Thesis Dissertation

SAFE AUTONOMOUS ROBOT NAVIGATION

Lampros Dionysiou

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2024

Πανεπιστήμιο Κύπρου
University of Cyprus

CYENS
CENTRE OF EXCELLENCE

Safe Autonomous Robot Navigation

Lampros Dionysiou

Advisor: Dr. Vasilis Vasiliades
Supervisor: Prof. Chris Christodoulou

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2024

Abstract

This thesis explores the implementation of safe autonomous navigation for robots within an indoor environment. Using TurtleBot 4, equipped with LiDAR and depth cameras, the study investigates the application of Simultaneous Localization and Mapping (SLAM), object detection algorithms and custom behavior trees to enhance robot navigation. The goal is to enable the robot to move autonomously, avoid unmovable obstacles, and navigate efficiently by moving lightweight objects when necessary. Through a series of experiments in a controlled environment, the effectiveness of the proposed navigation system is evaluated, demonstrating its potential to improve hospital logistics and operational efficiency.

Acknowledgements

I would like to thank my supervisors Prof. Chris Christodoulou and Dr. Vassilis Vassiliades for their consistent support and guidance during the running of this project. I would also like to thank Pieris Panayi for his advice and help during the project.

Contents

List of Figures

Listings

Chapter 1

Introduction

Most hospitals rely on human resources to move supplies, medicine, and food inside their facilities. The changing characteristics of the population and its rapid growth impose the need to increase hospitals' capacity, which leads to less efficiency in human resources[1]. From an economic viewpoint, the transfer of goods, linens, biological samples, medical equipment, pharmaceuticals, mail parcels, and medical waste, wastes more than 850 man-hours per week in a 500-bed hospital[2]. Therefore, autonomous robots could help with both clinical and non-clinical tasks, thus increasing the efficiency of hospitals and decreasing their overhead costs. However, hospitals are an unpredictable setting, unlike huge industrial environments where robots are often deployed.

The highly dynamic nature of hospital settings presents specific challenges to autonomous robots[3]. As opposed to the static and highly structured setting of industrial scenarios, the hospital environment possesses an intrinsic tendency to be a state of constant change, typically in an uncontrolled fashion. For example, medical staff, patients, and visitors are all highly mobile, and the position of equipment and supplies may change dramatically through the course of a day. In such a dynamic environment, the capability of a robot to adapt to changes in dynamic situations such that it can navigate around or move obstacles can be crucial[4]. This will not just ensure continued operation but also will lessen the dependence on human intervention and hence increase overall efficiency.

In emergency situations, the limitations of robots that "play it safe" by strictly avoiding obstacles become particularly evident. In cases of life and death, the robot's actual capacity to have the ability to push things aside and clear out the path can be lifesaving. For instance, a delivery robot tasked with delivering crucial medical items must move from point A to B in the least amount of time, effectively and sometimes that means moving chairs or other small

objects blocking it. Such conduct can drastically enhance response times and guarantee the delivery of adequate supplies to medical staff within a short span.

The benefits to be derived from the deployment of autonomous robots in hospitals go beyond efficiency and responsiveness to emergencies. Autonomous robots can do routine tasks, thus, it will enable healthcare providers to concentrate on patients. That implies that when robots do most of the work normally done by human beings, care levels will rise while reducing pressure among hospital employees through improving job satisfaction and minimizing instances of burnout.

In addition to this, having an autonomous robot has a significant impact on economics. Hospitals can carry out repetitive and mundane tasks without requiring human labor; this alone would be a significant drop in their operational costs. This cost-saving is especially significant in large hospitals where the volume of goods and supplies that need to be transported is high. Using robots in conjunction with other forms of automation will not only facilitate a more smooth and effective logistics system but it can be the key to improve the resource allocation as well as financial performance for healthcare institutions[2].

In summary, the integration of autonomous robots in hospital settings addresses the dual challenges of operational efficiency and adaptability in dynamic environments. Hospitals can improve the delivery of their service, significantly enhancing emergency response and saving a lot of money by giving robots the ability to get around more efficiently and interact with objects in their environments. To accomplish this goal, this thesis investigates the use of complex navigation strategies through behavior trees in order to improve robot autonomous navigation capabilities tailored for hospital environments.

1.1 Objectives and goals

In this thesis, I am going to train a robot (TurtleBot 4) that will be able to move autonomously inside a hospital. The robots must use SLAM to map the area of the hospital and object tracking algorithms to identify objects in their way. They should avoid collisions with unmovable objects or patients, but I will train the robots to move objects that are in the robot's path and are lightweight. The goal is to ensure the robots navigate efficiently in diverse dynamic scenarios.

1.1.1 Objectives

The general aim of this thesis is to make mobile robots more agile in dynamic indoor environments. This involves designing and implementing a robust navigation system using behavior trees to allow the robot to move around efficiently by interacting with and moving lightweight obstacles. The specific objectives include:

1. **Develop an Advanced Navigation System:** Implement a navigation system using behavior trees that can autonomously guide a robot through complex and dynamic environments.

2. **Integrate SLAM for Real-Time Mapping:** Utilize Simultaneous Localization and Mapping (SLAM) techniques to create and update maps of the environment in real-time.

3. **Implement Object Detection Algorithms:** Employ object detection algorithms, such as YOLO, to identify and classify objects and obstacles within the robot's path.

4. **Enhance Obstacle Interaction:** Develop mechanisms for the robot to safely move lightweight, non-fixed obstacles that block its path, ensuring continuous navigation.

5. **Evaluate Navigation Performance:** Conduct a series of experiments to assess the efficiency, reliability, and safety of the developed navigation system in various scenarios.

1.1.2 Goals

The broader goals of this research are to contribute to the field of autonomous robotics by demonstrating the practical application of advanced navigation strategies to improve robot maneuvering capabilities. The custom navigator developed should produce better results than the default navigator. The goals include:

1. **Improve Autonomous Navigation Efficiency:** Demonstrate how behavior trees can enhance the efficiency of autonomous navigation in complex and dynamic environments like hospitals. It is important to show that after customization the robot can adapt to the difficult circumstances in hospitals.

2. **Enhance Real-Time Decision Making:** Show how robots can make real-time decisions to interact with and move obstacles, ensuring smooth and continuous navigation in contrast with the 'play-it-safe' mechanism of the custom navigator.

3. **Increase Navigation Reliability and Safety:** Illustrate the potential improvements in reliability and safety of autonomous robots through the use of advanced object detection and navigation algorithms. Instead of applying the same strategy for all obstacles, it is better to customize it based on the type of obstacles (for example human or not). This way the navigation will be safer for patients in healthcare environments.

4. **Advance Research in Autonomous Robotics:** Contribute to the academic and practical knowledge base in the field of autonomous robotics, particularly in the use of behavior trees and SLAM for dynamic environments.

1.2 Methods of Investigation / Implementation

To achieve the goals mentioned above, I followed a structured approach explained below:

1. **Comprehensive Literature Review:** Investigate existing research on SLAM, object detection algorithms, and behavior trees to establish a knowledge base and identify potential improvements.

2. **ROS and Simulation Familiarization:** Set up and familiarize with the Robot Operating System (ROS) and simulation environments like Gazebo to provide a robust platform for development and testing.

3. **SLAM Techniques Integration:** Implement and compare various SLAM algorithms to enable real-time mapping and localization, selecting the most effective approach for dynamic environments.

4. **YOLO Algorithm Implementation:** Train and implement the YOLO algorithm for real-time object detection and recognition, ensuring the robot can accurately identify and classify obstacles.

5. **Development of Behavior Trees:** Create custom behavior trees using BehaviorTree.CPP to manage the robot's navigation decisions, focusing on efficient maneuvering around obstacles.

6. **System Integration and Experimental Testing:** Combine the SLAM, object detection, and behavior tree components into the TurtleBot 4. Conduct experiments to test the system's performance in various scenarios, including obstacle avoidance and interaction.

7. **Performance Evaluation:** Compare the custom navigator's performance to the default navigator's. Specifically, the comparison should be based in the success of obstacle handling and the overall navigation effectiveness.

8. **Iterative Refinement:** Use the results from testing to refine and enhance the navigation algorithms and behavior trees, ensuring continuous improvement in the robot's maneuvering capabilities.

Chapter 2

Background and Related Work

2.1 Turtlebot4

To fulfil the objectives of this thesis, a Turtlebot 4 equipped with a Lidar Sensor and a Depth Camera was utilised. While a substantial body of research on this subject has been conducted in simulated environments, the deployment of a physical robotic platform necessitates a detailed investigation into the processes of configuration and interaction within a real-world context. This methodology not only augments the empirical foundation of the study but also facilitates a critical examination of the practical challenges associated with the physical implementation of robotics, thereby contributing to a more comprehensive understanding of the topic.

2.1.1 Overview

The TurtleBot 4 [Figure 2.1] represents a significant advancement in the TurtleBot series, recognised as the world's most popular open-source robotics platform, now enhanced for modern robotics education and research. It comes in two models: the TurtleBot 4 Standard and the TurtleBot 4 Lite. Both models are built on the iRobot® Create® 3 educational robot base and equipped with a Raspberry Pi 4, making them powerful tools for both learning and practical applications in robotics[5]. This platform is designed to be user-friendly, fully assembled, and shipped with ROS 2 pre-installed, allowing users from educators to researchers to get started with robotics application development right out of the box[6][7][8].

Figure 2.1: TurtleBot4 [6]

2.1.2 Features

TurtleBot 4 has several built-in sensors, including a 2D LiDAR [Figure 2.2], an IMU, and an OAK-D spatial AI stereo camera [Figure 2.3]. These sensors are very important for tasks like navigation, mapping, and spatial analysis. These sensors are fully accessible through ROS 2, facilitating advanced robotics functions like simultaneous localization and mapping (SLAM) and autonomous navigation. The onboard sensors and the open-source software environment make it an ideal platform for developing complex robotic applications[6][7][9].

The mobile robot base of the TurtleBot 4 offers impressive specifications, including a 9 kg payload capacity, which can be upgraded to 15 kg with a custom configuration. It supports a range of movement speeds and comes with a durable chassis capable of handling various educational and research activities. The platform also includes educational support through comprehensive courseware, tutorials, and access to a vast community of ROS developers, enhancing its utility in academic and practical learning environments[6][7][8].

Additionally, TurtleBot 4 supports the latest updates of ROS 2, including the Humble Hawksbill version, which introduces features like namespace support for better organisation and scalability of robotics applications. This update allows for more complex multi-robot systems and advanced operational capabilities in both simulated and real-world environments, highlighting the platform's adaptability and readiness for future robotics challenges[10].

Figure 2.2: RPLidar A1M8

2.1.3 Sensors

RPLidar A1M8

The RPLIDAR A1M8 is a cheap, compact-sized laser scanner with a 360-degree 2D scanning capability, making it suitable for 3D mapping and obstacle detection. Its operation is efficient due to the use of low power and great precision, even at the maximum range of 6 metres. This device is particularly advantageous in environments without direct sunlight exposure, where it can accurately capture the layout and contents of a space[11][12].

Designed for both indoor and outdoor use, the RPLIDAR A1M8 adapts well to various environments with its adjustable scan rate of 5 to 10 Hz. It provides detailed measurements with a resolution of less than 0.5 mm up to 1.5 metres and maintains an angular resolution of ¡1 degrees. Its rapid sampling rate and the ability to deliver over 2000 samples per second make it an excellent choice for real-time applications in robotics and surveillance[11][13].

It's commonly used in applications such as robot navigation, obstacle avoidance, and interactive projects that require environmental perception. This device represents a balance between performance and cost, offering robust functionality for a variety of automated systems and robotics projects[13].

Luxonis OAK-D Pro

The Oak-D Pro by Luxonis is an advanced camera designed for robotic vision, featuring high-performance specs tailored for precise depth perception and complex computer vision tasks in varying light conditions. The camera integrates an IR laser dot projector for active stereo vision, which significantly enhances depth perception, especially on surfaces with low visual interest like blank walls. An IR illumination LED that complements this makes it ideal

Figure 2.3: OAK-D Pro

for night operations by facilitating vision in low-light or completely dark environments[14].

Equipped with substantial computational power, the Oak-D Pro boasts 4 TOPS (tera operations per second), with 1.4 TOPS dedicated to AI tasks. This allows it to run intricate AI models, which can be custom designed and integrated according to specific requirements. The camera supports high-resolution video capabilities up to 4K at 30 frames per second or 1080P at 60 frames per second, with encoding ranging from H.264 to H.265[15].

2.2 ROS2 Framework

2.2.1 Overview

Robot Operating System (ROS) [16] is an open-source middleware framework developed for the creation and control of robotic systems. ROS operates on top of existing operating systems and offers tools, libraries, and conventions for tasks like hardware abstraction, communication between components, device drivers, and package management.

ROS was not particularly well-liked in the business world, and it did not fulfil a number of the most essential needs, including real-time computing, safety, certification, and security[17].

These restrictions created the need for ROS2. One of the objectives of ROS2 is to ensure that it is compatible with applications used in industrial settings. The capacity of ROS 2 to provide real-time capabilities is one of the most noteworthy advances included into the operating system. One of the biggest changes in ROS2 is the lack of a master node. Each node runs independently, thus allowing us to create a decentralised system. Other changes include the decentralisation of parameters and the now-asynchronous services[17].

The necessity for real-time performance grew more crucial as robots moved into applica-

tions that were more time-sensitive and safety-critical. Some examples of these applications are autonomous automobiles and industrial robots.

2.2.2 RCLCPP and RCLPY

RCLCPP and RCLPY are essential components of the ROS 2 ecosystem, serving as client libraries for C++ and Python, respectively. RCLCPP offers a reliable framework for programmers to incorporate high-efficiency and immediate functionalities crucial in robotics, utilising the capabilities of C++ to manage intricate operations and data within ROS 2 environments. This system is designed specifically for situations that require predictable performance and effective communication between various components of a robotic system. Conversely, rclpy is designed for developers who favour Python due to its simplicity and user-friendly nature, allowing them to rapidly create and develop applications. While RCLPY facilitates the use of ROS 2 features, it lacks support for certain functionalities found in RCLCPP, such as the ability to create custom behaviour tree nodes. These nodes are essential for defining custom behaviours in robotic applications. [18]

2.2.3 ROS2 Components

RCLCPP and RCLPY offer a robust suite of communication patterns specifically tailored for developers working within the ROS 2 ecosystem. These patterns facilitate streamlined interactions between nodes, effectively encapsulating the most common communication scenarios in robotic applications. The underlying architecture of ROS 2 is structured as a graph where various components are interconnected, allowing for the simultaneous processing of data. This design ensures that developers can implement complex, real-time data handling and communication strategies efficiently and effectively. An example of a ROS Graph is shown in Figure 2.4

Nodes

In ROS 2, nodes are fundamental elements that represent individual processes within a robotic system, each tasked with specific functions like sensor data processing or motor control. Nodes in ROS 2 are designed to operate both independently and cooperatively within the ROS 2 graph, a network where components interact through various communication patterns such as topics, services, actions, and parameters, as shown in Figure 2.4. Each node is

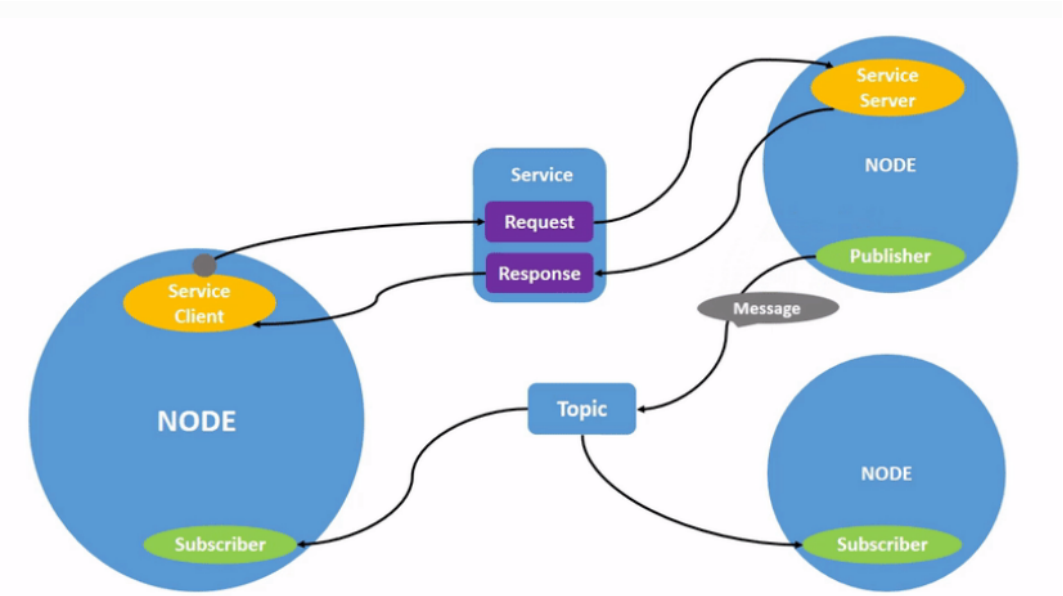

Figure 2.4: ROS Graph

responsible for a single, modular purpose, enabling efficient and effective data handling and computation.

Messages

Messages are a fundamental component for facilitating communication between nodes in a robotic system[19]. Each message in ROS2 is structured as a data type that can be defined by users according to their specific needs, using a format described by the ROS2 Interface Definition Language (IDL). The .msg files, where these definitions are stored, enable the specification of different data fields that can be standard primitive types or even other message types, as shown in Listing 2.1.

```
1 string firstname
2 string lastname
3 uint8 age
4 uint32 score
```

Listing 2.1: Example of a .msg message structure

Topics

ROS (Robot Operating System) topics are an integral component of the ROS communication architecture, facilitating the exchange of messages between different nodes within a network[20]. Using a publisher/subscriber model, ROS topics allow nodes to broadcast messages without requiring direct knowledge of the receiver's identity. This design supports a decoupled and highly scalable system where nodes can publish or subscribe to multiple topics simultaneously. For instance, in a robotic system, a sensor node might publish real-time sensor data on a specific topic while various other nodes subscribe to this topic to receive updates and react accordingly[21].

