POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING

MSc. Mechatronic Engineering

INSYSTEMS AUTOMATION GMBH

BERLIN, GERMANY

Sharing context information in a fleet of robots

University Tutor:
Prof. Alessandro Rizzo

Company Tutor:
Dipl. Ing. Jan Stefan Zernickel

Author:
Simone Noschese

POLITECNICO DI TORINO

Abstract

Mechatronic Engineering
Department of Control and Computer Engineering

Master of Science

Sharing context information in a fleet of robots

by Simone NOSCHESE

Autonomous Guided Vehicles are a product that saw a huge increase in the demand over the last decade. This is due to the fact that they provide an optimized solution to speed up the production and reduce the costs.

The project of this thesis was developed at InSystems Automation GmbH, which is a German company based in Berlin specialized in the production of autonomous guided vehicles. This project was developed as part of the Collaborative Embedded Systems (CrESt) project. CrESt, as a research project funded by the Federal Ministry for Education and Research, is concerned with the development of collaborative embedded systems, with a focus on dynamic software systems. InSystems Automation is one of 22 international partners from industry and science who take part in this research project.

The aim of this thesis is to provide the robots in a fleet with the context-awareness obtained through collaboration that allows each one of them to be aware of the presence of obstacles in a working environment without the need of directly observing them. This will allow the robots to react to the information which has been received and optimize their path calculation accordingly. The focus is on the obstacles that are blocking a pathway inside the facility, since those can significantly affect the capacity of the fleet.

The functionality has been implemented on a fleet of simulated AGVs with natural navigation. The foundation of this navigation system is on simultaneous localization and mapping algorithms (SLAM). These AGVs are able to freely navigate in a facility without the need for a predefined path. Moreover these AGVs are provided with the capability of recomputing their path according to their real-time perception of the surrounding area based on the readings of their sensors.

The solution that was found is based on the monitoring of areas in which the presence of an obstacle blocking a pathway has been detected. When an AGV in the fleet detects such an obstacle for the first time, each AGV in the fleet will then be able to monitor the presence of the obstacle and update its status upon new observations of the same.

Acknowledgements

I would like to thank my Professor Alessandro Rizzo for offering me this opportunity, for his advice, and for the support he has provided me during my time in Berlin.

I would like to thank the InSystems Automation family as they offered me a great opportunity to learn and to grow from both a professional and personal point of view. I would like to thank in particular my company supervisor Jan Stefan Zernickel for the guidance, encouragement and advice he has provided me during my time at InSystems. I would also like to thank my collegues for their support, in particular Daniele Indelicato for his guidance and his precious advice, especially during the early stages of the project, Christian Burghardt for his technical support about the simulator, and Antonio Misuraca for his precious graphic skills.

I want to thank my family for supporting me through every step of the process that has brought me to this very day (and in particular for their extensive sponsorship), none of this would have happened without them.

Last but not least I want to thank all of my friends for making this journey much more pleasant, from the night study sessions to the nights out, if I will remember these years as a great period of my life it will be thanks to you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As new generations of more capable autonomous systems are appearing, the possibility of having different systems interconnected and cooperating is sounding day by day as a less remote possibility for the future. Considering for example a possible future in which driver-less cars are interconnected, we can imagine an application where, in proximity of a crossroad, the cars are connecting to an interactive traffic light which shares with them the context information necessary to coordinate the traffic which can then proceed smoothly and safely.

Another example of a possible scenario would be a set of cars which are proceeding in a queue in an highway, when an unknown obstacle suddenly appears in front of the first car of the queue, having the information about the activation of an emergency braking procedure shared across the different cars would significantly reduce the risk of accidents involving multiple cars as an effect of the elimination of the human reaction delay.

Although the concept of human-robot collaborative systems is a field where a lot of research is being performed, the concept of automated robots collaborating with each other is still relatively new in the field of robotics. Let's consider a fleet of autonomous guided vehicles (AGVs) which are able to navigate freely in an environment such as a factory or a warehouse (i.e. without having to follow predefined paths in the facility), the capability of the single agents in the fleet to cooperate with the others can significantly improve the way in which the AGVs deal with uncertainty. This is done by dealing with uncertainty not as a set of individual robots but as a fleet. The concept of a collaborative fleet of AGVs can improve the performance of the fleet itself by taking advantage of the information which is collected by all the single agents in the fleet.

Let's consider, as an example, a situation in which the presence of a forklift or of materials waiting to be moved is blocking a corridor inside a warehouse. When a member of the fleet has to drive through that corridor in order to reach an assigned goal it will find its path blocked by the presence of an obstacle and will have to compute a new path to reach the assigned goal. The presence of this obstacle would then impair the capacity of the fleet as each robot that is then passing through that corridor would find its path blocked and would then have to recalculate a different way to reach the assigned goal. If the robots in the system have a way to share the information relative to the presence of an obstacle which is blocking a corridor in the facility then only the first robot would find its path blocked, as all the other agents

in the system would then receive the information and could optimize their path calculation according to the received information.

The aim of this thesis is to provide an implementation of the just described behaviour in a fleet of AGVs. The objective is to provide the robots in the fleet with the information relative to the presence of an obstacle which is blocking a specific pathway as, for example, a corridor in the facility, without having to directly observe them. This behaviour will eventually enhance the performance of the fleet in presence of obstacles in the environment, by reducing the impact they have on the fleet capacity as all the agents that belong to the fleet will then be able to use the received information and optimize their path calculation accordingly. At the moment, in fact, there is no functionality in place that allows a fleet of AGV to handle the presence of obstacles in the facility in a collaborative way.

A big role in this collaborative system implementation has been played by the increasing interest that decentralization has assumed over the recent years. The robots in the simulator on which the tests were performed are organized in a decentralized fashion, thanks to the Coaty platform that was developed by our partner Siemens as part of the collaborative embedded systems (CrESt) project and that is currently in the implementation stage on the AGVs at InSystems Automation.

As the process of decentralization moves the "intelligence" of the fleet from a central authority to the robots in the system. For example the robots are capable of tracking their battery status, or to perform an auction to decide which robot should perform a certain task when it is issued, and use these informations to optimize the performance of the fleet.

With this in mind, this thesis has the objective of bringing this concept of collaboration even further, where robots not only share the information regarding their status but are also capable of sharing information regarding the external environment.

1.1 About the AGVs

An AGV (Automated Guided Vehicle) is a mobile robot that is mainly used to move a large quantity of material, pallet loads between shipping/receiving docks and storage racks and move work-parts between machine tools and stations [12]. Their travel velocity is usually lower than the typical human walking speed, and there are several safety features which are implemented in order to make it possible for AGVs to work among people, such as obstacle detection with a related emergency stop trigger, emergency bumpers, warning lights and sounds, and emergency stop buttons. The application range of the AGVs is very wide. Nowadays they are being used in pharmaceutical, chemical, manufacturing, paper and print industries, as well as hospitals and warehouses.

AGVs play a huge role in terms of material handling optimization and are becoming more sophisticated over time as well as their capabilities to accomplish more complex tasks. The AGVs most common navigation systems are:

- Wire navigation: AGVs navigate along a continuous wire embedded into the floor of the facility. The vehicles detect the wire by means of an antenna.

- Magnetic navigation, that makes use of magnetic sensors equipped in the vehicle in order to follow a defined track made by a magnetic tape.
- Magnetic spot navigation, in which The AGVs go from a spot to the following one using sensors and controls such hall-effect sensors, encoders, counters, gyro sensor and other kinds of encoders to calibrate against steering angle errors.
- Optical Navigation, that makes use of cameras to properly identify and follow a strip located on the floor of the facility.
- Laser-Guided Navigation, in which the vehicle, equipped with a navigation laser positioned on top of a pole that interacts with target reflectors positioned in the AGV working area, The positioning of the vehicle is then computed via triangulation.
- Natural Navigation (SLAM or LiDAR navigation), where the AGVs are able to map and localize themselves into the environment by means of Simultaneous Localization and Mapping algorithms (SLAM).
- Vision Navigation, which makes use of triangulation of the signals coming from stereo cameras or makes use of time of flight cameras to perceive the surrounding environment.

Although the investment is very high at the beginning, in the long term the maintenance costs have proved to be way lower than hiring long term employees.

1.2 Natural navigation

The natural navigation system is based on a SLAM algorithm, that allows them to locate themselves in space. SLAM (simultaneous localization and mapping) is a technology for AGV mobile robots to create maps in completely unknown environments under the condition of uncertain position, and to use maps for autonomous positioning and navigation. SLAM has become the preferred advanced navigation mode for many AGV manufacturers.

SLAM mainly relies on LiDAR (Light Detection and Ranging) and camera to perceive environmental information. The AGV car of LiDAR can be positioned and mapped independently. It has high measurement accuracy, long measurement distance and not easily disturbed by external environment [12].

The transport robots that are developed at **InSystems Automation GmbH** navigate automatically thanks to a laser scanner, without ground loops or wall reflectors, and react to changes in their work environment. The AGVs are safe, drive around obstacles and avoid collisions with persons by going for alternative routes. Every AGV calculates the optimal path to its destination on its own with the help of a stored environment map.

FIGURE 1.1: ProANT AGVs produced by InSystems

InSystems automation

proANT
Automatisch Navigierendes Transportfahrzeug

FIGURE 1.2: InSystems and proANT logos

1.3 About InSystems Automation GmbH

The thesis project has been developed at **InSystems Automation GmbH** in Berlin, from April to September 2019. InSystems Automation develops innovative automatic solutions and special machines for production, material flow and quality tests.

The company was founded in 1999 by the managing directors Henry Stubert and Torsten Gast and grew constantly since. By now, more than 50 employees work at InSystems. The company is located in the science center Berlin-Adlershof and has offices, a workshop, an online shop and a showroom for industry 4.0.

Since 2012, InSystems specialized on the production of autonomous navigating transport robots, which are designed for loads from 30 to 1.000 kg, according to customer request, and implemented as a fleet into an existing production control. The transport robots are developed under the name proANT [8].

The company features a certified competence since 2006 and some of its partners are Siemens and Wago.

1.4 Case of study

The objective of this thesis is to implement a functionality which allows a fleet of robots to share information about the presence of obstacles. In particular the focus is on natural navigation AGVs, such as the ones that are developed at **InSystems Automation GmbH**. The tasks that have to be performed by the system are:

- The AGVs should be able to identify the presence of an obstacle that is blocking a pathway in the facility and to share the information relative to this obstacles. This means that a first step in the solution is to share the information only when the obstacle which has been identified is blocking a pathway, avoiding as much as possible to share information relative to the other obstacles.
- When an obstacle which information has been shared is no longer blocking a pathway in the facility and this is detected by any of the robots in the fleet (regardless if the robot is the one who first detected the presence of the obstacle or not), the information regarding the absence of the previously detected obstacle has to be shared.
- All the robots in the fleet should be affected by the received information regarding the presence of an obstacle.

1.5 Thesis structure

The structure of the thesis report has been divided in six chapters. The second chapter addresses the background informations that are needed in order to understand the functionality that was developed. In particular the first part will explain some basic concepts about the "Robot Operating System" (ROS) that is deployed on the AGVs, with particular focus on the aspects which are more relevant to the present work, such as:

- The navigation stack and the move_base package, that are responsible of the navigation of the AGV, with particular attention to the costmap_2d package that provides data about the environment that is then used by the planners in order to allow the AGV to reach an assigned goal position, and to the *obstacle_layer* that is responsible of detecting the presence of obstacles.
- The tools that were used in order to simulate the implemented functionalities.
- The Coaty framework which was used to perform the sharing of the information about the detected obstacles among the different AGVs in the fleet.

Chapter 3 further explains the specifics regarding the sharing functionality and the criteria that were adopted in order to perform the assigned task, then shows a first approach to the problem, based on the sharing of areas associated to the presence of obstacles.

Chapter 4 shows a new approach that was adopted in order to perform the sharing. The approach, though partially based on the one explained in the previous chapter, presents some substantial differences with respect to the first one. It is based on the sharing of the cells of the map that have been associated to the presence of an obstacle. Those cells are associated to subsections of the costmap that are monitored and updated according to criteria that are extensively discussed in the chapter.

In chapter 5 the results of the test that were performed are shown and documented. These tests provide an immediate way to verify and understand the functionalities that were implemented.

Chapter 6 summarizes and discusses the obtained results while opening the way to future improvements.

Chapter 2

State of the Art

2.1 About ROS: an overview of the system

ROS, which acronym stands for Robot Operating System, is an open source meta-operating system. The term "meta-operating" system underlines the fact that it is not an operating system in the traditional sense of process management and scheduling; rather, it provides a structured communications layer above the host operating system of a heterogeneous computer cluster [5].

The ROS runtime “graph” is a peer-to-peer network of processes that can be in principle be distributed across different machines. These processes, referred to as "nodes", are loosely coupled and exchange data by means of the ROS communication infrastructure. ROS implements several different styles of communication, such as synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server.

The main purpose of ROS is to support code reuse in robotics research and development. ROS is a distributed framework of processes (referred to as Nodes) that enables executables to be individually designed and loosely coupled at runtime. These processes can be grouped into Packages and Stacks, which can be easily shared and distributed. ROS also supports a federated system of code Repositories that enable collaboration to be distributed as well.
In order to achieve this goal of sharing and collaboration, there is a set of sub-goals that have been achieved:

- Thin: ROS is designed to be as thin as possible so that code written for ROS can be used with other robot software frameworks. A corollary to this is that ROS is easy to integrate with other robot software frameworks.
- ROS-agnostic libraries: the preferred development model is to write ROS-agnostic libraries with clean functional interfaces.
- Language independence: the ROS framework is easy to implement in any modern programming language. It has been already implemented in Python, C++, and Lisp, and there are experimental libraries in Java and Lua.
- Easy testing: ROS has a built-in unit/integration test framework called "rostest" that makes it easy to bring up and tear down test fixtures.

- Scaling: ROS is appropriate for large runtime systems and for large development processes.

For the purpose of the present thesis work the "language independence" property has been of fundamental importance, as the part that was developed relatively to the *costmap_2d* package has been implemented in C++, whereas the part relative to the communication has been developed in TypeScript and in Python. The streaming of data among the different nodes that were developed or modified has been performed mainly by means of Topics.

2.1.1 Nodes

A node is a process that performs computation [4]. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC (Remote Procedure Call) services, and the Parameter Server. These nodes are meant to operate at a fine-grained scale; a robot control system will usually comprise many nodes.

The use of nodes in ROS provides several benefits to the overall system. There is additional fault tolerance as crashes are isolated to individual nodes. Code complexity is reduced in comparison to monolithic systems.

All running nodes have a graph resource name that uniquely identifies them to the rest of the system. Nodes also have a node type, that simplifies the process of referring to a node executable on the *filesystem*. These node types are package resource names with the name of the node's package and the name of the node executable file. A ROS node is written with the use of a ROS client library, such as *roscpp* or *rospy*.

2.1.2 Messages

Nodes communicate with each other by publishing messages to topics. A message is a simple data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays [4].

Nodes can also exchange a request and response message as part of a ROS service call. These request and response messages are defined in srv files. Message ".msg" files are simple text files for specifying the data structure of a message. These files are stored in the "msg" subdirectory of a package. The code for including these messages as libraries in the code is automatically generated.

2.1.3 Topics

Topics are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic. Topics are intended for unidirectional, streaming

communication [4]. Each topic is strongly typed by the ROS message type used to publish to it and nodes can only receive messages with a matching type.

This mechanism of communication is asynchronous, as the message which is generated is stored in a buffer and is then read by the subscribers when the relative node is executing.

2.1.4 Services

The publish / subscribe model is a very flexible communication paradigm, but its many-to-many one-way transport is not appropriate for RPC request / reply interactions, which are often required in a distributed system. Request / reply is done via a Service, which is defined by a pair of messages: one for the request and one for the reply. A providing ROS node offers a service under a string name, and a client calls the service by sending the request message and awaiting the reply. Client libraries usually present this interaction to the programmer as if it were a remote procedure call [4].

2.1.5 Master

The ROS Master provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer. The Master also provides the Parameter Server [4].

2.1.6 Parameter server

A parameter server is a shared, multi-variate dictionary that is accessible via network APIs. Nodes use this server to store and retrieve parameters at runtime. As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters. It is meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary [4].

2.2 The navigation stack

The Navigation Stack is responsible of taking information from odometry and sensor streams and outputs velocity commands to send to a mobile base [7]. A stack is organized as a set of packages. Whereas the goal of packages is to create minimal collections of code for easy reuse, the goal of stacks is to simplify the process of code sharing. The packages which are part of the navigation stack and are of particular relevance to the developed functionality are the move_base and the costmap_2d.

The move base package provides an implementation of an action that, given a goal in the world, will attempt to reach it with a mobile base [3]. The move base node links together a global and a local planner to accomplish its global navigation

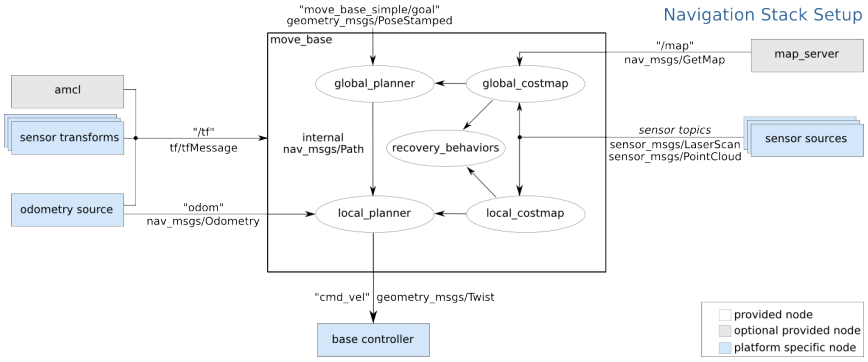


FIGURE 2.1: Navigation stack setup

task. The specific planners used as global and local planners must adhere to the interfaces specified in the nav core package. The move base node also mantains two costmaps, one for the local and one for the global planner.

The two different planners perform different functions and the result of their work combined is a trajectory for the mobile base:

- The global planner, given the global costmap and the assigned goal position in the working environment, is responsible of computing a valid path to bring the robot from the starting pose to the goal pose, where the path is simply a geometrical entity and the poses are defined as a set of coordinates in an orthonormal frame, followed by a quaternion describing the orientation. A path is valid when it does not cross obstacles that the global costmap is aware of.

- The local planner, given the local costmap and the assigned path computed by the global planner, is responsible of sending the command velocities to the mobile base.

The joint operation of the two planners gives as a result a trajectory, defined as a path with an assigned velocity vector. In accord with the different functionalities of the two planners, also their frequencies of operation differ. The frequency of the local planner which has been set for the simulation is 10 Hz, as it needs to compute the optimal trajectory on the basis of the global path and of the presence of the data which are coming from the local costmap, which carries information regarding the presence of obstacles which were not known at the moment of the path computation. The frequency of the global planner is sensibly lower with respect to the one of the local planner. The path is, in fact, generally computed just upon the assignation of a goal position and in case the robot is not able to reach the assigned goal due to the presence of obstacles which are blocking the path.

In order to optimize the robot operations the frequency of the global planner has been set to values in a 0.4 to 1 Hz range. In this way the robot is able to react more promptly to the presence of obstacles that makes a path invalid without necessarily having to perform the recovery behaviours. This would also allow the robots that receive the information about the presence of an obstacle to recalculate the path when a goal has been already received.

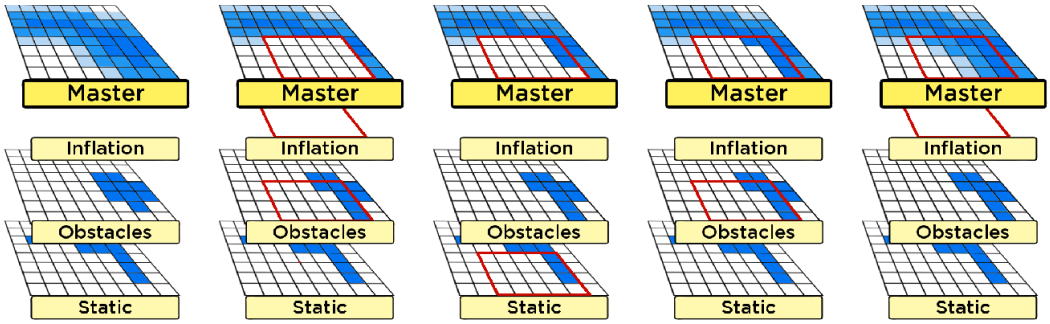


FIGURE 2.3: Costmap update example

2.2.1 The recovery behaviours

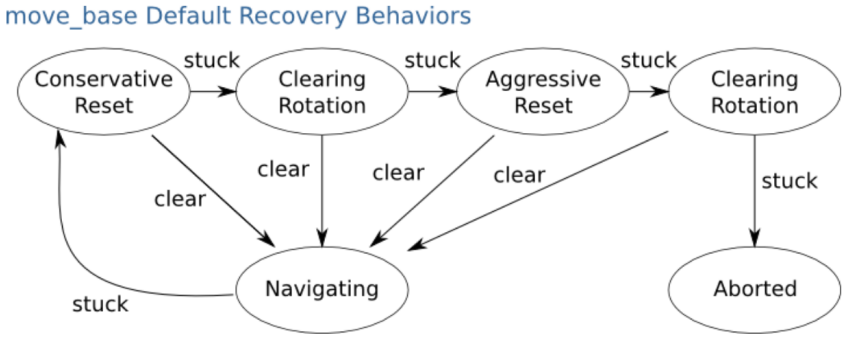


FIGURE 2.2: Default recovery behaviours of the move base

The recovery behaviours are triggered when the robot perceives itself as stuck. The operations that are performed are explained on a conceptual level in figure 2.2. In the moment the robot first perceives itself as stuck it will performed the so called "Conservative Reset", that consists in clearing from the map of the robot the obstacles outside of a user-specified region. At the end of this and all the other recovery behaviour operations a check will be performed in order to verify whether it is still stuck or if it is capable of reaching the assigned goal.

