Architectural Layering in Autonomous Robotics: Modular Design and Implementation on the Turtlebot3 Platform using ROS2.

Bachelor's thesis
in partial fulfilment of the requirements for the degree of
Bachelor of Science

Submitted by:

Marvin Kohnen

Matrikelnummer: 462 674

First assessor:
Prof. Dr. Malte Schilling

Second assessor:
Prof. Dr. Paula Herber

Münster, July 10, 2024

Contents

Abstract

This thesis presents a holistic approach to developing a layered architecture, aiming for a robust and performant system that also provides modularity to facilitate adaptability, maintainability, and scalability. After a comprehensive review of both theoretical and applied architectures, the work proposes its own design. This architecture is then tested in a Gazebo simulation and applied to a real-world application on the Turtlebot3 robot. The transition from simulation to real-world applications highlights some challenges, particularly in the higher layers of the architecture. Despite these issues, the proposed architecture demonstrates a reliable and performant core in both simulation and real-world environments.

1 Introduction

The field of autonomous robotics has seen significant advancements in recent years, driven by the need for robots that can operate independently in complex, dynamic environments. Autonomous robots are utilized in various applications, ranging from industrial automation and logistics to healthcare and service industries. Boston Dynamics develops agile, mobile robots to gather data and explore without boundaries, while OPEN AI's "Figure 01" races other companies in the creation of the first commercially viable autonomous humanoid robot. This rapid development raises the question of how to architect a robot, as creating robust and flexible robotic architectures is crucial to ensure these systems can adapt to a wide range of tasks and conditions.

One approach that has been frequently used to address these challenges is *layered architectures*. These consist of horizontal layers that each solve a designated task, upwards in complexity. In contrast, vertical dependencies between the layers are undesirable. This thesis will examine several of these systems and subsequently integrate their most advantageous features to design a layered architecture specifically for the Turtlebot3 platform. The Turtlebot3, a widely utilized educational robot, is often employed in the study of specific fields such as path planning and object detection. However, this work will adopt a more holistic approach, aiming to construct a system that is not only performant and robust, but also provides modularity and therefore allows for easy adapation, scalability and maintainabilty. Additionally, this layered architecture will be based on ROS2 Humble, the successor of ROS, which has not yet been extensively explored in recent research on the Turtlebot3 platform.

The following thesis is structured as follows: First, a presentation of existing robotic architectures, the need for layered approaches, and a brief explanation of the ROS2 framework are provided. Futhermore, a recent study is reviewed, that establishes guidelines formulated by a wide range of roboticists expressing their most crucial concerns when building a robotic architecture. Chapter 3 details the design and implementation of the proposed architecture, the experimental setup, and the evaluation methods. In Chapter 4, the findings from simulation and real-world deployment are displayed and the architectures robustness is examined by an ablation study. Moreover, the adherance to above mentioned guidelines is analysed.

These results are then discussed in chapter 5, while the last chapter summarizes the key contributions of the thesis and outlines limitations.

2 Related Work and Background

2.1 Robotic Architecture

2.1.1 The Need for Robotic Architecture

"Robot systems differ from other software applications in many ways. Foremost are the need to achieve high-level, complex goals, the need to interact with a complex, often dynamic environment, while ensuring the system's own dynamics, the need to handle noise and uncertainty, and the need to be reactive to unexpected changes." [2] This quote by Coste-Manière and Simmons perfectly introduces the various challenges robotic systems and their developers face. These needs shape their design, as all robotic systems embody *some* architectural structure and style. The architectural structure of a system outlines its division into various subsystems and the interactions between them, while the architectural style refers to the underlying computational principles of a system. [2] Different requirements and applications for robotic systems result in different architectures ("To date, the number of existing architectures has reached several hundred" [4]), some of which this thesis will explore in the following section.

2.1.2 The concept of Perceive-Reason-Act

At the foundation of most robotic architecture lies the concept of "Perceive-Reason-Act" (**PRA**). The principle takes its inspiration from neuroscience: **P**erception creates a model representing the current world state. The **R**eason component designs a plan using the model to find an appropriate action towards the goal. Finally, **A**ct refers to the translation of the action into actuator commands and therefore resulting in motor output. [20] An overview of the concept is given in figure 2.1. The rather intuitive PRA approach contains some drawbacks in real world applications: First, not every aspect of the environment is senseable (such as the inner state of other agents). Furthermore, key elements of the world may be difficult or rather impossible to model. Lastly, sensor and actuator measurements in the real world might differ vastly from the expected values in a clean lab situation. [20] These limitations prompted roboticists to develop more

sophisticated systems tailored to individual challenges and changing environments. The general approach to perceiving and navigating the environment remains the same.

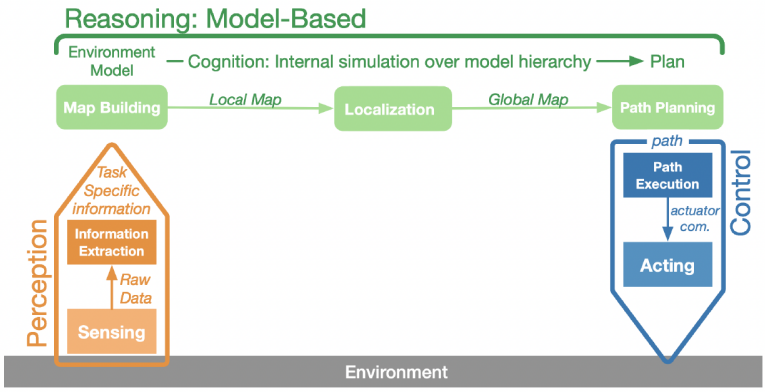


Figure 2.1: Overview of Perceive-Reason-Act [20]

2.1.3 Development of Cognitive Architecture

Cognitive Architecture is part of research in General AI and refers to the underlying frameworks that enable robots to mimic human-like cognitive processes. These architectures integrate perception, reasoning, and action to facilitate complex behaviors and decision-making in dynamic environments. By structuring robotic systems to process information, learn from interactions, and adapt to new situations, cognitive architectures aim to enhance the autonomy and intelligence of robots. Thus enabling them to perform tasks that require high levels of understanding and adaptability. [4] Given the dynamic and diverse nature of environments and applications, numerous definitions have been proposed for the key criteria that define a cognitive architecture. According to Newell [9], essential criteria for cognitive architectures include flexible behavior, real-time operation, learning, development, linguistic abilities, self-awareness, and brain realization. Sun [23] expanded this list to include broader concepts, such as ecological, cognitive, and bioevolutionary realism, modularity and synergistic interaction. However, differences in research goals, structure, operation, and application among existing cognitive architectures pose significant challenges for comparison and evaluation, making it nearly impossible to give a distinct definition. [4] Cognitive architectures, while powerful tools for modeling and understanding human cognition, come with several disadvantages. Despite their potential, the development and maintenance of cognitive architectures are complex and resource-intensive, demanding significant effort to build, debug, and extend. They often require substantial computational power and memory, which can limit their application in real-time scenarios or systems with constrained resources. Additionally, as these architectures grow in complexity, they can encounter

performance bottlenecks, making scalability a challenge for larger, more intricate tasks. Integrating new modules or functionalities can also lead to integration issues and conflicts. [5, 4]

Even though many of its advantages seem well-suited for our system, the drawbacks render cognitive architectures unsuitable for the purposes of this thesis, particularly in terms of adaptability, maintainability, and scalability. Consequently, we opted to explore alternative architectural types, with layered architectures emerging as the most viable option.

2.1.4 Layered Architectures

Layered architectures take a different approach to resolve the issues that the PRA model encountered. Horizontal layers of control systems are built to let the robot operate at increasing levels of competence and complexity and are assigned with different objectives. They consist of asynchronous modules which are allowed to communicate in a bidirectional manner. [1] Even though the higher levels can subsume the roles of lower levels by suppressing their output, the lower levels maintain their functionality as higher levels are added - resulting in a robust and flexible robot control system. [1] An example of a layered architecture can be seen in Figure 2.2, created by Brooks in 1986. [1] This figure demonstrates the integration of the PRA concept into architectural layering, with sensors representing perception, the layered architecture itself serving as reasoning, and actuators fulfilling the act component of PRA. Furthermore, horizontal layering is emphasized.

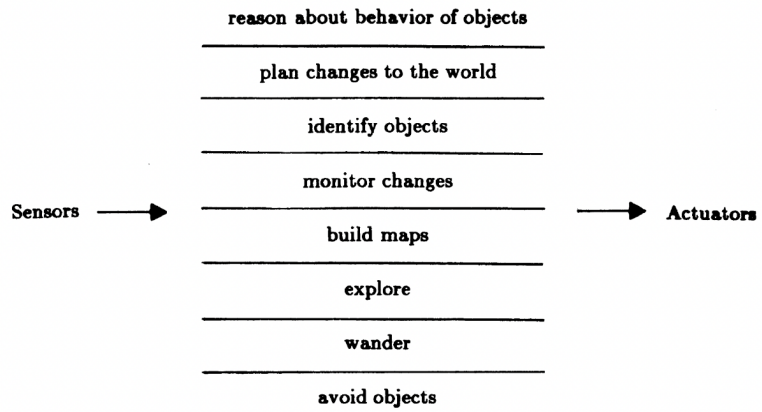


Figure 2.2: Early Layered architecture by Brooks in 1986 [1]

Historically, layered architectures are composed of three layers: Behavioral layer, Executive layer and Planning layer. [3, 10]

The Behavioral layer (also called the Hardware Abstraction- or Functional layer) is typically found at the bottom of the hierarchy inside a layered architecture and provides a low-level library or interface to access the robot hardware, such as sensors or actuators. [3, 22] It is often composed of very fast control loops, to create operations like path tracking or reflex-like reactions. [20]

The Planning layer sits at the top of most architectures, making the decisions to achieve high level goals. This more complex and slower layer then sends the plans to the *Executive layer*, which further divides tasks into subtasks. It is responsible for translating high-level plans into low-level behaviors, invoking behaviors at the correct times, monitoring execution, and handling exceptions. [20, 22, 24] As stated earlier, information and control is allowed to flow up and down through these layers, as the Behavioral layer sends back sensor data and actuator information to the Executive layer, which then informs the Planning layer on the current status of the assigned tasks. [3] A prototype of this structure can be seen in Figure 2.3. This architectural concept can even be extended to multiple robots, creating a sort of hive or swarm control, as Simmons et al. have shown in 2002. [22]

