Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Measurement

Offloading of route planning from autonomous vehicle to edge serves in mobile networks

Master's thesis

Bc. Adam Jáneš

Study program: Cybernetics and Robotics

Supervisor: prof. doc. Ing. Zdeněk Bečvář, Ph. D

Prague, May 2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Jáneš Adam** Personal ID number: **492356**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Measurement**

Study program: **Cybernetics and Robotics**

II. Master’s thesis details

Master’s thesis title in English:

Offloading of route planning from autonomous vehicle to edge serves in mobile networks

Master’s thesis title in Czech:

Plánování trasy pro autonomní vozidlo se zpracováním informací na hraně mobilní sítě

Guidelines:

Study principles and possibilities of computation processing for autonomous vehicles at the edge of the mobile network (multi-access edge computing, MEC) and with usage of mobile networks for vehicles to infrastructure communication. Implement a suitable existing map-based route planning algorithm for a model of the autonomous vehicle. Also, implement an existing algorithm for a detection of characteristics of moving obstacles (e.g., movement direction, speed, obstacle size, obstacle location). Furthermore, implement an existing algorithm determining an alternative route in the event of the obstacle detection.
Compare a delay and a success rate of these implemented algorithms for the case of their processing directly in the vehicle and for the case when these algorithms are processed in the MEC server. Consider, for example, quality of the communication channel and an availability of computing resources in the vehicle and in the MEC server. Test the effect of the place of processing (in the vehicle or in the MEC server) on the traveling time of the vehicle.

Bibliography / sources:

[1] P. Mach and Z. Becvar, “Mobile Edge Computing: A Survey on Architecture and Computation Offloading,” IEEE Communications Surveys & Tutorials, volume 19, no. 3, 2017.
[2] F. Borrelli, A. Bemporad, M. Fodor, and D. Hrovat, "An MPC/hybrid system approach to traction control. IEEE Trans. Control Systems Technology," volume 14, no. 3, pp. 541–552, May 2006.
[3] T. Verschoor, V. Charpentier, N. Slamnik-Krijestorac and J. Marquez-Barja, "The testing framework for Vehicular Edge Computing and Communications on the Smart Highway," IEEE Consumer Communications & Networking Conference (CCNC), 2023.
[4] J. Dolezal, Z. Becvar and T. Zeman, "Performance Evaluation of Computation Offloading from Mobile Device to the Edge of Mobile Network," IEEE Conference on Standards for Communications & Networking (IEEE CSCN 2016), 2016.

Name and workplace of master's thesis supervisor:

prof. Ing. Zdeněk Bečvář, Ph.D. Department of Telecommunications Engineering FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **06.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until:
by the end of summer semester 2024/2025

prof. Ing. Zdeněk Bečvář, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Declaration

I hereby declare that I have completed this thesis on my own and that I have only used the cited sources. I have no objection to use of this work in compliance with the act "§60 Zákon č. 121/2000 Sb." (copyright law), and with the rights connected with the copyright act including the changes in the act.

Prague, 24. May 2024

.....

Signature

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 24. května 2024

.....

Podpis

Acknowledgement

I would like to express my deepest gratitude to my thesis supervisor, prof. Ing. Zdeněk Bečvář, Ph.D., for his unwavering support, guidance, and encouragement throughout the development of this thesis. His expertise and insights were invaluable, and I feel incredibly fortunate to have had the opportunity to learn from him.

Abstract

This thesis focuses on the implementation of algorithms for autonomous driving with the ability to offload processing of these algorithms from the vehicle to the edge of the mobile network. Three algorithms for path planning, namely A*, Rapidly exploring Random Tree (RRT) and RRT* algorithms are implemented and evaluated. Furthermore, dynamic obstacle detection is investigated and implemented to the autonomous vehicle. The obstacle detection is performed based on a comparison of the map and lidar data. Tracking of the planned path is provided by Pure Pursuit algorithm and Model Predictive Control (MPC). The algorithms are tested on a model of autonomous vehicle enhanced with communication capabilities to offload processing to the edge servers via 5G mobile network. The aim of this paper is to investigate whether it is suitable to process algorithms for autonomous driving at the network edge. The conducted experiments show that path planning algorithms are suitable for offloading due to the high computational intensity. Algorithms for dynamic obstacle detection and path following are offloadable within a specified deadline, but the results are highly dependent on the current communication channel state.

Keywords:

Offloading, edge computing, 5G mobile network, path planning, obstacle detection, path following

Anotace

Tato práce se zaměřuje na implementaci algoritmů pro autonomní řízení s možností přenést zpracování těchto algoritmů z vozidla na hranu mobilní sítě. Jsou implementovány a vyhodnoceny tři algoritmy pro plánování trasy, a to algoritmy A*, Rapidly exploring Random Tree (RRT) a RRT*. Dále je zkoumána a do autonomního vozidla implementována detekce dynamických překážek. Detekce překážek se provádí na základě porovnání mapy a lidarových dat. Sledování naplánované trasy je zajištěno algoritmem Pure Pursuit a Model Predictive Control (MPC). Algoritmy jsou testovány na modelu autonomního vozidla rozšířeném o komunikaci přes 5G mobilní síť s možností zpracovávat úlohy na hraně mobilní sítě. Cílem této práce je zkoumat, zda je vhodné zpracovávat algoritmy pro autonomní řízení na hraně mobilní sítě. Uskutečněné experimenty ukazují, že algoritmy plánující trasu jsou vhodné pro offloadování z důvodu vysoké výpočetní náročnosti. Algoritmy pro detekci dynamických překážek a sledování trasy je možné offloadovat, avšak výsledky jsou velmi závislé na současném stavu komunikačního kanálu.

Klíčová slova:

Offloading, zpracování na hraně sítě, 5G mobilní síť, plánování trasy, detekce překážek, sledování trasy

Překlad názvu:

Plánování trasy pro autonomní vozidlo se zpracováním informací na hraně mobilní sítě

Table of Contents

List of Figures

List of Tables

1 Introduction

Modern cars incorporate increasing number of assistance systems and aspects of autonomous driving. Every new assistance system or aspects of the autonomous driving in the vehicle requires additional computation power, which increases the cost of car production. With the advent of the fifth-generation mobile networks, some computationally demanding tasks can be offloaded. Compared to Long Term Evolution (LTE), 5G offers promising prospects due to enhanced data rate, lower latency and lower power consumption [1]. It clears the way for offloading the computing tasks with strict time requirements such as autonomous driving.

Autonomous driving algorithms that can be offloaded are route planning algorithms [2], camera processing [3], or control algorithms. The advantage of offloading is the higher computational power that the Multi-access Edge Computing (MEC) server provides. The biggest performance increase is in image processing, since the offloading unit usually does not have a graphics card. If the image is offloaded to a server with a graphics card, the computation speed is increased several times. Offloading requires a connection with a sufficient data rate and, in the case of control algorithms, a low latency. If these conditions are not met, algorithms that require fast response miss the deadline. Another challenge of offloading algorithms that require fast response is the mobility management, since during a handover the data is forwarded to another MEC server. Data forwarding creates additional delays that can lead to missing time requirements [4].

The aim of this work is to implement path planning, dynamic obstacle detection and control algorithms with offloading capability on MEC server. The first part of the experiments compares the computation of path planning using the A* algorithm [5], Rapidly exploring Random Tree (RRT) and RRT*[6] algorithms locally on the vehicle and on the MEC server. The second part of the experiments compares the computation of the dynamic obstacle detection algorithm. The third part of the experiments investigates the offloading of Pure Pursuit algorithm and Model Predictive Control (MPC) [7] following a planned path. The path planning represents applications with soft deadlines. The control algorithms symbolize tasks with hard deadlines, since deadline violations in control algorithms lead to vehicle collisions.

This thesis explores the possibilities of offloading over the 5G mobile network and evaluates which autonomous driving tasks are suitable for offloading. The thesis focuses on meeting the deadlines of each task. In addition, it tests whether the MEC server is able to achieve more accurate results with the same deadlines by varying the parameters of each algorithm, such as the search depth in MPC or the reconnection distance in the RRT* algorithm.

The next section summarizes the recent works and results achieved in this field. The subsequent section of this thesis explains offloading and the advantages of the MEC server. The fourth section provides a description of the model of autonomous vehicle used in the thesis, covering mounted components and implemented algorithms. The practical section consists of experiments with a vehicle driving autonomously. Experiments compare algorithms for path planning, detecting dynamic obstacles and following the planned path computed locally and on the MEC server with regard to latency and precision of the algorithm, such as higher depth of the MPC algorithm or larger area for spotting dynamic obstacles. The concluding chapter summarizes whether it is possible to realize partial or full offloading for autonomous driving utilizing 5G mobile network.

2 Related works

Autonomous driving is mainly composed of path planning to the desired destination and following the planned path. Following the planned path should involve algorithms for monitoring the surrounding environment to detect other traffic participants and road signs.

Path planning can be formulated as the shortest path problem. The problem is based on a graph with edges between the vertices. Every edge has a weight representing the cost from one vertex to the second vertex. The cost is determined by Euclidian distance and vertices represent cells in the grid map. The most famous algorithms for solving this problem are Dijkstra's algorithm, Floyd-Warshall algorithm, and Bellman-Ford algorithm [8], [9]. Each mentioned algorithm searches for the shortest path with some difference. The first difference is that Dijkstra's algorithm does not support negative cost of edges as opposed to Floyd-Warshall algorithm and Bellman-Ford algorithm. Furthermore, Floyd-Warshall algorithm searches for the shortest path between every pair of vertices. For this reason, the time complexity of Floyd-Warshall algorithm is higher than the time complexity of Dijkstra algorithm and Bellman-Ford algorithm. The time complexity of these algorithm is shown in Table 1.

Table 1: Time complexity of the shortest path algorithms [8], [9]

Algorithm	Dijkstra's	Floyd-Warshall	Bellman-Ford
Time complexity	$O(m a n \log n)$	$O(nm)$	$O(n^3)$

One of the implemented algorithms in this work is A*. The A* algorithm is based on Dijkstra's algorithm. However, the A* algorithm has a concrete goal that should be reached. Furthermore, the A* algorithm uses priority queue and heuristic, which helps to prioritize direction to the goal position [5].

The significant disadvantage of the mentioned algorithms is that the algorithms do not take into account the dimension of the vehicle and steering restrictions which can lead to collision. The algorithm meeting this requirement is the RRT algorithm. However, the RRT algorithm searches for a path, which is not necessarily the shortest one. Moreover, the algorithm is not complete, which means that the existing path is not found every time. On the other hand, with the increasing run time of the algorithm the probability of path discovery increases as well. The asymptotically optimal variant of the RRT algorithm is the RRT* algorithm. The difference between RRT and RRT* algorithms is that the RRT* algorithm allows rewiring existing explored poses in proximity, which leads to shortening the resulting path. Another approach to path planning is the Probabilistic Roadmaps algorithm. This algorithm, unlike the RRT and the RRT* algorithm, first creates a big amount of poses and then the poses are connected based on distance. This approach is not implemented because it is not able to guarantee steering restrictions [6].

The main goal of the control algorithm is to ensure the vehicle follows the planned path. The unsophisticated way to track the path is to determine points on the path and compute the angle difference between the vehicle and the selected point. The Pure Pursuit algorithm [10] computes the point in defined look ahead distance in front of the vehicle. The chase of look ahead point improves behavior of path following because the vehicle reacts to turns in advance. In [10], the principles and properties of the algorithm are described. If the vehicle is moving at a higher velocity, the look ahead distance should be higher. For this reason, the vehicle cut corners on a curved path. The paper [11] describes approach to minimize path tracking error in turns. In the paper, the authors reduce cutting corners, which improves tracking performance. On the other hand, the experiments are not performed on several different environments.

The more complex approach than the Pure Pursuit algorithm is MPC. In contrast to the Pure Pursuit algorithm, MPC takes dynamic obstacles spotted in the surrounding environment into account. In [12], the authors propose an improved MPC for path tracking. The proposed controller forecasts future vehicle state to generate optimal steering. MPC can solve optimization problems not only for path tracking. In [13], the MPC and l_1-optimal hybrid controller is used to minimize slipping on challenging road surfaces, such as polished ice. The authors of the paper [12] improved MPC controller which can forecast future vehicle states. On the other hand, the controller has only been tested in simulation and it has not been tested how it works on real hardware.

Computation offloading represents a method for processing highly demanding tasks on the server. This approach allows users to extend the battery lifetime of the user device and run sophisticated applications even though the user device does not meet the hardware requirements. In this scenario, the running of the application requires a stable connection to the server. In [14], the authors describe possible mobile network architectures for offloading. Furthermore, the authors depict mobility management when MEC server is exploited by the user and scenarios when it is suitable to compute the task locally or to offload it. In [15], the authors formulate an optimization problem for offloading to minimize the time between the start of the offloading and receiving the result from the server. The paper [16] introduces implementation of an offloading framework and evaluates it using an Augmented Reality app. The result of the paper is that highly demanding tasks are recommended for offloading with regard to latency of computation and power consumption. On the other hand, during the testing of the Augmented Reality app, no mobile connection was used, but the results were provided using a Wi-Fi network. Moreover, the calculation did not have a deadline that should have been met.

The mentioned related works describe algorithms for autonomous driving computed locally on the vehicle or the authors introduce offloading in simulation or for application without deadlines. Therefore, the aim of this work is to implement and test algorithms for autonomous driving on real hardware with the ability to offload demanding tasks over 5G mobile network.

3 System model and background

The first subsection briefly describes the arrangement of the 5G network. The second subsection focuses on the MEC server and its characteristics. This is followed by a subsection about the communication channel and its impact on offloading.

3.1 5G network

The 5G network is divided into two parts. The first segment is the Next Generation Radio Access Network (NG-RAN) responsible for establishing connections with User Equipment (UE). NG-RAN involves base stations (gNB) providing connectivity to UEs. The UEs are represented by devices such as tablets, laptops, smartphones, industrial machines, autonomous vehicles, or IoT devices. Base stations arrange wireless communication with UEs and transfer UEs' demands to the 5G Core network (5GC), which is the second part of the 5G structure. The 5GC manages the cellular network and data routing, employing functions such as Quality of Service (QoS), policy rules, and mobility management [17].

The UE communicates with the MEC server through the 5G network. The network adjusts channel modulation to maximize data rate and minimize error rate. The channel is influenced by its bandwidth, signal level, noise level and interference level from other users in the network. Furthermore, the communication channel is affected by user movement and demands covered by Quality of Service.

In the frame of this thesis, the autonomous vehicle communicates with the base station for the purpose of offloading. In the practical part of the thesis, we test whether the offloading implementation is serviceable for stationary and moving vehicle in terms of autonomous driving.

3.2 MEC server

In recent years, the number of highly demanding applications that users want to run on their devices running on battery power has rapidly increased; for instance, augmented reality apps. This type of application implies that the battery lifetime is shortened. To mitigate this issue, users can offload their applications to a server, where the server handles the demanding tasks. At the beginning, the approach was realized as a centralized cloud (CC) accessible via the Internet. The centralized cloud allows users to use its high performance. On the contrary, user demands on the CC impose huge additional load on the radio and backhaul of the mobile network. Furthermore, results from the CC are received with high latency. Therefore, this approach is not suitable for real-time applications [14].

An alternative approach introduces smaller distributed units situated in proximity to the UEs. Its disadvantage is lower computation performance compared to the CC. On the other hand, the backhaul infrastructure is not loaded so much by user demands. Moreover, computation demands on these units are delivered with lower latency. In this approach, users can complete highly demanding tasks by offloading using less energy consumption and similar latency. Some implementations of this attitude, for instance cloudlets were not widely adopted due to the choice of technologies.

MEC is a concept of edge computing integrated into mobile network architecture. Additionally, many companies, such as IBM, Vodafone, Nokia, Huawei, and Intel, support this approach. The leading advantages of MEC servers are their proximity to users, integration into cellular network infrastructure, and sufficient performance. For this reason, the usage of MEC servers is a convenient approach to real-time applications.

MEC server, providing higher performance than the autonomous vehicle, is capable of planning routes with higher precision. Consequently, the MEC server realizes more possible steering in one iteration

and executes a higher number of iterations at the same time. Moreover, the vehicle conserves power consumption by executing only essential and simplified algorithms in scenarios when the connection to the MEC server is lost.

The most suitable use case for offloading is applications with highly demanding computational tasks and the short length of input parameters. The benefit of this application type is transfer speed to the MEC server owing to data size. Furthermore, the server can utilize its performance to compute the task, preferably using multi-core performance. In this situation, the task is completed faster remotely using the MEC server than locally on the user's device.

Another suitable application for offloading consists of many independent tasks. In this scenario, the user exploits maximal MEC server performance. Independent tasks are convenient for edge computing because the device can offload the whole or only part of the computation problem. Moreover, the tasks can be computed in parallel to further reduce latency.

However, the utilization of MEC servers presents several challenges. In the optimistic scenario, the user battery is discharged only by transmitting and receiving data, not by computing the task. However, the MEC server cannot process all user demands. For this reason, the script optimizing offloading should be included. The script decides which tasks is offloaded based on QoS and available resources.

Another challenge is resource allocation. The MEC server allocates resources for every user in terms of computation performance and time-frequency slots. It implies that as the number of users connected to a single MEC server increases, the number of resources available to each user decreases.