If, after performing the "Conservative Reset" the robot is still stuck, an in-place rotation will be performed in order to clear out space. If this too fails, the robot will more aggressively clear its map, removing all obstacles outside of the rectangular region in which it can rotate in place. This will be followed by another in-place rotation. If all this fails, the robot will consider its goal infeasible and notify the user that its path computation has aborted [3].

2.2.2 The costmap

The costmap is a configurable structure that retains information about where the robot should navigate in the form of an occupancy grid. The costmap uses sensor data and information from the static map to store and update information about obstacles in the world. The space is represented as an array of cells, each of those with an associated cost. An higher cost associated a cell discourages the planner from computing a path that leads the mobile base to cross the area which associated to it. If the cost of the cell is set to "lethal obstacle", the planner won't compute a path

crossing that cell as it would lead to a crash.

The purpose of the costmap is to provide the two planners with a configurable representation of the surrounding environment. As the planners have different purposes, the possibility of configuring each of the costmaps independently allows to tailor them to the tasks that they are designed to perform [2].

Each costmap can use different plugins as layers, each of those performing a separate task. For example, the configuration which was adopted was using three different layers:

- The *static layer*, which takes informations from the map obtained during the mapping procedure, provides the costmap with the static layout of the plant on which the AGV operates.
- The obstacle layer uses the information retrieved via the laser scanner to detect the presence of deterministic obstacles which were not known a-priori as in the static_layer.
- The *inflation layer*, that has the task of increasing the costs associated to cells which are near to obstacles, resulting in the creation of a buffer zone around lethal obstacles ("lethal obstacle" is the cost associated to a cell the crossing of which implicates a crash) this is done to discourage the robot to generate paths that are too close to obstacles.

The list of layers which are active in the costmap is defined as an "ordered list of layers", which means that the order in which the layers are scanned through depends on the order that was assigned to them in the configuration files and it will stay the same unless it is manually modified by the user.

The result of the merging of all the information relative to each single layer is stored inside the "master grid" of the costmap, which is then used for the related planner. The process of updating the costmap is performed with two passes through the ordered list of layers.

In the first pass, the *updateBounds* method is called, each layer is polled to determine how much of the costmap needs to be updated. When the layers are iterated over, they are provided with the bounding box that the previous layers need to update. Each layer can expand the bounding box as necessary according to the areas that carry new information. The bounding box is initialized as empty at each new iteration through the set of layers. The result of this operation is a bounding box representing the area of the costmap that needs to be updated.

In the second pass, the *updateValues* method is called, during which each successive layer will update the values within the area of the bounding box of the master costmap.

Some of the layers can mantain their own version of the costmap. For example the *obstacle_layer* keeps a private costmap of the same size as the master costmap to store the results of all the previous ray-tracing and marking steps. This is done in order to reduce how frequently the costmap must recalculate values that had previously been overwritten, as the values in the private costmap are only accessible to

the particular layer. The private costmap is, then, immune to the loss of information which may be caused by another data source writing over it.

Figure 2.3 shows an example of the update process of a three layers costmap, made of the static, the obstacles and the inflation layers as shown in the first figure from left. The static and the obstacle layers keep their own copies of their private costmap, while the inflation layer does not. As previously explained, the first step of the update process is performed calling the *updateBounds* method on each layer, following the ordered list and resulting in the bounding box shown in red in the master in the second image. In the following three images the updating process of the layered costmap is shown as it is performed on each single layer, resulting in the master which is shown in the last image on the right.

The different layers operate independently from one another but each layer will set the values in the master grid. By default the final value that a specific cell will assume in the grid is the maximum among the values that have been assigned to it by all the layers to that specific cell [10].

the extension of the area which is associated to a single cell is defined by the parameter called "resolution".

2.2.3 The obstacle layer

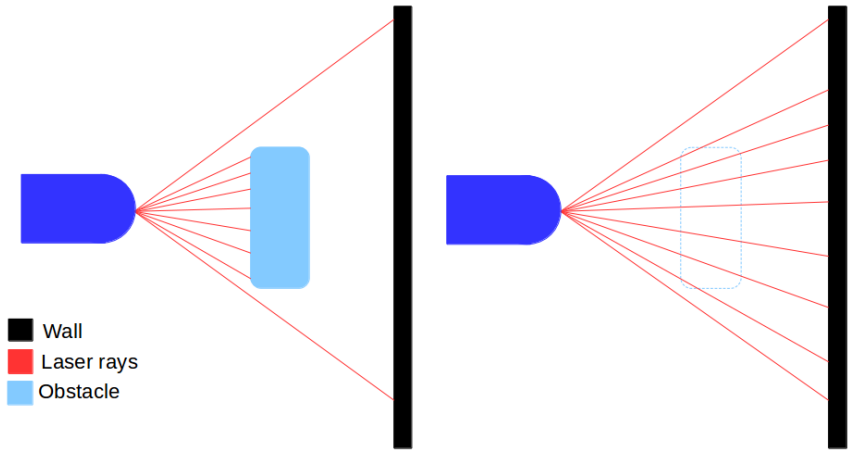


FIGURE 2.4: Marking and clearing operation, graphical representation

The operation of the obstacle layer is of the utmost importance for the aim of the thesis, since it implements the mechanism on how the robot perceives its surroundings based on the information coming from the sensors. Each sensor is used to either mark or clear an obstacle from the costmap.

Marking

Marking is the operation of inserting an obstacle in the costmap. This operation is performed on the endpoint of a laser beam that hits a surface, a graphical representation of it is shown in figure 2.4 in the left image. In particular, when the sensor

perceives an obstacle inside the range for obstacle detection (which is, in principle, lower that the maximum range for the sensor as the precision of the point which is associated to a laser reading in the visualization is affected not only by the laser scanner precision but also by the positioning precision of the robot and, therefore, of the scanner), it marks the corresponding pixel as an obstacle.

The identification of the points to mark in the AGVs produced by InSystems is done through the use of the information obtained from the laser scanner. The data are retrieved from the scanner as a *LaserScan* type of message, containing information about the minimum and the maximum angles of the observation, the angle increment for each laser beam, the maximum and minimum range for the observations and then two arrays containing the detected ranges and the associated intensities (the intensities are useful in case of the use of reflectors for the navigation).

This information is then transformed into a *PointCloud* type, which contains the tri-dimensional cartesian coordinates of the points of the observation. These coordinates are obtained considering the informations obtained by the scanner, such as points positions and angles and the frame from which the points were observed (i.e. the frame which origin corresponds to the sensor position). These points then go through a frame transformation in order to have a point cloud in the coordinate frame of the costmap that is being used.

The detected points, then, before being marked in the costmap, go through a set of checks:

- If the z coordinate of the point is too high the point will be discarded.
- If the point is too distant from the sensor the point will be discarded.
- If the point is outside of the map it will be discarded.

Regarding the second check, the necessity of defining a range for the marking of cells associated to obstacles in the costmap is not related to the precision of the laser readings but to the accuracy in the positioning of the robot in the environment. To give some numerical data the sensors which are mounted on the AGVs produced by InSystems have a range of 30 m but the range for marking a cell in the costmap associated to an obstacle has been set to 2.5 m. This exemplifies the fact that the laser precision is not the bottleneck in the accuracy of the readings.

Clearing

The clearing operation is based on a 2D implementation of Bresenham's algorithm. It is a line drawing algorithm that determines the points in a two dimensional raster that should be selected in order to form a close approximation to a straight line between two points. The two points, in the obstacle layer implementation are represented by the origin of the sensor and the ending point of the observation. For example, in the main use case of a laser scanner, the origin is the scanner position in the costmap and the ending point corresponds to the point where the laser has hit a surface. An intuitive representation of this mechanism is shown in figure 2.4 in the right image. All the costs of pixels of the map belonging to this line that are inside of a specific range (i.e. the clearing range) will be set to the value corresponding to

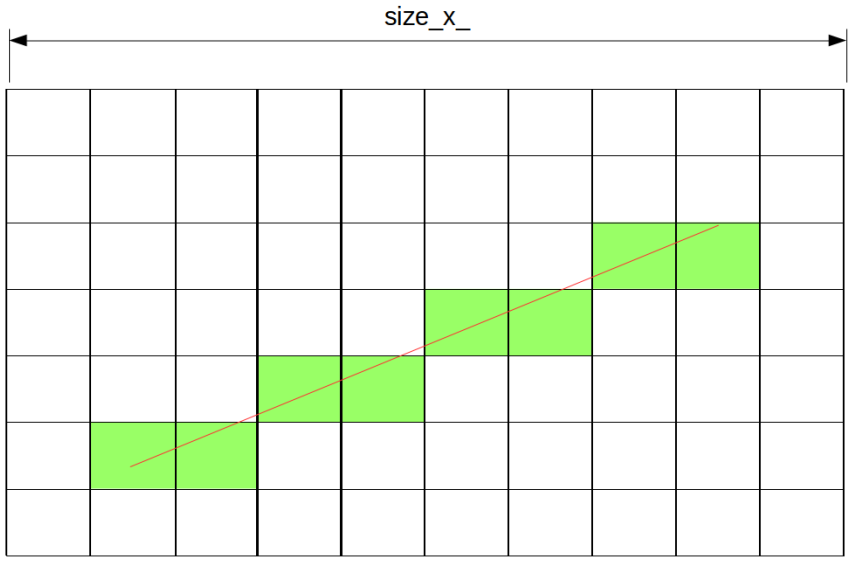


FIGURE 2.5: Graphical representation of Bresenham's algorithm

the free space.

Given the coordinates of two points $A \equiv (x_1, y_1)$ and $B \equiv (x_2, y_2)$. The task to find all the intermediate points required for drawing the line "AB" on the map. Note that every pixel has (unsigned) integer coordinates. Below are some assumptions to keep the algorithm simple.

- The line is drawn from left to right.
- $x_1 < x_2$ and $y_1 < y_2$
- Slope of the line is between 0 and 1. We draw a line from lower left to upper right.

Considering the equation of a line in the form $y = mx + c$, where m is the slope and c represents the intercept on the y axis, the idea of Bresenham's algorithm is to avoid floating point multiplication and addition in the computation of $mx + c$.

At each step of the iteration the x is increased and the Y_{k+1} value should assume the value among $Y_k + 1$ and Y_k that corresponds to the point which is closest to the original line. The value that Y_{k+1} assumes depends on the value of the slope error. If this value becomes greater than 0.5 , the line has moved one pixel upwards, therefore $Y_{k+1} = Y_k + 1$ and the new error will become equal to "old error -1".

In order to avoid floating point arithmetic, the slope $m = (y_2 - y_1)/(x_2 - x_1)$ is multiplied for its denominator $(x_2 - x_1)$. In order to get the comparison right, also the slope error gets multiplied for $(x_2 - x_1)$ and then again is multiplied for a factor 2, to compare with the two integer numbers 0 and 1 (thus avoiding comparison with 0.5). The initial value of *slope_error_new* is $2 * (y_2 \check{} y_1) \check{} (x_2 \check{} x_1)$.

In the real implementation the second and the third hypotheses are not always satisfied, therefore the algorithm has to be able to handle also negative slopes and slopes which absolute value is greater than 1.

The method that sets the correct values for the Bresenham's line algorithm to be properly called in its costmap implementation is called *raytraceLine* (Figure 2.10) and receives as input the maximum lenght that the line can assume (based on the ray-tracing range parameter) and the initial and the final points of the specific laser dot in the observation (respectively the sensor position and the point in which the laser hit the wall). Than it computes $dx = x_1 - x_0$ and $dy = y_1 - y_0$ and verifies which one has a bigger absolute value. This one will become the dominant dimension. The dominant dimension will then be scaled according to the maximum length that the line can assume and the *bresenham2D* method will be called. Note that in the definition of the offset there is the possibility to define negative slopes.

The $offset\_dy$ is computed as $offset\_dy = sign(dy) * size\_x\_$ (The parameter $size\_x\_$ can be graphically seen in figure 2.5).
This is done because the costmap array only has one dimension, the rows of the array are just concatenated and the value $size\_x\_$ keeps track of the length of what would have been the number of columns if the costmap indexes were arranged as a two dimensional array.

Figure 2.10 shows the details of the implementation of the bresenham2D method. the parameters in the function that are ending in "a" will be assigned to the dominant dimension, and all the parameters ending in "b" will be assigned to the other dimension. the offset parameter refers to the initial point in the array and the $offset\_a$ and $offset\_b$ parameters refer to the increment of the two dimensions.

2.3 About the simulator

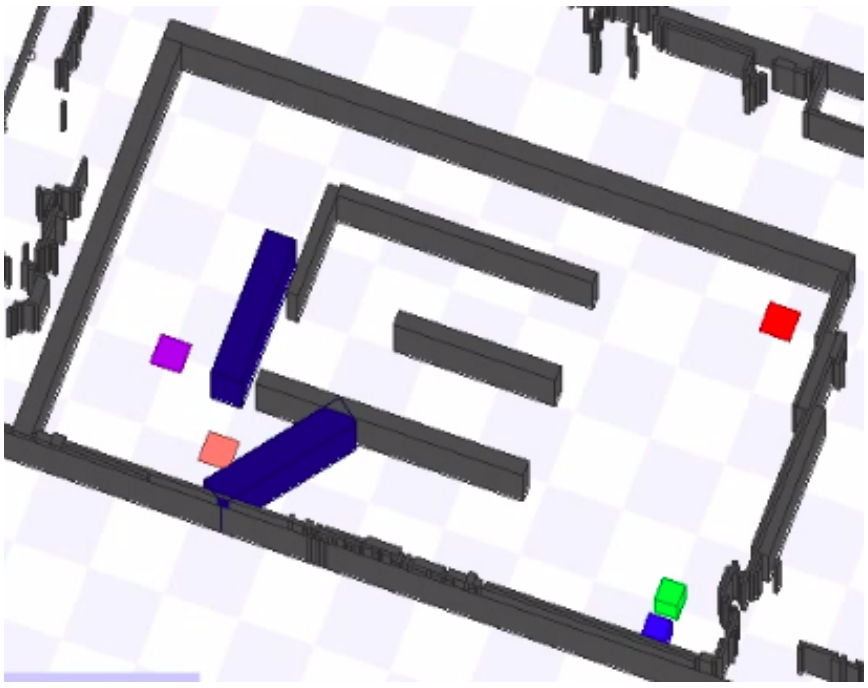


FIGURE 2.6: Simulation Environment

The functionalities that have been developed have been tested using the *Stage* simulation environment. Stage was designed with multi-agent systems in mind, so it provides fairly simple, computationally cheap models of multiple devices rather than attempting to emulate any device with great fidelity. This design is intended to obtain a compromise between conventional high-fidelity robot simulations and the minimal simulations. Stage is intended to be just realistic enough to enable users to move controllers between Stage robots and real robots, while still being fast enough to simulate large populations.

In the figure 2.6 a representation of the simulation environment is shown. The blue and green cubes on the lower right of the image represent the AGVs in their starting positions. The dark grey areas represent the walls layout (those are the representation of the a-priori known environment which is normally obtained through mapping), the orange, purple and red squares represent the goal positions and the navy blue parallelepipeds represent the obstacle positions. The information regarding the latter is not known a-priori and those obstacles will be used with the purpose to block a pathway in order to test the implemented functionalities.

2.4 Visualization in Rviz

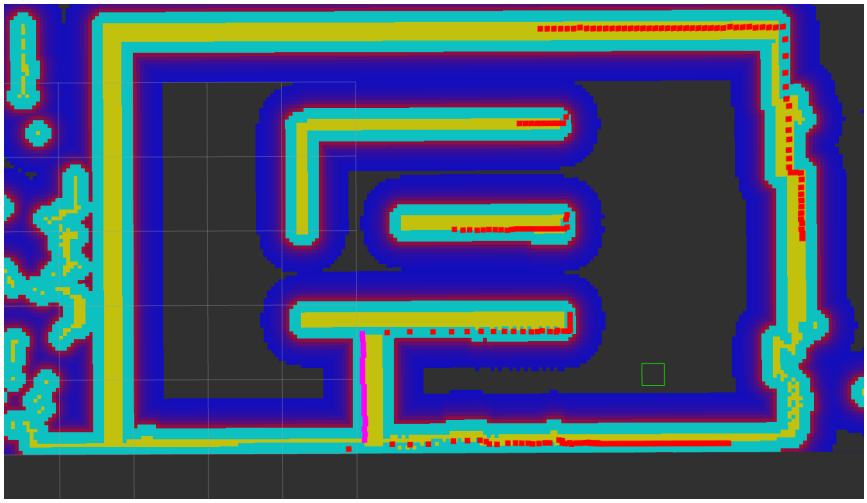


FIGURE 2.7: Example of visualization in Rviz

Rviz, which name stands for "ROS Visualization", is a 3D visualization environment. It provides a visualization of how the robot perceives the surrounding environment and it also gives information about its "thinking" process. The use of this visualization tool provides a very useful instrument for the debugging phase [6].

Figure 2.7 shows an example of a 2D visualization in Rviz. The robot footprint is indicated by a green square, the point cloud associated to the readings of the laser scanner is shown as a set of red points, the points identified as "lethal obstacles" which is the name associated to obstacles in the costmap, are displayed in yellow, while the gradation of cyan and blue are associated to the cells which cost has been increased by the inflation layer.

There are different possibilities for the choice of the set of colors for representing a costmap in Rviz. The costmap is actually set as an occupancy grid which cells are associated to a value that represents a different tone of grey in a fashion similar to a grey-scale representation. The normal set of colors that could be associated to a similar representation makes the distinction between the cells which are set as obstacles in the map and the surrounding areas difficult to identify. By using this set of colors for the costmap this distinction is definitely more human-readable.

Since in order to set up an obstacle in the simulator it was necessary to define it in a file which is also used by the robot, the laser scan points associated to that obstacle are displayed in pink. This affects only the way these obstacles are displayed, and not the reaction of the system to their presence, meaning that they are still not a-priori known, as it will be shown in the Chapter related to the tests that were performed.

2.5 CrESt

FIGURE 2.8: CrESt logo

Collaborative embedded systems will have a strong impact on significant technological development and re-define current factory processes. Tomorrow's factories will be able to flexibly react to changing production factors thanks to adaptive system architectures. Learning machines will take over the work of central order management systems and organise their transport order coordination autonomously. In-Systems Automation is one of 22 international partners from industry and science who take part in the research project *CrESt* (Collaborative Embedded Systems) [9].

CrESt, as a research project funded by the Federal Ministry for Education and Research, is concerned with the development of collaborative embedded systems, with a focus on dynamic software systems. The project's goal is to define methods for model-based descriptions of dynamic, scalable applications.

The research project is divided in six co-called "Engineering Challenges" and six interdisciplinary topics. The first three Engineering Challenges are concerned with the overarching question of how to best design the architecture of flexible, dynamic and adaptive systems.

The cases where such technology would be useful are many: autonomous robots, learning control systems and production machines are only a few examples. Such systems are meant to safely collaborate and take over repetitive work previously done by humans to optimize these processes in a goal-oriented manner.

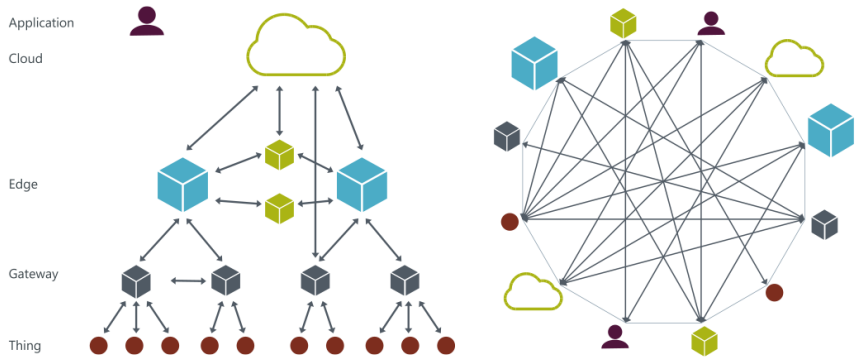


FIGURE 2.9: From cloud centered, hierarchical IoT to decentralized, non-hierarchical IoT

2.6 The Coaty communication framework

The concept of multiple autonomous vehicles collaborating towards the completion of a task is emerging as a key technology in mobile robotics that has been the focus of intense research over recent years. As system complexity tends to increase over time, a decentralized control approach is configuring itself as the best solution. With respect to a centralized server control, this approach counts several advantages, such as:

- removal of single point failure: in a centralized structure, when the server responsible of assigning the jobs fails, the whole system is shut down. In a decentralized system this does not happen since the jobs are handled independently by the agents.
- it is robust against failures of an agent as the system can continue to function with a reduced capacity.
- intelligence: in the centralized configuration the robots have no way of independently handling the assigned tasks as they depend on the server. In decentralized control the robots are provided with the tools to independently handle job assignation and completion.
- scalability: in a centralized control system, if the number of robots increases, the workload on the server will increase proportionally. Since in a decentralized control the job assignation and the control of the execution of the different tasks is handled by the agents, the increase in number of the agents in the system does not correspond to a significant increase of the workload on the single elements.