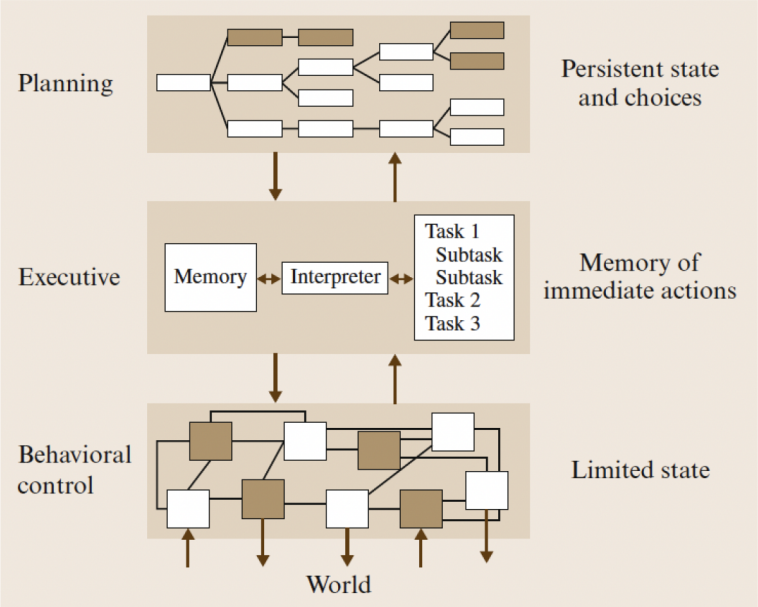


Figure 2.3: Prototype of a three-tiered architecture [21]

2.1.5 A different approach to Layered Architectures: CLARATy

While this typical three layered architecture is very popular and has been thoroughly investigated by developers, it does not come without its drawbacks. The developers of the "Coupled Layer Architecture for Robotic Autonomy" (**CLARATy**), a two layered architecture that is used for NASA's space robots [21], identified three problems when designing three layered architectures: Firstly, the developers "expand the capabilities and dominance of the layer within which they are working" [24]. This results in models, in which certain layers are predominantly active. Furthermore, there is still the need for research about the hierarchical superiority of the Planning and Executive layer of one over the other. [24] Another problem the authors state is the lack of access from the Planning level to the Behavioral control. This forces the system to carry own models on the Planning layer, which are not directly derived from the functional layer, to perform planning tasks. Eventually, this can lead to inconsistencies between the layers and endanger the integrity of the system. [24] Lastly, the authors claim that there is a misconception in equating greater intelligence in a system with increased granularity, suggesting that each part of a system can possess its own distinct hierarchy of granularity. They move on to explain that the Functional Layer contains many nested subsystems, the Executive layer encompasses multiple logic trees to coordinate these subsystems, and the Planning layer includes various timelines and planning horizons with different resolutions. [24] In order to fight these challenges, Volpe et al. proposed a two layered architecture. This offers two main benefits:
Firstly, an explicit Representation of Granularity: CLARATy introduces a third dimension for granularity, allowing for a clearer depiction of the system's complexity and hierarchy. This means that within the Functional layer there's a detailed organization showing how subsystems are nested or grouped together. Secondly, they blend the Planning and Executive layer, resulting in a *decision* layer which is more efficient and allowed the levels to share a common database (depicted in figure 2.4). [24]

After this concise overview of some existing architectural systems, this thesis will now present the integral middleware that serves as the backbone of the proposed architecture. Understanding the principles of a layered architecture provides a solid foundation for appreciating the architectural choices and design patterns employed in robotic systems. As seen, there is no definitive way to construct these systems. Rather, this architecture type provides some key characterics by its division of system functionalities into distinct layers. Each layer fulfills a specific role and offers clarity, modularity, and scalability. Transitioning the focus to the Robot Operating System (ROS) reveals how ROS embodies these principles through its comprehensive framework for robot software development.

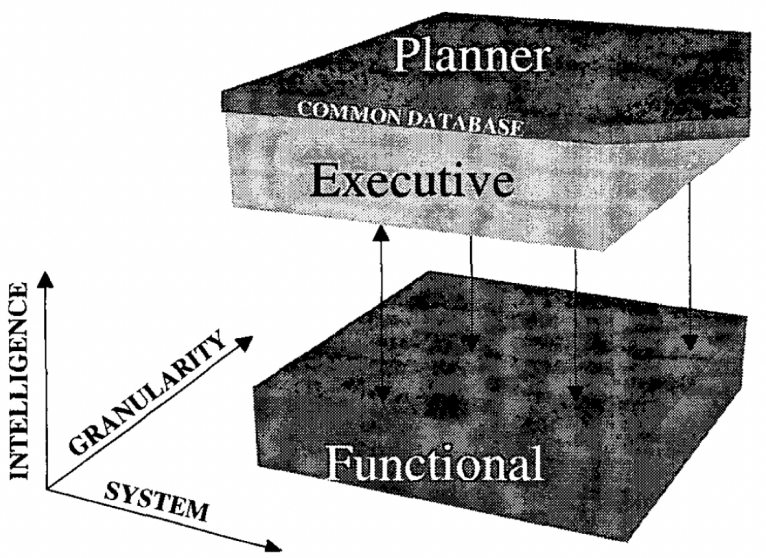


Figure 2.4: Overview of the CLARATy architecture [24]

2.2 Robot Operating System

2.2.1 ROS 1

The Robot Operating System (ROS) is an open-source operating system for robots. It provides the services expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. ROS is not an operating system in the traditional sense of process management and scheduling. Rather, it is a framework or middleware upon which robotics software is developed, offering a structured communications layer above the host operating systems of a heterogeneous compute cluster. [11, 19] ROS's architecture is designed around the notion of the intercommunication between numerous computing processes (often referred to as nodes) over a peer-to-peer network. This decentralized computing architecture enables the development and integration of complex robotic systems from modular and reusable components or packages. The system facilitates a distributed computing environment, allowing for the separation of tasks such as sensing, perception, decision-making, and actuation across multiple processors and machines. [11] Central to ROS is its communication and control flow infrastructure, which enables the exchange of messages in various forms. These fundamental concepts include messages, topics, services and above mentioned nodes. The latter communicate with one another through messages ("strictly typed data structures" [11]). This communication is implemented as a *publisher-subscriber system:* On the one hand Nodes send messages to a topic by publishing the information, on the other hand

nodes can receive the published information by subscribing to said topic. Multiple concurrent publishers and subscribers for the same topic may be active at any time. Also, a single node can publish and subscribe to many topics. [11] Additionally, ROS provides capabilities for package management and a vast ecosystem of community-contributed tools and libraries that address various robotics functions ranging from planning and perception to simulation and visualization. [11, 19]

2.2.2 ROS2 and its Improvements

Even though ROS laid the groundwork for the modern robotic industry, the growing interest of industrial applications show the limitations of ROS, which was built as a research platform. The urge for security, reliability in non-traditional environments and support for large scale embedded systems became the driver for the creation of ROS2, the second generation of the Robotic Operation System. [7] It introduces Node Lifecycle management, replaces the custom ROS1 middleware with industry standard middleware DDS (Data Distribution Service), better Real-Time support and adds Actions as a communicational tool. Actions in ROS2 are designed for long-duration tasks that require feedback to the caller during execution, which makes them distinct from the simpler service-call model. [7] A typical node interface for ROS2 can be seen in the figure 2.5 below.

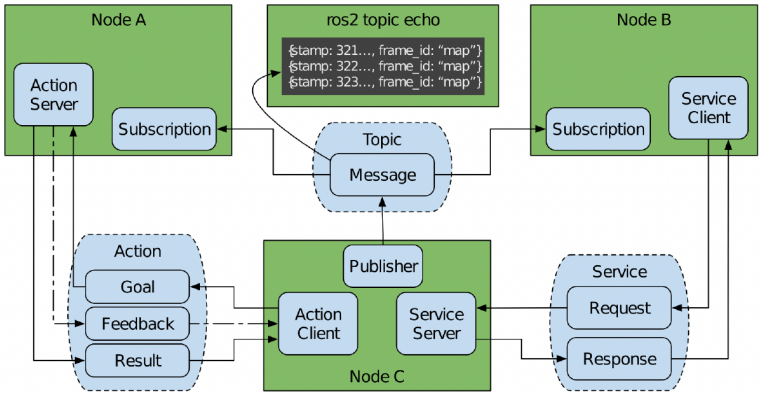


Figure 2.5: Overview of ROS2 Node interfaces. The main communicational tools of the ROS middleware: Topics, Services and Actions. [7]

2.3 How do you Architect your Robots?

2.3.1 State of the Art and Best Practices for ROS-based systems

Malavolta et al. [8] conducted a study in 2020 to answer three research questions: the architecture-relevant characteristics of ROS-based systems, the quality requirements considered in their design, and how to guide roboticists in architecting such systems. The authors used a two-part approach, mining ROS repositories and surveying developers who contributed to these repositories. The study found that ROS systems are becoming large and complex. While there are many open source packages and examples on how to use ROS, engineering robots with specific properties is still mostly an art of trial and error.The findings show that the most mentioned quality requirements related to ROS architecture were maintainability, performance, and reliability. The study also identified 39 guidelines for architecting ROS-based systems. These guidelines are useful for both roboticists aiming to develop high-quality robots and architecture researchers looking for evidence-based indications on how real-world ROS systems should be architected. This thesis will now present 7 out of those 39 guidelines, which were evaluated as the most impactful for architectural design of robotic systems. [8]

2.3.2 Architectural Guidelines

1. **Use standardized ROS message formats as much as possible, possibly supporting also their legacy versions.**

   This first guideline emphasizes the importance of using standardized ROS message formats, particularly those from the common_msgs and std_msgs packages. Adopting these standards facilitates easier reuse, upgrade, and replacement of ROS nodes by allowing for straightforward topic remapping. Standardized messages enhance development efficiency by making compatible tools, such as visualizers and SLAM algorithms, easier available. They also improve node testability through isolation testing with ROS bags and simplify the integration of new sensors and hardware, as many manufacturers support ROS out of the box. However, since the definitions of standardized message formats are subject to change, it's crucial to design systems to be as decoupled as possible from specific message formats to ensure future compatibility and ease of updates. [8]

