Wireless Sensor Network Integrated With ROS For Danger Avoidance In Mobile Robotics

Pedro Henrique Ferreira Mendes - 43608

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master Degree in Industrial Engineering. Work developed during the double degree exchange program between the Polytechnic Institute of Bragança (IPB) and the Federal Technological University of Paraná (UTFPR).

Work oriented by:

Prof. D.Sc. André Chaves Mendes

Prof. PhD. Luís Fernando Caparroz Duarte

Bragança

2022

Wireless Sensor Network Integrated With ROS For Danger Avoidance In Mobile Robotics

Pedro Henrique Ferreira Mendes - 43608

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master Degree in Industrial Engineering. Work developed during the double degree exchange program between the Polytechnic Institute of Bragança (IPB) and the Federal Technological University of Paraná (UTFPR).

Work oriented by:

Prof. D.Sc. André Chaves Mendes

Prof. PhD. Luís Fernando Caparroz Duarte

Bragança

2022

Acknowledgement

At first, I thank God for giving me the strength and health that allowed me to develop this work. I also want to express my gratitude to my parents Flávio and Silvia Mendes, siblings Flávia and João, and fiance Ana Paula Freitas Costa for all the unconditional love and support. Without our long video calls and chatting, this would be a much harder period, I love you all.

I also thank UTFPR and IPB for providing the opportunity to develop this work, this experience has made me a better student, professional, and human being. I especially thank my supervisor, Professor Ph.D. André Chaves Mendes, for all the support and knowledge you shared with me. I will always be grateful for the opportunity of working with you. And my co-supervisor Professor Ph.D. Luis Fernando Caparroz Duarte, our exchange of ideas and your perspective enriched the content of this work, thank you.

I am also very grateful to all my friends, Guido Berger, Alexandre Júnior, Arezki Chellal, Marina Ribas, Luis Piardi, Thadeu Brito, Jonas Queiroz, João Braun, Lucas Sakurada, Matheus Zorawski, João Mendes, Inês Sena, Felipe Teixeira, Gustavo Funchal, Victória Melo, Tiago Franco, Leonardo Sestrem, Adriano Silva, Tatiana Schreiner, Adriano Henrique, and Flávia Pires, thank you very much for all the support and the friendship.

I thank Leonardo Furst and Rômulo Martins, for welcoming me into the house, and making my adaptation in Bragança much easier. Thank you for your friendship and all the support.

Abstract

Environmental awareness is a crucial task that robots must perform to navigate autonomously. Moreover, it must be well executed to make navigation safer and collision-free. Since autonomous mobile robots are being used in dynamic scenarios where a simultaneity of events occurs, it becomes even more difficult to correctly sense and perceive the environment. This paper proposes the integration of a wireless sensor network with the Robotic Operating System – *ROS* – to incorporate advanced information from the environment into layered cost maps used by the robot to navigate. The system architecture was implemented, evaluated in a simulated environment, tested and validated, and the results obtained showed the effective gain in the computation of the paths and in the reduction of the computational load of the associated subsystems. With positive results in a simulated environment, the system was deployed on a robotic platform and evaluated in a real scenario, through experiments. The results obtained in the experiments prove the gain in the navigation process of the platform due to the better perception of the environment in the tested scenarios.

Keywords: Autonomous Mobile Robot, Wireless Sensor Network, Robot Operating System, Gazebo Simulator, IoT, TurtleBot3, Roomba©.

Resumo

A perceção do ambiente é uma tarefa crucial que os robôs têm de realizar para navegar de forma autónoma. Além disso, deve ser bem realizada para tornar a navegação mais segura e livre de colisões. Como os robôs móveis autónomos estão a ser empregados em cenários dinâmicos, onde ocorre uma simultaneidade de eventos, tornam-se ainda mais difícil a detecção e a percepção adequada do ambiente. Este trabalho propõe a integração de uma rede de sensores sem fios com o Sistema Operacional Robótico – *ROS* – para incorporar informações avançadas do ambiente em mapas de custos por camadas utilizados pelo robô para navegar. A arquitectura do sistema foi implementada, avaliada em ambiente simulado, testada e validada, e os resultados obtidos mostraram o ganho efetivo no cômputo dos percursos e na redução da carga computacional dos subsistemas associados. Com resultados positivos em ambiente simulado, o sistema foi implantado numa plataforma robótica e avaliado num cenário real, mediante experimentos. Os resultados obtidos nos experimentos comprovam o ganho no processo de navegação da plataforma devido a melhor percepção do ambiente nos cenários testados.

Palavras-chave: Robô Móvel Autónomo, Rede de Sensores Sem Fio, Robotic Operating System, Simulador Gazebo, IoT, TurtleBot3, Roomba©.

Contents

1 Introduction **1**

1.1 Objectives 2

1.2 Document Structure 3

2 State of the Art **5**

2.1 Autonomous Mobile Robotics 5

2.1.1 Perception (Sensing) 7

2.1.2 Localization 8

2.1.3 Simultaneous Localization and Mapping 10

2.1.4 Path Planning 10

2.2 Robot Operating System (ROS) 13

2.2.1 Navigation Stack 15

2.2.2 Layered Costmap 16

2.3 Internet of Things and Wireless Sensor Networks 17

2.3.1 Wemos Development Board and the ESP8266 18

2.3.2 MQTT 19

3 System Architecture **21**

3.1 Main Idea 21

3.2 Information Flow 23

3.3 Sensor Modules 24

3.4 MQTT Broker 27

3.5 Robot System 28
3.5.1 First Python Script 29
3.5.2 Plug-In C++ Code 31
3.5.3 Second Python Script 33

4 Simulation Development **36**
4.1 Gazebo and the TB3 36
4.2 Creating the Simulated World 39
4.3 Simulation Experiment 42
4.4 Main Script Controller 44
4.4.1 Inputs and Outputs 44
4.4.2 Algorithm Flowchart 45
4.5 Navigation Stack TB3's Configuration 49

5 Real Scenario Development **51**
5.1 Roomba© 51
5.2 RP Lidar 53
5.3 Sensor Modules 54
5.3.1 Case 56
5.4 Raspi Broker 57
5.5 Real Experiment 57

6 Results and Discussion **62**
6.1 Simulation Results 63
6.2 Real System Results 67
6.3 Discussion 69

7 Conclusion and Future Works **75**
7.1 Developed Works 75
7.2 Future Works 76

List of Figures

2.1 Steps to achieve autonomous navigation [4]. 7
2.2 Localization block diagram [7]. 8
2.3 Dijkstra example of path planning [15]. 12
2.4 A* example of path planning [15]. 12
2.5 Navigation stack structure [18]. 15
2.6 Layered costmap adapted from [20]. 16
2.7 IoT architecture layers [23]. 18
2.8 Applications of WSNs [25]. 18
2.9 Wemos D1 Mini [26]. 19
2.10 Architecture of MQTT [28]. 20
2.11 Process of MQTT [28]. 20

3.1 Main idea of the system. 23
3.2 Information flow of the system. 24
3.3 HC-RS04 and its functioning principle [31]. 25
3.4 Firmware flowchart. 27
3.5 Python script running on the broker machine flowchart. 28
3.6 Flowchart of the proposed system running on the robot. 29
3.7 Array created for updating the layer. 30
3.8 Flowchart of the first Python script. 30
3.9 Flowchart of the *"Update Costs"* function. 32
3.10 Flowchart of the second Python script. 34

4.1 Publications citing simulators between 2016 and 2020 [39]. 37
4.2 TB3 models [41]. 38
4.3 TB3 in a simulated environment. 38
4.4 TB3 standard components [41]. 39
4.5 TB3 dimensions [41]. 39
4.6 ESTIG blueprint. 40
4.7 ESTIG blueprint only with corridors. 40
4.8 Building editor with the image imported. 41
4.9 Simulated ESTIG building. 42
4.10 Experiments scenario. 43
4.11 Experiment runs flowchart. 44
4.12 Files generated by the Python script. 46
4.13 Flowchart of the Python controller script. 47
4.14 Bricks as obstacles blocking the pathway. 48

5.1 iRobot© Roomba© model 676 [44]. 52
5.2 Roomba© with the structure for ROS controlling. 53
5.3 RP Lidar [46]. 54
5.4 Parts of the wireless sensor node. 55
5.5 Sensor module circuit schematic designed in KiCad EDA. 55
5.6 Sensor node case in 3D model. 56
5.7 Sensor module complete. 56
5.8 Map generated by the Gmapping SLAM algorithm. 58
5.9 Edited map used in the experiments. 59
5.10 Module placed on the wall. 60
5.11 Module placed on the corridor wall. 60
5.12 Obstacles used to block the passage. 61

6.1 Default setup. 64
6.2 `Full Setup`. 64

6.3 Distance travelled each route. 65
6.4 Time to complete the routes. 65
6.5 Move base percentage of CPU consumption. 66
6.6 Move base percentage of memory usage. 66
6.7 Distance travelled on the routes. 67
6.8 Time to complete the routes. 67
6.9 Move base percentage of CPU consumption. 68
6.10 Move base percentage of memory usage. 69
6.11 Simulated and real distances. 70
6.12 Simulated and real times. 70

Acronyms

AMR	Autonomous Mobile Robot
AMCL	Adaptive Monte Carlo Localization
AP	Access Points
DWA	Dynamic Window Approach
ESTIG	Escola Superior de Tecnologia e Gestão
GIMP	GNU Image Manipulator Program
IPB	Instituto Politécnico de Bragança
IoT	Internet of Things
MQTT	Message Queuing Telemetry Transport
MCU	Microcontroller Unit
MVP	Minimal Viable Product
MCL	Monte Carlo Localization
RAM	Random Access Memory
RBPF	Rao-Blackwellized Particle Filter
RTT	Rapidly Exploring Random Tree
ROS	Robotic Operating System
SLAM	Simultaneous Localization and Mapping
SDK	Software Development Kit
TB3	TurtleBot 3
URDF	Universal Robotic Description File
VNC	Virtual Networking Computing
VM	Virtual Machine
WSN	Wireless Sensor Network

Chapter 1

Introduction

Robotics research has been evolving and growing since it was first defined. Its various applications boosted the developments from the beginning. Following the field tendency, mobile robotics has been in the spotlight for some time. Giving the robot the capability of movement through navigation only increases the possible tasks it can perform. However, navigation is not an easy mission to complete. It involves the integration of several systems that work closely together.

Indoor guidance is one of the fields where wheeled autonomous mobile robots (AMRs) are deployed. Museums, hospitals, restaurants, shopping centers, schools, and universities are only a few examples where a guidance robot could be beneficial and make people's lives easier.

To achieve their tasks, AMRs have to navigate through places. Navigating is not an easy mission. It depends on several algorithms that integrate various information acquired by the robot controller.

Navigation in these dynamic environments always implies avoiding unexpected obstacles, such as a group of people, objects left out of place, and several other dangerous situations. In industrial places, robots have other possible dangers, such as temperature or dirt, depending on the industry. Considering that, the more information about the environment the AMR could get, the better it will be able to navigate.

Although the quality of the navigation algorithms and software integration is always

progressing, the outputs are always heavily influenced by the quality and quantity of inputs, which in this case is data collected from sensors that sense the environment for the robot's controller to use as inputs to the algorithms. A good perception of the surroundings is crucial in every AMR operation.

Although several works have proposed navigation solutions and integration of sensor data, the vast majority of these approaches only consider data coming from sources on the robot's frame.

The Internet of Things (IoT) and AMRs are two of the most evolving fields in technology and engineering. Both sectors have ample possibilities of applications, which naturally leads to an overlapping characteristic, meaning that there are several cases where both technologies may be applied together to achieve better goals. The AMR market prior to COVID-19 accounted for US$4.98 billion, but predictions are that it will reach US$54 billion by 2023 [1]. The IoT market is also experiencing enormous growth. It is expected that by 2025 it will reach 75 billion connected devices worldwide, with an average growth rate of 35% per year [2]. As one can see, these figures show how important, and even crucial, the research in those areas is.

In that context, and acknowledging that there are predictions that by 2030 there will be around 100 trillion IoT sensors in the world [2], this work aimed at developing an architecture integrating information coming from distant sources, such as wireless sensor network (WSN) from IoT sensors, into the Robot Operating System (ROS) map, which would allow the robot to have advanced information about its environment. As a result, the robot senses the environment in advanced sections of the path it travels, far from the range of its platform's sensor array.

1.1 Objectives

This dissertation's primary purpose is to develop a way to integrate wireless sensor information into the robot's map for better navigation. The robot in which this project would be deployed is a guidance AMR that is supposed to run around the main building

of the Escola Superior de Tecnologia e Gestão (ESTIG) of the Instituto Politécnico de Bragança, Portugal.

A two-step approach was proposed to test and validate the developed architecture. At *first*, the work focused on developing the integration of data into the ROS structure, which was done in a simulated environment. *Once this step was completed*, the hardware was proposed and deployed in the real scenario. At this point, the sensor node's structure, firmware, hardware, and the software required to provide information between the machines were designed. The chronological steps of the work were as follows:

- Study the ROS environment and the Navigation Stack (software in ROS used for navigation in AMRs);
- Develop an integration between IoT nodes and the ROS mapping module;
- Build a simulated environment of the real scenario and the robot model to test and validate the architecture;
- Code a script in Python to control the simulation and automatically run tests, gather data, and analyze it;
- Run the modified ROS Navigation stack in a real robot and execute experiments in the physical environment; and,
- Adapt the Python script used in the simulation to control the experiments in the real world, gather data and analyze it.

1.2 Document Structure

The document is divided into seven chapters. The first is an introduction comprising contextualization and motivation.

The second chapter is about the state of the art, reviewing the critical literature around AMRs, ROS, and IoT, detailing the basic approaches and techniques in those research fields and challenges.

The third chapter is an overview of the system architecture, detailing the main ideas and how every piece of development is integrated with the others.

Chapter 4 details the work done in the simulations performed to validate the ideas before deploying them in the real scenario.

Chapter 5 describes how the work previously done in the simulation was launched on the real robot on the real site.

Chapter 6 presents the results for both real and simulated scenarios, as well as analysis and discussion. Finally, the work is concluded in Chapter 7.

Chapter 2

State of the Art

This chapter is a brief review of the AMRs, IoT, and WSN fields of research. At first, a study on AMRs is presented, comprising their essential characteristics and challenges. After that, an analysis of the ROS environment is performed. At first describing its global architecture, then focus on the navigation stack and layered costmaps. The last section aims at the IoT and wireless sensors network. Finally, the chapter will present the fundamentals of the IoT field and the related technologies used in this work: Message Queuing Telemetry Transport (MQTT) and the ESP8266 development board.

2.1 Autonomous Mobile Robotics

Robotics is a relatively new concept introduced into human society. The idea of a robot as humans know it today was conceived in science fiction literature. The work of the Capek brothers brought up the term "robota", a Czech word that means laborer. After that, Isaac Asimov, another literature icon of science fiction, was responsible for popularizing the term in his work "I, Robot". Since that, the concept of robotics has been evolving at an exciting pace. Research shows that academic publications with the term "robot" or "robotics" have been growing since the end of the last century [3].

The various possible applications for robotics developed that field of science a lot. A robot may walk, roll, jump, slide, skate, swim, and fly. According to how the system

is deployed, one can categorize it as stationary, land-based, air-based, and water-based. The land-based robots have inner classifications, wheeled, legged, track slip, and hybrid locomotion. In this work, the focus is on the wheeled robots. They are robots that use wheels to move around their habitat. These systems are easier to deploy and control than legged ones and are known for navigating easily through most types of surfaces, although they are not the best choice when the robot has to go over higher obstacles. Since the wheels are in touch with the floor, the robot's balance is less of a problem than in other architectures. Mobile robots can also be divided into other categories (according to their dynamics), such as differential drive, car-type drive, omnidirectional drive, and synchro drive [4].

As the development of robotics grows, the new technology allows these systems to perform better and better, to the point where human supervision is less required. In this context, the rise of autonomous mobile robots appears, systems that perform tasks autonomously without the guidance of a human user. These AMRs can operate in unpredictable scenarios and partially unknown habitats. Nowadays, one can find these systems deployed in medical care, personal and household services, entertainment, industrial automation, rescue operations, surveillance, construction, guidance, space exploration, and more. Executing tasks with no human commander implies that the robot has to achieve steps that were not within its scope. One can divide these steps (or tasks) into three main categories that are considered the basics of AMRs, locomotion, perception, and navigation. Locomotion involves all the kinetics aspects of the system, which means the actual physical control of the robot, comprising factors such as maneuverability, controllability, terrain conditions, efficiency, stability, and many others. Perception means sensing the environment, which makes it one of the most critical steps to achieving autonomous navigation. If the robot has a good awareness of the site, all the other tasks will be fed with accurate data improving the results, increasing the quality of the missions performed. Navigation comprises everything the robot does to go from one place to another. To achieve that, it needs to know where it is, where it needs to go, and how to get there, which involves localization and motion and path planning [5]. Figure 2.1

presents a flowchart of the phases AMRs go through to navigate.

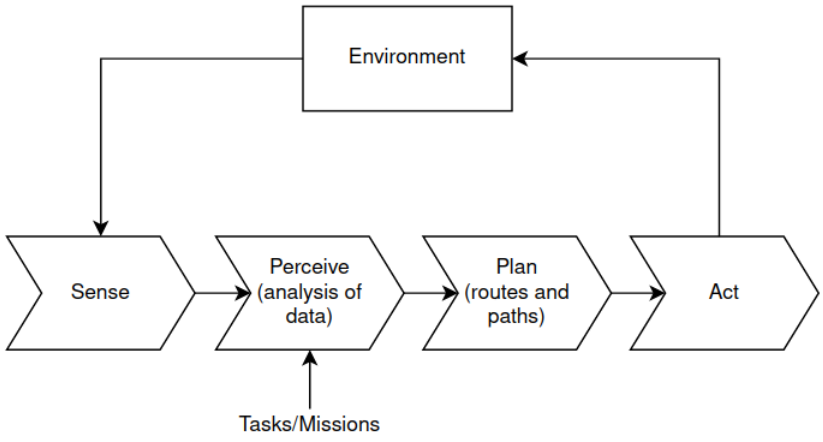

Figure 2.1: Steps to achieve autonomous navigation [4].

2.1.1 Perception (Sensing)

As mentioned before, perception is one of the most important tasks an AMR has to accomplish. The quality and quantity of data acquired directly influence the performance of the other phases of the operation. This gathering of information starts with measurements by, usually, several sensors. Afterward, algorithms are applied to extract relevant information from that data set. Some of the outputs of this sensing are object recognition, positioning, gesture and speech recognition, obstacle avoidance, and so on [6]. There are various approaches to exploit these inputs. For example, the academy can easily find applications with machine learning and artificial intelligence.

One of the most deployed techniques for the treatment of sensory information is sensor fusion (or data fusion). These techniques take more than one source of data as input and combine them to achieve better results than if the algorithms only had one source. Integrating multiple sources of knowledge can reduce uncertainty, increase accuracy, and reduce costs. An important detail of why this is a powerful tool is that by having more than one sensor, the eventual abnormalities in the readings of one can be completed by the others. Some popular sensor fusion algorithms are Kalman Filter, Particle Filter, Bayesian

Network, and Dampster-Shafer. The algorithms can be divided into state estimation methods and decision fusion methods, where the first are deployed to determine the state of an anticipated and continuously changing system. The second combines decisions of multiple classifiers to reach a mutual decision [5].

2.1.2 Localization

After sensing its environment, an AMR's next task is to localize itself. Some authors state that its the most challenging process to perform. The literature describes three fundamental problems of localization. The first is local pose tracking. The previous position is needed to track the current location, and it is resolved using a model where the robot's belief in its local is treated as a distribution. The second is global localization. The robot has to localize itself concerning a global map, which is solved by a uniform distribution as the initial pose. The third is the kidnapped robot problem, when a robot is taken to another location on the map without the robot's knowledge [7]. Figure 2.2 shows a diagram of the usual localization process.

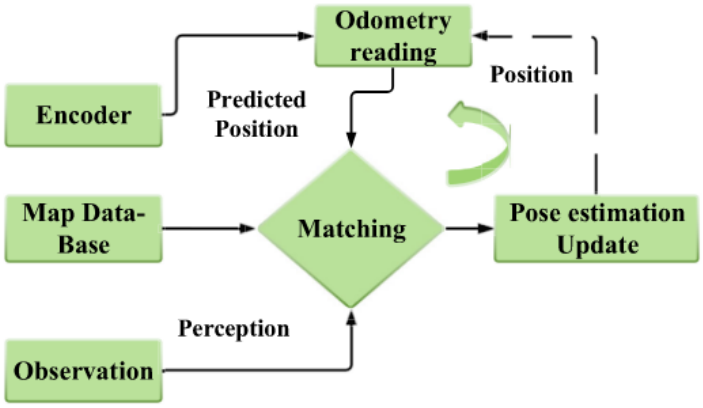

Figure 2.2: Localization block diagram [7].

