

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



OMRON Mobile Robot ROS-based Control and Management

Miguel Cardoso Félix

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Pedro Gomes da Costa

October 24, 2023

Resumo

A utilização de robôs móveis de vários fabricantes está muitas vezes limitada ao *software* fornecido pelo próprio fabricante. Esta limitação implica uma enorme desvantagem quando se tenta integrar estes robôs móveis em sistemas multi-robô específicos. Além disso, há uma necessidade crescente na interoperabilidade entre os robôs de múltiplos fabricantes e os respetivos sistemas de navegação. Esta necessidade levou ao desenvolvimento de uma nova abordagem com o objetivo de permitir que um gestor de frota de robôs baseado em grafos e ROS (*Robot Operating System*) possa controlar robôs móveis de navegação em trajetórias livres. Esta é uma abordagem que pode ser aplicada a qualquer robô móvel autónomo (AMR), sendo que são necessárias algumas adaptações conforme o tipo de informação fornecida pelos diferentes fabricantes de robôs móveis.

A presente dissertação tem como objetivo desenvolver um módulo de *software* capaz de controlar o robô móvel OMRON LD-90 e integrá-lo com um gestor de frota de robôs baseado em grafos e ROS. Assim, neste trabalho explora-se o estado-da-arte em tipos de navegação de robôs móveis, estudam-se tipos de planeamento de trajetórias e explica-se a forma como o gestor de frota de robôs utilizado está organizado. De seguida, é analisado o *Robot Operating System* (ROS) e é introduzida a interface *Advanced Robotics Command Language* (ARCL), desenvolvida pela OMRON. É, ainda, realizada uma análise das arquiteturas escolhidas para a aplicação, incluindo a *framework* de comunicação entre os componentes que constituem o sistema bem como a conexão entre os módulos de *software* desenvolvidos. Explica-se ainda, com detalhe, todo o processo de implementação do *software* bem como a forma como este realiza a troca de mensagens com o gestor de frota de robôs. Os resultados apresentam o comportamento do robô móvel ao receber tarefas específicas do gestor de frota de robôs, permitindo a verificação da integração bem-sucedida do robô com o gestor de frota.

Abstract

The use of mobile robots from various manufacturers is often limited to the software provided by the manufacturer itself. This is a major drawback when attempting to integrate these robots into specific multi-robot systems. Additionally, there has been an increasing need for interoperability between robots from multiple manufacturers and their respective navigation systems. This has led to the development of a new approach aimed at empowering a graph and ROS-based robot fleet manager for the management of free-path navigation mobile robots. This approach can be applied to every Autonomous Mobile Robot (AMR), but some adaptations are necessary due to the different types of information provided by different mobile robot manufacturers.

The present dissertation aims to develop a software module capable of controlling the OMRON LD-90 mobile robot and integrating it with the graph and ROS-based robot fleet manager. Firstly, this work explores the state-of-the-art in mobile robot navigation types, studies various types of path planning, and explains the organization of this specific robot fleet manager. Then, the Robot Operating System (ROS) framework is analysed, and the Advanced Robotics Command Language (ARCL), developed by OMRON, is introduced. Additionally, an analysis of the chosen architectures for the application is performed, including the communication framework of the components that constitute the system, and the connections between the developed software modules. Subsequently, a thorough explanation of the process of implementing the software and how it exchanges messages with the robot fleet manager is provided. The results demonstrate the behavior of the mobile robot when receiving specific tasks from the robot fleet manager, allowing verification of the successful integration of the robot with the fleet manager.

Acknowledgements

Firstly, I would like to express my deepest gratitude towards my supervisor, Pedro Gomes da Costa, for the unwavering support, profound expertise, and guidance provided throughout the course of this dissertation. His insightful feedback, encouragement, and knowledge imparted were crucial to the success of this project, and for that, once again, I express my heartfelt appreciation.

Furthermore, I would like to extend my most sincere recognition to my company supervisor, Paulo Rebelo. His commitment, guidance, trust, and dedication were essential in supporting me every step of the way while even taking time from his own projects to enhance the quality of this work and contribute significantly to the project's success. I would also like to express a special appreciation to company members Héber Sobreira and Cláudia Rocha for their help and support throughout the development of this dissertation.

My journey would not have been possible without my family, who have been my unwavering pillar of support throughout life. My parents, Amélia Cardoso and José Félix, and my sister Inês Félix have been my rock, always standing by my side through the highs and lows of life. Thank you for your unconditional love, understanding, belief, and patience, and for giving me everything to achieve all my goals. Since in a family no one is forgotten or left behind, I would like to extend my gratitude to all my family members and my brother-in-law Francisco Alexandre, who are with me daily and continuously support me, being a constant source of motivation and strength.

Finally, I would like to extend a heartfelt appreciation to my friends, who are a crucial part of my life, providing continuous support and help. With them by my side, the challenges that appeared along the way were easy to surpass. They have made this journey memorable, a journey I will remember forever.

In loving memory, I would like to honor my cousin Paulo, my grandmother Alzira, my grandfather Félix and my grandfather Jacinto, who will always be with me, guiding me through life.

Miguel Cardoso Félix

Para o meu primo Paulo

Contents

1	Introduction	1
1.1	Motivation and Context	1
1.2	Goals	2
1.3	Structure of the Document	2
2	Background and Fundamental Aspects	5
2.1	Mobile Robots	5
2.1.1	Fixed-Path Navigation	5
2.1.2	Free-Path Navigation	7
2.2	Path Planning	9
2.2.1	Configuration Space	9
2.2.2	Decomposition Graph-Based Methods	10
2.2.3	Graph Search Algorithms	13
2.2.4	Free Path Planning Algorithms	15
2.3	Exploration of the Mobile Robot Market	15
2.4	Multi-Robot Systems	16
2.4.1	Centralized Control Architecture	17
2.4.2	Decentralized Control Architecture	18
2.5	Fleet Managers	19
2.6	Communication Between a Fleet of Mobile Robots and the Fleet Manager	20
2.6.1	Robotics Middleware Framework	20
2.6.2	VDA 5050 Standard for AGVs	21
2.7	INESC TEC Navigation Stack and Fleet Manager	21
3	Overview of Implemented Technologies and Architectures	23
3.1	ROS	23
3.1.1	ROS Filesystem Level	24
3.1.2	ROS Computation Graph Level	25
3.1.3	RViz	26
3.1.4	TF Coordinate Conversion System	27
3.2	OMRON	27
3.2.1	ARCL	27
3.2.2	MobilePlanner	28
3.3	Communication Framework	28
3.3.1	<i>multimaster_fkie</i> ROS Package	29
3.4	Software Architecture	30
3.5	Conclusion	32

4	Software Implementation	33
4.1	Controller Node	33
4.1.1	ARCL Configurations	33
4.1.2	Connection to the LD-90	34
4.1.3	Command Receiving in ROS	36
4.2	Connection Receiver Node	37
4.2.1	ARCL Configurations	38
4.2.2	Receiving Data From the Robot	40
4.2.3	Data Processing and Handling	41
4.3	Visualisation Package	42
4.3.1	Map Visualisation Node	42
4.3.2	Robot State Display Node	48
4.3.3	Goal Sender Node	49
4.4	Trajectory Converter Node	50
4.5	Fleet Manager Communication Node	52
4.5.1	Path Messages from Fleet Manager	52
4.5.2	Messages Sent to the Fleet Manager	54
4.5.3	Messages Sent to the Robot	55
4.5.4	Structure of the Node	55
4.6	Conclusion	55
5	Results and Discussion	57
5.1	Controller Node Results	57
5.1.1	<i>goto</i> Command	58
5.1.2	<i>getGoals</i> Command	59
5.1.3	<i>dock/undock</i> Command	59
5.2	Connection Receiver Node Results	60
5.3	Visualisation Package Results	61
5.3.1	Map Visualisation Node Results	61
5.3.2	Robot State Display Node Results	63
5.4	Trajectory Converter Node Results	64
5.5	Fleet Manager Communication Node Results	65
5.6	Validation and Conclusion of the Results	66
6	Conclusions and Future Work	69
6.1	Accomplishment of the Proposed Goals	69
6.2	Future Work	69
	References	71
A	OMRON Mission Structure	75
B	Videos Taken	77

List of Figures

2.1	Example of a graph. Reprinted from [1].	10
2.2	Example of a trapezoid decomposition. Reprinted from [2].	11
2.3	Example of a fixed cell decomposition. Reprinted from [2].	12
2.4	Example of a Quadtree decomposition. Reprinted from [2].	12
2.5	Input map considering the multiple temporal layers (a) and the analysed neighbor cells focusing the cell with the robot position (b). Reprinted from [3].	14
2.6	Architecture of the robot fleet manager.	21
3.1	The ROS filesystem level. Reprinted from [4].	24
3.2	The ROS computation graph level. Reprinted from [4].	25
3.3	Illustration of the ROS topic publish/subscribe concept.	25
3.4	Illustration of the ROS service request/response mechanism.	26
3.5	Mobile Planner Interface. Reprinted from [5].	28
3.6	Proposed Communication Framework.	29
3.7	Illustration of the process of discovering and syncing. Reprinted from [6].	30
3.8	Proposed Software Architecture.	31
4.1	ARCL server setup parameter configuration.	34
4.2	Client/Server socket connection diagram.	35
4.3	Example of the process of sending a command to the LD-90 through the <i>omron_controller</i> node.	37
4.4	Outgoing ARCL connection setup configuration.	38
4.5	Outgoing ARCL commands configuration.	39
4.6	Diagram of the <i>omron_connection_receiver</i> node.	41
4.7	Representation of a map in MobilePlanner.	43
4.8	Occupancy Grid of the Workspace.	47
4.9	Diagram of the <i>data_marker</i> node.	48
4.10	Diagram of the <i>robot_state_publisher</i> node.	49
4.11	Diagram of the <i>goal_sender</i> node.	50
4.12	Illustration of a trajectory with end-point vertices.	51
4.13	Example of a trajectory and the publishing of a sequence of edges by the <i>path_supervisor</i> node.	53
4.14	Structure of a <i>execution_route_states</i> message.	54
4.15	Diagram of the <i>omron_edges_to_vertices</i> node.	56
5.1	Terminal output of the <i>omron_controller</i> node when initialized.	57
5.2	Publishing of a <i>goto</i> command to the <i>/omron_ld90/send_command</i> topic using the terminal.	58
5.3	Response messages from the ARCL server for the <i>goto</i> command.	59

5.4	List of goals returned by the ARCL server.	59
5.5	Response messages from the ARCL server for the dock command.	60
5.6	Response messages from the ARCL server for the undock command.	60
5.7	Messages published on the <code>/omron_ld90/status/location</code> topic.	61
5.8	Messages published on the <code>/omron_ld90/laser_points</code> topic.	61
5.9	Map displayed in the MobilePlanner interface.	62
5.10	Representation of the map in RViz.	62
5.11	Representation of the occupancy grid in RViz.	63
5.12	Representation of the robot's position in the map.	63
5.13	Representation of a graph-based trajectory in RViz.	64
5.14	Representation of new <code>.map</code> file with a graph-based trajectory in RViz.	64
5.15	Message Received in the <code>/omron_ld90/target_route</code> topic.	65
5.16	Message Received in the <code>/omron_ld90/execution_route_states</code> topic.	65
5.17	Completion of the edge sequence sent by the robot fleet manager.	66
5.18	Mission sent to the Fleet Manager.	66
5.19	Fleet Manager Diagram with the OMRON LD-90.	67
A.1	Diagram of the nodes executed during the demonstration.	75

List of Tables

4.1	Key Parameters of the <i>visualization_msgs/Marker</i> Message	45
-----	--	----

Abbreviations and Symbols

AGV	Autonomous Guided Vehicle
AMCL	Adaptive Monte Carlo Localization
AMR	Autonomous Mobile Robot
ARCL	Advanced Robotics Command Language
A*	A-star
C_{free}	Free Space
C_{obstacle}	Obstacle Space
C_{space}	Configuration Space
EKF	Extended Kalman Filter
FLOW	Fleet Operations Workspace
GUI	Graphical User Interface
INS	INESC TEC Navigation Stack
IP	Internet Protocol
LIDAR	Light Detection and Ranging
MES	Manufacturing Execution Systems
MiR	Mobile Industrial Robots
MQTT	Message Queuing Telemetry Transport
PM	Perfect Match
PRM	Probabilistic Roadmaps
RRT	Rapidly Exploring Random Trees
RMF	Robotics Middleware Framework
ROS	Robot Operating System
RViz	ROS Visualization
SLAM	Simultaneous Localization and Mapping
TCP/IP	Transmission Control Protocol/Internet Protocol
TEA*	Time-Enhanced A*
UDP	User Datagram Protocol
VDA	German Association of the Automotive Industry
VDMA	German Association for Materials Handling and Intralogistics

Chapter 1

Introduction

1.1 Motivation and Context

The field of robotics is continuously expanding in various industries, revolutionizing the way industrial operations are conducted. Therefore, by integrating automated processes with robotic systems, complex tasks traditionally performed by humans can now be executed efficiently and with high precision.

One of the most remarkable advancements in the field of robotics is the evolution of autonomous mobile robots. These robots can operate alongside humans, in dynamic working environments, through perception systems, navigation systems, and localization systems. This type of robot can execute tasks while navigating and interacting with its surroundings without any human intervention [7]. However, for a mobile robot to navigate autonomously, it is required for the robot to know its location, where it wants to navigate, and how to navigate to that position. To do this, path planning algorithms are necessary for robots to determine optimal paths, avoid obstacles, and adapt to dynamic environments.

Mobile robots are used in a wide range of contexts, such as industrial automation, surveillance, construction, agriculture, and medical environments. Subsequently, the demand for these robots is constantly growing, leading multiple renowned manufacturers in the field of robotics to produce various types of mobile robots.

While the production of mobile robots by multiple manufacturers has brought many benefits, it has also led to the issue of each robot being highly dependent on utilizing software specifically produced by its manufacturer. Consequently, configuring, changing, or applying new functionalities to multiple robots from different manufacturers may be challenging and result in a constant need to adapt systems completely.

Additionally, there is an increasing urge to have fleets of mobile robots working simultaneously in an environment to perform tasks that a single robot cannot execute effectively. This results in the need for robot fleet managers, responsible for managing all the robots in the system. However, when a fleet manager employs path planning for a specific type of mobile robot navigation, the difficulty of integrating robots that use different types of navigation emerges.

Therefore, this dissertation aims to address the two mentioned needs in the field of mobile robotics, enabling the control of a mobile robot without relying on the manufacturer's own software and integrating this robot with a specific navigation system into a robot fleet that utilizes path planning designed for a different type of mobile robot navigation.

1.2 Goals

This dissertation has two main goals: **to implement a software module capable of controlling the OMRON LD-90 mobile robot**, a non-ROS-based autonomous mobile robot that mainly relies on the software provided by its manufacturer, and **to integrate this robot**, that uses free path planning algorithms, **into a robot fleet manager** that utilizes graph-based path planning algorithms.

The implementation of this system, capable of controlling and managing the LD-90 mobile robot, will be done within the Robot Operating System (ROS) framework. The robot will be controlled by sending and receiving commands through the Advanced Robotics Command Language (ARCL) interface, developed by OMRON, using TCP/IP sockets.

Integrating this robot with the robot fleet manager enables the fleet manager to have a heterogeneous fleet of mobile robots that rely on different types of navigation and are built by different manufacturers, resulting in more versatility and flexibility.

1.3 Structure of the Document

This document is structured, in a particular form, to present the chronological process of the project's development, consisting of the current chapter and five additional chapters. The following is a brief introduction and explanation of each chapter:

- **Background and Fundamental Aspects:** Focuses on the study of relevant approaches in the field of mobile robots and provides an understanding of the underlying theories behind these approaches and how they can be applied to this dissertation.
- **Overview of Implemented Technologies and Architectures:** This chapter analyses the technologies used in this application. Additionally, it discusses the architectures considered for communication between the application components and how the developed software modules are interconnected. It explains the thought process behind achieving the proposed goals of this dissertation.
- **Software Implementation:** Presents a detailed explanation of each software module implementation, covering the process of two-way communication between the application and the mobile robot, data handling, information display, and the required message exchange to integrate the robot with the fleet manager.

- **Results and Discussion:** This chapter presents the tests conducted for each individual software module and their impact on the correct functioning of the entire application. It validates the developed algorithms and reviews the obtained results.
- **Conclusions and Future Work:** Provides an overview of the developed project and how the proposed goals were achieved. Additionally, possible improvements for the project are presented as future work.

Chapter 2

Background and Fundamental Aspects

In order to establish a strong foundation for understanding the context and key concepts that form the basis of this dissertation, this chapter provides a comprehensive overview of relevant theoretical aspects and fundamental knowledge related to the research topic. This is achieved by performing a profound analysis of the relevant state-of-the-art literature, theoretical frameworks, and historical developments, enabling a deeper comprehension of the research problem and facilitating exploration in the subsequent chapters.

2.1 Mobile Robots

Mobile robots have revolutionized various industries by introducing automation and autonomy into the realm of transportation and logistics. They encompass a wide range of robotic systems designed to navigate and operate in different environments and carry out tasks traditionally performed by humans. As a result, this field is experiencing rapid expansion in scientific research. For a robot to be autonomous, it must determine the sequence of actions required to complete a task and have a perception system to aid its decision-making process [8].

A significant step forward in the mobile robot industry is the emergence of Autonomous Mobile Robots (AMRs) as successors to Autonomous Guided Vehicles (AGVs). AMRs are capable of operating in dynamic and partially unknown environments [7], requiring them to navigate and avoid obstacles seamlessly, in contrast to AGVs. In terms of navigation methods, the AGVs rely on fixed paths such as electromagnetic, optical, and tape navigation, while others, the AMRs, employ free route types like inertial, laser, and visual navigation [9].

2.1.1 Fixed-Path Navigation

In fixed-path navigation, mobile robots are required to follow a predefined path. This approach encompasses various methods such as wire-guided navigation, which utilizes buried cable networks to guide the robots along the desired path, or line-guided navigation, where the robots follow

painted lines on the floor. These lines can be created using visible, invisible, or reflecting materials to ensure the robot's guidance. Fixed-path navigation is widely employed in warehouses and automatic factories due to its high reliability [10].

One significant challenge related to the associated drawbacks of this path navigation method is the difficulty of modifying the predefined paths once they are established, as it often requires physical modifications to the infrastructure. Additionally, the initial implementation cost of setting up the path infrastructure, such as laying down cables or painting lines, can be relatively high.

In the following, some fixed-path navigation methods will be presented, exploring their functioning principles, benefits, and drawbacks.

2.1.1.1 Electromagnetic Navigation

This method of fixed-path navigation involves placing metal wires along the path of the AGV and generating a magnetic field by applying low-frequency and low-voltage currents around these wires [11]. An induction coil on the robot allows it to detect and track the strength of the magnetic field, enabling navigation.

Electromagnetic navigation offers several advantages, including high reliability and precise control accuracy, making it suitable for applications where precise positioning is crucial. However, there are certain challenges associated with this method. It can be complex and time-consuming to initially set up the metal wires and generate the required magnetic field. Reconstructing or expanding the path may also pose challenges, as it requires additional wiring and modifications [11]. Moreover, the initial installation and maintenance costs of the system can be relatively high.

2.1.1.2 Optical Navigation

Another commonly used method for fixed-path navigation is optical navigation, which utilizes a continuous belt made of luminescent material laid on the ground or luminescent paint applied to the intended route [11]. This method requires two infrared sensors positioned symmetrically at the bottom of the AGV to detect reflected light. By measuring deviations from the expected light patterns, it becomes possible to enable control.

Optical navigation offers advantages in terms of flexibility and cost-effectiveness. The continuous belt or luminescent paint used in this method can be easily modified or extended to adapt to changing requirements or routes. Additionally, the use of infrared sensors allows for real-time monitoring of the reflected light, enabling precise control and adjustments. However, it is important to note that the optical navigation system may require adequate lighting conditions and periodic maintenance to ensure accurate detection and reliable operation.

2.1.1.3 Tape Navigation

As mentioned previously, another method of fixed-path navigation is tape navigation, which utilizes a magnetic induction navigation sensor and a magnetic guide belt laid on the ground [11]. The

controller utilizes the information obtained from the relative one-dimensional coordinate signal between the magnetic induction sensor and the magnetic guide belt to control the robot, enabling it to follow the magnetic guide belt based on the signal state.

This method offers several advantages. It provides a simple and cost-effective solution for achieving precise path following in indoor environments. The magnetic guide belt can be easily installed, allowing for quick modifications or expansions of the intended path by adding or repositioning the tape. Moreover, tape navigation systems often have low power requirements and can operate efficiently over extended periods.

However, it is important to consider some limitations of tape navigation. The system's performance can be affected by the presence of surrounding metals, which may introduce interference and affect the accuracy of the navigation [11]. Additionally, the tape used for the magnetic guide belt can be susceptible to contamination, potentially causing deviations or errors in the robot's trajectory. Regular inspection and tape cleaning are necessary to ensure reliable navigation.

2.1.2 Free-Path Navigation

In free-path navigation, the movement of the mobile robot is planned in real-time, allowing the robot to autonomously navigate through an environment without predefined paths or constraints. This approach provides flexibility and adaptability, as the robot can dynamically plan and adjust its path based on real-time perception of its surroundings.

2.1.2.1 Inertial Navigation

One common method used in free-path navigation is inertial navigation, which employs a gyroscope installed on the AMR. The gyroscope measures the robot's rotation and provides valuable information about its orientation. Additionally, a positioning block is installed on the ground within the robot's working environment. By combining the gyroscope's deviation signal with the ground positioning block signal, the AMR can accurately determine its position [11].

While inertial navigation offers numerous advantages, it is not without its difficulties. Using such advanced technology, including gyroscopes and sensor systems, contributes to its effectiveness but results in higher costs. Furthermore, gyroscopes are sensitive to vibrations, which can introduce errors in the navigation system [11]. Proper measures need to be taken to mitigate these vibrations, such as vibration-dampening mechanisms or integrating complementary sensors for robust navigation.

2.1.2.2 Laser Navigation

Laser navigation is another commonly used method for free path navigation. It involves the emission and reception of laser beams by a sensor, typically a LIDAR (Light Detection and Ranging) device. LIDAR sensors emit laser beams that sweep the surrounding environment, measuring the time the beam takes to travel and return. This enables the measurement of distances to objects by illuminating them with pulsed laser light and detecting the reflected pulses using a sensor [12].

By analysing these measurements, laser navigation allows for the determination of distances and angles to surrounding objects with high precision.

This method utilizes LIDAR-based Simultaneous Localization and Mapping (SLAM) algorithms. These algorithms process the scattered laser reflections collected by the LIDAR, generating a point cloud that represents the spatial distribution of objects in the environment [13]. By integrating the point cloud data with robot motion, SLAM algorithms simultaneously map the environment and accurately localize the robot within it.

LIDAR sensors come in various types, including 2D and 3D variants [12]. When comparing the modern 3D LIDARs to the traditional 2D LIDARs, the newer ones not only add an extra spatial dimension but also offer an increased number of scanning layers, resulting in wider fields of view and extended range capabilities.

It is important to note that laser navigation offers several advantages, including its ability to provide highly accurate and reliable results. Laser beams are not significantly affected by environmental lighting conditions, making this method suitable for various lighting environments. However, it is worth considering that laser navigation can be associated with high costs due to the expense of LIDAR devices and the computational requirements of SLAM algorithms [11].

