JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Alexander Raab

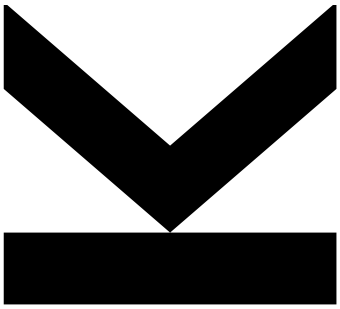
Submitted at
Institute of Robotics

Supervisor
Assoz.Univ.-Prof. DI Dr.
Hubert Gattringer

Co-Supervisor
DI Robert Fina

September 2022

Development of an experimental platform for autonomous ground vehicles



Bachelor's Thesis

for the acquisition of the academic degree

Bachelor of Science

Enrolled in the Bachelor's degree program

Mechatronics

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

Autonomous mobile robots are becoming increasingly relevant in educational, scientific and commercial settings. One software platform utilized to operate such systems is the Robot Operating System. While originally used only in research, its second generation, ROS2, is also applicable for projects outside of academic environments. For these reasons, this thesis aims to discuss the development of an autonomous ground vehicle utilizing ROS2 as its core software. In order to facilitate future research and educational applications, the setup and structure of the project focus on modularity and adaptability. The main goal is to enable the system to autonomously navigate by using an existing map, while also dynamically avoiding obstacles.

As an introduction to the robot, its kinematic model is discussed. In order to outline the implementation of additional trajectory tracking algorithms, the Stanley Lateral Controller is formulated for the system.

To better fit the application in this thesis, the central computer of a preexisting TurtleBot is exchanged and a new camera system is installed. By modifying the controller interfacing complementary sensors, the direct incorporation of embedded devices into a ROS2 system using the micro-ROS stack, is demonstrated.

For virtual testing, a ROS2 network is configured for the simulation using Gazebo in addition to the setup of the physical TurtleBot. To enable the navigation capabilities, this thesis relies on the Navigation 2 stack. The aforementioned path tracking controller is implemented using a Jupyter notebook and custom interfaces, in order to demonstrate the incorporation of external software platforms into the ROS2 framework.

By testing the simulation and by operating the physical robot in various different use cases, the functionalities of the setup are verified. With this it is proven, that the system is able to perform all desired tasks with sufficient reliability.

Kurzfassung

Autonome mobile Roboter gewinnen für Lehre, Forschung und Wirtschaft zunehmend an Relevanz. Eine Software-Lösung zum Betrieb solcher Systeme, ist das Robot Operating System. Während es ursprünglich nur in wissenschaftlichen Projekten eingesetzt wurde, ist seine zweite Generation, ROS2, für einen wesentlich breiteren Anwendungsbereich geeignet. Aus diesen Gründen beschreibt diese Arbeit die Entwicklung eines autonomen Fahrzeugs mit Hilfe von ROS2. Beim Aufbau des Projekts liegt der Fokus auf einer modularen, anpassbaren Struktur, um eine Weiterentwicklung in zukünftigen Lehr- und Forschungsprojekten zu erleichtern. Ziel ist es, ein bestehendes Robotersystem zu adaptieren und diesem zu ermöglichen, mit einer vorab bekannten Karte autonom zu navigieren. Gleichzeitig soll Hindernissen dynamisch ausgewichen werden.

Zur Einführung in das verwendete Robotersystem wird dessen kinematisches Modell beschrieben. Um in weiterer Folge die Implementierung zusätzlicher Regelungsstrategien zu diskutieren, wird ebenso ein Stanley Querdynamik Regler für den Roboter formuliert.

Die vorhandene Hardware wird für den Einsatz in dieser Arbeit angepasst und um ein neues Kamerasystem erweitert. Damit die direkte Einbindung von Mikrocontrollern mittels des micro-ROS Stacks gezeigt werden kann, wird die Ansteuerung der unterstützenden Sensorik ebenso erneuert.

Um den Roboter virtuell zu testen, wird zusätzlich zur Einrichtung des realen Systems auch eine Simulation mit Gazebo aufgebaut. Zur Implementierung der Navigation, wird auf den Navigation 2 Stack zurückgegriffen, welcher um den zuvor erwähnten Bahn-Regler erweitert wird. Um die Einbindung externer Software Plattformen in ROS2 Systeme zu demonstrieren, nutzt dieser ein Jupyter-Notebook und eine benutzerdefinierte Schnittstelle.

Simulation und physisches System werden in verschiedenen Anwendungsfällen getestet, um die Funktionsfähigkeit zu überprüfen. So zeigt sich, dass das System die gewünschten Anforderungen mit hinreichender Verlässlichkeit erfüllen kann.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years the field of mobile robotics is increasing in relevance due to its wide range of applications. Construction, logistics, reconnaissance or exploration are some examples of possible use cases. For many tasks, these robotic systems work autonomously, without a human operator in direct control. In such cases, robots need to be able to perceive their environments and compute the appropriate actions by themselves. Autonomous ground vehicles (AGVs), especially wheeled ones, are one of the most prevalent sub-categories of mobile robots. Amongst others, the advancements of self-driving cars draw scientific and commercial interest to these systems. For this reason, a vast number of different AGVs is available for all kinds of applications [13].

Due to the growing interest in wheeled mobile robots, they are often times used in research and education. From multi-wheeled all-terrain rovers, over car-like robots with four wheels to more simple Differential Drive platforms, systems with various wheel configurations exist [2]. Two-wheeled AGVs are a common choice for research projects or in educational settings, as these systems are typically less expensive in comparison to more complex robots. Their simple kinematics make it possible to focus on other aspects of mobile robotics like data acquisition or navigation [9].

The abundance of different robotic systems also entails a wide variety of software platforms, often limited to specific robotic systems. One framework aiming to support many different robots and use cases is the Robot Operating System (ROS). It is utilized for the setup of robotics applications and relies on community contributed packages as well as a modular structure. Its second generation, ROS2, improves on network security and other shortcomings of the first one in addition to greatly expanding the possible use cases. While originally used only for scientific applications, the development of ROS2 makes the open-source software suitable for systems outside of research [8].

For the aforementioned reasons, this thesis aims to develop an autonomous Differential Drive robot as a foundation for future research and educational applications. Similar robots are discussed in [9] and [7]. The system is based on the original TurtleBot 1 and relies on ROS2 to provide an adaptable software structure. All main functionalities are implemented using preexisting packages like the Navigation 2 (Nav2) stack. To

demonstrate the combination of ROS2 with external tools, the project incorporates Jupyter in its setup. This web based software platform is popular in different scientific fields like data science and is also becoming more relevant in robotics. By including tools for visualization, monitoring and user interaction in platform independent files, it provides convenient functionalities for robotics education and research [4].

As an introduction to the robot system, Chapter 2 discusses the kinematic model of the TurtleBot. Then the Stanley Lateral Controller is formulated for the Differential Drive model, in order to later implement this trajectory tracking algorithm for demonstrational purposes. Regarding the hardware of the TurtleBot, Chapter 3 describes the mobile base platform, the central computer and the environmental perception sensors as well as their respective adaptations in detail. Before covering the software setup for this project, the properties, capabilities and the general structure of ROS2 are outlined in Chapter 4. The micro-ROS stack for direct incorporation of embedded devices is also covered. The actual setup for the TurtleBot and its simulation with Gazebo are then described in Chapter 5. Here, this thesis discusses the structure and startup as well as all individual components of the ROS2 network. This chapter also includes the implementation of the Stanley Lateral Controller. Lastly, the results of this project and an outlook for further adaptations are described in Chapter 6.

Chapter 2

Robot Model

As an introduction to the robot discussed in this thesis, it is necessary to examine its mathematical model. The TurtleBot uses a two-wheeled vacuum robot as its base. The wheels are actuated by individual DC-motors and are positioned on both sides of the body, so that their rotational axis are coincident. Due to this configuration, the instantaneous centre of rotation (ICR) is bound to this axis. This model is commonly refereed to as a Differential Drive robot.

2.1 Kinematic Model

The state of the robot is defined by three minimal coordinates $\mathbf{q}$. In a Cartesian space these are the x- and the y-position of the body, as well as the orientation γ along the ${}_I z$-axis, see figure 2.1.

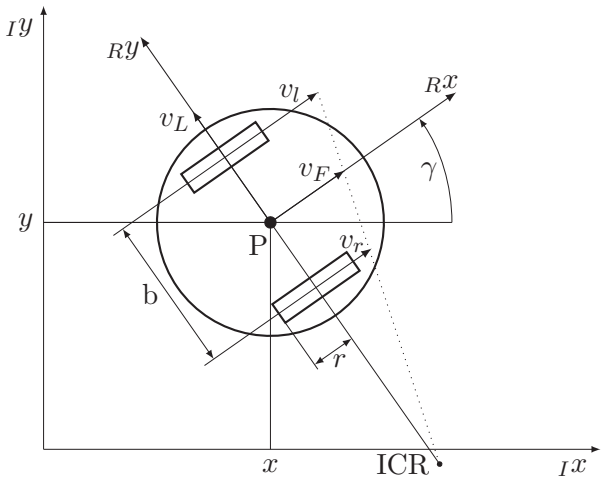


Figure 2.1: Kinematic model of a Differential Drive robot.

The choice of the minimal velocities $\dot{\mathbf{s}}$ however is ambiguous. For higher level control design such as path tracking, the linear velocity v_F and angular velocity $\dot{\gamma}$ of the robot body are typically used. This leads to

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\gamma} \end{pmatrix} = \begin{bmatrix} \cos\gamma & 0 \\ \sin\gamma & 0 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} v_F \\ \dot{\gamma} \end{pmatrix} \tag{2.1}$$

$$\mathbf{q}^T = (x, y, \gamma) \tag{2.2}$$

$$\dot{\mathbf{s}}^T = (v_F, \dot{\gamma}) \tag{2.3}$$

as the kinematic model. The non-holonomic constraint

$$v_L = -\dot{x}\sin\gamma + \dot{y}\cos\gamma = 0 \tag{2.4}$$

represents that cross slip is neglected.

Alternatively the angular wheel velocities $\dot{\beta}_l$ and $\dot{\beta}_r$ can also be used for the model description. Factoring in the relation between these and the tangential velocities of the wheels

$$v_l = r\ \dot{\beta}_l \tag{2.5}$$

$$v_r = r\ \dot{\beta}_r, \tag{2.6}$$

the transformation between both representations can be written as

$$\begin{pmatrix} \dot{\beta}_l \\ \dot{\beta}_r \end{pmatrix} = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ \frac{r}{b} & -\frac{r}{b} \end{bmatrix}^{-1} \begin{pmatrix} v_F \\ \dot{\gamma} \end{pmatrix}, \tag{2.7}$$

containing the wheel radius r and the wheel track b.

Due to the typically low operation speeds with Differential Drive robots, kinematic controllers often suffice. For real world application they are however commonly combined with dynamic motor controllers, where these wheel transformations are necessary [5].

The base robot of the TurtleBot also follows this scheme. It is controlled using only linear and angular velocity commands for the robot body and handles the motor control and the wheel transformations internally. As these processes can not be influenced, this thesis will focus on higher level control tasks, mainly path tracking.

2.2 Trajectory Tracking Control

One important aspects of robot control is following a pre-defined path. For mobile robots these are often globally computed and locally adjusted to avoid obstacles. As path tracking plays such a significant role in robotics, there are many different approaches for controlling the velocity and the steering of autonomous robots.

Geometric controllers, like the Pure Pursuit [1] approach, use only the relation between the current position and the trajectory to control the robot. Expending on these, kinematic controllers also incorporate the kinematic model in the control law. More complex approaches include dynamic, optimal, adaptive or model based strategies [1].

Stanley Lateral Controller

The ROS2 navigation stack Nav2, which is discussed at a later point in this thesis, already includes different types of pre-built path tracking controllers. To demonstrate how to implement additional control strategies into the existing framework, the Stanley Controller was chosen.

The Stanley Lateral Controller [14] is a geometric trajectory tracking approach named after the autonomous vehicle which won the DARPA Grand Challenge 2005. The control law was originally formulated for an Ackerman steering configuration, which can be simplified to a Bicycle model. To use the controller with the TurtleBot, it needs to be adjusted to the Differential Drive model.