Figure 2.2 presented one of the possible localization methods. There are other possibilities, such as beacon-based, light-based, landmark-based, and odor based. After gathering information, there is the aforementioned data fusion and filtering. The most deployed techniques are Kalman filter localization, Markov-Kalman localization, and Monte Carlo

localization (MCL) [7]. The proposed system uses the adaptive MCL to localize itself. The following paragraphs will review its basics.

The MCL algorithm is a probabilistic mobile robot localization approach. MCL is a particle-filter-based implementation of recursive Bayesian filtering localization. The process estimates a pose based on the map information, the robot's motion, and the robot's position related to objects in the environment. Based on the set of measurements the system can perform, a likelihood function is computed, which gives the probability of the robot being at a given position. Each iteration evaluates the likelihood function at sample points (called particles) randomly distributed according to the posterior estimate [8].

The density probability function $p(z \mid x, m)$ estimates the likelihood of the measurement z, if the robot is at the x position in the environment m. As mentioned before, each particle has a probability calculated by this function (defined by the system's sensory characteristics). After that, a weight is calculated, and only the higher weighted particles are kept as good predictions of the robot's pose [8] [9].

The algorithm assumes one of the particles is the actual pose of the robot, calculating only for the posteriors, making it more efficient, even though the computational load is usually high. These particles may be seen as pseudo-robots. The bigger the environment, the more particles are needed to achieve good results, although the computational burden increases. Since the calculations occur for posterior particles, when the robot is brought up at a known location, the algorithm has to be fed with the accurate pose of the system [10].

One evolution of MCL is Adaptive MCL (AMCL), which adjusts the number of particles according to the certainty of the pose estimation. When there is higher uncertainty, the algorithm uses more particles. When the estimations are more confident, it uses fewer particles. Therefore, it adjusts the calculations as it needs, achieving better computational usage [9].

2.1.3 Simultaneous Localization and Mapping

Simultaneous Localization and Mapping (SLAM) is a technique that allows the robot to localize and create a map while it goes through an environment. There are two main types of SLAM, lidar-based (lidar SLAM) and camera-based (visual SLAM). This work will use the lidar approach. This approach uses laser scans of the site and odometry information from the robot. With those data, it is possible to perceive landmarks and define the robot's position and orientation according to these landmarks. After that, a map is constructed with this laser information. Some algorithms were proposed, for example, the *Gmapping, Karto, and Hector* [11].

This dissertation will use the *Gmapping* algorithm, an open-source software that produces a grid-based map of the environment. This algorithm implements a Rao-Blackwellized Particle Filter (RBPF) to solve the SLAM problem [12].

2.1.4 Path Planning

To complete its missions, an AMR has to move from state "A" to state "B". This change in position has to be done safely, avoiding obstacles, and it is key to being as optimized as possible. Path planning is determining a collision-free route to guide the robot from its current position to the desired one. That is done using sensory information, map data sets, and localization knowledge. As the autonomous navigation field expands, path planning has become an area of focus. As one may infer, it is not an easy task to complete, especially when talking about AMRs in dynamic and ever-changing habitats. Some popular approaches stand out from others, but the truth is that for every application, there is a better fit [13].

The algorithms may differ based on previous knowledge of the robot's environment, the degrees of freedom of the system if the environment is static or dynamic, and others. Path planning may also be global or local. Global planning intends to produce a path based on extensive and complete site information. It works better for static environments since it usually takes static mapping information, not including dynamic objects. Local

path planning is usually deployed in unknown and dynamic environments. Global paths are usually calculated before the robot starts moving and local while the robot is moving [13].

This work deploys a combined approach of global and local path planning. The most common algorithms for global path planning are *Dijkstra, A Star*, including its variants [13], and *Rapidly Exploring Random Tree (RTT)* [14]. The local planner chosen was the Dynamic Window Approach, which will be presented in the following paragraphs.

The *Dijkstra* algorithm finds the shortest path in a graph from one point to another. It may calculate to every point in the map graph as it expands from the current point until it reaches the desired one. One may say it is a reliable path planner, even though it burdens the system's memory for calculating every possibility of movement to find the shortest one [13]. The first task is to set the cost for crossing to adjacent nodes (point in the map). Usually, the diagonal cost is higher. The next step is to define the starting node and set the costs to get to every node as infinite. Now, two sets of nodes are created, the "open" and the "closed". Here, the process calculates the cost to get to each node from the current (initial pose). After checking all the possibilities of crossing to neighbors, the algorithm places the current node in the closed list and adds to the "open" all the adjacent ones. This process occurs until there are no nodes in the open list and the current checking node is the target. Note that the algorithm has to know the neighbors for each node. The critical information is the cost to go from one node to another, which is computed back to the initial node, and the chosen path is the less costly trajectory. Besides the lowest cost to get to each node, the algorithm keeps the previous node, the node from where the cost is calculated coming. Figure 2.3 shows an example of Dijkstra. The colored nodes were the ones visited by the method. As one can see, it expands in all directions, checking the costs until it reaches the goal.

Dijkstra is one of the most fundamental algorithms for pathfinding. *A Star* is an evolution of the method. This process follows the same steps but evaluates the cost of each node differently. In *A Star*, the cost of each node follows a sum of the crossing cost already applied in *Dijkstra* and a heuristic cost. This heuristic cost represents a

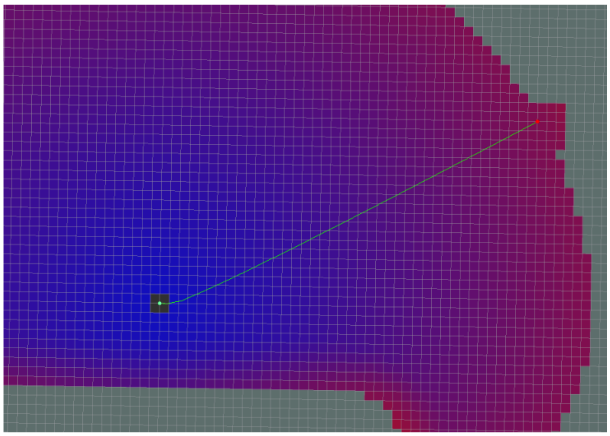

Figure 2.3: Dijkstra example of path planning [15].

measure that translates the distance from the current node to the goal node. It may be the euclidean distance or the Manhattan distance, for example. So basically, what it does is it guides the search to those nodes closer to the goal, saving memory as the *Dijkstra* algorithm computes the calculations to every neighbor node until it reaches the goal [16]. Figure 2.4 shows an example of the *A Star* search. As one can see, it guides the search to the closest nodes to the goal, unlike *Dijkstra*.

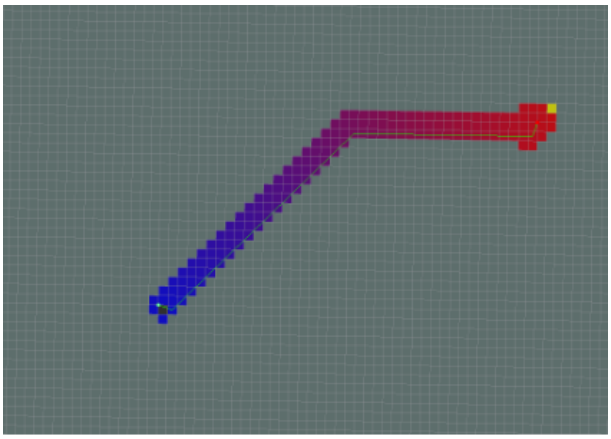

Figure 2.4: A* example of path planning [15].

The local planner used in this work is the dynamic window approach (DWA). It is one of the most deployed algorithms for local planning in AMRs. The method transforms the trajectory production into a constrained optimization challenge on the velocity vector state. It takes a set of multiple possible linear and angular velocities of the system and

simulates the trajectory for each of them. After that, it scores each trajectory based on a set of parameters the developer may produce. For example, the cost is infinite if the trajectory passes through an obstacle. If the trajectory goes far from the goal point, the cost is high, and so on. Then the less costly trajectory is chosen. Note that, to restrict the possibilities of velocities and trajectories produced, a limit of sampled velocities and a limit time for the simulation are set [14].

2.2 Robot Operating System (ROS)

As one can see, deploying an AMR or even a more straightforward robotic system depends on integrating several kinds of subsystems. From low-level drivers to data fusion and other sophisticated software as well as hardware and mechanical knowledge, a roboticist has to gather a massive amount of knowledge to develop and integrate all this set of subsystems. In that context, ROS was developed to solve some problems of deploying robots. It is one of the most popular open-source platforms providing dynamic middleware with publisher/subscriber communication and a remote-procedure-call mechanism. The different modules in the control architecture implement the functionalities mentioned above, also basic actions and report events about their state by subscribing and publishing messages. All the process is controlled by the *"rosmaster"* acting as a broker, making it much easier to integrate different executable codes, saving development time, and encouraging an open-source community of developers. That leads to code sharing, and a helping-each-other environment [17].

ROS was first released in 2009, and its philosophical goals were: peer-to-peer, tools-based, multi-lingual, thin, and open source [17].

Peer-to-Peer means that all the processes communicate directly with each other. These processes may be running on more than one host machine, and there is a "master controller" (called textitrosmaster) that ensures every process finds the others. As one may infer, multi-lingual means that ROS supports more than one programming language. Initially, there were four, Octave, Python, C++, and LISP. This cross-language support

was achieved by using a language-neutral interface definition language, which is used to describe the messages communicated between the modules. These messages comprise regular data structures, facilitating the multi-lingual characteristic. Tools-based means that the internal structure of ROS is modularized into tools used to build and run its tasks. They believe the gains in stability compensate for the possible loss in efficiency. By thin, they mean that ROS encourages developments that do not depend exclusively on its structures. That way, the software developed may be adapted and used outside the original environment. ROS performs modular builds using CMake, so all the software complexity is virtually inserted into the developer's libraries. Therefore, extracting it from the original environment for other applications is easier as ROS only integrates these "fat" (standalone) libraries [17].

There are four fundamental concepts behind ROS's structure. Nodes, messages, topics, and services. Nodes are software modules that perform computational tasks like the ones described in the last section. These nodes exchange information through messages that are strictly typed data structures. Nodes send messages by publishing them into topics. If a node needs information from a topic, it subscribes to it. A topic may have multiple subscribers and publishers. This kind of communication is considered a broadcast, which is inappropriate for synchronous processes, which may be crucial in some systems. To solve that, ROS has its services. A service is composed of two messages and a string name. One message is the request, and the other is the answer. Unlike topics, a service can only be processed by one node [17].

The software in ROS is organized into packages, a combination of code to perform some tasks, and XML files define them. Meta-packages are packages containing multiple packages inside. To launch all these executable nodes, ROS uses launch files written in XML, where one can run multiple nodes simultaneously, for example, to initialize the whole system [17]. The ROS version used for developments was the Noetic, the last release of ROS 1. ROS 2 is on its first release and is optimized for real-time systems. Not all platforms have already adhered to ROS 2, which is the case here. The drivers needed were not present in ROS 2 yet. Therefore the latest version of ROS 1 was used.

2.2.1 Navigation Stack

The navigation stack is a meta-package in ROS. This set of software integrates sensor information, such as odometry, laser scans, and mapping data, to control the robot. Figure 2.5 shows the structure of the navigation stack, and its modules [18].

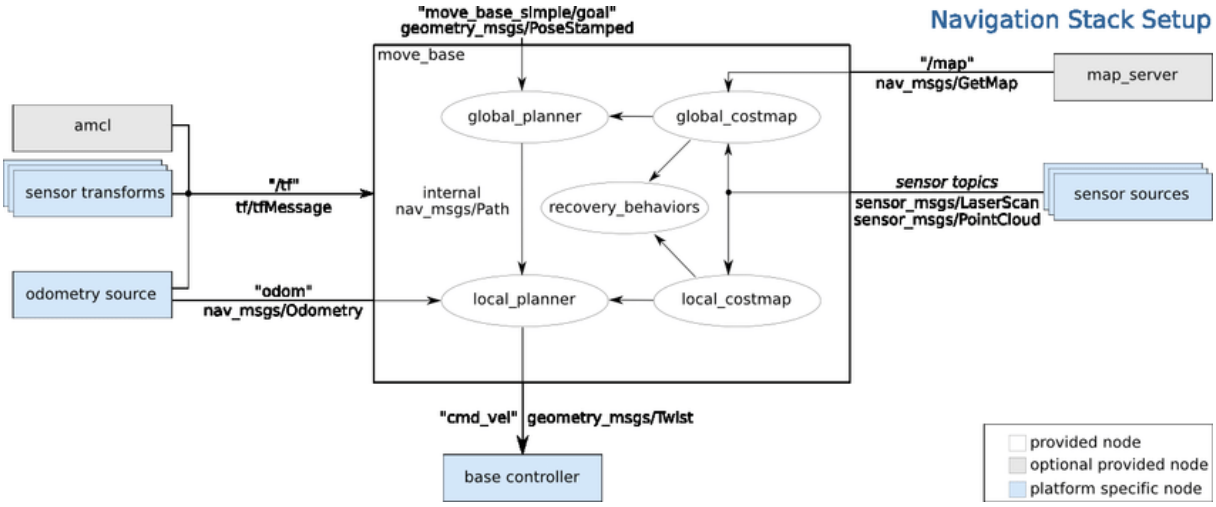

Figure 2.5: Navigation stack structure [18].

As shown in the figure, the navigation stack integrates all software needed for a robot to navigate autonomously. It includes mapping, sensing, perceiving, planning, and acting. Mapping is usually done in two ways: running a SLAM algorithm [19] or providing a static map of the environment to the robot. In this work, a static map produced with SLAM will be provided to the AMR. Since the Navigation Stack was developed to address any robotic platform in virtually any environment (as long as it runs ROS and provides the input data), almost all of its modules have a large set of parameters that the developer has to adjust to fit its desires better. These parameters interfere directly with the behavior of the AMR. Therefore, to produce the best response, a suitable parametrization is critical. The work developed in [15] presents the most important modules and parameters to adjust in the YAML configuration and launch files of the Navigation Stack, showing tests and simulations of different sets of parameters. It also brings up some problems one usually faces when setting up the stack on a system.

2.2.2 Layered Costmap

Although one can provide static maps to the platforms running ROS, the environments in which these robots operate are usually not static. Because of that, ROS allows us to use layered costmaps. Costmaps are grid-based maps that store information about their cells. That is, if the cell is free, occupied, or if its state is unknown, [20] show the basic idea for the implementation in ROS, compared with other techniques, and presents the benefits. Figure 2.6 illustrates this concept. The final map is a sum of all the values stored in the layers. There are three default layers in ROS, the static, the obstacles, and the inflation layer. The static layer is built if the static map information. The obstacles are updated with sensory information, and the inflation layer inflates these obstacles (static and dynamic) to give the robot some safety zone to go around these obstacles.

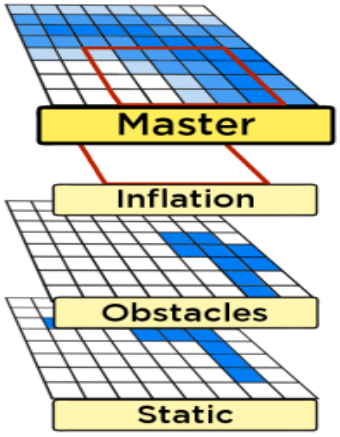

Figure 2.6: Layered costmap adapted from [20].

The algorithm has two significant steps for each layer. At first, the method *"Update Bounds"* is called and finds the area where there is a need for an update. After that, the method *"Update Costs"* sets the new costs of each cell and recalculates for the master layer [20]. As one can see, each layer added may take different information sources to contribute to the map construction. ROS allows users to create as many layers as they need (with the downside of computational cost) as plugin-ins [21]. They may be used for various purposes, for example, creating human interaction behavior, prohibited zones in the map,

and hallway movement alignment [20]. This dissertation proposes using wireless sensors to update a layer and create *"danger zones"*, giving the robot advanced information on the environment. The architecture will be presented in Chapter 3.

2.3 Internet of Things and Wireless Sensor Networks

The term "Internet of Things" was first coined in the 90s, and one of the most straightforward definitions may be *"things that are associated over the Internet"*. Associated means the transfer of information from various sources to destined places. Here, "things" may be any device or gadget with some sensing ability that produces data and sends this data to other locations through the Internet. So basically, IoT provides a bridge from the real world to the virtual one using machine-to-machine communication. This real-world information gets to where it needs to be, enabling the possibility of analysis and computation in different locations. Oxford Dictionaries defines it as "The interconnection via the Internet of computing devices embedded in everyday objects, enabling them to send and receive data" [22], [23].

The basic IoT architecture can be described in three main layers. The first is a perception/actuation layer composed of sensors and actuators (plus the needed components to control them). The second layer is the network part of the system, composed of routers, and gateways, basically the infrastructure that enables communication with the Internet. The last is the application layer, where the gathered data goes, where computation and analysis are performed. This layer is composed mainly of servers (usually cloud computing servers) [23]. Figure 2.7 presents the layers configuration.

A WSN is a group of sensor and routing nodes gathered to monitor some environment. These WSNs collect information and send it to a central processing unit that treats that information [24].

These nodes have usual modeling formed by three intercommunicating parts: sensing, processing, and communication. There is a fourth module, the power supply, usually limited to battery systems, making the management of power an important issue for the

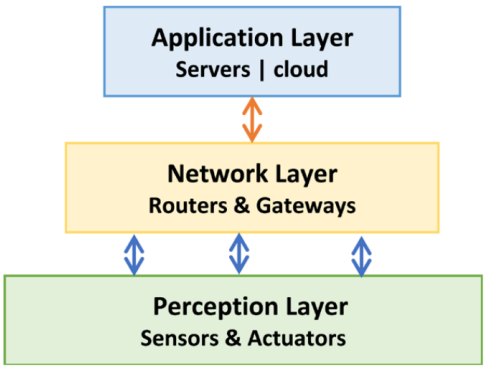

Figure 2.7: IoT architecture layers [23].

operational time of the node [25]. Figure 2.8 presents the described structure. The sensing component is the sensor of the module.

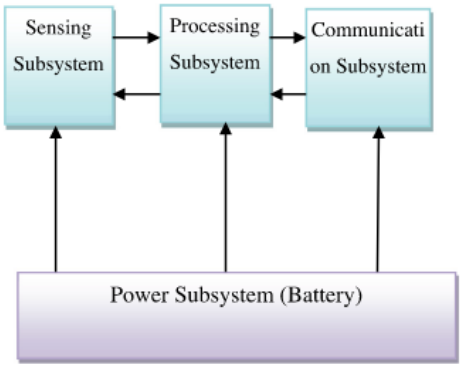

Figure 2.8: Applications of WSNs [25].

2.3.1 Wemos Development Board and the ESP8266

The processing module of the sensor nodes architecture presented above is usually deployed with microcontroller units (MCUs). In this work, a development board called Wemos D1 Mini was used. It is produced by the Chinese company Wemos. This board is based on the ESP8266, a popular MCU in the IoT world. The D1 Mini comprises all the hardware needed to prototype embedded systems rapidly. It brings 11 digital input/output pins with interrupts, PWM, I2C, one analog pin, SPI, UART, and WiFi. This module may come with an external antenna or a printed circuit antenna. It is a low-cost module

that became popular for its cost-benefit. It is a relatively powerful module for its size and capabilities, making it perfect for IoT and WSN applications. Figure 2.9 presents the board pin-out and some functionalities. The ESP8266 is based on the Tensilica CPU L106, a 32-bit processor [26][27].

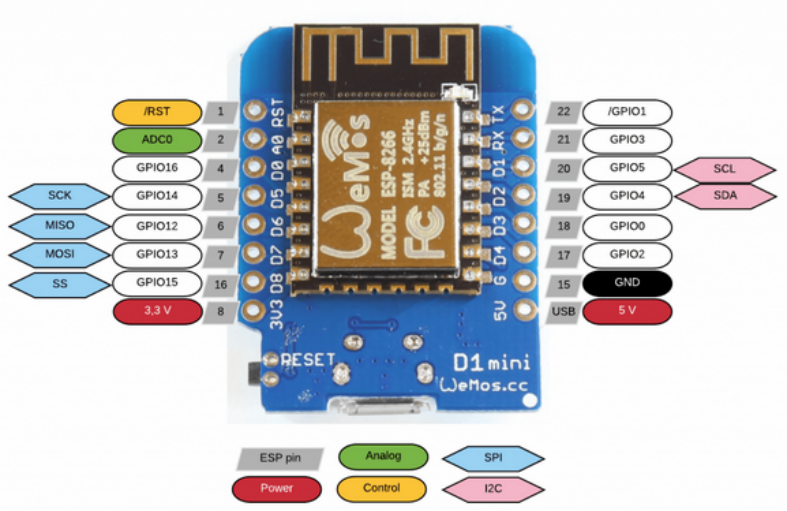

Figure 2.9: Wemos D1 Mini [26].