2.1.2.3 Visual Navigation

The visual navigation method is a rapidly developing method in the field of mobile robots, extensively used by AMRs. It involves the utilization of cameras or visual sensors to capture image information of the surrounding environment [11]. These images are then processed using computer vision techniques to extract relevant visual features, such as edges, corners, or textures.

With the aid of SLAM algorithms, visual navigation enables the robot to estimate its real-time position and orientation in real-time while creating a map of the environment. By comparing the extracted visual features with the map, the robot can accurately determine its location and plan the optimal path to its destination. Additionally, visual navigation plays a crucial role in obstacle avoidance by continuously analysing captured images to identify potential obstacles and hazards. The robot's path planning algorithm incorporates this visual information to generate collision-free paths, ensuring safe and efficient navigation.

Utilizing sensors and SLAM technology, it becomes possible to track the trajectory of the mobile robot and provide feedback position correction. In dynamic environments or encountering traffic, the AMR can adapt by adjusting its speed or coming to a stop. Consequently, all decisions pertaining to guide path selection, obstacle avoidance, and routing are autonomously made by the mobile robot itself. This empowers the mobile robot to navigate effectively and autonomously in complex environments, responding to real-time changes and ensuring the successful completion of its tasks [9].

2.2 Path Planning

One other crucial aspect of mobile robot navigation, closely related to the previous topic of navigation methods, is path planning. Path planning involves the use of algorithms that consider the spatial layout, obstacles, terrain, and other relevant information to create paths that robots can autonomously and safely follow within their workspace environment, with the goal of reaching a specific endpoint from a specific starting point [1].

In addition to path planning, it is important to distinguish two other concepts: trajectory planning and motion planning. Trajectory planning involves determining the robot's trajectory as a function of time, meaning that the position of the mobile robot is known at every time instant [2]. It focuses on generating a smooth and feasible path that considers time-varying constraints and objectives. Trajectory planning ensures precise control of the robot's motion, enabling it to navigate through complex environments while adhering to desired temporal constraints.

On the other hand, motion planning takes into account the kinematic and dynamic restrictions of the robot [2], considering factors such as its physical capabilities, velocity limits, acceleration limits, and turning radius. Motion planning algorithms ensure that the generated paths are executable by the robot, taking into consideration its physical constraints and capabilities. It aims to optimize the robot's motion while considering its kinematic and dynamic limitations, ensuring both safety and efficiency during navigation.

Furthermore, within the context of path planning, it is important to be familiarized with another term known as optimal path planning. Optimal path planning involves the use of a cost function that takes into account aspects such as distance traveled or time. The goal is to find a set of paths that optimize this cost function, identifying the path that minimizes or maximizes the desired criteria [1]. Optimal path planning algorithms employ various techniques, such as heuristic search or mathematical optimization, to efficiently explore the search space and identify the most favorable path based on the given criteria.

2.2.1 Configuration Space

Another important term to be familiar with when considering path planning is the configuration space. The configuration space, denoted as C_{space} , is a fundamental concept used to address the challenge of navigating in a given working environment with a mobile robot. It represents the set of all possible configurations and positions that the robot can assume [14].

To construct the configuration space, it is necessary to have knowledge of the map or layout of the working environment where the robot operates. The configuration space is then defined based on this information, representing the entire space in which the robot can move.

The C_{space} can be divided into two main components: the free space, C_{free} , and the obstacle space, C_{obstacle} . The C_{free} represents the regions within the configuration space where there are no obstacles, allowing the robot to move and traverse freely [2]. This space corresponds to the areas accessible by the mobile robot without encountering any obstacles or physical constraints.

On the other hand, the C_{obstacle} represents the positions or regions within the configuration space that are occupied by obstacles or are inaccessible to the mobile robot due to various constraints [1]. These obstacles could include walls, furniture, or any other objects present in the environment that pose restrictions on the robot's movement.

2.2.2 Decomposition Graph-Based Methods

There are various path planning methods, which can be divided into four categories: bio-inspired methods; mathematical model-based methods; sampling-based methods; and decomposition graph-based methods [1]. In the context of this dissertation, the methods that will be explored are the decomposition graph-based methods.

In decomposition graph-based methods, the fundamental idea is to represent the working environment as a grid, where the C_{space} is divided into a set of cells. It is determined which cells are free (C_{free}), which cells are occupied (C_{obstacle}), and which cells correspond to start or end nodes. By partitioning the configuration space into these categories, a graph structure can be constructed to establish connections between the cells [1].

The graph structure represents a map of the working environment as a collection of vertices and edges. Figure 2.1 illustrates an example of a graph structure.

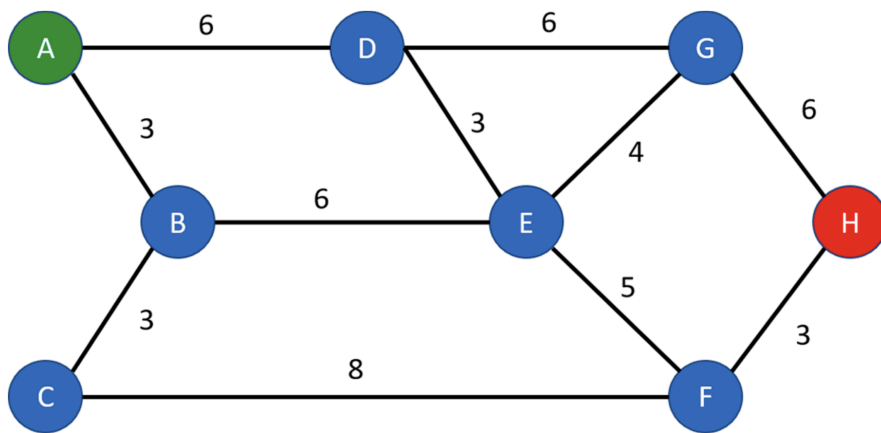


Figure 2.1: Example of a graph. Reprinted from [1].

2.2.2.1 Cell Decomposition

This method refers to dividing the C_{space} into cells and determining whether each cell is free or occupied. The connections between neighboring cells are then established, creating a graph structure. To find a path, a starting point is assigned to one cell and a destination point to another cell. Graph search algorithms are employed to determine the sequence of cells that must be traversed to follow the desired path [2].

The cell decomposition method can be further categorized into two types:

- Exact cell decomposition.
- Approximate cell decomposition.

2.2.2.2 Exact Cell Decomposition

In the exact cell decomposition approach, the configuration space, C_{space} , is divided into cells with precise boundaries. Each cell is rigorously determined as either free (C_{free}) or occupied (C_{obstacle}) based on the characteristics of the environment. The decomposition of the C_{space} aims to create cells with relatively simple geometries, such as convex polygons and trapezoids [2], which facilitate the calculation of paths between cells and the identification of neighboring cells.

By using cells with well-defined boundaries and simple geometries, path planning algorithms can efficiently determine paths within the environment. The simplicity of the cell shapes allows for easier computation of paths, as well as straightforward identification of neighboring cells. This approach simplifies the navigation process by providing a structured representation of the environment that is conducive to path planning and graph-based algorithms.

For example, in the case of trapezoid decomposition or vertical decomposition, the free space can be divided into trapezoidal regions using vertical lines originating from each of the obstacle vertices. By employing vertical lines from the vertices of obstacles, the free space is divided into trapezoidal regions [14]. The trapezoid decomposition approach provides a straightforward and efficient way to partition the free space based on the geometry of the obstacles. It is possible to observe an example of this decomposition in Figure 2.2.

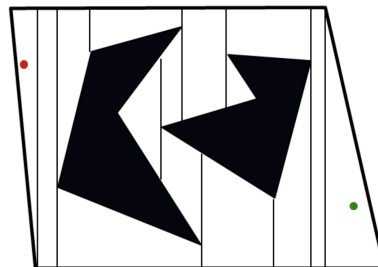


Figure 2.2: Example of a trapezoid decomposition. Reprinted from [2].

2.2.2.3 Approximate Cell Decomposition

This method adopts a more flexible approach to the division of the C_{space} , allowing for three types of cell classification: free, occupied, or semi-occupied. The cells in this method typically have basic geometric shapes, such as squares, which simplifies the construction of the C_{space} in a straightforward and efficient manner [2].

It is important to note that in approximate cell decomposition, the accuracy of the representation of the real working environment is influenced by the size of the cells.

There are various methods for approximate cell decomposition, but the main two are:

- Fixed cell decomposition.
- Quadtree decomposition.

For the approach of fixed cell decomposition approach, the C_{space} is divided into cells of a fixed and predefined size [2], typically squares, as depicted in Figure 2.3.

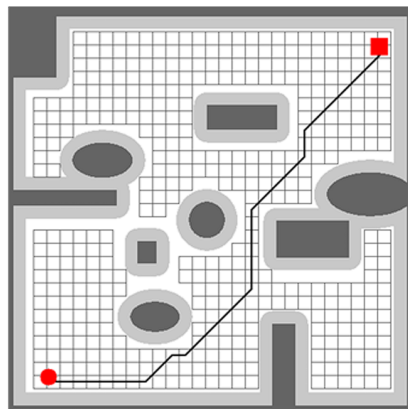


Figure 2.3: Example of a fixed cell decomposition. Reprinted from [2].

In the case of Quadtree decomposition, it takes a recursive approach to divide the C_{space} . It begins by dividing the space into four equal cells, and when a cell does not belong to the free space (C_{free}), it is further subdivided into four smaller cells [2], as shown in Figure 2.4. This method aims to reduce the number of points required to represent obstacles compared to a full grid representation [14], resulting in faster execution times for search algorithms. However, Quadtree decomposition comes with a higher implementation complexity.

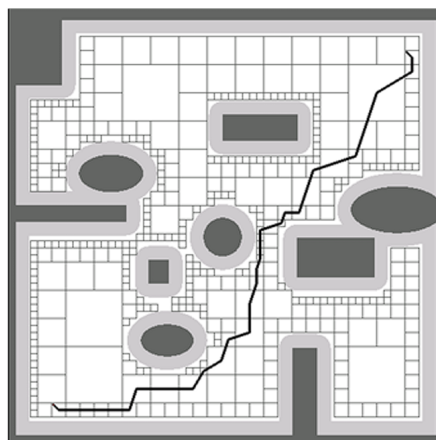


Figure 2.4: Example of a Quadtree decomposition. Reprinted from [2].

2.2.3 Graph Search Algorithms

After creating a graph representation of the configuration space (C_{space}) using one of the methods discussed earlier, the next step is to find the optimal path between two specified points. To accomplish this, a graph search algorithm must be employed. It is essential to select the appropriate algorithm that not only accurately determines the optimal path but also operates efficiently in terms of computational resources and runtime.

One major group of algorithms used in path planning is heuristic-based, which considers the cost of traversing edges between vertices. These algorithms incorporate a heuristic function that estimates the cost-to-go or the expected distance to the goal while considering the cost incurred until that point. The heuristic guides the algorithm to prioritize paths that are expected to lead to the goal more quickly, improving the efficiency of the search process.

Examples of heuristic-based algorithms for path planning include the A* algorithm and the TEA* algorithm.

2.2.3.1 A* algorithm

The main idea of the A* (A star) algorithm is to explore the search space by considering both the cost of the path already taken (the accumulated distance) and an estimate of the remaining cost to reach the goal (the heuristic estimate). For path planning, this heuristic estimate is typically based on some measure of distance between the current node and the goal. The Euclidean distance or the Manhattan distance are usually used [1].

The Euclidean distance, as shown in Equation 2.1, calculates the straight-line distance between two points, represented in their coordinates (x_n, y_n) and (x_g, y_g) .

$$\text{Euclidean Distance: } H(x_n, y_n) = \sqrt{((x_n - x_g)^2 + (y_n - y_g)^2)} \quad (2.1)$$

The Manhattan distance, as shown in Equation 2.2, calculates the distance between two points by summing the absolute differences of their coordinates.

$$\text{Manhattan Distance: } H(x_n, y_n) = |(x_n - x_g)| + |(y_n - y_g)| \quad (2.2)$$

In the A* algorithm, each node in the search space contains information about the initial distance from the starting node, and the sum of this distance and the estimated distance to the target node. During each iteration, the algorithm selects the node with the highest probability of being on the shortest path between the start and the target [1].

To calculate the cost for each node, the algorithm uses a heuristic function $h(n)$ that estimates the cost of the cheapest path from the starting node to the destination node. Additionally, a function $g(n)$, which may or not be heuristic, is used to represent the cost of the path already taken. The sum of these two functions determines the cost of a node [2], as represented in Equation 2.3.

$$f(n) = h(n) + g(n) \quad (2.3)$$

This algorithm has the advantage of being able to find multiple optimal solutions in the search space, but it can be computationally intensive. It is particularly suitable for static working environments where the layout remains unchanged. However, in the case of multi-robot systems, additional complexities arise due to the presence of obstacles and the need to handle dynamic changes in the environment, which pose challenges for path planning. To address this issue, multiple extensions of the A* algorithm have been developed, such as the TEA* algorithm.

2.2.3.2 TEA* Algorithm

The Time-Enhanced A* algorithm (TEA*) was developed as an extension of the A* algorithm to address the challenges of path planning in multi-robot systems [3]. It aims to navigate around obstacles, avoid deadlocks, and ensure the efficient execution of a set of tasks. The TEA* algorithm builds upon the principles of the A* algorithm while incorporating additional features.

In the TEA* algorithm, the input map is represented in three dimensions, consisting of the map's coordinates (x and y) and a temporal dimension, as shown in Figure 2.5(a). The temporal layer is represented by layers $k = [0, T_{\max}]$, where T_{\max} is the maximum number of layers [3]. The map can be obtained through cell decomposition of the C_{space} , resulting in an occupancy grid with free and occupied cells.

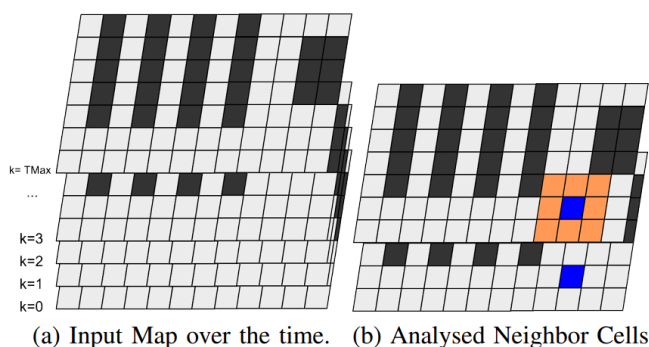


Figure 2.5: Input map considering the multiple temporal layers (a) and the analysed neighbor cells focusing the cell with the robot position (b). Reprinted from [3].

In TEA*, with the addition of the time domain, several features are incorporated [3]. The position of each robot is calculated in each temporal layer, and the analysed neighbor cells of the current position of the robot belong to the next temporal layer, represented in Figure 2.5(b). This ensures proper temporal coordination and movement planning. Additionally, the neighbor cells include the cell containing the robot's current position, which is depicted as the blue cell in Figure 2.5(b).

When applied to multi-robot systems, the TEA* algorithm initially converts the current positions of the robots into obstacles. This enables each robot to consider the positions of other robots as occupied cells, thereby avoiding deadlocks. These cells are treated as obstacles only in the temporal layers $k = \{0, 1\}$ [3]. By doing so, the robot acknowledges the occupation of a

specific path in the initial moments but can generate alternative obstacle-free paths in subsequent moments, effectively resolving the deadlock issue.

The algorithm then proceeds by analysing the list of missions, which consists of a robot number and a task assigned to it, and calculates the path for each robot. Before the next mission is analysed, the calculated path is converted into a moving obstacle for the remaining robots, ensuring collision avoidance and efficient coordination [3].

This iterative process continues in a loop, allowing for the updating of robot positions and the recalculation of paths for each subsequent mission on the list. By dynamically adapting to changing environments and mission requirements, the TEA* algorithm facilitates effective multi-robot path planning.

2.2.4 Free Path Planning Algorithms

In the case of certain mobile robots, the employed algorithms fall under the category of free path planning algorithms. These algorithms utilize various techniques, like potential fields, rapidly exploring random trees (RRT) [15], or sampling-based methods like Probabilistic Roadmaps (PRM) [16], to generate feasible paths. The primary goal of these algorithms is to plan collision-free paths while optimizing metrics such as distance and/or time.

All these algorithms take into consideration the robot's kinematics, such as its physical limitations and constraints on velocity and acceleration, to ensure that the generated paths are within the robot's capabilities. Additionally, they also consider the geometry of the environment and obstacles, enabling the robot to plan paths that navigate efficiently and safely through complex environments.

While free path planning algorithms use continuous techniques to explore the configuration space and find paths, graph search algorithms, as mentioned in 2.2.3, discretize the configuration space into a graph, where vertices represent valid robot configurations, and edges represent feasible transitions between configurations. One advantage of free path planning algorithms is their ability to handle dynamic and complex environments effectively, as they can adapt to real-time changes. They offer flexibility in handling different robot types and environmental conditions. On the other hand, graph search algorithms can be more computationally efficient, especially in scenarios with low-dimensional configurations spaces, where the graph can be constructed relatively quickly.

2.3 Exploration of the Mobile Robot Market

Today, numerous companies are at the forefront of developing advanced mobile robots, each offering unique features and capabilities. These robots employ different navigation methods, catering to diverse application requirements. Furthermore, the price ranges of these mobile robots vary based on their sophistication, functionalities, and intended use.

Several prominent companies have emerged as key players in the mobile robot market, delivering cutting-edge solutions for a wide range of industries, including [17]:

- **OMRON:** OMRON is a renowned manufacturer in the industrial automation industry with extensive expertise in sensing and control technology. The LD Mobile Robot is a self-navigating AMR designed for operation in dynamic and demanding environments. Unlike traditional navigation methods that require facility modifications like beacons or floor magnets, the LD Mobile Robot relies on advanced sensors for seamless navigation, which translates into cost savings and operational flexibility. The robot is equipped with laser scanners that allow it to detect and recognize natural features in its environment, using free path planning algorithms to generate paths. These sensors enable the robot to create a map of its surroundings and navigate through complex spaces with precision. Additionally, the LD Mobile Robot is equipped with a safety-rated laser that serves both for SLAM and safety functionality. When encountering an obstacle, the robot responds by appropriately reducing its speed or coming to a complete stop, ensuring safe operation. The LD mobile robot offers a reliable and efficient solution for autonomous navigation, enabling it to adapt to changing environments and operate smoothly in various industrial settings. With its robust sensor suite and intelligent navigation capabilities, it exemplifies the cutting-edge advancements in autonomous mobile robotics.
- **Locus Robotics:** Locus Robotics is a prominent company in the field of mobile robotics, specializing in AMRs for warehouse automation and fulfillment operations. Their robots are specifically designed to collaborate with human workers, enhancing efficiency and productivity in industries such as e-commerce, retail, and logistics. Locus Vector is an innovative and adaptable AMR solution tailored for high-productivity material handling and logistics applications. It has dual safety-rated LIDAR sensors and a three-stage safety system, ensuring safe operation in shared work environments. The robot boasts a high payload capacity of up to 600 lbs, further enhancing its capabilities.
- **Robotnik:** Robotnik is a company specialized in the design and manufacturing of mobile robots for industrial and research applications. One of their notable products is the RB-1, an AMR specifically designed for indoor environments. The RB-1 features a robust and versatile platform that can be customized with different payloads and sensors according to specific application requirements. The robot's software includes a control system, a laser-based localization system, and a navigation system. However, it does not have fleet management, job prioritization, communication to Manufacturing Execution Systems (MES), or a mobile planner. The RB-1's navigation is fully supported by ROS.

2.4 Multi-Robot Systems

When considering mobile robots, one crucial aspect to consider is how to effectively manage a fleet of mobile robots. A fleet of mobile robots refers to a group of robots designed to perform certain collective behaviors that a single robot is incapable of achieving. There are several advantages to utilizing a multi-robot system compared to a single-robot system, including [18]:

- **Resolving complex tasks:** Multi-robot systems can handle tasks that are inherently distributed in nature or require diverse capabilities. By distributing the workload among multiple robots, complex tasks can be efficiently accomplished.
- **Enhanced performance:** The collaboration and parallelism inherent in multi-robot systems can significantly improve performance. Multiple robots working in conjunction can reduce task completion time, increasing overall efficiency.
- **Increased reliability:** Multi-robot systems offer inherent redundancy. If a single robot experiences a failure or malfunction, the presence of other robots can ensure that the task is still carried out successfully. This redundancy enhances the reliability and robustness of the system.
- **Simplified design and cost-effectiveness:** Utilizing smaller, simpler robots in a multi-robot system can lead to cost savings and easier implementation. The modular nature of the system allows for individual robot units to be less expensive, while the overall system can achieve high performance through coordination.
- **Scalability, flexibility, and adaptability:** Multi-robot systems are highly scalable since additional robots can be added or removed from the fleet as per the requirements of the task or the environment. This flexibility allows for adaptability to dynamic environments and changing task requirements more effectively than single robots. The collective decision-making and coordination capabilities enable them to adjust their behavior and optimize their performance in response to real-time changes.

One fundamental decision that needs to be made in multi-robot systems is whether to use centralized, decentralized control or some combination of both. In fully centralized control approaches, high performance and efficiency are expected. However, these approaches come with the risk of single-point failures and poor scalability due to limitations such as communication bottlenecks [19].

On the other hand, fully decentralized control approaches address the limitations of centralized control by leveraging redundancy and parallelization. They eliminate the risk of single-point failures and offer better scalability. However, they may have disadvantages such as lower speed and efficiency compared to centralized control [19].

Finding the right balance between centralized and decentralized control is crucial in achieving the desired performance, fault tolerance, and scalability in multi-robot systems. Hybrid approaches combining centralized and decentralized control can provide a solution that leverages the benefits of each approach while mitigating their drawbacks.

2.4.1 Centralized Control Architecture

In centralized control architectures, there is a central control agent that possesses the global information of the environment and the robots and can communicate with all the robots to share tasks

and coordinate their actions. This central control agent, which can be a computer or a dedicated robot, serves as the decision-making entity for the entire fleet [20]. Each robot communicates continuously with the central control agent, receiving task allocations and providing status updates [18].

The central control agent processes the information received from individual robots and generates appropriate commands to be sent back to each robot, ensuring coordinated execution of the assigned tasks. One of the main advantages of this architecture is the ability to have a global view of the world, enabling the central control agent to generate globally optimal plans and allocate resources efficiently [20]. Furthermore, the centralized approach reduces duplication of effort, optimizes resource utilization, and can lead to cost and time savings [18].

However, it is important to note that centralized control architectures have certain limitations. Firstly, they are typically more suitable for systems with a small number of robots [20], as managing a large fleet can lead to communication bottlenecks and increased computational complexity. Additionally, centralized control architectures are vulnerable to failures in communication, dynamic environments, or uncertainties, as the entire system relies on the central control agent [20]. A single point of failure can have a significant impact on the entire fleet's performance. Scalability is also a concern, as the system's performance may degrade as the number of robots increases, further exacerbating the bottleneck issue [18].

2.4.2 Decentralized Control Architecture

When considering the decentralized control architecture in multi-robot systems, two main categories can be identified [20]: distributed architectures and hierarchical architectures. In distributed architectures, there is no central control agent, and each robot in the fleet possesses the same level of control and autonomy in the decision-making process. On the other hand, hierarchical architectures represent a hybrid approach that combines elements of both centralized and distributed control. In this case, the robots are organized into clusters, and there are one or more local central control agents responsible for coordinating the activities within their respective clusters.

