Dipartimento di Ingegneria e Architettura

CORSO DI LAUREA IN INGEGNERIA DELL'ENERGIA ELETTRICA E DEI SISTEMI

PROVA FINALE

AN IMPLEMENTATION OF OPEN-RMF IN AN INDUSTRIAL ENVIRONMENT

Laureando: Michele Belletti

Relatore: Stefano Seriani
Correlatori: Roberto Pugliese
Georgios Kourousias

ANNO ACCADEMICO 2024-2025

Contents

Part I

Introduction

This work presents a study on the new system, OPEN-RMF, which is still under active development but has reached a mature enough state to be deployed in mobile robots at Elettra Sincrotrone Trieste.

Chapter 1

Background and motivation

Elettra Sincontrone Trieste[1] is a major research facility that primarily provides users access to two light sources: the synchrotron Elettra and the free electron laser FERMI. In addition to operating these sources, the facility actively researches new technologies and techniques to enhance its capabilities and operations. Many recent advancements are based on robotics, which, especially after the COVID-19 pandemic and with the rapid rise of large language models (LLMs), have enabled new ways for users to interact with the infrastructure, including robot-assisted information retrieval and package delivery services. Previously, the facility developed a global planner, D-Star (D-star based optimized trajectory planner)[18], within the ROS 1 framework to enable robot navigation across large-scale environments. However, as ROS 1 has reached end-of-life, the codebase must now be migrated to ROS 2. Additionally, the facility employs robots from multiple vendors, each specialized in different tasks and not designed to coordinate with one another. An environment that has narrow passages and lifts can create significant coordination challenges during navigation. To address this issue, Open-RMF[2] has been developed in recent years as a multiplatform system for controlling and managing heterogeneous fleets of robots.

[1] https://www.elettra.eu/

[2] https://www.open-rmf.org/

Chapter 2

Research questions and objectives

The objectives of this thesis are: to integrate a Rover Mini robot from the company RoverRobotics [1] into the ROS 2 Nav2 ecosystem,

Figure 2.1: RoverRobotics Rover Mini

to port the D-Star global planner from ROS 1 to ROS 2, upgrade our Jobot rover from ROS 1 to ROS 2,

Figure 2.2: Jobot Rover

incorporate the robot into the Open-RMF framework for centralized fleet management and evaluate the suitability of the Open-RMF ecosystem for deployment at the Elettra Sincrotrone Trieste.

[1] https://roverrobotics.com/

Chapter 3

Scope and limitations

The principal limitation is the current lack of access to lift control systems, which prevents direct integration and operation by the robot fleet. Additionally, there is no automated system for opening and closing doors. While Open-RMF offers a high degree of customizability, and human-in-the-loop interactions can be configured—such as triggering a light or sending a notification when a robot approaches a door—this method requires a human to manually open the door or operate the lift. However, such an approach is not ideal for achieving full autonomy.

Part II

Literature Review

In this section, we provide a description of the key technologies used in this thesis. We begin by describing the Robot Operating System (ROS) and its functionality, followed by an overview of the navigation system employed, and a discussion of how the Open-RMF fleet management system operates.

Chapter 4

Overview of ROS1 and ROS2

The Robotics Operating System (ROS)[1] is an open-source collection of tools and libraries designed to support the development of robotics applications. Despite its name, ROS is not an operating system (OS) in the traditional sense, but rather a software development kit (SDK) that provides developers with a comprehensive set of tools for building robotic systems. It offers several features typically associated with an OS, including hardware abstraction, low-level device control, implementation of commonly used functionalities, inter-process communication, and package management. The ROS ecosystem is built upon the following core components:

- **Middleware communication layer (RMW)**: Built on the Data Distribution Service (DDS) standard, this layer enables nodes to exchange information using an anonymous publish/subscribe pattern. This design promotes fault isolation, separation of concerns, and modular interfaces. ROS also supports synchronous, Remote Procedure Call (RPC)-style communication between services.
  While the default DDS implementation is eProsima Fast DDS (`rmw_fastrtps_cpp`), this work uses Eclipse Cyclone DDS (`rmw_cyclonedds_cpp`) due to its better compatibility with the Nav2 navigation stack in ROS Humble.

- **Development tools**: ROS provides a suite of tools for managing and enhancing software development, including utilities for launching, introspection, debugging, visualization, plotting, logging, and playback.

- **Multi-machine support** ROS includes tools and libraries for retrieving, building, writing, and executing code across multiple computers and OS.

- **Community and governance**: ROS is supported by an active and extensive global community, coordinated primarily by Open Robotics[2], which serves as the central organization for its development and maintenance.

[1] https://www.ros.org/
[2] https://www.openrobotics.org/

4.1 Structure of ROS

The official ROS documentation[3] provides comprehensive information about the system. All communication in ROS is handled through the **ROS Graph** , a peer-to-peer network of processes (called nodes) that execute concurrently and exchange information with one another. Some components of this graph are described below. Note that some of the functionalities described are specific to ROS 2 and are not available in ROS 1:

- **Nodes**: A node is a modular process that performs a specific task. Nodes communicate with each other through topics, services, actions, and parameters. In ROS 2, nodes can be implemented as **LifecycleNode**, which provide a managed state machine with defined transitions for startup and shutdown, enabling more deterministic behavior.

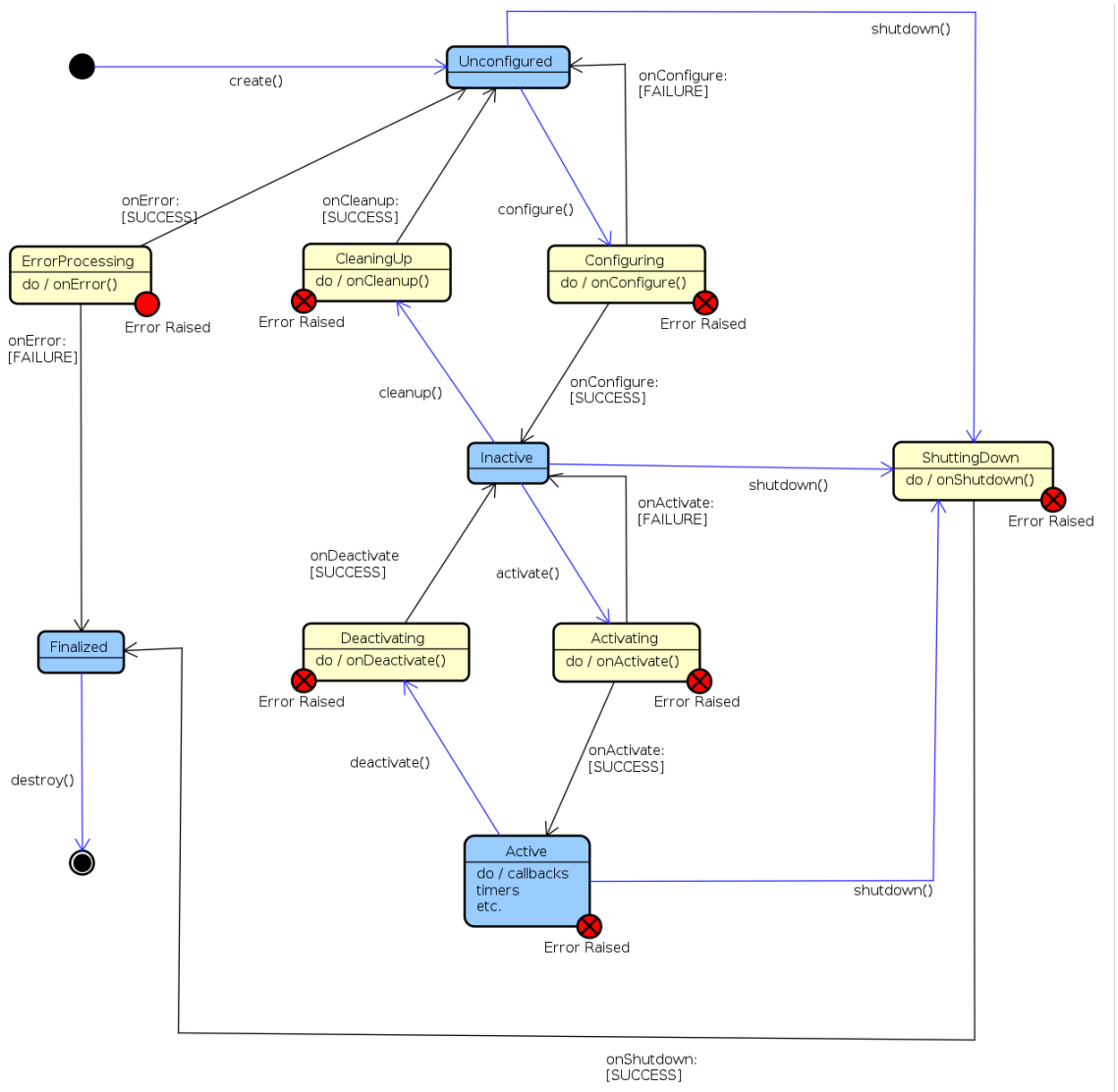

Figure 4.1: State Machine of LifecycleNode

Nodes can also be composed within the same process using the **Composition** feature, which improves performance by bypassing middleware overhead[7].

[3]https://docs.ros.org/en/rolling/index.html

- **Topics**: Topics are used for communication between nodes to share streams of messages in a publisher-subscriber structure.

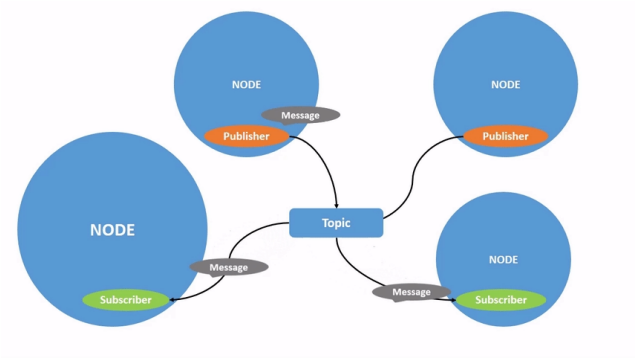

Figure 4.2: Topic process

- **Service**: Services are based on the request-response model. When a node with a service client sends a Request Message to a node with a service server, it receives a Response Message from that specific server.

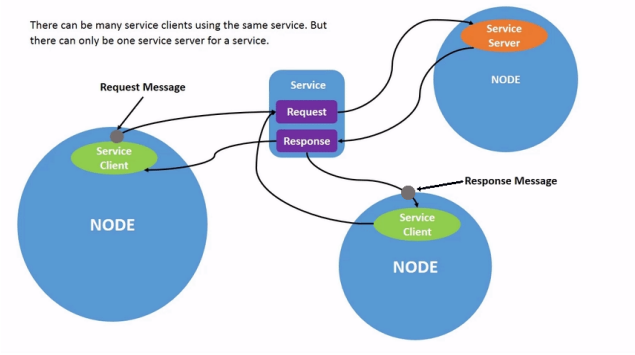

Figure 4.3: Service process

- **Parameters**: Parameters are configurations of a node that can be modified without changing the source code.

- **Actions**: Actions are designed to support long-running tasks that require feedback and the capability to interrupt when needed. They combine two services (goal and result) and a publisher-subscriber mechanism (feedback). A node sends a goal request and receives an acknowledgment, then starts publishing feedback messages while executing the task, and finally the action server responds with the result.

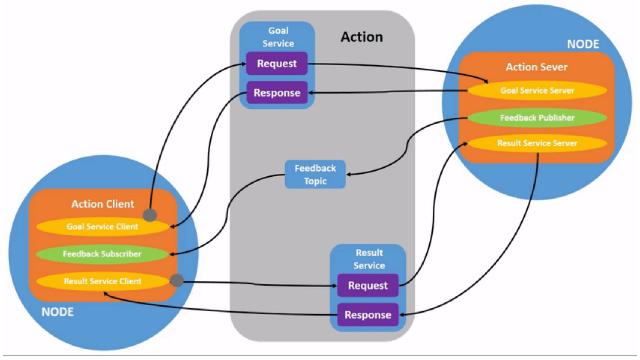

Figure 4.4: Action process

4.2 Tools of ROS

This section describes the tools available in ROS for developing robotic systems:

- **Launch**: A file structure that allows the simultaneous start of multiple nodes with a set of parameters.
- **rqt**: A graphical user interface (GUI) tool for ROS. It includes a series of plugins that facilitate development, monitoring, and diagnostics of the ROS system.
- **Rviz**: A 3D visualization tool for the Robot Operating System. It allows visualization of sensor data and state information from ROS.

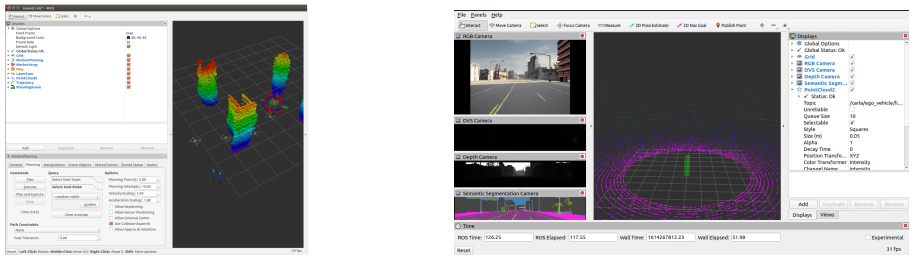

Figure 4.6: Examples of visualization of ROS data with Rviz

- **rosdep**: A command-line utility for installing system dependencies required by ROS packages.
- **sros2**: A security extension for ROS 2 that provides tools and instructions to apply security policies to DDS (Data Distribution Service).