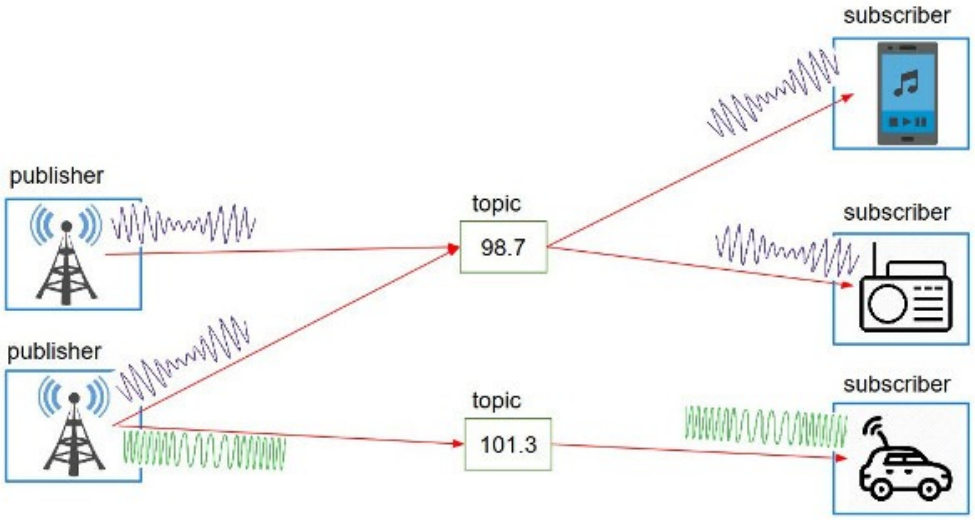

Figure 2.5: Real-world analogy of the publisher/subscriber model from The Robotics Back-End tutorial on ROS Topics [21]. Consider another radio transmitter broadcasting an AM signal at 98.7. It might be the same radio station or a different one. Occasionally, while driving, you may enter an area where two radio stations are transmitting on the same frequency.

Each ROS topic is designated to handle messages of a specific type, such as sensor data, state information, or control commands, which is determined at the time of the topic's declaration. This type consistency ensures that all communications on a topic are compatible and that the subscribing nodes can reliably process the received data.

The use of ROS topics exemplifies a many-to-many communication protocol where any node can act as a publisher or subscriber—or both—depending on the application's requirements. For example, in an autonomous vehicle, a node controlling the navigation might publish steering commands on one topic, while simultaneously subscribing to another topic that broadcasts sensor data from collision-detection sensors[21].

Services

Services diverge from the publisher-subscriber model typical of ROS topics. Instead, ROS services operate on a synchronous call-and-response model, providing data only upon direct request from a client. This ensures that data is delivered precisely when needed, enhancing system efficiency and control. To implement a service, developers define its structure and functionalities in .srv files, which specify both the request and response parameters[20].

```
bool data # e.g. for hardware enabling / disabling
---
bool success   # indicate successful run of triggered service
string message # informational, e.g. for error messages

```

Listing 2.2: SetBool service message definition

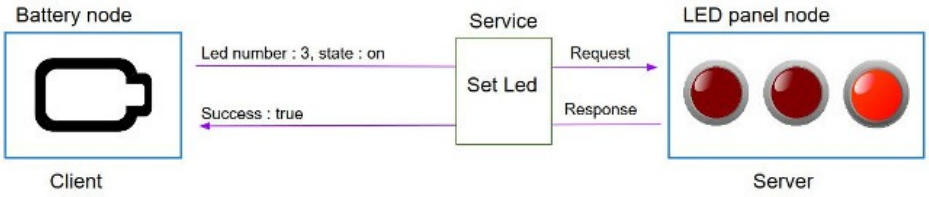

Figure 2.6: An example of a ROS service from "The Robotics Back-End" tutorial [22]. We have 2 nodes: a server node and a client node. The client node submits a request to the server to activate the third LED light. Upon receiving this request, the server processes and fulfills it, subsequently responding to the client node to confirm that the request was successfully executed.

Actions

Action servers and clients in ROS are specialized for facilitating multi-step, goal-oriented tasks, more intricate than the simple request-response interactions seen in Service-Client communications. These actions are well-suited for prolonged tasks, where action clients communicate with an action server, similar to services, to initiate and monitor the completion of these tasks. This process is structured around three phases: setting a goal, providing ongoing feedback, and delivering a final result[20]. Unlike typical services, action servers actively update the client about progress, ensuring detailed tracking and management of each step until the goal is achieved. This process is depicted in Figure 2.7.

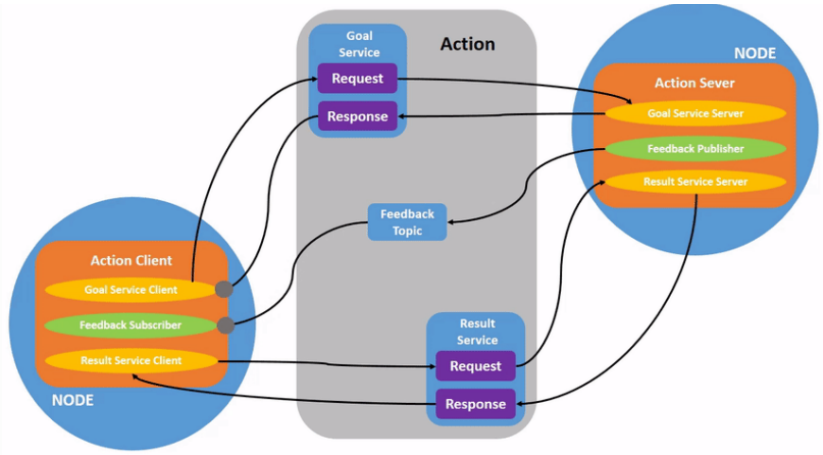

Figure 2.7: ROS Action Server and Client from ROS2 Documentation [23].

2.3 Robot Navigation

2.3.1 Nav2 Stack and Navigation server

The ROS 2 Navigation stack, commonly known as Nav2, leverages the ROS 2 framework to facilitate complex navigation tasks in autonomous robots. Nav2 features a collection of packages and libraries that enable robots to move, perceive, and respond instantly. Unlike its predecessor in ROS 1, which utilized a single process state machine, Nav2 employs a more sophisticated approach with behavior trees (BT). This allows for better management of multi-step or multi-state applications, improving scalability and user comprehension[24].

Nav2 integrates various elements such as odometry, sensor data, and velocity commands, which are vital for the robot to navigate from its starting position to a designated goal. The core of Nav2 includes a suite of action servers (explained above) like the planner, behavior, smoother, and controller servers, which interact with the top-level BT navigator. This navigator manages NavigateToPose action messages, coordinating the planning of paths, control efforts, and recoveries through smaller action servers connected to it[24][25].

BT Navigator

A Behavior Tree Navigator is an instrument for initiating and controlling the progress of planner, controller, and recovery servers for navigation[26]. It uses a behavior tree to coordinate these navigation tasks, thus making the development of specialized navigation behavior available by changing the behavior tree stored as an XML file. This reconfiguration is done

without programming because behavior trees can be efficiently designed with different types of control flow and condition nodes. Behavior trees are explained in great detail in Section 2.6. In this context, Nav2 uses the BT Navigator, which is built with the help of the BehaviorTree.CPP library. Thus, developing complex navigation behaviors is easy to overlay over top of the basic primitives. This creates more robot behaviors[27].

Planner Server

The Nav2 planner is one of the most fundamental Task Servers of the Nav2 framework and shall implement the nav2_behavior_tree::ComputePathToPose interface. This planning module establishes a feasible path, to be laid out between the current and final positions of the robot. For leading to that accomplishment, it loads a wide number of prospective planner plugins, which are capable to generate feasible paths best suited for different user-defined scenarios. In very simple words, a planner server calculates shortest or full-coverage or predefined routes in a complex environment. In return, these routes ensure that robots move safely, efficiently, and intelligently under different physical configurations and situational needs[26][28].

Controller Server

The controller server is the evolution of the local planners in ROS1, constituting the primary component for the execution of the navigation routes in the robotic system. The server maintains control of the stack's controller requests and the map of the plugin implementations to achieve the accurate execution of the navigation paths that are being generated by the planner server. By receiving path and plugin names for the controller, progress checker, and goal checker, the Controller Server calls the appropriate plugins, ensuring seamless operation. This server implements the nav2_msgs::action::FollowPath action server that receives its action goal as the computed path of the planner module in Nav2 and returns its command velocities. This server is designed to be flexible, hosting multiple plugins for path execution, each implementing functions from the virtual base class in the nav2_core package. The Controller Server builds and maintains the local cost map, dynamically updating the robot's trajectory for navigation through complex environments[28][29].

Behavior Server

The Behavior Server is seamlessly integrated with behavior trees, enabling a dynamic approach to managing robot actions. A crucial component of the Behavior Server is its ability to handle recovery behaviors, specifically designed to address unknown or failure conditions robots may encounter during navigation. These obstacles can include dynamic objects, temporary blockages, or unanticipated items not initially present on the navigation map. While planners and controllers guide the robot through expected environments, the recovery server tackles unforeseen challenges to ensure the robot can recover smoothly. This might involve actions such as backing up, spinning in place, attempting an alternative route, or moving from a problematic location to free space. Additionally, the Behavior Server manages various behaviors like recoveries and docking, hosting a vector of plugins implementing diverse C++ behaviors. While independent behavior servers can be created for each custom behavior, this unified server allows multiple behaviors to share resources such as costmaps and TF buffers, thereby reducing the incremental costs associated with new behaviors[28].

Smoother Server

The main purpose of a smoother server is to receive a path from the planner server and improve its quality by considering factors that affect the robot's movement, such as kinematics and acceleration. This server focuses on resolving problems associated with sudden movements that can occur during navigation, with the goal of reducing abrupt changes in speed or direction and maximising the distance from obstacles and areas with high costs. By incorporating a path, costmap, and other important data, the smoother server improves the path that various planning algorithms produce. This results in a more refined trajectory for the controller to track. As a result, the robot's navigation performance is improved through enhanced external behaviour, smoother turns, and the elimination of artefacts[28].

2.3.2 Turtlebot4 Navigator

The Turtlebot4 Navigator plays a vital role in the navigation system of the The Turtlebot Navigator initiates the essential services required for the robot to navigate and depends on the Nav2 Stack. It makes use of the capabilities of the ROS 2 Nav2 stack with behavior trees to implement a flexible and robust way of dealing with complex navigation scenarios. It adds on Nav2 Simple Commander. Depending on the Nav2 stack, the Turtlebot4 Navigator is capable of dynamic working during robotics tasks like path planning, obstacle avoidance,

and recovery behaviors[30].

2.3.3 Costmap2D

The concept of a costmap is crucial in navigation contexts, particularly in robotics. A costmap is a 2D grid representation of an environment where each cell contains a value indicating the occupancy probability. These values allow the planner to compute the most efficient path with minimal ”cost” by evaluating the likelihood of obstacles in each cell. There are two primary types of costmaps: the global costmap and the local costmap. The global costmap is derived from a map file and encompasses the entire map area. In contrast, the local costmap covers a smaller area and is updated in real-time using data from the robot's sensors, such as LiDAR and cameras[27][31].

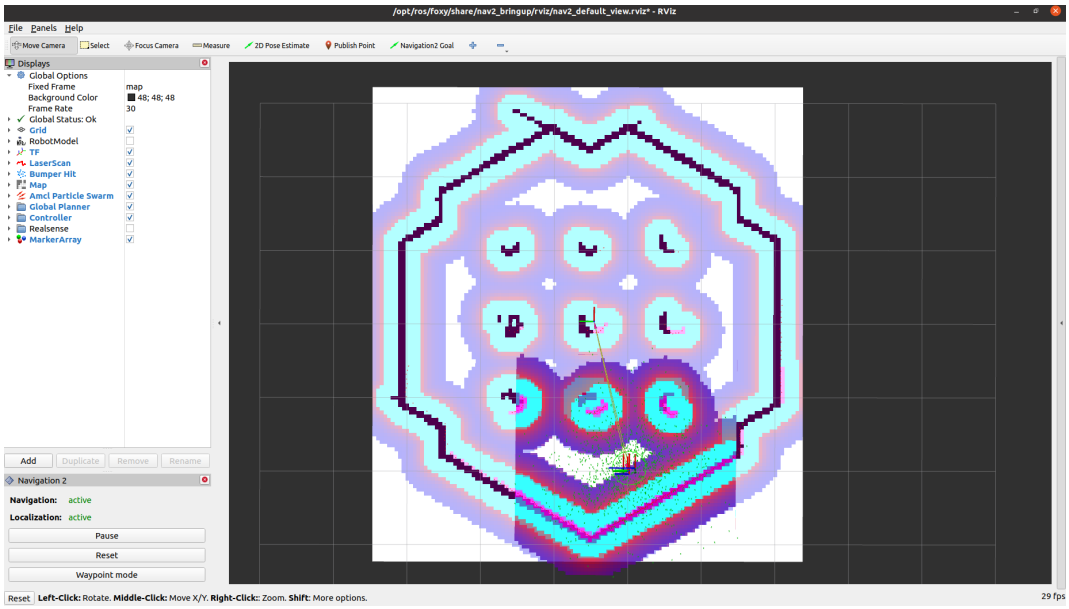

Figure 2.8: Costmap in RViz: The obstacles are shown with black color and the weights of the costmap around them are in color.

In the ROS 2 navigation stack, the costmap serves as the foundation for the planner and controller servers, enabling the robot to navigate through obstacles while minimizing a cost function. This functionality is facilitated by the nav2_costmap_2d package, which subscribes to sensor data and constructs a 2D or 3D occupancy grid with cell values ranging from 0 to 255[31]. Traditionally, a monolithic costmap was used, where all data was stored in a single grid. However, this approach had limitations, such as the loss of semantic context and difficulties in resolving conflicts between sensor data and global map values. To address

these issues, David V. Lu and colleagues developed the layered costmap approach, which organizes costmap data into semantic layers as shown in Figure 2.9. This method enhances the robot's ability to handle diverse contexts, ensuring more accurate and efficient navigation by segregating data based on its origin and significance.

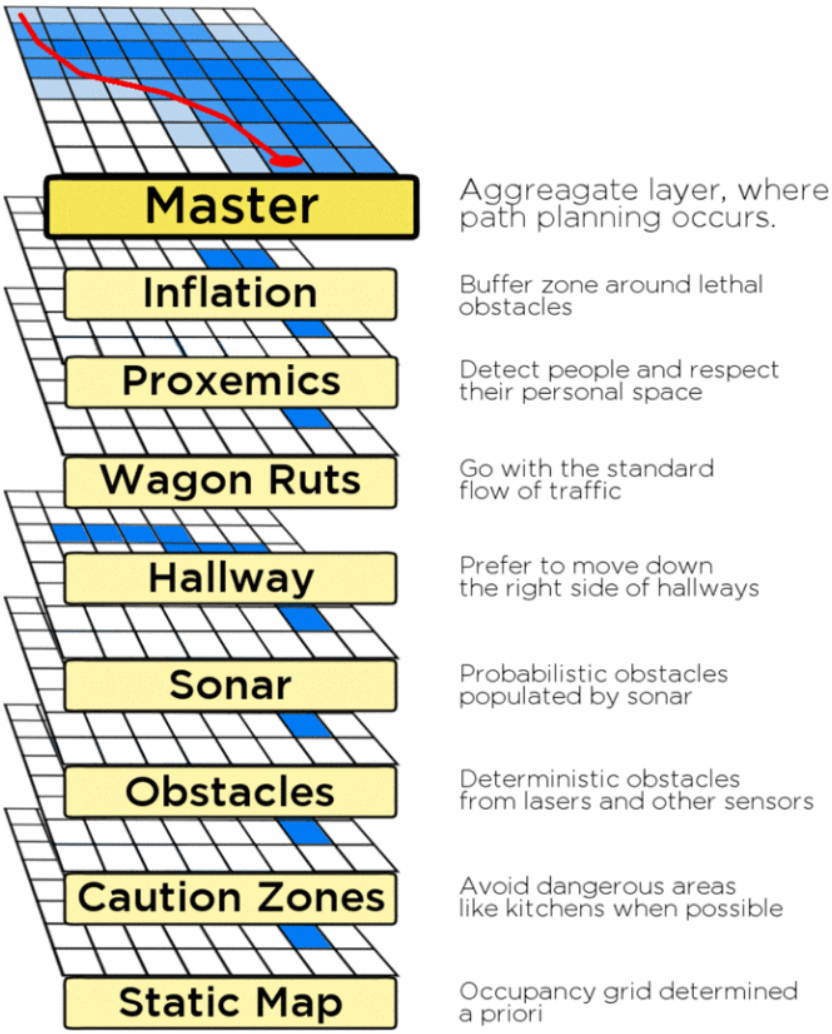

Figure 2.9: The stack consists of multiple costmap layers, which demonstrate the various contextual behaviours that can be achieved using the layered costmap approach[32].

2.4 SLAM

SLAM is a computational problem of constructing the map for an unknown environment with a robot/device simultaneously placed in that environment. This is, in general, what actually allows robots to conduct independent navigation within the environment without any pre-

built maps or external positioning system. The SLAM algorithms enable the creation of a map using sensory information from LiDAR, cameras, and IMUs. Updating will be done while correcting for errors and placing agent positions and the agent's actions. SLAM is applied in robotics, self-driving cars, drones, and augmented reality[28][33].

SLAM (Simultaneous Localization and Mapping) relies on odometry and external data to function effectively in any context. Sensors are crucial as they enable the mobile robot to gather information about the environment. Cameras and LiDAR collect visual data from the surroundings. Additionally, the IMU (Inertial Measurement Unit) is commonly employed in SLAM. This sensor measures various inertial parameters, including angular velocity, acceleration, Earth's magnetic field, and air pressure. IMUs are able to gather extensive internal and external data while being cost-effective. Unlike other sensors, IMUs are reliable in various conditions, whereas cameras require light, and GPS needs a signal to operate[34][35][36].

