POLITECNICO DI TORINO

Master's Degree in Mechatronics Engineering

Master's Degree Thesis

Robot Fleet Control for Mining Applications Using OpenRMF and OpenTCS

Prof. Fabrizio Lamberti

Prof. Javier Ruiz del Solar

Dr. Felipe Inostroza

Candidate

ANGELO SALZILLO

December 2024

Table of Contents

List of Figures

Chapter 1

Introduction

The mining industry is nowadays continuously evolving in order to face increasingly complex challenges and to improve the productivity while maintaining a high level of safety.
The mining environment indeed is a safety critical environment, since the tasks that have to be performed in order to extract materials and transport them are risky processes that could severely harm the people working inside such environment.
Along the years different solutions have been studied, developed and finally implemented to reduce the human contribution to the minimum possible, make the mine a safer and safer place and reduce the number of accidents.
If the mining tasks are executed by machines instead of humans, the safety level automatically improves, since in case of mine collapse only the machines should be damaged. Anyway, implementing a mining system where the tasks are executed by machines, implies necessarily the presence of a control system that allows to manage and supervise such machines
This thesis work aims at the study and development of a control system able to manage multiple machines inside a mining environment, allowing them to move and perform tasks autonomously or with the minimum human intervention.
In this thesis to any kind of autonomous machine will be referred to as a robot.

1.1 Hypothesis

One of the most important parameters, that allow to evaluate the production performances of a production system inside a mining environment, is the total cycle time. It is defined as the time needed to perform a cyclic operation, that is for instance the charge and discharge of the material that has to be transported inside the mine.
Currently, the mining system took under analysis consists of different fleets of

robots, each one controlled by a fleet management system and executing a specific task, for instance clean tasks and delivery tasks. If needed to execute both the tasks then, it is necessary to let the first fleet terminate the execution before allowing the second fleet to execute the second type of task.
The hypothesis that led to the development of this thesis work and that will be verified later on in the following chapters, is that it is possible to develop an application controlling at the same time different fleets of robots, devoted to the execution of different tasks, reducing the total cycle time with respect to the previous example.
Anyway, this hypothesis has to be verified because, while allowing the execution of all the tasks at the same time could effectively reduce the cycle time making the robots exploit the time available at its maximum, this also implies having more machines simultaneously in the environment. This in turn could cause the worsening of the traffic conditions, the creation of way more complex conflicts among the routes the robots have to follow, and also an increase of the amount of time the machines need to wait in idle state.

1.2 Objectives

The general objective of this thesis work is the development of a control system to monitor an ensemble of robots in charge of executing tasks in a mining environment. Specifically, the tasks took into account are the following:

- Patrol task, consisting in ordering the robot to move to a point or a set of points, letting possible to specify how many times the robot has to cover the same path.
- Clean task, consisting in cleaning an area of the environment.
- Delivery task, consisting in loading material at a specific location and then dropping it off at the drop off point.

Furthermore, the control system developed has to take into account the following assumptions: each robot in the environment is part of a fleet of robots, where a fleet of robots is an ensemble of machines that share the same physical features, implementation and the same area they can cover inside the map; a robot fleet can perform only one task between the delivery and clean ones; all the robots can perform patrol tasks.
To be precise, the just mentioned objective can be divided in more specific goals, that allow to better understand the aim of this thesis work, that are:

1. First of all the creation of the simulated mining environment, that has to be

a good representation of the mine itself, including the machines that will be used to perform the tasks as well.

2. The development of the control system, for which two different approaches have been studied:

   - Using the OpenTCS (Open Transportation Control System) framework, a fleet manager, designed specifically for single fleet systems. Using this framework denies the possibility to perform cleaning and delivery tasks at the same time, since will be executed by different robot fleets.
   - Using OpenRMF (Open Robotics Middleware Framework), that instead has an implemented interface that permits the communication between multiple robot fleets, allowing all the tasks to be executed at the same time.

3. The comparison of the two solution developed to assess which one yields the best results in terms of performance.

Chapter 2

State of the art

The first important thing needed in order to develop the applications mentioned in Chapter 1 is for sure a valid layout of a mine.
Starting from this point, is then possible to generate the simulated environment that will follow the aforementioned layout.
Specifically, the layout exploited for this thesis work is taken from the study of Cristofer Daniel Hernandez Larenas (2021) [1].
This study proposes a layout applicable for different extraction methods, specifically the block caving and panel caving methods.
This layout aims at maximizing the production performances considering the following improvements:

- The quantity of transported material.
- The uniformity of the transported material.
- The amount of interference among the LHDs (Load Haul and Dump), the machines exploited inside this mining environment.

A graphical representation of said layout is shown in Figure 2.1, where it is possible to notice all the key areas of the mine and how they are interconnected.

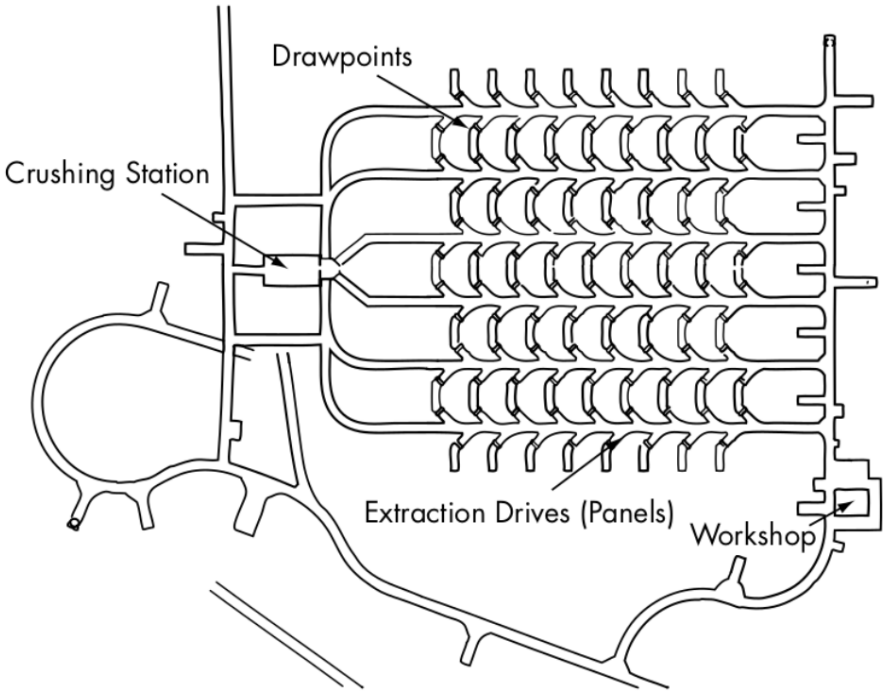

Figure 2.1: Representation of the mine layout [1]

According to this layout, it is possible to distinguish between four main areas, that are:

- The *Workshop*, that is the place where the machines of the environment can wait idle and where maintenance can be performed.
- The lanes in which the machines are displaced, represented by the six horizontal lanes in Figure 2.1, through which the machines can move and reach the places where they can perform tasks.
- The *Drawpoints*, where the machines can pick up the material to be transported an later on dropped off.
- The *Crushing Station*, where all the machines drop off the material taken from the drawpoints.

Following this layout then, all the delivery tasks should consider the Crushing Station as dropoff point, while picking up the material at the various drawpoints. Also, at the beginning of the simulation, the simulated machines should be generated at the Workshop, since this is the only area where they can be idle waiting for a

new task request to be allocated.
The loading process at the drawpoints is depicted in Figure 2.2.

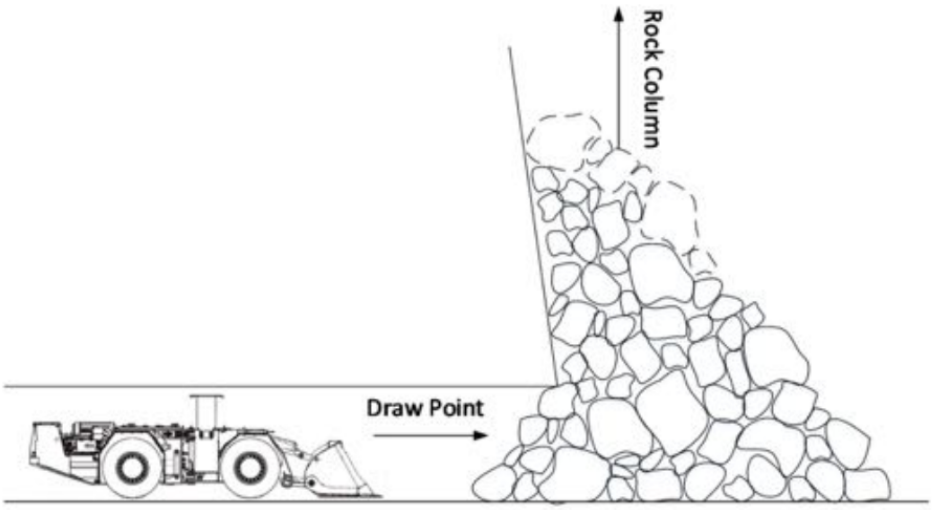

Figure 2.2: Loading process [2]

The second factor that has to be taken into account for the development of said applications is the kind of vehicle that will be used inside the mining environment and how it will accomplish the navigation issue within it.
The vehicle exploited in such a mining environment is the LHD (Load Haul and Dump), provided with a frontal bucket to load and unload the material. Specifically, according to [3], since the tunnels of the mining environment are GNSS-denied, the LHDs cannot rely on GNSSs (Global Navigation Satellite Systems) to localize themselves, and need to count on an alternative localization system.
The localization and navigation system implemented in this thesis work is derived from the one described the mentioned study, according to which it is necessary to:

- Have a map of the mining environment. This has to be built from the measurements taken by LiDAR sensors mounted on the LHDs, as they move inside the tunnels of the mine. As explained more in detail in Chapter 4, this will be done exploiting the *Cartographer* ROS2 (Robot Operating System 2) packages, that account for this.

- Have a topological representation linked to the map itself, providing names and properties to all the relevant locations within the mine and allowing route planning and self-localization. To be precise, the generation of the topological representations (one for OpenRMF and one for OpenTCS) has been done,

respectively, exploiting the *Traffic Editor* and the *Plant Overview Client* GUI (Graphical User Interface).

- Have a navigation and localization system that exploits the scan matching betweeen the current LiDAR measurements and the LiDAR landmarks contained in the topological representation of the map. For the development of the applications, the Nav2 ROS2 packages have been chosen for this purpose, since they provide all the necessary tools to carry out this issue, as explained in detail in the relative documentation [4].

The applications developed for this thesis work will assume the mining floor to be flat, reason for which the robot *.urdf* files exploit 2D LiDAR sensors. This is because OpenRMF and OpenTCS, even though accept maps with more that one floor, still do not support floors with irregularities or with a slope that is not equal to zero. In reality, as explained in [5], the process of generating a map and self localization presents harder challenges due to this irregularities, that can be addressed using jointly 2D and 3D LiDAR together with existing 3D simultaneous localization and mapping 2D mapping methods.

Chapter 3

Methods and technology

The purpose of this chapter is to illustrate the tools, libraries and plugins exploited to build the application developed with this thesis to manage and control multiple robots belonging to different fleets.
It will cover:

- The concepts at the heart of ROS2.
- The working principle of the OpenRMF framework, describing its internal structure, the communication systems among its blocks and the tools it provides.
- The working principle of the OpenTCS framework, the internal structure and the tools it provides.
- The main features of the open source fleet management system *free_fleet*.