Another important category of tools that are integrated with ROS is simulation environments. These environments allow developers to test their code on simulated robots under various conditions, which helps identify bugs that could potentially harm a real robot or its environment. Simulation tools provide realistic physical behavior for all components of a robotic system. According to the ROS documentation, two main simulation tools are commonly used: Gazebo and Webots. Among them, Gazebo is also maintained by Open Robotics and is widely adopted in many projects, including the one discussed in this thesis.
Gazebo offer:

- **ROS Integration**: A bridge that converts Protobuf messages to ROS messages, enabling communication between Gazebo and ROS.
- **Performance tools**: Features for distributed simulation across servers, dynamic asset loading, and adjustable time steps to optimize simulation performance.

- **Realistic simulation**: Support for a variety of sensors, including their noise models, within a 3D world rendered using OGRE 2.1[4] and powered by multiple physics engine, the default one is DART[5].
- **Extensibility**: A modular design that supports plugins, allowing developers to run custom code that interacts directly with the simulation environment.

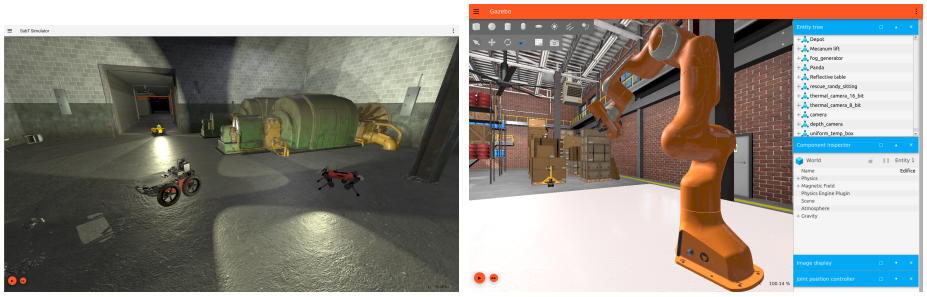

Figure 4.8: Examples of Gazebo

[4]https://wiki.ogre3d.org/Home
[5]https://dartsim.github.io/

Chapter 5

Introduction to NAV2 and Google Cartographer

Multiple tools are available to enable a robot to move autonomously within an environment. One of the most complete systems is Nav2 [1] [10], the successor to the ROS 1 Navigation Stack.

5.1 Description of Nav2

Nav2 provides a comprehensive set of tools for perception, planning[13], control, localization, and visualization, allowing any robot to navigate autonomously and reliably, even in complex environments. It enables building a model of the environment from sensor data, dynamically calculating the optimal path to avoid obstacles, setting motor velocities, and executing navigation tasks using Behavior Trees.

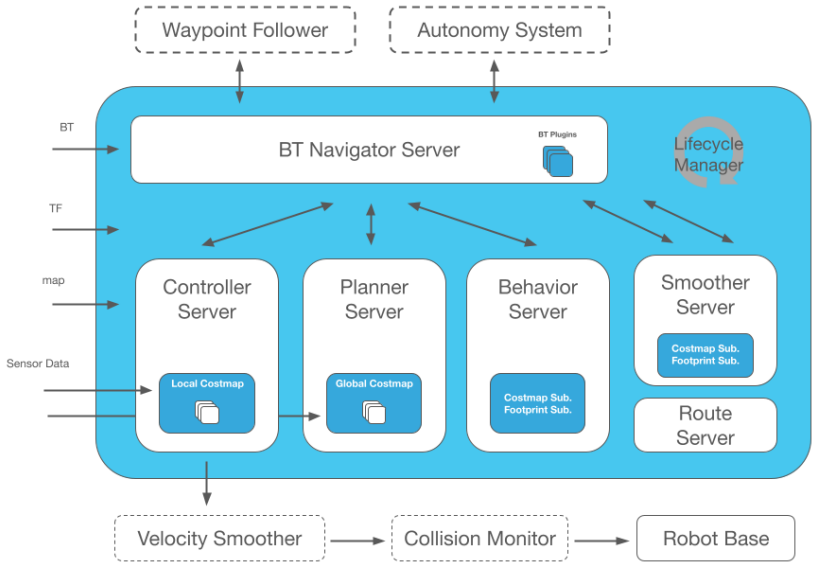

Figure 5.1: Nav2 architecture

[1] https://nav2.org/

Nav2 introduces several improvements over ROS 1, including:

- **Behavior trees**[3]: Nav2 uses Behavior Trees to structure and execute tasks in a human-readable and modular way. These trees enable the creation of complex systems and can be tested with various tools. The BehaviorTree.CPP[2] library is used to implement them, and it allows linking different trees for flexible behavior management.
- **nav2_util LifecycleNode**: This is a wrapper around the standard `rclcpp_lifecycle::LifecycleNode` described earlier 4.1. It adds a bond connection to the Lifecycle Manager, which monitors the state of each node and can safely transition nodes through their lifecycle states in the event of a failure.
- **Action Server**: Described in 4.1, Nav2 uses action servers for Behavior Tree interactions, such as NavigationToPose (navigate to a specific location) and NavigationThroughPoses (navigate through multiple waypoints).

Nav2 is composed of several key nodes that work together to control the robot:

- **Controller Server**: Handles movement requests. It uses plugin-based modules such as a progress checker (to verify if the robot is making progress), goal checker (to determine if the robot has reached its destination), and a controller (to generate velocity commands using the **local costmap** while avoiding collisions).
- **Planner Server**: Computes the global path to the destination using sensor data and a plugin-based global planner loaded with the **global costmap**.
- **BT Navigator Server**: Implements the NavigateToPose, NavigateThroughPoses, and other task interfaces. It is a Behavior Tree-based implementation of navigation that is intended to allow for flexibility in the navigation task and provide a way to easily specify complex robot behaviors.

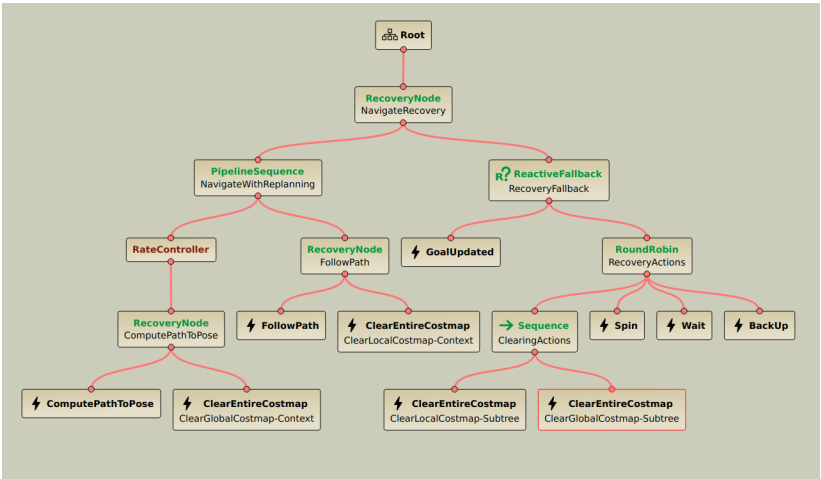

Figure 5.2: Humble Nav2 BT navigation `navigate_to_pose_w_replanning_and_recovery.xml` loaded with Groot2

[2]https://www.behaviortree.dev/

- **Smoother Server**: Smooth the path minimizing sharp turns and rapid rotations.

- **Behavior Server**: Handles specific behavior actions like spinning, backing up, and waiting.

- **Waypoint Follower**: An example of orchestrated control (an **Autonomy System**), this node takes a list of waypoints and uses the NavigateToPose action server to move between them, optionally performing custom tasks (e.g., taking a photo) at each waypoint through plugins.

- **Route Server**: Calculates routes through a predefined navigation graph instead of using free-space planning.

- **Velocity Smoother**: Smooths velocity, acceleration, and deadband signals sent to the robot to ensure safe and smooth motion.

- **Costmap** is a representation of the environment using a regular 2D grid of cells with a cost (unknown, free, occupied, or inflated) and is used for generate the global plan or to compute local control efforts.

The next things that Nav2 relies on for controlling the navigation are:

- **state estimation** of the robot that generate a suitable TF tree, a tree structure of coordinate frames generated using the transform library[4], based on the REP 105, that is should provide this structure: `map` → `odom` → `base_link`. According to the REP 105 they are defines as:

  - `map` : A world-fixed frame that may change in discrete jumps. It's useful for long-term global reference but not suitable for short-term localization.
  - `odom` : A continuous world-fixed frame that may drift over time. It provides short-term local reference.
  - `base_link`: A robot fix frame.

  The `odom` → `base_link` link can be estimated using various sensors. For improved accuracy, we use the `robot_localization` package[12], which provides nonlinear state estimation using Kalman Filter fusing the robot's odometry and the IMU data.
  The `map` → `odom` link can be generate with a global positioning system like Simultaneous Localization And Mapping (SLAM) method[2] or using the one provided by Nav2, Adaptive Monte-Carlo Localization (AMCL).

- **map** can either be preloaded or built dynamically using a SLAM method. One SLAM method systems is Google Cartographer, which will be described in the next section. It is also the SLAM method that was used to generate the map for navigation.

5.1.1 Cartographer

Cartographer[5] is a SLAM method that operates using two main subsystems: a **Local SLAM** (called **Frontend**) that build a series of locally consistent **submaps** from successive scans and from the pose extrapolator with the possibility that it can drift during acquisition and a **Global SLAM** (called **Backend**) that reduces drift by performing loop closure—detecting when the robot has returned to a previously visited location. It uses scan matching against submaps and applies global optimization to align all submaps consistently and build a globally accurate map. Cartographer is capable of achieving mapping accuracy of 5 cm, but it requires extensive parameter tuning to reach optimal performance.

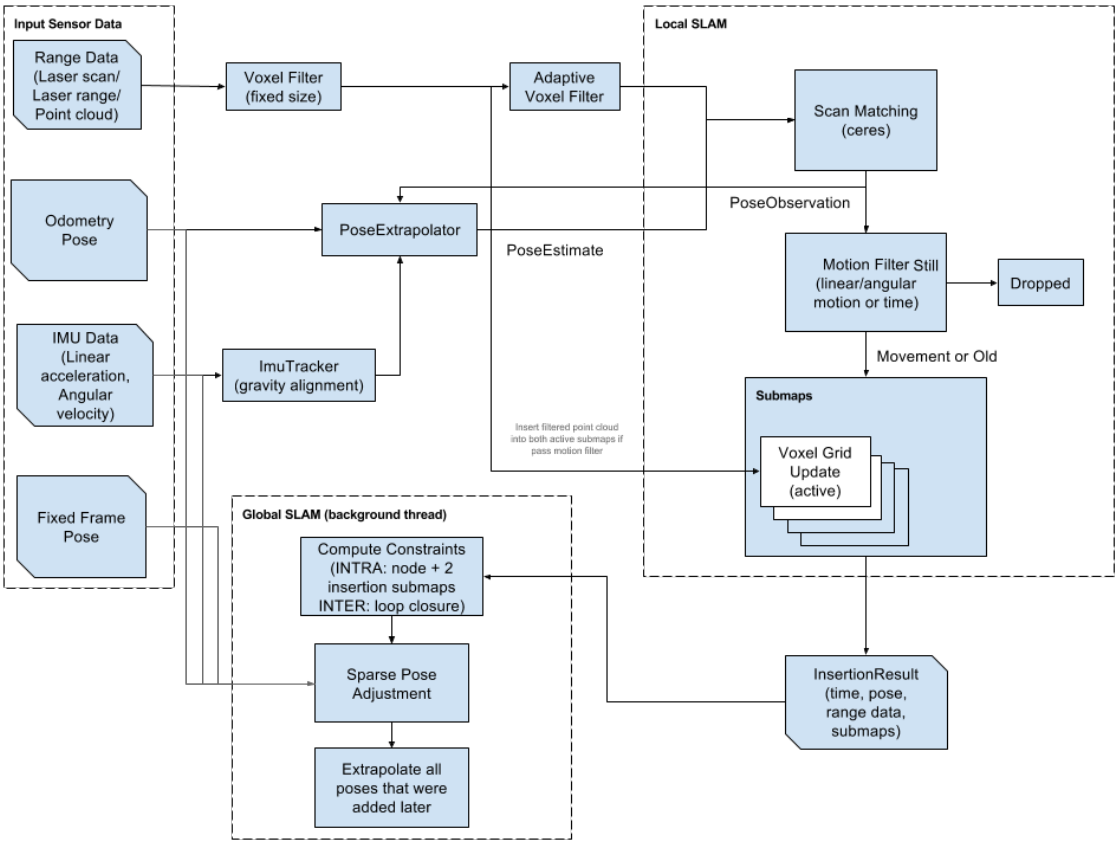

Figure 5.3: Cartographer Algorithm

Chapter 6

Review of related work on porting navigation stack from ROS 1 to ROS 2

The widespread adoption of the Robot Operating System (ROS) has introduced new use cases that were not originally considered during its development. Initially created as an academic project, ROS encountered limitations as users began applying it to more complex scenarios such as fleet management, deployment on embedded boards and microcontrollers, real-time systems, unreliable networks, and industrial applications. These emerging requirements highlighted architectural constraints in ROS 1, ultimately necessitating a complete redesign, which led to the development of ROS 2. Comprehensive documentation for ROS 2 is available at https://design.ros2.org/ . Many companies have adopted the ROS 2 ecosystem, noting improvements in development workflows[9] and in specific subsystems like in Nav2[13]. New initiatives, such as micro-ROS[1], aim to bring ROS 2 capabilities to microcontrollers. Additionally, features like Node Composition offer improved performance optimization[7] while sros2 allow for a better security[15]. As outlined in these references, ROS 2 offers substantial advantages over ROS 1, making it the preferred framework for modern robotic applications. With this in mind, we will migrate the global planner using the official ROS 2 documentation[2] and various secondary sources [3].

