is shown in the Figure 2.3.

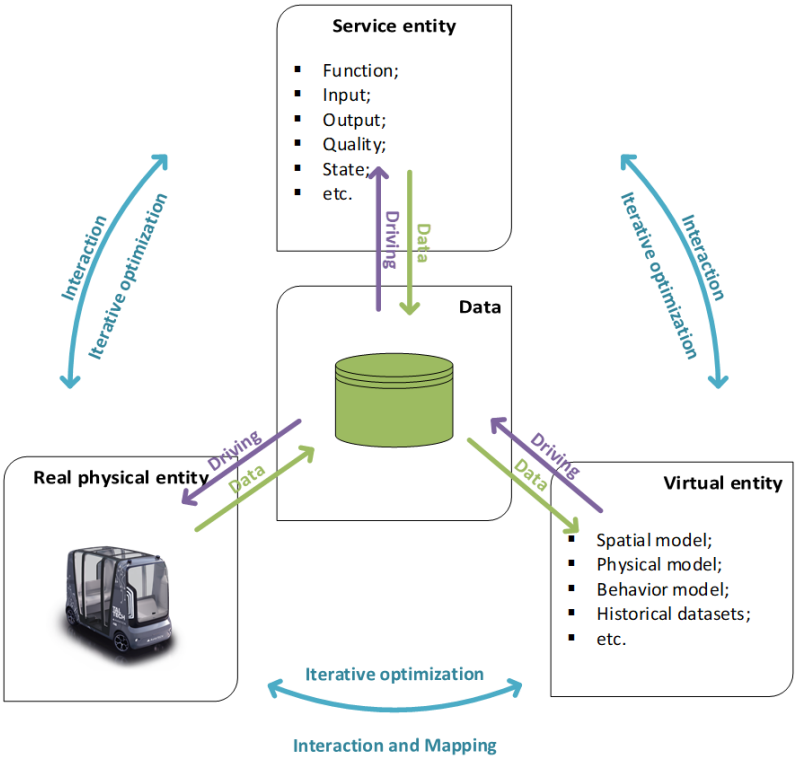
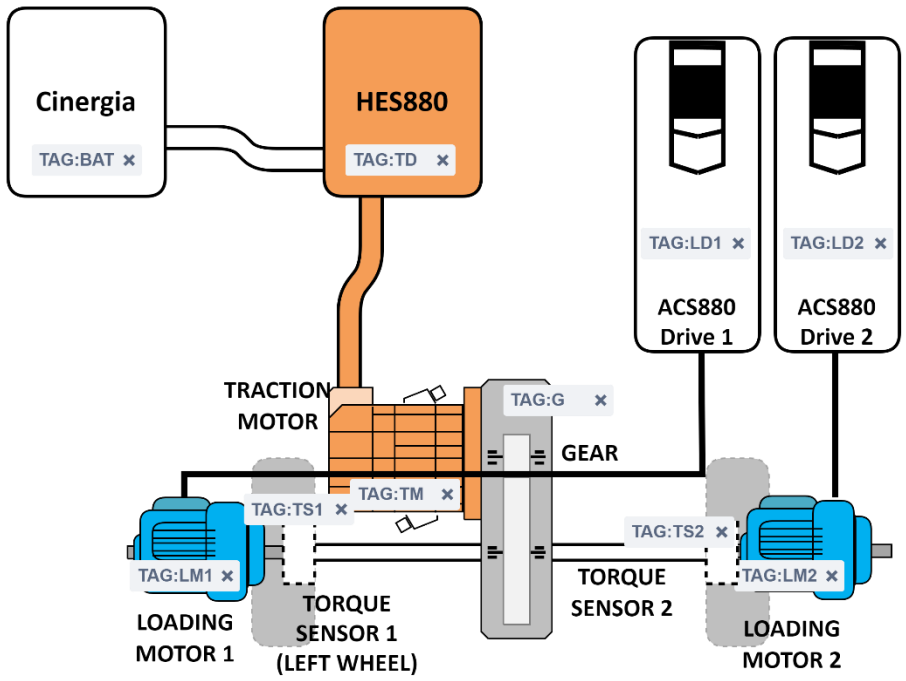
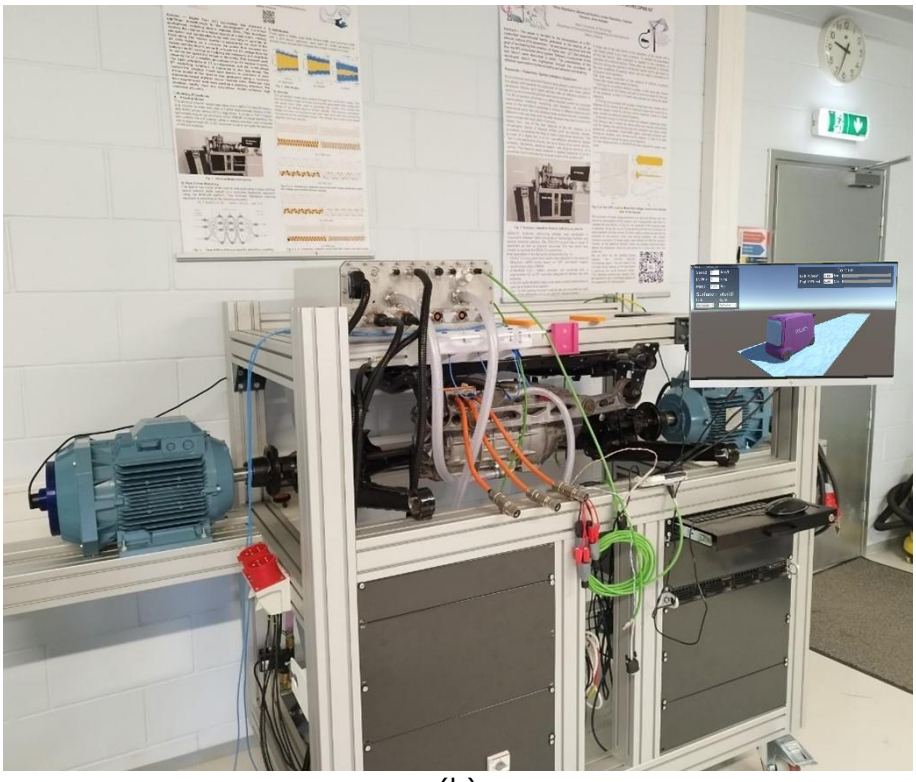


Figure 2.3 DT concept of the DT for PDS of autonomous EV project [9]

In the real world, the physical entity is presented by the ISEAUTO, but in order to be able to collect the data in a controlled environment, the reduced demonstrator or test bench (TB) was built to collect the data in the laboratory environment. TB consists of an EV PDS system identical to the ISEAUTO self-driving shuttle, which consists of a Mitsubishi i-MiEV PMSM traction motor drive and dedicated mechanical transmission. Commercial motor Y4F1 is controlled by an ABB HES880 frequency converter, which adjusts the power supplied to the motor according to predetermined parameters. The HES880, is powered by a Cinergia B2C+ battery tester and emulation system. The output of the Y4F1 motor is transmitted through a gearbox to a shaft. This shaft is connected to two loading motors (ABB induction motors 3GAA132214-ADE), which simulate the loads placed on the traction motor. Each loading motor is linked to an ABB ACS880 frequency converter, which regulates the torque supplied to the motor based on predetermined parameters. The principal diagram and general view of studied TB is shown in Figure 2.4.



(a)



(b)

Figure 2.4 Principal diagram (a) and general view in the laboratory (b) of the test bench

The virtual entity is represented by the virtual environment provided by the Unity engine, where the 3D model of the ISEAUTO vehicle and the road are presented. Through the UI, users can specify road conditions such as slope and road surface conditions for both the left and right sides of the road separately. Currently, available road surfaces are dry asphalt, snow, and ice. Additionally, users can control the vehicle speed and adjust the mass of the vehicle, which represents factors such as the number of occupants. As for feedback, the virtual environment can display the torque currently applied to each wheel. The virtual environment is represented in Figure 2.5.

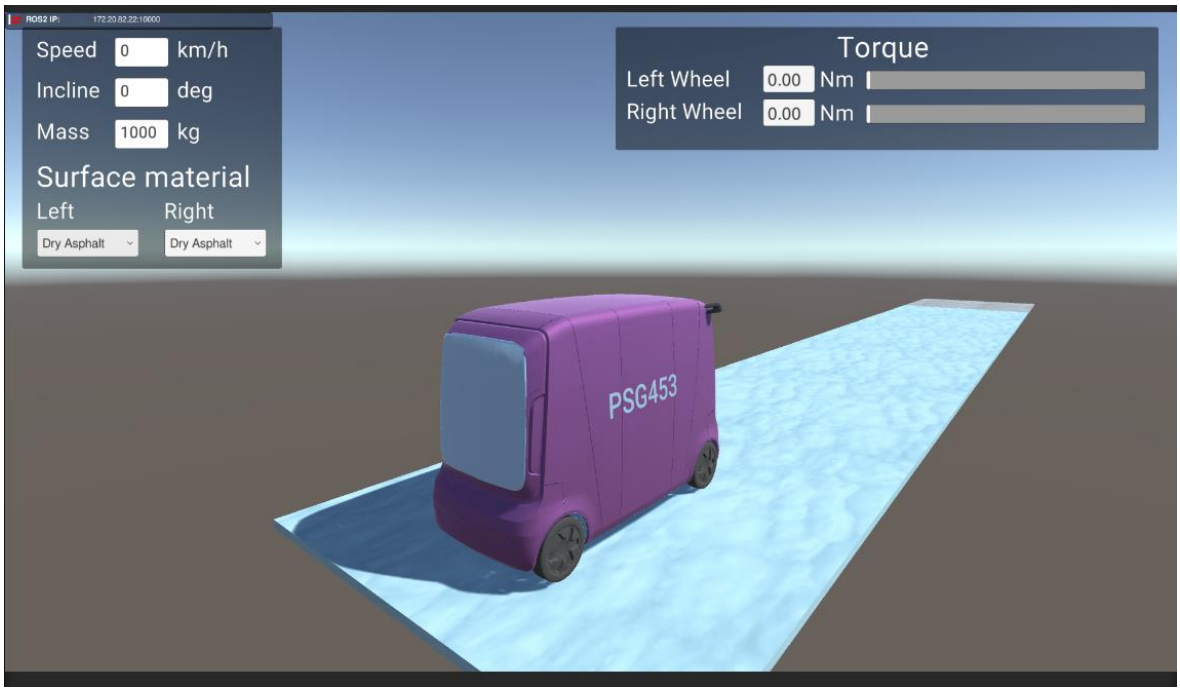


Figure 2.5 DT virtual environment with ISEAUTO 3D model and user interface

Service entity presents an integrated platform that supplies the needs of both physical and virtual systems. It encompasses systems that provide services related to data modeling, optimization, and prediction. The data generated by these systems, including physical, virtual, and service systems, is combined to create a more comprehensive and consistent dataset. The DT can utilize this combined data to optimize its performance, potentially employing deep learning tools for optimization before implementation on the physical machine [9].

2.2 Middleware layer

In modern technological solutions, many sensors and actuators are used to collect data about different physical conditions. These peripheral devices present a hardware layer. The gathered data is sent to the different programs or mathematical models that process the data, which present the software layer. If there is a lot of components on both of the layers the end working solution can be very complex and look like the scheme shown in Figure 2.6. For the user that needs to work with this system, it could be very hard to understand what data is sent to what program and what controls what.

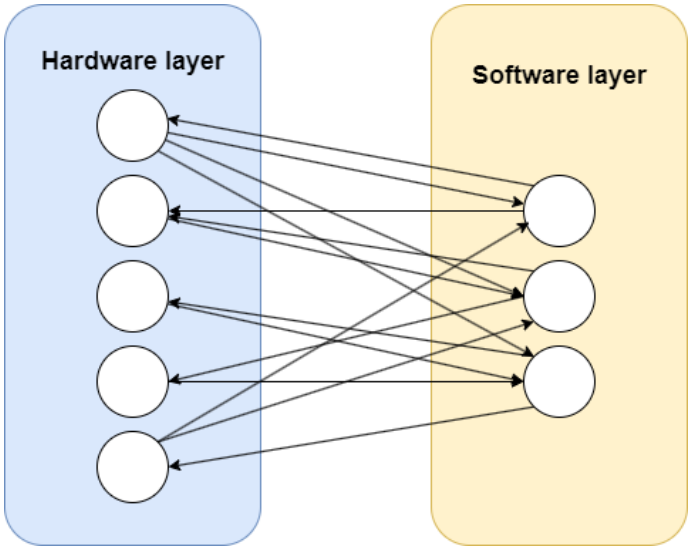


Figure 2.6 Example scheme of how can be arranged connection between hardware and software layer in the technical solution.

