POLITECNICO DI TORINO

Master of Science Degree in Mechatronic Engineering

Master Degree Thesis

Enabling Autonomous Multi-Floor Navigation for Robots in ROS2 using Behavior Trees

Supervisor
Prof. Marina Indri

Supervisors at LINKS Foundation
Dott. Gianluca Prato
Dott. Francesco Aglieco

Candidate
Minchu You

December 2023

Abstract

In today's rapidly evolving world, service robots are assuming critical roles in diverse fields, ranging from manufacturing and agriculture to healthcare. These mobile robots are required to process advanced navigation capabilities, allowing them to move seamlessly within multi-floor layouts. While traditional navigation solutions have excellent development in guiding robots through single-level environments, the challenge lies in enabling autonomous robots to understand the structural elements connecting different floors, such as elevators and stairs.

The research work addressing these problems has been developed in collaboration with the LINKS Foundation, which conducts exploration across various domains, such as AI, IoT, Cyber-physical and autonomous systems. This thesis aims to investigate existing navigation approaches and explore novel solutions to recognize and interact with the environment (e.g., with automated doors, furniture, and elevators) to enable navigation across floors in a building using Automated Guided Vehicle (AGV). The Robot Operating System 2 (ROS2) Navigation2 (Nav2) stack serves as a powerful tool for robot navigation, offering a wide range of functions and algorithms for path planning and obstacle avoidance. Additionally, behavior trees have emerged as a high-level control framework for designing and modelling the robot actions, enabling the implementation of complex behaviors while providing insight into the execution process.

This thesis extends the existing open-source ROS2 libraries by incorporating behavior-tree based behaviors and navigation planners/controllers to support navigation across floors. Simulation has been tested in Gazebo and Rviz with the two-storey office building at LINKS in Turin chosen as the environment. Following that, the application of these concepts to real-world robot navigation and a discussion on their applicability are included.

Contents

1 Introduction 4

1.1 Researches on multi-floor navigation . . . 5

1.2 ROS2 framework . . . 8

1.2.1 Main innovations . . . 8

1.2.2 Node-based architecture . . . 9

1.3 Navigation2 stack . . . 11

1.3.1 Navigation server . . . 12

1.3.2 TF tree . . . 13

1.3.3 Gazebo and Rviz . . . 15

2 Behavior trees 17

2.1 Behavior trees concepts . . . 17

2.1.1 History . . . 17

2.1.2 Nodes and Trees . . . 19

2.2 Behavior trees in ROS2 . . . 24

2.2.1 Behavior-tree library . . . 24

2.2.2 Behavior-tree Navigator . . . 28

3 Multi-floor navigation 32

3.1 Behavior tree plugin . . . 35

3.1.1 Action node plugin . . . 36

3.1.2 Decorator node plugin . . . 37

3.1.3 Custom behavior tree . . . 39

3.2 Navigator plugin . . . 41

3.3 Elevator server . . . 42

3.3.1 Open door service . . . 42

3.3.2 Move to floor service . . . 43

3.4 Simulation . . . 43

4 Implementation 49

4.1 SwitchBot . . . 49

4.1.1 MQTT protocol . . . 51

4.1.2 SwitchBot server . . . 53

4.2 Turtlebot3 . . . 57

4.2.1 TurtleBot3 Waffle Pi 57
4.3 Test 58
4.3.1 Map creating 59
4.3.2 Real environment test 63

5 Conclusion 67

Chapter 1

Introduction

In the realm of rapidly evolving robotics and autonomous systems, navigating through complex environments with precision and adaptability is a great challenge. While traditional navigation solutions have been well-developed in guiding robots through single-level environments [1], the demand for robots to operate in multifloor layouts has grown significantly. Such environments typically consist of several flat areas connected by elevators or stairs, requiring robots to make more intelligent and innovative solutions in the navigation process.

This thesis is developed in cooperation with LINKS Foundation, a non-profit organization that aims to promote, lead and strengthen the innovation processes in various areas such as: Artificial Intelligence, connected systems and IoT, cybersecurity, advanced calculation and satellite systems. The two-storey office building of LINKS in Turin serves as the simulation environment for multi-floor navigation. The structure's details and further information are provided in the main body of the text.

The Robot Operating System 2 is a set of software libraries and tools that are used to build robot applications. However, the navigation stack it provided does not support movement in complex multi-floor environments. This work enhances the ROS2 navigation stack by extending the libraries to enable autonomous robots to traverse between different floors by applying a novel behavior tree-based control system. Real-world scenarios are tested following successful simulations to evaluate the practicability of the approach.
Here is a brief description about the content of the chapters.

Chapter 1: This chapter explores the history of multi-floor navigation, outlining the framework of the ROS2 and its navigation stack.

Chapter 2: This chapter provides the concept of behavior trees and their development in robotics, with a particular focus on the structure of behavior trees and their integration with the navigation system.

Chapter 3: This chapter explains the limitations of the default navigation system and presents the method of integrating new nodes into the behavior tree library to facilitate an autonomous multi-floor navigation process.

Chapter 4: This chapter discusses the simulation method applied in real-world scenarios to evaluate the practicality of the multi-floor navigation process.

1.1 Researches on multi-floor navigation

The concept of robot navigation has been put forward in the 20th Century. Early robotic systems relied heavily on physical markers to define paths, and these were the first instances of programmed navigation [2]. These paths were basic and required a simple environment where changes were minimal and predictable. The first mobile robot (Fig. 1.1), known as 'Shakey the Robot' [3], demonstrated basic navigation and obstacle avoidance capabilities. It utilized a combination of basic movement and cameras to interact with its environment. The robot could perform tasks in a simple environment, moving around and manipulating objects based on its perception and internal map of the world.

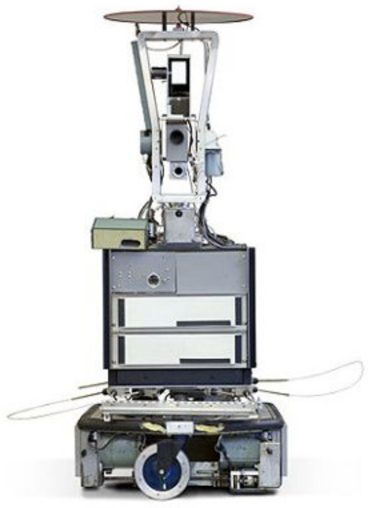


Figure 1.1: Shakey the Robot [3]

In these early stages, navigation systems were constrained by the limited processing ability [4]. Robots processed large amounts of sensor data to understand their surroundings, which was a challenge for the computation capabilities of that time. Despite the limitations, these efforts established the foundation for modern navigation systems, presenting the importance of perception and processing in robotics.

In the 1970s and 1980s, advancements in sensor technology enabled robots to detect nearby objects with higher precision and to create more accurate maps. The development of the Simultaneous Localization and Mapping (SLAM) algorithm attracted great attention during the 1980s and 1990s. Early SLAM algorithms were based on Extended Kalman Filters (EKF) [5], which integrated noisy sensor data to produce estimates of the robot's position and the map of the environment. As these techniques evolved, they overcame the challenges of robot localization in areas with unreliable or unavailable GPS signals by using landmarks to generate maps and determine the robot's position relative to these points of reference (Fig. 1.2).

The development of more advanced sensors, like Light Detection and Ranging (LIDAR), changed the field. LIDAR provided high-resolution distance measurements, allowing for more detailed and accurate maps [7]. These advancements in hardware required

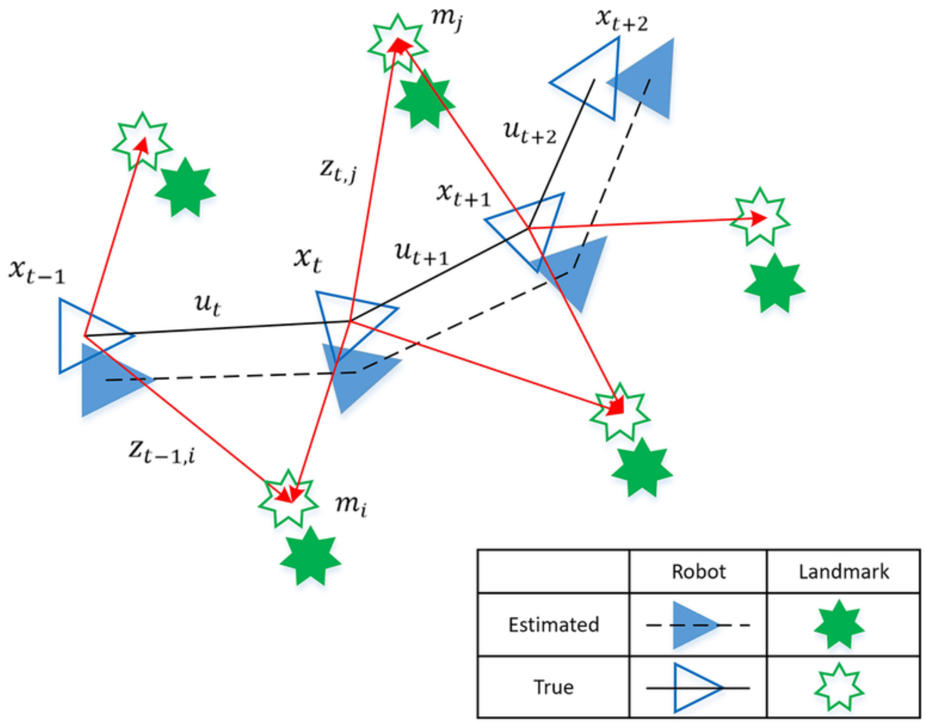


Figure 1.2: Landmark-Based navigation using SLAM [6]

improvements in computational methods. Graph-based SLAM methods were developed, allowing for more robust performance in larger and more complex environments [8]. The work of Kurt et al. [9] presented an approach for navigation in hybrid maps consisting of a topological graph overlaid with local occupancy grids, which can be efficiently optimized for very large environments and deal with changes to the map.

The rise of autonomous mobile robots appeared in the late 20th century. Robots began transitioning from predetermined paths and manual control to true autonomy, making decisions based on sensor input and internal models of the world. A significant development in this field was the invention of Dijkstra's algorithm, followed by the A* algorithm, which built upon it. Dijkstra's algorithm was developed by Edsger W. Dijkstra, which is a classic algorithm for finding the shortest path in weighted graphs and networks [10]. A* was created as part of the Shakey project by Peter Hart et al. [11] It combines the best features of Dijkstra's algorithm and achieves better performance by using intuitive strategies to guide its search.

Multi-floor navigation presents unique challenges in the field of robotics due to the three-dimensional complexity it introduces. Robots must not only navigate flat surfaces but also recognize and traverse vertical facilities, like stairs and elevators. The introduction of 3D SLAM technologies allowed robots to perceive and map environments in three dimensions. This improvement was crucial for multi-floor navigation. Researchers like Andrew [12] made significant contributions to this field by integrating vision-based and range-based sensors to create three-dimensional maps. Additionally, Nitin et al. [13]

introduced the concept of using a cost graph to represent a multi-floor structure, thereby enabling autonomous multi-floor navigation for a team of robots. Within this framework, a global planning mechanism calculates and compares the weights of each path, considering both the travel distance within the same floor and the cost of moving to a different floor via stairs or an elevator.

Today, multi-floor navigation is no longer limited to the research laboratories but is significantly enhancing efficiency and functionality in various domains. In the service and hospitality industry, robots like Savioke's Relay (Fig. 1.3a) are becoming common [14]. These robots can autonomously deliver items to guests' rooms in hotels, navigating corridors and using elevators. Similarly, in hospitals, robots such as Aethon's TUG (Fig. 1.3b) autonomously transport medications and supplies, interfacing with elevators and automatic doors to move between floors efficiently [15].

(a) Relay Autonomous Delivery Robot [16] (b) TUG Autonomous Mobile Robot [17]

Figure 1.3

In this implementation, the ROS2 Navigation2 packages are employed to execute the multi-floor navigation process. An elevator server is utilized to transport the robot between floors. The main innovations include:

- Development of an improved BT (Behavior-Tree) navigator integrated within the ROS2 Navigation2 framework, capable of identifying whether the destination is on the current floor or a different one, and adjusting the navigation strategy appropriately.

- Implementation of a decision-making process within the navigation system, allowing the robot to recognize and adapt to multi-floor environments.

- Utilization of behavior trees to control the robot's actions, ensuring smooth and autonomous navigation across various floors of a structure.
- Introduction of a custom elevator server that communicates with the robot, helping the use of elevators for efficient vertical traversal in buildings.

1.2 ROS2 framework

ROS2, or Robot Operating System 2, represents a significant evolution in the landscape of robotics software development. It is an open-source middleware that is specifically designed to provide a robust framework for the complex and varied demands of robotic software engineering. As the successor to ROS1, ROS2 preserves the original's capacity to promote the development and management of robot systems but also solves and fixes many of the limitations previously encountered [18].

With ROS2, engineers and researchers can efficiently manage the lifecycle of their robot's software, from initial development to deployment and maintenance. The system architecture of ROS2 is designed with a focus on distributed computing, enabling robots to operate autonomously within a networked framework.

1.2.1 Main innovations

The inception of the Robot Operating System, commonly referred to as ROS, marked an important moment in the field of robotics. ROS1 revolutionized the way that robotic applications were developed and deployed. Nevertheless, with the rapid advancement of the robotic field, ROS1 was not enough to support the usage in higher-demanding systems because of its limitations. ROS2, born out of the need for innovation and adaptation, has emerged as a significant evolution, addressing these limitations and accommodating to the dynamic and diverse requirements of modern robotic systems. Notable advancements in ROS2 include:

- Real-Time Capabilities. One of the most notable innovations in ROS 2 is its ability to support real-time capabilities, addressing the limitations of ROS 1 in this regard [19]. The need for real-time performance became increasingly significant as robots moved into more safety-critical and time-sensitive applications, such as autonomous vehicles and industrial robots.
- Lifecycle Management. ROS2 has an improved focus on the lifecycle management of nodes, which makes it easier to manage the state and transitions of various parts of a robotic system. By following a well-defined state machine, nodes behave predictably, which is essential for debugging and for systems that require high reliability.
- Middleware Abstraction. ROS2 introduced middleware abstraction as a fundamental feature, allowing developers to decouple the middleware from the core of the framework. This innovation enables ROS2 to ingratiate to different deployment needs by supporting various middleware implementations (e.g., DDS, FastRTPS) [20]. The difference structure between ROS1 and ROS2 can be seen in Fig. 1.4.

- Security Enhancements. ROS2 introduces several significant security enhancements to make it suitable for applications where security is a top priority. The mechanisms for authentication ensure that nodes can verify the identity of the communicating process [21]. This helps prevent unauthorized access and data tampering.

- Multiple Language Support. ROS1 mainly centered around the use of the Python programming language for scripting, while ROS2 introduces support for multiple programming languages, including C++. This enables the flexibility for developers to choose the language that best suits the requirements of their robotic application to enhance the performance.

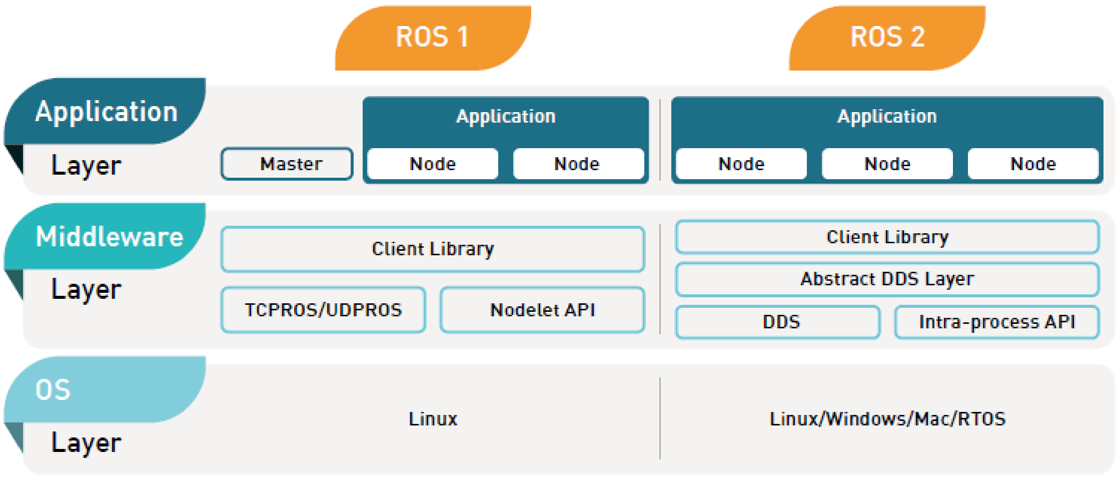


Figure 1.4: The difference between ROS1 and ROS2 [22]

1.2.2 Node-based architecture

As a powerful framework for robotic development, ROS2 offers a comprehensive set of core functions and capabilities that allow developers to create, control, and coordinate complex robotic systems. A node-based architecture serves as the fundamental structure that underpins the entire framework. In ROS2, a robotic system is composed of individual units known as nodes [23]. These nodes form the building blocks of the system, each with a specific role and responsibility. These nodes enable seamless communication, coordination, and control of robotic components. Here are some applications:

- Publish-Subscribe Communication. Publish-Subscribe communication facilitates the exchange of data between nodes within a robotic system. In this communication model, nodes are categorized into publishers and subscribers. Publishers broadcast messages, while subscribers receive and process those messages. This paradigm offers an efficient, event-driven method for nodes to share information, making it a base of ROS2's communication architecture. The process is shown in Fig. 1.5:

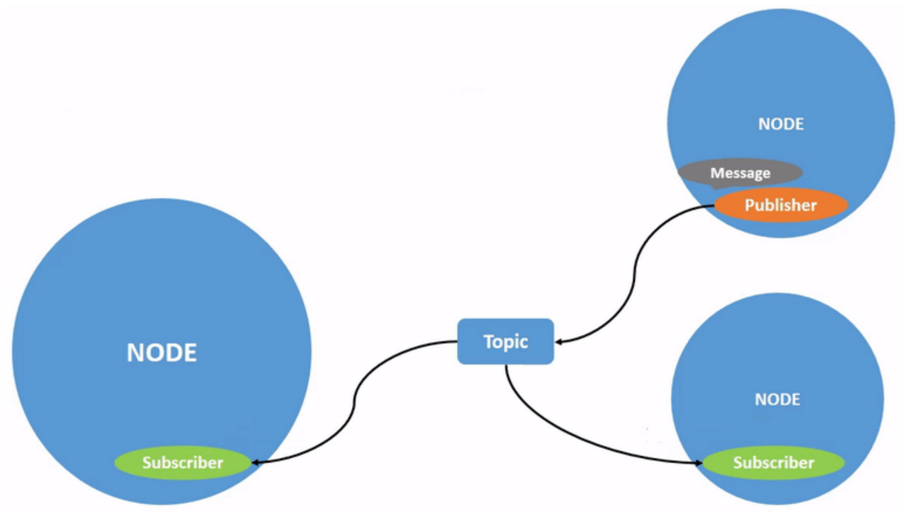


Figure 1.5: Publish-Subscribe Communication [24]

- Service-Client Communication. Service-Client communication facilitates request-response interactions between nodes. This communication model allows nodes to provide and consume services, where the service server offers a particular functionality, and another node (the service client) can request and receive that functionality. Service-Client communication is essential for arranging actions, making real-time requests, and executing tasks that involve multiple steps. The process is presented in Fig. 1.6:

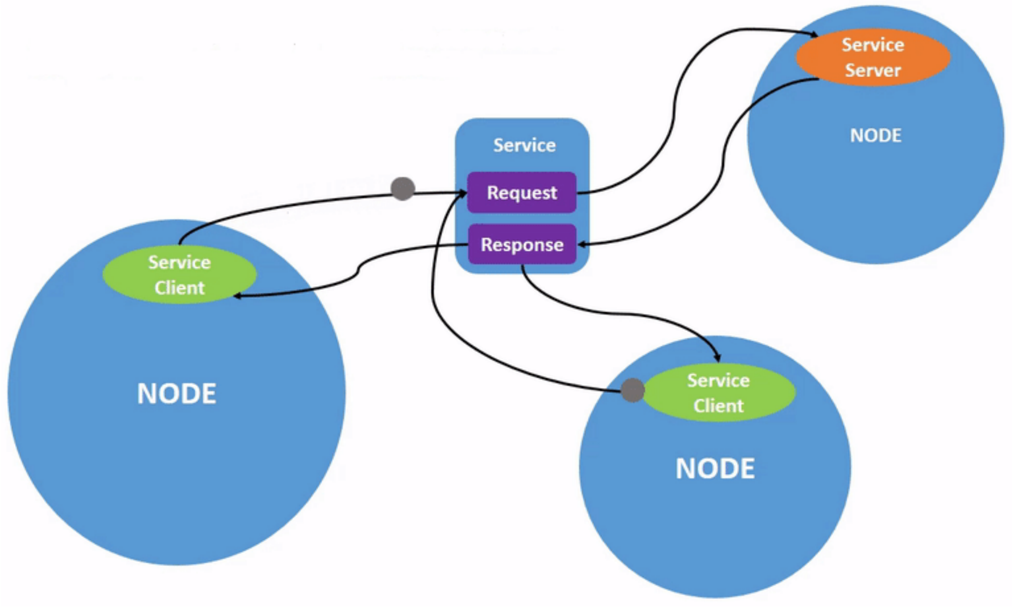


Figure 1.6: Service-Client Communication [25]

- Action Servers and Clients. Action servers and clients facilitate multi-step, goal-oriented actions. Actions are more complex compared with the request-response model of Service-Client communication, typically involving tasks that require multiple steps or feedback. This process consists of three parts: a goal, feedback and a result. In this communication model, one or more nodes act as action servers, offering specific action capabilities, while others act as action clients, requesting and monitoring the progress of these actions. The process is displayed in Fig. 1.7:

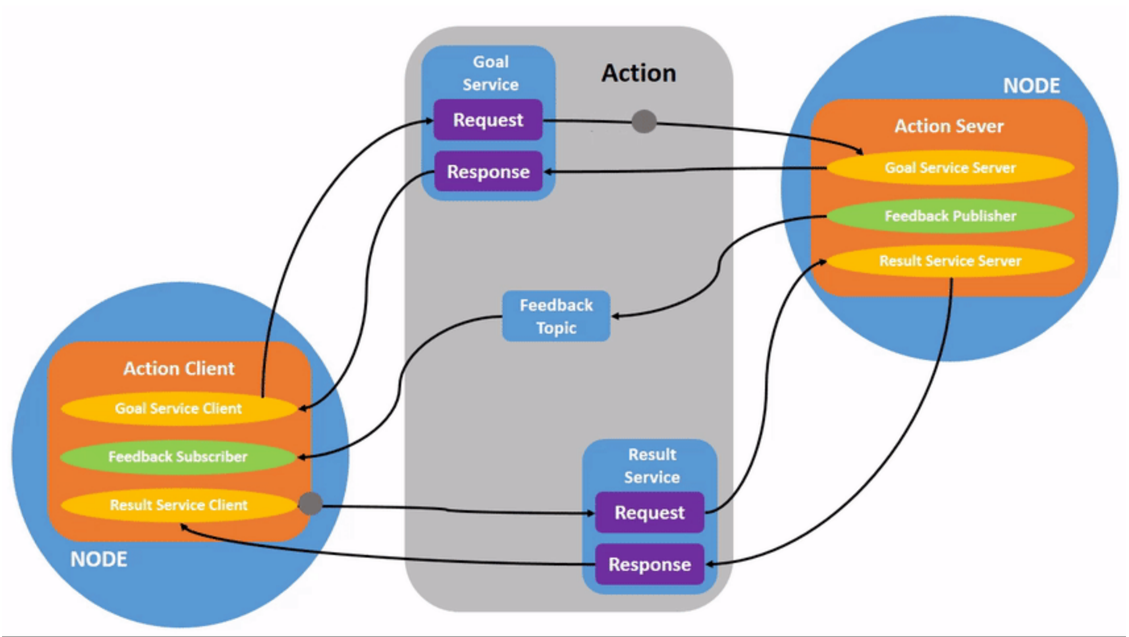


Figure 1.7: Action Servers and Clients [26]

- Parameter Management. Parameter management enables nodes to store and share configuration settings, variables, and other runtime parameters. Parameters are used to adjust and configure the behavior of nodes and the overall robotic system. ROS2 provides a centralized and standardized way to manage parameters (e.g., integers, floats, booleans, strings), making it easier to maintain and reconfigure nodes and applications.

1.3 Navigation2 stack

The Navigation2 stack is a critical and essential part of ROS2 framework. It addresses the significant challenges of enabling robots to navigate autonomously in diverse and unpredictable environments. As a core element of ROS2's ecosystem, Navigation2 includes a set of packages and libraries that equip robots with the ability to move, sense, and react in real time. The package provides an extensive collection of tools and algorithms to handle complex robotic tasks, such as path planning, obstacle avoidance and sensor data processing. Moreover, it makes use of the ROS2's architectural foundations, including its node-based architecture and communication functions, which enhance its effectiveness.

1.3.1 Navigation server

The navigation server serves as the foundation of the overall structure of Navigation2 stack. It is designed to guide robots through real-world spaces and is responsible for planning, controlling, and executing the autonomous movement of robots in diverse environments [27]. The structure of Navigation2 is present in Fig. 1.8. Controllers and planners are the "brain" of a navigation process. Recovery behaviors deal with the problem when robots encounter difficulties, offering solutions to potential issues and enhancing the system's fault tolerance. Smoothers optimize the planned path. A detailed introduction to these components and their roles in the navigation process is provided in this section.

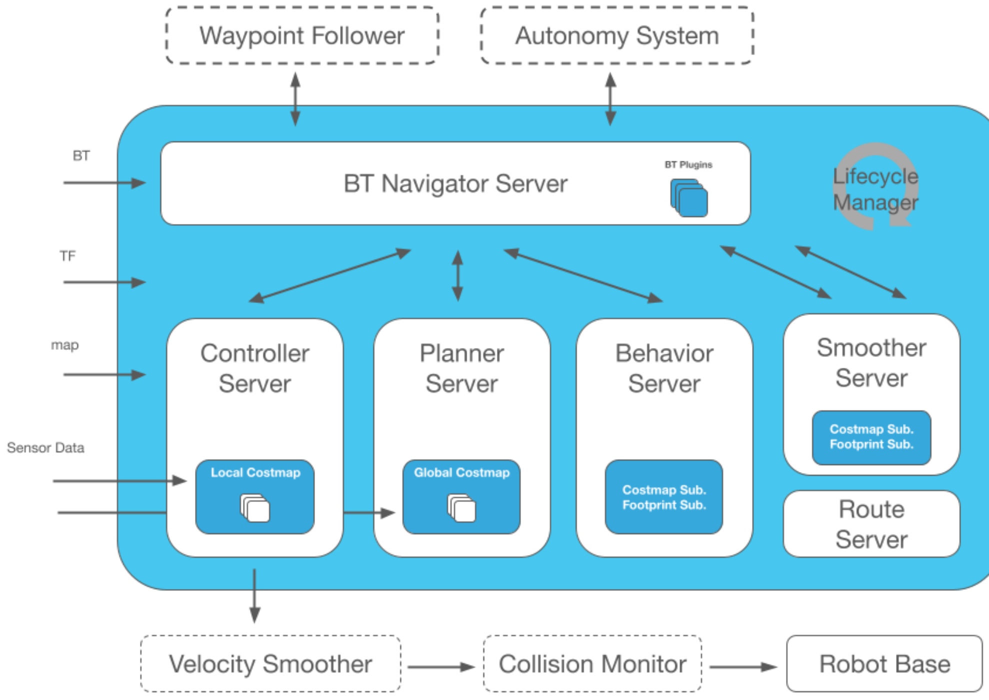


Figure 1.8: Nav2 architecture [28]

Planner Server

The primary function of the planner server is to consider the current location of a robot and its target position in terms of coordinates, and then to calculate optimal routes for guiding the robot to that predefined destination within a complex environment. The routes could be the shortest path, a complete coverage path, or along sparse or predefined routes, depending on the configuration of the planners. The objective of all routes is to enable robots to navigate safely, efficiently, and intelligently.

Controller Server

The controller server, previously known as local planners in ROS1, is responsible for ensuring the precise execution of navigation routes generated by the planner server. While the planner server calculates high-level navigation paths, the controller server translates these paths and breaks them down into a series of control commands that guide the robot's movements. Its primary function is to supervise the execution of these plans, adjusting the robot's trajectory as it navigates through the dynamic and challenging environment.

Behavior Server

The behavior server is integrated with the behavior trees, allowing for a dynamic approach to managing robot actions. One of the significant aspects of the behavior server is handling recovery behaviors. The recovery behavior server is designed to deal with unknown or failure conditions that robots may encounter during navigation. The obstacles may include dynamic objects, temporary blockages, or items that did not initially appear on the navigation map. While the planners and the controllers focus on guiding the robot through a known and expected environment, the recovery server manages the unforeseen challenges to ensure that the robot can recover gracefully. This could involve backing up or spinning in place, attempting an alternative path, or even moving from a poor location into free space.

Smoother Server

The general task of a smoother server is to receive a path from the planner server and output an enhanced version. It concentrates on improving the path's quality by considering factors that may influence the robot's movement, such as kinematics and acceleration. One of the crucial functions of the smoother server is to solve issues related to abrupt movements that may arise when following the navigation path. It aims to minimize these sudden changes in velocity or direction, and to increase the distance from obstacles and high-cost areas.