2.4.1 Different Types of SLAM

There are two kinds of SLAM algorithm: filter-based and optimization-based. Generally, the filter-based algorithms rely on the statistical foundations of the so-called Bayes filters, commonly related as probabilistic filters, and mostly develop and treat the SLAM problem as a state estimations problem. Optimization-based algorithms work to solve a direct-constrained state estimations problem, usually referred to as the least-squares problem, or an equivalent form, which may require a sparse solver[34].

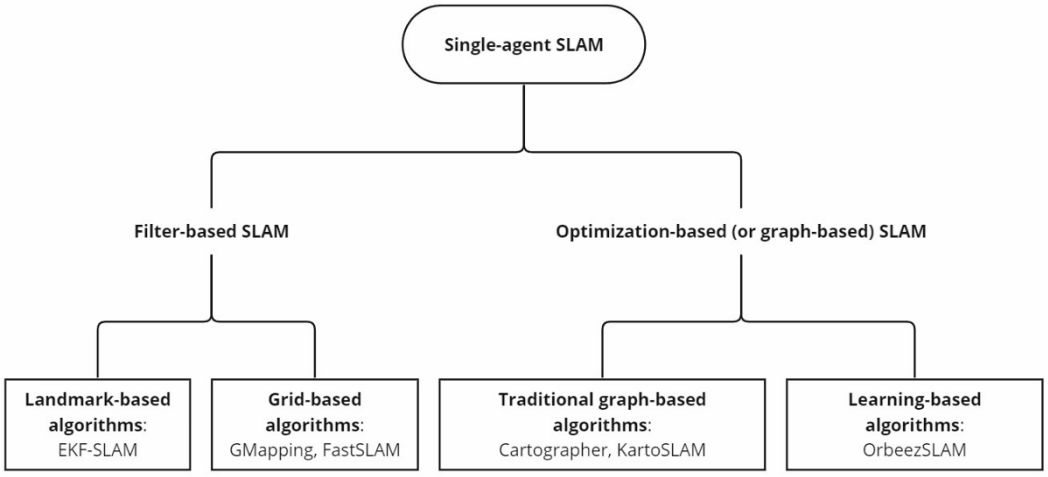

Figure 2.10: A rough classification of SLAM algorithms that are designed for a single mobile agent. [34]

Moreover, certain algorithms are tailored for specific sensor categories. The most frequently used technologies that are useful in the acquisition of frames of the surrounding in mobile robots are through the use of RGB-D cameras, stereoscopic visions, and LiDAR sensors. Thus, the SLAM algorithms are classified into two: visual SLAM (V-SLAM) and LiDAR SLAM. In turn, V-SLAM can be divided into monocular SLAM, and the stereo SLAM, depending on the type of camera sensor used.

Cameras are affordable, lightweight, and have broad detection ranges but are excessively dependent on light circumstances. RGB-D cameras are very sensitive to motion blur when mobile robots operate at high speeds. In practice, LiDAR SLAM has been more used than visual SLAM for indoor applications, and the use of LiDAR sensors has increased a lot on mobile robots. Laser sensors have better resolution but are less robust, more expensive, heavier, and particularly sensitive to the phenomena of refraction. Because the cameras are relatively cheaper and because of the advancements in computer vision, visual SLAM has become popular[37].

M. Filipenko and I. Afanasyev[38] compared different SLAM methods needed for indoor ROS-based robot mapping systems. They used lidar sensors, monocular cameras and stereo cameras.

- **Lidar SLAM:** GMapping, Hector SLAM[39] and Cartographer[40] are considered lidar slam systems. The first system does not provide reliable results in an indoor environment while the other two provide almost identical results[38].

- **Monocular SLAM:** Monocular systems like Parallel Tracking and Mapping (PTAM)[41], Semi-direct Visual Odometry (SVO)[42], Dense Piecewise Parallel Tracking and Mapping (DPP-TAM)[43], Large Scale Direct monocular SLAM (LSD SLAM)[44], ORB SLAM[45] and Direct Sparse Odometry (DSO)[46] can be used as additional sources of information but due to the lack of absolute scaling cannot be used by themselves for SLAM purposes[38].

- **Stereo SLAM:** Unlike monocular SLAM, stereo SLAM can solve localization problems for indoor robotics applications with sufficient accuracy[38]. Some stereo SLAM algorithms are: Real-Time Appearance-Based Mapping (RTAB map)[42], ORB SLAM[45] and Stereo Parallel Tracking and Mapping (S-PTAM)[47].

2.4.2 slam_toolbox

slam_toolbox is an open-source package for Robot Operating System, consisting of a collection of SLAM algorithms and utilities. It is a very robust framework that has been designed to implement SLAM in very different kinds of robotic applications. Other features are synchronous as well as asynchronous SLAM, multi-session mapping, lifelong mapping, and merging of maps. This acts as an advantage since the tool becomes quite versatile while working with a number of robots and use cases in 2D and 3D environments. By connecting with the Robotic Operating System, it provides powerful tools for processing sensor data, building maps in real time, and locating that greatly simplifies autonomous system development[48][34].

The slam_toolbox in ROS uses graph-based SLAM as its primary method. This type of SLAM constructs a graph of poses, which represent the position and orientation of the robot at different points in time. As the robot moves and gathers sensor data, it continually updates the graph to improve the map's accuracy and consistency by optimizing the relative poses between nodes in the graph[48]. This is an overview of how it works:

1. Graph Creation: When a robot is moving around the world, it is taking in sensor data and at different instants of time it is recording its poses inside the graph as nodes.

2. Loop Closure Detection: The system determines whether a robot, according to such information, after having left a place, has returned to that particular place by checking new sensor data against poses available on a graph. If this happens, an edge linking those poses will be drawn into the graph.

3. Optimization: The key part of graph-based SLAM is optimization. The relationship in the graph is fine-tuned to minimize total map error. This is mostly performed using Gauss-Newton or Levenberg-Marquardt-style algorithms—efficient optimization upon the raw form of links (edges) and positions (nodes) in a graph.

4. Map Updating: The map is continuously updated as new data comes in and optimizations are made in order to represent the robot's environment in the best way possible.

More specifically, the slam_toolbox algorithm is a big improvement over KartoSLAM, with a lot of differences from the original formulation. This newest implementation of KartoSLAM embeds the Ceres solver for more flexibility and to speed up optimization—a way to effectively solve non-linear least-squares problems inside the pose graph. Improvement

in the method used for the scan matcher from the slow Cholesky matrix decomposition has also been done. The SLAM Toolbox offers a reliable optimisation system that includes loop closure capabilities. This ensures that if there is any drift in the odometry, the measurements will be corrected. The package is entirely open-source[34].

Figure 2.11: slam_toolbox in action

2.4.3 Sychronous vs Asychronous SLAM

Synchronous and Asynchronous SLAM, both are the techniques in processing the sensor data to realize the mapping and localization. Synchronous SLAM processes sensor data as it comes in, allowing the sensing data to be synchronized and processed in a tight-coupled manner. The algorithm is able to wait for all sensor inputs required before updating the map and location of the robot to ensure high precision and consistency. Asynchronous SLAM processes sensor data independently and in parallel, dealing with latencies and irregularities in data collection from sensors. This feature makes it adaptive in relation to the environment and its changes in operation. While synchronous SLAM is often more accurate, the asynchronous one is much more robust and effective in dynamic environments with a high degree of uncertainty[48].

2.5 Object Detection and Recognition

One of the most important capabilities for a robot to perceive and interact with its environment is object detection and classification. Object detection identifies and localizes objects in an image or video stream, while classification ascribes labels to those objects based on

their categories. The process is quite vital in performing the necessary robotic applications, such as autonomous navigation, manipulation, and human-robot interaction. For example, an autonomous vehicle uses object identification and classification to be able to see pedestrians, other vehicles, and traffic signs in order to drive safely and smoothly. In an industrial set-up, the process of item identification and sorting through these methods is used with the goal of increasing productivity and accuracy in a product line.

In recent years, deep learning algorithms have gained popularity in the field of computer vision due to their superior performance compared to previous cutting-edge techniques in various tasks[49]. Object detection encompasses two primary tasks: object recognition and object localization[50]. When comparing the two-stage detectors, namely RCNN, Fast RCNN, and Faster RCNN, with the one-stage detectors YOLO v1, v2, v3, and SSD, it was observed that YOLO v3-Tiny improves the speed of object detection without compromising the accuracy of the results[51].

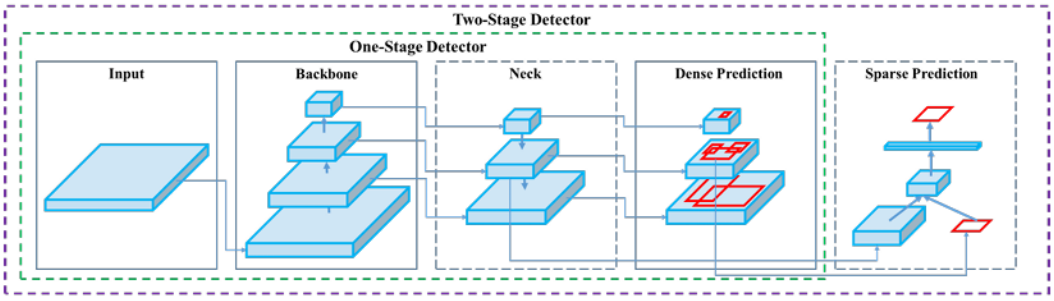

Figure 2.12: One-stage vs Two-stage Detectors

YOLO (You Only Look Once) is a cutting-edge system for object detection that has demonstrated exceptional speed and efficiency in real-time applications. This architecture is derived from Convolutional Neural Networks, but it restricts the system to perform only one forward propagation through the image analysis network. This technology is employed to analyse video feeds that consist of a series of images. Its purpose is to determine the location, identification, and classification of objects in each individual frame. The task at hand involves partitioning an input image into a grid of dimensions S × S. Within each grid cell, the system is designed to make predictions regarding the bounding boxes, along with their corresponding confidence scores and class probabilities. Subsequently, the network selects the particular number from these bounding boxes based on a confidence score that surpasses a predetermined threshold, signifying the detection of object locations. The YOLO algorithm approaches object detection as a regression problem, which allows for a streamlined process

and results in excellent computational efficiency. It achieves high processing speeds of over 30 frames per second on powerful computers[52].

The work of Vourkos et al.[53] presents the challenges in deploying a robot within the healthcare environment, which deviates substantially from structured industrial environments. The study centers on novel computer vision techniques for obstacles detection and navigation around dynamic obstacles, such as human beings and other robots, using deep learning–based methods developed over architectures like YOLO (You Only Look Once).

2.6 Behavior Trees

2.6.1 Overview

History behind their development

Behaviour Trees (BTs) were developed in the video game industry as a solution to the constraints faced with finite-state machines (FSMs) in managing non-player characters (NPCs). FSMs were commonly used for their simplicity but faced significant challenges in terms of modularity and scalability, especially when handling complex behaviors. The video game industry required a more flexible and modular approach, which led to the development of BTs. The implementation of this novel framework facilitated enhanced code reusability, incremental development of features, and improved efficiency in testing. Notable individuals who played a crucial role in the initial advancement of BTs were Michael Mateas and Andrew Stern. Early influential work in this field was done by Michael Mateas and Andrew Stern, creators of the interactive drama ”Façade”[54], and Damian Isla, who contributed to the AI in the game ”Halo 2”[55]. Nowadays, a wide variety of games including Spore[56], and GTA[57] use behavior trees.

When researchers realised BTs could handle the complexity and dynamic nature of robotic tasks, the BT industry moved from gaming to robotics. Academic papers discussing Behaviour Trees (BTs) in robotics were published by researchers such as Petter Ögren and Andrew Bagnell. These papers emphasised the benefits of BTs in developing modular and reusable behaviour modules for robots. Carnegie Mellon University played a pivotal role in this transition, utilizing BTs extensively for robotic manipulation and task planning. The modularity provided by BTs not only allows for the reuse of specific behaviours in various high-level tasks, but also enables non-experts to efficiently programme robots by taking advantage of the intuitive and adaptable characteristics of BTs[24].

How BTs work

Behavior Trees are structured as directed rooted trees where the internal nodes represent control flow nodes, and the leaf nodes represent execution nodes. A BT begins its execution from the root node, which sends activation signals, or ”ticks,” to its children nodes at a given frequency. The children nodes then execute based on these ticks, returning statuses of Success, Failure, or Running to their parent nodes. There are four main types of control flow nodes in BTs: Sequence, Fallback, Parallel, and Decorator[58].

Sequence nodes execute their children from left to right and return Success only if all children succeed. Fallback nodes also execute children from left to right but return Success if any child succeeds. Parallel nodes execute all children simultaneously and return Success or Failure based on predefined conditions. Decorator nodes modify the behavior of their child nodes, often used for tasks like repeating an action until success or imposing a limit on the number of executions.

BTs and Robotics

Behaviour Trees are employed in robotics to model and carry out intricate behaviours in a structured and hierarchical fashion. BTs empower robots to execute a diverse array of tasks by organising these tasks into reusable and independent modules. In the context of robotic task planning, Behaviour Trees (BTs) are employed to specify ordered sequences of actions, such as relocating to a specific position, gripping an item, and depositing it at a predetermined location. Every action can be depicted as a node in the behaviour tree (BT), enabling a clear and structured execution of tasks. Moreover, BTs enable the execution of actions without the need for synchronisation, allowing for safe interruption, which is essential for carrying out robotic operations in real-time. BTs are utilised in different areas of robotics, such as manipulation, navigation, and human-robot interaction. Open-source libraries and commercial frameworks offer the required resources for incorporating BTs into robotic software architectures[24].

Integration with ROS2

There is a great integration between ROS2 and behavior trees. An advantage of this integration is that the robot can take an alternative path at the time when the main one is blocked, without having to go through the replanning process using BTs. This would allow more agility in navigation for the robot. Besides, BTs can natively integrate with ROS2 actions

and services, treating them as nodes within the tree. This integration by BTs enables setting goals and monitoring progress effectively. For example, a BT can use a set of actions in order to control robot behavior, like moving to a waypoint or manipulating objects. The nodes in a BT can invoke services for clearing the cost map or updating the cost map with new data to improve the adaptability and responsiveness of the robot in changed environments[28].

2.6.2 Pre-defined Behaviour Tree Action Nodes

ComputePathToPose

The ComputePathToPose is one of the crucial nodes of behavior tree with respect to navigation stack in ROS 2. It sends a service call request to a global planner that is responsible for the calculation of the path from the robot's current position to a certain goal pose. The node takes in a goal pose, calls a global planner to calculate a path utilizing algorithms such A* or Dijkstra and then passes on an obtained series of the waypoints. In case of path computation failure, the node can direct fallback options or recovery processes[59][60].

FollowPath

The "FollowPath" behaviour tree (BT) action node is tasked with executing a pre-computed route by generating control instructions to guide the robot along the waypoints, while also ensuring that it avoids obstacles encountered along the way. When integrated into a behaviour tree, this system constantly checks the robot's progress, making adjustments to instructions depending on real-time sensor data and changes in the environment. The node collaborates with a local planner to guarantee seamless and secure navigation[61].

ClearEntireCostmap

The ClearEntireCostmap action behaviour tree node in ROS 2 is specifically meant to remove any data from the costmap (either the Global or Local costmap) that is used by the navigation system. This node is responsible for maintaining the robot's proper perception of the environment, particularly in situations when there have been large modifications or when the costmap has been congested with obsolete obstacle data. This node is integrated into the behaviour tree and is responsible for triggering the cleaning process. This procedure mitigates navigation challenges resulting from outdated or inaccurate information and is often used in situations when the environment has seen significant alterations, guaranteeing that the robot

can recalibrate and travel using the most up-to-date environmental data. This is helpful in our situation since once we move the obstacle, the environment will undergo a major change[62].

2.6.3 Pre-defined Behaviour Tree Control Nodes

Sequence

A Sequence node is a behavior tree control node that processes its child nodes one at a time, in a specific order. It starts by proceeding to the first of its children. If that child returns Success, then it moves on to the next child. If any child returns Running, then this Sequence node itself will return Running and will be resumed from exactly this tick point in the next tick. If any child reports Failure, the sequence node immediately relays that result without further checks. Only if all the children return Success does this node return Success[63].

Algorithm 1 Pseudocode for Sequence Node

```
procedure TICK
    for i = 1, ..., N do
        status ←child(i).Tick()
        if status == Running then
            return Running
        end if
        if status == Failure then
            return Failure
        end if
    end for
end procedure
```

Pipeline Sequence

Pipeline Sequence is similar to the Sequence node, but with the added feature that the preceding children are re-evaluated, mimicking the flow of water in a pipe[61].

If at any point a child returns FAILURE, all children will be halted and the parent node will also return FAILURE. Upon SUCCESS of the last node in the sequence, this node will halt and return SUCCESS.

Algorithm 2 Pseudocode for PipelineSequence Node

```
procedure TICK
    currentChild ← 1
    while currentChild ≤ N do
        for i = 1, ..., currentChild do
            status ←child(i).Tick()
            if status == Running then
                return Running
            else if status == Failure then
                return Failure
            end if
        end for
        currentChild ← currentChild + 1
    end while
    return Success
end procedure
```

Fallback

A Fallback node is a control node of behavior trees. It processes child nodes one by one in sequence until one of them returns Success. If first one returns Failure, it will tick to the next one, and so on. If one of the children returns Success, the Fallback node will immediately return with Success without executing the rest of the children. The Fallback node returns Success if a child returns Success, and Running otherwise; it will re-evaluate from that point on the next tick. It returns Failure only when all its children return Failure[64].

Algorithm 3 Pseudocode for Fallback Node

```
procedure TICK
    for i = 1, ..., N do
        status ←child(i).Tick()
        if status == Running then
            return Running
        else if status == Success then
            return Success
        end if
    end for
    return Failure
end procedure
```

Reactive Fallback

The Control Node is utilised to halt the execution of an asynchronous child node if any of the preceding Conditions transition from a state of FAILURE to SUCCESS. A ReactiveFallback node has a different behavior from a FallbackNode if it meets a running child. Unlike a typical FallbackNode that progresses to the next child when a child returns RUNNING, a ReactiveFallback begins the evaluation again from the first child. This means that the ReactiveFallback continues reevaluating all children from the beginning if any child is running and the highest priority action is performed[64].

