POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

MULTIAGENT SYSTEM FOR SMART META-SENSOR FRAMEWORK

Supervisors

Prof. MARINA INDRI

Dr. PANGCHENG DAVID CEN CHENG

Candidate

FABIO GIUSEPPE RIU

JULY 2024

Summary

In Industry 4.0 the AMRs (Autonomous Mobile Robots) have a key role. The AMRs need to share the same spaces with humans but only some of the AMRs can detect and recognize them through specific sensor data. Some types of sensors perform better than others in this context, but they might increase the cost of those robots. Robots equipped with only LIDAR (Light Detection And Ranging) perform well for indoor SLAM but still have limitations regarding obstacles that are below the height of the LIDAR. However, if they are also equipped with an RGB-D sensor, they offer a more limited but more detailed view. ROS (Robot Operating System) represents the state-of-the-art framework for the coordination and management of robots. ROS offers a Navigation Stack commonly used by the majority of commercial AMRs. The ROS Navigation Stack is fully modular and every manufacturer tunes this package depending on the characteristics of its own robot. The key idea for this thesis is to create a ROS framework to share information between different robot types of robots, allowing them to assist each other in the early identification of obstacles. This will enable robots with limited vision capabilities to perceive obstacles that they would not be able to detect on their own. The main information that needs to be shared is the robot's position and the dynamic obstacles independently of the implementation and algorithms used by the various robots. In this framework, each robot shares its real-time position with a node called `robot_info_manager_node`, which is responsible for forwarding this information to all other robots. This ensures that each robot can consider the positions of others during the path planning phases. This node not only receives and shares the positions of all the robots but also manages their`local_costmap`. These costmaps represent the real-time dynamic view of each robot and what its sensors perceive. To do this, each robot is registered with this node through a configuration file, and each robot is assigned a name. Using a service, each robot can send an information request to the node, specifying who is making the request. The node will respond by sending all the `local_costmap` and the positions of the other robots, excluding the requesting robot itself. The service is called through a new layer `global_costmap` called `positioning_layer`, which all robots that want to receive information from other robots must add to the list of layers in the

`global_costmap`, using the configuration file of the `move_base` node. This layer will be responsible for marking cells as occupied where various robots are positioned to avoid collisions between robots, and for copying their `local_costmap` to the correct position, including obstacles that, for structural reasons, cannot be observed. The only constraints are that the robots share the same map and the resolution of `local_costmap` and `global_costmap` are the same. This framework has been tested using a LoCoBot WX250S from Trossen Robotics and a Turtlebot3 Burger from Robotis, both in simulation using Gazebo and in a real-world environment in a laboratory.

Table of Contents

List of Tables

List of Figures

Acronyms

AMR
Autonomous Mobile Robot

AGV
Automated Guided Vehicle

SLAM
Simultaneous Localization and Mapping

ROS
Robot Operating System

LiDAR
Light Detection and Ranging

RGB-D
Red Green Blue-Depth

MAS
Multi-Agent Systems

AI
Artificial Intelligence

KF
Kalman Filter

EKF
Extended Kalman Filter

AKF

Adaptive Kalman Filter

UKF

Unscented Kalman Filter

PF

Particle Filters

SMC

Sequential Monte Carlo

RBPF

Rao-Blackwellized Particle Filter

PCL

Point Cloud Library

CPS

Cyber Physical Systems

MES

Manufacturing Execution System

CPHS

Cyber Physical Human Systems

HRI

Human-Robot Interaction

HRP

Human-Robot Perception

LIDAR

Light Detection and Ranging

RSSI

Received Signal Strength Indicator

MCL

Monte Carlo Localization

TOA

Time of Arrival

TDOA

Time Difference of Arrival

GUI

Graphical User Interface

ANSI

American National Standards Institute

HOG

Histogram of Oriented Gradients

ITSDF

Industrial Truck Standards Development Foundation

FPS

Frames Per Second

MMORPG

Massively Multiplayer Online Role-Playing Game

OSRF

Open Source Robotics Foundation

AMCL

Adaptive Monte Carlo Localization

RTAB-Map

Real-Time Appearance-Based Mapping

URDF

Unified Robot Description Format

PGM

Portable Gray Map

API

Application Programming Interface

TF

Transformation Frames

XML

eXtensible Markup Language

IDE

Integrated Development Environment

Q-Learning

Quality Learning

Nav2

Navigation stack 2

RViz

ROS Visualization

TF tree

Transformation tree

PC

Personal Computer

Chapter 1

Introduction

Industry 4.0 introduces the concept of robotic machines that are capable of autonomously moving in industrial environments shared with human operators. These robots, called Autonomous Mobile Robots (AMRs, shown in Figure 1.1) have the ability to collaborate with and assist humans during work, ensuring their safety. The main difference with respect to a traditional Automated Guided Vehicle (AGV) is that the latter follows only a pre-calculated path and its capabilities to avoid obstacles are very limited. In the best-case scenario, the robot stops and waits until the obstacle is removed, increasing processing times. A set of sensors placed in the robots gives AMRs the capability to address this problem more efficiently. These robots have the capability to perform simultaneous localization and mapping (SLAM), allowing them to build and update a map of their environment in real time. Additionally, AMRs have advanced obstacle detection and avoidance systems that enable them to recalculate their route when encountering an obstacle, ensuring they can reach their destination without collisions. AMRs can vary significantly from each other based on the quantity and types of sensors and their architecture, leading to varying levels of intelligence and adaptability. Some AMRs have the capability to detect and navigate around small obstacles with high precision. For example, a 360-degree LiDAR sensor placed at a certain height may not detect obstacles below that level, rendering them invisible to the robot. Consequently, different AMRs may offer different levels of accuracy and performance depending on the sensors used and their placement. To manage the complexity of integrating various sensors and achieving robust performance, many AMRs utilize the Robot Operating System (ROS) [2]. ROS provides a flexible framework for writing robot software, enabling the integration of multiple sensors and algorithms. This allows developers to easily implement SLAM, obstacle detection, and path planning functionalities, ensuring that AMRs can operate efficiently and safely in dynamic environments. ROS is a framework created for the development of robotic applications and offers a system called the ROS Navigation [3] that includes a collection

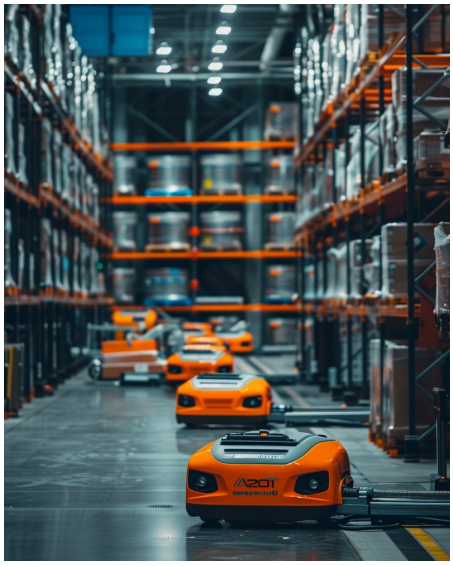


Figure 1.1: Example of AGVs inside a warehouse [1]

of packages that provide essential algorithms for autonomous navigation, such as localization, map building, path planning, and obstacle avoidance. By utilizing the Navigation Stack, developers can leverage pre-built components and tools to create sophisticated navigation systems, allowing AMRs to effectively navigate and adapt to complex, ever-changing environments. Generally, the ROS Navigation Stack can vary significantly from one robot to another, but its structure remains consistent across different implementations. This consistency ensures that developers can apply the same core principles and methods, even when customizing the stack to fit the specific needs and capabilities of different robots. Usually, a node provides a map, which is used to construct a `global_costmap`. This costmap is employed by a specific algorithm called `global_planner` to create a preliminary path to reach the desired goal. Another map is created during the execution of the plan using data received at that moment, the `local_costmap`, which represents the updated situation of the robot in real time. This costmap is employed by the `local_planner` algorithm to detect obstacles with sensors and avoid them during movement. The `global_costmap` is also used for localizing the robot in the map, using specific algorithms that attempt to determine the robot's position based on information provided by sensors and the data provided by this costmap. The methodology remains often the same over time. The aspects that can vary include the specific algorithms (`local_planner` and `global_planner`), the methods used to construct the `local_costmap` and `global_costmap`, and the types and number of sensors used by each individual robot. This thesis aims at proposing a method to make heterogeneous robots share relevant information about the environment, and

how this information can be used to improve the overall performance of navigation. Specifically, the thesis investigates how a robot with more sensors and a greater detection capability can assist other robots with limited vision of the environment, while also being able to anticipate the detection of obstacles before they become visible to them. The target environment of this application is indoor locations like warehouses. In this case of study, the two specific robots are a Locobot WX250, equipped with a 360-degree LiDAR and an RGB-D camera, and a Turtlebot3 Burger, equipped with only a 360-degree LiDAR. The two LiDARs are placed at different heights; for example, the Turtlebot's LiDAR is positioned at a height of about 180mm and cannot detect obstacles lower than this height. In contrast, the LiDAR on the Locobot is positioned at a height of 610mm, but it can detect obstacles of lower heights due to the RGB-D camera. So, the goal is to build a generic framework for sharing information, independently by the robot's type, sensors and algorithms. The constraint of this framework is that each robot uses a `costmap_2d` type for both local and global cost maps, and the resolution of these maps is uniform across all robots. Additionally, the robots share a common map.

1.1 Thesis structure

In Chapter 2, a review of the current state of the art is provided. Chapter 3 covers the ROS navigation stack package, including state-of-the-art algorithms for local and global planning. Chapter 4 describes the two robots used in the thesis: the Locobot WX250 and the Turtlebot3 Burger. Chapter 5 outlines the necessary steps to deploy and use multiple robots within the same ROS network.

In Chapter 6, the `Robot_position_info_manager` node is discussed, which is responsible for gathering information from various robots and their configuration requirements. Chapter 7 focuses on the positioning layer, which retrieves information from the node described in the previous chapter and processes it in a way that is useful for the robots.

Afterward, Chapter 8 details all software configurations and the choices of various algorithms used in the project. Chapter 9 presents the experiments conducted both in simulation and in the laboratory, along with the results obtained and the challenges encountered.

The final chapter, Chapter 10, discusses the achieved results and suggests potential future developments for the application.

Chapter 2

State of the art

In the field of robotics, Multi-Agents Systems (MASs) and intelligent agents play crucial roles in solving complex problems. An intelligent agent, as cited in [4], is a physical robot or virtual software entity capable of autonomous action in an environment to achieve specific goals. There are various types of intelligent agents designed to interact with their environment in distinct ways. Reactive agents are programmed to respond immediately to changes in their surroundings, focusing on sensing and acting in real time. Deliberative agents engage in more complex decision-making processes and can plan actions without external triggers. Hybrid agents represent a blend of both reactive and deliberative approaches. MASs consist of a collection of intelligent agents that may share common or conflicting goals, working together to achieve overall system objectives. MAS is a comprehensive term that encompasses a wide range of cases. The agents can be different from each other (heterogeneous) or can be all the same (homogeneous) and the communication structure can be centralized, hierarchical, decentralized or hybrid. Furthermore, another distinction can be made based on whether the agents are physical or not. In this first case, the correct term of this problem is Multi-robot system. According to [5], the problem in this thesis can be defined as a Heterogeneous Communicating Multi-Robot System. The analysis in [4] describes and classifies most of the existing work into four different levels of automation based on how many steps can be solved autonomously. The steps are the following:

- **decomposition of a task in sub-tasks**: Breaking down complex tasks into simpler, manageable sub-tasks based on the capabilities and requirements of the robots involved.

- **identify the agent coalition**: Forming teams or coalitions of agents that possess complementary skills and resources necessary to accomplish the sub-tasks efficiently.

- **assign all sub-tasks to an agent for execution**: Allocating each sub-task to the appropriate agent or coalition for execution, considering factors such as workload balance and specialization.

- **execution/planning/control the task**: Implementing the sub-tasks through a sequence of actions tailored to the specific requirements of each task type.

The greater the number of steps can resolve autonomously without human intervantion, the higher is the level of automation. The authors in [6] show the main differences between the AMRs and the AGVs and discuss the flow for constructing an AMR. Both ARMs and AGVs have the capability to navigate autonomously in an environment. However, the principal difference is that AMRs can move in the environment without any intervention by humans, introducing the concept of "autonomy" in addition to "automation". AGVs, on the other hand, often can only move along fixed paths and have critical issues in dynamic environments when obstacles appear. To do that, AMRs are equipped with a larger number of sensors and actuators that are controlled through a specific software that tries to observe what the robot sees through the sensors and find "the best" path, which is translated into actuator signals. ROS is indeed an example of this type of middleware software.

SLAM is the key for AMRs when they operate in an unknown environment. Essentially, SLAM involves a robot building a map of its surroundings while simultaneously keeping track of its own location within that map. This dual process is critical because it allows the robot to understand where it is and how to move through its environment without prior knowledge. Exists many approaches for solving SLAM [7] among which we can mention:

- **Kalman-based approaches**: There are many SLAM methods based on various Kalman Filter (KF). The Extended KF (EKF) is a common variant used in SLAM, handling nonlinearities by approximating them with linear models around current estimates. The Adaptive KF (AKF) adjusts filter parameters in real time, enhancing performance in varying or uncertain environments. For more accuracy in nonlinear systems, the Unscented KF (UKF) avoids linearization by using a set of representative points to capture nonlinearities more precisely.

- **Particle Filters**: Particle filters (PF), also known as Sequential Monte Carlo (SMC) methods, are advanced probabilistic techniques for robotics. They perform in handling nonlinearities and non-Gaussian noise distributions, offering flexibility in diverse environments. However, their computational demands increase with the state dimension and landmark count, limiting real-time use in complex scenarios.

- **FastSLAM**: FastSLAM integrates particle filters (PF) with Rao-Blackwellization, leveraging the Rao-Blackwellized Particles Filter (RBPF) for enhanced precision in robot localization and mapping. This approach utilizes PF to estimate the robot's trajectory while employing low-dimensional extended Kalman filters (EKF) to pinpoint landmark positions. FastSLAM excels in navigating non-Gaussian and nonlinear environments, though it necessitates higher computational resources in intricate scenarios. The ROS Gmapping package incorporates this type of algorithm [8].

- **Visual SLAM**: Vision-based SLAM has rapidly developed as a solution to the SLAM problem, leveraging cameras instead of laser scanners to reconstruct 3D maps of environments. Images provide rich feature information, enabling robots to perform a broader range of tasks. The advancement in visual sensors, including depth cameras and various types of visual sensors, has driven recent research and SLAM techniques toward utilizing cameras and visual sensory information extensively. The typical sensors used for this approach are monocular, RGB-D and stereo cameras. In [9] the authors propose an enhancement to Visual SLAM by fusing data from a LiDAR sensor and an RGB-D camera using the UKF. This strategy improves the fidelity and detail of maps with respect of using classic algorithms like cartographer or GMapping.

The choice of the algorithm depends largely on the characteristics of the environment and the sensors equipped on the robot. In indoor applications, a variety of sensors can be utilized to gather environmental information during SLAM [10]. Acoustic sensors are the cheapest ones available commercially. They have the advantage of being immune to darkness and transparency, making them reliable in various lighting conditions. However, their maximum range is limited to a few meters, and their accuracy ranges from 1% to 3% of the maximum depth. LiDAR sensors provide a wide field of view, up to 350 degrees, and offer a substantial depth range from 50 to 300 meters. These sensors are highly effective in creating detailed and accurate maps of the environment. Monocular cameras, which are standard RGB cameras, are also used for indoor applications. These sensors require data processing and very complex algorithms to extract useful information from the images they capture. Depth cameras can capture both color (RGB) and depth information of scenes. Various technological solutions can be adopted to do this [11]:

- Structured Light: Projects a known pattern (usually infrared) onto the scene and measures the deformation of this pattern to calculate depth. This one is similar to the Kinect.

- Time-of-Flight (ToF): Measures the time it takes for light to travel to an

object and back to the camera to calculate the distance.

- Stereo Vision: Uses two or more cameras at different locations to simulate human binocular vision and calculate depth by comparing the images.

All of these cameras need pre-processing before having a 3D image. For example, structured light requires time to recreate the 3D image from the projected points. Each of these sensors offers different advantages and limitations, making them suitable for various aspects of indoor SLAM applications. In fact, there are various SLAM algorithms that attempt to fuse data received from different sensors to improve accuracy and reliability

In Industry 4.0, it is important for humans to share the work environment with robots safely. Therefore, it is important that robots can detect and track humans in the environment; it can be useful to adopt a special policy for them. The difference between detection and tracking [12] is that detection consists only in finding a target, while tracking involves the robot following or tracking the target that was previously found using an algorithm from the detection category. The target detection problem can be divided into two main categories: static surveillance, where the sensors are fixed in the environment, and the mobile search, where the sensors can move in the environment. In static surveillance, the main problem is finding the minimum number of sensors and their positions to cover every part of the environment. Mobile search can be divided into three main categories based on the level of certainty of finding the target. The first category is the capture problem, where the worst-case scenario is considered. In this case, there are typically two approaches: one based on discretized graph representation and another based on continuous geometric representation. In the first approach, the main challenge is to represent the entire environment as a graph. The second approach provides less complexity, but each of these solutions results in centralized control. The second category is based on probabilistic search. In this case, the algorithm is based on a probability distribution over a grid map. The probability distribution represents the likelihood of finding the target in a given position. This approach is closer to real-world conditions. Within this category, patrolling can also be included, which has the same objective but follows a cyclic aspect, where the area is covered multiple times in a repetitive path. The third category is hunting, which has no search strategy and is based purely on randomness. In this approach, the robot moves randomly in the environment to find the target. Once the target is found, the second phase involves tracking or following it. Tracking involves only tracking the position of the target in the environment. Centralized approaches typically yield better results than decentralized ones, but they are less realistic to use in real environments. Once the target is traced, the robot stays near it and follows its movements. The case with more than one target is called the observation problem. The performance of multi-robot teams is better than one-on-one instances

in parallel.

Based on the type of sensors, there are various approaches for the detection problem. In [13], the authors compared a set of deep learning methods for human detection in embedded platforms (in this case they use NVIDIA Jetson). Human detection problems can be divided into two steps: object detection and object recognition (classification), but in the deep learning case, the problem can be solved using one single algorithm. For the authors, faster R-CNN and YOLO require large memory resources and in an embedded environment they would not work well. They use five models PedNet, Miltiped, SSD MobileNetV1, SSD MobileNetV2, SSD Inception V2. The camera resolution was reduced to 320x240 for real-time purposes. They tested the algorithms in various scenarios and evaluated them in terms of Precision Rate, Recall and F1score and in terms of FPS where:

$$PrecisionRate = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$Recall = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negative}}$$

$$F1score = 2 \times \frac{\text{Precision Rate } \times \text{Recall}}{\text{Precision Rate} + \text{Recall}}$$

Among those, SSD Inception V2 resulted to be the most accurate and one of the fastest. In [14] the authors compared various models for real-time vehicle type recognition. They considered R-CNN,Fast R-CNN and Faster R-CNN and concluded that the first one is not suitable for real-time applications. The main problem of YOLO is that it cannot catch small objects. The authors train Faster R-CNN, YOLO and SDD with a dataset containing about 2000 images of vehicles using a GeForce RTX 2080Ti. The results show that Faster R-CNN do not satisfy FPS for real-time operation. SDD is fast but have less accuracy with respect to other ones. Also, authors show that exists a limitation in the dataset, because the images of the vehicles are similar to each other. In [15], the authors show an excursus of principal models for human detection. The authors create a dataset of 10,972 images taken from a camera. They utilize two distinct computing platforms: one equipped with NVIDIA GeForce GTX 1080 Ti and the other with NVIDIA Jetson. They use 13 models including variants of YOLO, SDD, and RCNN. The result shows that TinyYOLO and SDD are the fastest models. Faster RCNN and RFCN are instead the most accurate models. The authors indicate YOLO V3-416 is the desirable model for human detection in embedded platforms. In [16], the authors aim to develop a standard for human detection on mobile service robots by integrating various detection methods within the ROS framework. The process starts with acquiring sensory data from 2D laser scans, RGB-D point clouds, and camera images. Specialized detectors identify people based on the

sensor type. These detections are then tracked over time to maintain consistent IDs for individuals, even during brief occlusions. Group tracking is done by estimating spatial and social relations. Multi-sensor fusion combines data from different sensors to improve robustness. The results are visualized using RViz and custom plugins, and test data are generated using a pedestrian simulator. Communication between pipeline stages is managed via ROS messages, facilitating easy interchangeability of components. For all these phases, the authors define a new standard ROS message for person detection. These messages are as generic as possible so that they are independent of the types of algorithms used at each level. The authors also show another human detection algorithm based on the type of sensors that the robot has. For the 2D laser scan, the authors mention the Leg detector that uses a subset of 2D features to track legs separately and performs optimally when the laser sensor is positioned close to the ground, typically below 0.5 meters in height. In the context of RGB-D sensors, the authors reference the Upper-Body detector and further mention the integration of the RGB-D detector from the Point Cloud Library (PCL). These detectors are tailored to detect human upper bodies and extract regions of interest from depth-based data. For monocular vision applications, the authors highlight the ground Histogram of Oriented Gradients (HOG) detector, which is GPU-accelerated and designed for medium- to far-range human detection. This detector requires a ground plane estimate to refine its detection capabilities in diverse environments. These algorithms are already configured to be used with their respective packages. The work presented in [17] discusses the evolving role of robots in smart factories, emphasizing their increasing collaboration with humans in shared workspaces. It introduces the concept of Cyber Physical Human Systems (CPHSs) as an evolution from Cyber Physical Systems (CPSs), highlighting the need for enhanced Human-Robot Interaction (HRI) and Human-Robot Perception (HRP) technologies. The importance of sensors, including vision (such as RGB-D and event-based cameras) and distance sensors (like LiDAR), is underscored for tasks ranging from collision avoidance to cooperative and collaborative actions between robots and humans. The authors explore the use of sensors in robotic systems for perceiving human operators in industrial environments. It focuses on vision sensors like RGB-D cameras for tasks involving human-robot collaboration, emphasizing their role in providing accurate image and depth information. Laser range finders are also highlighted for environment perception in mobile robots and manipulators, often combined with vision sensors for data fusion. They identify three levels of human-robot interaction:

- Coexistence: Basic interaction where robots and humans avoid obstacles but do not collaborate on tasks.
- Cooperative: Robots and humans perform different tasks but share a common objective. Robots perceive humans differently than mere obstacles.

- Collaboration: Humans and robots interact during task execution.