Waypoint Following

Waypoint following is often utilized in conjunction with planners. A path is typically represented as a sequence of waypoints that define the route for the robot. The primary function of waypoint following is to provide precise control of the robot's movement.

1.3.2 TF tree

The TF tree (transform tree) in Nav2 is a data structure that organizes the frames in a robot system and manages the relationships between them dynamically [29]. Each frame represents a coordinate system through which the positions of robots, sensors, and other objects can be expressed.

The TF library in ROS2 is responsible for maintaining the TF tree. It works by broadcasting and listening to transform messages that describe the relationships between different frames. These messages contain the relative positions and orientations between a child frame and its parent frame.

For instance, a simple robot (Fig. 1.9) might have a "base_link" frame representing its mobile base center, and a "base_laser" frame for the center of a mounted laser sensor. Data from the laser is transformed from the "base_laser" frame to the "base_link" frame to help the robot avoid obstacles. This relationship is defined in the TF tree,

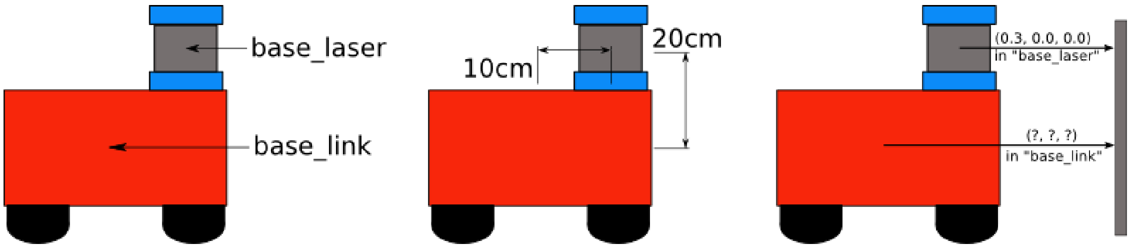


Figure 1.9: Base_link and base_laser frame in a robot

The structure of the TF tree used in the navigation process is presented in Fig. 1.10, with explanations provided for some of the main frames.

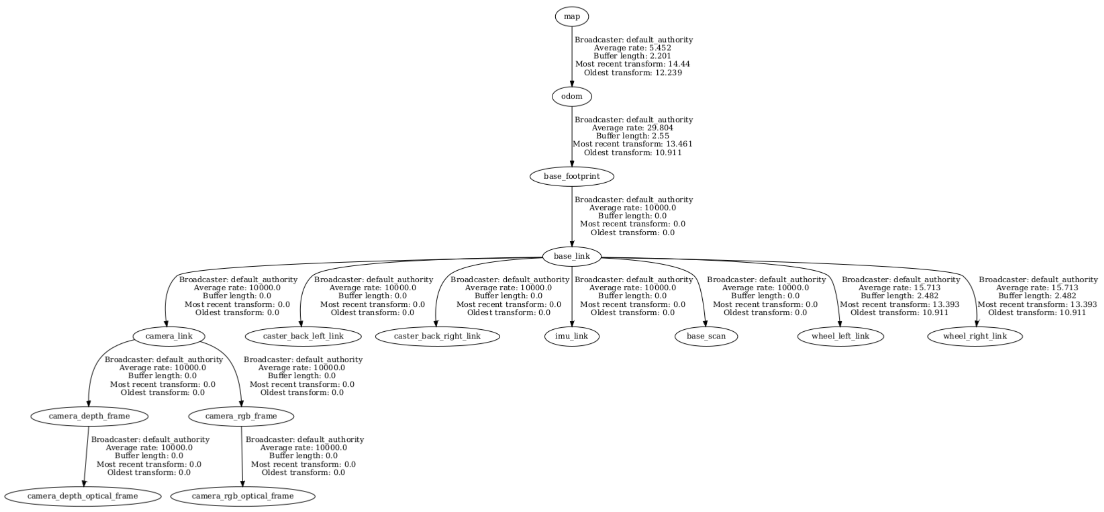


Figure 1.10: TF tree in Navigation2

- map frame: A global frame that represents a stationary reference on the map where all static objects are assumed to be fixed.
- odom frame: A local frame that represents the position and orientation of the robot based on odometry, which can move over time and provides relative motion information.
- base_footprint frame: This is usually a two-dimensional projection of the robot on the ground. It's the reference for the robot's contact with the floor or the ground.

- base_link frame: This is the robot's main body frame, from which positions and motions of all other parts of the robot are defined.
- base_scan frame: This frame is associated with a laser scanner or LIDAR sensor on the robot. It's the reference point from which scan data is measured.

1.3.3 Gazebo and Rviz

In the realm of robot navigation, the importance of testing and validation can not be ignored. Ensuring that robots navigate in the environment with accuracy and safety is a complex task that requires extensive experimentation. To facilitate this, some powerful tools are utilized: Gazobo and Rviz.

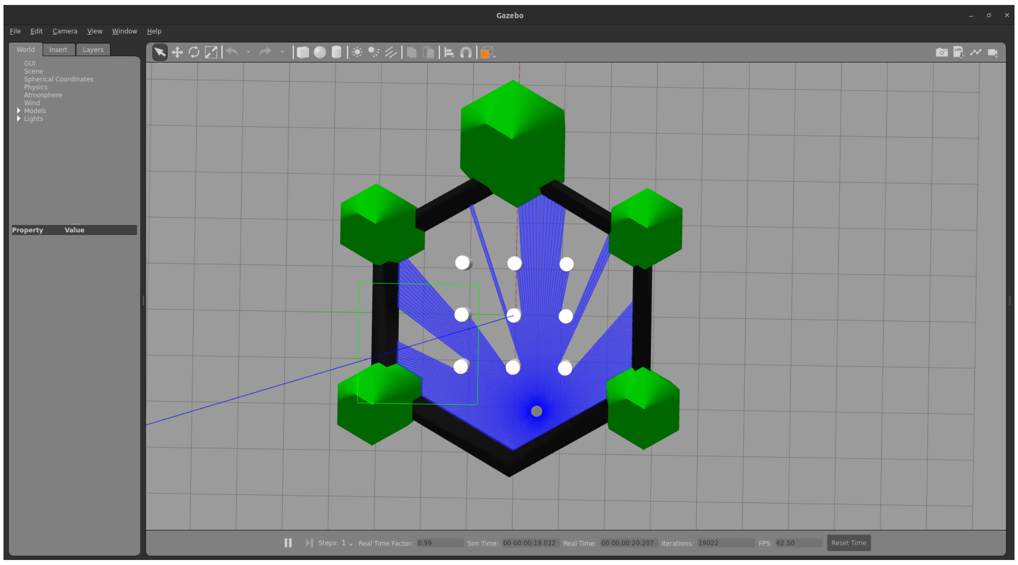
Gazebo is an open-source, 3D robotics simulator that has gained popularity in the research and development communities. It offers a comprehensive, physics-based simulation environment, enables the modeling of complex building structure and robotic platforms [30]. It also supports the integration of various sensors, such as LiDAR and cameras, which makes it a powerful tool for testing perception capabilities.

ROS Visualization (Rviz) is another necessary tool that complements Gazebo in the simulation environment. It enables real-time visualization of robot sensor data, assisting in the assessment of the navigation process performance [31]. With its interactive interface, users can manipulate robots and environments during the development and testing phases.

Fig. 1.11 is an example of the interface of Gazebo and Rviz. It is better to supply Rviz with a map for robot navigation. The map typically consists of two files: a `PGM` file and a `YAML` file. The `PGM` file is a gray scale image where black indicates obstacles, white represents free space, and shades of gray represent unknown areas. This file serves as the visual layout of the map. The accompanying `YAML` file (Listing 1.1) contains detailed data for the `PGM` file, including the path to the image file, the map's spatial resolution, a 3-element array that defines the 2D pose of the lower-left pixel in the map, and thresholds for occupied and free spaces. Pixels with an occupancy probability above the 'occupied' threshold are considered fully occupied, while those below the 'free' threshold are considered completely free.

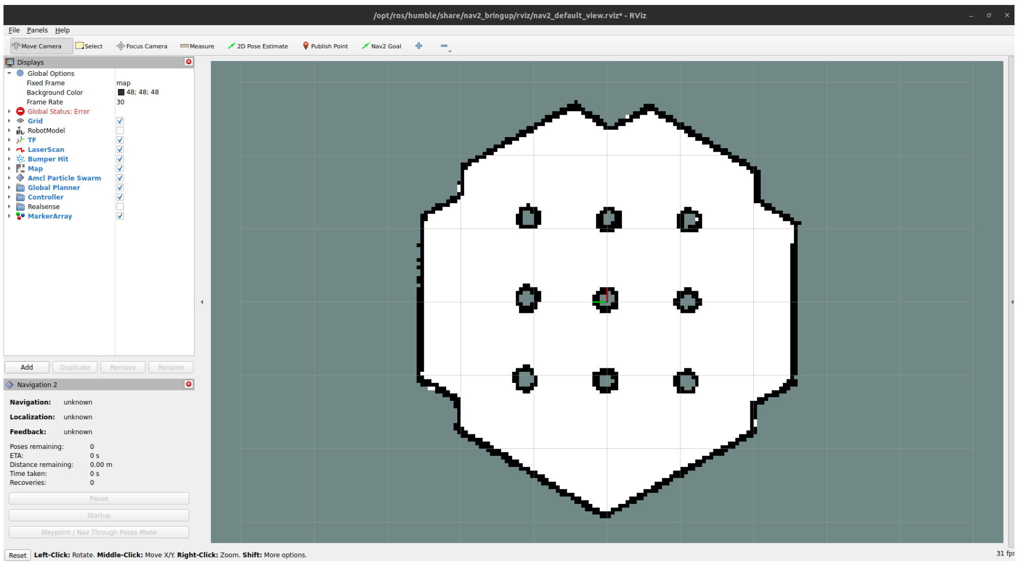
```
image: turtlebot3_world.pgm
resolution: 0.050000
origin: [-10.000000, -10.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Listing 1.1: Turtlebot3_world.yaml



Steps: 1 Real Time Factor: 0.99 Sim Time: 00 00:00:19.022 Real Time: 00 00:00:20.207 Iterations: 19022 FPS: 62.50

(a) Gazebo interface



Global Options
Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30
Global Status: Error
Grid ✓
RobotModel
TF ✓
LaserScan ✓
Bumper Hit ✓
Map ✓
Amcl Particle Swarm ✓
Global Planner ✓
Controller ✓
Realsense
MarkerArray ✓

Navigation: unknown
Localization: unknown
Feedback: unknown

Poses remaining: 0
ETA: 0 s
Distance remaining: 0.00 m
Time taken: 0 s
Recoveries: 0

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:** Zoom. **Shift:** More options.

(b) Rviz interface

Figure 1.11: Tools utilized in the simulation

Chapter 2

Behavior trees

The aim of this chapter is to give a comprehensive understanding of Behavior Trees. It begins with a historical overview, tracing the origins of behavior trees from the video game industry and their transition into the domain of robotics and AI. Then a detailed introduction about the nodes and structure of behavior trees is provided. Following this, the chapter focuses on the integration of behavior trees and ROS2, understanding the tools, libraries, and methodologies that have emerged within the ROS2 system to facilitate the implementation of BTs.

2.1 Behavior trees concepts

A Behavior Tree is a hierarchical and graphical representation of decision-making logic, mainly used to control the flow of decisions and actions in artificial agents. The tree structure includes a series of tasks and breaks them into nodes. Each node represents a decision or an action, and the tree effectively decides the sequence or priority in which these tasks should be executed.

One of the obvious advantages of behavior trees, compared with traditional decision-making structures like Finite State Machines (FSMs), is their modularity and flexibility [32]. Behavior trees allow for additions, modifications, or deletions of nodes without interfering with the overall flow of the tree, ensuring consistent performance when the environment or state changes.

2.1.1 History

The roots of behavior trees can be traced back to the video game industry of the early 2000s [33]. Game developers had an increased demand for non-player characters (NPCs) to exhibit more complex and realistic behaviors. They needed NPCs to react dynamically to player actions, environmental changes, and even other NPCs. At that time Finite State Machines were the model for NPC's behavior. However, as the number of states increased to accommodate the more sophisticated behaviors, FSMs became not only harder to manage but also difficult to modify or add new behaviors. As in the example illustrated in Fig. 2.1, when adding or removing a state, it is necessary to change the conditions of all other states that have a connection to the new or old one, which makes it easier to pass errors unconsciously.

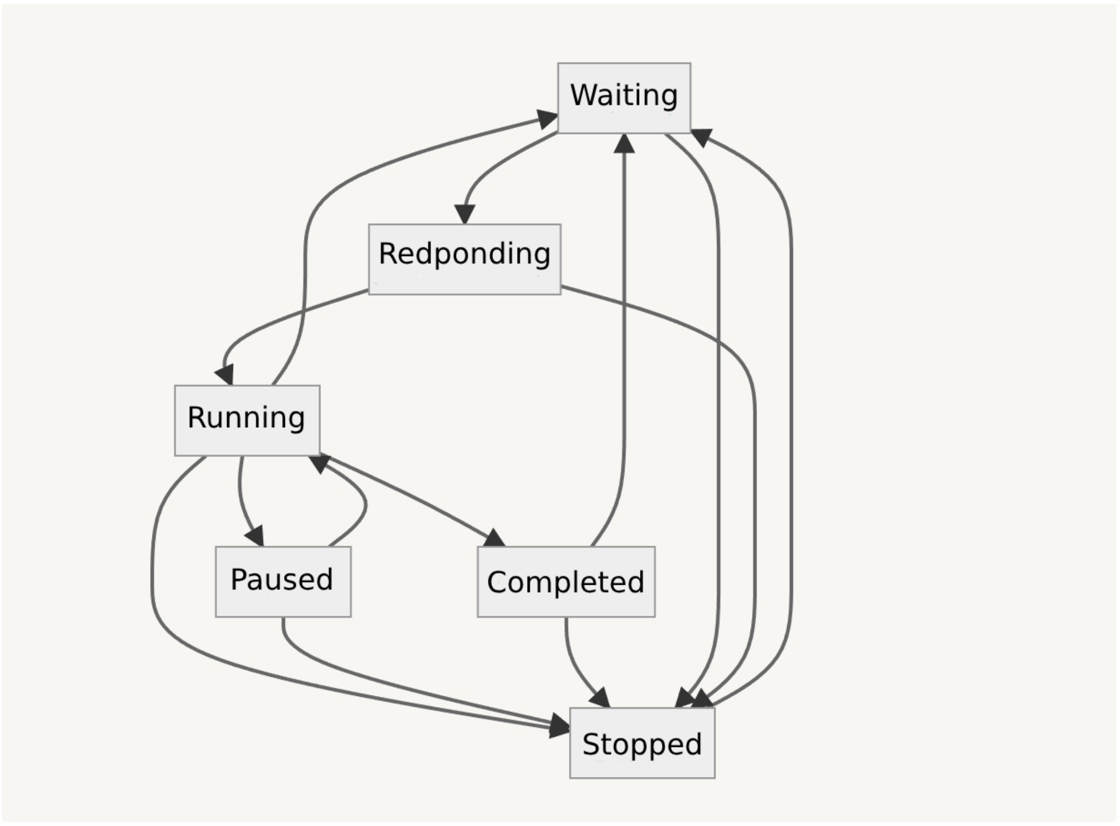


Figure 2.1: Finite State Machine [34]

When Bungie Studios began developing "Halo 2" [35], a 2004 first-person shooter game, they realized the FSM system used in the original "Halo" game would be insufficient. They wanted to provide more dynamic combat and smarter AI NPCs. Damian Isla, a notable AI developer at Bungie, supported the use of Behavior Trees for "Halo 2". A simplified BT of "Halo 2" is present in Fig. 2.2. BTs offered a hierarchical structure, allowing developers to define behaviors at multiple levels. This meant that high-priority behaviors (like taking over when in danger) would interrupt lower-priority behaviors (like patrolling an area), making NPCs more reactive and intelligent. The success of BTs in "Halo 2" led to their continued use in subsequent "Halo" games and served as a model for other game developers [36].

Researchers in robotics were facing challenges similar to game developers. Robots need to operate in dynamic environments, interact with unpredictable elements and make decisions in real time. The foundational idea for behavior-based control in robotics was proposed by Rodney Brooks [37]. He introduced a list of behaviors that could function alternatively, laying the foundation for behavior-based control in robotics. This initial concept was later expanded into a more structured, tree-like organization of behaviors. This structure allowed for more complex and dynamic control systems for robots. Recent

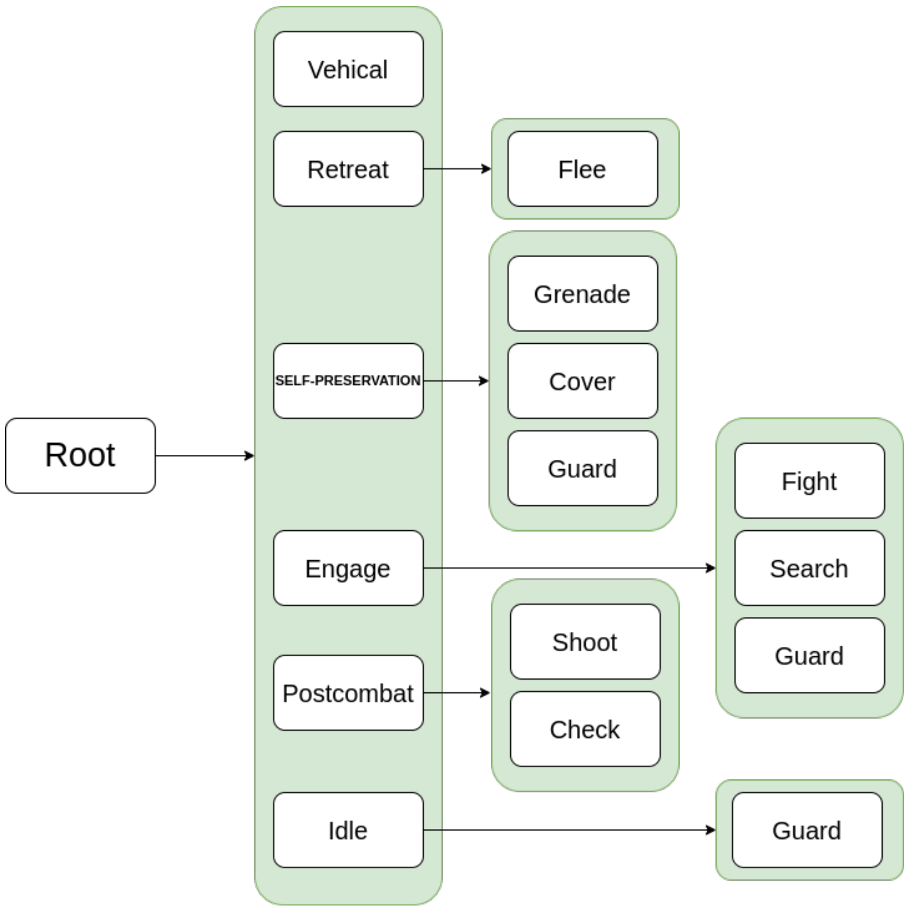


Figure 2.2: Simplified Halo 2 behavior tree

works propose BTs as a multi-mission control framework for various types of robots, including UAVs (Unmanned Aerial Vehicles) [38], robotic manipulation systems, and multi-robot systems [39].

ROS used the early integration of BTs as plugins or packages [40], which reduced the difficulty of the design and execution of complex robotic behaviors. ROS2 was designed with modularity, real-time capabilities, and enhanced security. These features are suited perfectly to the principles of BTs. ROS2 used deeper integration for BTs with tools and libraries, making it easier to create and design complex and layered robotic behaviors.

2.1.2 Nodes and Trees

Nodes are the core function of behavior trees, which represent tasks or decisions. The process by which a BT executes its nodes is controlled by a concept known as a "tick" [41]. A tick is actually a signal that activates a node, informing it to perform its function. During a BT's execution cycle, ticks are propagated from the root of the tree down to its branches, visiting nodes by sequence. When a node receives a tick, it executes its callback and returns a status: `SUCCESS`, `FAILURE`, or `RUNNING`. `RUNNING` means that the action needs more time to return a valid result. This status determines how the tree continues its execution. There are several different types of nodes. Root node is where the tick starts. Every behavior tree has one root, and it determines which child nodes get ticked based on the logic and their status. Detailed information about other nodes is presented below.

Control Node

- Sequence Node. The sequence node ticks its children sequentially, from left to right. It fails as soon as one child returns `FAILURE` and only succeeds if all children return `SUCCESS`. Sequence node includes "Sequence", "ReactiveSequence", and "SequenceStar" [42]. The difference between them can be found in Table 2.1. "Restart" means that the entire sequence is restarted from the first child of the list, and "Tick again" means that the next time the sequence is ticked, the same child is ticked again. Previous children, which returned `SUCCESS` already, are not ticked again.

Type of SequenceNode	Child returns FAILURE	Child returns RUNNING
Sequence	Restart	Tick again
ReactiveSequence	Restart	Restart
SequenceStar	Tick again	Tick again

Table 2.1: Type of SequenceNode

The tree in Fig. 2.3 represents the behavior of a robot entering a room. In this example, the first tick from the root node sets the Sequence node to `RUNNING`. Then the sequence node ticks the first child "OpenDoor", which returns `SUCCESS`. As a result, "EnterRoom" and later "CloseDoor" are ticked. Once the last child is completed, the sequence node changes from `RUNNING` to `SUCCESS`. If any child returns `FAILURE`, the subsequent children will not be ticked and the sequence node will switch from `RUNNING` to `FAILURE`.

- Fallback Node. Fallback node ticks its children from left to right, but succeeds as soon as one child returns `SUCCESS`. It only fails if all children return `FAILURE`. Fallback node includes "Fallback" and "ReactiveFallback" [43]. The difference between them can be found in Table 2.2. "Restart" means that the entire fallback is restarted from the first child of the list, and "Tick again" means that the next time the fallback is ticked, the same child is ticked again. Previous children, which returned `FAILURE` already, are not ticked again.

Type of FallbackNode	Child returns RUNNING
Fallback	Tick again
ReactiveFallback	Restart

Table 2.2: Type of FallbackNode

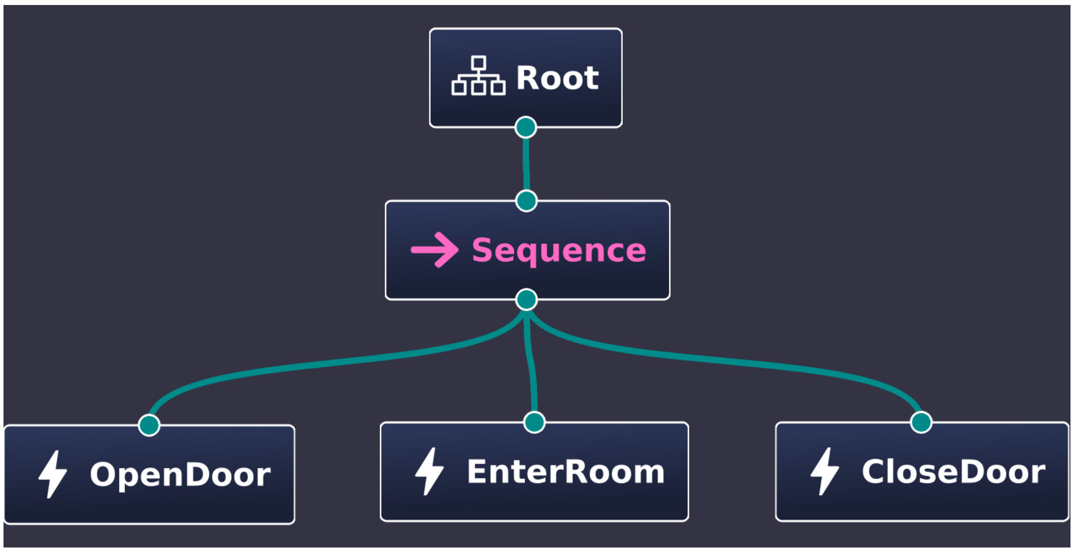


Figure 2.3: Example of a sequence node.

The tree in Fig. 2.4 represents different strategies to enter a room. In this example, the fallback node gets ticked by the sequence node and changes its status to `RUNNING`. If the first child returns `SUCCESS`, the subsequent children will not be ticked and the fallback node will return `SUCCESS`. On the contrary, if the first child returns `FAILURE`, "OpenDoor" and "SmashDoor" will be ticked. If all the children return `FAILURE`, the fallback node will return `FAILURE` and "EnterRoom" will not be ticked.

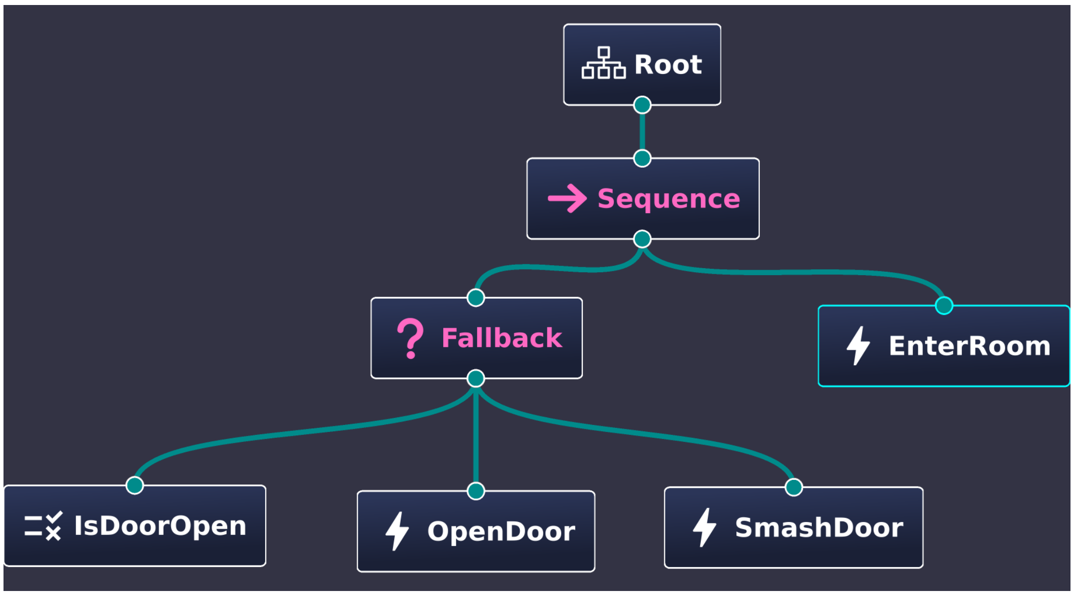


Figure 2.4: Example of a fallback node.

Decorator Node

It modifies the behavior or the status of their single child. A decorator can only have one child. Examples include "InverterNode", "RepeatNode", and "RetryNode" [44]. "InvertNode" will return `SUCCESS` if the child fails and return `FAILURE` if the child succeeds. "RepeatNode" ticks the child up to a preset time as long as the child returns `SUCCESS`, while "RetryNode" ticks as long as the child returns `FAILURE`.

The tree in Fig. 2.5 presents an example of decorator nodes. An inverter with a node "IsDoorOpen" is the same with "IsDoorClosed". The times of repeating or retrying are got from the port. In this case, the number is 3. The logic of this behavior tree is: Detecting whether the door is closed, and if so, try to open the door three times. Returning `FAILURE` if all attempts failed.

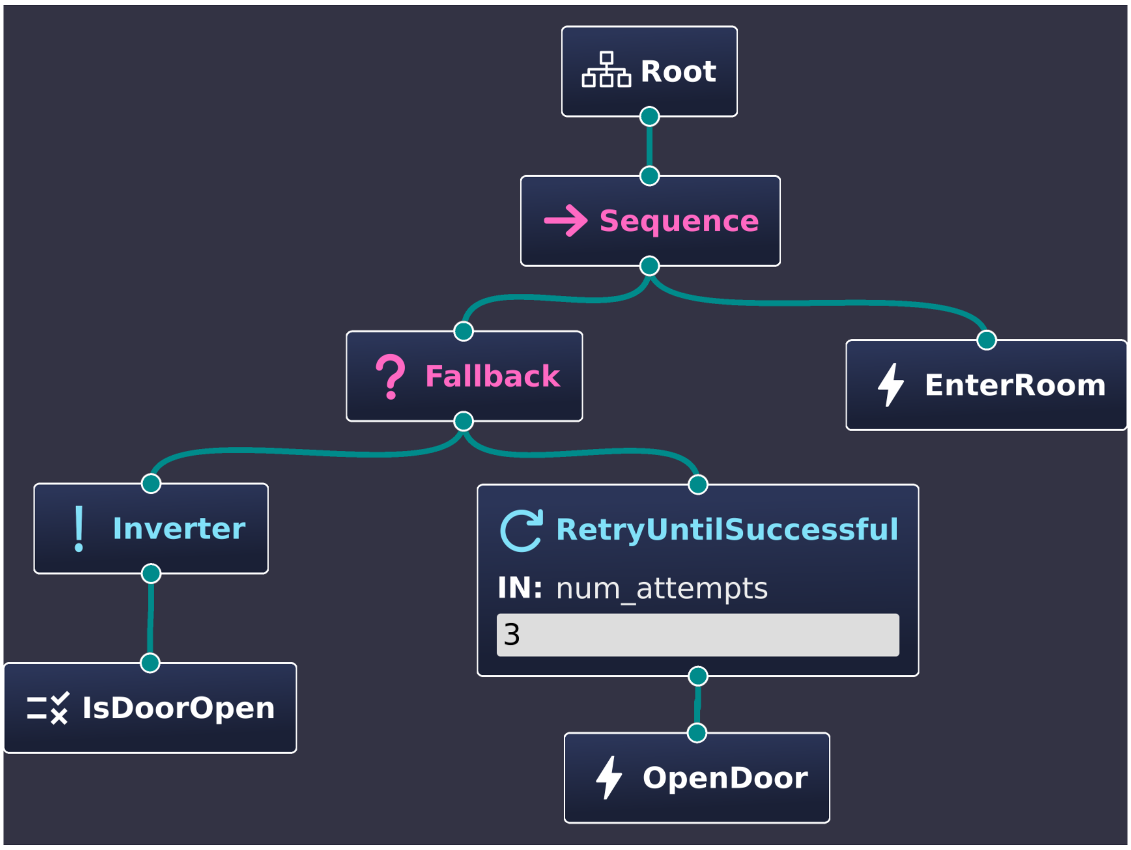


Figure 2.5: Example of decorator nodes.

Leaf Node

- Action Node. It represents an actual task, like "move forward" or "open the door".
- Condition Node. It checks a specific condition, like "is battery low?" or "Is the door open?".

Behavior Trees

A behavior tree is composed of nodes with a hierarchical structure. The root node is at the top and the leaf nodes are at the bottom. Control flow nodes and decorator nodes act as intermediaries, deciding the flow and behavior of ticks [45]. In each execution cycle, a tick is propagated from the root node, traveling through the tree based on node logic and statuses. The behavior tree is usually a `XML` file. Listing 2.1 is an example.