This approach was made possible through the use of *Coaty*, a middleware that provides the agents of the system with the capability of independently handling the information coming from the other agents without having the need for a central device with the task of handling all the communication.

Coaty is a collaborative IoT framework in JavaScript for Node.js and browsers. Its aim is to build distributed applications out of decentrally organized application components, defined as Coaty agents, which are loosely coupled and communicate with each other in (soft) real-time [1].

Coaty uses event-based communication flows with one-way/two-way and one-to-many/many-to-many event patterns to realize decentralized producer-consumer scenarios. Thereby, Coaty combines the characteristics of both classic request-response and publish-subscribe communication. In contrast to classic client-server systems, all Coaty participants are equal in that they can act both as producers/requesters and consumers/responders.

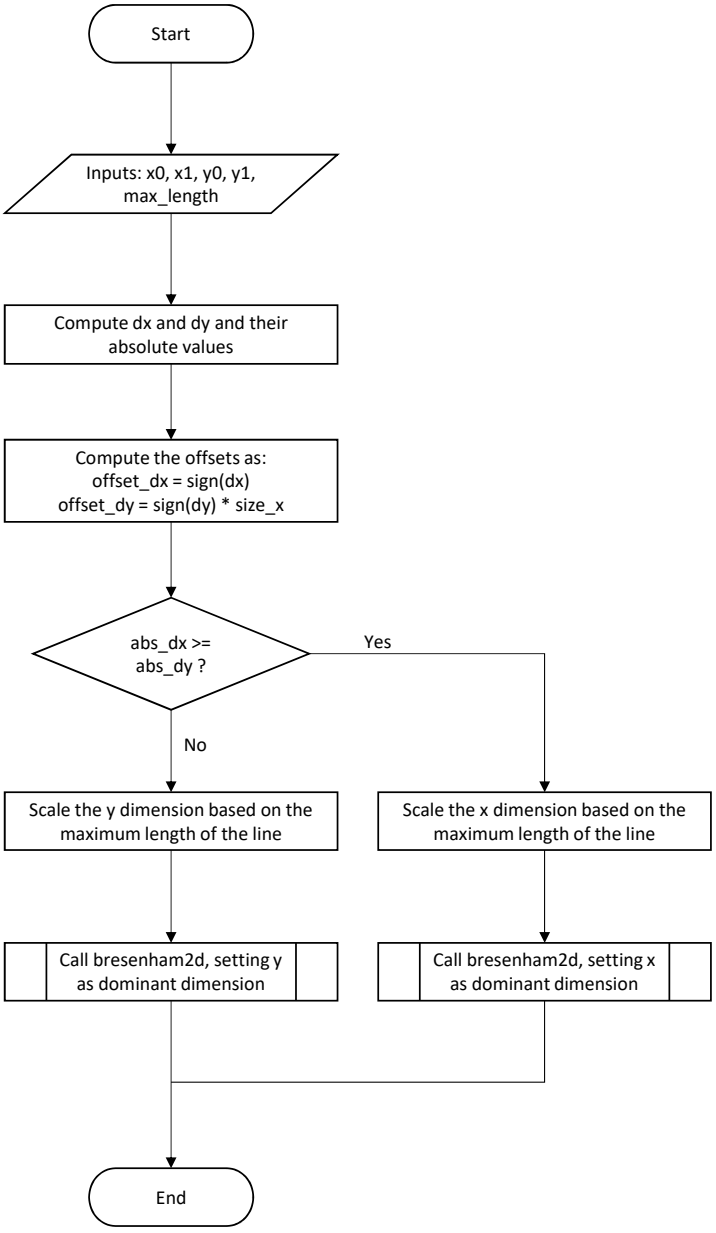


FIGURE 2.10: raytraceLine flow chart

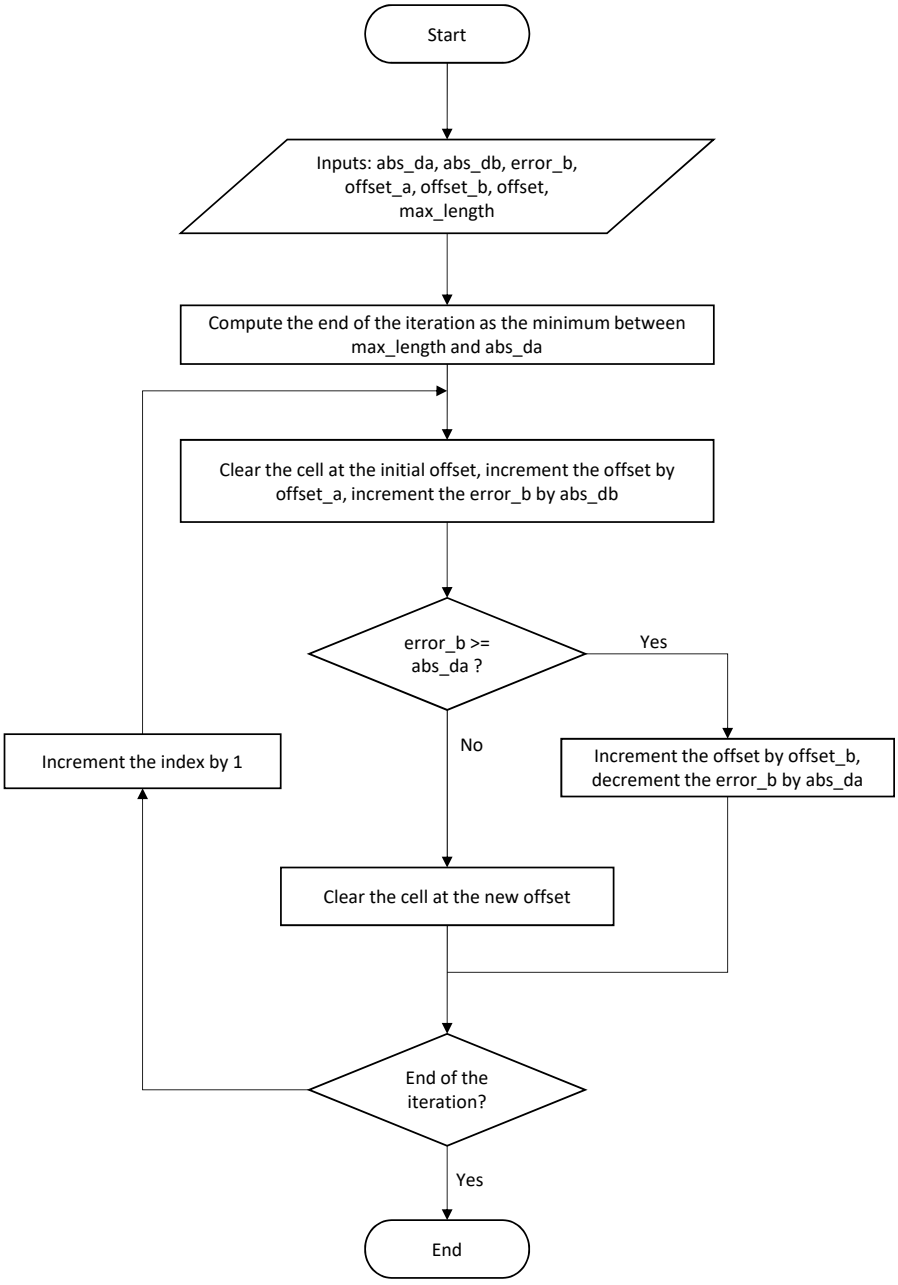


FIGURE 2.11: bresenham2D flow chart

Chapter 3

Specifics and First Approach to the Solution

3.1 Introduction

In order to perform the sharing of the information related to the presence of obstacles which are blocking a pathway in the working environment, two different approaches were investigated. The two approaches address the problem via the use of different solutions for the data structure that has been adopted for storing the information relative to the obstacles that prioritize different aspects of the sharing. The part related to the identification of the cells in the costmap that are associated to obstacles, as also the event that triggers the sharing, haven't seen any relevant modification in the optimization phase.

The first solution, which will be fully explained in this chapter, addresses the problem of the obstacle sharing by means of an approach based on the use of polygons to store the information. As a first solution this approach has used rectangles to identify the position of obstacles in the costmap, prioritizing the minimization of the amount of shared data over the precision of the representation of the detected obstacles. The natural step towards the implementation of a so defined solution on the AGVs would be the use of convex hull polygons. This latter solution, though, has not been investigated because of the technical problems that were found relatively to the implementation of the polygonal based representation of the obstacles.

The solution which was later found and identified as the best way to proceed towards the implementation of the sharing functionality is closer to the mechanics behind the implementation of the costmap, which is based on cells and not on the use of polygons to identify the obstacles. This last solution, discussed after this chapter, is based on the sharing of the cells associated to the detected obstacles and prioritizes the precision of the representation of the obstacles over the minimization of the data shared across the fleet of AGVs.

This chapter will describe the first solution that was adopted as a significant part of this solution was used as the basis for the implementation of the second one and as it will further explain the reasons that have led to the use of a different approach for the data structure that was used for the sharing.

3.1.1 Criteria for performing the sharing

The work of this thesis has the objective of enhancing the way robots handle uncertainty in a plant, in particular it focuses on making the AGVs receive information about obstacles when the scanner is not able to identify those obstacles by himself and making the robot react in a suitable way.

Since the number of obstacles that are identified during the operation of the robot is very high, sharing all of them could cause an excessive flow of data over the network, thus the necessity to identify an optimal criterion for triggering the obstacle sharing functionality arises.

With this consideration in mind, the first problem to address for the sharing functionality is to define a specific subset of all the detected obstacles which sharing can lead to an improvement of the way in which the fleet reacts to the presence of uncertainty in the environment related to the presence of obstacles which cannot be a-priori known. The obstacles which information needs to be shared are those which can have a significant impact on the path computation of the robot. For this reason the criterion which was chosen is to share only the information related to obstacles which are completely blocking a pathway. When a robot is led by is global path in a pathway which is blocked, the local planner will raise a flag indicating the failure in the computation of the velocities for the local base to follow the assigned path. Catching this flag indicates the presence of something that is blocking the path that was not known to the global planner, and therefore this signal will be used as a trigger for the sharing.

As the sharing functionality has to operate only to optimize the global path calculation, the functions that regulate the obstacles sharing have to operate only for the global planner, meaning that only the global costmap must be affected.

If the behaviour that avoids obstacles of an AGV is set, the robot tends to identify a vast number of obstacles during its operations. The identification of obstacles is done in the obstacle layer based on the readings of a laser scanner. The scanner has a range of 30m and an angular span of 270 degrees, but the points are added to the specific costmap according to a parameter that for the AGVs is set to 2.5m in order to limit the effect of a suboptimal angular localization of the robot.

Since each layer operates independently, the obstacle layer will mark the points that the laser scanner hits even if those are part of the wall. This behaviour is suitable for the normal operations of the single robot but when it comes to sharing the detected obstacles this would lead to sharing an increased amount of unnecessary information (sharing parts of the static layout of the plant is redundant since all the AGVs share the same map according to the plug-and-play functionality which has been developed at InSystems as part of the CrESt project).

In order to share the information regarding the obstacles which are compliant with the chosen criteria, there is the need to keep track of the points that are detected by the laser scanner in a suitable way. The *obstacle_layer* keeps the information in an array with a fixed length which is its private costmap. As we don't need to store information regarding all the cells in the costmap, it was chosen to use a vector to store the information regarding the observations. The details about the usage of this

vector are explained in the following.

3.2 The vector

The points that are detected by the laser scanner are added to a vector. Adding a cell to this vector means that the information regarding this cell can potentially be shared among the different AGVs. The downfall of using a vector is the dimension handling. It is important to keep the dimension of this vector as low as possible in order to reduce the time that is necessary to scan this vector. The criteria for adding a point to this vector are therefore more conservative with respect to the ones used for normal operations (explained in chapter 2 in the obstacle layer section). Moreover, the way in which the information of a cell is removed from is not only limited to the use of the raytracing algorithm.

3.2.1 Adding cells to the vector

As the layers usually operate in an independent way, they are usually unaware of the presence of the other layers in the configuration. This results in the possibility of having redundant information marked on the costmap. This behaviour doesn't represent an issue when a robot has to operate independently as the result of having redundant information regarding a cell won't affect in any way the navigation behaviour. On the other hand, for performing the sharing functionality it can represent an issue. In the obstacle layer, in fact, there is no check to verify whether a point has already been marked in the costmap or not as part of the *static_layer*. This means that the *obstacle_layer* has no way to determine whether a point that has been marked belongs to the static environment or if it is a not a-priori known obstacle.

This type of behaviour is not a problem for the normal operations of the *obstacle_layer* because in the final costmap each point assumes the highest value among all the values that were associated to it by the single layers. It also is the optimal configuration in terms of computational resources. For the sharing application, on the other hand, the presence of points of the wall as part of the wall in the vector introduces a redundancy in the information shared as the robots receiving the information already have those points marked in their respective costmaps.

It was then necessary to introduce an additional check in order to verify whether the points that are about to be added to the vector already belong to the static map. In this case, even if the points are marked in the costmap, they are not added to the vector.

During normal operations, though, the robot is never perfectly localized, and this introduces errors that, even if they don't significantly compromise the standard functionalities of the obstacle layer, they can significantly affect the sharing, since it may happen that points that are part of the static layout of the environment are treated as obstacles and therefore added to the vector used for sharing. In order to avoid this a tolerance was added in the check that is performed with the static map.

This tolerance increases the robustness of the system towards errors in the localization of the AGV.

3.2.2 Keeping the vector dimension low

The number of elements in the vector increases as the robot keeps track of all the obstacles that have been detected during its operations. The informations relative to points that are too far from the current position of the AGV doesn't have to be shared since they might not be still up to date. Sharing data which are not up to date can cause the robot to share informations about obstacles that have already been removed from the costmap, thus compromising the navigation of the whole fleet. For this reason, and in order to keep the dimension of the vector as low as possible, two additional checks were introduced:

- The points in the vector removed in the case they are no longer in the *obstacle_range*, which is the range for adding new obstacles
- to avoid redundancy, each time a new point is added, a check is performed to verify that the point that is about to be added is not already present in the vector.

The *obstacle_range* is a parameter that tells to the obstacle_layer which is the maximum distance at which an obstacle should be added to the costmap. This is necessary because the errors in the localization of the obstacles relative to errors in the angular localization of the robot increase linearly with distance.

3.2.3 Clearing points from the vector

The functions for the marking and the clearing are performed in the same function and are called periodically by the move_base_node. In this function *updateBounds* the clearing functions are performed before the marking functions.

When the *updateBounds* method of the *obstacle_layer* is called by the *move_base_node* for the marking and clearing operations of the costmap relative to the global planner, The custom raytrace function described in 2 is called once for each point of the clearing observation (the clearing observation contains the same type of data of the laser scan type and those data are then converted into a point cloud), giving as input to the *raytraceFreespace* function, that offers an implementation of Bresenham's algorithm (explained in the details in Chapter 2), each point of the point cloud. The function is then able to correctly compute the lines knowing the position of the sensor at the moment of the observation.

The points that belong to the so drawn lines are then added to a temporary vector that contains all the cells which value needs to be set equal to the free space value.

This vector is then compared to the vector containing all the cells that have been identified as compliant with the criteria for obstacle sharing. All the cells that are present in both the vectors are removed from the latter. After this operation the temporary vector is cleared.

When an obstacle is detected and the sharing has been triggered, this same obstacle will be added to the costmap and sent to all the other AGVs. If the same or another robot perceives an obstacle in the same position and the sharing is triggered, there is no need to send the same information to the other AGVs because this would create an useless redundancy in the information sent.

The problem of keeping the informations sent to the minimum is of the utmost importance because each obstacle that is sent needs time to be correctly processed and added in the costmaps of each robot and then sent to the other AGVs in the system over a local area network, potentially subtracting both computational resources to each of the robots in the system and capacity to the network. This creates the problem of identifying in an unique way each one of the obstacles. This identification problem is handled differently for the internal identification of an obstacle and for the identification when the obstacle is shared among different robots. This has been done to respond to issues that arose in the specific configuration that has been implemented and that will be explained further in this Chapter.

3.3 Triggering the sharing

With the aim to keep the flow of information to the minimum possible and also not to share obstacles that don't significantly influence the performance of the fleet (i.e. all the obstacles that can be easily avoided and therefore don't block the path to any of the AGVs that are observing them), the sharing should be triggered only when the robot is not able to avoid an obstacle and fails to reach the goal. This information can be retrieved from a message that is sent by the move base node. This message is streamed over the topic *l_plan_feedback* and it returns a "0" when the local planner is able to compute the velocity commands to send to the wheels and it sends a "1" when the robot fails to compute the velocities due to the presence of an obstacle that cannot be avoided without exceeding the limits of the local costmap (which is a subsection of the global costmap that is used from the local planner).

Since this message is sent with a 10 Hz frequency, in order to successfully read the message containing the "1" it would be necessary to read all the messages coming from the *l_plan_feedback* topic, which means having to process 10 messages per second. In order to avoid this unnecessary waste of computational resources, a new publisher was created in the move base node in order to publish on a new topic only in case the local planner fails to compute the velocity commands for the wheels. This topic was named *stuck_feedback*.

3.3.1 What should be shared

When a node is subscribed to a topic there is the need to define a function responsible of the handling of the data which are received from the topic. If the local planner fails to compute the velocities for the wheels it will start sending redundant messages on the topic that is responsible of triggering the sharing causing the obstacle layer to share the same obstacle multiple times.

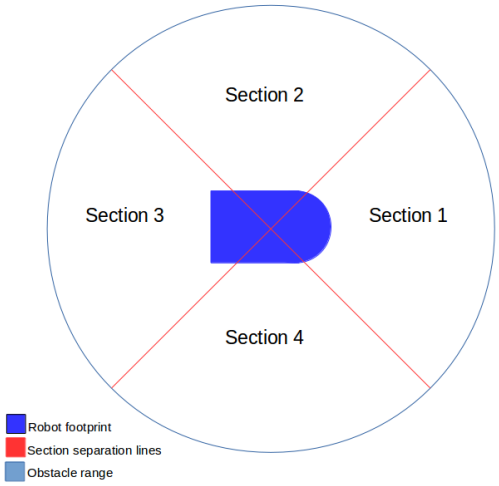


FIGURE 3.1: Section division for the sharing

In order to avoid this redundancy in the sharing only the first message received of a sequence will effectively trigger the sharing. This has been done by introducing a timer. The handler function will set a boolean variable to true and this will allow the execution of the part of the *updateBounds* method responsible of handling the sharing. This is done calling a function which is called *scanPoints*. This function splits the area surrounding the AGV in four different sections as shown in figure 3.1 and then creates four different temporary vectors, each containing the points associated to a specific section of the whole area. These sections are delimited by two lines, defined as follows:

$$y = (x - robot\_x) + robot_y \tag{3.1}$$

$$y = -(x - robot\_x) + robot_y \tag{3.2}$$

Each one of the four vectors is then associated to a rectangle that includes all the point of the vector in its area. Even if this solution is not precise to define the exact shape of the obstacle, it can achieve the goal of communicating the other AGVs that a path is blocked with the minimum amount of sent data possible. The such defined rectangles are then shared by publishing them on the associated topic.

The use of sectors was made necessary by the fact that the points that are shared are associated to rectangles. Let's consider a situation in which the robot has detected some points belonging to an obstacle which is now behind it. At this point the robot finds an obstacle in front of it which is blocking its path. Without using the sectors the two obstacles would be associated to the same area and shared as an unique entity. If the robot is standing in between the two obstacles (the previously detected one and the one which is blocking the pathway), the rectangle associated to them would include the robot, which is standing in between, resulting in blocking the robot as it would be, according to its costmap, stuck inside an obstacle.

3.4 The sharing handler node

The overall flow of data in the system, seen from the perspective of the single robot, includes data which are sent from the move base about obstacles that need to be

shared and data incoming from the other AGVs for adding the obstacles. Considering that there is also the need to remove the obstacles if they are not in the map anymore, the amount of data previously considered duplicates.

There is the need of creating a node responsible of handling all the data and making sure that:

- The obstacles that are set in the costmap of a robot are consistent with the data that are stored in the others.
- Each instance of an obstacle is unique and can uniquely be identified when it is added to the costmap regardless of the robot which has observed it (it can be the same robot that adds is to its costmap or it can come from an external source).
- Correctly handling the removal of each obstacle regardless of the robot which has sent the clearing informations.

The creation of an additional node can also answer to an optimization problem relative to the frequency of the move base node. The frequency of the move base node for the costmap update is set to 5 Hz. This frequency has to be set to a value sufficiently high in order to give the robot an up to date representation of its surrounding environment. Since the costmap is used for navigation purposes, it is of the utmost importance that the obstacles identified by the scanner of a robot moving along a path are added to the costmap as soon as possible.

On the other hand the sharing functionality has not the same necessity. This is because when an obstacle is shared it has already been detected and set in the local costmap of the robot making the observation so it no longer represents a threat for the correct navigation behaviour of the AGV. The sharing functionality is, in fact, supposed just to affect the capacity of the fleet by giving information about paths that are blocked. If an obstacle appears on the costmap of an AGV and the sharing is not immediately performed and the same obstacle appears also on the path of another AGV this in no way represents a threat for the navigation behaviour of the latter AGV, in the sense that the obstacle detection and avoidance behaviour is independent from the sharing functionality.