[1]https://micro.ros.org/

[2]https://docs.ros.org/en/rolling/How-To-Guides/Migrating-from-ROS1.html

[3]https://industrial-training-master.readthedocs.io/en/melodic/_source/session7/ROS1-to-ROS2-porting.html

Chapter 7

Overview of open source fleet management systems, particularly Open-RMF

Currently, several fleet management systems are available:

- **Isaac Mission Dispatch**[1]: A microservice-enabled fleet management system developed by NVIDIA that allows users to submit missions to multiple robots and monitor both robot and mission statuses. It provides connectivity between robots and the fleet management system but does not handle logistics such as task allocation or conflict resolution (e.g., intersecting robot paths). The implementation relies on the VDA5050 protocol, an industry standard for communication between cloud control services and mobile robots, and uses MQTT, a lightweight, publish-subscribe network protocol designed for devices with limited resources and bandwidth.
- **ROOSTER Fleet Management**[2]:An open-source ROS 1-based project that supports heterogeneous fleet management with task allocation, scheduling, and routing functionalities.
- **Open Robotics Middleware Framework (Open-RMF)**[3]: A free, open-source, and modular system that facilitates interoperability between multiple robot fleets and physical infrastructure such as doors, elevators, and building management systems.
- **Robofleet**[4][14]: An open-source system based on ROS 1 that supports inter-robot communication, remote monitoring, and remote tasking for heterogeneous fleets of ROS-enabled service robots. It is specifically designed to handle network variability and to incorporate security control features.
- **openTCS**[5]: open Transportation Control System) is a free control system software for coordinating fleets of automated guided vehicles (AGVs) and mobile robots. It can control any automatic vehicle,independent of their specific

[1]https://github.com/NVIDIA-ISAAC/isaac_mission_dispatch
[2]https://github.com/ROOSTER-fleet-management
[3]https://www.open-rmf.org/
[4]https://github.com/ut-amrl/robofleet
[5]https://opentcs.org

characteristics, with communication capabilities. Also, It can manage vehicles of different types and performing different tasks at the same time.

Other frameworks provide the infrastructure needed to initiate fleet control:

- **Transitive Robotics**[6]: An open-source framework for full-stack robotics, designed to simplify the development of capabilities that connect robots, cloud services, and web front-ends through a dynamically updating shared data model. It addresses common challenges in building full-stack robotic applications, such as unreliable networks, intermittently offline robots, and cross-device versioning.

- **Robot Web Tools**[7]: A collection of tools that enable communication with remote robots via web technologies, facilitating remote monitoring and interaction.

Given that Open-RMF offers control over multiple fleets, enables structured interaction between them to achieve optimal outcomes, provides a comprehensive toolset for system development, is based on ROS 2, and is open source, it will be used as the foundation for our fleet management architecture.

7.1 Open-RMF description

Open-RMF is a flexible and modular toolkit built on ROS 2 that facilitates interaction between heterogeneous robot fleets. It employs standardized protocols to enable communication with infrastructure, environments, and automation systems, optimizing the utilization of critical shared resources such as elevators, doors, narrow corridors, and other robots. By coordinating access to these resources, Open-RMF introduces system-level intelligence that helps prevent conflicts and ensures proper task and resource allocation. Figure 7.1 illustrates the architecture and operational workflow of Open-RMF.

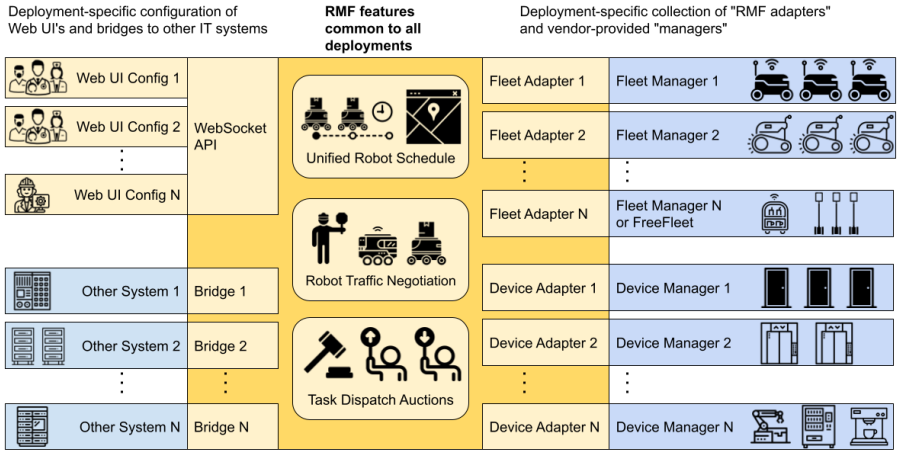

Figure 7.1: Open-RMF Structure

[6]https://transitiverobotics.com/
[7]https://robotwebtools.github.io/

The fleets that can be controlled through Open-RMF can be categorized into three types:

- *Full Control*: Fleets of this type provide full status updates and allow complete control over path planning and execution.
- *Traffic Light*: These fleets also provide full status updates, but only allow limited control, such as pausing and resuming operations. Open-RMF has control over certain behaviors, but not over the specific paths.
- *Read Only*: These fleets only provide status updates and cannot be controlled. Only one read-only fleet can exist in the system at any given time, as the presence of multiple read-only fleets would make it nearly impossible to avoid deadlocks or resource conflicts.

If a fleet lacks has *No Interface*, it can lead to deadlocks in resource allocation and is incompatible with an RMF system.
The core of Open-RMF, known as `rmf_core`, is composed of several modular packages, each serving a distinct role within the system:

- `rmf_traffic`: Provides core scheduling and traffic management capabilities. It is based on a platform-agnostic **Traffic Schedule Database** and includes two main mechanisms for traffic deconfliction:

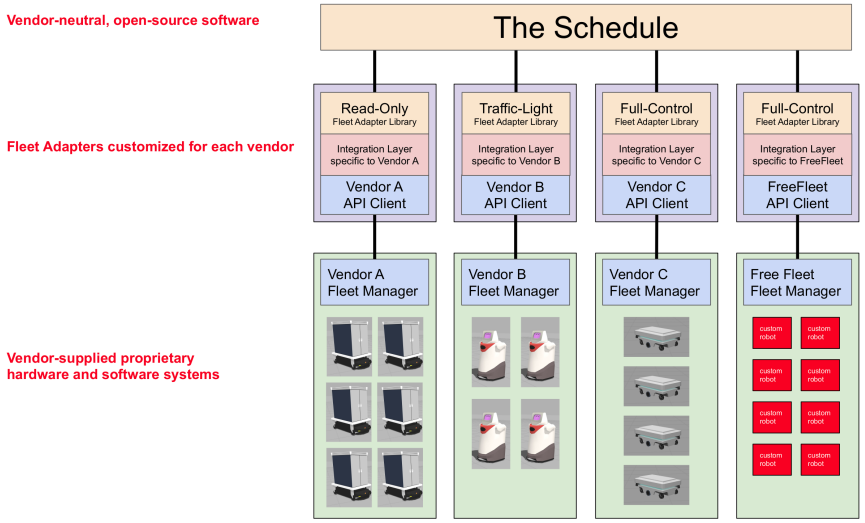

Figure 7.2: Open-RMF Schedule structure

  - *Prevention*: When changes occur in the database (e.g., route cancellations), fleet managers can proactively replan routes to avoid conflicts.
  - *Negotiation*: When a potential conflict is detected between two or more schedule participants, a conflict notice is issued to the relevant fleet managers. Each manager responds with a preferred path that accommodates other agents, and an external arbitration module selects the optimal path.

Conflict avoidance for Automated Guided Vehicles (AGVs) is implemented using a time-dependent extension to A* search. This algorithm considers both spatial and temporal dimensions, allowing it to generate conflict-free paths by accounting for the future movements of other agents. Support for Autonomous Mobile Robots (AMRs) is under active development.

- `rmf_traffic_ros2`: Provides the necessary ROS 2 interfaces for integrating rmf_traffic into ROS 2-based systems.
- `rmf_task`: Contains APIs and base classes for defining and managing tasks in RMF. The framework natively supports three types of task requests:
  - *Clean*: For robots capable of cleaning floor spaces.
  - *Delivery*: For robots tasked with transporting items between locations.
  - *Loop*: For robots required to repeatedly navigate between two or more points.
- `rmf_dispatcher_node`: Contains the logic responsible for task allocation and management across multi-fleet systems. When a user submits a new task request, the dispatcher intelligently assigns it to the most suitable robot by querying each fleet adapter capable of performing the task. These adapters respond with bids indicating which robot is best suited, and RMF uses this bidding mechanism to determine the optimal assignment, as illustrated in Figure 7.3.

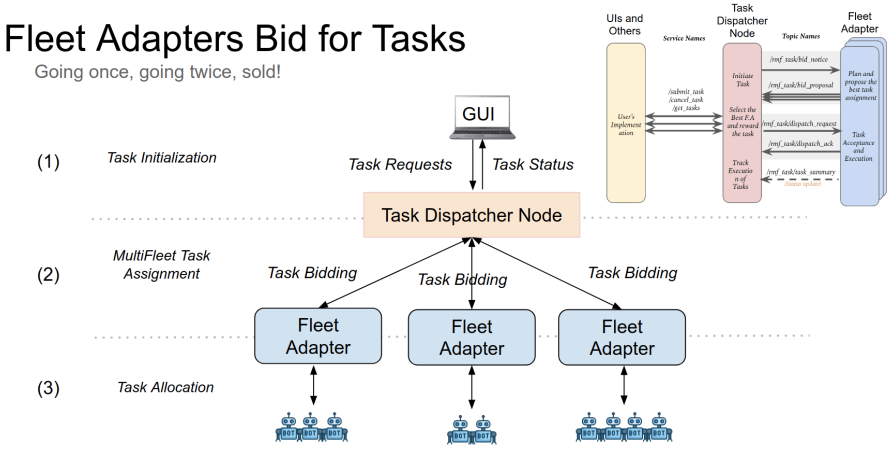

Figure 7.3: Open-RMF Bid Task

- `rmf_battery`: Provides battery consumption estimation models for different robot activities, aiding in energy-aware task planning.
- `rmf_ros2`: Supplies ROS 2 adapters, nodes, and Python bindings to integrate the RMF core components into ROS 2 applications.
- `rmf_utils`: A collection of utility functions and tools used across various RMF packages.

The following tools are available to support development with Open-RMF:

- **RMF demos**: A collection of demonstration packages showcasing the capabilities of Open-RMF. These serve as a useful starting point for development and experimentation.

- **Traffic Editor**: A graphical map editor used to define traffic patterns and interaction points (e.g., doors, lifts) that RMF uses for traffic management. It also supports the generation of Gazebo simulation worlds based on the designed maps.

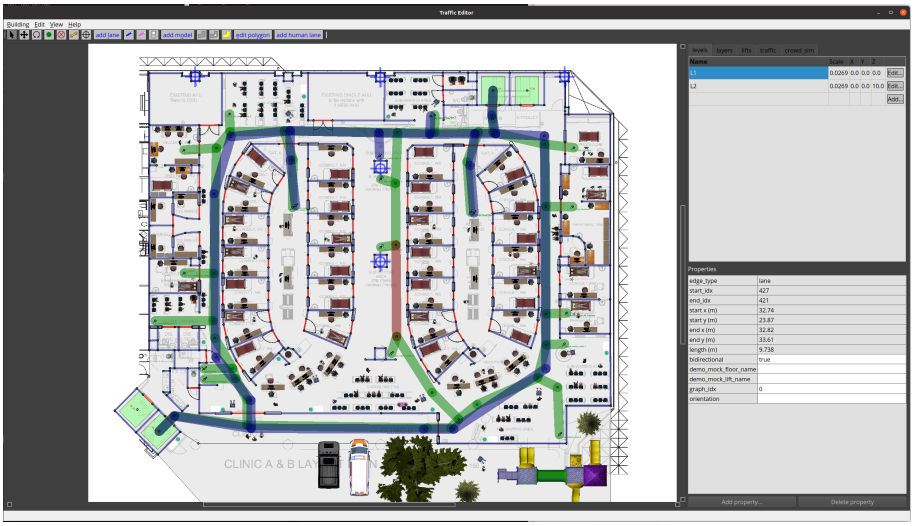

Figure 7.4: Traffic Editor

Another version of this software, **rmf_site**[8], is currently undergoing active deployment.

- **Free Fleet**: An open-source fleet adapter that enables the integration of standalone mobile robots into Open-RMF, allowing the creation of heterogeneous fleets.

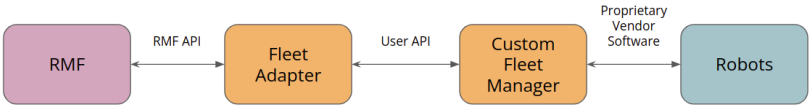

Figure 7.5: Fleet Adapter

- **RMF Schedule Visualizer**: An RViz plugin that provides visualization and monitoring of the RMF schedule, enabling users to observe and interact with traffic and task execution in real time.

[8]https://github.com/open-rmf/rmf_site

- **RMF Web UI** : A web-based application for visualization and control of the Robotic Middleware Framework (`rmf_core`). It provides a user-friendly interface for monitoring system state and interacting with fleet operations.