```xml
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <Sequence>
      <OpenDoor/>
      <EnterRoom/>
      <CloseDoor/>
    </Sequence>
  </BehaviorTree>
</root>
```

Listing 2.1: A simple behavior tree

Behavior Trees provide ports and a blackboard to ensure that data is appropriately shared and accessed by nodes during the tree's execution. They facilitate communication and data exchange within the tree, allowing for more dynamic and intelligent behaviors. The blackboard can be thought of as a data storage center or a shared memory space for the Behavior Tree. Ports can be used as the "input" and "output" interfaces of a node. They define what data a node needs to read from the blackboard (input) and what data it might write to the blackboard (output). An example of the usage of the blackboard and ports is represents in Listing 2.2. The frequency of the "RateController" is got from the port "hz", which is 1.0. The "ComputePathToPose" reads the "goal" and "path" from blackboard.

```xml
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <PipelineSequence name="NavigateWithReplanning">
      <RateController hz="1.0">
        <ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased"/>
      </RateController>
      <FollowPath path="{path}"  controller_id="FollowPath"/>
    </PipelineSequence>
  </BehaviorTree>
</root>
```

Listing 2.2: A behavior tree illustrating the blackboard utilization and ports integration

"BehaviorTree.CPP" is one of the most popular open-source libraries designed to implement Behavior Trees in C++ [46]. It provides a clean API to define custom nodes (both actions and conditions). With "BehaviorTree.CPP", ROS2 nodes (actions or services) can be integrated as action or condition nodes within a Behavior Tree. This allows developers to use existing ROS2 functionalities directly within the trees.

Using an additional tool named "Groot" [47], developers can visually design and create behavior Trees. It is part of the BehaviorTree.CPP suite. Groot2 includes a Behavior Tree editor that allows users to create and edit trees using a simple drag-and-drop interface. Fig. 2.6 is the behavior tree structure displayed in "Groot" corresponding to the forward `XML` file.

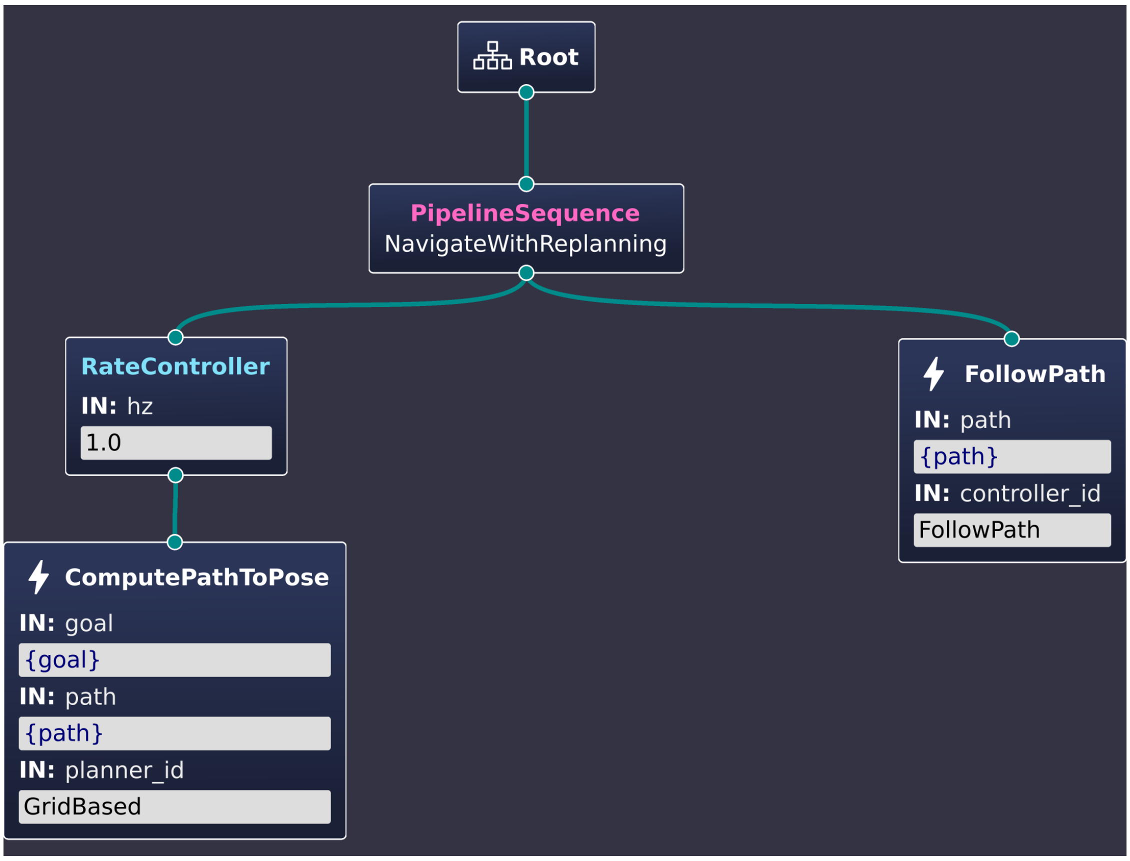


Figure 2.6: Behavior tree in Groot

2.2 Behavior trees in ROS2

The behavior trees in ROS2 allow developers to describe robotic tasks in a hierarchical and modular way. With Behavior trees, robots can make decisions based on real-time data [48]. For instance, a robot might choose a different path if the original path is blocked by obstacles, without needing to recompute the entire plan. Another advantage is the integration with ROS2 actions and services. With behavior trees, these actions can be utilized as nodes, allowing the tree to set goals and supervise the progress. Service also can be called by the nodes of a tree. For instance, clear the entire costmap or get new costmap.

2.2.1 Behavior-tree library

The Navigation2 stack provides a variety of behavior trees to enable a robot to navigate from an initial position to a goal pose. The tree files can be found in folder `nav2_bt_navigator/behavior_trees`. In this subsection the focus is primarily on `navigate_to_pose_w_replanning_and_recovery.xml` and `navigate_to_pose_w_replanning_goal_patience_and_recovery.xml`. The former is the default behavior tree of nav2, and the latter provides inspiration for the implementation of multi-floor navigation.

navigate_to_pose_w_replanning_and_recovery

It provides a detailed, step-by-step plan for a robot to reach the goal pose while continuously replanning its path. This structure enables the robot to deal with unforeseen obstacles and environment changes. Additionally, when the robot meets situations it cannot handle by replanning, recovery behaviors will be used to solve the problems. The behavior tree `navigate_to_pose_w_replanning_and_recovery` is displayed in Fig. 2.7.

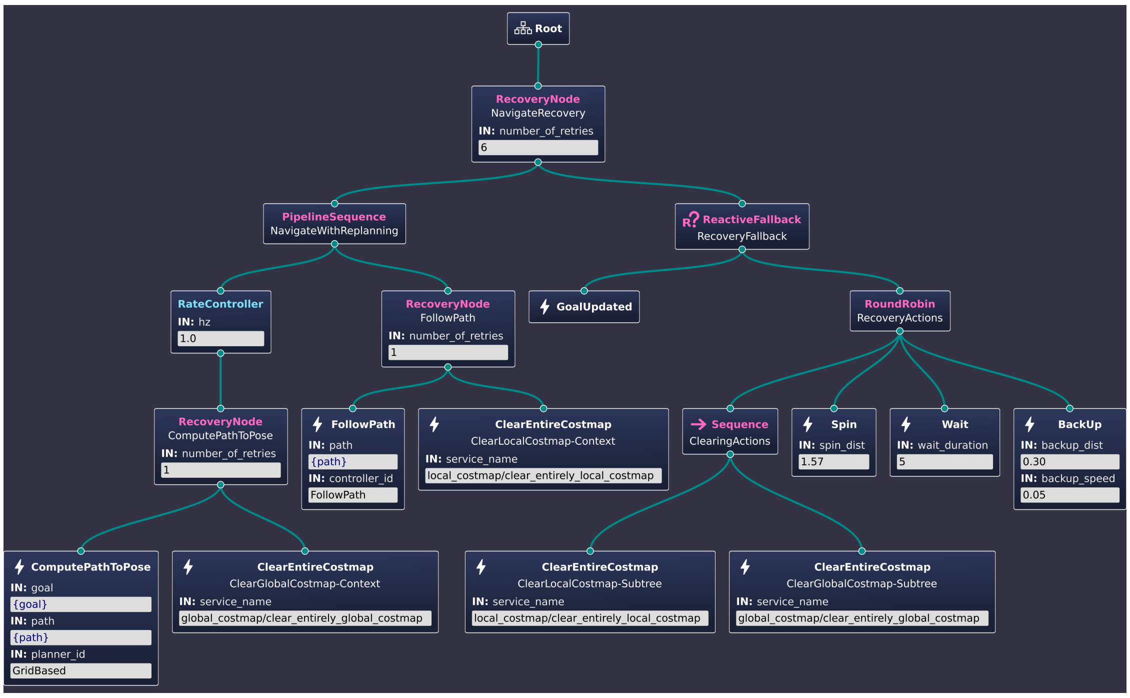


Figure 2.7: Navigate_to_pose_w_replanning_and_recovery.xml

The entire behavior tree can be divided into two parts: Navigation and Recovery. The navigation subtree is responsible for the core function of moving the robot from its current position to the goal pose. The structure is illustrated in Fig. 2.8. "PipelineSequence" is a control plugin based on the sequence node. It ticks the first child until it succeeds, then ticks the first and second children until the second child succeeds. The "RecoveryNode" has two children. It returns `SUCCESS` if and only if the first child returns `SUCCESS`. The second child will be ticked only if the first child returns `FAILURE` and if the second child succeeds, the first child will be executed again. In addition, it includes the following components:

- Path Planning. It generates a global path to the goal using algorithms like Dijkstra or A*. The path is written into the blackboard.

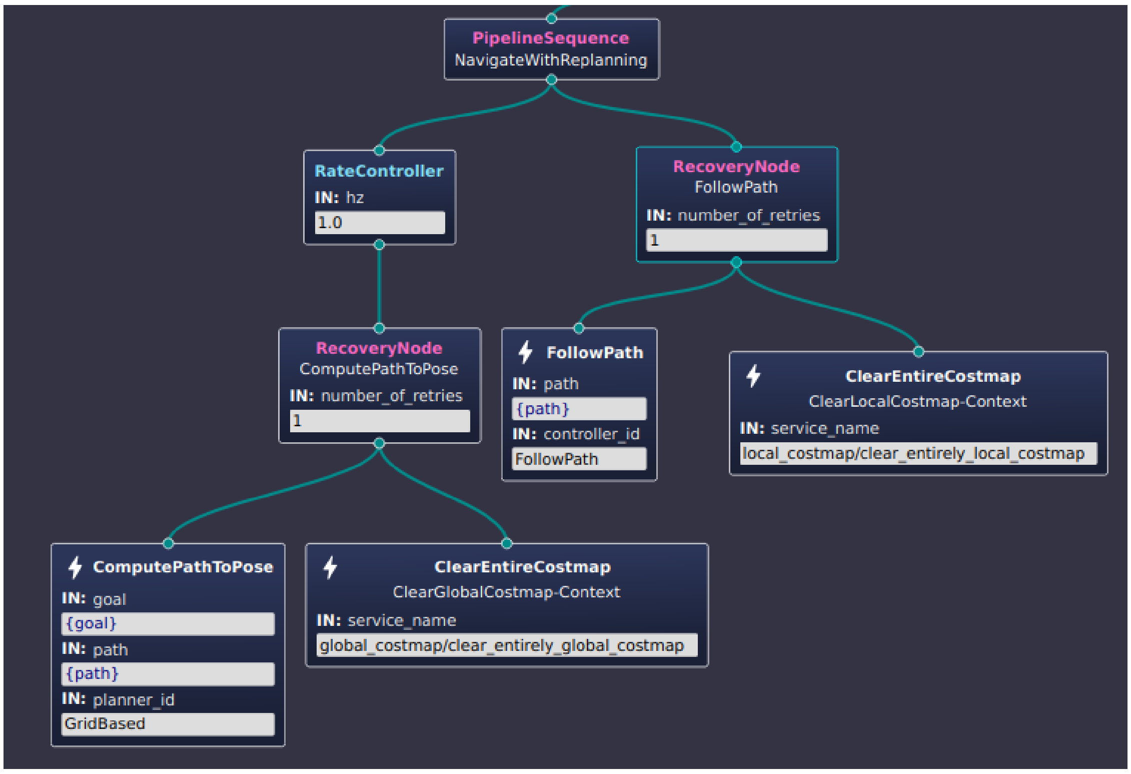


Figure 2.8: Navigation subtree of "Navigate To Pose"

- Local Planner. It takes the global path and computes velocity parameters to follow it. Makes dynamic adjustments to the path to adapt moving obstacles or changes in the environment.

- Goal Checker. It monitors the robot's position to determine whether the goal has been reached by calculating the distance between the robot and the goal.

- Replanning. It asks the path planning component to generate a new path to the goal in a specific frequency.

The recovery subtree is triggered when there is a failure in the navigation process. The structure is featured in Fig. 2.9. The "RoundRobin" node ticks its children in a round-robin [49] fashion until a child returns `SUCCESS`. It will give each child a chance to run before restarting the cycle. The primary functions of the recovery subtree include:

- Goal Updated. It is activated when a new goal has been passed to the navigation system through a preemption.

- Failure Detection. It determines when the navigation process cannot proceed as planned. For example, detecting that the robot has not moved for a certain amount of time.

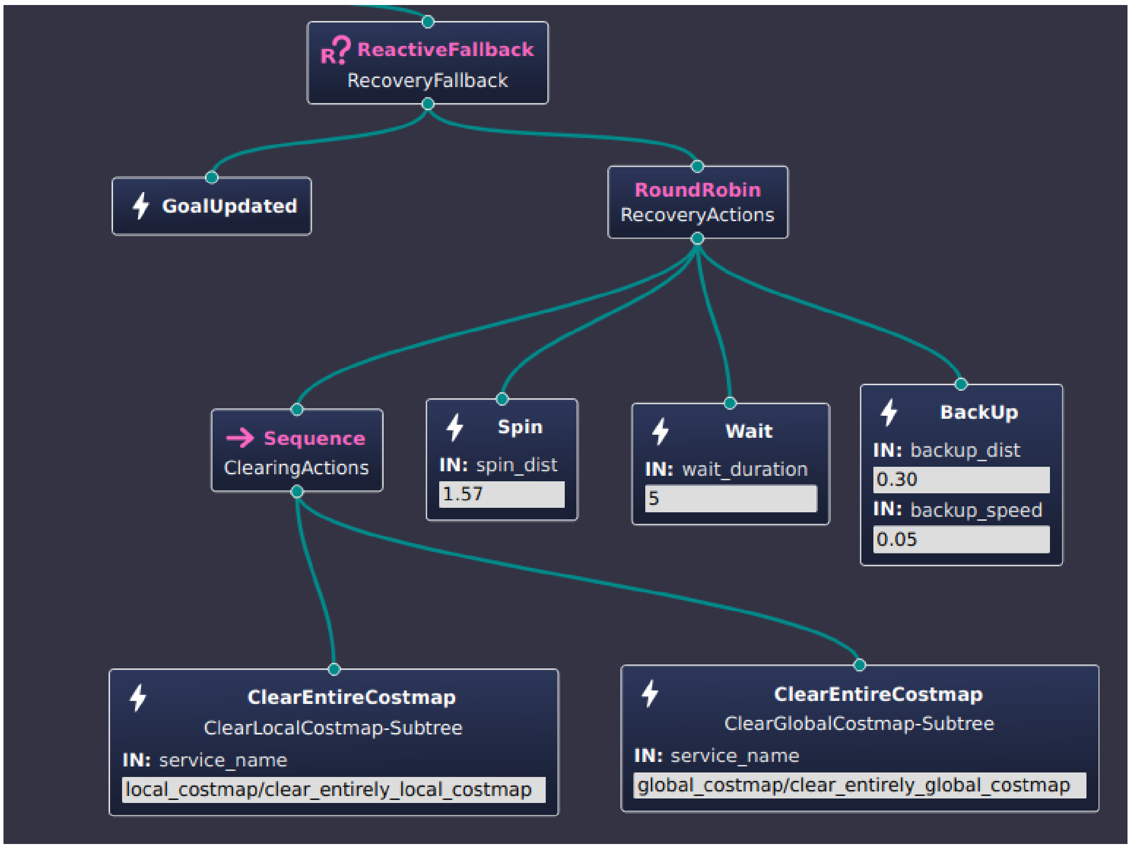


Figure 2.9: Recovery subtree of "Navigate To Pose"

- Recovery Action. It aims at moving the robot from a stuck position. For example, spinning in place or backing up a little.
- Clearing Behavior. It initializes the global costmap and local costmap after clearing them.

There is another default behavior tree provided by the Navigation2 stack: `navigate_through_poses_w_replanning_and_recovery.xml`, which is used in navigation starting from an initial pose, through some intermediary poses, to the final pose. It accepts and stores a list of waypoints into the blackboard and plans the path from the previous point to the next point. This behavior tree was initially analyzed as a potential solution, considering that multi-floor navigation could be viewed as a composition of several navigation processes. However, a more effective solution was eventually identified.

navigate_to_pose_w_replanning_goal_patience_and_recovery.

This behavior tree is an extension to the "Navigate To Pose", which not only directs a robot towards a goal but also allows the robot to adapt its path in response to dynamic changes. The structure is presented in Fig. 2.10.

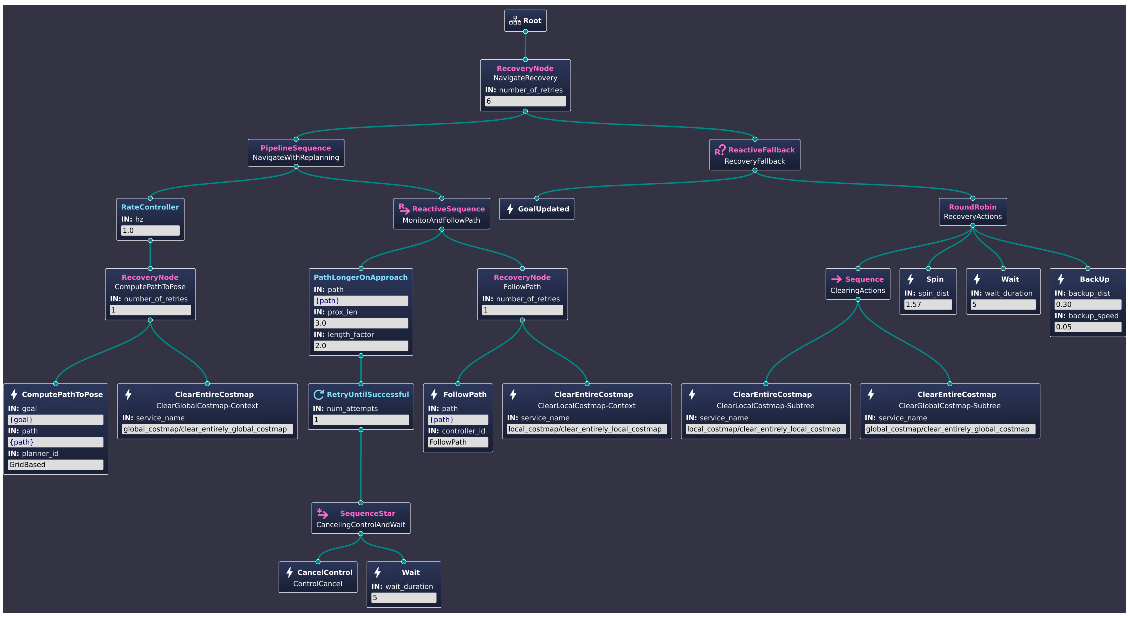


Figure 2.10: Navigate_to_pose_w_replanning_goal_patience_and_recovery.xml

The robot will pause the navigation and wait for a specific time when an obstacle (e.g., person, cargo) suddenly appears close to the goal. If the obstacle has moved during the wait time, the robot will follow the original path to the target. Otherwise, the robot will plan a new path which is longer than the old one.

This behavior tree is similarly divided into two sections: Navigation and Recovery. The structure of the navigation subtree is displayed in Fig. 2.11, and the recovery part is the same as "Navigate To Pose". It can be noticed that there is a new branch called "MonitorAndFollowPath", which is used by the node "PathLongerOnApproach". This particular node continuously checks if the global planner has planned a significantly longer path than the current one for the robot to approach the goal. If there is no longer path, "FollowPath" will be ticked, resulting in the robot's behavior being identical when employing the "Navigate To Pose" strategy.

Once there is a significantly longer path, the child of "PathLongerOnApproach" is ticked. This decorator node ticks the sequence node to stop the robot and wait. When the "CancelControl" node is ticked, the controller server stops the further navigation of the robot. The wait time is defined through a port. Since the "MonitorAndFollowPath" is a ReactiveSequence node, the "PathLongerOnApproach" needs to return `SUCCESS` so that the "FollowPath" node can be ticked again.

2.2.2 Behavior-tree Navigator

The behavior tree navigator primarily serves as the interface between the movement request and the corresponding logic, which is detailed in the behavior tree's XML. This involves implementing a specific action server for actions like "Navigate to Pose" or "Navigate Through Pose". It translates the abstract action "navigate to this goal" into specific

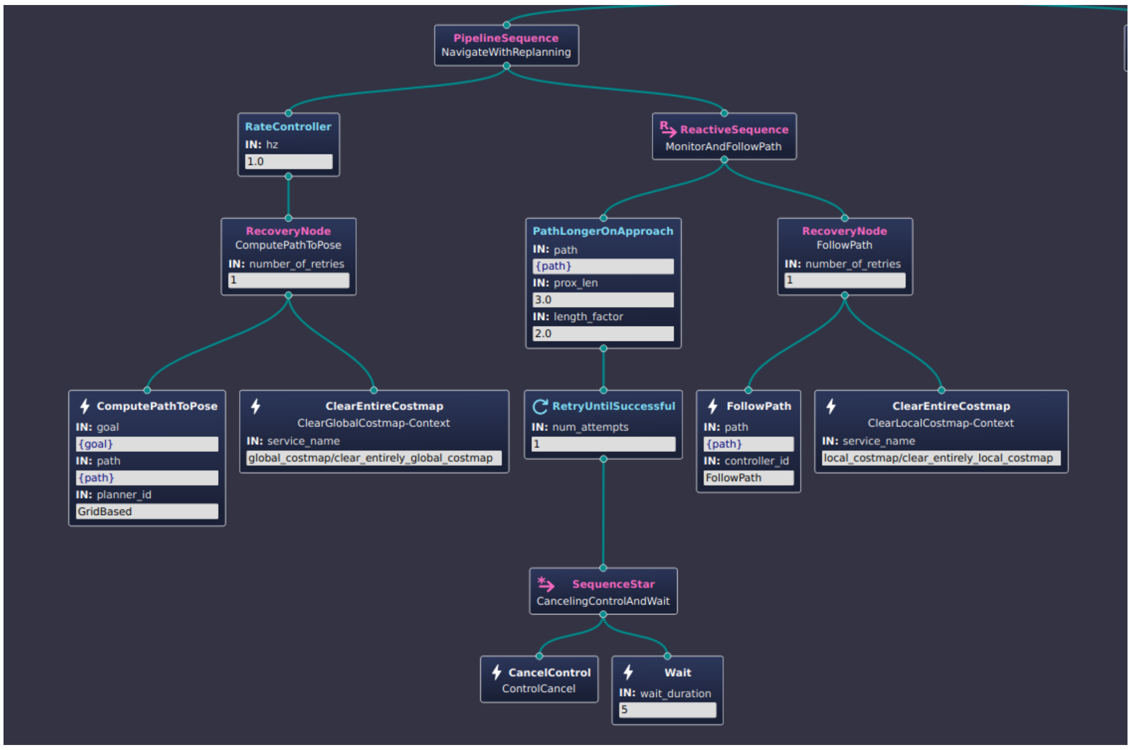


Figure 2.11: Navigation subtree of "Navigate To Pose and Pause Near Goal-Obstacle"

steps that the robot must perform, manages the lifecycle of this process, and handles communication between the navigation stack and other components that are requesting navigation actions. The typical functions and their descriptions are presented below:

- configure(): It allocates and initializes resources that the node will use, including initializing parameters, setting up the blackboard, creating clients for communication, and more. For example, create a client to subscribe to the goal topic (Listing 2.3).

```
// navigate_to_pose.cpp
bool
NavigateToPoseNavigator::configure(
  rclcpp_lifecycle::LifecycleNode::WeakPtr parent_node,
  std::shared_ptr<nav2_util::OdomSmoother> odom_smoother)
{
[...]
    goal_sub_ = node->create_subscription<geometry_msgs::msg::PoseStamped>(
    "goal_pose",
    rclcpp::SystemDefaultsQoS(),
    std::bind(&NavigateToPoseNavigator::onGoalPoseReceived, this, std::
    placeholders::_1));
  return true;
}
```

Listing 2.3: Configuration function within navigate_to_pose.cpp file

The Navigation2 stack involves a variety of messages for communication, among which `goal_pose` uses the message type `geometry_msgs/PoseStamped`. This message type is designed to represent a pose in 3D space. It consists of two main parts:

a header and a pose. The header specifies the frame_id of the goal, while the pose itself is divided into position and orientation. The position is defined by the "Point" message, which represents the 3D coordinates (x, y, z). The orientation is described by the "Quaternion" message, determining the orientation in space.

- getDefaultBTFilepath(): It provides a default Behavior Tree `XML` file path that the navigator can use in case a specific path hasn't been provided by configuration files (Listing 2.4).

```cpp
// navigate_to_pose.cpp
std::string
NavigateToPoseNavigator::getDefaultBTFilepath(
  rclcpp_lifecycle::LifecycleNode::WeakPtr parent_node)
{
  std::string default_bt_xml_filename;
  auto node = parent_node.lock();
  if (!node->has_parameter("default_nav_to_pose_bt_xml")) {
    std::string pkg_share_dir =
      ament_index_cpp::get_package_share_directory("nav2_bt_navigator");
    node->declare_parameter<std::string>(
      "default_nav_to_pose_bt_xml",
      pkg_share_dir +
      "/behavior_trees/navigate_to_pose_w_replanning_and_recovery.xml");
  }
  node->get_parameter("default_nav_to_pose_bt_xml", default_bt_xml_filename);
  return default_bt_xml_filename;
}
```

Listing 2.4: Function to set the behavior tree file path

- goalReceived(): It gets triggered when a new navigation goal is received by the action server, and checks if the received goal is valid (Listing 2.5).

```cpp
// navigate_to_pose.cpp
bool
NavigateToPoseNavigator::goalReceived(ActionT::Goal::ConstSharedPtr goal)
{
  auto bt_xml_filename = goal->behavior_tree;
  if (!bt_action_server_->loadBehaviorTree(bt_xml_filename)) {
    RCLCPP_ERROR(
      logger_, "BT file not found: %s. Navigation canceled.",
      bt_xml_filename.c_str());
    return false;
  }
  initializeGoalPose(goal);
  return true;
}
```

Listing 2.5: Initilize the goal after receiving

- onLoop(): It is commonly used within a loop to repeatedly execute tasks that must be checked or updated continuously during the node's operation.

- onPreempt(): This function is triggered when a new goal has been received while another goal is currently being processed. It stops the current behavior tree execution, cleans up resources or processes that were dedicated to the previous goal, and initiates the navigation process for the new goal (Listing 2.6).