Error Formulation

To formulate the error ${}_T\mathbf{e} = (e_x, e_y)^T$ with $e_x = 0$ between the target trajectory (x_T, y_T) and the current position (x, y), the centre of the robot is used as a reference point. The path can be approximated with a straight line in order to simplify the problem, leading to the transformation matrix

$$\mathbf{R}_{TI} = \begin{bmatrix} \cos\gamma_T & \sin\gamma_T \\ -\sin\gamma_T & \cos\gamma_T \end{bmatrix} := const. \; . \tag{2.8}$$

The target orientation γ_T is assumed to be constant. For low drive velocities, the changes in the transformation matrix are small, making this approximation valid. As shown in figure 2.2, the error between the robot position and the target trajectory can be written as

$${}_T\mathbf{e} = \mathbf{R}_{TI} \; {}_I\mathbf{e} = \mathbf{R}_{TI} \begin{pmatrix} x_T - x \\ y_T - y \end{pmatrix}. \tag{2.9}$$

With $\mathbf{R}_{TI} = const.$ (2.8), the time derivative of the positional error yields

$${}_T\dot{\mathbf{e}} = \mathbf{R}_{TI} \begin{pmatrix} \dot{x}_T \\ \dot{y}_T \end{pmatrix} - \mathbf{R}_{TI} \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix}. \tag{2.10}$$

Using the kinematic model (2.1), this equation can be simplified to

$${}_T\dot{\mathbf{e}} = \mathbf{R}_{TI} \begin{pmatrix} v_T \\ 0 \end{pmatrix} + \begin{pmatrix} -\cos e_\gamma \\ \sin e_\gamma \end{pmatrix} v_F. \tag{2.11}$$

The variable v_T represents the desired velocity in ${}_Tx$ direction and

$$e_\gamma = \gamma_T - \gamma \tag{2.12}$$

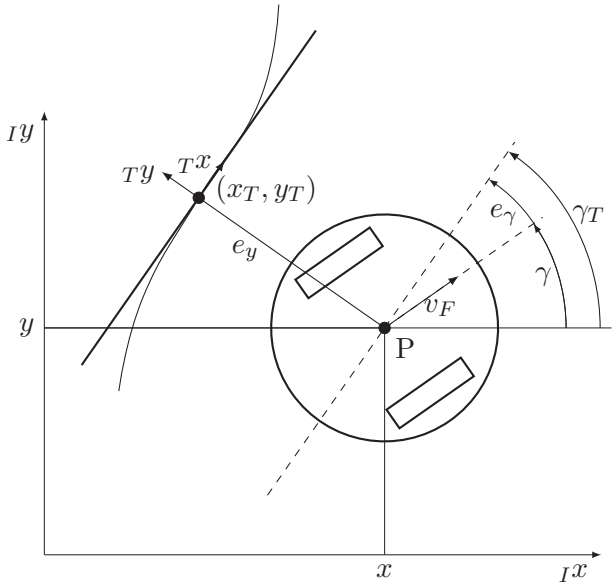


Figure 2.2: Error definitions for the Stanley Controller.
e_y: lateral error, e_γ: orientation error

the orientation error. Due to the approximation of a constant trajectory orientation γ_T, the angular velocity of the robot is also simplified to

$$\dot{\gamma} = \dot{\gamma}_T - \dot{e}_\gamma = -\dot{e}_\gamma. \tag{2.13}$$

With (2.11) and (2.13), the time derivatives of the lateral error

$$\dot{e}_y = v_F \ \sin e_\gamma \tag{2.14}$$

and of the rotational error

$$\dot{e}_\gamma = -\dot{\gamma} \tag{2.15}$$

can be explicitly stated.

Control Law

As mentioned previously, the Stanley Controller was originally designed for vehicles with an Ackerman steering. These models can be simplified to the Bicycle model illustrated in figure 2.3 by combining the front and back wheels respectively. The position P of the front wheel is chosen as a reference point.

The control law of the Stanley Controller is formulated as

$$\gamma_F = \begin{cases} e_\gamma + \arctan\left(\frac{k\ e_y}{v_F}\right) & \text{if } |\gamma_F| < \gamma_{F,max} \\ \gamma_{F,max} & \text{if } \gamma_F \geq \gamma_{F,max} \\ -\gamma_{F,max} & \text{if } \gamma_F \leq -\gamma_{F,max} \end{cases} \tag{2.16}$$

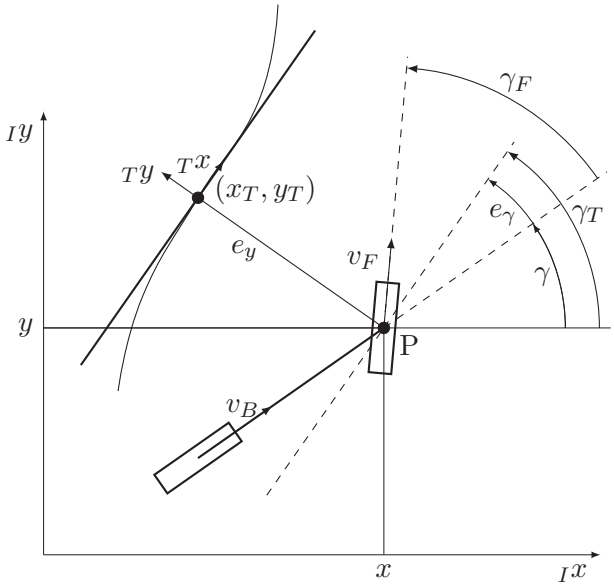


Figure 2.3: Definitions for the Stanley Controller of a Bicycle model.

yielding a steering angle γ_F depending on the lateral error e_y and the orientation error e_γ. It can be demonstrated that the controller is asymptotically stable for a front wheel velocity $v_F > 0$ and a maximum steering angle $0 < \gamma_{F,max} < \pi/2$ [6].

Applying the control law to the Differential Drive model, the steering direction and the current heading angle γ are equal. Here the Stanley Controller yields a new heading direction γ_F for the robot. The actual implementation of the controller and its integration into the Nav2 stack is discussed in Section 5.9 of the thesis.

Chapter 3

Hardware

The robot system used is based on the original TurtleBot 1, see figure 3.1. An adapted iRobot vacuum cleaner forms the mobile base platform. Built on top, there are multiple layers housing the power supply system and a central computer, along with various sensors. This thesis builds upon a preexisting robot, which was developed over the course of different preceding projects. In its former application it was operated as an office assistant using ROS1. To update the system to the latest generation of the Robot Operating System and to make the robot applicable for future projects, many components needed to be adapted or exchanged.

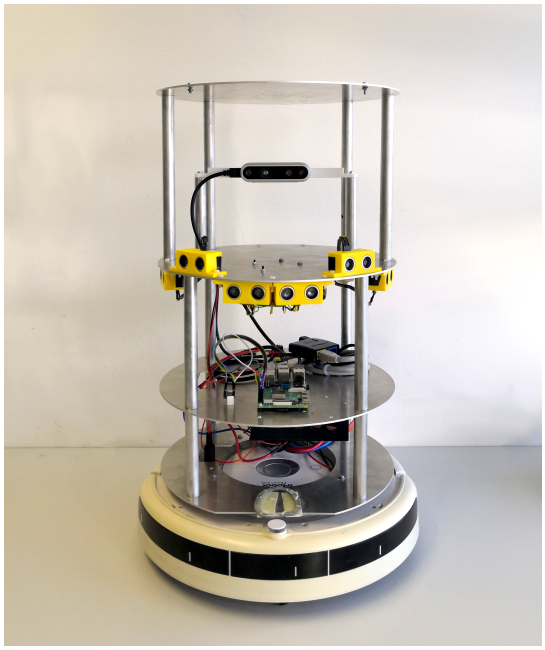


Figure 3.1: Depiction of the robot system.

3.1 Mobile Platform

As mentioned above, an iRobot Roomba 531 is used as the mobile robot base. Due to their well documented, freely accessible serial command interface (SCI) in addition to the various integrated sensors, the Roomba models are a good fit for this application. Via the Mini-DIN port on the top side, depicted in figure 3.2, a connection to the robot can be easily established. Over this interface, it is not only possible to control the Roomba with velocity commands, but also to gather data from its sensors. This includes odometry data from wheel encoders or signals from edge detectors and bumpers.

In previous works, multiple adjustments to the original vacuum robot itself were made. The built-in nickel-metal hybrid battery of the Roomba was replaced with a central lithium-polymer accumulator, to supply the whole robot system from one common source. Additionally, the vacuum unit was disabled to conserve energy and the floor brushes were removed in order to reduce friction.

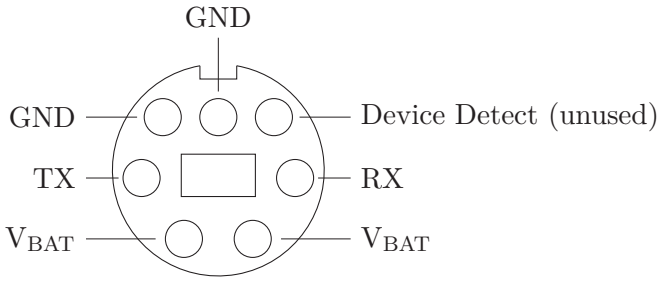


Figure 3.2: Mini-DIN port of the iRobot Roomba 531.

3.2 Central Computer

A Raspberry Pi (RPi) 4B single board computer with four gigabytes of RAM serves as the central computing unit of the robot. It was selected due to its compact design and numerous interfacing options. A network connection can be established via a built-in WLAN module or over an Ethernet socket. The RPi also supports up to four USB devices. However, its biggest advantage in comparison to conventional computers are the integrated GPIO ports. They are accessible over 40 header pins on the top side of the board and can be used to directly connect different peripherals like sensor and actuators. These ports also provide communication interfaces like SPI, I2C and UART. Lastly the RPi can also be powered over these connectors. In case of this project, the voltage is provided by a mini power supply connected to the battery.

Operating System

Due to its small size, the computing power of the RPi is rather limited. To minimize the impact of the operating system (OS), it is setup with Ubuntu Server 20.04 LTS. This specific Linux version is compatible with all, over the course of this thesis actively supported, ROS2 distributions and does not provide a graphical user interface

(GUI). On one hand, this conserves system resources for other computational tasks while operating the robot. On the other hand, it is intended to outsource simulation, visualization and remote control to external computers anyhow, because these types of processes typically require a dedicated graphics card.

Network

As indicated previously, other devices need to be able to communicate with the RPi remotely. It is generally possible to configure the single board computer as a wireless access point. As this would again expend important processing power, an external Edimax BR-6258n nano router is utilized. This adapter is powered by the RPi using an USB cable and also connects to the computer via LAN connection. The router provides a wireless network which enables external devices to communicate with the robot.

Raspberry Pi-Roomba Connection

The mobile base is also connected directly to the central computer. For this, one of the formerly mentioned GPIO-UART ports is used. These however work with logic levels of 3.3 V, in contrast to the 5 V signals of the Roomba's interface. To connect these devices, the bidirectional Adafruit TXB0104 logic level converter is utilized. The wiring scheme is shown in figure 3.3.

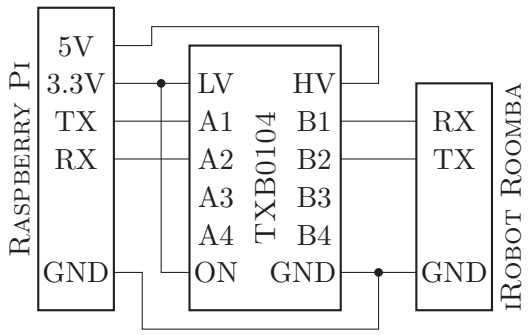


Figure 3.3: Connection scheme of the Adafruit TXB0104.

3.3 Depth Camera

As its main sensory component for environmental perception, the robot is equipped with an Intel RealSense D435 camera. In addition to capturing ordinary colour images, this model uses an integrated stereo camera for depth estimation. The D435 version is selected due to its ideal operating range of 0.3 m to 3 m. The robot will be used solely indoors, mainly in office and laboratory environments, where these distances are most common. Additionally the manufacturer provides a software library and working ROS2 packages, which greatly simplify the integration in this project.

The camera is mounted in the centre of the uppermost layer to utilize its full field of view. It is connected to the RPi via USB cable, enabling data transfer while also powering the device.