3.1 ROS2

ROS is an open-source framework developed by Open Robotics in cooperation with a big and open community of developers. The ROS software, provided with libraries, tools and GUIs, was intended, as the name suggests, for the development of robotic applications and robotics projects.
As the time passed since 2007, the year in which the framework was released, with the increasing demand for robotics applications and the need for more robust tools to rely on, Open Robotics released ROS2, a more powerful and high-performing version of ROS, providing many more features to be exploited for robotics projects. The following section deals with the core structure and concepts behind the ROS2 software, putting in evidence the communication among the building blocks of the framework and the various use cases.

A robotic system is a complex system, usually composed by many parts that interact among them, each one dedicated to the execution of a specific task or computation.
A generic wheeled robot for example, might need some motors to allow motion, a camera system, it may also rely on a vision system to process all the information coming from the camera images, a control system and other components, depending on the other tasks the robot was thought to accomplish.
The key aspect that leads to the choice of ROS/ROS2 for robot programming, is that it makes possible to treat the robotic system as an ensemble of multiple processes, all of them cooperating performing different tasks. This can be made possible through the execution of multiple nodes, where the node is the fundamental entity in the ROS environment.

3.1.1 Node

In ROS, the node is the core building block of the system's architecture. A ROS node is an executable that exploits the framework's functionalities to communicate and exchange information with other nodes.
Hence, nodes are processes that perform computations, such as controlling motors, reading sensors, managing user input, running algorithms and so on. Nodes are designed to be modular entities that focus on a specific job.
Nodes are language-independent, this means that it is possible to refer to various programming languages to write down the related executables, depending on the specific functionalities that a node has to provide. The most common choices are Python, for good readability, simplicity and rapid prototyping, or C++, for the development of applications for which the execution speed has primary importance and for tasks that require a high computational effort.
Other supported programming languages, even though way less common, are:

- Java, to integrate ROS with other Java-based systems.
- JavaScript, to develop applications that require ROS to interact with web applications.
- LISP, mainly used in the oldest versions of ROS.
- MATLAB, that supports a Robotic System Toolbox thanks to which it is possible to write scripts to implement ROS nodes.

One of the main differences between ROS2 and ROS is the absence of the *roscore* node. In ROS, the *roscore* node is the most important one and must be running before all the other processes, in order to allow further nodes to be executed.

This node indeed, prepares the needed environment for all the other nodes to register themselves and communicate. Since ROS2, this node is not necessary anymore, and it is possible to start the application by running directly the nodes that will execute the application-specific processes.

3.1.2 Communication

In ROS2, nodes can communicate through a DDS (Data Distribution Service) middleware, granting a flexible and non-centralized communication system.
The mechanism of communication provided by ROS2, introduced in the following subsections, are:

- Topics
- Services
- Actions

3.1.2.1 Topics

The first way nodes can communicate is through topics, that are named buses over which two or more nodes can exchange information.
The topic communication system is based on a publisher-subscriber structure among the nodes, or *pub-sub messaging*. In this scenario indeed, nodes can be classified as publisher nodes, if they are the ones sending messages, or subscriber nodes, if they are the one receiving or listening to the messages.
Specifically, nodes publish data by *advertising* a topic, where each topic is identified by a name and a type. The name and the type of a topic are used by the subscriber nodes to identify the topics they need to subscribe to. These are indeed two of the necessary parameters to be specified while writing down the code for the creation of a publisher/subscriber node, since it will search, among the active topics, the one that has the name and type specified in the code to establish the connection. Anyway, nodes can be at the same time both publishers ad subscribers, and this is what happens most of the times.
Furthermore, the pub-sub architecture allows more than just a dual communication type: a publisher indeed, can have one or multiple subscribers, allowing the same information to be exploited by multiple nodes to perform different computations.
The type of a topic is of crucial importance, first of all because it is necessary for the identification of the publisher-subscriber connection that has to be established, as mentioned before, but also for the syntax and the semantic aspect that a topic type carries. A topic type indeed specifies which are the types of data contained into the message, that could be, for instance, integers, floats or have a more complex

structure.
Even though ROS2 provides some already implemented topic types, such as the ones contained in the directory *std_msgs*, it is still possible to create new message types, characterized by a custom architecture and semantic.
The last thing about this kind of communication among nodes, is that it implements an *asyncronous* communication type: messages are indeed published at any time, with a specific publication frequency. This aspect, for instance, makes the pub-sub communication system perfect to manage the information coming from sensors. A schematic representation of the topic-based communication is given in Figure 3.1.

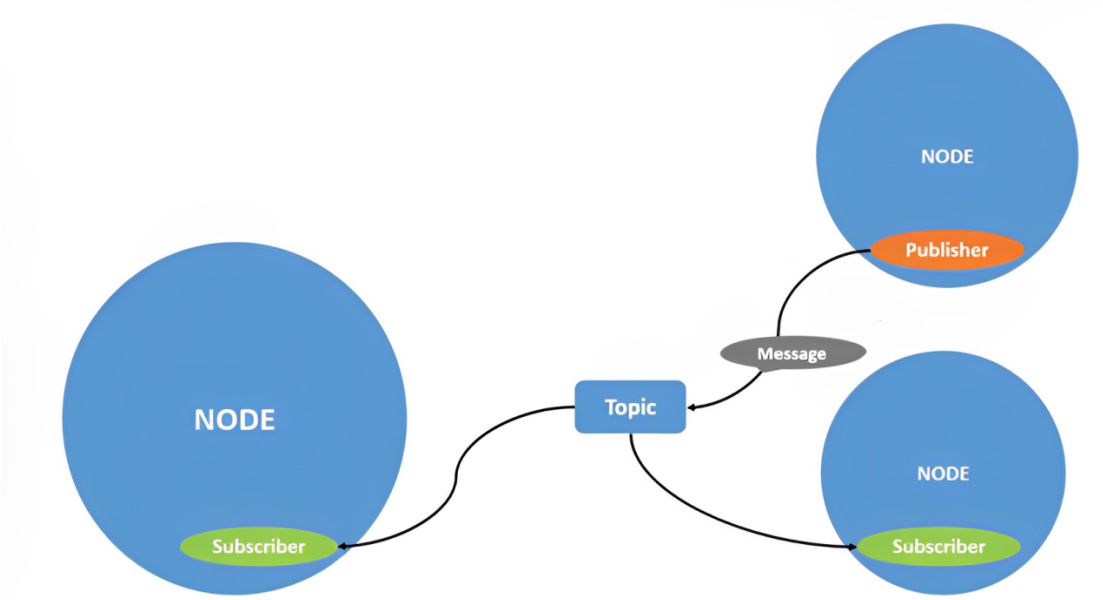

Figure 3.1: Diagram representation of the topic-based communication structure [6]

3.1.2.2 Services

While topics are suitable for streaming data and asyncronous communication, they offer a poor fit in case of on demand communication.
It might happen in fact, that a datum or a series of data is requested when a specific event takes place, or when a flag changes its value. The ROS service is another type of communication that implements this behaviour, called *request-reply communication*.
This mechanism is based on the presence of a server node, the one that offers the service, generally a computational process, and a client node, the one that asks for the service. The kind of information exchanged by the client and the

server is identified by the type of service: it is defined by two ROS messages, the *request*, specifying the input data types that the server node will exploit to perform computations, and the *response*, specifying the output data types that the server produces and that the client will receive as result of the service. The client-server communication is said to be an on demand communication type, since the server will perform its computation only after the client does a service call, differently from what happens in the publisher-subscriber communication type, where the data are streamed continuously.
As for the topics, ROS2 as well accounts for some already built-in service types; but even in this case, it is possible to declare more service types in order to allow to exchange data in a customized manner that better fits the requirements of the project circumstances.
A schematic representation of the service-based communication is given in Figure 3.2.

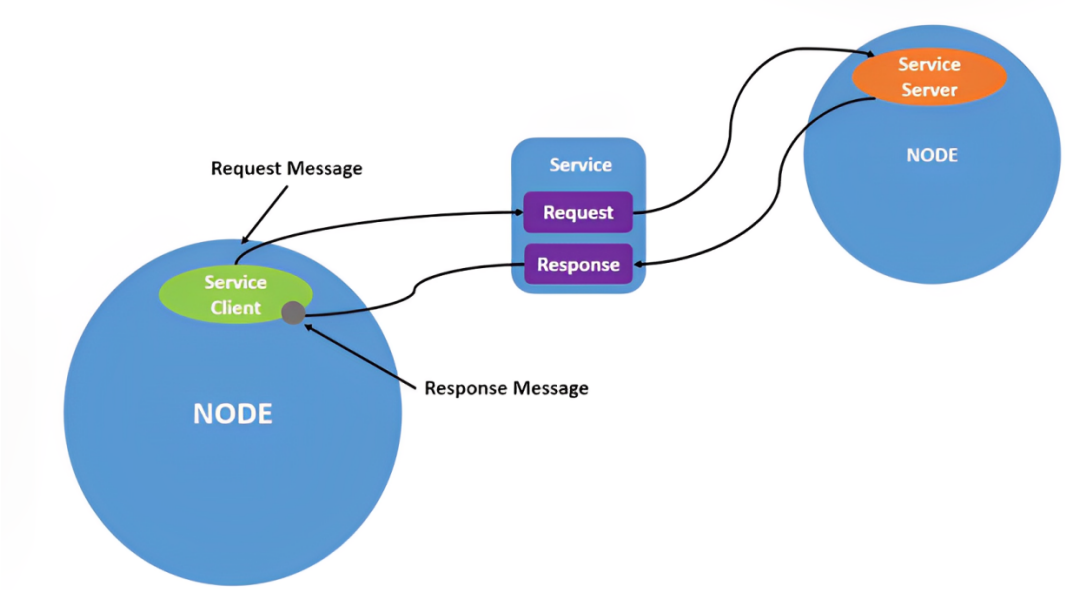

Figure 3.2: Diagram representation of the service-based communication structure [7]

3.1.2.3 Actions

Another communication type supported by the ROS2 framework is the action.
It consists in a more complex and structured mechanism to implement the communication, exploiting the advantages provided by both topics and services.
The action communication type could be compared to the client-service one, since

this case as well is characterized by the same hierarchy, having an action client node and an action server node. Anyway, the action client and server nodes present a higher complexity than a simple node, since the action-based communication is implemented through:

- A *Goal Service*, that requires a goal service client and a goal service server, respectively implemented by the action client node and the action server node.
- A *Result Service*, that requires a result service client and a result service server, respectively implemented by the action client node and the action server node.
- A *Feedback Topic*, that requires a subscriber and a publisher, respectively implemented by the action client node and the action server node.