```cpp
// navigate_to_pose.cpp
void
NavigateToPoseNavigator::onPreempt(ActionT::Goal::ConstSharedPtr goal)
{
  RCLCPP_INFO(logger_, "Received goal preemption request");
  if (goal->behavior_tree == bt_action_server_->getCurrentBTFilename() ||
    (goal->behavior_tree.empty() &&
    bt_action_server_->getCurrentBTFilename() == bt_action_server_->
    getDefaultBTFilename()))
  {
    initializeGoalPose(bt_action_server_->acceptPendingGoal());
  } else {
    bt_action_server_->terminatePendingGoal();
  }
}
```

Listing 2.6: Function for preempting goals

- goalCompleted(): It is called when the navigation has successfully reached its goal or if the goal is completed due to a cancel request.

Chapter 3

Multi-floor navigation

This chapter focuses primarily on the execution of a multi-floor simulation process. ROS2 humble on Ubuntu 22.04 has been selected as the operating environment [50]. Gazebo and Rviz are utilized to build the simulation environment. The initial part discusses the default navigation process, examining its operation and its limitations in the context of multi-floor navigation. Subsequently, a behavior tree-based solution is introduced. Custom behavior tree nodes have been created and integrated as plugins, which serve as the foundation for a new behavior tree architecture. Additionally, a novel navigator based on "NavigateToPose" has been implemented.

Fig. 3.1 presents the architecture of the LINKS building, serving as the simulation environment for the navigation exercise. This building includes two floors interconnected via an elevator system. The robot navigates from an initial position located at the first floor, employing the TurtleBot3 Waffle as the robotic model.

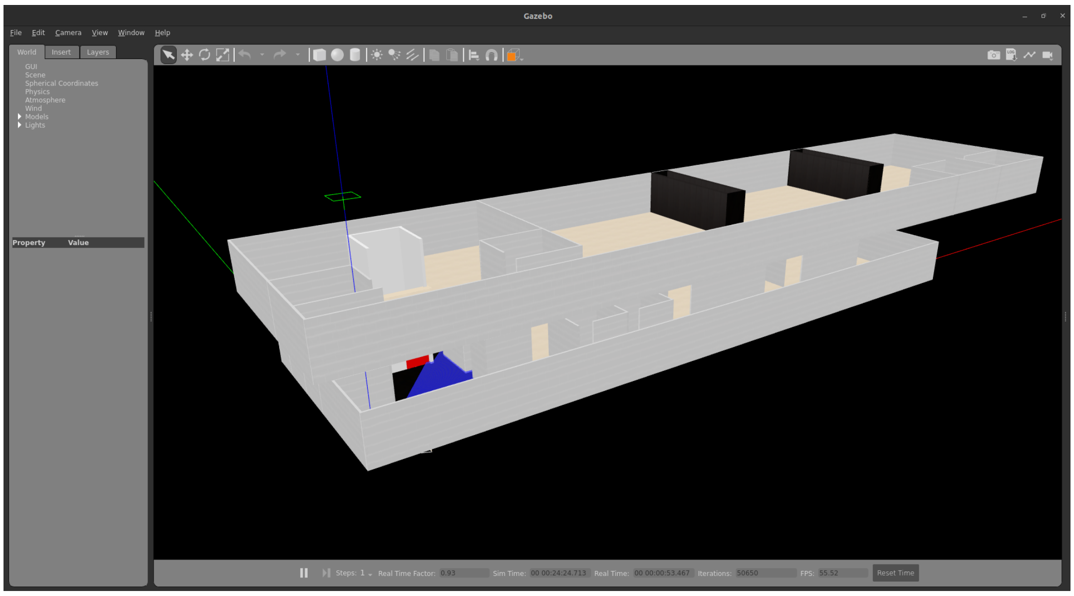
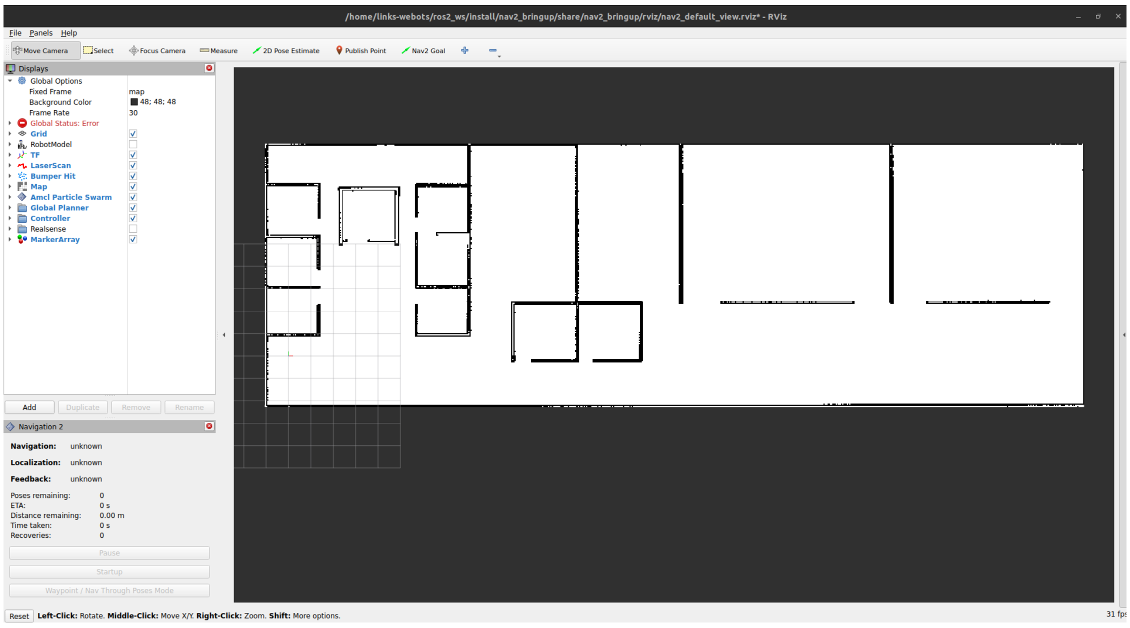


Figure 3.1: LINKS office in Gazebo

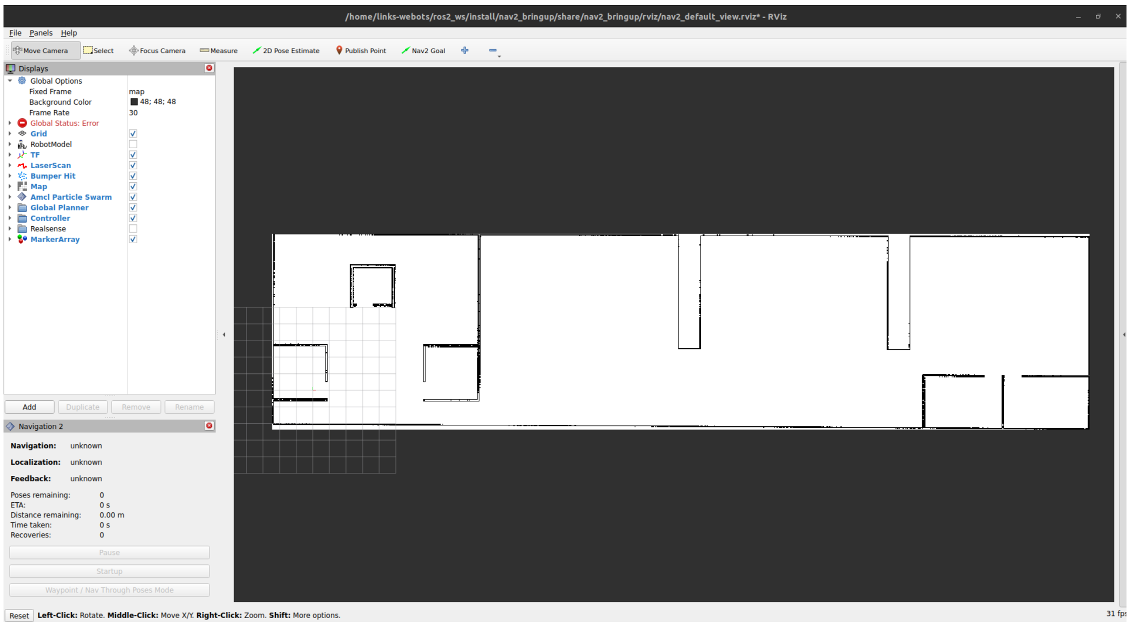
The maps of both floors are exhibited in Fig. 3.2. By default, the map that Rviz initializes reflects the floor 0, correlating with the robot's initial starting point.


/home/links-webots/ros2_ws/install/nav2_bringup/share/nav2_bringup/rviz/nav2_default_view.rviz* - RViz

```
Global Options
  Fixed Frame        map
  Background Color   48; 48; 48
  Frame Rate         30
Global Status: Error

Navigation:    unknown
Localization:  unknown
Feedback:      unknown

Poses remaining:     0
ETA:                 0 s
Distance remaining:  0.00 m
Time taken:          0 s
Recoveries:          0
```

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click: Zoom. Shift: More options.

(a) Floor 0 in Rviz


/home/links-webots/ros2_ws/install/nav2_bringup/share/nav2_bringup/rviz/nav2_default_view.rviz* - RViz

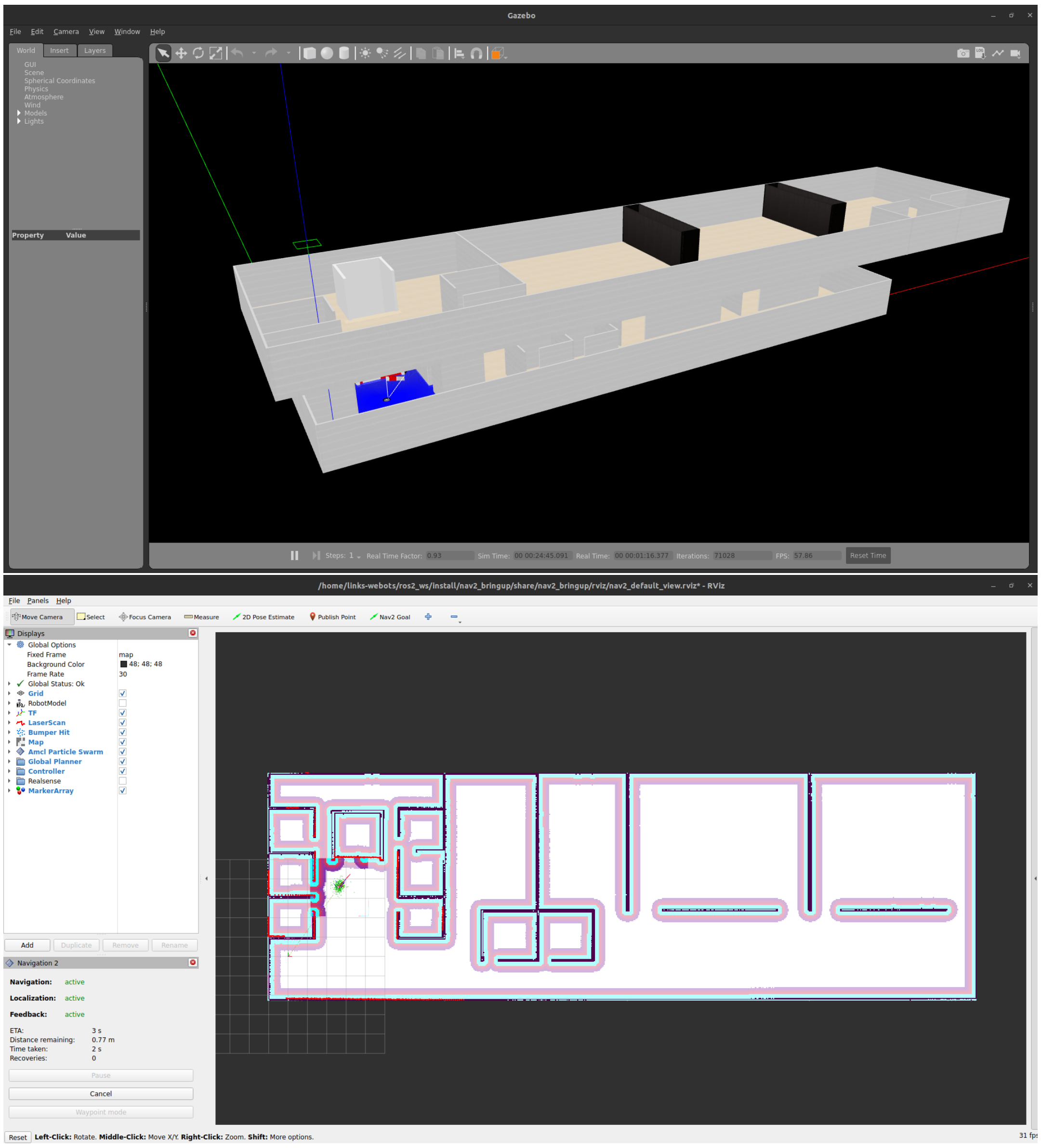
```
Global Options
  Fixed Frame        map
  Background Color   48; 48; 48
  Frame Rate         30
Global Status: Error

Navigation:    unknown
Localization:  unknown
Feedback:      unknown

Poses remaining:     0
ETA:                 0 s
Distance remaining:  0.00 m
Time taken:          0 s
Recoveries:          0
```

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click: Zoom. Shift: More options.

(b) Floor 1 in Rviz

Figure 3.2: Simulation environment of LINKS in Rviz

The initial section of the navigation procedure, after the launch of both Gazebo and Rviz, involves establishing the robot's initial pose. This step can be accomplished by utilizing the `2D Pose Estimate` function in Rviz, or by issuing a command to publish on the `/initialpose` topic using `ros2 topic pub`. After the initial pose setting, the bt navigator along with the costmap server is activated automatically. Following this, the navigation goal can be assigned either through the "`Nav2 Goal`" tool in Rviz or by publishing to the `/goal_pose`. The distinction is that: employing `Nav2 Goal` conveys the goal directly to the "NavigateToPose" action server, thereby using the default behavior tree in the navigation process. On the other hand, publishing over the topic sends the goal to the bt navigator defined within the configuration file, allowing for customized behavior tree usage.



Steps: 1 Real Time Factor: 0.93 Sim Time: 00 00:24:45.091 Real Time: 00 00:01:16.377 Iterations: 71028 FPS: 57.86

Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
Grid	✓
RobotModel	☐
TF	✓
LaserScan	✓
Bumper Hit	✓
Map	✓
Amcl Particle Swarm	✓
Global Planner	✓
Controller	✓
Realsense	☐
MarkerArray	✓

Navigation: active

Localization: active

Feedback: active

ETA: 3 s
Distance remaining: 0.77 m
Time taken: 2 s
Recoveries: 0

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:** Zoom. **Shift:** More options.

Figure 3.3: A navigation process in Gazebo and Rviz using the default behavior tree

The challenges presented by the default navigation process are evident from the previous scenario. Key issues include:

- Sending a navigation goal located on a different floor from where the robot currently is.
- Planning a path to this goal and interfacing with the elevator system.
- Switching the map and navigating to the goal once the robot reaches the target floor.

Here is the solution proposed by the thesis for multi-floor navigation:

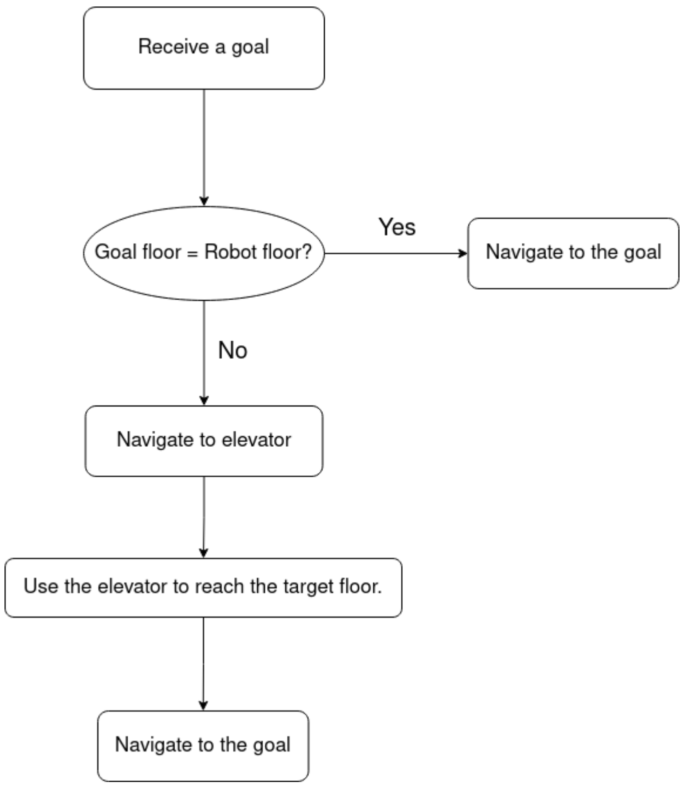


Figure 3.4: A simplified version of the multi-floor navigation method

3.1 Behavior tree plugin

Behavior tree plugins are a powerful mechanism for extending the functionality of the Navigation2 stack, enabling the customization of robot behavior to suit a wide range of applications. These plugins can be classified based on their function and role within the behavior tree: action plugin, condition plugin, control plugin, and decorator plugin.

3.1.1 Action node plugin

Action plugin executes operations such as moving the robot to a goal, following a path. These plugins are responsible for the active tasks in the behavior tree. Action nodes can be divided into two categories: those related to a ROS2 topic and those related to a ROS2 service. An action node associated with a ROS2 topic publishes a message each time it is ticked, while an action node using a ROS2 service calls the service upon being ticked. The action nodes utilized in this project include:

- Load new map. The "Load New Map" service node is designed to switch the active map to a new one, which is necessary when the robot transits to a different floor. Once ticking, this node triggers Rviz to load the specified new map. The file path for the new map must be supplied. Listing 3.1 is a detailed description of its structure:

```
// load_map_service.cpp
LoadMapService::LoadMapService(
        const std::string & service_node_name,
        const BT::NodeConfiguration & conf)
        : BtServiceNode<nav2_msgs::srv::LoadMap>(service_node_name, conf)
        {
        }

void LoadMapService::on_tick()
        {
            server_timeout_ *= 3;
            mappath = config().blackboard->get<std::string>("map_path");
            request_->map_url = mappath;
            increment_recovery_count();
        }

#include "behaviortree_cpp_v3/bt_factory.h"
BT_REGISTER_NODES(factory)
{
  factory.registerNodeType<nav2_behavior_tree::LoadMapService>("LoadMap");
}

// load_map_service.hpp
#include "nav2_behavior_tree/bt_service_node.hpp"
#include "nav2_msgs/srv/load_map.hpp"
```

Listing 3.1: BT service node: Load a new map

  The `nav2_behavior_tree` package includes the `bt_service_node.hpp` library, which standardizes the structure of service nodes. This library creates clients for designated services and sends requests when ticked. Additionally, it assesses the response from the service call and converts it into corresponding behavior tree statuses. In the end the `Behaviortree_CPP` library is used for node registering, which is necessary for all behavior tree plugins.

- Send a goal. The "Send a goal" action node is responsible for transmitting a new navigation goal. This action node has the function of sending a new navigation goal. This node possesses preemptive capabilities and will interrupt any ongoing navigation processes. When ticked, the robot initiates navigation towards the specified goal. This node is particularly useful after the robot has utilized the elevator and needs to proceed to its final destination. Meanwhile, the floor of the robot gets updated since it has moved to the target floor. The implementation details are displayed in Listing 3.2.

```
// send_goal_node.cpp
SetGoal::SetGoal(
    const std::string & name,
    const BT::NodeConfiguration & conf)
: BT::SyncActionNode(name, conf)
{
    node_ = config().blackboard->get<rclcpp::Node::SharedPtr>("node");
    publisher_ = node_->create_publisher<geometry_msgs::msg::PoseStamped>("
    goal_pose", 10);
}

BT::NodeStatus SetGoal::tick()
    {
        RCLCPP_INFO(node_->get_logger(), "Set the goal");
        goal_final = config().blackboard->get<geometry_msgs::msg::PoseStamped
    >("goalwithfloor");
        floorofrobot.pose.position.z = goal_final.pose.position.z;
        config().blackboard->set<geometry_msgs::msg::PoseStamped>("
    floor_robot", floorofrobot);
        goal_final.pose.position.z = 0;
        publisher_->publish(goal_final);
        return BT::NodeStatus::SUCCESS;
    }

#include "behaviortree_cpp_v3/bt_factory.h"
BT_REGISTER_NODES(factory)
{
  factory.registerNodeType<nav2_behavior_tree::SetGoal>("SetGoal");
}

// set_goal_node.hpp
#include "behaviortree_cpp_v3/action_node.h"
```

Listing 3.2: BT action node: Send a goal

In the constructor, the publisher for the corresponding topic is initialized. When the node is ticked, it publishes a message with the coordinates of the goal provided by the Blackboard. Unlike the service node, the status of the action node must be defined inside the `tick()` function.

- Interact with the elevator. The interaction with the elevator system includes commands to open the elevator door and to move the elevator to the desired floor. The elevator server facilitates these two functions. To utilize these services, the robot must specify the current floor when requesting the "OpenDoor" service and the target floor when requesting the "MoveToFloor" service. The architecture of this node closely mirrors that of the "Load New Map" service node.

3.1.2 Decorator node plugin

Decorator plugin modifies the behavior of its child node by adding conditions, changing the success or failure logic, or limiting the number of times a child can run. The behavior tree `navigate_to_pose_w_replanning_goal_patience_and_recovery.`, referenced in Section 2.2.1, incorporates a decorator node "PathLongerOnApproach". This node's purpose is to find out whether a newly calculated path is longer than the one currently being traversed. Similarly, a decorator node could be utilized to determine whether the robot is positioned in front of the door or has entered the elevator by checking if the robot's distance to a predefined point, representing the door or the elevator, falls below a specified threshold. The detailed structure is as follows:

The constructor initializes the behavior tree node, specifying its type along with the global and robot frame references (Listing 3.3).

```cpp
// robot_near_door.cpp
RobotNearDoor::RobotNearDoor(
        const std::string & name,
        const BT::NodeConfiguration & conf)
        : BT::DecoratorNode(name, conf),
        global_frame_("map"),
        robot_base_frame_("base_link")
    {
        auto node = config().blackboard->get<rclcpp::Node::SharedPtr>("node");
        node_ = config().blackboard->get<rclcpp::Node::SharedPtr>("node");
        tf_ = config().blackboard->get<std::shared_ptr<tf2_ros::Buffer>>("
    tf_buffer");

        getInput("global_frame", global_frame_);
        getInput("robot_base_frame", robot_base_frame_);
    }
```

Listing 3.3: Configuration function of the decorator node

Listing 3.4 obtains the robot's current position and establishes a distance threshold, subsequently verifying whether the distance between the robot and the door is less than this threshold.

```cpp
// robot_near_door.cpp
bool RobotNearDoor::isRobotNearDoor()
    {
        geometry_msgs::msg::PoseStamped current_pose;
        nav2_util::declare_parameter_if_not_declared(
            node_, "goal_reached_tol",
            rclcpp::ParameterValue(0.5));
        node_->get_parameter_or<double>("goal_reached_tol", goal_reached_tol_,
    0.5);
        tf_ = config().blackboard->get<std::shared_ptr<tf2_ros::Buffer>>("
    tf_buffer");
        node_->get_parameter("transform_tolerance", transform_tolerance_);
        if (!nav2_util::getCurrentPose(current_pose, *tf_, global_frame_,
    robot_base_frame_, transform_tolerance_))
        {
            RCLCPP_DEBUG(node_->get_logger(), "Current robot pose is not available
    .");
            return false;
        }
        pose_door = config().blackboard->get<geometry_msgs::msg::PoseStamped>("
    doorpose");

        double dx = pose_door.pose.position.x - current_pose.pose.position.x;
        double dy = pose_door.pose.position.y - current_pose.pose.position.y;
        return (dx * dx + dy * dy) <= (goal_reached_tol_ * goal_reached_tol_);
    }
```

Listing 3.4: Function to calculate the distance between the robot and the door

A flag is established to ensure that this node is ticked only when the robot is in front of the door for the first time. The status of this node is determined based on varying situations. The child node should include the interface with the elevator system for opening the door (Listing 3.5).

```cpp
// robot_near_door.cpp
inline BT::NodeStatus RobotNearDoor::tick()
    {
        if (status() == BT::NodeStatus::IDLE)
        {
            first_time_ = true;
        }

        setStatus(BT::NodeStatus::RUNNING);
        passed_door_ = config().blackboard->get<bool>("doorflag");

        if (!passed_door_ && isRobotNearDoor() && !first_time_)
        {
            const BT::NodeStatus child_state = child_node_->executeTick();
            switch (child_state)
            {
                case BT::NodeStatus::RUNNING:
                    return BT::NodeStatus::RUNNING;
                case BT::NodeStatus::SUCCESS: {
                    passed_door_ = true;
                    config().blackboard->set<bool>("doorflag", passed_door_);
                    return BT::NodeStatus::SUCCESS;
                }
                case BT::NodeStatus::FAILURE:
                    return BT::NodeStatus::FAILURE;
                default:
                    return BT::NodeStatus::FAILURE;
            }
        }
        first_time_ = false;
        return BT::NodeStatus::SUCCESS;
    }

#include "behaviortree_cpp_v3/bt_factory.h"
BT_REGISTER_NODES(factory)
{
  factory.registerNodeType<nav2_behavior_tree::RobotNearDoor>("RobotNearDoor");
}
```

Listing 3.5: Conditions for ticking the child node

3.1.3 Custom behavior tree

The new behavior tree implements the process of navigating between different floors. It includes two parts: a navigation subtree and a recovery subtree. The navigation subtree is responsible for the multi-floor navigation process and the recovery subtree handles the unexpected problems. The structure of the behavior tree and part of the corresponding file is shown below:

```xml
<!-- navigate_between_floors_w_replanning_and_recovery.xml -->
...
<ReactiveSequence name="MonitorAndFollowPath">
  <RobotNearDoor global_frame="map" robot_base_frame="base_link">
    <RetryUntilSuccessful num_attempts="1">
      <SequenceStar name="OpenTheDoor">
        <OpenDoor name="OpenTheDoorOfElevator" service_name="/open_door"/>
        <CancelControl name="ControlCancel"/>
        <Wait wait_duration="7"/>
        <MoveIntoElevator name="MoveInsideElevator"/>
      </SequenceStar>
    </RetryUntilSuccessful>
  </RobotNearDoor>
  <RobotInsideElevator global_frame="map" robot_base_frame="base_link">
    <RetryUntilSuccessful num_attempts="1">
      <SequenceStar name="ControlElevator">
        <SequenceStar name="CancelingControlAndWait">
          <CancelControl name="ControlCancel"/>
          <Wait wait_duration="4"/>
          <MoveToFloor name="UseElevatorToMoveToGoalFloor" service_name="/
    move_to_floor"/>
          <Wait wait_duration="3"/>
        </SequenceStar>
        <SequenceStar name="ChangingActions">
          <LoadMap name="LoadMap-Context" service_name="map_server/load_map"/>
          <Wait wait_duration="1"/>
          <SetGoal name="SetNewGoal-Context"/>
        </SequenceStar>
      </SequenceStar>
    </RetryUntilSuccessful>
  </RobotInsideElevator>
  <RecoveryNode number_of_retries="1" name="FollowPath">
    <FollowPath path="{path}" controller_id="FollowPath"/>
    <ClearEntireCostmap name="ClearLocalCostmap-Context" service_name="
    local_costmap/clear_entirely_local_costmap"/>
  </RecoveryNode>
</ReactiveSequence>
...
```

Listing 3.6: The navigation subtree and the recovery subtree of "navigate between floors"

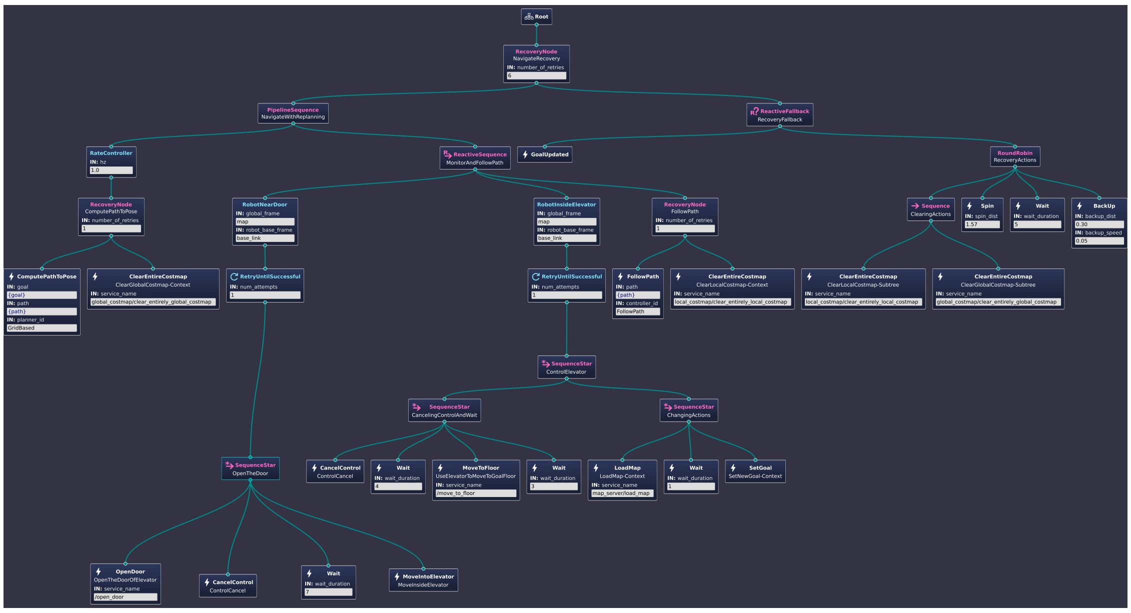

Root
RecoveryNode NavigateRecovery IN: number_of_retries 6
PipelineSequence NavigateWithReplanning
ReactiveFallback RecoveryFallback
RateController IN: hz 1.0
ReactiveSequence MonitorAndFollowPath
GoalUpdated
RoundRobin RecoveryActions
RecoveryNode ComputePathToPose IN: number_of_retries 1
RobotNearDoor IN: global_frame map IN: robot_base_frame base_link
RobotInsideElevator IN: global_frame map IN: robot_base_frame base_link
RecoveryNode FollowPath IN: number_of_retries 1
Sequence ClearingActions
Spin IN: spin_dist 1.57
Wait IN: wait_duration 5
BackUp IN: backup_dist 0.30 IN: backup_speed 0.05
ComputePathToPose IN: goal {goal} IN: path {path} IN: planner_id GridBased
ClearEntireCostmap ClearGlobalCostmap-Context IN: service_name global_costmap/clear_entirely_global_costmap
RetryUntilSuccessful IN: num_attempts 1
RetryUntilSuccessful IN: num_attempts 1
FollowPath IN: path {path} IN: controller_id FollowPath
ClearEntireCostmap ClearLocalCostmap-Context IN: service_name local_costmap/clear_entirely_local_costmap
ClearEntireCostmap ClearLocalCostmap-Subtree IN: service_name local_costmap/clear_entirely_local_costmap
ClearEntireCostmap ClearGlobalCostmap-Subtree IN: service_name global_costmap/clear_entirely_global_costmap
SequenceStar ControlElevator
SequenceStar CancelingControlAndWait
SequenceStar ChangingActions
SequenceStar OpenTheDoor
CancelControl ControlCancel
Wait IN: wait_duration 4
MoveToFloor UseElevatorToMoveToGoalFloor IN: service_name /move_to_floor
Wait IN: wait_duration 3
LoadMap LoadMap-Context IN: service_name map_server/load_map
Wait IN: wait_duration 1
SetGoal SetNewGoal-Context
OpenDoor OpenTheDoorOfElevator IN: service_name /open_door
CancelControl ControlCancel
Wait IN: wait_duration 7
MoveIntoElevator MoveInsideElevator

Figure 3.5: Navigate_between_floors_w_replanning_and_recovery

During navigation, as the robot approaches the elevator door, the first branch of "MonitorAndFollowPath" is ticked(Fig. 3.6). The robot stops its movement and holds its position. Subsequently, it calls the service to open the door and remains stationary until the door is fully open. Following this, a new point located inside the elevator is designated as a preemptive goal, prompting the robot to enter and proceed towards its objective.

As the robot enters the elevator and moves near the current goal, the second branch is ticked. The robot halts all movement and waits for further instructions. It then requests the "Move to Floor" service. Concurrently, it calls the service to update the navigation map to the new floor. Once the elevator reaches the destination floor and the map is updated, the final navigation goal is sent to the robot, converting the process into a single-floor navigation task.

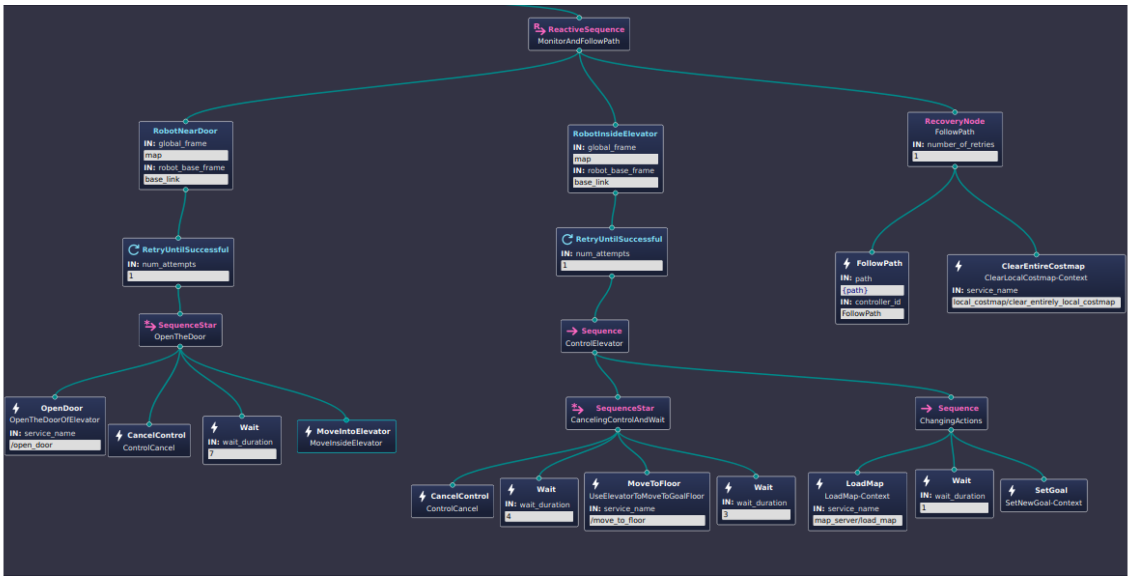


Figure 3.6: "MonitorAndFollowPath" branch of the behavior tree

3.2 Navigator plugin

The newly developed navigator, `"Navigate between floors,"` operates in conjunction with the behavior tree `navigate_between_floors_w_replanning_and_recovery`. It subscribes to the `"goal_pose"` topic, passing the goal to the behavior tree and triggering the navigation process. A primary function of this navigator is to determine whether the goal is on the same floor as the robot. This determination is facilitated by the fallback function `"goalReceived"`, which uses the z-coordinate to represent floor level. Once receiving a goal, the function compares the z-coordinate of the goal with that of the robot's current position. If they match, indicating that the navigation goal is on the same floor, the goal is sent directly to the behavior tree. Conversely, if they are different, the goal is stored in the Blackboard as the final destination. Meanwhile, the pose of the elevator on the robot's current floor is sent as the intermediate goal to the behavior tree, prompting the robot to navigate toward the elevator until the `"MonitorAndFollowPath"` branch is ticked.

An additional function of the navigator, as described in Section 3.1.2, is to reset the flag that ensures the robot stops only once in front of the door during a navigation task. This flag must be reset after the completion of the navigation process to ready the system for subsequent navigation tasks. The resetting occurs within the `goalCompleted` callback function. The values of these flags are maintained on the blackboard, allowing decorator nodes to access them as needed. The detailed structure is shown Listing 3.7:

```
void
NavigateBetweenFloorsNavigator::goalCompleted(
typename ActionT::Result::SharedPtr /*result*/,
const nav2_behavior_tree::BtStatus /*final_bt_status*/)
{
    auto blackboard = bt_action_server_->getBlackboard();
    door_flag_ = false;
    blackboard->set<bool>("doorflag", door_flag_);
    elevator_flag_ = false;
    blackboard->set<bool>("elevatorflag", elevator_flag_);
    std::cout << "SET THE Flags OF DOOR AS " << door_flag_ << std::endl;
    std::cout << "SET THE Flags OF ELEVATOR AS " << elevator_flag_ << std::endl;
}
```

Listing 3.7: Function to reset the flags after a navigation process

3.3 Elevator server

LINKS provides an API for interacting with the elevator system. The two services used in the navigation process are "OpenDoor" and "MoveToFloor". The "OpenDoor" service contains the steps to open the door, hold it open for a specified duration, and then close the door. The "MoveToFloor" service manages the elevator's movement to the designated floor, including the steps to open the door upon arrival, maintain it open for a set time, and subsequently close the door. To integrate the elevator system with the navigation package, an elevator server must be established to serve as the intermediary between the behavior tree nodes and the elevator system.

3.3.1 Open door service

Considering the scenario where the robot may not be on the same floor as the elevator, the request must contain information about the robot's current floor. Upon receiving the request, the server first calls for the elevator to move to the robot's floor and then opens the door to allow the robot to enter. Listing 3.8 is the message type for the `"OpenDoor"` service and Listing 3.9 is the detailed architecture of the server.

```
int32 rfloor   # stand for robot's current floor
---
bool success   # indicate successful run of open door service
```

Listing 3.8: OpenDoor.srv

```
void opendoor_handle_service(
    const std::shared_ptr<rmw_request_id_t> request_header,
    const std::shared_ptr<OpenDoor::Request> request,
    const std::shared_ptr<OpenDoor::Response> response)
{
    (void)request_header;
    RCLCPP_INFO(g_node->get_logger(),"Call the service to open the door");
    elevetor_pub.moveToFloor(request->rfloor);
    //elevetor_pub.openDoor();
    response->success = true;
}
```

Listing 3.9: Open door service

The functionality of the "OpenDoor" service could be enhanced by introducing a parameter that indicates the elevator's current floor. Upon receiving a request, the server could then compare the elevator's floor with the robot's floor. If they match, the server would simply execute the "OpenDoor" service; if not, it would proceed with the "MoveToFloor" service.

3.3.2 Move to floor service

The floor level within the elevator system is determined by height. For instance, if the vertical distance between two adjacent floors is 4 meters, moving to the first floor would involve elevating 4 meters above the ground level, while reaching the second floor would require ascending 8 meters. If the elevator already resides on the requested floor, it will execute the same sequence as the "OpenDoor" service. Listing 3.10 is the message type for the "MoveToFloor" service and Listing 3.11 is the detailed structure of the server.

```
int32 floor    # stand for request floor
---
bool success   # indicate successful run of move to floor service
```

Listing 3.10: MoveToFloor.srv

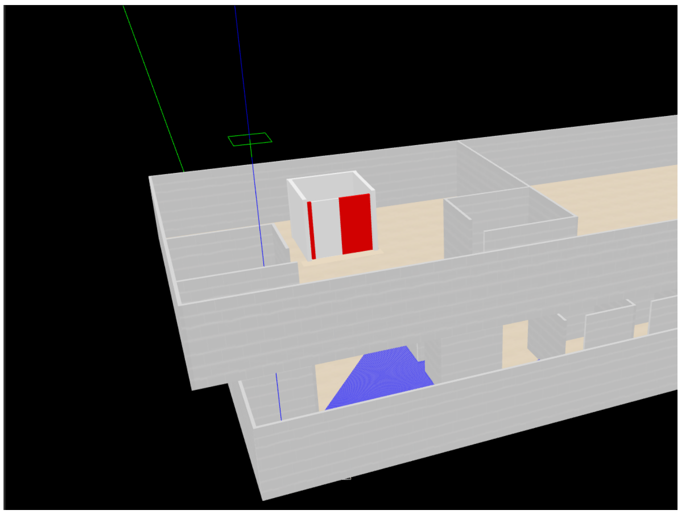
```
void movetofloor_handle_service(
    const std::shared_ptr<rmw_request_id_t> request_header,
    const std::shared_ptr<MoveToFloor::Request> request,
    const std::shared_ptr<MoveToFloor::Response> response)
{
    (void)request_header;
    RCLCPP_INFO(g_node->get_logger(),"Call the service to move to floor %" PRId32,
     request->floor);
    elevator_pub.moveToFloor(request->floor);
    response->success = true;
}
```

Listing 3.11: Move to floor service

The results of sending the request to floor 1 and floor 2 are presented in Fig. 3.7.

3.4 Simulation

After developing the behavior tree node plugins and the new bt navigator, the subsequent step involves integrating them into the navigation process. The behavior tree nodes must be added to the library in the CMakeList file. Within the parameter file, both the path to the default behavior tree and the names of the node plugins should be specified. Listing 3.12 is the BT navigator structure in the parameter file.


(a) Move to Floor 1

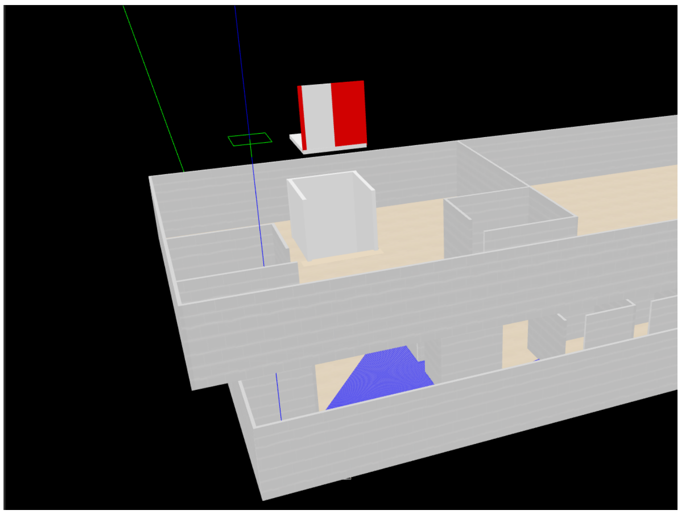

(b) Move to Floor 2

Figure 3.7: Execution of the "MoveToFloor" service

```yaml
# multi_floor_nav2_params.yaml
bt_navigator:
  ros__parameters:
    use_sim_time: True
    global_frame: map
    robot_base_frame: base_link
    odom_topic: /odom
    bt_loop_duration: 10
    default_server_timeout: 20
    # Simulation behavior tree
    default_nav_between_floors_bt_xml: "/home/links-webots/ros2_ws/src/navigation2
    /nav2_bt_navigator/behavior_trees/navigate_between_floors.xml"
    plugin_lib_names:
      - nav2_robot_inside_elevator_bt_node
      - nav2_robot_near_door_bt_node
      - nav2_load_map_service_bt_node
      - nav2_get_costmap_service_bt_node
      - nav2_set_initial_pose_bt_node
      - nav2_set_goal_bt_node
      - nav2_move_into_elevator_bt_node
      - nav2_open_door_service_bt_node
      - nav2_move_to_floor_service_bt_node
      - nav2_press_door_service_bt_node
      - nav2_press_target_service_bt_node
      [...]
```

Listing 3.12: BT navigator configuration file

To enhance practicality, an additional parameter file (Listing 3.13) is created to specify the robot's initial floor and other parameters used in navigation. Within this file, the coordinates of the elevator and its doors on each floor are stored in arrays, while the map paths are kept in a string array.

```
# nav2_multi_params.yaml
bt_navigator:
  #simulation params
   ros__parameters:
    initial_floor: 0

    x_elevator: [4.0, 4.0]
    y_elevator: [6.5, 6.5]

    x_door: [3.0, 3.0]
    y_door: [5.0, 5.0]

    floor_maps: ["/home/links-webots/ros2_ws/src/
    links_office_multifloor_simulation/maps/links_floor0_open.yaml",
                  "/home/links-webots/ros2_ws/src/
    links_office_multifloor_simulation/maps/links_floor1_open.yaml"]
```

Listing 3.13: Parameters used in the navigation

In this example, the coordinates of the elevator on the floor 0 are (4.0, 6.0), and the door is at (3.0, 5.0). The coordinates for the elevator and door remain consistent for floor 0 and floor 1 because the position of the map frame does not change between these two levels (Fig. 3.8).

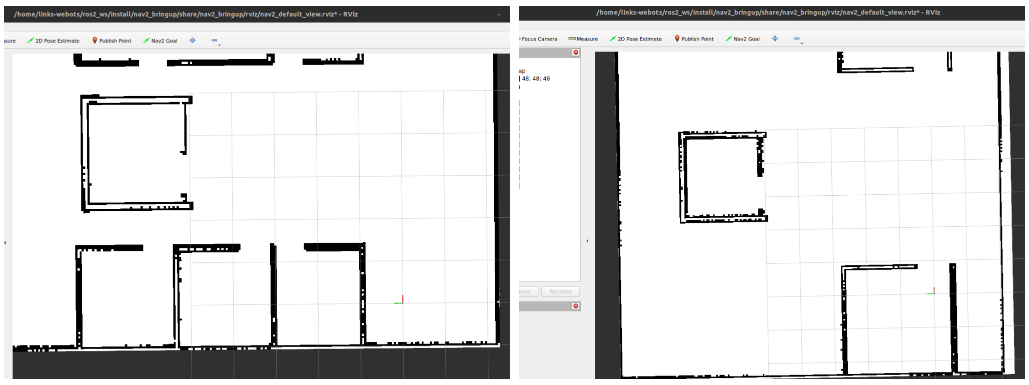

/home/links-webots/ros2_ws/install/nav2_bringup/share/nav2_bringup/rviz/nav2_default_view.rviz* - RViz
2D Pose Estimate
Publish Point
Nav2 Goal
Focus Camera
Measure

Figure 3.8: Map frame in floor 0 (left) and floor 1 (right))

To utilize this file in navigation, it must be declared in configuration files (Listing 3.14). Within the `nav2_bringup` folder, there is a launch file named `navigation_launch.py`, which is responsible for starting up various components required for the Nav2 stack. This launch file also allows for the configuration of various parameters and settings, crucial for customizing the navigation behavior. The file `nav2_multi_params.yaml` is included in these declarations.

```
# navigation_launch.py
def generate_launch_description():
    [...]
    # Create our own temporary YAML files that include substitutions
    multi_configured_params = RewrittenYaml(
            source_file=multi_params_file,
            root_key=namespace,
            param_rewrites=param_substitutions,
            convert_types=True)
    declare_multi_params_file_cmd = DeclareLaunchArgument(
        'multi_params_file',
        default_value=os.path.join(bringup_dir, 'params', 'nav2_multi_params.yaml
    '),
        description='Full path to the ROS2 parameters file to use for all launched
     nodes')
    [...]
```

Listing 3.14: Declare the parameters in the launch file

Use the following command to launch Gazebo and Rviz. This navigation process runs locally.

```
ros2 launch links_multifloor_simulation gazebo_sim.launch.py world:=/home/links-
    webots/ros2_ws/src/links_office_multifloor_simulation/worlds/
    links_office_multi_half.world
```

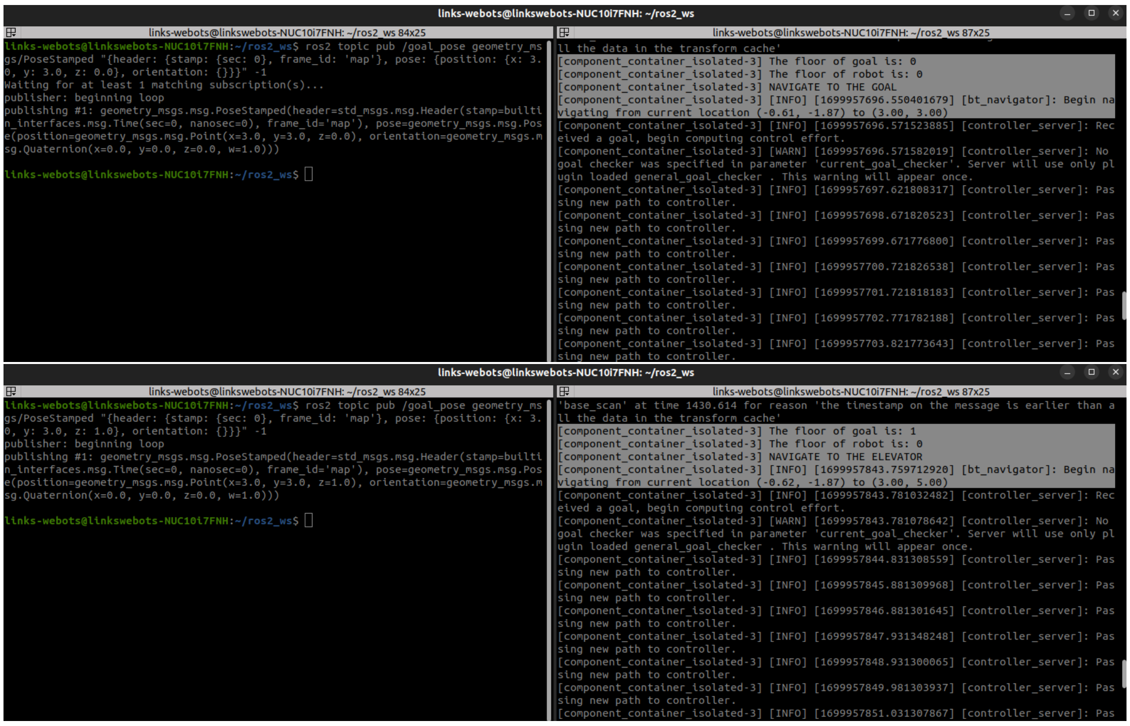
Listing 3.15: Launch Gazebo

```
ros2 launch links_multifloor_simulation tb3_multifloor_sim.launch.py use_simulator
    :=False params_file:=/home/links-webots/ros2_ws/src/behavior_tree/params/
    multi_floor_nav2_params.yaml multi_params_file:=/home/links-webots/ros2_ws/src
    /navigation2/nav2_bringup/params/nav2_multi_params.yaml
```

Listing 3.16: Launch Rviz

After starting the elevator server and setting the robot's initial position, the navigation goal can be transmitted via the `"goal_pose"` topic. The z-coordinate of the goal corresponds to the floor information. An example of the robot's response to goals on different floors is illustrated in Fig. 3.9.

Fig. 3.10 is the entire process of a multi-floor navigation from floor 0 to floor 1.


links-webots@linkswebots-NUC10i7FNH: ~/ros2_ws
links-webots@linkswebots-NUC10i7FNH: ~/ros2_ws 84x25
links-webots@linkswebots-NUC10i7FNH: ~/ros2_ws 87x25

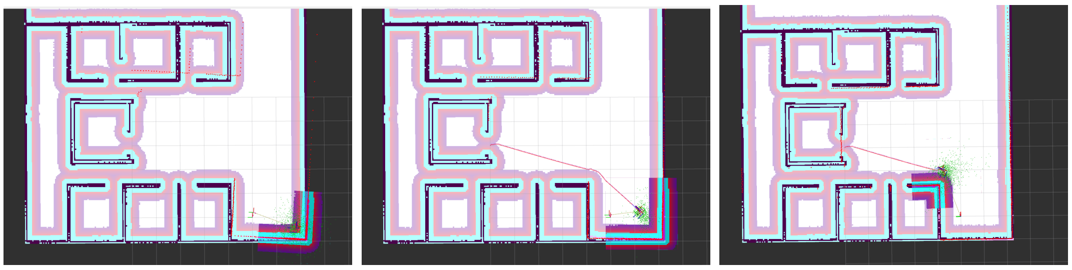
```
links-webots@linkswebots-NUC10i7FNH:~/ros2_ws$ ros2 topic pub /goal_pose geometry_ms
gs/PoseStamped "{header: {stamp: {sec: 0}, frame_id: 'map'}, pose: {position: {x: 3.
0, y: 3.0, z: 0.0}, orientation: {}}}" -1
Waiting for at least 1 matching subscription(s)...
publisher: beginning loop
publishing #1: geometry_msgs.msg.PoseStamped(header=std_msgs.msg.Header(stamp=builti
n_interfaces.msg.Time(sec=0, nanosec=0), frame_id='map'), pose=geometry_msgs.msg.Pos
e(position=geometry_msgs.msg.Point(x=3.0, y=3.0, z=0.0), orientation=geometry_msgs.m
sg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0)))

links-webots@linkswebots-NUC10i7FNH:~/ros2_ws$ 
```

```
ll the data in the transform cache'
[component_container_isolated-3] The floor of goal is: 0
[component_container_isolated-3] The floor of robot is: 0
[component_container_isolated-3] NAVIGATE TO THE GOAL
[component_container_isolated-3] [INFO] [1699957696.550401679] [bt_navigator]: Begin na
vigating from current location (-0.61, -1.87) to (3.00, 3.00)
[component_container_isolated-3] [INFO] [1699957696.571523885] [controller_server]: Rec
eived a goal, begin computing control effort.
[component_container_isolated-3] [WARN] [1699957696.571582019] [controller_server]: No
goal checker was specified in parameter 'current_goal_checker'. Server will use only pl
ugin loaded general_goal_checker . This warning will appear once.
[component_container_isolated-3] [INFO] [1699957697.621808317] [controller_server]: Pas
sing new path to controller.
[component_container_isolated-3] [INFO] [1699957698.671820523] [controller_server]: Pas
sing new path to controller.
[component_container_isolated-3] [INFO] [1699957699.671776800] [controller_server]: Pas
sing new path to controller.
[component_container_isolated-3] [INFO] [1699957700.721826538] [controller_server]: Pas
sing new path to controller.
[component_container_isolated-3] [INFO] [1699957701.721818183] [controller_server]: Pas
sing new path to controller.
[component_container_isolated-3] [INFO] [1699957702.771782188] [controller_server]: Pas
sing new path to controller.
[component_container_isolated-3] [INFO] [1699957703.821773643] [controller_server]: Pas
sing new path to controller.
```

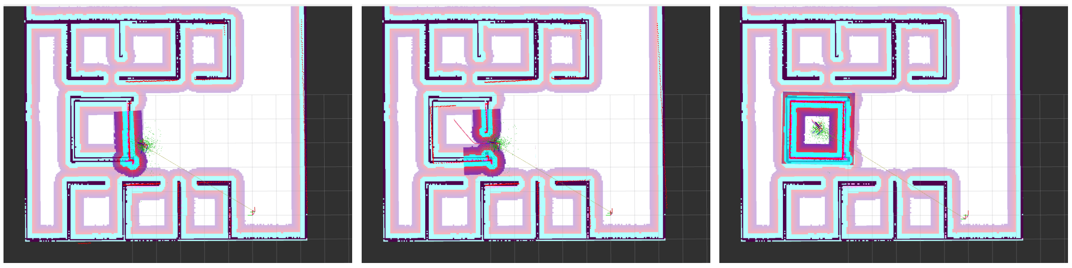
```
links-webots@linkswebots-NUC10i7FNH:~/ros2_ws$ ros2 topic pub /goal_pose geometry_ms
gs/PoseStamped "{header: {stamp: {sec: 0}, frame_id: 'map'}, pose: {position: {x: 3.
0, y: 3.0, z: 1.0}, orientation: {}}}" -1
publisher: beginning loop
publishing #1: geometry_msgs.msg.PoseStamped(header=std_msgs.msg.Header(stamp=builti
n_interfaces.msg.Time(sec=0, nanosec=0), frame_id='map'), pose=geometry_msgs.msg.Pos
e(position=geometry_msgs.msg.Point(x=3.0, y=3.0, z=1.0), orientation=geometry_msgs.m
sg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0)))