Decentralized control architectures offer unique advantages. In distributed architectures, one key advantage is the robustness of the system [18]. If one robot fails, the rest of the fleet can continue to operate effectively. Scalability is also a notable advantage, as there is no centralized control agent acting as a bottleneck, allowing for easy integration of new robots into the fleet. Additionally, decentralized control architectures provide flexibility and have lower communication demands compared to centralized approaches.

However, it is important to consider the potential drawbacks of decentralized control. One challenge is the possibility of sub-optimal solutions [18], as individual robots may make decisions based on local information that may not lead to the best global solution. Coordination and cooperation among the robots becomes critical to ensure overall efficiency and performance.

2.5 Fleet Managers

The main challenge to consider regarding mobile robot fleet managers on the market is that each manufacturer's fleet manager is typically designed to control their own robots. This approach ensures seamless integration and optimized performance for the specific robot fleet.

Some of the most well-known robot fleet managers are:

- Mobile Industrial Robots (MiR) has developed the **MiRFleet** fleet manager [21], designed to control MiR robots. It is a centralized control robot fleet manager that allows users to manage and coordinate their fleet of MiR robots. MiRFleet offers functionalities such as order handling by prioritizing and managing orders among multiple robots; traffic control, enabling the coordination of critical zones that consist of multiple robot intersections; and battery level control, which allows the user to monitor the robot's battery levels and automatically handle the recharging process. This fleet manager facilitates seamless communication and coordination between MiR robots, resulting in improved overall productivity.
- Otto Motors has developed the **Otto Fleet Manager** [22], designed to control and coordinate multiple AMRs within an industrial facility. This centralized control fleet manager offers various features, including: task assignment, enabling the assignment of specific tasks to individual robots or groups of robots, ensuring efficient use of resources; traffic control, ensuring smooth and safe interaction between the AMRs, avoiding collisions and congestion in busy areas; and job supervision, with the fleet manager continuously processing data about the fleet, providing information regarding each robot's status, such as charge level, location, and job status.
- Fetch Robotics has developed a unified control center for coordinating and monitoring their robot fleet, called **FetchCore** [23]. This fleet manager has several features, such as mapping, that allow the creation of detailed maps of facilities, enabling navigation and task planning for the robots. A real-time fleet status is also provided to continuously access robot information, including each robot's status and settings, ensuring up-to-date monitoring and control. Furthermore, it includes performance analytics that enhances productivity and efficiency, providing valuable insights into fleet performance and optimization opportunities.
- OMRON has developed the **OMRON Fleet Operations Workspace (FLOW)** [24]. This powerful tool is designed for managing and optimizing a fleet of OMRON mobile robots. FLOW features centralized control, facilitating intelligent job assignments and reducing wasted time and movement. It also includes managed motion, ensuring smooth operations in a dynamic working environment. Additionally, FLOW provides traffic control by notifying converging robots of their predicted paths, allowing them to calculate new paths efficiently and avoid collisions.

2.6 Communication Between a Fleet of Mobile Robots and the Fleet Manager

For the many mobile robot providers that exist, there is no single provider that can fulfill every task required in a specific context by a fleet of mobile robots. Therefore, there is a need to integrate different AGVs and/or AMRs from various vendors into a fleet. This type of heterogeneous fleet presents certain challenges, including [25]:

- **Individual integration effort:** Integrating each new robot from a different vendor requires individual effort and compatibility considerations. This can lead to increased complexity and time-consuming integration processes.
- **Traffic management:** In a heterogeneous fleet, where robots are sourced from different vendors, each robot may perceive the others as obstacles unless they have a common understanding of their respective trajectories. This lack of awareness can result in low operational efficiency and potential collisions.
- **Sharing the same environment:** Robots from different vendors sharing the same environment can be challenging due to differences in communication protocols, data formats, and control systems. Seamless coordination and collaboration between heterogeneous robots become more difficult to achieve.
- **Collaborative task execution:** Performing collaborative tasks with robots from different vendors may require additional coordination mechanisms and standardized interfaces. Achieving efficient collaboration and seamless task allocation across a heterogeneous fleet can be more complex and demanding.

With the increasing need for better communication and lower integration costs among robots provided by different vendors, interoperability standards have emerged. These standards aim to facilitate seamless integration, communication, and coordination among robots from diverse vendors, enabling efficient and effective operation of heterogeneous fleets [25].

2.6.1 Robotics Middleware Framework

To enable the coordination of fleets of robots and their integration with facility infrastructure, the Robotics Middleware Framework (RMF) was developed. RMF is a comprehensive framework comprising reusable open-source libraries and tools built on top of ROS 2 (Robot Operating System 2), which enables seamless interoperability among heterogeneous robotic systems [25]. With RMF, it is also possible to interface with various facility resources, such as doors, elevators, and passageways, enhancing the overall capabilities of the robot fleet.

RMF is designed to be flexible and robust, capable of operating on different communication layers. Its architecture facilitates scalability as the level of automation in a specific environment increases. Additionally, RMF offers cost savings by allowing resource sharing and minimizing integration efforts.

2.6.2 VDA 5050 Standard for AGVs

The VDA5050 is a standardized interface for AGV communication developed through collaboration between the German Association of the Automotive Industry (VDA) and the German Association for Materials Handling and Intralogistics (VDMA). This standard enables seamless communication between a fleet of AGVs and a master control system, facilitating the exchange of status updates and other relevant information.

One of the key features of the VDA5050 standard is the utilization of the Message Queuing Telemetry Transport (MQTT) protocol. MQTT employs a publish/subscribe architecture, allowing AGVs to interface with the master control system through MQTT brokers implemented by users. This protocol ensures efficient and reliable message transport, enabling timely and accurate communication between the AGVs and the master control system [25].

2.7 INESC TEC Navigation Stack and Fleet Manager

The INESC TEC Navigation Stack (INS) is a stable stack that was developed and tested throughout the years. This stack was developed on top of a ROS framework, supporting multiple traction modes, such as differential, tricycle, and omni-directional. The INS has several localization algorithms [26], such as Adaptive Monte Carlo Localization (AMCL), Perfect Match (PM) Localization, and Extended Kalman Filter (EKF) Beacons Localization, drivers that interface with the robot hardware, a path planner, a controller for parametric trajectories, and a map server. The INS supports multiple versions of Ubuntu, but in the context of this dissertation, it was installed the Ubuntu 18.04 LTS version, which is compatible with the version ROS melodic.

The INESC TEC robot fleet manager has centralized control, is graph-based, and is constituted by multiple ROS nodes, as depicted in Figure 2.6.

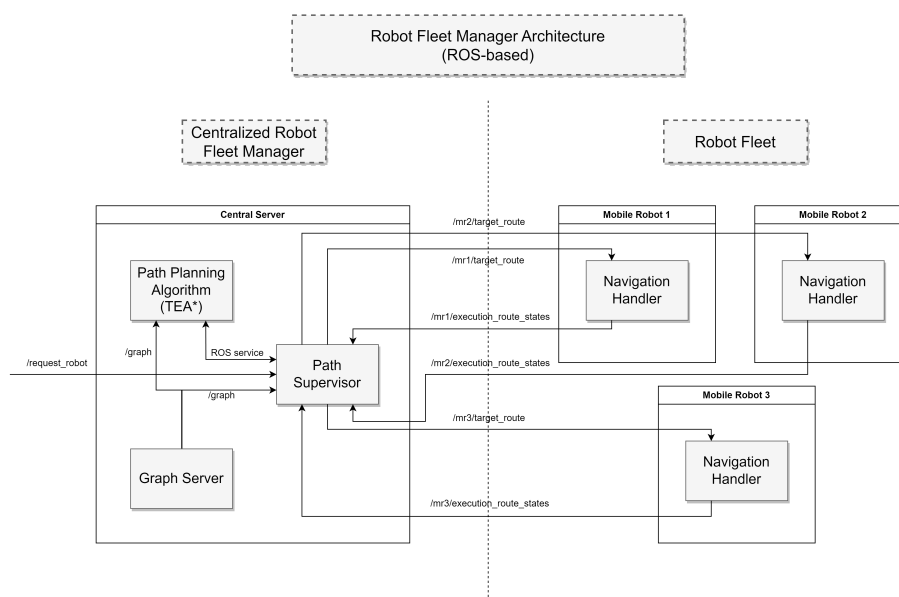


Figure 2.6: Architecture of the robot fleet manager.

The following briefly explains the responsibilities of each ROS node:

- **path_supervisor**: this node serves as the task manager. When a specific task requires a particular type of robot, the task manager searches through the available robots in the fleet with the desired type and calculates their distance to the designated point where the robot must navigate to execute the task. It then chooses the best-positioned robot to complete the intended task.
- **path_planner**: this node is responsible for planning the paths that robots must follow to execute a specific task. It uses the TEA* algorithm for path planning.
- **graph_server**: this node is responsible for sending the graphs and docking points to the nodes *path_planner* and *path_supervisor*.

Furthermore, the point of communication between each robot in the fleet and the fleet manager occurs through the *navigation_handler* node, as depicted in Figure 2.6. This specific robot fleet manager is used in the context of this dissertation.

Chapter 3

Overview of Implemented Technologies and Architectures

The main objectives of this dissertation were the development of a software module capable of controlling the OMRON LD-90 mobile robot and its integration with a ROS-based robot fleet manager. This chapter delves into the technologies and architectures adopted to achieve these goals, specifically, exploring the utilization of the Robot Operating System (ROS) and the Advanced Robotics Command Language (ARCL), a specific interface developed by OMRON.

In the forthcoming sections, an in-depth analysis of the communication mechanisms between system components and the overall software organization is provided.

3.1 ROS

This section discusses the Robot Operating System (ROS), including its definition, usage, and importance in project development.

ROS is an open-source framework that facilitates the development of robot systems. Despite its name, ROS is not a traditional operating system; rather, it consists of software libraries and tools that assist in building robot applications. It operates in conjunction with a conventional operating system. The primary objective was to establish a flexible and modular framework that promotes code, algorithm, and data sharing, fostering collaboration and expediting progress in robotics.

In addition to the core idea, the philosophy of ROS revolves around several key objectives:

- **Peer-to-peer communication:** ROS advocates a distributed architecture in which processes, known as nodes, engage in continuous communication through a publisher-subscriber messaging system. Without a central routing service, message exchange occurs directly between processes [27]. This approach enhances modularity and reusability, enabling independent development and testing of nodes that can be seamlessly integrated into larger systems.

- **Tool-based approach:** ROS offers a range of small, versatile tools for tasks such as debugging, visualization, and simulation. Instead of a monolithic runtime, ROS adopts a microkernel design, utilizing numerous small tools to construct and execute various ROS components [28].
- **Multilingual support:** Each process within ROS can be programmed in any language supported by ROS client libraries. High-performance tasks can be implemented in C++ or C, while other tasks can be implemented using Python or Java [29]. Furthermore, multiple languages can be combined through language-independent message processing, providing flexibility that is not readily available in other frameworks.
- **Thin:** ROS encourages the creation of separate libraries for each ROS component, driver, or algorithm with no ROS dependencies. These libraries are wrapped by a thin message-passing layer that allows them to make use of and be used by other ROS modules [27], resulting in seamless message exchange and facilitating greater reusability and simplification.
- **Free and open source:** ROS is an open-source framework released under a permissive license, allowing free access, modification, and distribution of its code and resources [27]. This fosters innovation, accelerates development, and promotes the widespread adoption of ROS.

3.1.1 ROS Filesystem Level

ROS can be referred to as a meta-operating system due to its OS-like features, in addition to its tools and libraries, such as hardware abstraction, package management, and a developer toolchain. Like a conventional operating system, ROS organizes its files in a specific way on the hard disk, as can be seen in Figure 3.1. So, the ROS filesystem refers to the organization and structure of files and directories within the ROS environment. It establishes a standardized way to store and access resources, including packages, source code, configuration files, data files, and more [29].

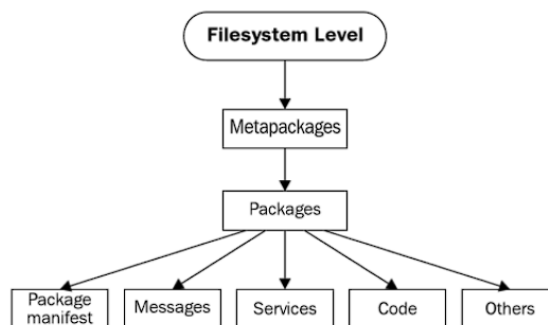


Figure 3.1: The ROS filesystem level. Reprinted from [4].

3.1.2 ROS Computation Graph Level

The computation graph level refers to how ROS organizes and manages the communication and interaction between different software components. At this level, ROS uses a distributed architecture that connects multiple software nodes to form a graph-like structure. The following diagram represented in Figure 3.2 shows how the ROS graph layer is structured.

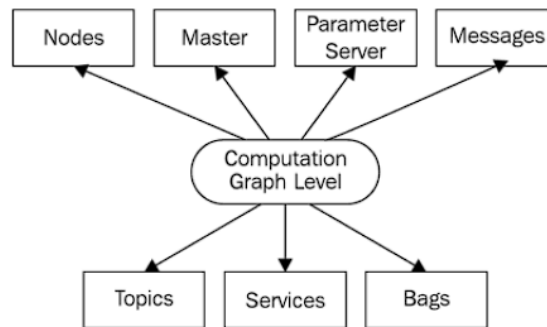


Figure 3.2: The ROS computation graph level. Reprinted from [4].

Nodes are independently compiled processes, also known as individual software modules, that perform computation. These nodes are written using ROS client libraries, which provide APIs for implementing various ROS functionalities, such as methods for communication between nodes. Rather than having a monolithic program, a typical ROS system consists of multiple ROS nodes, each designed to handle specific tasks or functionalities [29].

Communication between nodes is accomplished using messages. Messages are defined as simple data structures that contain field types supporting standard primitive data types and arrays of primitive data types [29]. Nodes exchange messages through topics, which implement a publish-subscribe communication mechanism. When a node wants to send a message, it publishes it to a specific topic. On the receiving end, a node interested in retrieving information of a particular data type subscribes to the relevant topic [28]. Multiple nodes can subscribe to the same topic or publish on the same topic concurrently [27], as it is possible to see in Figure 3.3.

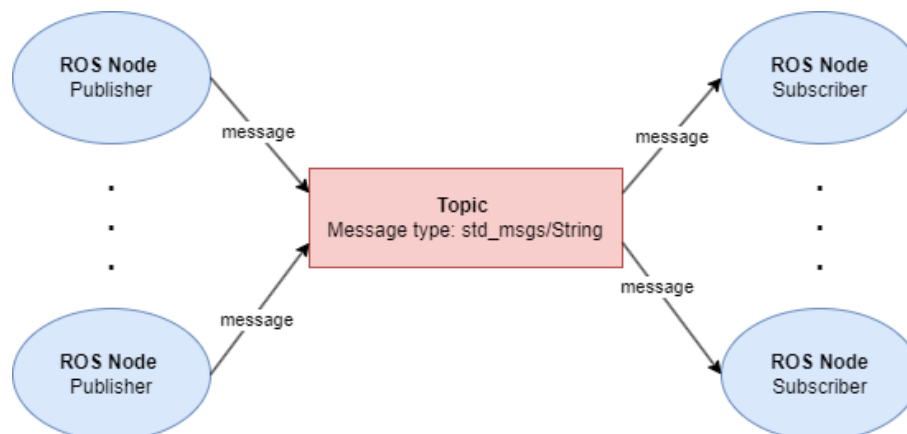


Figure 3.3: Illustration of the ROS topic publish/subscribe concept.

In the context of this dissertation, the representation of ROS nodes and ROS topics will be as depicted in Figure 3.3, with blue circles representing nodes and red rectangles representing topics.

In addition to the topic-based communication model, ROS supports synchronous interaction between nodes. This is achieved using ROS services, which enable nodes to call functions that are executed by other nodes [27]. Services are defined by a pair of messages that operate on a request/response mechanism, as shown in Figure 3.4. The request and response data types must be defined in a specific file. Unlike topics, only one node can advertise a service with a given name.

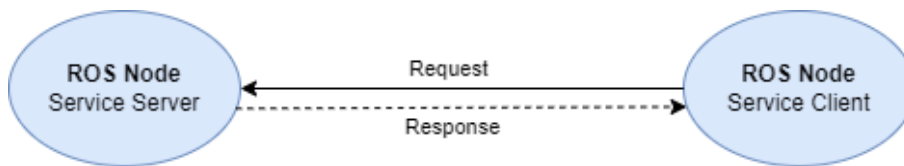


Figure 3.4: Illustration of the ROS service request/response mechanism.

3.1.2.1 ROS Master

The ROS master acts as a centralized coordination node for the communication between various nodes in a ROS system. It provides essential services to facilitate this coordination, including registration and naming services.

When a node initializes, it can register itself with the ROS master, providing information such as its name and the services and topics it offers or requires [29]. This registration allows the ROS master to maintain a record of active nodes in the system.

Regarding topics, nodes that wish to publish data on a specific topic must register this information with the ROS master. The ROS master then keeps track of publishers and subscribers for each topic by maintaining a list of active nodes interested in that topic.

When a node wants to subscribe to a topic, it sends a request to the ROS master, specifying the topic name of interest. The ROS master responds by providing a list of publishers for that topic. The subscriber node can then establish direct connections with the publishers using the provided information.

The command used to run the ROS master is as follows: `$ roscore`.

3.1.3 RViz

RViz (ROS Visualization) is a powerful visualization tool in ROS, capable of rendering three-dimensional (3-D) data including robot models, sensed images and point clouds. It is commonly used for visualizing robot states, sensor data, and planning information in a simulated or real robotic environment.

In addition to visualization, using *Marker* messages, it is also possible to interact with the user interface provided by RViz [30].

3.1.4 TF Coordinate Conversion System

The TF coordinate conversion system in ROS provides a framework for managing and transforming coordinate frames in a robotic system. It allows users to keep track of multiple reference coordinates under ROS over time.

A transformation in TF defines the relationship between two coordinate frames, representing how one frame is positioned and oriented relative to another frame. It consists of a translation vector and a rotation expressed as a quaternion or Euler angle. The implementation of TF is made on the topics */tf* and *tf_static*, through the publisher/subscriber mechanism. On top of that, it is possible to observe all coordinate frames and their relationships in a hierarchical structure called the TF tree, which represents the parent-child relationships between frames [31].

3.2 OMRON

3.2.1 ARCL

OMRON has developed the Advanced Robotics Command Language (ARCL), which serves as a communication interface for interacting with their mobile robots. ARCL is a straightforward, text-based operating language that enables users to send commands to the robot and receive responses from the robot [32]. To utilize ARCL, certain parameters in the OMRON MobilePlanner, a graphical user interface (GUI) for communication and configuration of the mobile robots, need to be accessed and modified.

To establish a connection with the ARCL server, a TELNET client is used. TELNET is a network protocol that provides bi-directional communication, handling eight-bit byte-oriented data transmission [33]. It relies on the Transmission Control Protocol (TCP), which enables the transmission of data along with interspersed TELNET control information [33]. TELNET operates within the TCP/IP (Transmission Control Protocol/Internet Protocol) protocol suite, which encompasses the set of protocols used for communication over the Internet.

Once connected to the ARCL server and upon successful login, the ARCL server provides a comprehensive list of supported commands, accompanied by corresponding descriptions. Each command can be executed, considering that certain commands require mandatory arguments to be sent to the server [32].

3.2.1.1 TCP/IP Sockets

There is the possibility of establishing a client/server connection to the ARCL server through a TCP/IP socket, and multiple connections are allowed [32]. A socket serves as a communication endpoint between two processes running in a network [34]. Applications can use sockets to send and receive data.

TCP/IP sockets can be categorized into two main types: stream sockets and datagram sockets. Stream sockets utilize TCP as the end-to-end protocol, layered on top of the Internet Protocol (IP),

providing a reliable byte-stream service [34]. Datagram sockets, on the other hand, use the User Datagram Protocol (UDP) for sending individual messages [34].

3.2.2 MobilePlanner

MobilePlanner is the software developed by OMRON to serve as a “control center” for their mobile robots [5]. The user interface of this software, represented in Figure 3.5, offers several functionalities. It allows users to create and edit map files, set goals and tasks for the robots, modify the configurations of the AMRs, and command individual AMRs for navigation and movement [5].

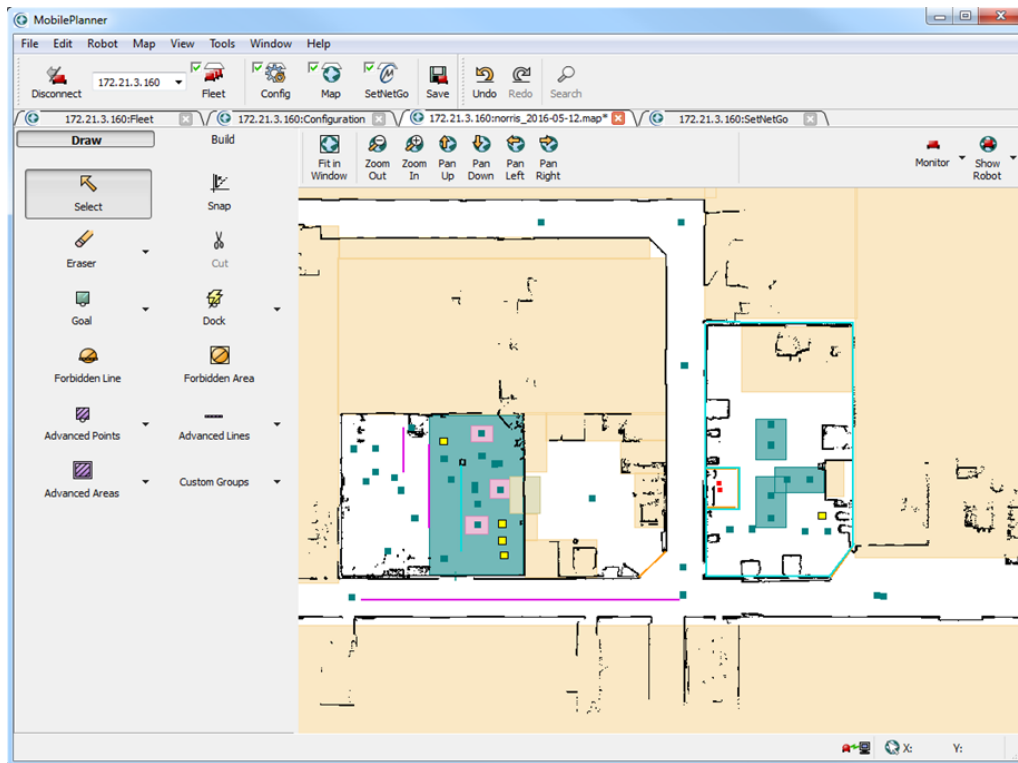


Figure 3.5: Mobile Planner Interface. Reprinted from [5].

One main feature of MobilePlanner is its map functionality, which represents a scanned representation of the floor plan in the mobile robot’s operating space. The map file contains crucial information used by the robot for navigation. It includes all points and lines scanned by the laser, as well as objects that influence the robot’s behavior, such as goal points, forbidden areas and lines, and docking stations. Additionally, the map file stores macros and tasks associated with goals [5]. The maps are stored in files with a *.map* extension.