3.4 Ultrasonic Distance Sensors

The third tower layer additionally houses seven HC-SR04 ultrasonic range sensors, which support the camera with complementary depth measurements. Whilst the RealSense D435 performs well for larger items and walls, it has difficulties to correctly detect thin objects like railing bars or translucent surfaces such as windows. In these cases, the ultrasonic sensor are used to avoid collisions.

The HC-SR04 uses time of flight measurements to perceive items inside its operation range of 4 m within a detection angle of 15°. Due to the measurement principle, it is possible that the sensor echoes interfere with each other when using all units at once. In order to avoid these unwanted interactions, only one sensor after another is activated, following the sequence illustrated in figure 3.4. This is possible, because the range sensors are only used to support the camera system, making a lower sample rate a viable option.

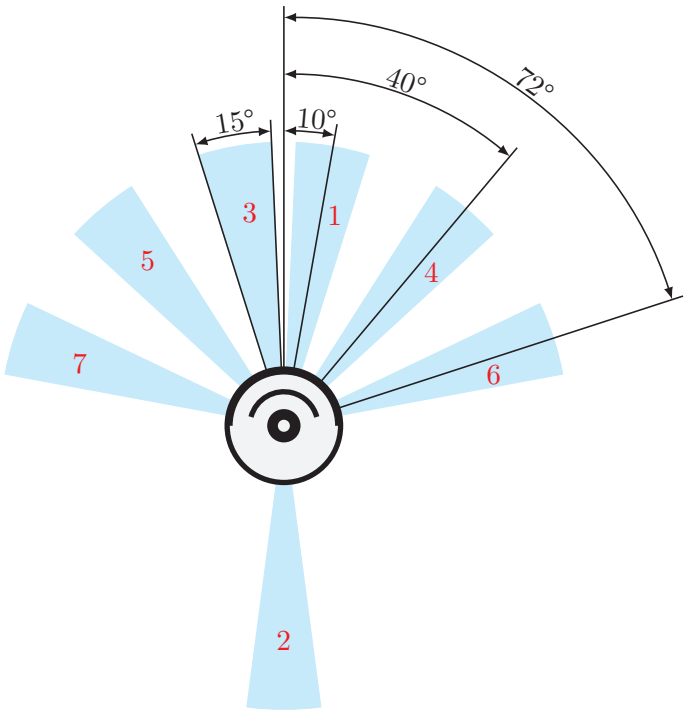


Figure 3.4: Placement and activation sequence of the ultrasonic range sensors.

The sensors are controlled with a Raspberry Pi Pico microcontroller. It is selected due to its compatibility with the micro-ROS framework, which enables embedded systems to directly exchange data with ROS2 networks. Like the RPi, the RPi Pico uses 3.3 V signals, while the sensors need 5 V to work reliably. To process the

sensor measurements, simple 2 : 3 voltage dividers are used to adjust the outputs to the correct signal levels. The wiring schematic for a single detector is illustrated in figure 3.5. The microcontroller is connected to the main computer with a USB cable, which supplies power and also allows data transfer using UART.

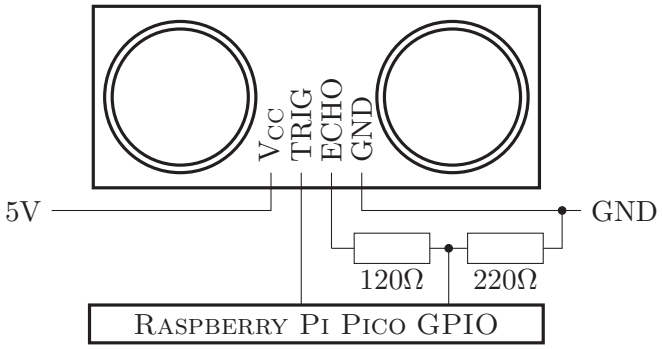


Figure 3.5: Wiring schematic of a single HC-SR04.

Chapter 4

The Robot Operating System

The Robot Operating System is a software platform designed for the development of robotics applications. The first version was originally invented for the Willow Garage PR2 robot and later adapted to support a wide variety of systems. Due to its development history, it is mainly used in research as it lacks vital features, necessary for many 'real world' or commercial applications, like real-time capabilities, network security or a standardized support of embedded systems. To make ROS applicable for a wider range of use cases, the second generation, ROS2, was designed with these shortcomings in mind.

4.1 ROS2

The new version of ROS focusses on modularity. The software is open source and relies on community contributed packages. Individual components, like device drivers and control systems, are independent from each other, making it possible to focus development on specific parts and building the rest of a robotic system from pre-existing ones. To support the modular approach, ROS2 works asynchronously. With its event-based system, multiple physical devices can be combined without the need of task synchronization. As a result of these design choices, ROS2 features multiple abstraction layers illustrated in figure 4.1. Different programming languages, mainly C++ and Python, are supported with client libraries. These depend on a common intermediate interface, the ROS2 Client Support Library (rcl), which in turn interacts with different ROS MiddleWare (rmw) implementations used for communication interfaces.

Notable changes

The differences between the two generations of ROS are numerous and it would go beyond the scope of this thesis to discuss them in detail. It is however necessary to highlight some of the most important design aspects and their relevance for the framework.

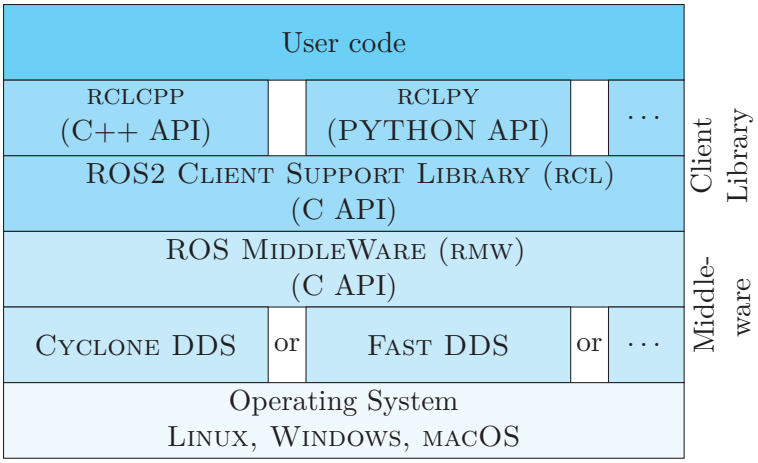


Figure 4.1: Abstraction layers in ROS2.

One of these are real-time capabilities significant for time and safety critical processes in systems like self-driving vehicles. For this purpose, ROS2 supports different real-time computing APIs. Another aspect is the integration of embedded systems into robotic applications. Many sensors, actuators and peripherals are based on microcontrollers (MCUs) and while there are experimental packages available for ROS1, the new generation offers a commercially supported solution, micro-ROS, for these kinds of tasks.

Extensive changes to the network architecture were also made. As shown in figure 4.1, ROS2 uses Data Distribution Service (DDS) implementations for data transport. This middleware choice enables many of the core aspects of ROS2 like peer-to-peer discovery of network participants without the need of a central name server, like the ROS Master in ROS1, or the aforementioned real-time capabilities. DDS uses UDP for data delivery and also introduces quality of service (QoS) settings which determine the reliability of data transmissions. In this way, ROS2 can deal with lossy and unstable connections encountered for example in wireless communication. Furthermore the middleware provides network security standards for message and participant identification, access control and data encryption. Together with additional security tools, these properties are essential for many robotics applications [8].

Naturally, these design choices also affect the network configuration. In ROS1, a single ROS Master with a clearly defined Uniform Resource Identifier (URI), is needed in order to manage all nodes of a ROS system. These also rely on URIs consisting of a hostname and a port. For setups using multiple machines inside the same network, each device needs an unique ROS-hostname or -IP to resolve connections. These settings are configured using environmental variables. Communication with devices in external networks is typically established using virtual private networks (VPNs), as ports are randomly assigned to nodes making direct connections difficult.

Since ROS2 relies on DDS for communication, these settings are no longer needed. Nodes can discover each other by default without the need of a ROS Master, even when distributed to different machines in the same network. An environmental variable,

the ROS Domain ID, identifies a ROS2 network. Only participants using the same ID can communicate, making it possible for multiple ROS2 systems to run in the same network. For connections with external networks, VPNs are still a viable option. With additional DDS configuration it is however possible, to expose specific ports for discovery, enabling direct transmission for example through firewalls and over the internet.

Data Distribution Service

While the previous paragraphs describe the relevance for ROS2, applications and capabilities of DDS are far more extensive. In short, DDS is a middleware protocol and API standard defined by the Object Management Group. Independent of the programming language and the system, it is used as an abstraction layer between the OS and the application. DDS handles discovery, connection and data transfer between individual components. Due to its data-centric architecture, it supports scalable systems with high reliability and low latency, while also dynamically discovering network participants. The software standard relies on a publish-subscribe pattern for communication and manages all data storage and distribution for the components. For this reason, the middleware is sometimes regarded as a 'global data space' accessible via API, even though information is stored locally at each participant. Utilizing various filters, data exchange only occurs when necessary, saving system resources and bandwidth as well as reducing message overhead. The aforementioned QoS policies are not only used to categorize reliability of data transmission, they also cover urgency for real-time systems and message priorities [3].

With DDS being a software standard, different implementations are available. While Eclipse Cyclone DDS is the default option of ROS2 Galactic, the second generation of ROS also supports other vendors listed in table 4.1 and even allows to combine them. The differences between these middleware implementations include licencing, performance, language support as well as additional features and tools. For the TurtleBot however, these aspects are currently not relevant and the default vendor is sufficient.

Implementation	Company	Key Features
Cyclone DDS	Eclipse Foundation	Open-source, high performance
Fast DDS	eProsima	Open-source, embedded support
Connext DDS	Real-Time Innovations	Commercial, feature-rich
GurumDDS	GurumNetworks	Commercial

Table 4.1: DDS vendors supported by ROS2.

4.2 Communication Patterns

While DDS handles the actual data transmission, usually hidden from the user, the communication provided by the ROS API is based on different core concepts [12].

Nodes

ROS2 networks, also referred to as ROS graphs, are organized in nodes. These elements perform specific tasks, transmit or receive data and communicate with each other. While simple processes can be handled by a single one, more complex tasks are typically executed by multiple nodes working together. This approach is consistent with the focus on modularity.

Topics

One communication pattern found in both generations of ROS are topics. Following the publish-subscribe pattern, strongly typed messages are sent to a topic by 'publishers'. 'Subscribers' listen to these named channels and receive all data which is published while they are subscribed. The data exchange is handled asynchronously, making it possible for many different participants to access the same topic as illustrated in figure 4.2.

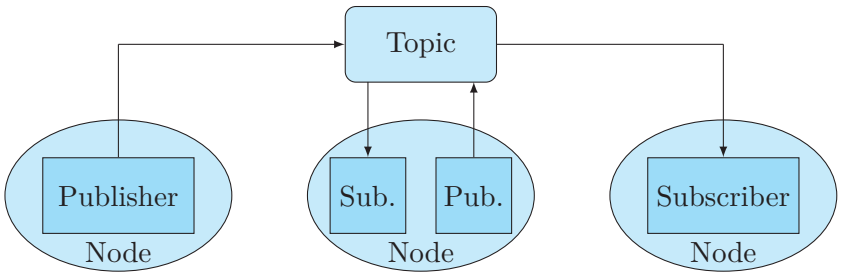


Figure 4.2: Communication via topics.

Services

In addition to the many-to-many transmission of topics, ROS also supports a call-reply style communication pattern in form of services. Like messages, these are also strongly typed and consist of a request and response part depicted in figure 4.3. Service 'clients' send requests to 'servers', which process the transmitted data and afterwards return answers. While different clients can call a service at the same time, only one server per service can exist.

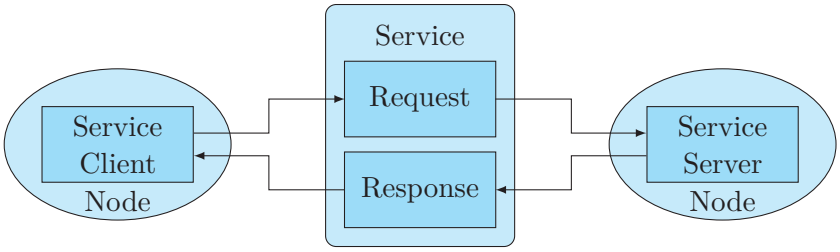


Figure 4.3: Communication via services.