2.3.2 MQTT

MQTT was first introduced in 1999 and standardized in 2013. It is a messaging protocol aiming to connect embedded IoT devices with other resources, such as serving or performing machine-to-machine communication. It uses a routing mechanism and is a connection protocol considered an excellent choice for IoT and WSNs. It is built based on the TCP protocol. The architecture of MQTT is based on a publisher/subscriber method, where the basic components are the subscriber, the publisher, and the broker [28]. Figures 2.10 and 2.11 show the architecture and the process of the protocol.

Publishers and subscribers are considered clients. They are responsible for connecting

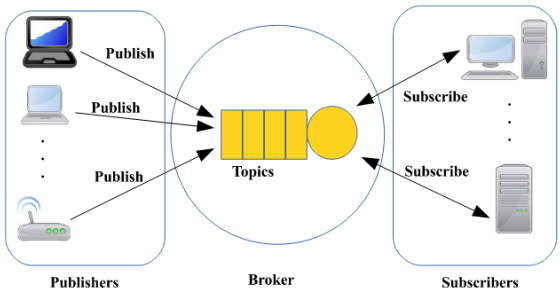

Figure 2.10: Architecture of MQTT [28].

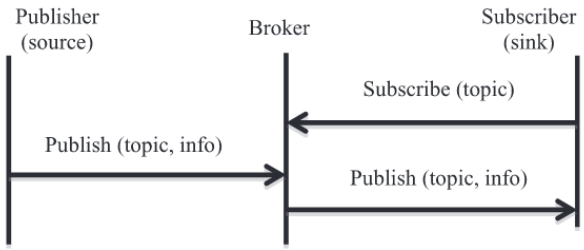

Figure 2.11: Process of MQTT [28].

to the broker. These clients may publish messages in topics, subscribe to topics, unsubscribe, and detach from the broker. The broker controls the flow of messages. It is responsible for receiving the messages and resending them to subscribed clients. Its primary responsibilities are to accept client requests, receive the messages, process clients' subscriptions and unsubscriptions as well as redirect the messages [29].

Chapter 3

System Architecture

This chapter addresses the system's architecture, describing the general ideas and presenting and describing all the components of the proposed system. The flow of information is also described for better understanding. As mentioned before, the leading general idea is to integrate wireless sensor nodes with the ROS mapping modules running on a robot that is supposed to guide people around the first floor of the ESTIG building. This integration aims to improve the perception system of the robot, allowing better navigation. Besides producing better navigation through better routes, and by better, one should understand safer and possibly faster. This work also proposes that with this better knowledge of the environment, the navigation stack may be more delicate parameterized to reduce computational load.

3.1 Main Idea

As studied in the past chapter, perception is probably the key component for a good AMR deployment. If the perception system is not solid, one may assume that the localization and navigation will not be adequate. Therefore, the more information about the robot's environment, the better. Depending on the AMR's habitat, this idea conflicts with most robots' architecture to the point where the perception modules usually depend on sensors mounted on the robot's frame. The problem with that is that the sensory capacity of the

system gets limited to the sensors' range. In sites like our study environment (inside of a building with corridors and few open spaces, which will be described later), the limitation would be even worst, as the corridors "restrict" the range of the sensors due to the walls. The only information the sensors would read would be literally around the robot, limiting the knowledge about the environment. Note here that there is no problem with the actual wall readings. They are essential for many reasons (localization, for example). However, they block the "vision" to other parts of the building where the robot would have to pass through and will only have the static map information.

To overcome this situation and give the robot advanced information about the environment, a wireless sensor system is proposed to monitor some corridors and send the information to the robot through MQTT communication. Any time the robot receives new information from the sensor nodes, it recalculates the route according to the new map data. Figure 3.1 illustrates the main idea. In that figure, there is an AMR on the left upper side and the goal on the right lower corner. The image resembles a warehouse environment with shelves creating corridors. A sudden danger to the AMR appeared in the middle of the first route calculated (red line). Note that this trouble could be anything that makes the robot's passage undesired, for example, obstacles left in the pathway, too high or too low temperatures, maintenance, and more. In the regular systems, the sensors on the AMR's frame would only detect the danger when it is just about to take the turn, giving the robot little time to readjust accordingly.

On the other hand, the green line simulates an AMR's behavior with this dissertation's proposed system. The sensor nodes would detect the trouble right when it became dangerous, so the robot has more time to adjust the path and behave to avoid the dangerous area. Observe also that having this advanced information gives the robot space to recalculate the route from a place where it can optimize it, meaning that the correction of the route as soon as possible may lead to a reduction in time and distance traveled.

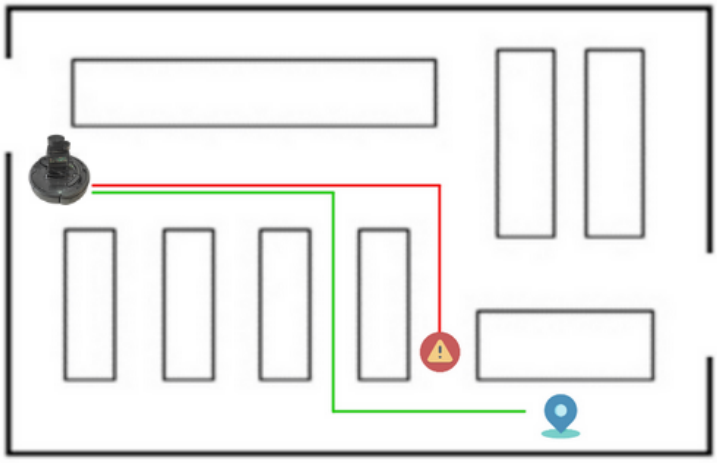

Figure 3.1: Main idea of the system.

The proposed system comprises three main modules, the robot running ROS, the wireless sensor nodes, and the central MQTT broker node. This separation of the broker from the robot system avoids adding a higher computational load on the robot's controller, which is an already computationally constrained embedded system. The following sections will cover the flow of information in the system and describe how each subsystem behaves to accomplish the tasks they are supposed to perform.

3.2 Information Flow

The sensor node is responsible for sensing the trouble. Once the danger is detected, the MCU classifies the area monitored as "blocked" or "clear". Once a decision is made, the wireless node publishes the result into the specific MQTT topic. Each surveilled area has a node and a respective MQTT topic. The broker machine is subscribed to every area topic and receives all the messages. Here, a script matches the nodes to their map localization points. If the information from the nodes is consistent, the code either places the danger area on the map or removes it. That is achieved through a WSN MQTT topic where the broker machine publishes the data for each area. The robot controller is subscribed to the WSN MQTT topic and receives the information from the broker. Once it receives the message, the robot arranges the data and inserts it in the ROS environment

by publishing it to a ROS topic called *wsn*. From now on, the WSN topic will be the MQTT one, and the wsn topic will be the ROS topic. The layer where the danger zones will be created is subscribed to *wsn* topic and uses that information to update the map.

At this point, the danger zone could already have been created, and all the information has reached its destination. However, depending on the parameters of the navigation stack, this upgrade on the map does not produce a recalculation of the route. To overcome this possibility, treatment has to be done to trigger the recalculations and achieve the possible gains. This procedure is accomplished through the ROS structure and uses the information already available in ROS topics. Each part is detailed in the following subsections. Figure 3.2 show a flowchart for better visualization.

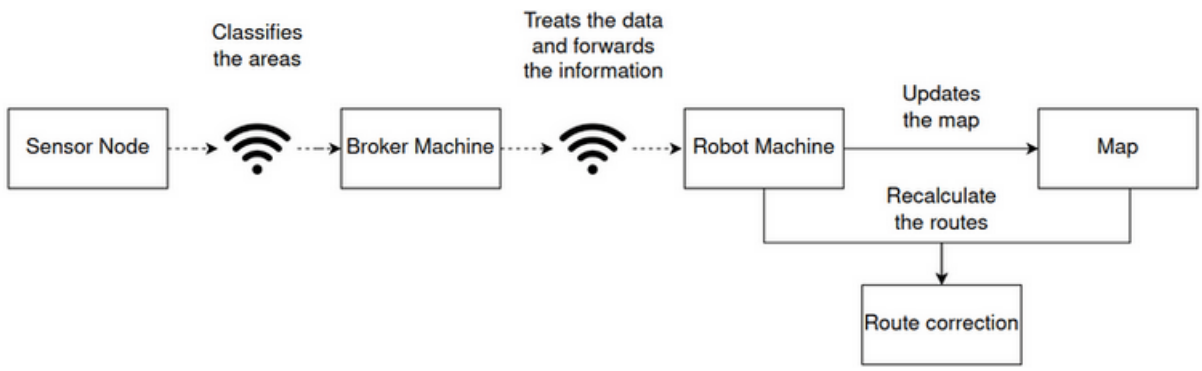

Figure 3.2: Information flow of the system.

3.3 Sensor Modules

The primary purpose of the sensor modules is to identify the danger and classify the area. The nodes comprise the MCU, the sensors, and the power unit. The firmware flashed into the MCU controls all the node's behavior, from collecting the sensory data to classifying the area and sending the information to the broker. Since the project's purpose is to validate the proposed architecture, the sensor chosen was the ultrasonic range sensor HC-SR04. This sensor can detect obstacles within a range of about 4 meters and at a 15-degree angle. It is a low-cost module and fits perfectly in the context of this

dissertation [30]. The sensor will be used only as an object detector. The area is classified as *"blocked"* if it perceives an obstacle and as *"clear"* if no obstacle is perceived. Figure 3.3 shows the sensor and the way it operates.

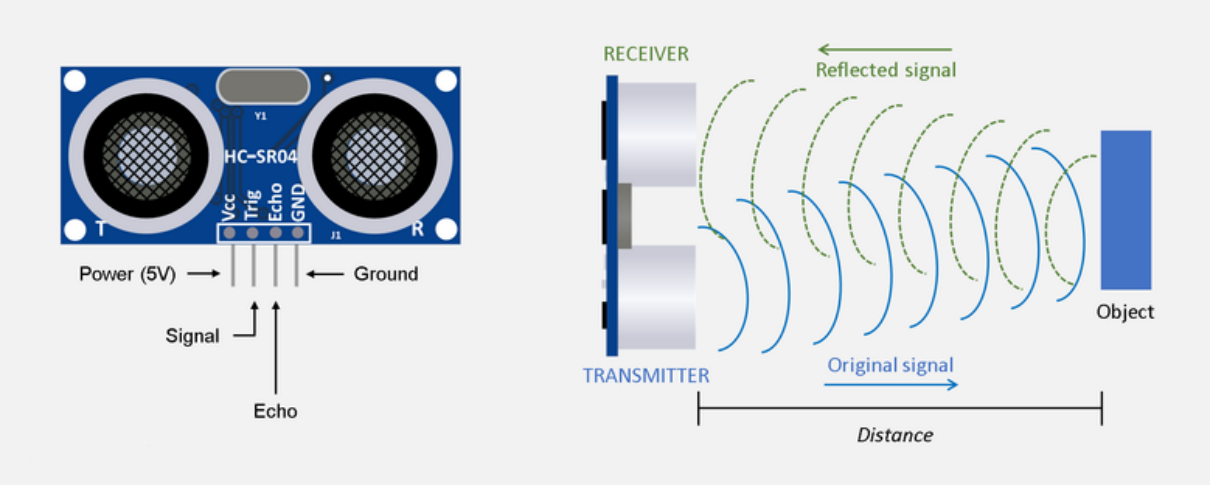

Figure 3.3: HC-RS04 and its functioning principle [31].

The code was developed in *Visual Studio Code* with the *PlatformIO* project extension, which is used for flashing the code into the Wemos D1 Mini through the *Arduino Framework* [32]. The firmware first connects to the wireless network that covers the whole floor of the building. A library called *ESP8266WiFi* was used to achieve that. That network is used to communicate with the broker and the robot. After the connection, the firmware uses an MQTT library to create the client, the entity that will publish into the area's MQTT topic. After it establishes the connection, the sensor starts its operation.

The *NewPing* library is deployed to control the HC-RS04 sensor. This library deals with the low-level aspects of the sensor, triggering the transmitter and reading the receiver determining the distance in centimeters from the obstacle. It uses the time of flight of the ultrasound wave. The maximum range of this sensor is four meters but it can be adjusted. Each node had adjustments regarding this range according to the characteristics of the monitored area. The process for that adjustment was carried with tests on site. Each module was placed on the site and readings were carried to find the range in which

the static obstacles would not interfere on the readings but the dynamic obstacles were detected. That experiment was executed for all modules generating different range values for each area situation.

The module HC-RS04 is internally controlled by an integrated circuit which is externally triggered by the *New Ping* library. This library sets the "Trig" pin of the module as a digital high for ten milliseconds, then, it is set to digital low. That activates the module's controller. The library also starts a timer once the internal controller is triggered and the transmitter is activated with a set of eight pulses. After starting the timer, the library starts to read the receiver sensor, accessed through the "Echo" pin on the module. Once it gets the signal back and the state of the pin is modified, the timer is stopped and the time of flight is computed. After that, it calculates the distance traveled using the velocity of the sound wave. The library implements the calculation in centimeters and in inches, but since we are only using it to detection and not distance measuring, it did not make difference which unit is chosen, this work used centimeters [33].

For better results and a minimal redundancy of the readings, two sensors were placed on the nodes. When both sensors complete the reading, they publish into the topic *"area x"* (x is the area number), either *"blocked"* or *"clear"*. It waits for 50 milliseconds and restarts the readings. Figure 3.4 presents the flowchart of the firmware developed. The circuit's schematic will be presented in Chapter 5.

The modules are going to be displayed in the building in a manner to cover the usually most busy corridors of the ESTIG. The proposal is that the nodes be placed on the walls pointing to the middle of the corridor. Since there are corridors with different widths, the sensor node will have to be adjusted for each. Besides that, the system must know the exact point on the robot's map where the nodes are placed. That way, it knows where each danger area has to be created on the map. This information is crucial and will be used by the broker machine and the robot.

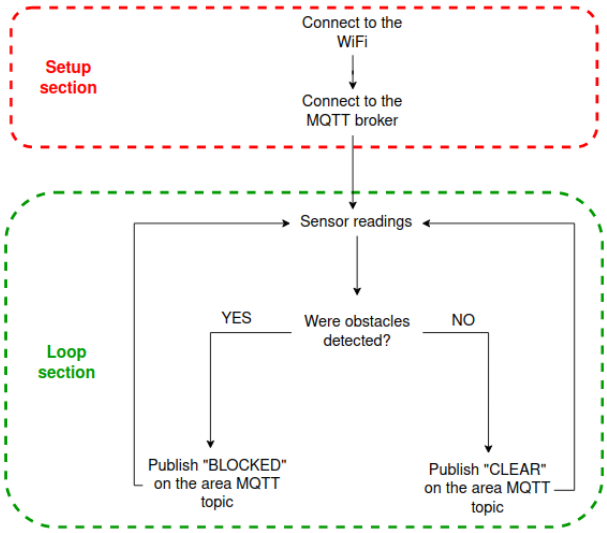

Figure 3.4: Firmware flowchart.

3.4 MQTT Broker

As explained in Chapter 2, MQTT needs a broker to control the message flow. For this work, the Mosquitto broker is perfect, it is open source, and lightweight, very suitable for low-power systems such as IoT and WSNs [34][35]. Even though the broker chosen is light, it does impose some computational load on the system. Since the robot system already has a heavy load, this broker is hosted in another machine hooked on the network, relieving the burden on the robot's machine. For that purpose, a Raspberry Pi (Raspi) running the Raspbian was deployed [36]. This Raspi is responsible for hosting the broker, receiving the messages on the area topics, analyzing them, and sending the info to the robot. Therefore it will act as the broker and client in the MQTT environment.

All that process is managed by a Python3 script developed. The *Paho-MQTT* library is used in the developed code. At first, it connects to the broker. After that, the script subscribes to every area topic. After the sensor nodes publish a message on an area topic, the script is called through an interruption routine called *"on_message"*. This procedure, in turn, calls another function called *"my_publish"*. This function receives as parameters the area of the last received message and the state of that area. After that, the method counts if the last five messages classified that area with the same state, so if the last

five messages classified the area as *"blocked"* the algorithm publishes on the WSN MQTT topic on which the robot will be subscribed.

The count starts every time there is a change in the area's state, so it will only publish when it changes the state, from *"clear"* to *"blocked"* or vice versa. Figure 3.5 show the flowchart of the process. The payload of the MQTT messages is composed of the monitored area's number and the X and Y coordinates. If the area is *"blocked"*, the X and Y will be the position of the sensor on the map. If the classification is *"clear"* the value -999 will be written for both coordinates. That is a non existing number on the maps coordinates. It is necessary for the update on the new layer (the next section will clarify that) and to make possible relocation of the sensor nodes if needed.

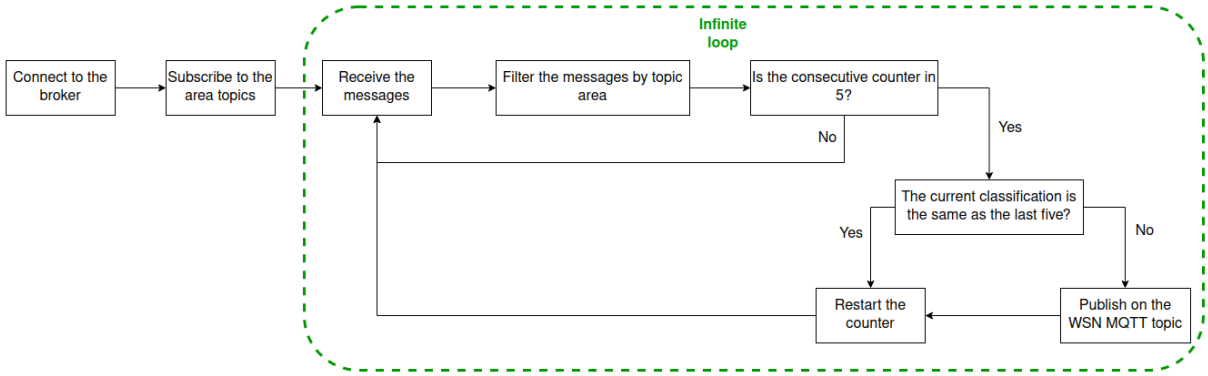

Figure 3.5: Python script running on the broker machine flowchart.

3.5 Robot System

At this point, the system process reaches its final section. Here, the danger has been sensed, the area classified, the classification sent to the central broker node, and the area point in the map sent to the robot. All that exchange of information between machines was performed with MQTT through a WiFi network. Now the robot has to receive the information, process it, and place the danger where the sensor nodes are detected. To perform those actions, two Python3 scripts were developed. Once again, the *Paho-MQTT*

library was deployed to deal with the MQTT processes. Besides, other Python libraries were employed to integrate the incoming information with the ROS environment. Aside from these two scripts, a ROS plug-in was added to the Navigation Stack. This plug-in is the one that creates the extra costmap layer where the wireless sensor information is going to be placed. It is developed in C++, and was adapted from the general code explained in the ROS-WIKI article called *"Adding a New Layer"* [21]. This adaptation (explained further ahead) allows the layer to receive information from ROS topics and insert it into the costmap. Figure 3.6 presents the essential software integrated for the architecture to work. The blue dotted marks highlight the development performed in this project scope. The orange delimiter shows the ROS Navigation Stack's regular standard routine, not developed by this work. Green and pink separators identify the Python and C++ developments, respectively. The red dots represent the robot system that comprises all the other blocks. The following subsections will describe in more detail each block separately.

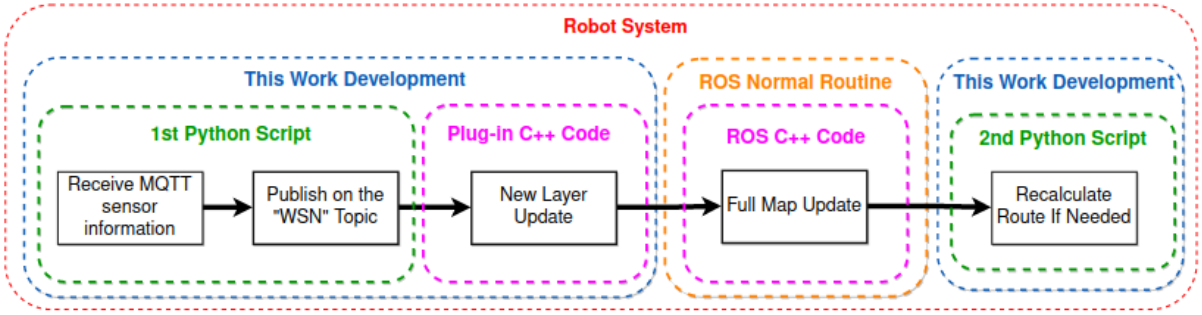

Figure 3.6: Flowchart of the proposed system running on the robot.