Figure 1 illustrates the difference between MEC and server computing utilizing CC. Using CC for server computation is inconvenient for real-time applications because the data goes through RAN, the Core Network, and the Internet to the Data Center, where it is processed. MEC servers, positioned between RAN and the Core Network, reduce latency due to their proximity to users and minimize additional network load.

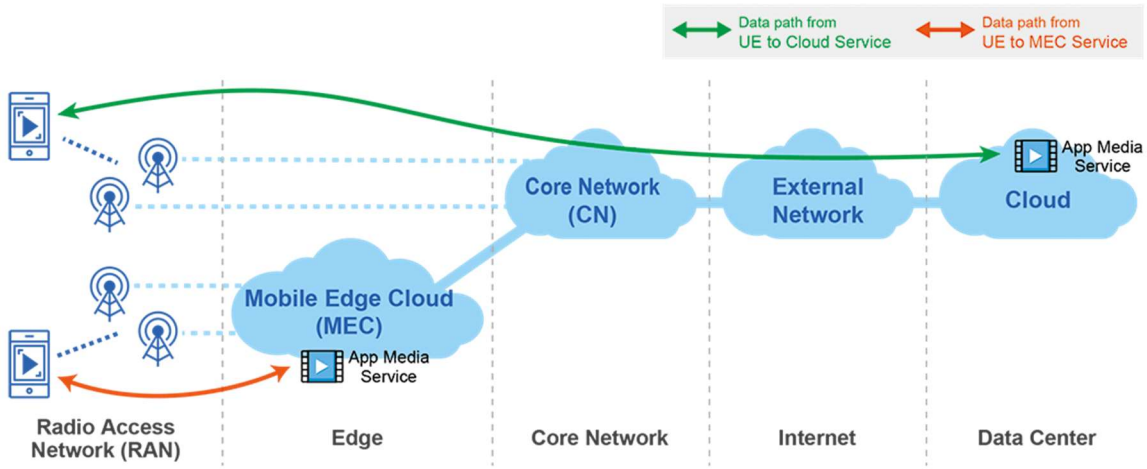


Figure 1: Comparison between Mobile Edge Computing and Cloud Computing [18]

4 Autonomous Vehicle

This section describes components mounted on the vehicle. At the beginning, all components are listed and the connections between them are depicted. Afterwards, each component is described with regard to its functionality. The components were mounted on the vehicle with my colleague Bc. Jan Daňek as part of the subject "Project" which is a subject intended for the preparation of the diploma thesis. The communication between the components and the implementation of the algorithms for autonomous driving is done individually within this thesis.

4.1 Architecture of autonomous vehicle

The vehicle used for testing offloading and autonomous driving is a commercial remote-controlled car modified for our purposes. The base vehicle is a chassis from Traxxas TRX-6 Mercedes G 63 6x6. The vehicle has six wheels allowing it to carry heavy loads. The computation part of the vehicle is represented by Raspberry Pi 4 model B, Arduino Uno, power bank Viking Smartech II and Rplidar A1M8 mounted on the chassis. The primary purpose of the vehicle is to serve as a mobile computing unit. However, its onboard computing capabilities are not sufficient for handling advanced algorithms for autonomous driving locally. Figure 2 depicts the connection among the components situated on the vehicle.

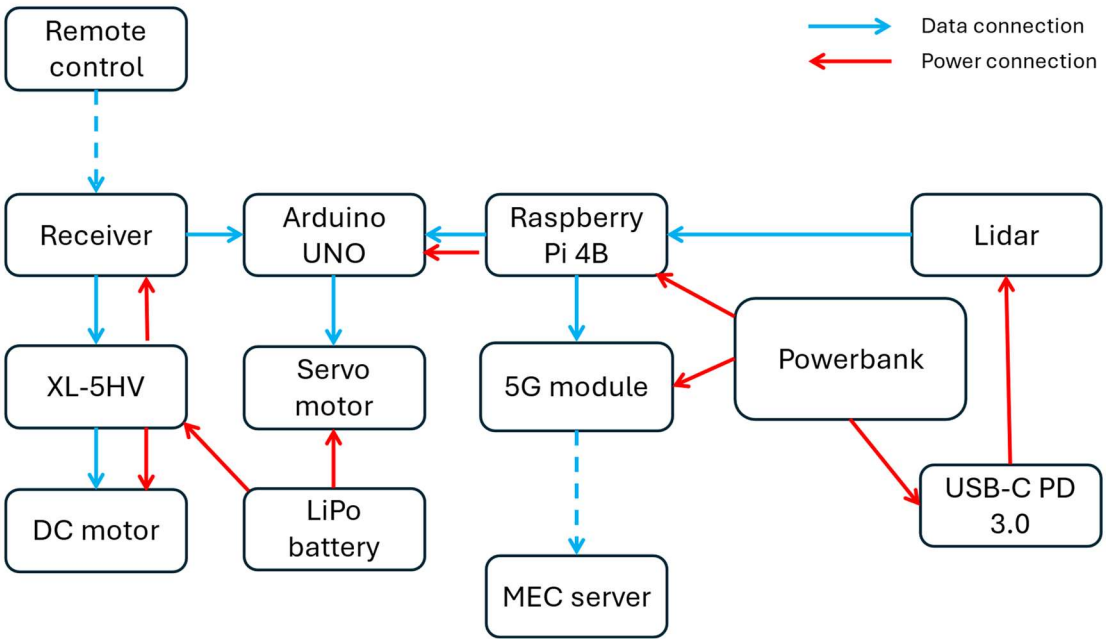


Figure 2: Data and power connection diagram

Figure 3 depicts the arrangement of component situated on the vehicle. The Arduino is positioned at the front of the car due to its proximity to the antenna, the servo motor and the regulator XL-5HV. The lidar is placed in an elevated position not to deem other components as obstacles. The position is achieved by 3D printed construction. Below the lidar, the 5G module is mounted in its 3D printed holder. The location of the powerbank is in the middle of the vehicle because it is the heaviest applied component. The remaining place is a convenient location for the Raspberry Pi 4B. 3D printed parts conceal cables between components and facilitate their attachment.

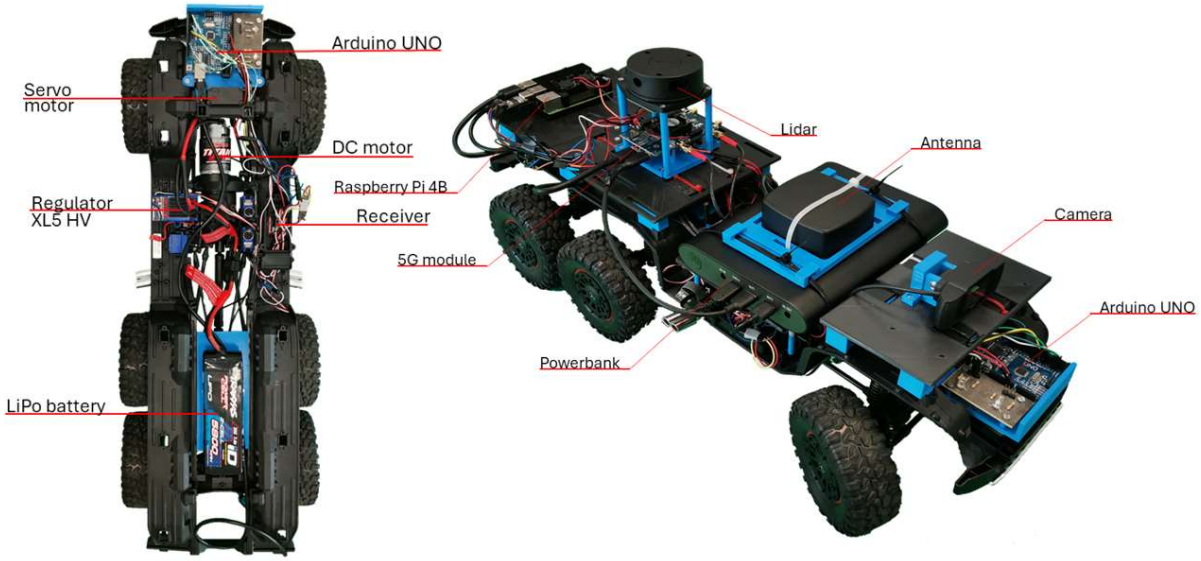


Figure 3: Vehicle components

4.2 Traxxas TRX-6 Mercedes G 63 6x6

The vehicle is a remote-controlled model on a scale of 1:10. The original remote control transmits a signal received by the antenna on the car. Consequently, the signal, in the form of Pulse Width Modulation (PWM), is sent to the original speed regulator XL-5HV or directly to the servo motor. The servo motor controls vehicle turning. The regulator converts the PWM signal to voltage and current suitable for DC motors. The DC motor located in the front of the vehicle controls movement forward and backward. During autonomous driving, the remote control is used as an emergency brake.

The regulator allows the car to switch between several modes and two maximum speeds. Regulator XL-5HV operates in Crawl Mode because it facilitates motor control. The principal benefit of this mode is that motor control is independent of the current state of the DC motor. In this mode, moving the lever forward or backward on the remote control causes car movement forward or backward. Returning the lever to its initial position activates the car's braking. In other modes, pulling the lever while the car is moving forward causes the vehicle to brake instead of reverse. The described behavior in other modes makes autonomous driving more difficult [19].

4.3 Arduino UNO

The board Arduino UNO is a microcontroller based on the ATmega328P. The primary reason for selecting the Arduino UNO is the large community and its own integrated development environment. In addition, the Arduino boards allow the use of analog PWM, which conserves significant computation power compared to digital PWM.

The Arduino UNO is utilized as a bridge between signals from remote control and scripts responsible for autonomous driving and motors. When the autonomous mode is deactivated, the user can control the vehicle through the remote control, with the antenna receiving the signal and transmitting it to the Arduino UNO. In this scenario, the Arduino UNO duplicates and forwards the signal to the motors. When the car drives autonomously, the Arduino receives steering and throttle commands from the script and adjusts the analog PWM signals accordingly. At the same time, the Arduino board monitors

the signal from the antenna. If the signal does not correspond to the remote control in the initial position, the board stops all analog PWM signals and deactivates the autonomous mode.

4.4 Power supply

Two power supplies are on the vehicle. The original Li-Po battery powers the regulator XL-5HV, the DC motor, and the servo motors. The second power supply is the Viking Smartech II powerbank. The principal purpose of the powerbank is to power all components for autonomous driving, including the Raspberry Pi 4 model B, RPLIDAR and Arduino UNO. The powerbank contains 2x USB-A output, 1x USB-C output, 1x DC output and 1x DC input port. The powerbank has high battery capacity and supports the Quick Charge 3.0 protocol. Additionally, the USB-A ports supply 5V/3A, which is essential for the trouble-free running of the Raspberry Pi 4B. Another crucial feature is a DC connector with output up to 94 W, capable of supplying additional performance to components such as an Intel NUC if needed. For these reasons, the powerbank Viking Smartech II was selected to power the computation part of the vehicle [20].

4.5 Rplidar A1M8

The RPLIDAR A1M8 is an affordable computer peripheral that provides information about the surrounding environment using a laser beam. The acquired information is transferred to the host via USB-A. Slamtec company provides an application for the Windows system, which establishes a connection and tests the device.

Figure 4 depicts the Slamtec RoboStudio application environment. Unfortunately, the application is not compatible with the Linux operating system. Nevertheless, communication between Robot Operating System (ROS) and RPLIDAR is straightforward due to the availability of a preconfigured ROS node [21]. The node receives data from RPLIDAR and converts it to ROS messages. Therefore, data visualization is not too complicated in the ROS environment. Figure 5 illustrates the lidar data in RVIZ, a visualization tool for ROS.

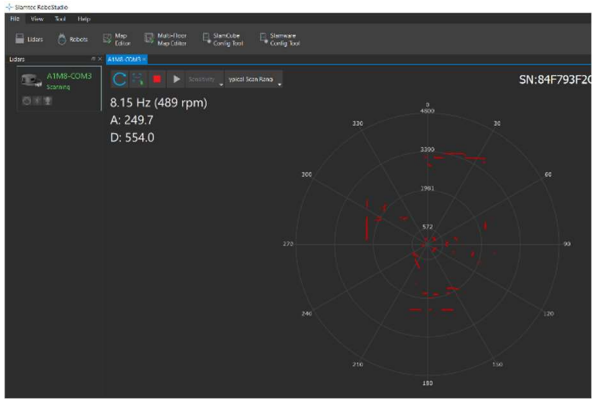


Figure 4: Slamtec RoboStudio

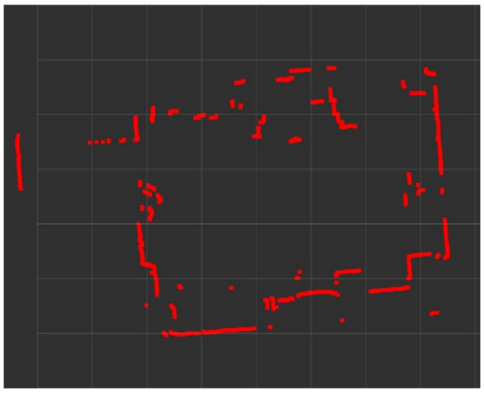


Figure 5: Lidar data in RVIZ

The head of the lidar turns around 360 degrees and recognizes obstacles up to a distance of 12 meters. The lidar captures more than 8000 samples per second of the surrounding environment and sends acquired data each turn. The initial speed of the lidar head is 5.5 Hz. For this reason, the angle between the two samples is very small. However, latency is more critical than the additional precision of data samples during autonomous driving. As a result, increasing the turning frequency leads to a higher frame rate of published data at the cost of resolution. The increase in the turning frequency is achieved by increasing the input voltage to the lidar. For this reason, the lidar is powered by the powerbank via the power delivery USB-C module. The module allows the user to adjust the voltage. The output voltage

of the USB-C module was set to 9 V, the highest possible option that still satisfies the maximum input voltage limit of 10 V [22].

4.6 Raspberry Pi 4 Model B

The Raspberry Pi 4B is a small desktop computer that can deal with two 4K displays. Model B features 8GB of RAM, 4x USB-A connectors and a gigabit Ethernet port. The crucial benefits of the Raspberry Pi 4B include high performance relative to its small size and support of the Linux operation system. The operation system was changed to Lubuntu, a lightweight version of the popular Ubuntu distribution. As a result, Lubuntu benefits from a similarly large community as Ubuntu.

The Raspberry Pi 4B serves as the core of autonomous driving. It collects and processes data from the lidar, adjusts the lidar's rotation speed and controls the motors by sending commands to the Arduino. The CPU of the Raspberry Pi was overclocked to achieve higher performance for navigation and motor control. To maintain a sufficient temperature, the Raspberry Pi 4B is cooled down by a cooling shield.

4.7 5G module

Communication with the server is provided by Quectel RM500Q-GL 5G HAT [23]. The module is connected to the Raspberry Pi 4B via a USB-A port. The board is also connected to the base station of the cellular network using 5G. The USB-C connector is utilized to power the board, as it can supply up to 15W, compared to the 2.5W provided through the USB-A connector. The higher power supply obtained from the powerbank results in a stronger signal and a more stable connection.

The module features 2 SIM slots, USB-A and USB-C ports, and four antenna connectors. In our setup, we use two antennas in a square case because the original antennas obstruct laser beams from the lidar. Additionally, the square antennas have more suitable emitting characteristics for our purpose since the base station's antennas are situated in the ceiling. The board supports Windows, Linux and Android operating systems. Moreover, the chip is configurable using AT commands via terminal.

5 Implemented algorithms

This section describes the Lubuntu operating system used on the Raspberry Pi 4B. Afterwards, the ROS environment and the nodes developed in frame of the thesis for autonomous driving are described. The nodes are divided into several groups according to their functionality. At the end of this section, the connection used in this thesis between the MEC server and the vehicle is described. All implemented algorithms are available on the git repository [24]. For a more detailed description of the git repository or the installation of the operating system using Ubuntu server and overclocking the CPU, see the appendix.

5.1 Lubuntu

Lubuntu is a complete operating system based on Ubuntu. It is known for being a lightweight version of Ubuntu, requiring less computational power. Additionally, plenty of applications and software are supported due to the Linux kernel. Furthermore, the Ubuntu community is one of the largest communities within the Linux distribution ecosystem. For these reasons, Lubuntu is chosen as an operating system to run on the Raspberry Pi 4B.

5.2 Robot Operating System (ROS)

Fundamentally, ROS consists of software libraries and tools designed to facilitate the development of robot applications. Each ROS distribution is tailored to a specific Linux version of the operating system. In our case, we utilize Lubuntu 20.04 LTS, thus necessitating the use of the ROS Noetic distribution designed for it. We have not chosen the latest version of the Lubuntu distribution because it should mean utilizing ROS 2 which has plenty of unsuitable solutions and errors. For this reason, we use ROS in this thesis.

The primary advantage of ROS and ROS2 is their ability to divide applications into smaller components and facilitate communication between them. They support two types of communication: messages and services. Every script creates a new node in the ROS environment. Nodes communicate with each other using topics, which are deified by name and the type of messages that occur in them. Each node can act as a publisher, subscriber or both in a topic. Communication in topic can be reliable (TCP type) or unreliable (UDP type). The type of communication is chosen during establishing a subscription to the topic. Last but not least, the ROS environment supports graphical tools, facilitating visualisation of acquired data from sensors.