To simplify and optimize the connection between hardware layer components and different software solutions, middleware can be used. Middleware is a middle layer between sensors, services, and applications that manages the data flow and allows them to interoperate [10]. Middleware can be considered as a roundabout that connects different roads and manages to send the vehicles to the needed endpoint instead of connecting all the houses with direct roads. The illustrative example of middleware implementation is shown in Figure 2.7. In this case middleware layer will take the data from the hardware layer and store it in its folders or files, where every software would be able to get this data, process it, and send the commands to another file or folder. Actuators that need to react to those commands will see them and act accordingly. This solution reduces the number of connections and structures of the end solution making it more clear for the end user to understand the data flow in the system.

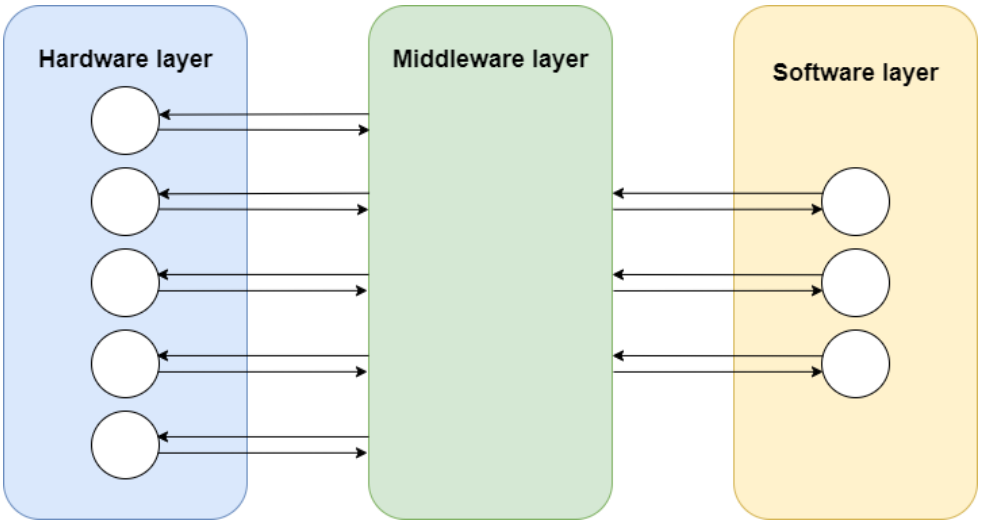


Figure 2.7 Example scheme of how can be arranged connection between hardware and software layer in technical solution by implementing middleware layer

This thesis is focused on the implementation of a middleware solution for the DT for PDS of autonomous EV project, aiming to establish a connection between the loading motors of DT TB, the middleware, and the virtual environment for UI and visualization.

2.2.1 Middleware choice for Digital Twin Project

The selection of middleware hinges on various factors, and as outlined in [11], the project criteria were comprehensively assessed while comparing different middleware options. Among frameworks such as Yet Another Reverse Proxy (YARP), Lightweight Communications and Marshalling (LCM), and Robot Operation System (ingl. ROS and ROS2), ROS2 emerged as the ideal choice due to its robust publish-subscribe messaging architecture, real-time support, distributed network topology, active developer community, native embedded compatibility, and extensive documentation. Moreover, ISEAUTO self-driving shuttle is using ROS based application Autoware [12] and further implementation of developed DT will be smoother.

2.3 ROS

ROS is open-source middleware that contains a set of software libraries and tools. Despite the name, ROS is not an operating system (OS) but rather a collection of tools and libraries that provide a standardized way for robot hardware to communicate with each other or to communicate with the environment [13].

ROS originated from projects at Stanford University in the mid-2000s, such as the Stanford AI Robot and Personal Robots programs. In 2007, Willow Garage Inc., a robotics incubator, played a significant role in extending the system's concepts and facilitating collaboration among researchers. The software was developed openly under a BSD license, leading to widespread adoption in the robotics research community. Initially viewed as challenging due to decentralized development across multiple institutions, this model eventually became a strength, as it allowed various groups to establish their repositories, fostering recognition, collaboration, and technical feedback. Today, ROS boasts tens of thousands of users globally, supporting diverse robotics applications ranging from hobbyist projects to industrial automation [14].

ROS has a list of different functionalities such as message-passing between processes, hardware abstraction, low-level device control, package management, and implementation of commonly used functionality. Additionally, ROS has many tools that help users in developing their robotics projects such as Gazebo 3D simulator, RViz, etc. ROS is used in academia, research institutions, and industry to develop, control, and integrate robotic systems into various platforms [14].

ROS has main concepts that provide a standardized way to build and integrate software for robotic systems, helping streamline development, ensure compatibility across projects, and leverage shared tools, libraries, and best practices. Mastering these concepts allows developers to create more innovative and efficient robotic applications that can interact seamlessly with other ROS-based systems. The main concepts of ROS [15]:

- **Nodes** - individual software processes that perform specific tasks. Nodes communicate with each other by publishing and subscribing to messages on topics.

- **Topics** - named buses over which nodes exchange messages. Nodes can publish messages to a topic or subscribe to receive messages from it.

- **Messages** - the data structures used to communicate between nodes. They can be predefined types or custom-defined by users.
- **Services** - allow nodes to send requests and receive responses. They are synchronous communication channels that enable more complex interactions compared to topics.
- **Actions** - provide a way to execute long-running tasks in a more flexible and robust manner compared to services. They enable asynchronous communication with feedback and goal tracking.
- **Launch files** - XML files used to launch multiple nodes with specific parameters and configurations in a single command.
- **Packages** - the organizational unit of ROS code. They contain nodes, launch files, configuration files, and other resources related to a specific functionality or robot component.

2.3.1 ROS vs ROS2

ROS has two main versions: ROS1 and ROS2. ROS1 is the first version of ROS, which was released in 2009 and is no longer being developed. Instead, ROS2 was released in 2017 to adapt to modern changes in robotics and improve ROS. The differences between ROS1 and ROS2 are presented in Table 2.1.

Table 2.1 Comparision of ROS1 and ROS2

ROS1	ROS2
Supported OS platforms	
Ubuntu	Ubuntu, macOS, Windows, Red Hat Enterprise Linux (RHEL)
Languages	
Python 2 and 3; C++ 98, 11, 14	Python 3; C++ 11, 14, 17
Real-time communication support	
No	Yes
Development	
Not developing anymore, the last distribution will reach the end of a lifetime in May 2025	The new distribution of ROS2 is released every year

As the table shows, ROS2 supports newer versions of programming languages, can be used on more OS platforms, supports real-time communication, and is still being developed by the community. Compared to ROS1, ROS2 is a more suitable choice for developing robotic systems nowadays, and it has real-time communication support, which is crucial for DT development.

2.3.2 ROS implementation in robotic systems

ROS has proven to be a versatile and widely adopted framework in the realm of robotics. The main real-world applications for ROS are industrial robots and autonomous vehicles. These fields are suitable for ROS implementation as they have a big data flow that needs to be structured and optimized for better performance.

In [16], the integration of the ABB industrial robot manipulator with ROS is described. The workflow involves connecting a ROS environment to a DT of ABB robots in an automated welding process station via RobotStudio. The work was tested in an automated welding process for steel looped hooks, demonstrating the efficiency of LERP-based planning for point-to-point trajectories. The results highlight the advantages of LERP-based planning, showing fast planning times, consistent path findings, and a 0% collision rate, outperforming widely used path planning algorithms.

One example of ROS usage with unmanned aerial vehicles (UAVs) is described in [17]. Research is focused on solving the problem of autonomous navigation when UAVs can not get external assistance in the complex environment of the coal mine. The UAV runs the visual simultaneous localization and mapping (vSLAM) algorithm under the framework of ROS, which helps to independently complete the positioning and navigation tasks.

ROS is widely used for different autonomous vehicles that ride on different surfaces (ground, sand, snow, water, etc.). The research of using ROS on an unmanned surface vehicle that rides on water is written in [18]. This study presents the development of a ROS-based autonomous navigation system for vessels, encompassing simultaneous localization and mapping (SLAM), path planning, and obstacle avoidance.

However industrial robots and autonomous vehicles are not the only fields where ROS can be used. In [1], a development of ROS-based software architecture for DT creation.

Its primary function is to allow direct communication and interaction, making it easier the exchange of data and information between diverse components, both hardware and software included, in a versatile robotized manufacturing environment. This work also shows that ROS can be used in DT solutions for communication and to provide a reliable connection for this kind of task.

Based on those works it is clear that ROS is a powerful tool that helps researchers to develop their ideas by providing different tools and libraries that can be used for robot systems development and other applications.

2.4 Section summary

In this section, the main concepts of DT technology and its implementation across various industries are explored. The focus is on the automotive industry, particularly how DTs offer optimization and monitoring solutions for electric vehicles. The potential of DTs in this sector is underscored by their role in advanced simulations and data-driven decision-making, which lead to improved vehicle efficiency and sustainability.

An overview of the DT framework for propulsion drive of autonomous EV project is provided. This framework requires a robust connection between the visualization and hardware components of the TB. The research gap identified involves the integration of visualization and hardware, emphasizing the need for seamless communication between them.

To address this gap, a comprehensive review of the ROS middleware and its application areas is conducted. ROS2 is presented as the ideal middleware solution for this project due to its ability to facilitate real-time, distributed communication across various components of the DT system. Its advanced features will enable efficient data flow and seamless integration between the physical and virtual entities, ultimately enhancing the performance and reliability of the DT system.

3 MICROCONTROLLERS

DT system for PDS architecture can be divided into five layers for a clear understanding of each layer and how they interact with each other. The architecture consists of visualization and UI, middleware, signal processing, hardware control, and actuators and sensors. The architecture is shown in Figure 3.1.

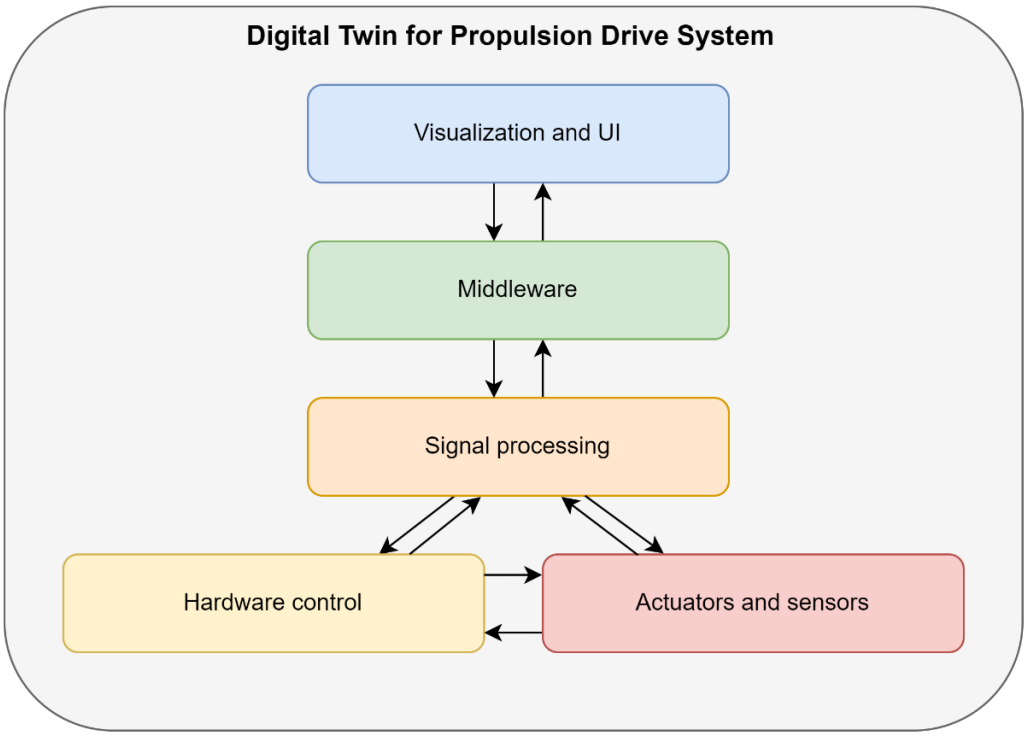


Figure 3.1 The architecture of the digital twin system for propulsion drive of autonomous electrical vehicle

Detailed description of each layer, as well as their connections:

- **The visualization and UI layer** serves as the front end of the DT system, providing graphical representations and UIs to interact with the digital model. It visualizes data, allows user inputs, and displays real-time system statuses. For this purpose, the Unity engine is used. It has a connection with middleware (in this case ROS2) to transmit user commands and receive feedback through the middleware layer to ensure the user's interactions are reflected in the system accurately.