Algorithm 4 Pseudocode for ReactiveFallback Node

```
Input: List of child nodes children
function REACTIVEFALLBACK(children)
    status ← Running
    while status = Running do
        for child ∈ children do
            childStatus ← child.tick()
            if childStatus = Success then
                return Success
            else if childStatus = Running then
                continue                          ▷ Restart from first child
            end if
        end for
        status ← Failure
    end while
    return Failure
end function
```

Recovery

A Recovery node is a Control node designed to add robustness by combining a main action with an alternative fallback recovery action. It only has two children: one is the main behavior, and the other one is the recovery behavior. The Recovery node ticks the main child. If he returns Success, the Recovery node returns Success. If the primary child returns Failure then, the Recovery node ticks the recovery child. This goes on until either the result of the primary child is Success—in that case, the Recovery node would return Success—or until the result coming from the child trying to recover is also failing, in which case it returns failure. The node thereby also regards a maximum number of retries to avoid infinite loops, lending to its appropriateness for cases when a primary action could fail, yet is correctable by the

corrective action[65].

Algorithm 5 Pseudocode for RecoveryNode

```
procedure TICK(number_of_retries)
    attempts ← 0
    while attempts < number_of_retries do
        status ← primaryChild.Tick()
        if status == Success then
            return Success
        else if status == Running then
            return Running
        else
            recoveryStatus ← recoveryChild.Tick()
            if recoveryStatus == Running then
                return Running
            else if recoveryStatus == Failure then
                return Failure
            end if
        end if
        attempts ← attempts + 1
    end while
    return Failure
end procedure
```

2.6.4 ROS2 Behaviour Trees Library

ROS2 offers a variety of pre-defined behavior trees that cover different needs. For example, it offers behavior trees with recovery mechanisms and without as well as different replanning mechanisms. The default behavior tree used is ”Navigate to pose with replanning and recovery” explained below.

Navigate_to_pose_w_replanning_and_recovery.xml

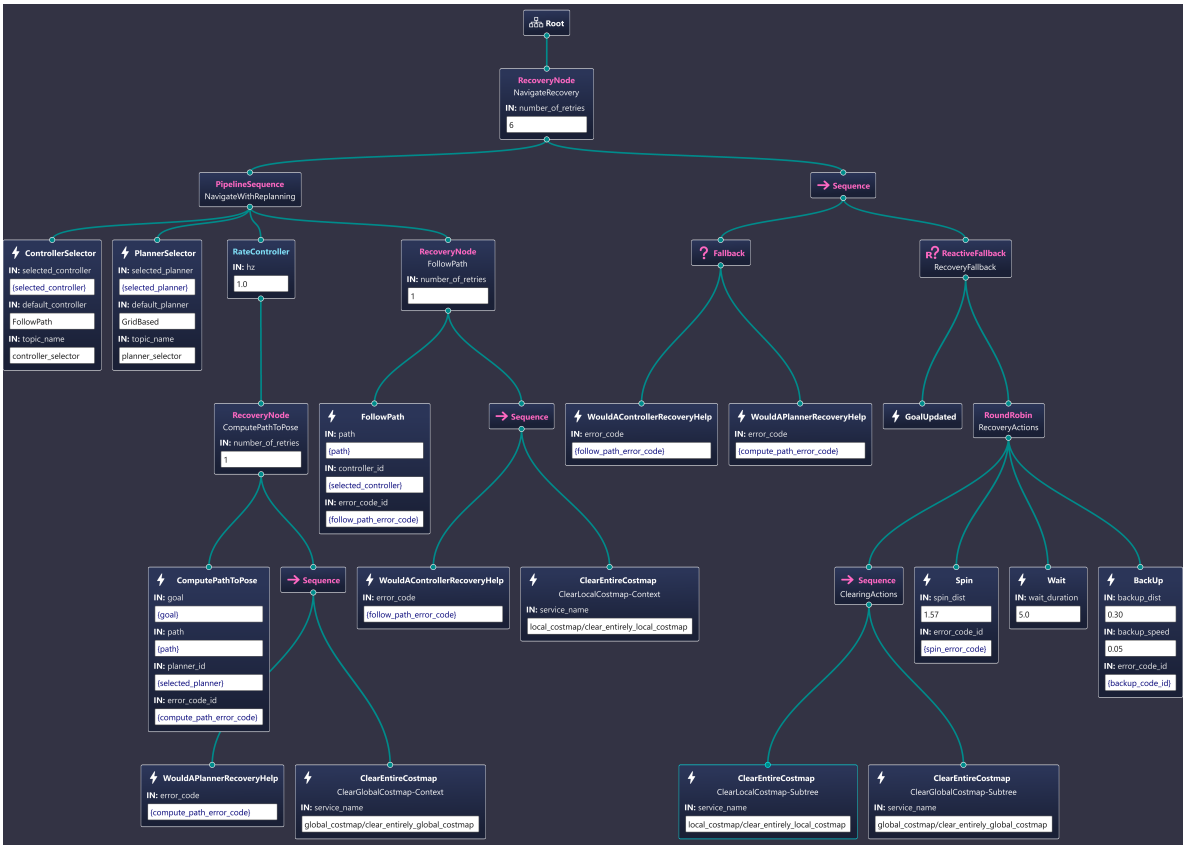

Figure 2.13: Visual representation of the default behavior tree in Groot

Left (Navigation) Subtree

- **Sequence 'NavigateWithReplanning':** This pipeline sequence node contains two children that will be executed the one after the other. However, in contrast to a simple seqeuence, it ticks the first child until it succeeds, then ticks the first and second children until the second child succeeds. If any of them fail, the tree will try executing the sequence node again for 6 times.
- **RateController:** This node is executed every one second. It ticks the 'ComputePathToPose' node. If the 'ComputePathToPose' fails it has a recovery mechanism to clear the costmaps.
- **FollowPath:** This node tries to follow the path calculated by the ComputePathToPose node. If it encounters an object along its path, it it has a recovery mechanism to clear the costmaps.

Right (Recovery) Subtree

- **ReactiveFallback 'RecoveryFallback':** This sequence node contains two children. As mentioned before, reactive fallback nodes return true if any of the children return true. However, unlike simple Fallback nodes, if at any point the goal is updated, it will interrupt the execution of the RoundRobin node and return true. This will prompt the BT to try and execute the left subtree again.
- **RoundRobin 'Recovery Actions':** The RoundRobin algorithm sequentially executes each child process until one of them returns a success status. It retains however the last ticked node, so it can continue from the next one. This means that during recovery it will first try to clear the costmaps. The next time the recovery actions are needed, it will tick the spin action node.

2.7 Similar projects

- **EG. Vourkos et al:** E. G. Vourkos et al.[53] discuss advanced techniques in detecting obstacles and navigating through dynamic healthcare environments using the YOLO algorithm. The paper further discusses the importance of real-time detection and obstacle tracking, which is fundamental for any autonomous robot in environments as complex and voluminous as hospitals. Using RGB-D sensor data in this method would enable better and enhanced accuracy and efficiency of detection and navigational tasks around moving obstacles. This approach is particularly beneficial in ensuring safe and reliable navigation for robotic systems in healthcare facilities, outperforming traditional vision-based methods in terms of speed and precision.
- **M. You:** The paper entitled "Enabling Autonomous Multi-Floor Navigation for Robots in ROS2 using Behavior Trees" by Minchu You[28] focuses on the problem of the autonomous navigation of robots in multi-floor environments. Building on the ROS2 Navigation2 stack and behavior trees, the study improves the robots' decision-making capabilities when it comes to dealing with environmental objects like elevators and stairs. The paper also includes adding behavior-tree based behaviours and navigation planners to existing ROS2 libraries; employing them in simulation and real-life scenarios.
- **M Colledanchise, L Natale:** "On the Implementation of Behavior Trees in Robotics"[66] discusses the use of Behavior Trees (BTs) as a tool to describe and implement robot

behaviors. The paper showcases the case of a robot that moves the arm in a pre-grasp position and then close the hand to fetch the object using behavior trees.

- **J Stüber, C Zito, R Stolkin:** The paper ”Let’s Push Things Forward: A Survey on Robot Pushing”[67] has as its goal the understanding of how pushing constitutes a primitive act in robotic manipulation. We look into some of the most common approaches for predicting and controlling the motion of pushed objects: analytical models, physics engines, and data-driven approaches, such as deep learning.

2.8 Useful software

2.8.1 Gazebo

Gazebo Simulator is an open-source 3D robotics simulator that creates realistic virtual environments for testing and validating robotic systems. It allows developers to model robots, simulate sensors and actuators, and conduct experiments in a dynamic, cost-effective manner. With features like physics-based rendering, Gazebo is widely used for prototyping and efficient development of robotic applications[68][69][70].

2.8.2 RViz

RViz is an essential visualization tool within the ROS (Robotics Operating System) ecosystem, used extensively in robotics to provide real-time visual feedback from a robot’s sensors. This information is published using topics. Users can view and interact with a variety of data types, including point clouds, depth maps, and path predictions, which are crucial for tasks such as navigation and manipulation. To facilitate effective navigation, RViz often utilizes a map consisting of a PGM file and a YAML file. The PGM file, a grayscale image, visually delineates the environment where black represents obstacles, white indicates free space, and shades of gray suggest unknown areas. These map files will be created after mapping the area using SLAM[69].

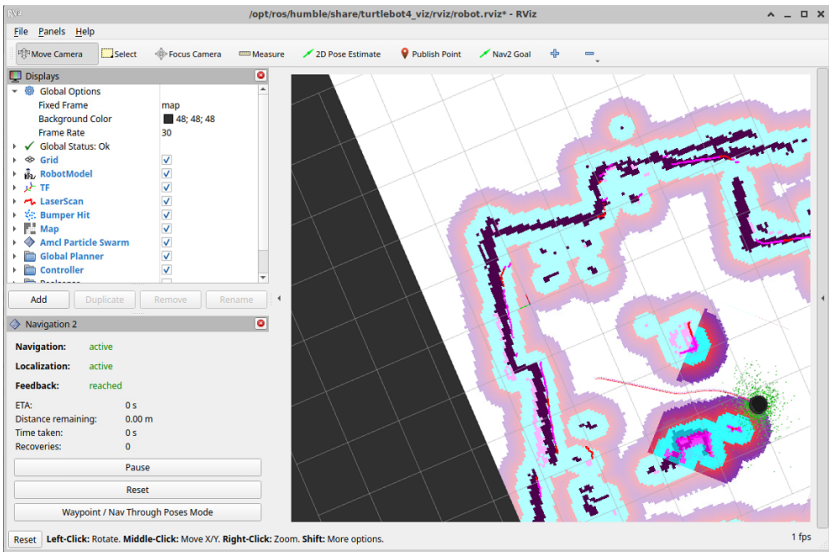

Figure 2.14: RViz interface

2.8.3 Groot

Groot is an advanced tool from the BehaviorTree.CPP suite designed for visually constructing and modifying behavior trees, which are crucial in robotics and AI decision-making processes. The Behavior Tree editor known as Groot2 allows users to effortlessly create and manage trees through a simple drag-and-drop interface. This editor is part of a broader toolset that supports live monitoring and on-the-fly editing, enabling developers to dynamically adjust and track behavior trees directly within robotic frameworks like ROS. Figure 2.15 showcases the structure of a behavior tree corresponding to the XML file shown in Figure 2.3, providing a clear visual representation of the tree's design and flow.

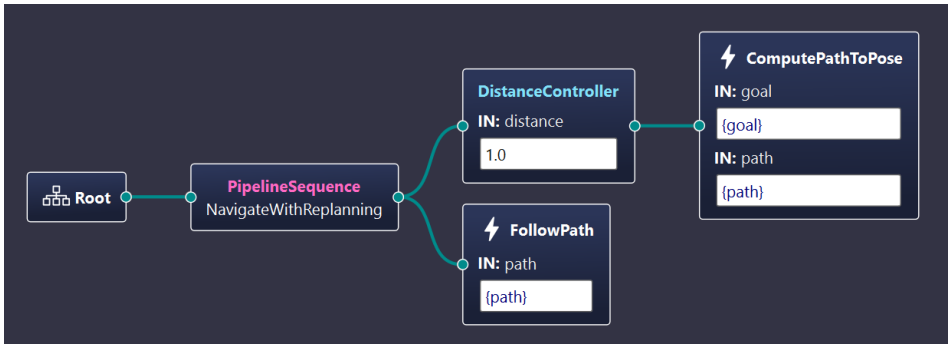

Figure 2.15: A simple behavior tree in Groot

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <PipelineSequence name="NavigateWithReplanning">
      <DistanceController distance="1.0">
        <ComputePathToPose goal="{goal}" path="{path}"/>
      </DistanceController>
      <FollowPath path="{path}"/>
    </PipelineSequence>
  </BehaviorTree>
</root>

```

Listing 2.3: A simple Behavior Tree representation in an XML file

Chapter 3

Implementation

3.1 Approach

In my project, rather than developing entirely new components, I aimed to maximize the reuse of existing elements from the TurtleBot ecosystem. This strategy eliminated the necessity for crafting bespoke navigation modules. Initially, I sought a high-level solution by implementing a Python script that integrated the Nav2 Stack to assign a destination goal. This script also utilized the robot's camera to detect obstacles in its path, attempting to halt its navigation and execute manual maneuvers when necessary. However, inherent limitations in the robot's built-in mechanisms and behaviors thwarted these efforts. For instance, the RPLidar sensor was capable of detecting obstacles from a considerable distance, prompting it to alter its course preemptively—a behavior that conflicted with my objective to physically interact with the object.

This challenge led me to examine the underlying mechanisms triggered by the nav_to_pose command. Through detailed investigation, I discovered that the nav_to_pose command, along with other navigational commands and the TurtleBot4 navigation scheme, relies on predefined behavior trees. These trees dictate the robot's actions both during routine navigation and in response to complications. Notably, the default behavior tree involved recalculating the navigation path every second, indicating a dynamic response to environmental changes. Recognizing the limitations of this approach for my specific goals, it became evident that crafting a custom behavior tree was essential, thereby shifting my approach from a high-level scripting solution to a more intricate, low-level customization of the robot's navigational behaviors.

To achieve this, I had to also implement custom behavior nodes and services. I needed

one node that would check the camera to detect and classify obstacles, and one node that would push the obstacles if they were objects.

3.2 Computer and Robot Setup

For the purpose of this thesis, I installed ROS2 Humble in the Turtlebot's Raspberry Pi 4B. The version used is a refined version of Turtlebot4 Standard, available here: http://download.ros.org/downloads/turtlebot4/.

For the remote PC, it's recommended to use Ubuntu 22.04 with ROS2 Humble installed. After installing the necessary packages, I tested connecting the wireless controller for manual navigation [Figure 3.1]. Some of the necessary packages were:

- ros-humble-turtlebot4-desktop
- ros-humble-turtlebot4-simulator

Figure 3.1: Navigation using a controller

In order for my approach to work, I had to disable the reflexes of the robot. The iRobot Create3 base has built-in mechanisms to prevent bumping into objects, moving backwards etc. I disabled ALL available reflexes and the limitations on speed.

3.3 Network Setup

On the TurtleBot 4 networking page, two main configurations are outlined for ROS 2 networking: Simple Discovery and Discovery Server.

- *Simple Discovery* is the default networking setup that utilises multicasting for communication. It allows all devices on the same network to automatically discover and communicate with each other's ROS 2 nodes. This setup is straightforward but may face issues with certain Wi-Fi networks that do not support multicasting well [Figure 3.2].

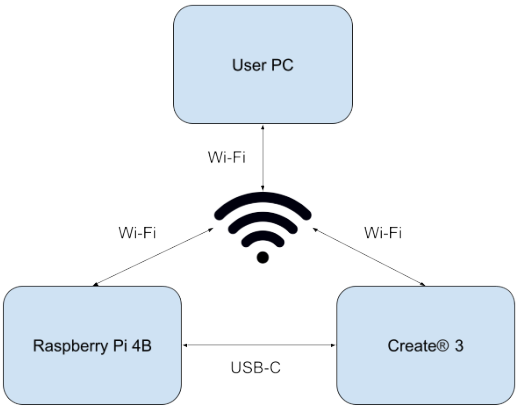

Figure 3.2: Simple Discovery [9]

- *Discovery Server*, on the other hand, designates one device as the server, which handles the discovery process for other client devices. This setup can bypass multicasting issues and does not require the Create® 3 to be connected to Wi-Fi. However, it requires more initial setup and only supports the FastDDS middleware [Figure 3.3].

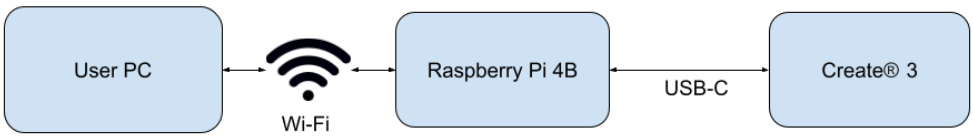

Figure 3.3: Discovery Server [9]

Due to restrictions in the lab's network, I chose Discovery Server. To test that the configuration worked, I checked the received topics from the computer [Figure 3.4].