3.5.1 First Python Script

The first Python script works as a bridge from the MQTT environment to the ROS environment. It is responsible for receiving the messages from the broker, organizing them into a data structure, and publish on the wsn ROS topic. Initially, it connects to the MQTT broker and subscribes to the WSN MQTT topic. After that, an interruption

routine is called every time a new message is published on the topic. This routine decodes the message into UTF-8 format and casts the message into a string. Then, another function is called. This function splits the string and filters the area number and the X and Y coordinates received. Once it has identified the area and the pair of values, it loads an array with the new information. Each monitored area is represented in this array by two cells, one cell is the X coordinate, and the other is the Y. *Area 1* information, for example, is stored in the first two cells, as shown in Figure 3.7. So, after filtering the area number, the algorithm loads the respective array cells and publishes the array on the wsn ROS topic.

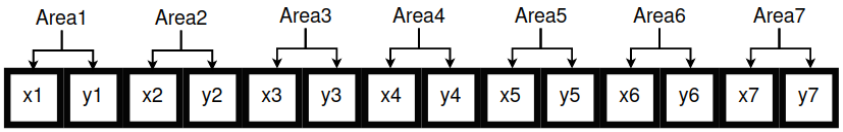

Figure 3.7: Array created for updating the layer.

Figure 3.8 presents the flowchart of the first Python script. Note that it runs in a loop but is only triggered by a message from the WSN.

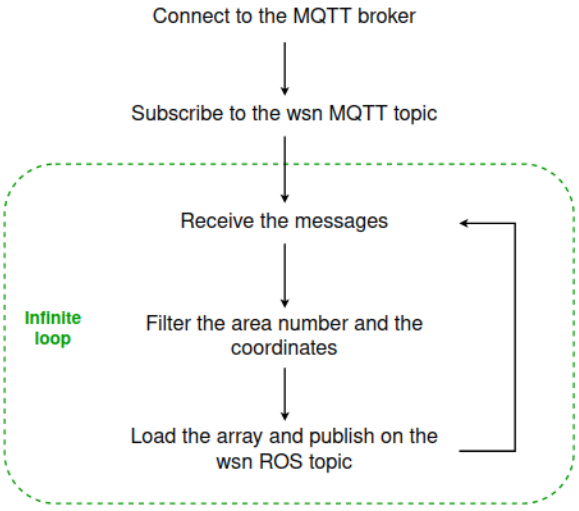

Figure 3.8: Flowchart of the first Python script.

3.5.2 Plug-In C++ Code

As mentioned before, the newly added costmap layer is a plug-in in the ROS environment, which means it is a class that the ROS system can reach, but it builds and works independently of the central core software. This plug-in is called by the Navigation Stack that uses its functions. It is a package that is registered in the system as a plug-in. Once the package with the plug-in designation is created, building the code makes it available for ROS to access. Every costmap layer has three principal class functions, the initialization, the update bounds, and the update costs. When updating the map with a fixed rate set on the parameters in the launch files, these routines are called from the Navigation Stack. The initialization and the *"Update Costs"* function were adjusted (explained ahead). The *"Update Bounds"* was not changed from the standard format. It is responsible for checking the area where changes were made and delimiting to update only the changed map cells. Besides the adjustments on the initialization and update costs functions, another function was created inside the class. It is a callback function that will be called every time the first Python script publishes on the *wsn* ROS topic and receives the array with the monitored area's information.

To summarize, inside the new layer class, a ROS node subscribing to the *wsn* ROS topic was created. This callback function is responsible for receiving the data array and storing this array for the *"Update Costs"* method to access it. That is the final step of the information traveling. This step is the one that enables the wireless node to be a data source for the costmaps through the reading of the wsn ROS node and making the information array available inside the layer class.

When the function *"Update Costs"* of the new layer is called, it travels through the array looking for valid sensor points. If there is a valid set of coordinates for a specific area, the cell costs must be set as obstacles. The possible costs are free, unknown, and lethal. When there are invalid X and Y values for an area (-999, a constant value nonexistent in the map), the method does not set any cost values for that zone, and therefore the layer does not have any information about those cells. When that happens, the Navigation

Stack does not consider those cells without information, meaning that the cell costs will stay the same as they are coming from the other layers. The other step the function takes is expanding that set X and Y to an area that covers the danger zone monitored by the wireless node. That is, it sets the costs of the cells around the point of the sensor to match the surveilled area. Now, the information about the danger zone is already on the map, and the robot can avoid it.

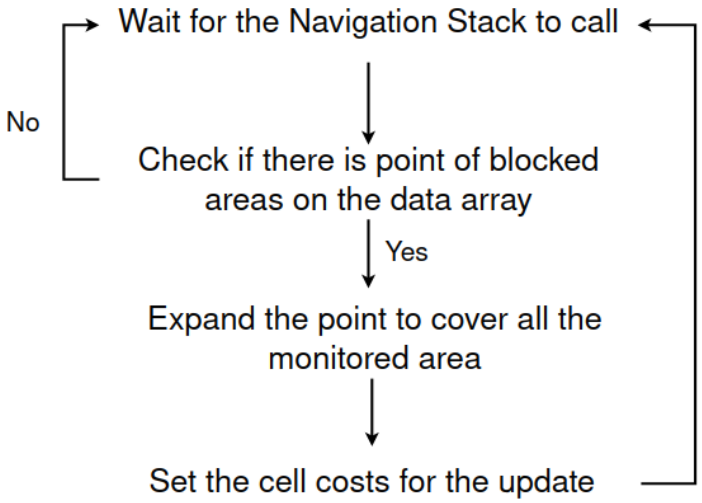

Figure 3.9: Flowchart of the *"Update Costs"* function.

For better understanding, Algorithm 1 presents the *"Update Costs"* logic of the *New Layer*. It is a simplified pseudo-code representing the logic of the flowchart above.

Algorithm 1 *"Update Costs" Function*

Input: Array with the areas' information (A) and Costmap Cells.

```
1:  function UPDATE_COSTS(A, COSTMAP_CELLS)
2:      for each area do
3:          if area_x = −999 and area_y = −999 then
4:              for each area cell do Set_Cost(area_cell, FREE_COST)
5:              end for
6:          else
7:              for each area cell do Set_Cost(area_cell, LETHAL_COST)
8:              end for
9:          end if
10:     end for
11:     end
```

3.5.3 Second Python Script

Although the information has already been written into the costmap values, the recalculation of the routes is not guaranteed. If the path planner is set to make one calculation per goal, it will not adjust the route and possibly get close to the danger area because when the path was calculated, the danger could not have been yet placed on the map. To overcome that possible situation, a second Python script was developed. The idea here is to spark, only if needed, recalculations of the path to consider the danger area created, which optimizes the route with the latest information.

To achieve that, the script takes some data from ROS topics. The code subscribes to the odometry topic, the current goal topic, and the calculated path topic. Once there is an update on the wsn information, the script takes the pose of the robot from the odometry information and checks if the robot is already inside the target tolerance or not. If the robot is on the target, the method cancels the goal and stops the robot if needed. If the robot is still not on the target, the goal is resent to the path planner to recalculate the

route accounting for the newest costmap knowledge. If the goal is not reachable anymore, the algorithm will cancel it and stop the robot. Otherwise, it will execute the new route.

The method takes the current X and Y coordinates of the robot and the current goal target. Then, subtracts the robot X value from the goal X value. The same happens for the Y. That way, there are two triangle sides and it calculates the hypotenuse, a straight line from one point to the other. That is considered as the linear distance of the robot to the desired goal. The Python *math* library is used to find the shortest angular distance to the goal. It takes both angles in radians, calculates the difference, and normalizes the value between - π and π. That is necessary when the robot is already close to the goal. If any calculation returns a value higher than the tolerance set on the Navigation Stack parameters, it triggers the path planner to recalculate the route. Figure 3.10 presents the flowchart of the process explained above.

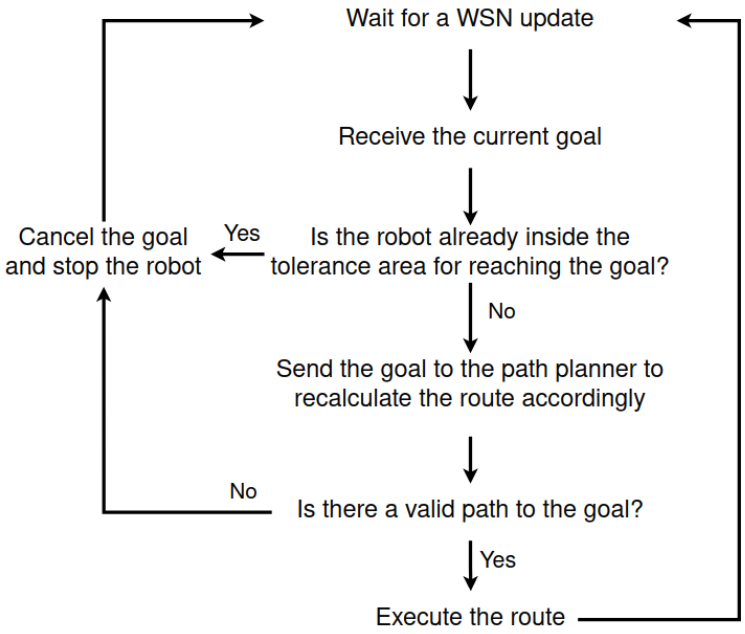

Figure 3.10: Flowchart of the second Python script.

Algorithm 2 presents a pseudo-code of the process described by the flowchart above. As mentioned before, this algorithm runs every time the WSN ROS topic updates the status of any area.

Algorithm 2 *Recalculate Route*

```
1: function RECALCULATE()
2:     Goal ← current_goal()
3:     Pose ← current_pose()
4:     Linear_diff ← Linear_Distance_Between_Pose_and_Goal()
5:     Angular_diff ← Angular_Distance_Between_Pose_and_Goal()
6:     if Linear_diff > tolerance AND Angular_diff > tolerance then
7:         Plan ← Calculate_Path()
8:         if Plan is valid then
9:             Execute_Path()
10:        else
11:            Stop_Robot()
12:        end if
13:    else
14:        Pass
15:    end if
16:    end
```

The past Sections presented the proposed system's architecture. The following Chapters will cover the implementation and tests in simulated and real environments respectively.

Chapter 4

Simulation Development

This chapter covers the simulation aspects of the work. It addresses the Gazebo simulator, the TurtleBot 3 (TB3) robot model simulated, the creation of the environment, the inputs and outputs, and how the controller deals with the tests. The simulation played a crucial part in the development of this work. It helped us learn and understand the ROS environment, validate the proposed architecture, and adjust many aspects of the Navigation Stack without needing to deploy the real robot. The simulations were automatized to perform several runs without human supervision.

4.1 Gazebo and the TB3

The simulation environment was developed in the Gazebo simulator with the TB3 as the robot model. Gazebo is an open-source project started at the University of Southern California around 2002 [37][38]. After the ROS creation, Gazebo was tightly integrated with the new framework (ROS) by the developers in the Willow Garage Project and nowadays is maintained by the Open Robotics Foundation. The critical concept of Gazebo is to allow rapid development and testing for robotic applications. The high integration with the ROS framework made it the most popular simulator in various robotics areas, for example, AMRs and legged robots. Gazebo offers multiple sensor libraries, the creation or importation of environments, and the possibility to use Universal Robotic Description

Files (URDF) to import robot models. All these features contributed to its popularity [39]. Figure 4.1 presents a chart with the number of citations of robotic simulators gathered from Google Scholar. It shows Gazebo has the most citations between 2016 and 2020.

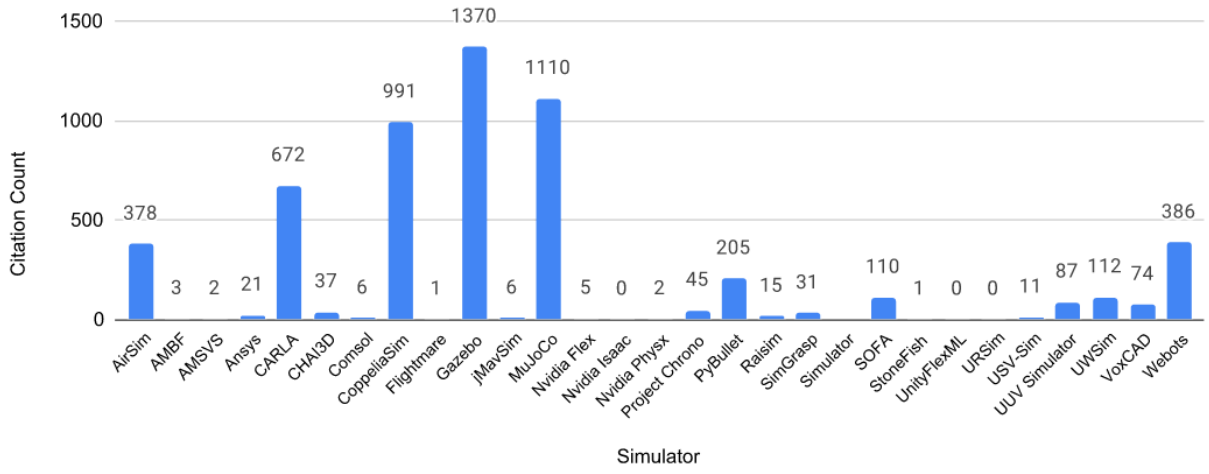

Figure 4.1: Publications citing simulators between 2016 and 2020 [39].

Gazebo was chosen because of the aforementioned tight integration with ROS and the intuitive user platform. The creation of the environment will be described in the next section.

As mentioned before, the robotic platform used in the simulation is the TB3, an open-source and collaborative effort led by ROBOTIS, the Open Robotics organization, other partners, and research centers. This work will be handled with the TB3 Burger model. However, the simulation can be performed with all three models since the files needed are available to the community by the developers [40]. The TB3 is not only a ROS standard platform but the most popular one, having an active community of developers facilitating educational projects and research [41]. Figure 4.2 shows the available models. All the TB3 models are differential drive robots, which means the movement is based on two separated wheels with velocities applied to each one, so if both receive the same values, it moves in a straight line (linear velocities only). If the values are different, the robot gains angular velocity.

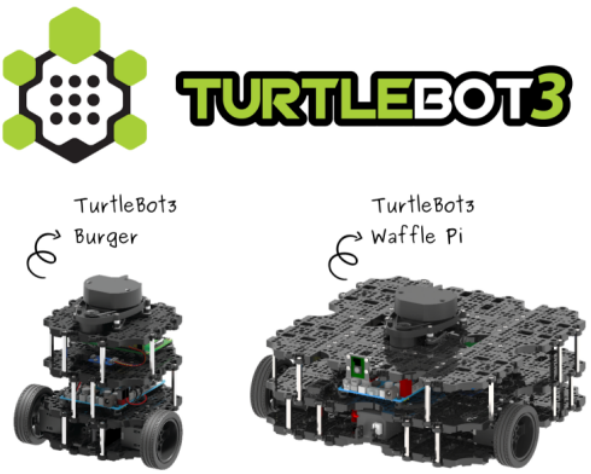

Figure 4.2: TB3 models [41].

All the configuration files to launch the TB3 simulation in Gazebo with the ROS navigation stack are available on the ROBOTIS GitHub found in [40]. There, one may find the URDF files, YAML configuration files, launch files, and the instructions to launch the TB3 in the Gazebo environment. Figure 4.3 presents the TB3 in the Gazebo simulation environment. Once the robot model and the ROS files were downloaded and prepared, the creation of the simulated scenario started.

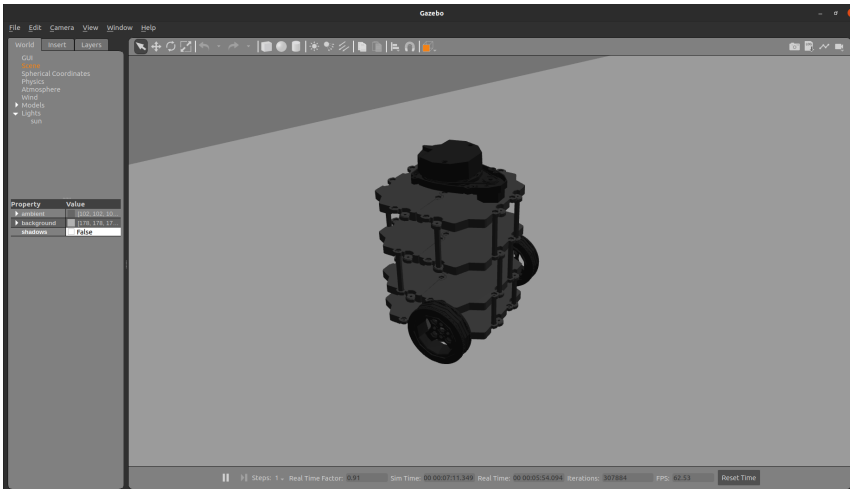

Figure 4.3: TB3 in a simulated environment.

Figures 4.4 and 4.5 present some characteristics of the TB3 Burger model hardware.

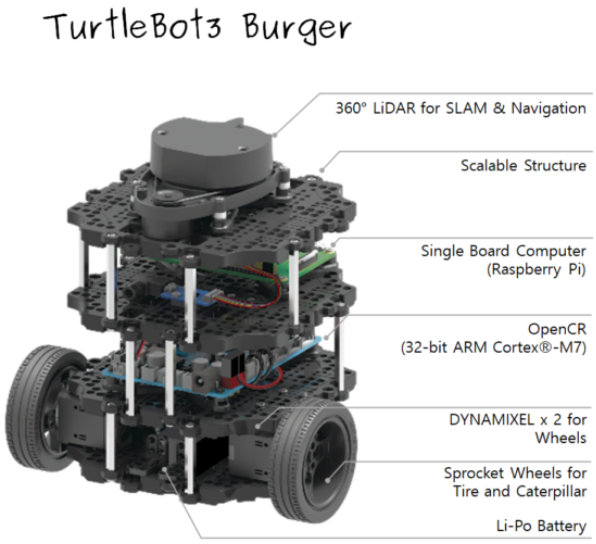

Figure 4.4: TB3 standard components [41].

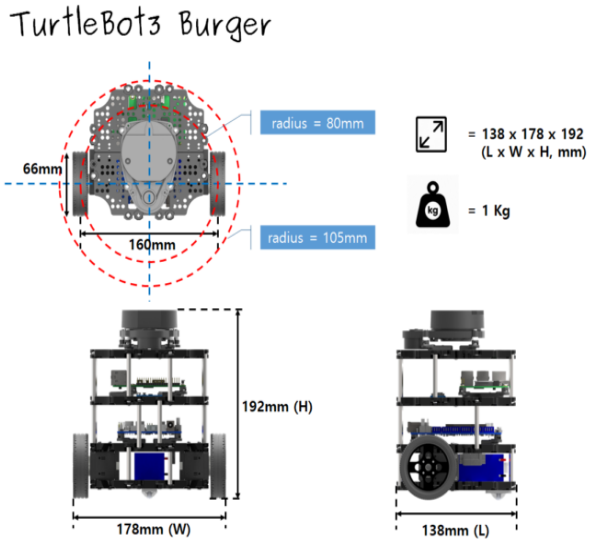

Figure 4.5: TB3 dimensions [41].

4.2 Creating the Simulated World

As mentioned before, the site of the study will be the first floor of the ESTIG building. That noted, the scenario in Gazebo to launch the TB3 with the navigation stack will simulate that place. For that, a blueprint of the level was used. Figure 4.6 shows the whole design of the floor. In the future, there is a project where an AMR will guide visitors. This work is in that context.

The idea is for the robot to navigate through the corridors where there are classrooms and professor's chambers. Those rooms are located in the lower part of the image. The right upper side is the library and the bar, not simulated by this work. The essential sections of the blueprint are the corridors, where the chambers and classrooms are. Because of that, the image was edited to have only the places the robot will travel through. The AMR is not supposed to go inside the rooms.

Using the GNU Image Manipulator Program (GIMP), an image editor, the parts not involved in the simulations (inside rooms for example) were excluded. Figure 4.7 shows the final image used to create the simulated world. That image guided the construction of the building corridors in Gazebo and the static map for the ROS navigation stack.

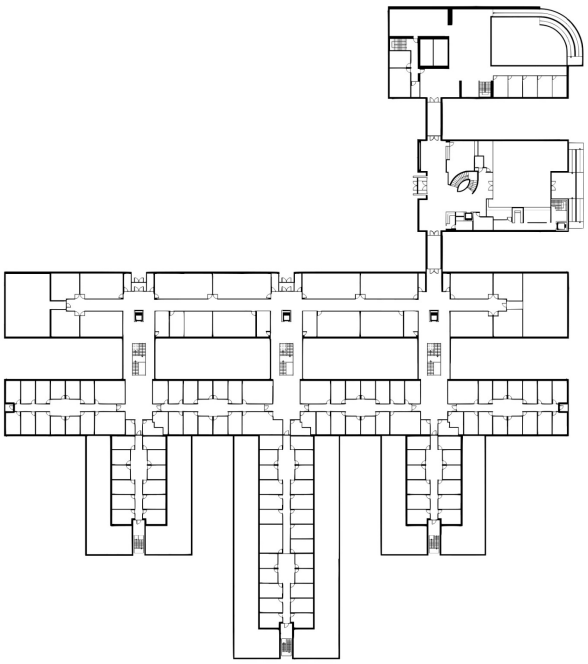

Figure 4.6: ESTIG blueprint.