2. **Group nodes and interfaces into cohesive sets, each with their own responsibilities and well-defined dependencies.**

Growing complexity poses a serious challenge while designing ROS-based software architectures, as it can result in hundreds of interdependent nodes leading to technical debt, limitations to certain ROS packages and reduce adaptability and extendability of the whole system. To mitigate these issues, the guideline suggests grouping nodes and interfaces into cohesive sets with defined responsibilities and dependencies. This structured approach aids in maintaining a clear understanding of the system's architecture, allowing for easier evolution and maintenance of the system. [8]

3. **The behavior of each node should follow a well-defined lifecyle, which should be queryable and updatable at runtime.**

   ROS inherits a certain flexibility that allows developers to create nodes without prescribed behaviors, offering great freedom but posing challenges for testability, reliability, and maintainability of stateful nodes. This guideline states the importance of treating node lifecycles as a crucial aspect of system design, suggesting documentation of node lifecycles to improve interaction, predictability, and test case development. As stated earlier, the development of ROS 2 already addresses some of these issues by supporting managed nodes with defined lifecycles (Unconfigured, Inactive, Active, Finalized) that can be inspected and controlled, enhancing testability and system management. Additionally, some ROS packages, even outside ROS 2's managed nodes, incorporate lifecycle considerations into their APIs, enabling runtime configurability and reflection to bolster system adaptability and autonomous capabilities. [8] Defining and enforcing the lifecycle of ROS nodes "can enhance the system in terms of run-time configurability and reflection, which can be exploited for providing autonomous capabilities". [8]

4. **Assign meaningful names to components (e.g., nodes, topics, services) and group them by adopting standard prefixes/suffixes.**

   This pretty self explanatory problem is especially important in ROS-based systems, as topics and services are created programmatically by the nodes at run-time and their identifiers are simple strings. [8]

5. **Nodes directly interacting with simulators and hardware devices should provide identical ROS messaging interfaces to the rest of the system**

   This guideline essentially boils down to creating uniformity in messaging interfaces, which allows developers to easily swap between simulations and hardware applications, while reducing modification cost to a minimum. [8]

6. **ROS nodes should be resilient with respect to the amount and frequency of the data received by sensors.**

This principle highlights the challenges developers face when working with hardware sensors in robotics: Sensors often produce data in bursts, can degrade, and become less accurate over time. To handle these challenges, developers are advised to design their systems with flexibility in mind, by implementing load balancing nodes to manage sudden data surges and designing nodes to be resilient to gaps in sensor data. This resilience is crucial for the reliability of state estimation nodes, where faults can cause significant system-level failures. I expect this guideline to not be of great relevance in my project, due to the rather small(-ish) dataload. Nonetheless, a cruicial thought to keep in mind when designing robotic systems. [8]

7. **Avoid persisting raw data if only part of it will be used.**

   This guideline refers to the obstacle of how extensive data logging or storage operations can negatively impact the system's performance during its operation. Developers are advised to selectively record only the necessary data to avoid these performance issues. [8]

The proposed architecture will try to adhere to these guidelines as much as possible, in order to create a robust and performant robotic system.

2.4 Examination of a Case Study: The Niryo One Arm Architecture

The Niryo One is a collaborative and open-source 6-axis robot designed for research and higher education in the context of the industry 4.0. Its architecture has been mentioned by Malavolta et al. as an example, that adheres very well to above stated guidelines. [8] Niryo One operates upon a layered architecture, which is composed of five layers. Each layer is further subcategorized into different packages, among them pre-built ROS packages, imported libraries or self-created packages.

At the lowest level we can find the **Hardware layer**. This layer contains a package *rpi* that handles all the external hardware (everything besides the motors) of Niryo One, and provides many utilities for the Raspberry Pi 3, such as Wi-FI, LEDs, Buttons etc. Furthermore, a *driver* package is built, which provides an interface to ros_control and handles the hardware control of motors. The other two packages found in figure 2.6 (*dynamixel_sdk* and *mcp_can_rpi*) are imported libraries to stabilize functionality of the Raspberry pi and control the actuators. [13]

Next up, a **control level** is implemented, which is entirely made up of the pre-built ros packages *ros_control* and *joint trajectory controller*. The former is a set of packages that include controller interfaces, controller managers, transmissions and hardware interfaces [18], while the latter provides a controller for executing joint-space trajectories on a group of joints. [15]

The actual motion control is invoked by the next higher level, the **Motion Planning layer**. Not only is there a *description* package, containing the URDF file and meshes for Niryo One (which is essentially a description for the robot), but also the widely used *moveit* packages, which are already part of the ROS ecosystem. The motion planning layer is responsible for finding inverse kinematics and building a path for the robot.[12, 17] For each point, each axis is given a specific position, velocity, and acceleration. This path is sent to the joint trajectory controller for the actual hardware execution. [12]

On top of that, the **Commander layer** provides a higher level interface between the client and the underlying robot commands. All commands, that the user inputs (e.g. "move joints"), go through this layer. The *commander* package handles concurrent requests, validates parameters and calls required controllers and then returns appropriate messages and status. Furthermore, a *python_api* package allows for easier access for other developers and tries to reduce the complexity of ROS for beginners. [12, 13]

On the highest level, the architecture features an **External communication layer**. This layer allows for communication outside the ROS ecosystem. First a TCP server is established in the *modbus* package. It offers a simple way for developers to create programs for robots to control them via remote communication on a computer, on a mobile or any device with network capability. Lastly, the ROS-packages *rosbridge* and *joy* are imported, in order to allow communication between ROS1 and ROS2 and the use of the joystick controllers. [12, 13]

After this thorough examination of existing architectures, we will now introduce our own approach to a layered robotic system.

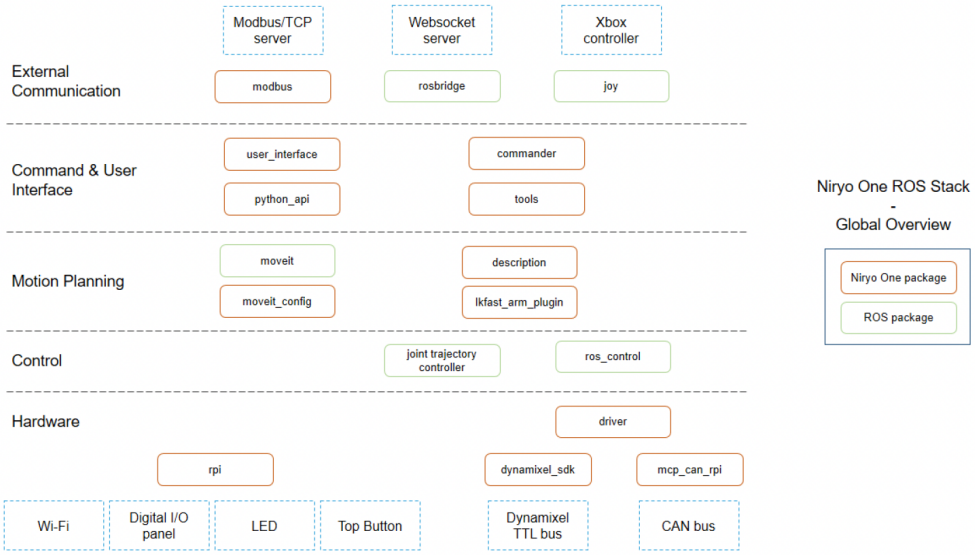


Figure 2.6: Niryo One ROS Stack overview [13]

3 Methodology

3.1 Frameworks and Setup

The architectural system is employed on the Turtlebot3 Burger Bot platform. This robot is frequently used in education and research because of its size, affordability and wide range of adaptability and expandability. A brief overview of the robots composition is shown in Figure 3.1.

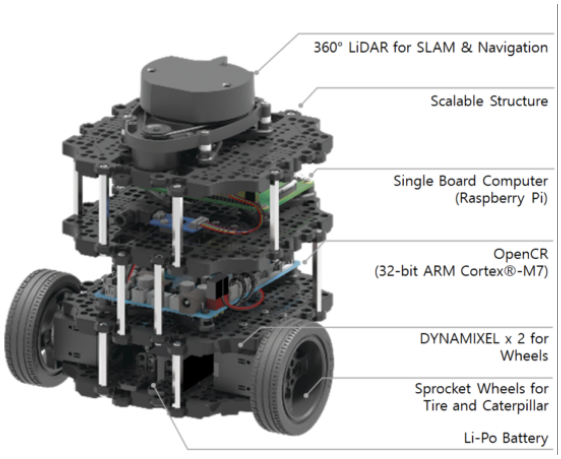


Figure 3.1: Turtlebot3 components and structure

The Architecture is built on the Ubuntu 22.04 LTS Operating System and ROS2 Humble for both the remote PC and the Raspberry Pi4 [1] on the Turtlebot3. The OpenCR board[2] is setup as instructed by the distributor of the Turtlebot3 platform ROBOTIS.[14]
Any simulation is done with Gazebo Fortress (as required for ROS2 Humble) and RVIZ. Gazebo is a robust simulation environment used for testing and developing robotic algorithms in complex, realistic settings. It offers a precise physics engine, advanced 3D graphics, and supports a variety of sensors and actuators. RViz on the other hand is a 3D visualization tool for ROS that displays real-time data from sensors and robot

[1] A Raspberry Pi is a small single-board-computer which is often used in robotics, because of its low cost, modularity and open design
[2] OpenCR is the Open-source Control Module for ROS. It is developed for ROS embedded systems to provide completely open-source hardware and software

models. It helps in debugging and analyzing robotic systems by providing insights into sensor perceptions and algorithmic decisions. RViz's customizable interface allows users to tailor visualizations to their specific needs, enhancing understanding and performance analysis of robotic systems. The different layers of the architecture are realised through Python and /Shell scripts, as well as some C/C++. Each layer is essentially a ROS package, containing not only python scripts, but also setup.py or CMakeLists.txt files to configure system entry points and package.xml to resolve any dependency issues. A simplified version of the architectures directory tree can be seen in Figure 6.1 in the Appendix.

In the next section, a detailed overview of the system is presented.

3.2 Architecture and Design

The architecture is structured into multiple distinct layers as shown in Figure 3.2, which grow in complexity the higher the layer is placed in the diagram. Closely following the example set by the Niryo One architecture 2.6, the overview distinguishes between pre-built ROS packages and custom-built components. Additionally, the control flow of the architecture is represented by arrows of different colors, each indicating a specific type of interaction. Green arrows signify a long-lasting invocation of a service, action, or publisher. Red arrows denote a one-time invocation of a new module using bash scripts. Grey arrows indicate suppression, while blue arrows represent access to a topic, service, action, or entire functions of another package.