```
lambros@ML-194235010821:~$ ros2 topic list
/amcl/transition_event
/amcl_pose
/battery_state
/behavior_server/transition_event
/behavior_tree_log
/bond
/bt_navigator/transition_event
/clicked_point
/cliff_intensity
/cmd_audio
/cmd_lightring
/cmd_vel
/cmd_vel_nav
/cmd_vel_teleop
/controller_server/transition_event
/cost_cloud
/diagnostics
/diagnostics_agg
/diagnostics_toplevel_state
/dock
/dock_status
/downsampled_costmap
/downsampled_costmap_updates
/evaluation
/function_calls
/global_costmap/costmap
/global_costmap/costmap_raw
/global_costmap/costmap_updates
/global_costmap/footprint
/global_costmap/global_costmap/transition_event
/global_costmap/published_footprint
/global_costmap/voxel_marked_cloud
```

Figure 3.4: Subscribed topics list

3.4 Mapping the area

Prior to the commencement of the coding phase of my project, I undertook the task of creating a detailed map of the laboratory space to facilitate the experiments. This map is depicted in Figure 3.5, where the walls and other immovable objects within the lab are marked in black.

The mapping was conducted in an empty laboratory setting, devoid of human presence and with all movable items removed. This approach ensures that the base map exclusively represents static obstacles, thereby enabling the robot to navigate around these fixed structures while remaining vigilant for any new or relocated objects that may appear subsequently.

This mapping process utilized the slam_toolbox, specifically employing a 2D SLAM technique paired with the RPLidar sensor to capture the spatial layout of the room. This method provides a reliable and accurate representation of the environment.

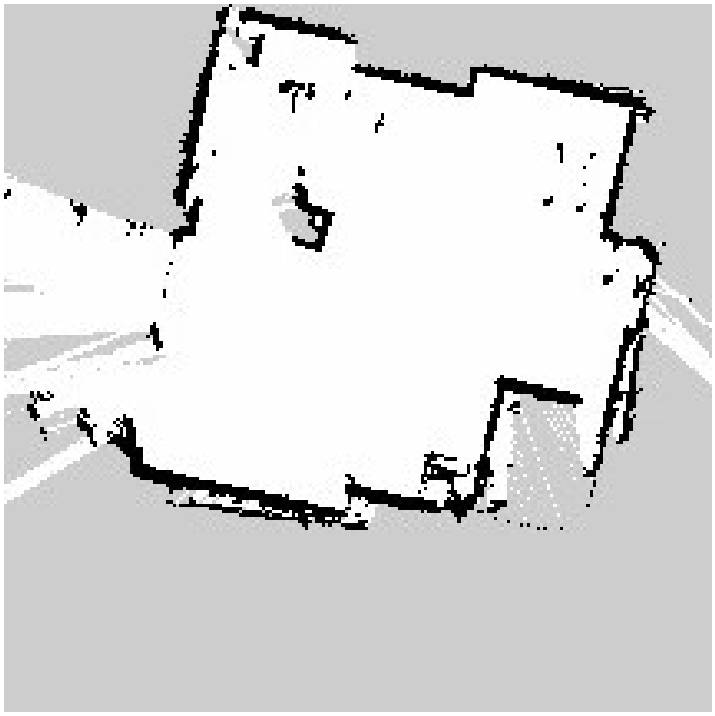

Figure 3.5: Map of the Robotics Lab

3.5 YOLO Model

Ideally, it would be beneficial to utilize a dataset capable of distinguishing between heavy and lightweight objects, thereby obviating the need for the robot to attempt manipulating heavier items. Regrettably, such a dataset was not available, necessitating the exploration of alternative strategies.

Initially, the identification of cardboard boxes was considered, assuming these to be inherently lightweight. However, this method is limited as cardboard boxes represent just one of numerous potential object types encountered within indoor environments.

A more comprehensive approach was subsequently adopted, focusing on the detection of lower body parts such as feet, thighs, and legs, including footwear. This methodology implies the presence of a human and thus prompts the robot to circumvent these figures. This enhanced detection capability significantly improves the robot's navigational efficiency in

complex indoor settings.

This approach can be extended to recognise other living entities commonly found indoors, such as dogs and cats, in order for the robot to effectively avoid all such obstacles.

3.5.1 Detecting Cardboard Boxes

For the detection of cardboard boxes I found a dataset in Roboflow which was labeled and ready to train a YOLOv8 model[71]. I downloaded it and trained the YOLOv8 model.

For the training I used batch_size 16 and the following filters:

- A.Rotate(limit = 10, p=0.5)
- Blur(p=0.1)
- MedianBlur(p=0.1)
- ToGray(p=0.01)
- CLAHE(p=0.01)
- ImageCompression(quality_lower=75, p=0.0)

After training, the model was tested out and had good overall results.

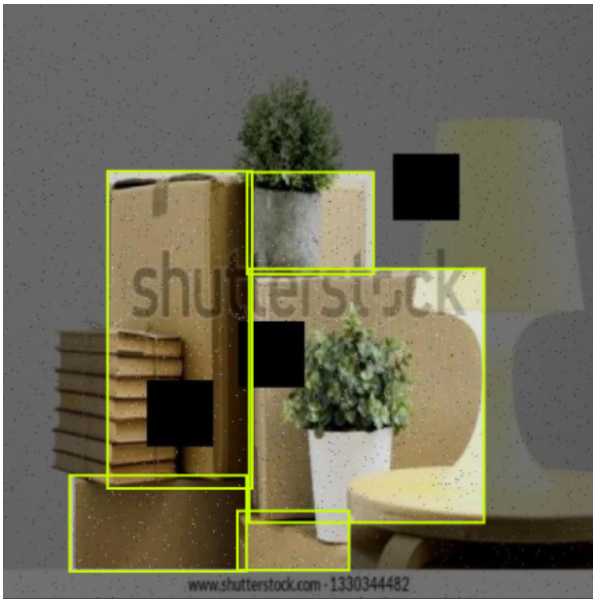

Figure 3.6: Cardboard detection

3.5.2 Detecting Lower-Body Parts

In order to implement this approach, I conducted an online search for a dataset containing images of ankles, legs, trousers, shoes, and knees. The rationale behind focusing on the lower body is that the camera on the robot is positioned at a height that makes it difficult to capture images of faces and arms.

I discovered a pre-trained model along with its corresponding dataset on Roboflow, specifically named "LowerBodyDetection"[72]. The dataset provided was of high quality, encompassing various versions that included pre-augmented images. Additionally, there was a pre-trained model available. Although the model used an older version of the dataset with far less images, it achieved 100% precision and 99.1% recall. The training metrics are shown in Figure 3.7. The existing model was trained in Roboflow using a YOLOv8s model pre-trained on the MS COCO dataset.

Figure 3.7: Predictions of the Roboflow Model for the "LowerBodyDetection" dataset

Despite the fact that a model already existed, I intended to improve it by utilising the updated augmented dataset containing over 5000 images.

1. **Roboflow Trained Model:** I uploaded the dataset to my Roboflow account and utilised a YOLOv8s model that was pre-trained on MS COCO to create a model. By doing this,

I was able to take advantage of the features provided by Roboflow and its pre-trained models, while also benefiting from the use of a larger and more up-to-date dataset.

The results of the training are shown in Figure 3.8.

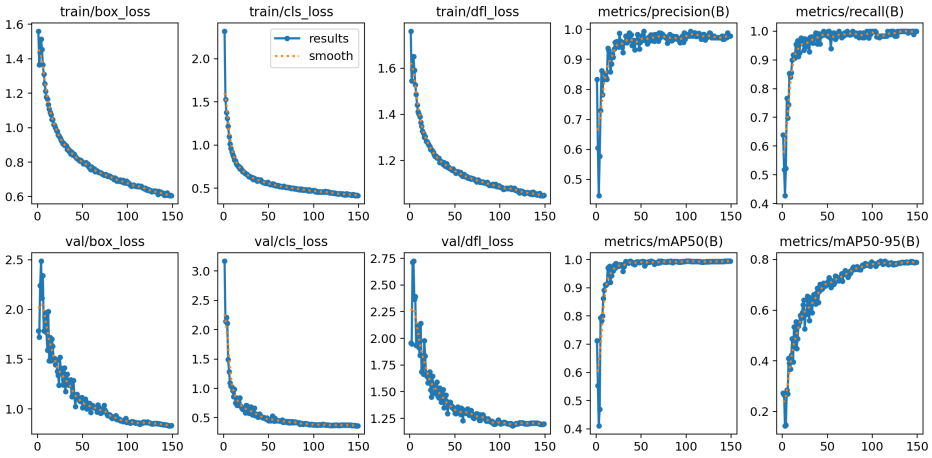

Figure 3.8: Roboflow Model for the ”LowerBodyDetection” augmented dataset

2. **Custom YOLOv8s Model:** I created a Google Colab notebook, downloaded the updated dataset and trained a YOLOv8s model pre-trained on MS Coco. While the outcomes would not surpass those of the pre-trained models in Roboflow, I chose to train a model manually and store its weights as a contingency plan in the event that the integration between Roboflow and my ROS2 subscriber service was not feasible.

   The results of the training are shown in Figure 3.9.

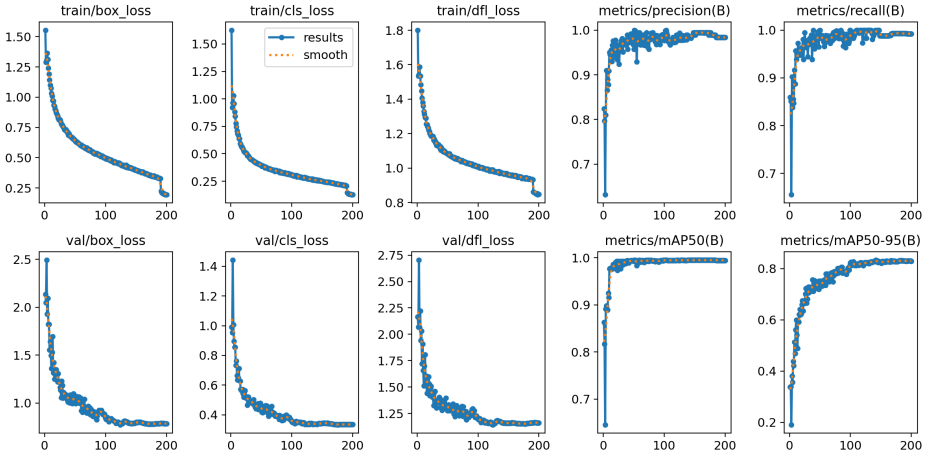

Figure 3.9: Google Colab Model for the ”LowerBodyDetection” augmented dataset

Upon completion of the training process, I observed that both models exhibit identical precision and recall metrics. Consequently, I now have the flexibility to choose either model for the construction of my YOLO service in the future.

3.6 Custom ROS2 Services

Prior to developing the custom behaviour tree nodes, I had to establish a service that would respond to requests for information from these custom nodes.

The initial concept involved the behaviour tree nodes subscribing to various ROS topics and retrieving the information independently. This would have left us with the task of just developing the behavior tree nodes. Our nodes were, however, hampered by the asynchronous nature of messages transmitted via topics. For instance, the node may experience a delay in its operation until it receives a message. After numerous attempts, I made the decision to abandon this plan and devise a more effective solution.

A more effective and simpler strategy involved developing two services that would subscribe to these topics and maintain the most recent state of the robot as shown in Figure 3.10. Upon receiving a request from the behaviour tree nodes, they would provide this information synchronously. The presence of services simplified the C++-written behaviour tree nodes and enabled the development of services in Python.

Instead of using custom service messages, we used SetBool, which is shown in Listing 2.1.

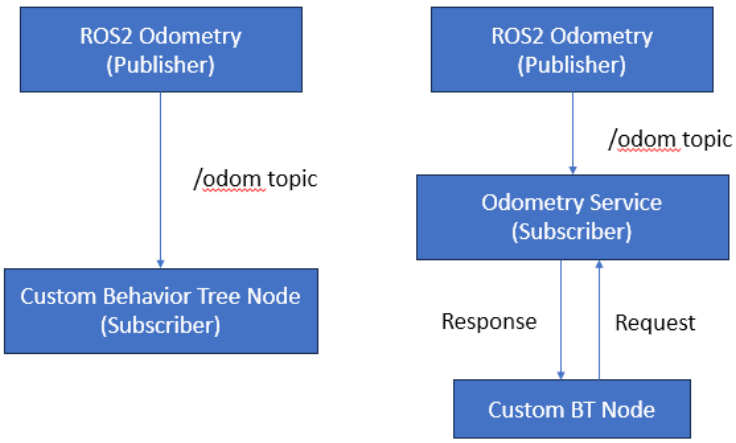

Figure 3.10: The initial concept without the utilization of services on the left and the updated approach on the right.

3.6.1 Odometry Service

This service susbcribes to the '/odom' ROS topic and receives messages about the position of the robot as shown in Listing 3.1. The QoS policy was set to 'BEST_EFFORT'. Every time a new message is received from topic '/odom', the callback method is called and the last known position (variable 'position') is updated.

```
def __init__(self):
    super().__init__('yolo_service')
    self.srv = self.create_service(SetBool, 'get_odom', self.
    get_odom_callback)
    self.position = None
    qos_policy = rclpy.qos.QoSProfile(reliability=rclpy.qos.
    ReliabilityPolicy.BEST_EFFORT, history=rclpy.qos.HistoryPolicy.
    KEEP_LAST, depth=1)
    self.subscription = self.create_subscription(
        Odometry,
        'odom',
        self.listener_callback,
        qos_profile = qos_policy)
    self.subscription  # prevent unused variable warning

def listener_callback(self, msg):
    if (self.position.x != msg.pose.pose.position.x or self.position.y !=
    msg.pose.pose.position.y):
        print(str(msg.pose.pose.position.x)+" "+str(msg.pose.pose.
    position.y))
    self.position = msg.pose.pose.position
```

Listing 3.1: Subscription to the odom topic and the callback function

Upon request, it provides the last known position of the robot to the MoveForwards node. It used the boolean value 'success' of SetBool to signal if it has a stored position or not and the string 'message' to pass both x and y to the callee. The code is shown in Listing 3.2.

```
def get_odom_callback(self, request, response):
    if self.position:
        response.message = f'Position: x={self.position.x}, y={self.
    position.y}, z={self.position.z}'
        response.success = True
```

```
5         self.get_logger().info('Requested. Replied "%s"' % response.
    message)
6     else:
7         response.message = 'Position not available'
8         response.success = False
9         self.get_logger().info('Requested. Replied "%s"' % response.
    message)
10    return response
```

Listing 3.2: Service callback function

3.6.2 YOLO Service

This service has two main objectives:

1. Use the camera sensor to detect humans.
2. Provide this information to the behavior tree nodes upon request.

There was no need to have two seperate entities doing these jobs, so I combined them in a single service.

Use of Image Topic instead of the camera feed

The OAK-D Pro camera installed on our robot has the ability to execute detection models within its processing unit. Roboflow offers a comprehensive tutorial on converting models online into a format that is compatible with deployment on OAK-D Pro. This can be done using the roboflowoak and depthai libraries. Nevertheless, after adhering to the instructions and attempting to execute the model with the camera, I encountered an error indicating that the resource was busy. Evidently, the other active Turtlebot modules were keeping the camera busy. Disabling the Turtlebot services was not feasible as I heavily depended on them for tasks such as navigation and SLAM. Accessing the camera just as an RGB input was also not possible.

Initially, an external camera was connected to the robot and employed as an RGB webcam, transmitting a live stream to the ROS service. After careful consideration, I determined that the most optimal course of action was to utilise the existing ROS topics. Turtlebot services publish an Image message that contains an RGB preview captured by the camera under

the topic 'oakd/rgb/preview/image_raw'. Instead of connecting an external camera and running the YOLO model on the Raspberry Pi of the robot, I could receive the image topics on another computer with more computational power and detect if there are humans present. This could significantly lower the latency of the detection.

Implementation

To implement this service, I created a subscriber for the 'oakd/rgb/preview/image_raw' topic. I also imported the model using the inference library for inference. To hold the last states of the YOLO detection, I created an array of length 5 that will keep the last 5 results of our model. By doing this, I strengthen my service against unintentional false positives or false negatives that the model might provide briefly. These are shown in Listing 3.3.

```
def __init__(self):
    super().__init__('yolo_service')
    self.srv = self.create_service(SetBool, 'get_yolo_state', self.
   get_yolo_state_callback)
    self.last_states = [False,False,False,False,False]
    self.subscription = self.create_subscription(
        Image,
        '/oakd/rgb/preview/image_raw',
        self.listener_callback,
        10)
    self.subscription  # prevent unused variable warning
    self.publisher = self.create_publisher(Bool, 'yolo', 10)
    self.model = get_model(model_id="lowerbodydetection/1")
```

Listing 3.3: YOLO Service Initialization

After initialising the service, we created two callback functions: one that would be executed every time an image is sent using the 'oakd/rgb/preview/image_raw' topic, and one that handles service requests.

Upon receiving an image, the first step is to convert it using cv_bridge and then pass it to the model for detection. By utilising the supervision library, we verify the presence of any detections. Subsequently, we proceed to shift the values in our array and save our updated boolean value at the final position. The value "true" signals that it is safe for the robot to push the obstacles, indicating the absence of humans. Conversely, the value "false" indicates that humans have been detected and the robot must recalculate its route.

```
def listener_callback(self, msg):
    bridge = CvBridge()
    img = bridge.imgmsg_to_cv2(msg, "bgr8")
    results = self.model.infer(img)
    detections = sv.Detections.from_inference(results[0].dict(by_alias=
    True, exclude_none=True))
    msg = Bool()

    for i in range(0,len(self.last_states)-1):
         self.last_states[i] = self.last_states[i+1]
    if detections.__len__() > 0:
        self.last_states[-1] = False
        msg.data = False
    else:
        self.last_states[-1] = True
        msg.data = True
    self.publisher.publish(msg)
    self.get_logger().info('Publishing: "%s"' % msg.data)
```

Listing 3.4: Image Callback Function

For the second one, upon a request from our behavior tree node, we check if the majority of the values in our last_states array are true. If true, it indicates that it is highly unlikely that a human is positioned in front of the robot. Consequently, we reply by employing a SetBool message, wherein we assign a value of true or false to its success parameter, denoting the absence or presence of a human.