5.3 ROS nodes interconnection

This subsection depicts the connection between particular nodes in the ROS environment, see Figure 6 . At the beginning, the acquired data from sensors are published into the ROS environment. The Laser Scan Matcher Node publishes odometry data from received lidar data. Slam Toolbox node creates a map from received odometry and lidar data. Position Publisher node simplifies transformations between frames and publishes information about the current vehicle position. A* and RRT nodes plan a path from the current position to the desired destination marked in RVIZ. The Detect Obstacles node and the Emergency Brake node detect dynamic obstacles from lidar data in the surrounding environment. According to the created path and spotted obstacles, the Pure Pursuit and MPC nodes track the path and publish commands for motors. Control Motor node converts ROS messages to string commands for Arduino UNO. Gateway and Control nodes communicate with server for the purpose of offloading and remote control of ROS environment on the vehicle. All nodes except Lidar node, Laser Scan Matcher node and Slam Toolbox node are implemented as part of this thesis. The server has a

similar node structure, but the nodes that publish sensor data, create the map, and control the engines are not included in the server.

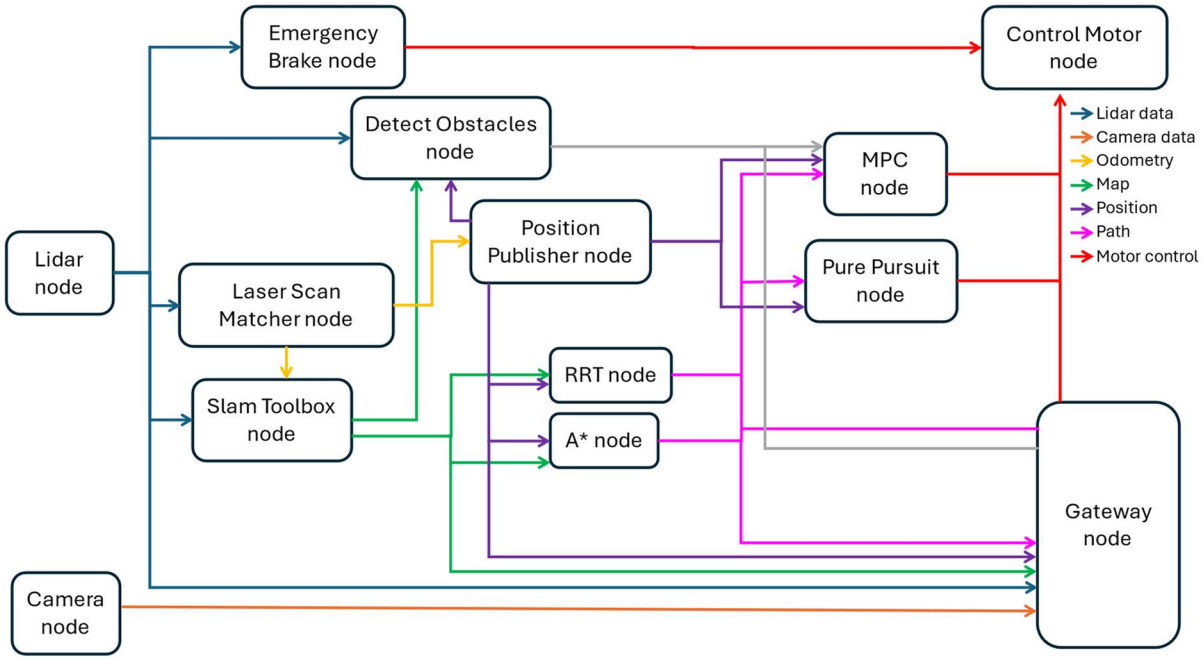


Figure 6: Linkage between the ROS nodes

5.4 ROS nodes

This section describes the individual nodes in the ROS environment and the algorithms that run in them.

5.4.1 Sensor nodes

This subsection consists of lidar and camera nodes. Both nodes transform data streams from the sensors into ROS messages and publish acquired data about the surrounding environment.

Information about the surrounding environment is acquired solely by the lidar. The Lidar node converts distance measurements into ROS messages. This node providing conversion is prepared by the producer on their GitHub. ROS messages published to the *scan* topic contain information about data length, the angle range or the time of publication.

The camera node takes data from the camera sensor and converts it into ROS messages. The ROS messages allow us to visualize the camera stream in RVIZ. Additionally, ROS messages provide a convenient format for subsequent processing.

5.4.2 Laser Scan Matcher node

The Laser Scan Matcher node subscribes to the *scan* topic to receive lidar data. From this data, the Laser Scan Matcher node generates data similar to odometry data. The odometry data are required for using the Slam Toolbox node, which creates maps.

Alternatively, odometry data can be reached by an IMU sensor as well. However, our experiments show that using the IMU sensor results in less accurate odometry data compared to those generated by the Laser Scan Matcher node. Odometry data contain information about position and rotation. For this

reason, the IMU sensor has to integrate measured values twice, which brings inaccuracy to the final values.

The data containing each transformation between particular frames are published to the *tf* topic. The frames determine several coordinate systems, such as lidar frame, odometry frame, vehicle frame or map frame.

5.4.3 Slam Toolbox node

The Slam Toolbox node is responsible for creating a map. The principal purpose of the node is to store information about the environment explored up to now. Information about the environment is stored in the form of a grid, see Figure 7. The black cells of the grid represent obstacles on the map, the white ones denote free space, and the grey area depicts unexplored space.

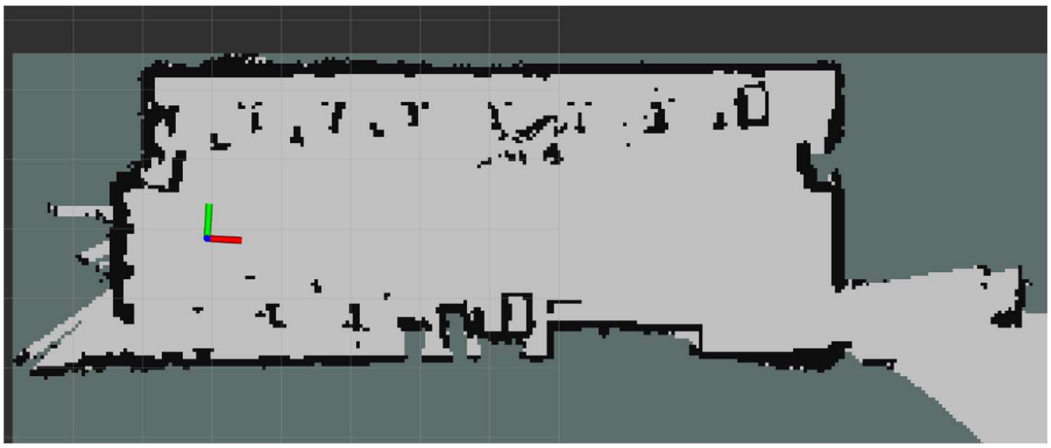


Figure 7: Occupancy grid representing map in RVIZ

Throughout time, the color of the boxes may change, for instance, when an obstacle disappears or changes its position. Map data are published to the *map* topic containing information about the size of the grid, the time of publication, resolution, map origin, etc. The grid, represented by an array, includes integers from -1 to 100. -1 indicates an unexplored cell. A number between 0 and 100 represents the probability of an obstacle's presence in that cell. After each optimization loop of the map, laser beams change the likelihood of obstacles in the cells according to the following equation:

$$probability = \frac{\#\ laser\ beams\ reflected\ in\ the\ cell}{\#\ reflected\ beams + \#\ beams\ that\ passed\ through\ the\ cell} \qquad (1)$$

Additionally, the node determines the vehicle's position. During each optimization loop, the node minimizes the distance between acquired laser data and obstacles depicted in the grid to reach the best estimate of the vehicle's position. Slam Toolbox uses a graph-based approach to creating and optimizing a pose graph [25]. For this reason, the Slam Toolbox node is one of the most power-consuming parts of autonomous driving.

We conducted tests with several simultaneous localization and mapping (SLAM) packages. In our experiments, the Slam Toolbox outperformed Google Cartographer and Hector SLAM. Hector SLAM had trouble expanding the map. Probably, it happened because Hector SLAM is based on a local approach. Google Cartographer accomplishes more accurate calculations than Hector SLAM. However, higher resolution causes higher computation load, which the Raspberry Pi 4B cannot offer. The precision and resolution of the map were decreased to lower the computation load. As we reduced computation demands, Google Cartographer had problems with localization on the straight corridors, which are a substantial part of the testing area. The Slam Toolbox provides a decent outcome without

noticeable issues in the same space with regard to computation demand. For this reason, the Slam Toolbox node is implemented for the best accuracy-to-demand ratio. Similar results are achieved in [26].

5.4.4 Position Publisher node

The Position Publisher node subscribes to *tf* and *map* topics to publish information about vehicle position in a more convenient way for other nodes. Its primary objective is to subscribe to position information, convert it and publish it with minimum latency. The current position of the vehicle can be accessed through two methods.

The first method subscribes to the *slam_toolbox/pose* topic published by the Slam Toolbox node. It is an effortless method to obtain the position, but this method suffers from significant latency, making it unsuitable for our requirements. The alternative method subscribes to frame transformations to reach a relative position between the map origin and the vehicle. Transformations are published much more frequently than *slam_toolbox/pose* messages, allowing us to achieve the same precision with considerably lower latency. To minimize latency further, we subscribe to transformations unreliably, which may result in occasional message loss. On the other hand, transformations are published in tens or hundreds of hertz. It follows that the unreliability of the communication does not cause inconvenience. The unreliable communication can be set by the following line:

```
ros::Subscriber subTf = n.subscribe("tf", 1, &Class::tfCb, this, ros::TransportHints().unreliable().tcpNoDelay());
```

This type of communication is not provided for Python scripts. For this reason, the majority of scripts are written in C++ language. Additionally, C++ is faster than Python in most scenarios [27].

In the transformation callback, we utilize transform listener [28] to compute the relative position and rotation between the vehicle and the map origin. The obtained position's X and Y axes are recorded since the vehicle operates in a 2D environment. The rotation between the frames is converted from quaternion notation to Euler angle notation. We focus solely on the yaw angle due to the 2D nature of the world. Lidar position is also determined using the transform listener. In the case of lidar, it is not necessary to compute lidar rotation because the yaw angle of the lidar is the same as the vehicle angle. Eventually, we append information about the map's origin and the car's position in the grid.

5.4.5 Path planning nodes

This section delineates algorithms for path planning. The A* represents a well-suited algorithm for local processing due to its relatively low computational demands. The RRT algorithm based on the Ackemann model and K-dimensional (K-d) tree structure is more computationally intensive and it is convenient to offload the algorithm on the MEC server.

Path planning according to the explored area is a fundamental aspect of autonomous driving. Firstly, the user specifies the finishing position for a path. The position is given by the graphical tool RVIZ. RVIZ visualizes ROS messages and allows users to observe the vehicle's current position with laser data in the grid. Additionally, RVIZ publishes ROS messages based on user interactions. To be concrete, the user picks the tool "2D Pose Estimate" or "2D Nav Goal" and clicks on the map. Subsequently, RVIZ publishes the clicked position to the *move_base_simple/goal* or *initialpose* topic. The *initialpose* topic is dedicated to adding new checkpoints to the path. The *slam_toolbox/goal* topic closes the path and initiates path computation.

5.4.5.1 A algorithm*

A* is a fundamental algorithm for path planning working with the grid provided by the Slam Toolbox node. The advantage of A* is that the algorithm finds the shortest path with permitted moves in the grid. Therefore, the algorithm is complete and optimal.

In the algorithm, the vehicle is considered a moving point without physical dimensions. It follows that the collision detection problem should have been solved previously. As a result, the obstacles on the map are dilated at the beginning of the algorithm. Therefore, the resulting path avoids collisions. This approach is convenient for round robots because they have an equal distance between the coordinate system's origin situated in the center and edge of the robot. The Traxxas vehicle has the shape of a rectangle. Furthermore, the coordinate system's origin of the vehicle is not located in the vehicle's center but on the front axle. The position of the origin facilitates the car's maneuvering because the rear axles follow the front axle. However, the origin's location noticeably increases the radius of the obstacle dilation. Therefore, the dilation is higher than necessary, and the path is not found in some scenarios. Because of this, the algorithm is not complete in this situation. On the other hand, the resulting paths do not cause collisions.

In Figure 8, we see dilated obstacles on the map preventing collisions. Figure 9 depicts excessive dilation related to the car's shape compared to Turtle Bot.

Figure 8: Dilatated obstacles in map

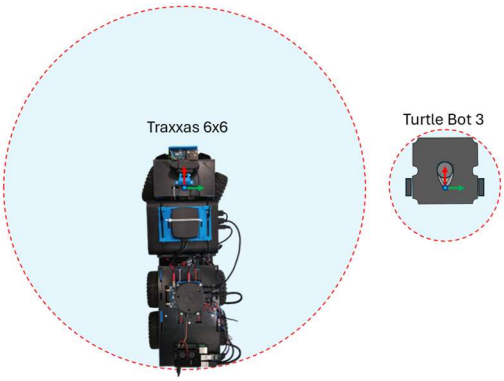


Figure 9: Illustration of the Traxxas and TurtleBot dilatation

The route planned by the A* algorithm is designed primarily for robots without steering constraints. The ideal example of the mentioned approach is TurtleBot because it can rotate on the spot. Moreover, the robot is round, a felicitous feature for map dilatation.

The Traxxas vehicle has constraints in the form of a turning radius. Therefore, the A* algorithm should be adjusted for 4-wheel and 6-wheel vehicles. The original A* algorithm does not take into account limited turning; consequently, the output path contains bends, which are challenging for a 6-wheel car, as depicted in Figure 10. The majority of the problem occurrence is in the proximity of the checkpoints.

To adapt the algorithm for 6-wheel cars, we introduce penalization for inconvenient moves. The algorithm's policy used for determining the shortest path is based on Euclidean distance. The added penalty for changing direction should minimize the number of turns. The second penalty is applied when the direction changes are close behind. This penalty should prevent exceeding the maximum turning radius. These modifications help to design a suitable path for a 6-wheel vehicle between 2 poses. The difference between the original and modified A* algorithm represents Figure 10 and Figure 11. The tuning of the A* algorithm is performed in several different environments, but the experiments are conducted in a single space to achieve constant conditions.

Figure 10: Original A algorithm*

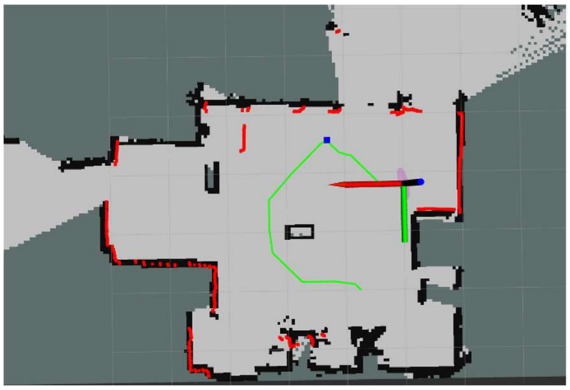


Figure 11: Modified A algorithm*

5.4.5.2 Rapidly exploring Random Tree (RRT) algorithm

The more suitable algorithm than A* for planning paths designed for 4-wheel and 6-wheel vehicles is the RRT algorithm. The advantage of the RRT algorithm is its ability to work with a specific vehicle model, resulting in a path directly tailored to the car.

The RRT algorithm starts by defining the start and end positions. Before the beginning, the maximum computation time is set, which influences the resulting path. Although the RRT algorithm is not complete, with increasing number of iterations, the probability of finding a path also increases. The RRT* algorithm enhances the RRT algorithm by introducing the possibility to refine the resulting path.

At the beginning of each iteration, the script generates a random pose on the map and searches for the nearest pose in the tree built from the start position. The Euclidean distance is computed in three dimensions: x coordinate, y coordinate and yaw angle. The nearest pose in the tree to the random pose serves as the starting point for the next move. Subsequently, the model simulates several maneuvers from the starting point to new positions. The new position with minimal distance from the random pose becomes a new leaf of the tree. At the end of the iteration, the script verifies whether the new leaf is closer to the goal position than the given limit. If the distance is lower than the limit, the RRT algorithm stops and publishes the obtained path.

The RRT* algorithm has a slightly different iteration flow than the RRT algorithm. After adding a new leaf to the tree, the RRT* algorithm inspects nodes in close proximity. If the path length to the nearby poses is shorter through the newly added leaf than the original path, the pose is reconnected to the new leaf. Consequently, the RRT* algorithm shortens the resulting path.

In Figure 12, the path generated by the RRT algorithm is depicted, while Figure 13 illustrates the resulting path computed by the RRT* algorithm. The following subsections describe a utilized vehicle model and the data structure of the tree.

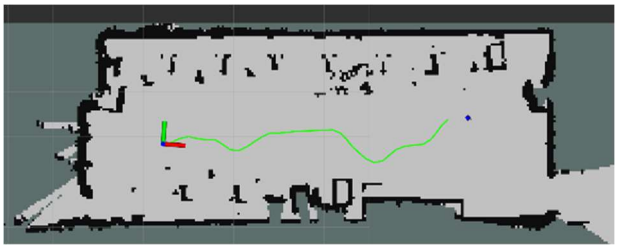


Figure 12: RRT algorithm

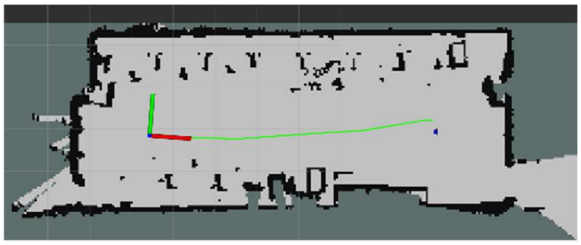


Figure 13: RRT algorithm*

5.4.5.3 Ackermann model

The simplified Ackerman model is utilized to calculate vehicle maneuvers, particularly suitable for 4-wheel cars. We implement the Ackermann model to determine the subsequent position given a specific steering angle.