links-webots@linkswebots-NUC10i7FNH:~/ros2_ws$ 
```

```
'base_scan' at time 1430.614 for reason 'the timestamp on the message is earlier than a
ll the data in the transform cache'
[component_container_isolated-3] The floor of goal is: 1
[component_container_isolated-3] The floor of robot is: 0
[component_container_isolated-3] NAVIGATE TO THE ELEVATOR
[component_container_isolated-3] [INFO] [1699957843.759712920] [bt_navigator]: Begin na
vigating from current location (-0.62, -1.87) to (3.00, 5.00)
[component_container_isolated-3] [INFO] [1699957843.781032482] [controller_server]: Rec
eived a goal, begin computing control effort.
[component_container_isolated-3] [WARN] [1699957843.781078642] [controller_server]: No
goal checker was specified in parameter 'current_goal_checker'. Server will use only pl
ugin loaded general_goal_checker . This warning will appear once.
[component_container_isolated-3] [INFO] [1699957844.831308559] [controller_server]: Pas
sing new path to controller.
[component_container_isolated-3] [INFO] [1699957845.881309968] [controller_server]: Pas
sing new path to controller.
[component_container_isolated-3] [INFO] [1699957846.881301645] [controller_server]: Pas
sing new path to controller.
[component_container_isolated-3] [INFO] [1699957847.931348248] [controller_server]: Pas
sing new path to controller.
[component_container_isolated-3] [INFO] [1699957848.931300065] [controller_server]: Pas
sing new path to controller.
[component_container_isolated-3] [INFO] [1699957849.981303937] [controller_server]: Pas
sing new path to controller.
[component_container_isolated-3] [INFO] [1699957851.031307867] [controller_server]: Pas
```

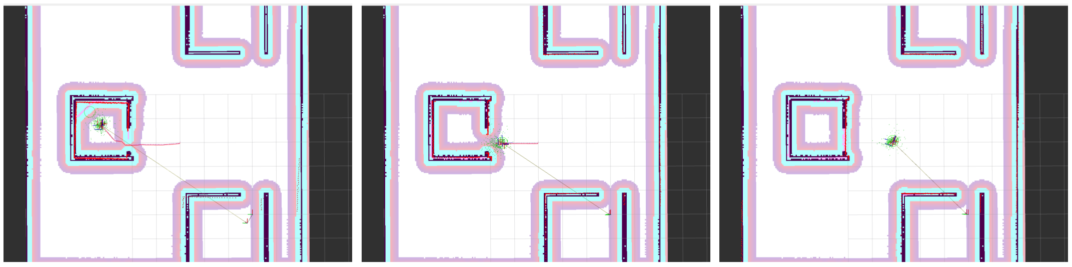
Figure 3.9: Single floor navigation(up) and multiple floor navigation(down)



(a) Navigate to the elevator in floor 0



(b) Navigate into the elevator and use it to move to floor 1



(c) Navigate to the final goal in floor 1

Figure 3.10: An entire multi-floor navigation process in Rviz

Chapter 4

Implementation

Building on the success of the multi-floor navigation simulation, this chapter introduces the transition from the virtual models to the real world application. Initially, the methodology employing a SwitchBot for elevator interaction is presented. Then this chapter introduces the TurtleBot used for testing, explaining the difference between the DWB (Dynamic Window Approach - B) controller and the MPPI (Model Predictive Path Integral) controller. Lastly, the creation of the map is described, followed by the test results.

4.1 SwitchBot

SwitchBot is a technology company focused on developing products that aim to enrich people's lives by enhancing interaction with the environment around them [51]. It offers devices that allow users to control home appliances via smartphones or voice commands, promote a smarter living experience. Their offerings include a variety of Bluetooth-compatible devices such as the SwitchBot Curtain, SwitchBot Blind Tilt (Fig.4.1).

(a) SwitchBot Curtain 3 [52]

(b) SwitchBot Blind Tilt [53]

Figure 4.1: SwitchBot products

One of the important products, the SwitchBot Bot (Fig.4.2), is a mechanical button pusher that can turn traditional appliances into smart devices [54]. It is designed for simplicity, with easy setup and compatibility with Android and iOS platforms. The device operates on a CR2 3V battery and can communicate over Bluetooth, making it a flexible addition to any smart home system.

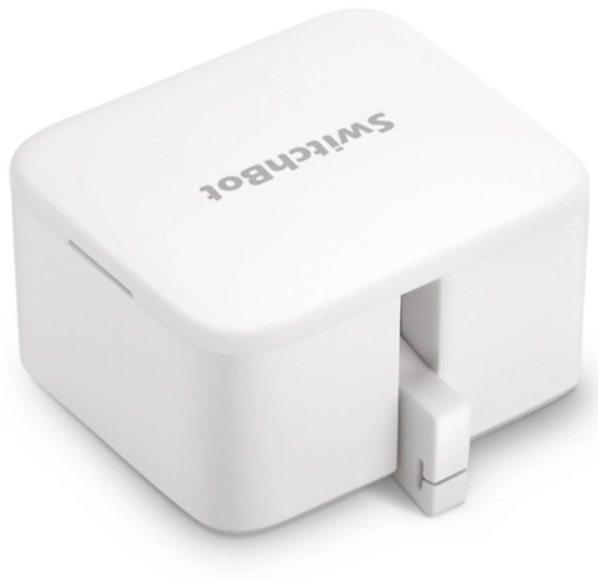


Figure 4.2: SwitchBot Bot [55]

Fig. 4.3 presents the work process of a SwitchBot Bot. It has a mechanical finger that can push or flip buttons and switches on various devices. SwitchBot is utilized to make interaction with the elevator throughout the navigation process. When the robot arrives in front of the elevator door, the first SwitchBot is activated to call the elevator. Once the door opens, the robot proceeds into the elevator, and a second SwitchBot is deployed to press the button for the desired floor.

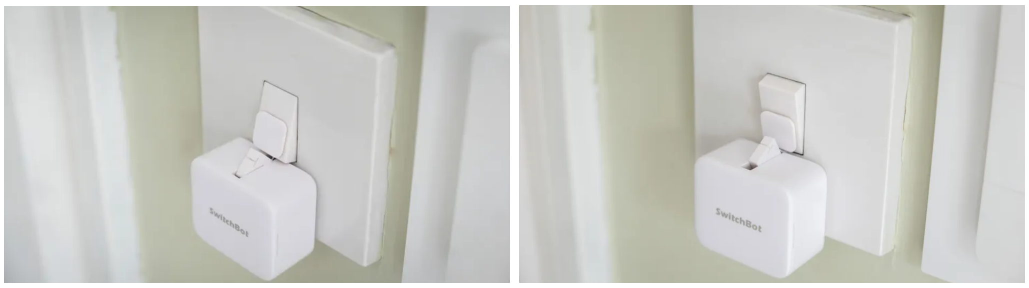


Figure 4.3: Use SwitchBot to turn on a light

Several methods exist for activating a SwitchBot device. The simplest approach is to manually control the device using the SwitchBot app on a smartphone, which connects via Bluetooth and is subject to a limited operating range. Alternatively, with the addition of a SwitchBot Hub, the device can be controlled remotely from any location via the internet. In this thesis, a MQTT broker is employed to issue commands to the SwitchBot locally.

4.1.1 MQTT protocol

MQTT (Message Queuing Telemetry Transport) is a protocol widely adopted for IoT communications, promoting data exchange between devices like sensors and actuators over the internet [56]. It is designed to be lightweight and efficient, demanding minimal resources, which makes it ideal for devices with limited processing capabilities and operating over networks with low bandwidth. Key components and features of MQTT include:

- MQTT Clients. These are devices or applications that publish messages to a broker or subscribe to message topics to receive information. Clients are lightweight and can be efficiently deployed on constrained devices.
- MQTT Broker. The broker is the server that receives all messages from the clients and then distributes them to the relevant subscribers. It acts as an intermediary that ensures messages are directed to the right destinations based on the topic subscriptions [57].
- Quality of Service (QoS) Levels. MQTT supports three levels of QoS to ensure message delivery even over unreliable networks: QoS 0: At most once delivery, QoS 1: At least once delivery, QoS 2: Exactly once delivery.

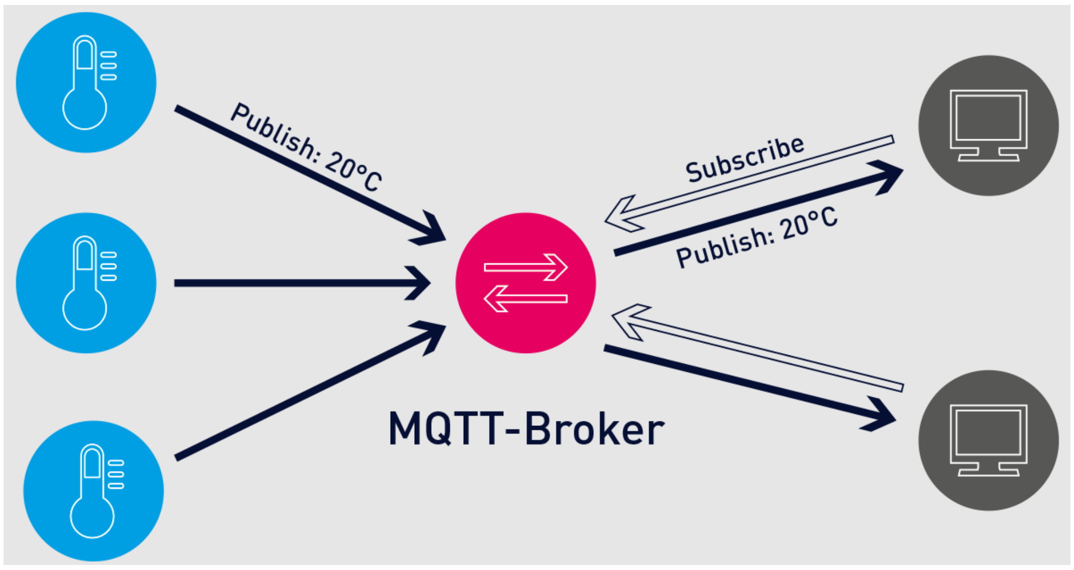
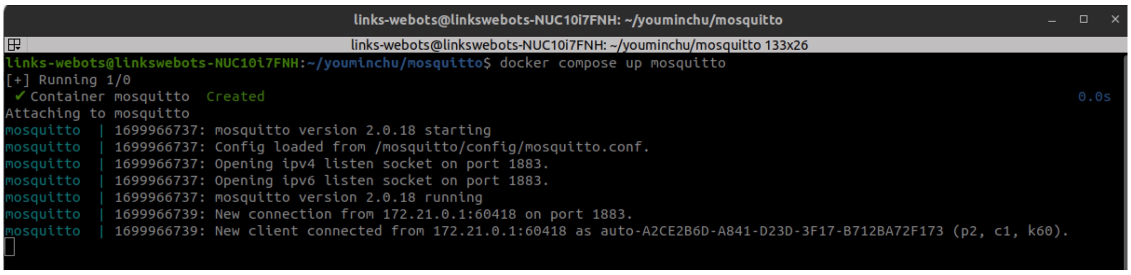


Figure 4.4: Working example of MQTT [58]

The MQTT broker is created through Docker, an open-source platform that automates the deployment of applications inside software containers, providing an additional layer of abstraction and automation of OS-level virtualization on Linux. Listing 4.1 is the detailed structure. This will create an MQTT broker on a local IP address (Fig. 4.5).

```yaml
version: "3.7"
services:
  mosquitto:
    container_name: mosquitto
    image: eclipse-mosquitto
    ports:
      - "1883:1883" #default mqtt port
    volumes:
      - ./config:/mosquitto/config:rw
      - ./data:/mosquitto/data:rw
      - ./log:/mosquitto/log:rw
```

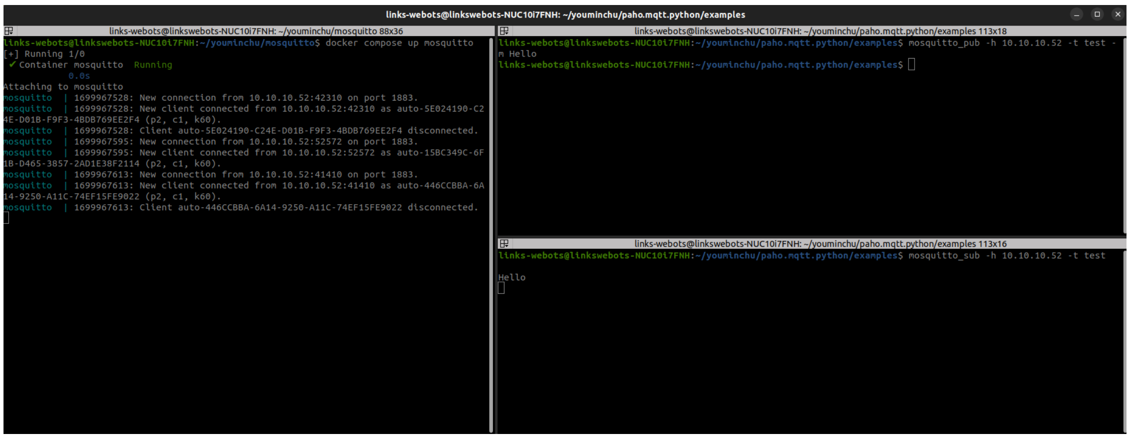
Listing 4.1: Docker-compose.yaml



```
links-webots@linkswebots-NUC10i7FNH:~/youminchu/mosquitto$ docker compose up mosquitto
[+] Running 1/0
 ✔ Container mosquitto  Created                                                                              0.0s
Attaching to mosquitto
mosquitto  | 1699966737: mosquitto version 2.0.18 starting
mosquitto  | 1699966737: Config loaded from /mosquitto/config/mosquitto.conf.
mosquitto  | 1699966737: Opening ipv4 listen socket on port 1883.
mosquitto  | 1699966737: Opening ipv6 listen socket on port 1883.
mosquitto  | 1699966737: mosquitto version 2.0.18 running
mosquitto  | 1699966739: New connection from 172.21.0.1:60418 on port 1883.
mosquitto  | 1699966739: New client connected from 172.21.0.1:60418 as auto-A2CE2B6D-A841-D23D-3F17-B712BA72F173 (p2, c1, k60).
```

Figure 4.5: Creation of an MQTT broker

Fig. 4.6 presents an example demonstrating the use of the Mosquitto broker for publishing and receiving messages. In this example, the broker's address is "10.10.10.52". The command `mosquitto_pub` is used to publish a message 'Hello' to the topic 'test'. The broker then transmits this message to all clients subscribed to this topic.



```
links-webots@linkswebots-NUC10i7FNH:~/youminchu/mosquitto$ docker compose up mosquitto
[+] Running 1/0
 ✔ Container mosquitto  Running
           0.0s
Attaching to mosquitto
mosquitto  | 1699967528: New connection from 10.10.10.52:42310 on port 1883.
mosquitto  | 1699967528: New client connected from 10.10.10.52:42310 as auto-5E024190-C2
4E-D01B-F9F3-4BDB769EE2F4 (p2, c1, k60).
mosquitto  | 1699967528: Client auto-5E024190-C24E-D01B-F9F3-4BDB769EE2F4 disconnected.
mosquitto  | 1699967595: New connection from 10.10.10.52:52572 on port 1883.
mosquitto  | 1699967595: New client connected from 10.10.10.52:52572 as auto-15BC349C-6F
1B-D465-3857-2AD1E38F2114 (p2, c1, k60).
mosquitto  | 1699967613: New connection from 10.10.10.52:41410 on port 1883.
mosquitto  | 1699967613: New client connected from 10.10.10.52:41410 as auto-446CCBBA-6A
14-9250-A11C-74EF15FE9022 (p2, c1, k60).
mosquitto  | 1699967613: Client auto-446CCBBA-6A14-9250-A11C-74EF15FE9022 disconnected.
```

```
links-webots@linkswebots-NUC10i7FNH:~/youminchu/paho.mqtt.python/examples$ mosquitto_pub -h 10.10.10.52 -t test -
m Hello
links-webots@linkswebots-NUC10i7FNH:~/youminchu/paho.mqtt.python/examples$
```

```
links-webots@linkswebots-NUC10i7FNH:~/youminchu/paho.mqtt.python/examples$ mosquitto_sub -h 10.10.10.52 -t test

Hello
```

Figure 4.6: Communication in the MQTT broker

4.1.2 SwitchBot server

To integrate the SwitchBot into the navigation process, a corresponding server, as the elevator server used in the simulation, must be established. This SwitchBot server processes requests from the robot to activate the Bot, thereby fulfilling the "OpenDoor" and "MoveToFloor" services. To accomplish this, multiple libraries are utilized.

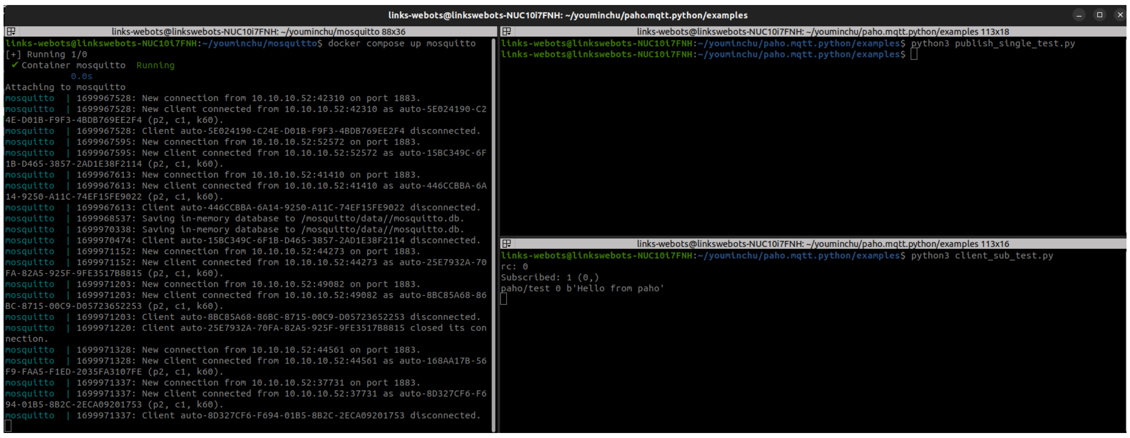
paho-mqtt

The "paho-mqtt" library is a client library for the MQTT protocol [59], providing an implementation for Python. It enables the development of applications that can communicate with an MQTT broker to publish messages to the broker, and to subscribe to topics and receive published messages (Fig. 4.7).

This library offers a module designed for the creation and utilization of a client, enabling the client to either publish or subscribe to a topic. An example of its usage is provided in Listing 4.2.

```python
# client_sub_test.py
import context  # Ensures paho is in PYTHONPATH
import paho.mqtt.client as mqtt
[...]
mqttc = mqtt.Client()
mqttc.on_message = on_message
mqttc.on_connect = on_connect
mqttc.on_publish = on_publish
mqttc.on_subscribe = on_subscribe

mqttc.connect("10.10.10.52", 1883, 60)
mqttc.subscribe("paho/test", 0)

mqttc.loop_forever()
```

Listing 4.2: Use paho-mqtt to create an MQTT client



links-webots@linkswebots-NUC10i7FNH: ~/youminchu/paho.mqtt.python/examples
links-webots@linkswebots-NUC10i7FNH: ~/youminchu/mosquitto 88x36
links-webots@linkswebots-NUC10i7FNH: ~/youminchu/paho.mqtt.python/examples 113x18
links-webots@linkswebots-NUC10i7FNH: ~/youminchu/paho.mqtt.python/examples 113x16

```
links-webots@linkswebots-NUC10i7FNH:~/youminchu/mosquitto$ docker compose up mosquitto
[+] Running 1/0
 ✔ Container mosquitto  Running
           0.0s
Attaching to mosquitto
mosquitto  | 1699967528: New connection from 10.10.10.52:42310 on port 1883.
mosquitto  | 1699967528: New client connected from 10.10.10.52:42310 as auto-5E024190-C2
4E-D01B-F9F3-4BDB769EE2F4 (p2, c1, k60).
mosquitto  | 1699967528: Client auto-5E024190-C24E-D01B-F9F3-4BDB769EE2F4 disconnected.
mosquitto  | 1699967595: New connection from 10.10.10.52:52572 on port 1883.
mosquitto  | 1699967595: New client connected from 10.10.10.52:52572 as auto-15BC349C-6F
1B-D465-3857-2AD1E38F2114 (p2, c1, k60).
mosquitto  | 1699967613: New connection from 10.10.10.52:41410 on port 1883.
mosquitto  | 1699967613: New client connected from 10.10.10.52:41410 as auto-446CCBBA-6A
14-9250-A11C-74EF15FE9022 (p2, c1, k60).
mosquitto  | 1699967613: Client auto-446CCBBA-6A14-9250-A11C-74EF15FE9022 disconnected.
mosquitto  | 1699968537: Saving in-memory database to /mosquitto/data//mosquitto.db.
mosquitto  | 1699970338: Saving in-memory database to /mosquitto/data//mosquitto.db.
mosquitto  | 1699970474: Client auto-15BC349C-6F1B-D465-3857-2AD1E38F2114 disconnected.
mosquitto  | 1699971152: New connection from 10.10.10.52:44273 on port 1883.
mosquitto  | 1699971152: New client connected from 10.10.10.52:44273 as auto-25E7932A-70
FA-82A5-925F-9FE3517B8815 (p2, c1, k60).
mosquitto  | 1699971203: New connection from 10.10.10.52:49082 on port 1883.
mosquitto  | 1699971203: New client connected from 10.10.10.52:49082 as auto-8BC85A68-86
BC-8715-00C9-D05723652253 (p2, c1, k60).
mosquitto  | 1699971203: Client auto-8BC85A68-86BC-8715-00C9-D05723652253 disconnected.
mosquitto  | 1699971220: Client auto-25E7932A-70FA-82A5-925F-9FE3517B8815 closed its con
nection.
mosquitto  | 1699971328: New connection from 10.10.10.52:44561 on port 1883.
mosquitto  | 1699971328: New client connected from 10.10.10.52:44561 as auto-168AA17B-56
F9-FAA5-F1ED-2035FA3107FE (p2, c1, k60).
mosquitto  | 1699971337: New connection from 10.10.10.52:37731 on port 1883.
mosquitto  | 1699971337: New client connected from 10.10.10.52:37731 as auto-8D327CF6-F6
94-01B5-8B2C-2ECA09201753 (p2, c1, k60).
mosquitto  | 1699971337: Client auto-8D327CF6-F694-01B5-8B2C-2ECA09201753 disconnected.
```

```
links-webots@linkswebots-NUC10i7FNH:~/youminchu/paho.mqtt.python/examples$ python3 publish_single_test.py
links-webots@linkswebots-NUC10i7FNH:~/youminchu/paho.mqtt.python/examples$
```

```
links-webots@linkswebots-NUC10i7FNH:~/youminchu/paho.mqtt.python/examples$ python3 client_sub_test.py
rc: 0
Subscribed: 1 (0,)
paho/test 0 b'Hello from paho'
```

Figure 4.7: Communication in a mosquitto broker using paho-mqtt library

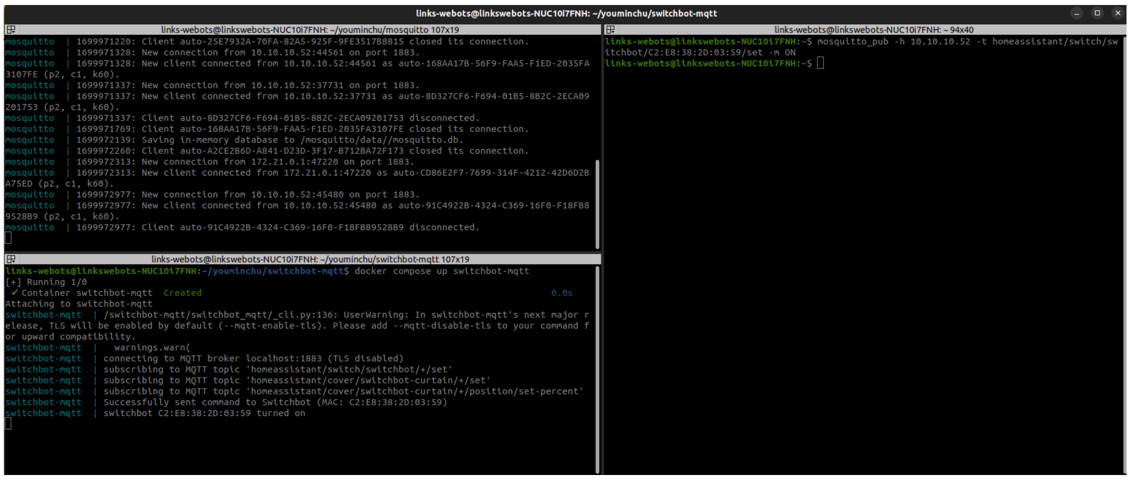
switchbot-mqtt

The 'switchbot-mqtt' library is an MQTT client for controlling SwitchBot automation buttons and curtain motors, and it's compatible with Home Assistant's MQTT Switch and MQTT Cover platform [60]. It can send 'ON' or OFF commands to a specified MQTT topic to control SwitchBot devices. There's also functionality to fetch and report the battery level of the devices. It is also created through Docker.

```
version: '3.8'
services:
  switchbot-mqtt:
    image: switchbot-mqtt
    container_name: switchbot-mqtt
    network_mode: host
    userns_mode: host
    environment:
    - MQTT_HOST=localhost
    - MQTT_PORT=1883
    #- MQTT_USERNAME=username
    #- MQTT_PASSWORD=password
    #- FETCH_DEVICE_INFO=yes
    restart: unless-stopped
    privileged: true
```

Listing 4.3: Docker-compose.yaml

Once it is activated, the "switchbot-mqtt" automatically scans the devices and creates an MQTT topic for each one, uniquely identified by their MAC address. Request to turn on or off the Bot can be sent to the topic `homeassistant/switch/switchbot/ADDRESS`.



```
links-webots@linkswebots-NUC10i7FNH: ~/youminchu/mosquitto 107x19
mosquitto  | 1699971220: Client auto-25E7932A-70FA-82A5-925F-9FE3517B8815 closed its connection.
mosquitto  | 1699971328: New connection from 10.10.10.52:44561 on port 1883.
mosquitto  | 1699971328: New client connected from 10.10.10.52:44561 as auto-168AA17B-56F9-FAA5-F1ED-2035FA
3107FE (p2, c1, k60).
mosquitto  | 1699971337: New connection from 10.10.10.52:37731 on port 1883.
mosquitto  | 1699971337: New client connected from 10.10.10.52:37731 as auto-8D327CF6-F694-01B5-8B2C-2ECA09
201753 (p2, c1, k60).
mosquitto  | 1699971337: Client auto-8D327CF6-F694-01B5-8B2C-2ECA09201753 disconnected.
mosquitto  | 1699971769: Client auto-168AA17B-56F9-FAA5-F1ED-2035FA3107FE closed its connection.
mosquitto  | 1699972139: Saving in-memory database to /mosquitto/data//mosquitto.db.
mosquitto  | 1699972260: Client auto-A2CE2B6D-A841-D23D-3F17-B712BA72F173 closed its connection.
mosquitto  | 1699972313: New connection from 172.21.0.1:47220 on port 1883.
mosquitto  | 1699972313: New client connected from 172.21.0.1:47220 as auto-CD86E2F7-7699-314F-4212-42D6D2B
A75ED (p2, c1, k60).
mosquitto  | 1699972977: New connection from 10.10.10.52:45480 on port 1883.
mosquitto  | 1699972977: New client connected from 10.10.10.52:45480 as auto-91C4922B-4324-C369-16F0-F18FB8
9528B9 (p2, c1, k60).
mosquitto  | 1699972977: Client auto-91C4922B-4324-C369-16F0-F18FB89528B9 disconnected.
```

```
links-webots@linkswebots-NUC10i7FNH: ~/youminchu/switchbot-mqtt 107x19
links-webots@linkswebots-NUC10i7FNH:~/youminchu/switchbot-mqtt$ docker compose up switchbot-mqtt
[+] Running 1/0
 ✔ Container switchbot-mqtt  Created                                                            0.0s
Attaching to switchbot-mqtt
switchbot-mqtt  | /switchbot-mqtt/switchbot_mqtt/_cli.py:136: UserWarning: In switchbot-mqtt's next major r
elease, TLS will be enabled by default (--mqtt-enable-tls). Please add --mqtt-disable-tls to your command f
or upward compatibility.
switchbot-mqtt  |   warnings.warn(
switchbot-mqtt  | connecting to MQTT broker localhost:1883 (TLS disabled)
switchbot-mqtt  | subscribing to MQTT topic 'homeassistant/switch/switchbot/+/set'
switchbot-mqtt  | subscribing to MQTT topic 'homeassistant/cover/switchbot-curtain/+/set'
switchbot-mqtt  | subscribing to MQTT topic 'homeassistant/cover/switchbot-curtain/+/position/set-percent'
switchbot-mqtt  | Successfully sent command to Switchbot (MAC: C2:E8:38:2D:03:59)
switchbot-mqtt  | switchbot C2:E8:38:2D:03:59 turned on
```

```
links-webots@linkswebots-NUC10i7FNH: ~ 94x40
links-webots@linkswebots-NUC10i7FNH:~$ mosquitto_pub -h 10.10.10.52 -t homeassistant/switch/sw
itchbot/C2:E8:38:2D:03:59/set -m ON
links-webots@linkswebots-NUC10i7FNH:~$
```

Figure 4.8: Using switchbot-mqtt to turn on a Bot

ROS2 SwitchBot service

The integration of SwitchBot into the ROS2 nav2 package requires the use of a SwitchBot server. This server establishes a client connection with the MQTT broker and sends requests to activate the SwitchBot as required. The service call message must include details of both the robot's current floor and the target floor, corresponding to the actions needed to call the elevator and press the desired floor button.

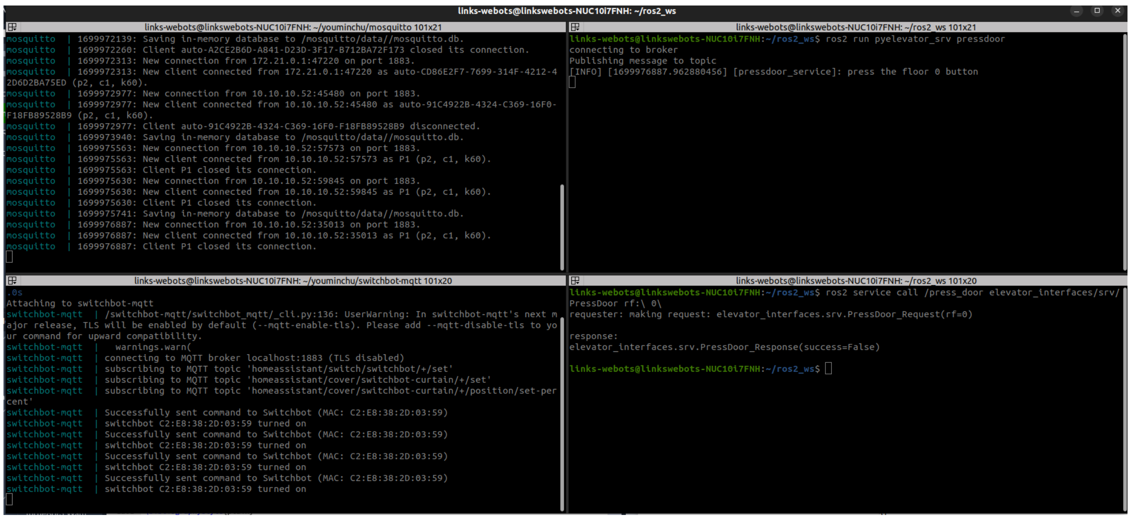
```
int32 rf # represent the floor information
---
bool success   # indicate successful run of open door service
```

Listing 4.4: PressDoor.srv

```python
from elevator_interfaces.srv import PressDoor
import paho.mqtt.client as mqtt
import rclpy

def press_door_callback(request, response):
    global g_node
    floor_info = request.rf
    # broker_address = "10.10.10.52"
    broker_address = "192.168.0.177"
    client = mqtt.Client("P1") #create new instance
    print("connecting to broker")
    client.connect(broker_address) #connect to broker
    print("Publishing message to topic")
    if floor_info == 0:
        g_node.get_logger().info("press the floor 0 button")
        client.publish("homeassistant/switch/switchbot/C2:E8:38:2D:03:59/set","ON"
    )
    elif floor_info == 1:
        g_node.get_logger().info("press the floor 1 button")
        client.publish("homeassistant/switch/switchbot/D1:35:33:35:43:8F/set","ON"
    )
    return response
[...]
```

Listing 4.5: Press door service

Before invoking the service, ensure that the Mosquitto broker and the SwitchBot-mqtt library are launched (Fig. 4.9). If set up correctly, the SwitchBot will activate upon receiving the command (Fig. 4.10).



links-webots@linkswebots-NUC10i7FNH: ~/ros2_ws
links-webots@linkswebots-NUC10i7FNH: ~/youminchu/mosquitto 101x21
links-webots@linkswebots-NUC10i7FNH: ~/ros2_ws 101x21
links-webots@linkswebots-NUC10i7FNH: ~/youminchu/switchbot-mqtt 101x20
links-webots@linkswebots-NUC10i7FNH: ~/ros2_ws 101x20