```
def get_yolo_state_callback(self, request, response):
    morethan3 = sum(self.last_states) >= 3
    self.get_logger().info('Requested. Replied "%s"' % str(morethan3))
    response.success = morethan3
    response.message = str(morethan3)  # Send state as a message (string)
    return response
```

Listing 3.5: Service Callback Function

3.7 Custom ROS2 Behavior Tree Nodes

3.7.1 Check Camera Node

This custom behavior tree node is a ConditionNode and receives no input parameters. When ticked, it sends a service request to the YOLO Service and waits for the response. If the success boolean value response message is true, the node returns BT::NodeStatus::SUCCESS. Otherwise, it returns BT::NodeStatus::FAILURE. Depending on this return value, the behavior tree follows a different execution order.

3.7.2 Move Forward/Backwards Node

This custom behavior tree node is also an ActionNode and receives two parameters: speed and distance. The speed parameter is optional.

When ticked, it sends a request to the Odometry Service to get the robot's initial position. After that, it publishes a Twist message under the topic 'cmd_vel' setting the forward speed of the robot. If the distance is negative, it sets a negative speed to move the robot backwards. It keeps sending this messages for the robot to keep moving until it covers the required distance. To check if the required distance is covered, the node requests the new position from the Odometry Service every 100 ticks.

Pushing heavy objects

Although I can tell if an obstacle is a human or an object, there is no way to tell if an object (e.g., a cardboard box) is heavy or light-weight. There is a possibility that the robot tries to push the object but fails to do so. Because of that, we must have a mechanism in place that will cancel the action and prevent the robot from pushing more. Given the code we have at this stage, the robot will continue to push until it covers a certain distance, something that will never be achieved if the object is heavy.

We introduce two mechanisms:

1. **Timeout:** The service will time out after a certain amount of time based on the speed and distance it has to cover. This will prevent the object from pushing forever.

2. **Insufficient Distance Checker:** Having a timeout is not enough. Until this time passes, the robot's wheel keep spinning, scratching the floor's surface and destroying the wheel themselves. An effective way to prevent this from happening sooner is to

check whether the robot has failed to move a sufficient distance. Every 100 ticks, i request the current x and y position from the Odometry Service and check if the distance between the previous and current position is close to 0. If yes, then we must abort.

In both of these cases, the service will return a BT::NodeStatus::FAILURE if the robot can not move the object. Otherwise, if the robot reaches it's goal, it will return BT::NodeStatus::SUCCESS.

3.8 Custom Behaviour Tree

3.8.1 Overview

In order for us to create custom navigation behavior for our robot we have to build our custom behavior tree using the nodes and services we created above. It is necessary to not only incorporate the logic for the robot's movement, but also to include contingency plans in case of any malfunctions or errors. Our xml representation of our behavior tree is shown in Listing E.1 and our graphical representation using Groot is shown in Figure 3.11.

According to the reasoning of our Behaviour Tree (BT), the robot will calculate its path at the start of the navigation, rather than doing so every second as the default BT does. Then it will start following the calculated path to the target. In the absence of replanning, the trajectory of the system will persist even if an obstacle is encountered. When the distance between the object and the obstacle becomes very small, we anticipate that the FollowPath node will abort. Because the FollowPath node is the first child of a RecoveryNode, that means that robot will follow some actions to overcome this failure. Initially, the system will examine the custom YOLO Node to determine whether the obstacle is a human or an inanimate object. If the node detects an object, it will indicate success, allowing the next action in the sequence to proceed. This action involves moving the robot forward and pushing the object aside. If the YOLO condition node yields a failure or if the robot exerts force but the object is of substantial weight, our sequence will be unsuccessful, resulting in the activation of the second node within the fallback node. This will prompt the robot to move in a backward direction. Regardless of the situation, once the object is moved or the robot moves backwards, it will proceed to reset its costmap and recompute its path. After this recovery section ends, the robot will continue following its path. If this recovery section fails as well, we are left with no choice but to proceed with the suggested recovery measures on the right subtree, such as spinning, clearing the costmaps, and backing up.

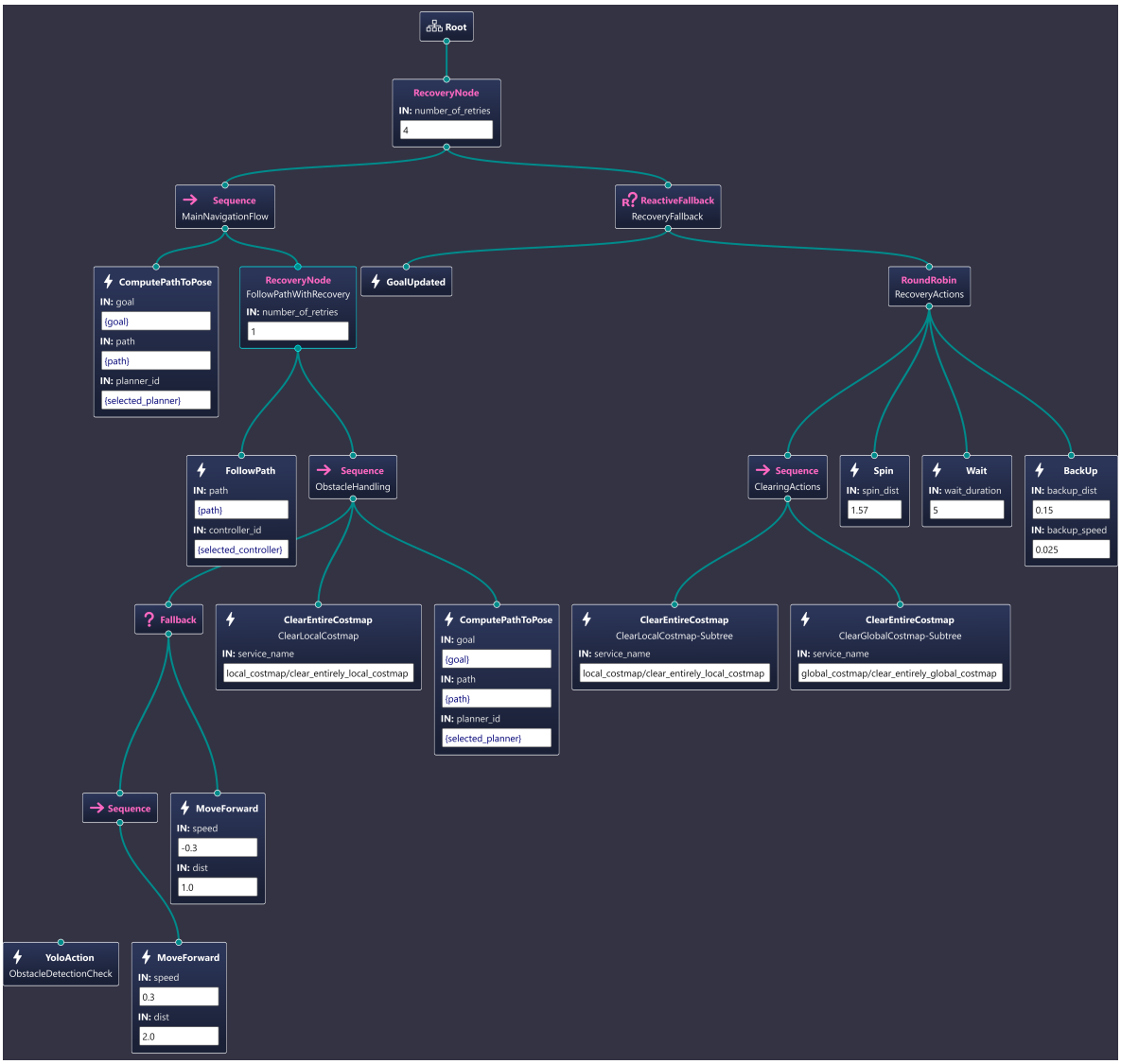

Figure 3.11: Graphical representation of Custom BT using Groot

3.8.2 Detailed Explanation

On the top, the BT has a recovery node, meaning that the left side of the tree is the regular subtree that we will follow and on the right we have the recovery sequence in case of failure. The left subtree will try to be executed 4 times before resorting to the fallback actions.

Left Subtree

- **Sequence 'MainNavigationFlow':** This sequence node contains two children that will be executed the one after the other. If any of them fails, we will try executing the sequence node again for 4 times.
- **ComputePathToPose:** This nodes calculates the best available trajectory to the goal.
- **Fallback 'FollowPathWithRecovery':** This node first ticks the FollowPath node. If it fails, it will tick its next child which is the sequence 'ObjectHandling'
- **FollowPath:** This node tries to follow the path calculated by the ComputePathToPose node. If it encounters an object along its path, it will return failure, executing the 'ObstacleHandling' Sequence.
- **Sequence 'ObstacleHandling':** This sequence contains a Fallback node ("CheckYolo") and some actions to be executed after it succeeds. The actions will clear the local costmap and recalculate the robot's path.
- **Fallback "CheckYOLO":** This Fallback node will check the YOLO Condition Node and move the robot forwards or backwards.

Right Subtree (Recovery)

The right recovery subtree of our custom behavior tree is the same as the one of the default behavior tree.

3.9 Navigation

As previously mentioned, I will utilise the default Turtlebot navigator but make alterations to its behaviour tree. In order to accomplish this, we need to register our custom behaviour nodes and create a custom behaviour tree. Once this is done, we must then configure the parameters of the navigator.

I generated a file named 'custom_params.yaml' and duplicated the default parameters of the navigator. I then added my custom nodes and set the default behavior tree of the nav_to_pose to mine. Each time the behavior server starts, it will load this custom tree and its nodes.