3.3 Communication Framework

One crucial aspect to consider while developing a software module is how all the components that constitute the system communicate among themselves. The implemented system consisted of the

LD-90 mobile robot, a local machine running the created software modules, and a server where the INESC TEC robot fleet manager is installed. The interaction between the components can be seen in Figure 3.6.

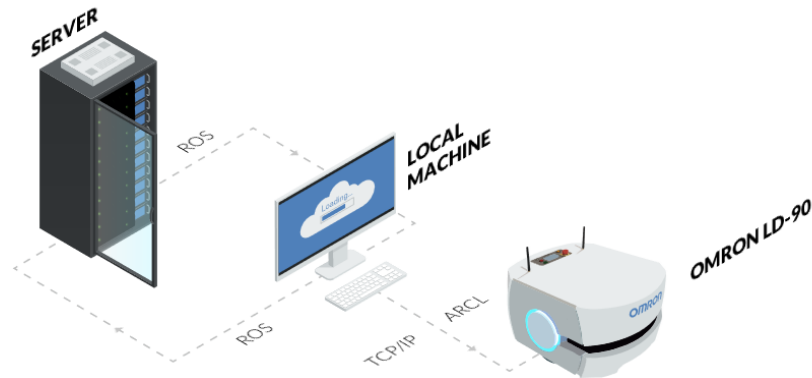


Figure 3.6: Proposed Communication Framework.

As mentioned in subsection 3.2.1, the communication with the LD-90 is established through the ARCL interface. Each LD-90 hosts an ARCL server, which remote clients can access through a TCP/IP connection. The local machine acts as the direct communication point between the system and the mobile robot, enabling users to send and retrieve information to and from the robot.

Furthermore, it was intended to integrate the LD-90 with the fleet manager. To achieve this, the local machine was used as a bridge for communication between the server and the mobile robot through ROS. Consequently, multiple ROS master nodes must be managed within the same network.

3.3.1 *multimaster_fkie* ROS Package

The *multimaster_fkie* package is a ROS package that provides a discovery mechanism for ROS master nodes to find and communicate with each other.

This package consists of two main nodes: the *master_discovery* node and the *master_sync* node. The *master_discovery* node periodically sends multicast messages to the common network, notifying other existing ROS masters of its presence and facilitating the detection of other available ROS masters. It also detects changes in the local network and notifies all other ROS masters in the common network of these changes [35].

On the other hand, the *master_sync* node utilizes the information from the *master_discovery* node to register remote topics and services to the local *roscore*, as well as update information on topics and services [35].

In this way, the *multimaster_fkie* package provides a solution for enabling ROS communication between the local machine and the server in the application design. It allows nodes from different master nodes to interact with each other as if they were part of a single ROS network. In Figure 3.7, it is possible to observe an illustration of the process for discovering and syncing used in the *multimaster_fkie* package.

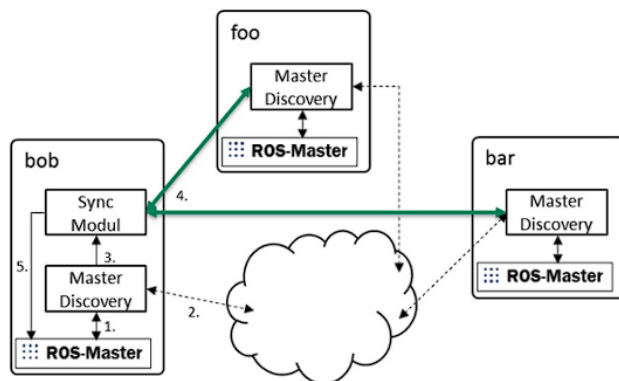


Figure 3.7: Illustration of the process of discovering and syncing. Reprinted from [6].

3.4 Software Architecture

The implementation of the application to control and manage the LD-90 mobile robot, while integrating it with the robot fleet manager, required the development of distinct ROS nodes and packages. Each node and package was designed to fulfill a unique role, contributing to the overall objectives of effectively controlling and managing the LD-90 robot.

In Figure 3.8, it is possible to observe the different ROS nodes and packages running in the local machine and their relation. Additionally, observing the communication point between the server and the local machine is possible.

The following provides a brief description of each node and package:

- ***omron_controller* node:** This node is responsible for establishing a connection to the ARCL server within the LD-90 mobile robot, enabling the sending of commands to the robot.
- ***omron_connection_receiver* node:** This node receives an incoming connection from the LD-90 mobile robot and handles all the information sent by the robot.
- ***omron_visualisation* package:** This package consists of three different nodes, each with different responsibilities for enabling real-time visualisation of the LD-90 mobile robot

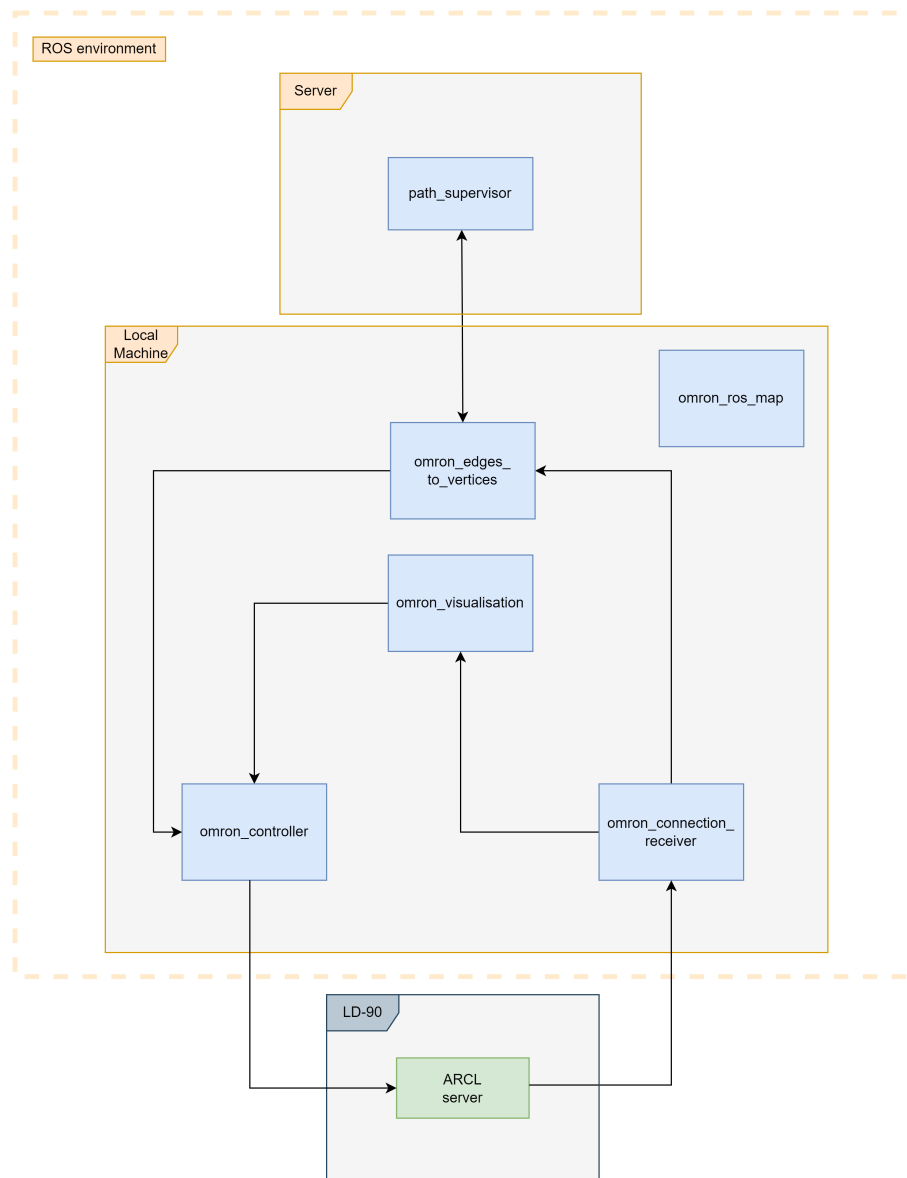


Figure 3.8: Proposed Software Architecture.

within a map, using RViz. Additionally, this node allows the sending of the robot to specific points through the RViz interface.

- ***omron_ros_map* node:** The fleet manager uses graph-based trajectories for robot navigation. This node handles the information of these trajectories to generate *.map* files, used by the LD-90 mobile robot.
- ***omron_edges_to_vertices* node:** This node is responsible for communicating with the robot fleet manager. It handles the information received from the fleet manager and sends information from the robot back to the fleet manager.

Furthermore, in Figure 3.8, it is also possible to observe the *path_supervisor* node that belongs to the robot fleet manager. This node is responsible for sending all the information related to path planning to the local machine. The information is then processed by the *omron_edges_to_vertices* node and sent to the LD-90 mobile robot.

Additionally, the *path_supervisor* node requires specific information from the robot, including its location in the graph, which explains the two-way arrow between this node and the *omron_edges_to_vertices* node.

3.5 Conclusion

The main objective of this chapter was to provide an in-depth analysis of the technologies used to develop the control and management of the OMRON LD-90 mobile robot, integrating it with the INESC TEC robot fleet manager. Additionally, the chosen architectures for the communication framework of the system and the organization of the implemented software were presented.

This chapter analysed the ROS framework, which facilitates the development of robot systems, and the ARCL interface, developed by OMRON to enable communication with their mobile robots. These technologies are fundamental for the proper functioning of the system, and multiple concepts from these technologies will be continuously referred to throughout the following chapters.

The system comprises the OMRON LD-90 mobile robot, a local machine, and a server with the robot fleet manager. The use of TCP/IP sockets enables seamless communication between the local machine and the LD-90. Additionally, communication between the local machine and the server is performed through ROS, using a specific ROS package that allows the management of multiple ROS master nodes within the same network.

The developed software modules that constitute the system were introduced, and their connections were presented. An in-depth analysis of these modules and their obtained results will be conducted in the subsequent chapters.

Chapter 4

Software Implementation

Related to the C++-based software implementation, corresponding to the management and control of the LD-90 mobile robot, the focus lies in developing and integrating various nodes and packages in ROS. The implementation phase is crucial, as it translates the previous theoretical concepts and algorithms into practical applications, effectively achieving the objectives.

The primary goal of this chapter is to provide a comprehensive overview of the ROS system created to facilitate the execution of complex robotic tasks. Each node and package will be discussed in detail, outlining their functionalities, design principles, and interconnections. By delving into the technical aspects of the implemented software, the aim is to demonstrate a single effective solution with practical viability.

4.1 Controller Node

The first ROS node implemented was the *omron_controller* node, responsible for direct communication with the LD-90 mobile robot. This was achieved by establishing a connection to the ARCL server on the robot and enabling the sending of commands to control the robot. In order to successfully connect with the ARCL server, specific configurations had to be made in the MobilePlanner software.

4.1.1 ARCL Configurations

The MobilePlanner, with its GUI, facilitates the modifications of the various configurations possible to do in the ARCL server. In the ARCL server setup section, multiple parameters can be seen under the Robot Interface tab, as shown in Figure 4.1.

The following parameters must be configured [32]:

- **OpenTextServer**: This parameter must be set to **True** in order to open the ARCL server.
- **PortNumber**: This parameter refers to the TCP port on which the ARCL server is opened. By default, the value is **7171**.

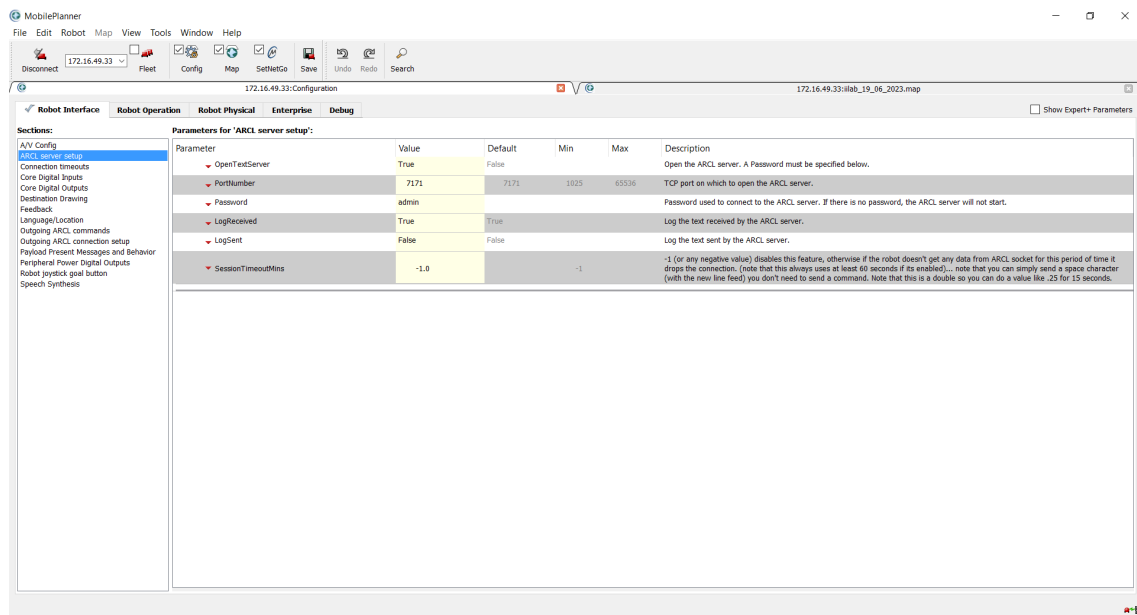


Figure 4.1: ARCL server setup parameter configuration.

- **Password:** This parameter refers to the password used to connect to the ARCL server. It is **required** to set a password; otherwise the ARCL server will not start.

By completing these configurations, it is possible enabling seamless communication between the *omron_controller* node and the LD-90 mobile robot, which will facilitate the execution of various ARCL commands.

4.1.2 Connection to the LD-90

As mentioned in subsection 3.2.1, it is possible to establish a connection to the ARCL server, on the LD-90 mobile robot, through the use of TCP/IP sockets. There are two steps to consider: socket creation and connection, since it is a remote client reaching a server.

A socket is created using the *socket()* function defined in the `<sys/socket.h>` header as:

```
int sockfd = socket(int domain, int type, int protocol);
```

It takes three arguments:

- **domain:** This parameter specifies the address domain requested in which a socket is created. It is set to `AF_INET` to indicate the use of IPv4 addresses.
- **type:** This parameter indicates the type of socket to be created. It is set to `SOCK_STREAM` since a TCP connection is intended.
- **protocol:** This parameter specifies a particular protocol to use within the socket. It is set to `0`, indicating that the default protocol for the specified address domain and socket type should be used, in this case, the IP protocol.

This function will return a non-negative integer that represents the socket file descriptor, if successful; otherwise, it will return a -1 value to indicate an error.

The connection procedure is performed using the `connect()` method, which is also defined in the `<sys/socket.h>` header:

```
connect(int socket, const struct sockaddr *address,
        socklen_t addresslen);
```

This method attempts to establish a connection on the socket and takes three arguments:

- **socket:** This parameter represents the socket file descriptor created using the `socket()` function.
- **address:** This parameter represents a pointer to a `sockaddr` structure, which contains the ARCL server's address, that refers to **the IP address of the robot on the network**, and the port number used to access the ARCL server, defined in subsection 4.1.1.
- **addresslen:** This parameter indicates the size of the `sockaddr` structure.

The `connect()` method returns 0 upon successful connection or -1 to indicate an error.

By establishing this connection to the ARCL server, it becomes possible to send commands to the server and receive response messages in return. The diagram illustrating how the connection is established can be seen in Figure 4.2, which also demonstrates the server side in a TCP/IP connection and the methods that must be executed to create the connection.

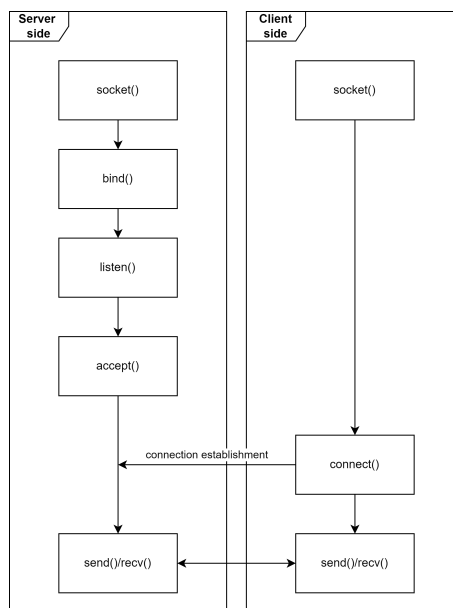


Figure 4.2: Client/Server socket connection diagram.

4.1.2.1 Logging into the ARCL server

As mentioned in subsection 4.1.1, a password is a required parameter to open an ARCL server. Therefore, when a remote client connects to an ARCL server, the server immediately requests the password. Subsequently, it is required that, before any commands are sent to the robot, a message containing the defined password has to be sent through the socket to the ARCL server.

4.1.3 Command Receiving in ROS

The *omron_controller* node subscribes to a single topic, */omron_ld90/send_command*, which receives messages of type *std_msgs::String*. The callback function implemented for each message received on this topic handles the message based on the command that it contains.

The ARCL reference guide introduces and explains the supported commands, their required arguments, and the ARCL server's response for each command. The following ARCL commands [32] were considered for sending instructions from the local machine to the ARCL server:

- **goto command:** This command sends the LD-90 mobile robot to a specific goal defined in the map loaded on the mobile robot. During the command's execution, the ARCL server continuously sends feedback messages until the LD-90 reaches the destination. The final message indicating the completion of the command is "Arrived at [goal name]".
- **getGoals command:** This command retrieves a list of goal names defined the loaded map loaded on the LD-90. The final message indicating the completion of the command is "End of goals".
- **dock command:** This command instructs the LD-90 to navigate to the docking station. The final message received from the ARCL server is "DockingState: Docked ForcedState: Unforced ChargeState: Overcharge".
- **undock command:** This command directs the LD-90 to move off of the dock station, positioning the robot in front of and facing the dock station. The final message received from the ARCL server is "Stopped".

Based on the command received through the */omron_ld90/send_command* topic, the node writes this command to the socket using the *send()* method defined in the *<sys/socket.h>* header.

Subsequently, after sending a command to the ARCL server, the feedback responses from the server can be verified. The completion of execution of each command can be ensured by checking for the presence of the final message associated with each command in the server's response messages.

As an example, Figure 4.3 illustrates the process of publishing a certain message on the */omron_ld90/send_command* topic:

In this example, a ROS node publishes a variable named *msg* of type *std_msgs::String* on the */omron_ld90/send_command* topic. The *msg* variable contains a member variable named *data* of type *string*. In the illustrated example, the value of *msg.data* is "goto 12".

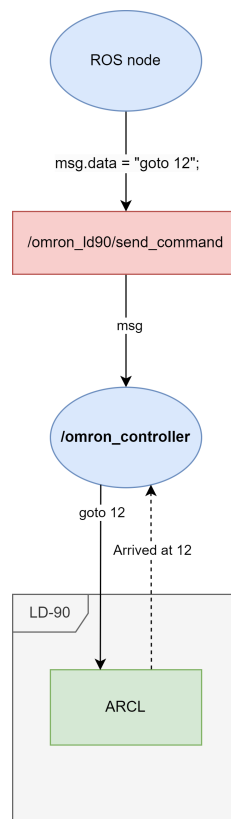


Figure 4.3: Example of the process of sending a command to the LD-90 through the *omron_controller* node.

The callback function for the */omron_ld90/send_command* topic, implemented in the *omron_controller* node, examines the content of the *msg.data* variable by comparing it with the list of considered ARCL commands. In this specific case, the command being sent to the ARCL server is "goto 12" and, as a result, the ARCL server produces response messages associated with this command. If the robot arrives at the destination goal, the line "Arrived at 12" is expected.

4.2 Connection Receiver Node

The *omron_connection_receiver* node plays a crucial role in the project's development as it provides the ability to receive information from the robot, through TCP/IP sockets, using a special connection available in MobilePlanner named "Outgoing ARCL connection".

Similar to the *omron_controller* node, specific configurations in the MobilePlanner software are required for the proper functioning of the *omron_connection_receiver* node.

4.2.1 ARCL Configurations

4.2.1.1 Outgoing ARCL Connection Setup

MobilePlanner allows for the configuration of an Outgoing ARCL connection, where the LD-90 mobile robot acts as the client and establishes a connection with a remote machine to send information. From the perspective of a TCP/IP socket connection, the robot assumes the role of the client and connects to a remote machine acting as a server.

Within the Outgoing ARCL connection setup section, under the Robot's Interface tab, as depicted in Figure 4.4, multiple parameters can be configured.

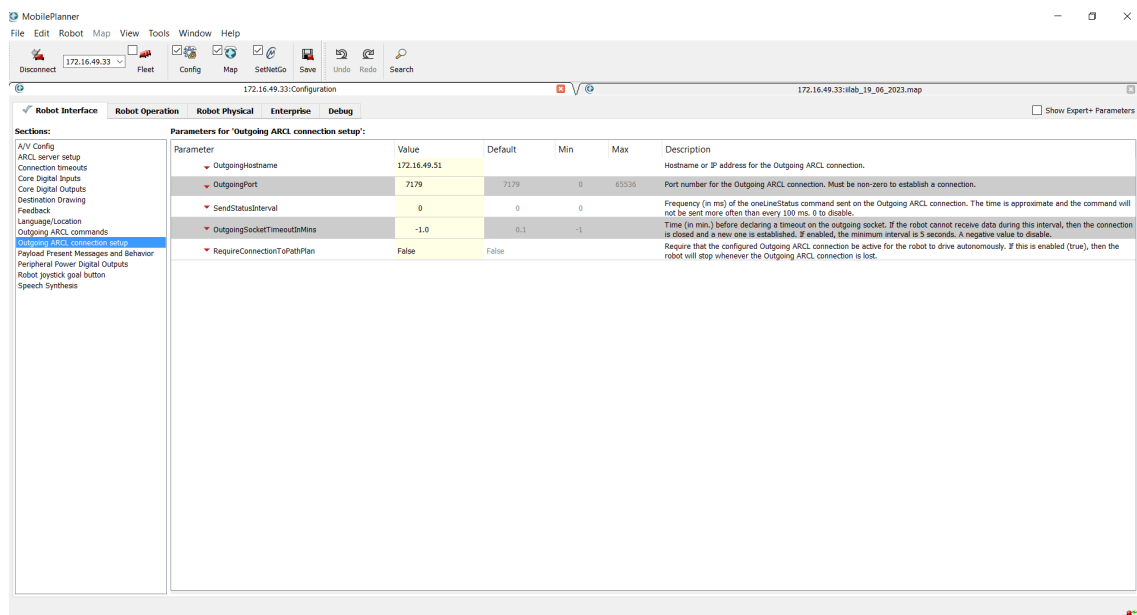


Figure 4.4: Outgoing ARCL connection setup configuration.

To establish an outgoing ARCL connection, the following parameters must be set [32]:

- **OutgoingHostname:** This parameter specifies the hostname or IP address with which the robot will attempt to establish the outgoing ARCL connection. The IP address is set to **the local machine's IP address within the network**.
- **Outgoing Port:** This parameter determines the port number for the outgoing ARCL connection, which is set to the default value of **7179**.
- **OutgoingSocketTimeoutInMins:** This parameter sets the duration in minutes that the LD-90 can remain without receiving data. If the robot does not receive any data beyond this specified time, the outgoing connection is closed. Since the robot may not receive data constantly and can experience prolonged periods of time without data, this parameter is set to **-1**, maintaining the outgoing connection open indefinitely until the socket is closed.