```
mosquitto  | 1699972139: Saving in-memory database to /mosquitto/data//mosquitto.db.
mosquitto  | 1699972260: Client auto-A2CE2B6D-A841-D23D-3F17-B712BA72F173 closed its connection.
mosquitto  | 1699972313: New connection from 172.21.0.1:47220 on port 1883.
mosquitto  | 1699972313: New client connected from 172.21.0.1:47220 as auto-CD86E2F7-7699-314F-4212-4
2D6D2BA75ED (p2, c1, k60).
mosquitto  | 1699972977: New connection from 10.10.10.52:45480 on port 1883.
mosquitto  | 1699972977: New client connected from 10.10.10.52:45480 as auto-91C4922B-4324-C369-16F0-
F18FB89528B9 (p2, c1, k60).
mosquitto  | 1699972977: Client auto-91C4922B-4324-C369-16F0-F18FB89528B9 disconnected.
mosquitto  | 1699973940: Saving in-memory database to /mosquitto/data//mosquitto.db.
mosquitto  | 1699975563: New connection from 10.10.10.52:57573 on port 1883.
mosquitto  | 1699975563: New client connected from 10.10.10.52:57573 as P1 (p2, c1, k60).
mosquitto  | 1699975563: Client P1 closed its connection.
mosquitto  | 1699975630: New connection from 10.10.10.52:59845 on port 1883.
mosquitto  | 1699975630: New client connected from 10.10.10.52:59845 as P1 (p2, c1, k60).
mosquitto  | 1699975630: Client P1 closed its connection.
mosquitto  | 1699975741: Saving in-memory database to /mosquitto/data//mosquitto.db.
mosquitto  | 1699976887: New connection from 10.10.10.52:35013 on port 1883.
mosquitto  | 1699976887: New client connected from 10.10.10.52:35013 as P1 (p2, c1, k60).
mosquitto  | 1699976887: Client P1 closed its connection.
```

```
links-webots@linkswebots-NUC10i7FNH:~/ros2_ws$ ros2 run pyelevator_srv pressdoor
connecting to broker
Publishing message to topic
[INFO] [1699976887.962880456] [pressdoor_service]: press the floor 0 button
```

```
.0s
Attaching to switchbot-mqtt
switchbot-mqtt  | /switchbot-mqtt/switchbot_mqtt/_cli.py:136: UserWarning: In switchbot-mqtt's next m
ajor release, TLS will be enabled by default (--mqtt-enable-tls). Please add --mqtt-disable-tls to yo
ur command for upward compatibility.
switchbot-mqtt  |   warnings.warn(
switchbot-mqtt  | connecting to MQTT broker localhost:1883 (TLS disabled)
switchbot-mqtt  | subscribing to MQTT topic 'homeassistant/switch/switchbot/+/set'
switchbot-mqtt  | subscribing to MQTT topic 'homeassistant/cover/switchbot-curtain/+/set'
switchbot-mqtt  | subscribing to MQTT topic 'homeassistant/cover/switchbot-curtain/+/position/set-per
cent'
switchbot-mqtt  | Successfully sent command to Switchbot (MAC: C2:E8:38:2D:03:59)
switchbot-mqtt  | switchbot C2:E8:38:2D:03:59 turned on
switchbot-mqtt  | Successfully sent command to Switchbot (MAC: C2:E8:38:2D:03:59)
switchbot-mqtt  | switchbot C2:E8:38:2D:03:59 turned on
switchbot-mqtt  | Successfully sent command to Switchbot (MAC: C2:E8:38:2D:03:59)
switchbot-mqtt  | switchbot C2:E8:38:2D:03:59 turned on
switchbot-mqtt  | Successfully sent command to Switchbot (MAC: C2:E8:38:2D:03:59)
switchbot-mqtt  | switchbot C2:E8:38:2D:03:59 turned on
```

```
links-webots@linkswebots-NUC10i7FNH:~/ros2_ws$ ros2 service call /press_door elevator_interfaces/srv/
PressDoor rf:\ 0\
requester: making request: elevator_interfaces.srv.PressDoor_Request(rf=0)

response:
elevator_interfaces.srv.PressDoor_Response(success=False)

links-webots@linkswebots-NUC10i7FNH:~/ros2_ws$
```

Figure 4.9: Using ROS2 service to turn on a Bot

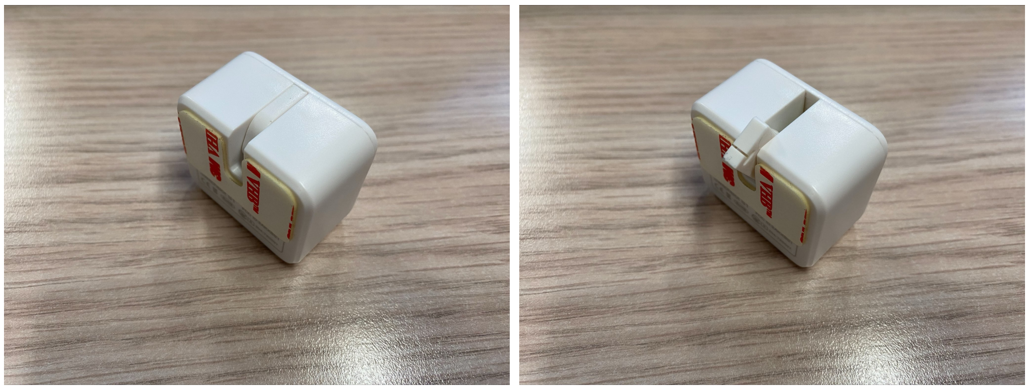


Figure 4.10: The process by which a Bot is activated

To integrate the service into the navigation process, a corresponding behavior tree node must be added. This node replaces the one that interacted with the elevator server in the simulation. The new node is designed to activate the SwitchBot when the robot requires elevator usage. Fig. 4.11 is part of the behavior tree implemented in the real-world test.

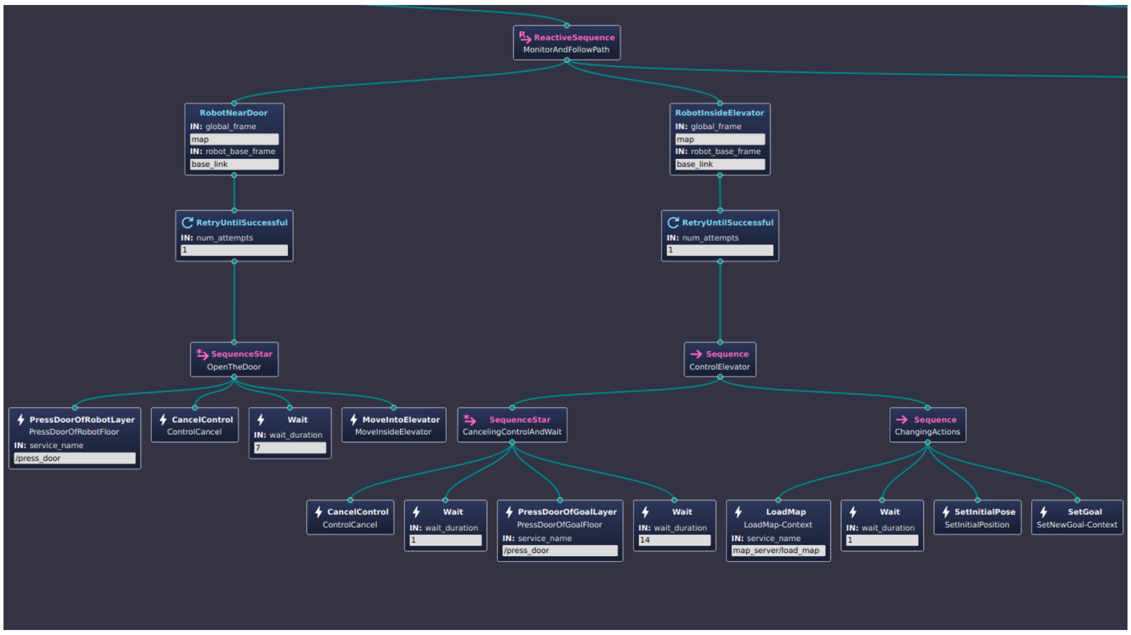


Figure 4.11: The branch that interacts with the elevator in behavior tree navigate_between_real_floors.xml

4.2 Turtlebot3

The TurtleBot3 is a collaborative project that represents the evolution of the TurtleBot series [61]. The original TurtleBot, introduced around 2010, was designed as an affordable, personal robot kit primarily for educational and research purposes. It was designed to be easy to assemble and program. In 2012, TurtleBot2 was released. It was equipped with better sensors, a more robust platform, and was designed to be more user-friendly. TurtleBot 2 continued to support ROS, which allowed users to take advantage of the growing ROS software packages [62]. The TurtleBot3, developed by ROBOTIS in collaboration with Open Robotics - the organization behind ROS, was officially announced in 2017.

The TurtleBot3 is a popular robotic platform, especially within the context of ROS2. It serves as an ideal choice for conducting real-world simulations and experiments in the field of robotics research. The TurtleBot3 is available in different models, like Burger, Waffle, and Waffle Pi, each with varying specifications (Fig. 4.12). Common components to all models include a Raspberry Pi as the main controller, sensors like a 360-degree LIDAR, and motors with encoders for precise movement control.

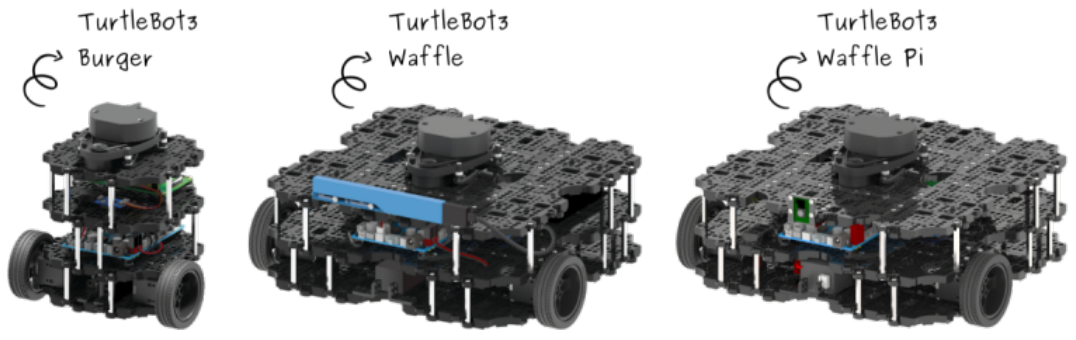


Figure 4.12: TurtleBot3 various models [63]

4.2.1 TurtleBot3 Waffle Pi

The TurtleBot3 Waffle Pi model has been selected to use in this test. Offering greater computing power, the Waffle Pi is capable of handling more complex computational tasks. It is equipped with a comprehensive set of sensors, enhancing its navigation and perception capabilities. In addition to the 360-degree LIDAR (LDS-01), it typically includes a camera, an inertial measurement unit (IMU), and other sensors that enable sophisticated operations. This makes the Waffle Pi an excellent choice for advanced robotics projects. The primary components are presented in Fig. 4.13 and some important parameters of the TurtleBot3 Waffle Pi are displayed in Table 4.1

Sensors play an important role in the functionality of the TurtleBot3, particularly in its navigation and mapping capabilities. They emit laser beams to measure distances to objects in the surrounding environment. By rotating and scanning, these sensors construct a 2D map that represents the distances to various obstacles and features. The

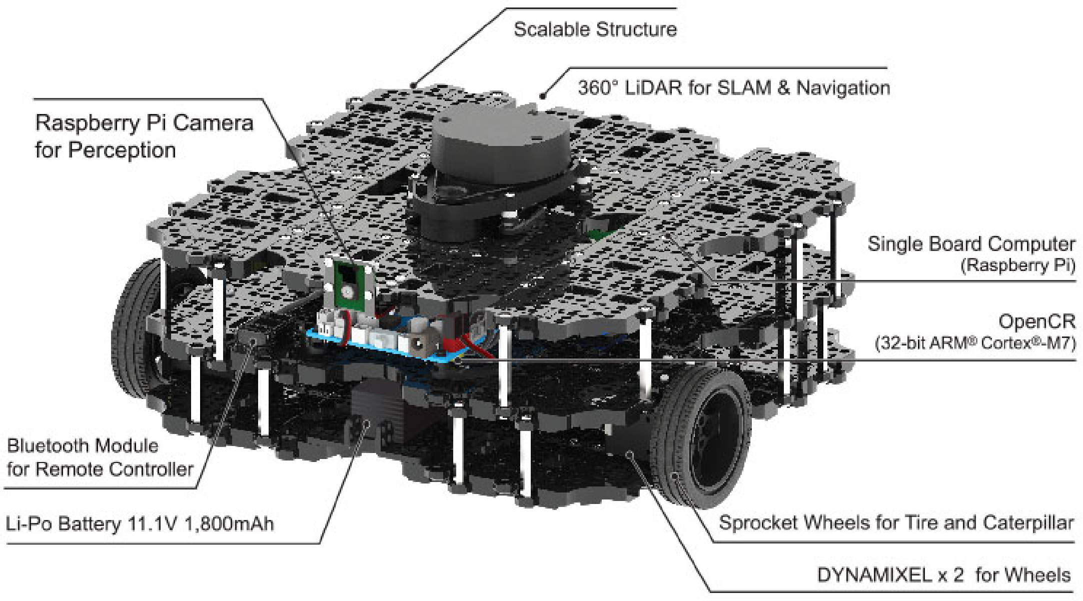

Scalable Structure
360° LiDAR for SLAM & Navigation
Raspberry Pi Camera for Perception
Single Board Computer (Raspberry Pi)
OpenCR (32-bit ARM® Cortex®-M7)
Bluetooth Module for Remote Controller
Li-Po Battery 11.1V 1,800mAh
Sprocket Wheels for Tire and Caterpillar
DYNAMIXEL x 2 for Wheels

Figure 4.13: TurtleBot3 Waffle Pi main components [64]

Items	Waffle Pi
Maximum Translational Velocity	0.26m/s
Maximum Rotational Velocity	1.82rad/s (104.27deg/s)
Maximum Payload	30kgs
Size (L*W*H)	281mm*306mm*141mm
Sensor	Raspberry Pi Camera, 360-degree LIDAR

Table 4.1: TurtleBot3 Waffle Pi main specifications

data gathered is crucial for SLAM algorithms, providing essential information for both mapping the environment and determining the robot's location within it.

For the test, a modified version of the TurtleBot3 was used, as shown in Fig. 4.14. The primary modification lies in the sensor: the test's Waffle Pi model is equipped with an RPLIDAR A3M1 sensor, replacing the standard LDS-01 sensor. The RPLIDAR A3M1 offers advantages such as a longer detection range and higher resolution [65], enabling it to detect objects at greater distances and provide more detailed environmental data. Additionally, its higher scanning rate and frequency contribute to more precise and comprehensive environmental mapping. Table 4.2 compares these two sensors.

4.3 Test

The successful execution of the entire navigation process in a real-world environment relies on maintaining a stable connection between the TurtleBot and a controlling computer.

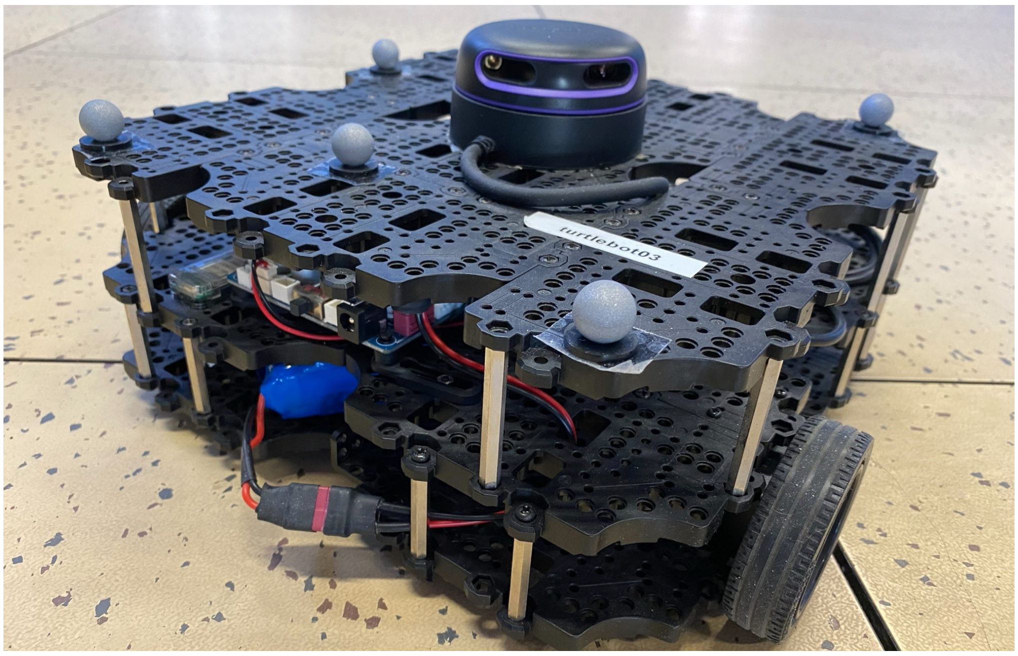


Figure 4.14: TurtleBot3 Waffle Pi used in test

Feature	RPLIDAR A3M1	LDS-01
Range	Up to 25 meters	Up to 3.5 meters
Sample Rate	16000-64000 samples per second	2000 samples per second
Angular Resolution	0.225°to 0.9°	1°
Scanning Frequency	10-20 Hz	5-10Hz

Table 4.2: Comparison chart for RPLIDAR A3M1 and LDS-01

For this purpose, a laptop and a mobile hotspot are utilized. The laptop is responsible for executing the navigation process and sending commands to activate the SwitchBot. This connection between the SwitchBot and the laptop is established using the SSH (Secure Shell) protocol.

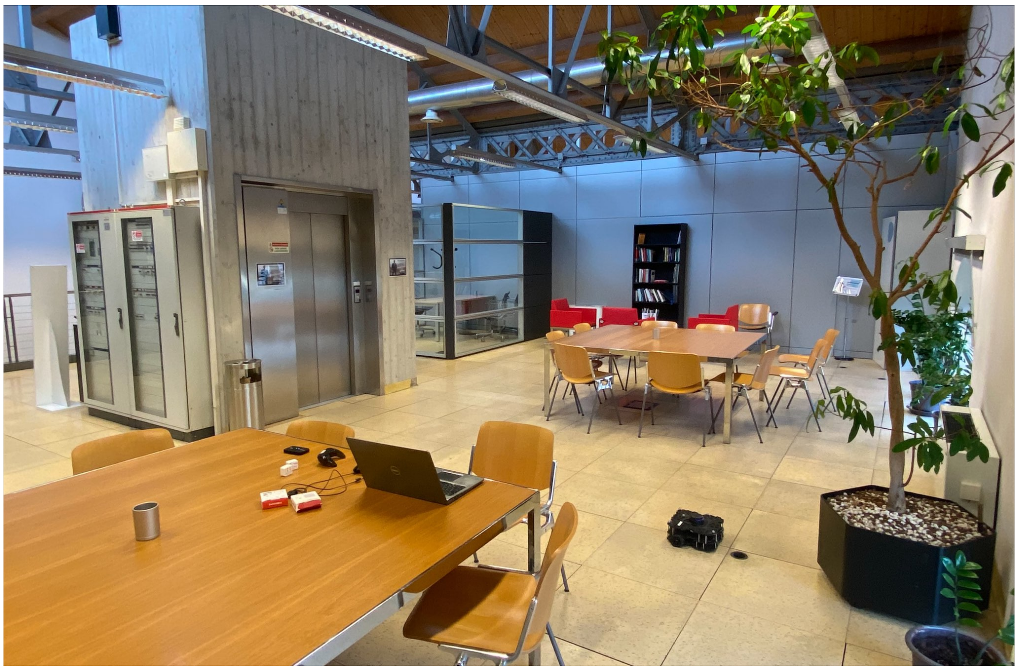
4.3.1 Map creating

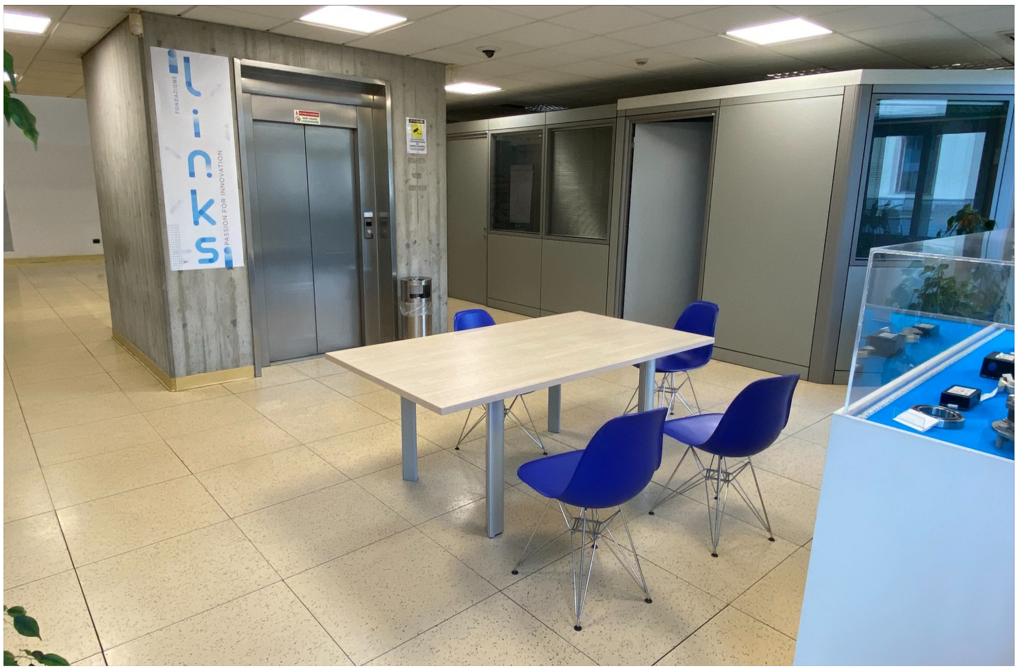
The chosen environment for this test is the two-floor leisure area in LINKS (Fig. 4.17). These floors are connected by an elevator.

The initial step before the navigation involves creating the map and updating the elevator's position within it. Throughout this process, the SLAM Toolbox is utilized. Establish a connection with the TurtleBot and execute the commands to launch Rviz and the toolbox. Subsequently, control the robot to explore different areas.

```
ros2 launch nav2_bringup navigation_launch.py
```

Listing 4.6: Launch Navigation2


(a) Floor 1



(b) Floor 0

Figure 4.15: LINKS office

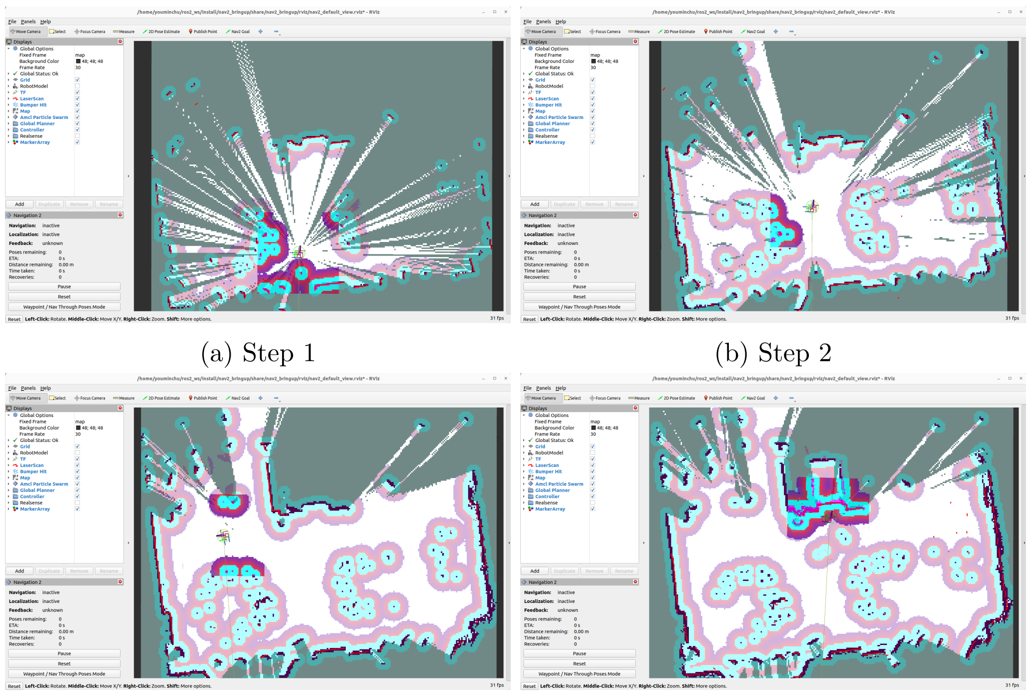
```
ros2 launch slam_toolbox online_async_launch.py
```

Listing 4.7: Launch SLAM Toolbox

```
ros2 run rviz2 rviz2 -d $(ros2 pkg prefix nav2_bringup)/share/nav2_bringup/rviz/
    nav2_default_view.rviz
```

Listing 4.8: Launch Rviz

Fig. 4.16 illustrates the process of mapping floor 1. As the TurtleBot navigates, the unknown areas gradually decrease until the entire floor is mapped, including the interior of the elevator.



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 4

Figure 4.16: Floor1 map creation

Before utilizing the maps for navigation, two additional steps are required. The first step involves verifying whether the elevator door is marked as a permanent obstacle on the map. The map obtained from the map server is shown in Fig. 4.17a. In this figure, the elevator door is represented as a black line, indicating an impassable barrier for the robot, even when the door is open. A feasible solution to this issue is to use a photo editor to remove this line. A comparison of the maps, before and after editing, is displayed in Fig. 4.17b.

The second step involves adjusting the position of the map frame. When the robot transits to a new floor and the map is changed, its coordinates remain unchanged. For instance, if the robot's coordinates in the elevator on floor 0 are (1.5, 2.5), these coordinates will persist even after switching to the floor 1 map. However, this location on floor

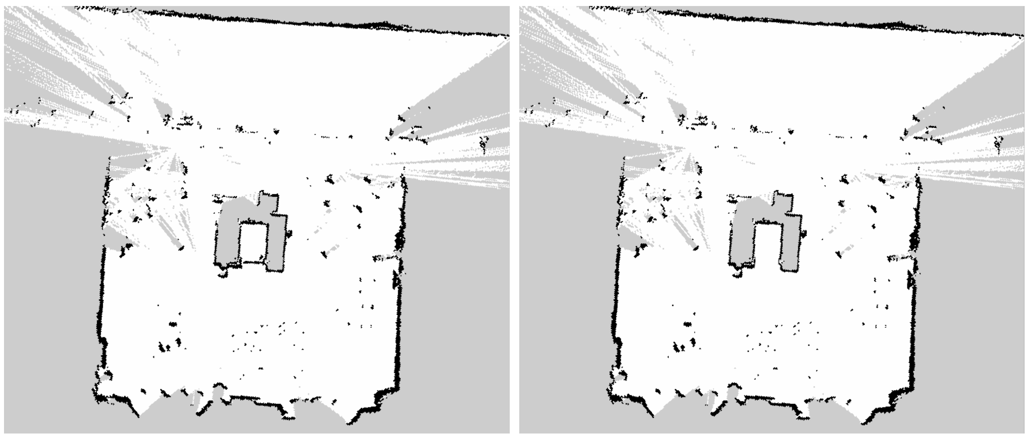

(a) Before editing the door (b) After editing the door

Figure 4.17: Floor 0 map comparison

1 might not correspond to the elevator but to a different area. Therefore, it is necessary to adjust the map frame's position to ensure the robot is accurately placed within the correct area after the map changes.

This adjustment can be made by modifying the `YAML` file associated with the map. The 'origin' parameter in this file, specified as [x, y, yaw], determines the coordinates of the map's bottom-left corner and its orientation. Appropriately defining these parameters is crucial for accurately positioning the map frame. Fig. 4.18 presents the map utilized in the navigation.