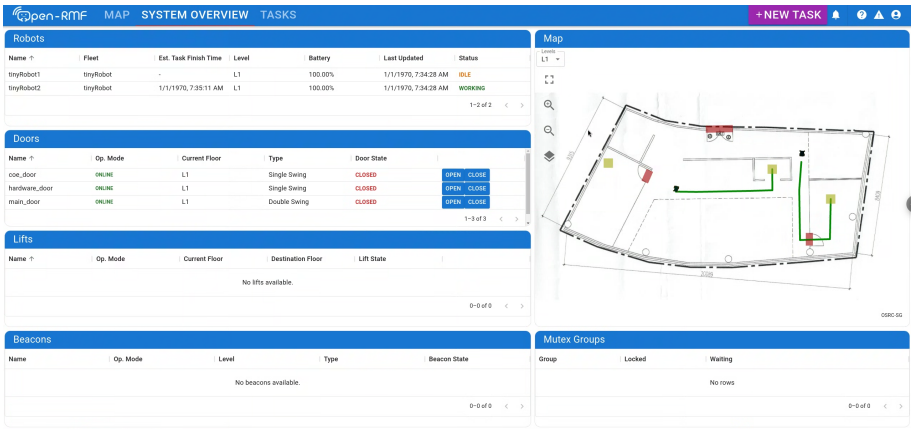

Figure 7.6: RMF Web

- **RMF Simulation**: A set of Gazebo plugins designed to simulate various aspects of building infrastructure—such as doors, lifts, and crowds—as well as robot behavior, including navigation and interactions with workcells.
- **Simulation assets**: Freely available 3D models and environment assets used to support simulation development and testing in Gazebo.

Part III

Methodology

In this part, we describe the entire procedure followed in the development of this project:

- The setup of our testing platform, based on the Rover Mini and a suite of onboard sensors.
- The configuration of the Nav2 navigation stack.
- The procedure used to port the global planner from ROS 1 to ROS 2, and its integration into the Nav2 framework.
- The tools and methods used to integrate the system into Open-RMF.

All source code developed for this project is publicly available in the following GitHub repository: `michbelle/myTesiCode`[9]. The repository also includes the necessary components for simulating the system.
The simulation environment is containerized using Docker, which provides a consistent and OS-independent base system. The Docker image is built on top of `osrf/ros:jazzy-desktop-full` and includes all the packages developed for this thesis, as well as the required Open-RMF tools and its demo simulation.

[9]https://github.com/michbelle/myTesiCode

Chapter 8

Description of research platform and hardware used

The robot used in this thesis is a four-wheel drive (WD) skid-steering vehicle (see Figure 2.1). It has a maximum speed of 12.87 km/h and a payload of 45,36 kg. Its operational time ranges from 60 to 90 minutes under active usage, and up to 6 hours in idle mode. The manufacturer provides control and simulation code on GitHub[1], compatible with the Gazebo simulator. However, this code is officially tested only for the ROS 2 Humble distribution, which has therefore been selected for use in this thesis.
The robot features a USB interface that allows connection to a PC or single-board computer for control purposes. In this work, a mini PC is used to control the robot and also used to manage the entire navigation stack based on Nav2. The Mini PC used is a **Beelink SER3-E-16512EJ0W64PRO** with a Ryzen 7 3750H processor which is a quad-core 8-thread 2.3 GHz mobile processor boosting to 4.0 GHz with a Radeon RX Vega 10 Graphics. The mini PC draws power directly from the robot, which supplies a 19V output, thereby reducing the overall operational time.

Figure 8.1: Mini PC used

To improve odometry estimation, an additional sensor is integrated: the **IMU Witmotion (WIT) JY901B**. This device is a 9-Axis Combined IMU/Magnetometer/Altimeter (measure linear accelerations, angular velocities, Euler angles, magnetic field, barometry, altitude) that communicates using the TTL/UART protocol. Sensor data are acquired using the Witmotion Sensors UART Connection

[1] https://github.com/RoverRobotics/roverrobotics_ros2

Library[17], using a USB-to-serial converter, and are published in the ROS 2 environment through a dedicated node, `witmotion_ros`, avaible on GitHub at `ElettraSciComp/witmotion_IMU_ros`[2].

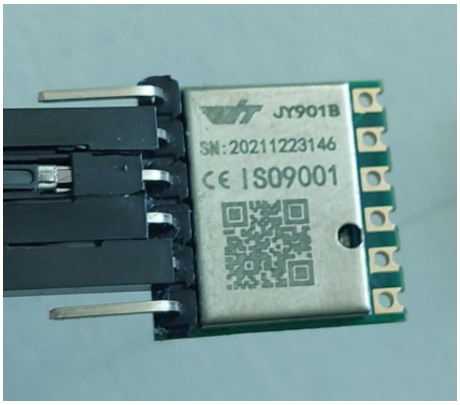

Figure 8.2: IMU used

For localization 5.1, a **2D laser scanner** is employed to provide environment perception and compensate for odometry drift by providing a transformation link between the `map` and `odom` TF frames. The 2D laser scanner provides a cost-effective solution for environmental perception, providing dense spatial information that enables obstacle detection and basic environment recognition. Compared to alternative technologies [1][11], it requires significantly less computational power, making it a practical choice for lightweight and resource-constrained robotic platforms. The Laser scan used is **RPLIDAR S1** a 360 degree 2D laser scanner(LIDAR) with a distance range of 40m on white object and 10m on black ones. It features a sample rate of 9.2kHz and a scan rate ranging from 8 to 15 Hz,resulting in an angular resolution from 0.313° to 0.587°, with an accuracy of ±5cm and resolution of 3cm. The ROS 2 driver for the RPLIDAR S1 is available on Github at: `Slamtec/sllidar_ros2`[3].

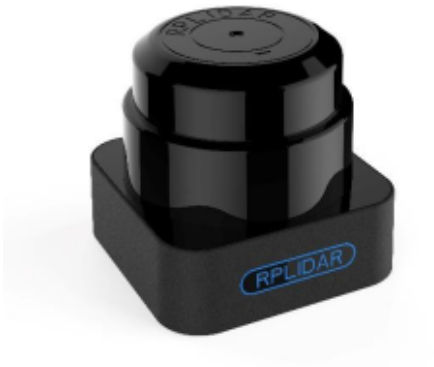

Figure 8.3: LIDAR used

[2]https://github.com/ElettraSciComp/witmotion_IMU_ros/tree/ros2
[3]https://github.com/Slamtec/sllidar_ros2

All components are physically mounted on the rover using custom 3D-printed parts, as shown in the following image.

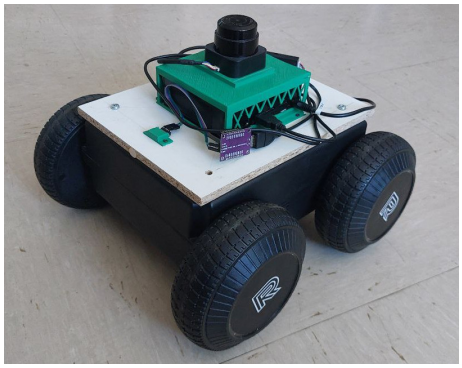

Figure 8.4: Robot used

To reflect the updated hardware configuration, the original URDF (Unified Robot Description Format) file provided by the manufacturer was modified. This ensures accurate sensor placement relative to the `base_link` frame, which is essential for proper operation of sensor-dependent ROS 2 nodes.
A 3D model of the robot, including all coordinate frames (TF), is shown in this picture:

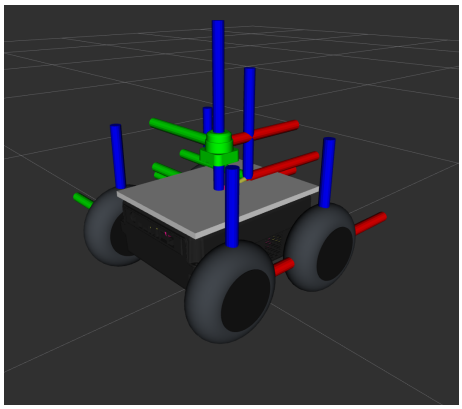

Figure 8.5: 3D model with TF

With the system fully assembled, the following ROS 2 launch files are used to retrieve data from the sensors connected to the robot's onboard mini PC:

- `0.1r_sensorLaunch.launch.py`: Launches the nodes responsible for publishing data from the IMU and LIDAR sensors.
- `0.2r_mini.launch.py`: Publishes odometry data based on the built-in encoders of the robot and subscribes to the `/cmd_vel` to move the robot.

While, for simulation, the following launch file can be used for starting the simulation:

- `0s_ign_mini_gazebo.launch.py`: Launches the Gazebo simulation of the robot with all configured sensors. The `ros_gz_bridge/parameter_bridge` node is used to share simulated data between Gazebo and the ROS 2 environment.

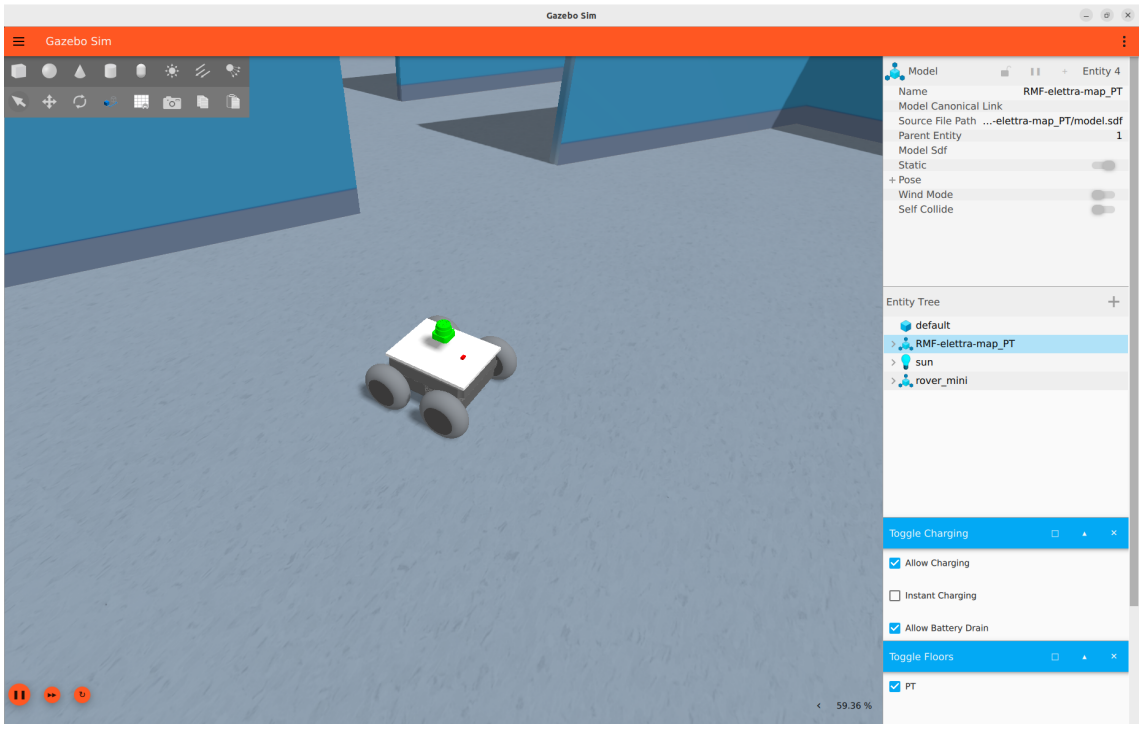

Figure 8.6: Model of the robot in Gazebo Ignition

Chapter 9

Steps taken to port the navigation stack from ROS 1 to ROS 2

In this chapter, we outline the requirements for the operation of Nav2 and provide a comprehensive overview of the packages utilized by Nav2. Particular emphasis is placed on the process of porting the Global Planner module from ROS 1 to ROS 2, with a detailed discussion of the required adaptations and the challenges encountered during the transition.

9.1 Odometry Estimation `odom` → `base_link`

As previously discussed, the raw odometry data provided by the rover are not sufficiently accurate, particularly in rotational movements[12]. This is a well-known issue in odometry estimation for skid-steering vehicles, which tend to produce significant drift and errors when turning using only the data from the encoders.
To address this problem, the `robot_localization` package[12] was employed, specifically its `ekf_node` Node. This package enables the fusion of multiple sensor, even multiple instances of the same sensor type, and supports output in various ROS message formats, such as `nav_msgs/Odometry`. It also allows preprocessing of sensor messages prior to inserting into the estimation process. The filter performs continuous state estimation and is robust to intermittent sensor data loss, maintaining a coherent position estimate over time.
To launch the EKF node, the following launch file was wrote:

`1<r/s>_robot_localizationEKF.launch.py`

9.1.1 Configuration Odometry Estimation in `ekf_node`

As previously described and detailed in the node's documentation[1], we added the information about the sensors used to the configuration file. The principal information to include is the type of sensor and on which topic subscribe, `odom0:odometry/wheels` and `imu0:imu/data`, and, for each sensor, we need to choose which components of the state measurement data we want to fuse using the following array by setting the corresponding values to `true` for those we wish to include and `false` for those we do not:

[1] https://docs.ros.org/en/noetic/api/robot_localization/html/index.html

`sensorX_config`:$[X, Y, Z, roll, pitch, yaw, \dot{X}, \dot{Y}, \dot{Z}, \dot{roll}, \dot{pitch}, \dot{yaw}, \ddot{X}, \ddot{Y}, \ddot{Z}]$
In our case, we fused the yaw angular velocity from the IMU while for the odometry we select only the velocity along the x and y in ENU (East North Up) coordinates.

Additionally, the process noise covariance matrix Q can be tuned. This matrix represents the noise we add to the total error after each prediction step. Generally, the larger the value of Q relative to the variance of a given variable in an input message, the faster the filter will converge to the value indicated by the measurement. In the next section we will evaluate the performance obtained setting these parameters.

9.1.2 Parameter changing result

As describe in the paper, it allow the fusion of multiple sensor data and also expose the process noise covariance to tune it.