Actions

Based on both of the previously discussed communication types, action are used to perform more complex, usually long running tasks. They consist of a goal and a result service as well as a feedback topic. The action client sends a goal request to the action server, which starts a process to achieve said goal and returns an acknowledgement. Typically this is followed by a result request from the client. During execution, the server publishes a periodic feedback data stream to the topic. When finished or in case the task is premature cancelled, the action server returns a result response. This sequence is illustrated in figure 4.4.

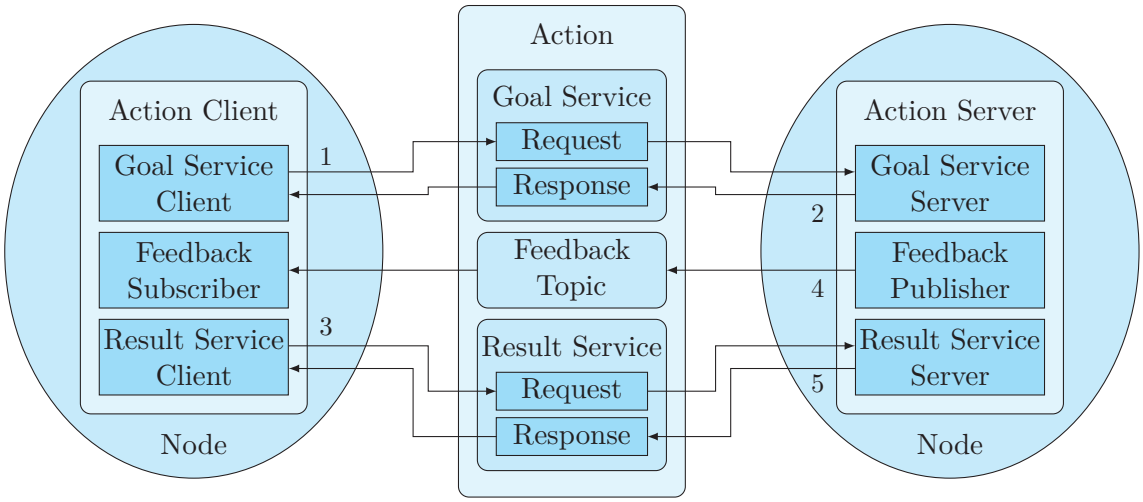


Figure 4.4: Communication via actions.

4.3 micro-ROS

As microcontrollers are essential parts of many robotic systems, the MICRO-ROS stack [10] is developed to port most of the core functions of ROS2 to work with embedded systems. The 'MICRO-ROS agent', which runs on the same device the MCU is connected to, bridges the gap between external MICRO-ROS nodes and the rest of the ROS2 network. The connection between different devices can be established for instance over ETHERNET, BLUETOOTH or via UART.

This direct integration of embedded systems is possible, because micro-ROS is built with the same architecture as ROS2, see figure 4.5. Its client library consists of the original RCL and is extended by the ROS2 Client Library package (rclc). Both APIs and the DDS For Extremely Resource Constrained Environments (DDS-XRCE), which acts as the middleware, are optimized for the use on microcontrollers. Latter also handles the connection to the main ROS2 network. Due to the need of real-time capabilities, especially in embedded systems, micro-ROS is typically used together with Real Time Operating Systems (RTOS).

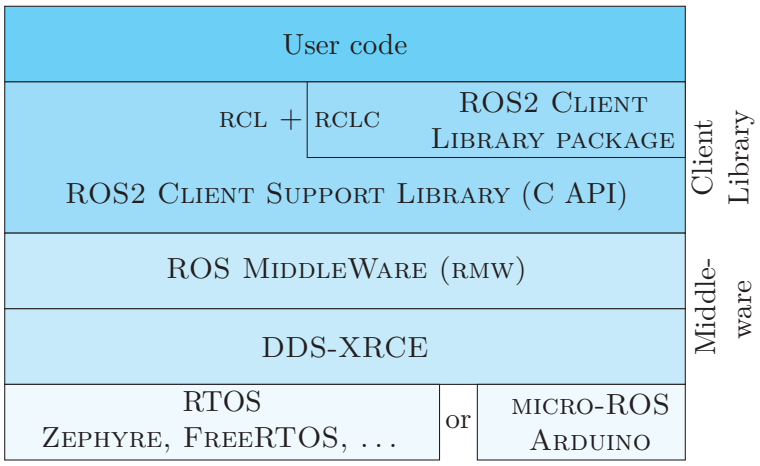


Figure 4.5: Architecture of the micro-ROS framework.

Chapter 5

Software

Setting up the ROS2 network for the TurtleBot is one of the main objectives of this thesis. Each year a new distribution is being released, often including drastic changes. With this, selecting a version for the project is a central decision. The releases alternate between Short Term Support (STS) and Long Term Support (LTS) distributions. The versions supported during the course of this thesis are listed in table 5.1. It was initially planned to start the project with the development version Rolling Ridley and switch to Humble as soon as the new distribution was released. With this release, ROS2 changes its main operating system support for Linux to Ubuntu 22.04. Some of the essential software libraries, like the drivers for the depth camera, are however currently not supported by this Linux distribution and many relevant ROS2 packages for Humble are incompatible with the older version of the OS. In the end Galactic was chosen over Foxy, because of changes in the standard DDS implementation for subsequent versions making it easier to update the project in the future.

Distribution	Release date	End Of Life date
Foxy Fitzroy (LTS)	June 5th, 2020	May 2023
Galactic Geochelone (STS)	May 23rd, 2021	November 2022
Humble Hawksbill (LTS)	May 23rd, 2022	May 2027
Rolling Ridley (development version)	-	-

Table 5.1: List of the possible ROS2 distributions.

The target functionality the ROS2 network needs to fulfill is enabling the robot to autonomously navigate in an environment using an existing map. The project is designed to contain only the basic packages necessary to complete this task, while focusing on modularity and exchangeability of individual parts. This structure not only simplifies the transition from a simulated environment to the actual robot, it also keeps the structure adaptable for future changes.

5.1 Simulation system

A standard approach in robotics is to test systems using simulators before operating the physical robots. In this thesis, Gazebo, a physics based simulation environment designed for robotic applications, is used for testing. Due to the modular project structure, the core ROS2 network remains the same for both, the simulation and the actual robot. Only specific parts, like nodes for physical sensors need to be added or replaced.

Gazebo Simulator

Currently, few robotics simulators support the direct integration of ROS2, Webots and Gazebo being two of the most prominent ones. Latter was selected, since it was already used in the preceding TurtleBot project, making parts of the simulated robot model and the virtual environment reusable.

Gazebo utilizes a physics engine for its simulations. It provides command-line tools as well as a GUI and displays a 3D rendered visualization. Multiple pre-existing plugins for sensors and actuators are available to simulate numerous different systems. The simulation environment is defined by a 'world file' in Simulation Description Format (SDF) format. Additional to objects with geometric, physical and visual properties, it also contains rendering components like cameras and lights. Robots and their individual parts are defined by Unified Robot Description Format (URDF) representations, which will be discussed in detail in Section 5.4. Both SDF and URDF are Extensible Markup Language (XML) file formats.

5.2 Structure of the ROS2 network

Before discussing the individual parts of the ROS2 network in detail, it is advantageous to outline its general structure and which devices are involved. It is intended for ROS2 and Gazebo to run on the same machine for the simulation. As discussed previously, the simulator needs a dedicated graphics card to operate smoothly, which is why the RPi is not suitable for this task. The network for the actual robot will be executed on the single board computer, some tasks are however performed on external devices.

In both cases the network largely contains the same components. As illustrated in figure 5.1, for the simulation, all elements are present on the same device. Gazebo provides simulated sensor data from its virtual environment via different plugins. These send messages to the same topics as the actual sensors interface. The virtual robot is operated in the same way as the physical one using another plugin. In addition to the components of the real system, an inertia measuring unit (IMU) is simulated for demonstration purposes.

Comparing this structure to the network of the physical TurtleBot depicted in figure 5.2, the changes mainly effect parts which interact with the robots hardware.

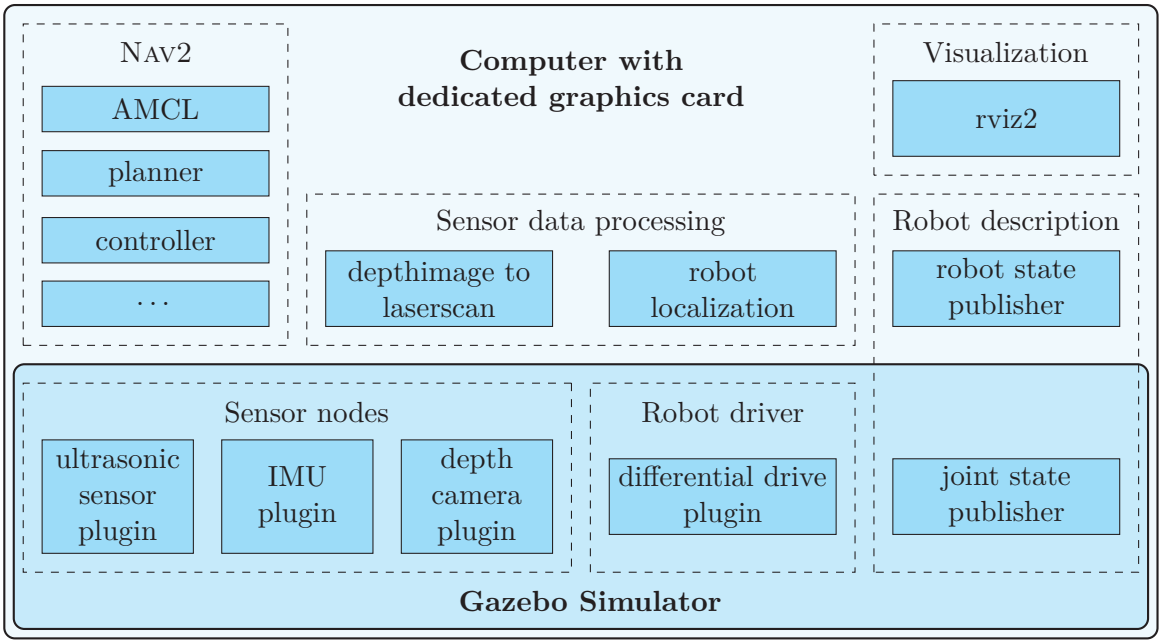


Figure 5.1: ROS2 network of the simulation.

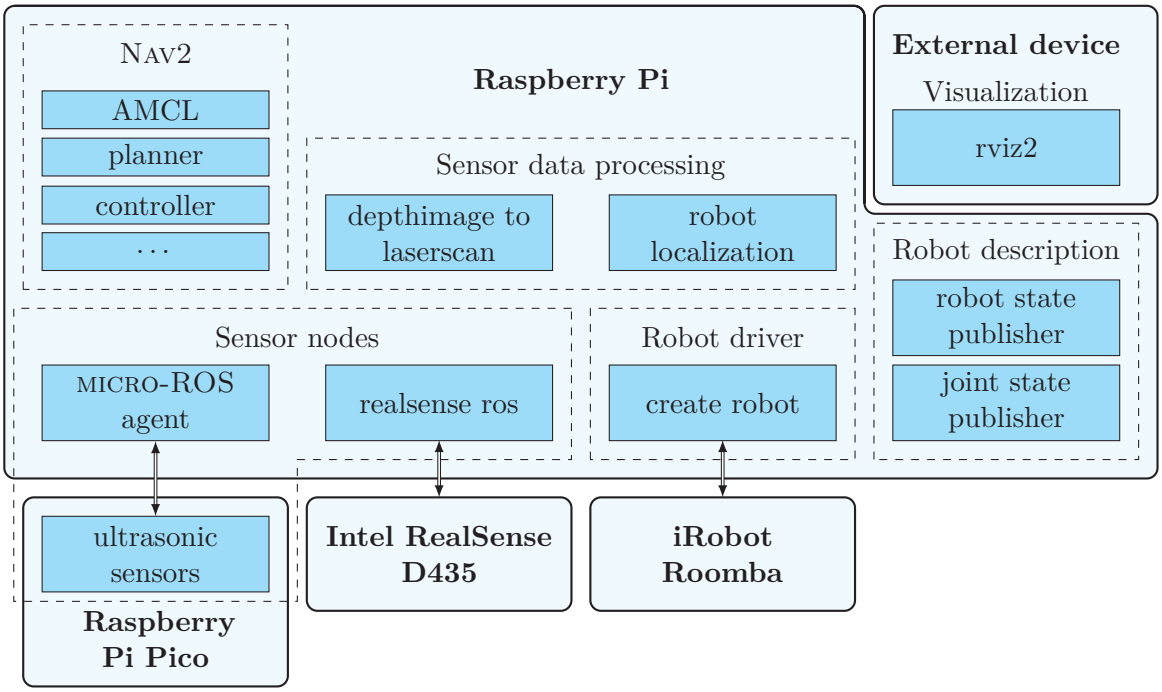


Figure 5.2: ROS2 network of the physical robot.