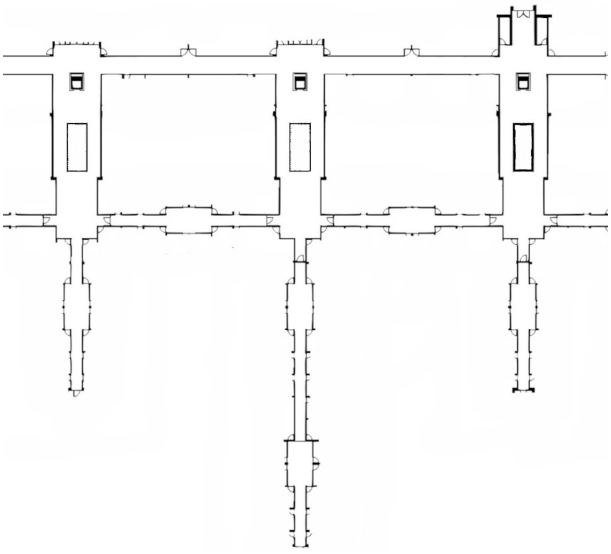

Figure 4.7: ESTIG blueprint only with corridors.

Once defined the site of the study, the simulated world began to take shape. Gazebo has a handy tool called *"Building Editor"*. It allows the users to create walls and add features such as windows, doors, and stairs. The tool presents three types of building styles

(textures): wood, tiles, and bricks. What makes this tool remarkable is the possibility of importing a blueprint. That option allowed us to import the image in Figure 4.7 with the correct resolution and to respect the actual size of the building. After importing the image, it is placed in the background of the working window, making it easy to create the walls following the blueprint of the floor, as shown in Figure 4.8.

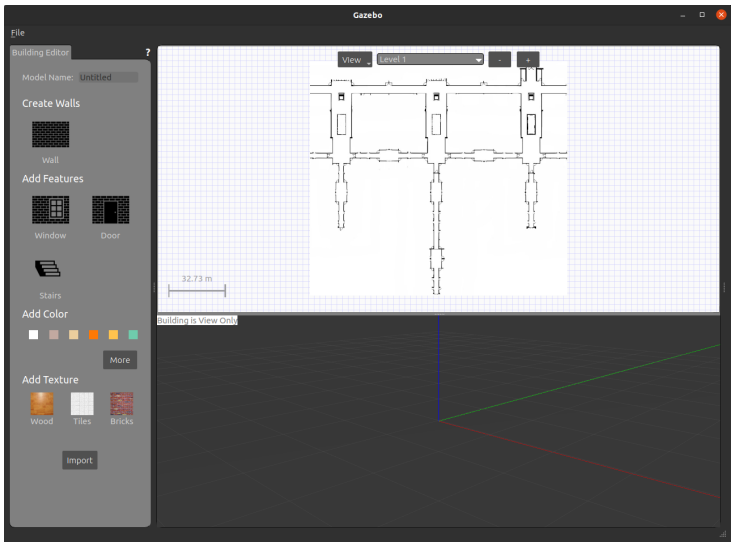

Figure 4.8: Building editor with the image imported.

At this point, the walls were designed on top of the blueprint. The result was very accurate and is shown in Figure 4.9. As one can see, it only simulates the walls. Other details, such as doors, were considered closed, and the windows were considered out of the robot's "vision". Adding more details would only increase the heaviness of the simulation and not produce substantial gains that justified that. It is also important to note that the scale is one-to-one, which makes the simulated environment very close to the real one.

After completing the design of the building, Gazebo generates an XML file containing all the information about the built world. That file is used in the ROS launch files to deploy the robot inside the building. These launch files are the ones that start the processes in ROS. They are responsible for linking the TB3 configuration files (description and ROS parameters) to the Gazebo building, as well as triggering the ROS nodes to initiate.

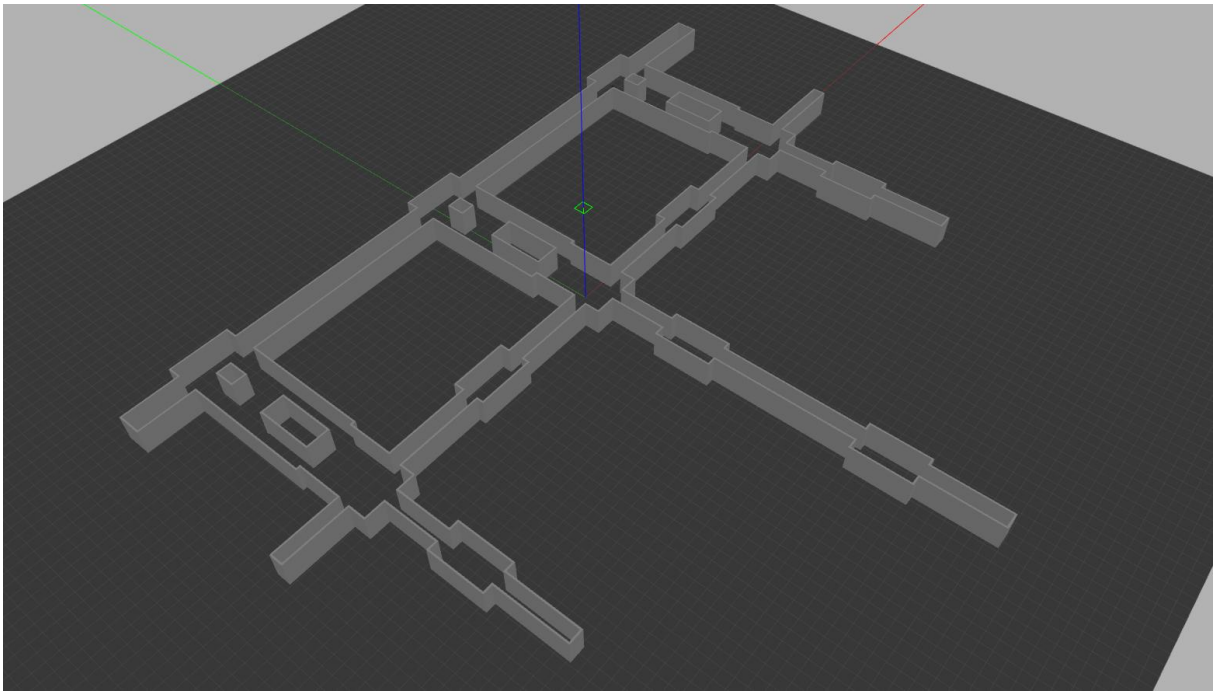

Figure 4.9: Simulated ESTIG building.

4.3 Simulation Experiment

After the simulation environment and the robot model were ready, a Python script was developed to conduct the simulation tests automatically. The idea is that this script controls some aspects of the simulation and gathers and stores data for later analysis. The experiments will consist of the robot traveling through three routes previously chosen. These routes will simulate the paths a person usually may take around the building. Besides that, seven zones were chosen to be monitored to detect obstacles. All these areas were chosen considering the areas students and professors usually stand while in conversation or waiting for classes. Figure 4.10 presents the stopping points of the routes, named from "A" to "D", the monitored zones "Z1" to "Z7", the central Raspi broker, and the sensor nodes. Note also the red collared area. It shows the approximate coverage area of the "Z4".

As mentioned before, the robot will perform three routes in the experiments. The first

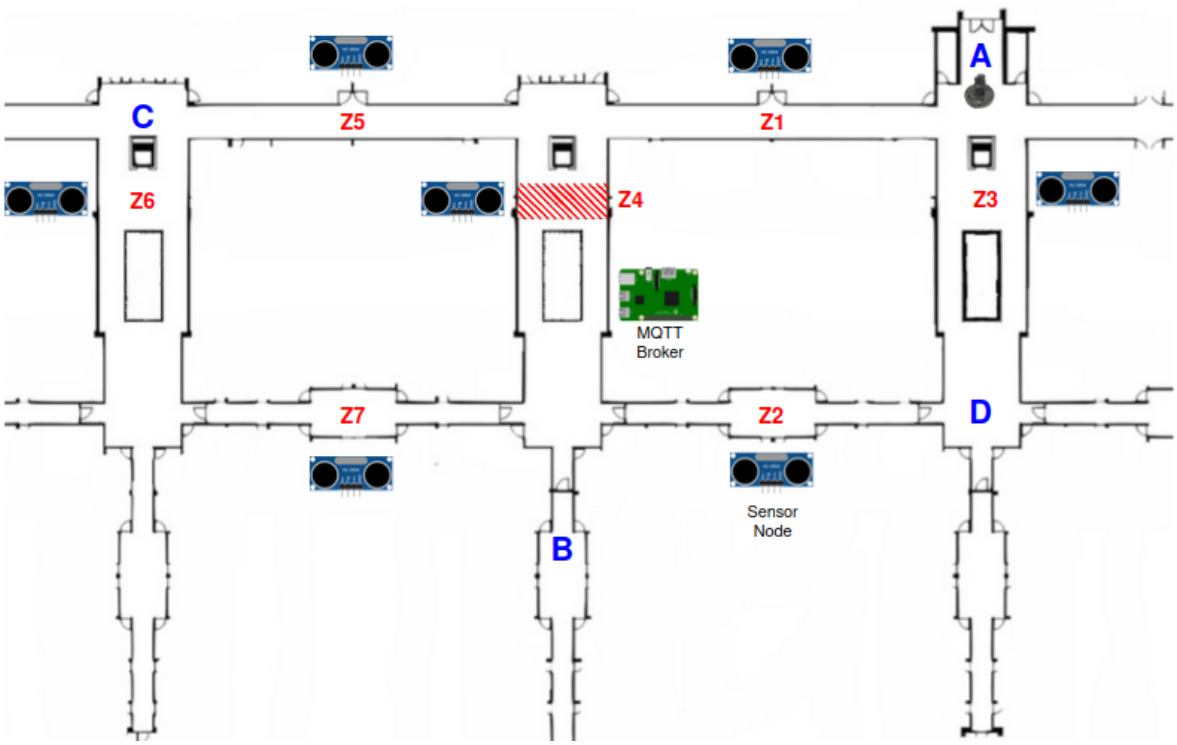

Figure 4.10: Experiments scenario.

route starts at point "A" and finishes at point "B". The second goes from point "B" to "C", and the third "C" to "D". For each route, obstacles are raffled into the monitored zones to block the passage. For the first route, the method draws obstacles either in zone one or four. For the second, the raffle is between zones four and seven. The draw for the third route is for zones five, one, or three. Note that the obstacles will only appear inside the monitored areas, that is because if the obstacles appear outside the monitored zones, the behavior of the robot will remain the same, so for the experiment, the focus is on the situation where the system changes the behavior of the AMR to find its possible gains. The experiment follows the flowchart presented in Figure 4.11.

The process described above will be called a run of the simulation. Therefore, a run is considered the completion of the three routes. The simulation experiment will perform fifty runs for both configurations, the one with the proposed monitoring architecture and optimized parametrization, called `Full Setup`, and the one without the system and

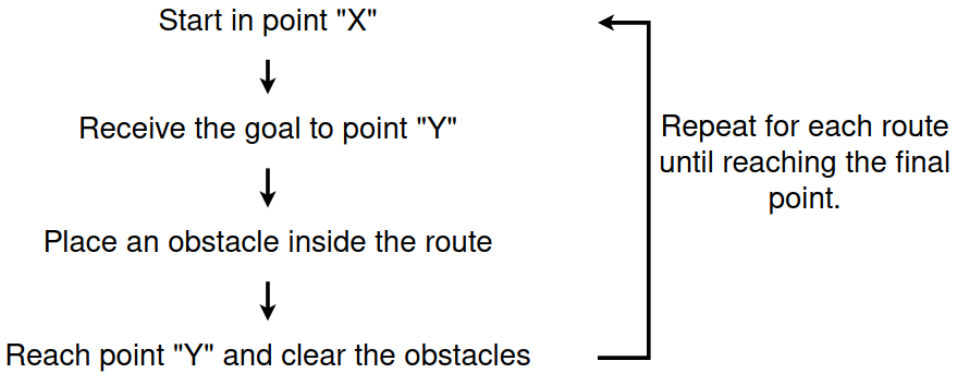

Figure 4.11: Experiment runs flowchart.

standard parameters, called default setup. For that number of runs, it is critical to have them running autonomously without supervision. To achieve that, the Python controller script was developed.

4.4 Main Script Controller

The experiment was held to compare the AMR behavior, percentage of CPU consumption, and memory usage for the main ROS processes running for navigation. Therefore, this Python script must gather those data. Besides that, it uses ROS services to spawn obstacles, send navigation goals, and ROS topics to receive and publish information.

4.4.1 Inputs and Outputs

This algorithm takes in some arguments that set directions for the runs. They are listed below and explained after.

- Setup type (-t)
- Number of runs (-n)
- Rviz (-rv)
- Gazebo client (-gz)

- Analysis (-a)

The *"setup type"* is the argument for setting which configuration will run, default or `Full Setup`. There is another possibility to run both setups one after the other. That is used to call the ROS launch files that trigger the simulations to begin. The *"number of runs"* is a counter that will keep restarting the routes until it reaches that value. Each setup was run 50 times. The *"Rviz"* is a flag that starts the visualization tool of ROS. That is mainly used for debugging. The experiment does not use that tool since it implies a high computational burden (graphical interface). The *"Gazebo"* argument is also a flag that may launch the graphical simulation window. Similar to the Rviz, it represents a high computational load and was not set for the experiments. The analysis argument is responsible for calling another script process after the simulation if needed. It was advantageous when constructing the experiment since not all the runs were supposed to be analyzed (for example, when debugging). Stopping the analysis saved a good amount of time. This analysis takes all the data and organizes the results.

The script outputs are shown in the file tree in Figure 4.12. As one can see, text files store the data gathered. Each route has its information inside corresponding folders. The distance and time are placed out of those folders because they store all routes' distances and time spent, making it easier for later analysis. Note that the default and `Full` setups have separated text files containing the memory and CPU percentage for each route performed. Besides that, it stores the parameters passed to the script and the Navigation parameters set for the TB3. The folders amcl, map_server, move_base, and rosmaster correspond to the main ROS processes analyzed. At this point, they are empty. The after-treatment will generate files placed inside those folders for us to have each process information separately. The distance is recorded in meters and the time in seconds.

4.4.2 Algorithm Flowchart

The control script was designed with the primary process initially called to create the files presented in Figure 4.12, launch the simulation, and finish the sub-processes to restart

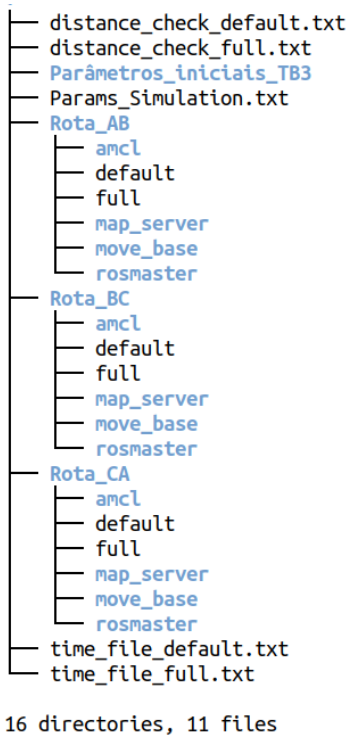

Figure 4.12: Files generated by the Python script.

each run. This main procedure triggers three child routines using the Python library called *multiprocessing*. This library has a method named *"Process"*, which launches functions in a parallel process. As mentioned before, three algorithms are called that way. They will be explained in the following subsections. They were named *"Goal Check"*, *"Spawn and Goals"*, and *"Distance Check"*. Figure 4.13 present the flowchart of the main process.

Spawn and Goals Process

The *"Spawn and Goal"* process is responsible for spawning and deleting obstacles, sending navigation goals, timing the routes, and publishing into the WSN ROS topic information about the environment. This procedure starts after the launch files are deployed, and the AMR is ready to navigate.

At first, the algorithm raffles the area where the obstacle will appear, blocking the passage. It is done for each route. For example, for route "A" to "B", the monitored zones where the AMR may pass are one and four, so the script randomly chooses one of them

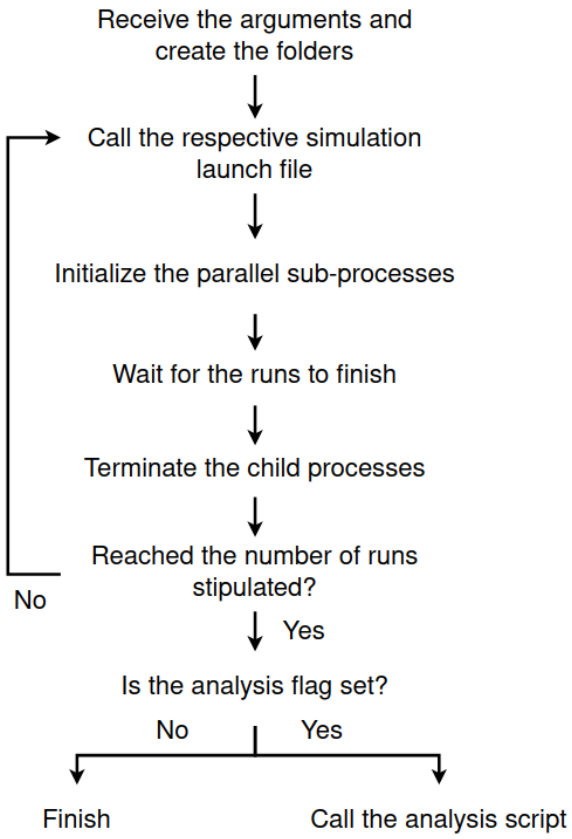

Figure 4.13: Flowchart of the Python controller script.

to spawn the obstacles inside. After that, the algorithm launches a sub-process that runs a Linux *Top* command. This command displays the system's information, such as %CPU consumption and percentage of memory consumed, displayed by processes running. Those data are logged into a text file inside the route folder, shown in Figure 4.12. *Top* was called to perform the readings once every two seconds. After this data gathering is initiated, the navigation is started alongside a reading of the system time. The process waits until a flog of goal reached is set. Once that flag is set, the method reads the time, stops the Top process, and clears the obstacles for the following route to begin. Note that the time spent is calculated by subtracting the first time reading from the second. This whole algorithm is repeated for every route. Figure 4.14 shows the obstacles used to block the passage.

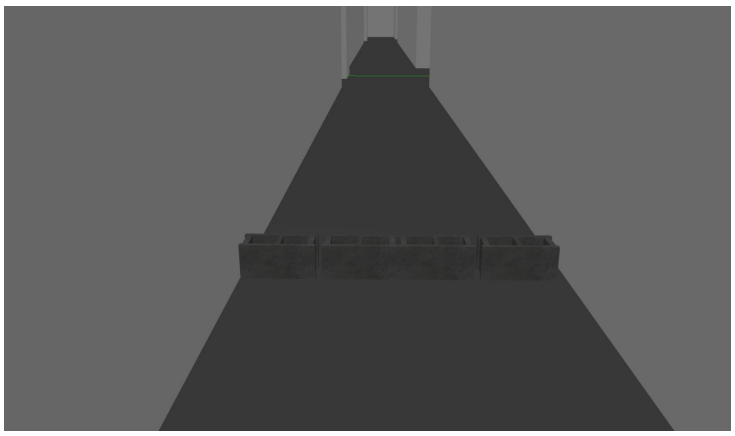

Figure 4.14: Bricks as obstacles blocking the pathway.

Goal Check Process

As one can see, the procedure described above depends on the script to know when the AMR reached the goal. To perform that, the *"Goal Check Process"* was created. The Navigation Stack has a topic called *status*. This topic receives the status of the last goal sent to the Navigation Stack. The method to check if the goal was reached or not is trivial in the ROS environment. The process subscribes to the status topic, and when it receives a message saying "Goal reached", it sets a flag to which the *"Spawn and Goals"* has access. Although simple, this process is crucial for controlling the experiments since it is how the controller knows when the AMR has reached each goal and behaves accordingly.

Distance Check Process

This process is not as trivial as the *"Goal Check"* one. That is the method responsible for calculating the distance traveled. This algorithm listens to the odometry topic, where the Navigation Stack publishes the poses based on the robot's odometry data. Every time the pose is updated, this process takes the coordinates and stores them in a list. After the robot reaches the goal, the method calculates the Euclidean distance between each point registered and sums all the distances, which gives the total length traveled. The path is broken down into small ones, and the sum of those little displacements is the value needed.