- **The middleware layer** acts as the communication hub of the system, linking the UI with lower-level operations. It processes data from the UI and translates

it into actionable commands for the lower layer while also handling data coming back from lower layers to deliver it to the end user.

- **The signal processing layer** processes the commands and data into a form that can be used to control hardware devices. It can convert digital commands into analog signals, filter and analyse sensor data, and perform necessary calculations or data transformations.

- **The hardware control layer** includes the physical controllers (like motor drivers and frequency converters) that directly interface with the hardware components. It interprets the signals from the signal processing layer and translates them into specific actions performed by the hardware.

- **The actuators and sensors layer** is the physical layer that interacts with the real-world environment. Actuators execute physical actions based on the commands received via the hardware control layer, and sensors monitor conditions and send data back through the system.

This structured approach allows each layer to focus on its specific role while supporting the seamless function of the overall system. By understanding how each layer operates and connects, developers and engineers can better optimize each part of the DT for efficiency, performance, and reliability.

For the signal processing layer, one of the most common solutions is using MCUs. MCUs are compact, integrated circuits that serve as the brains of countless electronic devices, providing the computational power necessary for their functionality, which can be easily integrated with PCs through USB or other types of connections. These boards usually come with a central processing unit (CPU), memory, and input/output peripherals. The variety of MCUs nowadays is extensive, with numerous companies producing different kinds of MCUs with different functionalities, because of which choosing the suitable one could be challenging[19].

This thesis focuses on controlling the loading motors of a DT system that receives road load information from the Unity virtual environment associated with the DT. The control of the loading motor is facilitated by an ABB ACS880 frequency converter, which utilizes analog input signal pins to regulate the motor's torque. The voltage range of the analog signal can be adjusted on the inverter to align with the capabilities of the MCU. Consequently, a crucial criterion for selecting an MCU is its ability to generate the required voltage output. This can be achieved through either a Digital-to-Analog

Converter (DAC) or Pulse Width Modulation (PWM). The seamless flow of this data into the motor control system is crucial for the accurate physical representation of virtual scenarios.

3.1 Overview of relevant microcontrollers

One important criterion for selecting proper MCUs for the final solution is their capability to connect with ROS2. For this purpose, ROS2 has micro-ROS that can be installed on the 32-bit MCUs, and only certain models of MCUs can support that. Table 2.2 provides a comprehensive list of all MCUs officially supported by ROS2, comparing them across various factors[20]. According to [21], the Espressif ESP32 series boards are supported. However, it is important to note that a successful connection has specifically been demonstrated with the Espressif ESP32-DevKitC-32E. Therefore, this board will be used for comparison in tests. Additionally, from the list of comparisons, the Crazyflie 2.1 Drone has been excluded since its application is specifically targeted at drones and not suitable for broader MCU testing scenarios.

Table 3.1 Overview of microcontrollers compatible with micro-ROS

MCU	CPU	Dimensions (mm)	Price (euros)*
Renesas EK RA6M5	ARM Cortex M-33 core @ 200 MHz	80 x 180	199.00
Espressif ESP32-DevKitC-32E	Xtensa dual-core 32-bit LX6 microprocessor, up to 240 MHz	54.4 x 27.9	10.00
Arduino Portenta H7	Dual-core Arm Cortex-M7 and Cortex-M4	25.40 x 66.04	99.00
Raspberry Pi Pico RP2040	Dual-core Arm Cortex-M0+	21.0 x 51.3	5.00
ROBOTIS OpenCR 1.0	ARM Cortex-M7 STM32F746ZGT6	105 x 75	210.00
Teensy 3.2	ARM Cortex-M4 MK20DX256VLH7	17.80 x 36.26	20.00
Teensy 4.0	ARM Cortex-M7 iMXRT1062	17.80 x 36.26	25.00
Teensy 4.1	ARM Cortex-M7 iMXRT1062	17.80 x 61.66	32.00
STM32L4 Discovery kit IoT	ARM Cortex-M4 STM32L4	250 x 210	50.00
Olimex LTD STM32-E407	STM32F407ZGT6 Cortex-M4F	101.6 x 86	30.00

*Prices are given for 08.12.2023

Considering the future utilization of the MCU within the DT TB and, ultimately, in a real vehicle, the board should be as compact as possible to facilitate easy installation in spaces where availability may be limited. Furthermore, since multiple boards will be employed for various tasks and controls, cost-effectiveness is also a priority to ensure budget efficiency. Based on these criteria—size and cost—the selection process focused on identifying the smallest and most affordable MCUs. The boards selected for testing include the ESP32-DevKitC-32E, Raspberry Pi Pico RP2040, Teensy 3.2, Teensy 4.0, and Teensy 4.1. It is noted that the Teensy 3.2 shares the same dimensions as the Teensy 4.0 but features a less powerful CPU. Between these two, the Teensy 4.0 presents more potential due to its superior processing capabilities.

All these MCUs support various communication protocols compatible with micro-ROS. However, they all share one common protocol, USB, which will be utilized for speed testing. The primary distinction between the Teensy 4.1 and the Teensy 4.0 is that the former offers additional pins and includes an Ethernet module[22], which is not necessary for this phase of testing as it focuses solely on USB communication protocols. Given the similarity in CPUs between these two boards and the additional features of the Teensy 4.1 not being required, the Teensy 4.1 will be excluded from the testing.

The final list of MCUs that will be used for testing includes:

- Raspberry Pi Pico RP2040
- ESP32-DevKitC-32E
- Teensy 4.0

All the chosen MCUs are shown in Figure 3.2 for their size comparison.

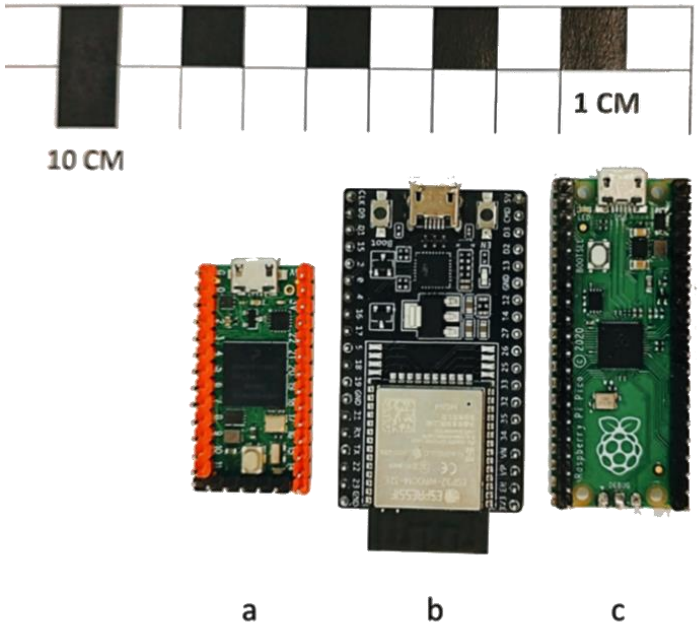


Figure 3.2 MCUs chosen for the testing: a - Teensy 4.0, b - ESP32-DevKitC-32E, c - Raspberry Pi Pico RP2040

This selection is tailored to ensure that the boards meet the compact size and cost requirements essential for their intended use in limited spaces and varied control applications within the DT framework.

3.1.1 Raspberry Pi Pico RP2040

The Raspberry Pi Pico RP2040 is a compact and powerful microcontroller designed by the Raspberry Pi Foundation. Pico is built around the RP2040, a dual-core ARM Cortex-M0+ processor clocked at up to 133 MHz, making it a versatile board suitable for various applications from hobbyist projects to more complex industrial controls. It comes equipped with 264 KB of SRAM and 2 MB of onboard Flash memory, which can be used for running programs and storing data. The board measures just 21 mm x 51.3 mm, making it highly compact and easy to incorporate into a variety of projects. The Pico provides 26 multi-function GPIO pins, including 3 analog inputs. It supports a range of interfaces such as SPI, I2C, UART, and USB. This makes it easy to connect a wide array of external devices. The board can be programmed using C/C++ SDK or the official MicroPython port, which is tailored specifically for the RP2040. This allows for flexibility

in development approaches according to the user's expertise or project requirements. The Raspberry Pi Pico can be powered via the micro-USB connection or an external source, which can be anywhere between 1.8 V and 5.5 V, providing flexible power management options. Its low cost and easy-to-use interface make it ideal for different fields, from education to industry [23].

3.1.2 ESP32-DevKitC-32E

The ESP32-DevKitC-32E is a versatile and powerful development board based on the ESP32 chipset, designed and manufactured by Espressif Systems. It is part of the ESP32 series, which is renowned for its high performance, wide range of features, and suitability for various applications. The ESP32-DevKitC-32E is equipped with the ESP32 chip, which includes a dual-core Tensilica Xtensa LX6 microprocessor that can be clocked at up to 240 MHz. It features substantial memory capabilities with 520 KB of internal SRAM and external flash memory that can be expanded up to 16 MB, depending on the module version. The board measurements are 54,4 x 27,9 mm which makes it compact, making it easy to embed in a variety of devices or projects. The board offers 34 GPIO pins with support for various functions such as ADC, DAC. Comprehensive interface support includes SPI, I2C, UART, CAN, Ethernet, IR, PWM, DAC, and ADC. Notably, the ESP32-DevKitC-32E supports Wi-Fi and Bluetooth 4.2/LE, making it ideal for IoT applications. This development board can be programmed using the ESP-IDF, Arduino IDE, and can also support scripting languages like MicroPython and JavaScript, providing flexibility for various development preferences. Power can be supplied via micro-USB or through an external supply pin. The ESP32-DevKitC-32E stands out for its powerful ESP32 chip, robust I/O capabilities, and comprehensive network connectivity options, making it a versatile platform for developers looking to explore advanced IoT, wearable, or industrial applications[24].

3.1.3 Teensy 4.0

The Teensy 4.0 is a high-performance microcontroller board designed and developed by PJRC. It stands out due to its incredible processing power, compact form factor, and the

extensive range of features it offers, making it an excellent choice for both hobbyist projects and professional applications requiring significant computational capabilities. Teensy 4.0 features an ARM Cortex-M7 processor at 600 MHz, one of the fastest microcontrollers in the market in terms of clock speed. It is equipped with 1 MB of flash memory and 512 KB of RAM, plus an additional 2 MB of flash memory accessible via the SPI bus. The board is notably small, with dimensions of just 17,8 x 36,26 mm, making it incredibly versatile for size-sensitive projects. Teensy 4.0 offers 40 digital I/O pins, including several capable of PWM, and others supporting various serial protocols. Includes support for multiple communication interfaces like SPI, I2C, UART, and CAN bus. Can act as a USB host or USB device, offering significant flexibility for interfacing with other USB devices. Features 14 high-resolution analog inputs, which is substantially more than typical microcontrollers. Contains a built-in real-time clock for time-sensitive applications even when disconnected from power. Programmable via the Arduino IDE with a Teensyduino add-on, simplifying the development process while offering advanced features for power users. Teensy 4.0 stands out in the microcontroller world due to its impressive processing speed, extensive I/O capabilities, and compact size. Its support for advanced audio processing and robust interfacing options makes it particularly valuable in projects where performance and space are critical[22].

3.2 Testing microcontrollers

3.2.1 Latency test

After selecting the MCUs, the next step is to conduct tests to determine the most suitable MCU for dedicated tasks and future project plans. The most critical criterion for the MCU selection is the response time, or latency, since in the DT development project, this parameter is crucial for effectively monitoring the power consumption of the motors and detecting problems in the motor system. Additionally, one of the project's key criteria is real-time communication, which necessitates that latency be as minimal as possible to ensure accurate and timely data transmission.

Another important factor is the data transmitting distance. Since MCUs may be installed on the DT TB close to the devices they control or monitor (which is more convenient), it is essential to consider how latency is affected by cable length. The USB cables used

can vary in length, and understanding how this impacts communication latency is crucial. Based on these considerations, the latency test will include different MCUs and cable lengths.

The latency test framework utilizes the ROS Publish-Subscribe messaging type. This test aims to measure the time it takes for an MCU to receive a message from a PC and respond to it. The test code for the MCUs was developed using the PlatformIO extension for Visual Studio Code, which allows code to be written for all boards on the same platform without the need for different integrated development environments (IDEs). Both the PC and the MCU will have one publisher and one subscriber. The PC publisher sends a message containing a PC timestamp and message ID to a topic "latency_ping" that the MCU is subscribed to. Upon receiving a message, the MCU publishes the same message to a "latency_ping" topic to which the PC is subscribed. After receiving the message, the PC sends a new message with a new timestamp and a message ID that is incremented by one. Figure 3.3 shows the latency test code scheme and Table 3.2 describes the structure of both "latency_pong" and "latency_ping" messages as they are identical. The full code of the latency test is shown in Appendix 1.