Figure 9.1: Difference in the odometry between raw data and filtered (recorded data 001)

9.2 Localization `map` → `odom`

In this section, we describe the process of acquiring a map and establishing the transformation between the `map` frame and `odom` frame using a localization method. This is necessary because, for long-term operation, the position of the robot in the world, using only odometry data, tends to accumulate errors and is not sufficiently accurate. Therefore, a localization technique is required to compensate for these errors[2].

9.2.1 Mapping

Although a map is already available, for completeness we provide instructions to run the following SLAM packages: `slam_toolbox`[6] (GitHub: SteveMacenski/slam_toolbox[2]) and `cartographer_ros`[5] (GitHub: ros2/cartographer_ros[3]). These packages can be launched using the following launch files:

```
X<r/s>_mappingW<slam/cartographer>.launch.py
```

To save the generated map, use the following command, which works for both `slam_toolbox` and `cartographer_ros`. This will save the map as two files: a `.pgm` image and a `.yaml` metadata file:

```
ros2 run nav2_map_server map_saver_cli -f map
```

When using Cartographer, it is also possible to save the map as a `.pdstream` format with the following command:

```
ros2 service call /write_state cartographer_ros_msgs/srv/WriteState
  "{filename:  "map.pdstream", include_unfinished_submaps :false}"
```

[2]https://github.com/SteveMacenski/slam_toolbox/tree/ros2
[3]https://github.com/cartographer-project/cartographer_ros

9.2.2 Only localization

~~The map that was built using Cartographer is the following:~~ The map create using Cartographer was obtained by the robot Jobot while navigating through Building T of Elettra, which serves as an office building. The map is as follows:

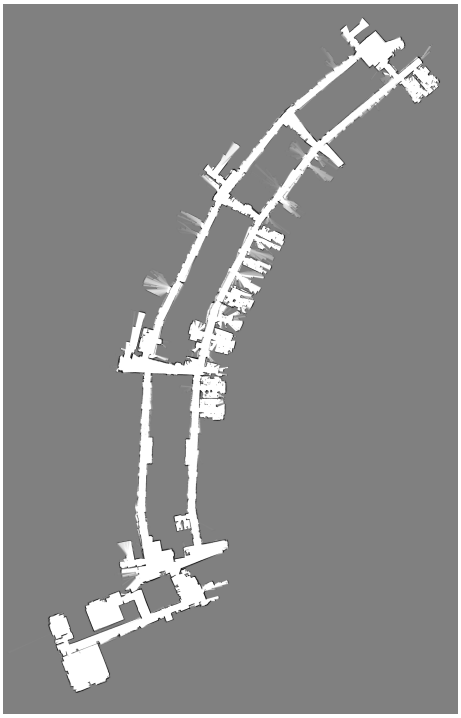

Figure 9.2: Map acquired with Cartographer

We initially attempted to use Cartographer solely for localization; however, due to the project's limited maintenance, we encountered difficulties in using it effectively. As a solution, we opted for the method provided by the Nav2 package, specifically `nav2_amcl` and the `amcl` node.
One limitation of this approach is the need to convert the map acquired with Cartographer from the `.pbstream` format into a `.pgm` image and `.yaml` metadata pair, which are required by the `amcl` node. This conversion can be performed using the following command:

```
ros2 run cartographer_ros cartographer_pbstream_to_ros_map \
        -pbstream_filename map.pbstream \
        -map_filename map.pgm \
        -resolution 0.05
```

Since certain walls in the Cartographer generated map where not solid black, due to the uncertainty of their position during mapping, the conversion process introduced errors that led to these areas to be misinterpreted as free space. As a result, the planner generated paths that passed through them. To address this issue, the walls were manually corrected using GIMP, an image editing tool, by modifying the `.pgm` map file.
Additionally, our planner is designed with the assumption that unknown spaces are not usable for planning. Therefore, we set the parameter `track_unknown_space` in the global costmap node to `true` (`false` by default) to achieve optimal results from our planner. aggiungere teletransport per la modifica della mappa? The corrected map is shown below:

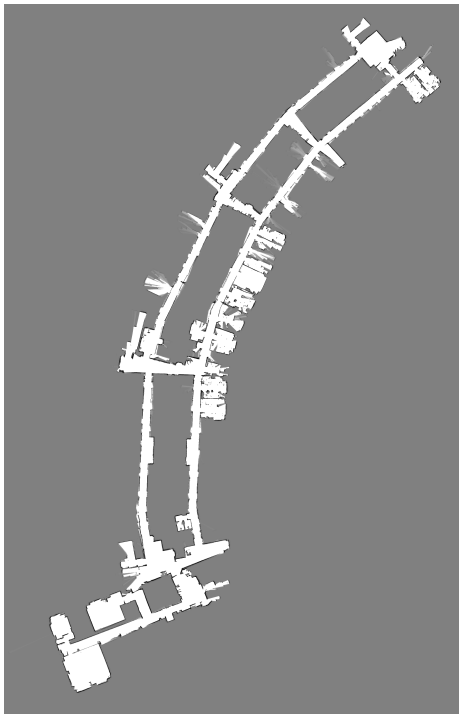

Figure 9.3: Map edited with Gimp and used

With the map properly adjusted, all necessary tools for localization are now available. We created the following launch file to initiate localization:

`3<r/s>_L<amcl/Cartographer>.launch.py`

At this stage, all requirements are met to utilize the Nav2 navigation stack.

9.3 Structure Navigation Stack Nav2

In this section, we will discuss the components used in the Nav2 navigation stack and describe the process of porting the Global Planner from ROS 1 to ROS 2.

9.3.1 Navigation Launch file

The launch file contains all the necessary processes required for the Navigation stack Nav2 to operate. It also implement the composition structure (4.1) and the lifecycle manager (4.1). Below is a list of the core components used, along with a brief

description of the plugins associated with each. Additional documentation can be found in [4] , and a migration guide is available in[5] :

- `nav2_controller::ControllerServer` that use the following plugins:
  - As `controller_plugins` use `nav2_mppi_controller::MPPIController`: a Model Predictive Path Integral (MPPI) algorithm to track a path with adaptive collision avoidance.
  - As `progress_checker_plugins` use `nav2_controller::SimpleProgressChecker` that checks whether the robot has made positional progress.
  - As `goal_checker_plugins` use `nav2_controller::SimpleGoalChecker` that checks whether the robot has reached the goal pose.
- `nav2_smoother::SmootherServer` that use the following plugins:
  - As `smoother_plugins` use `nav2_smoother::SimpleSmoother` that takes an input path and applies a lightweight smoothing algorithm. It weights the initial path points and the smoothed path points to create a balanced result where the path retains its high level characteristics but reduces oscillations or jagged features.
- `nav2_planner::PlannerServer` that use the following plugins:
  - As `planner_plugins` use `dstar_global_planner/DStarGlobalPlanner` that is our global planner plugin and is a implementation of the D* informed incremental search algoritm with external optimizers[16].
- `nav2_behaviors::BehaviorServer` that use the following plugins:
  - As `behavior_plugins` use: `nav2_behaviors/Spin` performs an in-place rotation by a specified angle, `nav2_behaviors/BackUp` moves the robot backward by a set distance, `nav2_behaviors/Wait` is used to pause navigation for a specified duration, allowing time for temporary occlusions to clear before resuming path planning.
- `nav2_bt_navigator::BtNavigator` that use the following plugins:
  - As `navigators` use `nav2_bt_navigator::NavigateToPoseNavigator` and `nav2_bt_navigator::NavigateThroughPosesNavigator` that utilize the default behavior trees (BT) for navigation.
- `nav2_lifecycle_manager::LifecycleManager` that handling the lifecycle transition states for the stack in a deterministic way.

The simulation configuration file contains minor differences due to the use of different versions of ROS 2: Jazzy in the simulated environment and Humble on the physical robot. In the next section, we will describe the key steps taken to port the global planner `DStarGlobalPlanner` from ROS 1 to ROS 2.

[4] https://docs.nav2.org/setup_guides/algorithm/select_algorithm.html
[5] https://docs.nav2.org/migration/index.html

9.3.2 Porting ROS 1 DStarGlobalPlanner to ROS 2

As previously mentioned, the `DStarGlobalPlanner`[6] is a implementation of the D* informed incremental search algoritm with external optimizers[16] for the ROS 1 `move_base` package[7] and we want to analyze if there are some advantages of using this planner compared to the default planner.
Since the code is written using ROS 1, which is not compatible with ROS 2, it will need to be ported to the newer version. This process was divided into two main phases. In the first phase, we migrated the base structure of the package from ROS 1 to ROS 2, following these ROS 2 porting guidelines [8][9]. In the second phase, we adapted the code to comply with the structure required by Nav2 planner plugins, based on the Nav2 documentation titled: "Writing a New Planner Plugin"[10]. The following subsections describe these two phases in detail. The first focuses on the structural migration to ROS 2 and the integration into the Nav2 framework. The second details the modifications needed to adapt the planner logic and interfaces to match ROS 2 conventions and the plugin architecture.

Porting code from ROS 1 to ROS 2

In the `package.xml` file, which describe the package as an organizational unit for your ROS 2 code, the following changes were made:

- `buildtool_depend` : `catking` → `ament_cmake`: `ament_cmake` is the new build system for CMake based packages. Compared to `catkin`, it introduces many utilities for managing packages. More information is available in the documentation[11].
- `roscpp` → `rclcpp`: `rclcpp` is the new ROS client library for C++, built on `rcl`, which provides the foundation for language-specific client libraries. `rcl` communicates with `rmw`(ROS middleware interface), which handles the communication with `DDS` (Data Distribution Service).
- Updated package naming to follow the ROS 2 conventions.
- To inform `colcon` that the package uses `ament_cmake`, the build type must be explicitly specified in the `build_type`.

Subsequently, the `CMakeLists.txt` file was updated with the following major changes:

- Package names were updated to reflect ROS 2 naming conventions.
- C++ standard was set to 17.
- `find_package()` is required to be called for each package.
- `ament_export_dependencies()` is used to declare runtime dependencies.

[6] https://github.com/ElettraSciComp/DStar-Trajectory-Planner/tree/ros2_port
[7] https://wiki.ros.org/move_base
[8] https://docs.ros.org/en/rolling/How-To-Guides/Migrating-from-ROS1.html
[9] https://industrial-training-master.readthedocs.io/en/melodic/_source/session7/ROS1-to-ROS2-porting.html
[10] https://docs.nav2.org/plugin_tutorials/docs/writing_new_nav2planner_plugin.html
[11] https://docs.ros.org/en/rolling/Concepts/Advanced/About-Build-System.html

- The package must be declared in ROS with `ament_package()`.

Other minor changes are:

- Log calls were updated from `ROS_` to `RCLCPP_`
- Message type structures changed (e.g., `geometry_msgs::PoseStamped` → `geometry_msgs::msg::PoseStamped`).
- The structure of the C++ program now required the use of smart pointer.

Additional changes specific to the ROS 2 Navigation stack Nav2 are discussed in the following section.

Porting code from ROS 1 Navigation stack to Nav2

The following changes were made to port the planner to the Nav2 framework:

- Since the planner is a **global planner**, it now inherits from the new abstract interface `nav2_core::GlobalPlanner`. As a result, we updated the `global_planner_plugin.xml` file to change the `base_class_type` accordingly.
- As a plugin based on `nav2_core::GlobalPlanner`, the planner must implement five pure virtual methods required for proper operation: `configure()`, `activate()`, `deactivate()` and `cleanup()`, `createPlan()`. Below is a description of each method:
  - `configure()`: it's called when the node enters `on_configure` state. This method is responsible for declaring ROS parameters and initializing the planner's member variables. It takes the following input parameters:
    * `rclcpp_lifecycle::LifecycleNode::WeakPtr` A weak pointer to the base lifecycle node.
    * The planner's name as a `std::string`.
    * A share pointer to the `tf` buffer `std::shared_ptr<tf2_ros::Buffer>`.
    * A shared pointer to the costmap `std::shared_ptr<nav2_costmap_2d::Costmap2DROS>`.
    * For jazzy also: `std::function<bool()> cancel_checker` that allow for the cancellation of the current planning task.

    This replace the `initialize` method and the class initialization used in ROS 1.
  - `activate()`: Called when the node enters `on_activate` state, used for implement operations which are necessary before planner goes to an active state. In our case, we allocate memory and fill our internal costmap `state_map`, which is used by the D* (`dstar`) algorithm.
  - `deactivate()`: Called when the node enters `on_deactivate` state. This method is used for implementing operations which are necessary before planner goes to an inactive state. In our case, this method is not used and is left empty.

  - `cleanup()`: Called when the node enters `on_cleanup` state. This method is used for cleaning up resources which are created for the planner. In our case we destroys the allocated memory used by `state_map` and `dstar`.
  - `createPlan()`: Invoked when the planner server requests a global plan between a start and goal pose. It required the start and goal position as `geometry_msgs::msg::PoseStamped` and return a path `nav_msgs::msg::Path`. It replace the `makePlan` used in ROS 1.

- At the end of the C++ code, we added the following lines to export the planner as a plugin:

```
#include "pluginlib/class_list_macros.hpp"
PLUGINLIB_EXPORT_CLASS(<type of the plugin class>, <type of the base class>);
```

- Because it is a plugin,it is necessary to provide a description using a plugin declaration file. This is done by creating a file named `global_planner_plugin.xml` in the same directory as the `package.xml`. The `global_planner_plugin.xml` file specifies the plugin's class type and the base class it inherits from, following the format required by the pluginlib system. Then we edit the CMake file as described in the documentation[12].

Building the project

The system was built using the new build tool, `colcon`[13]. It is a command line tool to improve the workflow of building, testing and using multiple software packages. It automates the process, handles the ordering and sets up the environment to use the packages.
After successful compilation, the global planner is ready for use within the Nav2 framework.