Nodes for communicating with sensors and the robot replace the virtual ones. These handle the data exchange with the peripherals and in case of the micro-ROS agent incorporate an external node into the ROS2 network. As described previously, the visualization is now performed on an external device connected for example wirelessly to the TurtleBot. Although this part of the system is optional, some form of remote connection to the robot is necessary to transmit control commands.

5.3 Launch files

The simplified networks illustrated in the previous section depict, that many different nodes need to be loaded when starting up a ROS2 project. Due to internal dependencies, the initialization sequences need to be exactly defined. As this becomes relevant especially for larger projects, ROS2 provides the so called 'launch system' for these tasks. The startup sequence is defined by 'launch files' written in PYTHON. Nodes together with their parameters, executables and even other launch files can be started using event based actions. More complex systems, such as the NAV2 stack, typically include external configuration files containing parameters for each node.

For this thesis, launch files for different use cases, like the startup of the simulation or of the physical robot, were compiled. These all launch the same base file, handling the whole system with all possible nodes and extension, except with different parameters. Most settings are defined in external configuration files, making the system easily adaptable. A flowchart of the startup sequence is illustrated in figure 5.3.

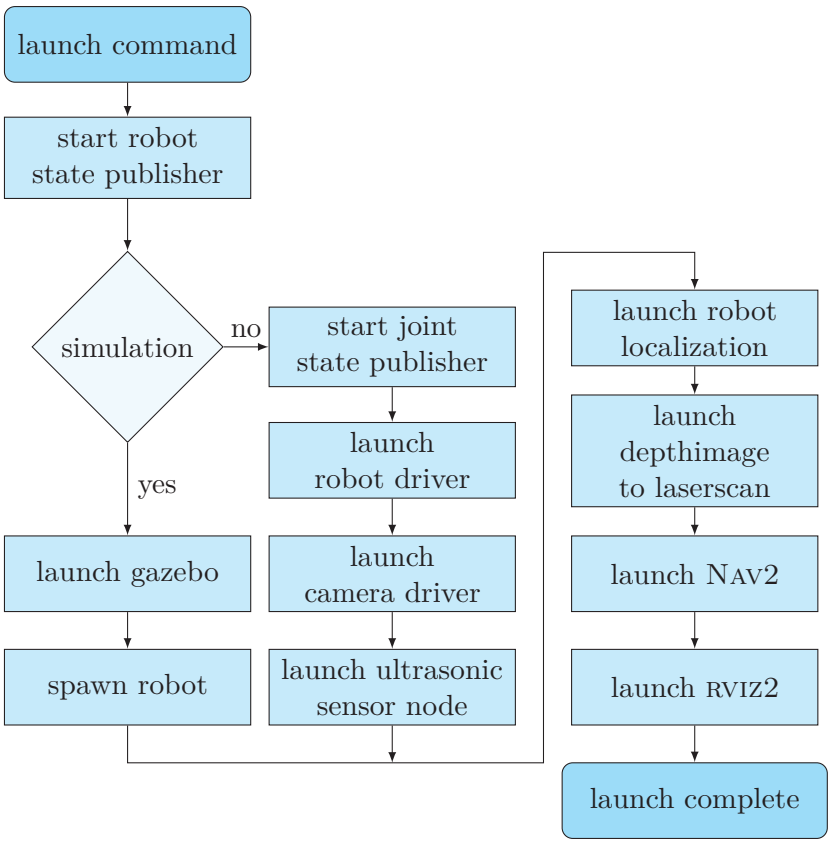


Figure 5.3: Launch sequence of the ROS2 network.

5.4 Robot Description

A fundamental aspect for many ROS2 networks is the description of the robot itself. This is not only necessary for simulation and visualization, but rather for describing positions and orientations of system components relative to each other. Operations in different 3D coordinate frames are handled by the GEOMETRY2 package. The TF2 sub-package is used to broadcast, listen to and transform these frames for all components, as there is no central server tracking this information in ROS2.

Unified Robot Description Format

For defining the geometric, visual and physical properties of robots or individual items, URDF files are used. Structured in XML format, they also specify the coordinate frames of the system. The representation of the whole robot is typically split into various smaller files for each component. In case of the TURTLEBOT, the models of the ROOMBA and the REALSENSE are contained in their respective driver package, while other parts are reused from the original project or are newly defined. The main URDF file imports these and, by specifying their relative position to a common reference point, forms the robot model.

Conveniently, GAZEBO uses the same format for its simulation. Plugins and definitions relevant for the simulator can be compiled in separate files and included in the definitions of the actual components.

Robot State Description

While transforms are broadcast and utilized by different parts of ROS2 networks, the basic description of a robot is defined by joint states and robot states. The 'joint state publisher' node is used for non-fixed joints of a system, like those found in wheels. Using the robots description, it publishes information like position and velocity of those. The 'robot state publisher' then utilizes the URDF definitions to process the current state of the robot and its coordinate frames. This structure is shown in figure 5.4.

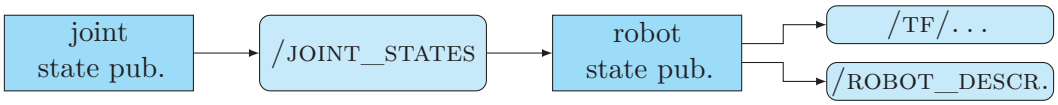


Figure 5.4: Interfaces of the robot state description nodes.

5.5 Robot Driver

One of the main parts of the ROS2 network for interacting with hardware components is the driver for the mobile base platform. It is responsible for sending velocity commands to the robot and reading and transmitting sensor data to their respective topics. The ROS2 package CREATE_ROBOT implements these functions for both, the simulation in GAZEBO and the actual IROBOT ROOMBA. It is originally designed for the IROBOT CREATE models. As these utilize the same command interface, the package can be reused by adjusting its configuration to match the robot in use.

For the simulation, the packages includes different GAZEBO plugins in the URDF file of the robot. The DIFFERENTIAL_DRIVE_CONTROLLER adds functionalities of a Differential Drive robot to the virtual one. Its settings include the geometric properties as well as velocity and acceleration limits. Additionally, it simulates wheel odometry sensors. The URDF file also utilizes plugins for various integrated sensor like bumpers and cliff detectors.

To communicate with the actual ROOMBA, CREATE_ROBOT relies on a C++ library, LIBCREATE, for handling data exchange over the robots SCI. The driver node subscribes to a control topic and publishes sensor data to the ROS2 network, the library is used to manage all data exchange with the ROOMBA, like requesting and parsing sensor data or transmitting velocity commands. The driver relies on UART communication. For this reason, the serial device and the baud rate of the robot need to be passed as parameters when launching the node.

In both cases, the plugin and the node interface the same ROS2 topics. As illustrated in figure 5.5, the velocity commands are received by listening to the CMD_VEL topic and the odometry data is published to ODOM. While the current setup does not use the other sensors or the diagnostic messages, they are still distributed to the ROS2 network. These include data from the edge detectors or status information about the driver itself.

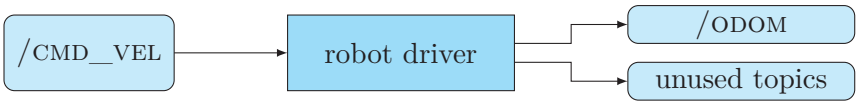


Figure 5.5: Interfaces of the robot driver.

CREATE_ROBOT is only available for ROS2 FOXY. Due to changes between the ROS versions, adjustments to the URDF file and the driver node were necessary. As the package is still in development, not all sensors of the real robot can be interfaced. This does not affect the current setup, but needs to be considered for future projects.

5.6 Sensor Nodes

In addition to the integrated sensors of the Roomba, the TurtleBot is equipped with devices for environmental perception. As described in Chapter 3, a depth camera and multiple ultrasonic range sensors are used for this purpose.

RealSense Driver

As the main component for gathering environmental data for localization, mapping and obstacle avoidance, the driver for the Intel RealSense is a vital part of the ROS2 network. It is responsible for publishing image and depth data, as well as information about the camera itself. The usage of the device in combination with ROS2 is supported by a package, realsense-ros, which is developed by the manufacturer. Like the driver for the Roomba, it also relies on a software library.

In contrast to create_robot, this package does not contain any support for simulators. While there are specific plugins for RealSense products in development, they are currently not available for ROS2 Galactic. realsense-ros however contains a URDF representation of the camera, which can be utilized as a base for modelling the functionality of the sensor using existing Gazebo plugins. By combining a simulated camera and a virtual depth sensor, the behaviour of the actual device can be replicated with sufficient accuracy. The settings of these plugins are adjusted to match the specifications of the RealSense D435 and use the same ROS2 topics the driver node interfaces.

The realsense-ros package can be used for all RealSense cameras and provides various parameters for adjusting the ROS2 integration. These include support for different messages, like depth images and point clouds, filters and calibration files. The driver node is started with a launch file and for this project is set to automatically detect the connected camera.

For the TurtleBot application, only the camera information and rectified depth images are relevant. The ROS2 interface of the node and the virtual model is shown in figure 5.6. For better organization of the network, all topics are combined under a common namespace, /camera.

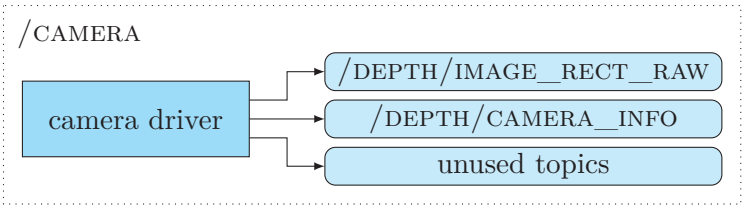


Figure 5.6: Interfaces of the camera driver.

Ultrasonic Node

Similar to the driver of the depth camera, the node responsible for the ultrasonic sensors publishes measurement data to specific topics. In this case however, the detector can not be directly incorporated into the ROS2 network using a driver package. For this reason, the setups for the simulation and the actual system display greater differences compared to the previously discussed components.

The ultrasonic sensors were already present on the existing robot, making it possible to reuse their URDF models. Originally, only the visual representations of the detector units were defined. With another default plugin specific for time-of-flight range sensors, the models were modified to simulate these measurements. Again, the settings are adjusted to fit the properties of the physical sensor, so that their behaviours match.

The actual HC-SR04 detectors are operated using the RPi Pico. Utilizing the micro-ROS framework, a ROS2 node is executed on the microcontroller. A micro-ROS agent running on the main computer connects to the peripheral system over the DDS layer using UART for data transmission. The code is written in C++ and uses an interrupt-based approach to handle periodic measurements and data publishing as displayed in figure 5.7.