Table 3.2 Description of "latency_ping" and "latency_pong" messages structure

Parameter type	**Parameter name**
builtin_interfaces/Time	pc_stamp
int32	message_id

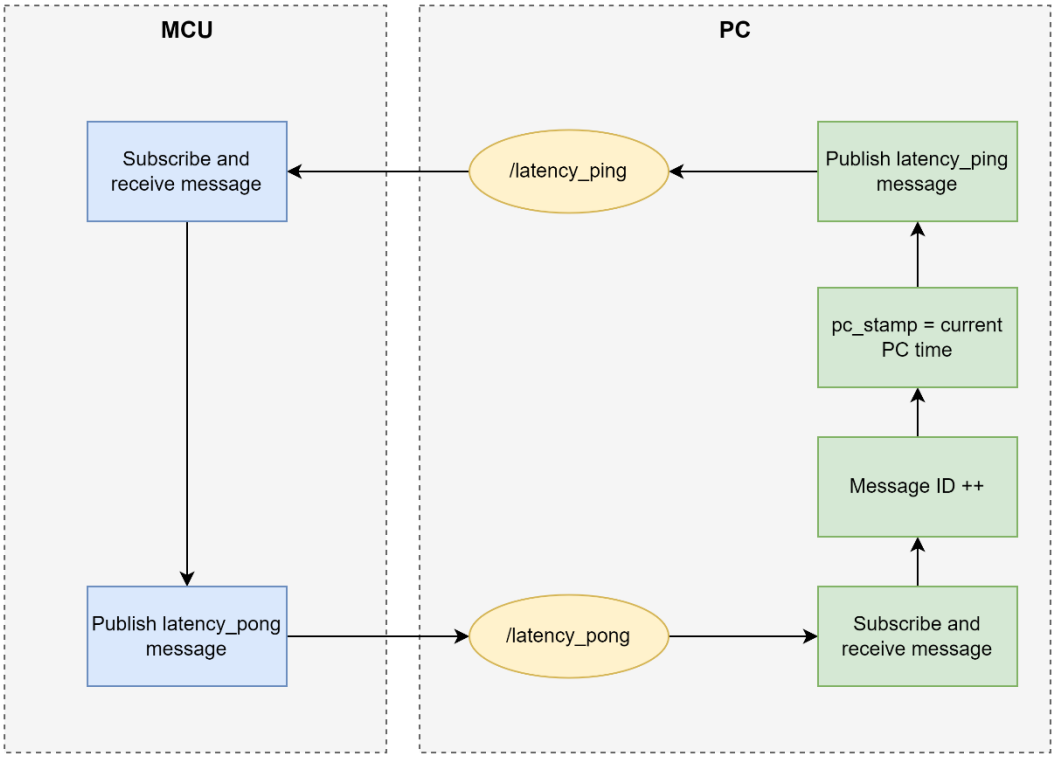


Figure 3.3 Scheme showing latency test working principle.

To avoid clock desynchronization and further complications, only the PC timestamp is used. This process is repeated until the PC and MCU have exchanged 100 000 messages. The test was conducted with all the chosen MCUs and USB cable lengths of 5 m, 2 m, 1 m, and 0.2 m. The difference between the average time for passing one message is shown in Figure 3.4.

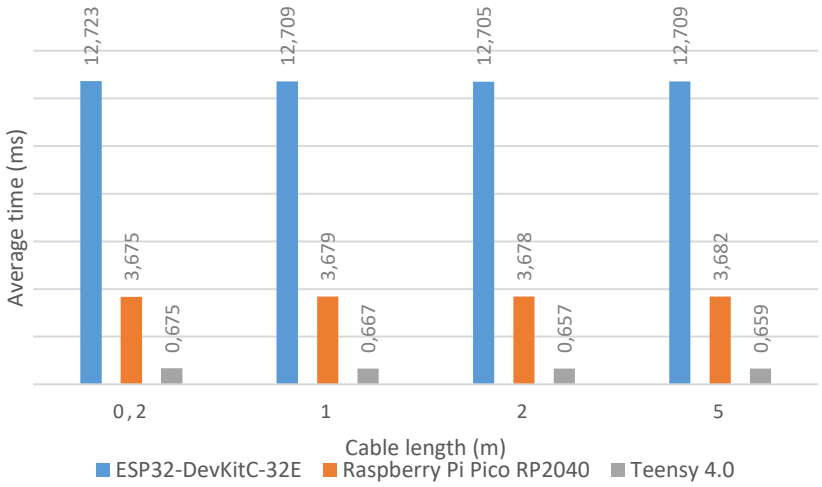


Figure 3.4 Comparison of the average time taken by different MCUs to transmit a single message

Among the tested boards, the Teensy 4.0 consistently demonstrated the shortest average message transmission time, clocking in at approximately 0,6 ms across all cable lengths. The Raspberry Pi Pico RP2040 came in second, with an average transmission time of around 3,6 ms, also consistent across varying cable lengths. Even though ESP32-DevKitC-32E has a bigger clocking speed of the CPU than Raspberry Pi Pico RP2040, it exhibited the longest transmission time, averaging about 12,7 ms regardless of the cable length. These results indicate that the Teensy 4.0 is the fastest performer in terms of message passing on average, followed by the Raspberry Pi Pico RP2040 and the ESP32-DevKitC-32E is the slowest out of three.

Additional parameters gained during the latency tests are presented in Table 3.3, along with the minimum and maximum time for one message exchange and the total amount of time it took to send 100 000 messages.

Table 3.3 Additional results of latency test for each MCU

MCU	Cable length (m)	Minimum time (ms)	Maximum time (ms)	Total time for 100 000 messages (min)
ESP32-DevKitC-32E	0.2	10.490	36.427	21.204
	1	10.439	27.737	21.182
	2	10.799	26.204	21.176
	5	10.653	114.047	21.181
Raspberry Pi Pico RP2040	0.2	2.609	104.592	6.126
	1	2.659	104.403	6.132
	2	2.740	105.455	6.130
	5	2.599	104.697	6.136
Teensy 4.0	0.2	0.316	101.891	1.125
	1	0.322	102.073	1.112
	2	0.335	101.461	1.095
	5	0.329	101.633	1.098

Based on the testing results, the Teensy 4.0 board consistently exhibited the shortest latency among the devices tested. The disparity in data transmission speeds between the boards is apparent from the average time and the total time taken to send all messages. Specifically, it took the ESP32-DevKitC-32E 21 times longer to complete the

message exchanges compared to the Teensy 4.0, and 3.5 times longer than the Raspberry Pi Pico RP2040.

Interestingly, the length of the USB cable did not significantly affect message latency across most test cases, as results remained within a consistent range regardless of cable length. However, a notable exception was observed with the ESP32-DevKitC-32E when using a 5 m cable, where the maximum time recorded deviated significantly from the expected latency pattern. This anomaly suggests that while cable length generally does not impact latency, certain configurations or specific conditions may lead to unexpected latency increases.

It was also noticed that during the latency test, all the MCUs had some distinct jumps in latency values. In Figures 3.5-3.7 are shown latency test results for each board with a cable length of 0,2 m and the average value of latency.

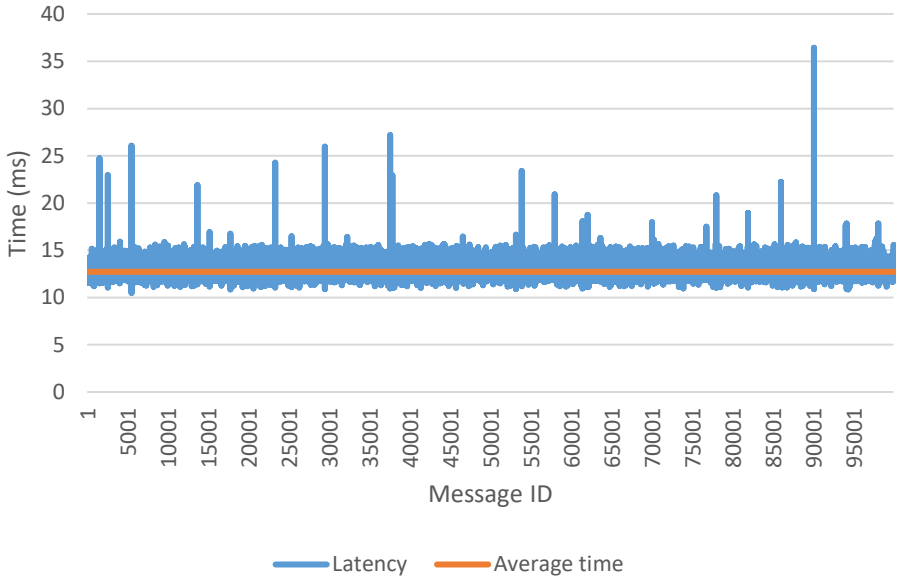


Figure 3.5 Message transmitting time for ESP32-DevKitC-32E with cable length 0,2 m

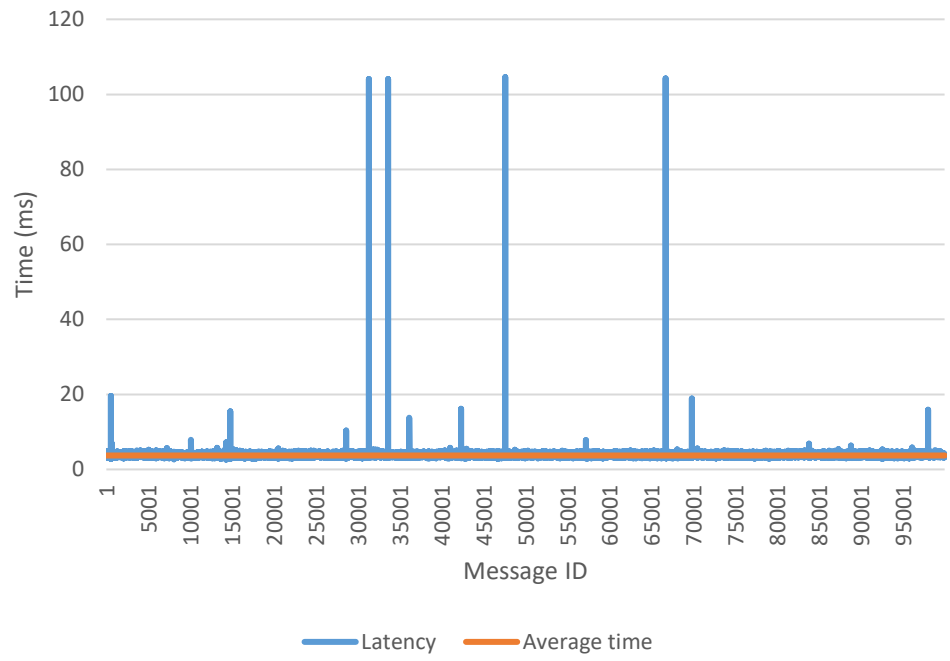


Figure 3.6 Message transmitting time for Raspberry Pi Pico RP2040 with cable length 0,2 m

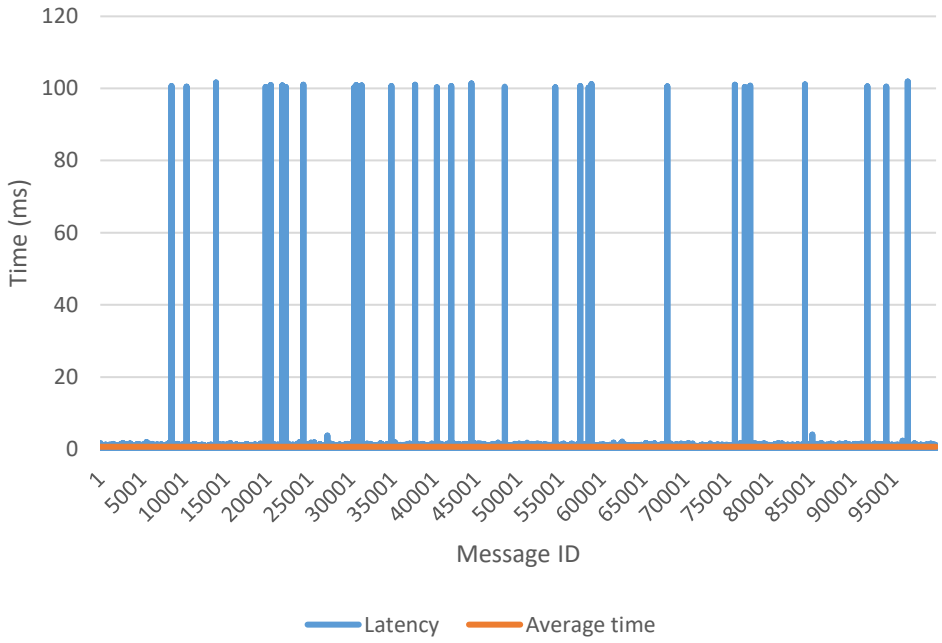


Figure 3.7 Message transmitting time for Teensy 4.0 with cable length 0,2 m

The latency test graphs reveal that some messages took significantly longer to transmit compared to the average latency time. For the ESP32-DevKitC-32E, the message transmission time remained closer to the average, with smaller spikes indicating more consistent performance.