```
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
    default_nav_to_pose_bt_xml: ~/custom_tree.xml
    plugin_lib_names:
      - nav2_move_bt_node
      - nav2_yolo_bt_node
```

Listing 3.6: Custom parameters file

In order to start navigation, we must launch the Turtlebot's naviagator using **ros2 launch turtlebot4_navigation localization.launch.py map:=map.yaml** where 'map.yaml' is the map of our lab.

After the localisation service launches, we must launch nav2 using **ros2 launch turtlebot4_navigation nav2.launch.py**.

To set the initial position, we can either publish a message from terminal or open rviz to visualize the area and set the initial position on the map. Rviz can be launched using **ros2 launch turtlebot4_viz view_robot.launch.py**.

We can set the initial position using '2D Estimate Pose' and send a goal using 'Nav2 Goal' as shown in Figure 3.12.

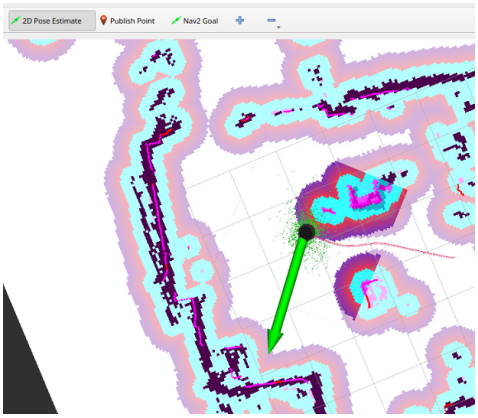

Figure 3.12: Setting the initial position in RViz

Chapter 4

Experimental Results

4.1 Scenario 1: Path Blocked by a Movable Object

4.1.1 Overview

The first scenario consists of 2 rooms connected by a door. When mapping the area, no obstacle was in place and the robot was able to move freely and map the 2 rooms. Then, a movable obstacle (a cardboard box) was placed in front of the door, blocking the entrance to the other room. Our turtlebot wants to go into the other room. This scenario is shown below in a vector (Figure 4.1) and in real life (Figure 4.2).

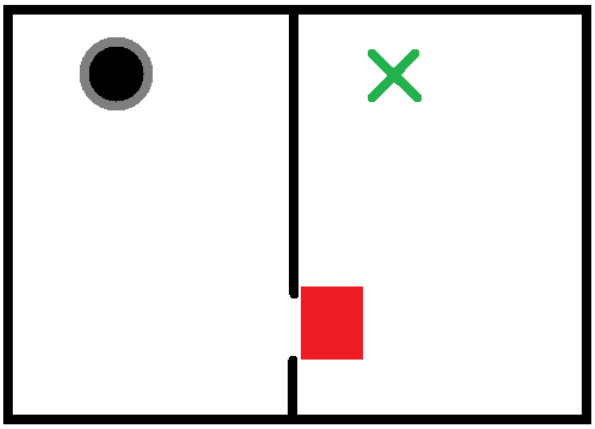

Figure 4.1: Scenario 1: Map of the proposed layout. The TurtleBot4 is shown as a black circle and the destination as a green cross. The obstacle is marked in red.

Figure 4.2: Scenario 1: Layout in real life

4.1.2 Results

The proposed trajectory goes right through the obstacle. The same happens for the default navigator as well. The robot has not seen the obstacle yet.

As the robot gets closer to the obstacle the lidar sensor detects it, and it is shown on the map.The default navigator tries to replan its route, fails to do so, enters recovery mode and eventually aborts its mission. The custom navigator goes right in front the object and stops as showin in Figure 4.3.

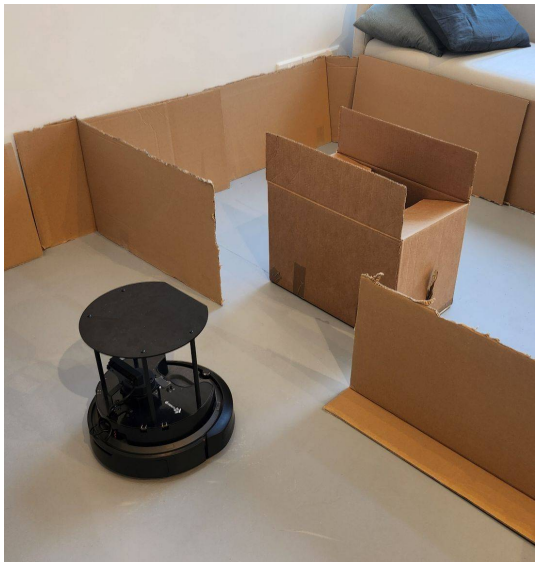

Figure 4.3: Scenario 1: The robot gets closer to the obstacle and stops

After it stops, the robot turns left and right. This is due to the fact that the ”FollowPath” node has not aborted yet. It is still trying to find an alternative to follow the predefined path. After it returns failure, our robot checks the camera, detects no human and moves forward 1 meter pushing the object out of the way as shown in Figure 4.4.

Figure 4.4: Scenario 1: The robot pushes the cardboard box

It then replans its route. Since the object has moved out of the way, the robot can now reach its destination as shown in Figure 4.5.

Figure 4.5: Scenario 1: The robot reaches its destination

4.2 Scenario 2: Path Blocked by a Heavy Object

4.2.1 Overview

This scenario is the same as the one depicted in Figures 4.1 and 4.2. However, in this case the object is heavy (a cardboard filled with items).

4.2.2 Results

The default navigator fails as in the previous scenario. The custom navigator approaches the object and tries to push it but the obstacle is not moving as shown in Figure 4.6.

Figure 4.6: Scenario 2: The robot tries to push the object

After 3 seconds, the robot moves back and tries to recalculate its route as shown in Figure 4.7. Since no route is available, it will enter recovery mode.

Figure 4.7: Scenario 2: The robot moves back

If there's no path for some time, it will abort the goal just like the default navigator. However, if we remove the object before this timeout, the robot finds the new path and follows it until it reaches the destination.

4.3 Scenario 3: Path Blocked by a Human

4.3.1 Overview

This scenario is the same as the one depicted in Figures 4.1 and 4.2. However, in this case path is blocked by a human.

4.3.2 Results

The default navigator fails as in the first scenario. The custom navigator approaches the human as shown in Figure 4.8.

Figure 4.8: Scenario 3: The robot stops in front of the human obstacle

The YOLO Service notifies the behavior tree node that a human is standing in front of the robot. The robot moves back without trying to push the human and tries to recalculate its route as shown in Figure 4.9. Since no route is available, it will enter recovery mode.

Figure 4.9: Scenario 3: The robot moves back from the human obstacle

If there's no path for some time, it will abort the goal just like the default navigator. However, if the human moves before this timeout, the robot finds the new path and follows it until it reaches the destination.

4.4 Scenario 4: Encountering a Lightweight Object

4.4.1 Overview

In the third scenario, the obstacle blocks the fastest route to the target, but there are still alternative paths for the robot to follow. This scenario is shown below in a vector (Figure 4.10) and in real life (Figure 4.11).

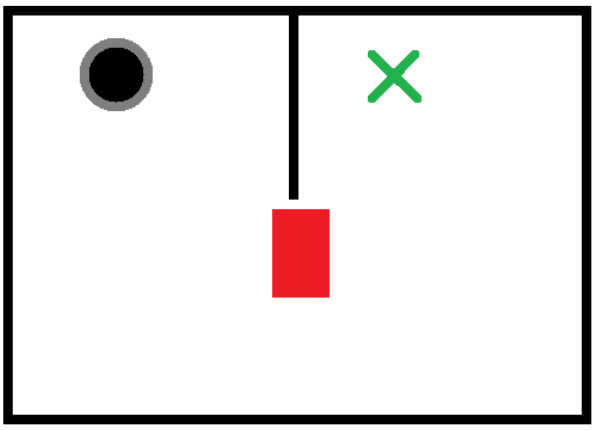

Figure 4.10: Scenario 4: Map of the proposed layout. The TurtleBot4 is shown as a black circle and the destination as a green cross. The obstacle is marked in red.

Figure 4.11: Scenario 4: Layout in real life

4.4.2 Results

The proposed trajectory for the custom navigator goes right through the obstacle. The same happens for the default navigator as well. The robot has not seen the obstacle yet. As the robot gets closer to the obstacle the lidar sensor detects it.

The default navigator re-plans its route along the way so it avoids the obstacle by going around it as shown in Figure. Although it is a longer route, it reaches the final destination without having to stop or push the object.

Figure 4.12: Scenario 4: Longer route followed by the default navigator

The custom navigator goes right in front the object and stops. it checks the camera, detects no human and moves forward 1 meter pushing the object out of the way as shown in Figure 4.13.

Figure 4.13: Scenario 4: The robot pushes the cardboard box

It then replans its route. Since the object has moved out of the way, the robot can now reach its destination as shown in Figure 4.14.

Figure 4.14: Scenario 4: The robot reaches its destination

4.5 Scenario 5: Encountering a Heavy Object

4.5.1 Overview

This scenario is the same as the one depicted in Figures 4.1 and 4.2. However, in this case the object is heavy (a cardboard filled with items).

4.5.2 Results

The default navigator behaves as in the previous scenario and although it follows a longer route it reaches its destination. The robot with the custom navigator tries to push the object but fails to do so. After 3 seconds it moves back as shown in Figure 4.15.

Figure 4.15: Scenario 5: The robot moves back

After that, it re-plans its route, takes the longer route the default navigator followed and reaches its final destination.

We can see that the custom behavior of the navigator does not work out always for the best. The robot lost significant time when it tried to push the object.

4.6 Scenario 6: Encountering a human

4.6.1 Overview

In this scenario, we have an empty room with people moving inside it. The robot must cross the whole room before reaching its destination as shown in Figures 4.16.

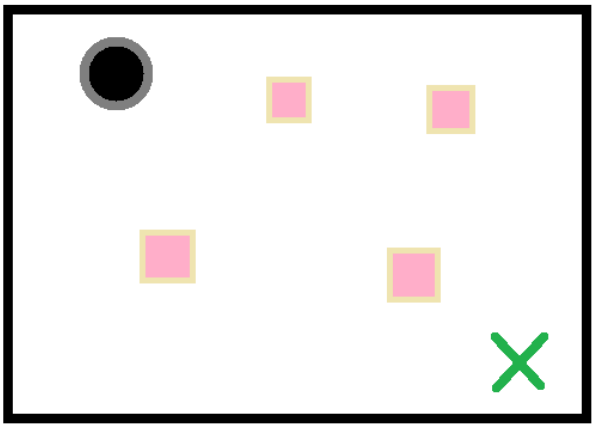

Figure 4.16: Scenario 6: Map of the proposed layout. The TurtleBot4 is shown as a black circle and the destination as a green cross. The moving people are marked in beige.

4.6.2 Results

The robot with the custom navigator follows its original route. When it comes close to a person, it checks its camera, identifies the person and moves a bit back as shown in Figure 4.17.

Figure 4.17: Scenario 6: The robot moves back after it detected a person

During this time, it gives the person enough time to move away before replanning the route. If the person does not move, the custom navigator plans around them as shown in Figure 4.18

Figure 4.18: Scenario 6: The robot re-plans its route and moves around the human obstacle

Chapter 5

Discussion

The evaluation of the performance of the custom navigator with the default navigator helps to understand the performance of the robotic navigation system in dynamic environment and especially in the hospital environment where there is always high level of unpredictability and interaction with humans.

5.1 Dynamic Environments with Movable Obstacles

The custom navigator demonstrated superior performance compared to the default navigator when dealing with lightweight and portable obstacles. This is exemplified in Scenario 1, where the customised navigator successfully pushed aside a cardboard box to clear its path and reached the final destination. In contrast, the default navigator failed to generate a successful new route and went into recovery mode and eventually aborted its mission. The fact that the custom navigator can move light obstacles and interact with its environment allows it to perform in dynamic conditions, hence increasing the real-world applicability such as for hospital logistics.

5.2 Handling Heavy Objects

The custom navigator did not manage to do better than the original one when it came to heavier objects, as can be seen in Scenario 2. The custom navigator tries to push the heavier object but cannot. The robot finally goes to its state of recovery due to the stall behavior and finally, aborts if there is no viable path. The default navigator, in its less aggressive strategy, does not even try to push the massive obstacle, thereby avoiding unnecessary behavior. This

means that, while the custom navigator is very good with light obstacles, it must improve handling heavier and immovable obstacles.

5.3 Crowded Indoor Environments

A person blocking the way is common in environments such as hospitals. Its effectiveness in such a situation is seen to be the most promising part of the custom navigator, as shown in Scenario 3 and 6. When encountering a human, the custom navigator gets close and, on detection, moves backward to allow the person to pass, after which it either re-plans or waits for the path to clear. This behavior is highly suitable for hospital settings in which, most of the time, individuals block paths. Hence, in this way, the default navigator does not prove to be very suitable in human-related, dynamic obstacle situations.

5.4 Path Recalculation and Patience

In such densely crowded settings, recalculating the path at every instant is not possible due to the very fast-changing surroundings. This makes the wait-and-then-recompute strategy advantageous when applied by the custom navigator. The waiting often provides time for human movement along the path, resulting in clear passage for the robot, which continues with its task without the need to re-plan. People usually give way to robots, and the robot will follow the trajectory without too much obstruction. Such a strategy of patience reduces unnecessary path computation and increases the robot's efficiency in dynamic human environments.

Chapter 6

Conclusion

6.1 Summary of Findings

This thesis aimed at advancing robots in such a manner that robots can move independently in dynamic indoor environments, like hospitals, using behavior trees for advanced strategies in navigation. The design and implementation of the custom navigator in this research, therefore, show significant improvement and have thus effectively realized the said objectives in improving the robot's ability to handle different kinds of obstacles.

Behavior trees are a powerful tool to use for the structuring of decisions, making the behavior modular, scalable, flexible, and valuable about the navigation of the robot. The value brought by a behavioral tree-structured, custom navigator that enabled the robot to physically interact and move obstacles in a scene with lightweight, movable obstacles was that the robot could navigate through the scene and maintain its path without any external intervention, thus showcasing a significant improvement over the default navigator.

Implementing such complex behaviors was greatly facilitated by the introduction of behavior trees, from proactive handling of obstacles to smooth transitions between different strategies for navigation. Structuring the flow of navigation into manageable nodes and their sequences facilitated more control of actions so that each decision is taken with relevance to the current environmental state.

Other advanced object detection features have been tapped into with behavior trees, such as YOLO, to detect and recognize humans and other obstacles. This has enabled the robot to well differentiate between barriers that it can move and those that it should avoid, thus increasing safety and efficiency.

The results of this work clearly illustrate the potential that behavior trees can introduce for

the next level of autonomous navigation. It fully met the main goal it was developed for: the implementation of an ad hoc navigator for enhancement in navigation efficiency. But it also sheds light on the flexibility and robustness properties of behavior trees under dynamic and uncertain environments. This approach offers a promising direction for future research and development in autonomous robotics, particularly in applications requiring high adaptability and reliability.

6.2 Contributions to the Field

The research presented in this thesis contributes to the field of autonomous robotics by demonstrating the practical application of advanced navigation strategies. The use of behavior trees for dynamic and unpredictable environments has shown to be effective in improving the maneuverability of autonomous robots. This work provides valuable insights and advancements in the following areas:

1. **Enhanced Autonomous Navigation:** The developed system showcases how behavior trees can be utilized to enhance the efficiency of autonomous navigation in complex environments.

2. **Improved Real-Time Decision Making:** The ability of the robot to make real-time adjustments to its path by recognizing and interacting with obstacles represents a significant advancement in autonomous navigation. The custom navigator, in some cases, outperforms the default one which follows a 'play-it-safe' strategy.

3. **Increased Safety and Reliability:** The use of advanced object detection and navigation algorithms has demonstrated potential improvements in the safety and reliability of autonomous robots.

6.3 Future Work

The implementation and deployment of the custom navigator using behavior trees presented in this paper has shown several of the possibly promising ways that can be further investigated and developed for better capacity and performance of robots in the hospital environment.

- **Enhanced Obstacle Detection:** One of the most important directions for further development is the improvement of algorithms for detecting obstacles. Such algorithms

should work at a proper level of reliability, robust enough for different lighting conditions and uncertain environments. These models can differentiate efficiently between different types of obstacles. Thus the robot is well-prepared to make an informed decision in the face of obstacle avoidance or interaction with objects in its path.

- **Integration of Reinforcement Learning:** Another promising area for future work is integrating reinforcement learning into optimizing a robot's decision-making process. The learning should take place in such a way that, after each evaluation, the robot shall know whether to decide to push an object or to recalculate the path toward accomplishing the goal. In this way, adaptive learning can improve the overall efficiency of robots and their potential to deal with a more general class of problems in dynamic environments.

- **Expanded Testing Environments:** The experiment should be carried out in a more diversified and realistic setting of hospital environments to generalize the robot's capabilities and identify the limits. Realistic hospital environments need to be accommodated within the testing of general wards and emergency rooms, among others, with varying levels of activity and obstacle configurations will provide valuable insights into the robot's performance. This more general approach to testing will identify potential hitches and points that need improvement to make sure the robot copes with the complexity of a real-life hospital environment.

- **Utilization of Gazebo Simulator:** The Gazebo Simulator is a robust testing and verification tool for robotic systems under controlled and cost-effective conditions. Future work will utilize the Gazebo Simulator in testing different scenarios and configurations of the robots under deployment before actual deployment in natural hospital settings.

References

[1] G. S. PERROTT and D. F. HOLLAND, "Population trends and problems of public health," *The Milbank Quarterly*, vol. 83, no. 4, p. 569–608, Nov. 2005.

[2] I. Vrabková and I. Vaňková, "Efficiency of human resources in public hospitals: An example from the czech republic," *International Journal of Environmental Research and Public Health*, vol. 18, no. 9, p. 4711, Apr. 2021.

[3] N. Ramdani, A. Panayides, M. Karamousadakis, M. Mellado, R. Lopez, C. Christophorou, M. Rebiai, M. Blouin, E. Vellidou, and D. Koutsouris, "A safe, efficient and integrated indoor robotic fleet for logistic applications in healthcare and commercial spaces: the endorse concept," in *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 2019, pp. 425–430.

[4] E. Toulkeridou, A. Kourris, E. Christoforou, R. J. Ros, M. Bosch, R. Lopez, A. Perrot, A. Godart, N. Ramdani, C. Pattichis *et al.*, "Safe robot navigation in indoor healthcare spaces," in *IEEE-EMBS International Conference on Biomedical and Health Informatics, BHI*, 2022.

[5] S. Krause and A.-L. Henk, "Selecting an educational robot: a comprehensive guideline," EasyChair, Tech. Rep., 2024.

[6] Clearpath, "TurtleBot4 Overview," [Accessed: 02.12.2023]. [Online]. Available: https://clearpathrobotics.com/turtlebot-4/

[7] S. Guillen, "Clearpath Robotics announces Turtlebot 4," 2021, [Accessed: 02.12.2023]. [Online]. Available: https://clearpathrobotics.com/blog/2021/10/clearpath-robotics-announces-turtlebot-4/

[8] ——, "Clearpath Robotics launches Turtlebot 4," 2022, [Accessed: 03.12.2023]. [Online]. Available: https://clearpathrobotics.com/blog/2022/05/clearpath-robotics-launches-turtlebot-4/

[9] Clearpath, "TurtleBot4 User Manual," [Accessed: 01.12.2023]. [Online]. Available: https://turtlebot.github.io/turtlebot4-user-manual/overview/features.html

[10] S. Munir, "Turtlebot 4 now supports ROS2 Humble," 2022, [Accessed: 03.12.2023]. [Online]. Available: https://clearpathrobotics.com/blog/2022/05/clearpath-robotics-launches-turtlebot-4/

[11] "Slamtec RPLIDAR A1 - SLAMTEC Global Network — slamtec.ai," [Accessed 29-05-2024]. [Online]. Available: https://www.slamtec.ai/product/slamtec-rplidar-a1/

[12] "RPLIDAR A1M8 - 360 Degree Laser Scanner Development Kit — generationrobots.com," [Accessed 29-05-2024]. [Online]. Available: https://www.generationrobots.com/en/402778-rplidar-a1m8-360-degree-laser-scanner-development-kit.html

[13] M. S. Aslam, M. I. Aziz, K. Naveed, and U. K. uz Zaman, "An rplidar based slam equipped with imu for autonomous navigation of wheeled mobile robot," in *2020 IEEE 23rd International Multitopic Conference (INMIC)*. IEEE, 2020, pp. 1–5.

[14] Luxonis, "OAK-D Pro — shop.luxonis.com," [Accessed 29-05-2024]. [Online]. Available: https://shop.luxonis.com/products/oak-d-pro

[15] "Luxonis OAK-D Pro Camera (auto-focus) — Génération Robots — generationrobots.com," [Accessed 29-05-2024]. [Online]. Available: https://www.generationrobots.com/en/404059-luxonis-oak-d-pro-camera-auto-focus.html

[16] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," vol. 3, 01 2009.

[17] ed, "ROS1 vs ROS2, Practical Overview For ROS Developers - The Robotics Back-End — roboticsbackend.com," [Accessed 29-05-2024]. [Online]. Available: https://roboticsbackend.com/ros1-vs-ros2-practical-overview/

[18] "Client libraries - ROS 2 Documentation: Rolling documentation," [Accessed 12-05-2024]. [Online]. Available: https://docs.ros.org/en/rolling/Concepts/Basic/About-Client-Libraries.html

[19] J. M. O'Kane, "A gentle introduction to ros," 2014.

[20] OpenRobotics, "Topics vs Services vs Actions - ROS 2 Documentation: Foxy documentation — docs.ros.org," [Accessed 29-05-2024]. [Online]. Available: https://docs.ros.org/en/foxy/How-To-Guides/Topics-Services-Actions.html

[21] The Robotics Back-End, "Multiple publishers/subscribers inside one node," [Accessed 12-05-2024]. [Online]. Available: https://roboticsbackend.com/what-is-a-ros-topic/#Multiple_publisherssubscribers_inside_one_node

[22] ——, "What is a ROS Service?" [Accessed 12-05-2024]. [Online]. Available: https://roboticsbackend.com/what-is-a-ros-service/

[23] "Understanding actions - ROS 2 Documentation," [Accessed 12-05-2024]. [Online]. Available: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html

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

[25] R. K. Megalingam, A. Rajendraprasad, and S. K. Manoharan, "Comparison of planned path and travelled path using ros navigation stack," in *2020 International Conference for Emerging Technology (INCET)*. IEEE, 2020, pp. 1–6.

[26] F. Schena, "Development of an automated benchmark for the analysis of nav2 controllers," Ph.D. dissertation, Politecnico di Torino, 2024.

[27] M. De Rose, "Lidar-based dynamic path planning of a mobile robot adopting a costmap layer approach in ros2," Ph.D. dissertation, Politecnico di Torino, 2021.

[28] M. You, "Enabling autonomous multi-floor navigation for robots in ros2 using behavior trees," Ph.D. dissertation, Politecnico di Torino, 2023.

[29] C. Barbara, "Path planning algorithm for an autonomous air sanitizing mobile robot in indoor scenarios," Ph.D. dissertation, Politecnico di Torino, 2023.

[30] OpenRobotics, "TurtleBot 4 Navigator · User Manual — turtlebot.github.io," [Accessed 29-05-2024]. [Online]. Available: https://turtlebot.github.io/turtlebot4-user-manual/tutorials/turtlebot4_navigator.html

[31] V. Mayellaro, "Person-aware autonomous navigation for an indoor sanitizing robot in ros2," Ph.D. dissertation, Politecnico di Torino, 2022.

[32] D. V. Lu, D. Hershberger, and W. D. Smart, "Layered costmaps for context-sensitive navigation," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014, pp. 709–715.

[33] F. Maresca and A. Ragazzo, "Ros-based autonomous navigation and object recognition for a mobile manipulator operating in a warehouse environment," Ph.D. dissertation, Politecnico di Torino, 2022.

[34] P. Vanella, "Implementation of ros-based multi-agent slam centralized and decentralized approaches," Ph.D. dissertation, Politecnico di Torino, 2023.

[35] J. Leonard and H. Durrant-Whyte, "Simultaneous map building and localization for an autonomous mobile robot," in *Proceedings IROS '91:IEEE/RSJ International Workshop on Intelligent Robots and Systems '91*, 1991, pp. 1442–1447 vol.3.

[36] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Transactions on Robotics*, vol. 32, no. 6, pp. 1309–1332, 2016.

[37] I. Z. Ibragimov and I. M. Afanasyev, "Comparison of ros-based visual slam methods in homogeneous indoor environment," in *2017 14th Workshop on Positioning, Navigation and Communications (WPNC)*. IEEE, 2017, pp. 1–6.

[38] M. Filipenko and I. Afanasyev, "Comparison of various slam systems for mobile robot in an indoor environment," in *2018 International Conference on Intelligent Systems (IS)*, 2018, pp. 400–407.

[39] S. Kohlbrecher, O. von Stryk, J. Meyer, and U. Klingauf, "A flexible and scalable slam system with full 3d motion estimation," in *2011 IEEE International Symposium on Safety, Security, and Rescue Robotics*, 2011, pp. 155–160.

[40] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2d lidar slam," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 1271–1278.

[41] G. Klein and D. Murray, "Parallel tracking and mapping for small ar workspaces," in *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, 2007, pp. 225–234.

[42] M. Labbé and F. Michaud, "Online global loop closure detection for large-scale multi-session graph-based slam," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014, pp. 2661–2666.

[43] J. Engel, V. Koltun, and D. Cremers, "Direct sparse odometry," 2016.

[44] J. Engel, T. Schöps, and D. Cremers, "Lsd-slam: Large-scale direct monocular slam," in *European conference on computer vision*. Springer, 2014, pp. 834–849.

[45] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos, "Orb-slam: a versatile and accurate monocular slam system," *IEEE transactions on robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

[46] J. Engel, V. Koltun, and D. Cremers, "Direct sparse odometry," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 3, pp. 611–625, 2017.

[47] T. Pire, T. Fischer, G. Castro, P. De Cristóforis, J. Civera, and J. J. Berlles, "S-ptam: Stereo parallel tracking and mapping," *Robotics and Autonomous Systems*, vol. 93, pp. 27–42, 2017.

[48] S. Macenski and I. Jambrecic, "Slam toolbox: Slam for the dynamic world," *Journal of Open Source Software*, vol. 6, no. 61, p. 2783, 2021.

[49] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis, "Deep learning for computer vision: A brief review," *Computational intelligence and neuroscience*, vol. 2018, 2018.

[50] M. Kulkarni, P. Junare, M. Deshmukh, and P. P. Rege, "Visual slam combined with object detection for autonomous indoor navigation using kinect v2 and ros," in *2021 IEEE 6th international conference on computing, communication and automation (ICCCA)*. IEEE, 2021, pp. 478–482.

[51] P. Adarsh, P. Rathi, and M. Kumar, "Yolo v3-tiny: Object detection and recognition using one stage improved model," in *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2020, pp. 687–694.

[52] P. Jiang, D. Ergu, F. Liu, Y. Cai, and B. Ma, "A review of yolo algorithm developments," *Procedia computer science*, vol. 199, pp. 1066–1073, 2022.

[53] E. G. Vourkos, E. Toulkeridou, A. Kourris, R. J. Ros, E. G. Christoforou, N. Ramdani, and A. S. Panayides, "Safe robot navigation in indoor healthcare workspaces," in *International Conference on Computer Analysis of Images and Patterns*. Springer, 2023, pp. 56–64.

[54] M. Mateas and A. Stern, "Façade: An experiment in building a fully-realized interactive drama," in *Game developers conference*, vol. 2. Citeseer, 2003, pp. 4–8.

[55] D. Isla, "Handling complexity in the halo 2 ai," in *Game Developers Conference*, vol. 12, 2005.

[56] C. Hecker, "My liner notes for spore: Spore behavior tree docs," 2009, [Accessed: 2024-05-20]. [Online]. Available: https://www.chrishecker.com/My_Liner_Notes_for_Spore/Spore_Behavior_Tree_Docs

[57] A. J. Champandard, "Behavior trees for next-gen game ai," in *Game Developers Conference, Audio Lecture*, December 2007.

[58] AurynRobotics, "Nodes Library — BehaviorTree.CPP — behaviortree.dev," [Accessed 29-05-2024]. [Online]. Available: https://www.behaviortree.dev/docs/category/nodes-library

[59] R. Bernardo, J. M. Sousa, M. A. Botto, and P. J. Gonçalves, "A novel control architecture based on behavior trees for an omni-directional mobile robot," *Robotics*, vol. 12, no. 6, p. 170, 2023.

[60] OpenNavigation, "ComputePathToPose - Nav2 1.0.0 documentation — docs.nav2.org," [Accessed 29-05-2024]. [Online]. Available: https://docs.nav2.org/configuration/packages/bt-plugins/actions/ComputePathToPose.html

[61] ——, "Introduction To Nav2 Specific Nodes - Nav2 1.0.0 documentation — docs.nav2.org," [Accessed 29-05-2024]. [Online]. Available: https://docs.nav2.org/behavior_trees/overview/nav2_specific_nodes.html

[62] ——, "Navigation Plugins - Nav2 1.0.0 documentation — docs.nav2.org," [Accessed 29-05-2024]. [Online]. Available: https://docs.nav2.org/plugins/index.html#behaviors

[63] AurynRobotics, "Sequences — BehaviorTree.CPP — behaviortree.dev," https://www.behaviortree.dev/docs/nodes-library/sequencenode/, [Accessed 29-05-2024].

[64] ——, "Fallbacks — BehaviorTree.CPP — behaviortree.dev," [Accessed 29-05-2024]. [Online]. Available: https://www.behaviortree.dev/docs/nodes-library/fallbacknode/

[65] OpenNavigation, "RecoveryNode - Nav2 1.0.0 documentation — docs.nav2.org," https://docs.nav2.org/configuration/packages/bt-plugins/controls/RecoveryNode.html, [Accessed 29-05-2024].

[66] M. Colledanchise and L. Natale, "On the implementation of behavior trees in robotics," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5929–5936, 2021.

[67] J. Stüber, C. Zito, and R. Stolkin, "Let's push things forward: A survey on robot pushing," *Frontiers in Robotics and AI*, vol. 7, p. 8, 2020.

[68] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ international conference on intelligent robots and systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3. Ieee, 2004, pp. 2149–2154.

[69] S. P. Thale, M. M. Prabhu, P. V. Thakur, and P. Kadam, "Ros based slam implementation for autonomous navigation using turtlebot," in *ITM Web of conferences*, vol. 32. EDP Sciences, 2020, p. 01011.

[70] H. Maghfiroh and H. P. Santoso, "Online navigation of self-balancing robot using gazebo and rviz," *Journal of Robotics and Control (JRC)*, vol. 2, no. 5, 2021.

[71] C. Box, "Carboard box dataset," https://universe.roboflow.com/carboard-box/carboard-box, sep 2023, visited on 2024-05-30. [Online]. Available: https://universe.roboflow.com/carboard-box/carboard-box

[72] moveUp care, "Lowerbodydetection dataset," https://universe.roboflow.com/moveup-care/lowerbodydetection, apr 2023, visited on 2024-05-30. [Online]. Available: https://universe.roboflow.com/moveup-care/lowerbodydetection

Appendix A

Odometry Service