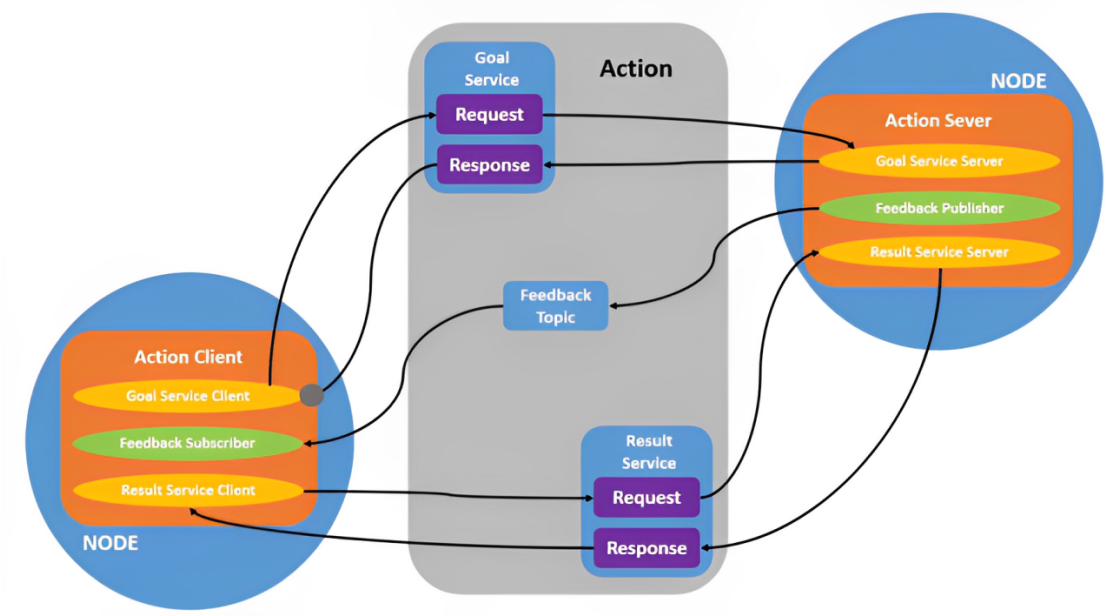

Figure 3.3: Diagram representation of the action-based communication structure [8]

As it could be inferred by the names, the action client sends the goal to the action server, that is in charge of providing a result. Meanwhile, a stream of feedbacks is sent back from the action server to the action client, in order to monitor the entire action process. Differently from the service indeed, through the action it is possible to implement a stoppable mechanism, maintaining the on demand communication type.
This type of communication can be used to implement long running tasks for which

it is needed to monitor the progress (for instance, a robot navigating to a specific location). A schematic representation of the action-based communication is given in Figure 3.3.

3.2 OpenRMF framework

OpenRMF is a set of extendable libraries and tools applicable to a variety of different use cases. The purpose of this framework is to facilitate the control of a multitude of robots beloging to different and heterogeneous fleets within the same environment.
OpenRMF is built on top of ROS2, thing that allows to take advantage of the topic-based and service-based communications and to organize the execution of concurrent processes through the execution of multiple nodes.
One of the biggest advantages of this OpenRMF consists in its modularity, being strictly organized in repositories for any area of interest: more specifically, each repository contains ROS2 packages, messages or tools.
According to [9], the repositories OpenRMF is composed of are:

- *rmf_api_msgs*, collection of *.json* message schemas which bridges the C++ and python components of RMF to the web interface.
- *rmf_battery*, consists of a single package that provides the necessary APIs to model a robot's battery life and its change in state-of-charge.
- *rmf_building_map_msgs*, provides ROS message and service types for communicating about building infrastructure.
- *rmf_internal_msgs*, the packages contained in this repository provide the internal messages of the core of RMF. Developers extending RMF or producing alternative implementations of existing components will need to use these messages. In general, they are not intended to be used by users of RMF.
- *rmf_ros2*, collection of packages that integrate the core algorithms and data structures of RMF into a ROS2-based distributed system.
- *rmf_simulation*, contains simulation plugins used in Open-RMF. It currently supports Gazebo Classic 11 and Gazebo Fortress.
- *rmf_task*, provides APIs and base classes for defining and managing tasks in RMF.
- *rmf_traffic*, implements the algorithms and data structures that are used for scheduling and negotiating mobile robot traffic between multiple agents.

- *rmf_traffic_editor*, a repository accounting for a GUI for annotating floorplans to create traffic patterns, Python-based tools to use and manipulate map files and Gazebo model thumbnails.

- *rmf_utils*, provides some low-level C++ programming utilities that are used across all Open RMF C++ packages. This package is a tool for the developers of Open RMF, and not necessarily intended for external developers.

- *rmf_visualization*, contains several packages that aid with visualizing various entities within RMF via RViz.

- *rmf_visualization_msgs*, messages for visualizing aspect of OpenRMF.

The following subsections of this chapter will be helpful for a general understanding on how to build a RMF-based application, delving into the structure and the components that such application should have.

3.2.1 Traffic deconfliction

One of the key advantages of using OpenRMF to manage a multi-fleet robotic system is its capability to adress the problem of traffic deconfliction.
Specifically, this issue can be handled through the combination of two strategies: conflict prevention and conflict resolution.
An efficient way to prevent traffic conflicts is to take route decisions based on the paths of the other robots: for this reason the core RMF is provided with a platform agnostic *Traffic Schedule Database* (or more simply traffic schedule), implemented through a set of classes condensed in the node *rmf_schedule_node* defined in the repository *rmf_traffic*.
The traffic schedule is a dynamic database whose contents change over time storing all the information relative to the robots' routes, priorities, eventual path abolitions and rescheduling. An important aspect to underline is that the database looks into the future, storing all the intended trajectories instead of the actually confirmed ones. All the stored informations can then be exploited to compute the best routes in order to avoid conflicts, if possible.
By the way, relying only on the *Traffic Schedule Database*, hoping it will be sufficient to manage this issue, is not the correct approach. Conflicts indeed can arise for a huge number of reasons, such as delays, dynamic objects moving over time, changes in task priorities due to emergencies and so on.
Luckily, RMF is provided with a *Negotiation Scheme* that comes to play whenever the traffic schedule detects an incoming conflict.
It is based on the exchange of different messages published over different topics according to the following procedure:

1. After the detection of a conflict between the schedule participants, the *rmf_schedule_node* sends a notification over the topic */rmf_internal_msgs/rmf_traffic_msgs/msg/NegotiationNotice*.

2. Each fleet adapter acknowledged of the issue will publish the optimal itineraries compatible with the others, for each one of the robots belonging to the respective fleet, over the topic */rmf_internal_msgs/rmf_traffic_msgs/msg/NegotiationProposal*.

3. After every fleet adapter has sent its proposal, the *rmf_schedule_node* evaluates the best one notifying the fleet adapters of the corrections.

4. If the conflict is solved, it is notified over the topic */rmf_internal_msgs/rmf_traffic_msgs/msg/NegotiationConclusion*; otherwise a newer attempt is requested through the topic */rmf_internal_msgs/rmf_traffic_msgs/msg/NegotiationRepeat*.

3.2.2 Fleet adapter

The fleet adapter is a key component developed to integrate various robotic fleets cooperating with the core of OpenRMF.
If considering the communication process when a task needs to be executed as orchestrated in levels, it is possible to identify three distinct layers:

Highest, here all the strategic decisions are taken, for example which fleet of robots is in charge of executing the said task (or set of tasks) and which eventual conflict needs to be prevented or resolved.
The *Traffic Schedule Database*, the *Task Dispatcher* (more later on this) and all the function and classes used to populate and manage its internal mechanisms belong to this layer.

Middle, at this layer takes place the fleet adapter, one for every fleet of robots of the system. It acts as a communication bridge, being in charge of dispatching the tasks to the robots belonging to the fleet, translating the said tasks into actions to be executed such as reaching a specific location, sending path goals, lifting objects, docking, charging and so on.
Furthermore, it is dedicated to status reporting in a format that RMF can understand, including battery level, robots' locations, task completion status.

Lowest, it is the layer where all the most robot-specific instructions are taken, such as spinning the motors, reading statuses through sensors, obstacle avoidance. The fleet management system of a fleet of robots belongs to this layer.

Figure 3.4 depicts this hierarchy in a diagram in the case of four fleets being coordinated by one OpenRMF system.

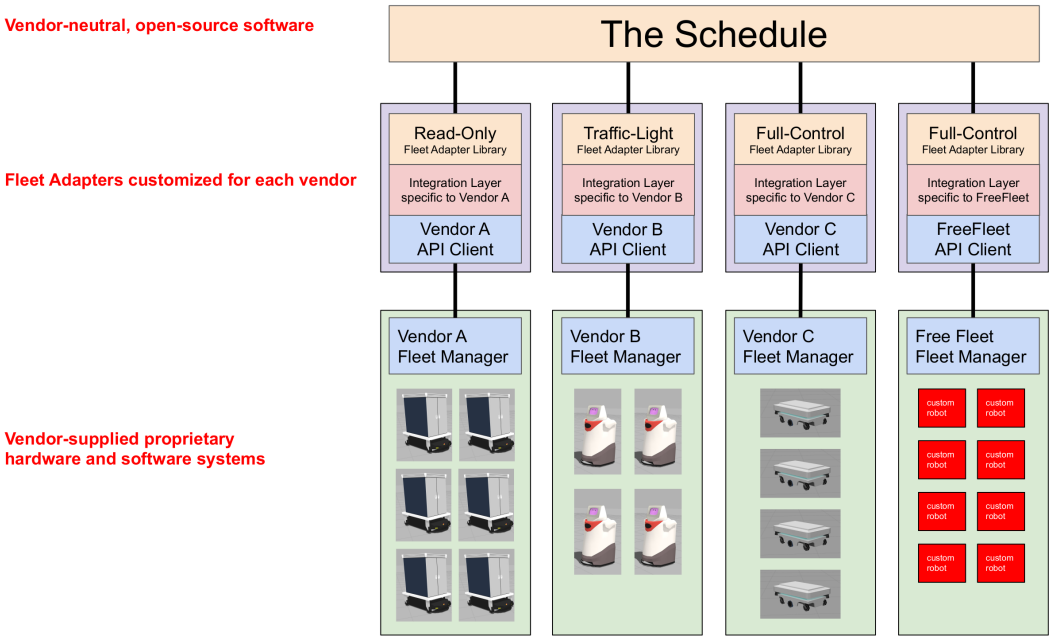

Figure 3.4: Diagram representation of the communication structure [10]

The biggest advantage of this communication architecture is that it is possible to operate with many diverse fleets, having different features and eventually even different communication protocols among its robots, since the fleet adapter can be configured to translate all the fleet-specific information in a general and understandable structure that can be correctly interpreted by the system. While indeed OpenRMF is a vendor-neutral open source software, the robot fleets in general are already provided with a vendor-specific fleet management system that is already configured by the vendor company and that needs to be correctly adapted to be integrated with the rmf framework.
As mentioned before, each robot fleet is required to have a fleet adapter to link its fleet-specific API to the interfaces of the scheduling and negotiation system.
By the way, according to the amount of information that the fleet adapter can access and communicate to RMF, that depends on how the fleet management systems were designed, it is possible to distinguish 4 classes of fleet adapters:

Full Control, RMF is provided with live status updates and full control over the paths that each individual mobile robot uses when navigating through the

environment. This control level provides the highest overall efficiency and compliance with RMF, which allows RMF to minimize stoppages and deal with unexpected scenarios gracefully. [10]

Traffic Light, RMF is given the status as well as pause/resume control over each mobile robot, which is useful for deconflicting traffic schedules especially when sharing resources like corridors, lifts and doors. [10]

Read Only, RMF is not given any control over the mobile robots but is provided with regular status updates. This will allow other mobile robot fleets with higher control levels to avoid conflicts with this fleet. Note that any shared space is allowed to have a maximum of just one "Read Only" fleet in operation. Having none is ideal. [10]

No Interface, without any interface to the fleet, other fleets cannot coordinate with it through RMF, and will likely result in deadlocks when sharing the same navigable environment or resource. This level will not function with an RMF-enabled environment (not compatible). [10]

A more detailed description of what kind of control a fleet adapter allows according to its type is described in Table 3.1

Fleet adapter type	Robot/fleet-manager API feature set
Full Control	-Read the current location of the robot [x, y, yaw] -Request robot to move to [x, y, yaw] coordinate -Pause a robot while it is navigating to [x, y, yaw] coordinate -Resume a paused robot -Get route/path taken by robot to destination -Read battery status of the robot -Infer when robot is done navigating to [x, y, yaw] coordinate -Send robot to docking/charging station -Switch on board map and re-localize robot. -Start a process (such as clean zone) -Pause/resume/stop process -Infer when process is complete (specific to use case)
Traffic Light	-Read the current location of the robot [x, y, yaw] -Pause a robot while it is navigating to [x, y, yaw] coordinate -Resume a paused robot -Read battery status of the robot -Send robot to docking/charging station -Start a process (such as clean zone) -Pause/resume/stop process -Infer when process is complete (specific to use case)
Read Only	-Read the current location of the robot [x, y, yaw] -Pause a robot while it is navigating to [x, y, yaw] coordinate -Read battery status of the robot -Infer when process is complete (specific to use case)
No Interface	-The fleet adapter has no access to the robot status

Table 3.1: Informations a fleet adapter can access/transmit according to its type [10]

3.2.3 Task dispatching

The process of assigning a specific task or a set of tasks to a fleet of robot is called task dispatching, and it is assigned to the *Task Dispatcher*, a RMF-based entity that is implemented through the *rmf_dispatcher_node*.
It selects the best fleet that can execute the task among all, following a precise procedure based on the interchange of several messages over topics.
It is graphically represented in the diagrams in Figure 3.5, Figure 3.6 and Figure 3.7, and it can be described in five simple steps:

1. As soon as the *Task Dispatcher* receives a task request, that comes directly from the user (either through the terminal or a dashboard), it sends a bid notice message over the topic
   /rmf_internal_msgs/rmf_task_msgs/msg/BidNotice to all the fleet adapters of the system since they subscribe to the same topic. This is message is necessary to notify the fleet adapters that a new task request has arrived.