In the case of the Raspberry Pi Pico RP2040, there were more pronounced spikes, with some exceeding 100 ms, but these occurred intermittently and didn't greatly affect the overall average latency. When values above 100 ms were excluded, the average latency time for the Raspberry Pi Pico RP2040 decreased slightly, from 3,675 ms to 3,671 ms.

Teensy 4.0 also exhibited significant spikes reaching 100 ms, but these occurred more frequently, which impacted the average latency time more heavily. After excluding values above 100 ms, the average latency time of the Teensy 4.0 decreased from 0,675 ms to 0,646 ms.

The stark difference between the average values and the latency spikes, consistently around 100 ms, suggests potential issues in data transmission, such as noise interference, code problems, or other technical challenges that warrant further investigation. Understanding the root cause of these spikes is crucial for optimizing the latency performance of these devices.

Conclusively, while the Teensy 4.0 board demonstrated the shortest latency overall, making it a superior choice for applications where speed is critical, the influence of cable length on message latency, particularly under specific conditions, warrants further investigation. Additional tests with a focus on varying environmental conditions, different types of USB cables, and perhaps even alternative communication protocols could provide deeper insights into optimizing the system’s real-time communication capabilities.

3.2.2 MCU temperature test

During the latency test, it was observed that some MCUs began to generate heat. Given that in the future, the MCU may be installed in a test bench within a closed housing to protect the MCU and conceal connections, managing the MCU's temperature becomes critical. If the housing is to be 3D printed, selecting a material capable of withstanding the temperature generated by the MCU is crucial to prevent MCU overheating or equipment damage. Consequently, additional tests will be conducted to measure each MCU's temperature during data transmission.

Furthermore, although power consumption tests are typically prioritized for devices operating on batteries or limited power supplies, in this scenario where MCUs operate

via USB, power consumption is deemed less critical. However, it may be considered for future tests to ensure the overall efficiency and sustainability of the system.

For the MCU temperature measurements, it was decided to conduct these simultaneously across all MCUs to ensure uniform test conditions. The temperatures were measured using a FLIR C3-X thermal camera, which can measure temperatures ranging from 20°C to 300°C with an accuracy of ±3°C/3% [25]. This level of accuracy is suitable for this test as the precise temperature is less critical than understanding the general temperature range.

The same latency test setup was used for temperature monitoring, but instead of stopping at 100 000 messages, the test ran continuously for two hours. Before starting the test, the ambient temperature was recorded using a different kind of temperature sensor, which showed an environmental baseline of 22°C. Figure 3.8 demonstrates the measurement process of the MCUs during the temperature test from the thermal camera.

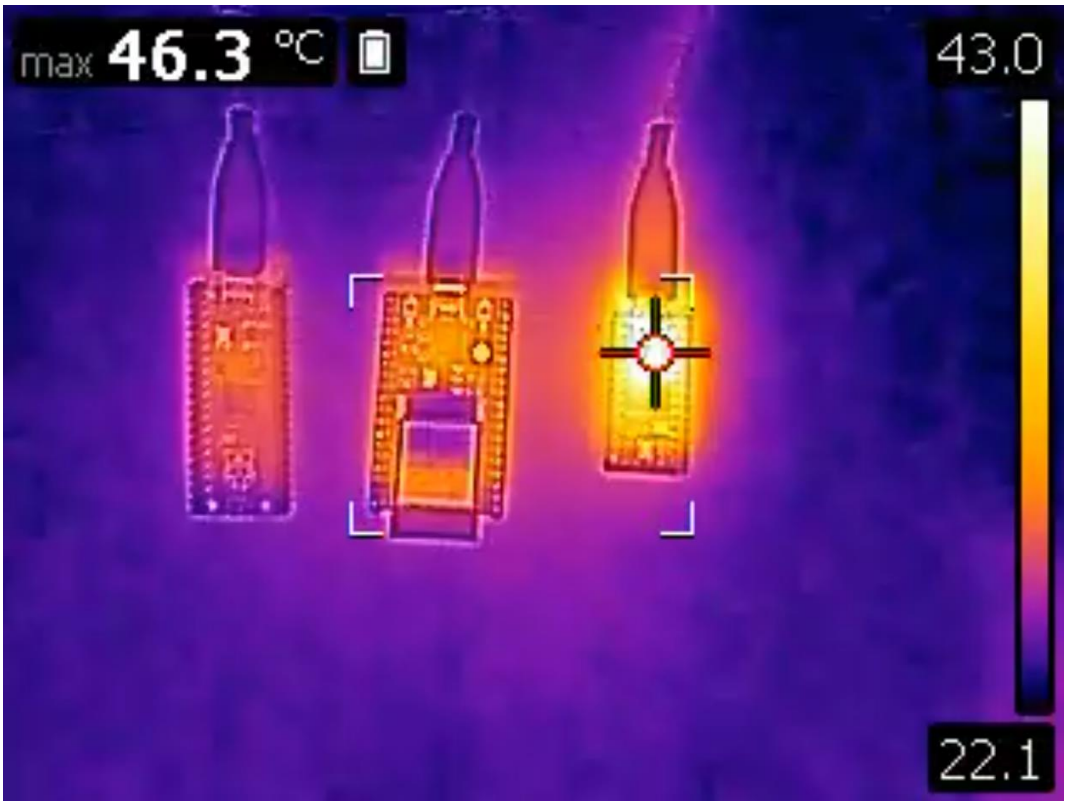


Figure 3.8 View from the thermal camera during the MCU temperature test

The results of the temperature measurements are presented in Table 3.4.

Table 3.4 Results of temperature measurement of the MCUs

MCU	Maximum temperature during test (°C)
Raspberry Pi Pico RP2040	29
ESP32-DevKitC-32E	33
Teensy 4.0	47

This test was conducted without any casing surrounding the MCUs, in an environment free from additional heating sources nearby, and with the MCUs not operating at full computational capacity. Given these conditions, the temperature of the Teensy 4.0 was notably high, signalling potential thermal management challenges in future deployments.

Considering a scenario where the MCU is enclosed within a casing and potentially surrounded by other heat-generating equipment, the Teensy 4.0 board could experience even higher temperatures. This raises significant concerns for thermal management, especially in tightly confined or poorly ventilated spaces. Therefore, selecting an appropriate material for the casing that can withstand higher temperatures without deforming is crucial.

For instance, Polylactic Acid (PLA) plastic, which is commonly used in 3D printing, begins to deform at around 60 °C [26]. This characteristic highlights the risk of using PLA for enclosures housing components like the Teensy 4.0, which may approach these temperatures under normal operational stress. Alternative materials should be considered for 3D printing that can withstand higher temperatures than PLA. Using a metal casing could also be beneficial, as it can serve as a heat sink, helping to dissipate some of the heat away from the MCU. These solutions provide better safety margins for thermal management, ensuring the stability and longevity of both the MCUs and their casings.

Other solutions to reduce the temperature of MCU could include the implementation of active cooling solutions, such as small fans or heat sinks, which are essential for maintaining safe operational temperatures. Another consideration is designing the casing with adequate ventilation openings or using thermally conductive materials to help dissipate heat more effectively. Additionally, integrating temperature sensors within the housing could provide continuous monitoring, enabling preventive measures if temperatures approach critical thresholds.

3.3 Selection of microcontroller for the current project

After conducting a series of tests, it was found that the Teensy 4.0 exhibited the shortest delay time compared to other MCUs, which is important for real-time communication. However, it had some high spikes in message passing time that should be considered in the future and reduced as much as possible. Additionally, it also tended to heat up significantly more than its counterparts. This observation underscores the necessity for careful selection of materials and the potential inclusion of extra cooling methods in projects that incorporate Teensy 4.0, especially in test bench applications.

Despite the heating issue, the Teensy 4.0's superior latency performance makes it the preferred choice for this thesis and future project endeavours. Consequently, addressing the heating concerns is crucial to fully benefit from Teensy 4.0's capabilities. Implementing effective thermal management strategies is essential to ensure the device operates within safe temperature ranges without compromising its performance.

3.4 Section summary

In this section, the foundational architecture of the DT project is outlined, along with the rationale behind implementing various MCUs. To establish a more reliable and faster connection, different types of MCUs were tested. Based on the choice of middleware, only specific types of MCUs could be assessed, and three were selected for testing.

Latency emerged as the most crucial test, and the communication distance was also evaluated to understand how latency changes with increasing distance. Interestingly, cable length did not significantly affect communication speed, but there was a considerable latency difference among the boards. The fastest board was the Teensy 4.0, followed by the Raspberry Pi Pico RP2040, and finally, the ESP32-DevKitC-32E.

A temperature test was also conducted to determine how much each board heats up during continuous operation. The Teensy 4.0 reached a temperature of 47°C, which should be carefully considered in future work to prevent overheating and potential equipment damage. The Teensy 4.0 was chosen to be used for the current work with

consideration of the potential overheating and choosing the proper materials for the mounting because of that.

4 METHODOLOGY

The primary objective of this thesis is to establish a connection between the test bench loading motors, which simulate the load experienced by a bus during travel, and the Unity virtual environment. In this virtual setting, users can select various driving conditions to test motor behaviour. The complete system is designed to automatically control the torque produced by the loading motors based on the parameters set by the user in the virtual entity and then provide feedback to the user.

For this task, the ROS2 Humble Hawksbill distribution was chosen. Although the newer stable ROS2 distribution, Iron Irwini, is available, it was not selected due to its approaching end-of-life date in November 2024. In contrast, Humble Hawksbill is supported until May 2027, offering a longer period of utility and support[27]. Although ROS2 is compatible with various OSs, Ubuntu 22.04 OS was specifically chosen for installation due to its stability and the availability of a dedicated Debian package that facilitates integration with ROS2 Humble Hawksbill. Ubuntu 22.04 is the latest long-term support version, ensuring a smooth and stable operation of ROS2.

Ubuntu 22.04 was installed on the computer as the second OS alongside with the Windows 11 OS. Another choice was to install Ubuntu on a Virtual Machine. The decision to install Ubuntu as the second OS was to avoid problems that could appear with communicating with devices through a Virtual Machine.

4.1 Controlling the loading motor

The loading motor is linked to an ABB ACS880 frequency converter, which represents the hardware control level of the system and is capable of controlling various parameters of the motor. The module receives analog input signals to determine the torque value. The Teensy 4.0 generates voltage signals ranging from 0 to 3.3 V using PWM. The analog input signal of the converter accepts a voltage value that can be adjusted as needed; thus, the voltage range was set from 0 to 3.3 V accordingly. The converter's logic is designed such that the maximum value represents 100% torque in one direction, while the minimum value signifies 100% torque in the opposite direction, with 1.65 V serving as the zero point.

For testing purposes, a smaller TB was used, as shown in Figure 4.1. It comprises an ABB ACS880 frequency converter, also employed in the main TB, and an ABB three-phase 0,75 kW motor. In the main TB, the motors used for road load simulation have a power rating of 7,5 kW. The purpose of the smaller TB is to verify that the connection works and that the motor can be controlled via Unity before conducting further tests in the laboratory using the main TB.

Figure 4.1 Small loading motor test bench used to create and test the connection between Unity and hardware

The Teensy 4.0 board is going to be subscribed to the topic where the car and road conditions will be published. The torque calculation will be done on the board itself, and after, the corresponding voltage signal will be sent to the converter, and the torque value will be sent to another topic. To write the code for the Teensy 4.0 board, PlatformIO extension was used for Visual Studio Code, and the code was written based on the micro-ROS. For computer to be able to communicate with the MCU, on the PC the micro-ROS agent needs to be launched. The agent is responsible for data exchange from the MCU to the whole ROS system.