At the foundational level, the *Hardware Control* package serves as the primary interface with the robot's physical components. It is the only layer [apart from the nav2 package, which we will get to later on] that processes and publishes commands directly to the robot's motors and sensors. This module's design includes a prioritization mechanism that manages conflicting commands to ensure that higher priority tasks, such as emergency stops, preempt other actions. The Hardware Control acts as a library of functions for the other layers to call upon, in order to create movement, e.g. "Drive Forward", "Turn Left", "Turn Right", "Turn by Angle" etc.).

Above that is the rather straightforward *Obstacle Avoidance* layer, which continuously reads sensor output from the 'LaserScan' message [16] and calls upon the hardware layer to immediately stop and rotate the robot. This command is sent with the highest possible priority. If the Obstacle Avoidance layer is missing or dysfunctional, a flag is set and the Hardware Control is prohibiting all movement commands, to ensure robot safety.

For map management, the architecture integrates a SLAM Toolbox. This tool is critical for building and updating the map while the robot navigates through its environment. The SLAM process simultaneously localizes the robot within the map and updates the map's structure based on new information gathered from the robot's sensors. This data is published through the /map topic, permitting access to other layers.

In order to make use of the SLAM Toolbox and create a map, the robot has to wander around in its environment. This is realised through the *Random Explore* layer, which does exactly what the name suggests: Randomly explore the surrounding area by invoking movement through the Hardware layer with lowest priority commands.

In the same package as the SLAM Node, a *Map Change Detection* Module is located, which continuously monitors the environment for changes, updating the navigation map on a set frequency (currently 30 seconds). It uses a change detection algorithm that compares the current map state with new data collected by the robot, ultimately deciding when the map is robust enough to request the navigational layer to take over the robots movements. This request is executed via a bash script that launches a new terminal and invokes the desired navigational module.

At the highest level, the *Navigation Layer* provides autonomous navigation capabilities. It is important to note, that two concurrent navigational modules are placed inside this layer: One is custom built while the other employs the nav2 package, a commonly used navigational framework which provides a wide range of algorithms and functionalities to facilitate path planning and directed movement. [6] Both modules dynamically calculate the most efficient paths to the designated targets, considering the current environmental layout and obstacles as previously mapped by the SLAM Toolbox. While the Nav2 package overrides any movement commands independently, the Custom Navigation module relies on the Obstacle Avoidance and Hardware Control packages to facilitate movement. Any other navigational logic is handled inside the Custom Navigation itself. In contrast, the navigation module using nav2 invokes an Action client, a tool exclusive to ROS2, in order to communicate with the nav2 package. This action server uses asynchronous services for setting a goal, canceling a goal and requesting results.
Moreover, both navigational systems suppress the Random Explore package by setting a flag and therefore preventing any callbacks inside the Random Explore layer.

Lastly, there are a couple more modules which provide analytical output for this thesis such as Maps, Heatmaps or position logging. These modules dont impact the robots behaviour in any case. Nonetheless, they are part of the architecture and are worth mentioning. Most importantly, a Heatmap Generator is placed in this *Analysis layer*. As soon as any navigational functionality is invoked, it maps the robots position from the /odom frame to the /map frame and therefore creates a heatmap that matches the

given dimensions by the SLAM Toolbox. Some examples of these heatmaps will be shown in Chapter 4.3.

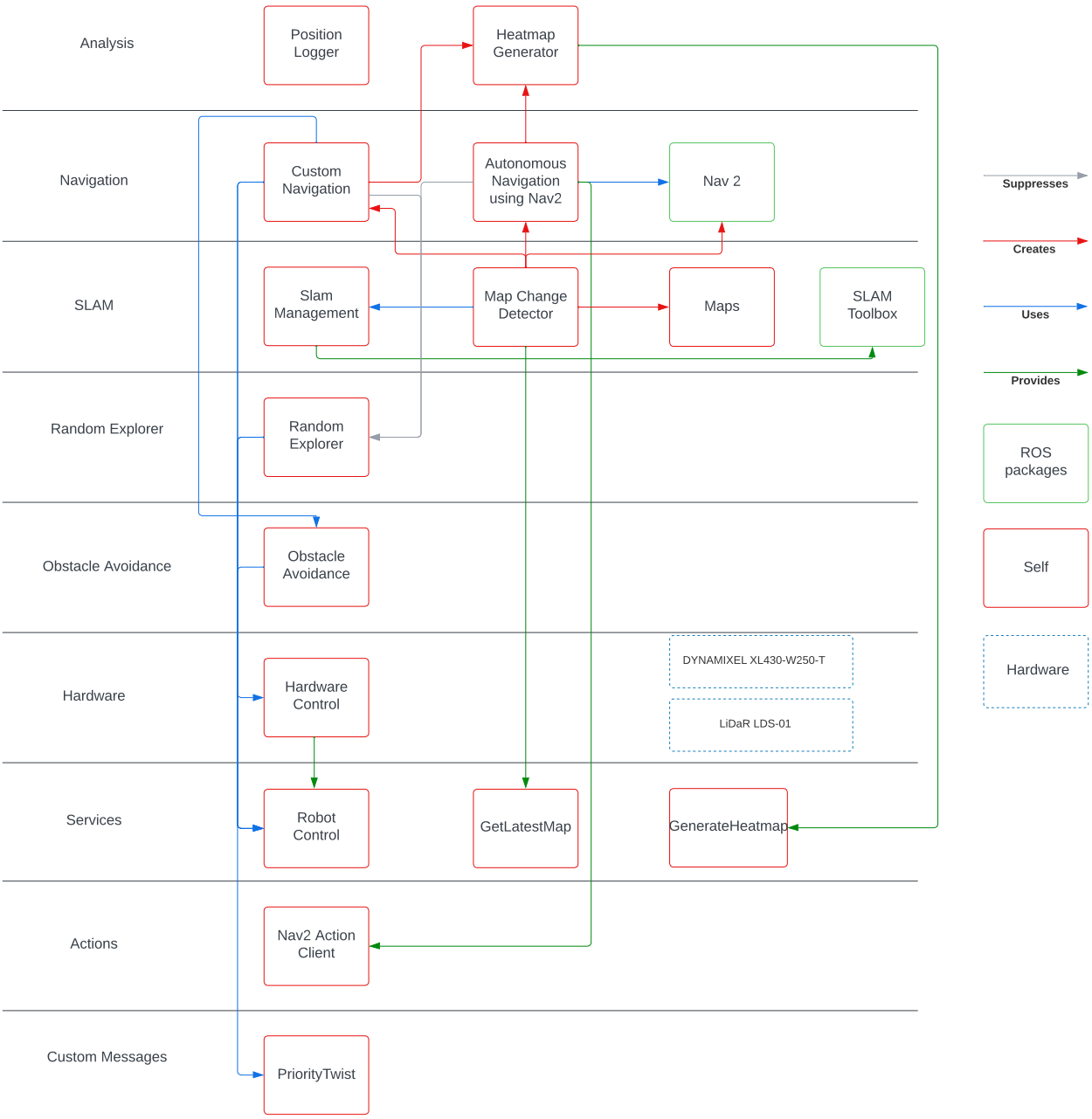


Figure 3.2: Architecture Overview. Boxes symbolize modules inside the layers. Arrows define control flow.

4 Results

With the specifics of the architecture now established, we can analyze its adherence to the guidelines introduced by Malavolta et al. Subsequently, we move on to qualitative research in form of an ablation study. Since performing quantitative research on a layered architecture for robotic systems is relatively limited by the quantifiability of certain layers, this thesis then analyzes the performance of the two navigational systems. While their performance is undoubtedly significant, it is not the primary emphasis of this work. Instead, the thesis demonstrates the successful implementation of two interchangeable systems within the architecture, highlighting its modularity and adaptability.
This analysis is mainly conducted in Gazebo and RViz, the prevalent simulation environments for robotic systems and ROS based applications. Following this, the application of the architecture on the real Turtlebot3 platform will be presented.

4.1 Evaluation of Guideline Adherence

1. **Use standardized ROS message formats as much as possible, possibly supporting also their legacy versions.**

   The architecture uses mostly standardized ROS message formats, such as LaserScan and Odometry. The external tools used in the system, such as SLAM Toolbox and Nav2, also use standardized ROS message formats: LaserScan, Odometry, Map and Twist. The only time the architecture deviates from this guideline is through the introduction of the custom PriorityTwist message. This design choice was made consciously, since the other possible solution for the prioritization mechanism involved environmental variables or global variables which not only are not common practice in Python, but also are very hard to maintain and reduce portability of the system. Furthermore, the priority is rather straightforward to understand, since it basically contains a normal Twist message at its core with only an added integer field.

2. **Group nodes and interfaces into cohesive sets, each with their own responsibilities and well-defined dependencies.**

Since each layer provides a specific functionality in the greater system, dependencies are kept to a minimum and easily imported wherever needed. Apart from the standard ROS message dependencies, most layers only depend on the Hardware Control package (or rather the Robot Control service). An exeption is the Navigational layer, which is connected to the vast framework of Nav2 and the SLAM Toolbox. These dependencies are necessary and therefore not within the inherent control of the architecture's design.

3. **The behavior of each node should follow a well-defined lifecyle, which should be queryable and updatable at runtime.**

   Each package in the system invokes and manages its own nodes, which is a result of the ROS2 Humble middleware with its managed nodes in order to enable runtime configurability. Furthermore, any node that is not initialized by the user at the start of robot operation has a well-defined lifecycle. Since the architecture is autonomous, most nodes are not designed to be updatable by user input at runtime, once their lifecycle has been invoked.

4. **Assign meaningful names to components (e.g., nodes, topics, services) and group them by adopting standard prefixes/suffixes.**

   The architecture establishes a standardized name convention. Any package is named "turtlebot3 packagename", with the corresponding node being named after the package name. Moreover, the entry points in the respective setup.py files follow the same name convention, resulting in rather straightforward terminal commands (e.g. ros2 run turtlebot3_hardware_control hardware_control).