An example of a robot that can detect a human is shown in [18], where the authors combine different types of data from various sensors to reduce error effectively. Different sensors exist, some of which are more accurate but often more expensive. AMRs are considered as meta-sensors, designed to support AGVs. For this research work, the authors equipped the robot with a monocular camera and a 2D laser. Artificial Intelligence (AI) is used to detect humans. When a human is detected, the AMR sends information to other agents and defines a reaction rule. It is necessary to calibrate the camera with the laser. The first phase involves human detection. Information about human presence is obtained from the Sensors Synergy Center and the AGV coordination center, ensuring all AGVs and AMRs follow the same obstacle avoidance rules. The chosen neural network for this project is YOLO, which generates a bounding box around the detected human and saves the coordinates in a text file. This file is then published to a ROS topic. The laser sensor publishes information about a specific area on the image plane. A node monitors the human presence topic. Data synchronization is managed using the ApproximateTimeSynchronizer from the message filters ROS package. The positions are marked on the global map. To avoid human obstacles, Elastic Bands with three bands is employed, selecting the shortest one each time. In [19], a robot is used for monitoring designated areas. If a human enters in a specific area, the robot follows and tracks the human until the latter one is leaving the working area. To distinguish a human from a generic obstacle, it is used a face detector algorithm. The algorithm for patrolling the area is divided into two stages. The first stage is necessary to create the working map. The second stage identifies waypoints that represent the entire working environment using Rviz plugin. The robot moves from one way point to the next in a loop. A set of ROS nodes is deployed. One node processes the image from a camera to identify possible human presence based on well-known Viola-Jones face detection. When a human is found, the patrolling is stopped. In the considered case study a depth sensor camera is used to track and follow the human. The Controller node is the main hub. It switches from follower movement to goal-based navigation and vice versa. A Patrol node is responsible for waypoints handling. It is important to focus on the delay time used to switch robot's navigation mode. The authors highlight the necessity to take into account the various delays that may occur for real-time application. The first delay is the one used for face detection, the second is the time to switch between Patrol node and Follower node. Similarly, in [20], an algorithm is presented for a robot to track the position of the target and follow. The robot used in this paper is equipped with a Kinect sersor (deph-camera). The robot tries to draw the skeleton of a human. This method works only if the human stand in front of the robot. When the primary tracker (drawing the skeleton) looses the target, the tracking method changes using CAMshift. To estimate the position of human, it is used

an EKF. The authors impose a safe distance between the target and the robot. In [21], the authors improve the reliability of trajectory prediction of a worker to improve the maximum speed of mobile robots. To predict a human's destination intention in a manufacturing environment, authors use a naive Bayes classifier. The process involves integrating multiple data sources: the human's current position, the environmental map, and the manufacturing schedule from the Manufacturing Execution System (MES). It uses this classifier because it requires less data with respect to other neural classifiers. In the environment it is necessary to identify the Points-Of-Interest (POI) that are the nodes of the naive Bayes classifier. The output of this model is the set of POIs with the probability direction of the human direction. The number of POIs can be very large and it can change over time. For this reason, the authors abstract the model in only four points. Work-shift-end, break, new task and last/current task. With this model, it is possible to predict and calculate the probability transitions to and from these few states. In this way, it is not necessary to recalculate the transition probability for every change in the model and it takes less time to calculate the final destination probability. After using the model to give information about the probability destination it is possible to use a standard forward-planning method to obtain the predicted trajectory.

Using a multi-robot system can provide several advantages. As discussed previously, the robots can be different or of the same type. In paper [22], the authors present a multi-robot version of the MCL algorithm using robots equipped with LiDAR sensors and RSSI. MCL was chosen because it is the most commonly used algorithm for single-robot localization. To determine the pose, each robot combines information from MCL, the relative positions of other robots through RSSI, and the positions calculated by other robots. This algorithm increases the precision of the location and can prevent the kidnapped robot problem. This algorithm has been compared with other algorithms like TOA or TDOA in a MATLAB simulation in a rectangular field. Following that, the authors tested the performance in a real-world scenario using a TurtleBot equipped with an ESP8266 chip with a rod antenna for measuring the RSSI. They implemented the algorithm using ROS and Gazebo frameworks. In [23], the heterogeneous robots can be hierarchically organised to reduce manufacturing costs. Robots in this paper is divided into three different classes, each one with a specific job. The first class of robots is "the explorers". These robots send request for a new job to the manager. After that, they receive a new local environment and a path to reach it from the planner. The local environment is a section of the entire map. They reach the new local environment and receive the old local map from the manager and update that during a new exploration of this area. The second type of robot is the "planners". Planners are delegated to assign the explorers in local environments and manage global path planning for all the explores. When a new job is sent from the manager, it is necessary for all planners to collaborate to avoid path collisions and assignment

conflicts. The third and last type of robots is “the manager”. The major purposes of the manager are maintaining local environment updates and managing the job queue. The robots are divided in based on processing, communication, sensing and actuation capability. Another aspect that the author considers in this paper is the capability of each robot to traverse different types of terrain and the different size of the robots. In [24], the authors demonstrate the importance of robots collaborating with each other and sharing information both amongst themselves and with other sensors fixed in the environment. A good way to fuse the data is to use the KF to increase the robustness. To reduce the replacement cost of old AGVs is possible to use AMR to support them. AMRs have to detect the presence of humans in the environment and update the shared map. The paper now describes the architecture of this system. AGVs are the meta sensors that allow to identify the human in the environment, using an IP camera through YOLO algorithm, and a laser-camera to track them. Sensors Synergy Center receives information about the position of AGVs from AGV Coordination Center to take decision for the AMR on the basis of the current task of AGV. AMRs localize themselves implementing the Adaptive Montecarlo Localization algorithm. When AMRs do not have the sentry role, they can do traditional AMR tasks. According to ANSIITSDF B56.5 "Safety Standard for driverless, automatic guided industrial vehicles and automated functions of manned industrial vehicles", they identify 3 types of zone based on the probability of finding a human. The three types are:

- **Critical area**: Zones with no visibility for AGVs or where humans are likely to be present.
- **Human presence area**: Zones where human activity is less static, making it more likely to find people.
- **Human-free area**: Zones where no humans are expected to be present.

The human behaviour can be classified as static if the human moves less 1 m from the first detection position, otherwise is defined as dynamic. When an AGV is assigned a new task, an AMR is chosen from the list based on its distance from the required area and the priority of its current task.

A key idea is the exchange of information between robots. Systems that use real-time information exchange (such as position) include online multiplayer video games. In [25], the authors compare the performance of various interest management algorithms in MMORPG games. Instead of transmitting all state changes to every player, only the relevant ones are sent, improving efficiency. The results show that data obtained from computer-controlled players can simulate those from real players. The authors conclude that techniques for message limitation depend on the increase in the game’s world size, the number of players, and the morphology of the map itself.

Chapter 3

Robot Operating System

3.1 Robot Operating System

Robot Operating System (ROS) is an open-source framework used for building and controlling robotic systems. ROS is not a real operating system but a middleware that facilitates communication between various components of a robotic system. ROS [2] was originally developed at Stanford University by Eric Berger and Keenan Wyrobek. Their efforts, supported by contributions from colleagues and early funding, eventually led to collaboration with Willow Garage, where ROS was formally launched in 2007. They developed the PR2 robot and ROS, establishing ROS as a multi-robot platform. In 2012, ROS became a global standard with the release of the first official ROS distribution and the birth of the Open Source Robotics Foundation (OSRF). The primary objective of ROS is to provide a standardized platform for sharing code and research outcomes. ROS [26] has three levels of concepts: the Filesystem level, the Computation Graph level, and the Community level. At the filesystem level, all code, data, and configuration files are encapsulated into a wrapper called a ROS Package, which is structured as shown in Listing 3.1.

```
1
2       my_robot_package/
3       |-- CMakeLists.txt
4       |-- package.xml
5       |-- src/
6       |       |-- my_robot_node.cpp
7       |       |-- another_file.cpp
8       |-- include/
9       |       |-- my_robot_package
10      |       |-- my_robot_node.h
11      |-- scripts/
```

```
12      |       |-- my_script.py
13      |-- launch/
14      |       |-- my_robot_launch.launch
15      |-- config/
16      |       |-- my_robot_config.yaml
17      |-- msg/
18      |       |-- MyMessage.msg
19      |-- srv/
20      |       |-- MyService.srv
21      |-- worlds/
22              |-- my_world.world
```

Listing 3.1: Example of the file system structure of a typical ROS package

The ROS community level includes all the people and groups who contribute to and support the ROS development and distribution. The most complex level is the computation graph level. The architecture of ROS is graph-oriented, where processing takes place in nodes that can communicate with other ones using messages through topics and services (a more detailed explanation will be given later). A key component that enables this communication is the ROS Master, which is made to facilitate various functionalities [27]:

- **Name Resolution**: The ROS Master maintains a registry of names for nodes, topics, and services. Nodes use the Master to find each other by name, facilitating seamless communication across the system.
- **Topic/Service Registration**: The Master tracks which nodes are publishing and subscribing to which topics. This information allows nodes to efficiently establish communication channels by discovering publishers and subscribers. Additionally, it manages the offered services within the ROS system.
- **Parameter Server**: It hosts a parameter server where nodes can store and retrieve configuration parameters dynamically during runtime. This central repository enables nodes to share and update settings as needed.

ROS has two major versions, ROS1 and ROS2. The main difference between ROS1 and ROS2 [28] is that ROS1 utilizes a client-server structure, where the ROS Master interfaces with all the other nodes, while ROS2 does not have master and slave nodes but the infrastructure is based on peer-to-peer model, making ROS2 decentralized and improving ROS1 from different points of view [29]. Nodes are the fundamental building block for ROS and each node is a process that performs computation; they are written in various programming languages including C++ and Python. Each node can represent an entity or functionality like robotic arms. For instance, a node may embody a sensor that gathers environmental data, a node could control a robot joint, or it could also represent a more abstract functionality,

such as a higher-level control algorithm to choose the best path in the environment. Nodes in ROS are autonomous entities that can operate independently or interact with other nodes to achieve complex robotic behaviors. They communicate with each other using tools provided by the ROS framework, with topics and services being the primary mechanisms facilitating ROS node communication. The first tools for sharing information between nodes are the topics. A topic in ROS operates on the basis of publish/subscribe semantics, which can be conceptualized as a pipe. Publishers push messages into this pipe, and nodes that are subscribed to the topic receive these messages. Adopting a publish-subscribe model, ROS topics allow nodes to publish messages into a shared space and subscribe to receive messages of interest. This flexible communication paradigm supports various patterns, including one-to-many and many-to-many relationships among nodes. The nature of this type of communication is asynchronous, in fact, the messages are published and received independently of each other. The second ROS tool that offers synchronous communication between two ROS based on the request/response paradigm is the service. In ROS, a node can provide a service that other nodes in the ROS network can call upon. The server node waits for receiving specific requests, typically defined in a .srv file. In .srv it is necessary to define the type of messages for the input and for the output, that can be only one. When a request is received, a handler processes the request data and formulates a response based on the input. It is important to know in advance the ROS message definition, whether it is a service or a topic, because this allows for proper handling and processing of the data. A message is a simple data structure comprising typed fields. You can create new messages in ROS that consist of default message types or simple data by defining a .msg file. This file specifies the data fields that compose the custom message structure. When defining a topic or specifying the output message type for a service, you can only use one message type or, for the topics, an array of the same messages. If more messages must be sent back, it is necessary to define a new wrapper message whose fields are all necessary messages.

3.2 The ROS navigation stack

ROS navigation stack is a flexible ROS framework designed for the autonomous navigation of mobile robots. This framework provides the necessary tools and libraries to allow robots to move intelligently in the environment, create paths, avoid obstacles and in general for solving the SLAM problem. This ROS package is composed of a set of modular parts that are easily interchangeable in order to choose the correct algorithm for each case. As can be seen in Figure 3.1, for the correct use of the Navigation Stack, it is necessary to implement a set of nodes that are different from robot to robot:

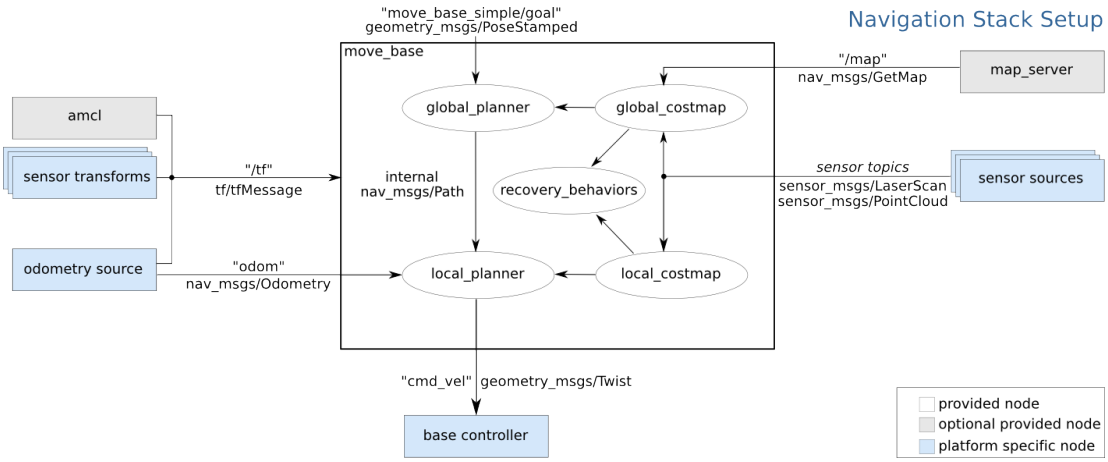


Figure 3.1: Overview of ROS Navigation Stack [30]

- **sensor transforms**: this node is necessary for publishing the transformation between sensor frames and the robot's base frame in the /tf topic; it is necessary to know the relative positions and orientations of these sensors with respect to the robot.

- **odometry source**: this node is necessary for publishing the transformation between the robot's base frame and the odometry frame in the /tf topic; it is necessary to know the robot's estimated position and orientation based on the data received from the wheel odometry data.

- **base controller**: this node is necessary for receiving the result of the Navigation Stack, that are the instructions for moving the robots in the environment, and translating the given velocity into signals that the robot's wheels can execute.

In addition to these data, each sensor publishes specific information on dedicated topics providing data such as sensor readings, status updates or diagnostic information. The core of the Navigation Stack is in the package that provides the `move_base` [30]. `The move_base` package in ROS provides the autonomous navigation capability for the robot using global and local planners to create the relative best path without collision. A diagram in Figure 3.2 shows the algorithm used by this node for recovery behaviors when the robot finds obstacles or gets stuck.

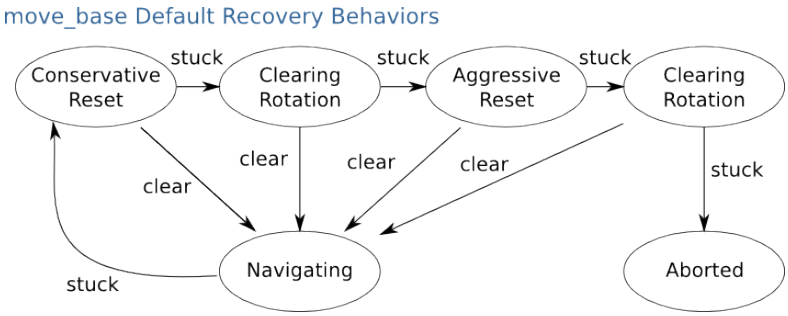


Figure 3.2: Recovery behaviors graph [30]

3.2.1 Costmap2D, layers and global and local costmap

In ROS, `costmap_2d` is a tool that provides a structured representation of the robot's environment. To store this information, `costmap_2d` utilizes a 2D grid map and categorizes cells representing free space, obstacles or unknown terrain. ROS navigation stack uses a wrapper that contains the `costmap_map` data called `Costmap2DROS`, which provides a two-dimensional representation. In fact, `costmap_2d::Costmap_2DROS` contains among its attributes a `costmap_2d::LayeredCostmap`, that contains principally a `Costmap2D` object and a vector of `Layer` objects. `costmap_2d::Layer` is an abstract class that defines the interface and basic behavior for layers used in the creation and management of costmaps in ROS. Each layer represents a specific aspect of the environment. Examples of these layers include `ObstacleLayer`, `InflationLayer`, and `StaticLayer`. To create and update a costmap, `Costmap2DROS` calls `Costmap2DROS::updateMap()`, which, in turn, calls `LayeredCostmap::updateMap()`, which, in turn, calls the `updateCosts` function for each of its layers. Each `Layer::updateCosts` directly updates the `Costmap2D` object of the `LayeredCostmap` to which it belongs. The association of a `costmap_2d::Layer` with a `costmap_2d::LayeredCostmap` occurs through the `addPlugin`function. The same `Layer` object will have a pointer to the `LayeredCostmap` to which it belongs.

The `move_base` node will then handle the creation of two `Costmap2DROS` objects: one representing the global costmap and one representing the local costmap. The global costmap is generally created from a map, and represents the environment with only static obstacles and it is used to create a priori path that can be modified during the execution based on the possible changes found. The local costmap instead is continually created from the information given by the sensors during the execution created previously from the global planner. It represents the environment that the sensors of the robot can perceive during the exploration, including new or dynamic obstacles. Using this map the robot can adjust the trajectory based on

the changes found. The local costmap and the global costmap are usually formed by the same type of principal layer

- `ObstacleLayer`: This layer is responsible for managing and representing obstacles detected in the robot's environment. It typically receives sensor data or other inputs indicating the presence of obstacles and updates the costmap accordingly. The costmap assigns higher costs to grid cells occupied by obstacles, influencing the robot's path planning to avoid collisions.
- `InflationLayer`: This layer is tasked with expanding the influence of obstacles in the costmap. It works by assigning increasing costs to grid cells near obstacles, creating a buffer zone or safety margin around them.
- `StaticLayer`: The StaticLayer handles static elements in the environment that are not expected to move or change often. This can include features such as walls, buildings, or other permanent structures. The layer updates the costmap with information about these static elements. The StaticLayer usually creates or updates the costmap based on the occupancy grid map received from the map server.

The global costmap usually includes all the layers mentioned above, while the local costmap typically consists of one ObstacleLayer per sensor and an InflationLayer.

Map_server

The map_server node in ROS is essential for robot navigation and localization. It manages occupancy grid maps, typically loaded from files like .pgm and .yaml, and provides them to other nodes via ROS messages. Firstly, localization nodes such as AMCL (Adaptive Monte Carlo Localization) rely on the gridmap from map_server to estimate the robot's precise position and orientation within its environment. By comparing sensor data, such as LiDAR scans, against the gridmap, localization algorithms can accurately determine the robot's location. Furthermore, the StaticLayer in ROS's costmap_2d framework directly subscribes to the gridmap provided by map_server. This layer specifically represents permanent static elements in the environment, crucial for long-term navigation planning and obstacle avoidance strategies.

3.2.2 Global and local planners

Various algorithms exist for selecting the optimal path from a global costmap, a process called global planning. Similarly, algorithms used to correct the trajectory are referred to as local planning. It is possible to create a customized algorithm,

including the `nav_core::BaseGlobalPlanner` interface. One of the most common global planners used with `move_base` in ROS is the `global_planner`. It allows for parameter management to customize the path generation process, optimizing it according to specific project requirements.

- `use_grid_path`: if this parameter is set to true, the generated path follows the grid boundaries of the environment map.
- `use_dijkstra`: if this parameter is set to true, the path is generated using the Dijkstra algorithm that guarantees the shortest path but is slow [31].
- `use_a_star`: if this parameter is set to true, the path is generated using the A* heuristics that is more efficient than Dijkstra [31] and is therefore more suitable for a dynamic environment. In [32] a new relaxed version of this algorithm was create, further decreasing the converging time.

Another common global planner is NavFn. It employs a potential field method to generate paths by representing the environment as a grid of potential values. These values guide the robot from its current location to a specified goal while avoiding obstacles using Dijkstra's algorithm [33]. Usually this planner offers better performance [34]. To a better understanding of how to create a customized plugin and how to use it, the authors in [31] show step by step how to create the plugin and how to set the correct parameters to activate it. The work in [35] shows the Reinforcement learning algorithm, which is a machine learning algorithm with trial-and-error nature to create a new global planner. With the advent of deep learning, this algorithm is used to approximate training for other deep learning algorithms. This algorithm works trying to maximize a reward cost by taking the best possible action. There are a set of deep learning evolutions like Q-Learning (that have problems with high dimension) and Deep Q-Network. The authors set a hyper-parameter and train the net with 15000 episodes (every episode has a specific cost that expresses how the episode is done) simulated in ROS Gazebo.

Local planning refers to the process of generating and executing short-term navigation paths for a robot in real time, typically within its immediate surroundings. It adjusts or refines the trajectory provided by global planning to ensure the safe and efficient movement of the robot. A variety of local planners are available in ROS, with three of the most commonly used ones illustrated in [36] for their accuracy in robotic navigation.

- **DWA Local Planner**: The DWA algorithm focuses on reactive collision avoidance by maximizing an objective function. This function considers factors such as progress toward a target, clearance from obstacles, and the robot's forward velocity. Operating within a dynamic window of achievable velocities, DWA discretely samples potential velocity pairs (linear velocity and

angular velocity) and simulates their application over short time intervals. It evaluates these trajectories using a cost function to select the optimal velocity pair that minimizes risk and maximizes progress. It is implemented in the `dwa_local_planner` package in ROS.

- **EBand Local Planner**: The EBand algorithm addresses real-time collision-free motion control using "contraction" and "repulsion" forces. These forces adjust the robot's trajectory dynamically to avoid obstacles while maintaining a smooth path. The algorithm stretches an "elastic band" along the desired path, deforming it as needed to circumvent encountered obstacles. It is implemented in the `eband_local_planner` package in ROS. It performs high accuracy in path following, effectively navigating complex environments with minimal deviations.

- **TEB Local Planner**: Building upon EBand, the TEB algorithm extends its capabilities by optimizing trajectories closer to the robot's current position. TEB aims for time-efficient path execution while considering robot dynamics, geometric constraints, and obstacle avoidance. It generates a sequence of intermediate robot poses, known as "timed elastic bands" to minimize trajectory execution time and optimize multiple trajectories simultaneously. Initially supporting non-holonomic robots and later expanding to holonomic robots in the ROS Kinetic version, TEB is implemented in the `teb_local_planner` package. It performs high accuracy in path following, effectively navigating complex environments with minimal deviations. The work in [37] shows the passage for the process of installing and setting up the TEB for mobile robot navigation in dynamic environments. This setup is particularly beneficial due to TEB's speed, which achieves superior results in highly dynamic environments.

3.2.3 Localization

Localization is a critical aspect of autonomous robotic systems, enabling robots to determine their position and orientation within their environment. In the context of ROS, localization refers to the process through which a robot estimates its pose (position and orientation) relative to a known map or environment. This capability is essential for robots to navigate effectively, interact with objects, and perform tasks autonomously in dynamic and unknown environments. ROS provides a comprehensive framework for implementing various localization algorithms and techniques, catering to different types of robots and sensor configurations. The default localization node [38] is AMCL. It enables robots to accurately determine their position and orientation in known environments. Utilizing a probabilistic approach known as Monte Carlo localization, AMCL maintains a distribution of particles representing possible robot poses based on sensor data and a pre-existing

map of the environment. This adaptive nature allows AMCL to dynamically adjust the number of particles, optimizing computational resources for efficient localization while ensuring robust performance in the presence of sensor noise and environmental changes. AMCL estimates the transformation between the base_frame and the global frame (/map), but it publishes the transformation between the odometry frame and the global frame.

3.2.4 Real-Time Appearance-Based Mapping

Real-Time Appearance-Based Mapping (RTAB-Map) is a SLAM solution. Unlike traditional SLAM methods [39], [40] that primarily rely on geometric data from sensors like LiDAR, RTAB-Map integrates visual appearance information extracted from RGB-D cameras. This approach enhances mapping accuracy by capturing detailed textures and colors of the environment alongside geometric features. This node can be used for creating a map using SLAM by setting the parameter `localization = false`. Setting `localization = true`, instead, allows the node to take the place of both the `map_server` node and the localization node. Therefore, the RTAB-Map node will localize the robot implementing it and publish the topic containing the gridmap based on the map that was created in a previous SLAM phase.