4.2.1.2 Outgoing ARCL Commands Configuration

MobilePlanner also provides the ability to configure the commands that are intended to be sent from the robot to the local machine, as well as the rate at which they should be sent [32]. In the Outgoing ARCL commands section, under the Robot's Interface tab, it is possible to add commands to be sent.

Figure 4.5 illustrates the defined commands to be sent and their corresponding sending intervals.

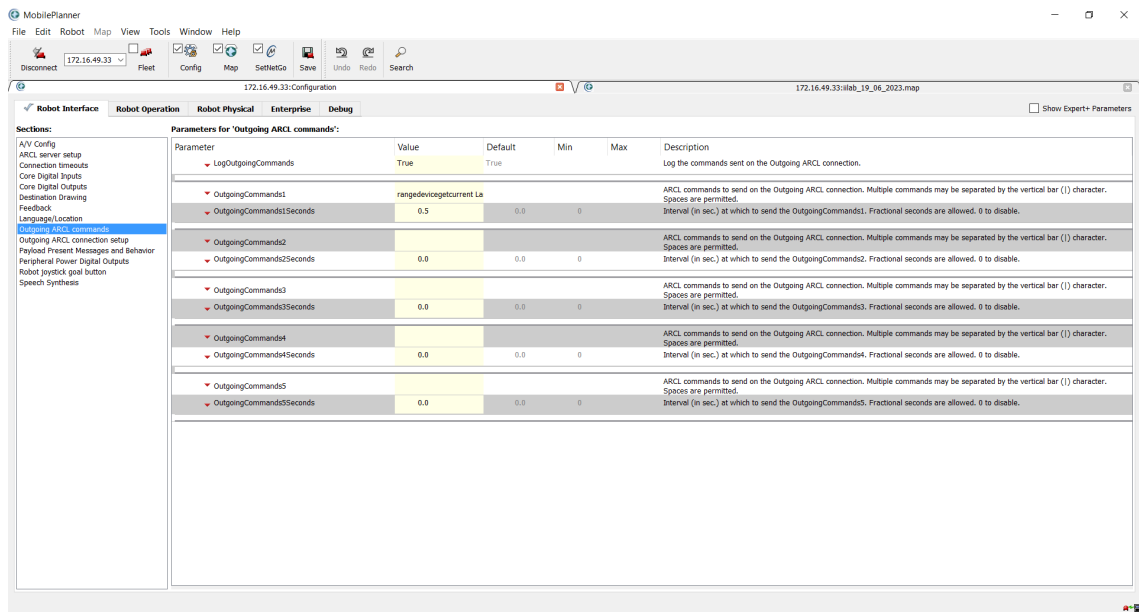


Figure 4.5: Outgoing ARCL commands configuration.

Two commands, namely *Status* and *RangeDeviceGetCurrent*, were used to send information from the robot to the local machine. These commands were added to the **OutgoingCommands1** parameter, separated by the vertical bar (|) character as: "rangedevicegetcurrent Laser1 | status". Additionally, these commands are being sent every half second, as specified in the **OutgoingCommands1Seconds** parameter.

Each command returns valuable information that is utilized in the development of the project. The *Status* command provides five different pieces of information [36]: the operational state of the LD-90, the battery charge level, the LD-90's location in the map as a pair of coordinates (X, Y) and an angle θ , the accuracy of the LD-90's location in the loaded map (reflecting how much the working environment has changed), and the operating temperature of the LD-90.

On the other hand, the *RangeDeviceGetCurrent* command retrieves a set of absolute (X, Y) map coordinates related to the active detection readings from the named ranging sensor [36]. In this case, the primary laser used by the LD-90 for mapping, referred to as *Laser_1*, was employed as the ranging device.

4.2.2 Receiving Data From the Robot

In order to receive data from the robot, the *omron_connection_receiver* node creates a TCP/IP socket and acts as a server. The robot, on the other hand, acts as the client, as mentioned in 4.2.1.1, and establishes a TCP/IP connection with the *omron_connection_receiver* node.

The creation of the server socket involves several steps, as depicted in Figure 4.2, including socket creation, binding the socket to a specific address and port, enabling the socket to listen for incoming connections, and accepting connections from clients. These steps allow the *omron_connection_receiver* node to receive data from the robot over the established TCP/IP connection.

The process of creating the socket follows the same steps as previously explained in subsection 4.1.2. The binding process involves associating a specific socket with a particular address and port. This is accomplished using the *bind()* method, which is defined in the `<sys/socket.h>` header as:

```
int bind(int socket, const struct sockaddr *address,
        socklen_t address_len);
```

The *bind()* method takes three arguments, which are similar to the ones used in the *connect()* method explained in subsection 4.1.2. The *address* variable contains the address and port, as defined in MobilePlanner, to which the socket should be bound.

For the listening phase, the server needs to listen for incoming connections on a socket. This can be achieved using the *listen()* method, which is also defined in the `<sys/socket.h>` header. The *listen()* method allows the socket to listen for incoming connections and specifies the maximum length of the connection queue. It is defined as follows:

```
int listen(int socket, int backlog);
```

The *listen()* method takes two arguments: the file descriptor of the created socket and the maximum length of the connection queue.

Finally, the accepting phase involves accepting incoming connections on the socket. This can be achieved using the *accept()* method, also in the `<sys/socket.h>` header, defined as:

```
int accept(int socket, struct sockaddr *address,
          socklen_t *address_len);
```

The *accept()* method extracts the first connection request from the queue of pending connections for the listening socket specified by the *socket* argument. It then creates a new socket and returns the file descriptor associated with that socket. From this point onward, the connection between the client and the server is established, allowing for the exchange of messages between them.

4.2.3 Data Processing and Handling

From the moment the *omron_connection_receiver* node is initialized and the robot establishes a connection to the server side of the TCP/IP socket, the robot begins transmitting data, which is then received on the socket. This is accomplished using the *recv()* method, which is also defined in the `<sys/socket.h>` header. The received information is a single string, as both the return of the *Status* command and the *RangeDeviceGetCurrent* are sent together at the rate defined in 4.2.1.2.

Subsequently, all the received and stored data was published on multiple topics, allowing other nodes to retrieve the desired information. However, a challenge arose as the received data arrived as a single string. Therefore, it was necessary to split this string into smaller strings, each corresponding to a specific type of information. Consequently, every time a message was received on the socket, the string containing the data was divided into six distinct strings, representing the following information: status, state of charge, location, localization score, temperature, and laser scan points.

With the data divided into multiple strings, it becomes possible to publish each type of information on different topics. The process of the *omron_connection_receiver* node is depicted in Figure 4.6.

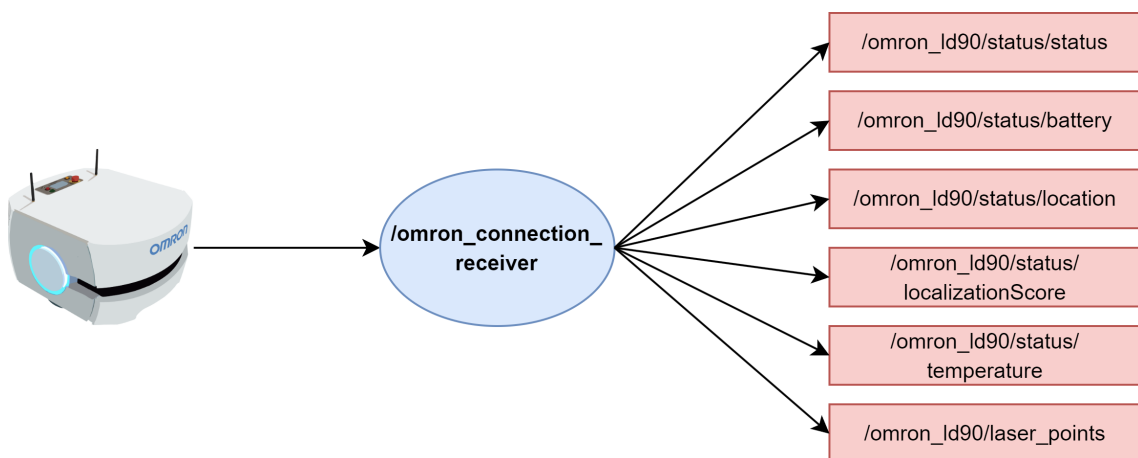


Figure 4.6: Diagram of the *omron_connection_receiver* node.

The *omron_connection_receiver* node receives data from the robot, which is processed by the method developed in the node responsible for creating individual strings for each type of information. These strings are then published on the respective topics, represented by the red rectangles in Figure 4.6.

By developing the *omron_connection_receiver* and the *omron_controller* node, as introduced in section 4.1, two points of direct communication between the application and the robot were established. These nodes enable the application to control the LD-90 robot and retrieve information from it.

4.3 Visualisation Package

This section focuses on the implementation of a ROS package that encompasses three distinct nodes, each serving a different purpose. The package is designed to recreate the basic functionalities of the MobilePlanner interface, such as the visualisation of a loaded map, a visual representation of the robot's position on the map, and the functionality of sending the robot to specific goals within the map by clicking on points within the RViz interface. By integrating these nodes, the package offers a comprehensive solution for map visualisation, robot tracking, and direct interaction with the LD-90 mobile robot in a ROS environment.

The following presents the nodes comprising this package and provides a brief description for each:

- **map_visualisation:** This node recreates the map loaded on the robot in RViz. It utilizes the information available in the *.map* file, generated by MobilePlanner, to display the map.
- **robot_state_publisher:** This node is responsible for tracking the robot's position and displaying it in RViz.
- **goal_sender:** This node allows the user to send the LD-90 robot to a specific goal within the map using specific the */clicked_point* topic provided by RViz.

4.3.1 Map Visualisation Node

The MobilePlanner software, as mentioned in subsection 3.2.2, offers a user interface primarily designed for representing scanned representations of the robot's working space as maps. These maps can be edited, and various components can be added to them.

The following introduces some of the components that can be added to maps using MobilePlanner [5]:

- **Goals:** These points on the map specify destinations to where the LD-90 robot can be sent. Additionally, the desired orientation for the robot at the goal location can also be defined.
- **Forbidden Areas:** Since the LD-90 robot's scanning laser is situated about 200 millimeters above the ground, there may be obstacles that the robot cannot detect. To prevent collision situations, it is possible to define areas on the map known as "forbidden areas" that block the robot's navigation within those regions.
- **Dock:** This component designates the point on the map to where the robot travels in order to connect itself to its charging station. The dock point should be positioned close to the charging station, between 1 and 1.5 meters, and its orientation must face the charging station.

In Figure 4.7, a map of a working environment in MobilePlanner is depicted. The map was scanned with LD-90 mobile robot. All the points and lines scanned by the robot's laser and the

previously mentioned components are illustrated in the map, which the red boxes and corresponding red numbers can identify. In this case, a goal is indicated by the number "1", a forbidden area by the number "2", and a dock by the number "3". If a goal has a small line, it indicates that it has associated to it a desired orientation for when the robot reaches the goal.

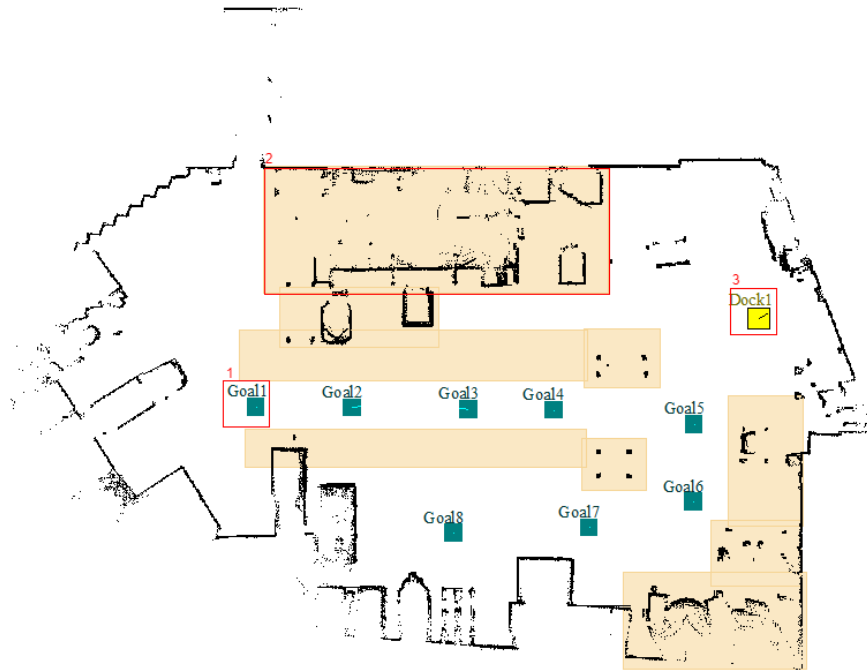


Figure 4.7: Representation of a map in MobilePlanner.

MobilePlanner generates *.map* files that contain comprehensive information about the map. This includes details about the scanned points and lines, as well as information about the components utilized in the map.

The objective of the *map_visualisation* node is to represent the map and its content within a ROS environment, specifically using RViz. To achieve this, the node utilizes the information present in the *.map* file and recreates the visual elements displayed by MobilePlanner in RViz.

4.3.1.1 Information Stored in a *.map* file

To accurately recreate the map in RViz, it is crucial to understand the information contained within the *.map* file and its corresponding representations. The *.map* file stores four different types of data related to the working environment in which the robot operates [5]:

- The points and lines scanned by the robot's laser of the working environment.
- Components inserted in the map using the MobilePlanner software, such as goal points, forbidden areas, and dock points.

- Functionalities associated with individual goals.
- Data specifying special goal types properties.

For the *map_visualisation* node, the relevant information is associated with the scanned points and the components inserted in the map. It is important to familiarize oneself with the parameters required by each component, as defined in MobilePlanner. For example, a goal component has five parameters [5]:

- **Name:** Indicates the name of the goal.
- **Description:** An optional parameter that provides a description of the goal.
- **Type:** Specifies that the component type is a goal.
- **Position:** Refers to the coordinates of the goal, with separate X and Y coordinate parameters.
- **Heading:** An optional parameter that specifies the desired orientation, in degrees, when the robot reaches the goal. If this parameter is used, the goal type changes to *GoalWithHeading*.

Each goal defined in a map using MobilePlanner corresponds to an individual line in the generated *.map* file, containing all five parameters. Similarly, the dock component also has the same five parameters as the goal component, with the *type* field set to *DockLynx*. Additionally, the *Heading* parameter is required for the dock component, as a dock must face the charging station.

On the other hand, the forbidden area component does not have the *Position* and *Heading* fields like the goal component. Instead, it has the following parameters in addition to the ones shared with the goal component [5]:

- **Angle:** Represents the rotation angle, in degrees, of the forbidden area within the map.
- **Corner:** Refers to the X and Y coordinates of the starting corner of the forbidden area.
- **Opposite:** Indicates the X and Y coordinates of the ending corner of the forbidden area.

These parameters define the geometry and position of the forbidden area component within the map. For every forbidden area defined in a map using MobilePlanner, there is a corresponding individual line in the generated *.map* file containing the respective parameters.

Additionally, all the points and lines scanned by the robot's laser of the working environment are stored in the *.map* file, with the map points being represented as pairs of (X, Y) coordinates and the map lines by two pairs of (X, Y) coordinates: $(X1, Y1); (X2, Y2)$.

4.3.1.2 Map Representation in RViz

As mentioned earlier, the *map_visualisation* node is responsible for recreating a map generated using the MobilePlanner software within a ROS environment using RViz. To achieve this, the node utilizes the relevant information from the corresponding *.map* file by extracting the required data. The extraction process involves utilizing file handling methods and consists of the following steps: opening the *.map* file for reading and iterating through the file line by line. During the iteration, each line is checked to see if it contains specifications that match the parameters of the components or the points and lines. If a particular line of the file does contain those parameters, the information within it is stored for further processing.

Furthermore, it is important to understand how to store the extracted data so it can be used by RViz. ROS provides a message type called *visualization_msgs/Marker*, which is specifically designed for visualizing various primitive shapes in RViz. The *Marker* type allows for the definition and specification of characteristics, such as shape, color, position, orientation, and scale. Commonly used *Marker* types include points, lines, spheres, cubes, cylinders, and text.

The *Marker* message comprises several key fields:

Field	Type	Description
header	std_msgs/Header	Contains information about the ROS message, including the timestamp and coordinate frame.
ns	string	Provides a way to group related markers together using a namespace.
id	int32	Assigns a unique identifier to each marker within its namespace.
type	int32	Specifies the shape or type of the marker. Common values include: - ARROW (0) - CUBE (1) - LINE_LIST (5) - POINTS (8) - ...
action	int32	Describes the action to be performed on the marker. Common values include: - ADD (0) - MODIFY (1) - DELETE (2)
pose	geometry_msgs/Pose	Represents the position and orientation of the marker in 3D space.
points	geometry_msgs/Point[]	Represents the points of the marker.
scale	geometry_msgs/Vector3	Specifies the size or scale of the marker.
color	std_msgs/ColorRGBA	Defines the color of the marker using RGBA values.

Table 4.1: Key Parameters of the *visualization_msgs/Marker* Message

The *map_visualisation* node utilizes the *Marker* message type to visualize the different components defined in the map within RViz. Multiple instances of this message type are created to

represent the various map components.

For example, one instance is specifically designed to display all the goals of the map. This is achieved by using the *points* field within the *Marker* message type, which represents a vector of the *geometry_msgs/Point* message type. The coordinates of each goal in the map are stored as *geometry_msgs/Point* objects in this vector associated with the respective *Marker* instance.

Each *Point* message contains three fields: (x, y, z) . Therefore, to associate a goal with a *Point* message, the (X, Y) coordinates of the goal are assigned to the x and y fields, respectively. Since the z coordinate is not relevant in this context, it is set to 0. For every goal in a map, a *Point* message with the goal's coordinate is added to the *points* vector of the *Marker*. The *type* field for this *Marker* is set to "Points".

Similar approaches were followed for the remaining map components, with each component represented as a separate *Marker*, using different types of the one used for the goals. In addition, *Markers* were created for the text associated with the name of each goal and for an arrow representing the heading (in cases where a goal has a specified heading). These additional *Markers* were also visualized in RViz.

It is important to consider the *header* field in the *Marker* message since it contains information about the coordinate frame associated with the data. It is essential that all the *Marker* messages are defined within the same frame to ensure proper visualization.

After creating the *Marker* messages to represent the various map components, each marker is published to a separate topic. Through RViz, it is possible to subscribe to these individual topics to render the *Markers*.

4.3.1.3 Laser Scan Points in RViz

The *map_visualisation* node is also responsible for displaying the points scanned by the robot's laser on the map. Therefore, the node subscribes to the */omron_ld90/laser_points* topic. As mentioned in subsection 4.2.3, the *omron_connection_receiver* node is responsible for publishing data on this topic.

When new laser scan points are received on this topic, the callback method associated with it in the *map_visualisation* node is triggered. The process for displaying the laser scan points is similar to that used for the other components. A new *Marker* instance is created to represent these points, and it is published on a specific topic for visualisation in RViz.

4.3.1.4 Creation of an Occupancy Grid

One of the objectives of this dissertation was to integrate the LD-90 mobile robot with the INESC TEC robot fleet manager. To enable interoperability with other robots in the fleet, it is necessary for the map generated by MobilePlanner to be compatible with their mapping systems. However, MobilePlanner produces a specific file format that is not compatible with robots from other manufacturers, which typically use the standard *.png* file format for map representation.

To address this compatibility issue, the *map_server* ROS package is employed [37]. This package provides utilities for converting maps to image and metadata files. The *map_saver* node within the package facilitates the conversion of a message of type *nav_msgs/OccupancyGrid* to a *.png* file. The *OccupancyGrid* message represents the map in the form of a 2D grid with occupancy probabilities.

The creation of the *OccupancyGrid* message was based on the map points extracted from the *.map* file. This process involved the following steps:

- Determining the size and resolution of the occupancy grid. This information was obtained from the *.map* file, which includes the map's resolution as well as the maximum and minimum points. These values were used to calculate the size of the occupancy grid.
- Creating an empty occupancy grid data structure.
- Converting each map point (X, Y) to grid cells indices while ensuring that the indices are valid.
- Updating the corresponding cell in the occupancy grid to represent the occupied state.

These steps enabled the creation of the *OccupancyGrid* message, which is published by the *map_visualisation* node on a specific topic. By subscribing to this topic, other robots can utilize the *map_saver* node to convert the map from its original *.map* file format to a *.png* file format, as represented in Figure 4.8.

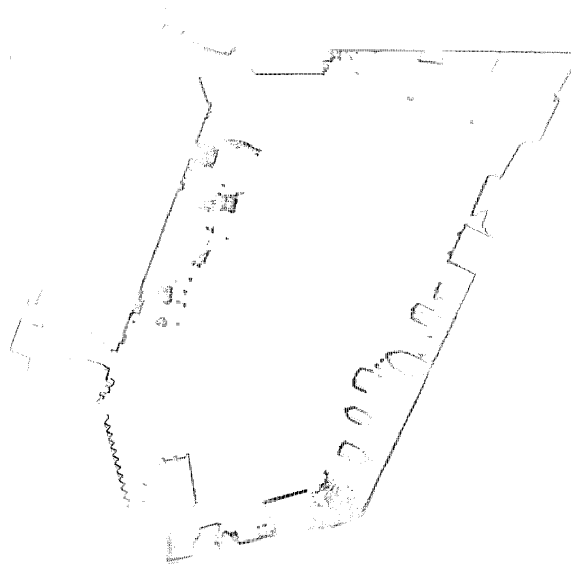


Figure 4.8: Occupancy Grid of the Workspace.

4.3.1.5 Structure of the *map_visualisation* node

For a better understanding of the processes that occur in the *map_visualisation* node, refer to the diagram in Figure 4.9.



Figure 4.9: Diagram of the *data_marker* node.

The *map_visualisation* node utilizes the information from the *.map* file, and publishes it on specific topics that RViz subscribes to, enabling the representation of the map in RViz. Additionally, it is important to note the relationship between the *map_visualisation* node and the *omron_connection_receiver* node, through the */omron_ld90/laser_points* topic, where the data containing the laser scan points is published.

4.3.2 Robot State Display Node

The *robot_state_publisher* node is responsible for the visualisation of the robot's position within the map displayed in RViz. It accomplishes this by subscribing to the */omron_ld90/location* topic, which contains information about the robot's (X, Y) coordinates and its orientation angle θ . As mentioned in 4.2.3, the *omron_connection_receiver* node is responsible for publishing this information on the topic.

Whenever a new message is published on the */omron_ld90/location* topic, a callback method implemented in the *robot_state_publisher* node is triggered. This method converts the robot's coordinates and angle into a *geometry_msgs/PoseStamped* message type. This message type is suitable because it consists of two main fields: a *geometry_msgs/Pose* object, which allows the

specification of coordinates and orientation, and a *Header*, which ensures that the robot's position is represented in the same frame as the map created by the */map_visualisation* node.