The newly created node, which was implemented using Python, has the advantage of being independent from the behaviour of the *move_base_node*. This means that the frequency can be set to a lower value to handle the sharing functionality with respect to the move base frequency, leading to a configuration which is less resource consuming from a computational point of view. The frequency of this node can be rapidly reconfigured from the associated configuration file.

3.4.1 The list

During the testing phase it is very useful to have a visual representation of all the obstacles that are currently set in the individual AGVs. The *yaml* format was chosen for the implementation of the table because it is more human-readable with respect to the *xml* format.

The obstacles are registered in the *yaml* table as a couple of (x, y) coordinates indicating the two opposite vertices positions, a time span that indicates the lifetime of the obstacle, and an ID that is used internally in the robot to uniquely identify each obstacle. The table plays an important role in the debugging phase, as it makes the checking of which obstacles have been shared and which obstacles have been removed as immediate as opening a text file. When the system is correctly functioning, all the different tables, each one associated to a different AGV, contain the same obstacles that are stored inside the other ones.

It is also possible to manually add a new obstacle directly in the table and check whether this is properly shared (meaning that it also appears in the other tables) and marked inside the costmap of each one of the robots.

3.4.2 The interaction with move base

The Sharing handler node, developed in Python (all the part relative to the costmap and to *move_base* is developed in C++), has the task of handling the communication between the different elements of the system responsible for the sharing of the obstacles. In particular, with respect to the *move_base_node*, it has the task of keeping track of all the obstacles that have been detected both by the same AGV the handler node is running into and the other AGVs. Therefore the main role of this node is to grant or deny the authorization to add a new obstacle to the costmap or to remove an existing one.

Handling data coming from move base

The position of the newly detected obstacles is streamed by the *move_base_node* to the other ROS nodes via publishing on the *obstaclePosition* topic. The "Sharing handler" node retrieves the informations associated to that topic via subscription.

The time interval between the publishing and the retrieval of the information depends on the frequency at which the python node is running. The frequency was set to 0.2 Hz but this value is easily reconfigurable.

The function that is called for the handling of the data incoming from the *obstaclePosition* topic is named *obstacleFromMoveBaseCallback*. The flow chart relative to this function is shown in Figure 3.4 The message this function receives as input is an array of single precision float numbers. The first four elements of the array represent the coordinates of the two points that are used to generate the rectangle, whereas the last two elements of the array represent the coordinates of an additional point that is used to simplify the clearing operation via raytracing. Since the coordinates refer to the map frame, only positive coordinates are defined. When the received additional point has negative coordinates it means that the points associated to the rectangle are just the first two.

The function first performs a check to avoid adding multiple instances of the obstacles into the list. If the obstacles have not been already added to the list, two checks are performed in order to verify how many different point have been received as raytracing informations. The data necessary to clear the obstacle is then

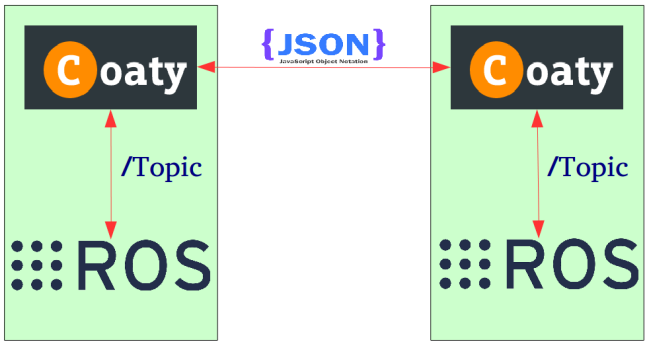


FIGURE 3.2: Coaty communication structure

sent to the *move_base_node* receive. Finally the function responsible of the outbound communication with the Coaty framework is called. The details about Coaty and the working principle of this function will be explained in the Coaty section of this chapter.

The received obstacle can be built with one, two or more points. In the first two cases, the only points chosen for the raytracing are all the points that have been used to generate the obstacle. In the general case of having an obstacle generated with more than two points, the points associated to the obstacle for its raytracing are only three and are the two vertices that generated the rectangle and an additional point chosen as the closest point to the middle position of the rectangle. The message also contains an ID in order to simplify the operations of raytracing.

Sending data about new obstacles to move base

The list containing all the obstacles is periodically scanned. When a new obstacle is found in the list it is then added to a vector that contains a list of all the obstacles that have already been scanned and which duration is not expired. The data relative to this obstacle are then streamed on the *SharedObstacleMessages* topic. The messages streamed on this topic are structured as two points represented in planar cartesian coordinates, and an associated ID.

3.4.3 How is the sharing performed in Coaty

In the configuration that is implemented in both the simulation environment and the proANT AGVs, Coaty is responsible of handling the communications between the different agents in the system. The communication used in this application is an asynchronous type publisher/subscriber.

Since all the informations are gathered inside the Python node that handles the decisional part of the sharing functionalities that have been implemented, the role of the Coaty framework is translating the information incoming from the AGV the node is running in and sending them to the other Coaty agents in a suitable format. When the communication is coming from other agents, instead, the Coaty node has to translate the received informations in a suitable way and send them to the other ROS nodes. An overview of the structure is shown in figure 3.2

The sharing between different Coaty agents

The sharing between two different Coaty agents is achieved with a topic mechanism similar to the one used internally with ROS, the main difference being that the data have to be sent in the JSON format, which stands for JavaScript Notation Format.

The object which is associated to an obstacle inside the Coaty framework and is used for sharing purposes among different agents has been defined with the following attributes:

- The two points defining the position
- An additional point used for raytracing
- The duration that was assigned to the obstacle
- An additional variable that determines whether that obstacle should be added to the costmap of the AGVs receiving it or if it has to be cleared.

Whilst the identification mechanism that was implemented makes use of an ID for the internal identification of the obstacle, for an unambiguous identification of the obstacles between the different AGVs the position of the two opposite vertices is used (since only one obstacle at a time can occupy a specific position).

the additional couple of coordinates is used to simplify the raytracing operations in almost the same way in which it is used internally in the robot for when an obstacle is identified in the *obstacle_layer*. The additional point can assume two different negative couples of values $(-1, -1)$, and $(-2, -2)$, those two couple of values indicate respectively that the obstacle was generated by using only two points (therefore there is no additional point to be used for raytracing), and that the obstacle was artificially added into the map of a robot by manually modifying the *yaml* list (therefore it is used only for testing purpose and should not be cleared via raytracing).

Since the set up and the clearing of an obstacle is handled with the same message format and with the same handler function, an additional variable that tells if the obstacle position is new or it is already there and needs to be cleared has been used.

The interaction between the sharing handler node and the Coaty framework

When an obstacle is received from the *move_base_node* and it is identified as compliant with all the necessary checks in order to be added to the costmap (see Figure 3.4), the *sendToCoaty* method is called (Figure 3.5). The purpose of this function is to share the data relative to the obstacle by publishing on a topic named *ObstacleToCoaty*. The message sent has the same structure for both adding an obstacle and for clearing it. The value of the *kill* variable that carries the information relative to whether the obstacle should be marked or cleared from the other agents.

The messages that are sent by the Coaty node to the Python node have the same structure of the ones that are sent by ROS to Coaty. Those messages are handled by the *obstacleFromCoatyCallback* function. This handler performs almost the same functions that the *obstacleFromMoveBaseCallback* performs when an obstacle is received from the move_base (details in Figure 3.4). The differences are:

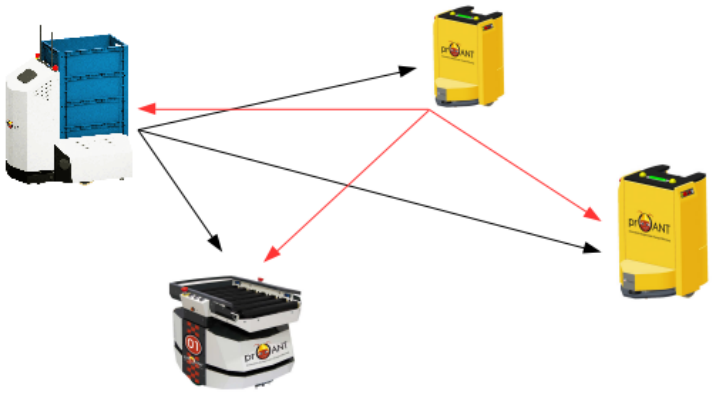


FIGURE 3.3: Message redundancy example

- In this case there is no need to assign a duration to the obstacle since it has been already assigned from the AGV that sent it.
- The obstacle doesn't have to be sent back to the other AGVs.

How to avoid sending back to the other AGVs the received obstacles

In the sharing handler node there are 3 ways to add a new obstacle to the list of the existing ones:

- The obstacle is detected from the robot that the node is running into.
- The obstacle is received via Coaty from another agent in the system.
- The obstacle is generated by directly modifying the list.

These three cases have to be handled differently since in the first and in the third case the detected obstacle has been created internally to the robot and therefore it has to be sent to the other AGVs. In the second case the obstacle was received from another AGV and must not be sent back. Figure 3.3 shows an example of what can happen if this is not correctly handled. The AGV on the left identified an obstacle and sent its information to the other AGVs in the system (indicated with the three black arrows). Each one of those added this new obstacle to its list and then sent it back to the others (as indicated by the red arrows). If the number of AGVs in the system is n, the number of messages sent would then be $n(n-1)$. When the information about the obstacle is received for the second time, the obstacle won't be added again to the map since it is already there and therefore it won't be sent back to the other AGVs in the system. This means that even in case of a mishandling of the information, the number of messages sent would be still limited. Even if the worst case scenario just described doesn't lead to a crash of the system, this situation is still not acceptable and a way to avoid sending back the received obstacles back to the other robots must be found.

The problem has its foundations in the fact that the both the functions responsible of handling the obstacles received from the *move_base_node* and from Coaty, write in the list to add the received obstacles. When the function that has the task of scanning the list is called (it is called periodically, whilst the handlers are event based) it will perceive these new obstacles as new, then it will send their information to the

move_base_node in order to allow their marking in the costmap and finally will send the informations to the other AGVs in the system via Coaty. This sequence of operation is correct when the obstacle was detected internally, but it has to be avoided when the obstacle is coming from the other agents.

This issue was solved by means of an array (properly, a list) that contains the identification numbers of the obstacles that don't need to be sent to the other AGVs. The sendToCoaty function described in the previous chapter will not send the obstacles which ID is already present in the array and, as last operation, it will append the sent id at the end of the list. On the other hand, when the obstacle is received from an external source, the handler *obstacleFromCoatyCallback* after generating the ID of the obstacle, it will append this id at the end of the before mentioned array. In this way, the *sendToCoaty* function will not send the obstacle back to the other AGVs.

3.5 How is a shared obstacle cleared

When an obstacle is shared it means that it is blocking a path. This is done in order to notify the other AGVs that that path is blocked. When a robot is passing nearby the obstacle at a distance which is suitable for the raytracing, if the obstacle is no longer present in the environment, the other robots should be notified about this and it should be removed from the costmaps of all the AGVs in the system. The problem that arises relatively to an obstacle blocking a path is that when a path is blocked the AGVs won't travel across it and therefore, if the obstacle is not visible from the robots that pass nearby, it will never be cleared. For this reason it is necessary to have a predefined duration for each added obstacle. When that duration expires the obstacles are removed. The duration that is assigned to the detected obstacles can be modified from a configuration file.

The obstacles are handled internally by means of an ID and when the clearing request comes from another AGV the obstacle is identified by its position in the environment.

When an obstacle is removed from the costmap it is of the utmost importance to remove its information from all the lists (the one in *yaml* format and the ones that are only active at runtime) in which the obstacle has been added. If it is not correctly done there might still be an ID associated with some information that could be reassigned to a new obstacle, causing a malfunction of the system.

3.5.1 Duration expiration

Considering the sharing functionality that was implemented as part of this thesis' work, since the obstacles can in principle be spawned in areas which are far from the current position of the AGV in which the functions are running, this may cause to update a very large area of the costmap only to add a small area corresponding to a newly spawned obstacle.

In order to avoid this, the obstacles that are added by the sharing functionality are marked directly in the master grid. To eliminate these obstacles it is sufficient to stop adding the obstacles to the master grid and as a result they won't appear in the

visualization.

The duration expiration that is assigned to the obstacles is handled by the Sharing Handler node. When a new obstacle is spawned, this script keeps track of its arrival time. When the duration of the obstacle has expired, the command to clear it from the costmap is sent via message from the python node and is read by *move_base_node*. The message contains the internal ID of the obstacle that needs to be removed from the costmap.

The duration expiration of the obstacles is handled independently in each robot. This means that when an obstacle is removed because of the deadline expiration, no message is sent. When an obstacle is received the robots start counting the arrival time from the moment that the obstacle is added to their list. This may cause a slightly asynchronous behaviour of the obstacles that is related to the frequency at which the Sharing handler nodes are running and the time to receive the obstacle. This behaviour, however, doesn't constitute a threat to the normal functioning of the system, since an asynchronous behaviour in the order of magnitude of seconds is neglectable with respect to the duration of the obstacles that is at least superior by 2 orders of magnitude.

3.5.2 External clearing

The clearing and adding of obstacles in Coaty has intrinsic redundancy problems. As was explained in the Coaty section of this chapter regarding the marking operation, the problem relative to sending back to the other AGVs the obstacles received from Coaty was solved using a vector containing the IDs that don't have to be sent to the other AGVs. A similar solution was used for the raytracing, the difference is that when an obstacle is removed the ID has to stay in the list even if the obstacle is not there anymore to avoid the case in which the obstacle removal information is received from Coaty, the obstacle is removed, and then the information is sent back to the other agents.

The different cases in the handling of the *alreadySentToCoaty* and of the *alreadySentKillToCoaty* vectors is shown in the table 3.1. Whilst the way in which new IDs are appended at the end of the two lists is similar, the way in which the removal of IDs from the lists is handled is different:

- The *alreadySentToCoaty* list, that handles the informations about newly spawned obstacles, is cleared upon the removal of an obstacle, regardless of the source of the deletion request (it can be either move_base or Coaty).
- the *alreadySentKillToCoaty* list, that handles the informations about the clearing of an obstacle, is cleared when an ID is reassigned.

The difference is in the fact that when an obstacle is removed from the list, an information is sent to all the other AGVs in order to remove the obstacle from all the costmaps. When the obstacle is deleted, the information about its removal should be kept in order to avoid sending it back to the other AGVs multiple times.

The messages regarding the spawning or the deletion of obstacles are sent from Coaty on the same topic and therefore are handled by the same function *obstacleFromCoatyCallback* (the relative flow chart is shown in figure 3.6). In order to correctly

List	Append to list	Remove from list
alreadySentToCoaty	new obstacles from Coaty obstacle sent to Coaty	ID removed by Coaty ID removed by move_base
alreadySentKillToCoaty	send remove to Coaty receive remove by Coaty	reassigning an ID

TABLE 3.1: The handling of the lists relating to the sharing with Coaty

handle the messages, there is an associated variable that tells whether to remove or to add the obstacle. In case the obstacle needs to be removed, from the received position the associated internal ID is found, it is added to the list *alreadySentKillToCoaty* in order to avoid sending the removal information back to the other AGVs, and finally the informations for its deletion are sent to *move_base_node* and the obstacle is removed from the *yaml* list.

3.5.3 Internal clearing

When the request for the deletion of the obstacle is handled internally from the same AGV, the request is sent by move_base_node via topic. The sent message contains just an ID which is sufficient to identify the obstacle that needs to be removed. The information relative to this deletion is then sent to the Coaty framework (sendToCoaty, figure 3.5) in order to be shared with the other agents in the fleet.

3.6 Technical problems of the configuration

The reasons for which this first configuration was discarded in favor of a configuration based on the sharing of the points is related to the precision of the localization of the robot. Even a slight de-localization of the AGV can significantly affect the cell of the visualization to which a point in the real environment is associated.

When a robot identifies the presence of an obstacle, this obstacle will be associated to a certain set of cells in the costmap. As the robot gets closer to the obstacle, the cells to which the obstacle is associated can be different according to the tolerance on the positioning precision of the robot. The same point in space can be associated to a cell or to another according to where the robot perceives itself in the localization.

Let's consider a possible scenario in which a robot first identifies an obstacle to be in a specific position. During the robot movement it will get closer to the obstacle and will be able to better identify its position (the precision in the detection is heavily affected by the angular positioning precision of the robot as the related error linearly increases proportionally to the distance from the detected obstacle) and will therefore associate the position of the detected obstacle to different cells in the visualization. In the moment in which the cell that was previously associated to the obstacle is labeled as "free space" by the raytracing algorithm the obstacle is removed. This error can lead to the removal of obstacles that are still present in the environment.

Another problem relative to this configuration is the fact that since the points that are detected are then used to form a shape, the presence of detection errors can lead to the generation of virtual obstacles in the representation that are not corresponding to the presence of any obstacle in the real environment.

A solution to the first of these issues could be the use of a polygon which shape is constantly modified according to the readings of the sensors. This solution, though, would undermine the very purpose of using a configuration which is based on the use of polygons as its purpose is the minimization of the shared data.

These considerations led to the adoption of a system which is based on sharing the indexes of the costmap associated to the cells that have been identified as obstacles, without using the information regarding the cells to generate polygons. This system will be described in the next chapter.