The subsequent position of the vehicle is computed in the function *ackemanMove*. The initial segment of the code computes a simple maneuver without steering. If the steering is non-zero, the script calculates the circle radius. The circle imagines a path for the rear axle. Afterward, the circle center is determined. At the end, we compute the subsequent vehicle position using the circle center and the center angle. The center angle is equal to the steering angle [29].

The move is divided into several sections to verify whether the maneuver is collision-free. The collision-free position does not intersect with obstacles on the map. The algorithm inspects the grid cells spread around the edge of the vehicle. Figure 14 illustrates the computation of the subsequent vehicle position.

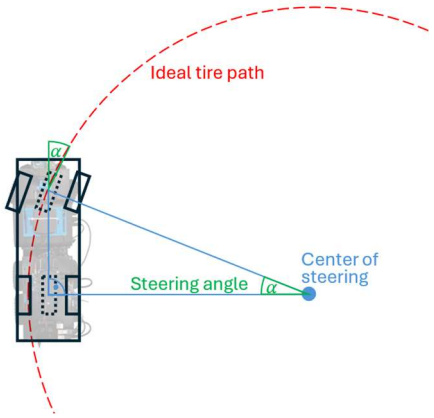


Figure 14: Illustration of the computation of the Ackermann steering

5.4.5.4 K-Dimensional (K-D) tree

K-D tree is a data structure convenient for handling multi-dimensional data, commonly used for searching for the nearest node within the structure. The K-D tree follows a tree-based structure with a root node serving as the starting point. Each node, except for the root, has one parent node and a maximum of two children nodes, making it a binary tree. If the node has no successor, it is a leaf of the tree. In our case, the K-D tree searches for the nearest point in the tree to the random point within the RRT algorithm. While both linear search and K-D tree methods have a worst-case time complexity of O(n) for nearest neighbor search, the K-D tree offers the advantage of pruning, resulting in an average time complexity close to (2^K + log(n)), where K is number of dimensions and n is number of nodes. The searching for the nearest neighbor using the K-D tree is efficient when 2^K is significantly smaller than the number of nodes. In our scenario, the algorithm works with three dimensions, and the tree includes thousands of nodes, which means that the condition is satisfied [30].

We implemented our K-D tree because the libraries for C++ and Python only support storing multi-dimensional data. Our application requires adding information about the path to tree nodes, which is not permitted by libraries.

The K-D tree consists of nodes represented by struct *Node*. The node struct contains information about left and right children and the node's position. The position in the format of x coordinate, y coordinate, and yaw angle is stored in the *Pose* struct among the information about the previous position on the way and the length of the path up to now. The fundamental operations of the tree are inserting nodes, searching nodes in the tree, and finding the nearest point to the reference. Our implementation incorporates the rebalancing and clearing functions. The remaining functions support the mentioned fundamental operations and the correct behavior of additional functions, such as reconnecting nodes.

The insert function adds a new node to the tree according to the rules. Nodes are compared based on one of the dimensions at a given depth. The left child node has a smaller or equal value of the given dimension than its parent. The given dimension of the right child node is higher than its parent's value. Dimensions, determining characteristic parameters for a given level, change in the following order: x coordinate, y coordinate, yaw angle, x, y, yaw, etc. First of all, the function inspects the root node. If the root node is empty, it is a primitive occurrence. In this scenario, we insert the first node into the tree. Otherwise, the function compares the root node and the new node according to the given dimension. If the child node exists, the new node is compared with the child node. Progress continues until an empty child slot is found. Figure 15 draws the progress of the insert function. On the side, the characteristic dimension is marked.

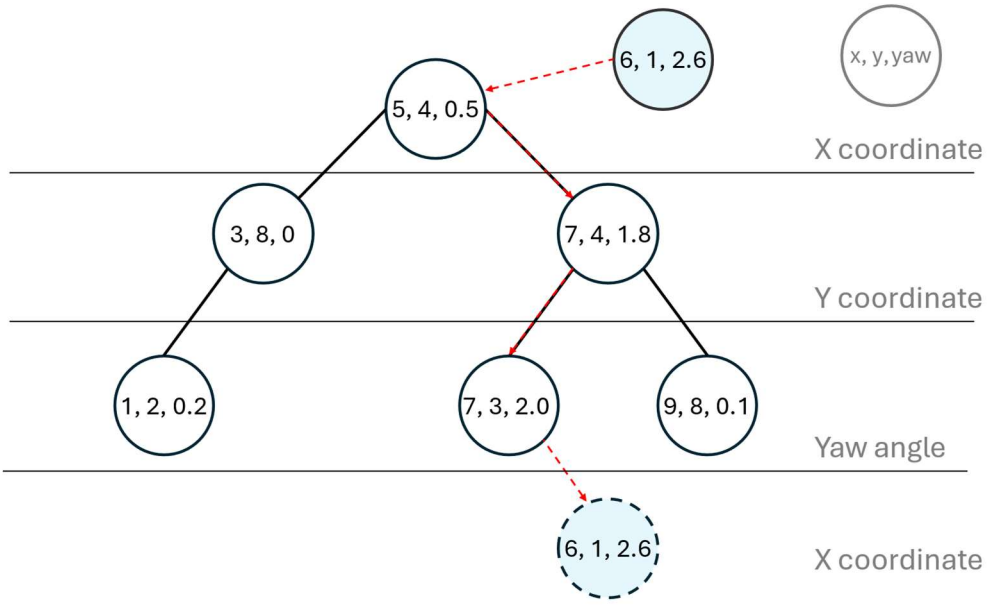


Figure 15: Progress of the insert function

The search function is similar to the insert function. Firstly, we compare the reference node and the parent node. If the nodes are not equal, we compare the reference node and the right or left child node based on the given dimension. The function continues until we find the searched node in the tree or reach the leaf of the tree.

The nearest neighbor search function is based on a depth-first search algorithm [31]. Firstly, we dive deep into the tree similarly to the insert function and record the minimum distance from a reference point. Afterward, the algorithm returns and inspects unexplored branches. If the minimal possible distance of the branch area to the random point is higher than the recorded minimum distance, the

branch is pruned. This pruning process ensures efficiency in the nearest neighbor search problem. Figure 16 represents the branch area and its distance to the reference point. In Figure 17, the progress of nearest neighbor search function is depicted.

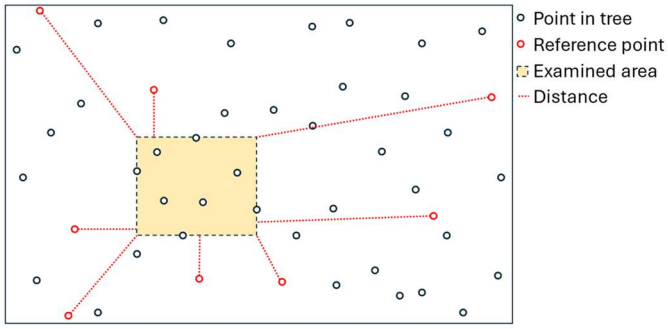


Figure 16: Branch area and distances to reference points

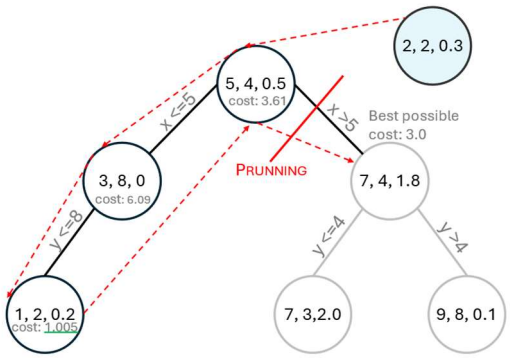


Figure 17: Progress of the nearest neighbor search function

Our implementation does not contain a function for deleting nodes because there is no need for node deletion in our scenario. Every time the RRT algorithm starts, a new K-D tree is built. The clearing function is utilized to clear the whole tree. The rebalancing function is called when the difference between the minimum and maximum depth of the tree exceeds the given limit. The rebalancing function erases the original structure and builds a new balanced tree. Rebalancing minimizes the depth of the tree and improves structure efficiency because the efficiency of the balanced tree is much higher than that of the imbalanced one. The imbalanced state happens when the starting position is close to the map edge. In this situation, the number of left children nodes is much higher than the number of right children nodes or vice versa.

The reconnection function is applied to shorten the resulting path. When a new position is added to the tree, the function finds nodes in the tree close to the inserted node. The found nodes are inspected if the path through the inserted node to the found node is shorter than the existing one. The function also takes into account the model constraints.

5.4.6 Emergency Brake node

The node for emergency braking subscribes only to the *scan* topic to receive information about surrounding obstacles, focusing on imminent collision avoidance. The size of the vehicle is known; therefore, we can distinguish the laser points located in front of the car and define a limit, ensuring collision-free movement. The algorithm compares the distance between the distinguished points and the given limit that ensures collision-free movement. If any point is closer than the limit, the car stops until the obstacle disappears. If the space in front of the vehicle is not cleared in 15 seconds, the car returns to the map origin.

5.4.7 Detect Obstacles node

The node detects dynamic obstacles using the *map* and *scan* topics. The lidar data are transformed into the map frame. Consequently, the subscribed grid and transformed data are combined into one grid by using the XOR logical operator. The result of the merge is a map representing only dynamic obstacles that are not depicted in the original grid.

In the next step, the algorithm distinguishes particular obstacles. One obstacle is represented by cells in the grid, which have a common side or vertex. The connection of the cells is reached by implemented

the "two-pass" algorithm [32]. We have optimized the algorithm to inspect each cell only once. For this reason, the time complexity is O(n). The algorithm inspects the cell and its surrounding cells. If the cell represents a dynamic obstacle, the algorithm gives the cell a number symbolizing the ID of the obstacle. If the cell adjoins cell representing different obstacle, the obstacles are merged by the "union-find" approach [33]. The benefit of the "union-find" approach is that we do not copy cells, only reconnect the head of the first obstacle to the head of the second obstacle.

The previous step assigned cells to obstacles. The next part of the algorithm computes characteristic obstacle features, such as position and radius. Afterward, the algorithm associates the obstacles with historical data, minimizing the sum of distances between current and previous positions. The connection to historical data improves the prediction and monitoring of dynamic obstacles.

The node publishes an array containing information about dynamic obstacles to the *dyn_obs* topic. Each new dynamic obstacle increases the length of the array, with information about the obstacle inserted into a new cell. If the spotted obstacle is connected to historical data, the features are updated in the proper cell of the array. If information about the obstacle is not updated, the node publishes an empty cell. For this reason, the length of the message during driving is increasing. The array is cleared when no dynamic obstacles are spotted.

5.4.8 Path following nodes

This section delineates nodes following the planned path. The Pure Pursuit node represents a low computationally demanding task convenient for local processing. The MPC node represents a more sophisticated algorithm because it takes into account the reference path and dynamic obstacles published by the Detect Obstacles node. Furthermore, the MPC node utilizes the Ackermann model for path tracking which increases computation load. The Control Motor node transforms ROS messages into string commands for Arduino UNO.

5.4.8.1 Pure Pursuit node

The Pure Pursuit node focuses on tracking the planned path. For this reason, the node subscribes to the *robot_position* and *path* topics. The *robot_position* topic provides information about the current vehicle position, while the *path* topic includes information about the planned path that the car should follow. The output of the node is published commands for motors.

When the node receives a new planned path, the vehicle starts to follow it. The principle for path tracking is based on the "follow-the-carrot" algorithm [34]. The name derives from the situation when camels pursue carrots, which are in front of them on a string. The similarity with this situation is that the vehicle follows the point on the path in front of it. Furthermore, as well as the camel, the car never reaches the chasing point, except the end position.

The algorithm starts after receiving a new path. Afterward, the node prepares a list of breaking points along the path. These points are flagged once the vehicle approaches within a specified distance threshold. To determine the chasing point, the algorithm calculates the intersection between the circle with a given radius and the path nearby the current vehicle position. The circle center is located in the vehicle and the circle radius determines the distance between the chasing point and the car.

Once the chasing point is computed, the algorithm calculates the angle between the vehicle's orientation and the chasing point. Afterward, the angle is utilized as the input parameter for a regulator. We apply the proportional-integral (PI) regulator for its simplicity and lower computational overhead compared to more complex methods like MPC. At the end of the algorithm, the node publishes

commands for motors generated by the regulator to the *controlMotor* topic. Figure 18 depicts the computational process of the "Pure Pursuit" approach.

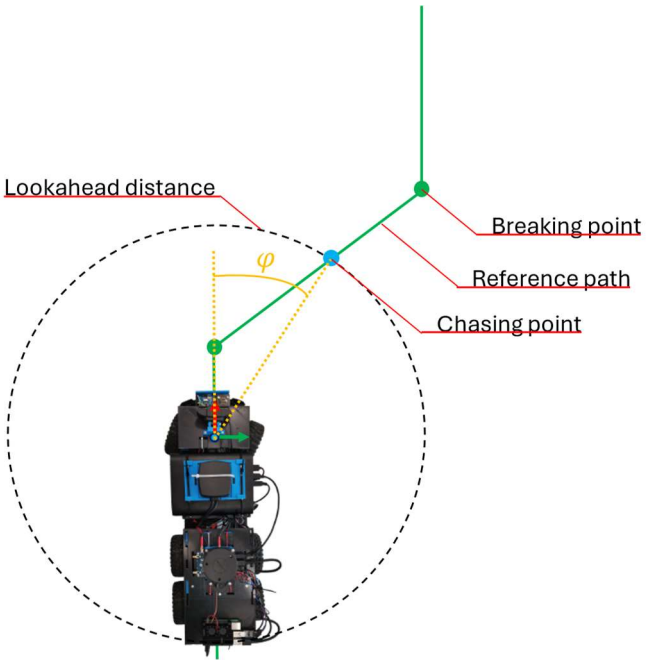


Figure 18: Illustration of the Pure Pursuit algorithm

5.4.8.2 MPC node

The MPC node represents a more sophisticated approach to path tracking compared to the Pure Pursuit node. The node subscribes to the *robot_position* topic, the *path* topic and the *map* topic. Additionally, the node subscribes to the *dyn_obs* topic, which contains information about dynamic obstacles processed by the Dynamic Obstacles node. The principal advantage of this approach is that the MPC node takes into account avoiding dynamic obstacles. The Pure Pursuit node is not able to dodge a dynamic obstacle. When the Pure Pursuit node is used to follow the path, only the Emergency Brake node prevents collision in the immediate vicinity. On the contrary, the MPC node registers surrounding obstacles, estimates their movements and tries to avoid them.

The implemented MPC algorithm is based on the depth first search (DFS) approach. The higher depth causes that the algorithm optimizes the longer part of the path in front of the vehicle. The longer optimized path improves the car's behavior in terms of avoiding dynamic obstacles and following the reference path. On the other hand, the higher depth means that the task is more computationally demanding. Furthermore, the too long path hardly changes the optimized path because most of the path is far from the current position and does not affect the next vehicle move. In our case, the algorithm executes at a maximum of eight layers in depth, which poses 1.6 m of path.

The algorithm begins at the current vehicle position. The subsequent vehicle position is computed using the Ackermann model. The input parameter of the model is the steering angle reached by dividing the vehicle's steering range. The division of the range determines the possible steering utilized for the next iteration. Therefore, the possible tracks are branching symmetrically.

When the new possible subsequent position is computed, the algorithm verifies whether the position is feasible. If the position collides with an obstacle, the branch is closed. Otherwise, the cost of the position is determined. The computation of cost contains the distance from the reference path and the distance from dynamic obstacle. The distance from reference path should be as small as possible. The distance from a dynamic obstacle should be higher than the given limit. If the distance from the

obstacle is exceeded, the cost of the position is penalized. Figure 19 depicts possible tracks with two layers and three possible steering angles. Moreover, the picture illustrates the distances utilized for the cost computation.