[12]https://docs.ros.org/en/rolling/Tutorials/Beginner-Client-Libraries/Pluginlib.html
[13]https://github.com/colcon

Chapter 10

Jobot porting procedure to ROS 2

Elettra has also a Jobot, a 2WD rover still based on ROS 1, it was decided to port it to ROS 2 in order to effectively evaluate the use of Open-RMF.

10.1 Jobot Description

Jobot is a 6-wheeled rover with a 2WD configuration. It is equipped with a mini PC that collects data from an RPLIDAR A2 laser sensor and acceleration data from a serial bus connected to an Arduino microprocessor, which is linked to an IMU. An image of the robot is available in 2.2:
The steps taken to port the code in ROS 2 are described in the following section.

10.2 What changes where made in the code

The code use the serial ROS 1 library to communicate to the Arduino and retrieve the data from the encoders and the IMU. This library is no longer available in ROS 2, so to avoid rewriting the structure of the Jobot code, we utilized the same library that has been ported to ROS 2, which is available on GitHub: https://github.com/iwelch82/serial-ros2.git . The retrieved data is used to publish the odometry pose estimation of the robot, while a secondary node handles the IMU data.
The secondary node for publishing the IMU data was updated to ROS 2 by simply using the base node logic, whereas the main node was rewritten using the lifecycle structure. This logic is employed to configure the node during the on_configure method, and if it fails or a request for deactivation or cleanup is made, it stops and cleans everything. This structure allow to remotely control the passage through the states, for now and in our case it will be connected to an automatic Node manager used in nav2, the `lifecycle_manager` in `nav2_lifecycle_manager`. This node check all the states of the nodes that are connected to it and sequentially configures and activates them. If a crash occurs or if we stop a specific node's process, the lifecycle communicates the change of state and activates a procedure to stop the connected nodes. This node is only called in the launch file.
In addiction we updates the custom messages to ROS 2 that the Jobot use internally.
Once everything is configured, the node starts to publish the robot's state odometry

but does not publish static transforms between the sensors. These transforms are manually published using the `static_transform_publisher` provided by `tf2_ros`.

To minimize changes, the software is built inside a Docker container, and all applications are launched from it.

This, along with the same software used for navigation, enables this platform to function as an autonomous mobile robot (AMR).

10.3 Create a model for the simulation

For simulation purpose, we need to create a model of the Jobot in the URDF format, which will be used by Gazebo to build a 3D representation and generate data from the sensors equipped on it.
Since Jobot has a 2WD configuration, we will add two cylinders as wheels, which will be controlled by the Gazebo plugin `ignition::gazebo::systems::DiffDrive`. This plugin allows the robot to receive velocity commands and move accordingly.
Next, we add the sensors gazebo type`IMU` and `gpu_lidar` to simulate the IMU and the lidar on the robot. This procedure is similar to the one used for the Mini Rover from the Rover Robotics. At the end we obtain the following model:

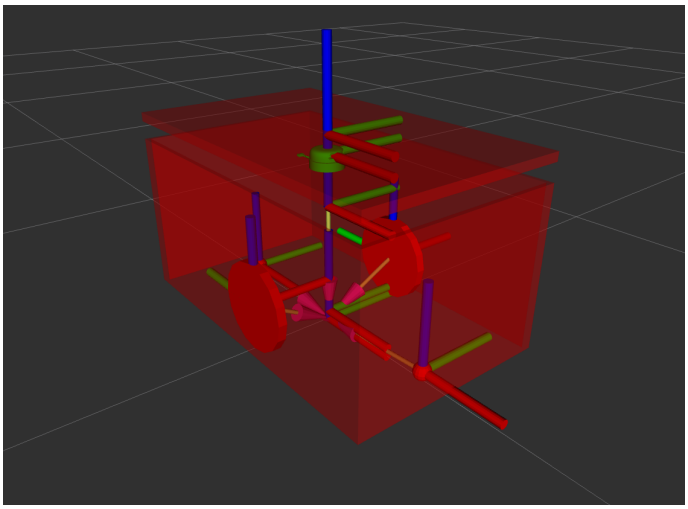

Figure 10.1: Jobot robot URDF model

Chapter 11

Integration of the navigation stack with Open-RMF

To integrate a fleet in Open-RMF, we first need information about the building's infrastructure, such as automatic doors or lifts that can be controlled, and the **route maps** that robots can use. **Route maps** are essential for predicting the paths robots will take, enabling Open-RMF to perform effective **traffic management** and avoid conflicts. All these information have to be stored in a file that can be generated using the **traffic editor tool**[1] [2], a tool provided by Open-RMF. The following section describes how the Traffic Editor works and how we used it in our implementation.

11.1 Traffic Management - Route Maps Generation

To begin describing the environment for Open-RMF, we created a new project in the Traffic Editor by generating a `.building.yaml` file. We then imported both the map produced by Cartographer and the floor plan of the facility.After importing the map,we defined the positions of key elements and planned the traffic flow for the floor, following best practices as outlined in the Open-RMF documentation[3]. The following components of the Traffic Editor were used to encode the necessary environment data:

- **vertex**: They are waypoints or a part of the description of walls, doors or else. Each vertex can have several properties:
  - `is_holding_point` : Indicates that the robot is allowed to wait at this waypoint for an indefinite period of time.
  - `is_parking_spot`: Designates a parking location for a robot.
  - `is_passthrough_point`: A waypoint where the robot should not stop.
  - `is_charger`: Identifies a charging station.
  - `is_cleaning_zone`: Used to indicate that this waypoint belong to a cleaning zone for the `Clean` task.
  - `dock_name`: Used for triggering docking behavior via RMF.

[1] https://github.com/open-rmf/rmf_traffic_editor
[2] https://osrf.github.io/ros2multirobotbook/intro.html
[3] https://osrf.github.io/ros2multirobotbook/integration_nav-maps-strategies.html

  - `pickup_dispenser`: Specifies a dispenser workcell for `Delivery` tasks.
  - `dropoff_ingestor`: Specifies an ingestor workcell for `Delivery` tasks.
  - `spawn_robot_type`: Used in simulation to spawn a robot model.
  - `spawn_robot_name`: Used in simulation to identify a spawned robot.

- `measurement`: Connects two vertex and is used to set the scale of the drawing. In our case, we used the width of a door.

- `door`: Connects two vertex and is used for interacting with the real world and simulate doors. If a robot encounters a door on its route, RMF sends a command to open it. When generating the simulation world model, a controllable door model is automatically added.

- `traffic lane`: Connects two waypoints and define the paths that fleets will follow. Lanes can be grouped, assigned to specific fleets, and configured as unidirectional or bidirectional. An optional "orientation" setting can be specified to require robots to move forward or backward along the lane. During planning, Open-RMF assumes that all traffic lanes are straight lines.

- `fiducials`: Used for multi-floor alignment and as anchors for the ROS 2 map. They provide spatial reference points between floors.

- `lift`: Is used for interacting with the real world and simulate elevators. If a robot goes to a lift, RMF sends a command to call it and, when the robot reach the waypoint, it send the lift to the designed floor. When generating the simulation world model, a controllable lift model is automatically added.

- `walls`: Connect two waypoints to define the location of a wall in the environment. When generating the simulation world, a wall is created between these points.

- `floor`: Add the ground plane for robots in the simulation world. This component can be modified to include holes (e.g., for elevator shafts).

- `environment assets`: Adds Gazebo models of various furniture and objects to the simulation, helping to create a realistic environment for testing and validation.

In the process of designing route maps, several strategic choices were made:

- Only one robot is allowed in a hallway at a time.

- Holding points (leaf nodes) were added along corridors or near their entrances and exits to allow one robot to yield to another already in the hallway. For certain task-specific waypoints, it is recommended to assign a very low speed limit to discourage RMF from using them as generic holding points.

- Mutex groups were defined on lanes and waypoints to ensure exclusive access, avoiding deadlocks by allowing only one robot to occupy a mutex-assigned group at any time.

At the end we obtained the following configuration map:

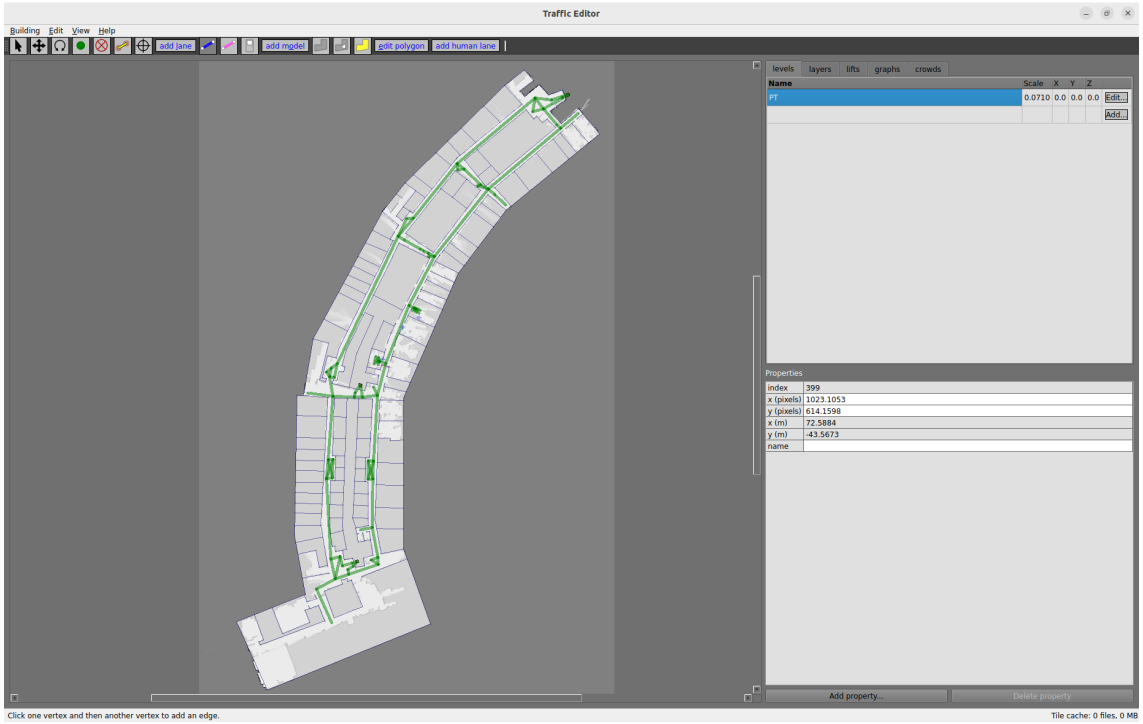

Figure 11.1: Traffic editor with Elettra's route map

The route map `.yaml` files for each graph can be generated using the following command:

```
ros2 run rmf_building_map_tools building_map_generator nav \
      ${building_map_path} ${output_nav_graphs_dir}
```

For the simulation, the world can be generated using the following command:

```
ros2 run rmf_building_map_tools building_map_generator ignition \
      ${building_map_path} ${output_world_path} ${output_model_dir}
```

And the following command can be used to download the models required for the simulation world:

```
ros2 run rmf_building_map_tools building_map_model_downloader \
      ${building_map_path} -f -e ~/.gazebo/models
```

The simulation environment generated from the configuration resulted in the following world:

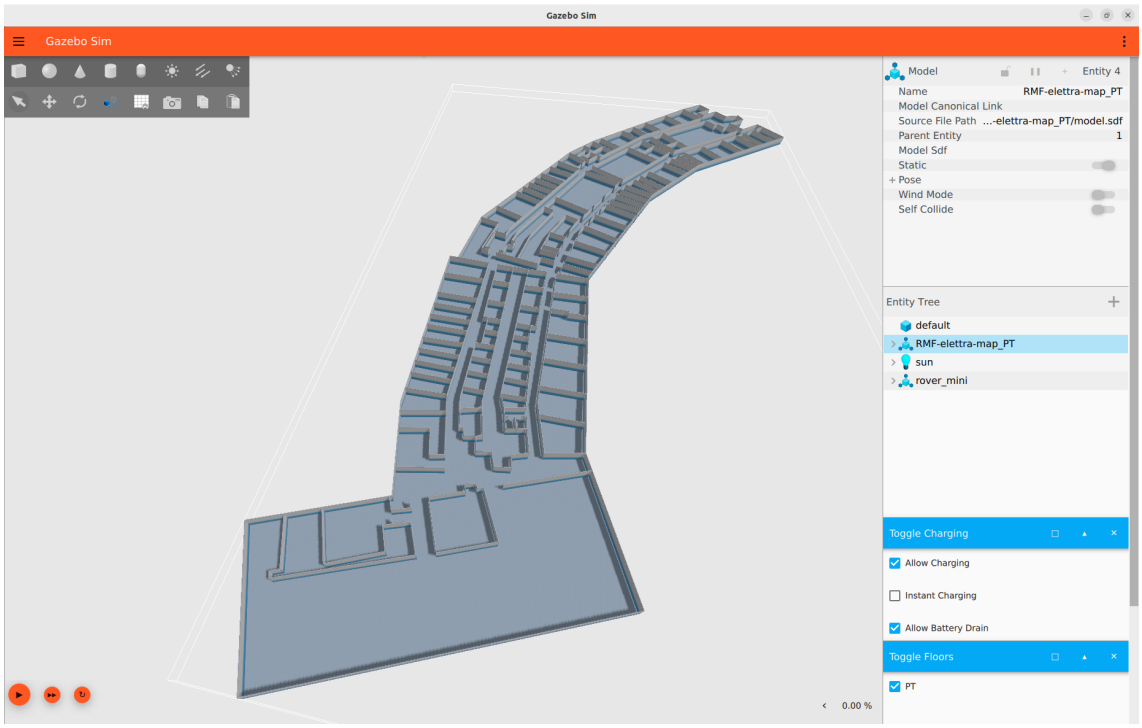

Figure 11.2: World generated using traffic editor

We can now launch the core components of RMF.