The orientation in the *Pose* object is represented by a *geometry_msgs/Quaternion* message type. This means that the orientation angle θ , obtained from the */omron_ld90/location* topic, needs to be converted. To achieve this, a method can be used to directly convert the yaw angle around the Z-axis to a *Quaternion* message. This conversion method is available in the *tf* library of ROS.

After creating the *PoseStamped* message, the *robot_state_publisher* node publishes it to a specific topic that RViz subscribes to. This allows for the continuous visualisation of the robot's position in RViz. Figure 4.10 illustrates the described process of the *robot_state_publisher* node, where it is possible to observe the interaction between this node and the *omron_connection_receiver* node, through the */omron_ld90/status/location* topic.

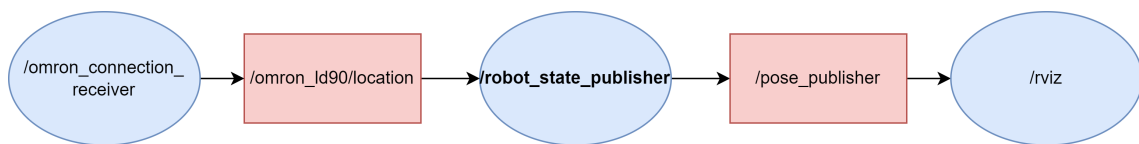


Figure 4.10: Diagram of the *robot_state_publisher* node.

4.3.3 Goal Sender Node

The main purpose of the *goal_sender* node is to enable control of the LD-90 robot through the RViz interface, similar to the functionality provided by the MobilePlanner interface. In MobilePlanner, users can send the robot to specific goals or the dock by double-clicking on their corresponding location. The *goal_sender* node aims to provide similar functionality within the ROS environment using RViz.

To achieve this, the *goal_sender* node subscribes to the specific RViz topic */clicked_point*. When a user clicks on a point in the RViz interface, RViz captures the 3D coordinates of that point and publishes them as a message on the */clicked_point* topic.

Furthermore, the *goal_sender* node utilizes the information from the *.map* file to extract the goal coordinates, following the procedure described in 4.3.1.2. The callback method implemented in the *goal_sender* node is triggered upon message publishing in the */clicked_point* topic. This callback method creates a filter by comparing the coordinates of the clicked point with the coordinates of every goal plus a small tolerance. If the clicked point falls within the range of any goal coordinates plus the small tolerance, the corresponding goal is identified. A message is published on the */omron_ld90/send_command* topic with the command to send the robot that specific goal, which is handled by the *omron_controller* node. The sequence of these steps is illustrated in Figure 4.11.

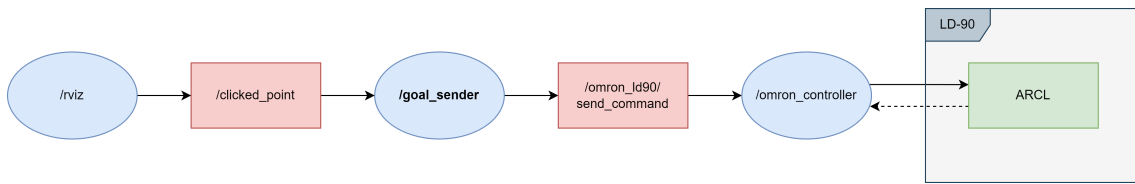


Figure 4.11: Diagram of the *goal_sender* node.

4.4 Trajectory Converter Node

In addition to controlling the LD-90 mobile robot, another primary objective of this dissertation was to integrate the LD-90 into the robot fleet manager. The fleet manager utilizes graph structures to create trajectories for robot navigation. Therefore, it is necessary to reproduce these types of trajectories in a *.map* file that the LD-90 mobile robot can use.

As mentioned in subsection 2.2.2, graph structures are represented as a collection of vertices and edges. However, since the LD-90 is a free-path navigation robot that employs free path planning algorithms, this graph-based approach is not directly applicable to the robot. Instead, an approximation was made.

The *omron_ros_map* node manages the trajectory data, which includes information about the edges and vertices, to generate goals in a *.map* file that can be loaded into the LD-90. When a trajectory is created, a *.yaml* file is generated with the following information:

Edges :

- {CurveType: spline, Destination_ID: 2, Id: 3, Origin_ID: 1, ParamB: 0.25, ParamF: 0.25, VelocityBackwards: 0.3, VelocityForward: 0.3}
- {CurveType: spline, Destination_ID: 4, Id: 5, Origin_ID: 2, ParamB: 0.89, ParamF: 1.04, VelocityBackwards: 0.3, VelocityForward: 0.3}

Vertices :

- {FrameId: map/nn0, Id: 8, Label: pass, Theta: 1.64, ThetaHolonomic: 2.74, X: 4.17, Y: -3.86}
- {FrameId: map/nn0, Id: 10, Label: pass, Theta: 2.79, ThetaHolonomic: 2.74, X: 5.93, Y: -2.99}

The *.yaml* file includes information such as the origin/destination vertices that form an edge, the *Id* of each vertex, the (X, Y) coordinates of each vertex, and the angle *Theta* corresponding to the orientation of the vertex in the reference frame. The *omron_ros_map* node utilizes this information to generate goals in a *.map* file, using the same coordinates as the vertices in the trajectory.

Using file handling methods, this node opens a *.map* file and defines all the vertices in a trajectory as goal components. As mentioned in 4.3.1.1, each goal component has five parameters: *name*, *description*, *type*, *position*, and *heading*. However, the *description* parameter is optional, and the *type* parameter simply specifies that the component is a goal. Therefore, the *omron_ros_map* node only needs to consider the other three parameters.

The goal's *name* and *position* are associated with a vertex's *Id* and (X,Y) coordinates, respectively. As for the goal's *heading*, there are additional considerations to take into account. The *heading* parameter is used only for particular vertices. These vertices can be divided into dock points, charge points, and logistics points. The specificity of these types of vertices is that they are only associated with one edge, implying that when a robot reaches that vertex, the only possible path is to return along the edge that led to that vertex but in the opposite direction. Given that the LD-90 robot is unidirectional, it must perform a turn-around maneuver at a certain point to execute this action.

The approach adopted considers assigning a *heading* to every goal associated with these particular vertices so that when a robot reaches the goal, it will perform the required rotation to orient itself towards the edge it must go through. Thus, when the robot receives another task, it does not waste time with this initial rotation, avoiding delays in the fleet manager.

Figure 4.12 illustrates a simple trajectory that contains some vertices that can be one of the particular vertices mentioned above.

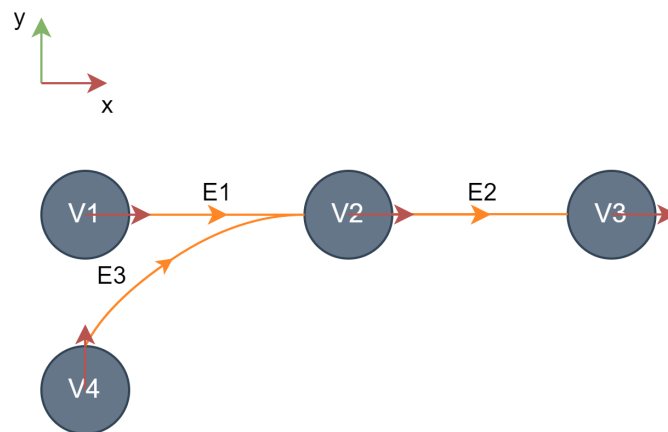


Figure 4.12: Illustration of a trajectory with end-point vertices.

In this trajectory, the particular vertices are V1, V3, and V4. Each of these vertices is associated with only one edge, unlike vertex V2, which is associated with three edges. Since these particular vertices are associated with only one edge, it means that in the edge list defined in the *.yaml* file, each particular vertex will only appear associated with one edge, meaning that the vertex is either the origin or the destination of that edge. In the specific case of the trajectory in Figure 4.12, vertices V1 and V4 are the origin vertices of their corresponding edges, and vertex V3 is the destination of its corresponding edge.

Vertices have an orientation in relation to the reference frame, corresponding to the angle θ seen in the *.yaml* file for every vertex. This orientation points to the orientation of the edge, as represented in Figure 4.12 by the red arrows. As mentioned previously, the approach taken considers that the final orientation of the robot must point to the edge that it must go through. To achieve this, the value assigned to the *heading* of a goal depends on whether the particular vertex is the origin or the destination of its corresponding edge. For the cases where the vertex is an origin, the *heading* corresponds to the angle θ of the vertex, and for the cases where the vertex is a destination, the *heading* corresponds to the angle θ of the vertex plus 180 degrees.

Furthermore, the *omron_ros_map* node verifies each *Id* of every vertex in the list of vertices in the *.yaml* file. It looks for a predefined vertex *Id* corresponding to the dock component to be defined in the *.map* file. The definition of this component in a *.map* file, as mentioned in 4.3.1.1, contains the same five parameters as the goal component, with the change that the *type* parameter now refers to *DockLynx*. The assignment of values to each parameter follows the same logic used for the goal components, considering that the *heading* parameter of the dock component must make the mobile robot face the charging station.

By accurately defining the list of vertices present in the *.yaml* files as goal or dock components in a *.map* file, it is possible to replicate the vertices of a graph-based trajectory on the LD-90 mobile robot.

4.5 Fleet Manager Communication Node

As explained in Section 3.4, the communication between the server containing the robot fleet manager and the local machine containing the developed ROS environment occurs through the *path_supervisor* node and the *omron_edges_to_path* node. These nodes exchange information to enable the LD-90 mobile robot to execute a specific path in a trajectory.

4.5.1 Path Messages from Fleet Manager

Firstly, it is important to understand where the fleet manager publishes the messages containing the path it wants the robot to travel and what information these messages contain.

The robot fleet manager uses the topic */omron_ld90/target_route* to publish the message containing information about the path it wants the robot to navigate. To generate this path, the fleet manager uses the TEA* algorithm to find the optimal path for the robot to reach its destination. The message sent to the topic contains the sequence of edges that the mobile robot has to traverse to get to a specific vertex.

The *omron_edges_to_vertices* node subscribes to the */omron_ld90/target_route* topic. Therefore, when the robot fleet manager publishes a message on this topic, a callback function implemented in the node is triggered to handle the sequence of edges contained in the message. However, it is worth noting that the concept of edges is not directly applicable to the LD-90 mobile robot. As a result, the callback function iterates through the received sequence of edges and

checks the origin and destination vertices of each edge in the sequence. This process allows the creation of a sequence of vertices associated with goal components that the LD-90 can be sent to.

Figure 4.13 illustrates an example of a trajectory and the publishing of a message containing a sequence of edges by the *path_supervisor* node.

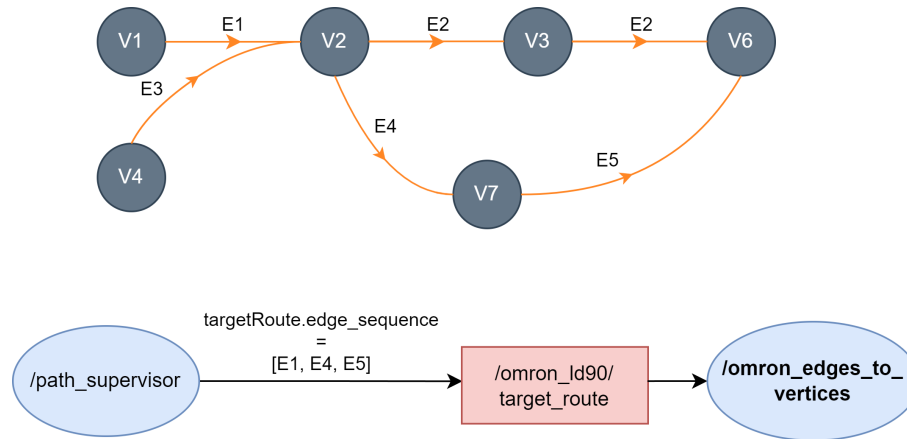


Figure 4.13: Example of a trajectory and the publishing of a sequence of edges by the *path_supervisor* node.

In this example, it is assumed that the current location of the robot is vertex V1, and the fleet manager publishes the sequence of edges [E1, E4, E5], indicating that the LD-90's destination vertex is V6. As mentioned earlier, the origin and destination vertex of each edge in the sequence are checked, resulting in the extended sequence of vertices [V1, V2, V2, V7, V7, V6]. However, considering the robot's initial position, the first vertex of the extended sequence is not used. Moreover, since there are repetitions of vertices in adjacent edges, it is necessary to exclude these duplicates. After taking all these considerations, the *omron_edges_to_vertices* node converts the initial sequence of edges [E1, E4, E5] into the vertex sequence [V2, V7, V6]. With this sequence of vertices, it is possible to guide the robot to the destination vertex of the generated path.

Additionally, the robot fleet manager can send a sequence of edges containing negative edges, indicating that the robot is intended to navigate in the opposite direction of the edge. The *omron_ros_map* takes this possibility into account while generating the sequence of vertices by swapping the origin and destination vertices of the negative edges.

However, there is no defined ARCL command that allows the sending of multiple goals to which the LD-90 has to go. This limitation leads to two approaches that have been considered to address this problem using the **goto** command:

- Sending the robot to every goal of the created sequence of vertices individually, which means that the robot will stop every time it reaches a goal.
- Alternatively, sending the robot directly to the final goal of the created sequence of vertices.

As mentioned in Section 4.4, the LD-90 mobile robot is not suitable for graph-based trajectories since it relies on free path planning algorithms. The definition of goal components in a *.map*

file allows sending the LD-90 to specific points within the map. Nevertheless, it is not possible to predict the robot's behavior until it reaches its destination. Consequently, having goal components alone is insufficient for the LD-90 to accurately follow the defined edges between vertices.

To address this challenge, a solution was found by utilizing the forbidden area component to create restricted zones on the map, preventing the robot from entering these regions. By doing so, the robot can be indirectly guided to navigate approximately along the defined edges. The creation of these forbidden areas was done using the MobilePlanner software.

By creating forbidden areas in the map that force the robot to navigate only along the edges of a trajectory, it becomes possible to employ the two approaches considered for the sending of multiple goals to the LD-90.

4.5.2 Messages Sent to the Fleet Manager

The robot fleet manager requires certain information from the robot to work properly. It is essential that the fleet manager knows the robot's location in the graph at all times. Additionally, when a sequence of edges is published for the robot to follow a specific path, the fleet manager must be informed about which edges the robot has already traversed. This information ensures efficient coordination and enables the fleet manager to manage and monitor the robot's movements effectively.

The fleet manager retrieves this information by subscribing to the `/execution_route_states` topic. In this topic, the published messages have the type `itrci_nav/execution_route_states`. This message type has the structure represented in Figure 4.14.

itrci_nav/execution_route_states
symbolic_pose.edge_id
symbolic_pose.percentage_edge
edge_sequence[].id
edge_sequence[].state

Figure 4.14: Structure of a `execution_route_states` message.

The `omron_edges_to_vertices` node is responsible for publishing this information to the `/execution_route_states` topic. When the node receives a sequence of edges, it must continuously specify the state of each edge in the sequence. There are two possible states: "processing" and "complete". When a new sequence of edges is received on the topic, every edge is initialized with the "processing" state. To determine if the robot has finished traversing through an edge, the `omron_edges_to_vertices` node subscribes to the `/omron_ld90/status/location` topic, which provides the (X, Y) coordinates of the robot on the map. Using this information, the node continuously compares these coordinates with the coordinates of the next vertex in the produced sequence of

vertices corresponding to the received sequence of edges. If the coordinates match within a defined tolerance, it indicates that the state of the edge is "complete"; otherwise, the state of the edge remains as "processing".

Additionally, the current location of the robot is specified using the edges in the received sequence, and with the same procedure used for determining the state of the edges, it is possible to specify which percentage of the edge has been covered. However, the taken approach considers that when the state of the edge changes to "complete", it means that the edge has been 100% covered; otherwise, it means that the edge has been 0% covered. Since the concepts of a graph-based trajectory are not directly applicable to the LD-90 mobile robot, it is not possible to precisely determine the percentage of edge that has been covered. Thus, the edge is either fully complete or not complete at all.

4.5.3 Messages Sent to the Robot

With the creation of the sequence of vertices corresponding to the sequence of edges received in the */omron_ld90/target_route* topic, the LD-90 mobile robot can complete the intended path generated by the fleet manager. To achieve this, the two previously considered approaches for sending the LD-90 to multiple goals were implemented.

The first approach involves publishing a **goto** command for every goal corresponding to a vertex in the sequence of vertices to the */omron_ld90/send_command* topic. In this first approach, the LD-90 receives a new command with a new goal destination every time it reaches a goal in the sequence of vertices.

For the second approach, the LD-90 is directly sent to the last goal in the sequence of vertices. Because there are restricted zones that the robot cannot navigate to due to the created forbidden areas on the map, it is guaranteed that the robot will pass by the other goals in the sequence of vertices.

4.5.4 Structure of the Node

Figure 4.15 illustrates the structure of the *omron_edges_to_vertices* node. This node serves as the direct point of communication between the developed application and the INESC TEC robot fleet manager, utilizing the */omron_ld90/target_route* topic and */omron_ld90/execution_route_states* topic. Additionally, it is possible to verify that *omron_edges_to_vertices* node also communicates with the *omron_controller* node and the *omron_connection_receiver* node through the */omron_ld90/send_command* topic and the */omron_ld90/status/location* topic, respectively.

4.6 Conclusion

This chapter explained the implementation of each individual ROS node that was developed for the application and presented the structure of each node in order to provide a deep understanding of how the system operates.

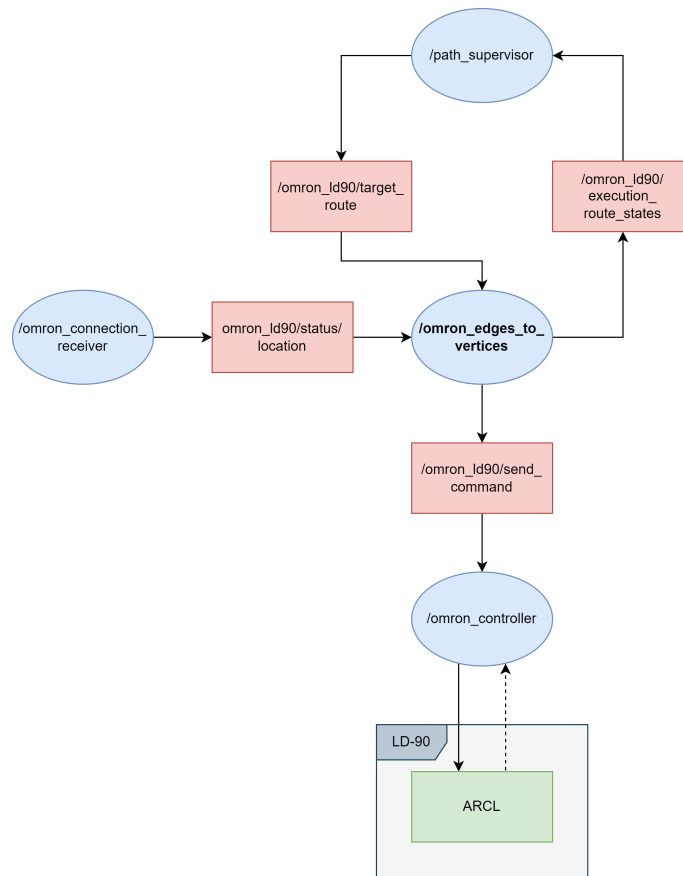


Figure 4.15: Diagram of the `omron_edges_to_vertices` node.

The system comprises four individual ROS nodes and a ROS package that encompasses three different ROS nodes. These nodes are divided into three groups: the nodes that enable direct interaction with the LD-90 mobile robot, which are the `omron_controller` node and the `omron_connection_receiver` node; the nodes responsible for the visualisation of the map and the robot within RViz, which are the nodes that compose the `Visualisation` package; and the nodes responsible for the integration of the mobile robot with the robot fleet manager, in this case, the `omron_ros_map` node and the `omron_edges_to_vertices` node.

Additionally, the processes of converting graph-based trajectories into a `.map` file that the LD-90 mobile robot can use and exchanging messages between the system and the robot fleet manager were also presented in this chapter.

Chapter 5

Results and Discussion

This chapter delves into a comprehensive project analysis, presenting and scrutinizing the results obtained through every developed ROS node and their interactions.

Additionally, an in-depth discussion of the influence of each node's performance on the overall behavior of the LD-90 mobile robot and its integration with the fleet manager will be present. This helps the identification of potential areas for optimization and refinement.

5.1 Controller Node Results

The *omron_controller* node enables establishing a connection to the ARCL server in the LD-90 mobile robot. Figure 5.1 shows the terminal output of this node when it is first launched.

```
[ INFO] [1689171728.636730728]: Boas-vindas ao socket connection OMRON LD-90
Robot Namespace: /omron_ld90
Connected...

Enter password:
Welcome to the server.
You can type 'help' at any time for the following help list.
Commands:
addCustomCommand          Adds a custom command that'll send a message out ARCL when called
addCustomStringCommand    Adds a custom string command that'll send a message out ARCL when cal
led
analogInputList           Lists the named analog inputs
analogInputQueryRaw       Queries the state of an analog input by raw
analogInputQueryVoltage   Queries the state of an analog input by voltage
applicationBlockDrivingClear  Clears an application blockDriving [abdc]
applicationBlockDrivingSet  Sets an application blockDriving [abds]
applicationFaultClear      Clears an application fault [afc]
applicationFaultQuery       Gets the list of any application faults currently triggered [afq]
applicationFaultSet        Sets an application fault [afs]
  arclSendrxt              Sends the given message to all ARCL clients
  arclStat                 Show ARCL status [ast]
  centralServer            Gets information about the central server connection
  configAdd                Adds a config file line to the config values being imported.
  configParse              Parses the imported config values and terminates the configStart command.
  configStart              Starts importing config values. Use with configAdd and configParse.
connectOutgoing           (re)connects a socket to the given outside server
  createInfo               creates a piece of information
  dock                     Sends the robot to the dock
  doTask                   does a task
  doTaskInstant            does an instant task (doesn't interrupt modes)
  echo                     with no args gets echo, with args sets echo
  enableMotors             Enables the motors so the robot can drive again
```

Figure 5.1: Terminal output of the *omron_controller* node when initialized.

By analyzing the output in Figure 5.1, it is possible to extract multiple pieces of information that are relevant to understand.

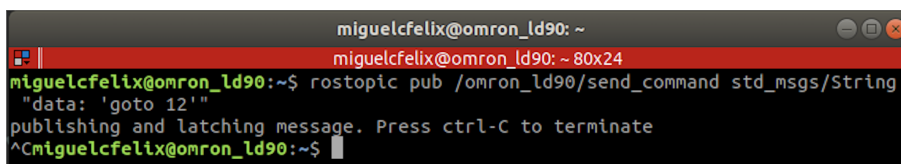
- The line "Connected..." indicates that the TCP/IP client socket created in this node has successfully established a connection with the ARCL server.
- The first message received from the ARCL server is "Enter password:", requesting the password defined in Section 4.1.1. As mentioned in 4.1.2.1, it is required that the password must be the first message sent through the socket to the ARCL server, so, in the implementation of this node, this message is automatically sent to the socket every time this node is launched.
- The line "Welcome to the server" indicates that the received password matches the one that was defined in the ARCL server configurations. Below this welcome line, it is possible to observe a list of available commands, each command being followed by a brief description.