3.2.5 RVIZ visualization

To develop ROS applications, there exist efficient tools that simplify the user interface and the management of all the parameters of the robot. One of the most used tools is RVIZ [41]. Like any ROS feature, this software is a node that subscribes to the main topics of the ROS navigation stack. From this software, once properly set, it is possible to:

- visualize the global map sent to the robots
- interprets the data received from the robot's sensors
- visualize the global and the local map, and the current path
- set the goal of the robot directly in the map

3.2.6 Gazebo Simulation

In some cases, it is preferable not to work directly with robots but in a simulated environment opportunely created. There exists a tool called Gazebo [42] that permits to simulate not only the environment but the entire robot's dynamics. It is necessary to create a structure, called Unified Robot Description Format (URDF)

that is an XML-based file used to describe the kinematic and dynamic structure of the robot denoting joints, links and frames. With this information, this software can recreate the node and the topics like sensors, odometry and motor wheels.

Chapter 4

Robots Description

This Chapter provides a detailed description of the robots used in this thesis: the Locobot WX250S-6DOF and the Turtlebot3 Burger. Each robot will be described in terms of both hardware and software components. The hardware description includes the main components, sensors, and computational units, while the software description covers the operating system and the software packages provided by the robots' manufacturers.

4.1 Locobot WX250S Description

4.1.1 Hardware Description

The Locobot WX250S-6DOF shown in Figure 4.1 is produced by Trossen Robotics. The robot is 622.8 mm tall and is equipped with an RPLIDAR A2M8 360° Laser Range Scanner positioned on top. This robot is equipped with a 6-degree-of-freedom WindowX Robot Arm, which, for the purposes of the thesis, is not necessary. At the core of this robot is an Intel NUC NUC8i3BEH Mini PC, which includes an Intel Dual-Core i3-8th Gen processor, 8GB of DDR4 RAM, and a 240GB Solid State Drive (SSD). The only GPU available on this system is the Intel Iris Plus Graphics 655, which lacks the necessary power to handle Convolutional Neural Networks (CNNs) for detecting and recognizing people. The NUC is shown in Figure 4.2. The Locobot WX250S-6DOF from Trossen Robotics features the Kobuki platform as its mobile base, as shown in Figure 4.3, specifically designed for indoor applications. This base includes two independent driving wheels, enabling precise and flexible maneuverability. The Kobuki platform is equipped with an active bumper sensor, which detects collisions with obstacles, allowing the robot to detect a contact with an object, and cliff sensors that prevent the robot from driving off edges with a depth of 50mm or more, ensuring it does not fall down

Figure 4.1: Locobot WidowX-250 6 DOF (Kobuki) [43]

Figure 4.2: Intel NUC NUC8i3BEH Mini PC [43]

stairs or other drop-offs. These sensors enhance the robot's safety and operational capabilities. This mobile base is made only for indoor applications. The Locobot

Figure 4.3: Kobuki mobile base [43]

WX250S-6DOF is distinguished by its Intel® RealSense™ Depth Camera D435 (shown in Figure 4.4), a critical sensor that significantly enhances its capabilities. This RGB-D camera is part of the Intel® RealSense™ D400 series, known for providing high-quality depth perception and RGB imagery. This sensor enables the robot to perceive obstacles in front of it, facilitating safer and more efficient navigation.

4.1.2 Software Description

The robot is powered by an Intel NUC NUC8i3BEH Mini PC running Ubuntu 20.04. The software packages provided by Trossen Robotics are available in both

Figure 4.4: Intel® RealSense™ Depth Camera D435 [43]

ROS and ROS 2. However, since the Locobot with the Kobuki mobile base does not support ROS 2, this thesis has chosen to use the ROS version. The Locobot ROS 1 packages offer a comprehensive ROS navigation stack, which includes the NavfnROS global planner and the TrajectoryPlannerROS local planner, both of which utilize `costmap_2D`. For localization and map storage, the package includes the RTAB-Map ROS node, which creates a 3D map of the environment by fusing data from the LIDAR and the Intel® RealSense™ Depth Camera D435. This 3D map is used to generate a `costmap_2D` and to localize the robot within the environment using a Visual Odometry-based algorithm. The global cost map incorporates ObstacleLayer, InflationLayer, and StaticLayer, while the local cost map includes ObstacleLayer and InflationLayer only. The Locobot WX250S-6DOF also integrates MoveIt, a powerful open-source software for mobile manipulation, which facilitates planning, manipulation, and control of the robot arm. MoveIt supports motion planning, kinematics, and collision checking, allowing for complex manipulation tasks to be performed efficiently. The integration with MoveIt enhances the robot's capability to interact with objects and perform precise movements. Additionally, the provided packages support complete management of namespace and `tf_prefix` in the robot's launch files, ensuring compatibility with multi-robot environments. This capability is crucial for complex operations where multiple robots are operating simultaneously within the same space.

4.2 Turtlebot3 Burger Description

4.2.1 Hardware Description

The Turtlebot3 burger [44] is the result of a collaborative project among several companies, including ROBOTIS, Open Robotics, Intel, Onshape, and OROCA. Structurally, it differs significantly from the Locobot WX250s, particularly in size and features. Unlike the Locobot WX250s, the Turtlebot3 Burger does not include a robotic arm. Its primary sensor is the LiDAR 360, specifically the LDS-02 model mounted on top at a height of approximately 120 mm from the ground. The Turtlebot3 Burger is powered by a Raspberry Pi 3, which lacks a GPU, resulting in

limited computational power. The Turtlebot3 Burger is shown in Figures 4.5. The custom moving platform is powered and controlled by an OpenCR board, which integrates various sensors and actuators for autonomous navigation. It utilizes two DYNAMIXEL wheels for precise and efficient movement. This version highlights that the OpenCR board not only controls the platform but also integrates sensors and actuators for autonomous navigation.

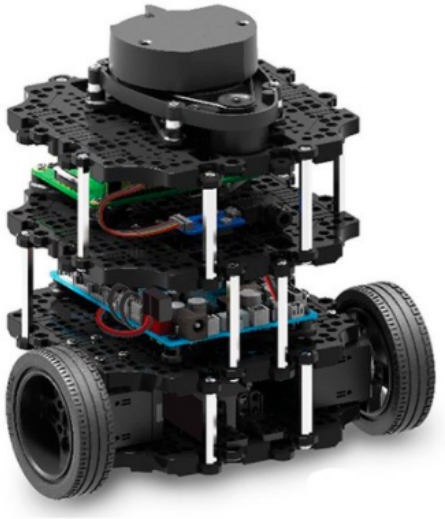


Figure 4.5: Turtlebot3 Burger

4.2.2 Software Description

The provided package is very basic for ROS navigation. This robot is compatible with both ROS 1 and ROS 2. The map created is saved using the `map_server` ROS node, which creates and saves a 2D costmap of the environment using the LiDAR. The launch file does not provide any management of namespaces or `tf_prefix`. If one tries to launch this robot with others, the system may fail, because the node and topic names are the same for all robots. The YAML files for configuration are compatible with versions before Hydro.

Chapter 5

ROS framework for multi robot

The development of multi-robot systems has garnered significant attention in robotics research and industry due to their potential to increase efficiency and performance in various applications, such as exploration, surveillance, and cooperative tasks. ROS2, along with the Navigation stack (Nav2), offers features and tools that enhance multi-robot coordination [45]. ROS2 [46] improves the performance for real-time communications and is more efficient for large-scale systems. However, it is still possible to create a multi-robot system using ROS1. Careful attention is required when launching nodes for multi-robot to avoid issues and conflicts with duplicated names. In this case, tools such as launch files can be used not only to simplify the launching of multiple nodes belonging to a single robot, but also to launch the same nodes more times with different namespaces to avoid name conflicts [47]. This ensures that each instance of a node operates independently, facilitating better coordination and management of multi-robot systems. In ROS, a launch file is an XML file that describes the information and parameters required to start nodes. It allows users to launch multiple nodes and set parameters quickly using a single command line. A typical launch file can include various elements such as node declarations, parameters, re-mappings, and groupings that help to manage the complexity of launching a robotic system:

- **Node Declarations**: Specify which nodes to launch, including the package name, executable, and any necessary arguments. It is possible to launch another launch file.
- **Parameters**: Define parameters to be passed to nodes, either directly in the launch file or from external YAML files.
- **Groups**: Organize nodes and settings into logical groups, which can also

include namespaces to avoid name conflicts.

- **Remappings**: Redirect topics, services, and other resources to avoid conflicts and ensure correct communication between nodes.
- **Conditional Launches**: Enable or disable the launching of nodes based on specific conditions or parameters.

To use a launch file, the roslaunch command is used:

```
roslaunch <package_name> <launch_file_name>
```

Often, robot manufacturing companies provide launch files to enable the correct functionality of their robots. However, these companies do not always provide parameters that prevent conflicts when launching multiple robots. Potential conflicts include:

- Robots having the same transformation (tf) names
- Nodes having the same names for multiple robots
- The launch file starting a node several times, while it should be launched only once, such as the map server node
- Launch the same node several times for different nodes, each with different parameters.

Using namespaces and careful parameter management can help mitigate these conflicts. Launch files can include parameters that allow setting unique names or namespaces for each robot instance, ensuring smooth operation in multi-robot systems.

5.1 ROS launch file for multiple robots

If one tries to launch the nodes belonging to a robot more times, it is likely that some errors will occur. A guideline for solving this type of problem both in simulation and in a real environment is provided in [47], [48]. The principle for launching multiple robots is that each node belonging to a specific robot should share the same namespace. This ensures that every node launched for a particular robot has a consistent prefix (namespace), which distinguishes it from nodes belonging to other robots. For example, let's consider a scenario with two robots, "robot1" and "robot2" that want to navigate using the same `move_base` node in the same ROS network. It is possible to use namespaces to differentiate their nodes. By using

them in the launch file, robot1's move_base node becomes "robot1/move_base" and robot2's becomes "robot2/move_base". The XML files can vary between different robots, but they typically share a core structure when using the ROS navigation stack. For instance, the launch file provided by ROBOTIS [44] for the TurtleBot3 and the launch file provided by Trossen Robotics [49] for the Locobot WX250S have similar foundational patterns, albeit with noticeable differences.

```
<launch>
    <arg name="robot_name" default="name" />
    <arg name="init_pose" default="-x 0 -y 0 -z 0" />
    <arg name="model" default="model" />

     <!-- Set the robot description -->
    <param name="robot_description" command="$(find xacro)/xacro.py
    urdf/robot.urdf.xacro" />

    <!-- Spawn the robot model in Gazebo -->
    <node name="spawn_robot_model" pkg="gazebo_ros" type="spawn_model
    " args="-urdf -param robot_description -model $(arg model) $(arg
    init_pose)" respawn="false" output="screen" />

    <!-- Publish robot state -->
    <node pkg="robot_state_publisher" type="robot_state_publisher"
    name="robot_state_publisher" output="screen"/>

    <!-- AMCL node for localization -->
    <node pkg="amcl" type="amcl" name="amcl">
        <param name="global_frame_id" value="/map" />
        <!-- Add other required parameters for AMCL -->
    </node>

    <!-- Move Base node for navigation -->
    <node pkg="move_base" type="move_base" respawn="false" name="
    move_base" output="screen">
        <rosparam file="params.yaml" command="load" />
        <remap from="map" to="/map" />
    </node>

    <!-- RViz for visualization -->
    <node pkg="rviz" type="rviz" name="rviz" args="-d file.rviz"
    required="true"/>

</launch>
```

Listing 5.1: Example of a robot launchfile, called single_robot.launch

The XML in 5.1 represents a generic launch file for a robot utilizing the ROS navigation stack in a simulated environment using Gazebo. The robot is initially

spawned in the Gazebo world, followed by the launch of nodes essential for navigation such as AMCL, Move Base, and RViz for visualization. If the robot is used in the real world, nodes for robot control and sensors would be launched instead of those for Gazebo. It is important to notice the line:

```
<remap from="map" to="/map" />
```

This line is necessary to share a single map among all the robots by redirecting the map topic (/map) accordingly. If you attempt to launch this launch file twice, you may encounter the errors mentioned above. The key to successfully launch two or more robots is to set the correct namespace for each robot and ensure that each robot has a different tf_prefix. Listing 5.2 is an example of how it should be managed.

```
<launch>
     <!-- Start Gazebo with a specific world -->
    <node name="gazebo" pkg="gazebo" type="gazebo"
        args="map.world" respawn="false" output="screen" />

    <!-- Run the map server -->
    <node name="map_server" pkg="map_server" type="map_server" args="
    $(find your_pkg)/map/map.yaml" >
        <param name="frame_id" value="/map" />
    </node>

    <!-- LAUNCH FIRST ROBOT -->
    <group ns="robot1">
        <param name="tf_prefix" value="tf1" />
        <include file="single_robot.launch">
            <arg name="init_pose" value="-x 0 -y 0 -z 0" />
            <arg name="robot_name" value="robot1" />
            <arg name="model" value="model1"/>
        </include>
    </group>

    <!-- LAUNCH SECOND ROBOT -->
    <group ns="robot2">
        <param name="tf_prefix" value="tf2" />
        <include file="single_robot.launch">
            <arg name="init_pose" value="-x 1 -y 1 -z 1" />
            <arg name="robot_name" value="robot2" />
            <arg name="model" value="model1"/>
        </include>
    </group>

</launch>
```

Listing 5.2: Example of launch file for multiple robots

This launch file begins by initializing the Gazebo node to simulate the environment described in map.world. Following this, the map server node is started to provide the map topic required for all robots. Each robot's node is launched using the single_robot.launch (Listing 5.1) file within a group, which defines a distinct namespace for each robot to ensure namespace isolation. This namespace acts as a prefix for all nodes within the group. For instance, a node named NavigationNode inside a group with namespace Robot1 would be launched as Robot1/NavigationNode. Additionally, each robot's parameters can be configured independently within its namespace. Additionally, each robot's parameters can be configured independently within its namespace. The tf_prefix parameter plays a crucial role similar to the namespace, providing a prefix in the TF tree specific for each robot. It is essential that all robots' TF trees are interconnected through a common root, typically /map, to establish a shared reference point for their coordination. In order to achieve this, it is necessary to set a proper configuration of the node responsible for transforming /map to /odom, typically handled by the localization node, such as AMCL in this case. An example of what we can achieve is illustrated in Figure 5.1.

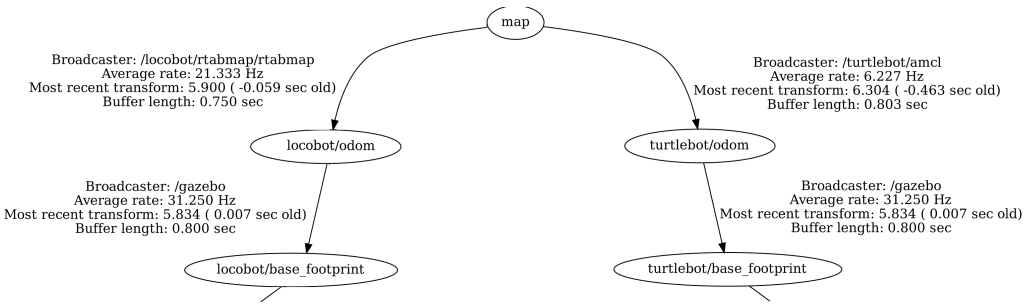


Figure 5.1: A representation of part of TF tree

5.2 Launching Locobot and Turtlebot

Based on the provided information, two main packages have been created: one for launching the Locobot and Turtlebot in map creation mode, and another for launching the robots in localization mode.

5.2.1 Map Creation Modality

The first mode is used only once to create a map of the environment. In this mode, each robot creates its own map, resulting in two different maps. The chosen map is created by the Locobot using both a depth camera and a laser scanner, utilizing the RTAB-Map package. RTAB-Map is selected because it creates a 3D map of the environment [39] and then generates a 2D grid map by projecting the 3D data from all sensors onto the ground plane. This process involves using a voxel grid filter to merge points projected into the same cell and applying 2D ray tracing to fill the empty spaces between obstacles and the camera. This approach ensures that the maps created are highly detailed and accurate, leveraging the strengths of both the depth camera and the laser scanner to capture various aspects of the environment. The integration of these data sources and the advanced processing techniques used by RTAB-Map contribute to the robustness and precision of the generated maps. This implies that the Locobot has sufficient dimensions to explore and map the entire environment. While the map created by the Turtlebot using only LiDAR might not be as detailed as the one created by the Locobot, the maps are still sufficiently similar. Significant differences between the environment viewable by the Turtlebot with only LiDAR and the map sent from the Locobot can create problems during localization if the two maps are very different. RTAB-Map offers a parameter called `Grid/MaxObstacleHeight` to set the maximum height for considering an object as an obstacle. By adjusting this parameter, it is possible to create various mapping scenarios.

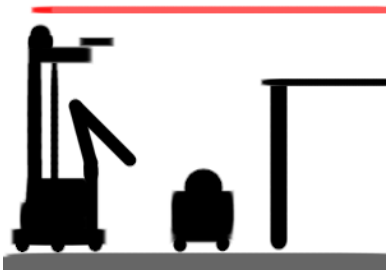


Figure 5.2: The red line represents the MaxObstacleHeight that, in this case, is set to the exact height of Locobot

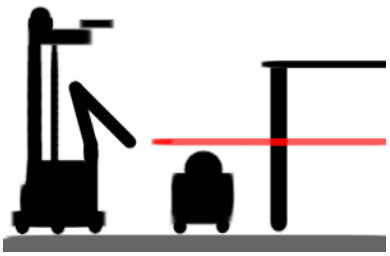


Figure 5.3: The red line represent the MaxObstacleHeight that, in this case is set to the exact height of Turtlebot

In the first case, shown in Figure 5.2, the `MaxObstacleHeight` parameter is set to the height of the Locobot, which is taller than the Turtlebot. Consequently,

the grid map generated by the Locobot can be significantly different from the Turtlebot's view. For instance, a table with a top height between the Locobot's height and the Turtlebot's height would be represented as a rectangle in the grid map created by the Locobot (Figures 5.4, 5.5). On the contrary, the Turtlebot would perceive this area as four circles, representing the table legs. This discrepancy occurs because the Turtlebot's LiDAR cannot detect the tabletop, but only the legs.

In the second case, shown in Figure 5.3, the `MaxObstacleHeight` parameter is set to the height of the Turtlebot's LiDAR position. In this scenario, the Turtlebot's view matches the grid map provided by RTAB-Map (Figures 5.6, 5.7, 5.8). This alignment means that a path marked as possible for the Turtlebot might appear feasible, but during movement, the Locobot may discover the path is unfeasible and change its trajectory accordingly. However, this does not pose a problem for the Locobot's localization, because it relies on the 3D information of the environment for accurate positioning.

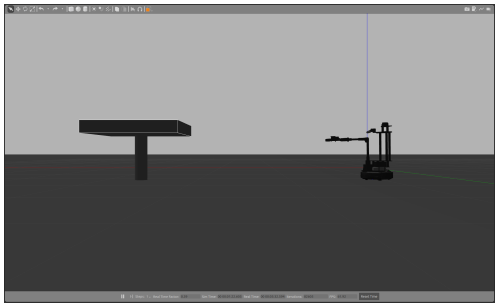


Figure 5.4: The simulated environment with the MaxObstacleHeight set to 0.7

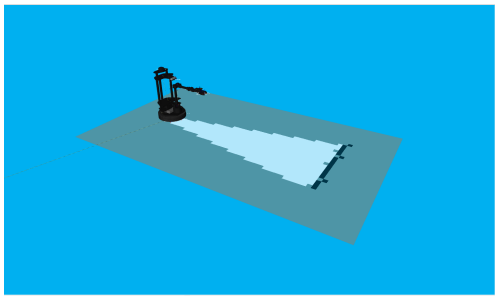


Figure 5.5: The map generated, the obstacles are placed in the map in rectangular shape

The second option has been chosen because the Locobot is considered as a support for the Turtlebot. The thesis aims at demonstrating how a robot with more sensors can send information about the environment to a robot that would otherwise be unable to gather such information. This choice depends on the environment's morphology and must be evaluated on a case-by-case basis. It is possible to merge the two maps to create a new one, but merging heterogeneous maps poses several challenges [50]:

- The chances of incorrect merging are higher with heterogeneous maps. Therefore, it is necessary to search mechanisms to reduce the risk of map corruption due to errors. Possible solutions include multi-level map storage solutions or meeting strategies to confirm merging decisions.

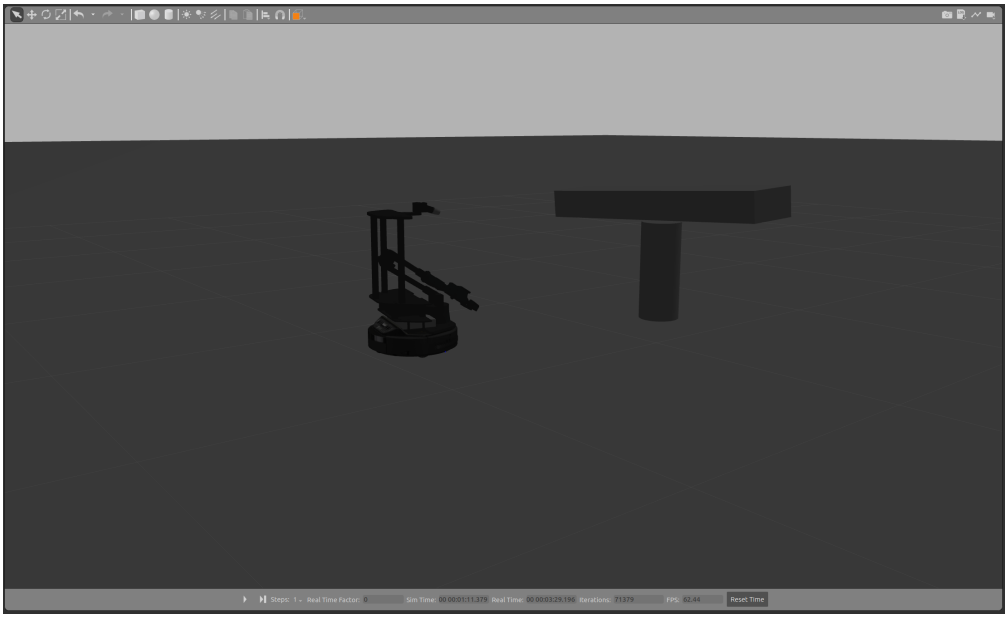


Figure 5.6: The simulated environment with the MaxObstacleHeight set to 0.2 and laser_scan parameter set to false

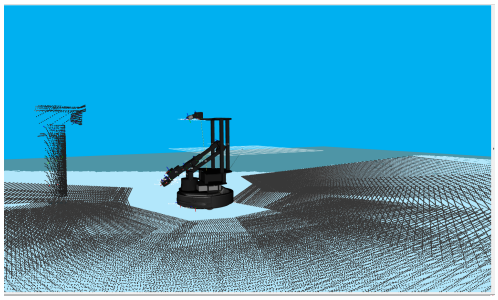


Figure 5.7: The Locobot takes information from all the figure

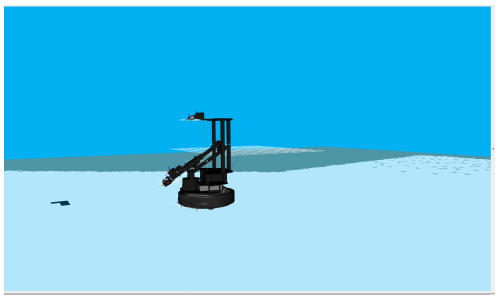


Figure 5.8: The map generated, the obstacles are placed in the map in circle shape and not with rectangular despite the robot see the rectangular place in top of the cylinder

- Merging of different quality maps is currently lacking even for same-type maps. To facilitate the propagation of higher quality maps, map quality assessment algorithms are necessary.
- There are limited solutions for asymmetric merging, where a separate global map is produced for each involved robotic platform. This means each robot

may end up with a different version of the map, complicating coordination and navigation.