```
[...]
# origin: [-5.37, -11.2, 0] # before
origin: [-10.37, -7.2, 0] # after
```

Listing 4.9: floor1_open.yaml

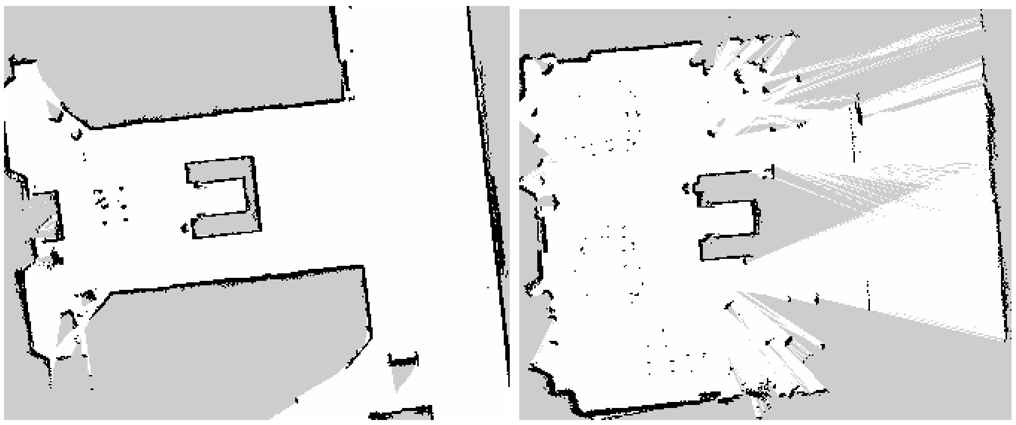
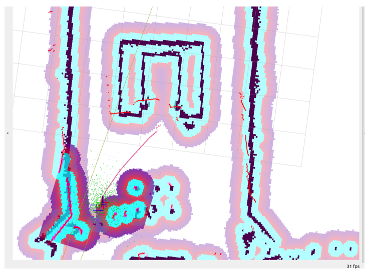

(a) Map of floor 0 (b) Map of floor 1

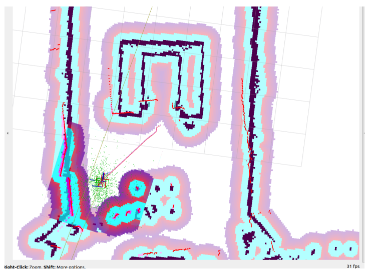
Figure 4.18: LINKS maps used in navigation

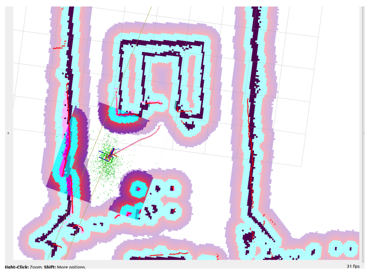
4.3.2 Real environment test

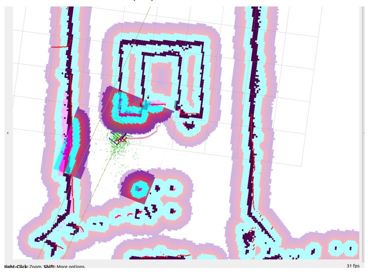
By default, ROS2 Humble employs eProsima's Fast DDS as its middleware. However, for enhanced performance and more stable connectivity, Eclipse Cyclone DDS has been selected. Cyclone DDS is designed to be both simple and lightweight, which makes it an ideal choice for applications with limited resources or where simplicity is a priority. It is known for its good performance, especially in terms of latency and throughput, making it suitable for real-time applications.

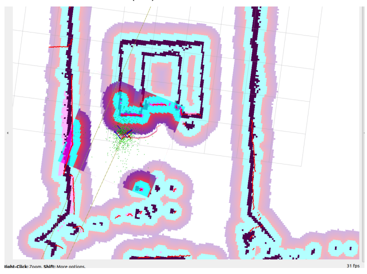
Another difference from the simulation relates to the selection of the controller. In Nav2, the default controller is the DWB controller. As a local planner, the DWB controller is responsible for making immediate, short-term decisions to navigate the robot along a global path while avoiding dynamic obstacles. It calculates the robot's velocities by taking into account its current velocity, acceleration limits, and the destination. These velocities are optimized to avoid obstacles while efficiently navigating towards the goal. The process of using DWB controller to navigate to the elevator on floor 0 is displayed in Fig. 4.19.

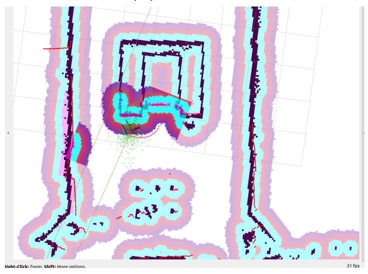

(a) Step 1

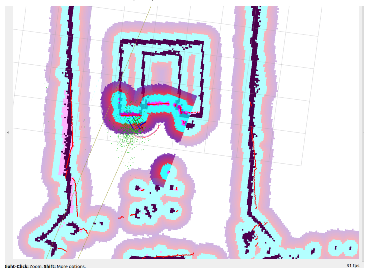

(b) Step 2

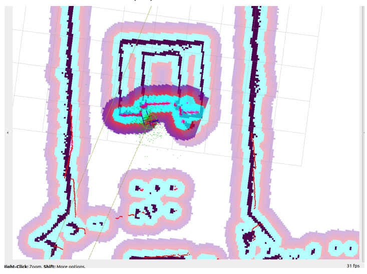

(c) Step 3


(d) Step 4


(e) Step 5


(f) Step 6


(g) Step 7


(h) Step 8

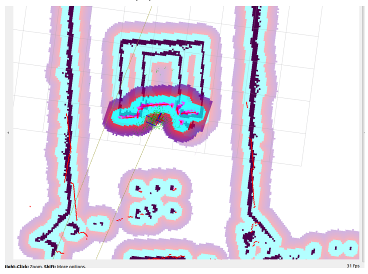

(i) Step 9

Figure 4.19: Navigation using DWB controller

The MPPI Controller generates multiple trajectories based on a random process, evaluates them, and selects the best trajectory considering both the current state and future predictions. MPPI can be used for both local and global planning, offering a more integrated approach to path planning and obstacle avoidance. The process of using the MPPI controller to navigate to the elevator in floor 0 is displayed in Fig. 4.20.

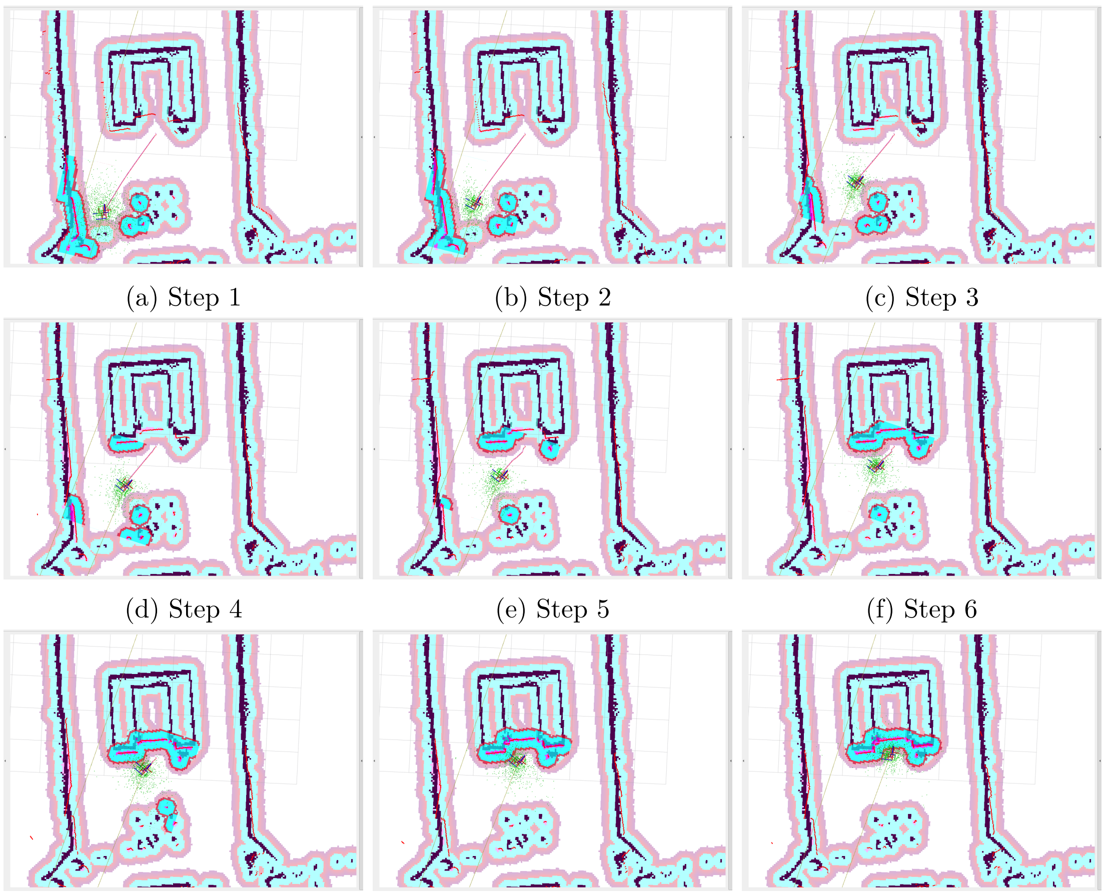

(a) Step 1 (b) Step 2 (c) Step 3
(d) Step 4 (e) Step 5 (f) Step 6
(g) Step 7 (h) Step 8 (i) Step 9

Figure 4.20: Navigation using MPPI controller

It can be observed that the path from the robot's position to the goal is mostly a direct line. During the navigation task, the MPPI controller directed the robot along the shortest, most direct route to the goal, in contrast to the DWB controller, which took a more complex path to achieve the same objective. The observed behavior of the DWB controller suggests a preference for certain velocities or directions, possibly due to its design focusing on local velocity planning and obstacle avoidance. This result might indicate a limitation or a configuration issue in the DWB approach when navigating in specific environments. Conversely, the MPPI controller's effective and direct approach can be attributed to its model predictive capabilities, evaluating and selecting optimal trajectories based on future state estimations. This feature allows it to navigate efficiently and directly towards the goal in clear environments. Therefore, the MPPI controller has been selected to use in the test.

The test began on floor 1 and proceeded to floor 0. After the initial pose of the TurtleBot was set, the command detailed in the Listing 4.10 was used to publish the goal. The entire navigation process could be divided into several distinct stages.

```
ros2 topic pub /goal_pose geometry_msgs/PoseStamped "{header: {stamp: {sec: 0},
    frame_id: 'map'}, pose: {position: {x: -5.0, y: -2.0, z: 0.0}, orientation:
    {}}}" -1
```

Listing 4.10: Publish a goal

- The TurtleBot navigated from its initial pose on floor 1 to the elevator, where the first SwitchBot interacted with the elevator system (Fig. 4.21).

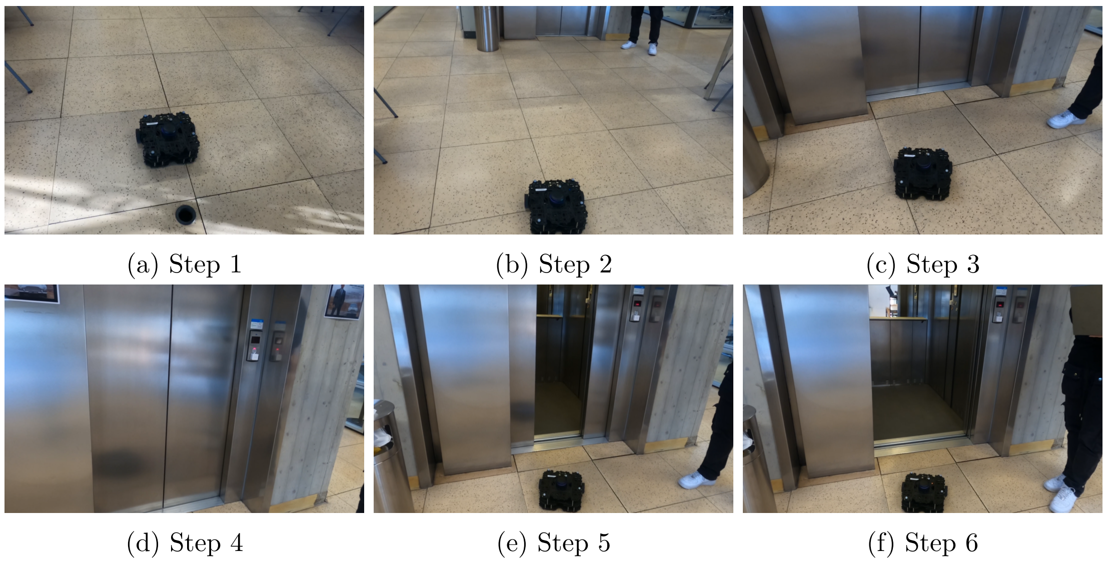
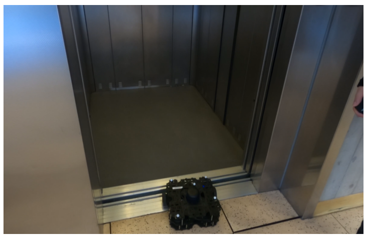
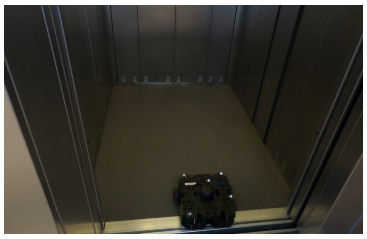
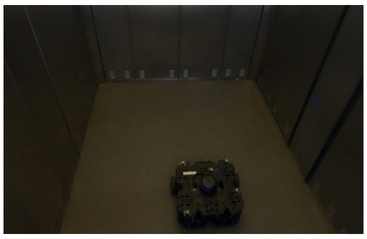

(a) Step 1 (b) Step 2 (c) Step 3
(d) Step 4 (e) Step 5 (f) Step 6

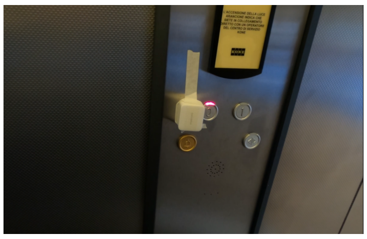
Figure 4.21: Stage 1 of the multi-floor navigation process

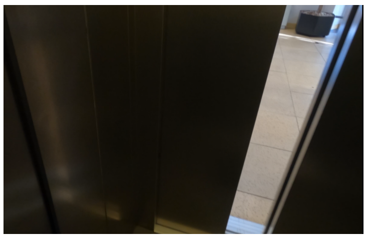
- The TurtleBot navigated inside the elevator, and the second SwitchBot pressed the button for the target floor (Fig. 4.22).
- The elevator went down to floor 0, and the TurtleBot navigated to the final goal (Fig. 4.23).


(a) Step 1


(b) Step 2


(c) Step 3


(d) Step 4


(e) Step 5

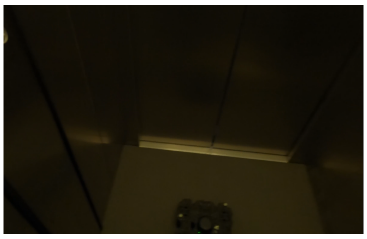
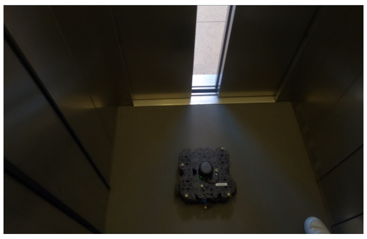
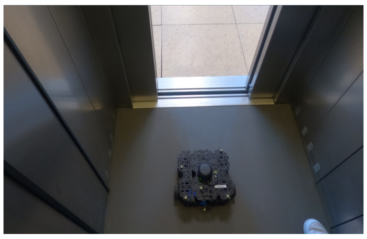
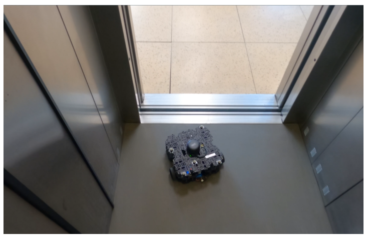
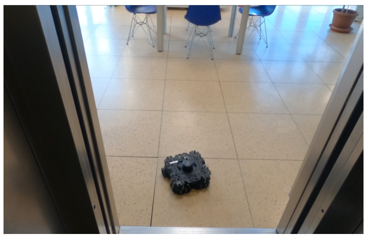

(f) Step 6

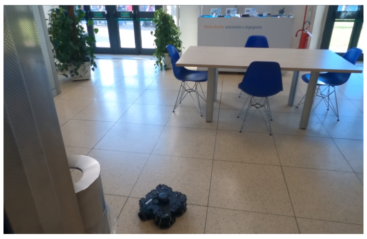
Figure 4.22: Stage 2 of the multi-floor navigation process


(a) Step 1


(b) Step 2


(c) Step 3


(d) Step 4


(e) Step 5

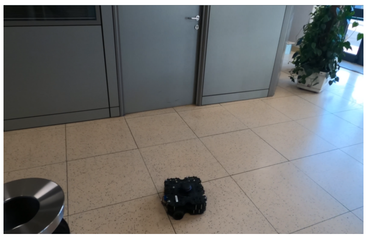

(f) Step 6

Figure 4.23: Stage 3 of the multi-floor navigation process

Chapter 5

Conclusion

The concept of multifloor navigation in robotics has evolved significantly over the years. Initially, robotic navigation was largely limited to single-level environments, focusing on challenges like obstacle avoidance and path optimization. As technological advancements in robotics and software development progressed, the need for more complex navigation capabilities, including multifloor navigation, became apparent. This thesis contributes to this evolving field by implementing and testing a multifloor navigation system in ROS2 using behavior trees. This approach not only addresses the complicated challenges of navigating between different floors but also presents the adaptability and efficiency of behavior trees in complex robotic tasks.

In the simulation part, a new navigator and a custom behavior tree are created. The navigator plays an important role in analysing the goal information and sending the goal to the behavior tree. It integrated with the behavior tree, ensuring an efficient navigation process. This tree is designed to handle the complex process required for multifloor navigation. The behavior tree's architecture allows for a dynamic and flexible response to varying environmental conditions. A critical component in the simulation is the incorporation of an elevator server. This server is responsible for controlling the simulated elevator's movements to transition the robot between floors.

Then the theoretical concepts and software developed in the simulation are applied to the real world test. A Turtlebot is selected as the model for its reliability and adaptability in various environments. To interact with the elevator in the real world, a SwitchBot is employed. This solution allows the robot to physically interact with the elevator buttons, a necessary function for transitioning between floors. This real world test is crucial in validating the effectiveness of the custom behavior tree and the new navigator developed in the simulation, proving their applicability beyond the theory.

As a component of LINKS projects, this thesis serves as a foundational basis for potential future works in various directions.

In the simulation, the local controller encounters challenges when the robot traverses to the first floor or higher. One possible explanation for this issue is that the robot's z-coordinate exceeds the local controller's threshold when the robot is transported to a higher floor via the elevator. A solution involves utilizing a TF broadcaster to maintain a consistent z-coordinate. Additionally, enhancing the elevator server could be beneficial.

Incorporating a function that communicates the current floor information to the robot would enable it to decide whether it has reached the correct floor.

In the real world test at the LINKS office, which includes only two floors, the initial plan involved positioning one SwitchBot on floor 0 and another on floor 1. The idea was to activate the SwitchBot on the current floor when the robot needed to use the elevator, and then, after the robot entered the elevator, to activate the SwitchBot on floor 1 instead of pressing the button for floor 1. However, this approach encountered a problem due to the limited range of Bluetooth connectivity, which could not support the connection between floors. Moreover, in environments with more than two floors, this method would be impractical with a traditional elevator system. To address this, developing a smart elevator system that could be integrated into the Nav2 package might be a solution.

Bibliography

[1] A. Pandey, S. Pandey, and DR Parhi. "mobile robot navigation and obstacle avoidance techniques: A review". *International Robotics Automation Journal*, 2, 2017.

[2] R. Crespo, J.C. Castillo, O.M. Mozos, and R. Barber. "semantic information for robot navigation: A survey". *Applied Sciences*, 10(2), 2020.

[3] SRI Internatiional. Shakey the Robot. https://www.sri.com/hoi/shakey-the-robot/. [Online; accessed April 2023].

[4] H. Choset, K.M. Lynch, S. Hutchinson, G.A. Kantor, W. Burgard, L.E. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. The MIT Press, 2 edition, 2005.

[5] J. Sola. "simulataneous localization and mapping with the extended kalman filter: A very quick guide". 2014.

[6] X. Lei, B. Feng, G. Wang, W. Liu, and Y. Yang. A novel fastslam framework based on 2d lidar for autonomous mobile robot. *Electronics*, 9(4):695, April 2020.

[7] T. Raj, F.H. Hashim, A.B. Huddin, M.F. Ibrahim, and A. Hussain. A survey on lidar scanning mechanisms. *Electronics*, 9(5), 2020.

[8] G. Grisetti, R. Kummerle, C. Stachniss, and W. Burgard. "a tutorial on graph-based slam". *IEEE Intelligent Transportation Systems Magazine*, 2(4):31–43, winter 2010.

[9] K. Konolige, E. Marder-Eppstein, and B. Marthi. "navigation in hybrid metric-topological maps". *2011 IEEE International Conference on Robotics and Automation*, 2011.

[10] H. Wang, Y. Yu, and Q. Yuan. "application of dijkstra algorithm in robot path-planning". *2011 Second International Conference on Mechanic Automation and Control Engineering*, 2011.

[11] S. Sedighi, D. Nguyen, and K. Kuhnert. "guided hybrid a-star path planning algorithm for valet parking applications". *2019 5th International Conference on Control Automation and Robotics (ICCAR)*, pages 570–575, 2019.

[12] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1052–1067, 2007.

[13] N. K. Dhiman, D. Deodhare, and D. Khemani. "a ros based framework for multi-floor navigation for unmanned ground robots". *AIR 2019: Proceedings of the Advances in Robotics 2019*, (44):1–6, July 2019.

[14] J. Huang, T. Lau, and M. Cakmak. Design and evaluation of a rapid programming system for service robots. In *2016 11th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 295–302, 2016.

[15] R. Bloss. Mobile hospital robots cure numerous logistic needs. *Industrial Robot*, 38(6):567–571, 2011.

[16] Steve Cousins. 5 ways relay autonomous delivery robots are so cost-effective. `https://www.relayrobotics.com/blog/2019/11/6/5-ways-relay-autonomous-delivery-robots-are-so-cost-effective`. [Online; accessed April 2023].

[17] Tug autonomous mobile robots for healthcare and hospitality. `https://aethon.com/products/`. [Online; accessed April 2023].

[18] Ros1 vs ros2, practical overview for ros developers. `https://roboticsbackend.com/ros1-vs-ros2-practical-overview/`. [Online; accessed April 2023].

[19] J. Kay and A.R. Tsouroukdissian. "real-time control in ros and ros 2.0". *ROSCon15, 2015*, 2015.

[20] Y. Liu, Y. Guan, X. Li, R. Wang, and J. Zhang. "formal analysis and verification of dds in ros2". *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2018.

[21] V. Mayoral-Vilches, R. White, G. Caiazza, and M. Arguedas. "sros2: Usable cyber security tools for ros 2". *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, December 2022.

[22] Ros teaching series (3) - the difference between ros 1 and ros 2. https://www.circuspi.com/index.php/2022/09/26/ros1-ros2-difference/. [Online; accessed April 2023].

[23] Understanding nodes. https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html. [Online; accessed April 2023].

[24] Understanding topics. https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html. [Online; accessed April 2023].

[25] Understanding services. https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html. [Online; accessed April 2023].

[26] Understanding actions. https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html. [Online; accessed April 2023].

[27] Navigation concepts. https://navigation.ros.org/concepts/index.html#navigation-servers. [Online; accessed April 2023].

[28] Nav2 nav2 1.0.0 documentation. https://navigation.ros.org/. [Online; accessed April 2023].

[29] Tully Foote. tf: The transform library. In *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6, 2013.

[30] N. Koenig and A. Howard. "design and use paradigms for gazebo, an open-source multi-robot simulator". *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, February 2005.

[31] H.R. Kam, SH. Lee, and T. Park. "rviz: a toolkit for real domain data visualization.". *Telecommun Syst 60*, pages 337–345, 2015.

[32] M. Colledanchise and P. Ogren. *Behavior trees in robotics and AI: An introduction.* CRC Press, 2018.

[33] Y. Fu, L. Qin, and Q. Yin. A reinforcement learning behavior tree framework for game ai. In *Proceedings of the 2016 International Conference on Economics, Social Science, Arts, Education and Management Engineering*, pages 573–579. Atlantis Press, 2016/08.

[34] State machines vs behavior trees: designing a decision-making architecture for robotics. https://www.polymathrobotics.com/blog/state-machines-vs-behavior-trees. [Online; accessed May 2023].

[35] Halo 2 - wikipedia. https://en.wikipedia.org/wiki/Halo_2. [Online; accessed May 2023].

[36] M. Nicolau, D. Perez-Liebana, M. OâNeill, and A. Brabazon. "evolutionary behavior tree approaches for navigating platform games". *IEEE Transactions on Computational Intelligence and AI in Games*, 9:227–238, 2017.

[37] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, 2(1):14–23, 1986.

[38] P. Ogren. Increasing modularity of uav control systems using computer game behavior trees. *AIAA Guidance Navigation and Control Conference*, pages 13–16, 2012.

[39] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ogren. Towards a unified behavior trees framework for robot control. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5420–5427, 2014.

[40] Behavior tree - ros wiki. https://wiki.ros.org/behavior_tree. [Online; accessed May 2023].

[41] K. French, S. Wu, T. Pan, Z. Zhou, and O.C. Jenkins. Learning behavior trees from demonstration. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 7791–7797, 2019.

[42] Nodes library: Sequences. https://www.behaviortree.dev/docs/3.8/nodes-library/SequenceNode. [Online; accessed May 2023].

[43] Nodes library: Fallbacks. https://www.behaviortree.dev/docs/3.8/nodes-library/FallbackNode. [Online; accessed May 2023].

[44] Nodes library: Decorators. https://www.behaviortree.dev/docs/3.8/nodes-library/DecoratorNode. [Online; accessed May 2023].

[45] A.J. Champandard and P. Dunstan. *Game AI Pro 360: Guide to Architecture: The Behavior Tree Starter Kit*. CRC Press, 1st edition edition, 2019.

[46] About | behaviortree.cpp. https://www.behaviortree.dev/docs/3.8/intro. [Online; accessed May 2023].

[47] Groot2. https://www.behaviortree.dev/groot. [Online; accessed May 2023].

[48] R. Ghzouli, T. Berger, E.B. Johnsen, A. Wasowski, and S. Dragule. "behavior trees and state machines in robotics applications". *IEEE Transactions on Software Engineering*, 49:4243–4267, September 2023.

[49] Rasmus V. Rasmussen and Michael A. Trick. Round robin scheduling â a survey. *European Journal of Operational Research*, 188(3):617–636, 2008.

[50] Matti Kortelainen. A short guide to ros 2 humble hawksbill. 2023.

[51] Switchbot - your simple switch to a smart home. https://eu.switch-bot.com/. [Online; accessed September 2023].

[52] Switchbot curtain 3 | automatic curtain opener, supports matter. https://eu.switch-bot.com/products/switchbot-curtain-3. [Online; accessed September 2023].

[53] Switchbot electric blind tilt | make existing blinds smart. https://eu.switch-bot.com/products/switchbot-blind-tilt. [Online; accessed September 2023].

[54] T.V. Tran, H. Takahashi, T. Narabayashi, and H. Kikura. An application of iot for conduct of laboratory experiment from home. In *2020 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*, pages 1–4, 2020.

[55] Switchbot bot | smart button pusher. https://eu.switch-bot.com/products/switchbot-bot. [Online; accessed September 2023].

[56] D. Soni and A. Makwana. A survey on mqtt: a protocol of internet of things (iot). In *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*, volume 20, pages 173–177, 2017.

[57] B. Mishra. Performance evaluation of mqtt broker servers. In *International Conference on Computational Science and Its Applications*, pages 599–609. Springer, 2018.

[58] What is mqtt? definition and details. https://www.paessler.com/it-explained/mqtt. [Online; accessed September 2023].

[59] M. Bender, E. Kirdan, M.O. Pahl, and G. Carle. Open-source mqtt evaluation. In *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–4. IEEE, 2021.

[60] switchbot-mqtt 3.3.1. https://pypi.org/project/switchbot-mqtt/. [Online; accessed September 2023].

[61] R. Amsters and P. Slaets. Turtlebot 3 as a robotics education platform. In *Robotics in Education: Current Research and Innovations 10*, pages 170–181. Springer, 2020.

[62] A. Koubaa, M.F. Sriti, Y. Javed, M. Alajlan, B. Qureshi, F. Ellouze, and A. Mahmoud. Turtlebot at office: A service-oriented software architecture for personal assistant robots using ros. In *2016 International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 270–276, 2016.

[63] Turtlebot3. https://www.turtlebot.com/turtlebot3/. [Online; accessed September 2023].

[64] Turtlebot3 waffle pi | ros components. https://www.roscomponents.com/en/mobile-robots/turtlebot-3-waffle. [Online; accessed September 2023].

[65] M. Bouazizi, C. Ye, and T. Ohtsuki. Activity detection using 2d lidar for healthcare and monitoring. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 01–06. IEEE, 2021.

Acknowledgements

I'd like to start by thanking Professor Marina Indri for all the help and advice on my thesis. The guidance has been a huge part of this project, and I really appreciate it.

Then a big thank you to Dott. Gianluca Prato and Dott. Francesco Aglieco for their help with decision-making and coding. Their expertise has been incredibly helpful and has really helped me get through some tough parts of my work.

I also want to thank Assistant Professor Pangcheng David Cen Cheng for the help in polishing up the writing in my thesis, which has made a big difference.

A special Thanks to the LINKS office for providing a great place to work and all the equipment that needed for the tests.

To my parents, thanks for the love and support. The encouragement have been my power throughout this journey.

To Olivia Zhang, thank you for your patience, love, and understanding. Your support and encouragement have meant a lot.

And to my friends, thank you for all the laughs, the chats, and for always being there. You guys have been a source of strength and joy.

To everyone who has supported me, whether I've named you here or not, thank you from the bottom of my heart. Your support has been key in getting this thesis across the finish line.