5. **Nodes directly interacting with simulators and hardware devices should provide identical ROS messaging interfaces to the rest of the system**

   When moving from simulator to hardware devices, the user only has to change a single flag in the bringup. Namely, the SLAM Toolbox expect a a boolean parameter "use sim time", which is set to true in simulation. This flag is queryable via the terminal, leaving the codebase untouched when transitioning environment.s Other than that, the system is indifferent to the usage of simulation or hardware application.

6. **ROS nodes should be resilient with respect to the amount and frequency of the data received by sensors.**

   As expected, the system encounters a rather small dataload from the lidar sensor. Furthermore, this dataload is comparably consistent. The absence of big sensor data has lead to little emphasis on this guideline. Therefore any addition to this

system that increases sensor load needs to be tested thoroughly. Since the Obstacle Avoidance has its own managed node, faces a very small data load and operates on highest possible priority, the architectures robustness to system-level failure is suppsoedly quite high.

7. **Avoid persisting raw data if only part of it will be used.**

   Apart from the map created by the SLAM Toolbox and any data generated for user visualization, no sensor data is persisted. However, there is no garbage collection system in place, which means that over many iterations of the model, a significant accumulation of maps and map configuration files may occur.

4.2 Ablation Study

In order to further discuss the robustness of the architecture, an ablation study was conducted (see Figure 4.1). The effect of an omitted layer on the other modules is categorized into three cases. Firstly, a layer can be completely unaffected, resulting in no changes to performance or robustness. Secondly, a layer can be indirectly affected by changes in other layers—meaning the layer itself does not directly access the contents or functionality provided by the omitted package but faces secondary consequences, such as an incomplete mapping of the environment. Thirdly, directly affected layers explicitly access the ommited layer and face severe repercussions in terms of unmet dependencies, unavailable services or other integral parts of the architecture. On the one hand, the omission of higher layers [1] has no effect on the levels below it, which accounts for the lower triangular matrix of *unaffected layers*. On the other hand, the removal of lower layers has varying impact on the levels above it.
As expected, the ablation of the Hardware Control layer has a direct negative impact on all layers that rely on it to invoke movement, namely Obstacle Avoidance, Random Explore and the Custom Navigation. It also indirectly affects layers that rely on the robot being moved around (SLAM layer) and therefore by extension also the Navigation using Nav2, even though it overwrites any other movement commands and controls the robot on its own.
As a result of ommiting the Obstacle Avoidance module, the Random Explore layer cannot safely navigate the robot in an unknown environment, as the robot is very likely to crash into the first obstacle in its path. This culminates in an unfinished map, making any kind of navigation obsolete. The Custom Navigation also continuously accesses certain flags set in the Obstacle Avoidance package and is therefore directly affected.
The ablation of the Random Explore layer inherits the same problems we just described, with the exception that the Custom Navigation is not directly accessing said layer.
Clearly, the exclusion of the SLAM and Map Change Detector layer results in both navigational systems failing on multiple levels: Not only is no data being published to the /map topic, that both navigation systems heavily rely on, but both packages are also never invoked in the first place.

Lastly, it is evident that the two navigational packages do not interfere with one another and can functionally coexist - provided a given priority between the two navigational system is established. This means they are practically interchangeable and therefore qualify as a unaffected layer. Note that the ablation of either navigational package during runtime does not interfere with the robot's ability to navigate safely in both simulation

[1] *Higher* and *lower* layers are to be interpreted as shown in Figure 3.2.

and real-world environments. Since a flag is set during the runtime of the navigation packages to suppress the Random Explore layer, the removal of the navigational module leads to a fallback to random exploration. Also, if the Obstacle Avoidance layer is missing, a similar flag is set to prohibit movement commands issued by the Hardware Control layer, ensuring that robot safety is prioritized above all else.

Figure 4.1: Ablation Study on layer dependency. The horizontal axis of the diagram represents the layers of the architecture that are omitted one by one (i.e., the ablated layers). The vertical axis represents the layers that remain active during the ablation study. The intersections indicate whether the active layers are unaffected, indirectly affected, or directly affected by the omission of the corresponding horizontal layer.

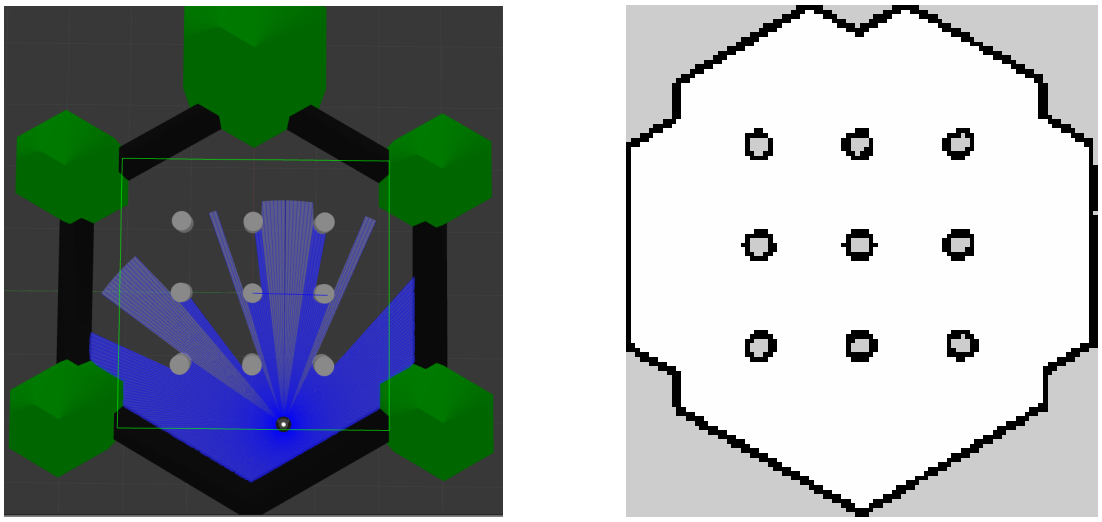
It becomes evident that the lower levels are vital for the performance and robustness of the architecture, while higher layers are more likely to have less devastating effects on overall stability or simply enhance the system's functionality. This additive nature results in a more adaptive and flexible design, as higher-level modules can easily be replaced by others with similar purposes. In contrast while examining the upper triangular matrix of figure 4.1, it is inevitable to mention the dependency of the higher levels on the lower layers. This interdependence limits the overall flexibility of the architecture by how resilient and adaptable the lower layers are designed.

After this thorough analysis of the architectures resilience and robustness, we now move on to assess the systems performance. This evaluation will be divided into a simulated environment and an application to a real surrounding.

4.3 Simulation

The Gazebo simulation uses the standard map *Turtlebot3 world*, which is part of the Turtlebot3 simulation package. This world is shown in Figure 4.2. The left image displays the Turtlebot3 world within the Gazebo simulation environment. The right image shows the same world as mapped by the SLAM Toolbox and Map Change Detection module, rendered in the Portable Gray Map (PGM) format.

The nine pillars form obstacles for the robot to avoid.



(a) Turtlebot3 World in Gazebo

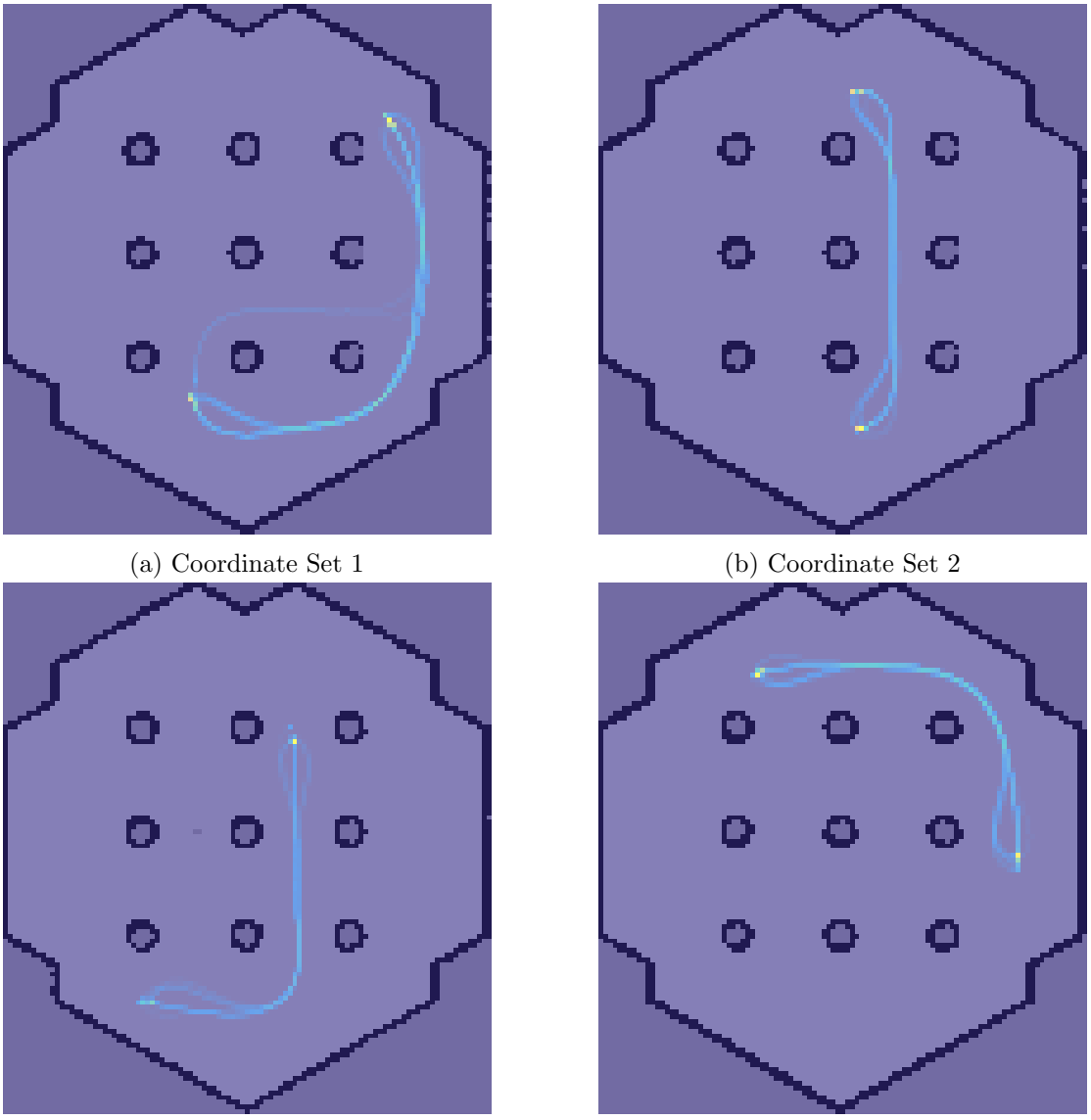
(b) World as Portable Gray Map

Figure 4.2: Simulated Environment for the Turtlebot3

The robot was provided with four distinct coordinate sets to navigate between. Each tuple of coordinates was targeted 20 times, resulting in 10 movements towards the goal-coordinate and 10 movements back towards the start-coordinate. While moving, two key metrics were collected: Distance traveled and the time used to get there. The full datasets for each iteration can be seen in the Appendix. Note that all coordinates were chosen at random and do not contain any bias towards one another, but both navigation systems where given the same coordinates. To condense the findings, the mean and standard deviation of the two metrics are calculated and shown in their respective tables. Furthermore, the movement of the robot was monitored via a heatmap, that was then projected onto the map provided by the SLAM Toolbox and Map Change Detection module.