In practice, RTAB-Map heavily relies on the LiDAR system, even when the `MaxObstacleHeight` is set lower. To make the system work effectively, it would be necessary to disable the LiDAR for map creation, but this significantly impacts the map's accuracy. While the result might be sufficiently good in simulation, in a real environment, the map quality would be too poor.

5.2.2 Navigation Modality

This modality is used to enable robots to navigate within the environment. The map utilized has already been created in a previous mode. In this modality, both robots are launched while adhering to the precautions mentioned earlier regarding namespaces and tf_prefix. This launch file directly invokes the launch file provided by Trossen Robotics, ensuring the correct parameters are set as detailed in Table 5.1. It initiates the Gazebo world, generating and saving it as house.world file.

Parameter	Value
rtabmap_args	_init_pose:=[0,0,0,0,0,0]
robot_name	locobot
robot_model	locobot_wx250s
localization	true
world_name	$(find all_robot_gazebo)/worlds/house.world
paused	true
use_position_controllers	true

Table 5.1: Parameter table for Locobot

The launch file provided by ROBOTIS to simulate the Turtlebot has been revised to correctly configure parameters. Initially, the original launch file lacked support for namespaces and tf_prefixes. Introducing these into the initial launch file caused nodes to not connect correctly, as absolute tf IDs (preceded by "/") were used in the configuration files for both topics and tf frames. To address this issue, adjustments were made to ensure that namespaces and tf_prefixes were appropriately handled, allowing for proper node connectivity and TF frame resolution. Furthermore, the map server node launch has been removed from the setup because the RTAB-Map node provides the map instead of map_server. Similarly, modifications were applied to the amcl.launch file to align the global_frame_id with the value specified by RTAB-Map. Additionally, updates were made to the move_base parameter

files, originally designed for pre-hydro versions, to accommodate new layers and functionalities required for enhanced navigation capabilities, a shown in Listing 5.4.

```
local_costmap:
  global_frame: odom
  robot_base_frame:
    base_footprint

  update_frequency: 10.0
  publish_frequency: 10.0
  transform_tolerance: 0.5

  static_map: false
  rolling_window: true
  width: 3
  height: 3
  resolution: 0.05

```

Listing 5.3: Pre-Hydro parameter for local_costmap

```
local_costmap:
  global_frame: turtlebot/odom
  robot_base_frame: turtlebot/
    base_footprint

  update_frequency: 3.0
  publish_frequency: 3.0
  transform_tolerance: 0.5

  rolling_window: true
  width: 3
  height: 3
  resolution: 0.05

  plugins:
    - name: obstacle_layer
      type: "
    costmap_2d::ObstacleLayer"

    - name: inflation_layer
      type: "
    costmap_2d::InflationLayer"

```

Listing 5.4: Same parameter is the new format

5.2.3 Launch Locobot and Turtlebot in real environment

In ROS network setup used in this thesis, we have three devices: two robots (Turtlebot and Locobot) and a PC serving as the ROS master. The Locobot runs all ROS nodes locally on the robot itself, while the PC is configured to use RVIZ solely for visualizing main topics and sending commands. Conversely, the Turtlebot runs state nodes locally, while the nodes from the ROS navigation stack operate on the PC. This configuration adheres to the namespace and tf_prefix clauses discussed earlier to ensure proper node communication and TF frame management. The map used for navigation was created directly by the Locobot using its LiDAR sensor. This approach was chosen because attempting to create the map without LiDAR resulted in an insufficiently accurate map for localization purposes. In the PC, the node responsible for sharing information, `robot_position_info_manager node`, will also be run, as discussed in the Chapter 6. A representionton of the

ROS network is shown in Figure 5.9

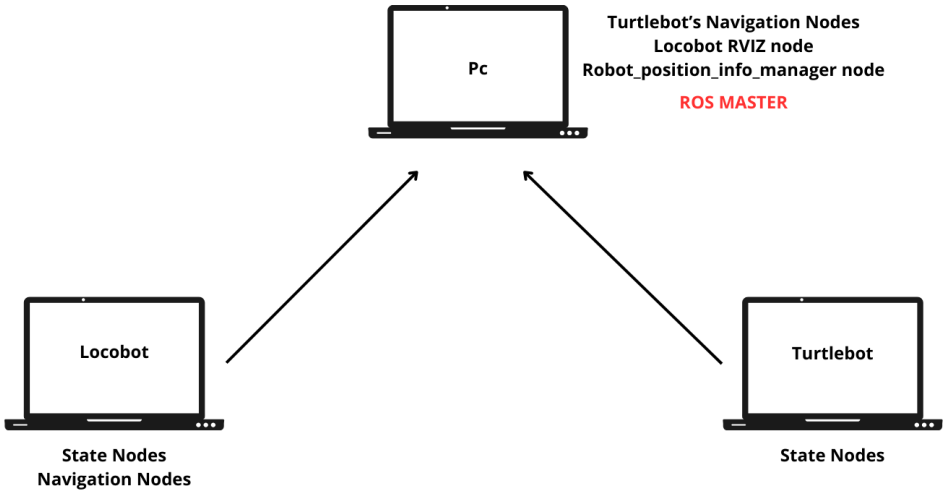


Figure 5.9: Representation of the ROS network and where nodes are run

Chapter 6

ROS node for information position management

The main questions to address in this thesis are: How can information be transmitted between the two robots? In what ways can this information be shared? How can this information be utilized to gain some advantages? What benefits could arise from the heterogeneity of the robots? It is important to consider the meaning given to the local costmap and how it is created, which involves using the robot's sensors for detecting obstacles in real time [51]. The key idea is that the robots share their local maps with each other, essentially sharing information about what they themselves are observing, including the obstacles encountered during path execution. Therefore, it is necessary to find a way to manage and share these local costmaps and how to use them to create paths that consider the information received from other robots. Another piece of information that can be shared is the robots' own positions. By sharing their positions, a robot can use this information to create a path that takes into account the positions of other robots in advance. This chapter will focus on how to share the local costmap.

6.1 Local costmap and position management

The main idea for sharing information is to create a ROS node that manages the local costmap data. A general schema of this node can be found in Figure 6.1. This node will handle the collection, management, and dissemination of local costmap data and position data between the robots. The local costmap is usually published by move_base node and in this topic is published in nav_msgs/OccupancyGrid type of message. The definition of this type of message contains :

- header (std_msgs/Header): it contains timestamped information, such as the

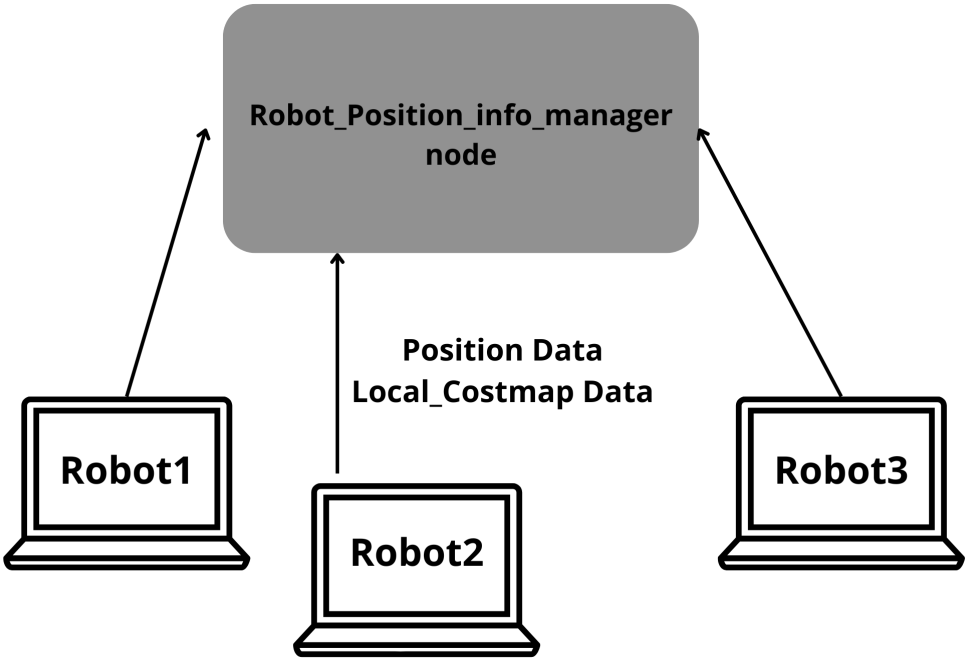


Figure 6.1: Robot_position_info_manager node receive position and local_costmap data from all the robots

time when the local costmap was created

- info (nav_msgs/MapMetaData): it contains metadata about the map like resolution, dimensions and the origin.
- data (int[]): this field is the most important, it contains the local costmap information in the form of grid map where, in row-major order, starting from [0,0] position, each value is set as follows:
  - 0: free cell
  - 1-100: the probability to find an obstacle in this cell
  - -1: unknown cell

The local costmap topic already exists, so it is necessary for the created node to subscribe to these topics to receive the information. To achieve this, a configuration file has been created to select the correct topics and assign a unique name to each robot. This unique name will later serve to identify the robot that is sending this information. The configuration file is a .yaml file named robot_position_info_param.yaml, an example of which can be found in Listing 6.1. The file contains a list, called subscribers, that includes the following fields:

- **name**: The name of the robot that associates the information received from the other fields to the robot.
- **topic**: The topic where it can be found the position of the robot.
- **type**: The message type corresponding to the localization topic. This was created to handle different types of messages that may indicate the robot's position.
- **localtopic**: The topic where it can be found the local costmap.

```
subscribers:
  - name: "robot1"
    topic: "/robot1/position"
    type: "geometry_msgs/PoseStamped"
    localtopic: "/robot1/move_base/local_costmap"
  - name: "robot2"
    topic: "/robot2/position"
    type: "geometry_msgs/PoseStamped"
    localtopic: "/robot2/move_base/local_costmap"
```

Listing 6.1: Example of robot_position_info_manager parameters

When this node starts, it tries to read the robot_position_info_param.yaml file. If the file is not found, the node stops and displays an error on the screen. If the configuration file is loaded correctly, the node subscribes to the topics for each robot listed in the file. For every robot specified in the file, the node will subscribe to the topic containing the robot's position and the local costmap topic. After that, this node executes the command:

```
ros::spin();
```

The node remains listening and receiving information from the subscribed topics. When new data are sent from a topic regarding an updated local costmap, the node has a data structure for saving the latest information about the robot that triggers a callback function. When the callback function manages this event, the node takes the new data and pushes them into the correct position, replacing the old data with the new data. The data structure for saving this information is a map where the key is the robot's name and the data corresponds to the entire message received. This choice was made to speed up data retrieval by the robot's name every time new data are received and every time a robot requires specific position data. This data structure could cause slowdowns during another phase where almost the entire contents of the data structure are printed. In such cases, other structures like simple vectors would have better performance. However, these latter events

generally occur less frequently. In fact, the frequency at which data arrive on the local costmap topic depends on a parameter called `publish_frequency` defined in the local costmap parameters. The frequency of the other event described before, which will be explained in detail later on, depends on the `publish_frequency` of the global costmap, which is generally lower. This is because local planning and obstacle avoidance prefer updated information about the immediate surroundings of the robots, and with a higher value of `publish_frequency`, the robot can react quickly to changes in the environment. If the number of robots grows significantly, it may be considered to replace the data structure with an ordered vector. This would allow for efficient searching using binary search, while still achieving better performance when printing the entire data structure. A similar discussion can be made for the management of the poisoning robot: when new data are received from the robot, a dedicated callback function is called and the data are pushed into the data structure, which is the same previously used for the same reason, and replace the old data with the new. The code structure allows for differentiation in callback handling based on the type of data received from the position topic. The handler I created responds to the `PoseWithCovarianceStamped` type, saving the position while discarding any uncertainty-related data. The map contains objects of a new class called RobotInfo which includes:

```
class RobotInfo {
public:
    geometry_msgs::Pose pose;
    std::shared_ptr<nav_msgs::OccupancyGrid> local_costmap;
    mutable boost::shared_mutex mtx;
};
```

Listing 6.2: RobotInfo Class

This structure allows storing the position and local costmap of each robot while ensuring thread safety with a shared mutex. Multiple threads are allowed to read information from RobotInfo, but only one thread is allowed to modify it. The information will be read by threads from various robots while they create the global costmap, whereas it will be written by only the thread managing the node. To achieve this, get methods have been implemented using shared_lock, allowing multiple threads to concurrently read information from RobotInfo. Meanwhile, setter methods use unique_lock, ensuring that only one thread at a time can modify the data within RobotInfo.

It has already seen how these data are received and stored by the node, but how the data are sent from this node to the other ones? ROS typically facilitates communication through two primary mechanisms: topics and services. However, in certain projects, communication may also occur via external files [18]. In this scenario, where information needs to be sent only when requested by the robot, ROS services offer synchronous communication and appear to be the most appropriate

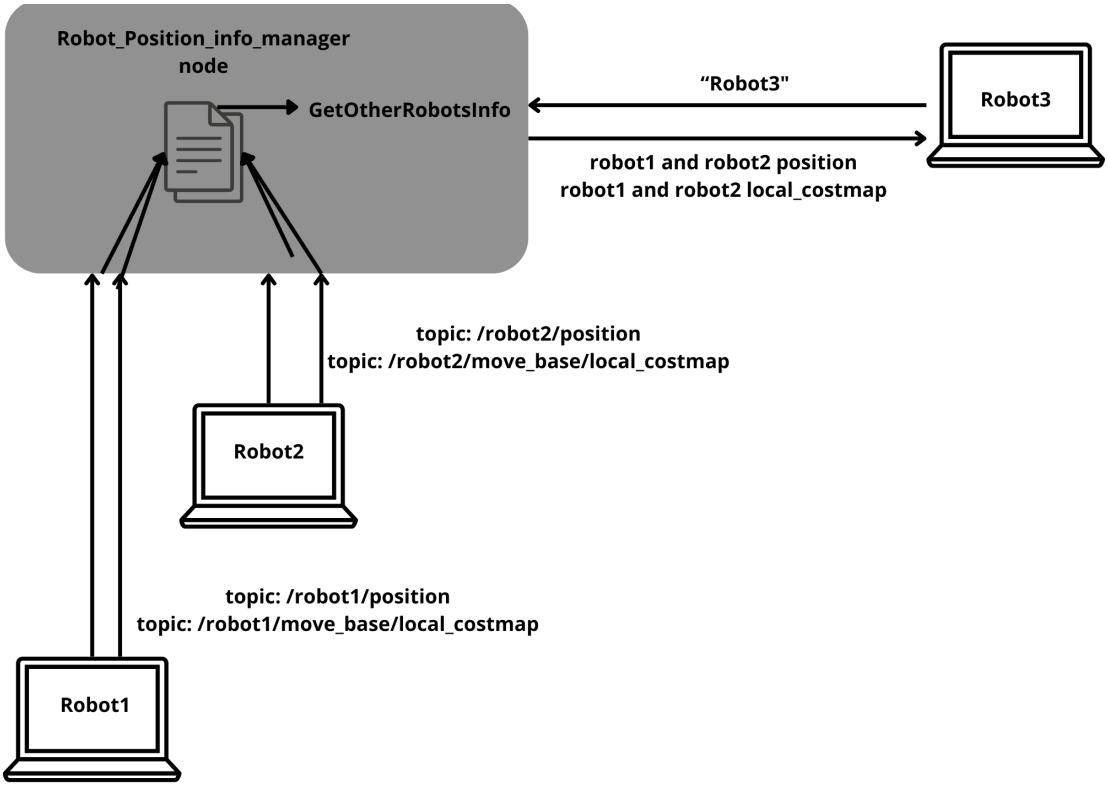


Figure 6.2: Robot_position_info_manager node description

choice. These services will respond to requests sent by robots with the correct information. I have created two main services named GetOtherRobotInfo and GetRobotPosition. The diagram in Figure 6.2 shows the main functionalities of the node.

6.1.1 GetRobotPosition

This service is responsible for sending information to the requesting robot about the latest positions of other robots indicated in the request by name. If the requested robot name does not exist or is null, the service responds with false. The template of this service is illustrated in Listing 6.3.

```
1 string requester_name
2 ---
3 geometry_msgs/Pose robot_position
```

Listing 6.3: The GetRobotPosition service template

As mentioned earlier and as indicated in the first line of Listing 6.3, the input of this service is a string representing the name of the robot requesting the service.

The output of this service is the position of the other robot indicated in the request, in the form of a `geometry_msgs/Pose` message.

6.1.2 GetOtherRobotsInfo

This service is responsible for sending information to the requesting robot about the last known positions and local costmaps of other robots. The node excludes information about the requesting robot itself when responding. For this purpose, the input required to obtain this information includes the name of the requesting robot. If the specified robot name does not exist or is null, the service responds with information about all robots. The template for this service is illustrated in Listing 6.4.

```
1 string requester_name
2 ---
3 positioning_layer/OtherRobotInfo[] other_robots_info
```

Listing 6.4: The GetOtherRobotsInfo service template

As mentioned above and as indicated in the first line of Listing 6.4, the input of this service is a string representing the name of the robot requesting the service. The output consists of two parts: positions and local costmaps. However, ROS services can only respond with a single message or a vector of the same type of message. To accommodate sending both types of information in a single response, a new message type called `positioning_layer/OtherRobotInfo`. This new message is simply composed, as can be seen from Listing 6.5, by two parts:

- **pose** (`geometry_msgs/Pose`): is the message that contains the robot position
- **local_costmap** (`nav_msgs/OccupancyGrid`): is the message that contains the local costmap

```
1 # msg/OtherRobotInfo.msg
2
3 geometry_msgs/Pose pose
4 nav_msgs/OccupancyGrid local_costmap
```

Listing 6.5: The positioning_layer/OtherRobotInfo message template

So, OtherRobotInfo service responds with a vector of the new message type `positioning_layer/OtherRobotInfo`, which contains the positions and local costmaps of all other robots requested by the name provided.

Chapter 7

Positioning Layer

In this chapter, a new costmap_2d layer, called Positioning Layer, is introduced. This layer is responsible for acquiring data from the Robot_position_info_manager node and processing it to gain benefits. To create a new layer, you need to create a new class that implements the `costmap_2d::Layer` interface and overrides three main functions.

- `virtual void updateCosts(costmap_2d::Costmap2D& master_grid, int min_i, int min_j, int max_i, int max_j) = 0;`

  - **master_grid**: A reference to the main costmap that combines all the layers. This object is passed by reference from the `costmap_2d::LayeredCostmap` object that subscribes to the layer.
  - **min_i, min_j, max_i, max_j**: These parameters define the boundaries within which the main costmap can be modified by the function.

  This function is used to update the cost values of the main costmap within a specific region. It is called by every layer of the costmap during their update process when `LayeredCostmap` calls `updateMap`. The function ensures that each layer can modify the cost values within the defined boundaries, allowing for a composite costmap that reflects the contributions of all individual layers. `LayeredCostmap` calls this function one by one for each layer, so the order in which these layers are registered is important.

- `virtual void updateBounds(double robot_x, double robot_y, double robot_yaw, double* min_x, double* min_y, double* max_x, double* max_y) = 0;`

  - **robot_x, double robot_y, double robot_yaw**: Represent the position and orientation of the robot to which this layer belongs within the `LayeredCostmap`.
  - **double* min_x, double* min_y, double* max_x, double* max_y**. They are the references to the limits.

- `virtual void onInitialize() = 0;`
  It configures and initializes the layer's parameters and data structures, such as connecting to topics, creating data structures, or defining other necessary components. This initialization occurs once when the layer is inserted into the `LayeredCostmap`.

7.1 Functions

7.1.1 The updateBounds function

The developed layer utilizes bounds similar to those used by the `static_layer`, which typically writes the costmap based on the map received from the map server. These bounds cover the entire costmap area and do not adjust based on the robot's position or orientation. To achieve this, the `useExtraBounds` function is invoked. This function takes into account any additional restrictions imposed by other layers, which may limit the area further. Its primary role is to translate the coordinates of these bounds from the map's perspective to the world frame of reference.

7.1.2 The onInitialize function

Each robot must initialize its layer properly to ensure it correctly receives data from the Robot_position_info_manager node and sets the necessary parameters accordingly. Firstly, it is necessary to establish a connection for this layer to receive data. This is achieved by creating an object of type `ros::ServiceClient`, which will be invoked each time the map needs to be updated. This `ros::ServiceClient` provides a direct connection to the `getOtherRobotsInfo` service of the `Robot_position_info_manager` node, as detailed in Chapter 6. In this function, the layer retrieves information on how to manage and process data. To properly configure this layer, it is necessary to pass parameters through the configuration file of the `global_costmap` .YAML file. An example can be found in Listing 7.1. The parameters that can be used to configure the layer are three:

- **robot_name**: Specifies the name of the current robot. This parameter is typically used to identify the robot to the `Robot_position_info_manager`

and to make requests for information via the `getOtherRobotsInfo` service. If the node fails to retrieve this information because it doesn't exist or due to any other issue, a default empty string is chosen as the name. Setting an empty string as the parameter for the `getOtherRobotsInfo` service means requesting all available data from the node.

- **other_robot_radius**: This parameter defines the radius in meters of the circle around each other robot's position that marks surrounding areas as occupied on the map. Essentially, it extends the occupied region beyond each robot's exact position, creating a buffer zone where these areas are considered as occupied. If this parameter is omitted or not loaded correctly, it will be set to 0 by default, meaning no cells will be marked as occupied in the robot's position. A demonstration of this parameter is shown in Figures 7.1, 7.2.

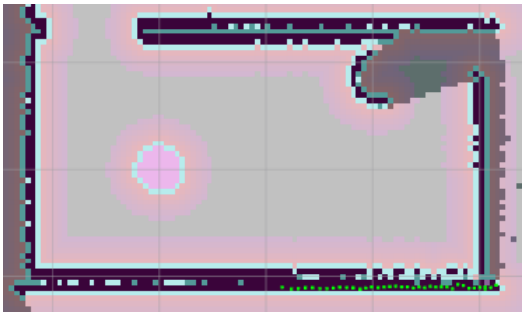


Figure 7.1: Representation of a robot created by another robot with the Positioning Layer active and an `other_robot_radius` set to 0.2

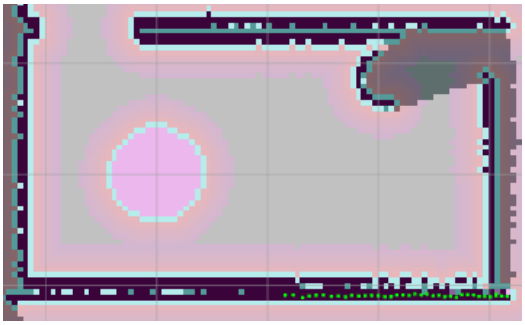


Figure 7.2: Representation of a robot created by another robot with the Positioning Layer active and an `other_robot_radius` set to 0.4

- **discard_radius**: This parameter specifies the radius around the robot where information from other robots is discarded or not considered. It's not exactly a radius but rather half the side length of a square constructed around the robot. If this parameter is omitted or not loaded correctly, it will be set to 0 by default, meaning the discard area is disabled. A representation of the discard_radius is shown in Figure 7.3.