2. The fleet adapters will publish their proposal back over the topic
   rmf_internal_msgs/rmf_task_msgs/msg/BidProposal : this message contains a cost associated to the execution of the task, that depends on the length of the path that the robot should cover, its battery level, the duration of the execution of an eventual process nested within the task, and so on.
   From now on it will be referred to this cost with the name of *Bid Cost*.

3. The *rmf_dispatcher_node* behaves as an external judge and evaluates all the proposal and assigns the execution of said task to the fleet adapter that yields the best proposal, if it is enabled to perform this type of task (more later on this). The way the dispatcher select the fleet adapter, that is the way it evaluates the *Bid Cost* is configurable (i.e. fastest to finish, shortest path, etc). Hence, the *Task Dispatcher* communicates the winner publishing a message over the topic
   /rmf_internal_msgs/rmf_task_msgs/msg/DispatchRequest, specifying the name of the fleet the task is awarded to.

4. The fleet adapter, that in turn is in charge of dispatching the task among the robots the fleet is composed of, sends back a confirmation message over the topic
   /rmf_internal_msgs/rmf_task_msgs/msg/DipatchAck to certify if the task execution request is correctly received.
   If there are free robots in the fleet the task is executed immediately, otherwise it is added to a queue and executed according to a FIFO protocol.

5. Finally, during all the time between the task submission and its completion, the winning fleet adapter sends a message over the topic

/rmf_internal_msgs/rmf_task_msgs/msg/TaskSummary, communicating the real time status of the task, specifying fleet and robot that will execute it, submission time, start and end time of the execution, if the task is queued, active, completed, failed, cancelled, etc.

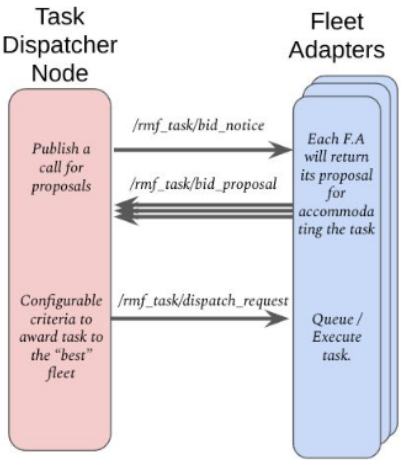

Figure 3.5: Bidding process before the task assignment [11]

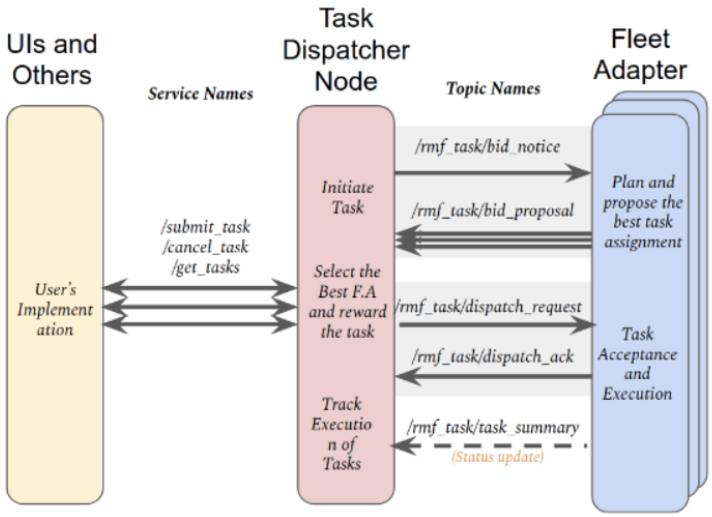

Figure 3.6: Bidding process after the task assignment [11]

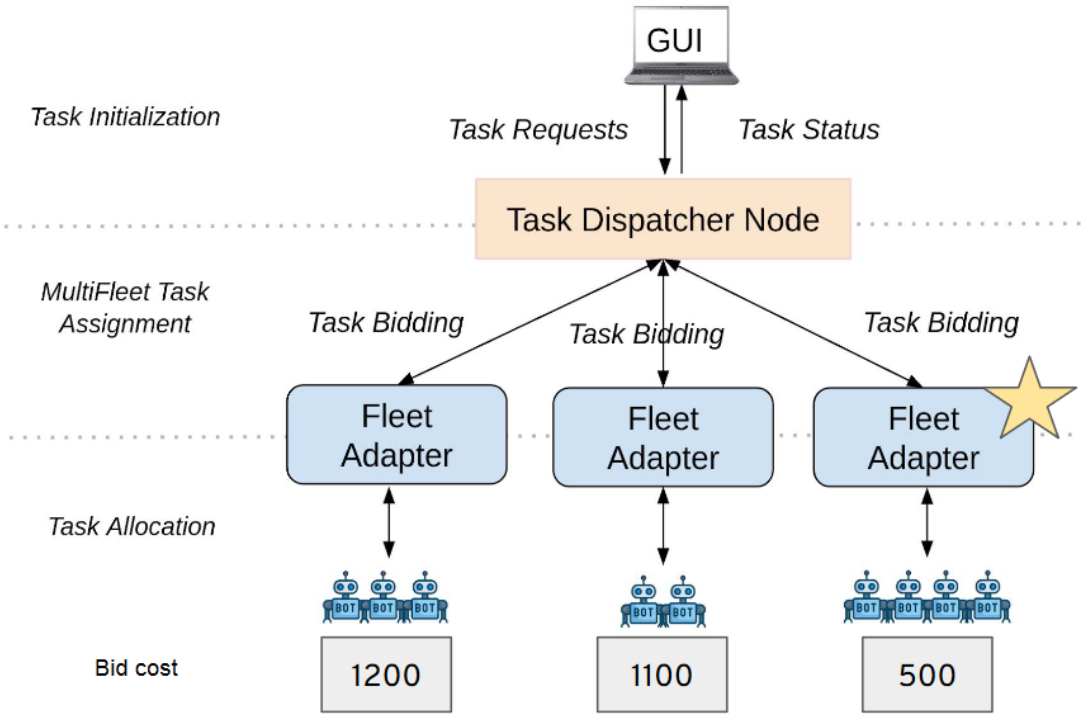

Figure 3.7: Selection of the winning fleet in the bidding process [11]

The task features and characteristics are specified in the *task description*, that is serializable data structure (in *json* format) that can be interpreted by the *Task Dispatcher*.
The tasks can be simple, when they consist in one activity only, or composed, when they come out of the sequence of more activities (such as cleaning, going to a place and then performing a delivery). If the task is composed, it has to be specified in the json format of the task description with the key word *compose*, and with the description of each activities it is made of.
Two examples of this json format are shown in Figure 3.8 and Figure 3.9.
Each activity is defined by a *category* that serves as a label element: each fleet adapter indeed must be configured to perform or not perform a specific task by specifying the *categories* that it can support. For example, if a fleet is not configured to support a task, because it does not support all the *categories* it is composed of, it will be excluded by the bidding process and the *Task Dispatcher* will not wait for any proposal coming from the respective fleet adapter.

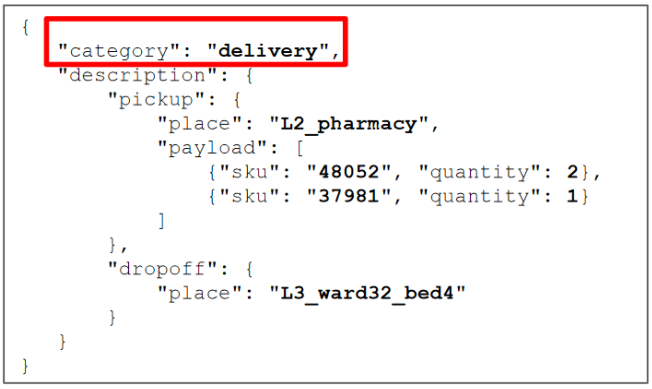

Figure 3.8: *json* format of a simple task that performs a delivery [11]

Figure 3.9: *json* format of a composed task that consists of a pickup task, followed by a dropoff task, ending with a greeting task [11]

3.2.4 Traffic-Editor

One of the main challenges in the management of a multi-fleet robotic system consists in the definition of the characteristics of the map of the environment in which the robot operate.
There might be indeed sections of the environment where the robots are permitted and others where they are not, or simply areas where just some selected robots are allowed instead of all of them.
It might also be useful to specify a route direction (the routes can indeed be traversed unidirectionally or bidirectionally) and the physical orientation a robot has to take while covering a path.
Furthermore, in order to facilitate the execution of a task in a specific point of the map it is useful to set a property for that point: the location of the charger for example needs to be specified by setting a property, so to distinguish it from the other points on the map.
Lastly, if the robots need to interact with the surrounding environment, such as doors, it is still necessary to define a procedure to make the interaction possible.
To deal with all these issues, the implementation of OpenRMF is provided with the repository *rmf_traffic_editor* that counts for several tools and a GUI. *Traffic Editor* indeed allows the generation of *navigation graphs*, that are files containing all the information related to the the paths the robots have to follow.
The output of the GUI is a file with the extension *.building.yaml*, but still does not represent the navigation graph. Thanks to the tool *building_map_generator*, contained in the package *rmf_traffic_editor/rmf_building_map_tools*, according to the input parameters it is fed with, it is possible to convert the *.building.yaml* file into a *.world* file, suitable for Gazebo, or into a *.yaml* file, cointaining the navigation graph. The following subsections explain more in detail the functionalities of *Traffic Editor* for the generation of navigation graphs, these are:

- Level creation
- Waypoints definition
- Traffic lanes definition

3.2.4.1 Level creation

The first step for the generation of a traffic-editor map is the creation of a level. The definition of the measurement units can be done by setting reference coordinates of by exploiting a reference image (in *.png* format). The second case is very helpful to draw the lanes a robot can traverse directly on the image of the map of the environment, allowing a more precise and user-friendly job. Additionally, if a reference image is used, it is also necessary specify the pixel-meters ratio, that

expresses how many meters (or portions of a meter) represents a pixel in the *.building.yaml* output file. An interesting aspect of this tool, is that it is possible to generate a multilevel environment as well, consisting in an environment with more than a floor, eventually linked through stairs or lifts.