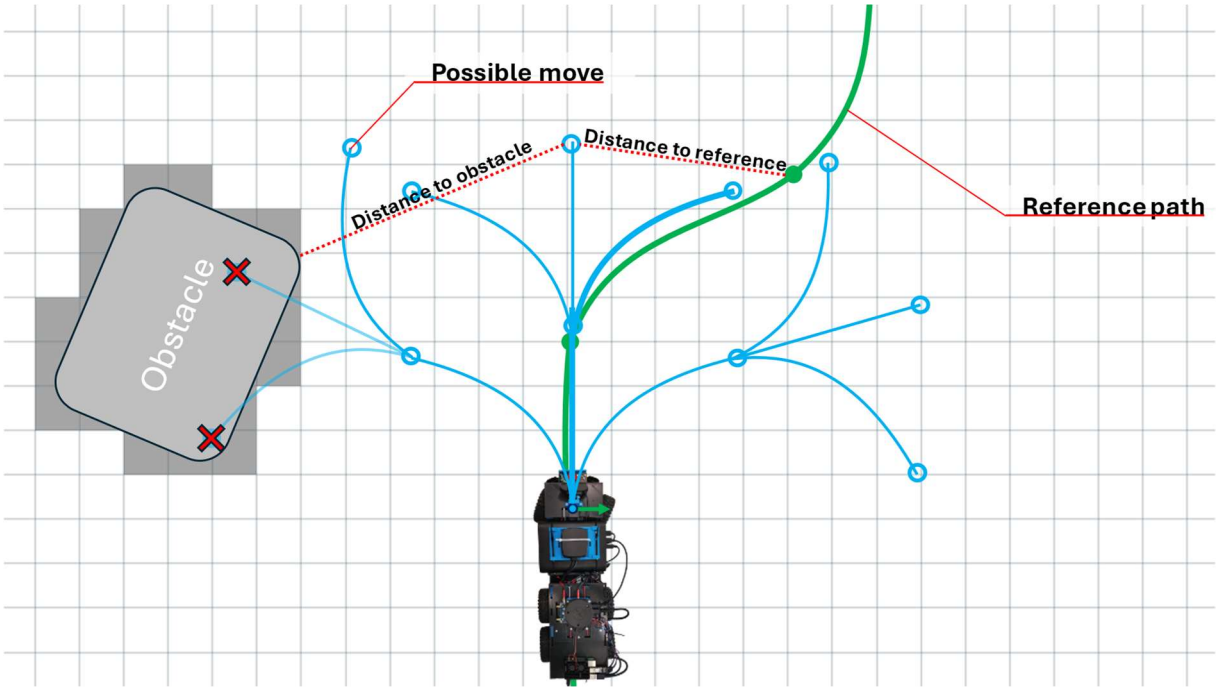


Figure 19: Illustration of the MPC algorithm

The DFS approach enables efficient pruning of the track tree, optimizing computational resources. When the algorithm reaches the tree leaf, it records the cost of the leaf. If the following computed cost of subsequent positions is higher than the recorded cost, the tree branch is pruned. Otherwise, if the cost of the position is lower than the recorded cost, the value of the recorded cost is changed to the lowest computed one. To support the pruning of the tree, we prioritize positions with minimal cost and maximal depth. In the case of no dynamic obstacles, only one route of the tree should be explored, and other routes should be pruned.

5.4.8.3 Control Motor node

The Control Motor node is a bridge between computation nodes and the Arduino Uno microcontroller. The node subscribes to the *controlMotor* topic containing control commands for the motors from the MPC node, the Pure Pursuit node or the Emergency Brake node.

At the beginning of the node, a connection with the Arduino is established. The subscribed messages are converted into commands suitable for the Arduino. These commands are categorized into three types: enabling and disabling autonomous driving mode, adjusting steering, and controlling forward speed. For compatibility with the Arduino, commands follow a specific format.

The commands start with a letter that determines the motor. The letter 's' represents steering, and 'v' changes vehicle velocity. The letter is followed by white space. The subsequent part of the command is a number between -100 and 100, representing power in percentage. In the context of speed control, the positive number determines the power of movement forward, whereas the negative number indicates movement backwards. In the case of steering, the negative number represents turning left, and the positive number represents turning right. Each command concludes with an end-of-line character.

5.4.9 Communication nodes

The vehicle is connected to a 5G cellular network via a 5G module. If the algorithm for path planning, detecting dynamic obstacles or path following is offloaded, the vehicle sends all necessary information for the offloaded algorithm to the gNB. The communication between the vehicle and the gNB is arranged by a script on the vehicle. The script establishes a TCP connection and receives server's decision which algorithm will be offloaded. When the gNB obtains the necessary data from the vehicle, the gNB forwards it to the MEC server. When the MEC server receives all the necessary information, the algorithm is started. Once the algorithm finishes, the MEC server transfers the result to the gNB which forwards it to the vehicle.

Each algorithm for autonomous driving is computed in ROS environment. On the MEC server, the ROS environment is launched in Docker. Docker allows users to run scripts on the device regardless of the operating system and installed software libraries. On the other hand, Docker does not support a graphical interface. It follows that the user cannot set a desired destination for the autonomous car from the graphical RVIZ application in Docker.

For this reason, on the computer controlling gNb is installed ROS2 environment without Docker, because the operation system of the computer is Ubuntu 22.04 LTS which already does not support ROS environment. This solution for experiments is applied because the autonomous vehicle cannot connect to the 5G cellular network and Wi-Fi simultaneously. Wi-Fi was used for controlling the ROS environment on the vehicle through SSH. For this reason, the vehicle also communicates with the MEC server for the purpose of launching and shutting down ROS nodes. In a real car, controlling the system will be ensured by a display mounted on the car dashboard. The diagram of the connection is displayed in Figure 20.

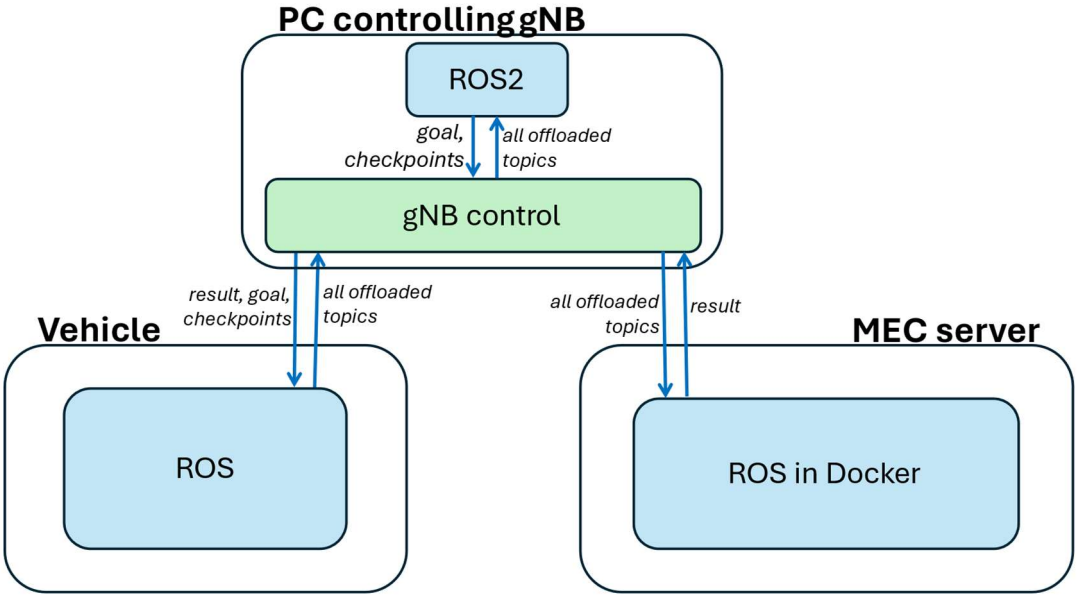


Figure 20: Connection between the vehicle and the MEC server

5.4.9.1 Gateway node

The Gateway node acts as a crucial link between the ROS environment and the server. At the beginning, a connection to the server is established. Afterward, the node expects a offload decision message from the server. The message determines the list of nodes that are going to be offloaded. When the offload decision message is received, the mentioned nodes are muted. The muted node computes no tasks and does not react to the subscribed topic except the offload decision. For offloaded nodes, all

subscribed topics are rerouted to the server for processing. For this reason, the gateway node subscribes to nearly all topics within the ROS environment.

When the offload decision message from the server is received, the node publishes the information to the *controlNode* topic to mute the node mentioned in the message. Afterward, the node starts resending ROS messages intended for the offloaded node. The process of resending is divided into two steps. In the first step, the node converts the ROS message into JSON format. Furthermore, the node appends the name of the topic to the JSON message. Subsequently, the JSON message is transmitted to the server. On the server, the JSON message is converted to the ROS message and published to the particular topics.

The last task of the node is listening to the server. When the server sends the offloading result, the node converts the JSON message to the ROS message and publishes the result to the proper topic.

5.4.9.2 Control node

The node is responsible for communication with the server and management of the ROS environment. If the vehicle does not utilize the 5G module, the vehicle is handled by the SSH protocol. However, the simultaneous use of the SSH protocol and the 5G module is not feasible. For this reason, the Control node establishes a connection with the server using the 5G module. Commands received from the server are interpreted and executed within the ROS environment. The list of possible commands is depicted in Table 2.

Table 2: List of the commands accepted by Control node

Command	Description
run <node>	Launch the node or the launch file if the program is not running yet
kill <node>	Stop the node or the launch file if the node is running
reset	Shut down the ROS environment and launch new ROS master node
q	Shut down the ROS environment and terminate the control script

The node utilizes the library *roslaunch* representing API for ROS environment. The principal benefit of the ROS API is the appropriate behavior of nodes in the environment. The alternative method to launch and terminate nodes involves the *Popen* function in the *subprocess* library. However, if a node or the ROS master is terminated by the *subprocess* library, the node process is still running in the Linux environment. For this reason, the node process has to be closed by the command *pkill node* as well. The second approach is less elegant than the first one because the nodes are terminated by brute force. Furthermore, the second approach causes trouble in systems with several ROS environments or when the ROS environment includes multiple anonymous same nodes.

When the command sent by the server is executed, the newly launched node broadcasts the status of its launch. The described behavior applies to all nodes except SLAM, lidar and scan_matcher nodes. The mentioned nodes are encapsulated and it is not facilate to add code to verify the node launch. In that case, the Control node waits for a message from the newly launched node. The lidar node should publish a message to the *scan* topic, the scan_matcher node should provide a new transformation to the *tf* topic and the Slam Toolbox node should distribute an occupancyGrid message to the *map* topic. If the proper message is not posted within ten seconds after launch, the Control node reports an error to the server. Otherwise, the node informs the server that the node is launched correctly.

The vehicle is controlled by a server in the research phase. However, in a real-world scenario, the node would communicate with a local device in the car, such as a touch screen mounted on the car dashboard. This local approach enhances cyber security by minimizing reliance on external server commands.

6 Experiments

This section describes the experiments, which are divided into three categories. The first category focuses on path planning and compares the path planning using the A* algorithm and RRT algorithm. The second category describes experiments working with the algorithm detecting dynamic obstacles. The third category of experiments examines the pure pursuit algorithm and the MPC algorithm. The experiments in all categories are performed both locally on the car and by offloading to the MEC server.

6.1 Scenarios and metrics

The experiments investigate the behavior of algorithms for path planning, dynamic obstacle detection, and tracking the planned path computed locally and on the server. The experiments compare the computation time of the algorithms locally and on the server. The computation time is defined as the difference of times between the start and the end of the algorithm. Total offloading time is the difference between the time of sending the offloading request to the server and the time of receiving the result from the server. Total offloading time can be divided into the actual computation time on the server and the communication time, which is the time taken to send data to and from the server. In addition to comparing the individual computation and total offloading times, it is examined whether the given times meet predetermined deadlines. A deadline is met if the time is shorter than a predetermined deadline. The resulting paths of each algorithm are compared with respect to their length. The path length represents the length of the trajectory of the planned path between the start and the goal position.

A*, RRT and RRT* algorithms are compared in path planning. The RRT* algorithm is investigated with different reconnection distances. The reconnection distance refers to the maximum distance around a newly added point at which the points are reconnected to shorten the path. The RRT* algorithm is investigated with a reconnection distance of 0.4 m, 0.7 m and 1.0 m. The dynamic obstacle detection algorithm is compared with different maximum distances from the vehicle at which dynamic obstacles are spotted. The maximum distances for dynamic obstacle detection are given as 2 m, 5 m, 8 m and no limit. The algorithms compared for path tracking are Pure Pursuit and MPC. MPC is launched in two modes. The modes differ as to whether MPC takes dynamic obstacles into account. MPC is tested for different maximum depths during the experiments. Maximum depths of 1, 3, 5 and 8 steps are investigated in the experiments.

6.2 Theoretical requirements for the communication channel

If the user offloads the algorithm, it sends the necessary data to the server. For this reason, the quality of the network matters a lot. If the network is not able to provide low enough latency and high enough transfer rates, this results in deadlines not being met and put road users at risk.

Table 3 and Table 4 depict the calculation for the minimum bit rate. The parameters are calculated for a scenario where the user offloads path tracking computation with and without dynamic obstacles. Path planning is not considered in the examples because the path planning algorithm is usually executed only once. Moreover, we assume that we are moving through a mapped environment and therefore we just need to send the map once at the beginning. For this reason, only position and lidar data are sent to the server. The current position and lidar data are sent at the maximum frequency so that the control algorithm is not negatively affected. The response from the server is a control command that sets the motors on the vehicle. Control commands are sent in response to a received vehicle position. If we consider the computation time to be constant, the control commands will be sent at the same frequency as the current position messages.

All values in the table are measured during the experiments. We consider a deadline of 2 s for route planning, 100 ms for dynamic obstacle detection and 50 ms for the control algorithm. For the calculation, we consider that the downlink has twice the speed of the uplink. Algorithm computation time is taken as the average time of the most complex measured algorithm from the experiments described in the following sections. Thus, RRT* of 1.0 m is used for path planning, we do not crop the map for obstacle detection, and we consider MPC with a maximum depth of 8 steps for control. The worst case option is considered for calculating the transfer rate. So during the smallest deadline for message transmission, all other possible messages are placed before it.

Table 3: Data rate requirements – without obstacles

Message	Frequency [Hz]	Message Size [kb]	Maximum transmission time [ms]	Minimum data rate [kb/s]
Position	20	2	38.89	51.44
Control motor	20	0.5	4.86	102.88

Table 4: Data rate requirements – with obstacles

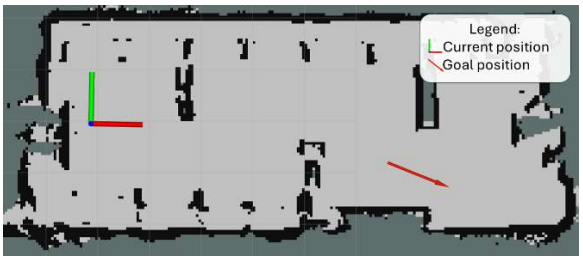
Message	Frequency [Hz]	Message size [kb]	Maximum transmission time [ms]	Minimum data rate [kb/s]
Position	20	68.23	43.59	1565.03
Scan	10			
Control motor	20	0.5	0.16	3125

Table 3 presents a simple example of an offloading control algorithm without dynamic obstacles. Since only messages containing the current location are sent to the server periodically, the minimum data rate is obtained as a quotient of 1000 and the maximum transmission time and multiplied by the size of one message. The downlink is calculated in the same way, which should be twice as large.

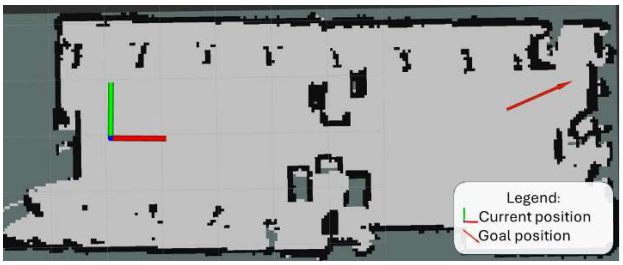
Table 4 presents a more complex problem where dynamic obstacle detection is considered in addition to the control algorithm. In the worst case, the request is sent both lidar data and position to the server at the same time. Since the control algorithm responds to the position, the server should receive 68.23 kb (66.23 kb lidar data and 2 kb position) before computing. We still assume that the downlink rate is twice as fast as the uplink rate. This implies that 68.23 kb should be transferred in 43.59 ms. Using a similar calculation as in Table 3, the minimum data rate for uplink is 1565 kb/s and downlink is 3125 kb/s.

6.3 Path planning

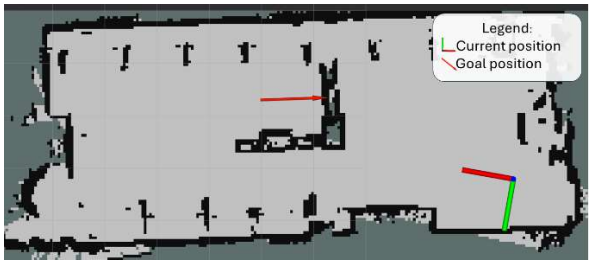
This subsection explores path planning using A*, RRT and RRT* algorithms. In order to run the experiments under constant and controlled conditions, the rosbags are recorded before the experiments. The rosbags are files in ROS environment which subscribe to selected topics and record all the published data to them. Therefore, the ROS environment can simulate the same situation by playing rosbag files. A great advantage of rosbag is that all data recorded are acquired from the real environment. The rosbag files include information about mapped surrounding environment, current position and user request for the goal destination. The experiments are performed on five different maps, which are shown in Figure 21.