11.2 RMF core

This is the launch file used to start the core components of RMF:
`0_rmf_core.launch.xml`

This launch file starts the following nodes, which form the core structure of RMF:

- `rmf_traffic_schedule` : Manages traffic scheduling by controlling path structures, publishing itinerary topics, and handling conflict resolution.
- `rmf_traffic_blockade` : Acts as the Traffic Blockade Moderator, managing checkpoint information.
- `building_map_server` : Publishes map data, including points of interest such as parking spots.
- `visualization.launch.xml`: Optional nodes that launches visualization tools for monitoring and controlling RMF via rviz2.
- `door_supervisor`: Publishes control requests for doors using ROS communication structures.

- `lift_supervisor`: Publishes control requests for lifts using ROS communication structures.
- `rmf_task_dispatcher`: Allow to dispatch submitted task to the best fleet/robot within RMF.
- `queue_manager`: Manages reservations for parking spots and helps prevent conflicts.

Once the core system is launched, the next step is to connect our fleets using **fleet adapters**. Open-RMF provides the Free Fleet adapter, which is compatible with ROS 2 and Nav2. The following section will detail how this adapter works and how we integrated it into our system.

11.3 Free Fleet

Free fleet[4] is a python implementation of the `full_control` Open-RMF Fleet Adapter[5], which serves as a general template for communicating with a fleet manager. Its primary goal is to control robots running under both ROS 1 and ROS 2. It uses `zenoh` as a communication layer between each robot and the fleet adapter. Below is an overview of how communication is managed using the Free Fleet adapter:

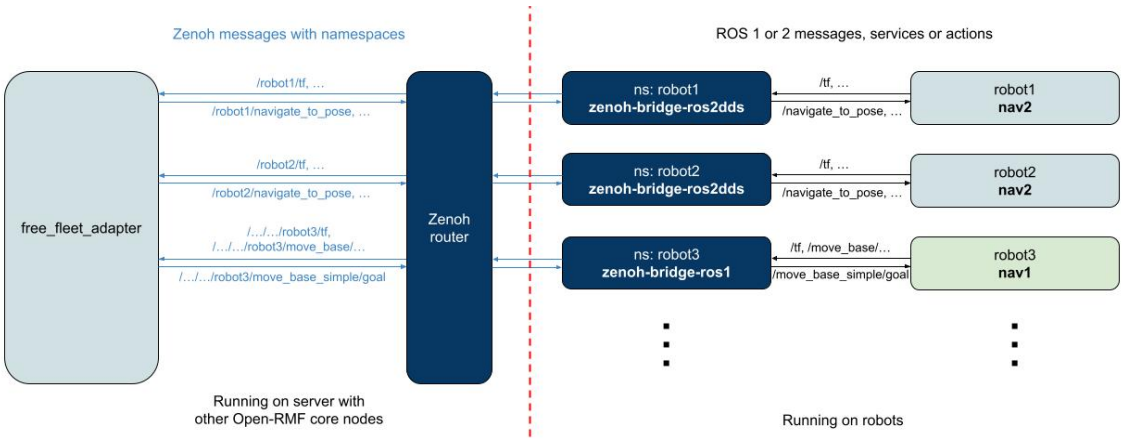

Figure 11.3: Free Fleet Adapter Communication

11.3.1 Setup fleet

To use the Free Fleet, it is necessary to create a `.yaml` configuration file for the fleet. This file describes the robots within the fleet that share the same characteristics. The configuration includes essential information such as the fleet name, velocity limit of the fleet, robot dimensions and mechanical details, power management parameters, the tasks the fleet are capable of performing, a list of all robots in the fleet with their individual names and their assigned chargers. It also need the positional correlation between the robot's map and the RMF map.
After setting up the fleet configuration, the next step is to configure Zenoh for communication.

[4]https://github.com/open-rmf/free_fleet
[5]https://github.com/open-rmf/fleet_adapter_template

11.3.2 Setup Zenoh

Zenoh[6] is a pub/sub/query protocol that unifies data in motion, data at rest and computations. It combines traditional pub/sub with geo distributed storage, queries and computations, while maintaining high efficiency in both time and space. Zenoh allows selective topic transmission over the network, reducing bandwidth usage to specific servers.
To set up Zenoh for use with Free Fleet, the following steps are required:

- Zenoh router must be running on a server. This router can be launched using the command: (`$ zenohd`). Multiple instances of the Zenoh router can be run for redundancy.
- On each robot and on the RMF server, the standalone executable `zenoh-bridge-ros2dds` must be launched with the associated configuration file. This executable bridges all ROS 2 communications using DDS over Zenoh. The configuration of this file depend on the machine. On the server it's configured to accept all the traffic from each robot, while on the robot we need to setup:
  - The `namespace`, which must match the fleet configuration file namespace.
  - Enabling the publication of the robot's frame status topics (`.*/tf` and `.*/tf_static`) and battery information (`.*/battery_state`)
  - Enabling communication with action servers controlling navigation (`.*/navigate_to_pose`)
  - Zenoh-specific configuration details, such as the communication mode, the server address, the protocol and the port for connection endpoints (e.g., `tcp/<ip address server>:7447`)

11.3.3 Launching Free Fleet

After completing the setup, we can use the following launch file to start the Free Fleet adapter:
`1rmf_mini_fleet_adapter.launch.xml`
This launch file takes as parameters the fleet configuration file and the route maps that the fleet will use. Once launched, it is possible to monitor the robot topics on the RMF server.
Also this is replicated for the Jobot fleet, launching the following file
`1rmf_jobot_fleet_adapter.launch.xml`

[6] https://zenoh.io/

11.4 Web interfaces

Open-RMF also offers a web-based interface, `rmf-web`[7], to visualize and control the system on a browser. It consists of two main components:

- api server : Based on the OpenAPI specification and implemented with the Python FastAPI library, it provides REST API methods to communicate with Open-RMF.
- dashboard : In our case, a demo dashboard that connects to the API server and allows control of the fleet.

Both components are available as Docker images and can be launched with the command:

```
docker compose -f /path/to/compose.yml up
```

Then we can connect to the web dashboard:

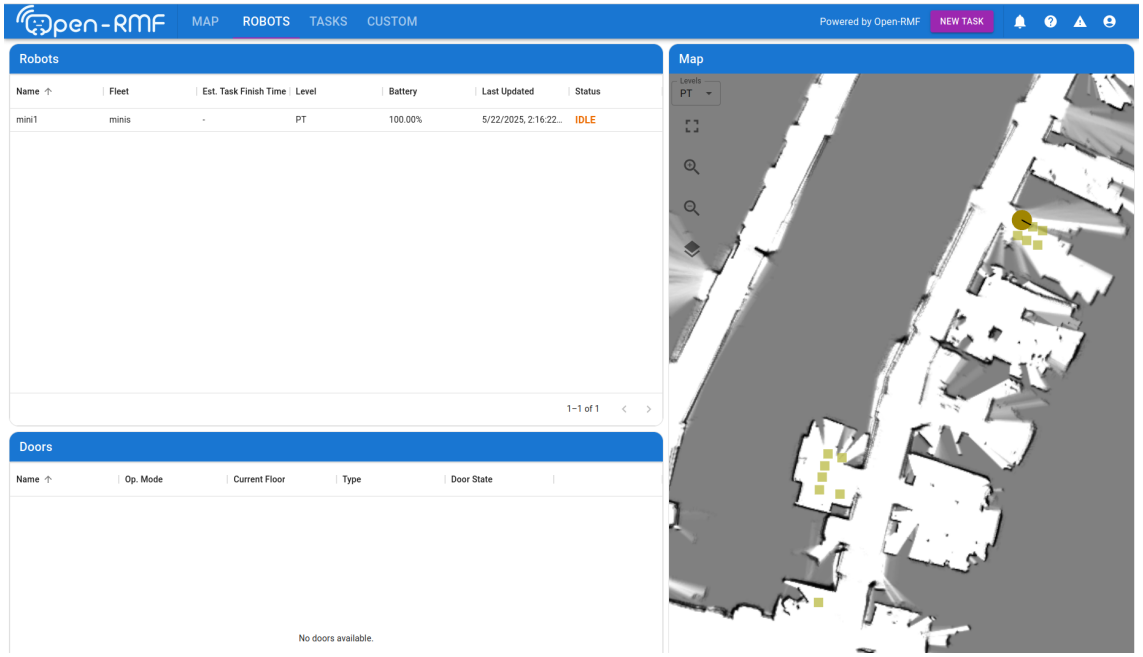

Figure 11.4: Open-RMF Web Dashboard

[7]https://github.com/open-rmf/rmf-web

Chapter 12

Testing and evaluation of the system

After some testing, these are the problems we found:

- The testing revealed some design issues with the rover. When operating in an office environment, the robot's laser sensor cannot detect certain parts of obstacles, as shown in the image below.

Figure 12.1: Laser position error

One possible solution is to reduce the free space from the occupancy map. However, the dimensions of the chair legs are quite long, significantly reducing the available space to pass through doors, which the planner currently prevents. Another option is to relocate the LIDAR sensor underneath the robot. This would require a low-profile LIDAR and would create blind spots by the wheels if no additional sensors are added on top. Alternatively, adding a depth

camera could provide a 3D occupancy grid, which can then be processed to generate a simpler 2D map.

- The RMF commands from waypoint to waypoint: RMF commands the robot from waypoint to waypoint, and it leaves the low level navigation to each robot's navigation stack. So if the robot chooses to use a non-straight route, it means that the RMF navigation graph might need to be re-drawn or reconsidered (if there are constant obstacles around), or the navigation stack of the robot should be tuned to allow tighter tolerances for static/dynamic obstacles. but how the robot chooses to navigate between them is out of RMF's control, since RMF does not have dynamic obstacle information, or if there are new static obstacles, it needs to be reflected into RMF's traffic navigation graph

- the communication is not the same as the ROS 1 with a central node and all node has to communicate to hei to share data, the ROS 2 infrastructure now allow that throught multicasting all devices within the same subnet are connected. to avoid to share data multiple solution are avaible some depending from the middleware used (cycloneDDS, FastRTS) or associate each robot with a `namespace` or the fastest one is to associated each robot with a unique `ROS_DOMAIN_ID`. The last solution is used, the only downside is the maximum number of robot allowed in the subnet is 255.

- The D-star planner is not designed if its path is completely blocked because it not update the costmap used internally at each request of the planner

- Setting up collision monitor for improve safety reason

- Additionally, we discovered misalignment between the planimetry image and the Cartographer map, as shown in the image below. It appears that at each major rotation, the system overestimates the robot's rotation angle during mapping.

- For feaster the deployment of the robots, we used the functionality of rosbag, a tool that allow to record the data published by some or all nodes and then republished after also with fastest velocity. This allot us to configure obtain the best outcome of process noise covariance of the localization node and the

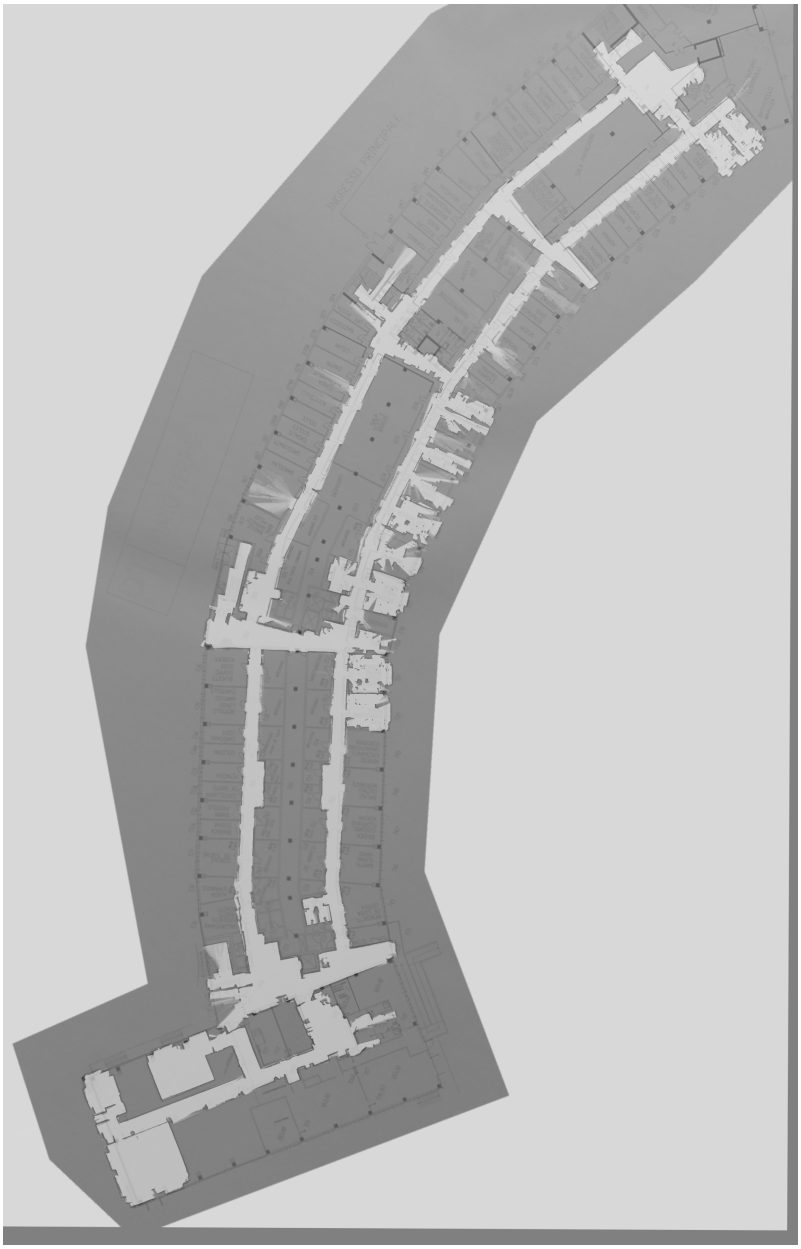

Figure 12.2: Overlap Cartographer Map and photo of floor plan

Part IV

Results and Analysis

Chapter 13

Presentation of results from testing and evaluation

This part is divided based on the differents thing that have been completed:

- porting the planner DStarGlobalPlanner from ROS 1 to ROS 2.
- given the robot platform rover mini build an autonomous mobile robot (AMR).
- given an the AMR jobot based on ROS 1 port it to ROS 2.
- implement the Open-RMF middleware using these robots.