4.5 Navigation Stack TB3's Configuration

The Navigation Stack, as mentioned before, has a set of parameters that are key for the conduct of the AMR. The list below shows some of the most critical YAML files containing parameter values to initialize the navigation. The launch files point to these files and gather the parameters. Some parameters will be explained after the listing.

- *global_planner_params.yaml*
- *local_planner_params.yaml*
- *common_costmap_params.yaml*
- *local_costmap_params.yaml*
- *global_costmap_params.yaml*
- *move_base_params.yaml*
- *map.yaml*

The global planner YAML file sets parameters for the global planner to use. The most important one is the algorithm to calculate the paths. In this work, the *A Star* method was chosen, explained in Chapter 2. The Navigation Stack has the option to use the *A Star*, the *Dijkstra*, and the *Carrot Planner* by default. Since the *A Star* is an evolution of *Dijkstra*, and the *Carrot Planner* is too simple (only calculates straight line paths), the *A Star* was the best option to choose.

The same happens to the local planner file, but this one has more important parameters. Here the maximum linear velocities (0.3 m/s) and accelerations (2.5 m/s^2) are set, the angular values, respectively, (2.75 m/s) and (3.2 m/s^2). This file sets the goal tolerance to 10cm for linear distance and 0.17 $radians$ for angular. These values were set considering the capabilities of the TB3. Aside those, the *local_planner_params.yaml* file receives the simulation period for the *DWA* local planner, which was set to 1.5 seconds to reduce CPU burden (default was 1.7), and the number of sampled velocities set to 25

linear and 35 angular (explained in Chapter 2). The *DWA* planner is one of the options the Navigation Stack has implemented and is available for users. The other option is the *Time Elastic Band* algorithm. The *DWA* was chosen because it showed a smoother behavior in the simulation.

The common costmap configuration file set values that will be used for both costmaps, global (which stores information from the static map) and local (which stores information from the sensors), the sum of both costmaps is used for the robot to navigate. In the common_costmpap_params.yaml, the measures of the robot are set, which gives the robot knowledge of its size.

The update frequency of the costmaps, the sources of information, the time tolerance, and the reference frame are set in *global_costmap_params* and *local_costmap_params*. The update frequency for the global costmap is set to $2Hz$, and the local to $5Hz$. The data received is considered if it was produced in the last second (older data is thrown away). The costmap layers are set in the global parameters file. Here, the new WSN layer was inserted.

The *move_base_params.yaml* sets the parameter of the main controller node called *Move Base*. It sets the control frequency (default is $10Hz$, but was adjusted to $5Hz$ for reduction of CPU burden), and the path planner frequency, which by default is $10Hz$. In the *Full Setup* is set to zero, meaning it will only calculate the route once. Recalculations will be triggered only if needed. All these settings of parameters were chosen considering the work in [15], mentioned in Chapter 2. All the files described above, containing all the parameters, may be found on the Gitlab repository of the project [42].

Chapter 5

Real Scenario Development

After the simulated experiments, the developments for the real scenario testing was started. This chapter will cover the work sustained to deploy and integrate the subsystems in the study site. At this point, all the architecture was already designed. Now, the hardware will be put together for validation in the study site. The idea of this work is to propose minimal viable hardware that allows us to develop, deploy, and conclude rapidly. Also, the idea is to have a low-cost wireless sensor module.

The chapter will present the IRobot's Roomba© robotic platform with its adapted modules, the broker hosted in a Raspi, the sensor module, the experiment, and the parameters of the navigation stack for both setups (`full` and default).

5.1 Roomba©

The robotic platform used in simulations was the TB3, as presented in Chapter 4. The first models of the TurtleBot were built upon a house cleaning robot called Roomba©. Because of that, all the physical and dynamic characteristics of the TB3 are very close to the Roomba© robot. The differences, at first, were only extra sensors added to the Turtlebot platform. The TB3 has evolved into a different structure but maintained the size and the main physical characteristics. The Roomba© platform is a decent candidate for comparison with the TB3.

Roomba© is a differential drive robotic platform developed by an American company called iRobot©. The company is the leader in consumer robots, having offices in Europe, America, and Asia. The company focuses on cleaning robots, which is the case of Roomba©. It is a vacuum cleaner [43]. Figure 5.1 present the model used in this work, the iRobot© Roomba© 600 Series.

Figure 5.1: iRobot© Roomba© model 676 [44].

This Roomba© model 676 comes with an odometry sensor, a tactile sensor in the bumper, and distance sensors in the front. For the navigation with ROS, the platform used the odometry (already on the Roomba©) and an extra Lidar sensor, presented in the next Section. For the controlling element, a Raspi was used. The system was placed together by a 3D printed structure mounted on the robot. Figure 5.2 shows the actual robot with the structure mentioned. The figure shows the three stories frame. On the first, there is the power bank, which will feed the Raspi (the robot has an exclusive battery), the second holds the Raspi, and the third, the Lidar.

The Raspi controls the robot via a USB port (Raspi) connected to the USB-Serial port of the Roomba©, as one can see in Figure 5.2 on the left lower side. All the drivers for

Figure 5.2: Roomba© with the structure for ROS controlling.

controlling the robot with ROS may be found in [45], a GitHub page containing the driver files and the instructions for deployment. These drivers are responsible for integrating the ROS ecosystem into the Open Interface Specification of the iRobot system. They translate the information published into ROS topics into the commands of the low-level robot controller. With them, one can control the robot using the ROS Navigation Stack.

The Raspi used to control the robot is a model 4B with 4GB of Random Access Memory (RAM). It runs Linux Ubuntu 20.04 with ROS Noetic. This Raspi is connected to the wireless network to receive the sensor module data through MQTT. The power bank's capacity was 10000mAh and 5 volts output.

5.2 RP Lidar

The Lidar used in this work is the RP Lidar model A1. That is a 360-degree lidar with a detection range of 12 meters and can produce eight thousand samples per second. It

has a resolution of one degree. It communicates and is powered via a USB cable, in this case, connected to the Raspi. This a sensor produced by the Chinese company Slamtec. It has open-source Software Development Kit (SDK) tools and is compatible with ROS [46]. Slamtec has a repository on GitHub with all the drivers for deploying it with ROS, as well as tutorials found in [47]. Figure 5.3 presents the RP Lidar model A1. This lidar model uses a digital signal processor to analyse the reflected signal received by its vision system. The signal is generated by infrared lasers reflected on the environment. The module outputs the distance and angle of the detected obstacles.

Figure 5.3: RP Lidar [46].

5.3 Sensor Modules

The sensor module, as mentioned before, is composed of the Wemos D1 board with ESP8266 and antenna, two ultrasonic sensors, the power bank, and the case, all shown in Figure 5.4. In the project, the sensors were kept at a distance where the readings would not interfere with the other. The idea of the system is to be discrete and small.

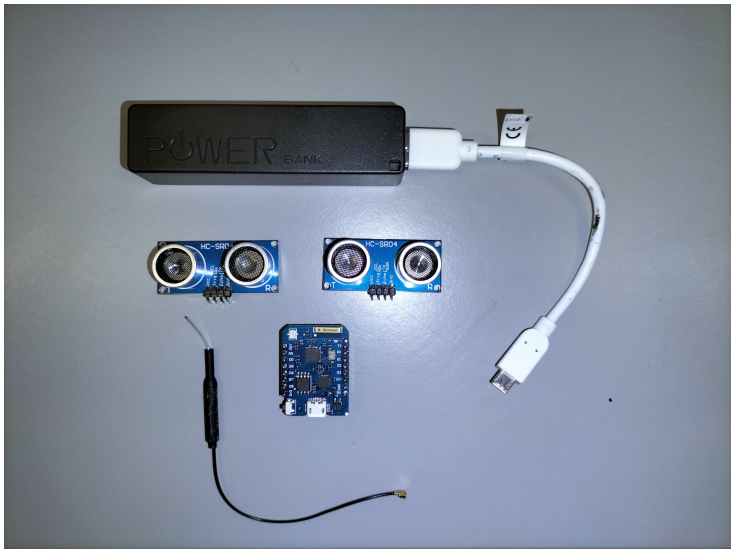

Figure 5.4: Parts of the wireless sensor node.

Figure 5.5 presents the circuit of the sensor module, with the connections between the HC-SR04 sensors and the Wemos D1 mini. As one can see, the first sonar uses the digital pins D0 and D1, and the second uses D3 and D6. A 5 volts power bank powers both sensors and the Wemos.

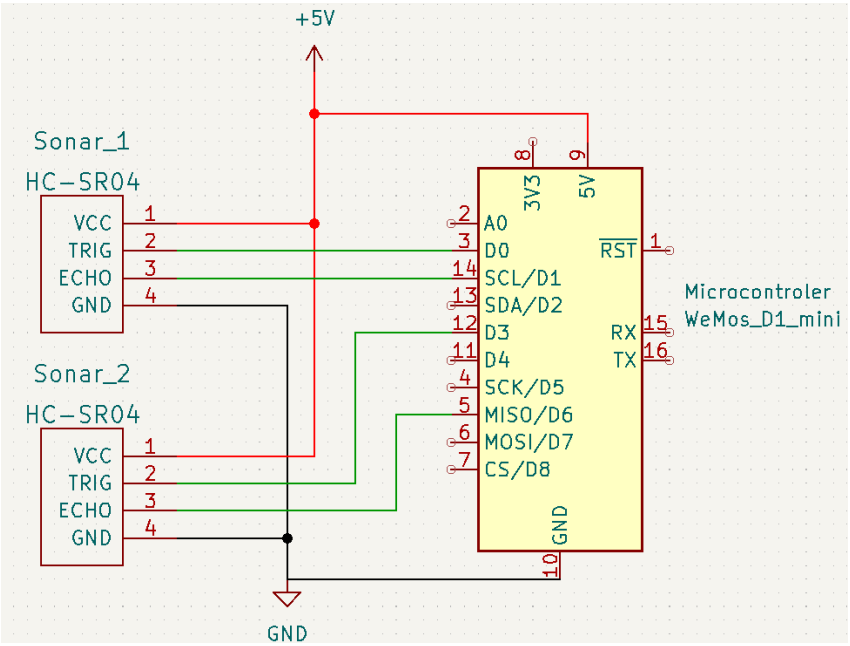

Figure 5.5: Sensor module circuit schematic designed in KiCad EDA.

5.3.1 Case

To fit these components, a case was printed in a 3D printer. As mentioned before, the case will hold the sensors at a distance apart because the readings can interfere with one another if they are too close. The project was adapted from an existing one in TinkerCAD. The model is presented in Figure 5.6.

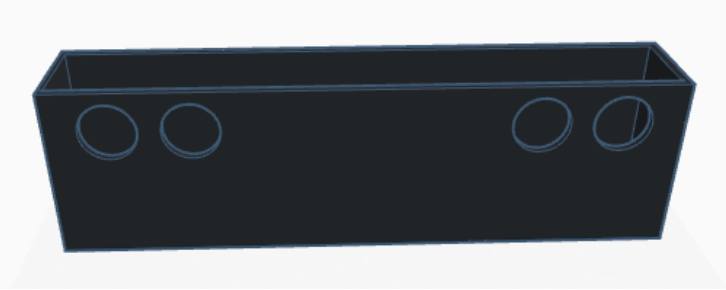

Figure 5.6: Sensor node case in 3D model.

The designed dimensions accounted for the sizes of the sensors, the Wemos, and a small power bank. The Wemos antenna was kept outside the case. Figure 5.7 presents the final version of the wireless sensor node. Seven wireless nodes were produced to monitor the seven zones.

Figure 5.7: Sensor module complete.

5.4 Raspi Broker

As mentioned in Chapter 3, the MQTT broker will be running in a Raspi hooked in the wireless network. That was done to remove that computational burden from the robot's Raspi. That machine may be placed anywhere as long as it has WiFi network coverage. This machine acts as the MQTT broker but may be used for other processes in future projects, as it may operate as a fog computing node to serve the robot machine if it needs higher computational power than it can provide. The concept of fog computing is relatively simple. It is a server structure similar to cloud computing but closer to the final user, which in robotics is interesting because it reduces the latency in communication [48].

A Raspberry Pi model 3 B was used to host the MQTT broker. The bridge script that receives the messages from the sensor nodes and sends them to the robot is started via SSH. The Mosquitto broker is a service that starts every time the Raspi initializes.

5.5 Real Experiment

The experiment with the real robot was performed on the same site of study, the first floor of the ESTIG building. Unlike the simulation, the map was not a straightforward blueprint. A SLAM mapped the whole building level. A ROS package called *Gmapping* was used to perform that. This package runs the algorithm and saves the final map. To drive the robot around, a package called *Teleop Twist Keyboard* was used, which allows the user to send velocity commands to the robot through the keyboard. All the interaction with the Raspi on the robot was executed via SSH or through a Virtual Networking Computing (VNC) software server. The SLAM result is shown in Figure 5.8.

After that, some corrections were performed to enhance the image, reducing the possible problems for the localization algorithms. Since the map is made of black, white, and gray pixels, it is possible to edit the image using the GIMP, clearing it for the robot to better identify the static obstacles. For example, each large corridor in the vertical axis of the image has a set of stairs. In the original map produced, these stairs were not captured

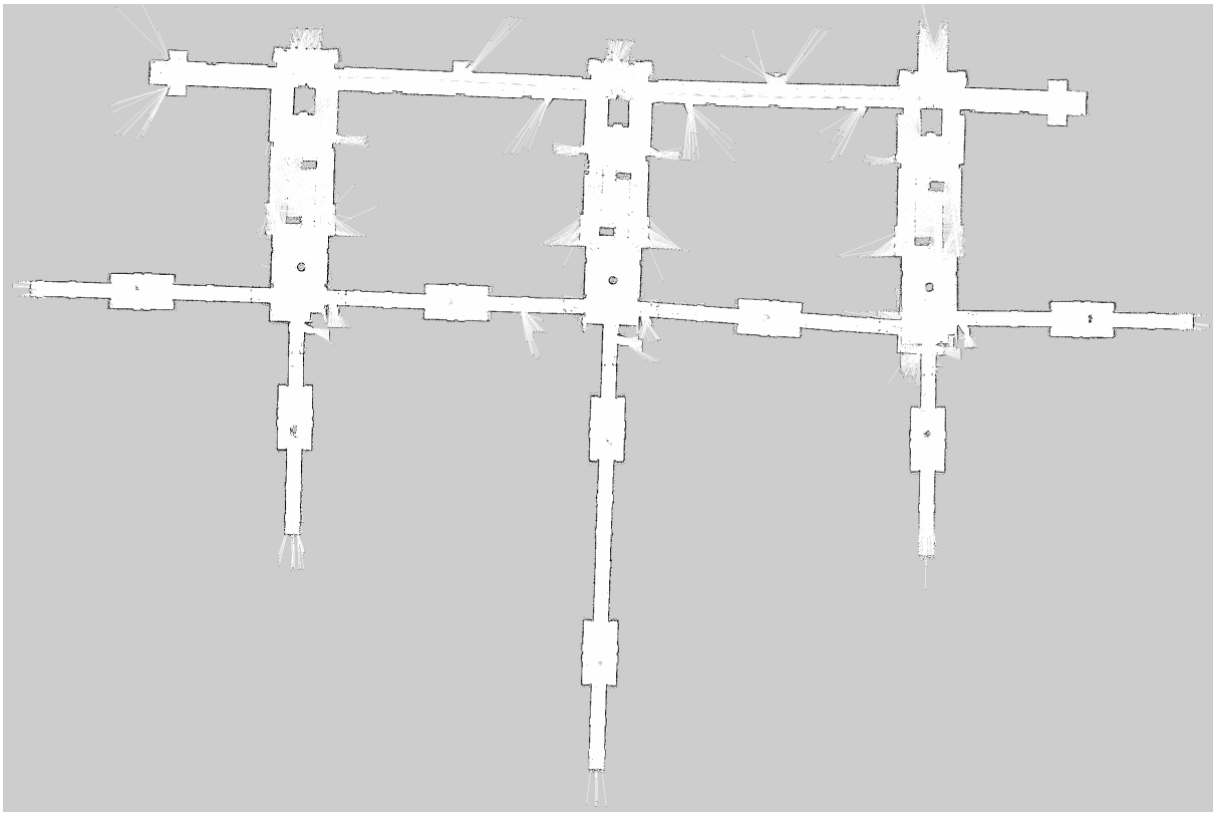

Figure 5.8: Map generated by the Gmapping SLAM algorithm.

by the lidar. Therefore, they were not well defined as obstacles. Another issue detected was a set of chairs located in the same corridors. The problem with them is that the legs of these chairs, because of their geometry, were not captured by the lidar either, so the robot would not see them as obstacles. The manipulations were performed to help the robot navigate with less trouble. Figure 5.9 show the edited file. Note that all the map flaws were corrected, contributing to noise reduction in the navigation process.

The Python controller script used in the simulation was adapted to perform the experiments with the Roomba©. The significant change in the script is the launch commands. Now, it launches the Roomba© nodes instead of the simulation ones, and the obstacles spawning were removed. All the data gathered was maintained the same, computational load, memory consumed, time spent, and distance traveled. It generates the same information folders and files for later analysis. That way, the analysis script used in the

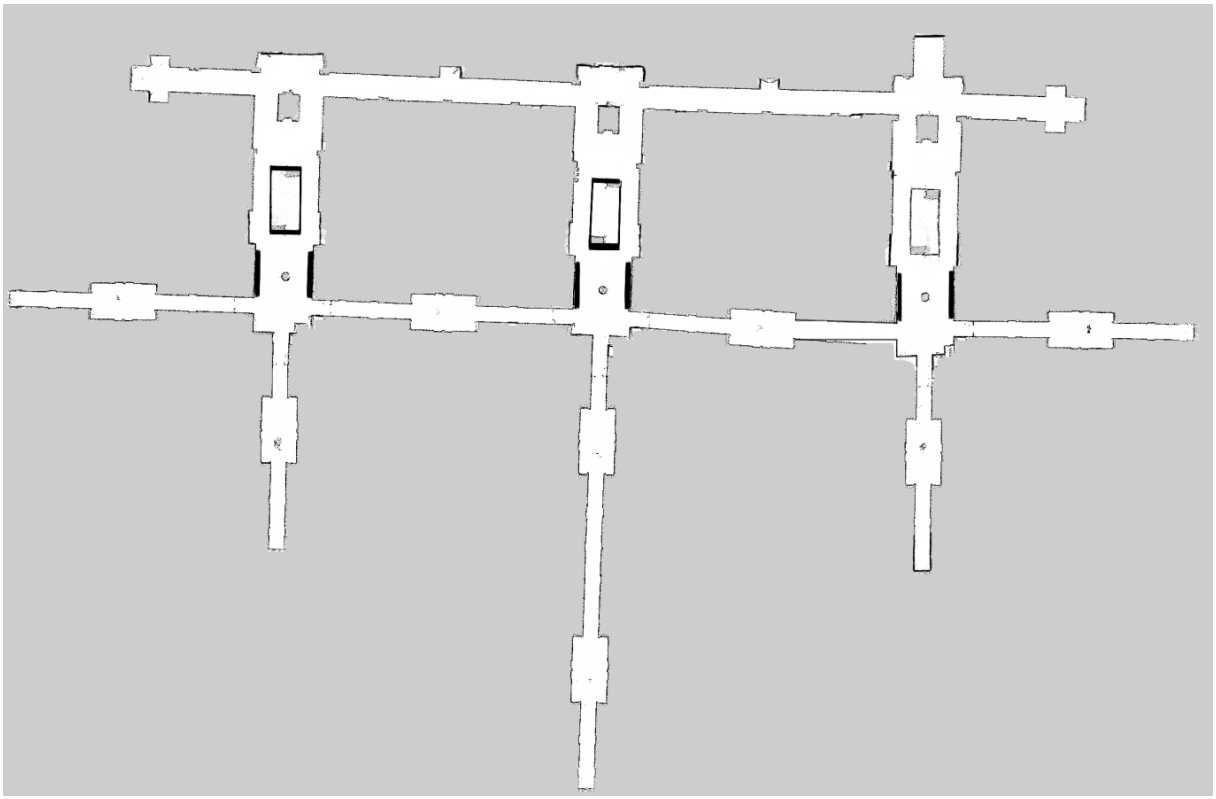

Figure 5.9: Edited map used in the experiments.

simulation will be used again for the real-site experiment results. The same routes were performed. Note that this script, in the simulation, performed the raffling of obstacles. Now, the same function in Python was raffling the places, but it is in a separate script and was executed before the runs. All the routes and areas were maintained the same, except for route "C" to "D", in which one possible location for the raffling of the obstacle was changed. Zone four was included as the robot calculated paths going through that zone instead of zone one, shown in Figure 4.10.