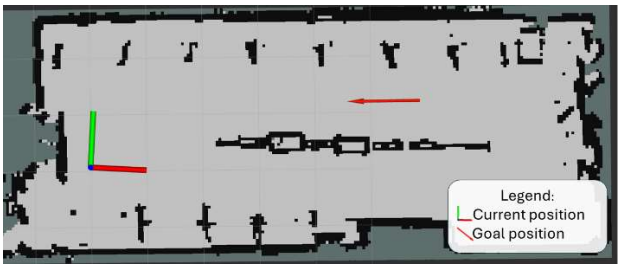
A: Map1 - path planning experiments



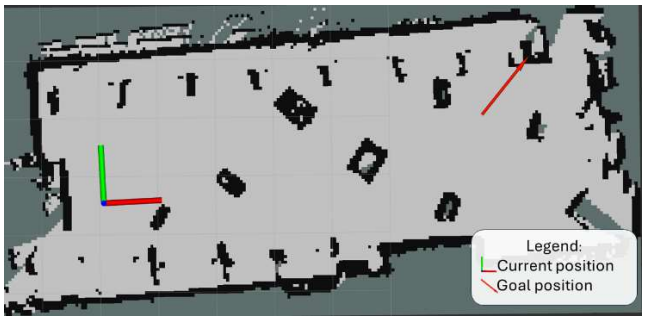
B: Map2 - path planning experiments



C: Map3 - path planning experiments



D: Map4 - path planning experiments



E: Map5 - path planning experiments

Figure 21: Maps – path planning experiments

The maps are chosen to test several different scenarios that the vehicle could get into. Each map shows the current vehicle location (coordinate axes), which does not change during the path planning experiments. The red arrow indicates the user destination. In addition to the X and Y coordinates, the user specifies the rotation of the vehicle at the destination.

The first map tests the reaction to obstacles and verifies if the dilatation of the obstacles is sufficient. The second map represents the situation of passing through a narrow space. Narrow passage creates a challenge for the planning algorithms [35]. Due to the forced dilatation, the A* algorithm cannot plan a path through these narrow passages. On the other hand, algorithms using random points to branch the search tree such as RRT algorithms usually need a larger number of iterations to overcome the narrow passage. The third map tests how greedy the given algorithms are and try to follow the target as the crow flies. Both of the two algorithms follow the target, but a part of the algorithms involves searching the map that is not exactly between the start and goal positions. The fourth map verifies the algorithms work with the final vehicle rotation. The final fifth map contains several scattered obstacles that make the overall path finding more difficult.

The experiments are repeated 15 times for each scenario by playing rosbag. The number of iterations plays almost no role for the A* algorithm, because the A* algorithm is deterministic and therefore always finds the same path. In contrast, RRT and RRT* select a random point to which the algorithm searches for a point in the K-D tree with the minimum distance. For this reason, in some cases, the path is not found. Moreover, the RRT and RRT* algorithms publish different paths with a different length. One of the aspects of experiment is the percentage of met the maximum time (deadline) to find the

path. The deadlines for path planning are set to 0.5 s, 1.0 s, 1.5 s, 2.0 s, 2.5 s, and 3.0 s. Another aspect of measurement is the length of the path. The MEC server should return a shorter path because the higher performance of the MEC server ensures more executed iterations in the same amount of time. The higher number of iterations increases the probability of finding the path or shortening it.

Figure 22 compares the average computation time for each algorithm processed locally on the vehicle, on the server, and total offloading time.

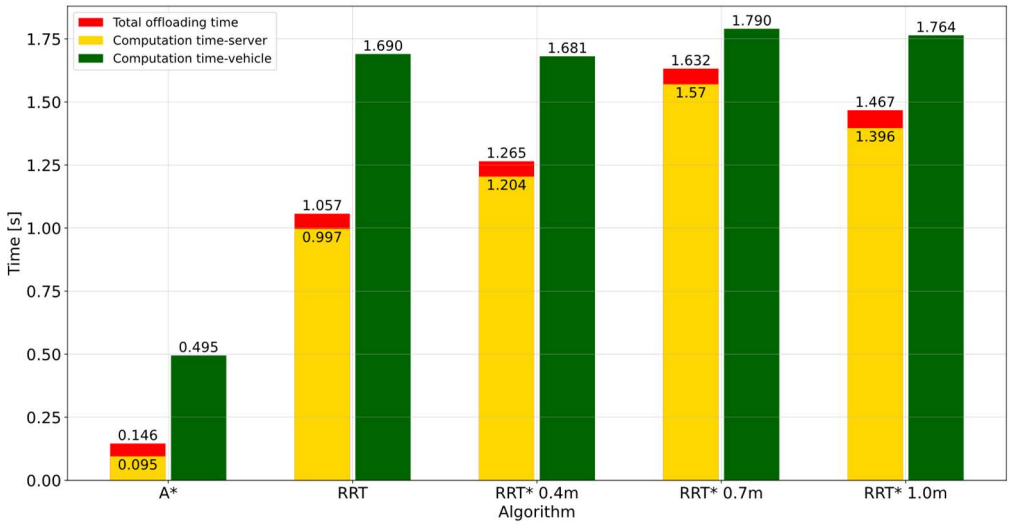


Figure 22: Average computation times – path planning

The A* algorithm runs 5.2 times faster on the server, and after adding the time spent sending and receiving data from the server, the result is obtained 23.5% (349 ms) earlier than the local computation. In addition to the fact that the user gets the result sooner, the Raspberry Pi is only used to send the request and, therefore, the power consumption should be reduced. The RRT algorithm runs 41% faster on the server and after including the time taken to send data, the result is received 37.5% (633 ms) earlier than the local computation. Similar results occurred for the RRT* algorithm, which returns the result from the server 24.7% (416 ms), 8.8% (158 ms), and 16.8% (297 ms) earlier than local computations for 0.4 m, 0.7 m, and 1.0 m reconnection distances.

In Figure 23 and Figure 24, the success rate of finding the path within the given time limit is depicted. The results show that computation on the server do not improve the path finding success rate using the A* algorithm except for a deadline of 0.5 s. The A* algorithm provides a path to the target in only 88%, because the obstacle dilatation in the narrow passage on map 2 obstructs the path to the target. On the other hand, the algorithm returns the information in the deadline that a path to the target does not exist using the A* algorithm and therefore the maximum success rate is 100% not 88%. The RRT and RRT* algorithms running on the server increase the success rate of the path found within the specified deadline in all cases. This is due to the fact that the server has a more powerful CPU, which allows a higher number of iterations to be performed in the same amount of time, thus increasing the probability of finding a path.

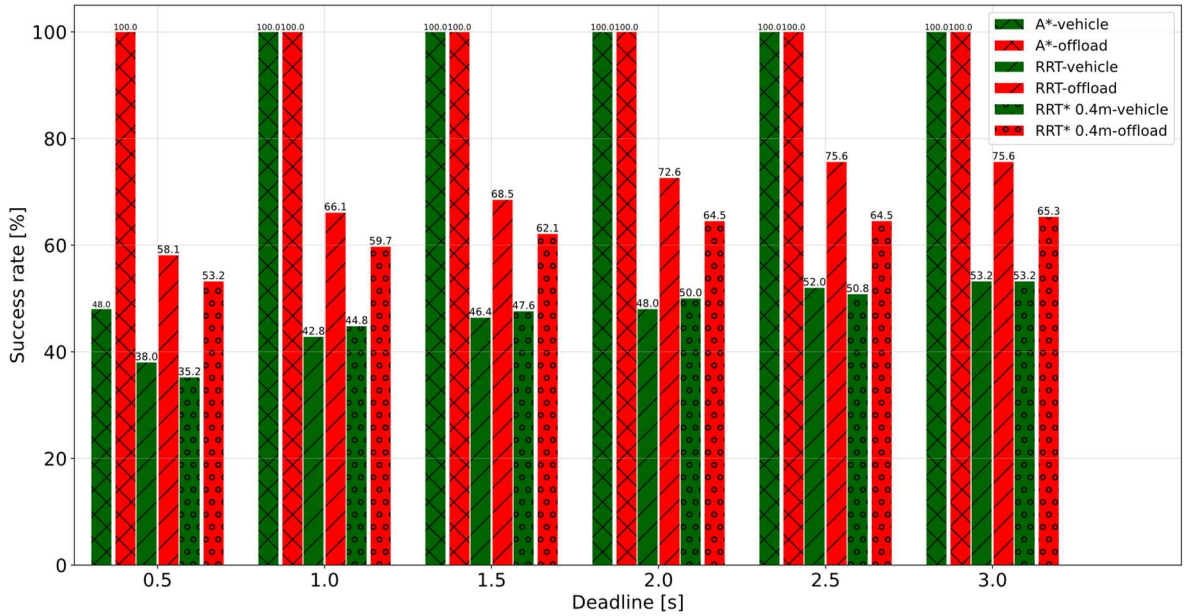


Figure 23: Success rate of algorithms for different deadlines – part 1

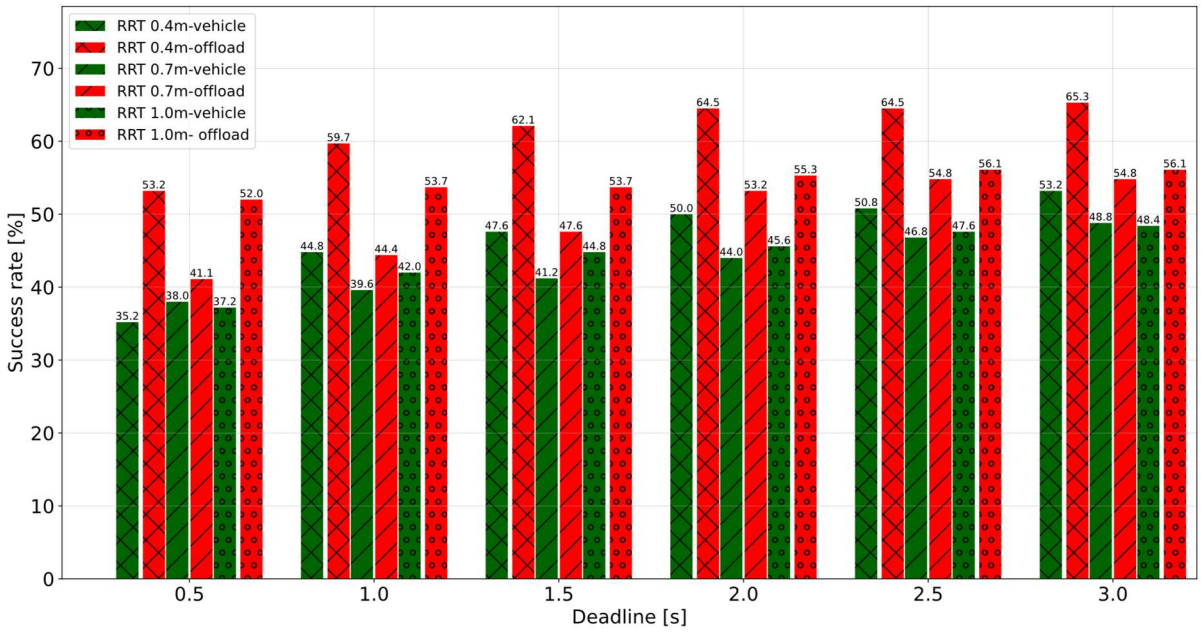
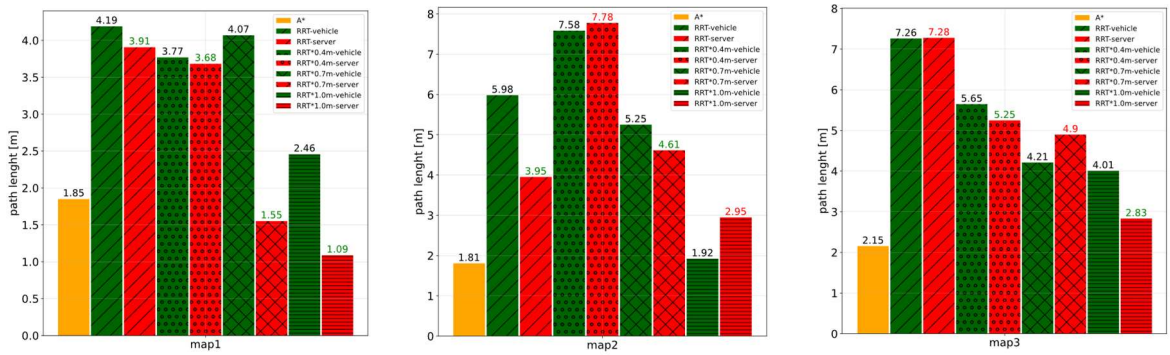
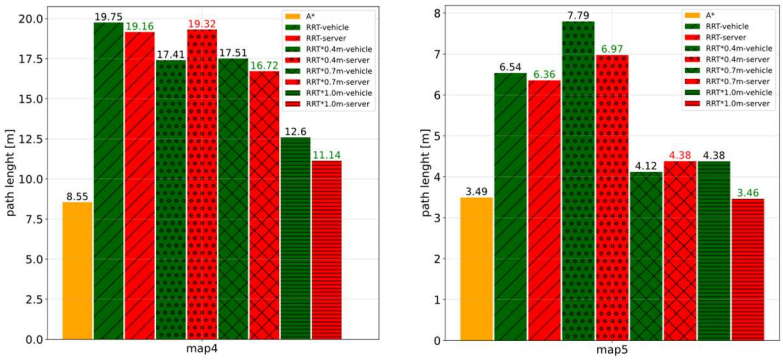


Figure 24: Success rate of algorithms for different deadlines – part 2

Figure 25 compare the average path lengths found by each algorithm.



A: Map 1-path lengths *B: Map 2-path lengths* *C: Map 3-path lengths*



D: Map 4-path lengths *E: Map 5-path lengths*

Figure 25: Comparison of path lengths

According to Figure 25, the shortest paths are achieved by the A* or RRT* algorithm. In general, the A* algorithm returns the shortest possible path. In our case, this is the shortest path including penalties for inappropriate behaviour in terms of vehicle dynamics. The longest paths are generated using the RRT algorithm. During iterations, the RRT* algorithm reconnects the points in the built K-D tree and thus reduces the final path length. The higher the reconnection distance of RRT* algorithm leads to the shorter final path on average. The lengths of the paths found by the RRT* algorithm with a reconnection distance of 1 meter are close to the paths found by the A* algorithm or they are shorter. In addition, the RRT* algorithm works with the vehicle model, unlike the A* algorithm, and therefore the path planned by RRT* should be better adapted to vehicle dynamics.

Figure 25 show that the average length of the path found on the server is shorter than the path computed by the same algorithm locally in 14 out of 20 cases (marked with green numbers in Figure 25 above the bars). This is due to the fact that the higher computational power of the MEC server guarantees more iterations at the same time.

The higher the distance for reconnection in the RRT* algorithm, the shorter paths are achieved on average. On the other hand, point reconnection is a very challenging operation. First, the points in the neighborhood are found, then checked if the connection is possible without collision and without violating vehicle dynamics. For this reason, the one iteration time in RRT* algorithm is increased. In Table 5 and Table 6, we see the average time of one iteration for RRT and RRT* algorithms, the average number of iterations per second and the average number of reconnections for each configuration of RRT* algorithm computed on the vehicle and on the server. The tables summarize information about the iteration time and number of reconnections that are related to the resulting path length. The A*

algorithm is deterministic, so the iteration time does not affect the resulting path length. For this reason, the A* algorithm is not included in Table 5 and Table 6.

Table 5: Comparison of iterations and reconnections - vehicle

Algorithm	One iteration time [ms]	Number of iterations per second	Average number of reconnections
RRT	1.503	665.335	0
RRT* 0.4m	2.318	431.406	1559.6
RRT* 0.7m	2.995	333.889	3451.4
RRT* 1.0m	3.879	257.798	4727.8

Table 6: Comparison of iterations and reconnections - server

Algorithm	One iteration time [ms]	Number of iterations per second	Average number of reconnections
RRT	0.826	1210.653	0
RRT* 0.4m	1.157	864.304	1623.2
RRT* 0.7m	1.704	586.854	5243.0
RRT* 1.0m	2.492	401.284	10164.6

The tables demonstrate that the server can process 56% to 100% more iterations than the Raspberry Pi can process locally. The path finding success rate increased by 16% on average in specified deadlines. The measured path lengths computed on the server are on average 6.3% (0.46m) shorter than paths computed locally. Hence, the main benefit of higher performance on the server is a slight increase in the success rate of the RRT and RRT* algorithms.

6.4 Dynamic obstacle detection

The testing of the dynamic obstacle detection is conducted in a similar manner to the testing of the path planning algorithms. In the experiments, the environment is mapped using a Slam Toolbox. The vehicle stands still in the space collecting lidar data. Under these conditions, the rosbag is recorded to achieve constant and controlled conditions during the experiments. Three courses are recorded. In the first run, there is only a single person moving in the space. In the second case, the dynamic obstacle is represented by two walking people. In the last scenario, there are 3 people moving around the room. The movement of dynamic obstacles is random at arbitrary distances from the vehicle. Figure 26 demonstrates the mapped environment. The red points indicate the data measured by the lidar. The figure shows the third scenario where 3 people are moving around the vehicle (marked with a blue ellipses).