4.1.1 Torque calculation

In [28] is described as a model development for the PDS load for ISEAUTO that considers different parameters that can influence the load, such as vehicle geometry, road surface, gravitational acceleration, etc. The forward motion of the vehicle is powered by the tractive effort, F_{TE}. This force must surpass several resistances, including rolling resistance (F_{RR}), aerodynamic drag (F_{AD}), climbing resistance force (F_{CR}), and the force needed for acceleration (if velocity isn't constant). Consequently, the total road load, F_{RL}, comprises the sum of rolling resistance, aerodynamic drag, and climbing resistance force as follows:

$$F_{RL} = F_{RR} + F_{AD} + F_{CR}. \tag{4.1}$$

The uphill resistance force caused by the road's incline relies on factors such as the vehicle's mass (m), the angle of the road in degrees (α), and the gravitational acceleration (g):

$$F\_CR = mg \cdot \sin(\alpha\pi/(180°)). \tag{4.2}$$

The rolling resistance depends on the coefficient of the rolling friction between the road and the tire C_{rf} and the normal force F_N which is represented by the vehicle's weight (m) and gravitational acceleration (g):

$$F\_RR = C\_rf \cdot mg \cdot cos(\alpha\pi/(180°)). \tag{4.3}$$

Aerodynamic drag is important, especially it becomes increasingly significant at higher velocities. It is contingent upon air density (ρ), the coefficient of drag (Cd), the frontal area of the vehicle (A), and the relative velocity of the vehicle (v) with respect to the air:

$$F\_AD = 1/2\ C\_d\ \rho A v\text{^}2. \tag{4.4}$$

To accurately compute the torque applied on a wheel, it is necessary to perform a multiplication of the road load force (F_{RL}) by the radius of the wheel (r). This calculation effectively determines the rotational force, or torque, that influences the wheel's movement.

$$T = F_{RL} \cdot r \tag{4.5}$$

The constant parameters presented in Table 4.1 are used for calculation and are taken from ISEAUTO vehicle parameters. From [29] are taken the rolling resistances for different types of roads for high-performance tires.

Table 4.1 Constant parameters used for road load torque calculations for ISEAUTO

Parameter	**Value**	**Unit**
Gravitational acceleration	9.81	m/s^2
Dry asphalt rolling friction	0.014	-
Wet asphalt rolling friction	0.017	-
Gravel rolling friction	0.02	-
Hard-packed snow rolling friction	0.016	-
Ice rolling friction	0.014	-
Air density	1.225	kg/m^3
Aerodynamic drag coefficient	0.8	-
Frontal area	2.77	m^2
Wheel diameter	0.57	m

Considering all the parameters and the fact that the ISEAUTO's maximum speed is 20 km/h, the road load torque can be calculated based on speed. The graph in Figure 4.2 illustrates the changes in torque value on a 0-degree slope across different road types. Dry asphalt and ice are represented by a single line due to their identical rolling friction coefficients. From this graph, can be observed the maximum and minimum road load torque values that can be applied to the vehicle on a 0-degree slope.

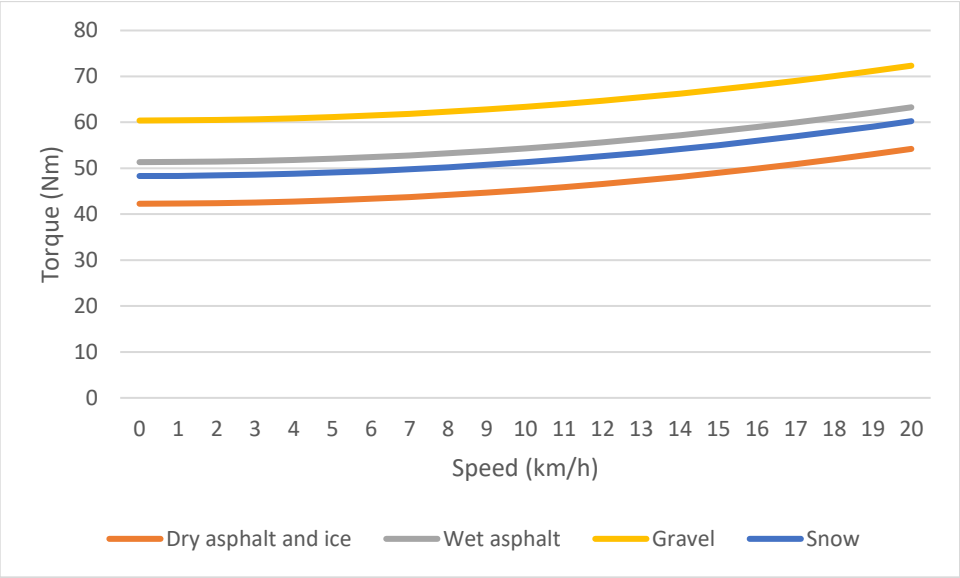


Figure 4.2 Road load torque changes depending on the speed of ISEAUTO on different types of roads with road incline of 0°

The code subscribes to the "car_param" topic where the parameters from the Unity virtual environment will be published. The parameters are sent as numbers. Each type of road surface has its identifying number based on what the proper rolling friction is selected. The torque calculation is done on an MCU that calculates all the forces and the torque value which is then converted to voltage. 3,3 V corresponds to motor nominal torque and 0 V corresponds to the negative value of motor nominal torque. To calculate the nominal torque on a smaller TB that is shown in Figure 4.1 is needed to calculate the angular velocity of the motor:

$$\omega = (2\pi \cdot n)/60, \tag{4.6}$$

where

ω - angular velocity (rad/s)

n – motor speed (RPM).

To calculate the nominal torque:

$$T = P/\omega, \tag{4.7}$$

where

P – power of the motor (W).

The motor speed at 50 Hz is 2840 RPM, and the power of the motor is 750 W. Based on those parameters, the nominal torque of the motor is equal to 2,52 Nm. This torque value is not enough to simulate the actual torque that will be applied to the car wheels during the simulation of the ride. This motor is used for verification purposes to see that the whole system works because it is controlled by the same frequency converter that is controlling loading motors on the main TB. Signals are sent to the motor, the motor starts spinning, its rotation can be controlled, and the torque values are sent back to the visualization.

The real test bench currently uses more powerful motors that have, according to [30], a rated torque of 49 Nm. The full code used on the MCU that subscribes to the "car_param" topic, calculates the road load, and sends the value to the frequency converter and to the ROS topic is shown in Appendix 2.

4.2 Hardware connection with visualization model

To seamlessly integrate the Unity virtual environment with the DT system, establishing a connection with ROS2 is essential. Thankfully, Unity offers a dedicated solution: the ROS TCP Connector package. This package serves as a vital bridge between Unity and ROS, leveraging TCP connections for communication. Installation involves deploying the ROS TCP Connector on the Unity side, while on the ROS side, the ROS TCP Endpoint is necessary. Together, these components facilitate the exchange of messages between the Unity environment and ROS. Given that Unity is predominantly coded in C#, while ROS primarily employs C++ or Python, the ROS TCP Connector handles the critical task of message serialization and deserialization. This ensures smooth communication between the two environments, overcoming language barriers seamlessly.

To ensure a successful connection on the Unity side, it is imperative to specify in Unity both the version of ROS being utilized and the IP address associated with ROS. This step is crucial for establishing connectivity. By specifying the IP address, users gain the flexibility to run the ROS network and visualize data in Unity on separate computers, enhancing scalability and versatility. Moreover, it is essential on the Unity side to define the types of messages being exchanged. This involves providing the file paths to the ROS message files containing the message structure, enabling Unity to properly

interpret and handle the data being transmitted. This step ensures seamless communication and accurate interpretation of messages between Unity and ROS, facilitating efficient collaboration and interaction within the integrated system.

A C# script was developed to enable Unity to subscribe to ROS topics. This script establishes a ROS connection and manages both a publisher and a subscriber. The publisher is responsible for transmitting user-defined parameters from the interface—such as speed, mass, incline, left-wheel road surface, and right-wheel road surface—to the "car_param" topic. The parameters are sent only when some of the parameters change their value by the user. The structure of the message sent to the "car_param" topic is shown in Table 4.2.

Table 4.2 Structure of the message sent to the "car_param" topic from the Unity

Parameter type	**Parameter name**
float32	speed
float32	mass
float32	incline
int32	surface_l
int32	surface_r

The subscriber component of the script listens to the " torque_left_wheel" and "torque_right_wheel" topics. Those topics receive torque values calculated by the MCU for the left and right wheels. The messages sent to those topics have only one float type parameter. The received torque values are then displayed to the user on the main visualization screen, providing real-time feedback on the system's operation. The whole system scheme is shown in Figure 4.3. The full code used on the Unity side to publish to "car_param" topics and subscribe to the torque topics is shown in Appendix 3.

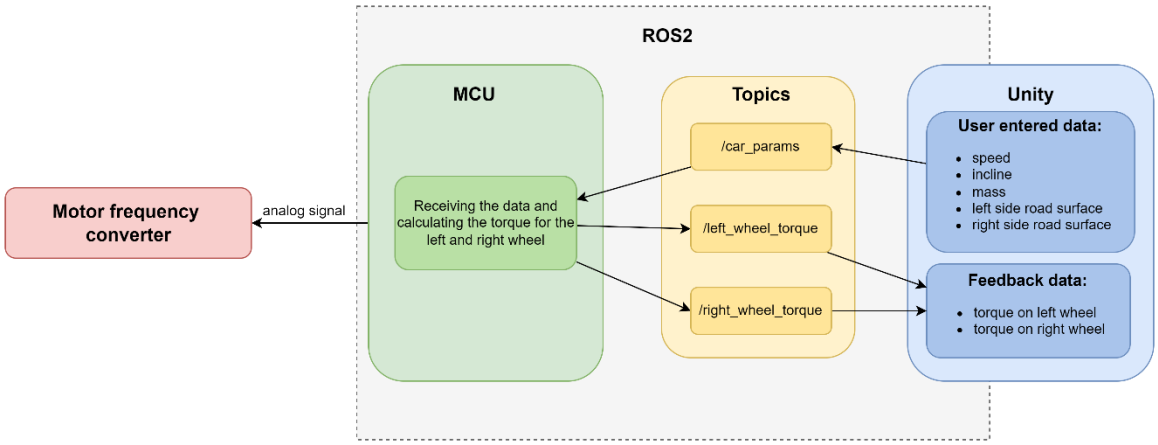


Figure 4.3 Overview scheme of created connection between virtual entity and hardware

4.3 Overview of the final solution

The results of the current thesis demonstrate the successful implementation of the ROS2 middleware and establish a working connection between the TB loading motors and the visualization software. This connection operates bidirectionally, allowing data to flow seamlessly from Unity to the MCUs and vice versa. Specifically, data gathered by the MCUs can be transmitted back to Unity, while Unity can also send information to the MCUs.

This solution is versatile and can be easily adapted to hardware that supports control through analog or digital signals. By leveraging the flexibility of the ROS2 middleware, it enhances the interoperability between different components, enabling efficient data exchange for various applications. Ultimately, this implementation lays the groundwork for scalable and robust system architectures that can accommodate a wide range of devices and control protocols.

4.4 Section summary

This section offers a comprehensive overview of the final solution, highlighting the integration between the ROS-MCU connection and the implementation of road load torque calculations. It delves into how road load torque was calculated for various road surface types, providing insight into the factors influencing performance. The ROS-Unity connection was also established, enabling seamless data exchange and real-time control. Testing demonstrated the effectiveness of this setup by controlling a real motor through the Unity UI, showcasing the system's capability to interact with the physical environment. Further refinement of these connections could involve enhancing communication protocols, streamlining data transmission, and optimizing system responsiveness, resulting in a robust, high-performance solution.

5 FUTURE WORK

In the future, the plan is to establish connections and test different communication protocols with all physical components of the TB system (including Cinergia B2C+ battery tester and emulator, the main traction motor and inverter, mechanical transmission, and all necessary sensors) and virtual component (developed models). On the software side, the aim is to connect the system not only with Unity but also with MATLAB and custom neural network code to enhance the data analysis and processing capabilities. This comprehensive integration will enable real-time monitoring and seamless data exchange across all critical components.

Moreover, the wheel-testing TB, also developed in the lab, will provide the DT with crucial information about the wheel pressure status, offering valuable insights into real-world performance. Establishing a reliable connection between the DT and this testing TB will strengthen predictive analytics and allow for more accurate diagnostics.

As more components and sensors are integrated, the system's scalability must be carefully considered. It is essential to ensure that the system architecture remains efficient as new elements are added. This will involve developing strategies to handle increasing data flows while maintaining low latency and reliable message delivery.

Regarding the MCUs, further testing is necessary to determine the root cause of occasional latency spikes that lead to longer messaging times than usual. One possible explanation is overheating, which requires additional investigation. Additionally, the performance of one-way communication, where no acknowledgment is sent confirming message receipt, should be assessed to identify potential latency issues in this setup. Adding additional cooling methods to the MCUs could also help establish a more reliable and faster connection.

Besides the USB communication testing, additional tests using different types of communication protocols can be also performed. For example, ESP32-DevKitC-32E also supports communication through Wi-Fi, which can be useful in applications where using cables will not be possible or they will be really long.