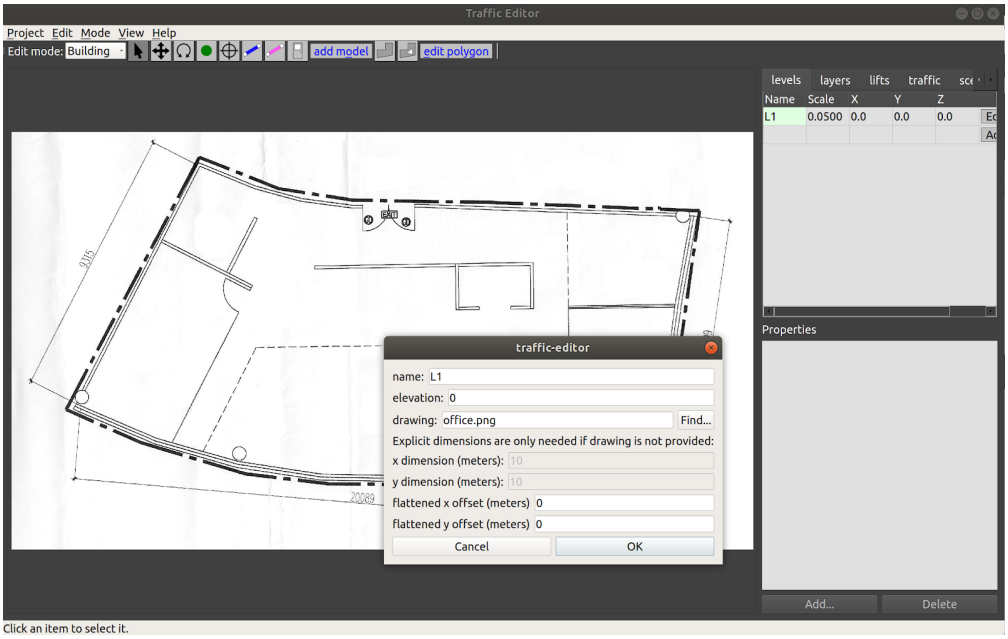

Figure 3.10: Creation of a level with reference image [12]

3.2.4.2 Waypoints definition

A waypoint is a location, defined by its x and y coordinates and indentified by a unique name, in which a robot can park and execute actions.
In order to specify the behaviour that the robot can assume in that specific waypoint, *Traffic Editor* allows to specify its properties, that according to the documentation [12] can be:

- *is_holding_point*: if true and if the waypoint is part of a traffic lane, the *rmf_fleet_adapter* will treat it as a holding point during path planning, allowing the robot to wait at this waypoint for an indefinite period of time.
- *is_parking_spot*: if true this waypoint is treated as a parking spot. Parking spots are used when an emergency alarm goes off, and the robot is required to park itself.
- *is_passthrough_point*: if true the system is informed that the robot should not stop at this waypoint.

- *is_charger*: if true and if the waypoint is part of a traffic lane, the *rmf_fleet_adapter* will treat this as a charging station.
- *is_cleaning_zone*: indicate if current waypoint is a cleaning zone, specifically for *Clean* Task.
- *dock_name*: if specified and if the waypoint is part of a traffic lane, the *rmf_fleet_adapter* will issue an *rmf_fleet_msgs::ModeRequest* message with *MODE_DOCKING* and *task_it* equal to the specified name to the robot as it approaches this location. This is used when the robot is executing their custom docking sequence (or custom travel path).
- *spawn_robot_type*: indicates the name of the robot model to spawn at this waypoint in simulation. The value must match the model's folder name in the assets repository.
- *spawn_robot_name*: a unique identifier for the robot spawned at this waypoint. The *rmf_fleet_msgs::ModeRequest* message published by this robot will have name field equal to this value.
- *pickup_dispenser*: name of the dispenser workcell for *Delivery* Task, typically is the name of the model.
- *dropoff_ingestor*: name of the ingestor workcell for *Delivery* Task, typically is the name of the model.

3.2.4.3 Traffic lanes

Once all the waypoints are defined, it is possible to join them by drawing a *Traffic Lane*. A multitude of lanes representing the path that a robot can cover is called *graph*, and each one of them is identified by a unique name. Said that, it is possible that different fleets of robots are allowed to cover different paths (if each one of them is associated to a different graph) or the same one (if they share a graph). As for the waypoints, properties can be associated to traffic lanes as well. Specifically a lane can be:

- unidirectional, if it can be traversed in a single direction, that has to be specified.
- bidirectional, if it can be traversed in both directions.
- forward oriented, if the robot can only be in this traffic lane moving forwards.
- backward oriented, if the robot can only be in this traffic lane moving backwards.

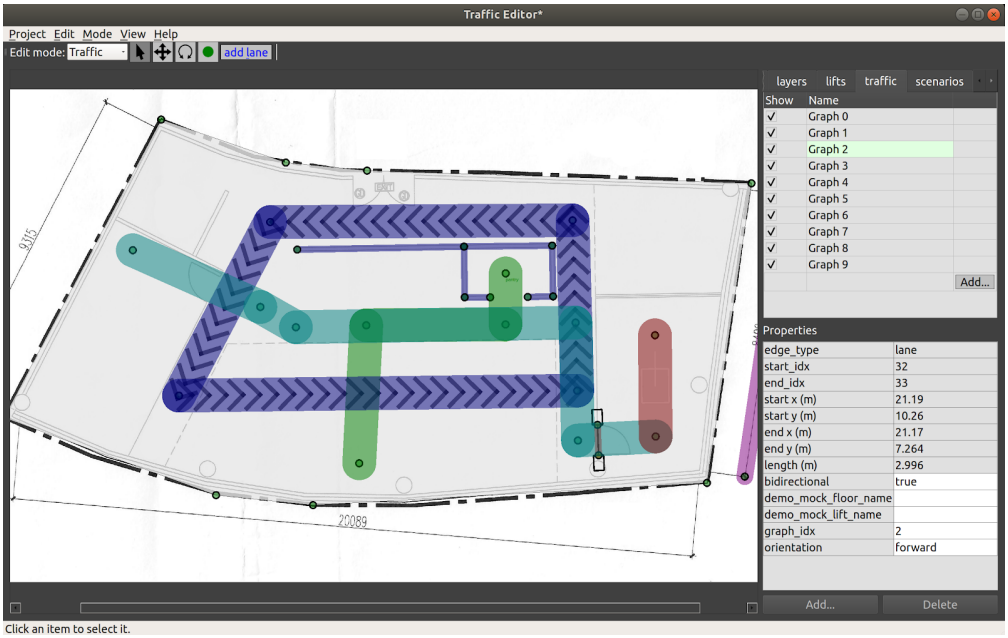

Figure 3.11: Traffic lanes [12]

3.3 Fleet management system

As explained in the previous sections, the OpenRMF framework allows the communication among multiple fleets of robots, and manages the task allocation through the bidding process previously mentioned, exploiting the fleet adapter structure. Anyway, the fleet adapter does not submit the commands needed to perform a task directly to the robots, there is instead an other intermediate layer that needs to be considered, that is the fleet management system. The fleet management system acts as an interface between OpenRMF and the fleet of robots itself: it receives the commands from the the fleet adapter and translates them in a form that can be understood by the robots. Of course, every fleet needs to be provided with a fleet management system to function properly. Figure 3.4 depicts how this interface fits in the communication scheme.
Generally, a robotic fleet comes with its own fleet management system provided by the vendor company, but by all means this does not apply to every case and some times it is necessary to develop a custom fleet management system.
In the event that the user wishes to integrate a standalone mobile robot which does not come with its own fleet management system, the open source fleet management system *free_fleet* could be used.

3.3.1 Free fleet

The *free_fleet* system is based on the client-server structure, where:

- The client part has to be run on every robot of the fleet with their navigation software.
- All the robots of the fleet refer to one unique server (one server for each fleet).

The structure of the *free_fleet* system and how it is interconnected to the OpenRMF framework is represented in Figure 3.12.
The client's implementation is intended to permit the interaction between robots that do not necessarily share the same implementative features. For instance, exploiting the *free_fleet* packages it is possible to have in the same fleet robots that implement the communication through ROS topics and services and other robots that use ROS2 ones instead, or to have robots that exploit different navigation stacks or onboard communication protocols, and so on.
In other words, *free_fleet* allows implementative diversity within the same robot fleet.
The server operates on a central computer and receives the status updates coming from each client, in such way they can be displayed through a GUI, or passed upstream to OpenRMF. The server is also in charge of transmitting the commands

to be executed, coming from the user through the UI, or coming from OpenRMF, to the clients. Each server can operate with more than a client at a time, behaving as a fleet management system. The server indeed can be exploited to work with more complex systems, such as OpenRMF, adapting each robot's interface and API's to OpenRMF's interfaces, or it can be used as its own fleet management system.
Concerning the communication system between the client part and the server part, it is implemented exploiting the *CycloneDDS* [13] communication protocol , reason why there is no need to be concered if the robots or the central computer runs different version of ROS.

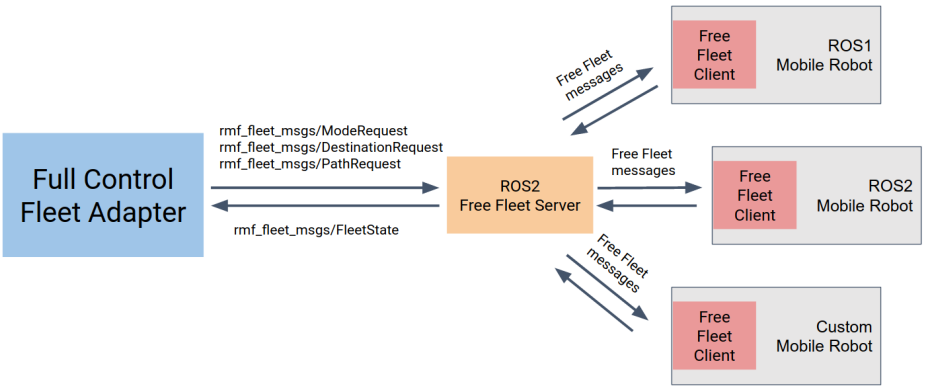

Figure 3.12: Diagram representation of the free fleet structure [14]

3.4 OpenTCS

The OpenTCS software is a fleet management system specifically designed for automatic machines.
It was mainly though for the control of AGVs (Automated Guided Vehicles), to facilitate their coordination while executing tasks, for instance, in a production plant. Anyways, its functionalities can be exploited to control other other automatic vehicles, such as mobile robots or quadrocopters.
OpenTCS allows to coordinate these machines independently of their individual features, such as the navigation stack or handling systems. OpenTCS can manage vehicles having different characteristics and performing different tasks at the same time. This is done via the integration of the machines with the core system through pluggable drivers, comparable to device drivers in operating systems.
This section will explain briefly the core implementation of OpenTCS, the tools and GUI it provides, and the vehicle driver OpenTCS-NeNa that allows to exploit the functionalities of OpenTCS in the ROS2 environment.

3.4.1 System overview

OpenTCS is a composite system composed by different elements running as separate processes and operating together according to a client-server structure.
The aforementioned elements and their purposes are

Server : The unique server process of this hierarchy is the Kernel, running vehicle-independent strategies and drivers for controlled vehicles [15]

Clients :

- Plant overview, provided with a GUI, for modelling and visualizing the plant model. [15]
- Kernel control center, for controlling and monitoring the kernel and for providing a detailed view of vehicles and their associated drivers. [15]
- Arbitrary clients for communicating with other systems, for instance for process control or warehouse management. [15]

Drivers : For managing the communication channels between the vehicles and the Kernel. They are vehicle-specific and can have different implementative features.