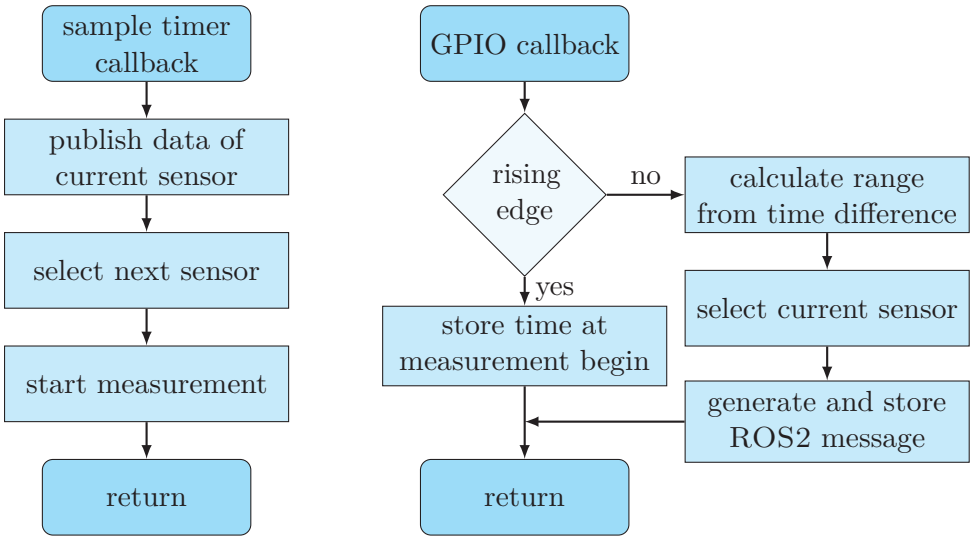


Figure 5.7: Interrupt Service Routines for the ultrasonic range measurements.

Like the camera driver, the ultrasonic node and its topics are combined under the same namespace. As figure 5.8 shows, each sensor has its own dedicated publisher and topic to distribute the measurement data.

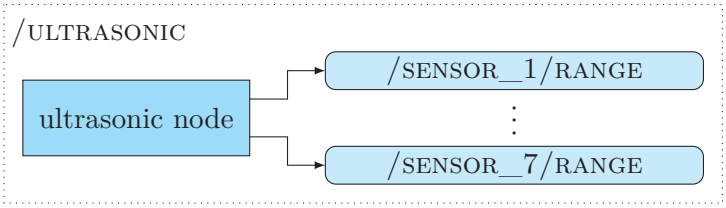


Figure 5.8: Interfaces of the ultrasonic node.

Simulated IMU

Inertia measuring units (IMUs) are commonly built into mobile robots. Their data can be utilized to improve the odometry estimation using sensor fusion algorithms. Even though the actual TurtleBot is not equipped with such a device, an IMU plugin for the simulation is included in the URDF description of the robot to demonstrate such a setup. Combining the functionality of a gyroscope and an accelerometer, it is used to simulate measurements of the angular velocity and the linear acceleration. An actual sensor would use the same interface as the virtual one depicted in figure 5.9.

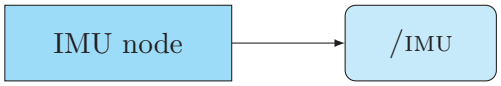


Figure 5.9: Interfaces of the IMU node.

5.7 Sensor Data Processing

Sensor measurements are converted to ROS2 messages by their respective nodes and drivers. In many cases, these then need to be processed, filtered or converted to other data types for the use in different system components. Consistent with the modular setup, these tasks are carried out by separate nodes.

Odometry Estimation

Considering the importance of accurate odometry estimation, a typical approach in mobile robotics is to utilize sensor fusion algorithms. By combining data of different sensors instead of using only a single source, the precision of the results can be improved.

For the TurtleBot, the robot_localization package provides multiple non-linear state estimation options. For this thesis, an extended Kalman filter (EKF) node is used to combine the odometry data of the base robot with the outputs of the simulated IMU. It is selected due to its lower computational effort and simpler configuration in comparison to the other alternatives.

As depicted in figure 5.10, the node uses the sensory data to compute and publish filtered odometry messages in addition to transforms between the robot and its odometry frame. The estimator is used for both, the simulation and the actual system,

even though no IMU data is available for the latter one. In this case, the filter still publishes estimated odometry data, which represents the smoothed input values.

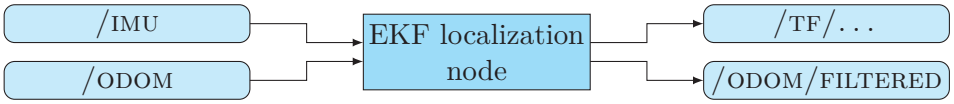


Figure 5.10: Interfaces of the extended Kalman filter node.

Depth Image Processing

Due to the low computational power of the RPi, it is practical to also adjust the image data received from the camera. While the device provides depth information in multiple message formats, these all consist of large data structures. Compared to other tasks, manipulating these consumes significant amounts of system resources. To still be able to process the depth measurements at a sufficient sample rate, they are filtered. Additionally, the localization algorithm used in this thesis also relies on messages in a different format than directly available from the camera driver. For these reasons, the DEPTHIMAGE_TO_LASERSCAN package is utilized to adjust the measurement data.

As its name suggests, it is used to convert the three dimensional depth image messages to 2D LaserScans. A configurable number of horizontal rows is sliced from the centre of the image and converted to a planar scan by selecting the minimum depth value of each column. These measurements then represent the environment approximately 45 cm above the ground. The node illustrated in figure 5.11 subscribes to the topic provided by the camera driver and publishes the conversion results to the /SCAN topic.

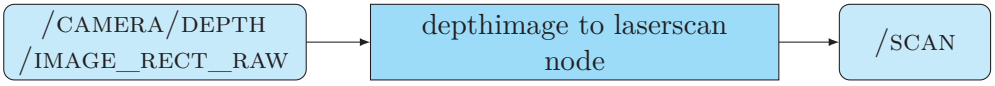


Figure 5.11: Interfaces of the depth image conversion node.

5.8 Navigation2 Stack

An essential part of the TurtleBots ROS2 network is the Navigation 2 stack [11]. As a successor of a ROS1 project, it provides tools and functionalities necessary for localization, navigation and also mapping tasks.

Since these applications typically involve complex processes, Nav2 relies heavily on actions as its main communication pattern between components. Its core functions like path planning or robot control are performed by individual modules communicating with a central behaviour tree (BT) provided by the BT Navigator. This structure is depicted in figure 5.12. The TurtleBot utilizes a default tree included in the software stack, which enables the system to navigate in dynamic environments. To implement different behaviours within the navigation modules, plugins written in

C++ are used. More complex tasks require the development of additional ones, for this project however, pre-existing plugins are largely sufficient.

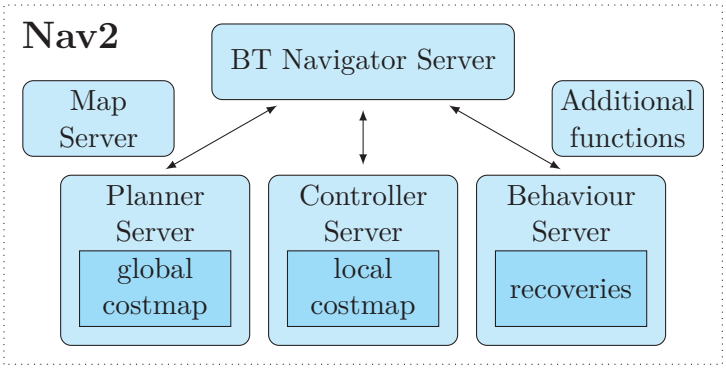


Figure 5.12: Structure of the Nav2 stack.

5.8.1 Localization

While the robot_localization package computes the transform between the robot and its odometry, the Adaptive Monte Carlo Localization (AMCL) module is used to localize the TurtleBot in the global map. The relation between these frames is shown in figure 5.13.

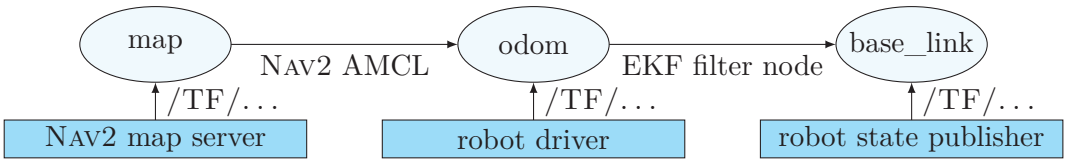


Figure 5.13: Frames and transforms relevant for the localization with AMCL.

The AMCL is based on particle filters and processes LaserScan data to estimate the robots position in a static map. It is utilized to compensate the unpreventable drift of the odometry sensors and continuously adjusts the position of the TurtleBot. In order for AMCL to yield satisfactory results, the provided static map needs to match the actual environment sufficiently well. The interfaces of the AMCL module are illustrated in figure 5.14.

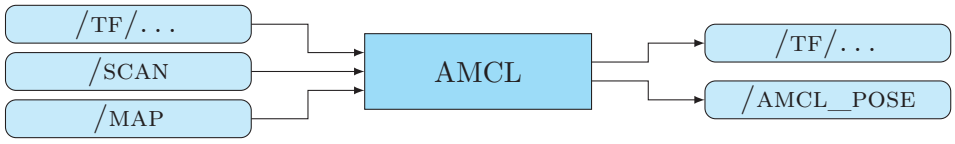


Figure 5.14: Interfaces of the AMCL module.

5.8.2 Map and Costmaps

Nearly all modules of the navigation stack require information about the surroundings in form of a 2D map, represented by an image file and a YAML description. These can be dynamically updated with simultaneous localization and mapping (SLAM) algorithms or used as a static plan. Utilized for fixed maps, the Map Server reads these files and publishes the information to the /MAP topic as shown in figure 5.15.

Additionally for path planning and collision avoidance, NAV2 relies on costmaps. These are layered on top of the global map and contain a cost value for each grid. Path planners use them to generate paths by minimizing cost functions. Collisions can be avoided for instance by penalizing occupied spaces. NAV2 supports global and local costmaps with multiple plugins each, see table 5.2.

Plugin	Functionality
Static Layer	Assigns costs to obstacles in the static map
Inflation Layer	Increases costs in the vicinity of obstacles
Obstacle Layer	Uses 2D data to mark obstacles
Voxel Layer	Uses 3D data to mark obstacles
Range Sensor	Uses range sensor data to mark obstacles

Table 5.2: Overview of the available costmap plugins.

Global Costmap

The global one typically contains static objects like walls, where the position is known. For the TURTLEBOT, the STATICLAYER plugin defines the costs of all spaces in the global map. To increase the penalty in close proximity of walls, the INFLATIONLAYER is utilized.

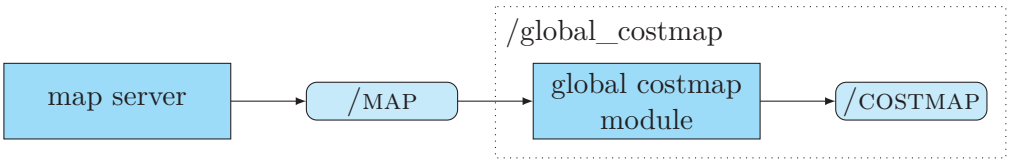


Figure 5.15: Interfaces of the map server and the global costmap modules.

Local Costmap

For dynamic environments and objects not marked in the static map, local costmaps are necessary. As their name suggests, they are not layered upon the global map, instead these costmaps are locally built around the robot and only active in its vicinity. In this project, the local costmap is updated with the converted LASERSCAN measurement inputs from the camera using the VOXELLAYER, which places markers at spaces where objects are detected. The data of the ultrasonic sensors is processed by

the RangeLayer, applying a probabilistic approach for marking occupied positions. Figure 5.16 depicts the relevant sensor topics, the local costmap module accesses.

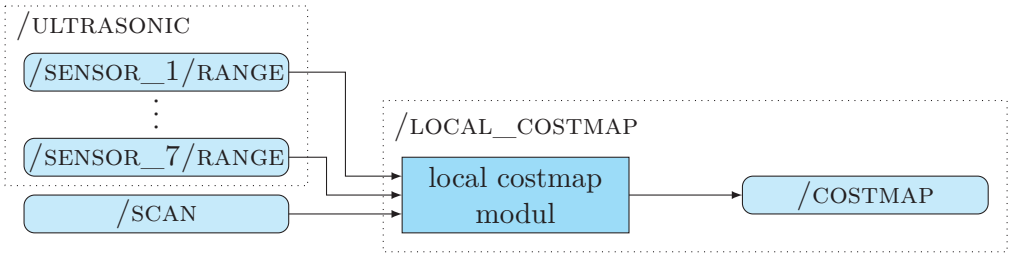


Figure 5.16: Interfaces of the local costmap module.