```
import rclpy
from rclpy.node import Node
from nav_msgs.msg import Odometry
from std_srvs.srv import SetBool


class YoloService(Node):
    def __init__(self):
        super().__init__('yolo_service')
        self.srv = self.create_service(SetBool, 'get_odom', self.
    get_odom_callback)
        self.position = None
        qos_policy = rclpy.qos.QoSProfile(reliability=rclpy.qos.
    ReliabilityPolicy.BEST_EFFORT,
                                          history=rclpy.qos.HistoryPolicy
    .KEEP_LAST,
                                          depth=1)
        self.subscription = self.create_subscription(
            Odometry,
            'odom',
            self.listener_callback,
            qos_profile = qos_policy)
        self.subscription  # prevent unused variable warning

    def listener_callback(self, msg):
        if (self.position and (self.position.x != msg.pose.pose.position.
    x or self.position.y != msg.pose.pose.position.y)):
```

```
24              print(str(msg.pose.pose.position.x)+" "+str(msg.pose.pose.
    position.y))
25          self.position = msg.pose.pose.position
26
27      def get_odom_callback(self, request, response):
28          if self.position:
29              response.message = f'Position: x={self.position.x}, y={self.
    position.y}'
30              response.success = True
31              self.get_logger().info('Requested. Replied "%s"' % response.
    message)
32          else:
33              response.message = 'Position not available'
34              response.success = False
35              self.get_logger().info('Requested. Replied "%s"' % response.
    message)
36          return response
37
38  def main(args=None):
39      rclpy.init(args=args)
40      yolo_service = YoloService()
41      rclpy.spin(yolo_service)
42      yolo_service.destroy_node()
43      rclpy.shutdown()
44
45  if __name__ == '__main__':
46      main()
```

Listing A.1: Odometry Service

Appendix B

YOLO Service

```
import rclpy
from rclpy.node import Node
from cv_bridge import CvBridge
from sensor_msgs.msg import Image
from std_msgs.msg import Bool
import cv2
from std_srvs.srv import SetBool
from inference import get_model
import supervision as sv


class YoloService(Node):
    def __init__(self):
        super().__init__('yolo_service')
        self.srv = self.create_service(SetBool, 'get_yolo_state', self.
    get_yolo_state_callback)
        self.last_states = [False,False,False,False,False]
        self.subscription = self.create_subscription(
            Image,
            '/oakd/rgb/preview/image_raw',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning
        self.publisher = self.create_publisher(Bool, 'yolo', 10)
        self.model = get_model(model_id="lowerbodydetection/1")

    def listener_callback(self, msg):
        bridge = CvBridge()
```

```
28          img = bridge.imgmsg_to_cv2(msg, "bgr8")
29          results = self.model.infer(img)
30          detections = sv.Detections.from_inference(results[0].dict(
    by_alias=True, exclude_none=True))
31          msg = Bool()
32
33          for i in range(0,len(self.last_states)-1):
34               self.last_states[i] = self.last_states[i+1]
35          if detections.__len__() > 0:
36              self.last_states[-1] = False
37              msg.data = False
38          else:
39              self.last_states[-1] = True
40              msg.data = True
41          self.publisher.publish(msg)
42          self.get_logger().info('Publishing: "%s"' % msg.data)
43
44      def get_yolo_state_callback(self, request, response):
45          morethan3 = sum(self.last_states) >= 3
46          self.get_logger().info('Requested. Replied "%s"' % str(morethan3)
    )
47          response.success = morethan3
48          response.message = str(morethan3)  # Send state as a message (
    string)
49          return response
50
51  def main(args=None):
52      rclpy.init(args=args)
53      yolo_service = YoloService()
54      rclpy.spin(yolo_service)
55      yolo_service.destroy_node()
56      rclpy.shutdown()
57
58  if __name__ == '__main__':
59      main()
```

Listing B.1: YOLO Service

Appendix C

MoveForwards/MoveBackwards BT Node

```
#include "rclcpp/rclcpp.hpp"
#include "behaviortree_cpp_v3/action_node.h"
#include "geometry_msgs/msg/twist.hpp"
#include "std_srvs/srv/set_bool.hpp"
#include <sstream>
#include <string>

class MoveForward : public BT::SyncActionNode
{
public:
    MoveForward(const std::string &name, const BT::NodeConfiguration &
    config)
        : BT::SyncActionNode(name, config)
    {
        node_ = rclcpp::Node::make_shared("move_forward_bt_node");
        publisher_ = node_->create_publisher<geometry_msgs::msg::Twist>("
    cmd_vel", 10);
        client_ = node_->create_client<std_srvs::srv::SetBool>("get_odom"
    );
    }

    static BT::PortsList providedPorts()
    {
        return {BT::InputPort<float>("speed"), BT::InputPort<float>("dist
    ")};
```

```
22      }
23
24      virtual BT::NodeStatus tick() override
25      {
26
27          if (!client_ready_)
28          {
29              client_ready_ = client_->wait_for_service(std::chrono::
    seconds(1));
30              if (!client_ready_)
31              {
32                  RCLCPP_ERROR(node_->get_logger(), "Service not available
    after waiting");
33                  return BT::NodeStatus::FAILURE;
34              }
35          }
36
37          std::ostringstream oss;
38
39          if (!initial_position_set_)
40          {
41              RCLCPP_INFO(node_->get_logger(), "Initial position requested"
    );
42
43              auto request = std::make_shared<std_srvs::srv::SetBool::
    Request>();
44              auto result = client_->async_send_request(request);
45
46              // Spin until the future is resolved
47              rclcpp::spin_until_future_complete(node_, result);
48
49              RCLCPP_INFO(node_->get_logger(), "Service replied");
50              auto res = result.get();
51              if (res->success)
52              {
53                  initial_position_set_ = true;
54
55                  sscanf(res->message.c_str(), "Position: x=%lf, y=%lf", &
    initial_x_;, &initial_y_);
56
57                  oss << "Initial Position: " << initial_x_ << " " <<
    initial_y_;
```

```
58                 RCLCPP_INFO(node_->get_logger(), "%s", oss.str().c_str())
   ;
59             }
60             else
61             {
62                 RCLCPP_ERROR(node_->get_logger(), "Unable to retrieve
   initial position");
63                 return BT::NodeStatus::FAILURE;
64             }
65         }
66
67         float speed;
68         getInput("speed", speed);
69         float dist;
70         getInput("dist", dist);
71
72         auto start_time = node_->now();
73         geometry_msgs::msg::Twist move_cmd;
74         move_cmd.linear.x = speed; // speed value
75
76         double target_distance = dist; // meters to move forward
77         double current_distance = 0.0;
78
79         bool moved = false;
80
81         double x1 = 0.0;
82         double x2 = 0.0;
83
84         double y1 = 0.0;
85         double y2 = 0.0;
86
87         int i = 0;
88
89         while (current_distance < target_distance)
90         {
91             if ((node_->now() - start_time).seconds() > dist / std::abs(
   speed) + 5)
92             { // Timeout for safety
93                 move_cmd.linear.x = 0;
94                 publisher_->publish(move_cmd);
95                 initial_position_set_ = false;
96                 moved = false;
```

```
97                  RCLCPP_ERROR(node_->get_logger(), "TIMEOUT.");
98                  return BT::NodeStatus::FAILURE;
99              }
100
101             publisher_->publish(move_cmd);
102             rclcpp::spin_some(node_);
103             std::this_thread::sleep_for(std::chrono::milliseconds(10));
    // short sleep to yield CPU
104
105             if (i % 100 == 0)
106             {
107                 RCLCPP_INFO(node_->get_logger(), "New position requested"
    );
108             }
109
110             auto request = std::make_shared<std_srvs::srv::SetBool::
    Request>();
111             auto result = client_->async_send_request(request);
112
113             // Spin until the future is resolved
114             rclcpp::spin_until_future_complete(node_, result);
115
116             auto res = result.get();
117             auto new_x_ = 0.0f;
118             auto new_y_ = 0.0f;
119
120             if (res->success)
121             {
122                 sscanf(res->message.c_str(), "Position: x=%lf, y=%lf", &
    new_x_, &new_y_);
123
124                 current_distance = std::sqrt(std::pow(new_x_ - initial_x_
    , 2) + std::pow(new_y_ - initial_y_, 2));
125
126                 if (current_distance > 0 && !moved)
127                 {
128                     moved = true;
129                 }
130
131                 if (i % 100 == 0)
132                 {
133                     x2 = x1;
```

```
134                     x1 = new_x_;
135                     y2 = y1;
136                     y1 = new_y_;
137 
138                     std::ostringstream oss1;
139                     oss1 << "New Position: " << new_x_ << " " << new_y_;
140                     RCLCPP_INFO(node_->get_logger(), "%s", oss1.str().
    c_str());
141                     oss1.str("");
142 
143                     oss1 << "Distance travelled: " << current_distance;
144                     RCLCPP_INFO(node_->get_logger(), "%s", oss1.str().
    c_str());
145                     oss1.str("");
146 
147                     double distance_last_ticks = std::sqrt(std::pow(x2 -
    x1, 2) + std::pow(y2 - y1, 2));
148                     oss1 << "Distance travelled the last 100 ticks: " <<
    distance_last_ticks;
149                     RCLCPP_INFO(node_->get_logger(), "%s", oss1.str().
    c_str());
150                     oss1.str("");
151 
152                     if (distance_last_ticks < 0.02 && moved)
153                     {
154                         initial_position_set_ = false;
155                         moved = false;
156                         RCLCPP_ERROR(node_->get_logger(), "The robot did
    not move the distance it should.");
157                         return BT::NodeStatus::FAILURE;
158                     }
159                 }
160             }
161             else
162             {
163                 RCLCPP_ERROR(node_->get_logger(), "Unable to retrieve new
     position");
164                 return BT::NodeStatus::FAILURE;
165             }
166 
167             i++;
168             // start_i++;
```

```
169         }
170
171         // Stop the robot
172         move_cmd.linear.x = 0;
173
174         publisher_->publish(move_cmd);
175
176         initial_position_set_ = false;
177         moved = false;
178         RCLCPP_INFO(node_->get_logger(), "Returning SUCCESS");
179         return BT::NodeStatus::SUCCESS;
180     }
181
182 private:
183     rclcpp::Node::SharedPtr node_;
184     rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr publisher_;
185     rclcpp::Client<std_srvs::srv::SetBool>::SharedPtr client_;
186     bool client_ready_;
187     double initial_x_;
188     double initial_y_;
189     bool initial_position_set_;
190 };
191
192 #include "behaviortree_cpp_v3/bt_factory.h"
193 BT_REGISTER_NODES(factory)
194 {
195     BT::NodeBuilder builder = [](const std::string &name, const BT::
    NodeConfiguration &config)
196     {
197         return std::make_unique<MoveForward>(name, config);
198     };
199     factory.registerBuilder<MoveForward>("MoveForward", builder);
200 }
```

Listing C.1: MoveForwards BT Node

Appendix D

CheckYOLO BT Node

```
#include "rclcpp/rclcpp.hpp"
#include "behaviortree_cpp_v3/condition_node.h"
#include "std_srvs/srv/set_bool.hpp"

class YoloActionListener : public BT::ConditionNode
{
public:
    YoloActionListener(const std::string &name, const BT::
    NodeConfiguration &config)
        : BT::ConditionNode(name, config), client_ready_(false)
    {
        node_ = rclcpp::Node::make_shared("check_yolo_condition");
        client_ = node_->create_client<std_srvs::srv::SetBool>("
    get_yolo_state");
    }

    BT::NodeStatus tick() override
    {
        if (!client_ready_)
        {
            client_ready_ = client_->wait_for_service(std::chrono::
    seconds(1));
            if (!client_ready_)
            {
                RCLCPP_ERROR(node_->get_logger(), "Service not available
    after waiting");
                return BT::NodeStatus::FAILURE;
            }
```

```
25          }
26          RCLCPP_INFO(node_->get_logger(), "Service requested");
27
28          auto request = std::make_shared<std_srvs::srv::SetBool::Request
    >();
29          auto result = client_->async_send_request(request);
30
31          // Spin until the future is resolved
32          rclcpp::spin_until_future_complete(node_, result);
33
34          auto res = result.get();
35          if (res->success)
36          {
37              return BT::NodeStatus::SUCCESS;
38          }
39          else
40          {
41              return BT::NodeStatus::FAILURE;
42          }
43      }
44
45      static BT::PortsList providedPorts()
46      {
47          return {};
48      }
49
50  private:
51      rclcpp::Node::SharedPtr node_;
52      rclcpp::Client<std_srvs::srv::SetBool>::SharedPtr client_;
53      bool client_ready_;
54  };
55
56  #include "behaviortree_cpp_v3/bt_factory.h"
57  BT_REGISTER_NODES(factory)
58  {
59      factory.registerNodeType<YoloActionListener>("YoloAction");
60  }
```

Listing D.1: CheckYOLO BT Node

Appendix E

Custom Behavior Tree

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <RecoveryNode number_of_retries="4">
      <Sequence name="MainNavigationFlow">
        <ComputePathToPose goal="{goal}" path="{path}" planner_id="{selected_planner}"/>
        <RecoveryNode number_of_retries="1" name="FollowPathWithRecovery">
          <FollowPath path="{path}" controller_id="{selected_controller}" />
          <Sequence name="ObstacleHandling">
            <Fallback name="CheckYolo">
              <Sequence>
                <YoloAction name="ObstacleDetectionCheck"/>
                <MoveForward speed="0.3" dist="2.0"/>
              </Sequence>
              <MoveForward speed="-0.3" dist="1.0"/>
            </Fallback>
            <ClearEntireCostmap name="ClearLocalCostmap" service_name="local_costmap/clear_entirely_local_costmap"/>
            <ComputePathToPose goal="{goal}" path="{path}" planner_id="{selected_planner}"/>
          </Sequence>
        </RecoveryNode>
      </Sequence>
      <ReactiveFallback name="RecoveryFallback">
        <GoalUpdated/>
        <RoundRobin name="RecoveryActions">
```

```
24          <Sequence name="ClearingActions">
25            <ClearEntireCostmap name="ClearLocalCostmap-Subtree"
    service_name="local_costmap/clear_entirely_local_costmap"/>
26            <ClearEntireCostmap name="ClearGlobalCostmap-Subtree"
    service_name="global_costmap/clear_entirely_global_costmap"/>
27          </Sequence>
28          <Spin spin_dist="1.57"/>
29          <Wait wait_duration="5"/>
30          <BackUp backup_dist="0.15" backup_speed="0.025"/>
31        </RoundRobin>
32      </ReactiveFallback>
33    </RecoveryNode>
34  </BehaviorTree>
35 </root>
```

Listing E.1: Custom Behavior Tree