Figure 3.13 shows the architectural structure of the building blocks of OpenTCS.

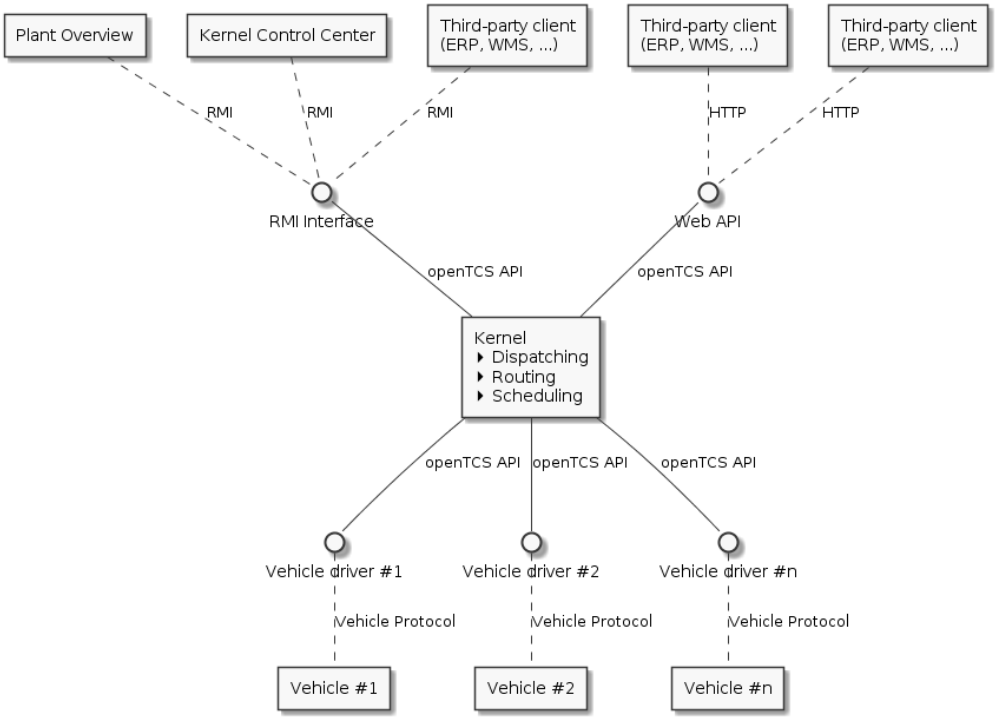

Figure 3.13: diagram representation of the client-server architecture of OpenTCS [15]

3.4.1.1 Server

The Kernel is the heart of the OpenTCS system, being in charge of all the main computational processes and determines the appropriate actions for the robots in various scenarios. More specifically, the Kernel could be better understood if considered in its composing entities, the Dispatcher, the Router and the Scheduler:

Dispatcher: it assigns the execution of a task or a transport order to the vehicles present in the environment, selecting the best one in terms of resources consumption. In order to make this choice, it takes into account not only the length of the paths of each vehicle, but also their speed and battery level. Finally, it manages the behaviour of the other vehicles that are not executing any tasks, that are, for instance, parked or in idle state.

Router: it is in charge of computing the optimal paths of each vehicle for the execution of each task, that will be exploited by the Dispatcher for task assignment.

Scheduler: it is responsible of managing the all the traffic issues that can arise

in case of multiple tasks being executed at the same time, for instance by ordering to some vehicles to park or stop in a specific waypoint.

Clearly, the OpenTCS distribution includes default settings for each one of these elements of the Kernel, but they can be modified by the developer to adjust it to the specific requirements of the environment.

3.4.1.2 Clients

The plant overview client included in the openTCS distribution can be activated through the terminal, and according to the mode in which it is run, it can be used for different purposes. More precisely, two are the modes of the plant overview client, the *Modelling Mode* and the *Operating Mode*.
When in *Modelling Mode*, it can be used for the editing of plant models, which can then be used as environment on which the Kernel can perform all the scheduling, dispatching and routing computations. More specifically, the plant overview is provided with a GUI through which it is possible to define all the features of the simulated world, for example waypoints, allowed paths and directions, locations with specific properties, including the features of the vehicles themselves.
Once all the modifications to the environment have been set, through this GUI it is possible to persist the model in the Kernel, so that the Kernel knows exactly which model has to be considered for all the operations.
On the other hand, when in *Operating Mode*, the plant overview client is utilized to show the overall status of the transportation system and any ongoing transport process, as well as to create new transport orders interactively.
Said that, the user can exploit the *Operating Mode* of the plant overview client to allocate manually new tasks when needed. Then the Kernel will execute all the computational processes to guarantee the correct and compliant execution of said tasks.
Another process among the ones already mentioned is the Kernel control center client. It is designed for monitoring and controlling the Kernel, providing functionalities such as:

- Assigning vehicle drivers to vehicles, ensuring that each vehicle is operated and managed correctly inside the system.
- Controlling the vehicle communication, making sure that they can receive correctly the commands and send back their status updates.
- Monitoring vehicle state information, displaying all the real time information relative to the vehicles, such as status, velocity, location, and so on, to efficiently manage the fleet.

As the plant overview client, the Kernel control center client as well can be activated by the terminal, and it is provided with a GUI to facilitate the visualization of the status of the vehicles and tasks.
Morover, other third-party clients can be included to cooperate with the core system and to exploit more functionalities that are not included in the basic ones provided by OpenTCS.
In the case of Java clients, the Kernel provides an RMI (Remote Method Invocation) interface to consent the communication, the same exploited by the Kernel control center and plant overview clients, but in addition it provides web APIs as well for creating and withdrawing transport orders and retrieving transport order status updates.

3.4.1.3 Drivers

The driver framework that is part of the openTCS Kernel manages communication channels and associates vehicle drivers with vehicles. A vehicle driver is an adapter between Kernel and vehicle and translates each vehicle-specific communication protocol to the kernel's internal communication schemes and vice versa. Furthermore, a driver may offer low-level functionality to the user via the kernel control center client, for instance, manually sending telegrams (specific messages or commands) to the associated vehicle. By using suitable vehicle drivers, vehicles of different types can be managed simultaneously by a single openTCS instance [15].
More specifically, the driver exploited to adapt the OpenTCS core system to the simulated vehicles is the OpenTCS-NeNa driver.
The OpenTCS-NeNa software acts as a vehicle driver that implements the interface between ROS2 and the OpenTCS system.
This allows to take advantage of all the ROS2 functionalities that many AGV share, such as support for sensors, cameras and SLAM (Simultaneous Localization and Mapping).
The features implemented through the OpenTCS-NeNa driver, according to the documentation [16], are the following:

- Fully supported OpenTCS Operations.
- Fully supported OpenTCS Transport Orders.
- Properly handle ROS2 AGV navigation failures, such an unreachable destination.
- Live position tracking of a ROS2 AGV for showing in the OpenTCS Plant Overview.

- Live orientation of the ROS2 AGV for showing in the OpenTCS Plant Overview.
- Configurable ROS2 namespaces, which allows usage for multiple ROS2 AGVs simultaneously.
- Configurable ROS2 domain IDs.
- Dispatch the ROS2 AGV to a user-defined coordinate.
- Dispatch the ROS2 AGV to an OpenTCS Plant Model point.
- Set the initial position of a ROS2 AGV.
- Show a continuously updated ROS2 navigation status in the control center panel.
- Show the connection status in the control center panel.

3.4.2 Plant model elements

A plant model in OpenTCS can be described by an ensemble of elements specifying the properties of the environment, the locations where it is allowed to execute tasks, the vehicles themselves, and if there are any areas with particular features.
These elements can be either defined manually into an *.xml* file, or added through the plant overview client GUI, that automatically produces an *.xml* output that can be correctly interpreted by the Kernel. Certainly, to exploit this functionality, the plant overview client has to be in *Modelling Mode*.
A plant is then indentified by the combination of:

- Points
- Paths
- Locations
- Vehicles

A representation of these elements in a plant model using the GUI provided by the plant overview is shown in the example of Figure 3.14.

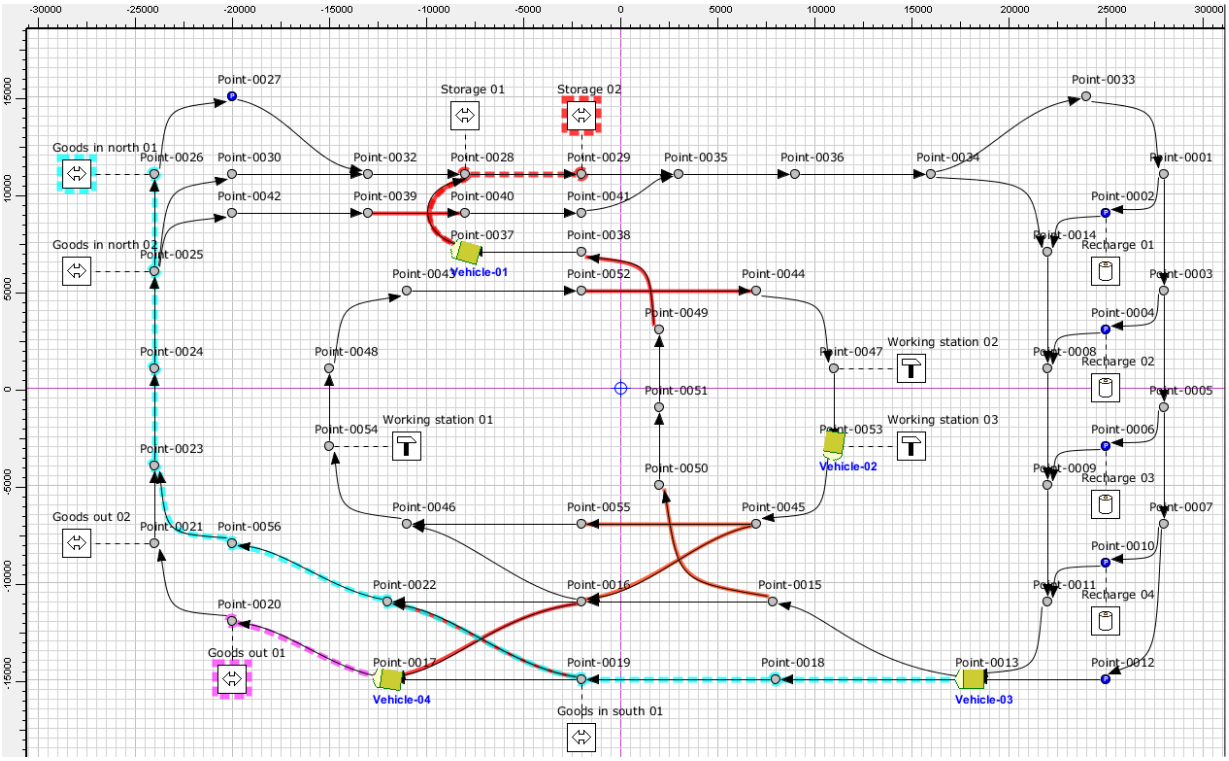

Figure 3.14: example of plant model in plant overview client GUI [15]

3.4.2.1 Points

In OpenTCS points represent the positions that the discrete vehicles in the plant can occupy. In plant operation mode, vehicles are generated in a specific point, and can follow a path by moving from one point to another one. According to the behaviour each vehicle should have when approaching a point, it is possible to define the following properties:

Halt points: indicate points at which a vehicle could wait temporarily while processing an order, for instance, to perform an operation. The vehicle then is expected to report the position when it arrives at said point.
Halt position is the default type for points when modelling with the plant overview client.