5.8.3 Planner

For the robot to navigate in its environment, the Planner Server provides tools for path planning. Like the previous functionalities, the actual computing algorithms are implemented as C++ plugins, providing interfaces for the server. Multiple add-ons can be defined for individual behaviours in more complex applications. The TurtleBot however only uses the planner to compute a path from its current position to a goal locations. For the standard setup, the NavFn plugin generates paths inside the map by referencing the global costmap and applying a Dijkstra algorithm. For the implementation of the Stanley Controller, another planner is utilized. The reason for this substitution is discussed in Section 5.9.

5.8.4 Controller

The computation of velocity control commands for the robot is performed by the Controller Server and its plugins. Nav2 provides several basic controllers for robots with various kinematics. Some of these include local planners, processing the local costmap to adjust the global path. As suggested in [11], the DWB controller plugin is chosen for the Differential Drive model. Based on the Dynamic Window Approach (DWA), it is the successor of the DWA controller of ROS1s navigation stack. Additionally, the server also includes so called 'Goal Checkers' to verify, if the goal position and the target orientation are reached. These plugins compare the robots location and alignment with the goal pose and notify the system, when both match in consideration of predefined tolerances.

5.8.5 Behaviours

Autonomous robots need to be able to handle uncertain conditions and system failures. For these cases, the Recovery Server is implemented. Trying to free the robot when trapped by rotating and backing up are some of the standard behaviours provided by Nav2. For complex systems, the server could notify a human operator in the event of critical errors.

5.9 Implementation of the Stanley Lateral Controller

In order to demonstrate, how additional plugins for the Nav2 stack are developed, the Stanley Controller described in Section 2.2 is implemented as an add-on for the Controller Server. Plugins are by default written in C++. In case of this thesis, the controller add-on is structured in an more complex way, involving services in a python application. This serves the purpose to illustrate how other software platforms, in this instance Jupyter, can be incorporated into the ROS2 network.

Jupyter Notebooks are browser based applications supporting multiple programming languages, including Python. They can be run on remote servers, do not need a specific OS and contain code, visualization and GUI elements as well as explanatory comments. The open-source project is gaining popularity in scientific fields like data science, machine learning and also robotics, which is why it was selected for this thesis. Additional tools for the use of Jupyter together with ROS1 are available in python packages, however the support for ROS2 is still in development.

Nav2 Controller Plugin

Nav2 plugins are written as custom classes inheriting from their respective base class, in this case `nav2_core::Controller`. These parent abstract classes provide a structure of necessary methods, like configuration or activation of the add-on, every custom one needs to implement. For the Stanley Controller plugin, these are illustrated in figure 5.17. In order to communicate with python nodes, the add-on contains a service client with a custom service definition. The method relevant for computing the control law, `computeVelocityCommands()`, utilizes this to call the service server running in a Jupyter Notebook. The plugin is built along with the other ROS2 packages and selected via a launch configuration file.

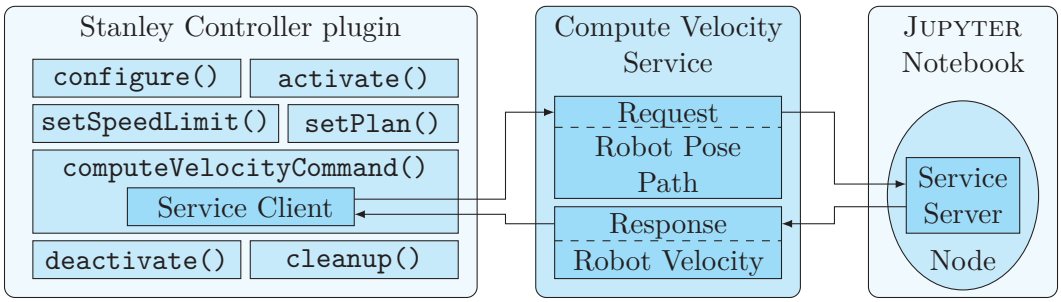


Figure 5.17: Structure of the Stanley Controller implementation.

Control Law Implementation

The service request consists of the robot position in its own coordinate frame and the target trajectory in the frame of the global map I. Before the actual computation, it is verified, that the received path contains at least two poses. To calculate the lateral error e_y needed for the control law, the robots position P is transformed to the same frame as the map. Then the closest point on the path T_1 is selected as displayed

in figure 5.18. For the orientation error, a second pose on the trajectory is needed, as the planner only supplies a set of positions without orientation. This additional point T_2 is chosen at a certain lookahead distance. The orientation error e_γ can then be approximated by substituting the trajectory with a line through both of these positions.

The coordinates of T_1 and T_2 in the frame of the global map are represented by ${}_I x_{T_i}, {}_I y_{T_i}$ with $i = 1, 2$ and ${}_I x_P, {}_I y_P$ describe the position of the robot transformed to the same coordinate system. Referring to these notations, the angle of the linearized trajectory is calculated with

$$\gamma_T = \arctan2({}_I y_{T_2} - {}_I y_{T_1}, {}_I x_{T_2} - {}_I x_{T_1}) \tag{5.1}$$

and

$$e_\gamma = \gamma_T - \gamma \tag{5.2}$$

yields the orientation error. To compute the lateral error, the absolute distance between the robot and the closest point on the path is computed with

$$|e_y| = \sqrt{({}_I x_{T_1} - {}_I x_P)^2 + ({}_I y_{T_1} - {}_I y_P)^2} \ . \tag{5.3}$$

In order to determine the sign of e_y, the orientation γ_L of the line $\overline{T_1 P}$ is assessed using

$$\gamma_L = \arctan2({}_I y_P - {}_I y_{T_1}, {}_I x_P - {}_I x_{T_1}) \ . \tag{5.4}$$

Its yaw difference to the linearized trajectory is calculated by evaluating

$$\gamma_{LT} = \gamma_T - \gamma_L \tag{5.5}$$

and normalizing the result to a value $-\pi < \gamma_{LT} \leq \pi$. The lateral error is then computed with

$$e_y = \begin{cases} |e_y| & \text{if } \gamma_{LT} > 0 \\ -|e_y| & \text{else} \end{cases} \ . \tag{5.6}$$

Lastly, the control law described in (2.16) is applied yielding the new heading angle γ_F. Using the sample rate f_s of the Controller Server and an additional smoothing factor k_1, the angular velocity can be computed with

$$\dot{\gamma}_F = f_s \ k_1 \ \gamma_F \ . \tag{5.7}$$

The robots velocities are then returned to the plugin via the service response and subsequently passed on to the robot. This computation sequence is depicted in figure 5.19. Additionally, the JUPYTER node contains a publisher distributing the current lookahead point for visualization. To handle possible errors regarding the service itself, warnings are displayed in case it becomes unavailable or when the time necessary to compute the control law exceeds the sample rate of the controller.

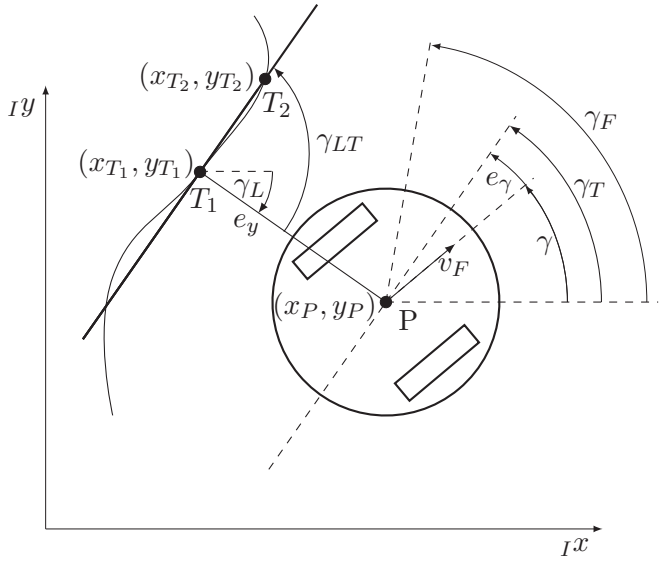


Figure 5.18: Illustration of the geometrical error definitions for implementation of the Stanley Controller.

The DWB planner described in Section 5.8.4 generates a plan toward the goal position and with only the last pose representing the target orientation. For the Stanley Controller this can cause errors, as the robot drives with a constant linear velocity. At the end of the trajectory, sudden orientation changes can not be compensated and the Goal Checker does not validate the robots position. In these situations, the TurtleBot overshoots the goal. To avoid missing the target pose, the Smac Hybrid-A* planner is utilized. Its main applications are robots with an Ackerman drive, as this plugin computes trajectories considering a minimum turning radius. Due to this, the robot approaches the goal position not in a straight line, but on a circular path. The end orientation is reached gradually and the Goal Checker correctly identifies, as soon as the target is reached.

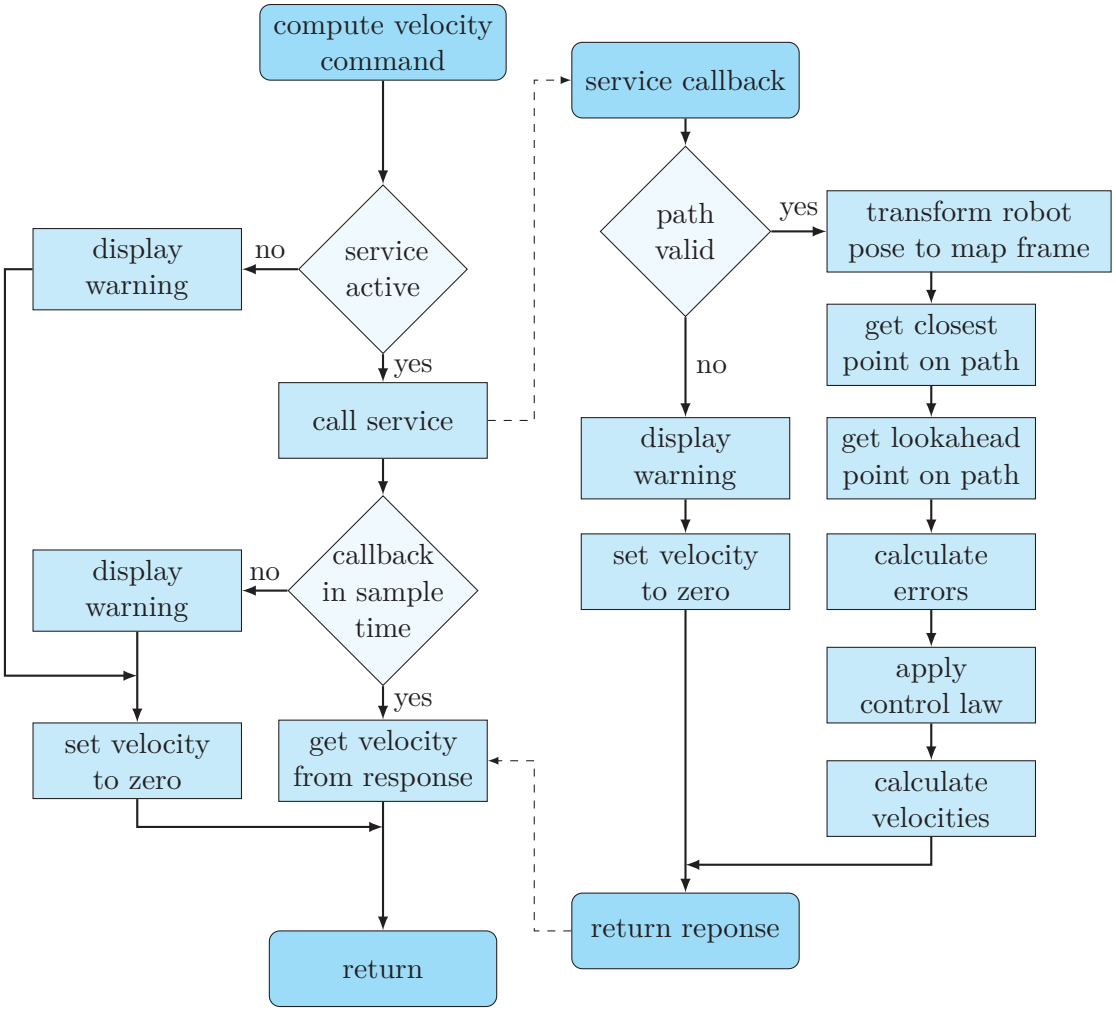


Figure 5.19: Computation sequence of the Stanley Controller.