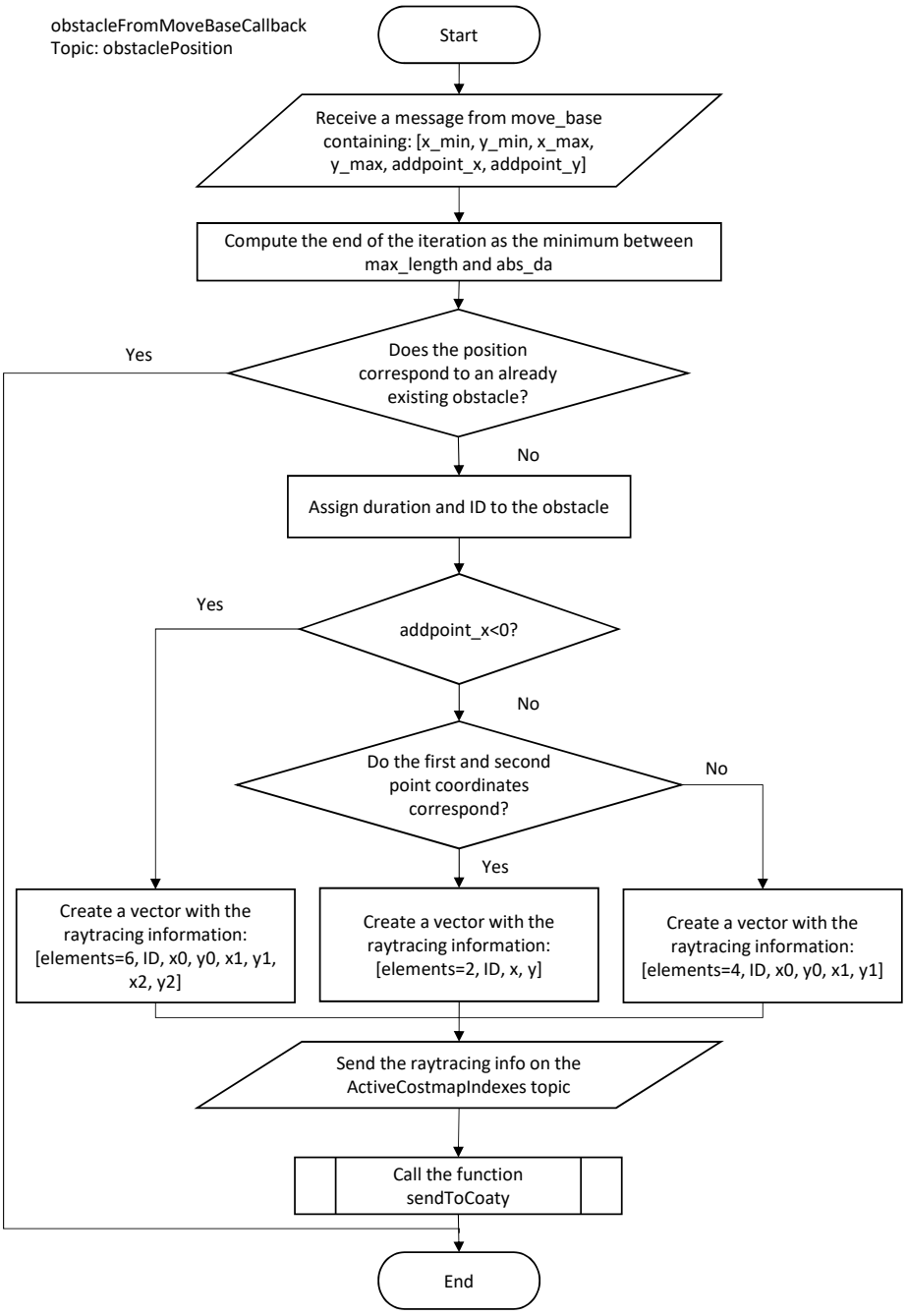


FIGURE 3.4: obstacleFromMoveBaseCallback flow chart

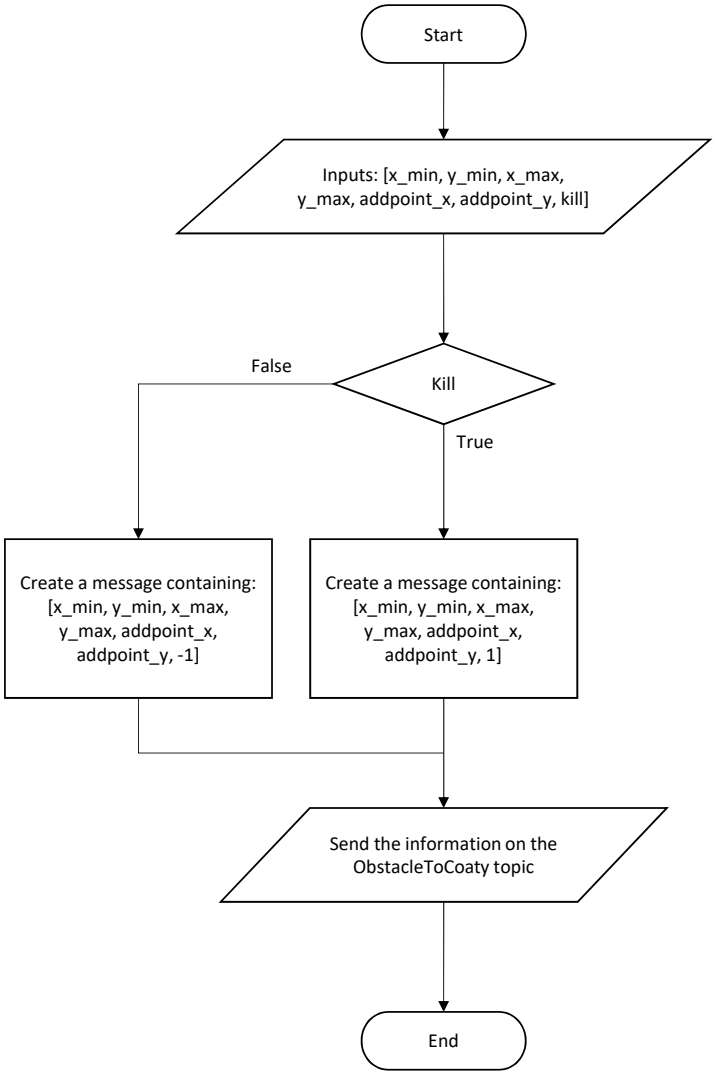


FIGURE 3.5: sendToCoaty flow chart

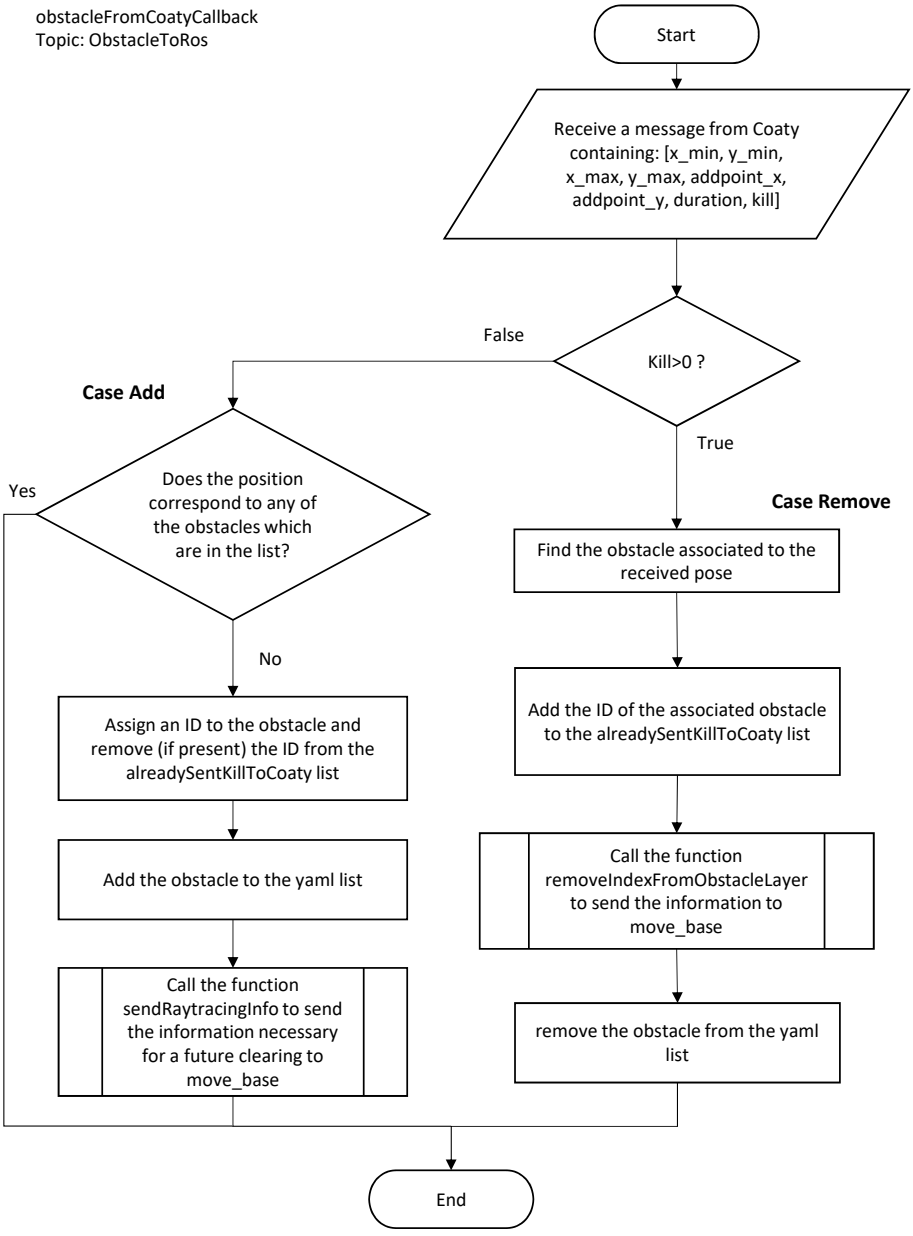


FIGURE 3.6: obstacleFromCoatyCallback flow chart

Chapter 4

Optimization and Configuration Changes

4.1 Introduction

In the first configuration, the issues that arose were mainly related to the lack of precision of the robot in its localization. Since the robot is never perfectly localized, the coordinates of the points that are added to the AGV visualization can differ from the point that should have been detected if the robot was perfectly localized. When those points are shared, according to the first configuration that was implemented, a rectangle was associated with them.

The problem of having errors in the identification of the points is then strictly related to the way in which the obstacles are marked. In fact the points are connected by lines, resulting in the creation of an obstacle with a different shape from the real one. This can heavily compromise the functionality of the system by blocking pathways with virtual obstacles that aren't matched by real obstacles in the working environment. Therefore it is clear that the so shaped system manifests a lack in robustness with respect to the presence of errors among the identified points.

A possible solution to this problem would be to modify the shape of the obstacle on the go, adjusting its shape according to the latest readings of the sensors. This could be done by modifying the system in order to use convex hull polygons instead of rectangles in the representation of obstacles and modifying the associated points when they are inside of the minimum range which has been defined as suitable to perform the marking of the obstacles.

However, with this configuration, the flow of data among the AGVs would drastically increase, since, instead of sharing only one obstacle at a time when the obstacle is detected, the modifications to the obstacle would then be shared with a much higher frequency, in the same order of magnitude of the frequency at which the costmaps are running. As this could not be a problem in terms of CPU usage, it would undermine the very point of using polygons to define the shape of the obstacles instead of directly sharing the points that are forming the obstacle.

Since the errors in the localization precision solution is outside of the aim of the present work (and could be also not feasible with relation to the hardware at our disposal), another solution has to be found. Since the presence of errors in the detected

```
string UUID
uint32[] marked_indexes
uint32[] cleared_indexes
uint32 deadline
float32[] rectangle_shape
```

FIGURE 4.1: Structure of the custom defined message

(and therefore shared) points cannot be consistently avoided, in order to guarantee a functionality of the system which is robust against localization errors, instead of associating the points to polygons and then share them, the option of directly sharing the points associated to the obstacle was chosen.

The solution just described does not completely solve the problem in the sense that it reduces the effect of a wrongly detected point by not considering it a vertex of an artificially created obstacle, but, in order to properly deal with uncertainties in the detection of points of the laser scanner it is necessary to remove the wrongly detected points. This is done by updating the obstacle after it has been already added. In this way the newest readings of the laser scanner will remove the points that were present in a previous observation but that are not detected anymore. This solution will also provide a realistic representation of the real shape of the obstacle even if this is only slightly moved from the position in which it was first detected.

The potential issues related to this new configuration are strictly related with the computational resources that are necessary to handle an increased data flow (therefore also the network could possibly become a bottleneck for the system) and therefore an increased amount of data to process. The solutions which were adopted in order to handle these possible issues in the configuration and the way in which the configuration was handled will be explained in the following.

The first part of this Chapter will show in the details the structure that was defined in order to share the information regarding the obstacles and the reasons that led to the definition of the structure of the messages that was defined. Then the modifications that were performed on each node of the configuration will be described.

4.2 The structure of the shared data

In order to keep the configuration as simple and standardized throughout the system as possible, a new message structure was defined: this structure has been used in all of the messages that have the purpose of adding an existing obstacle or modifying its shape in some way which are sent internally in the robot. The structure of the information used for data transfer between Coaty agents is also shadowing this structure.

The message structure has been defined as a custom message named *SharedObstacle* inside the *costmap_2d* package. The structure of the message is shown in figure 4.1. The role of each element in this structure will be explained in the details in the following.

4.2.1 The UUID

When the information about multiple obstacles is shared, there are several operations that need to be performed on the elements that were shared such as updating the shape of the obstacles either internally (which means from the obstacle_layer of the robot that has the information) or externally (which means that the update is incoming from another agent in the fleet of AGVs), or removing the obstacle after its expiration date. In order to simplify the operation of identification of an obstacle we decided to use an identification system.

With respect to the initial configuration, the use of a globally unique identifier that can be used both internally in the robot and among different agents in the fleet was chosen. This choice brings to an increment of the size in bytes of each single message which is shared (in the initial configuration the ID was defined only internally in the robot, therefore it was a part of the message shared over the network), but it significantly simplifies the operation of identifying an obstacle which information have been shared among the list of all the shared obstacles.

The acronym *UUID* stands for Universally Unique Identifier (reference) and it is used to identify the obstacles in an unambiguous way across the different AGVs in the system. It is treated as a string for the comparison (this means that in order to find a specific identifier in a list a comparison with the elements in the list is performed, and in this case the identifiers are treated as strings) but it is generated as a 32 hexadecimal digits, displayed in 5 groups separated by hyphens, in the form 8-4-4-4-12 for a total of 36 characters. A detailed explanation of the role of the bits is provided in table 4.1. In particular the 48-bit node id what the table refers to is represented by the media access control (MAC) address of the device, which is a unique identifier assigned to a network interface controller (NIC) [13].

Name	Length (bytes)	Length (hex digits)	Contents
time_low	4	8	integer giving the low 32 bits of the time
time_mid	2	4	integer giving the middle 16 bits of the time
time_hi_and_version	2	4	4-bit "version" in the most significant bits, followed by the 12 bits of the time
clock_seq_hi_and_res clock_seq_low	2	4	1 to 3-bit variant in the most significant bits, followed by the 13 to 15-bit clock sequence
node	6	12	the 48-bit node id

TABLE 4.1: UUID record layout [13]

4.2.2 Marked and cleared cells

The *marked_indexes* is an array of unsigned integers that should contain the information about all the indexes that are in the obstacle that have to be marked. When new

points of the obstacle are detected they should be added to that list.

the *cleared_indexes* is an array of unsigned integers that should contain all the informations related to the obstacles that have to be cleared. This is used only in the updating phase of the obstacles. When a point was detected in a position and the raytracing algorithm has detected that that point no more corresponds to an obstacle, it is added to this array. The information which is contained in this array is used for sharing the data about the clearing of indexes with the other robots, whilst the updating of the points that belong to an obstacle with the clearing of some points are directly handled by the custom functions of the *obstacle_layer*.

The necessity to use two lists to handle the indexes of the costmap associated to an obstacle arises from the need to avoid the loss of data when an obstacle is observed from an angle which is different from the angle from which it was initially observed.

Let's consider, as an example, the situation in which an obstacle is observed a first time as in figure 4.2 (A), and the relative information is shared. The information that is detected by the robot is represented by all the points belonging to the two sides of the obstacle which have been highlighted in red. The shared obstacle structure will then contain the indexes relating to the two highlighted sides of the rectangle.

Upon a new observation of the obstacle, the angle from which the observation is taken can in principle be different from the one from which the obstacle was first observed. Let's consider, then, an observation taken from the angle represented in figure 4.2 (B). In this case the only side of the obstacle which is observed is only one.

At this point there are two possible ways to handle the observation with only one list, and those are:

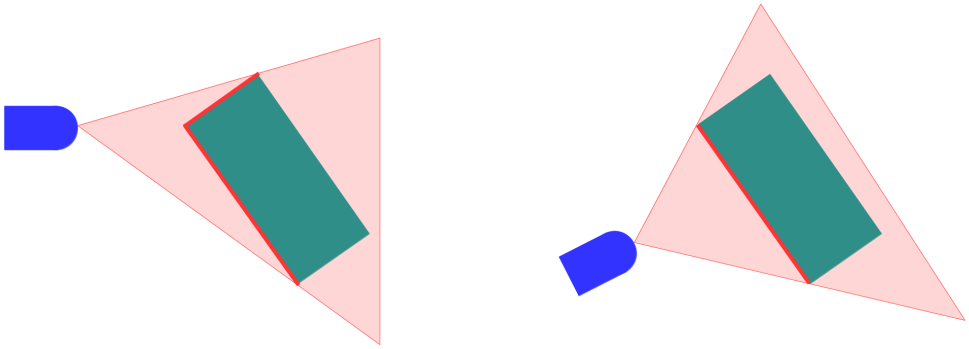
- Overwriting the data on the list.
- Appending the newly observed cells at the end of the list.

Neither of those two different approaches works.

In the case in which we are overwriting the data on the list, a situation in which the obstacle is observed from two different angles would cause a loss of information, as, for example, in the situation shown in figure 4.2 in which the data relative to the short side of the obstacle would be lost. This missing information regarding the obstacle that was detected could then possibly provide the other agents in the system with an information which is telling them that a pathway would be free when it is actually blocked by the presence of an already detected obstacle.

On the other hand, if the points that are detected are just added to the points of the list this would nullify the very purpose of updating the area associated to the obstacles since there would be no way of clearing the points of the area and therefore no way of passing along the information that an obstacle was removed to the other agents in the fleet.

To avoid these possible situation the configuration takes into account the presence of two lists, one for the indexes which have to be marked in the costmap section belonging to the obstacle, and one for the indexes which were previously in the list of marked indexes that have been explicitly cleared by the raytracing algorithm (this means that the point where the index was previously has been observed and it has identified as "free space"). The detailed working principle of the two lists will be explained in a separate section with an example.



(A) first observation

(B) second observation

FIGURE 4.2: Observation and update example

4.2.3 The expiration date

If an obstacle is blocking a corridor in a facility and it is identified and the information regarding its presence is shared among the agents, this corridor will likely not be crossed again by any robot as long as the obstacle is present. If this obstacle is in an area which is never observed except if a robot passes in that specific corridor, the obstacle will likely occupy that position in the visualization for a virtually unlimited amount of time. In order to avoid this problem, an expiration date, to assign to each shared obstacle, was defined. This expiration date is set according to a parameter which is configurable for the specific use case scenario.

The expiration date, named deadline in the message structure, represents the duration (expressed in seconds) that is assigned to the obstacle by the robot who first detects it. From the moment an obstacle is added to the costmap by an AGV, the arrival time of the obstacle is stored. When the time that the obstacle has spent in the costmap exceeds its assigned duration it is removed. This behaviour was not changed from the first configuration explained in 4 in the part related to the clearing of the obstacles by deadline expiration.

4.2.4 Costmap Subsection

Upon sharing the information relative to a newly detected obstacle, a subsection of the costmap is defined. This subsection, represented by a rectangle (and therefore defined as rectangle_shape in the message structure) represents the area in which the marked indexes of the obstacle are contained upon its first observation. It is

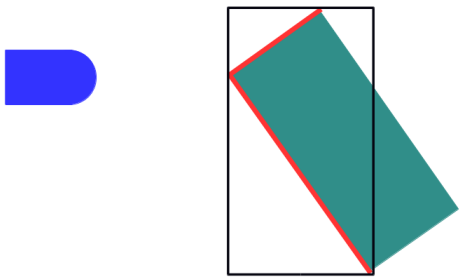


FIGURE 4.3: Costmap subsection example

computed in the same way as the obstacle shape was in the previous configuration, explained in 4, but in this new configuration it covers a different role in the sharing process of the obstacle information. In fact, in this new configuration, the obstacle is not represented by a rectangle which is entirely marked but the rectangle is only used to identify the position of the obstacle. Only the points that are contained in this rectangle are marked or cleared in the obstacle and their updates are shared in real time.

Therefore, this area represents the zone of the costmap which is monitored whenever a robot is close to it. All the information relative to the presence of points to mark or to clear that differs from the data which are already known to the agent will then trigger an update of the obstacle associated to the area.

For example, considering the figure 4.3, upon observing the two sides of an obstacles which are highlighted in red, the rectangle associated to the obstacle will be the one shown in black. All the cells of the costmap in this rectangle will be subject to updates whenever the rectangle enters the zone which is associated to the marking of obstacles on the costmap.

4.2.5 Identification and update example

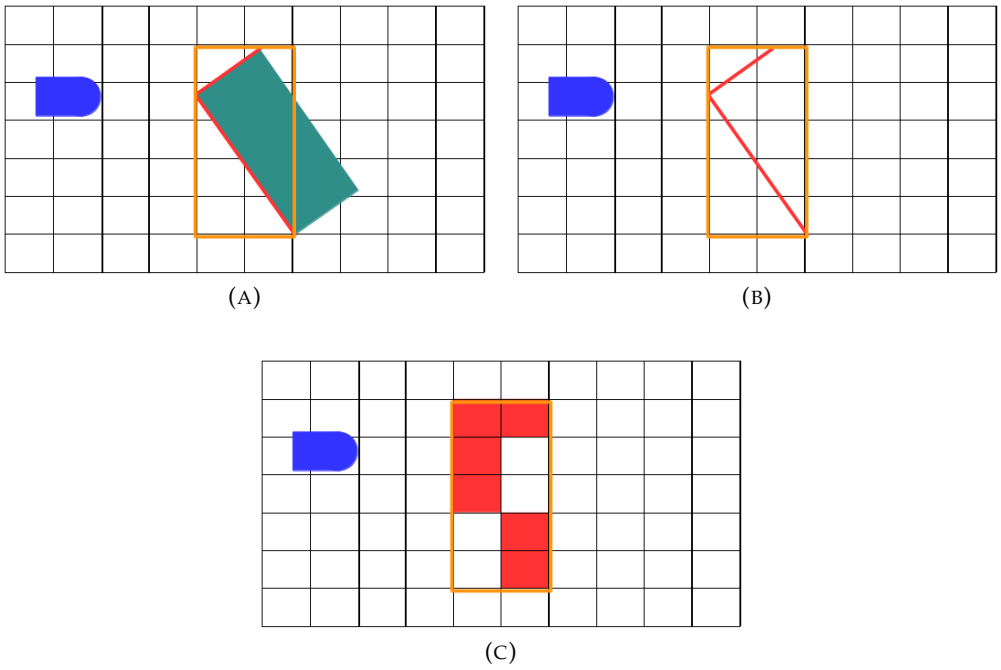


FIGURE 4.4: Obstacle sharing example

Figure 4.4 shows an example of how the data relative to an obstacle which has to be shared is stored. In the frame (A) of the figure the robot, drawn in blue, has identified an obstacle. The detected sides of the obstacle are highlighted in red, and they are all contained in the orange rectangle which represents the costmap subsection which has to be associated to the obstacle. In the frame (B) the sides of the obstacle that contain the information that has to be marked in the costmap are isolated for clarity. The cells that are associated to the sides of the obstacle are shown in frame (C).

As a result the obstacle is represented, referring to figure 4.4 (C), as:

- UUID: generated and assigned by the *SharingHandler* node.
- marked_indexes: all the indexes associated to the red cells in the figure.
- *cleared_indexes*: empty vector.
- *rectangle_shape*: the orange rectangle shown in figure.
- *expiration_date*: assigned by the *SharingHandler* node according to the configured parameter.

It is important to notice that the *cleared_indexes* vector associated to the obstacle is empty, as the clearing of the indexes is reserved to the updates. Only the cells that were previously marked can be cleared.