Reporting points: indicate points at which a vehicle is expected only to report its position, while halting or even parking at such a point is not permitted. Therefore a route that only consists of reporting points will be unroutable because the machine is not able to halt at any position, and therefore a task can be executed only if exists a path that contains at least a halt point.

Parking points: indicate points where a vehicle may stop for a long time, when it is not executing tasks, entering in park mode. As for the halt point, the vehicle is expected to report its position when it arrives at such a point.

Morover, points in OpenTCS carry the information related to the position, that is their x,y and z coordinates, and it is possible to specify the vehicle orientation angle, that represents the orientation said vehicle should assume when occupying the point.
Specifically, talking about coordinates, it is important to distinguish between *model coordinates* and *layout coordinates*.
While the *model coordinates* represent the actual coordinates of the plant and that the Kernel uses to perform its computations, the *layout coordinates* only serve to represent the points graphically in the GUI. Said that, the two coordinates type do not need to be exactly the same, thing that allow, for instance, to represent very big or very small plants on a different scale, or simply to visually enlarge or shorten a specific path with respect to the others.

3.4.2.2 Paths

A path is an interconnection between two points that can be followed by the vehicles to move from one position to another.
One of the most important attributes of a path is its length, that is computed considering the difference between two model coordinates, and it is used by the Kernel to compute the moves of the vehicles. The length indeed imposes a weight on a path, so that this can be taken into account to find the less resource-consuming routes. Furthermore, it is possible to specify a maximum velocity and a minimum velocity for each path, or set a *locked flag*: it is a flag that, when set, specifies the router that this path should not be considered when computing the routes for the vehicles. This is an useful functionality especially in case of emergencies, to simulate, for instance, the closure of a path for safety reasons.
Lastly, since every path is implicitly unidirectional, because it is not possible to specify a direction attribute, in order to generate a bidirectional path in OpenTCS plant overview client, it is necessary to overlap two opposite paths, the first one having the starting and finish points the opposite way as the second one.

3.4.2.3 Locations

Locations are marker for points at which vehicles may execute special operations, such as load or unload an object, clean the area, charge the battery level and so on. The kind of operation that can be executed at a specific location is defined by the

location type, defining all the task sequences that need to be followed in order to complete said operation. This allows variability in the definition of tasks, since it is possible to define composite tasks made by multiple operation ordered in a sequence. Every location is connected directly to a point through a *location link*: to be of any use for vehicles in the plant model, every location has to be linked to at least one point.

3.4.2.4 Vehicles

Vehicles are abstract models used to simulate real machines in order to allow the communication with them and to visualize their positions and other statistics. As specified in [15], a vehicle comes with the following attributes:

- A *critical energy level*, which is the threshold below which the vehicle's energy level is considered critical. This value may be used at plant operation time to decide when it is crucial to recharge a vehicle's energy storage.
- A *good energy level*, which is the threshold above which the vehicle's energy level is considered good. This value may be used at plant operation time to decide when it is unnecessary to recharge a vehicle's energy storage.
- A *fully recharged energy level*, which is the threshold above which the vehicle is considered being fully recharged. This value may be used at plant operation time to decide when a vehicle should stop charging.
- A *sufficiently recharged energy level*, which is the threshold above which the vehicle is considered sufficiently recharged. This value may be used at plant operation time to decide when a vehicle may stop charging.
- A *maximum velocity* and *maximum reverse velocity*. Depending on the router configuration, it may be used for computing routing costs/finding an optimal route to a destination point.
- An *integration level*, indicating how far the vehicle is currently allowed to be integrated into the system. According to its *integration level*, a vehicle can be:
  - ignored, if the vehicle and its reported position will be ignored completely, thus the vehicle will not be displayed in the plant overview. The vehicle is not available for transport orders.
  - noticed, if the vehicle will be displayed at its reported position in the plant overview, but no resources will be allocated in the system for that position. The vehicle is not available for transport orders.

  - respected, if the resources for the vehicle's reported position will be allocated. The vehicle is not available for transport orders.
  - utilized, if the vehicle is available for transport orders and will be utilized by the openTCS.

- A set of *allowed transport order types*, which are strings used for filtering transport orders (by their type) that are allowed to be assigned to the vehicle.

- A *route color*, which is the color used for visualizing the route the vehicle is taking to its destination.

3.4.3 Plant operation elements

Once the environment, inclunding the machines that will operate within it, is all set, the system is ready for the execution of eventual tasks.
First of all, it is necessary to define the tasks the machines will have to perform in a way that is compliant with the OpenTCS fleet management system. OpenTCS indeed offers two different way to define a task or a sequence of tasks, they are the *Transport Order* and the *Order Sequence*.

3.4.3.1 Transport Order

The first template for task execution offered by OpenTCS is the Transport Order. It consists in a sequence of actions and movements to be performed by a specific vehicle. With the allocation of a transport order, it is possible to set the following properties, as in [15]:

- A sequence of *destinations* that the processing vehicle must process (in their given order). Each destination consists of a location that the vehicle must travel to and an operation that it must perform there.

- An optional *deadline*, indicating when the transport order is supposed to have been processed.

- An optional *type*, which is a string used for filtering vehicles that may be assigned to the transport order. A vehicle may only be assigned to a transport order if the order's type is in the vehicle's set of allowed order types. (Examples for potentially useful types are "Delivery" and "Cleaning").

- An optional *intended vehicle*, telling the dispatcher to assign the transport order to the specified vehicle instead of selecting one automatically.

- An optional set of *dependencies*, as references to other transport orders that need to be processed before the transport order. Dependencies are transitive, meaning that if order A depends on order B and order B depends on order C, C must be processed first, then B, then A. As a result, dependencies are a means to impose an order on sets of transport orders. (They do not, however, implicitly require all the transport orders to be processed by the same vehicle. This can optionally be achieved by also setting the intended vehicle attribute of the transport orders).

An example of dependencies among different transport orders is shown in Figure 3.15.

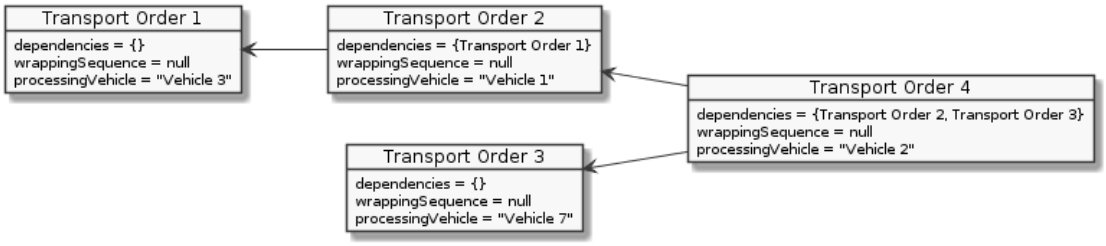

Figure 3.15: Diagram representation of dependencies among transport orders [15]

3.4.3.2 Order Sequence

The second way of defining a task proposed by OpenTCS is through an Order Sequence. An Order Sequence is a series of tasks that must be completed following a specific order. Specifically, it defines a sequence of Transport Orders, each one with their dependencies.
This way of defining tasks is very useful when a complex job needs to be broken down into several operations that cannot be expressed through a single Transport Order, and when these operations need to be executed by the same vehicle.
As the Transport Order, the Order Sequence carries some attributes as well, that are as in [15]:

- A sequence of transport orders, which may be extended as long the *complete flag* is not set yet.

- A *complete flag*, indicating that no further transport orders will be added to the sequence. This cannot be reset.

- A *failure fatal flag*, indicating that, if one transport order in the sequence fails, all orders following it should immediately be considered as failed, too.

- A *finished flag*, indicating that the order sequence has been processed (and the vehicle is not bound to it, anymore). An order sequence can only be marked as finished if it has been marked as complete before.
- An optional *intended vehicle*, telling the dispatcher to assign the order sequence to the specified vehicle instead of selecting one automatically. If set, all transport orders added to the order sequence must carry the same intended vehicle value.

Figure 3.16 shows an example representing the structure of an Order Sequence.

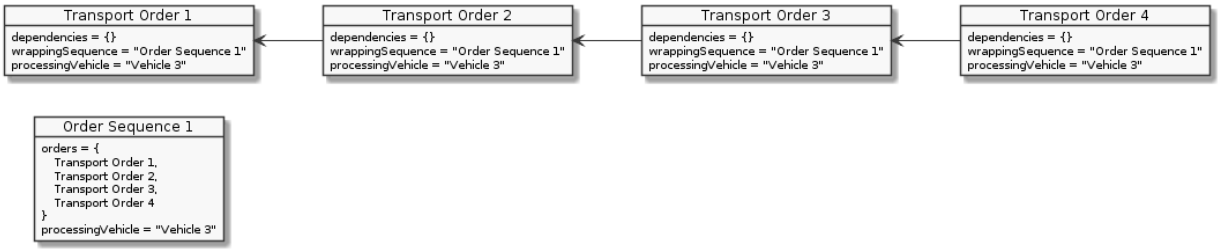

Figure 3.16: Diagram representation of an Order Sequence [15]

3.5 OpenRMF vs OpenTCS comparison

While both OpenRMF and OpenTCS can be used as control systems for a multi—robot environment, they both come with a different implementative structure and present important differences that need to be pointed out clearly.
The main difference consists in the fact that while OpenRMF is provided with an inter—fleet communication interface, allowing the coordination among multiple fleets, OpenTCS is not, and is specifically thought to function as a single fleet control system.
OpenRMF instead relies on the *Task Dispatcher*, the *Traffic Schedule Database* and on the bidding process to coordinate each robot in each of the diverse fleets in the environment, on the other hand OpenTCS is not provided with any of the just mentioned structures.
This is the reason why, if using OpenTCS as management system to control multiple fleets, it is necessary to have a control application for each one of them, in order to control any fleet independently.
Furthermore, each fleet can be controlled only after ensuring that no other robots belonging to a different one are still performing a task or are out of the *Workshop* area, that is saying in the area where the robots can move. This explains why it is necessary to wait that all the delivery tasks are finished and that all the active robots have come back to the *Workshop* before initiating a cleaning task, if using

OpenTCS.
Another important difference consists in how to request the execution of a task using the two frameworks.
With OpenRMF it is possible to exploit a web application toolkit, provided with an API server, called *RMF Web*, that allows allocating tasks through a more user friendly GUI. Alternatively, it is possible to directly allocate tasks via terminal, specifically:

- It is possible to allocate a single task opening a new terminal.
- It is possible to allocate multiple tasks simultaneously in the same terminal, through a *.launch* file containing all the desired tasks with the appopriate initial parameters.

With OpenTCS instead it is possible to exploit the *Plant Overview* client set to *Operating Mode* to allocate tasks, exploiting the GUI provided by the same client.

Chapter 4

Realization

The purpose of this chapter is to illustrate how the tools, libraries and plugins described better in detail in Chapter 3 were used to build the applications developed for this thesis work.
Specifically, two applications were developed:

- The first one, exploiting the functionalities provided by the OpenRMF framework, allowing the simulation of a complex system where multiple fleets of robots, each one dedicated to the execution of a specific task, can operate at the same time.
- The second one, using the OpenTCS fleet management system, where tasks of different types cannot be executed simultaneously, since there is no higher level layer allowing communication among the various fleets of the environment.