4.3.1 Navigation using Nav2

Deploying the architecture with the Navigation based on the Nav2 package shows very coherent data. The Robot has used the same route through every iteration of the simulation, with the exception of one run in Coordinate Set 1 (see Figure 4.3a). Near the destinations, the paths show some slight variance, which is attributed to the robots orientation when aligining its position to the correct coordinates.



(a) Coordinate Set 1

(b) Coordinate Set 2

(c) Coordinate Set 3

(d) Coordinate Set 4

Figure 4.3: Heatmaps for Navigation using the Nav2 package. Separated by Coordinate Sets with 20 iterations each. (n=20)

Another contributing factor to the frayed path is the robot's need to turn at the be-

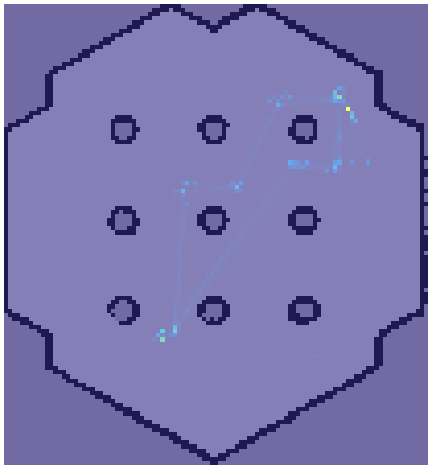
ginning of each iteration. The direction of the robot's turn appears random, causing significant variations in the time taken and the turning radius. Nevertheless, table 4.1 shows minimal deviation in distance traveled in both distance traveled and time used, with the largest deviation in distance traveled being approximately 0.47 meters and the longest deviation in time used being roughly 1.6 seconds.
To summarize the results, the long-established and relatively sophisticated Nav2 package offers excellent performance for navigation on the Turtlebot3 platform, especially when compared to the Custom Navigation system.

Coordinate Set	Mean Distance Traveled (m)	Standard Deviation Distance Traveled (m)	Mean Time Used (s)	Standard Deviation Time Used (s)
0	5.306683	0.386026	27.932736	1.595587
1	4.065978	0.471940	21.998405	1.201836
2	4.395432	0.387008	22.895346	1.303297
3	4.173309	0.385454	22.000601	1.200653

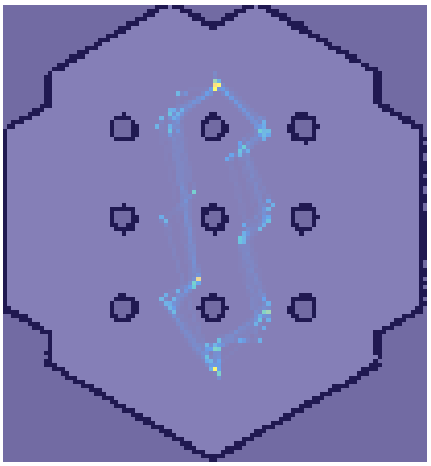
Table 4.1: Navigation using Nav2 package. Mean and standard deviation for distance traveled and time used. (n=20)

4.3.2 Custom Navigation

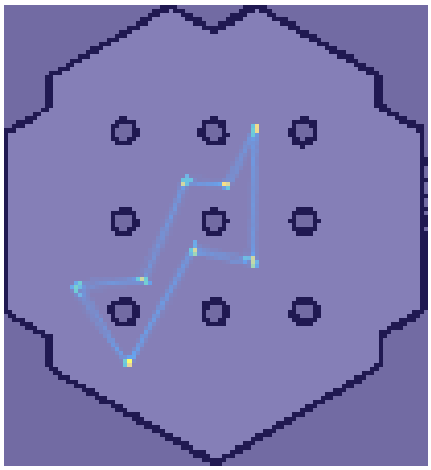
The heatmaps for Custom Navigation are shown in Figure 4.4. The data demonstrates consistent pathfinding, despite the differences between the paths from start to goal and from goal to start. Each subfigure is labeled to indicate the direction of each path. Given the less precise nature of Custom Navigation compared to Nav2, the paths appear more spread out. This is due to the "turn_by_angle" function of the Hardware Control layer, which is not as accurate as the turning commands issued by the nav2 package. Nevertheless, the paths are consistent and the given coordinate is reached reliably.



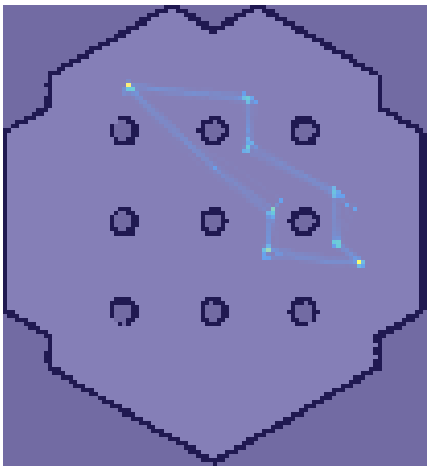
(a) Coordinate Set 1. Right path: Direction Start to Goal. Left path: Direction Goal to Start



(b) Coordinate Set 2. Right path: Direction Goal to Start. Left Path: Direction Start to Goal.



(c) Coordinate Set 3. Right path: Direction Start to Goal. Left path: Direction Goal to Start



(d) Coordinate Set 4. Right path: Direction Start to Goal. Direction Goal to Start.

Figure 4.4: Heatmaps for Custom Navigation. Separated by Coordinate Sets with 20 iterations each. (n=20)

The logged data regarding distance traveled and time used to reach coordinates supports the previously mentioned statement. To facilitate more meaningful deductions, the data for the two paths in each heatmap is divided into two tables (see tables 4.2 and 4.3). Both tables show similar path length when compared to the Nav2 Navigation. For the direction from start to goal, the Custom Navigation shows even shorter distance traveled, whereas the direction from goal to start displays longer distances. Standard deviation is down for both paths, when compared to the Navigation using Nav2, indicating an even more consistent route from coordinate to coordinate. On the other hand, both directions reveal longer time used than the Nav2 navigational system. Considering the number of turns required to follow the Custom Navigation paths compared to the Nav2 system, this is likely the cause of the slower performance. Furthermore, the large standard deviation of approximately 5.12 seconds for coordinate set 1 in table 4.3 is likely caused by an outlier (see 6.6 in the Appendix).

Coordinate Set	Mean Distance Traveled (m)	Standard Deviation Distance Traveled (m)	Mean Time Used (s)	Standard Deviation Time Used (s)
1	3.749091	0.061393	30.548182	1.804333
2	4.180000	0.172337	40.887273	2.639118
3	4.020909	0.194600	31.364545	2.103618
4	4.003636	0.110025	31.101818	1.603950

Table 4.2: Mean and standard deviation for distance traveled and time used for Custom Navigation. Direction: Start to Goal. (n=10)

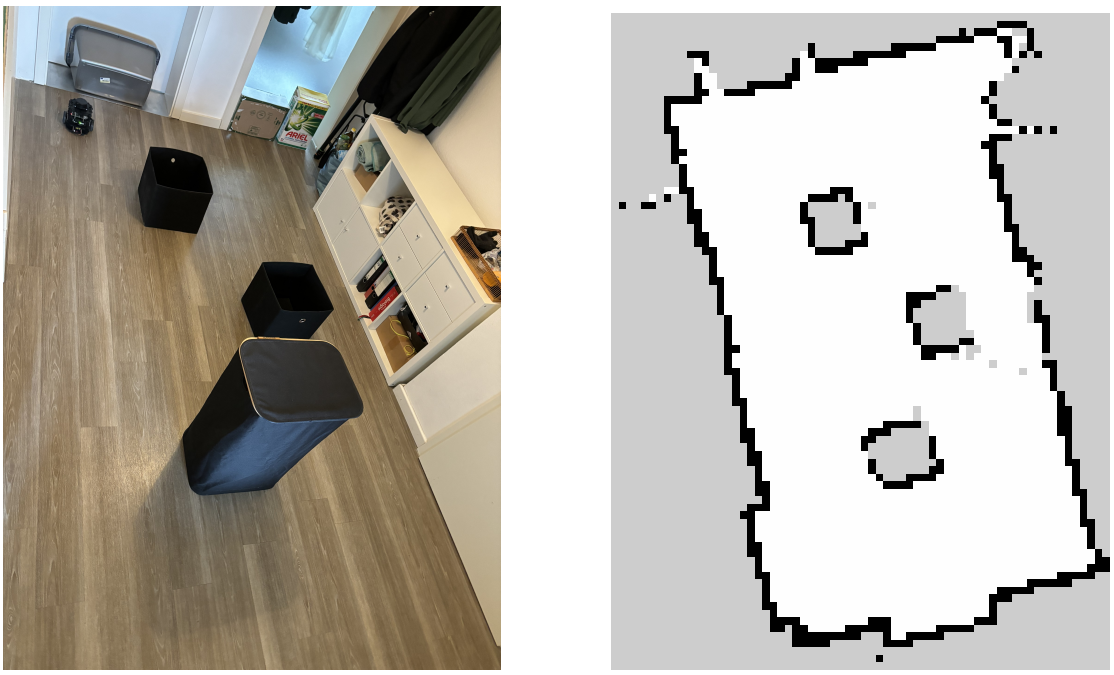
Coordinate Set	Mean Distance Traveled (m)	Standard Deviation Distance Traveled (m)	Mean Time Used (s)	Standard Deviation Time Used (s)
1	4.438000	0.152082	37.835000	5.122185
2	4.214000	0.093	40.711000	1.834000
3	4.428000	0.090897	40.292000	1.962826
4	4.214000	0.092999	40.711000	1.833791

Table 4.3: Mean and standard deviation for distance traveled and time used for Custom Navigation. Direction: Goal to Start. (n=10)

In conclusion, although the Custom Navigation may be inferior compared to the Nav2 package regarding speed, it still ensures reliable navigation to the desired coordinates. Additionally, there are instances where it outperforms the Nav2 package in terms of the distance traveled.