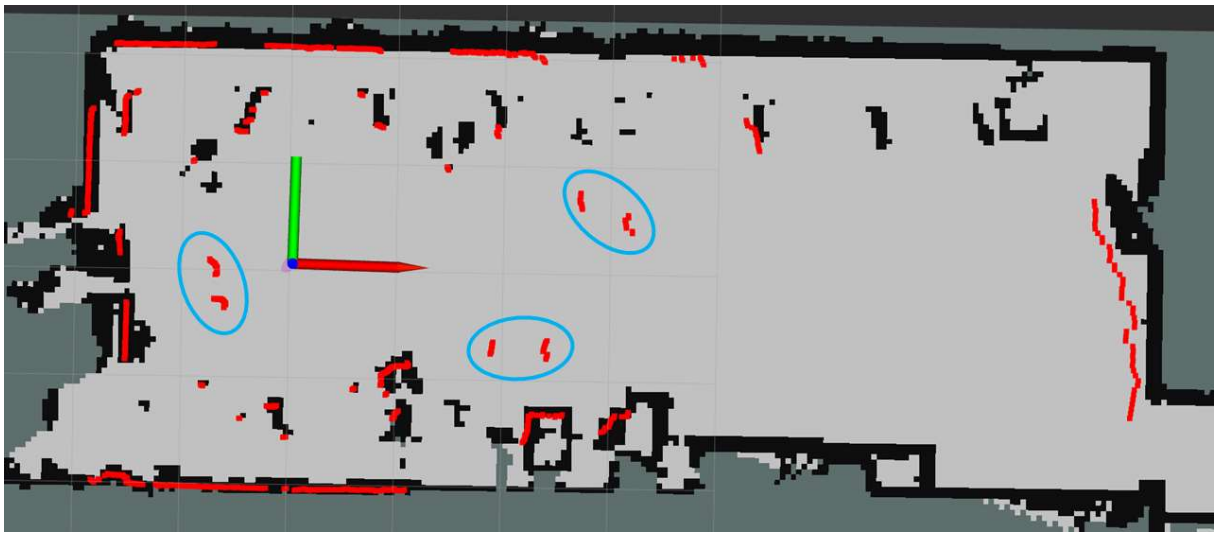


Figure 26: Third scenario testing dynamic obstacle detection

After receiving the lidar data, the algorithm checks if there is a dynamic obstacle around the vehicle. The size of the examined area is determined by the distance of half of the side of the square, where the vehicle is located in the middle of the mentioned figure. The length determining the size of the examined area is set to 2 m, 5 m, 8 m and infinity. A larger space helps to better predict the movement of dynamic obstacles. On the other hand, the algorithm working with larger space represents a more computationally demanding task. Due to the 9V power supply (see section 4.5), data from the lidar arrives every 100 ms, so the computation time should not exceed 100 ms. If the lidar is powered by USB from a Raspberry Pi, due to the lower voltage, data would come in at 5.5 Hz, which would represent 182 ms between messages. In our experience, we can set the lidar message frequencies up to 15.5 Hz. For these reasons, the deadlines for dynamic obstacle detection are set to 64ms, 100ms and 182ms.

Figure 27 demonstrates the average computation times for dynamic obstacle detection computed locally and on the server. Total offloading time is depicted in Figure 27 as well.

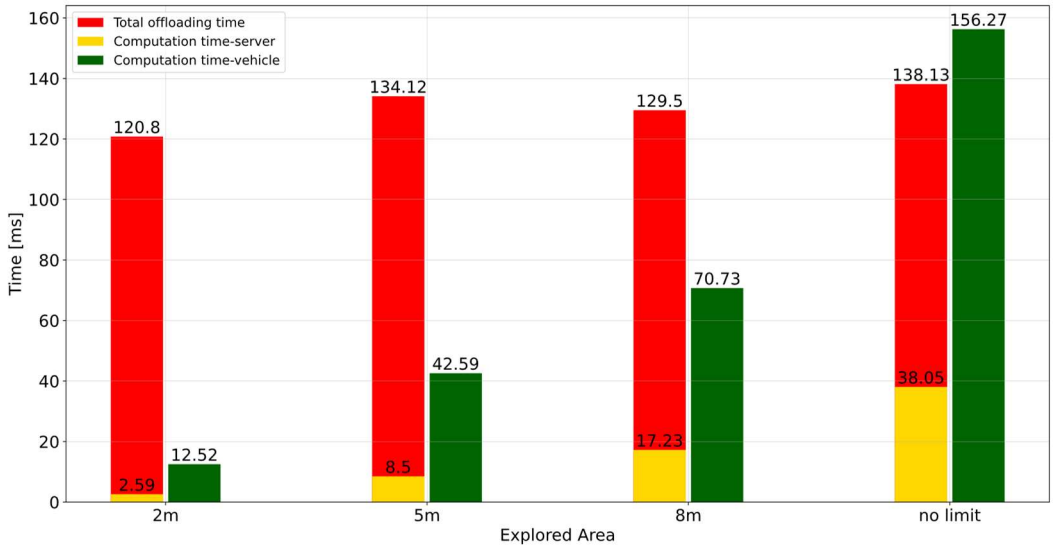


Figure 27: Comparison of computation time of dynamic obstacle detection

Figure 27 demonstrates that the computation on the server itself is on average 4.3 times faster than on the vehicle. On the other hand, the average total offloading time is lower than the local computation only when algorithm operates with a whole map. The figure shows that the largest part of the total offloading time is the communication part. Moreover, the values of total offloading time are not monotonic because the task is not computationally demanding enough, and therefore the results rather indicate the current channel quality. For this reason, the average total offloading time is 76.4% (53.9 ms) longer than the local computation.

Figure 28, depicting the detection success rate for each deadline, demonstrates that some of the results had large delays, hence the high average offloading time.

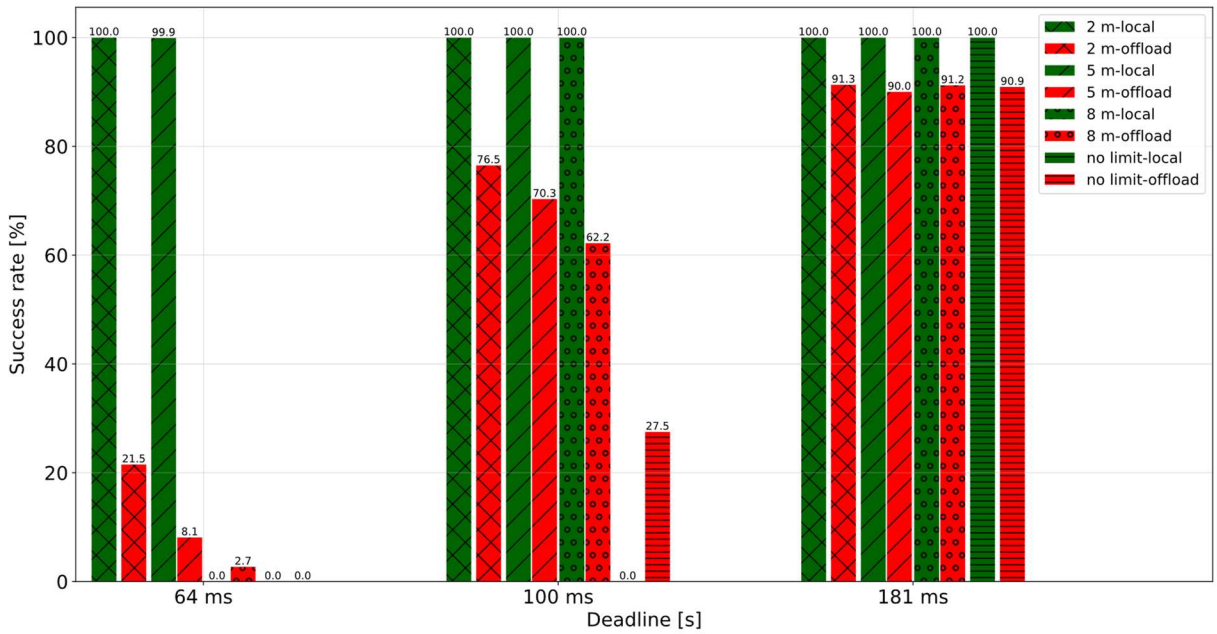


Figure 28: Success rate of obstacle detection for different deadlines

The results shown in Figure 28 indicate that in only 2 cases is the deadline met by offloading with a higher success rate than when the algorithm is executed locally. These are cases where the deadline is short and the problem is enough computationally demanding. Therefore, the vehicle does not meet the deadline even once and offloading meets some deadlines.

During the experiment, the shortest offloading time is 26 ms and the longest offloading time is 1.5 s. The high delay was due to the degradation of the transmission speed. An algorithm that optimizes the offloading decision should respond to this situation by reducing the number of messages sent to the server. This algorithm is not used in the experiments because in that case the deadline of each offload performed would have been met, thus significantly affecting the results.

6.5 Path following

Recorded rosbags are used to test tracking of the planned path and avoidance of dynamic obstacles. Firstly, the surrounding environment is mapped. Afterwards, a path is created which the car starts to follow. During the following the path, a dynamic obstacle appears. The dynamic obstacle is simulated by walking human. Figure 29 illustrates the scenario of the experiments.

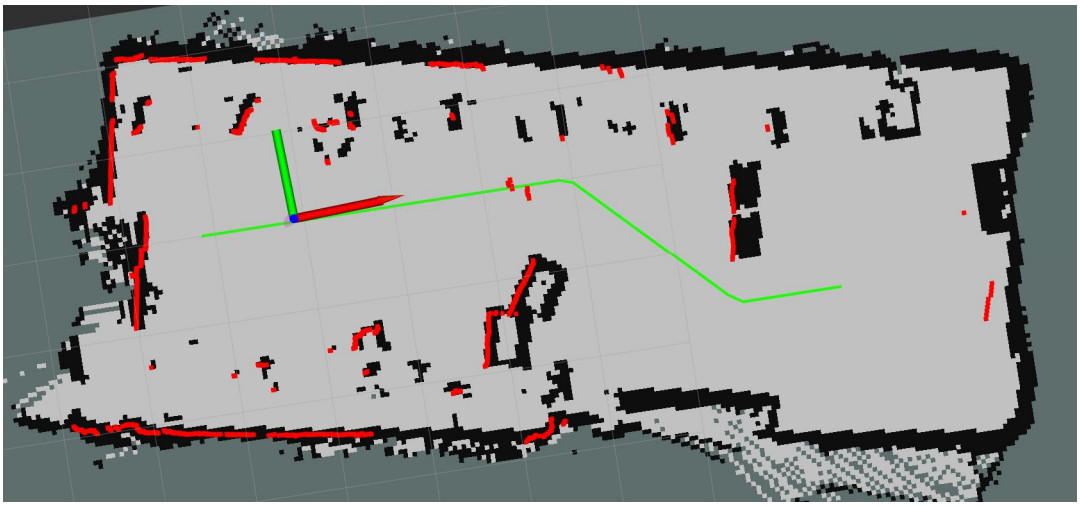


Figure 29: Scenario testing path following with obstacles

The behaviour of the Pure Pursuit algorithm and the MPC algorithm is investigated on the recorded data. Pure Pursuit is a simple version of the algorithm that follows a planned path. The disadvantage of the Pure Pursuit algorithm is its inability to respond to dynamic obstacles. To compare the MPC and Pure Pursuit algorithms, the MPC algorithm is launched in two modes. In the first mode, it does not react to dynamic obstacles and only follows the planned path in the map. This mode is closest to the Pure Pursuit algorithm except for the fact that MPC works with a vehicle model. The second mode responds to dynamic obstacles, increasing its computational and time complexity.

The MPC algorithm searches in depth all possible routes from the current location. The greater the depth to which MPC searches, the better the algorithm responds to the surrounding environment and dynamic obstacles. For this reason, MPC is run with four different maximum depths. The depths are 1,3,5 and 8 steps, where 1 step plans 20 cm ahead. The maximum time to execute the algorithm is 100 ms because new information about dynamic obstacles arrives in this time interval.

Figure 30 compares the Pure Pursuit algorithm and MPC disregarding dynamic obstacles computed locally and on the server. Figure 31 compares different depths of MPC taking into account dynamic obstacles. Total offloading time is included in the comparison as well.

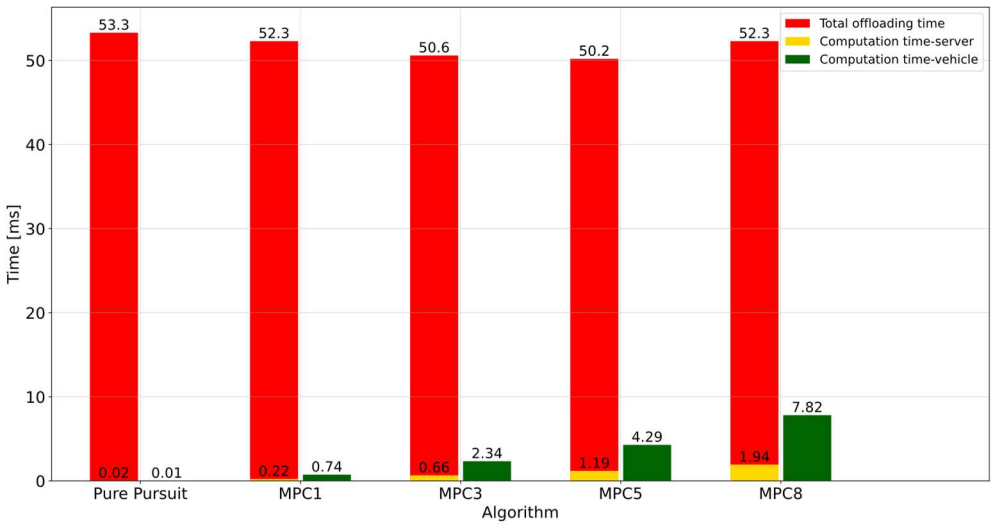


Figure 30: Comparison of computation times – path following without obstacles

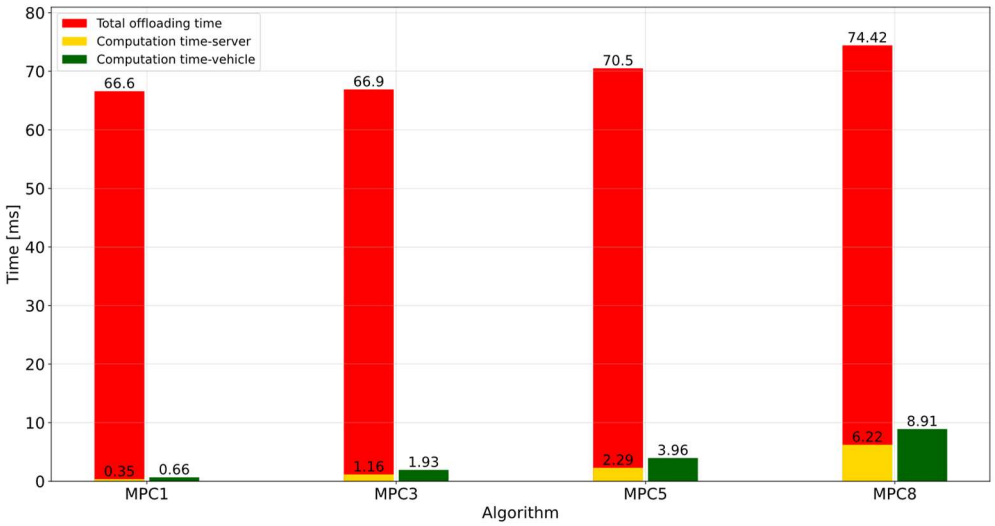


Figure 31: Comparison of computation times – path following with obstacles

The computation of algorithms itself is 2.18 times faster on the server than locally. Because the computation of the control algorithms is very fast, the transmission of data makes up the largest part of offloading. Moreover, the values of total offloading time are not monotonic because the task is not computationally demanding enough, and therefore the results rather indicate the current channel quality. For this reason, the average offloading times are 17.5 times (56.3 ms) longer than the local computation.

Figure 32 and Figure 33 demonstrate the success rate of the algorithms for each deadline. Figure 32 compares algorithms that do not take dynamic obstacles into account. Figure 33 compares MPC algorithms that take dynamic obstacles into account. The MPC algorithms differ in the maximum depth of the optimal path search.