SUMMARY

DT technology holds significant potential across various fields, ranging from agriculture to aerospace and healthcare. Particularly in the automotive industry, its implementation can optimize production processes and enhance the effectiveness of industry monitoring. The DT for the propulsion drive of autonomous EV project focuses on developing and creating an unsupervised prognosis and control platform for the EV PDS, aimed at performance estimation and monitoring. This framework necessitates a robust connection between the visualization (Unity) and hardware components of the TB.

In this thesis, the overall architecture of the DT project was outlined, and from this, the choice of a suitable MCU was considered to ensure compatibility with different layers of the DT system. Various MCUs were reviewed and tested to identify the most suitable for real-time communication. Latency tests were conducted for each MCU using different cable lengths to examine how communication speed is affected by changes in cable length. The Teensy 4.0 demonstrated the best performance, maintaining an average latency of 0.6 ms regardless of cable length. In comparison, the Raspberry Pi Pico RP2040 and ESP32-DevKitC-32E showed average latencies of approximately 3.7 ms and 12.7 ms, respectively.

Additionally, a temperature test was performed to determine how much the MCUs heat up during operation. The Teensy 4.0 heated up the most, reaching up to 47°C, while the ESP32-DevKitC-32E and Raspberry Pi Pico reached maximum temperatures of 33°C and 29°C, respectively. Considering latency test values Teensy 4.0 was chosen as the final solution, with caution over temperature management.

The implementation uses the ROS2 Humble Hawksbill distribution. The Unity ROS-TCP package manages the connection between ROS and Unity, while the connection between ROS and the MCU is facilitated by micro-ROS. User inputs in the Unity UI are published to a ROS topic, which the MCU is subscribed to. Based on the received data about road conditions, the MCU calculates the road load and converts it into an analog signal sent to a frequency converter that controls the loading motor. The torque values for both wheels are also sent to ROS topics, to which Unity subscribes and displays the values to the user.

The primary objective of the thesis has been achieved—to establish a connection between the DT hardware and visualization software. Additionally, a comparison of various MCUs was conducted. The final solution still holds considerable potential for further research and development. One significant area for future work is the continued

testing of the connection and implementation of MCUs. Another area is that the DT project has many additional components and sensors that still need to be connected to the virtual entities.

KOKKUVÕTE

Digitaalse kaksiku tehnoloogial on märkimisväärne potentsiaal erinevates valdkondades, alates põllumajandusest kuni lennunduse ja tervishoiuni. Autotööstuses võib selle rakendamine optimeerida tootmisprotsesse ja tõhustada tööstuse järelevalvet. Autonoomse elektrisõiduki veoajami digitaalse kaksiku projekt keskendub elektrisõiduki veoajami süsteemi järelevalveta prognoosi- ja juhtimisplatvormi väljatöötamisele ja loomisele, mis on suunatud jõudluse hindamisele ja jälgimisele. See raamistik nõuab tugevat ühendust testplatvormi visualiseerimise (Unity) ja riistvarakomponentide vahel.

Käesolevas lõputöös toodi välja digitaalse kaksiku projekti üldine arhitektuur ning sellest lähtuvalt kaaluti sobiva mikrokontrolleri valikut, et tagada ühilduvus digitaalse kaksiku süsteemi erinevate kihtidega. Vaadati üle ja testiti erinevaid mikrokontrollereid, et leida reaalajas suhtluseks sobivaim. Iga mikrokontrolleri jaoks viidi läbi latentsus testid, kasutades erinevaid kaabli pikkusi, et uurida, kuidas kaabli pikkus mõjutab sidekiirust. Teensy 4.0 näitas parimat jõudlust, säilitades keskmise latentsuse 0,6 ms olenemata kaabli pikkusest. Võrdluseks näitasid Raspberry Pi Pico RP2040 ja ESP32-DevKitC-32E keskmised latentsusajad vastavalt ligikaudu 3,7 ms ja 12,7 ms.

Lisaks viidi läbi temperatuuritest, et teha kindlaks, kui palju mikrokontrollerid töötamise ajal kuumenevad. Teensy 4.0 kuumenes kõige rohkem, ulatudes kuni 47 °C-ni, samas kui ESP32-DevKitC-32E ja Raspberry Pi Pico saavutasid maksimumtemperatuurid vastavalt 33 °C ja 29 °C. Arvestades latentsus testi väärtustele, valiti lõplikuks lahenduseks Teensy 4.0, arvestades mikrokontrolleri temperatuuriga.

Lõplik lahendus kasutab ROS2 Humble Hawksbilli distributsiooni. Unity ROS-TCP pakett haldab ühendust ROS-i ja Unity vahel, samas kui ROS-i ja mikrokontrolleri vahelist ühendust hõlbustab mikro-ROS. Unity kasutajaliidese kasutajasisendid avaldatakse ROS-i topic-us, mida mikrokontroller jälgib. Teeolude kohta saadud andmete põhjal arvutab mikrokontroller teekoormuse ja teisendab selle analoogsignaaliks, mis saadetakse laadimismootorit juhtivale sagedusmuundurile. Mõlema ratta pöördemomendi väärtused saadetakse ka ROS-I topic-utele, mida Unity jälgib ja kuvab saadud väärtusi kasutajale.

Lõputöö esmane eesmärk oli saavutatud – luua ühendus digitaalse kaksiku riistvara ja visualiseerimistarkvara vahel. Lisaks viidi läbi erinevate mikrokontrollerite võrdlus. Lõplik lahendus sisaldab endiselt märkimisväärset potentsiaali edasiseks uurimis- ja arendustegevuseks. Üks oluline valdkond edaspidiseks tööks on mikrokontrollerite

ühendamise ja juurutamise jätkuv testimine. Teine valdkond on see, et digitaalse kaksiku projektil on palju lisakomponente ja andureid, mida tuleb veel ühendada virtuaalseteüksustega.

LIST OF REFERENCES

[1] C. Saavedra Sueldo, I. Perez Colo, M. De Paula, S. A. Villar, and G. G. Acosta, "ROS-based architecture for fast digital twin development of smart manufacturing robotized systems," *Ann Oper Res*, vol. 322, no. 1, pp. 75–99, Mar. 2023, doi: 10.1007/s10479-022-04759-4.

[2] W. Purcell and T. Neubauer, "Digital Twins in Agriculture: A State-of-the-art review," *Smart Agricultural Technology*, vol. 3. Elsevier B.V., Feb. 01, 2023. doi: 10.1016/j.atech.2022.100094.

[3] M. Alazab *et al.*, "Digital Twins for Healthcare 4.0 - Recent Advances, Architecture, and Open Challenges," *IEEE Consumer Electronics Magazine*, vol. 12, no. 6, pp. 29–37, Nov. 2023, doi: 10.1109/MCE.2022.3208986.

[4] F. Dembski, U. Wössner, M. Letzgus, M. Ruddat, and C. Yamu, "Urban digital twins for smart cities and citizens: The case study of herrenberg, germany," *Sustainability (Switzerland)*, vol. 12, no. 6, Mar. 2020, doi: 10.3390/su12062307.

[5] G. Bhatti, H. Mohan, and R. Raja Singh, "Towards the future of ssmart electric vehicles: Digital twin technology," *Renewable and Sustainable Energy Reviews*, vol. 141. Elsevier Ltd, May 01, 2021. doi: 10.1016/j.rser.2021.110801.

[6] S. Almeaibed, S. Al-Rubaye, A. Tsourdos, and N. P. Avdelidis, "Digital Twin Analysis to Promote Safety and Security in Autonomous Vehicles," *IEEE Communications Standards Magazine*, vol. 5, no. 1, pp. 40–46, Mar. 2021, doi: 10.1109/MCOMSTD.011.2100004.

[7] N. D. K. M. Eaty and P. Bagade, "Digital twin for electric vehicle battery management with incremental learning," *Expert Syst Appl*, vol. 229, Nov. 2023, doi: 10.1016/j.eswa.2023.120444.

[8] X. Liao *et al.*, "Driver Digital Twin for Online Prediction of Personalized Lane-Change Behavior," *IEEE Internet Things J*, vol. 10, no. 15, pp. 13235–13246, Aug. 2023, doi: 10.1109/JIOT.2023.3262484.

[9] A. Rassõlkin, T. Vaimann, A. Kallaste, and V. Kuts, "Digital twin for propulsion drive of autonomous electric vehicle," in *Prociding of IEEE 60th International Scientific Conference on Power and Electrical Engineering of Riga Technical University (RTUCON)*, 2019. doi: 10.1109/RTUCON48111.2019.8982326.

[10] J. Cardoso, C. Pereira, A. Aguiar, and R. Morla, "Benchmarking IoT Middleware Platforms," in *Proceding of IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2017. doi: 10.1109/WoWMoM.2017.7974339.

[11] S. Jegorov, "Middleware framework for Digital Twin entities communication Master's thesis," Master thesis, Tallinn University of Technology, Tallinn.

[12] A. Rassõlkin, R. Sell, and M. Leier, "Development case study of the first estonian self-driving car, iseauto," *Electrical, Control and Communication Engineering*, vol. 14, no. 1, pp. 81–88, Jul. 2018, doi: 10.2478/ecce-2018-0009.

[13] G. Beraldo *et al.*, "ROS-Health: An Open-Source Framework for Neurorobotics," in *Proceding of IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2018. doi: 10.1109/SIMPAR.2018.8376288.

[14] M. Quigley, B. Gerkey, and W. D. Smart, *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. O'Reilly Media, Inc., 2015.

[15] ROS Documentation, "ROS Concepts." Accessed: Mar. 20, 2024. [Online]. Available: https://docs.ros.org/en/foxy/Concepts.html

[16] M. R. Diprasetya, S. Yuwono, M. Loppenberg, and A. Schwung, "Integration of ABB Robot Manipulators and Robot Operating System for Industrial Automation," in *IEEE International Conference on Industrial Informatics (INDIN)*, Institute of Electrical and Electronics Engineers Inc., 2023. doi: 10.1109/INDIN51400.2023.10217964.

[17] M. Zhao, Q. Chen, and S. Zhang, "Research on VSLAM of UAV in Coal Mine Based on ROS," in *2021 4th International Conference on Artificial Intelligence and Big Data, ICAIBD 2021*, Institute of Electrical and Electronics Engineers Inc., May 2021, pp. 637–640. doi: 10.1109/ICAIBD51990.2021.9458969.

[18] W. N. W. Zakaria, I. A. T. Mahmood, A. U. Shamsudin, M. A. A. Rahman, and M. R. M. Tomari, "ROS-based SLAM and Path Planning for Autonomous Unmanned Surface Vehicle Navigation System," in *2022 IEEE 5th International Symposium in Robotics and Manufacturing Automation, ROMA 2022*, Institute of Electrical and Electronics Engineers Inc., 2022. doi: 10.1109/ROMA55875.2022.9915665.

[19] K. Ramu, M. Ramachandran, and M. Selvam, "Microcontroller Based Sensor Interface and Its Investigation," *Electrical and Automation Engineering*, vol. 1, no. 2, pp. 92–97, Jul. 2022, doi: 10.46632/eae/1/2/4.

[20] micro-ROS Documentation, "micro-ROS supported hardware." Accessed: Mar. 10, 2024. [Online]. Available: https://micro.ros.org/docs/overview/hardware/

[21] micro-ROS Documentation, "micro-ROS porting to ESP32." Accessed: Mar. 10, 2024. [Online]. Available: https://micro.ros.org/blog/2020/08/27/esp32/

[22] Teensy Documentation, "Teensy Technical Specification." Accessed: Mar. 10, 2024. [Online]. Available: https://www.pjrc.com/teensy/techspecs.html

[23] Raspberry Pi Documentation, "Raspberry Pi RP2040 Specification." Accessed: Apr. 09, 2024. [Online]. Available: https://www.raspberrypi.com/products/rp2040/specifications/

[24] Espressif Documentation, "ESP32-DevKitC V4 Getting Started Guide." Accessed: Mar. 10, 2024. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html#

[25] "FLIR C3-X Compact Thermal Imaging Camera." Accessed: Apr. 01, 2024. [Online]. Available: https://www.tester.co.uk/flir-c3-x-compact-thermal-imaging-camera#:~:text=It%20can%20detect%20and%20measure,3°C%2F3%25.

[26] J. Suder, Z. Bobovsky, M. Safar, J. Mlotek, M. Vocetka, and Z. Zeman, "Experimental analysis of temperature resistance of 3d printed PLA components," *MM Science Journal*, vol. 2021, no. March, pp. 4322–4327, Mar. 2021, doi: 10.17973/MMSJ.2021_03_2021004.