Listing 7.1: An example of how set up paramter for the positioning_layer

```
global_costmap:
  positioning_layer:
    robot_name: name_of_the_robot
    other_robot_radius: 0.2
    discard_radius: 0.6
```

7.1.3 The updateCosts function

The core function of this layer resides in this function. It is responsible for integrating various local costmaps into the global costmap and marking cells occupied by other robots. The first task it performs is to check if the `discard_radius` is activated (`discard_radius` >= 0). If active, the function attempts to retrieve information about its own position from the `Robot_position_info_manager` node. If the layer fails to obtain this information or if it does not exist, the layer disables itself even if `discard_radius` >= 0. The rationale for this step will be explained shortly. After that, this function makes a request to the `Robot_position_info_manager` node to retrieve information about other robots. If the service call fails, `updateCost` does not modify the global costmap in any way and triggers an error message. Once the data are received, the layer processes each robot individually, extracting information such as the position, resolution, and dimensions of each local costmap one by one. The critical step involves centering the local costmap precisely at the robot's location. This requires accurately translating the indices of the local costmap, represented as a matrix, into real-world coordinates, and then converting them to indices suitable for the global costmap matrix. To begin, we calculate the corresponding starting cell in the global costmap that corresponds to the first cell of the local costmap. This is achieved using the `worldToMap` function provided by `costmap_2d::Costmap2D`, which converts points from world coordinates to matrix indices. Given the robot's position in real-world coordinates, adjusting for half the real length and width of the local costmap (computed by multiplying the number of rows and columns of the local costmap by its resolution) gives us the starting point where modifications to the global costmap should begin. To obtain the indices in the global costmap you simply utilize the `worldToMap` function. Once these indices are found, it will be necessary to copy the content from the local costmap to the global costmap represented by `master_grid`. Before using the `setCost` function on `master_grid`, two conditions must be checked:

- Ensure that the local costmap is within the bounds of the global costmap. If it exceeds these bounds, refrain from writing to the global costmap to prevent segmentation faults during the write operation.
- If the `discard_radius` is activated and the preliminary checks are successful, avoid writing to cells near the robot that is updating the global costmap within the square area defined by the discard_radius. This precaution prevents cells directly under the robot from being marked as occupied, potentially hindering its movement. This situation can occur when the robot updating the global costmap appears in the local costmap of another robot.

Only cells marked as definitely occupied in the local costmap are copied into the global costmap.

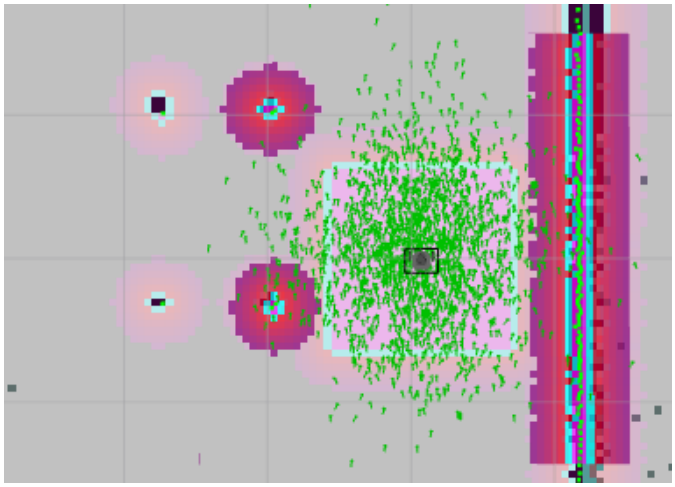


Figure 7.3: The square in the image represents the clearance area where the costmap of other robots cannot write information about the presence of obstacles with discard_radius = 0.6

7.1.4 Layer Integration Process

This layer is responsible for retrieving the position and local costmap of other robots from the `Robot_position_info_manager` node. It marks as occupied the cells where other robots are located, and copies the local costmaps of these robots into its own global costmap (Figure 7.4). However, it avoids copying cells that are too close to its own robot, ensuring that the robot's movement is not hindered. To mitigate errors rather than improve the quality of information, it is necessary to insert this layer before the inflation layer adds padding around obstacles. This provides a higher error margin, which can help to compensate for inaccuracies in localization or distance measurements by the robot. The reason why the local costmap and global costmap need to have the same resolution is to avoid issues related to pixel scaling, as there is no rescaling of the local costmap to match the resolution of the global costmap.

7.2 Configuration and Correlation with Robot_position_info_manager node

It is important to understand the two key parts of this framework. The list of robots that are subscribed to Robot_position_info_manager node are the robots that decide to send data about position and their local_costmap. Every robot that wants to receive data from another robot has to install this ROS new package containing this layer and insert it into its own layers the positioning layer. The

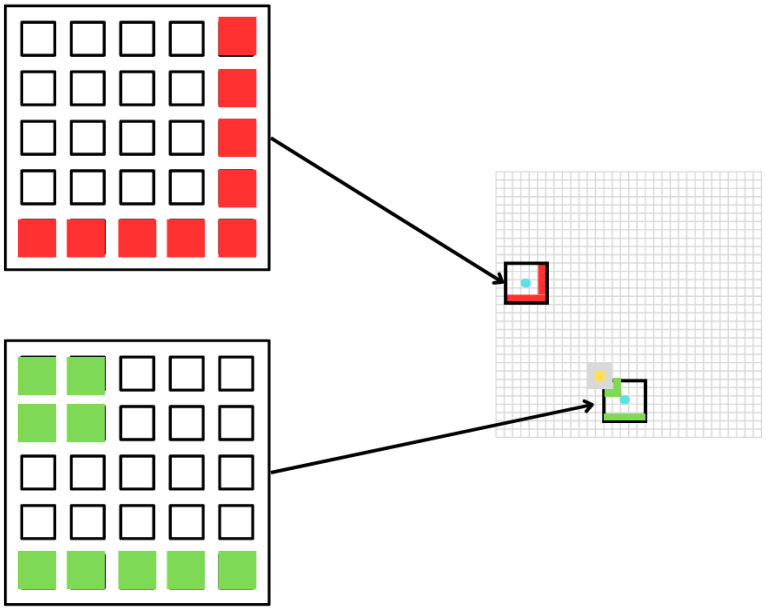


Figure 7.4: In this image, the functionality of the layer is demonstrated. On the left, the local maps where two robots have detected obstacles that are marked in red and green. On the right, these areas are shown as they are integrated into the layer. The blue dots represent the areas where the robots are positioned, marking those locations as occupied. The local costmap with green obstacles is not fully written because part of it is too close to the robot updating the global costmap (yellow dot). The discard area, where data is not written, is marked in light gray

robot_name parameter of this layer is necessary for configuring well this framework. If the robot is subscribing in the list of Robot_position_info_manager node source, the two name parameters are properly set to the same name to avoid conflict. In fact, if the names are different, the robot receives the same position as itself and is marked as occupied by the cells close to him, thus creating problems during the path planning of the robot. A single robot can independently decide whether to receive only data about other robots and add it to our global map without sending information about itself, or to send only data without receiving information from others. It is important, for example, to exclude data received from robots with lower precision. It is necessary that a robot with less quality of data is not placed in the Robot_position_info_manager node source list. If a robot only wants to receive the local costmap, simply set `other_robot_radius` to 0 or omit it from the layer's parameters. Similarly, if the discard area should not be activated, set `discard_radius` to 0 or omit it. Diagram is shown in Figure 7.5

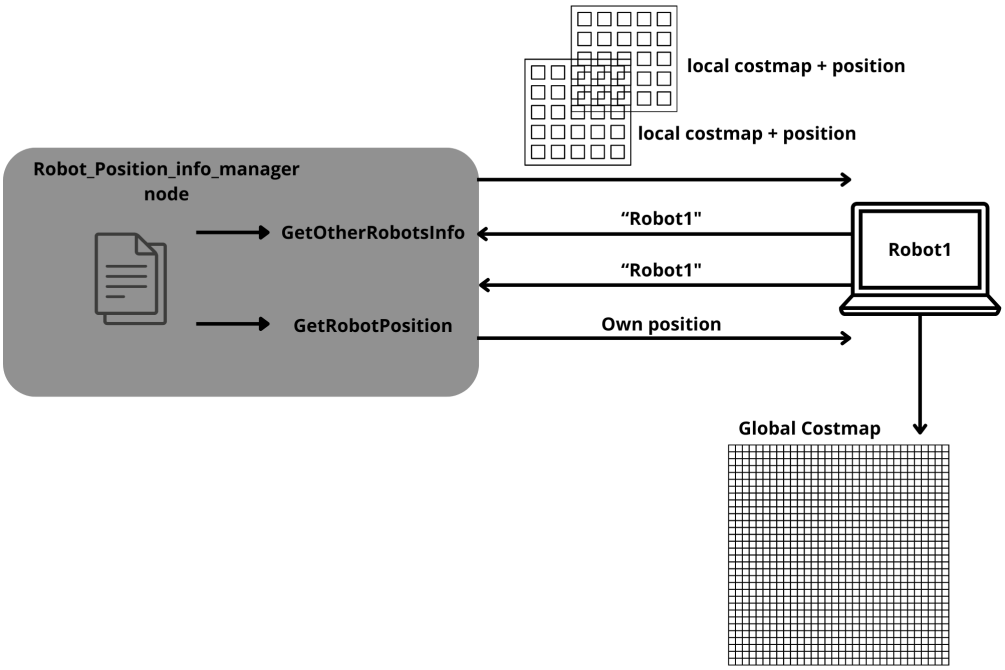


Figure 7.5: A representation of what the Positioning layer does and how it communicates with the Robot_position_info_manager

Chapter 8

Software configuration

This Chapter illustrates all the configurations used by the robots. The map creation is considered done with all the precautions that were discussed in Chapter 5. This configuration is used both in simulation and in real application

8.1 Robots configuration

SLAM and localization package

The Locobot WX250s uses the RTAB-Map package for creating maps and localizing itself in the environment. The robot is sold with this software integrated and provides nodes for creating and storing maps, as well as a node that localizes the robot in the environment during the localization phase. This node is able to fuse the data received from the RPLIDAR A2 and the Intel® RealSense™ Depth Camera D435 to create a better representation of the environment. As discussed in Chapter 5, this robot was chosen with RTAB-Map to create the map, with some differences between the lab and the simulated environment. While in the simulated environment using Gazebo, creating the map with `use_lidar:=false` and `MaxObstacleHeight` still resulted in a high-quality map, in the real environment, setting `use_lidar:=false` and thus creating the map using the Intel® RealSense™ Depth Camera D435 resulted in a map that was too noisy and unusable. Therefore, it was decided to create the map using the LiDAR of the LoCoBot to make the best use of RTAB-Map's capabilities.

Global Planner

The files provided by the manufacturer of the Locobot use NavFn as the default global planning algorithm. As mentioned in Chapter 3, this package uses the Dijkstra algorithm as default to evaluate the best path. This implies that the

results obtained from navFn, although better [34], may require more time for processing [52]. In this scenario, where the global map can change very dynamically, it is important to generate a global path as quickly as possible. For this reason, it has been chosen the `global_planner` package that offers various options including the possibility to use the A* algorithm to evaluate the path, requiring less time for finding a path with performance similar to the performance offered by navFn [34]. Actually, based on the tests conducted, the resulting environment is not very dynamic. However, due to the nature of the package, which frequently updates the global costmap, having a faster global planner is advantageous.

Local Planner

The files provided by the manufacturer of the Locobot use `base_local_planner/TrajectoryPlannerROS` as the default local planning algorithm. This package [53] is a new implementation of old `dwa_local_planner` [54] that improves this last one, makes it more modular and allows, for example, adding your own cost function. As mentioned in Chapter 3, there exist other approaches, each of which brings various benefits and strengths. It is important to understand which environment you want to simulate. In the tests, the environment can change over time, but once a path is traced, it is difficult to change. Generally, in the case of a highly dynamic environment, the TEB algorithm is preferred. For the tests, it is not necessary to have high execution speed at the expense of precision and smoothness, and for this reason, it has been chosen to maintain the algorithm originally provided by Trossen Robotics.

Configuration files

It is therefore necessary to make modifications to the `move_base_params.yaml` configuration file so that the move_base node belonging to the Locobot WX250s correctly incorporates all the changes discussed earlier. Table 8.1 shows these modifications.

The global and local planners also have configuration files. In Table 8.2, `use_dijkstra` is set to false to force the global planner algorithm to use A* as the global planning algorithm. `use_grid_path` set to false is used to make the path less jagged and smoother. The configuration files of the local planner have not undergone any changes compared to the original configuration file.

The other two configuration files are properly set as follows: `global_costmap_all_params.yaml` and `local_costmap_all_params.yaml`. In these two files, it is necessary to increase the `update_frequency` and `publish_frequency` to enhance the frequency at which the robot updates its internal data of the costmap and publishes the costmap, respectively. In the local costmap, it is crucial to set `publish_frequency` equal to `update_frequency` to

Parameter	Value
controller_frequency	10
controller_patience	3
planner_frequency	2
planner_patience	5
oscillation_timeout	10
oscillation_distance	0.2
base_local_planner	"base_local_planner/TrajectoryPlannerROS"
base_global_planner	"global_planner/GlobalPlanner"

Table 8.1: Parameter table for move_base Locobot

allow_unknown	false
use_dijkstra	false
use_grid_path	false

Table 8.2: Parameter table for global_planner Locobot

make the local costmap available to other robots as soon as possible. Another important parameter to set is the resolution of both the local and global costmaps, ensuring this value is shared across all costmaps and robots. For the global costmap, it is essential to add the `position_layer` as cited in Chapter 7 to incorporate the local costmap of the Turtlebot into the Locobot's global costmap.

```
global_costmap:
  global_frame: map
  update_frequency: 2.0
  publish_frequency: 2.0
  positioning_layer:
    robot_name: locobot
  plugins:
  - {name: static_layer, type: "costmap_2d::StaticLayer"}
  - {name: positioning_layer, type: "
    positioning_layer_namespace::PoseLayer"}
  - {name: laser_layer, type: "costmap_2d::ObstacleLayer"}
  - {name: depth_layer, type: "costmap_2d::ObstacleLayer"}
  - {name: inflation_layer, type: "costmap_2d::InflationLayer"}
```

Listing 8.1: the global_costmpa_all_params.yaml file used by the Locobot

```
local_costmap:
  update_frequency: 4.0
  publish_frequency: 4.0
```

```
  rolling_window: true
  width: 4.0
  height: 4.0
  resolution: 0.05
  plugins:
  - {name: laser_layer, type: "costmap_2d::ObstacleLayer"}
  - {name: depth_layer, type: "costmap_2d::ObstacleLayer"}
  - {name: inflation_layer, type: "costmap_2d::InflationLayer"}
```

Listing 8.2: the local_costmpa_all_params.yaml file used by the Locobot

8.1.1 Turtlebot configuration

SLAM and localization package

The Turtlebot does not use any SLAM algorithm, as the map considered when using this robot is the one generated during the Locobot's SLAM session. Regarding localization, we have chosen to use the AMCL package, which is provided by the robot's manufacturers. To simplify matters, both in simulation and in the laboratory, the initial localization point is always correct. This is to avoid incorrect updates of the local costmap to the Locobot, which would affect its global costmap.

Global Planner and Local Planner

The same considerations made for the LoCoBot can be applied to the Turtlebot. The Turtlebot uses `navfn` and `dwa_local_planner` as the global and local planner algorithms, respectively. However, unlike the choices made for the Locobot, default values were retained for the Turtlebot's algorithms to emphasize the diversity and heterogeneity between the two robots.

Configuration files

Just as with the Locobot, configuration files have been changed for the Turtlebot, despite the Turtlebot using default algorithms for local and global planning. The main files modified here are those that adjust parameters for the global costmap and local costmap, namely `global_costmap_params.yaml` and `local_costmap_params.yaml`. As discussed in Chapter 5, these files were adapted to accommodate the addition of new layers, including the `positioning_layer` in the global costmap for this robot as well. Additionally, the resolution of both the local and global costmaps, as well as their update and publication frequencies, were also modified, according to the value used by the Locobot.

Listing 8.3: the global_costmpa_params.yaml file used by the Turtlebot

```
global_costmap:
```

```
  global_frame: map
  robot_base_frame: base_footprint

  update_frequency: 2.0
  publish_frequency: 2.0
  transform_tolerance: 0.5

  positioning_layer:
    robot_name: turtlebot

  plugins:
    - name: static_layer
      type: "costmap_2d::StaticLayer"

    - name: positioning_layer
      type: "positioning_layer_namespace::PoseLayer"

    - name: obstacle_layer
      type: "costmap_2d::ObstacleLayer"

    - name: inflation_layer
      type: "costmap_2d::InflationLayer"
```

Listing 8.4: the local_costmpa_params.yaml file used by the Turtlebot

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_footprint

  update_frequency: 2.0
  publish_frequency: 2.0
  transform_tolerance: 0.5

  rolling_window: true
  width: 3
  height: 3
  resolution: 0.05

  plugins:
    - name: obstacle_layer
      type: "costmap_2d::ObstacleLayer"