After this first output from the node, it is possible to send multiple ARCL commands to the socket. As explained in subsection 4.1.3, this node subscribes to the `/omron_ld90/send_command` topic, and the messages published in this topic correspond to the ARCL commands considered for execution. To verify the proper functioning of this node with each command, the ARCL commands were directly published in the topic using a terminal.

For every considered ARCL command listed in subsection 4.1.3, the behaviour of the `omron_controller` node was tested by publishing every command to the `omron_ld90/send_command` topic. This can be accomplished using the **rostopic command-line tool and the pub command**, which allows users to publish data to a topic.

5.1.1 goto Command

Figure 5.2 demonstrates the use of the `rostopic` command-line tool and the `pub` command by publishing the command **goto 12** to the `/omron_ld90/send_command` topic, in order to send the LD-90 to the goal **12** defined in the map loaded on the mobile robot.



```
miguelcfelix@omron_ld90: ~
miguelcfelix@omron_ld90: ~ 80x24
miguelcfelix@omron_ld90:~$ rostopic pub /omron_ld90/send_command std_msgs/String
"data: 'goto 12'"
publishing and latching message. Press ctrl-C to terminate
^Cmiguelcfelix@omron_ld90:~$
```

Figure 5.2: Publishing of a `goto` command to the `/omron_ld90/send_command` topic using the terminal.

The `omron_controller` subscribes to the `/omron_ld90/send_command` topic. When the message shown in Figure 5.2 is published to this topic, the `omron_controller` node retrieves the command from the message and sends it to the socket. Subsequently, it receives response messages from the ARCL server through the socket, as depicted in Figure 5.3.

As mentioned in subsection 4.1.3, sending any command to the ARCL server results in specific feedback messages from the ARCL server. In the specific case of Figure 5.3, when the ARCL server received the **goto 12** command, and since the LD-90 was initially located at the docking

```

goto 12
DockingState: Undocking ForcedState: Unforced ChargeState: Float
DockingState: Undocking ForcedState: Unforced ChargeState: Not
DockingState: Undocking ForcedState: Unforced ChargeState: Overcharge
DockingState: Undocking ForcedState: Unforced ChargeState: Not
DockingState: Undocking ForcedState: Unforced ChargeState: Bulk
DockingState: Undocking ForcedState: Unforced ChargeState: Not
DockingState: Undocked ForcedState: Unforced ChargeState: Not
Going to 12
Arrived at 12

```

Figure 5.3: Response messages from the ARCL server for the *goto* command.

station, the first seven responses given by the ARCL server referred to the undocking process of the mobile robot from its docking station. With this process completed, the ARCL server then sends the feedback message that the mobile robot is navigating toward the destination goal. Upon arrival at the destination goal, the ARCL server sends the feedback message "Arrived at 12", meaning that the command was successfully executed.

5.1.2 *getGoals* Command

This command was also published on the `/omron_ld90/send_command` topic using the same process as used for the *goto* command. By using the *getGoals* command, it is possible to retrieve from the ARCL server the list of all goal components defined in the map loaded on the LD-90. The list of goals received in the `omron_controller` node is illustrated in Figure 5.4, containing the "End of goals" response line, indicating the execution of the command successfully.

```

getGoals
Goal: 12
Goal: 6
Goal: 14
Goal: 8
Goal: 22
Goal: 18
Goal: 26
Goal: 4
Goal: 10
Goal: 24
Goal: 35
Goal: 20
Goal: 2
Goal: 27
Goal: 33
Goal: 52
Goal: 1
Goal: 49
Goal: 39
Goal: 47
Goal: 32
Goal: 42
Goal: 40
Goal: 41
End of goals

```

Figure 5.4: List of goals returned by the ARCL server.

5.1.3 *dock/undock* Command

The previously used procedure used for publishing messages to the `/omron_ld90/send_command` topic was also used for the *dock* and the *undock* commands. Figures 5.5 and 5.6 illustrate the

feedback messages received on the *omron_controller* node for the *dock* and *undock* commands, respectively.

```
dock
DockingState: Undocked ForcedState: Unforced ChargeState: Not
DockingState: Docking ForcedState: Unforced ChargeState: Not
DockingState: Docking ForcedState: Unforced ChargeState: Bulk
DockingState: Docked ForcedState: Unforced ChargeState: Bulk
DockingState: Docked ForcedState: Unforced ChargeState: Overcharge
```

Figure 5.5: Response messages from the ARCL server for the *dock* command.

```
undock
DockingState: Undocking ForcedState: Unforced ChargeState: Overcharge
DockingState: Undocking ForcedState: Unforced ChargeState: Not
DockingState: Undocking ForcedState: Unforced ChargeState: Overcharge
DockingState: Undocking ForcedState: Unforced ChargeState: Not
DockingState: Undocking ForcedState: Unforced ChargeState: Bulk
DockingState: Undocking ForcedState: Unforced ChargeState: Not
DockingState: Undocked ForcedState: Unforced ChargeState: Not
Stopping
Stopped
```

Figure 5.6: Response messages from the ARCL server for the *undock* command.

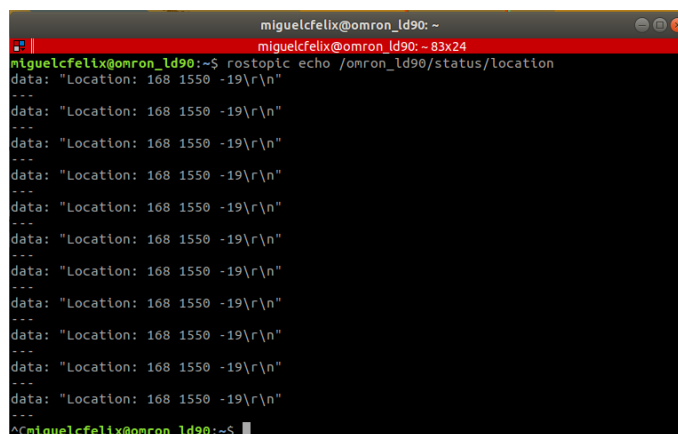
The feedback messages obtained on the *omron_controller* node for both commands contain the line associated (mentioned in subsection 4.1.3) with the completion of the execution of the command by the LD-90 mobile robot.

5.2 Connection Receiver Node Results

The *omron_connection_receiver* node is responsible for receiving information sent by the robot. As explained in Section 4.2, this node creates a TCP/IP server socket, to which the LD-90 will establish a connection. By successfully establishing a connection, the robot starts sending to the socket two defined ARCL commands: *status* and *RangeDeviceGetCurrent*. Subsequently, the *omron_connection_receiver* node splits the received information on the socket and publishes it into multiple topics, as depicted in Figure 4.6.

The obtained results are verified using the *rostopic* command-line tool and the *echo* command, which allows displaying all the messages being published on a certain topic in the terminal. There are two topics where the *omron_connection_receiver* node publishes information that are used by other nodes in the application. These topics are the */omron_ld90/status/location* and */omron_ld90/laser_points*.

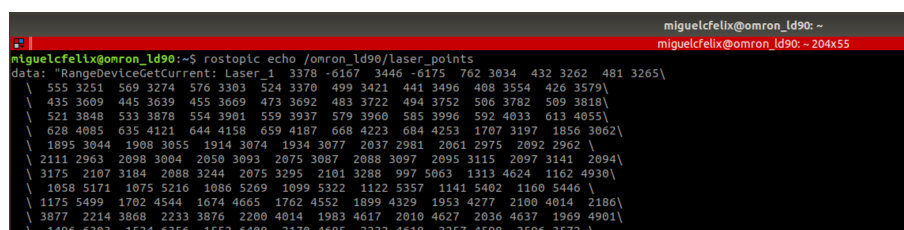
As mentioned in Section 4.2, the information received in the socket is sent by the robot as a single string that contains all the information from the two used ARCL commands. Figure 5.7 depicts the information published on the */omron_ld90/status/location* topic by the *omron_connection_receiver* node, consisting only of the *Location* field of the *status* command, as intended. From this topic, other nodes can retrieve the robot's (X, Y) coordinates and its orientation angle θ .



```
miguelcfelix@omron_ld90: ~
miguelcfelix@omron_ld90: - 83x24
miguelcfelix@omron_ld90:~$ rostopic echo /omron_ld90/status/location
data: "Location: 168 1550 -19\r\n"
----
data: "Location: 168 1550 -19\r\n"
----
data: "Location: 168 1550 -19\r\n"
----
data: "Location: 168 1550 -19\r\n"
----
data: "Location: 168 1550 -19\r\n"
----
data: "Location: 168 1550 -19\r\n"
----
data: "Location: 168 1550 -19\r\n"
----
data: "Location: 168 1550 -19\r\n"
----
data: "Location: 168 1550 -19\r\n"
----
data: "Location: 168 1550 -19\r\n"
----
^Cmiguelcfelix@omron_ld90:~$
```

Figure 5.7: Messages published on the `/omron_ld90/status/location` topic.

Furthermore, the messages published by the `omron_connection_receiver` node on the `/omron_ld90/laser_points` topic are displayed in Figure 5.8. Each message contains multiple (X, Y) coordinates corresponding to the points scanned by the robot’s laser at a specific moment.



```
miguelcfelix@omron_ld90: ~
miguelcfelix@omron_ld90: - 204x55
miguelcfelix@omron_ld90:~$ rostopic echo /omron_ld90/laser_points
data: "RangeDeviceGetCurrent: Laser_1 3378 -6167 3446 -6175 762 3034 432 3262 481 3265\
 \ 555 3251 569 3274 576 3303 524 3370 499 3421 441 3496 408 3554 426 3579\
 \ 435 3609 445 3639 455 3669 473 3692 483 3722 494 3752 506 3782 509 3818\
 \ 521 3848 533 3878 554 3901 559 3937 579 3960 585 3996 592 4033 613 4055\
 \ 628 4085 635 4121 644 4158 659 4187 668 4223 684 4253 1707 3197 1856 3662\
 \ 1895 3040 1908 3055 1914 3074 1934 3077 2037 2981 2061 2975 2092 2962\
 \ 2111 2963 2098 3004 2050 3093 2075 3087 2088 3097 2095 3115 2097 3141 2094\
 \ 3175 2107 3184 2088 3244 2075 3295 2101 3288 997 5063 1313 4624 1162 4930\
 \ 1058 5171 1075 5216 1086 5269 1099 5322 1122 5357 1141 5402 1160 5446\
 \ 1175 5499 1702 4544 1674 4665 1762 4552 1899 4329 1953 4277 2100 4014 2186\
 \ 3877 2214 3868 2233 3876 2200 4014 1983 4617 2010 4627 2036 4637 1969 4901\
 \ 1496 6303 1524 6356 1552 6408 2170 4685 2222 4618 2257 4598 2586 3572 \
```

Figure 5.8: Messages published on the `/omron_ld90/laser_points` topic.

5.3 Visualisation Package Results

The developed visualisation package, constituted of the three nodes mentioned in Section 4.3, enables the visualisation of the map created with the MobilePlanner software, along with all its components. Additionally, it provides the visualisation of the LD-90’s position on the map and facilitates sending the robot to a specific goal or dock in the map.

5.3.1 Map Visualisation Node Results

The `map_visualisation` node enables the recreation of a map displayed in the MobilePlanner interface, within a ROS environment using RViz. Therefore, this node was used to create a RViz representation of the map illustrated in Figure 5.9.

As explained in Section 4.3, the `map_visualisation` node publishes `Marker` messages in specific topics to represent the components of a `.map` file. By subscribing to these topics in RViz, it is possible to obtain the map representation in Figure 5.10. The goal and dock components are

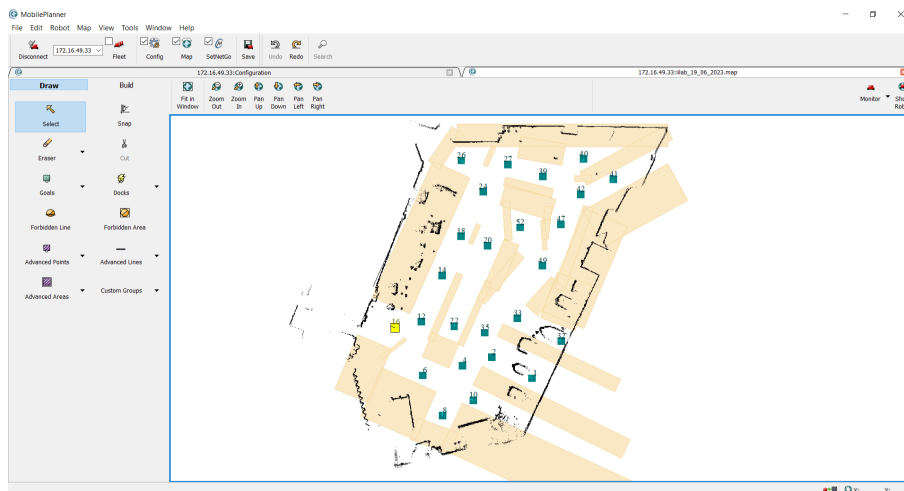


Figure 5.9: Map displayed in the MobilePlanner interface.

represented by green squares, with their names written in white on their left side. Additionally, some goals have an associated *heading*, represented by black arrows pointing to the final orientation that the robot must have when it reaches those specific goals. Forbidden areas are represented by orange squares, scanned points by small black dots, and scanned lines by red lines.

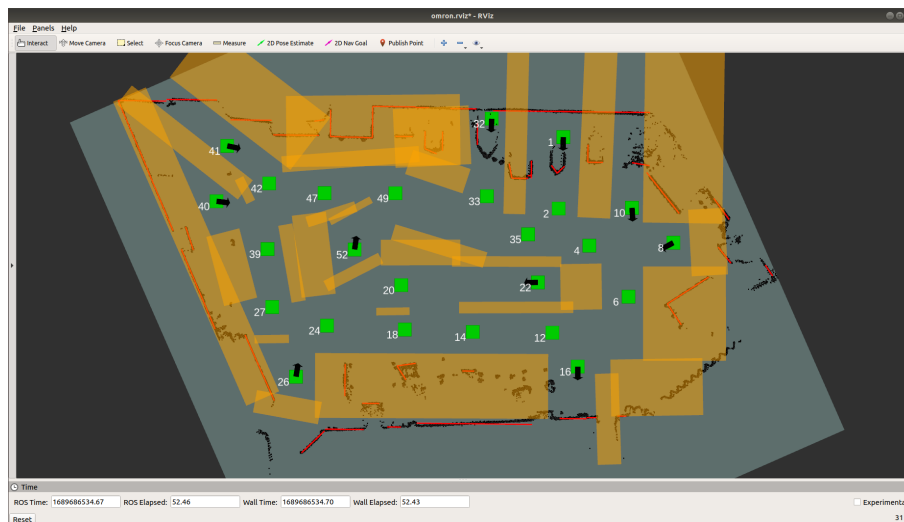


Figure 5.10: Representation of the map in RViz.

Furthermore, as mentioned in 4.3.1.4, this node creates an *OccupancyGrid* message to enable other robots to use the same map as the LD-90 mobile robot. This message is published on a specific topic and, by subscribing to this topic in RViz, it is possible to visualize the *OccupancyGrid* as depicted in Figure 5.11.

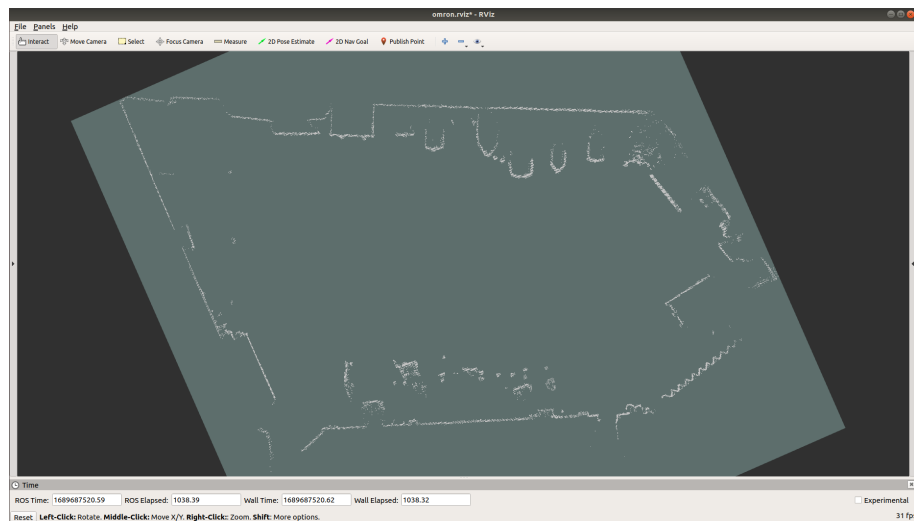


Figure 5.11: Representation of the occupancy grid in RViz.

5.3.2 Robot State Display Node Results

The *robot_state_publisher* node continuously represents the robot's location in the map by subscribing to the */omron_ld90/status/location* topic. Therefore, when the *robot_state_publisher* node is launched, it is necessary to also launch the *omron_connection_receiver* node to ensure that information is being published to the */omron_ld90/status/location* topic. The *robot_state_publisher* node uses this information to display the robot's location on the map.

In Figure 5.12, it is possible to observe a **red arrow** pointing to the goal *16* that represents the robot's position on the map. Additionally, the points scanned by the robot's laser are represented by light-blue points.

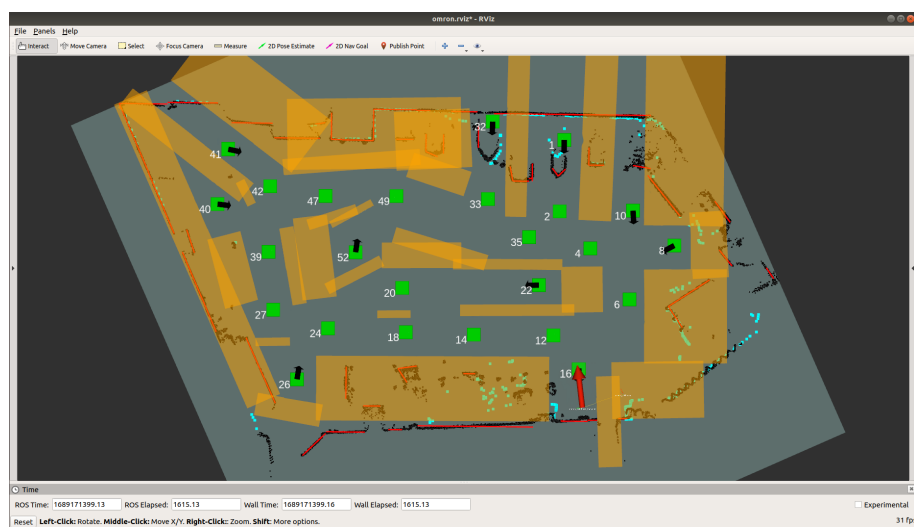


Figure 5.12: Representation of the robot's position in the map.

5.4 Trajectory Converter Node Results

The *omron_ros_map* node converts graph-based trajectories into *.map* files. To achieve this, as mentioned in Section 4.4, the node reads a *.yaml* file that is generated upon the creation of a trajectory. Thus, the trajectory depicted in Figure 5.13 was created.

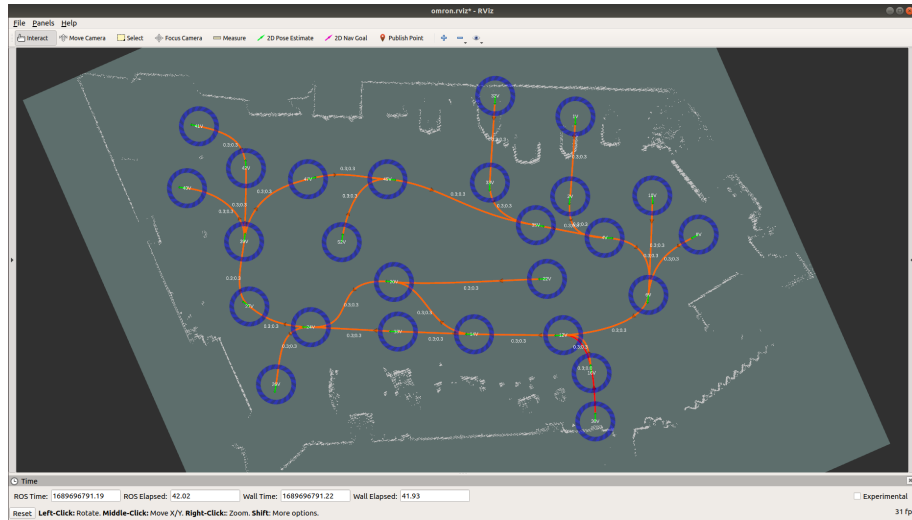


Figure 5.13: Representation of a graph-based trajectory in RViz.

When the *omron_ros_map* node is launched, a new *.map* file is created and the goals defined in it match exactly with the vertices in the trajectory. Figure 5.14 illustrates the created *.map* file displayed in the RViz interface, using the *map_visualisation* node. Additionally, the *omron_ros_map* node handles the particular vertices in a trajectory by assigning headings, represented by the **black arrows**, to the goals that are associated with those vertices.

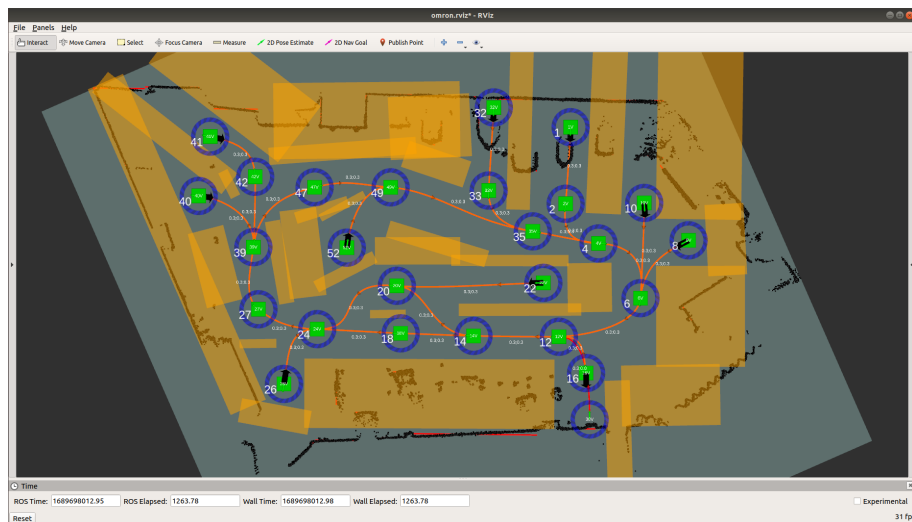
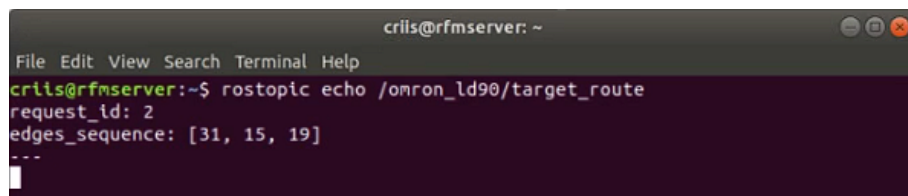


Figure 5.14: Representation of new *.map* file with a graph-based trajectory in RViz.

5.5 Fleet Manager Communication Node Results

The *omron_edges_to_vertices* node serves as the direct point of communication between the LD-90 mobile robot and the robot fleet manager. Therefore, the correct functioning of this node indicates the possibility of successfully integrating the LD-90 mobile robot with the fleet manager.