The modules were placed on the walls of each monitored zone. To hold them on the walls, double-sided tape was used. All the zones had good coverage of the wireless network, so the connection for the module was not a problem, even though the Raspi on the Roomba had some spots where it had to switch access points and lost connection for a couple of seconds, which did not affect the navigation process. Figure 5.10 show the

module on the wall. The height was chosen to detect a group of standing persons in the corridor.

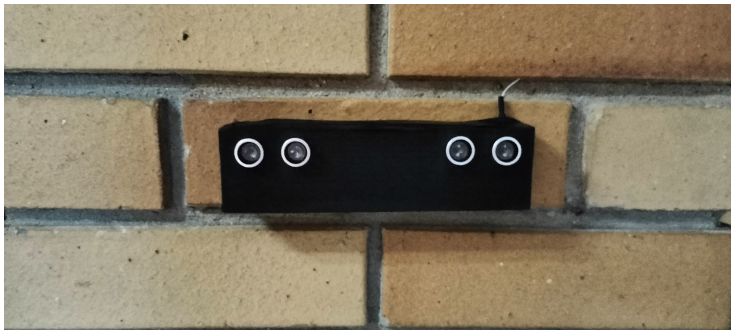

Figure 5.10: Module placed on the wall.

Figure 5.11 shows the corridor of the monitored zone one. As one can see, the module is placed on the wall at a height that can detect a standing person.

Figure 5.11: Module placed on the corridor wall.

The experiment was conducted the same way as in the simulation, with the change in route "C" to "D" and the reduced number of runs, which was defined to be ten for each setup. The default setup (without the proposed architecture and standard parameters) was run with wooden obstacles placed. Figure 5.12 shows the barriers blocking the passage of the robot. This obstacle was chosen to make it impossible for the robot to pass, just as danger would act in the environment. That's the case this work is focused on, a "danger" that the robot can not go through or is very hard to overcome.

Figure 5.12: Obstacles used to block the passage.

Roomba© were configured in the same YAML files shown in Chapter 4, in Section 4.5. In the simulations, the starting point was the default parameters for the TB3, found in the official repository. Now, the starting point was the parameters defined in the Roomba© repository [49]. The local and global path planners were maintained the same as in the simulation, *DWA*, and *A Star*. The parametrizations for the `Full Setup` and default setups were almost the same in the real scenario. The parametrizations on the Roomba© were more sensible than in the simulated TB3. Therefore, the parameters were adjusted to a point where it could safely navigate. Then, the parameters that the simulation showed were the most rewarding in terms of computational load (the size of the local costmap and the recalculations of the global path) were set accordingly. The size of the local costmap went from 3 to two meters, and the frequency of path calculations from 10 hertz to zero.

Chapter 6

Results and Discussion

This chapter will present the results obtained in simulated and real environments. At first, they are presented and discussed separately, comparing the proposal, called `Full Setup`, and the default. After that, it discusses and compares both scenarios, simulated and real. As the past chapters stated, the simulation and the real experiments were held with differential drive robotic platforms, the TB3, and the iRobot's Roomba©. As one can see, both platforms have very similar characteristics, which made the porting from the simulation to the real site much more straightforward. The parameters for navigation were minimally adjusted, as shown in Chapters 4 and 5. The possibility to simulate the study site and a very similar robotic system before the deployment of the real robot allowed a familiarization with the system, as well as debugging of parametrizations and testing of software developed.

In that context, ROS structures were decisive. They run the same way in simulation and the real system. The only change is the source of information. In the first case, it comes from the simulator, and the second comes from the real robot. That allowed us to test every subsystem in the architecture before launching it in the real scenario. Even the nodes' firmware could be debugged with the simulation help. This achievement is significant. It saved time and resources when deploying the real robot because much of the "trouble" was fixed before, in simulation. The parametrization and adjustments of the navigation stack were the main problems solved in the simulation, besides validating the

integration of the subsystems.

The architecture can produce two lines of improvements as a result. The first is the change in the behavior of the robot. This architecture ensured that the robot avoided the area sensed as a danger zone by placing the area's lethal costs in the costmaps through the new layer and recalculating the paths afterward. The second line is the computational improvements achieved by adjusting the ROS parameters. This proposed parametrization is done considering the system implemented. Without that architecture, some adjustments could not have been made. For example, the reduction of the local costmap and the frequency of global path calculations only worked well because of the advanced information on the already known troubled area.

6.1 Simulation Results

The simulated experiments were executed in a virtual machine (VM) in the IPB's infrastructure, which provides adjustable computing power through the virtualization of a computing cluster. This VM had four cores and 8GB of RAM.

Figures 6.1 and 6.2 show the visualization tool used in ROS to illustrate the change in the conduct of the robot. In both, the first moments of the simulation are presented, and the robot is pursuing the first route ("A" to "B"). Here, obstacles obstructed the passage in zone 4 (in both of them), but only the second had already placed the danger on the map (pink-colored area). The `Full Setup` already avoids the problematic area, and the default has not yet detected the danger in the route. One can say that integrating the ROS topic with the costmap through the new layer worked well. Also, the images bring a red line indicating the AMR calculated route. As one can see, in the first, the computed path passes through the problematic area, guiding the robot to the trouble. In the second figure, the path has already been recalculated. The blue-colored area around the robot is the local costmap. In the `Full Setup`, its size was reduced to lower the computational load.

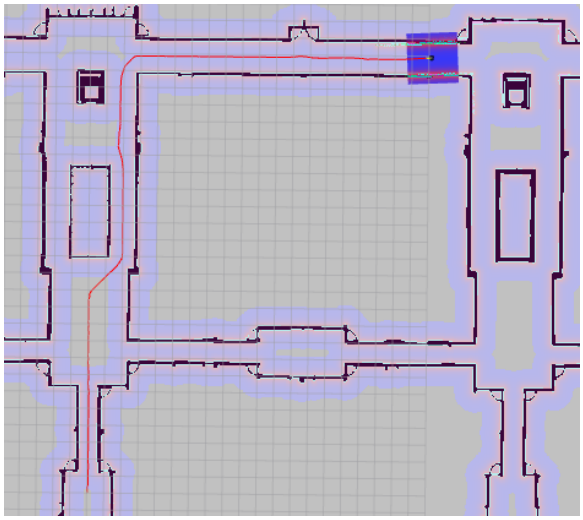

Figure 6.1: Default setup.

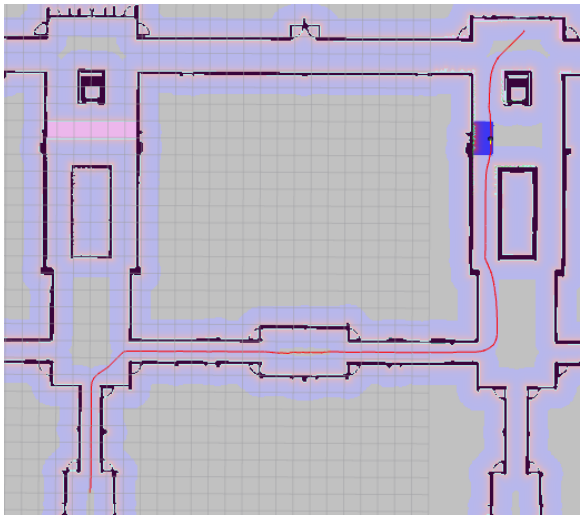

Figure 6.2: `Full Setup`.

As these results show, the proposed system allows the AMR to take alternative routes to avoid dangerous environmental zones. Without our architecture, the robot would only detect the problem by getting closer to it when the trouble appeared in the sensors' range. This anticipation reduces the time taken to complete the routes and diminishes path distance in cases where the robot's passage near the unsafe area must not happen or is physically impossible.

To illustrate that result, the charts in Figures 6.3 and 6.4 were produced. They show, respectively, the mean and 95% confidence interval of the distance traveled and the time spent on each route. The blue bar represents the `Full Setup`. The orange bar illustrates the default setup. These results show that in the case that the danger makes the crossing impossible for the robot, the advanced information permits anticipation and correction of the routes in time to reduce the time and distance traveled. The ability to "predict" what is going to be in the way, in this case, produces these reductions as the obstacles or danger does not have to be inside the sensor's range, so if the robot has to change the path, it saves time and distance to detect the problematic area. In all the routes, the AMR saved around eighty meters in the distance traveled and reached the goals around 20 seconds earlier.

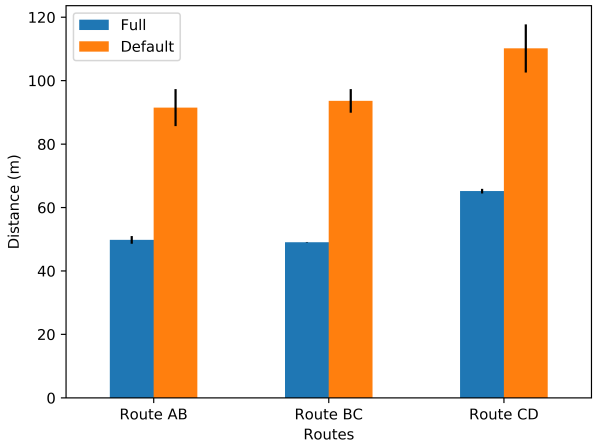

Figure 6.3: Distance travelled each route.

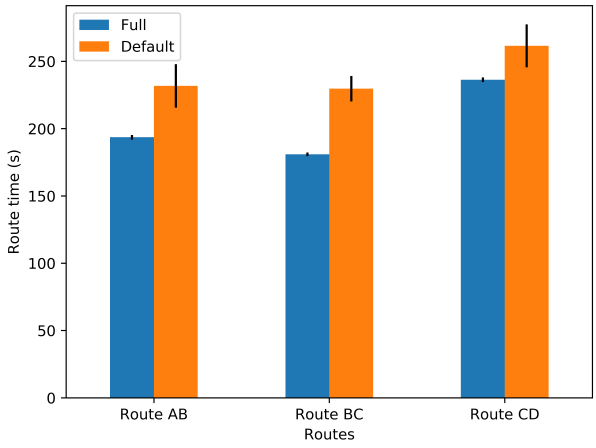

Figure 6.4: Time to complete the routes.

To compare the parametrization, the Python control script monitored the computational load used by the *"move_base"* process, which is the central node that integrates the algorithms and sends the controlling commands to the drivers. Besides that process, it monitored three other essential processes in the Navigation Stack, the *"map_server"*, responsible for passing the static map to the AMR, the *"amcl"* (localization algorithm process), and the *"rosmaster"*, the central broker in the ROS system. The values for the *"move_base"* process are the ones that showed significant improvements. The other processes maintained, statistically, the same behavior.

Figure 6.5 shows a substantial reduction in computational load. The blue triangle represents the `Full Setup`. As one can see, the mean values for all the routes are between 25 and 50 percent. The orange triangle shows the values for the default setup, the mean falls, for all routes, higher than 175 percent CPU consumption, much higher than the `Full Setup`. This achievement was possible by tuning some parameters considering the advanced information of our architecture. For example, the size of the local costmap was reduced, frequency of path planning and map updates. It is a notable result since embedded boards with limited processing power usually control mobile robots. The chart below shows CPU percentage values over one hundred, which happens in multi-core systems when the process runs in more than one core.

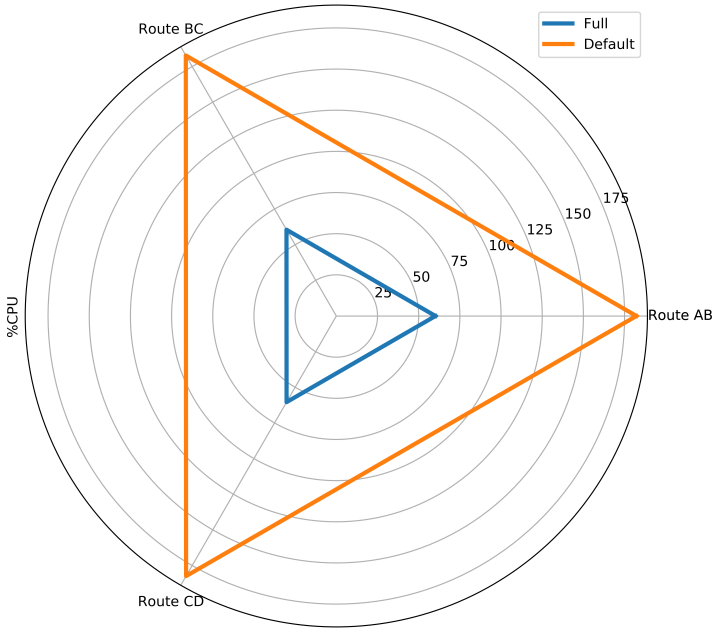

Figure 6.5: Move base percentage of CPU consumption.

Besides monitoring the CPU, the controller script gathered memory information from the *Top* command regarding the same ROS processes. The adjustments to the parameters had no substantial impact on memory usage. The only process that showed a slight reduction was the *"move_base"*, but the analysis showed that statistically, the mean values had no difference. Figure 6.6 present the chart with the values. The blue bars represent the `Full Setup`, and the orange is the default.

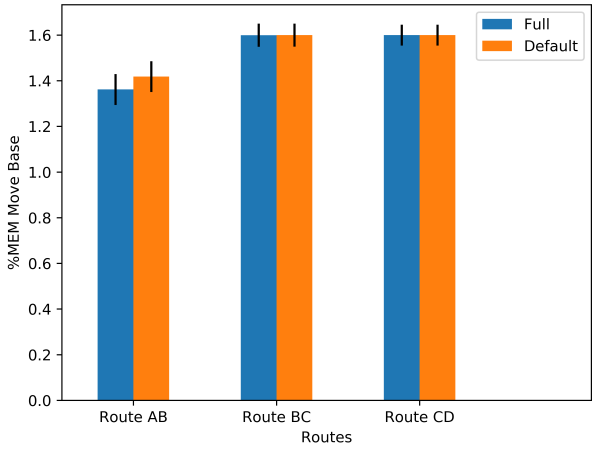

Figure 6.6: Move base percentage of memory usage.

6.2 Real System Results

The real environment experiments were executed in the ESTIG building in hours when there were fewer people. That was necessary because of the localization system implemented. The robot sometimes would get lost if people blocked the way in narrow pathways. To avoid these localization problems, which are out of the scope of this study, the tests were performed at night, when the building had fewer people around.

As mentioned, the same information gathered in the simulation was stored in the real scenario experiments. Executing fifty runs for each setup would consume an enormous amount of time. Because of that, ten runs were executed for each setup on the real site.

The results of the real scenario experiments were as good as the simulation ones. Figures 6.7 and 6.8 present the charts produced with the real experiment data. Like in the simulation charts, the blue represents the `Full Setup`, and the orange is the default. Again, the results show improvements in time and distance, as the mean time to complete the routes and the mean distance traveled were smaller with our system. The first two routes showed a better improvement than the last one. While the third saved around thirty meters and 15 seconds, the first two saved around seventy meters. The first lowered the time spent by sixty seconds, and the second by 80 seconds.

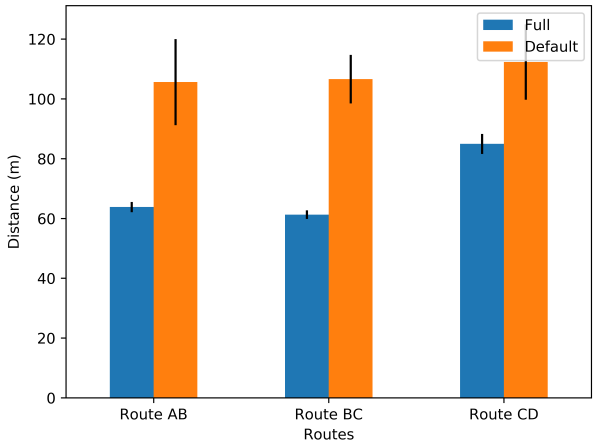

Figure 6.7: Distance travelled on the routes.

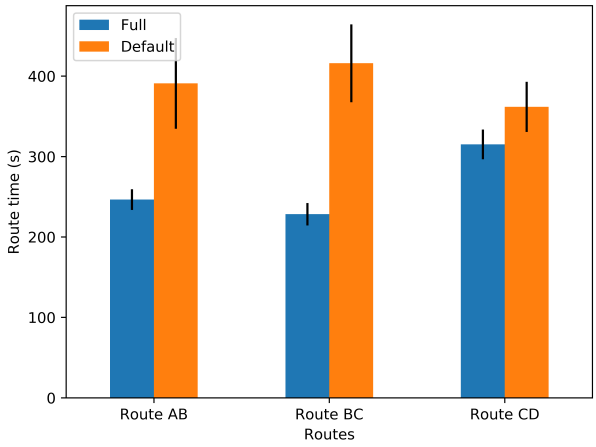

Figure 6.8: Time to complete the routes.

Looking at the computational improvements, the same tendency from the simulation

appeared in the data collected. Figure 6.9 presents the chart for the percentage of CPU consumption of the *"move_base"* process, mentioned in the last section. The blue triangle is the `Full Setup`, and the orange is the default setup. The default values fall around 110 percent of CPU consumption. The `Full Setup` reduced that value to around twenty percent. As one can see, the CPU burden introduced by this ROS process is drastically reduced by the better parametrization allowed by the proposed architecture.

Like in the simulation, the size of the costmap was reduced, which makes the cost value matrix smaller, reducing the number of calculations the robot has to perform. The global path planner frequency was reduced to zero (10Hz is the default value), meaning that it only recalculates the path when an obstacle is detected obstructing it, made possible by the mechanism introduced by this work. The size of the costmap is crucial because it is the area that the local path planner uses to determine the best velocity commands. The bigger that area, the bigger the area of possible obstacles for the planner to perceive. With the advanced information introduced by the proposed architecture, and considering that the maximum linear velocity the deployed AMR can execute is 0.3 meters per second (relatively low), it was possible to reduce that size without any trouble navigating (as the experiments showed), but reducing the computational burden. The default size is 3 meters. In the `Full Setup`, it was two meters.

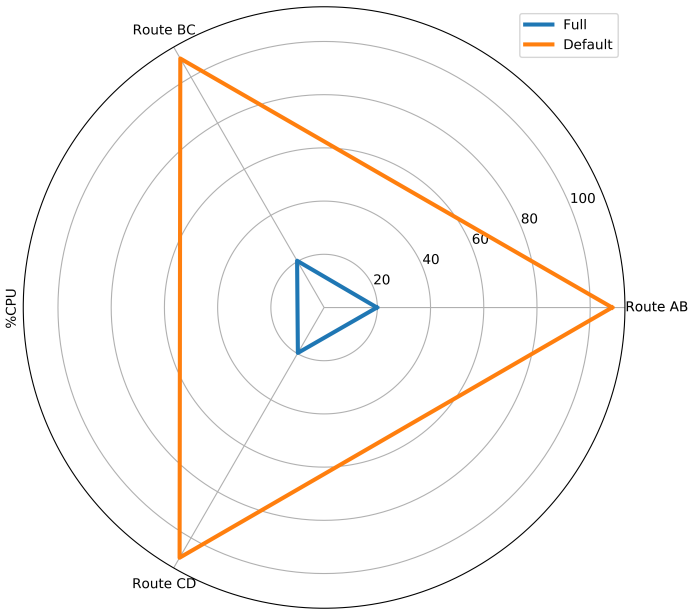

Figure 6.9: Move base percentage of CPU consumption.

The results in terms of the memory usage for the real robot were different from the ones simulated. The processes *"rosmaster"*, *"map_server"*, and *"amcl"* maintained the same mean values when comparing the `Full Setup` and the default. However, the *"move_base"* process showed a higher usage of memory with the `Full Setup`. Figure 6.10 present the mean values, the blue bar is the `Full Setup`, and the orange is the default.

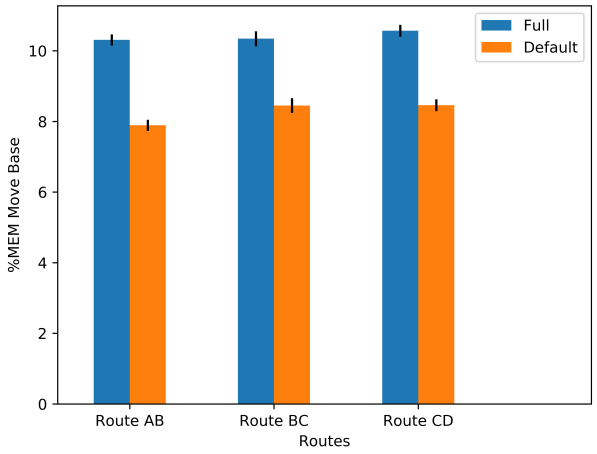

Figure 6.10: Move base percentage of memory usage.

6.3 Discussion

When looking at the results, one can see that the simulated and the real site experiments converged to the same overall results. In both cases, the results show that the parametrization directly influences CPU consumption as it interferes with the calculations the AMR has to perform.