    - name: inflation_layer
      type: "costmap_2d::InflationLayer"
```

Chapter 9

Simulation and experimental results

This Chapter explores the evaluation of the `robot_position_info_manager` node and the Positioning Layer in both simulated and real-world environments. The primary focus is on assessing the functionality and performance of the application under controlled conditions. In the simulated environment, tests were conducted using the Turtlebot house model in Gazebo, designed to simulate a simple household environment suitable for to testing robot navigation and perception capabilities. These tests involve scenarios where robots operate in separate areas, unable to directly observe each other, yet exchanging critical position and obstacle information. The simulation setup ensures that the robots can accurately model their surroundings and transmit relevant data to each other's global costmaps, crucial for collaborative path planning and obstacle avoidance.

9.1 Working in a simulated environment

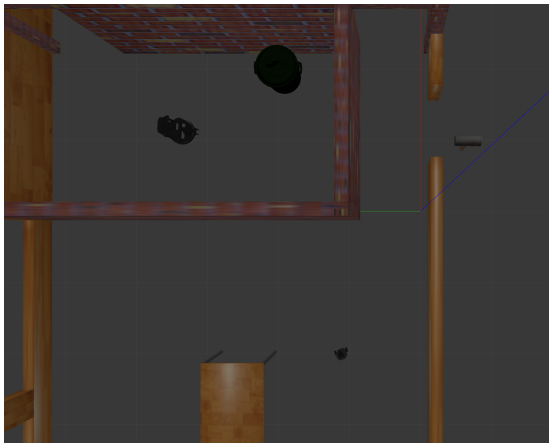
9.1.1 Simulation Setup

This Section examines the tests conducted to evaluate the robot_position_info_manager node in a simulation environment. The primary objective is to analyse the effectiveness and correct operation of the application, as well as to discuss the results obtained during the testing process. All the tests are conducted in a simulated environment using Gazebo. The environment used is the Turtlebot house that represents a simple house. It was important for these tests to have an environment that allowed the two robots to be in two areas not directly visible to each other. The map generation is already done, using the process described in Chapter 8.

9.1.2 Test results

Test 1

The first test evaluates how the robots exchanged their positions reciprocally and that, on its own global costmap, each robot indicated the cells where the other robot is located as occupied. Figures 9.1, 9.2, 9.3 show such functionality.



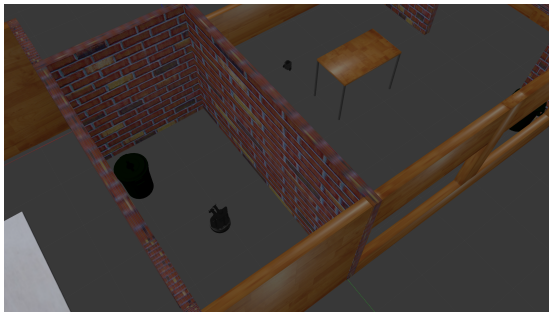


Figure 9.1: The two robots are positioned in such a way that they cannot directly see each other. In fact, there is a wall between the two robots.

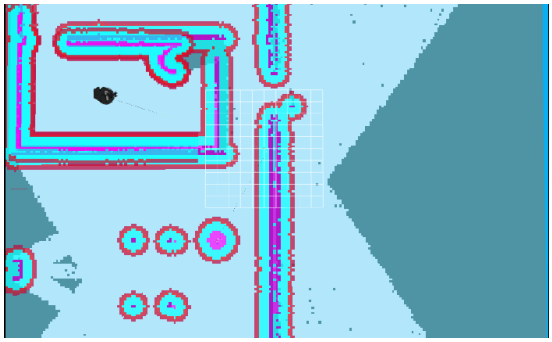


Figure 9.2: The Locobot RVIZ visualization.The Locobot, in the global costmap, marks with occupied the cells where the Turtlebot is located

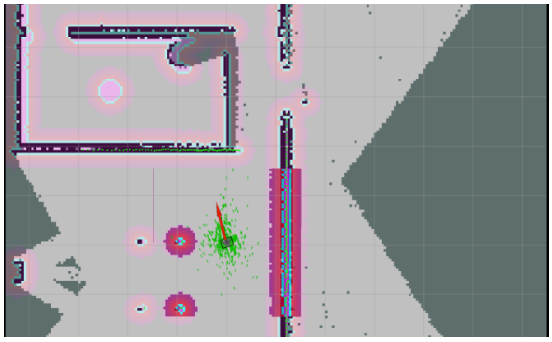


Figure 9.3: The Turtlebot RVIZ visualization. The Turtlebot, in the global costmap, marks with occupied the cells where the Locobot is located

Test 2

The second test verifies that the robots shared their local costmap reciprocally and put in the correct way the local costmap in their own global costmap. To verify that, a cylindrical object has been put near the Turtlebot and a large cube near the Locobot (Figure 9.4). The objects are placed in such a way that the objects or part of them are placed inside the local costmap. Only the surface of the object that the robot can detect by its sensors is transmitted to the other robot. Figure 9.5 shows the local costmap of the Locobot. It is possible to notice the cube, represented by a line located to the right of the robot. The same line is visualized by the Turtlebot in own global costmap, as shown in Figure 9.8. A path planning will then take this information into consideration. In a similar way, it is shown in Figures 9.6 and 9.7, how the Turtlebot detects the upper part of the cylinder and this information is found in the global costmap of the Locobot.

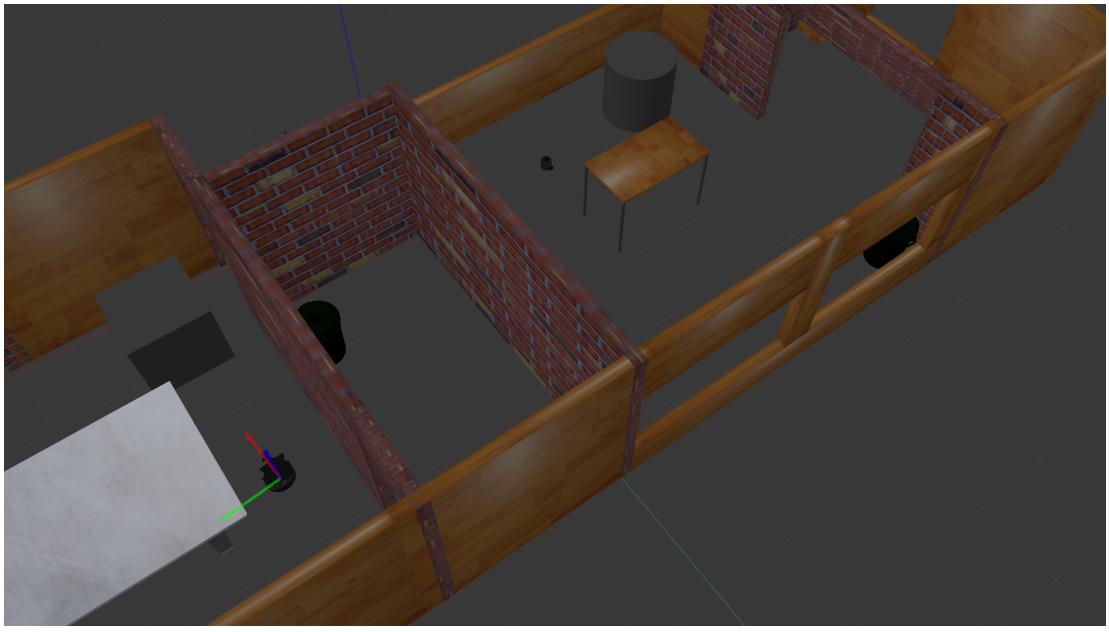


Figure 9.4: The two robots are positioned in such a way that they cannot directly see each other. In fact, there is a wall between the two robots. A Cube is positioned near the Locobot and the cylinder is positioned near the Turlebot. The Turtlebot can't see the cube and the Locobot cannot see the cylinder

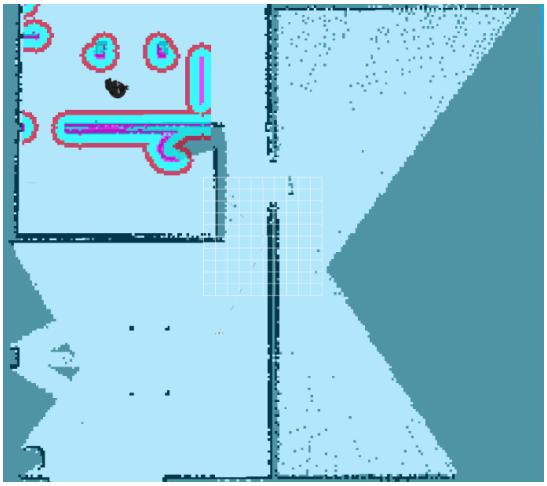


Figure 9.5: Local costmap of Locobot

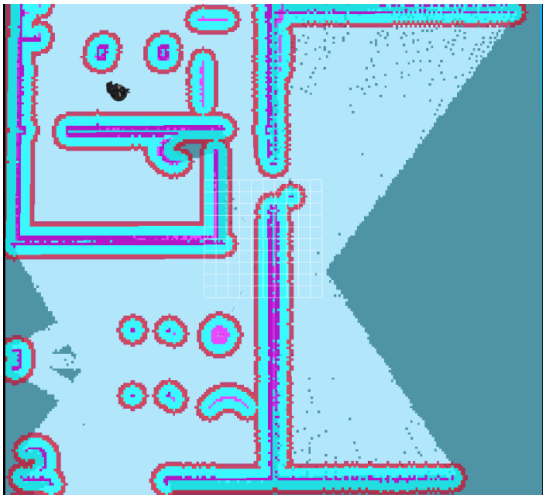


Figure 9.6: Global costmap of Locobot

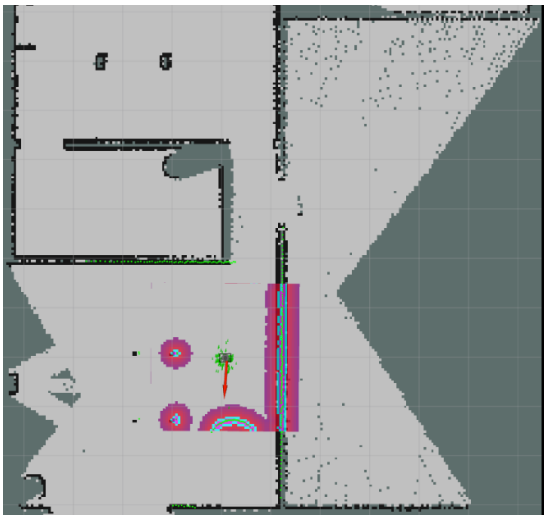


Figure 9.7: Local costmap of Turtlebot

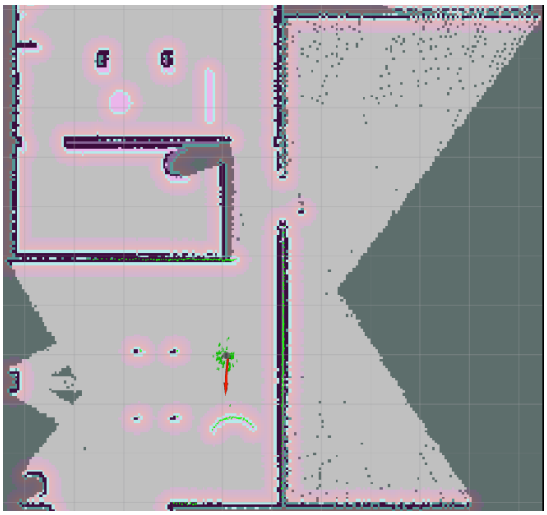


Figure 9.8: Global costmap of Turtlebot

Test 3

The last test shows a scenario where the robot cannot perceive the obstacles but they can be detected by another robot that have the possibility to see that object. For this reason, a new wooden cube of 10 cm has been added in Gazebo. The cube, as can be seen in Figures 9.9, 9.11, is invisible to the Turtlebot, which mounts a

LiDAR sensor placed at a height of 18 cm. The Locobot, is equipped with RGB-D cam with which it can, despite its height, correctly visualize the obstacle. In this scenario, the Locobot not only anticipates information about obstacles that the Turtlebot would typically encounter only during path execution, but also identifies obstacles that are completely invisible to the Turtlebot. Without the Locobot detecting these obstacles on its behalf, a collision that the Turtlebot would be unable to foresee would have occurred. To test that, a set of cubes have been placed in a line, and the Locobot was positioned to observe these cubes (Figures 9.10, 9.12, 9.13, 9.14). The positions of these obstacles were added to our local costmap and transmitted to the Turtlebot, which incorporated them into its own global costmap. This allows the Turtlebot to plan a path while considering these otherwise invisible cubes, as shown in Figure 9.15.

Figure 9.9: Comparison between the cube and the Turtlebot

Figure 9.10: The Locobot is in position for identify the cubes.

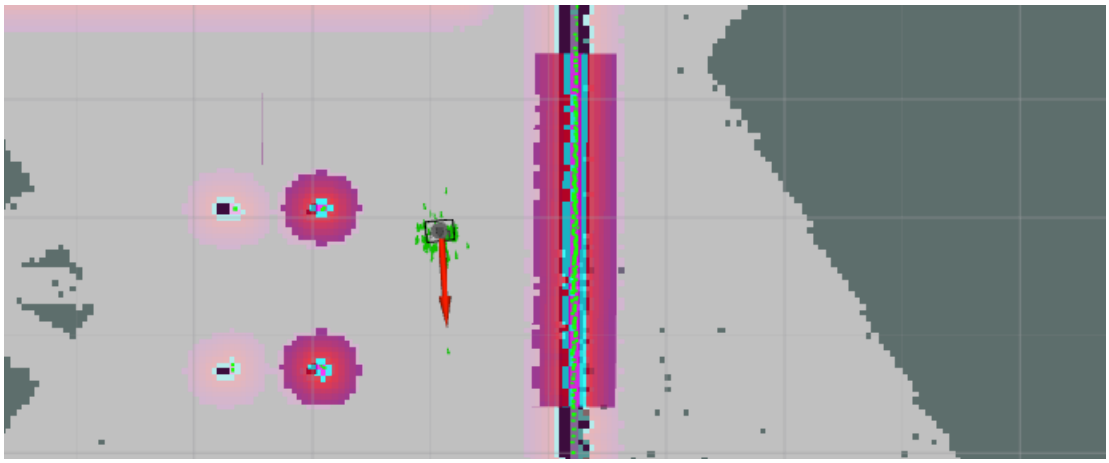


Figure 9.11: The cube is invisible for the Turtlebot

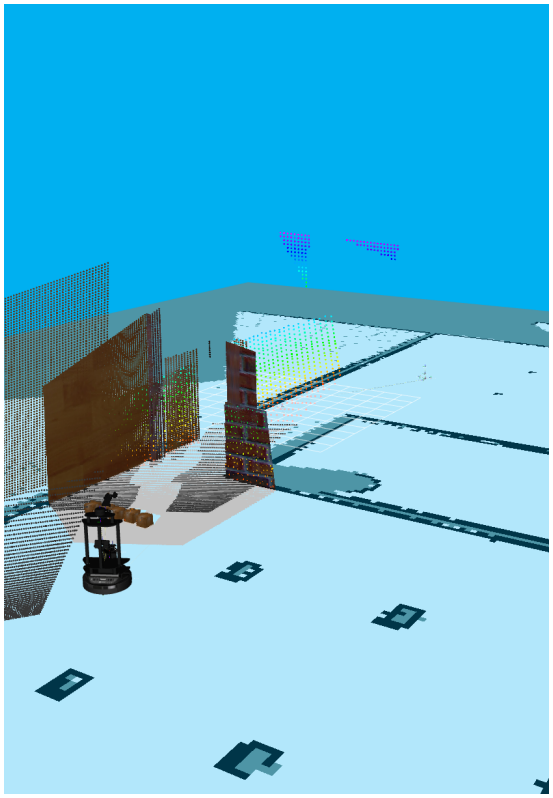


Figure 9.12: The representation recreated by RTAB-Map of the environment that the Locobot can see through the RGB-D cam

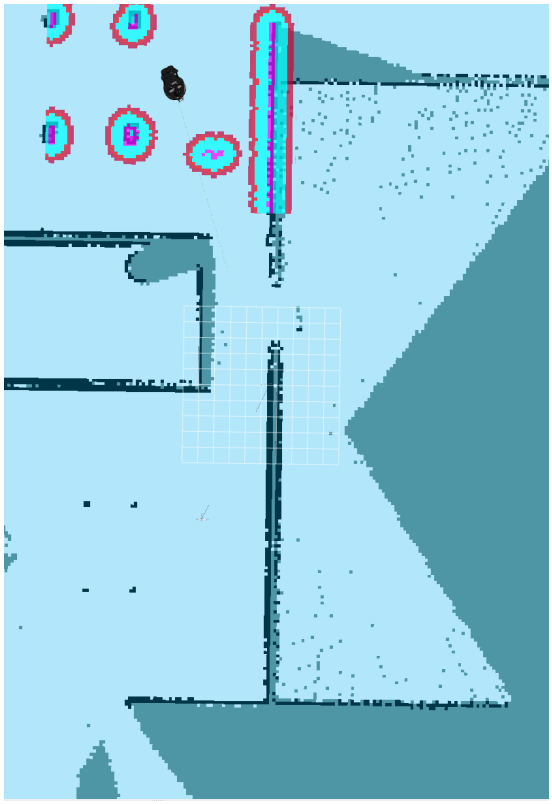


Figure 9.13: The local costmap created by the Locobot

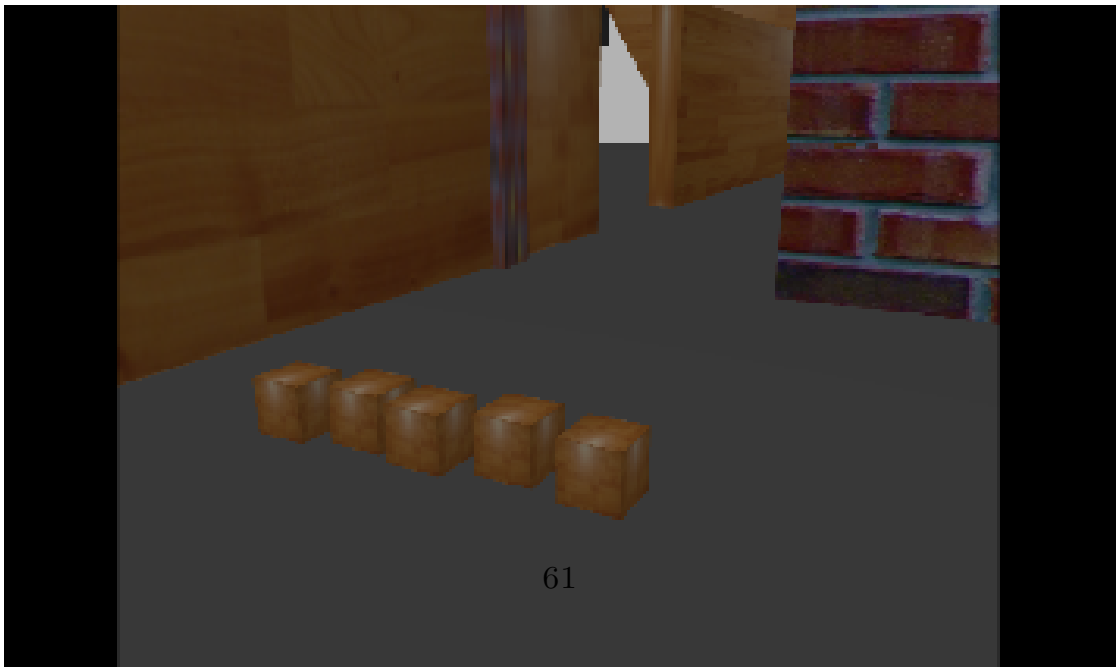


Figure 9.14: The view of the camera positioned on the robot

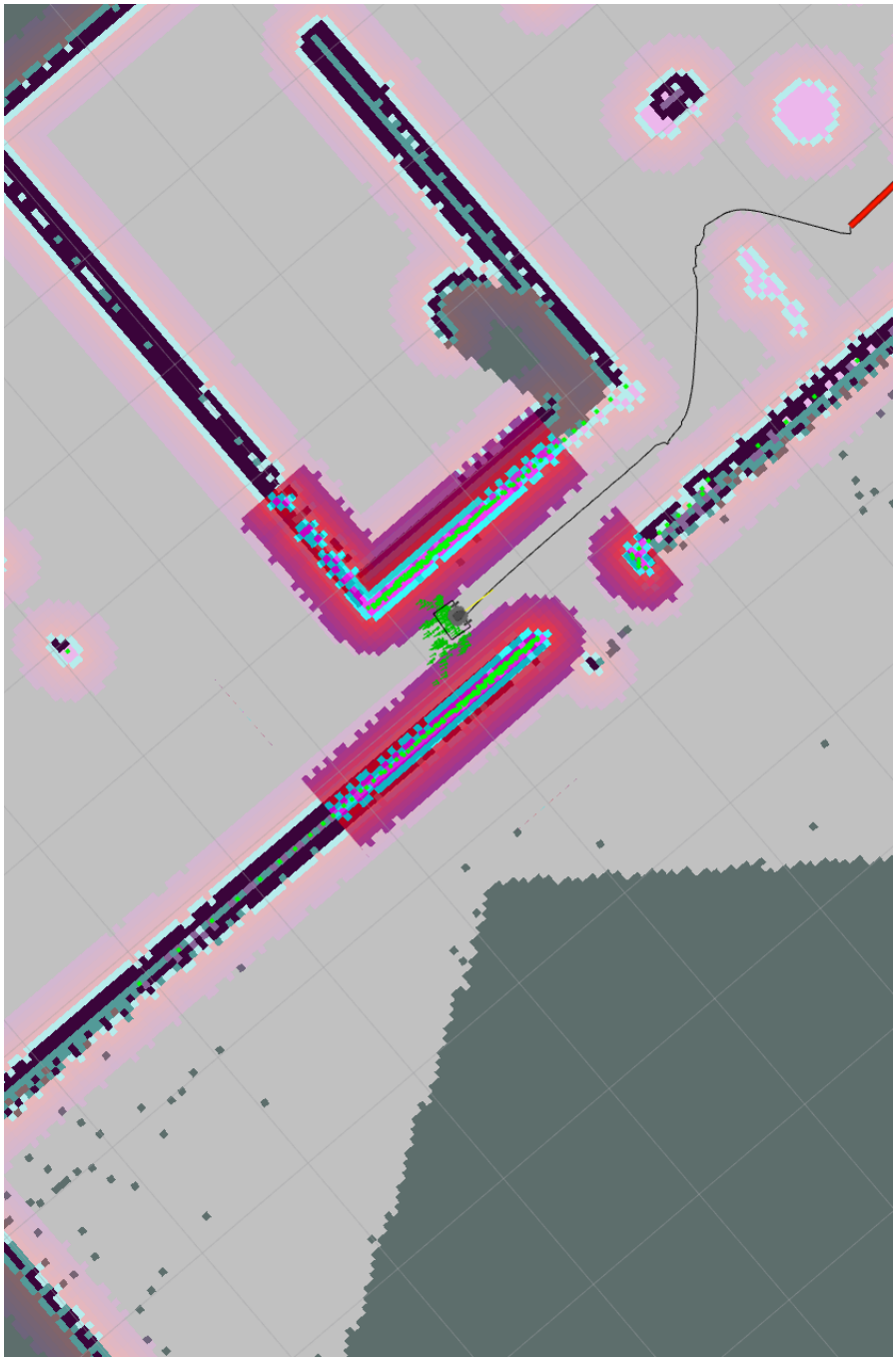


Figure 9.15: The Turtlebot creates the free-collision path, including the information sent by the Locobot

9.2 Working in real environment

9.2.1 Laboratory Setup

This section examines the tests conducted to evaluate the `robot_position_info_manager` node in a real-world laboratory environment. The primary objective, as with the ROS simulation part, is to analyze the effectiveness and correct operation of the application, as well as to discuss the results obtained during the testing process. The tests performed are the same as those conducted during the simulation phase. The environment for these tests is a laboratory, where two distinct areas were constructed to ensure that the robots could not observe each other directly (Figure 9.16), and each robot could have access to information that only it could observe. To create a barrier, cardboard boxes were used, with heights exceeding that of the Locobot's LiDAR (Figure 9.17) . The Turtlebot will be positioned on one side, while the Locobot will be on the one. It is important that the initial orientation of the robots is the same to ensure that the local costmaps of both robots are correctly aligned with respect to the global costmap. The map generation is already done, using the process described in Chapter 8, specifically for a real-world environment.

Figure 9.16: The initial setup of the experiments in the laboratory

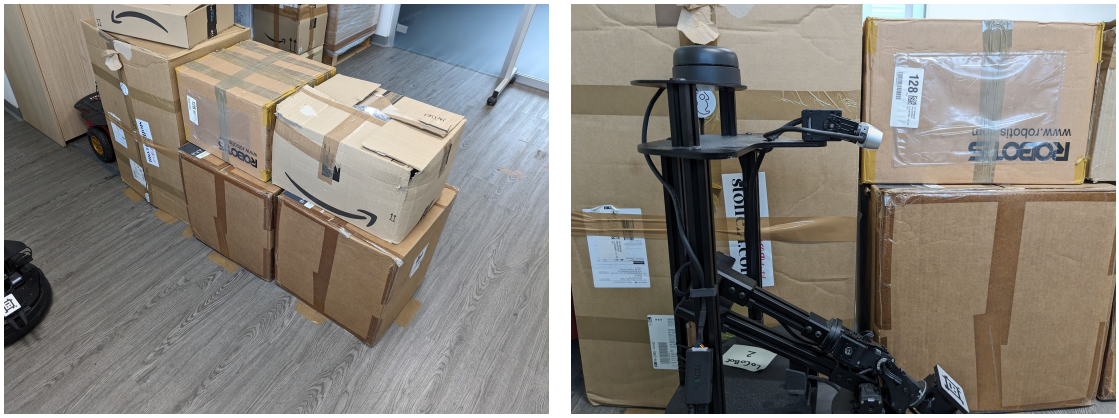


Figure 9.17: On the left, the illustration of the wall created to separate the spaces between the two robots; on the right, the comparison of the Locobot's height relative to that of the wall.

9.2.2 Test results and performance

Test 1

The first test conducted was to evaluate how the robots exchanged their positions reciprocally and that each robot indicated, on its own global costmap, the cells where the other robot is located as occupied. Figures 9.18, 9.19, 9.20, 9.21 show such functionality.

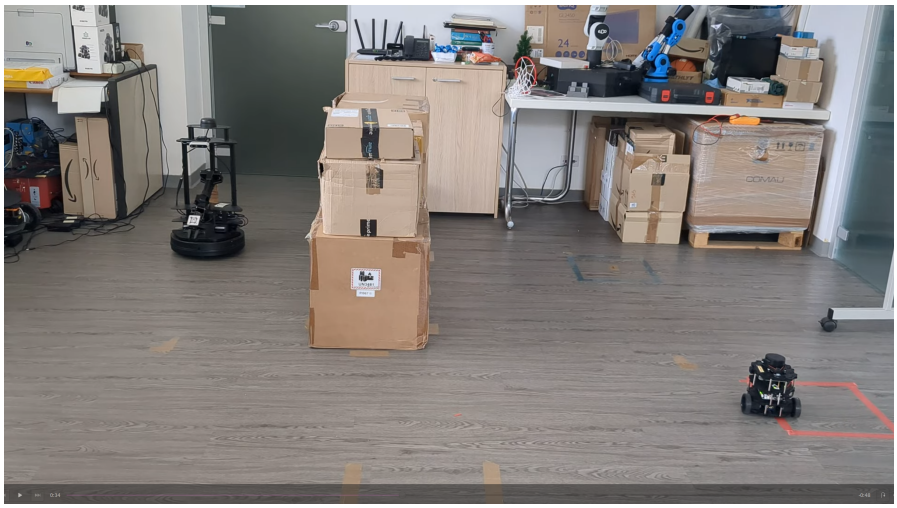


Figure 9.18: In this image, the Turtlebot is positioned closer to the camera than its initial position.

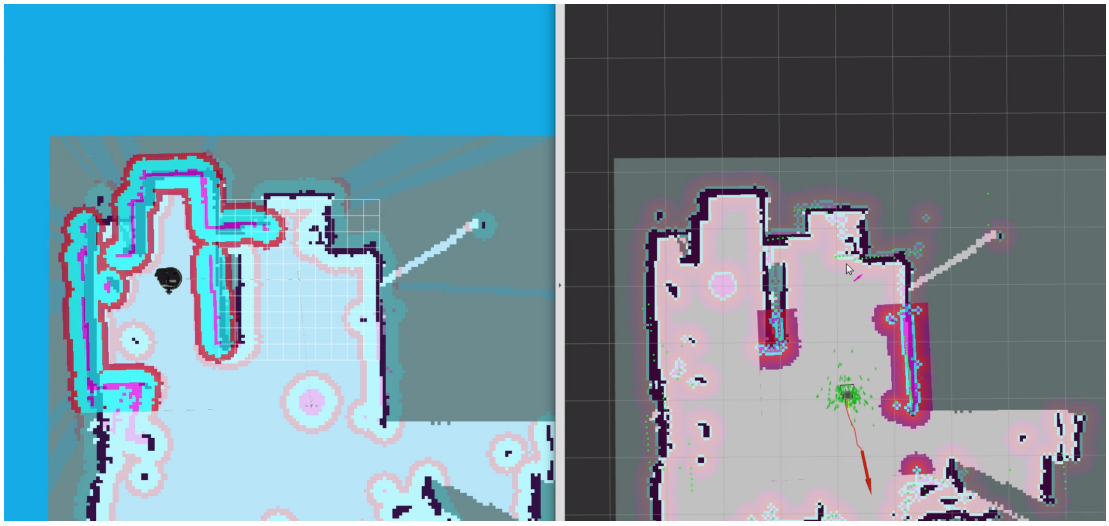


Figure 9.19: On the left, the global costmap and local costmap of the Locobot; on the right, those of the Turtlebot. As can be observed, it mirrors the view in Image 9.18, and the Turtlebot receives information about the Locobot's position, marking the cells where it detects the Locobot as occupied. Similarly, the Locobot performs the same action.

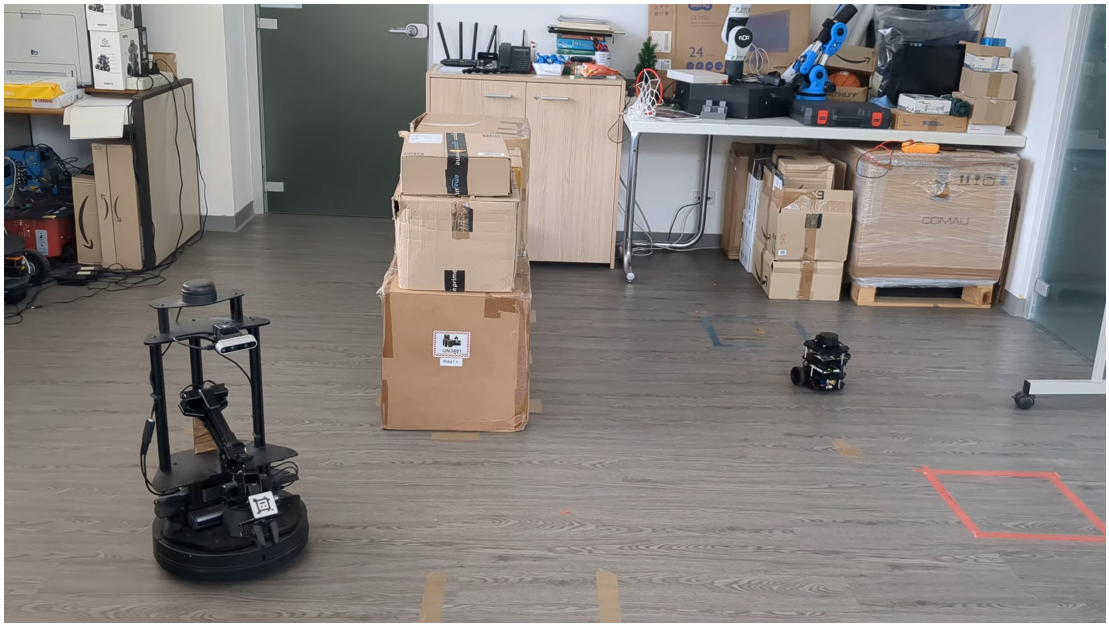


Figure 9.20: In this image, the Locobot is positioned closer to the camera than its initial position

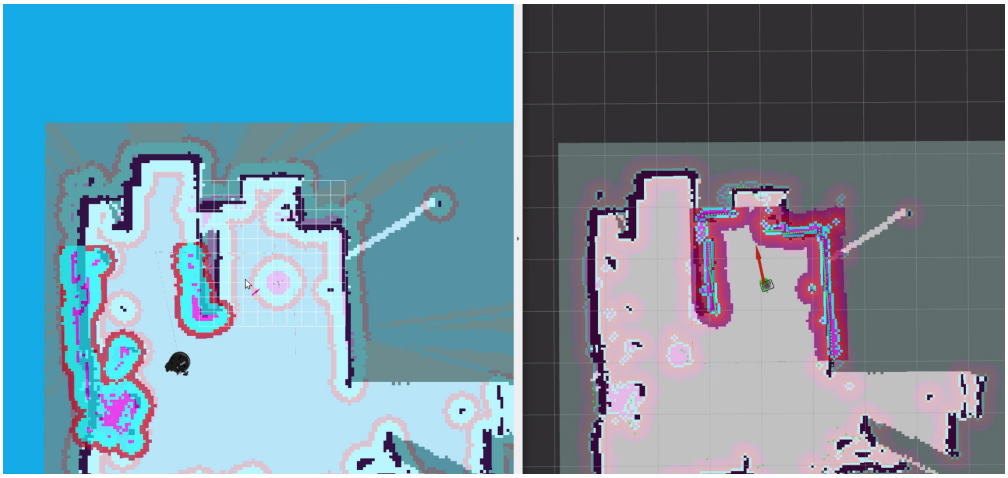


Figure 9.21: On the left, the global costmap and local costmap of the Locobot; on the right, those of the Turtlebot. As can be observed, it mirrors the view in Image 9.20, and the Turtlebot receives information about the Locobot's position, marking the cells where it detects the Locobot as occupied. Similarly, the Locobot performs the same action.

By disabling the discard area on both robots, setting the `discard_radius = 0`, it is possible to verify that the information exchanged between the robots is correct, as there will be overlap between what each robot sees and what is transmitted by the other robot. Evidence of what has been said can be observed in Figure 9.22.

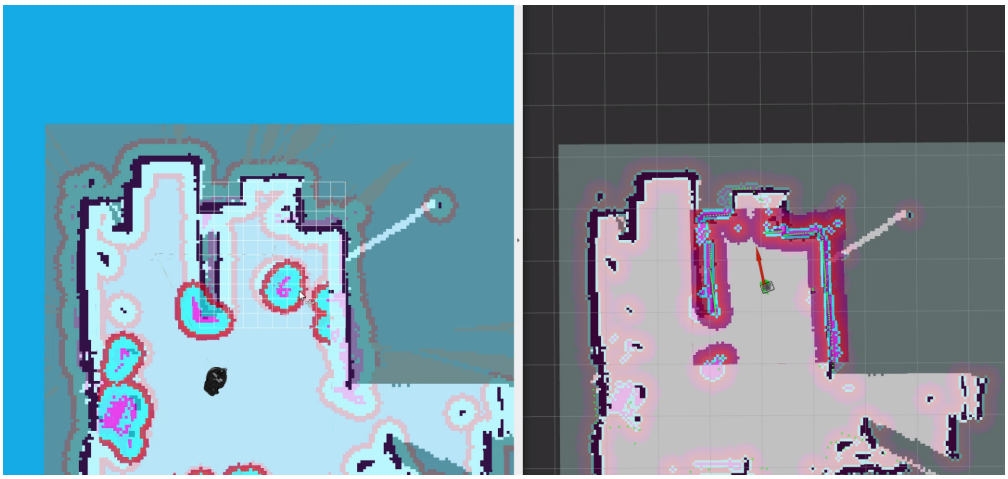


Figure 9.22: Matching between the information sent by the other robot and the information obtained by the other robot.

Test 2

The second test is to verify that the robots shared their local costmap reciprocally and put in correct way the local costmap in the own global costmap. To verify that, a box has been added near the Turtlebot (Figure 9.23). During path execution, the Turtlebot encounters the obstacle. When the box appears within its local costmap, information about the obstacle's presence is transmitted to the Locobot, which can utilize this information during path planning. Here, a challenge was identified: although the `global_planner` algorithm is faster, it often generates paths that are too narrow and close to obstacles, leading to collisions. For this reason, the global planning algorithm chosen was the one provided directly by the robot manufacturer, as discussed in Chapter 8. Figure 9.24 shows when the Turtlebot positions itself with respect to the obstacle, and the Locobot receives and incorporates this information into its global costmap. A new route is then set for the Locobot considering the new obstacle. As seen in Figures 9.25, 9.26 the Locobot will create a path that anticipates both the obstacle and the Turtlebot's position at that moment.