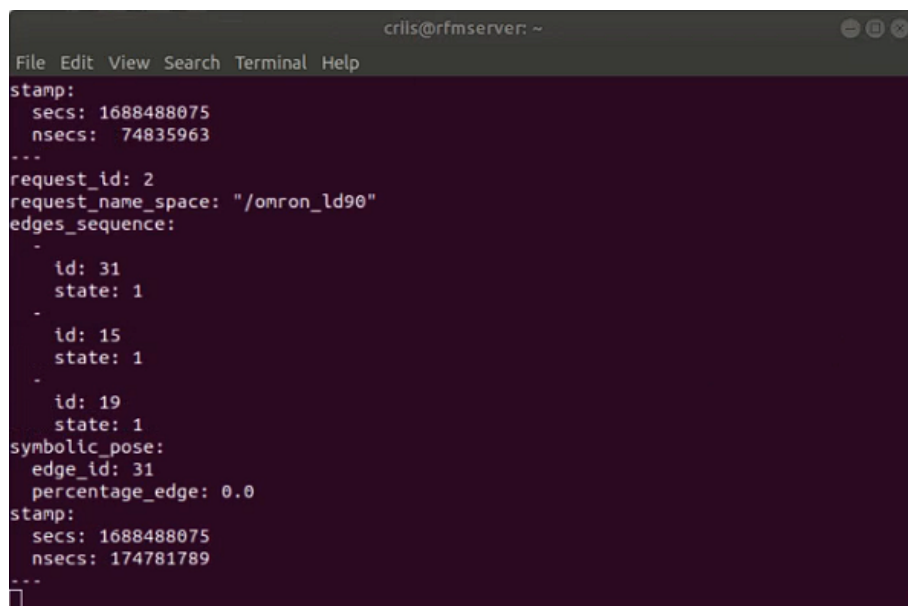
As mentioned in Section 4.5, a sequence of edges is published by the fleet manager on the */omron_ld90/target_route* topic, as illustrated in Figure 5.15. Then, the *omron_edges_to_vertices* node converts this sequence of edges into a sequence of vertices that the LD-90 must be sent to.



```
criis@rfmserver: ~  
File Edit View Search Terminal Help  
criis@rfmserver:~$ rostopic echo /omron_ld90/target_route  
request_id: 2  
edges_sequence: [31, 15, 19]  
---  
|
```

Figure 5.15: Message Received in the */omron_ld90/target_route* topic.

Considering the *.map* file represented in Figure 5.9 loaded into the robot and the sequence of edges in Figure 5.15, the node generates the sequence of vertices [12, 14, 16]. Furthermore, as explained in Section 4.5, the node must send information to the fleet manager while executing the received path. Figure 5.16 displays the messages being published by the *omron_edges_to_vertices* node on the */omron_ld90/execution_route_states* topic, which contains information regarding the current position of the LD-90 in the graph and the state of each edge in the received sequence of edges.



```
criis@rfmserver: ~  
File Edit View Search Terminal Help  
stamp:  
  secs: 1688488075  
  nsecs: 74835963  
---  
request_id: 2  
request_name_space: "/omron_ld90"  
edges_sequence:  
  -  
    id: 31  
    state: 1  
  -  
    id: 15  
    state: 1  
  -  
    id: 19  
    state: 1  
symbolic_pose:  
  edge_id: 31  
  percentage_edge: 0.0  
stamp:  
  secs: 1688488075  
  nsecs: 174781789  
---  
|
```

Figure 5.16: Message Received in the */omron_ld90/execution_route_states* topic.

The complete execution of the received sequence of edges by the LD-90 mobile robot implies that the state of every edge must be equal to 2, i.e., the state is "complete". Additionally, the position of the robot in the graph, represented by the *edge_id* parameter in the *symbolic_pose* field of the message published on the */omron_ld90/execution_route_states* topic, must be equal to the last edge in the received edge sequence and indicate that the edge is 100% complete.

The changes that occur in the message published on the *omron_ld90/execution_route_states* topic until the robot finalizes the received path are represented in Figure 5.17.

```

---
request_id: 2
request_name_space: "/omron_ld90"
edges_sequence:
- id: 31
  state: 2
- id: 15
  state: 1
- id: 19
  state: 1
symbolic_pose:
  edge_id: 15
  percentage_edge: 0.0
stamp:
  secs: 1688488081
  nsecs: 274838060
---
request_id: 2
request_name_space: "/omron_ld90"
edges_sequence:
- id: 31
  state: 2
- id: 15
  state: 2
- id: 19
  state: 1
symbolic_pose:
  edge_id: 19
  percentage_edge: 0.0
stamp:
  secs: 1688488085
  nsecs: 274773371
---
request_id: 2
request_name_space: "/omron_ld90"
edges_sequence:
- id: 31
  state: 2
- id: 15
  state: 2
- id: 19
  state: 2
symbolic_pose:
  edge_id: 19
  percentage_edge: 1.0
stamp:
  secs: 1688488090
  nsecs: 674775662
---

```

Figure 5.17: Completion of the edge sequence sent by the robot fleet manager.

5.6 Validation and Conclusion of the Results

The individual behavior of each node was tested in the previous sections. However, in order to validate and demonstrate all the algorithms developed in this dissertation, which enable the control and management of the LD-90 mobile robot by integrating it with the robot fleet manager, a test was conducted by sending a mission directly to the fleet manager.

The sending of missions to the fleet manager consists of a message with three parameters: *request_id*, which indicates the number that this mission request is associated with; *destination_vertices_id*, which represents the vertices in the graph to which the robot has to navigate; and *robot_type*, indicating the type of robot required to fulfill the request. The tested mission sent to the fleet manager is represented in Figure 5.18.

```

crlis@rfmserver: ~
File Edit View Search Terminal Help
crlis@rfmserver:~$ rostopic pub /request_MAGS itrcl_nav/go2vertex_magvs_request
"request_id: 1
destination_vertices_id: [18, 39, 52, 6, 16]
robot_type: 2"
publishing and latching message. Press ctrl-C to terminate

```

Figure 5.18: Mission sent to the Fleet Manager.

This mission indicates to the robot fleet manager that the LD-90 mobile robot must navigate to the vertices [18, 39, 52, 6, 16]. The fleet manager then plans the path the robot must follow to

reach each one of the vertices defined in that sequence. The *.map* file loaded into the robot is the one illustrated in Figure 5.9, associated with the trajectory depicted in Figure 5.13. Therefore, the robot will travel through multiple vertices in the trajectory, ending in vertex 16, corresponding to the dock.

As mentioned in Section 4.5, two approaches were considered for sending multiple goals into the robot: sending the robot vertex by vertex in an edge sequence or sending the robot directly to the final vertex in the edge sequence. Although both approaches work correctly, the first introduces delays on the robot fleet manager, as the robot stops when it reaches every vertex in a sequence of edges. Therefore, the adopted approach was to send the LD-90 mobile robot directly to the final vertex in the edge sequence. A complete diagram of the nodes executed during this mission and their interaction can be seen in Appendix A. Additionally, in Appendix B, a video demonstration can be seen, representing the complete execution of this mission.

The development of every ROS node enables the control of the LD-90 mobile robot and the integration of this robot with a robot fleet manager that utilizes a type of trajectory that is not directly applicable in robots that use free path planning algorithms, like the LD-90 mobile robot. This leads to the successful integration of the LD-90 with the INESC TEC robot fleet manager, as depicted in Figure 5.19.

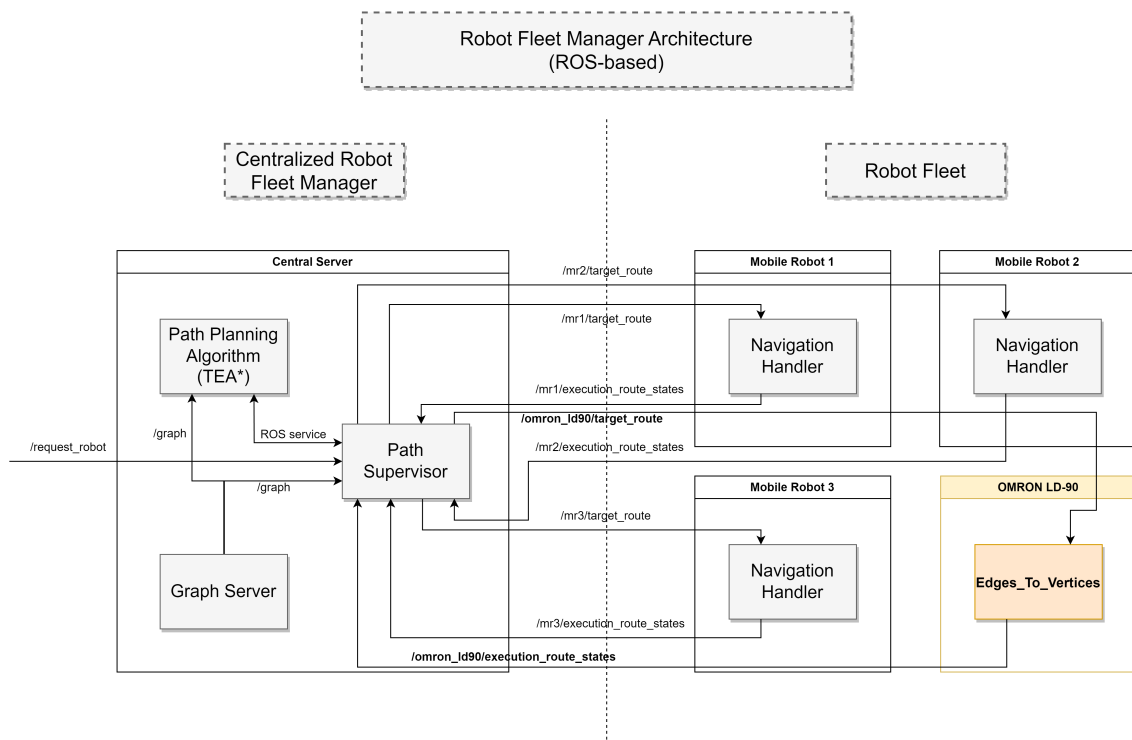


Figure 5.19: Fleet Manager Diagram with the OMRON LD-90.

Chapter 6

Conclusions and Future Work

6.1 Accomplishment of the Proposed Goals

The main objectives of this dissertation were to develop a software module to control the OMRON LD-90 mobile robot, which is a non-ROS-based robot primarily relying on the software produced by its manufacturer, and to integrate this robot that employs free path planning algorithms with a robot fleet manager that utilizes graph-based path planning algorithms, all within the ROS framework.

The created system consists of several ROS nodes, each with different responsibilities for ensuring the proper functioning of the entire system. The results obtained confirm the successful implementation of a communication interface between this non-ROS-based AMR and a ROS-based architecture. The development of these software modules was based on the ARCL interface, developed by OMRON, and involved the use of TCP/IP sockets.

Furthermore, integrating this robot into a fleet manager that uses graph-based path planning algorithms, indicates the possibility of having a heterogeneous fleet of robots managed by this robot fleet manager.

6.2 Future Work

Although the determined goals for this dissertation were achieved, there are some considerations that can improve the developed application. Firstly, it is important to note that the LD-90 mobile robot does not precisely follow the edges in the graph-based trajectory; instead, an approximation was made. To achieve this approximation, forbidden areas were created using the MobilePlanner software. These forbidden areas forced the robot to navigate only in specific zones corresponding to the edges of the trajectory.

Therefore, in the **Trajectory Converter Node**, it would be important to develop an algorithm that automatically creates these forbidden areas in the newly generated *.map* file. By doing so, the created software would not require any use of the robot's manufacturer's own software, and it would improve the accuracy of the robot's movement along the graph-based trajectory.

Additionally, another aspect that would improve the efficiency of integrating the LD-90 with the robot fleet manager is having access to the free paths generated by the LD-90 path planner. This access could enable another approach to verify which edges the robot had already passed through while executing a mission. Consequently, the use of forbidden areas could be eliminated. By leveraging the free-path information from the LD-90 mobile robot path planner, the developed application could send the fleet manager more accurate and real-time feedback on the robot's trajectory and progress. This would lead to a reduced dependency on the MobilePlanner software.

Furthermore, OMRON has developed a new version of the ARCL interface, which includes a command that allows sending a robot to a goal while enforcing it to pass through intermediate goals. However, this version has not been utilized with the LD-90 mobile robot yet. By incorporating this command in future work, it would enable the robot to navigate along the edges of graph-based trajectories, eliminating the need for using forbidden areas entirely.

References

- [1] Ángel Madridano, Abdulla Al-Kaff, David Martín, and Arturo de la Escalera. Trajectory planning for multi-robot systems: Methods and applications. *Expert Systems with Applications*, 173:114660, 2021. doi:<https://doi.org/10.1016/j.eswa.2021.114660>.
- [2] Pedro Luís Cerqueira Gomes da Costa et al. Planeamento cooperativo de tarefas e trajectórias em múltiplos robôs. 2011.
- [3] Joana Santos, Pedro Costa, Luís F. Rocha, A. Paulo Moreira, and Germano Veiga. Time enhanced a*: Towards the development of a new approach for multi-robot coordination. In *2015 IEEE International Conference on Industrial Technology (ICIT)*, pages 3314–3319. IEEE, 2015. doi:[10.1109/ICIT.2015.7125589](https://doi.org/10.1109/ICIT.2015.7125589).
- [4] Enrique Fernandez, Luis Sanchez Crespo, Anil Mahtani, and Aaron Martinez. *Learning ROS for robotics programming*. Packt Publishing Ltd, 2015.
- [5] OMRON. I614-e-01 mobile robot software suite, 2007.
- [6] Alexander Tiderko. multimaster_fkf. http://wiki.ros.org/multimaster_fkf. Accessed on July 5, 2023.
- [7] Mary B. Alatise and Gerhard P. Hancke. A review on challenges of autonomous mobile robot and sensor fusion methods. *IEEE Access*, 8:39830–39846, 2020. doi:[10.1109/ACCESS.2020.2975643](https://doi.org/10.1109/ACCESS.2020.2975643).
- [8] Francisco Rubio, Francisco Valero, and Carlos Llopis-Albert. A review of mobile robots: Concepts, methods, theoretical framework, and applications. *International Journal of Advanced Robotic Systems*, 16(2):1729881419839596, 2019. doi:[10.1177/1729881419839596](https://doi.org/10.1177/1729881419839596).
- [9] Giuseppe Fragapane, Rene De Koster, Fabio Sgarbossa, and Jan Ola Strandhagen. Planning and control of autonomous mobile robots for intralogistics: Literature review and research agenda. *European Journal of Operational Research*, 294(2):405–426, 2021. doi:<https://doi.org/10.1016/j.ejor.2021.01.019>.
- [10] Zuo L. Cao, Sung J. Oh, and Ernest L. Hall. Dynamic omnidirectional vision for mobile robots. *Journal of Robotic Systems*, 3(1):5–17, 1986. doi:<https://doi.org/10.1002/rob.4620030103>.
- [11] Jianqi Zhang, Xu Yang, Wei Wang, Jinchao Guan, Ling Ding, and Vincent C.S. Lee. Automated guided vehicles and autonomous mobile robots for recognition and tracking in civil engineering. *Automation in Construction*, 146:104699, 2023. doi:<https://doi.org/10.1016/j.autcon.2022.104699>.

- [12] Tao Yang, You Li, Cheng Zhao, Dexin Yao, Guanyin Chen, Li Sun, Tomas Krajnik, and Zhi Yan. 3d tof lidar in mobile robotics: A review. *arXiv preprint arXiv:2202.11025*, 2022.
- [13] Y Xiang, XQ Yang, WW Yang, and WH Miao. Localization and mapping algorithm for the indoor mobile robot based on lidar. In *IOP Conference Series: Materials Science and Engineering*, volume 831, page 012021. IOP Publishing, 2020. doi:10.1088/1757-899X/831/1/012021.
- [14] Chad Goerzen, Zhaodan Kong, and Bernard Mettler. A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems*, 57:65–100, 2010. doi:10.1007/s10846-009-9383-1.
- [15] Steven M LaValle, James J Kuffner, BR Donald, et al. Rapidly-exploring random trees: Progress and prospects. *Algorithmic and computational robotics: new directions*, 5:293–308, 2001. doi:10.1201/9781439864135-43.
- [16] Robert Martin C. Santiago, Anton Louise De Ocampo, Aristotle T. Ubando, Argel A. Bandalá, and Elmer P. Dadios. Path planning for mobile robots using genetic algorithm and probabilistic roadmap. In *2017IEEE 9th international conference on humanoid, nanotechnology, information technology, communication and control, environment and management (HNICEM)*, pages 1–5. IEEE, 2017. doi:10.1109/HNICEM.2017.8269498.
- [17] Con Cronin, Andrew Conway, and Joseph Walsh. State-of-the-art review of autonomous intelligent vehicles (aiv) technologies for the automotive and manufacturing industry. In *2019 30th Irish signals and systems conference (ISSC)*, pages 1–6. IEEE, 2019. doi:10.1109/ISSC.2019.8904920.
- [18] Anis Koubâa and J.Ramiro Martínez-de Dios, editors. *Cooperative Robots and Sensor Networks 2015*, volume 604 of *Studies in Computational Intelligence*. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-18299-5.
- [19] A Jamshidpey, M Wahby, MK Heinrich, M Allwright, W Zhu, and M Dorigo. Centralization vs. decentralization in multi-robot coverage: Ground robots under uav supervision. 2021.
- [20] Zhi Yan, Nicolas Jouandeau, and Arab Ali Cherif. A survey and analysis of multi-robot coordination. *International Journal of Advanced Robotic Systems*, 10(12):399, 2013. doi:10.5772/57313.
- [21] Mobile Industrial Robots. Mir fleet. Accessed on July 20, 2023. URL: <https://www.mobile-industrial-robots.com/solutions/mir-accessories/mir-fleet/>.
- [22] OTTO Motors. Otto fleet manager. Accessed on July 20, 2023. URL: <https://ottomotors.com/fleet-manager>.
- [23] Fetch Robotics. Fetchcore cloud software. Accessed on July 20, 2023. URL: <https://fetchrobotics.com/cloud-software/>.
- [24] OMRON. Fleet manager | omron. Accessed on July 20, 2023. URL: <https://industrial.omron.co.uk/en/products/fleet-manager>.
- [25] Qing Rebecca Lia, Zachary Dydek, and Daniel Theobald. Why interoperability is critical to the warehouse of the future. In *ISR Europe 2022; 54th International Symposium on Robotics*, pages 1–7. VDE, 2022.

- [26] André R. Baltazar, Marcelo R. Petry, Manuel F. Silva, and António Paulo Moreira. Autonomous wheelchair for patient's transportation on healthcare institutions. *SN Applied Sciences*, 3:1–13, 2021. doi:[10.1007/s42452-021-04304-1](https://doi.org/10.1007/s42452-021-04304-1).
- [27] Morgan Quigley, Brian Gerkey, and William D Smart. *Programming Robots with ROS: a practical introduction to the Robot Operating System*. " O'Reilly Media, Inc.", 2015.
- [28] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.
- [29] Lentin Joseph and Jonathan Cacace. *Mastering ROS for Robotics Programming: Design, build, and simulate complex robots using the Robot Operating System*. Packt Publishing Ltd, 2018.
- [30] David Gossow, Adam Leeper, Dave Hershberger, and Matei Ciocarlie. Interactive markers: 3-d user interfaces for ros applications [ros topics]. *IEEE Robotics & Automation Magazine*, 18(4):14–15, 2011. doi:[10.1109/MRA.2011.943230](https://doi.org/10.1109/MRA.2011.943230).
- [31] Zebang Shen, Xiaowei Xu, Peng Zhi, and Rui Zhao. Chapter 2 - introduction to robot operating system. In Qingguo Zhou, Zebang Shen, Binbin Yong, Rui Zhao, and Peng Zhi, editors, *Theories and Practices of Self-Driving Vehicles*, pages 27–62. Elsevier, 2022. doi:<https://doi.org/10.1016/B978-0-323-99448-4.00002-3>.
- [32] OMRON. I618-e-02 advanced robotics command language, 2017.
- [33] Jon Postel and Joyce K Reynolds. Telnet protocol specification. Technical report, 1983. doi:[10.17487/RFC0854](https://doi.org/10.17487/RFC0854).
- [34] Michael J Donahoo and Kenneth L Calvert. *TCP/IP sockets in C: practical guide for programmers*. Morgan Kaufmann, 2009.
- [35] Sergi Hernandez Juan and Fernando Herrero Cotarelo. Multi-master ros systems. *Institut de Robotics and Industrial Informatics*, pages 1–18, 2015.
- [36] OMRON. Adept arcl reference guide, 2016.
- [37] Tony Pratkanis Brian Gerkey. map_server. http://wiki.ros.org/map_server. Accessed on July 9, 2023.

Appendix A

OMRON Mission Structure

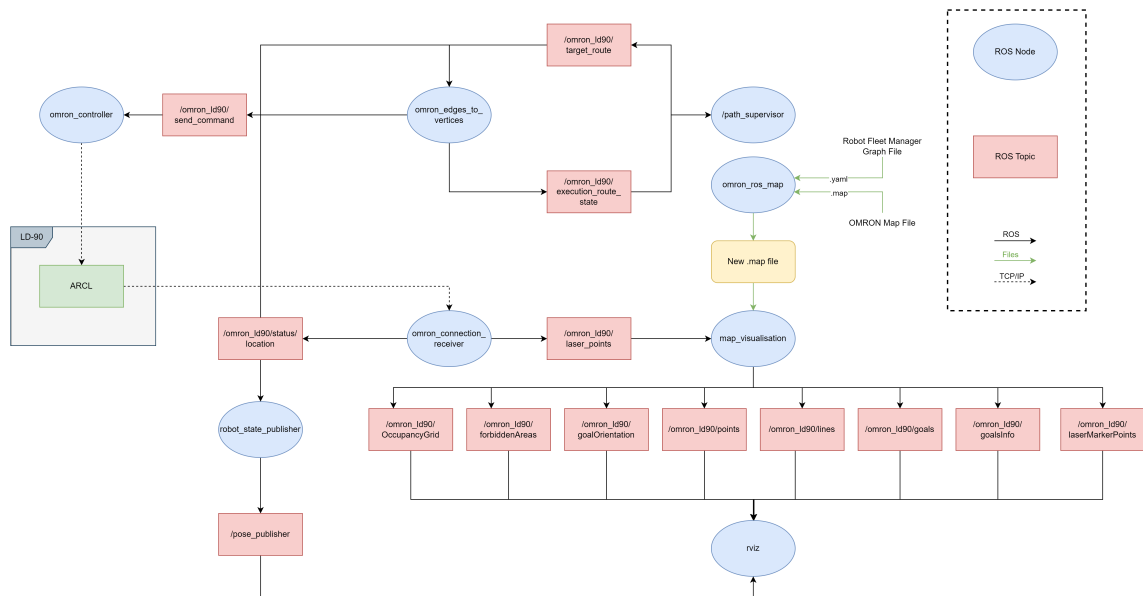


Figure A.1: Diagram of the nodes executed during the demonstration.

Appendix B

Videos Taken

Video demonstration of the complete execution of a mission, sent by the robot fleet manager, performed by the OMRON LD-90 mobile robot:

- Link: <https://youtu.be/igFFSyZQV80>