The improvement in the navigation process was also evidenced in both scenarios. The fact that the AMR avoided the danger zones faster implies that the navigation was safer since the system kept the platform further away than the default setup, that to change the path had to get closer to the danger.

Noting that the results in the real environment followed the results acquired in simulation, and the integration and testing of the system could be validated before deploying, the virtual experiment played a crucial role in the success of the implementation in the

real scenario. Figures 6.11 and 6.12 present the comparison of the times and the distances between the simulated and the real experiments for the `Full Setup`. As it shows, even though the values were not the same, the tendency obtained in the simulation appeared in the real scenario, which confirms that Gazebo is a good platform for simulating ROS systems. The fact that the simulations were run only with walls of the site may have caused the difference between the absolute values of the simulation and the real site experiment. Having fewer things around the robot leads to less noise for the navigation algorithms, which may work faster and the robot travels freer. Besides that, the computational power of the VM is higher than the Raspi on the robot, which also could have influenced.

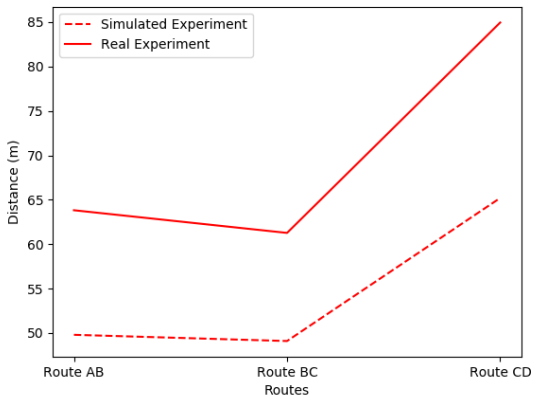

Figure 6.11: Simulated and real distances.

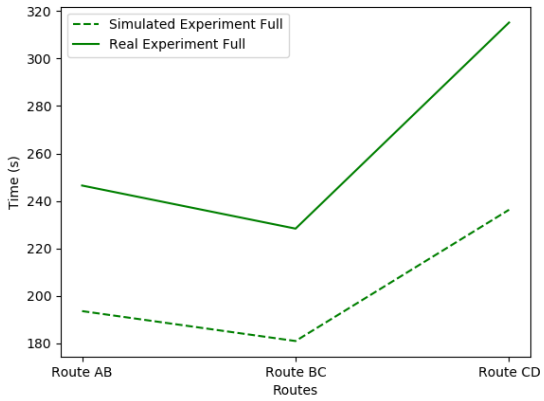

Figure 6.12: Simulated and real times.

The Python controller script used for the simulations and adapted to the real experiments opened vast possibilities for testing. It launches, controls, gathers data, and calls the second script to analyze the information. It is a powerful tool for any ROS developer. Parametrizations may be compared, different scenarios may be deployed, and virtually any situation can be run several times and studied.

A WiFi network problem was encountered when launching the AMR in the real environment. The ESTIG building is big. Therefore, several access points (APs) are needed for the network to cover all the sites. The problem was that the Raspi on the AMR would sometimes disconnect from one AP and not connect to the next closest one. Changing the

configuration of the WPA supplicant solved the issue. It was set to keep trying to connect to the network until it was successful. The change of APs introduced some seconds where the robot was offline. However, those seconds did not affect the performance in our experiments since the architecture is an addition to the system and the AMR regular processes run offline.

Another trouble faced was the irregularities in the building floors. The IRobot Roomba has a low structure with small wheels. The building has some floor inconsistencies where the robot sometimes gets stuck. Besides that, some doorways have a separation on the floor. These separations introduced an unevenness that sometimes also grabbed the robot's wheels, which resulted in distorted odometry measurements, creating situations where the robot is unable to locate itself or even move. When this happened, the experiment had to be reset. This problem did not greatly influence our work, but showed the shortcomings of the design of some robots, making them specific to certain applications.

Note that our implementation problems were unrelated to our proposed integrated sensory architecture. All the validations had already been done in the simulation environment, which saved us time.

The AMR avoided all the areas where the obstacles were placed and was able to recalculate the routes the same way it did in the simulation. The danger detection from sensor nodes took very few seconds (around 3) to detect the standing persons in the pathway and reach the AMR. Therefore, danger avoidance successfully improved the navigation process by making it safer.

Although this work was limited to one site scenario, the architecture may be deployed in several indoor environments and various situations. The use of MQTT delivered the flexibility to add as many sensor sources as needed. Scalability is an important feature these wireless sensor systems have to provide. To implement the system with a ROS robot, one would have to adjust the location of the sensor modules and the inflation values on the layer plug-in to cover the sensed area. The source of information is also flexible. This work focused on the data gathered by the wireless sensor network and the integration, but the flexibility of MQTT, mentioned above, allows the information to come from any

source. For example, one could have a system where users would choose areas for the robot to avoid for any possible reason. They would have to publish the area's location into the MQTT topic (related to the map frame) and adjust the size of the inflation area for the robot to avoid. In industrial sites, where maintenance happens periodically, this architecture could inform the AMR in advance that the area is blocked. That is a perfect example to show how flexible the system can be.

Another essential characteristic is that the wireless sensor nodes and the MQTT subsystems can work with several robotic platforms. The structure designed with MQTT allows any robot connected to the broker to receive dynamic information about the environment. Note that any robot running the ROS navigation stack, and an MQTT client, is capable of adhering to this system and benefiting from it.

The sensor modules implemented in this project were developed to be cost-effective. Applying the concept of the Minimal Viable Product (MVP), it was possible to assemble a module using simple hardware easily found in the market, with relatively low cost, that delivered good results and fulfilled the expectation for a prototype. Considering the components and the 3D printed case, the module cost would be around 25 euros. Since it is an addition to the existing AMR system, the nodes were kept as simple as possible.

The Navigation Stack parameters must be adjusted for each robotic platform. A parametrization that worked for AMRs studied in this dissertation may not work for others. That noted, the proposed parametrization is relevant for the iRobot's Roomba©, TB3, and for very similar platforms, it may be used as a guide. It is essential to say that, because this parametrization has to be done for each robot individually, the improved computational behavior worked well because it was trimmed for our specific system. One should not expect the same results implementing this parametrization on other platforms. Note also that there is always a trade-off between reducing the computational burden and the quality of the navigation process. It was possible to reduce the computational load safely because of the advanced information about the environment. Reducing the path calculations and the size of the costmap in the default setup would also reduce the computational burden. Without the architecture, the navigation would not be as safe

because detecting problems in the routes would be slower. That is why already known problematic areas were chosen to be monitored. That way, it ensures these areas are continuously surveilled.

As mentioned, the values obtained about the memory usage in simulation showed no difference between the `Full Setup` and the default setup. However, the real experiment resulted in higher memory usage by the *"move_base"* process. This result was expected since adding the new layer increases the data the system has to keep, in this case, about the map. The fact that it did not appear in the simulation may be explained by analyzing the differences between the computing environments. The virtualization and the use of a simulator could have diluted the memory usage, and the monitoring would have had to be more detailed. Since the difference did not affect the navigation, and the focus of the work was not on memory usage, this detailed analysis was not performed.

Like any other monitoring or sensory system, the performance is directly related to the amount of information it can receive and treat. In this case, since there was limited access to materials, as mentioned before, only the usually busier areas were monitored in the building. In an exemplary implementation, a more extensive area would be surveilled. Increasing the sensor coverage would boost the results in terms of safety in navigation, and the system would be even more robust. This extrapolation would not affect memory usage since the layer is already created, even though it could lead to a higher CPU consumption. That is why the reduction through parametrization is relevant.

The problems encountered in the real experiments, with the robot being unable to overcome the floor unevenness, were related to the habitat not being prepared for receiving AMRs. The adjustment made to the SLAM-generated map also reveals that issue. The stairs were not at the Lidar range in the specific site, and neither was some furniture mentioned in past sections. Because of that, the robot could not identify these obstacles. Although the AMR navigated safely (did not crash into static obstacles), some adjustments in the habitat are always welcomed. It only helps the results. The experiments were held with the doors closed, and some windows captured by the Lidar were blocked with paper. That was necessary to reduce localization problems. The SLAM was executed

with the same environment, but in an actual operation (not in experimental scenarios), one can not guarantee that the environment is the same. Doors could be left opened or closed, for example, which inserts noise to the AMCL localization algorithms, possibly leading to the AMR getting lost. Therefore, a more robust localization algorithm could improve the navigation process in a continuous operation.

The project's environment permitted at least two possible paths to get to the targets. This scenario was perfect for testing this prototype because it allowed the robot to choose other paths when one was blocked. In the case of a site that does not have that possibility, and blocking the entire area is not an option, instead of classifying the danger areas and closing them completely, other strategies for placing the danger on the maps would have to be developed. That implies having better sensors and better ways to identify the size and structure of the obstacles, something that is not quickly done with low-cost sensors.

In this work, the Python programming language was crucial in speeding up the developments as its library capabilities perfectly fulfilled many tasks needed to deploy the system. Although in ROS the C++ language was used in a critical subsystem (the plug-in layer), the vast majority of the software produced by this work is in Python 3. The Rospy library provided access to the ROS framework through Python. The programming knowledge acquired while developing these scripts is one of the best results this work delivered.

Besides the results presented, the architecture opens the exciting possibility of implementing *Fog Computing*. The fact that the MQTT broker is running in a separate machine communicating with the robot allows the developers to use this machine for any computational tasks, transferring processes from the robot to the *Fog*. That provides more computational power to the platform but faster data transfer than cloud connections. Although the time limitations did not allow it to be developed in this work, using this concept could lead to more robust and complete systems since it expands the capabilities of the usual restrained embedded systems.

Chapter 7

Conclusion and Future Works

7.1 Developed Works

This dissertation presented a concept in sensory design applied to mobile robotics, which integrates advanced dynamic environment information with AMRs navigating with ROS. This work proposed and implemented a wireless sensor network that communicated with the ROS environment running on an AMR through MQTT. The introduced architecture was designed with layered costmaps and ROS topics that receive information about the environment from sources outside the robot's sensory structure. This fusion was validated through simulated and real experiments, presenting some of the possible gains. The system produced behavior that avoids danger zones with faster detection and reaction. This conduct opened the possibility of adjusting critical parameters of the navigation stack, reducing the computational load. The final result is an AMR with less CPU burden and better danger avoidance.

Looking at the objectives proposed in Chapter 1, it is fair to state that they were successfully achieved. The ROS environment and the Navigation Stack were fully understood and comprehended. ROS played an essential role as it accelerated the developments with the possibility to reuse the already functioning autonomous navigation structures, the drivers, and URDF files of the platforms deployed. If all those steps had to be done

in the context of this dissertation, the time limitation would hit the developments much harder. Integrating the wireless sensor nodes and the ROS costmap layer was accomplished, which is crucial to holding the subsystems together. The simulated environment created in Gazebo could effectively represent the real scenario. The results showed that the values followed the same tendency. The simulation controlling script, adapted to the real scenario, could perform all the critical tasks needed. This development was crucial for the success of this work as the tests were automated and did not require the user to perform tasks while the tests were in execution. The experiments in the real environment could be successfully performed and compared between the setups analyzed. The work developed in this context delivered a fully functioning autonomous robotic platform, which may be used in various other future projects. Because of the modularity of ROS, even different robotic platforms may benefit from this dissertation's achievements.

7.2 Future Works

This dissertation presented an initial prototype of the proposed system architecture. The architecture's adaptability allows the developer to implement it in several situations, as mentioned before. Several developments may be carried out in the future as the time limited the developments in this work context.

The Python script controlled the experiments. It was responsible for setting specific navigation goals. In future work, to deploy the guidance operation, a human-machine interface could be developed for receiving the location where the user wants the robot to go. For that, a database linking the location to the map coordinates. An application could receive that data and send the goal to the Navigation stack. Besides, implementing a more robust robotic platform and creating a body structure to receive the interacting platform.

As mentioned in the last chapter, the sensor modules employed in this work were a prototype to validate the system with few resources. In future projects, new sensors could be evaluated for this purpose. A cost-benefit analysis would enrich the outcome and guide

the deployment of new systems with different sensors. This evaluation could address the cost, range, and type of danger detected for several types of sensors.

Since this work was confined to only one study site (ESTIG's building), an interesting future development would be to deploy it in other scenarios, with more than one AMR, with more sensor sources, and evaluate the results. Analyzing the network traffic would also be interesting to understand how the system affects WiFi communication.

The discussion section cited a user interface to inform the AMR of possible danger areas. That could result in other projects. This platform could be used by maintainers of the system in situations where the robot has to keep away from any area on the map. It would extend the information sources the robot can listen to, which enriches the navigation process.

The localization problem in this work is solved using the AMCL algorithm. It takes in odometry and laser information to localize the AMR in the map. Although it works, it is not the most robust technique. Integrating this system with other localization processes, such as fiducial markers, would significantly improve this dissertation. Because of that, future work could be carried out in that direction.

Bibliography

[1] M. Cardona, A. Palma, and J. Manzanares, "Covid-19 pandemic impact on mobile robotics market", in *2020 IEEE ANDESCON*, IEEE, 2020, pp. 1–4.

[2] M. Yousif, C. Hewage, and L. Nawaf, "Iot technologies during and beyond covid-19: A comprehensive review", *Future Internet*, vol. 13, no. 5, p. 105, 2021.

[3] N. G. Hockstein, C. Gourin, R. Faust, and D. J. Terris, "A history of robots: From science fiction to surgical robotics", *Journal of robotic surgery*, vol. 1, no. 2, pp. 113–118, 2007.

[4] F. Rubio, F. Valero, and C. Llopis-Albert, "A review of mobile robots: Concepts, methods, theoretical framework, and applications", *International Journal of Advanced Robotic Systems*, vol. 16, no. 2, p. 1 729 881 419 839 596, 2019.

[5] M. B. Alatise and G. P. Hancke, "A review on challenges of autonomous mobile robot and sensor fusion methods", *IEEE Access*, vol. 8, pp. 39 830–39 846, 2020.

[6] N. A. K. Zghair and A. S. Al-Araji, "A one decade survey of autonomous mobile robot systems", *International Journal of Electrical and Computer Engineering*, vol. 11, no. 6, p. 4891, 2021.

[7] M. A. Niloy, A. Shama, R. K. Chakrabortty, M. J. Ryan, F. R. Badal, Z. Tasneem, M. H. Ahamed, S. I. Moyeen, S. K. Das, M. F. Ali, *et al.*, "Critical design and control issues of indoor autonomous mobile robots: A review", *IEEE Access*, vol. 9, pp. 35 338–35 370, 2021.

[8] P. Pfaff, W. Burgard, and D. Fox, "Robust monte-carlo localization using adaptive likelihood models", in *European robotics symposium 2006*, Springer, 2006, pp. 181–194.

[9] A. d. Oliveira Júnior, *Combining particle filter and fiducial markers in a slam-based approach to indoor localization of mobile robots*, 2022.

[10] L. Zhang, R. Zapata, and P. Lépinay, "Self-adaptive monte carlo localization for mobile robots using range sensors", in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2009, pp. 1541–1546.

[11] Z. Xuexi, L. Guokun, F. Genping, X. Dongliang, and L. Shiliu, "Slam algorithm analysis of mobile robot based on lidar", in *2019 Chinese Control Conference (CCC)*, IEEE, 2019, pp. 4739–4745.

[12] *Open slam*, Available: `https://openslam-org.github.io/gmapping.html`, Accessed: June 2022.

[13] K. Karur, N. Sharma, C. Dharmatti, and J. E. Siegel, "A survey of path planning algorithms for mobile robots", *Vehicles*, vol. 3, no. 3, pp. 448–468, 2021.

[14] L.-s. Liu, J.-f. Lin, J.-x. Yao, D.-w. He, J.-s. Zheng, J. Huang, and P. Shi, "Path planning for smart car based on dijkstra algorithm and dynamic window approach", *Wireless Communications and Mobile Computing*, 2021.

[15] K. Zheng, "Ros navigation tuning guide", in *Robot Operating System (ROS)*, Springer, 2021, pp. 197–226.

[16] M. M. Costa and M. F. Silva, "A survey on path planning algorithms for mobile robots", in *2019 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, IEEE, 2019, pp. 1–7.

[17] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, *et al.*, "Ros: An open-source robot operating system", in *ICRA workshop on open source software*, Kobe, Japan, vol. 3, 2009, p. 5.

[18] *Navigation robot setup*, Available: http://wiki.ros.org/navigation/Tutorials/RobotSetup, Accessed: August 2022.

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

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

[21] *Creating a new layer*, Available: http://wiki.ros.org/costmap_2d/Tutorials/CreatingaNewLayer, Accessed: January 2022.

[22] A. Khanna and S. Kaur, "Internet of things (iot), applications and challenges: A comprehensive review", *Wireless Personal Communications*, vol. 114, no. 2, pp. 1687–1762, 2020.

[23] R. A. Mouha, "Internet of things (iot)", *Journal of Data Analysis and Information Processing*, vol. 9, no. 2, pp. 77–101, 2021.

[24] M. Majid, S. Habib, A. R. Javed, M. Rizwan, G. Srivastava, T. R. Gadekallu, and J. C.-W. Lin, "Applications of wireless sensor networks and internet of things frameworks in the industry revolution 4.0: A systematic literature review", *Sensors*, vol. 22, no. 6, p. 2087, 2022.

[25] K. Gulati, R. S. K. Boddu, D. Kapila, S. L. Bangare, N. Chandnani, and G. Saravanan, "A review paper on wireless sensor network techniques in internet of things (iot)", *Materials Today: Proceedings*, 2021.

[26] L. Miller, *Which controller should i use... wemos d1 mini or nodemcu?*, Available: https://www.learnrobotics.org/blog/which-controller-should-i-use-wemos-d1-mini-or-nodemcu/, Accessed: August 2022.

[27] *Esp8266ex datasheet*, Available: https://espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf, Accessed: August 2022.

[28] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications", *IEEE communications surveys & tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[29] D. Soni and A. Makwana, "A survey on mqtt: A protocol of internet of things (iot)", in *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*, vol. 20, 2017, pp. 173–177.

[30] V. Zhmud, N. Kondratiev, K. Kuznetsov, V. Trubin, and L. Dimitrov, "Application of ultrasonic sensor for measuring distances in robotics", in *Journal of Physics: Conference Series*, IOP Publishing, vol. 1015, 2018, p. 032 189.

[31] *Connecting an ultrasonic sensor (hc-sr04 5v) to a micro:bit*, Available: https://www.teachwithict.com/hcsr045v.html, Accessed: June 2022.

[32] *What is platformio?*, Available: https://docs.platformio.org/en/latest/what-is-platformio.html, Accessed: August 2022.

[33] ——, "Application of ultrasonic sensor for measuring distances in robotics", in *Journal of Physics: Conference Series*, IOP Publishing, vol. 1015, 2018, p. 032 189.

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

[35] *Eclipse mosquitto an open source mqtt broker*, Available: https://mosquitto.org/, Accessed: August 2022.

[36] H. D. Ghael, L. Solanki, and G. Sahu, "A review paper on raspberry pi and its applications", *Int. J. Adv. Eng. Manag*, vol. 2, no. 12, pp. 225–227, 2021.

[37] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator", in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, IEEE, vol. 3, 2004, pp. 2149–2154.

[38] *About gazebo*, Available: `https://gazebosim.org/about`, Accessed: August 2022.

[39] J. Collins, S. Chand, A. Vanderkop, and D. Howard, "A review of physics simulators for robotic applications", *IEEE Access*, vol. 9, pp. 51 416–51 431, 2021.

[40] *Robotis-git/turtlebot3*, Available: `https://github.com/ROBOTIS-GIT/turtlebot3`, Accessed: January 2022.

[41] *Robotis e-manual*, Available: `https://emanual.robotis.com`, Accessed: June 2022.

[42] *Simulacao_pedro*, Available: `https://gitlab.estig.ipb.pt/_cedri/andre_mendes/pedro_mendes/simulacao_pedro`.

[43] *Irobot*, Available: `https://www.irobot.com`, Accessed: September 2022.

[44] *Roomba 600 series*, Available: `https://www.irobot.com.br/roomba/600-series`, Accessed: September 2022.

[45] *Autonomylab/create_robot*, Available: `https://github.com/AutonomyLab/create_robot`, Accessed: June 2022.

[46] *Rplidar a1*, Available: `https://www.slamtec.com/en/Lidar/A1`, Accessed: June 2022.

[47] *Slamtec/rplidar_ros*, Available: `https://github.com/Slamtec/rplidar_ros`, Accessed: June 2022.

[48] H. F. Atlam, R. J. Walters, and G. B. Wills, "Fog computing and the internet of things: A review", *big data and cognitive computing*, vol. 2, no. 2, p. 10, 2018.

[49] *Create_robot*, Available: `https://github.com/AutonomyLab/create_robot`, Accessed: June 2022.