4.4 Real World Application

Applying the layered architecture to a real environment is certainly among the most difficult tasks during the creation of this thesis. While simulations provide a controlled and repeatable setting for initial development and testing, they cannot fully capture the complexities and unpredictability of a real environment. As seen in Figure 4.5a, the robot was placed in a square room with a couple of obstacles to avoid. The lower layers, namely Hardware Control, Obstacle Avoidance, Random Explore, SLAM Toolbox and Map Change Detector, worked as expected, resulting in the creation of the map as seen in Figure 4.5b. However, transitioning all layers from simulation to a real-world scenario presented significant difficulties. Although the SLAM Toolbox provides a .yaml file containing the essential details for mapping the robot's position (published by the /odom topic) to the frame of the /map topic, the navigational systems were either dysfunctional or imprecise, depending on the iteration. The efforts to debug these issues were severely constrained by the extensive time required for setting up the entire system, which includes charging, map creation, and actual navigation attempts.



(a) Real World Setup (JPEG)

(b) Map by Turtlebot3

Figure 4.5: Real Environment and corresponding Map created by Turtlebot3

5 Discussion

The primary aim of this thesis is to develop a layered architecture that is not only performant and robust but also modular, easily adaptable and maintainable. Supposedly, the system is designed to be horizontally structured with minimal vertical dependencies to enhance modularity and facilitate a plug-and-play environment.

In order to evaluate the architecture, the first step is to examine the conformity to the proposed guidelines by Malavolta et al. (see 2.3.2 and 4.1). Overall, the architecture adheres well to said principles: The majority of message formats used are standardized ROS messages, with only one exception: the custom Priority Twist message. Many of the signaling flags used in the architecture, such as the Random Explore suppression, are simple publisiher-subscriber systems, which do not need custom message formats. The nodes follow a well-defined lifecycle and clear naming conventions, are agnostic to the use of simulators or hardware devices, and are logically grouped within their packages. It is important to note that Guideline 6 fell a little bit out of scope. Due to the relatively small amount of data and the infrequency of sensor data outages, the primary focus was directed towards other aspects. Consequently, no fallback plans, aside from retrying or waiting for the next data load, were implemented. Above statements are partially supported by the results of the ablation study and quantitative analysis. On the one hand, the ablation study 4.1 demonstrates that the system is robust when scaling upwards, indicating that adding new layers is safe and does not compromise the overall stability or performance of the architecture. On the other hand, the ablation study illustrates that creating a complex layered architecture is not feasible without vertical dependencies. These vertical dependencies become especially prevalent the lower the layer is set in the architecture, as these layers provide the most basic and essential functionalities such as movement and obstacle avoidance. A possible solution would be to overwrite those lower levels, as exemplified by the nav2 package. However, this bypass would not come without consequences, as complexity and responsibility in the higher levels would drastically increase, defeating the purpose of this architecture type. This expansion of capabilities and dominance of certain layers is also what the authors of the CLARATy architecture deemed the most problematic in past attempts to build robust architectures.

Furthermore, the successfull employment of two concurrent navigational modules underlines the architecture's modularity and adaptability, as there is no upward dependency from a lower level to the navigational layer. This means that any navigational system can be implemented, provided it either adapts to the Hardware Control to invoke movement or suppresses the Hardware Control and invokes movement itself. The difference in performance between the two navigational systems also indicates that any future adaptations of the architecture are granted the flexibility to prioritize different aspects, such as speed versus distance traveled in navigation. This allows for customization based on specific needs and goals.

Transitioning from simulation to real-world applications is a common challenge in the robotics community. Therefore, it is not surprising that the real-world implementation of the architectural system encountered some issues. Two critical insights were gathered: although no layer required codebase adaptation to transition from simulation to hardware, only the lower layers, closer to the sensors and motors, were able to provide their designated functionality. In contrast, the higher, more complex layers failed to provide their designed functionality. Nonetheless, the modularity of the architecture enabled the system to still gain basic operational capabilities.

6 Conclusion and Outlook

In this work, a layered architecture was implemented with the aim of achieving a modular design to ensure scalability, adaptability, and reliability. Furthermore, the system is intended to be robust and performant, with clear tasks designated to the specific layers in order to avoid problems other roboticists have encountered in the past.

These requirements were evaluated through adherence to guidelines proposed and weighed by a community of roboticists on GitHub, an ablation study to assess the system's robustness, and performance checks in the form of multiple navigational endeavors in both simulated and real-world environments on the Turtlebot3 platform. The analysis of the architecture confirmed its adherence to most of the guidelines. In the simulation, both navigational modules demonstrated their performance and reliability, proving the system's capabilities. Moreover, the ablation study demonstrated a certain robustness in the system. This robustness is, outside of simulations, currently exclusive to the lower layers, as evidenced by the real-world application. When transitioning from simulation to the real world, the navigational layer, the highest and most complex layer, failed to provide its intended functionality. However, since the lower levels in the system were fully functional, the modularity of the architecture was validated and the system remained to some extend operational. Furthermore, the two triangular matrices of the ablation study demonstrated that adding new layers does not interfere with existing ones. However, vertical dependencies became particularly apparent in the higher layers, indicating that creating a layered architecture without vertical linkages is a challenging task that needs further investigation.

The next step would be to improve the architecture's behavior when moving from simulation to the real world. On the one hand, this could include a better handling of odometry and map data. On the other hand, transitioning the Hardware Control from a service to an action server could improve the handling of multiple concurrent movement requests.
This leads to the next major consideration: the scalability of the architecture. Currently, only two complex layers are implemented—SLAM and navigation. By adding more sophisticated packages and layers to the architecture, one could evaluate how the

system manages the increased strain and probable rise in vertical dependencies. Investigating the aforementioned issues will have a great impact on how to improve the modular design of layered architectures, continuing the holistic approach of this thesis.

Bibliography

[1] R. Brooks. "A Robust Layered Control System for a Mobile Robot". In: *IEEE Journal on Robotics and Automation* 2.1 (1986), pp. 14–23. ISSN: 0882-4967. DOI: 10.1109/JRA.1986.1087032. URL: http://ieeexplore.ieee.org/document/1087032/ (visited on 11/06/2023).

[2] E. Coste-Maniere and R. Simmons. "Architecture, the Backbone of Robotic Systems". In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. 2000 ICRA. IEEE International Conference on Robotics and Automation. Vol. 1. San Francisco, CA, USA: IEEE, 2000, pp. 67–72. ISBN: 978-0-7803-5886-7. DOI: 10.1109/ROBOT.2000.844041. URL: http://ieeexplore.ieee.org/document/844041/ (visited on 03/07/2024).

[3] Jun-Young Jung et al. "Three-Layered Hybrid Architecture for Emotional Reactive System". In: *2008 International Conference on Control, Automation and Systems*. 2008 International Conference on Control, Automation and Systems (ICCAS). Seoul, South Korea: IEEE, Oct. 2008, pp. 2728–2731. ISBN: 978-89-950038-9-3 978-89-93215-01-4. DOI: 10.1109/ICCAS.2008.4694221. URL: http://ieeexplore.ieee.org/document/4694221/ (visited on 03/10/2024).

[4] Iuliia Kotseruba and John K. Tsotsos. "40 Years of Cognitive Architectures: Core Cognitive Abilities and Practical Applications". In: *Artificial Intelligence Review* 53.1 (Jan. 2020), pp. 17–94. ISSN: 0269-2821, 1573-7462. DOI: 10.1007/s10462-018-9646-y. URL: http://link.springer.com/10.1007/s10462-018-9646-y (visited on 03/07/2024).

[5] Pat Langley, John E. Laird, and Seth Rogers. "Cognitive Architectures: Research Issues and Challenges". In: *Cognitive Systems Research* 10.2 (June 2009), pp. 141–160. ISSN: 13890417. DOI: 10.1016/j.cogsys.2006.07.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S1389041708000557 (visited on 07/08/2024).

[6] Steve Macenski et al. "The Marathon 2: A Navigation System". Version 2. In: (2020). DOI: 10.48550/ARXIV.2003.00368. URL: https://arxiv.org/abs/2003.00368 (visited on 05/08/2024).

[7] Steven Macenski et al. “Robot Operating System 2: Design, Architecture, and Uses in the Wild”. In: *Science Robotics* 7.66 (May 11, 2022), eabm6074. ISSN: 2470-9476. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/10.1126/scirobotics.abm6074 (visited on 11/05/2023).

[8] Ivano Malavolta et al. “How Do You Architect Your Robots?: State of the Practice and Guidelines for ROS-based Systems”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. ICSE ’20: 42nd International Conference on Software Engineering. Seoul South Korea: ACM, June 27, 2020, pp. 31–40. ISBN: 978-1-4503-7123-0. DOI: 10.1145/3377813.3381358. URL: https://dl.acm.org/doi/10.1145/3377813.3381358 (visited on 11/05/2023).

[9] Allen Newell. “Précis of Unified Theories of Cognition”. In: *Behavioral and Brain Sciences* 15.3 (Sept. 1992), pp. 425–437. ISSN: 0140-525X, 1469-1825. DOI: 10.1017/S0140525X00069478. URL: https://www.cambridge.org/core/product/identifier/S0140525X00069478/type/journal_article (visited on 07/08/2024).

[10] R. Peter Bonasso et al. “Experiences with an Architecture for Intelligent, Reactive Agents”. In: *Journal of Experimental & Theoretical Artificial Intelligence* 9.2-3 (Apr. 1997), pp. 237–256. ISSN: 0952-813X, 1362-3079. DOI: 10.1080/095281397147103. URL: http://www.tandfonline.com/doi/abs/10.1080/095281397147103 (visited on 03/10/2024).