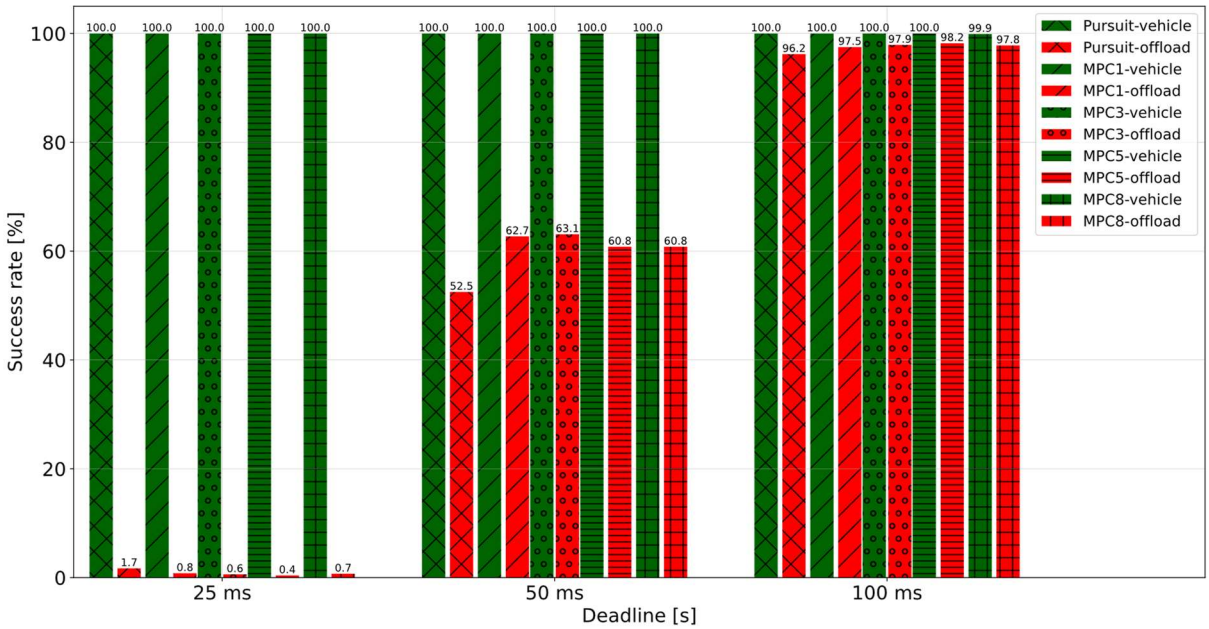


Figure 32: Success rate of control algorithms for different deadlines – without obstacles

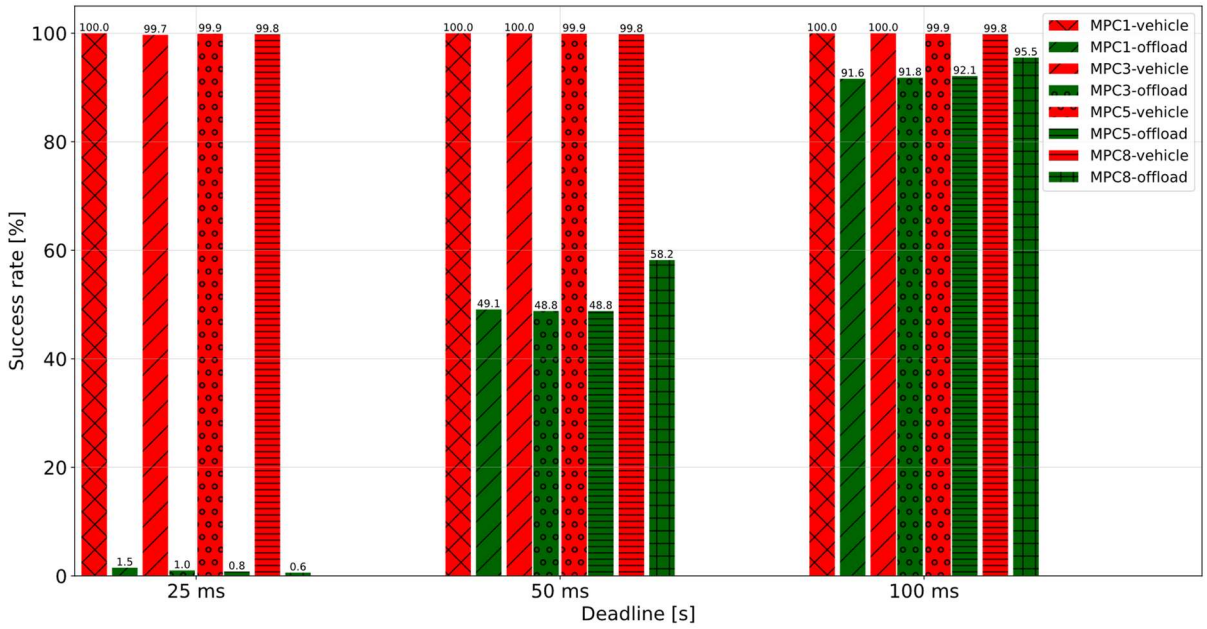


Figure 33: Success rate of control algorithms for different deadlines – with obstacles

The results of both experiments are similar. With rare exceptions, all local computations meet the specified deadlines. At most 1.7% are met by the shortest deadline of 25 ms. The second deadline are met in up to 63.1%. During the 100 ms deadline, offloading is successfully executed in more than 91%.

Due to the low complexity of the task and the short time limits, the success rate of the algorithms mainly based on the time it takes to transfer the data. This property is reflected in the figure comparing the algorithms taking into account dynamic obstacles, where the most complex algorithm for deadlines of 50 ms and 100 ms achieves the highest success rates. In order to achieve a higher success rate, an offload decision algorithm that takes into account the quality of the transmission channel should be used.

6.6 Dynamic experiment

During the dynamic experiment, a connection is established with the MEC server where the user input a request for the destination. Subsequently, a map and the current location of the vehicle is sent from the car. The path computation is performed on the MEC server. Once the route is planned, the car starts following the planned path. Messages containing the current position and lidar data are periodically sent from the vehicle to the MEC server. The computation of the control algorithms is also performed on the MEC server and the car only receives commands for the motors.

Dynamic tests are carried out, but no conclusion can be drawn from this because a controlled and constant environment cannot be guaranteed. The demonstration of the dynamic experiments is captured on video [36]. Figure 34 illustrates the recorded course of dynamic experiments.

Figure 34: Demonstration of offloading[1]

[1] Video available on https://www.youtube.com/watch?v=aPKcAR9Qli4

7 Conclusion

In this thesis, we have implemented algorithms for autonomous driving with the possibility of offloading of various driving-related functionalities to the edge server. The autonomous driving is divided into three parts: path planning, dynamic obstacle detection and following the planned path. Path planning is performed using modified A*, RRT and RRT* algorithms. Path planning is done in the map that is created before the experiments. Dynamic obstacle detection is performed using a "two-pass" algorithm with a Union-Find approach. Then, MPC and Pure Pursuit algorithm is used for path following.

During the offloading of path planning, the computation result is received from the server on average 23.5% (349 ms) earlier than during the local computation. In addition, the RRT and RRT* algorithms achieve a higher deadline success rate of 16% on average due to the higher computational power of the MEC server, resulting in a higher number of executed iterations. The higher number of iterations for the RRT and RRT* algorithms on the server also leads to 6.3% (0.46 m) shorter paths on average. Dynamic obstacle detection is faster using offloading only if it is a whole-map detection. In other cases, the average local computation is faster than the offloading. The local computation has a higher success rate in meeting the specified deadline, except when the local computation is not completed even once within the specified time limit. The computation of the control algorithm by the MEC server does not outperform the local computation in any experiment. The control algorithm is not computationally intensive enough to make the delay caused by transmitting data for processing in MEC server worthwhile.

In future work, we will focus on dynamic experiments and more complex cooperation between the server and the autonomous vehicle. The K-D tree created during path planning can be stored and used to reduce the time of the next run of the RRT and RRT* algorithms in the same environment. In addition, we will focus on the cooperation of multiple autonomous cars in the same space. Vehicles can share information between each other about a planned path, a mapped area or a future maneuver. Furthermore, they can share communication and computing resources.

8 Bibliography

[1] A. Gunawan, B. Geraldo, K. Honggiarto, and F. L. Cahyadi, “Understanding the Uses and Potential of IoT with 5G Technology Compared to 4G LTE: A Systematic Literature Review,” Aug. 2023, doi: https://doi.org/10.1109/icimtech59029.2023.10277995.

[2] D. Spatharakis, M. Avgeris, N. Athanasopoulos, D. Dechouniotis and S. Papavassiliou, "A Switching Offloading Mechanism for Path Planning and Localization in Robotic Applications," 2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics), Rhodes, Greece, 2020, pp. 77-84, doi: 10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics50389.2020.00031.

[3] Y. Yamato, T. Demizu, H. Noguchi and M. Kataoka, "Automatic GPU Offloading Technology for Open IoT Environment," in IEEE Internet of Things Journal, vol. 6, no. 2, pp. 2669-2678, April 2019, doi: 10.1109/JIOT.2018.2872545.

[4] P. Mach and Z. Becvar, “Cloud-aware power control for real-time application offloading in mobile edge computing,” Transactions on Emerging Telecommunications Technologies, vol. 27, no. 5, pp. 648–661, Dec. 2015, doi: https://doi.org/10.1002/ett.3009.

[5] T. Svoboda and M. Hoffmann, “Problem solving by search II,” 2021. [Online]. Available: https://cw.fel.cvut.cz/b202/_media/courses/b3b33kui/prednasky/03_search_live_withnotes.pdf [Accessed: April 20, 2024]

[6] V. Vonásek, “Motion planning: sampling-based planners I.” [Online]. Available: https://cw.fel.cvut.cz/b222/_media/courses/aro/lectures/2023-planning-samplingi.pdf [Accessed: April 19, 2024]

[7] “What is Model Predictive Control? - MATLAB & Simulink,” www.mathworks.com. https://www.mathworks.com/help/mpc/gs/what-is-mpc.html

[8] M. Kairanbay and H. M. Jani, “A review and evaluations of shortest path algorithms,” ResearchGate, Jan. 2013, [Online]. Available: https://www.researchgate.net/publication/310594546_A_Review_and_Evaluations_of_Shortest_Path_Algorithms [Accessed: May 10, 2024]

[9] Z. Hanzálek, “Shortest Paths Introduction Problem Statement Negative Weights YES and Negative Cycles NO,” 2022. [Online]. Available: https://rtime.ciirc.cvut.cz/~hanzalek/KOA/SPT_e.pdf [Accessed: April 20, 2024]

[10] R. C. Coulter, “Implementation of the Pure Pursuit Path Tracking Algorithm,” 1992. Available: https://www.ri.cmu.edu/pub_files/pub3/coulter_r_craig_1992_1/coulter_r_craig_1992_1.pdf

[11] M. -W. Park, S. -W. Lee and W. -Y. Han, "Development of lateral control system for autonomous vehicle based on adaptive pure pursuit algorithm," 2014 14th International Conference on Control, Automation and Systems (ICCAS 2014), Gyeonggi-do, Korea (South), 2014, pp. 1443-1447, doi: 10.1109/ICCAS.2014.6987787.

[12] H. Wang, B. Liu, X. Ping and Q. An, "Path Tracking Control for Autonomous Vehicles Based on an Improved MPC," in IEEE Access, vol. 7, pp. 161064-161073, 2019, doi:10.1109/ACCESS.2019.2944894.

[13] F. Borrelli, A. Bemporad, M. Fodor and D. Hrovat, "An MPC/hybrid system approach to traction control," in IEEE Transactions on Control Systems Technology, vol. 14, no. 3, pp. 541-552, May 2006, doi: 10.1109/TCST.2005.860527.

[14] P. Mach and Z. Becvar, "Mobile Edge Computing: A Survey on Architecture and Computation Offloading," in IEEE Communications Surveys & Tutorials, vol. 19, no. 3, pp. 1628-1656, thirdquarter 2017, doi: 10.1109/COMST.2017.2682318.

[15] A. Ndikumana, K. K. Nguyen and M. Cheriet, "Age of Processing-Based Data Offloading for Autonomous Vehicles in MultiRATs Open RAN," in IEEE Transactions on Intelligent Transportation Systems, vol. 23, no. 11, pp. 21450-21464, Nov. 2022, doi: 10.1109/TITS.2022.3192098.

[16] J. Dolezal, Z. Becvar and T. Zeman, "Performance evaluation of computation offloading from mobile device to the edge of mobile network," 2016 IEEE Conference on Standards for Communications and Networking (CSCN), Berlin, Germany, 2016, pp. 1-7, doi: 10.1109/CSCN.2016.7785153.

[17] A. Sultan, “5G System Overview,” www.3gpp.org, Aug. 08, 2022. https://www.3gpp.org/technologies/5g-system-overview

[18] Gigabyte.com, 2024. https://www.gigabyte.com/FileUpload/Global/ModelPlugin/ModelSectionChildItem/469/469.png?_=33e442d4d68381a6462ed0e93ba6177b [Accessed: March 26, 2024].

[19] “MODEL 88096-4 owner’s manual MERCEDES-BENZ G 63 AMG 6X6.” [Online]. Available: https://traxxas.com/sites/default/files/88096-4-OM-EN-R01.pdf [Accessed: March 28, 2024].

[20] “VIKING SMARTECH II QC3.0 40000mAh User manual,” [Online]. Available: https://www.best-power.cz/user/related_files/user_manual_smartech_ii_en_cz_sk_de_hu-1.pdf [Accessed: March 28, 2024].

[21] “RPLIDAR ROS package,” GitHub, Jul. 16, 2022. https://github.com/Slamtec/rplidar_ros.

[22] “RPLIDAR A1.” [Online]. Available: https://bucket-download.slamtec.com/d1e428e7efbdcd65a8ea111061794fb8d4ccd3a0/LD108_SLAMTEC_rplidar_datasheet_A1M8_v3.0_en.pdf [Accessed: April 1, 2024].

[23] “RM500Q-GL 5G HAT - Waveshare Wiki,” www.waveshare.com. https://www.waveshare.com/wiki/RM500Q-GL_5G_HAT [Accessed: April 4, 2024].

[24] “Autonomous Driving” GitLab. https://gitlab.fel.cvut.cz/mobile-and-wireless/autonomous-driving [Accessed: May 10, 2024].

[25] S. Macenski and I. Jambrecic, “SLAM Toolbox: SLAM for the dynamic world,” Journal of Open Source Software, vol. 6, no. 61, p. 2783, May 2021, doi: https://doi.org/10.21105/joss.02783.

[26] B. K. Luknanto, “A review of 2D SLAM algorithms on ROS,” M. S. thesis, School of Industrial and Information Engineering, Milano, Italy, 2020.

[27] P. Fua and K. Lis, “Comparing Python, Go, and C++ on the N-Queens Problem,” arXiv.org, Jan. 08, 2020. https://arxiv.org/abs/2001.02491

[28] “tf: tf::TransformListener Class Reference,” docs.ros.org. https://docs.ros.org/en/diamondback/api/tf/html/c++/classtf_1_1TransformListener.html [Accessed: April 1, 2024].

[29] J. Sprinkle, “How and why to use the Ackermann steering model,” YouTube. May 24, 2016. [YouTube Video]. Available: https://www.youtube.com/watch?v=i6uBwudwA5o [Accessed: April 7, 2024].

[30] M. Berezovský and R. Mařík, “Search trees, k-d tree,” Pokročilá Algoritmizace, vol. 4, no. 33, 2012, Available: https://cw.fel.cvut.cz/old/_media/courses/a4m33pal/paska13.pdf

[31] A. Cormen, *Introduction to algorithms*. Cambridge, Mass.: Mit Press, 2003.

[32] L. G. Shapiro and G. C. Stockman, *Computer vision.* Upper Saddle River Prentice Hall, 2001.

[33] J. Vyskočil and R. Mařík, “Advanced algorithms topological ordering, minimum spanning tree, Union-Find problem,” 2012. [Online]. Available: https://cw.fel.cvut.cz/b221/_media/courses/b4m33pal/lectures/2011pal02.pdf [Accessed: April 2, 2024].

[34] Carnegie-Mellon University. Robotics Institute and R. Craig Coulter, “Implementation of the Pure Pursuit Path Tracking Algorithm, ” 1992.

[35] Z. Sun, D. Hsu, T. Jiang, H. Kurniawati and J. H. Reif, "Narrow passage sampling for probabilistic roadmap planning," in IEEE Transactions on Robotics, vol. 21, no. 6, pp. 1105-1115, Dec. 2005, doi: 10.1109/TRO.2005.853485.

[36] 6Gmobile research lab, “Autonomous Driving with Offloading to MEC - Phase II,” *YouTube*, May 22, 2024. https://www.youtube.com/watch?v=aPKcAR9Qli4 [accessed May 23, 2024].

9 Apendix

9.1 The Installation of Lubuntu

Firstly, the user should prepare the following items: the Raspberry Pi 4B, an Ethernet cable, a micro HDMI cable, a keyboard, an SD card, and a computer. On the desktop, download the Ubuntu server and software for flashing SD cards, such as Rufus or Balena Etcher. Afterward, flash the Ubuntu server onto an SD card using Rufus or Balena Etcher application.

In the next step, connect all peripherals to Raspberry Pi 4B and plug in the Ethernet cable. Insert a flashed SD card into the Raspberry Pi and power on the device. Set name and password and type following commands:

sudo apt-get update
sudo apt-get install lubuntu-desktop

Finally, reboot the device by typing the command sudo reboot.

9.2 Overclocking CPU

Overclocking involves increasing the CPU clock rate to boost computational power. When the user overclocks the CPU, it tends to generate higher temperatures. For this reason, a cooling system is necessary. In the described situation, the cooling system consists of 2 fans and ribbing. The aluminium fans actively cool down CPU and RAM storage. The ribbing serves as passive cooling by enlarging the device surface to enhance heat dissipation.

When the operation system is installed, the user should paste the following lines setting up voltage and CPU frequency into the file /boot/firmware/config.txt.

over_voltage=6
arm_freq=2000

These lines specify that the CPU frequency should be 2000 MHz. The first line sets up the addition voltage to a value of 6.

9.3 Git repository

Algorithms run locally on the vehicle are located in the *autonomous-driving/trax_repo/src* folder. The ROS package is located in this folder. Below is a list of what each package contains:

- camera - running the camera on the vehicle
- control_motor - communication with Arduino and motor control
- detect_obstacle -detect dynamic obstacles
- fake_odom - launchfiles and config files
- gateway - establishing a connection to the server
- path_follower - trace the route (Pure Pursuit, MPC)
- rplidar_ros-master - launching lidar on the vehicle
- services - publisher of current position and definition of custom messages
- slam_toolbox - create and maintain map

The individual scripts can be found in the *autonomous-driving/trax_repo/src/[package]/src* folder. The *msg* folder in the package folder contains definitions of custom messages. The *config* folder in the package folder contains config files defining parameters. The *launch* folder in the package folder contains launch files that allow several ROS nodes to be launched at once.

The algorithms that are run on the server are in the autonomous-driving/MEC/ROS-Kafka/trax_repo/src folder. The structure of the packages and files is the same as in the repository intended for local execution.