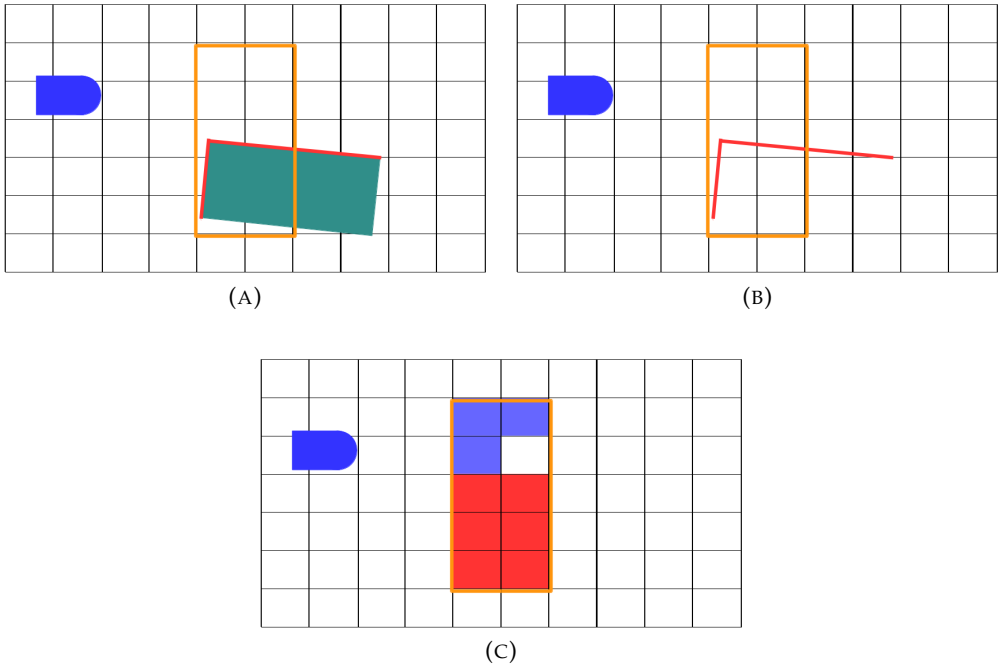


FIGURE 4.5: Obstacle updating example

Let's consider now the case in which this same obstacle has to be updated at a later moment. The obstacle has been moved from the original position as can be seen in figure 4.5 (A). The sides of the obstacle that are visible to the robot are the highlighted in red (Figure 4.5 (A) and (B)). It is possible to immediately see that the area of the costmap which is being monitored is still the same as before. This area, in fact, is not modified after it has been set upon first identification of the obstacle.

Considering figure 4.5 (C) The *cleared_indexes* vector is made of the cells which are highlighted in blue, That are all the cells that were previously part of the *marked_indexes* list and that have been cleared by the raytracing algorithm. Note in fact that only the cells that are cleared through the raytracing algorithm will be set as free space". The cells that are not visible at the moment (such as the two cells marked in red in the lower right) will remain untouched as there is no new information regarding them. The *marked_indexes* list contains the cells marked in red. As a result the obstacle is represented as:

- UUID: not modified with respect to the previously generated one.
- *marked_indexes*: all the indexes associated to the red cells in the figure.
- *cleared_indexes*: all the indexes associated to the blue cells in the figure.
- *rectangle_shape*: same as before.
- *expiration_date*: The time at which the obstacle will be removed from the visualization has not been modified.

It is important to notice that even if the obstacle is still present in the map, if the area occupied by the obstacle does not block the pathway anymore, all the agents in the fleet will be able to compute a path that crosses the area which is no longer blocked.

4.3 The identification in the obstacle layer

It is possible to divide the functionalities of the *obstacle_layer* relative to the sharing in three subcategories:

- The functionalities related to the detection of the points.
- The functionalities related to the first sharing.
- The functionalities relative to the updates.

The part related to the detection and clearing of the cells is strongly linked to the standard functionality of the *obstacle_layer*. The marking is performed on the end points of the observation data retrieved via the laser scanner and the clearing is performed via an implementation of Bresenham's algorithm for computing a line on a raster receiving two points as input (which are the source of the observation and the end point of the laser beam). This part was explained in chapter 2. The part that was implemented regarding the way in which obstacles are detected is the same that was implemented in the first configuration and that was explained in chapter 3.

The part relative to the first sharing has gone through some modifications with respect to the configuration that was explained in 3, the main difference being that the points that are identified in the obstacle layer, when the sharing functionality is triggered, are not associated to rectangles that have to be marked entirely but are shared according to the structure that was explained previously in this chapter. The presence of the rectangle, though, still remains in order to properly identify the area associated to the obstacle. This costmap subsection that is present in the new structure is determined in the same way as the area of the rectangle was determined in the first configuration, explained in chapter 3.

When an obstacle is shared it is sent to the *sharingHandler* node which has to assign an ID to the obstacle and determine if the obstacle needs to be marked. When the obstacle is sent back to *move_base*, it is marked inside the costmap and its shape is updated when the obstacle is inside of the obstacle marking range.

In this behaviour, related to the updating of the obstacle which information have been already shared, there are two main differences with respect to the previous configuration which was described in chapter 3.

The first is that the obstacle cannot be deleted via raytracing, meaning that even if the obstacle has no points to be marked on the map, it will not be definitively removed until its expiration date, which is assigned by the *sharingHandler*.

The main reason that led to this choice is the poor robustness of the previous configuration towards errors in the identification of points in the sensor readings. These errors are caused by the presence of a tolerance on the precision in the localization of the robot. The error on the angular positioning of the robot causes an error in the identification of the points in the surrounding environment. Therefore a point in the surroundings of the robot that has to be associated to an obstacle can be erroneously associated to a cell in the costmap which does not belong to the obstacle. This error increases linearly with the distance between the point that is being considered and the sensor of the robot.

The problem arises when the robot observes the same obstacle from a different position. In fact, in this case, the cell to which the obstacle is associated could change and the point which was first associated to that obstacle would now be detected as "free space", this would cause the removal of the obstacle, even if the obstacle has not moved at all from the position in which it was first detected.

The second main difference is in the fact that, by sharing the points, the shape of the obstacle can be updated when the area associated to the obstacle falls inside the range for obstacle marking. In practice if the obstacle bas been moved from the original position, the information relative to its position will be updated and all the AGVs in the fleet will receive the updated position, knowing in real time which obstacles are blocking a pathway, and which obstacles are not blocking a pathway anymore, even if they are still present. The area which was initially associated to the obstacle will be monitored until the expiration of the deadline of the obstacle.

When the information about an obstacle has been received from another agent in the fleet, the obstacle can either be a new obstacle which has been detected by another agent in the fleet or an update regarding an obstacle which information was already received. The difference is related to the presence of the unique id associated to the obstacle in the list. If the UUID which is associated to the obstacle which is received is already associated to an obstacle which is in the list, the received message represents an update. In the case the UUID is not already in the list the obstacle was not already present in the visualization of the robot, it has to be added to it. In case an update is received, first the *marked_indexes* are appended at the end of the indexes which are already part of the object instance associated to the obstacle (a check is performed in order to avoid adding the same index multiple times). At that point the indexes that are present in the *cleared_indexes* are removed from the *marked_indexes* vector. At the end of this process the robot has an updated version of the obstacle that was already present in its visualization and this will reflect in the costmap representation as soon as it is updated according to the update frequency that was set in the configuration.

When an obstacle is updated from the *obstacle_layer*, the procedure to modify the *marked_indexes* and the *cleared_indexes* vectors that are associated to the obstacle consists of the same operations that are performed when an obstacle is received from an external source, explained just above. The difference is that the update is shared with the other robots before performing the operation of appending to the vector and clearing (In the former case the update was received but not shared back as it would cause an unnecessary redundancy of data across the agents in the fleet).

4.4 The sharing handler node

The first change relative to the Python node of the configuration, defined as *Sharing-Handler*, was removing the presence of a YAML list that stores information regarding the obstacles which have been spawned in the visualization.

The advantage of having a human-readable list was to be immediately able to check if all the functionalities relative to the sharing were working properly and to easily detect the presence of malfunctions in the system. A list in the *yaml* format,

though, has no utility if not the one of debugging. It is not necessary to have a list that stores data even when the robot is shut down, as the data which are stored in it would not be up to date at the next restart anyway.

The role of the node in the configuration is to handle the updates which are coming from the *move_base_node* and from the other AGVs via Coaty. In particular the obstacles are received from the *move_base* in two separate topics. When the obstacle is newly detected and needs to be added to the costmap it is handled differently with respect to the case in which the obstacle information is already existing and needs to be updated.

In the "adding" case, first a check is performed on the costmap subsection which is associated to the obstacle in order to verify that the obstacle is not being associated to an area which is already being monitored as part of another shared obstacle. This can be considered as a non redundancy check.

At this point, if the obstacle is not already in the costmap, an UUID is assigned to it and it is sent back to *move_base* and to Coaty. In fact, In order to avoid redundancies in the shared obstacles, when the *obstacle_layer* (and therefore *move_base*) detects a new obstacle and triggers the sharing, before adding the relative information to the costmap it awaits for an authorization by the *SharingHandler* node.

The case relative to the "update" of an existing obstacle can be divided in two sub-cases: one relative to the "external update", i.e. when the obstacle updated information is received from another robot that belongs to the fleet through Coaty, and the other relative to the "internal update", i.e. when the obstacle is updated upon direct observation.

When the obstacle needs to be updated from an external source as another agent in the system, the associated callback function is described in figure 4.6. At first a check is performed in order to verify if the received message contains a newly spawned obstacle in the visualization to add to the costmap or if it is already present and it needs to be updated.

It is important to notice that in the updating case, the indexes that have to be associated to a value corresponding to an obstacle in the visualization, are not overwritten on the previously present indexes in the list but are appended at the end of it. After that the obstacle is added to the vector containing the obstacles that have to be sent to *move_base* and finally the indexes that have to be cleared are removed from the marked indexes list. This is done because an AGV, upon observing an obstacle, could have only a partial observation and overwriting the marked indexes of the obstacle can potentially mean a loss of the data that are relative to the obstacle. For this reason the only two ways that can remove indexes from the obstacle are the direct clearing of an index via raytracing or the removal of the whole obstacle after its deadline expiration.

In the case of "internal update", the obstacle information is handled in the same way as the "external update", with the difference that the obstacle is not sent back to *move_base* but it is sent to the other agents in the fleet via Coaty.

4.5 The handling in Coaty

The role of Coaty in the configuration has not gone through many conceptual modifications. The functions that are performed in Coaty are essentially the same that were performed in the previous configuration, described in chapter 3. The main difference is that all the functions that were implemented had to be adapted to the new configuration. The structure of the objects that were shared among the different Coaty agents is an implementation of the message structure that was previously described in this chapter. As it is used only for the transferring of data, all the information either relative to "adding" or "updating" is simply routed to the *SharingHandler* node or to the other agents, according to the source of the information that can be respectively "external" (another Coaty agent) or "internal" (direct observation of the obstacle that leads to an update).

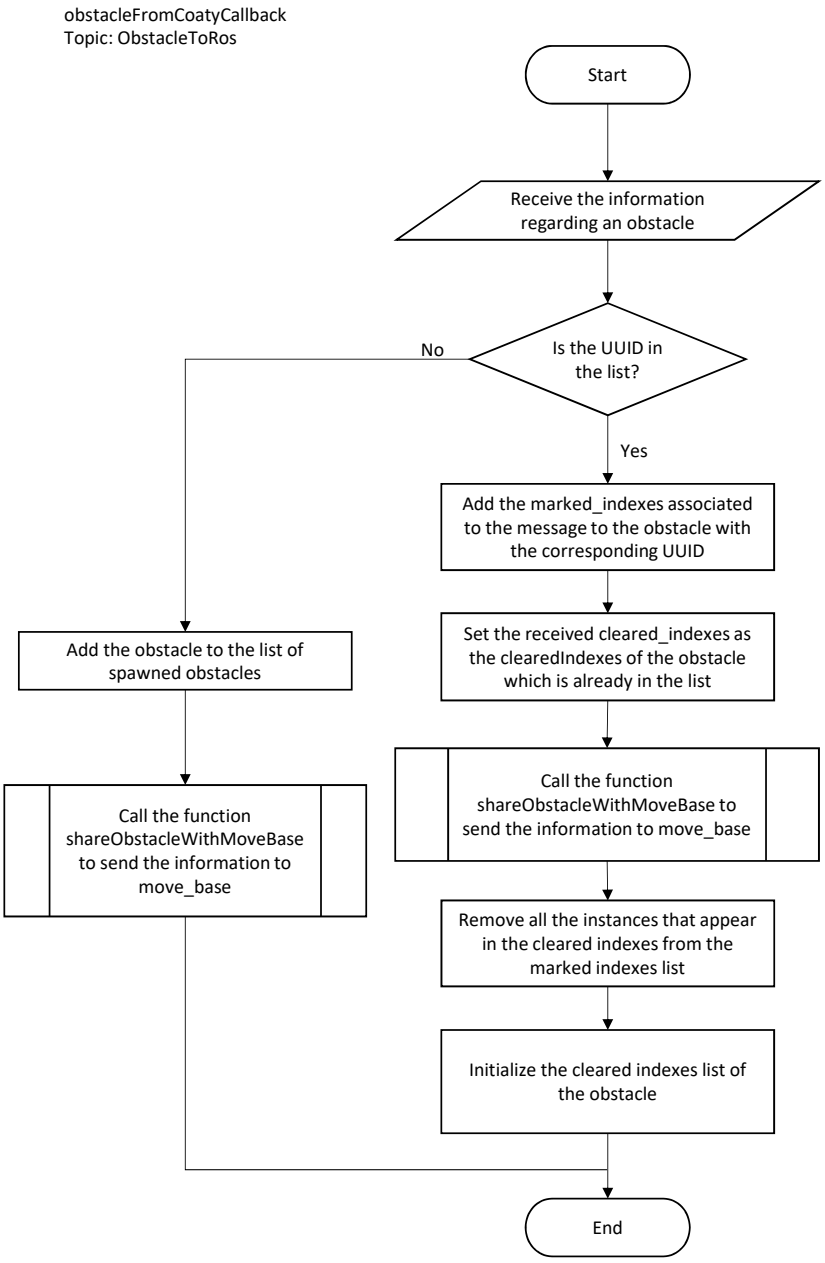


FIGURE 4.6: Flow chart of the obstacleFromCoatyCallback function

Chapter 5

Simulation Tests

5.1 Introduction

In this chapter the results relative to the tests that were performed are reported. The tests have the objective to verify the functionalities that were implemented. In particular the functionalities to test are:

- Sharing information about a detected obstacle
- Reacting upon receiving information about an obstacle
- Being able to clear points of an obstacle and sharing the update
- Removing an obstacle after its deadline has expired

These functionalities were tested in the Stage simulator [11]. The visualization in the two simulated robots is shown by using RVIZ.

5.2 Test 1

The functionalities that have to be validated with this test are:

- Sharing information about a detected obstacle
- Reacting upon receiving information about an obstacle

In figure 5.1, the dark grey area of the map represent the a-priori known walls. The blue parallelepipeds represent the obstacles which are not known in advance. The two cubes in the lower right of the image represent the two AGVs in the simulated environment.

In this test the blue robot will first be sent to reach the goal associated to the orange rectangle in figure 5.1. The blue robot has no information about the presence of two obstacles that are blocking the two routes that would normally be faster than the one highlighted with a red arrow in figure 5.1. For this reason it will first try and reach the goal position via the two blocked routes before finding the only route that allows it to reach the goal.

After the first robot has reached the orange goal, the green robot will be sent to the goal position associated to a purple square in figure 5.1. As the information

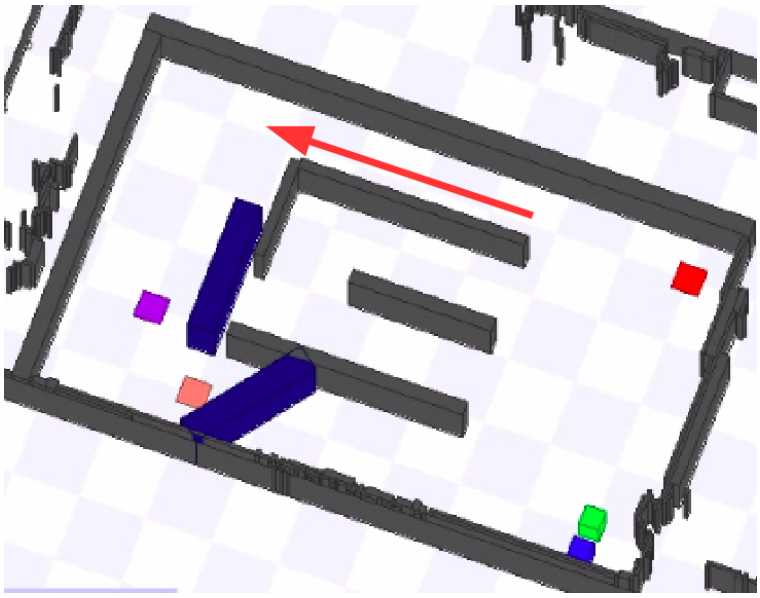


FIGURE 5.1: The highlighted path is the only valid path to reach the goal

about the first two path being blocked will be shared according to the implemented sharing functionality, if the test is successful the green robot will immediately find the only available route.

In the figure 5.2 the first image of the sequence is shown. On the lower part the simulation environment is shown. on the upper part there are the RVIZ visualizations, of the blue and the green robot. In the visualizations the walls are represented in yellow. The areas that are circling the walls are coloured with a different color depending on the proximity to the obstacles. These values represent the effect of the *inflation_layer* as their cost has been increased according to the proximity to obstacles. The green empty squares represent the robot footprint that gives the position of the robot in its own visualization of the surrounding environment. The red and pink dots in the blue robot visualization represent the points where the laser scanner of the blue robot has hit a surface. It is possible to notice that these dots do not precisely correspond to the points of the wall in the a-priori known environment which is marked on the visualization by the *static_layer*. This misalignment represents the effect of the tolerance on the precision of localization.

In figure 5.3 the blue robot received the goal position and has started moving towards it. As it gets closer to the first obstacle, the relative points are added in its costmap but are not shared with the other robot as the obstacle has not been completely detected yet, therefore the blue robot does not have enough information to know that that specific obstacle is blocking its path. It is important to notice that, even if the laser scanner could already detect the presence of the obstacle from the starting position, it is added to the costmap only after it is inside the maximum range for adding an obstacle in the visualization, that has been set to 2.5 m in the test. The reason for adding the obstacles to the costmap only if they are within a certain range from the sensor and not using all the 30 m range of the sensor is that the precision of the positioning of the robot, expecially regarding the angular positioning, generates an error on the detection which increases linearly with distance proportionally to the

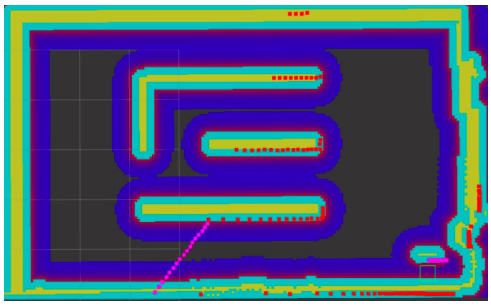
angular positioning error of the robot.

In figure 5.4 the blue robot has finally detected that the obstacle is blocking its path and therefore the relative information has been shared with the other AGV. It is important to notice that even if the green robot is perceived as an obstacle in the visualization of the blue robot, the information relative to it is not shared as the green robot is not blocking its path.

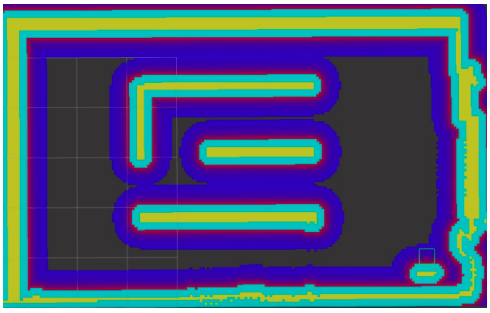
In figure 5.5 the fourth frame of the test is represented. The blue robot has reached the second obstacle and it has shared the relative information with the other agent in the system.

At this point the robot is able to identify the only route which is not blocked and is able to reach the assigned goal, as can be seen in figure 5.6 and in figure 5.7. While the robot is moving from the position that is shown in figure 5.6 to the one that is shown in figure 5.7, the whole shape of the second obstacle is detected and its shape is updated only regarding the costmap sector that was previously shared. This can be seen from the change in the shape of the obstacle in the green robot visualization. As can be noticed, the complete shape of the obstacle is not visible in the visualization of the blue robot in figure 5.7. This because all the information relative to the obstacle_layer is deleted upon reaching a goal. The only information that remains is the one associated to the obstacles that are currently detected by the scanner and are within the range associated to the obstacle marking. As can be noticed in figure 5.7, the obstacles which information has been shared are not affected by this clearing behaviour.

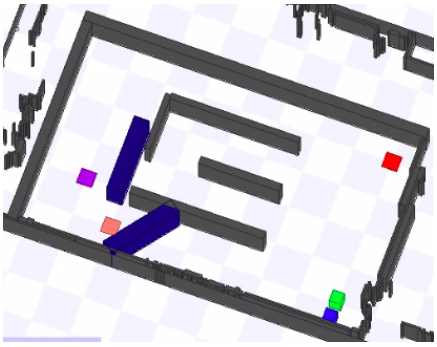
Once the blue robot has reached the assigned goal, the green robot is sent to the goal position associated to the purple square in the environment representation in figure 5.8 (C). In figure 5.8 (B) it is possible to visualize the path computed by the green robot as a red line. This path, as expected, takes already in consideration the presence of the two obstacles which data had been previously received. Figure 5.9 and figure 5.10 show the green robot following the previously computed path and reaching the assigned goal.