13.1 Porting the planner

We have successfully port the code to ROS 2 and used for navigation. These are some output plan generated by the planner with different configuration parameters. More info about of the funtionality of the pramaters are described in its paper.

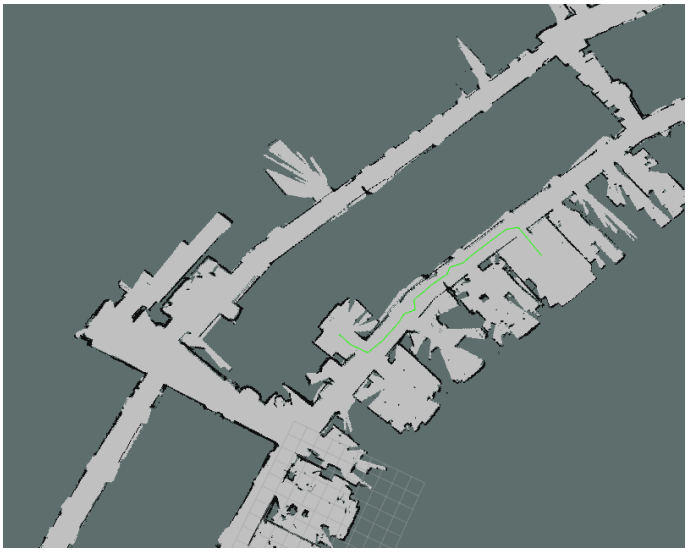

Figure 13.1: results obtained by the Dstar planner

In the end, the logic transformation between the move_base logic from ROS 1 to the

Planner Server used in ROS 2 for the global planner are based on the implementation of the state machine Lyfecycle and the application of more advanced functionality of the languages used for programming.

13.2 RoverRobotics Mini navigation result

This are some images that display the Mini during navigation:

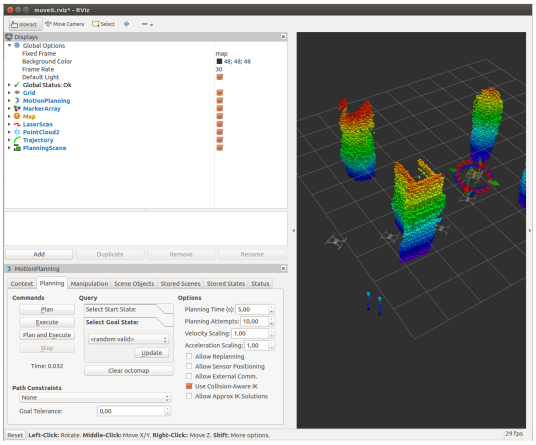

Figure 13.2: results navigation of Mini

The implementation was enough stray forward, building the robot and then apply the corrected nodes allow as to build a AMR. The tuning of the localization node and the localization node takes some time but is was reduced with the use of rosbag.

13.3 Jobot navigation result

This are some images that display the Jobot during navigation:

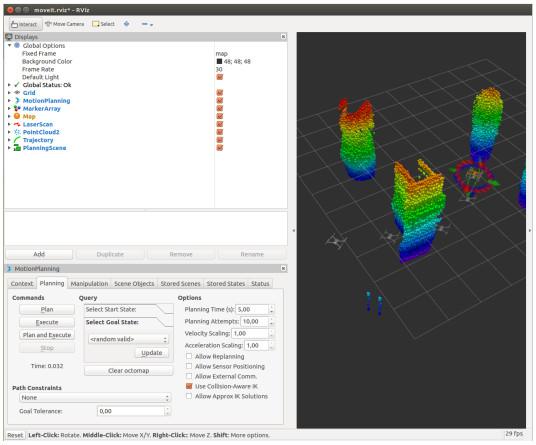

Figure 13.3: results navigation of jobot

Similar to the Mini, just more tuning.

13.4 Open-RMF navigation result

These are some images of the navigation of the 2 robots

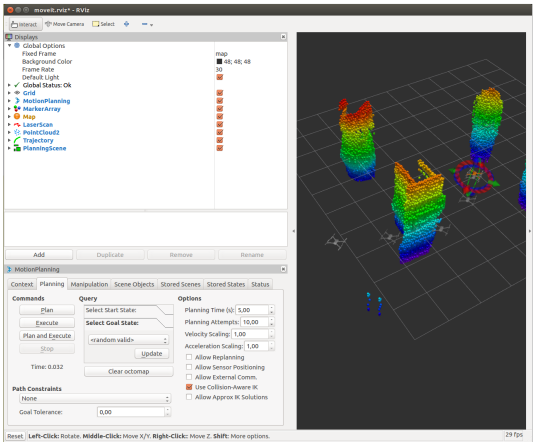

Figure 13.4: images navigation of the 2 robots with open-rmf

Chapter 14

Comparison of system performance before and after the porting process

This part is divided in section, in each section we evaluates the following:

- calculation time for the planner on Mini rover
- navigation response from ROS 1 to ROS 2

14.1 Planner calculation time comparisation before and after and vs new nav2 planner

In this section, we present the time required for the global planner node to calculate the path from the planners, comparing the performance between the Jobot robot with ROS 1 built-in and the Docker ROS 2 image using the standard planner.

Start->End	ROS 1 D-start	ROS 2 D-start	ROS 2 NavfnPlanner
ufficio -> magazzino	16.86	1.35	0.046
ufficio -> ingresso	73.238	1.35	0.073
ingresso -> disegnatori	60.41	1.35	0.081

Table 14.1: result planner first request

After the first iteration:

Start->End	ROS 1 D-start	ROS 2 D-start	ROS 2 NavfnPlanner
ufficio -> magazzino	0.025	1.35	0.052
ufficio -> ingresso	0.091	1.35	0.069
ingresso -> disegnatori	0.158	1.35	0.084

Table 14.2: result planner first request

these are some results generated using the planner Dstar

and these are some generated by the NavfnPlanner
the result showed us:

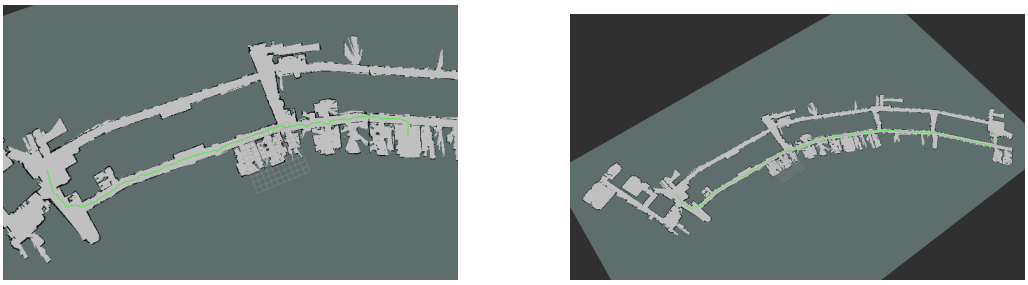

Figure 14.2: some of the plan generated by Dstar planner

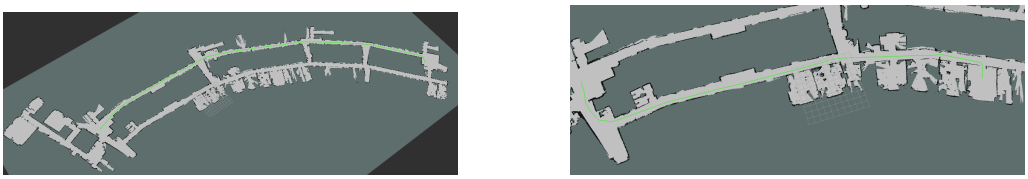

Figure 14.4: some of the plan generated by NavfnPlanner planner

- time to calculate the first plan using Dstar is very higher than the NavfnPlanner
- after the first calculation the performances are similar
- without a fine tuning of the parameters of the Dstar planner hasnt returned an sub-optimal path

while porting there are some advantages that suggested an sub optimal application for open-rmf. it has an implemented a load from a file of paths but these need to reverse ingegneer to wrote down a json file compatible with that. A solution is to implement in the code a saving option for the planner.

14.2 comparisation jobot navigation

the test are performed using the same points as used before we obtain the following

Start->End	ROS 1 D-start	ROS 2 D-start
ufficio -> magazzino	1.25	1.35
ufficio -> pausa lontana 1	1.25	1.35
magazzino -> pausa lontana 2	1.25	1.35

Table 14.3: result planner asking the node

conclusion

14.3 result analysis from Open-RMF

Chapter 15

Discussion on the challenges encountered and how they were addressed

the chalenges are:

- high error on localization moving forward, it over estimate the movement comparing the map
- communication throught the wifi, since that the

Part V

Conclusion and Future Work

Over this work we conclude the lot of benefits that the use of ROS 2 has made for an industrial standard.

Chapter 16

conclusion

advantages using ROS 2 and his middleware structure (Zenoh)

Chapter 17

Future work

improvement in the planner

Part VI

Appendices

Chapter 18

Simulation

In this part i will write some useful information for running the simulation. All test were done on debian based machine: To see the graphical interface launched in the docker container you need to have to share the access to the X server using the command `xhost local:root`.

today the tool for converting the data are https://www.ros.org/blog/noetic-eol/
implement also planning benchmark
describe and use composition in nav2 to obtain better performance [8]

Bibliography

[1] Sy-Hung Bach, Phan-Bui Khoi, and Soo-Yeong Yi. "Application of QR Code for Localization and Navigation of Indoor Mobile Robot". In: *IEEE Access* 11 (2023), pp. 28384–28390. DOI: 10.1109/ACCESS.2023.3250253.

[2] Luca Carlone et al. "Part1 Prelude". In: *SLAM Handbook. From Localization and Mapping to Spatial Intelligence*. Ed. by Luca Carlone et al. Cambridge University Press.

[3] Michele Colledanchise and Petter Ögren. *Behavior Trees in Robotics and AI*. July 2018. DOI: 10.1201/9780429489105. URL: http://dx.doi.org/10.1201/9780429489105.

[4] Tully Foote. "tf: The transform library". In: *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. Open-Source Software workshop. 2013, pp. 1–6. DOI: 10.1109/TePRA.2013.6556373.

[5] Wolfgang Hess et al. "Real-Time Loop Closure in 2D LIDAR SLAM". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1271–1278.

[6] Steve Macenski and Ivona Jambrecic. "SLAM Toolbox: SLAM for the dynamic world". In: *Journal of Open Source Software* 6.61 (2021), p. 2783. DOI: 10.21105/joss.02783. URL: https://doi.org/10.21105/joss.02783.

[7] Steve Macenski et al. *Impact of ROS 2 Node Composition in Robotic Systems*. 2023. arXiv: 2305.09933 [cs.RO]. URL: https://arxiv.org/abs/2305.09933.

[8] Steve Macenski et al. "Impact of ROS 2 Node Composition in Robotic Systems". In: *IEEE Robotics and Automation Letters* 8.7 (2023), pp. 3996–4003. DOI: 10.1109/LRA.2023.3279614.

[9] Steven Macenski et al. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[10] Steven Macenski et al. "The Marathon 2: A Navigation System". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020.

[11] Alexey Merzlyakov and Steven Macenski. "A Comparison of Modern General-Purpose Visual SLAM Approaches". In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2021.

[12] T. Moore and D. Stouch. "A Generalized Extended Kalman Filter Implementation for the Robot Operating System". In: *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Springer, 2014.

[13] DV Lu A. Merzlyakov M. Ferguson S. Macenski T. Moore. "From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the robot operating system 2". In: *Robotics and Autonomous Systems* (2023).

[14] Kavan Singh Sikand et al. *Robofleet: Open Source Communication and Management for Fleets of Autonomous Robots*. 2021. arXiv: 2103.06993 [cs.RO]. URL: https://arxiv.org/abs/2103.06993.

[15] Victor Mayoral Vilches et al. *SROS2: Usable Cyber Security Tools for ROS 2*. 2022. arXiv: 2208.02615 [cs.CR]. URL: https://arxiv.org/abs/2208.02615.

[16] Andrey Vukolov. "D-Star-Based Optimized Trajectory Planner for Mobile Robots Operating in Dense Environments". In: *New Trends in Mechanism and Machine Science*. Ed. by Giulio Rosati, Alessandro Gasparetto, and Marco Ceccarelli. Cham: Springer Nature Switzerland, 2024, pp. 294–301. ISBN: 978-3-031-67295-8.

[17] Andrey Vukolov. *ElettraSciComp/witmotion_IMU_QT: Version 0.9.17*. Version v0.9.17-alpha. 2022. DOI: 10.5281/zenodo.7017118. URL: https://doi.org/10.5281/zenodo.7017118.

[18] Andrey Vukolov et al. "Universal Configurable Navigation and Control System for Industrial Unmanned Ground Vehicles with Differential Chassis". In: *Advances in Mechanism and Machine Science*. Ed. by Masafumi Okada. Cham: Springer Nature Switzerland, 2023, pp. 287–301. ISBN: 978-3-031-45770-8.