[11] Morgan Quigley et al. “ROS: An Open-Source Robot Operating System”. In: *ICRA workshop on open source software* 3.3.2 (May 2009), p. 5.

[12] Niryo Robotics. *Get Started with Niryo One ROS Stack*. Mar. 8, 2024. URL: https://niryo.com/docs/niryo-one/developer-tutorials/get-started-with-the-niryo-one-ros-stack/.

[13] Niryo Robotics. *Niryo One ROS*. Mar. 8, 2024. URL: https://github.com/NiryoRobotics/niryo_one_ros.

[14] Robotis. *E-Manual ROBOTIS*. July 8, 2024. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/opencr_setup/#opencr-setup.

[15] ROS.org. *Joint Trajectory Controller*. Mar. 5, 2024. URL: http://wiki.ros.org/joint_trajectory_controller.

[16] ROS.org. *LaserScan Message*. Mar. 5, 2024. URL: http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/LaserScan.html.

[17] ROS.org. *MoveIt Concepts*. Mar. 5, 2024. URL: https://moveit.ros.org/documentation/concepts/.

[18] ROS.org. *ROS Control*. Mar. 5, 2024. URL: http://wiki.ros.org/ros_control.

[19] ROS.org. *ROS Introduction*. Mar. 5, 2024. URL: https://wiki.ros.org/ROS/Introduction.

[20] Malte Schilling. "Autonome Systeme Und Mobile Roboter: Architectures" (Universität Münster). June 13, 2023. URL: https://www.uni-muenster.de/LearnWeb/learnweb2/course/view.php?id=69302.

[21] Bruno Siciliano and Oussama Khatib, eds. *Springer Handbook of Robotics*. Springer Handbooks. Cham: Springer International Publishing, 2016. 285-291. ISBN: 978-3-319-32550-7 978-3-319-32552-1. DOI: 10.1007/978-3-319-32552-1. URL: https://link.springer.com/10.1007/978-3-319-32552-1 (visited on 03/08/2024).

[22] Reid Simmons et al. "A Layered Architecture for Coordination of Mobile Robots". In: *Multi-Robot Systems: From Swarms to Intelligent Automata*. Ed. by Alan C. Schultz and Lynne E. Parker. Dordrecht: Springer Netherlands, 2002, pp. 103–112. ISBN: 978-90-481-6046-4 978-94-017-2376-3. DOI: 10.1007/978-94-017-2376-3_11. URL: http://link.springer.com/10.1007/978-94-017-2376-3_11 (visited on 03/10/2024).

[23] Ron Sun. "Desiderata for Cognitive Architectures". In: *Philosophical Psychology* 17.3 (Sept. 2004), pp. 341–373. ISSN: 0951-5089, 1465-394X. DOI: 10.1080/0951508042000286721. URL: http://www.tandfonline.com/doi/abs/10.1080/0951508042000286721 (visited on 07/08/2024).

[24] R. Volpe et al. "The CLARAty Architecture for Robotic Autonomy". In: *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*. 2001 IEEE Aerospace Conference Proceedings. Vol. 1. Big Sky, MT, USA: IEEE, 2001, pp. 1/121–1/132. ISBN: 978-0-7803-6599-5. DOI: 10.1109/AERO.2001.931701. URL: http://ieeexplore.ieee.org/document/931701/ (visited on 03/08/2024).

Appendix

Figure 6.1: Project Directory Tree

Coordinate Set	Distance Traveled (m)	Time Used (s)
1	5.130298	27.998535
1	4.869894	26.000022
1	5.453286	26.000290
1	4.974142	26.000858
1	5.387319	25.999217
1	5.434525	27.999560
1	5.122722	26.000072
1	5.448368	28.000674
1	5.180251	25.999200
1	5.629328	29.999985
1	5.352114	28.000899
1	5.404368	28.000448
1	5.164623	25.998860
1	5.398202	27.974977
1	5.169252	26.025875
1	5.619464	29.999127
1	5.327660	27.999871
1	5.592663	29.760104
1	5.213084	26.239961

Table 6.1: Simulated Nav2 Data - Coordinate Set 1

Coordinate Set	Distance Traveled (m)	Time Used (s)
2	5.311640	27.996754
2	4.080332	22.000514
2	3.818398	19.998617
2	3.761347	19.999943
2	3.828356	19.999907
2	3.963116	22.000073
2	3.884740	19.999886
2	4.296424	24.000082
2	3.873013	20.002518
2	3.839976	19.997566
2	3.811117	20.000155
2	3.832355	20.000026
2	3.866210	19.999748
2	3.848264	20.001474
2	3.854485	19.998312
2	3.844553	20.001375
2	3.891537	19.998770
2	3.788880	20.000184
2	3.852515	19.999944

Table 6.2: Simulated Nav2 Data - Coordinate Set 2

Coordinate Set	Distance Traveled (m)	Time Used (s)
3	7.566510	39.683514
3	4.266853	22.316174
3	4.407925	22.550062
3	4.395692	23.449422
3	4.502186	23.999573
3	4.534671	23.999918
3	4.417903	24.000293
3	4.435943	23.999711
3	4.346739	24.000350
3	4.278771	23.999819
3	4.281278	22.000120
3	4.267426	21.999995
3	4.340623	21.999955
3	4.260607	21.999936
3	4.306719	22.000231
3	4.243368	22.000067
3	4.409689	21.999886
3	4.227193	22.000825
3	4.373677	21.998897

Table 6.3: Simulated Nav2 Data - Coordinate Set 3

Coordinate Set	Distance Traveled (m)	Time Used (s)
4	6.377680	31.996258
4	4.325679	22.000514
4	4.237888	19.998617
4	3.761347	19.999943
4	4.828356	19.999907
4	4.963116	22.000073
4	4.884740	19.999886
4	4.296424	24.000082
4	3.873013	20.002518
4	3.839976	19.997566
4	3.811117	20.000155
4	3.832355	20.000026
4	3.866210	19.999748
4	3.848264	20.001474
4	3.854485	19.998312
4	3.844553	20.001375
4	3.891537	19.998770
4	3.788880	20.000184
4	3.852515	19.999944

Table 6.4: Simulated Nav2 Data - Coordinate Set 4

Coordinate Set	Distance Traveled (m)	Time Used (s)
1	3.68	27.5
1	3.67	34.55
1	3.78	30.84
1	3.74	31.07
1	3.84	31.49
1	3.75	30.1
1	3.69	28.15
1	3.73	30.63
1	3.71	30.74
1	3.84	30.5
1	3.81	30.46

Table 6.5: Simulated Custom Navigation Data. Coordinate Set 1. Direction Start to Goal

Coordinate Set	Distance Traveled (m)	Time Used (s)
1	4.37	40.3
1	4.73	50.45
1	4.17	34.83
1	4.46	41.09
1	4.35	34.04
1	4.57	35.91
1	4.42	37.18
1	4.32	32.86
1	4.49	35.64
1	4.5	36.05

Table 6.6: Simulated Custom Navigation Data. Coordinate Set 1. Direction Goal to Start

Coordinate Set	Distance Traveled (m)	Time Used (s)
2	4.07	42.79
2	4.03	41.46
2	4.13	38.9
2	3.95	39.86
2	4.46	46.09
2	4.52	43.57
2	4.2	39.07
2	4.13	36.49
2	4.11	41.97
2	4.24	40.07
2	4.14	39.49

Table 6.7: Simulated Custom Navigation Data. Coordinate Set 2. Direction Start to Goal

Coordinate Set	Distance Traveled (m)	Time Used (s)
2	4.31	44.58
2	4.18	41.22
2	4.25	38.42
2	4.14	41.25
2	4.1	40.77
2	4.07	41.59
2	4.33	41.55
2	4.31	40.03
2	4.18	39.41
2	4.27	38.29

Table 6.8: Simulated Custom Navigation Data. Coordinate Set 2. Direction Goal to Start

Coordinate Set	Distance Traveled (m)	Time Used (s)
3	3.5	27.64
3	4.03	35.86
3	4.12	32.1
3	4.09	31.76
3	3.98	29.45
3	4.02	30.5
3	4.05	31.83
3	4.23	32.48
3	3.93	29.62
3	4.21	31.97
3	4.07	31.8

Table 6.9: Simulated Custom Navigation Data. Coordinate Set 3. Direction Start to Goal

Coordinate Set	Distance Traveled (m)	Time Used (s)
3	4.24	41.0
3	4.36	39.9
3	4.4	44.22
3	4.52	40.61
3	4.55	41.64
3	4.4	39.72
3	4.44	40.64
3	4.48	40.03
3	4.39	36.73
3	4.5	38.43

Table 6.10: Simulated Custom Navigation Data. Coordinate Set 3. Direction Goal to Start

Coordinate Set	Distance Traveled (m)	Time Used (s)
4	3.86	30.27
4	4.19	34.86
4	4.03	31.79
4	3.95	31.18
4	3.94	29.1
4	4.12	32.91
4	3.86	30.46
4	4.0	30.19
4	3.91	31.11
4	4.08	30.16
4	4.1	30.09

Table 6.11: Simulated Custom Navigation Data. Coordinate Set 4. Direction Start to Goal

Coordinate Set	Distance Traveled (m)	Time Used (s)
4	4.31	44.58
4	4.18	41.22
4	4.25	38.42
4	4.14	41.25
4	4.1	40.77
4	4.07	41.59
4	4.33	41.55
4	4.31	40.03
4	4.18	39.41
4	4.27	38.29

Table 6.12: Simulated Custom Navigation Data. Coordinate Set 4. Direction Goal to Start

Declaration of Academic Integrity

I hereby confirm that this thesis, entitled *Architectural Layering in Autonomous Robotics: Modular Design and Implementation on the Turtlebot3 Platform using ROS2.* is solely my own work and that i have used no sources or aids other than the ones stated. All passages in my thesis for which other sources, including electronic media, have been used, be it direct quotes or content references, have been acknowledged as such and the sources cited. I am aware that plagiarism is considered an act of deception which can result in sanction in accordance with the examination regulations.

(Ort, Datum) (Unterschrift)

I consent to have my thesis cross-checked with other texts to identify possible similarities and to have it stored in a database for this purpose.

I confirm that I have not submitted the following thesis in part or whole as an examination paper before.

(Ort, Datum) (Unterschrift)