[27] ROS Documentation, "ROS Distributions", Accessed: Mar. 20, 2024. [Online]. Available: https://docs.ros.org/en/rolling/Releases.html

[28] A. Rassõlkin, L. Gevorkov, T. Vaimann, A. Kallaste, and R. Sell, "Calculation of the Traction Effort of ISEAUTO Self-Driving Vehicle," in *Proceeding of 25th International Workshop on Electric Drives: Optimization in Control of Electric Drives (IWED)*, 2018. doi: 10.1109/IWED.2018.8321397.

[29] "Tire friction and rolling resistance coefficients", Accessed: Apr. 09, 2024. [Online]. Available: https://hpwizard.com/tire-friction-coefficient.html

[30] ABB Documentation, “ABB 3GAA132314-ADE Specification.” Accessed: Apr. 09, 2024. [Online]. Available: https://new.abb.com/products/3GAA132314-ADE/3gaa132314-ade

APPENDICES

Appendix 1 Latency test code

```
#include <Arduino.h>
#include <micro_ros_platformio.h>

#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>

#include <std_msgs/msg/float32.h>
#include <std_msgs/msg/int32.h>
#include <psg453_interfaces/msg/latency_ping.h>
#include <psg453_interfaces/msg/latency_pong.h>


#if !defined(MICRO_ROS_TRANSPORT_ARDUINO_SERIAL)
#error This example is only avaliable for Arduino framework with serial
transport.
#endif

// Define message types
psg453_interfaces__msg__LatencyPing msg_in;
psg453_interfaces__msg__LatencyPong msg_out;

// Create ROS concepts
rcl_publisher_t publisher;
rcl_subscription_t subscriber;
rclc_executor_t executor;
rclc_support_t support;
rcl_allocator_t allocator;
rcl_node_t node;


#define RCCHECK(fn) { rcl_ret_t temp_rc = fn; if((temp_rc !=
RCL_RET_OK)){error_loop();}}
#define RCSOFTCHECK(fn) { rcl_ret_t temp_rc = fn; if((temp_rc !=
RCL_RET_OK)){}}


// Error handle loop
void error_loop() {
  while(1) {
    delay(100);
  }
}
```

```
// Function to be executed when a new message is published to latency_ping
topic
void subscription_callback(const void * msgin)
{
  // Cast received message to used type
  const psg453_interfaces__msg__LatencyPing * msg = (const
psg453_interfaces__msg__LatencyPing *)msgin;
  // Rewrite received message to another message and publish it to the
latency_pong topic
  msg_out.message_id = msg->message_id;
  msg_out.pc_stamp.sec = msg->pc_stamp.sec;
  msg_out.pc_stamp.nanosec = msg->pc_stamp.nanosec;
  RCSOFTCHECK(rcl_publish(&publisher, &msg_out, NULL));
}

void setup() {
  // Configure serial transport
  Serial.begin(230400);
  set_microros_serial_transports(Serial);
  delay(2000);

  allocator = rcl_get_default_allocator();

  // Create init_options
  RCCHECK(rclc_support_init(&support, 0, NULL, &allocator));

  // Create node
  RCCHECK(rclc_node_init_default(&node, "micro_ros_platformio_node", "",
&support));

  // Create publisher
  RCCHECK(rclc_publisher_init_default(
    &publisher,
    &node,
    ROSIDL_GET_MSG_TYPE_SUPPORT(psg453_interfaces, msg, LatencyPong),
    "latency_pong"));

  // Create subscriber
  RCCHECK(rclc_subscription_init_default(
    &subscriber,
    &node,
    ROSIDL_GET_MSG_TYPE_SUPPORT(psg453_interfaces, msg, LatencyPing),
    "latency_ping"));

  // Create executor
  RCCHECK(rclc_executor_init(&executor, &support.context, 2, &allocator));
  RCCHECK(rclc_executor_add_subscription(
```

```
    &executor,
    &subscriber,
    &msg_in,
    &subscription_callback, ON_NEW_DATA));

}

void loop() {
  // Run the executor
  RCSOFTCHECK(rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100)));
}
```

Appendix 2 Code used on MCU to calculate road load

```
#include <Arduino.h>
#include <micro_ros_platformio.h>

#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>

#include <std_msgs/msg/float32.h>
#include <std_msgs/msg/int32.h>
#include <psg453_interfaces/msg/car_param.h>



#if !defined(MICRO_ROS_TRANSPORT_ARDUINO_SERIAL)
#error This example is only avaliable for Arduino framework with serial
transport.
#endif

// Creating ROS concepts
rcl_publisher_t left_wheel_publisher;
rcl_publisher_t right_wheel_publisher;
rcl_subscription_t subscriber;
rclc_executor_t executor;
rclc_support_t support;
rcl_allocator_t allocator;
rcl_node_t node;

// Defining message types
psg453_interfaces__msg__CarParam msg_in;
std_msgs__msg__Float32 left_wheel_msg;
std_msgs__msg__Float32 right_wheel_msg;

// Definig constants used for calculation
const float g = 9.81;               // Gravitational acceleration
const float air_dens = 1.225;       // Air density
const float drag = 0.8;             // Aerodynamic drag coefficient
const float area = 2.77;            // Frontal area
const float dry_asphalt = 0.014;    // Rolling friction of dry asphalt
const float wet_asphalt = 0.017;    // Rolling friction of wet asphalt
const float gravel = 0.02;          // Rolling friction of gravel
const float snow = 0.016;           // Rolling friction of packed snow
const float ice = 0.014;            // Rolling friction of ice
const float pi = 3.1415927;
const float wheel_radius = 0.285;   // Wheel radius in meters
const float motor_torque = 49;      //Calculated motor nominal torque (Nm)
```

```
const int motor_pin = 2;            // Pin to which motor frequency
converter is connected
float surface_l;
float surface_r;


#define RCCHECK(fn) { rcl_ret_t temp_rc = fn; if((temp_rc !=
RCL_RET_OK)){error_loop();}}
#define RCSOFTCHECK(fn) { rcl_ret_t temp_rc = fn; if((temp_rc !=
RCL_RET_OK)){}}


// Error handle loop
void error_loop() {
  while(1) {
    delay(100);
  }
}

// Function to be executed when a new message is published to car_param
topic
void subscription_callback(const void * msgin)
{
  // Cast received message to used type
  const psg453_interfaces__msg__CarParam * msg = (const
psg453_interfaces__msg__CarParam *)msgin;
  // Defining values received from car_param topic
  float speed = msg->speed;
  float mass = msg->mass;
  float incline = msg->incline;

  // Define what type of surface is used for the left wheel
  switch (msg->surface_l){
    case 0:
      surface_l = dry_asphalt;
      break;

    case 1:
      surface_l = wet_asphalt;
      break;

    case 2:
      surface_l = gravel;
      break;

    case 3:
      surface_l = snow;
```

```
      break;

    case 4:
      surface_l = ice;
      break;
  }

  // Define what type of surface is used for the right wheel
  switch (msg->surface_r){
    case 0:
      surface_r = dry_asphalt;
      break;

    case 1:
      surface_r = wet_asphalt;
      break;

    case 2:
      surface_r = gravel;
      break;

    case 3:
      surface_r = snow;
      break;

    case 4:
      surface_r = ice;
      break;
  }

  // Calculation of road load
  float climb_resistance = mass*g*sin((incline*pi)/180);
  float left_rolling_resistance = surface_l*mass*g*cos((incline*pi)/180);
  float right_rolling_resistance = surface_r*mass*g*cos((incline*pi)/180);
  float aerodynamic_drag = (drag*air_dens*area*pow(speed, 2))/2;

  float left_wheel_road_load = (climb_resistance + left_rolling_resistance +
aerodynamic_drag)*wheel_radius;
  float right_wheel_road_load = (climb_resistance + right_rolling_resistance
+ aerodynamic_drag)*wheel_radius;

  // Converting torque value to voltage
  int voltage = map(left_wheel_road_load, -motor_torque, motor_torque, 0,
255);
  if (voltage > 255){
    voltage = 255;
  }
```

```
  else if (voltage < 0){
    voltage = 0;
  }
  analogWrite(motor_pin, voltage);

  // Sending the data to torque_left_wheel and torque_right_wheel topics
  left_wheel_msg.data = left_wheel_road_load;
  right_wheel_msg.data = right_wheel_road_load;

  RCSOFTCHECK(rcl_publish(&left_wheel_publisher, &left_wheel_msg, NULL));
  RCSOFTCHECK(rcl_publish(&right_wheel_publisher, &right_wheel_msg, NULL));
}

void setup() {
  // Configure serial transport
  Serial.begin(230400);
  set_microros_serial_transports(Serial);
  delay(2000);

  // Defining motor pin as an output
  pinMode(motor_pin, OUTPUT);

  allocator = rcl_get_default_allocator();

  // Create init_options
  RCCHECK(rclc_support_init(&support, 0, NULL, &allocator));

  // Create node
  RCCHECK(rclc_node_init_default(&node, "torque_calculation_node", "",
&support));

  // Create torque_left_wheel publisher
  RCCHECK(rclc_publisher_init_default(
    &left_wheel_publisher,
    &node,
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Float32),
    "torque_left_wheel"));

  // Create torque_right_wheel publisher
  RCCHECK(rclc_publisher_init_default(
    &right_wheel_publisher,
    &node,
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Float32),
    "torque_right_wheel"));

  // Create subscriber
  RCCHECK(rclc_subscription_init_default(
```

```
    &subscriber,
    &node,
    ROSIDL_GET_MSG_TYPE_SUPPORT(psg453_interfaces, msg, CarParam),
    "car_param"));

  // Create executor
  RCCHECK(rclc_executor_init(&executor, &support.context, 3, &allocator));
  RCCHECK(rclc_executor_add_subscription(
    &executor,
    &subscriber,
    &msg_in,
    &subscription_callback, ON_NEW_DATA));

}

void loop() {
  // Run the executor
  RCSOFTCHECK(rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100)));
}
```

Appendix 3 Code used on Unity to connect to ROS

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.Robotics.ROSTCPConnector;
using RosMessageTypes.Psg453Interfaces;
using Float = RosMessageTypes.Std.Float32Msg;

public class RosBridge : MonoBehaviour
{

    ROSConnection ros;
    public string topicName = "car_param";


    // Setup for variable access from GUI and SurfaceDetector
    public GameObject interfaceObject;
    private GuiManager gui;
    public GameObject carObject;
    private SurfaceDetector surfaceDetector;

    // All variables to be transmitted
    public float speed;
    public float mass;
    public float incline;
    public int surface_l;
    public int surface_r;
    public float torqueL;
    public float torqueR;

    public bool valueChanged = true;    // flag changes when speed, mass or
incline are changed from UI

    void Start()
    {
        // Start the ROS connection
        ros = ROSConnection.GetOrCreateInstance();
        // Create publisher and subscribers
        ros.RegisterPublisher<CarParamMsg>(topicName);
        ros.Subscribe<Float>("torque_left_wheel", TorqueLeftChange);
        ros.Subscribe<Float>("torque_right_wheel",
TorqueRightChange);

        gui = interfaceObject.GetComponent<GuiManager>();
        surfaceDetector = carObject.GetComponent<SurfaceDetector>();
    }
```

```
    // Rceives the data from torque_left_wheel topic and displays it in UI
    void TorqueLeftChange(Float torqueLeftMessage)
    {
        torqueL = torqueLeftMessage.data;
        Debug.Log("Torque left: " + torqueLeftMessage.data);
    }

    // Rceives the data from torque_right_wheel topic and displays it in UI
    void TorqueRightChange(Float torqueRightMessage)
    {
        torqueR = torqueRightMessage.data;
        Debug.Log("Torque right: " + torqueRightMessage.data);
    }

    void Update()
    {
        // if ANY (speed, mass, incline, 2x surface) value is changed then
assing new values and publish them to the topic
        if ((gui.valueChanged != valueChanged) ||
            (surface_l != surfaceDetector.detectedRoadTypeL) ||
            (surface_r != surfaceDetector.detectedRoadTypeR))
        {
            // Assigning new values
            speed = gui.speed;
            mass = gui.mass;
            incline = gui.incline;
            surface_l = surfaceDetector.detectedRoadTypeL;
            surface_r = surfaceDetector.detectedRoadTypeR;
            Debug.Log("speed = " + speed + "| mass = " + mass + "| incline =
" + incline + "| surfaceL = " + surface_l + "| surfaceR = " + surface_r);

            CarParamMsg carParams = new CarParamMsg(
                speed,
                mass,
                incline,
                surface_l,
                surface_r
            );
            // Send the message to car_param topic
            ros.Publish(topicName, carParam);
            valueChanged = gui.valueChanged;    //flag reset
        }

    }
}
```