Comparison to the standard setup

In the documentation of Nav2 [11], it is recommended to utilize the NavFn planner together with the DWB plugin for Differential Drive robots. To compare this combination to the setup of the custom Stanley Lateral Controller along with the Smac-Hybrid-A* planner, both configurations are tested in the simulation. The systems are tasked to navigate to three identical waypoints. To be able to directly compare the different setups, the maximum velocities of the utilized controllers are set to the same values.

Plotted in figure 5.20 are the positions and orientations of the robot while completing the navigation. As a ground truth, the dashed line represents the shortest path to the individual goals when tracked without any errors and with the same constant maximum velocities. Here, the yaw value is approximated by dividing these ideal trajectories into multiple straight line segments. The angles are calculated using two reference points per line, similar to (5.1). The graph is discontinuous, because these shortest connections do not consider the goal orientations. The circles indicate where

waypoints are located in the diagram and the horizontal lines represent their exact values.

It can be seen, that both configurations track approximately the same trajectories and reach the goals within a predefined tolerance. The setup with the Stanley Controller is slightly faster to reach the end point, as the SMAC planner computes a more direct path between waypoints. Additionally the Stanley Controller always drives with a fixed forward velocity, while the DWB plugin is able to lower the overall speed of the robot if necessary.

Most of the times, both setups are able to correctly navigate to a given waypoint. In some cases however, the robot can get stuck, especially when passing through narrow doorways at sharp angles. The configuration using the DWB plugin sometimes collides with obstacles, when the goal is positioned close to a door, but behind a wall in another room. This controller utilizes a costmap for navigation and it may get stuck at local cost minima in these situations. The Stanley Controller on the other hand has a slower reaction time, as its maximum sample rate is limited to 5 Hz due to possible delays in the communication between the NAV2 plugin and the JUPYTER node. For this reason, it sometimes bumps into obstacles marked in the local costmap, as the controller is unable to adjust the velocities to abrupt changes of the trajectory. This shortcoming is amplified by the fact, that the current implementation of the Stanley Controller always drives with a constant forward velocity and is unable to avoid collisions by for example rotating on the spot.

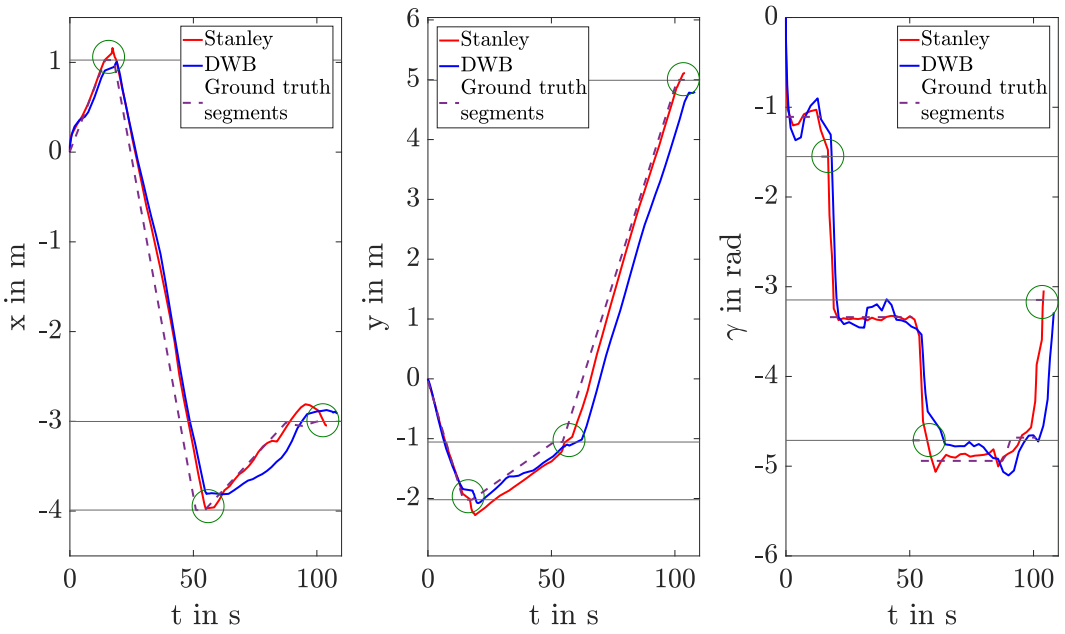


Figure 5.20: Simulated positional and orientational data of both setups.

5.10 Visualization

A ROS2 network consisting only of the aforementioned components would be sufficient to operate the TurtleBot. In many cases however, some form of visualization and remote control is desired. For this purpose, one of ROS2s standard tools, rviz2, is utilized. It provides a GUI depicted in figure 5.21 and displays the robot model with its transforms by listening to the robot_description topic. Maps and costmaps, sensor data and video streams as well as trajectories and the estimations of the AMCL node can be visualized in the same manner.

For additional functions like remote control, various plugins can be added. Default ones include the abilities to publish points by interacting with the displayed map or estimating the robots pose. For the TurtleBot, a Nav2 add-on is included to control the robot in two different ways. The first one publishes a goal position selected on the map to the ROS2 network. In contrast to this simple method, the Waypoint Follower is used to accumulate multiple target positions. By also enabling this component in Nav2, the robot navigates to each individual goal in sequence, making longer, more complex routines possible. Furthermore, actions, executed when waypoints are reached, can be specified. For demonstration purposes, a short delay at each goal is implemented.

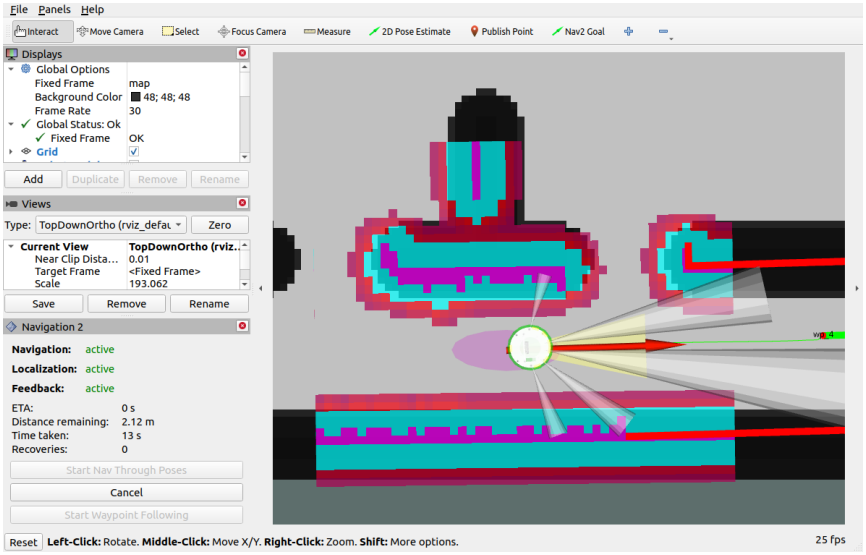


Figure 5.21: Visualization with rviz2.

Chapter 6

Conclusion and Outlook

To confirm its functionality, the ROS2 system was tested using the simulator and the TurtleBot was operated in an actual office environment. Providing different goal positions and target trajectories with varying complexity, it was proven, that the robot can navigate in a dynamic environment using an existing map. As illustrated in previous chapters, the ROS2 network is easily adaptable due to its modular structure and can be started with predefined launch files. The robot successfully combines different sensors for perception and localization and demonstrates many core capabilities of the Nav2 stack. Furthermore, with the addition of the Stanley Controller plugin, an example for the combination of ROS2 and Jupyter is given. Thus, the main goals of this thesis are considered achieved.

While the TurtleBot can perform all desired functionalities and is able to serve as a platform for future projects, further adaptations are recommended to keep the system up-to-date and to improve the current setup.

For one, the odometry sensors of the Roomba are rather inaccurate resulting in deviations of the actual robot position. To improve the localization, the approach discussed in Section 5.7 can be used. Adding an IMU to the physical robot and estimating its position using an EKF, like demonstrated in the simulation, could yield better results than the current setup. As many different models of these sensors are available, there is a number of ways to incorporate them into the TurtleBot. Common options include connecting them to the RPi via USB or utilizing the RPi Pico for handling the data exchange with the sensors.

In addition to improving the odometry estimation, the inclusion of the Roombas other internal sensors like bumpers and cliff detectors into the navigation functionalities could proof valuable. For better obstacle avoidance, the settings of the costmap plugins could be adjusted further. As this thesis uses only the converted LaserScan data of the camera and the range measurements of the ultrasonic sensors, obstructions outside of the detection range can not be processed. For the current setup, items above the scan layer like tabletops or smaller object like bins need to be marked in the static map. By utilizing the complete depth image of the RealSense, these obstacles could be detected and avoided using the local costmap.

On the software side it could be beneficial to upgrade to a newer version of ROS2, considering active development is focused on these distributions. This is possible as soon as the camera driver and all necessary parts of Nav2 are updated. Regarding the robots functionalities, the implementation of SLAM capabilities could be a starting point for subsequent projects, enabling the robot to operate in unknown environments.

Regarding the Stanley Controller, it is advisable to further improve its implementation. By adjusting the calculations of the velocity commands to utilize more efficient algorithms, the sample rate of the controller could be increased. Further adaptations are necessary to make the implementation viable for other path planners. This could be achieved directly in the Jupyter node. For example only a rotation could be performed, when the target position is reached, but the orientation still needs to be corrected. Another option could utilize the capabilities of the Goal Checker to switch between different path planners. In this case, the TurtleBot could rely on the NavFn plugin for general navigation and switch to the Smac-Hybrid-A* planner in the vicinity of the goal for a smooth approach.

Bibliography

[1] N. H. Amer et al. “Modelling and Control Strategies in Path Tracking Control for Autonomous Ground Vehicles: A Review of State of the Art and Challenges”. In: *Journal of Intelligent and Robotic Systems* 2.86 (2017), pp. 225–254.

[2] W. Chung and K. Iagnemma. “Wheeled Robots”. In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Springer International Publishing, 2016, pp. 575–594.

[3] *DDS Foundation*. `https://www.dds-foundation.org/`. Accessed on 08-16-2022.

[4] T. Fischer et al. “A RoboStack Tutorial: Using the Robot Operating System Alongside the Conda and Jupyter Data Science Ecosystems”. In: *IEEE Robotics and Automation Magazine* 29.2 (2022), pp. 65–74.

[5] H. Gattringer. *Vorlesungsskriptum Grundlagen der Robotik*. 2021.

[6] H. Gattringer. *Vorlesungsskripum Robotik*. 2021.

[7] G. Karalekas, S. Vologiannidis, and J. Kalomiros. “EUROPA: A Case Study for Teaching Sensors, Data Acquisition and Robotics via a ROS-Based Educational Robot”. In: *Sensors (Basel, Switzerland)* 20.9 (2020).

[8] S. Macenski et al. “Robot Operating System 2: Design, architecture, and uses in the wild”. In: *Science Robotics* 7.66 (2022).

[9] M. Merdan et al. *Robotics in Education: Current Research and Innovations*. Advances in Intelligent Systems and Computing. Springer International Publishing, 2019.

[10] *micro-ROS Documentation*. `https://micro.ros.org/`. Accessed on 08-01-2022.

[11] *Navigation 2 Documentation*. `https://navigation.ros.org/index.html`. Accessed on 08-10-2022.

[12] *ROS 2 Documentation*. `https://docs.ros.org/en/galactic/index.html`. Accessed on 08-01-2022.

[13] F. Rubio, F. Valero, and C. Llopis-Albert. “A review of mobile robots: Concepts, methods, theoretical framework, and applications”. In: *International Journal of Advanced Robotic Systems* 16.2 (2019).

[14] S. Thrun et al. “Stanley: The Robot that Won the DARPA Grand Challenge”. In: *Journal of Field Robotics* 9.23 (2006), pp. 661–692.

Curriculum Vitae

Personal

Name	Alexander Raab
Born	09 September 1998
Address	Stockhofstraße 23/206, 4020 Linz
E-Mail	alex.raab@liwest.at
Citizenship	Austria

Education

2019–2022	Bachelor's program of Mechatronics JKU Linz
2013–2018	Higher college of Mechatronics HTL Wels

Civil Service

09/2018–06/2019	Wels Caritas Austria