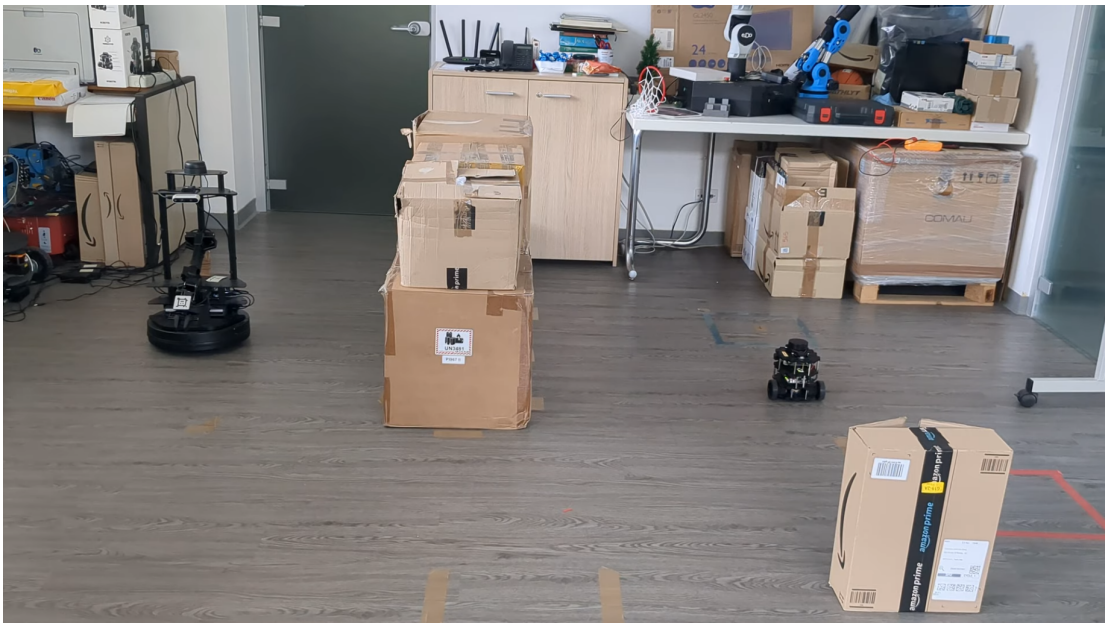


Figure 9.23: The initial setup of the second test in the laboratory.

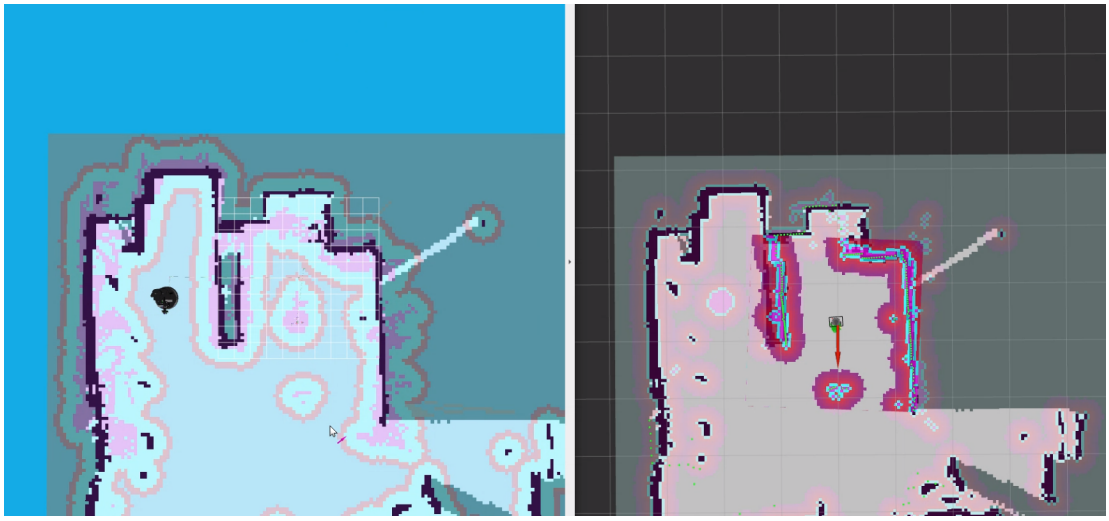


Figure 9.24: The Turtlebot is near the new obstacle as shown in Image 9.23 and is correctly displayed in its own local costmap.

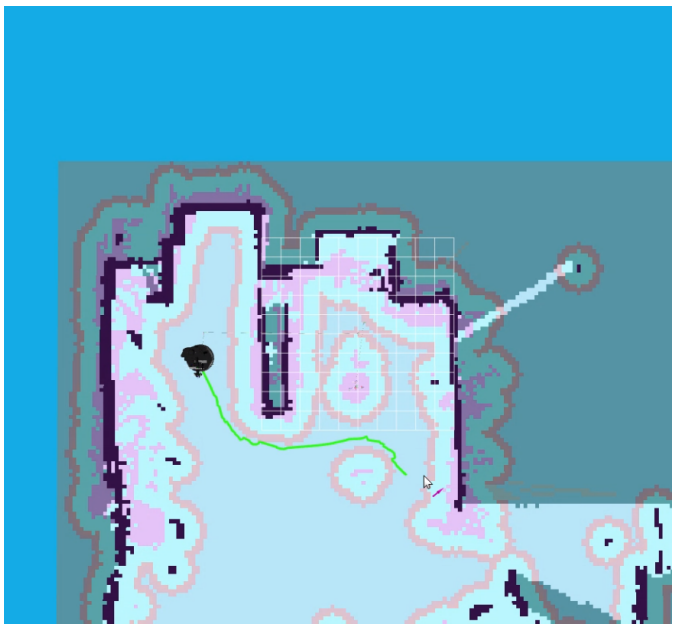


Figure 9.25: The Locobot plans a trajectory, taking into account what the Turtlebot previously observed. The Locobot would have also noticed the obstacle, but only once it was in front of it. In this case, however, it manages to plan a preemptive trajectory even before starting the path execution.

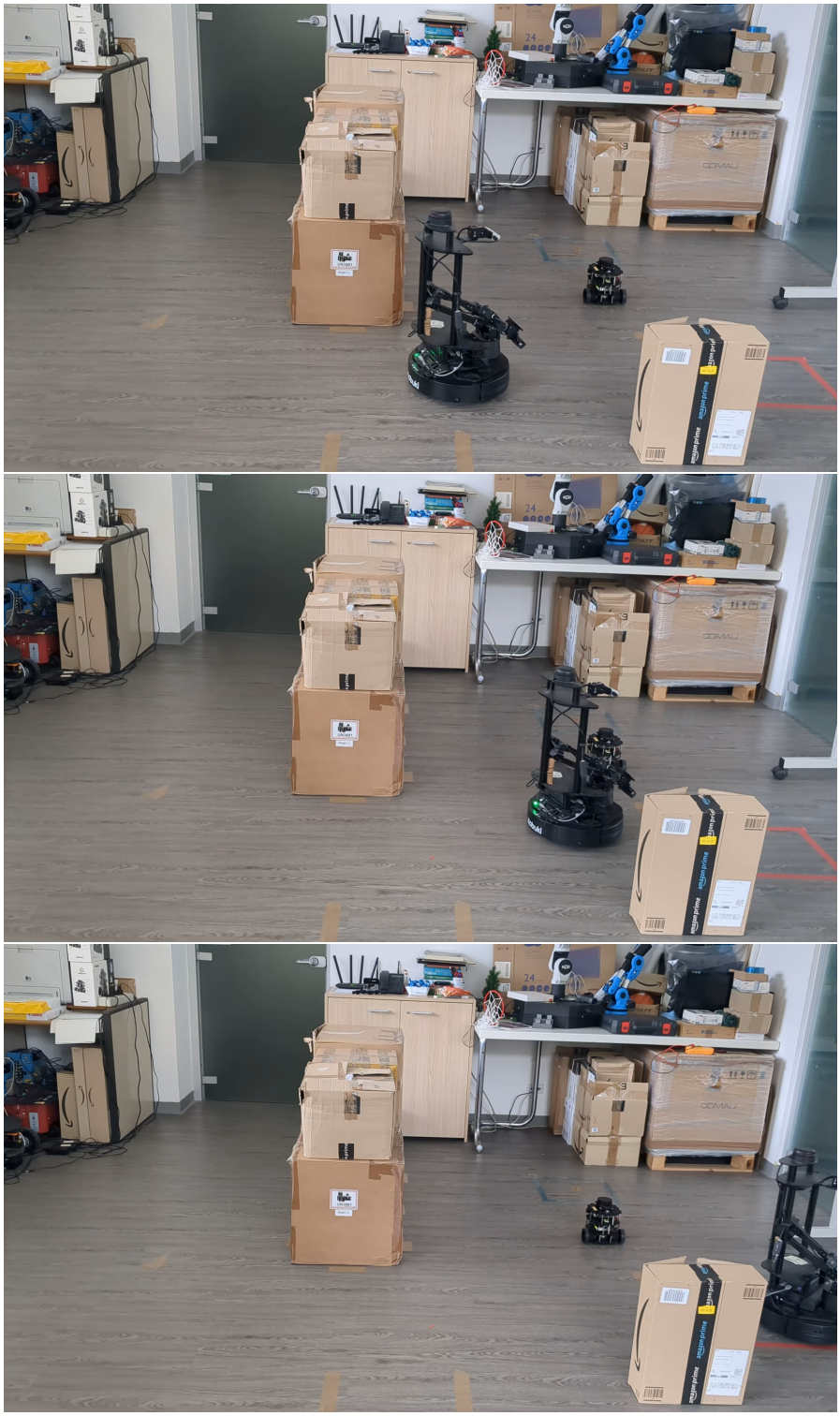


Figure 9.26: The execution of the Locobot's path, which correctly avoids the object.

Test 3

The last test conducted revealed that the obstacles that one robot cannot see can be detected by another robot, which has the capability to see the object. For this reason, a 16 cm high box was placed in front of the Turtlebot, as can be seen in the Figure 9.28. The box is invisible to the Turtlebot (Figure 9.27), which mounts a LiDAR sensor placed at a height of 18 cm. If the Turtlebot traces a path towards a point crossing the box, it does not have the ability to avoid the obstacle and will therefore collide with it, as can be observed in Figure 9.29. It is possible to see that the Locobot, equipped with RGB-D cameras, can correctly visualize the obstacle despite its height. In this case, the robot not only perceives information that is visible only when the path is being executed, but can also avoid a collision that could not have been avoided otherwise. The objective is therefore to bring the Locobot closer so that it can see the obstacle and then transmit the information to the Turtlebot, enabling it to avoid the obstacle.

Figure 9.27: The Turtlebot LiDAR is positioned higher than the height of the box.

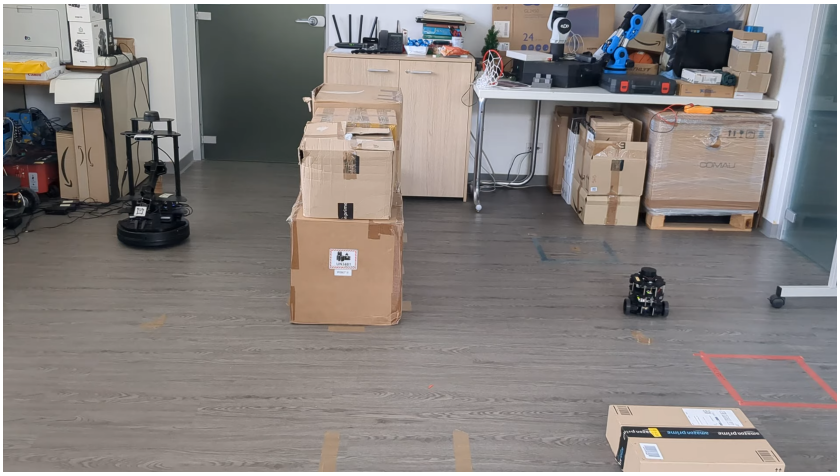


Figure 9.28: The initial setup of the third test in the laboratory.

Figure 9.29: The path created by the TurtleBot does not take the box into account because it cannot perceive it.

During the trials of this test, several issues arose. When the LiDAR detects an obstacle, it appears much more "stable" in the costmaps compared to how it is visualized using the RGB-D camera. In fact, there is a difficulty in ensuring that the Locobot visualizes the obstacle in a stable manner. While the Locobot is moving towards the obstacle, the obstacle appears sufficiently clear, although the quality with which it is represented in the costmap is quite poor, almost never allowing the obstacle's shape to be recognized, resulting in a very distorted figure. This may be sufficient for the Locobot itself to navigate around the obstacle while in motion, but it is almost useless for transmitting information about the obstacles to another robot. Despite this, as shown in Figures 9.30, 9.31, 9.32, and 9.33, it is possible to help the Turtlebot avoid an obstacle invisible to it thanks to the Locobot, though not without difficulties. Compared to the simulation, where the RGB-D camera behaves ideally, it is less effective in reality. Additionally, during the execution, a limitation arose. In the case of large obstacles, the various sensors are not able to accurately capture their depth. This does not create a significant problem for obstacles that both robots can observe, as the information, even if partial, can help the other robots to anticipate problems caused by the obstacles. However, for obstacles that are invisible to other robots, partial information may not be sufficient for the robot to correctly avoid the obstacle.

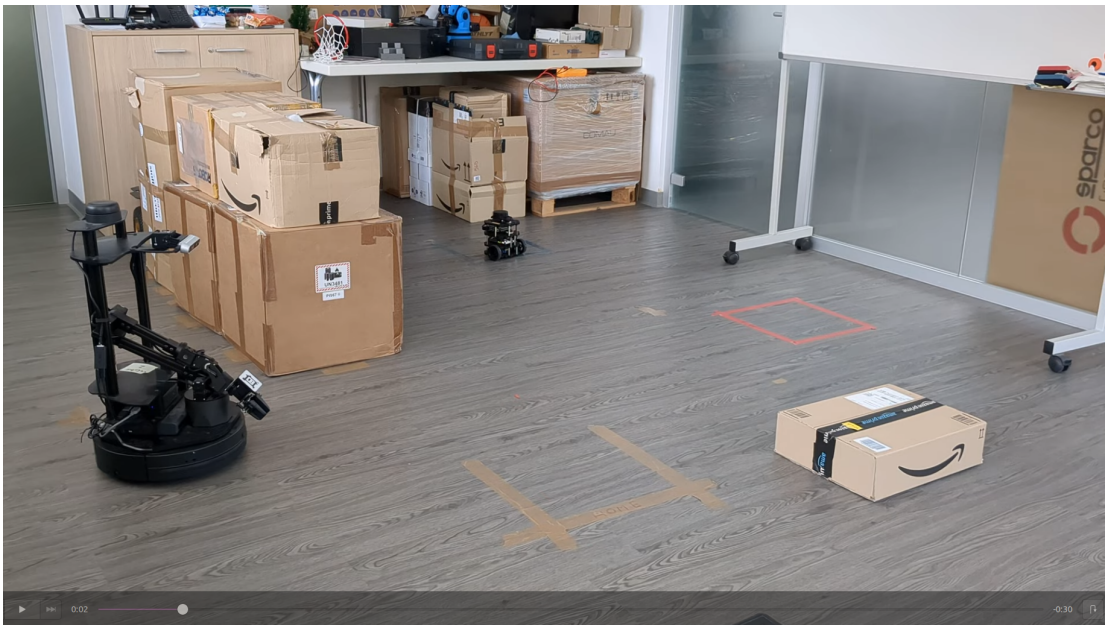


Figure 9.30: The Locobot positioned itself to successfully localize the obstacle.

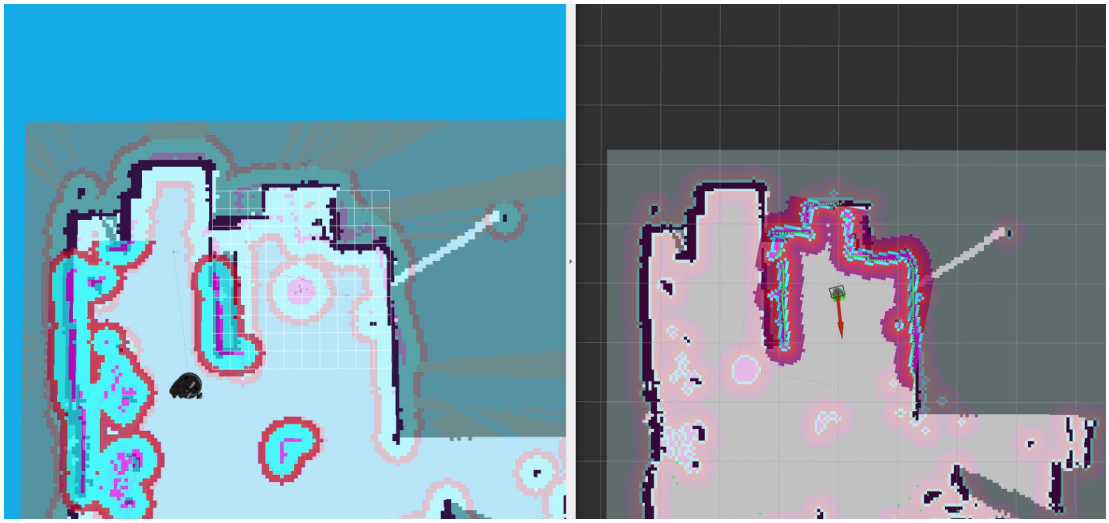


Figure 9.31: On the left, the Locobot places in its local costmap what it sees of the obstacle. On the right, the Turtlebot places in its global costmap the information received from the Locobot, even though the obstacle is completely invisible to the Turtlebot itself.

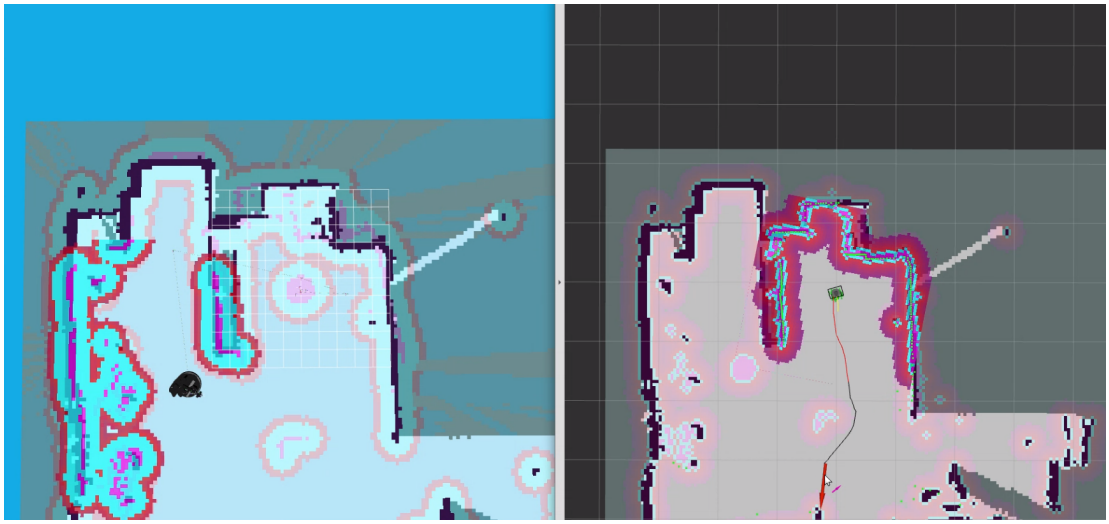


Figure 9.32: The Turtlebot creates a path, taking into account the information received from the Locobot, thus avoiding an obstacle that it could not see on its own.

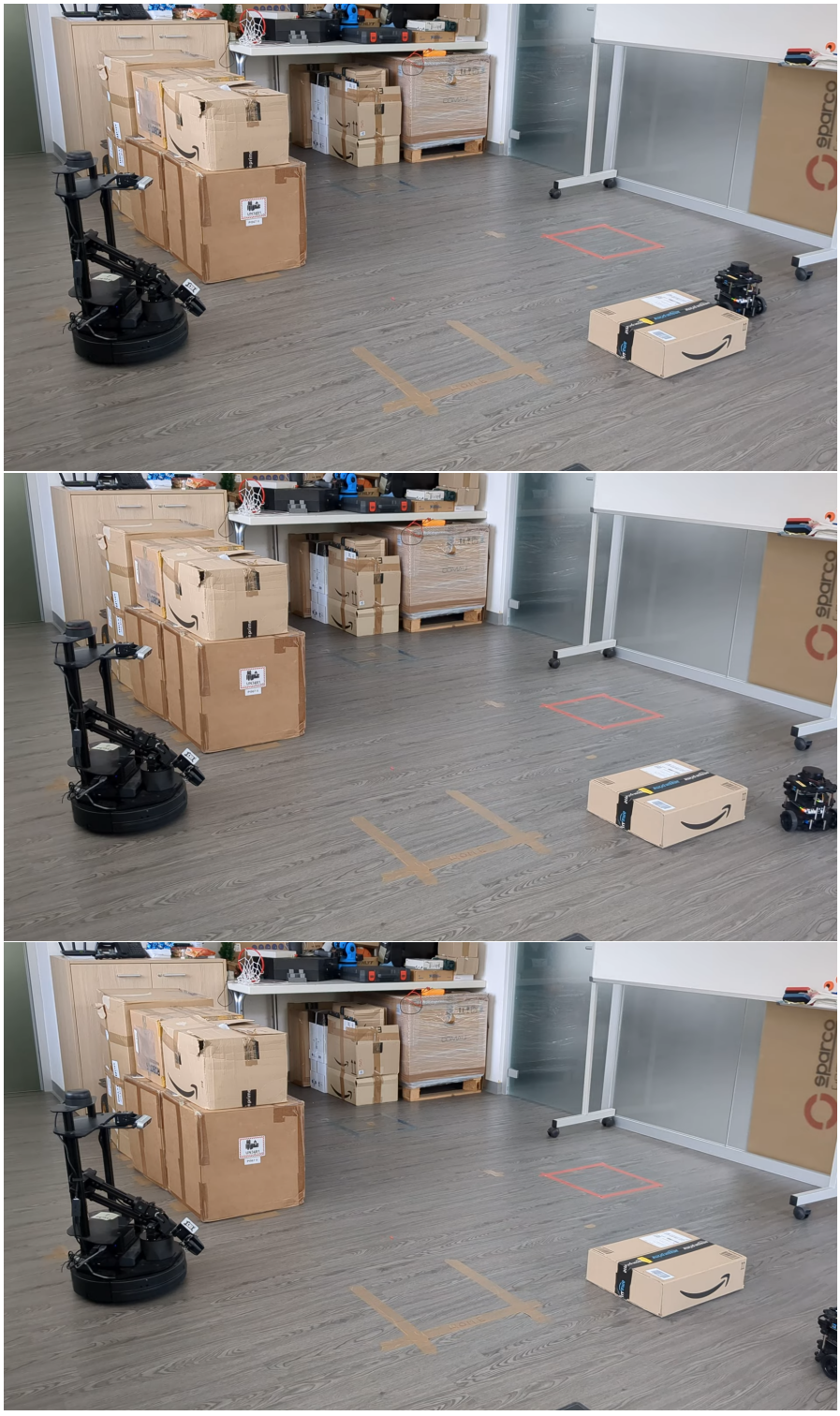


Figure 9.33: The execution of the Turtlebot's path in Image 9.32, which correctly avoids the object.

Chapter 10

Conclusions and future works

The thesis aims at developing a ROS framework that facilitates information sharing among robots of different types, enhancing early obstacle identification. This framework will allow robots that cannot directly observe obstacles to receive information from other robots with obstacle detection capabilities. The framework consists of two distinct components: `robot_info_manager_node` and the `positioning_layer`.

- `robot_info_manager_node` is responsible for collecting and forwarding information to all other robots. This node collects data on the positions of robots and their local costmaps
- The `positioning_layer` incorporates the local costmaps of other robots into its own global costmap. It achieves this by placing them in the correct position and marking the cells where other robots are located.

This new ROS package, which includes the layer and the `robot_position_info_manager` node, facilitates to share useful information between multiple robots. This information sharing is beneficial for anticipating obstacles and has several other advantages. It not only aids in identifying obstacles, but also significantly enhances path planning by providing real-time positions of other robots, allowing for trajectory planning in advance, avoiding collision. Regarding the exchange of heterogeneous information, the main issue during my tests seemed to stem from the low precision of the Locobot's RGB-D camera. For more accurate obstacle identification, LiDAR is much more effective due to its precision and stability, and it does not need to be directly pointed at the obstacle. As mentioned in Chapter 9, this information will always be partial

since only the side of the obstacle visible to the robot is detected. This partial view can be mitigated by having multiple robots observe the environment from different angles. For instance, a team of robots like the Turtlebot, equipped with LiDAR positioned low enough, can support other robots by acting as sentinels, patrolling the work environment. In this way the obstacle will be tracked by multiple robots from different perspectives, obtaining a more accurate picture of the situation in the environment. However, RGB-D cameras are useful for directly identifying people and marking the cells where the person is detected as occupied. Utilizing people detection as cited in [16], combined with the `social_navigation_layers::ProxemicLayer`, which adds Gaussian costs around the detected person increasing in the direction of their motion, allows for transmitting important information to other robots that cannot identify humans themselves. The simulated and experimental tests indicate that this shared information is useful. However, the current layer does not account for the dynamics of obstacles or the robots themselves. Tools similar to the `social_navigation_layers::ProxemicLayer`, which feature an elongated Gaussian shape in the direction of movement, can help to provide a more dynamic approach to obstacle information. This is also applicable to non-human obstacles, such as the robots themselves, which could utilize the same concept for navigating around other moving robots. It is possible to achieve this by adapting the ProxemicLayer input topic to receive not only the position and velocity of humans but also that of robots. This information can be obtained whenever a new robot position is received by comparing the previous position with the current one to estimate direction and velocity before updating the data. Additionally, it is possible to enrich the information exchanged with the `robot_position_info_manager` node, for example, by including the complete paths of all robots. This would enable path planning that avoids collisions based on these paths. Using an approach based on MES [21] can help to better understand who is crossing the work environment and when, optimizing the deployment of robots for patrolling. In any case, the exchange of the local costmap remains fundamental for sharing all this information.

Bibliography

[1] StockCake. *Automated Warehouse Robots*. Accessed: 2024-07-02. 2024. URL: https://stockcake.com/i/automated-warehouse-robots_835120_933116 (cit. on p. 2).

[2] Wikipedia contributors. *Robot Operating System*. https://en.wikipedia.org/wiki/Robot_Operating_System. Accessed: 2024-06-03. 2024 (cit. on pp. 1, 13).

[3] Open Source Robotics Foundation. *ROS Navigation Stack*. Accessed: 2024-07-02. 2024. URL: http://wiki.ros.org/navigation (cit. on p. 1).

[4] Yara Rizk, Mariette Awad, and Edward W Tunstel. «Cooperative heterogeneous multi-robot systems: A survey». In: *ACM Computing Surveys (CSUR)* 52.2 (2019), pp. 1–31 (cit. on p. 4).

[5] Peter Stone and Manuela Veloso. «Multiagent systems: A survey from a machine learning perspective». In: *Autonomous Robots* 8 (2000), pp. 345–383 (cit. on p. 4).

[6] Murat Köseoğlu, Orkan Murat Çelik, and Ömer Pektaş. «Design of an autonomous mobile robot based on ROS». In: *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*. IEEE. 2017, pp. 1–5 (cit. on p. 5).

[7] Hamid Taheri and Zhao Chun Xia. «SLAM; definition and evolution». In: *Engineering Applications of Artificial Intelligence* 97 (2021), p. 104032 (cit. on p. 5).

[8] Yassin Abdelrasoul, Abu Bakar Sayuti HM Saman, and Patrick Sebastian. «A quantitative study of tuning ROS gmapping parameters and their effect on performing indoor 2D SLAM». In: *2016 2nd IEEE international symposium on robotics and manufacturing automation (ROMA)*. IEEE. 2016, pp. 1–6 (cit. on p. 6).