(A) robot_0 visualization (blue robot)



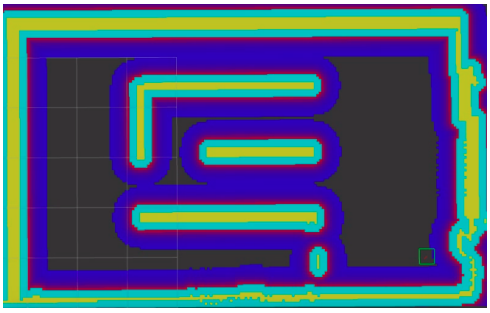
(B) robot_1 visualization (green robot)



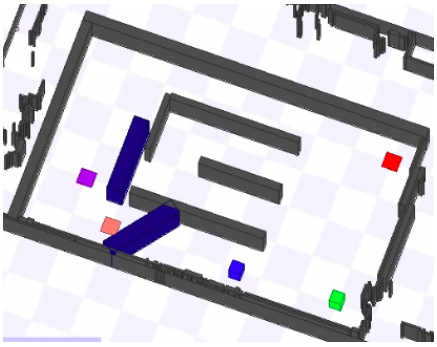
(C) simulation environment

FIGURE 5.2: test 1, frame 1

(A) robot_0 visualization (blue robot)



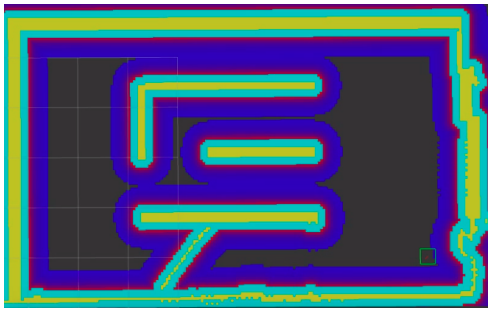
(B) robot_1 visualization (green robot)



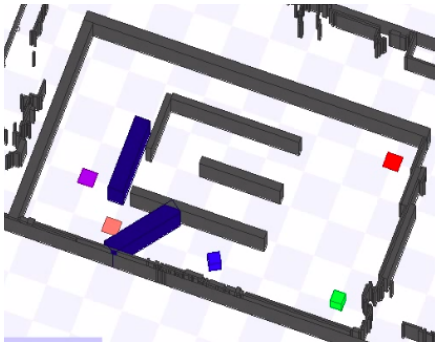
(C) simulation environment

FIGURE 5.3: test 1, frame 2

(A) robot_0 visualization (blue robot)

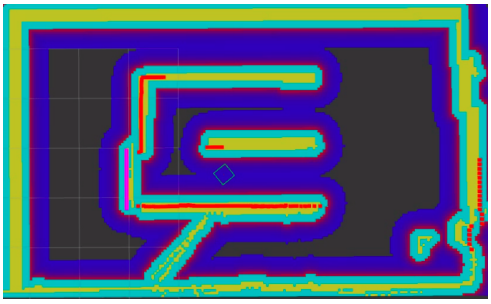


(B) robot_1 visualization (green robot)

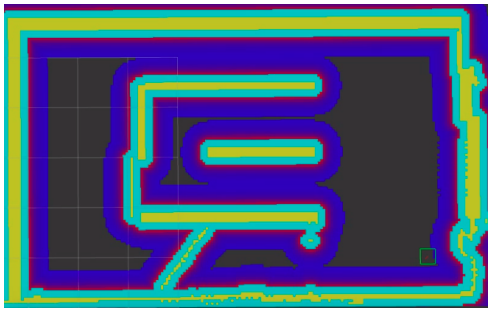


(C) simulation environment

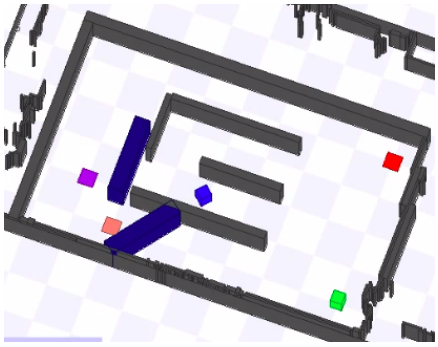
FIGURE 5.4: test 1, frame 3



(A) robot_0 visualization (blue robot)

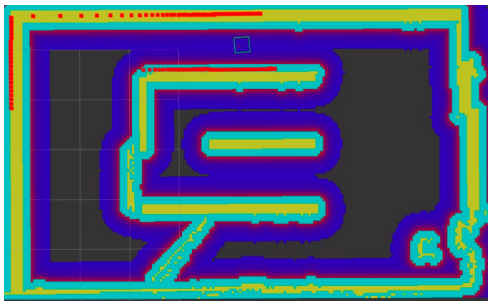


(B) robot_1 visualization (green robot)

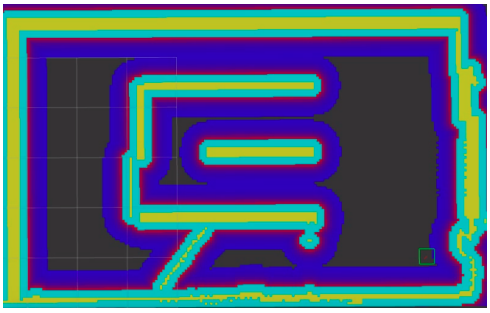


(C) simulation environment

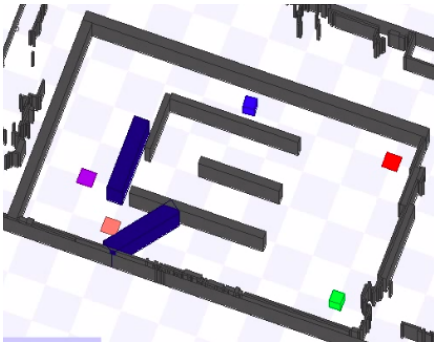
FIGURE 5.5: test 1, frame 4



(A) robot_0 visualization (blue robot)

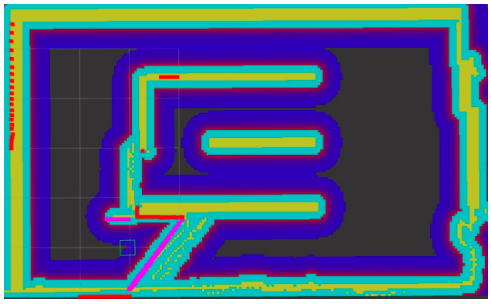


(B) robot_1 visualization (green robot)



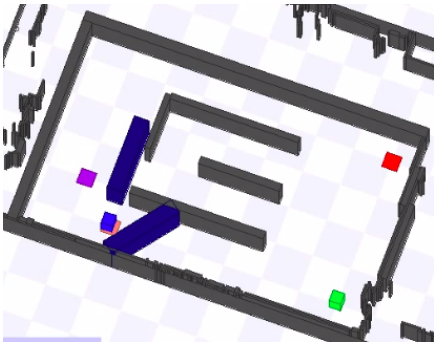
(C) simulation environment

FIGURE 5.6: test 1, frame 5



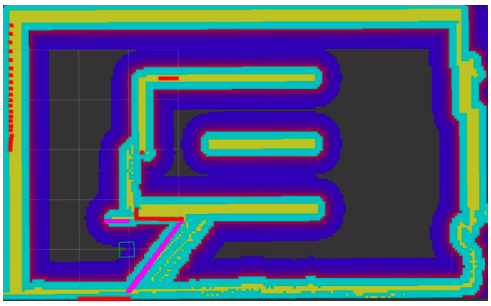
(A) robot_0 visualization (blue robot)

(B) robot_1 visualization (green robot)

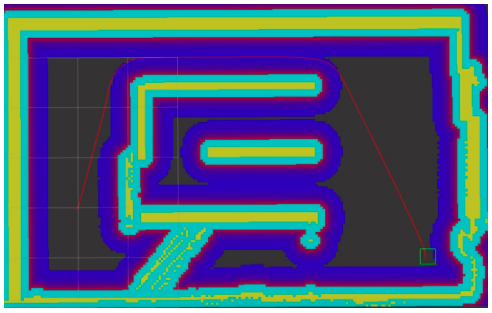


(C) simulation environment

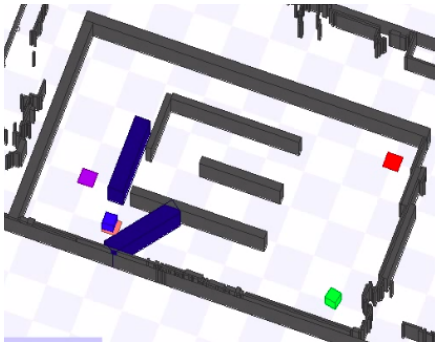
FIGURE 5.7: test 1, frame 6



(A) robot_0 visualization (blue robot)

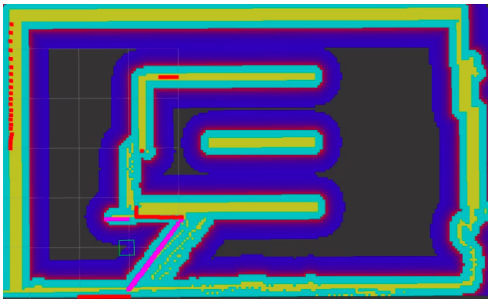


(B) robot_1 visualization (green robot)

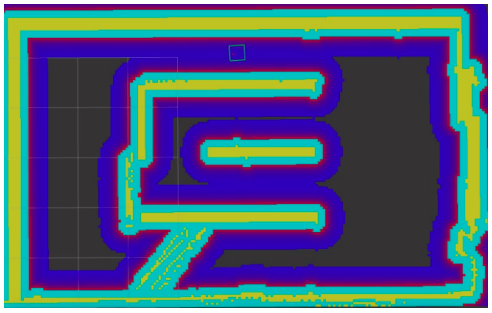


(C) simulation environment

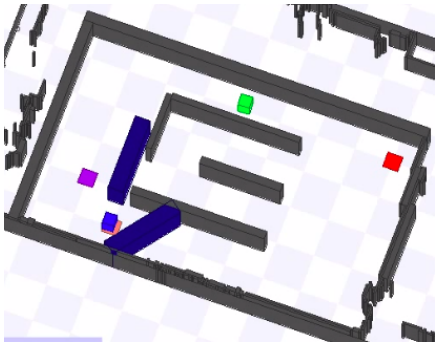
FIGURE 5.8: test 1, frame 7



(A) robot_0 visualization (blue robot)



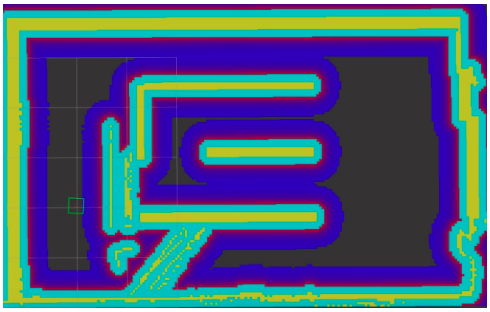
(B) robot_1 visualization (green robot)



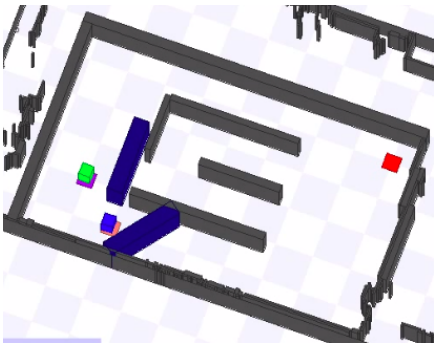
(C) simulation environment

FIGURE 5.9: test 1, frame 8

(A) robot_0 visualization (blue robot)



(B) robot_1 visualization (green robot)



(C) simulation environment

FIGURE 5.10: test 1, frame 9

5.3 Test 2

The functionalities that have to be validated with this test are:

- Being able to clear points of an obstacle and sharing the update
- Reacting upon receiving an update about an obstacle which information had been already shared

In this test we will verify the behaviour of an agent of the system upon receiving an update about an obstacle which information was already shared. In particular the blue robot will be sent to the orange goal and will find an obstacle blocking its path, as shown in figure 5.11. Once the goal position has been reached, the obstacle, that is visible to the laser scanner of the robot, will be removed. The information will be sent to the green robot. At that point the blue robot will be sent to the goal position associated to the red square. Finally the green robot will be sent to the goal position associated to the orange rectangle. If the information about a change in the position of the shared obstacle has been successfully shared, the green robot will know that the route is now clear and will proceed straight to the goal position.

In figure 5.12 the starting position of the test is shown. A detailed description of the role of the different elements in the visualization of the agents in the system was given previously in this chapter (in the "Test 1" section).

In figure 5.13 It is possible to observe that the obstacle has been already detected by the blue robot and the relative information has been shared, at this point, the blue

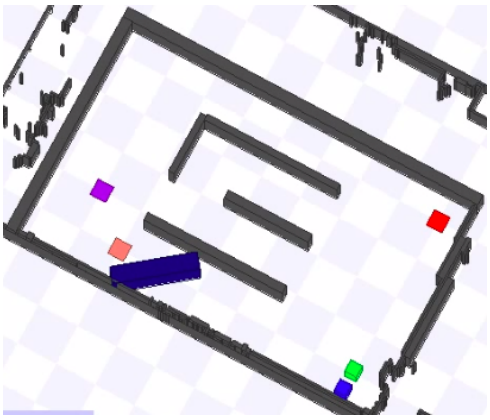


FIGURE 5.11: Test 2 setup

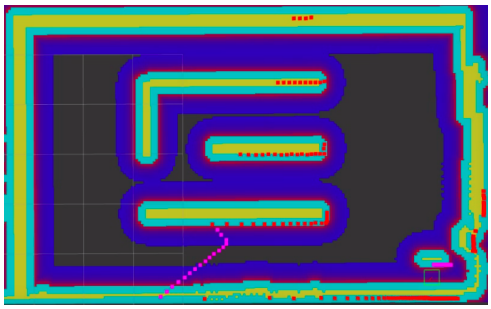
robot will recalculate a path to reach the assigned orange goal, as can be seen in figure 5.14.

In figure 5.15 the blue robot has reached the assigned goal. The next part of the performed test has the aim to verify the ability of the system to handle the removal or the shape modification of an obstacle. If the obstacle is moved from its original position, and this could mean removing either completely or partially the obstacle, the robot that is able to perceive this modification has to communicate it to the other agents in the system. This allows all the agents in the system to have a representation of the working environment that adheres as much as possible to reality.

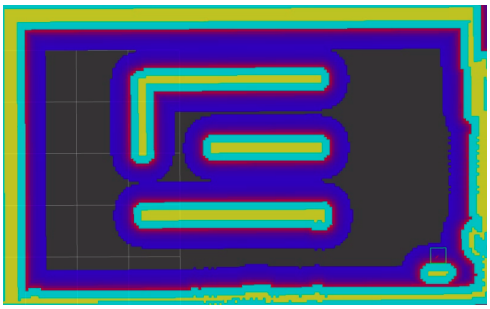
Figure 5.16 (C) shows that the obstacle has been removed from the working environment. The absence of the obstacle has been detected by the laser scanner of the blue robot, which has removed the relative information from its global costmap as can be seen in figure 5.16 (A), and has shared the update with the green robot, as can be seen in figure 5.16 (B).

In absence of obstacles, it is clear to see that the fastest route that leads the green robot to the orange goal is the one that goes straightforward from the starting position to the goal position. As we already checked (in "Test 1") that the presence of an obstacle which information has been shared is capable of affecting the computation of a path from the robot which is receiving the information, we want to verify that also an update of the shape of the obstacle in the visualization is able to affect the path computation.

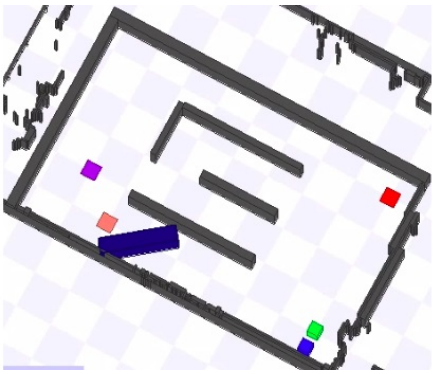
In figure 5.17 (C), once the blue robot has been sent to the red goal position, the green robot is sent to the orange goal position. From what is shown in figure 5.18 and in figure 5.19, we can see that the path that the robot is following is affected by the removal of the obstacle which information were previously shared by the blue robot.



(A) robot_0 visualization (blue robot)

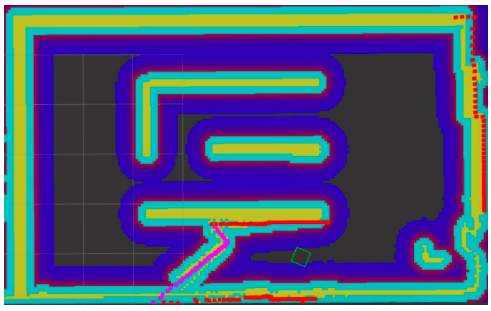


(B) robot_1 visualization (green robot)

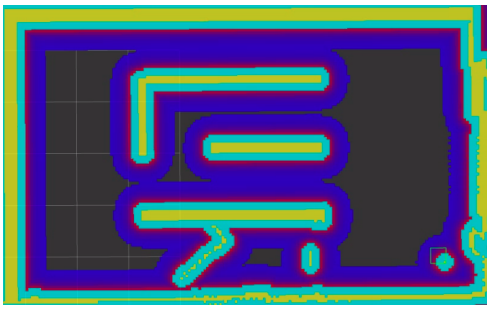


(C) simulation environment

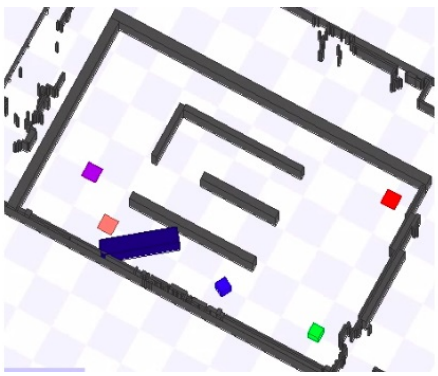
FIGURE 5.12: test 2, frame 1



(A) robot_0 visualization (blue robot)

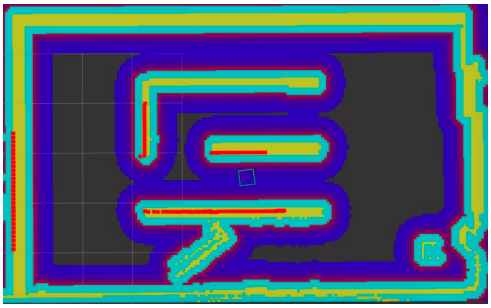


(B) robot_1 visualization (green robot)

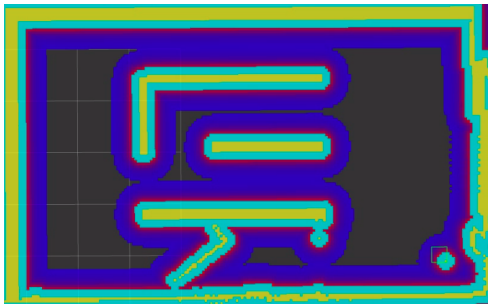


(C) simulation environment

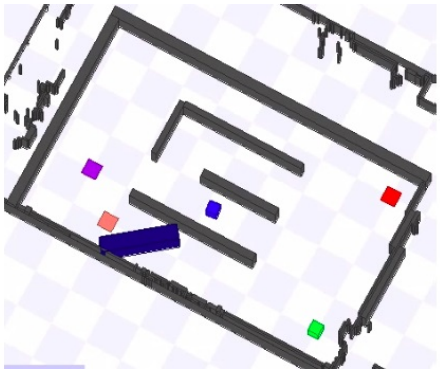
FIGURE 5.13: test 2, frame 2



(A) robot_0 visualization (blue robot)

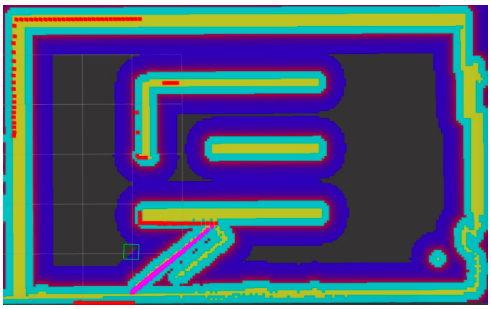


(B) robot_1 visualization (green robot)

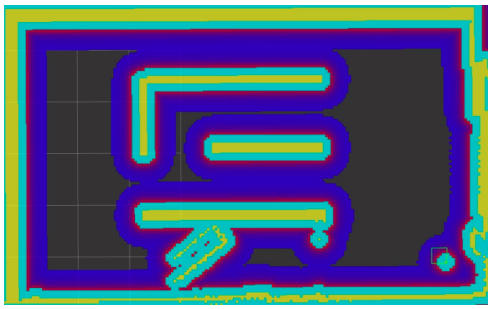


(C) simulation environment

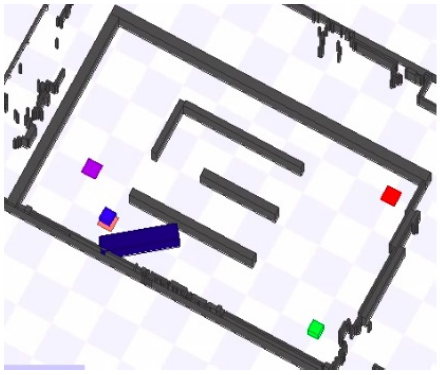
FIGURE 5.14: test 2, frame 3



(A) robot_0 visualization (blue robot)