[9] Lili Mu, Pantao Yao, Yuchen Zheng, Kai Chen, Fangfang Wang, and Nana Qi. «Research on SLAM algorithm of mobile robot based on the fusion of 2D LiDAR and depth camera». In: *IEEE Access* 8 (2020), pp. 157628–157642 (cit. on p. 6).

[10] Mubariz Zaffar, Shoaib Ehsan, Rustam Stolkin, and Klaus McDonald Maier. «Sensors, slam and long-term autonomy: A review». In: *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE. 2018, pp. 285–290 (cit. on p. 6).

[11] Lulu Chen, Hong Wei, and James Ferryman. «A survey of human motion analysis using depth imagery». In: *Pattern Recognition Letters* 34.15 (2013), pp. 1995–2006 (cit. on p. 6).

[12] Cyril Robin and Simon Lacroix. «Multi-robot target detection and tracking: taxonomy and survey». In: *Autonomous Robots* 40 (2016), pp. 729–760 (cit. on p. 7).

[13] Wahyu Rahmaniar and Ari Hernawan. «Real-time human detection using deep learning on embedded platforms: A review». In: *Journal of Robotics and Control (JRC)* 2.6 (2021), pp. 462–468 (cit. on p. 8).

[14] Sandro Augusto Magalhães, Luís Castro, Germano Moreira, Filipe Neves Dos Santos, Mário Cunha, Jorge Dias, and António Paulo Moreira. «Evaluating the single-shot multibox detector and YOLO deep learning models for the detection of tomatoes in a greenhouse». In: *Sensors* 21.10 (2021), p. 3569 (cit. on p. 8).

[15] Chloe Eunhyang Kim, Mahdi Maktab Dar Oghaz, Jiri Fajtl, Vasileios Argyriou, and Paolo Remagnino. «A comparison of embedded deep learning methods for person detection». In: *arXiv preprint arXiv:1812.03451* (2018) (cit. on p. 8).

[16] Timm Linder and Kai O Arras. «People detection, tracking and visualization using ros on a mobile service robot». In: *Robot Operating System (ROS) The Complete Reference (Volume 1)* (2016), pp. 187–213 (cit. on pp. 8, 76).

[17] Andrea Bonci, Pangcheng David Cen Cheng, Marina Indri, Giacomo Nabissi, and Fiorella Sibona. «Human-robot perception in industrial environments: A survey». In: *Sensors* 21.5 (2021), p. 1571 (cit. on p. 9).

[18] Marina Indri, Fiorella Sibona, and Pangcheng David Cen Cheng. «Sensor data fusion for smart AMRs in human-shared industrial workspaces». In: *IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*. Vol. 1. IEEE. 2019, pp. 738–743 (cit. on pp. 10, 41).

[19] Emil-Ioan Voisan, Bogdan Paulis, Radu-Emil Precup, and Florin Dragan. «ROS-based robot navigation and human interaction in indoor environment». In: *2015 IEEE 10th Jubilee International Symposium on Applied Computational Intelligence and Informatics*. IEEE. 2015, pp. 31–36 (cit. on p. 10).

[20] Qimin Ren, Qingjie Zhao, Hui Qi, and Lingrui Li. «Real-time target tracking system for person-following robot». In: *2016 35th Chinese Control Conference (CCC)*. IEEE. 2016, pp. 6160–6165 (cit. on p. 10).

[21] Andreas Löcklin, Falk Dettinger, Maurice Artelt, Nasser Jazdi, and Michael Weyrich. «Trajectory Prediction of Workers to Improve AGV and AMR Operation based on the Manufacturing Schedule». In: *Procedia CIRP* 107 (2022), pp. 283–288 (cit. on pp. 11, 76).

[22] Ziyu Zhu, Kongtao Zhu, Zhentan Zheng, Shitao Chen, and Nanning Zheng. «Multi-L: A Novel Multi-Robot Cooperative Localization Method in Indoor Environment». In: *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2022, pp. 2436–2443 (cit. on p. 11).

[23] Praneel Chand and Dale A Carnegie. «Mapping and exploration in a hierarchical heterogeneous multi-robot system using limited capability robots». In: *Robotics and autonomous Systems* 61.6 (2013), pp. 565–579 (cit. on p. 11).

[24] Marina Indri, Fiorella Sibona, and Pangcheng David Cen Cheng. «Sen3Bot Net: A meta-sensors network to enable smart factories implementation». In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. IEEE. 2020, pp. 719–726 (cit. on p. 12).

[25] Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. «Comparing interest management algorithms for massively multiplayer games». In: *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*. 2006, 6–es (cit. on p. 12).

[26] ROS Wiki contributors. *ROS/Concepts*. `http://wiki.ros.org/ROS/Concepts`. Accessed: 2024-06-03. 2024 (cit. on p. 13).

[27] ROS Wiki. *ROS Master*. `http://wiki.ros.org/Master`. Accessed: 2024-06-25 (cit. on p. 14).

[28] David St-Onge and Damith Herath. «The Robot Operating System (ROS1 &2): Programming Paradigms and Deployment». In: *Foundations of Robotics: A Multidisciplinary Approach with Python and ROS*. Springer, 2022, pp. 105–126 (cit. on p. 14).

[29] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. «Exploring the performance of ROS2». In: *Proceedings of the 13th international conference on embedded software*. 2016, pp. 1–10 (cit. on p. 14).

[30] ROS Wiki Contributors. *move_base - ROS Wiki*. http://wiki.ros.org/move_base?distro=noetic. Accessed: 2024-06-03 (cit. on pp. 16, 17).

[31] Zhanying Zhang and Ziping Zhao. «A multiple mobile robots path planning algorithm based on A-star and Dijkstra algorithm». In: *International Journal of Smart Home* 8.3 (2014), pp. 75–86 (cit. on p. 19).

[32] Maram Alajlan and Anis Koubâa. «Writing global path planners plugins in ROS: A tutorial». In: *Robot Operating System (ROS) The Complete Reference (Volume 1)* (2016), pp. 73–97 (cit. on p. 19).

[33] ROS Wiki Contributors. *Navigation Function (NavFn)*. http://wiki.ros.org/navfn. Accessed: 2024-06-10. 2024 (cit. on p. 19).

[34] Alexandros Filotheou, Emmanouil Tsardoulias, Antonis Dimitriou, Andreas Symeonidis, and Loukas Petrou. «Quantitative and qualitative evaluation of ROS-enabled local and global planners in 2D static environments». In: *Journal of Intelligent & Robotic Systems* 98 (2020), pp. 567–601 (cit. on pp. 19, 52).

[35] Adithya Balachandran, Anil Lal, and Pramod Sreedharan. «Autonomous Navigation of an AMR using Deep Reinforcement Learning in a Warehouse Environment». In: *2022 IEEE 2nd Mysore Sub Section International Conference (MysuruCon)*. IEEE. 2022, pp. 1–5 (cit. on p. 19).

[36] B Cybulski, Agnieszka Wegierska, and Grzegorz Granosik. «Accuracy comparison of navigation local planners on ROS-based mobile robot». In: *2019 12th International Workshop on Robot Motion and Control (RoMoCo)*. IEEE. 2019, pp. 104–111 (cit. on p. 19).

[37] Franz Albers, Christoph Rösmann, Frank Hoffmann, and Torsten Bertram. «Online trajectory optimization and navigation in dynamic environments in ROS». In: *Robot Operating System (ROS) The Complete Reference (Volume 3)* (2019), pp. 241–274 (cit. on p. 20).

[38] ROS Development Team. *AMCL Package Documentation*. http://wiki.ros.org/amcl. Accessed: 2024-06-26 (cit. on p. 20).

[39] Mathieu Labbé and François Michaud. «RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation». In: *Journal of field robotics* 36.2 (2019), pp. 416–446 (cit. on pp. 21, 32).

[40] ROS Development Team. *RTAB-Map ROS Package Documentation*. http://wiki.ros.org/rtabmap_ros. Accessed: 2024-06-26 (cit. on p. 21).

[41] ROS Community. *rviz*. http://wiki.ros.org/rviz. Accessed: 2024-07-07. 2024 (cit. on p. 21).

[42] Open Source Robotics Foundation. *Gazebo*. https://gazebosim.org/home. Accessed: 2024-07-07. 2024 (cit. on p. 21).

[43] Trossen Robotics. *Interbotix X-Series LoCoBots Specifications: Hardware*. Accessed: 2024-07-02. 2024. URL: https://docs.trossenrobotics.com/interbotix_xslocobots_docs/specifications.html#hardware (cit. on pp. 24, 25).

[44] ROBOTIS. *TurtleBot3: Official repository for TurtleBot3*. https://github.com/ROBOTIS-GIT/turtlebot3. Accessed 2024-06-05 (cit. on pp. 25, 29).

[45] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. «Robot Operating System 2: Design, architecture, and uses in the wild». In: *Science robotics* 7.66 (2022), eabm6074 (cit. on p. 27).

[46] Open Robotics. *Real-Time Programming*. https://docs.ros.org/en/jazzy/Tutorials/Demos/Real-Time-Programming.html. Accessed: 2024-06-05. 2024 (cit. on p. 27).

[47] Corey Williams and Adam Schroeder. «Utilizing ROS 1 and the turtlebot3 in a multi-robot system». In: *arXiv preprint arXiv:2011.10488* (2020) (cit. on pp. 27, 28).

[48] ROS Answers Community. *Multiple Robots Simulation and Navigation*. https://answers.ros.org/question/41433/multiple-robots-simulation-and-navigation/. Accessed: 2024-06-05. 2012 (cit. on p. 28).

[49] Interbotix Labs. *Interbotix ROS X-Series LocoBots: ROS packages for Interbotix X-Series LocoBots*. https://github.com/Interbotix/interbotix_ros_rovers/tree/main/interbotix_ros_xslocobots. Accessed 2024-06-05 (cit. on p. 29).

[50] Ilze Andersone. «Heterogeneous map merging: State of the art». In: *Robotics* 8.3 (2019), p. 74 (cit. on p. 33).

[51] Kaiyu Zheng. «Ros navigation tuning guide». In: *Robot Operating System (ROS) The Complete Reference (Volume 6)* (2021), pp. 197–226 (cit. on p. 38).

[52] Imen Chaari, Anis Koubaa, Hachemi Bennaceur, Adel Ammar, Maram Alajlan, and Habib Youssef. «Design and performance analysis of global path planning techniques for autonomous mobile robots in grid environments». In: *International Journal of Advanced Robotic Systems* 14.2 (2017), p. 1729881416663663 (cit. on p. 52).

[53] ROS Wiki Contributors. *base_local_planner*. http://wiki.ros.org/base_local_planner. Accessed: 2024-06-11 (cit. on p. 52).

[54] ROS Wiki Contributors. *dwa_local_planner*. http://wiki.ros.org/dwa_local_planner. Accessed: 2024-06-11 (cit. on p. 52).

Acknowledgements

In primo luogo, vorrei ringraziare i miei relatori, la Professoressa Marina Indri e il Dottor Pangcheng David Cen Cheng, per aver creduto in me e per avermi affidato questo progetto. Inoltre, li ringrazio sinceramente per la loro disponibilità nel correggere la mia tesi con rapidità ed efficienza, permettendomi così di rispettare le scadenze previste.

Desidero ringraziare la mia famiglia e i miei cari, per il loro affetto e incoraggiamento. La vostra presenza mi ha dato la forza di superare i momenti più difficili e di raggiungere questo importante traguardo.

Non posso dimenticare i miei compagni di studi, in particolare Luca e Andrea, per la collaborazione, il sostegno reciproco e i momenti di confronto che hanno arricchito questa esperienza.

Infine, ma non per importanza, desidero ringraziare la persona che è mi stata più vicino in questo percorso e che è stata capace di sopportare le mie ansie e le mie paranoie (cosa non facile). La persona su cui so di poter sempre contare: la mia fidanzata Jeje.