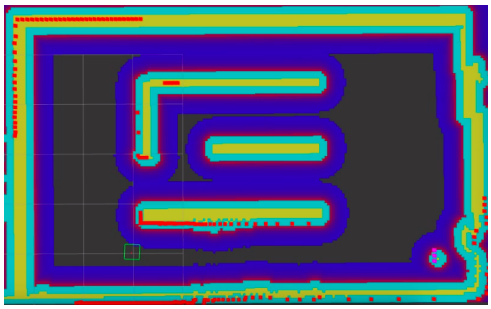


(B) robot_1 visualization (green robot)

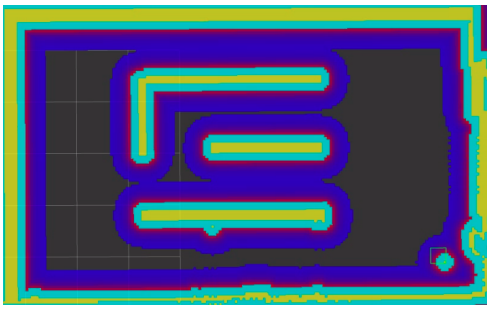


(C) simulation environment

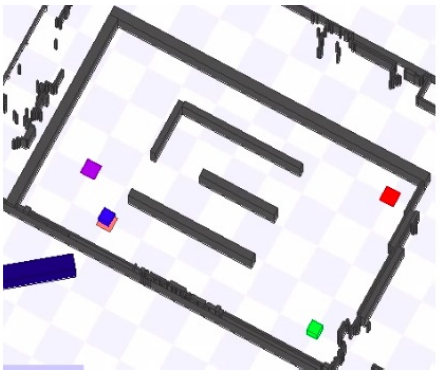
FIGURE 5.15: test 2, frame 4



(A) robot_0 visualization (blue robot)

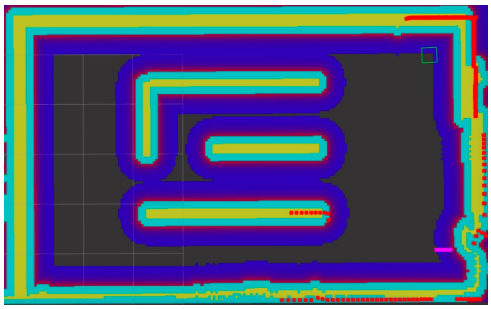


(B) robot_1 visualization (green robot)

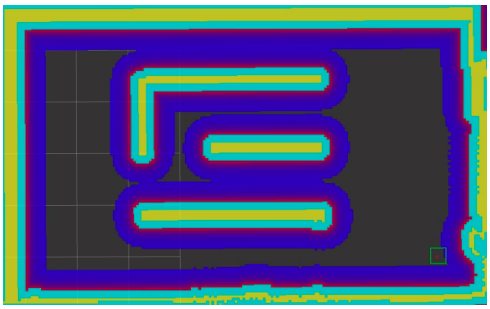


(C) simulation environment

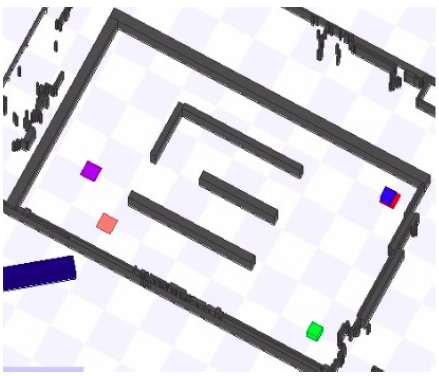
FIGURE 5.16: test 2, frame 5



(A) robot_0 visualization (blue robot)

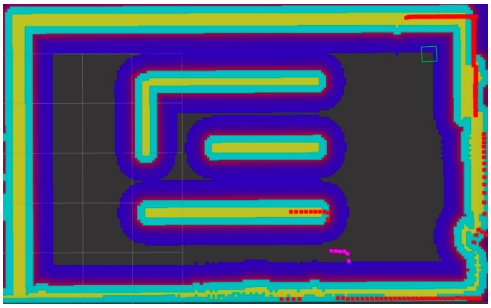


(B) robot_1 visualization (green robot)

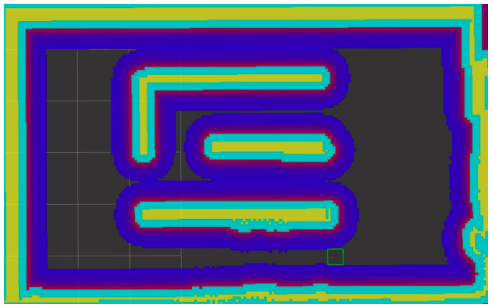


(C) simulation environment

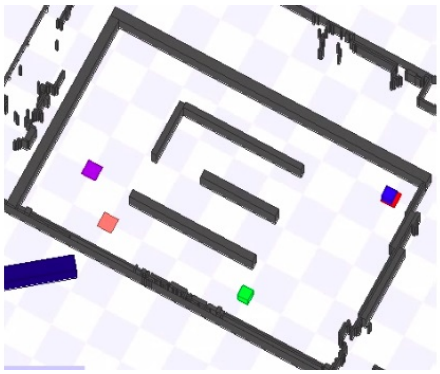
FIGURE 5.17: test 2, frame 6



(A) robot_0 visualization (blue robot)

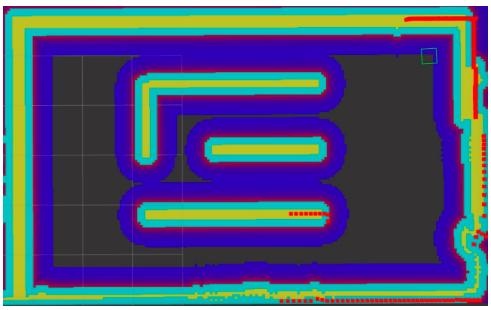


(B) robot_1 visualization (green robot)

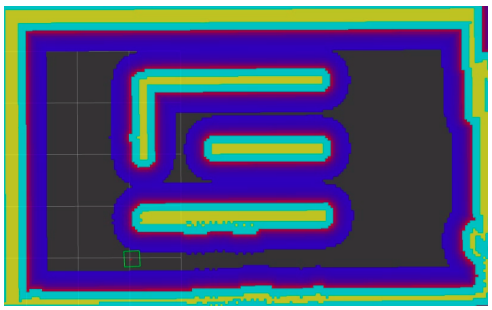


(C) simulation environment

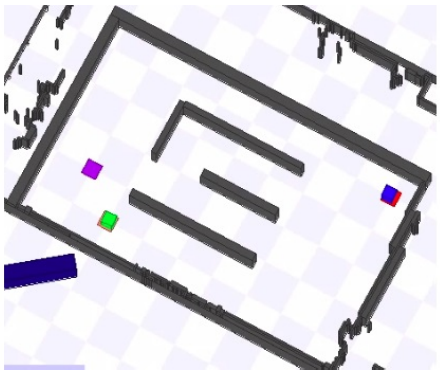
FIGURE 5.18: test 2, frame 7



(A) robot_0 visualization (blue robot)



(B) robot_1 visualization (green robot)



(C) simulation environment

FIGURE 5.19: test 2, frame 8

5.4 Test 3

The purpose of this test is to demonstrate the behaviour of the expiration date that is assigned to the obstacles which information is shared among the agents in the system. The starting position of this test is obtained at the end of the sequence of "Test 2". Therefore, starting from the ending position of the previous test, shown in figure 5.19, the obstacle that was removed will be added again in the environment and will be detected by the green robot that will share the information with the other robot. At this point the green robot will be sent to the goal associated with a purple square (visible in figure 5.19 (C)). At this point the objective is to see what happens after the expiration date that was assigned to the sharing of the information relative to that specific obstacle.

When an obstacle is detected to block a route in the working environment the information that is shared is not only made by the points that were associated to it at the moment of its identification. It would be more correct to say that when an obstacle like the one just described is detected, the functionality that is triggered will monitor the area in which the obstacle was first detected for a configurable amount of time. This means that, in the moment the obstacle is removed from that position, the area which was initially assigned to the obstacle will continue to be monitored until the expiration date. In the case a new obstacle appears in that area, each robot in the fleet will be able to share the relative information in case the obstacle enters in its range for obstacle detection.

There is, though, the possibility that an obstacle that blocks, for example, a corridor in the facility is never detected anymore because, since that corridor is blocked, no robot will pass near enough to the position that was associated to the obstacle in order to share an update about its presence. The purpose of the expiration date is to manage this situation, avoiding that the presence of an obstacle in the visualization of the robots in the fleet can permanently block a corridor even when the obstacle is removed.

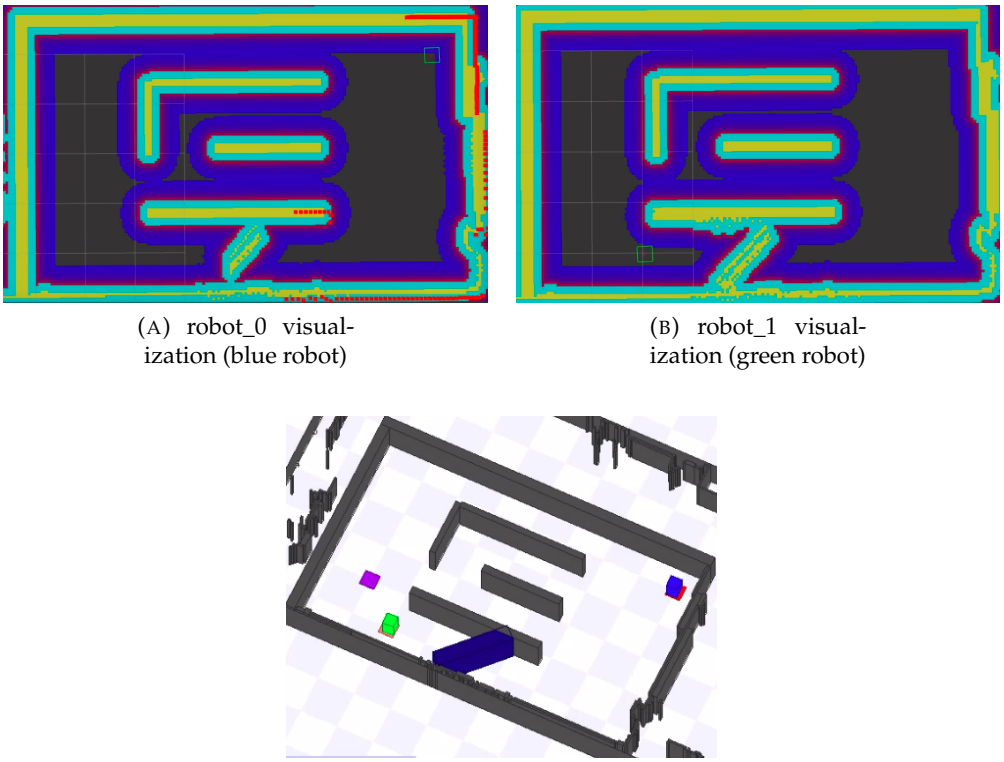
In figure 5.20 the first frame of the sequence is shown. The obstacle which was previously removed has been added again and, as the are associated to the obstacle is inside the obstacle marking range of the green robot, the update relative to the presence of the obstacle has been shared with the blue robot. It is possible to note that the obstacle is formed by multiple lines. This is because in the process of adding back the obstacle to that position it was moved towards the green robot, those lines are the positions where the obstacle was detected while it was moving. The points belonging to the previous positions are not cleared because the obstacle is moved towards the robot and, therefore, there is no way of clearing the points that are not visible to the laser scanner via raytracing.

another thing that has to be noted is that the obstacle that is shown in the representation of the blue robot in figure 5.20 (A) is slightly different in shape from the obstacle which is visualized by the green robot in figure 5.21 (B). This happens because when the obstacle was moved back in the simulation environment map, it was moved in a position that does not coincide precisely to its previous position. For this reason only the information relative to the section of the obstacle that entered in the area that is being monitored has been shared. The rest of the information is just

present in the visualization of the green robot as a normal obstacle.

In figure 5.21 in fact, the green robot has been sent to the goal position associated with the purple rectangle in the simulator, and the obstacle is now outside of the obstacle marking range of the green robot. As the data relative to the *obstacle_layer* is cleared upon reaching a goal, the only information remaining in the visualization is the one relative to the obstacles which information has been shared. The information that the two robots have about the obstacle is, in fact, from what can be seen in figure 5.21, the same.

In figure 5.22, since the the expiration date associated to the obstacle has passed, the obstacle has been removed from the costmap of both the robots. If one of the two robots will be sent to a goal it will have no information about the presence of an obstacle in the position that is shown in figure 5.22 (C). If the path will then be blocked by the presence of that obstacle, a new sharing will be triggered.

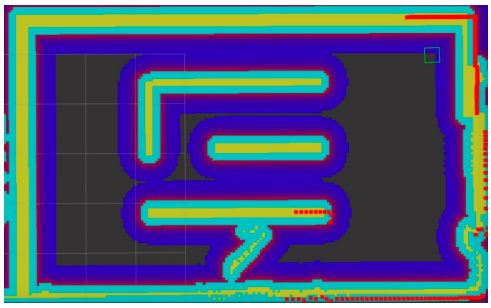


(A) robot_0 visualization (blue robot)

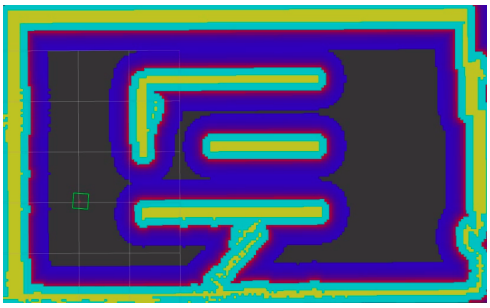
(B) robot_1 visualization (green robot)

(C) simulation environment

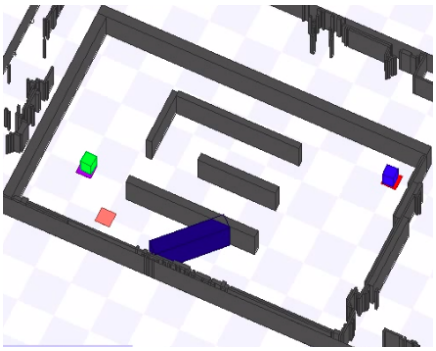
FIGURE 5.20: test 3, frame 1



(A) robot_0 visualization (blue robot)

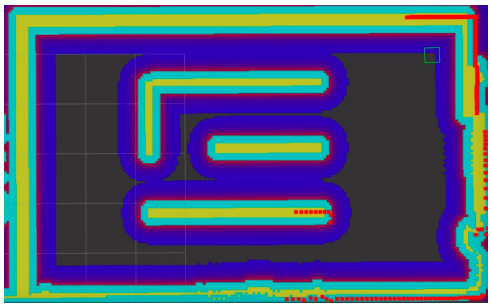


(B) robot_1 visualization (green robot)

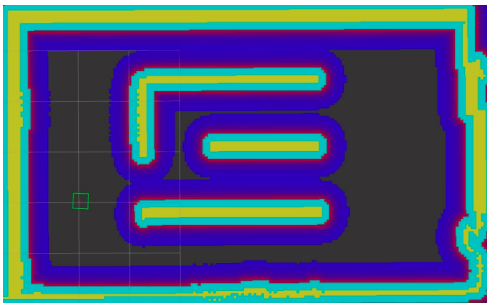


(C) simulation environment

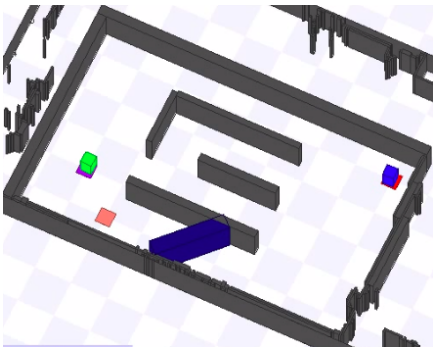
FIGURE 5.21: test 3, frame 2



(A) robot_0 visualization (blue robot)



(B) robot_1 visualization (green robot)



(C) simulation environment

FIGURE 5.22: test 3, frame 3

Chapter 6

Conclusions & Future Work

6.1 Results

The purpose of this project was to develop a functionality that provided the robots in a fleet with the capability to handle the presence of obstacles in a working facility in a collaborative way. The AGVs in the fleet have to be capable to share the information regarding the presence of obstacles that are blocking a pathway in a facility with the others and to react upon receiving such information. As a result the robots will then be able to be aware of the presence of obstacles without self-observation. This will lead to an increased capacity of the fleet in presence of obstacles as the AGVs will, then, be able to optimize their path calculation with an increased awareness of the context obtained through collaboration. A system like this represents a big step forward in the development of collaborative robots which has been the main focus of the research over the past years at **InSystems Automation**.

The system that was developed had to overcome the fact that the visualization system which is implemented in ROS was not meant to perform such functionalities, and therefore the need to adapt the current system to the goal that was assigned for this project.

The solution is based on the monitoring of areas in which the presence of an obstacle blocking a pathway has been detected. When an AGV in the fleet detects such an obstacle for the first time, all the AGVs in the fleet will then be able to monitor the presence of the obstacle and update its status. Another possible problem that needed to be addressed is the fact that when an obstacle is blocking, for example, a corridor in a plant, it is likely that the other AGVs in the fleet, having the information about the corridor being blocked, will not pass through that pathway, thus having no way of checking whether the obstacle is still present or if it has been removed. This would cause a malfunction of the system that would not be able to delete the obstacle from the map even when the obstacles have been removed. To address this problem an expiration date has been introduced, which is configurable according to the specific use-case scenario and that is assigned to each obstacle which is subject to sharing.

The functionalities that were developed and that have been tested in simulation are the following:

- Sharing information about a detected obstacle.

- Reacting upon receiving information about an obstacle.
- Being able to clear points of an obstacle and sharing the update.
- Removing an obstacle after its expiration date.

The major issue that was detected in the system and had to be solved is the positioning precision of the robot, that generates two side problems in the configuration:

- The default operation of the *obstacle_layer* is related only to the information coming from the sensors. As part of the configuration a way to discern between obstacles and parts of the static environment which is known a-priori was implemented. As a side effect of the presence of a tolerance on the positioning precision of the AGV, the *obstacle_layer* will not be able to correctly associate the points detected on the wall with the a-priori known information given by the mapping process of the static layout of the plant and will try to perform the sharing on those as well. In order to solve this problem, a tolerance was introduced that compensates this error.
- Another side problem was determined by the fact that if an obstacles is identified and is associated to a set of points, as the robot moves, the obstacle could be then associated to a different set of cells and therefore it would not be detected any more as the "same" obstacle. This has been solved with the introduction of the monitoring of the area associated to the obstacle. In this way, if the cells associated to the obstacle change they can be simply updated and the update shared across the AGVs in the fleet.

The solution that was adopted has been verified to grant a good robustness against the issues that were just explained.

6.2 Next steps

The natural next step regarding this case of study is the deployment of the designed functionality on the AGVs, which will be the focus of the further research. Moreover, there still are some points open for improvement relating in particular to the assignment of the expiration date. In fact the system performance relating to the sharing functionality is connected to the ability of the AGVs to set an expiration date for the obstacles which is as close as possible to the real moment in which the obstacle is removed. This way of dynamically changing the expiration date can be achieved through an adaptation algorithm. In the current system the expiration date can be manually configured but an optimal configuration of this parameter would require to have data indicating the mean time in which an obstacle blocking a path is removed. **InSystems Automation** has already shown interest towards further research in this field.

Another possible improvement that needs further investigation is the ability of the robot to differentiate between obstacles. At the moment the robot will share everything in the selected areas that does not belong to the a-priori known environment and will treat the areas as distinct entities. An improvement to this could be achieved by implementing an algorithm that allows the AGV to associate each obstacle to a different area to monitor independently.

Bibliography

[1] Siemens AG. *Coaty documentation*. 2018. URL: https://coaty.io/nutshell.

[2] Open Source Robotics Foundation. *costmap_2d documentation*. URL: http://wiki.ros.org/costmap_2d.

[3] Open Source Robotics Foundation. *move_base documentation*. URL: http://wiki.ros.org/move_base.

[4] Open Source Robotics Foundation. *ROS Concepts*. URL: http://wiki.ros.org/ROS/Concepts.

[5] Open Source Robotics Foundation. *ROS Introduction*. URL: http://wiki.ros.org/ROS/Introduction.

[6] Open Source Robotics Foundation. *Rviz documentation*. URL: http://wiki.ros.org/rviz.

[7] Open Source Robotics Foundation. *the navigation stack*. URL: http://wiki.ros.org/navigation.

[8] InSystems Automation GmbH. URL: http://www.insystems.de/en/insystems-2/.

[9] InSystems Automation GmbH. *CrESt – Collaborative Embedded Systems*. URL: http://www.insystems.de/en/crest-kollaborierende-eingebettete-systeme/.

[10] David V. Lu , Dave Hershberger , William D. Smart. "Layered Costmaps for Context-Sensitive Navigation". In: *IEEE RSJ International Conference on Intelligent Robots and Systems* (2014).

[11] Richard Vaughan. *Stage simulator documentation*. URL: http://rtv.github.io/Stage/.

[12] Wikipedia. *Automated guided vehicle*. URL: https://en.wikipedia.org/wiki/Automated_guided_vehicle.

[13] Wikipedia. *Universally unique identifier*. URL: https://en.wikipedia.org/wiki/Universally_unique_identifier.