Through both these applications it is possible to simulate the behaviour of multiple robots in a mining environment, simulate the execution of tasks of various types and record the performances resulting from the usage of both OpenRMF and OpenTCS as control systems of the robot fleets.
In the end, the two applications will be evaluated to figure out which of them offers the best performances, assessing if it is possible to build an autonomous system with OpenRMF having all the robot fleets cooperating simultaneously, or it is necessary to resort to a more simple solution using a fleet manager like OpenTCS.

4.1 Creation of the environment

The first step to be considered in order to simulate any robot fleet control is the creation of the environment in which all the simulations will be performed, all the robots generated, all the tasks executed.
Since the applications developed for this thesis work are intended to simulate the behaviour of multiple robots into a mine, the simulated environment will be a mining environment.
Specifically, the model of the plant will refer to the mine layout shown in Chapter 2 (Figure 2.1), and will consist in a three-dimensional version of the same two-dimensional layout.
The simulation software used for the creation of the environment and for simulating the robots' behaviours is Gazebo 11, since it offers various advantages when developing and testing robotic entities in a virtual space. It is indeed first of all completely open-source, it integrates with ROS/ROS2, and provides realistic physics and sensor simulation, allowing to simulate from simple robots to robots with a more complex design and with more complex features.
Specifically, Gazebo 11 offers very useful tools for world generation (where world is a word to indicate the environment in Gazebo 11), that is Building Editor and Model Editor.
Exploiting the functionalities offered by Building Editor it was possible to:

- Have a reference design image over which it is possible to sketch the layout of the mine: this way the mine layout can be as similar as possible to the design contained in the initial reference image. As mentioned before, the figure considered to accomplish this task is Figure 2.1.

- Set the scale of the reference image. This allows to adapt the measurements of the reference image to the real world ones, by setting the number of pixels per meter. In this case, it was chosen 20 pixels per meter.

- Generate walls to identify the lanes where the robots can move in the mine. In this case the walls' dimensions are 2 meters in height and 0.5 meters in depth.

Model Editor instead, as the name suggests, allows the generation of models, Gazebo entities that, after being created, can be allocated and copied in any point of the simulated environment.
More specifically, Model Editor was used to generate the model of the pile of rocks present at the *drawpoints* indicated in Figure 2.1.
Figure 4.1 and Figure 4.2 show, respectively, the model of the rock pile generated through Model Editor and overall world used for simulations. Once the environment

is all done, it can be saved in a *.world* file to be opened when needed.

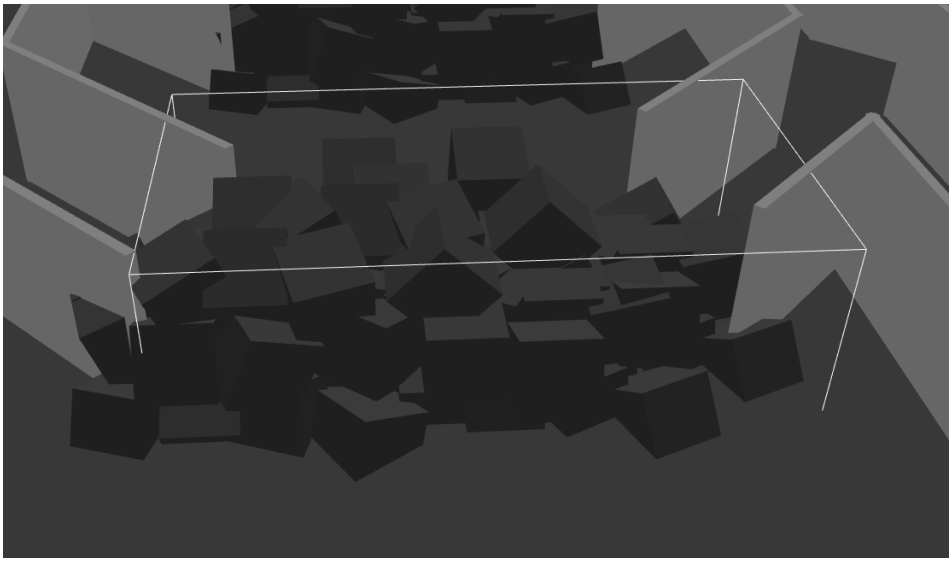

Figure 4.1: Pile of rocks

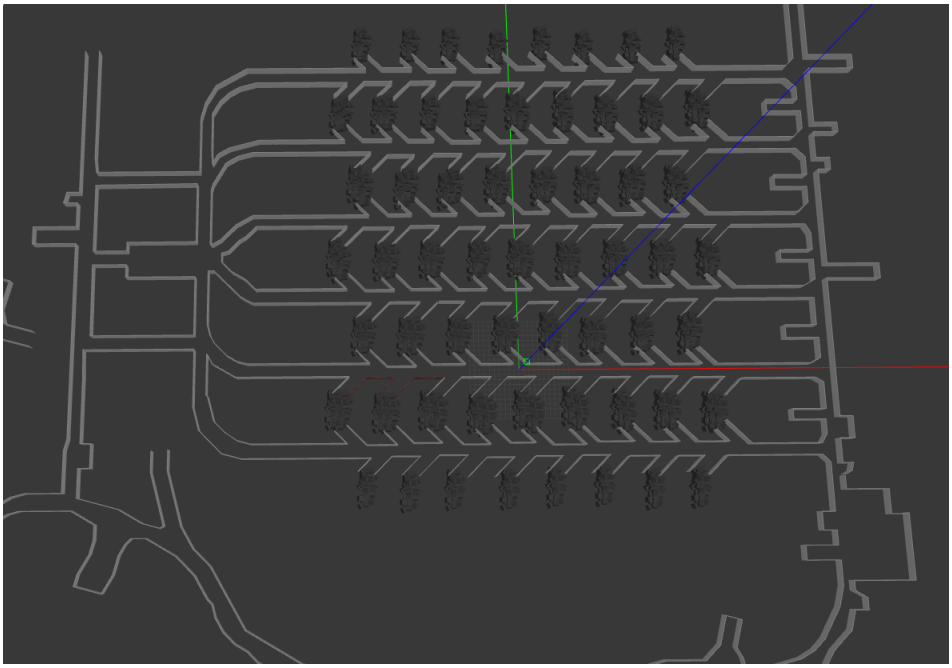

Figure 4.2: Environment generated through Gazebo

The origin of the reference system of the environment contained in the *.world* file generated trough Gazebo is in the second lane from the bottom of the layout shown in Figure 4.2, where the red, green and blue axes represent, respectively, the positive x, y and z axes.
It is very important to keep track of the origin of any reference system, since all along this thesis work many reference systems have been used, thing that implied the usage of coordinate transformations to remap them.

4.2 Map generation

Once the environment has been generated, it is necessary to build a map of the same environment, so to make the robots able to move within it and recognize eventual obstacles.
Hence, it is possible to provide the navigation stack, in this case *Nav2*, with a *.png* image and let it be in charge of the navigation tasks.
To be precise, the *.png* file in question should have the following properties:

- Having just three color levels:
  - White, representing the free space, that is to say the space that can be occupied by the robots and within they can move freely.
  - Black, representing the barriers of the environment, such as the walls, the rocks or other eventual objects.
  - Grey, representing the unknown space, for which it is not possible to determine if there are obstacles in it or if it is free space.

  This way, the navigation stack can interpret the information coming from the map and process it properly.
- Being an accurate representation of the three-dimensional environment.
- Having a scale parameter representing the number of pixels per meter and a reference system associated to it (not necessarily the same reference system of the *.world* file generated through Gazebo).

As it comes evident from these statements, the image shown in Figure 2.1 cannot be used as map of the environment for many reasons. It indeed is not a 2D replica of the world, since the walls created in Gazebo cannot be curved but only straight; it does not contain any simulated rock pile, while they are present in the environment; the writings on the picture, in black, will be interpreted by the navigation stack as obstacles.
Then, it is necessary to provide a new image having the aforementioned properties in order to provide the system with an actual map.

4.2.1 Cartographer

One of the tools that can be used for the purpose of map generation is *Cartographer*, a set of ROS packages used for SLAM (Simultaneous Localization and Mapping). It allows to generate accurate maps of the simulated environment while keeping track of the position of the robot simultaneously.

4.2.1.1 Turtlebot3

Cartographer relies on the robot models present in the *Turtlebot3* packages. In fact, it is necessary to have a robot generated in the environment to proceed with the mapping. Specifically, in the *Turlebot3* packages are present the *.urdf* models of three different robots, respectively named burger, waffle and waffle pi. Figure 4.3 and Figure 4.4 show the models of *Turtlebot3* burger and waffle in the mining environment generated through Gazebo 11.

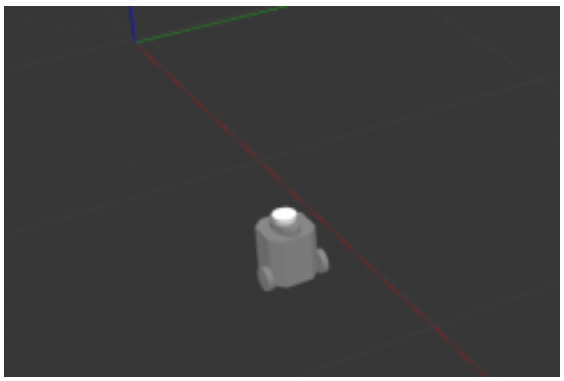

Figure 4.3: Turtlebot3 burger

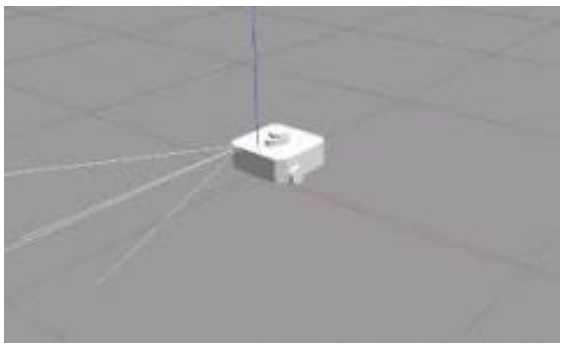

Figure 4.4: Turtlebot3 waffle

They differ from each other for shape and physical features, such as velocity and acceleration, that anyway can be changed by just adjusting these values modifying the *.urdf* files.
Anyway they have an important thing in common: they are all provided with a LiDAR (Light Detection and Ranging) sensor, that uses a laser to measure variable distances of the surrounding environment. This sensor is exploited during the SLAM process by *Cartographer* to build the map image, since it uses the distances of the surrounding walls and obstacles detected on the path to build the map.

4.2.1.2 Mapping process

Three processes have been running, respectively in three different terminals, to start mapping, namely they are:

1. turtlebot3_world.launch.py, that runs Gazebo 11 and reads the *.world* file that will be the environment of the simulations. Furthermore, allocates the robot entity in the world and activates the LiDAR sensor, whose measurements are published over the topic */scan*.
   Before running the *.launch* file, it is necessary to properly specify in it the name of the *.world* file along with its path (in this case, it was used the aforementioned file containing the mining environment), the *Turtlebot3* model chosen (burger was chosen), and the coordinates of the location where the robot will be generated.
   Figure 4.5 shows the robot generated in the mining environment with the LiDAR sensor activated.

2. teleop_keyboard, in charge of the teleoperation task. This allows the robot to move where requested, in order to map all the corners of the environment by simply using the computer keyboard or a joystick.

3. cartographer.launch.py, that activates the SLAM nodes in charge of building the map while the robot moves within the environment.
   This process also opens a Rviz window that facilitate the task, allowing the visualization of the generating map, showing the areas that are already mapped and the ones that still need to be discovered.

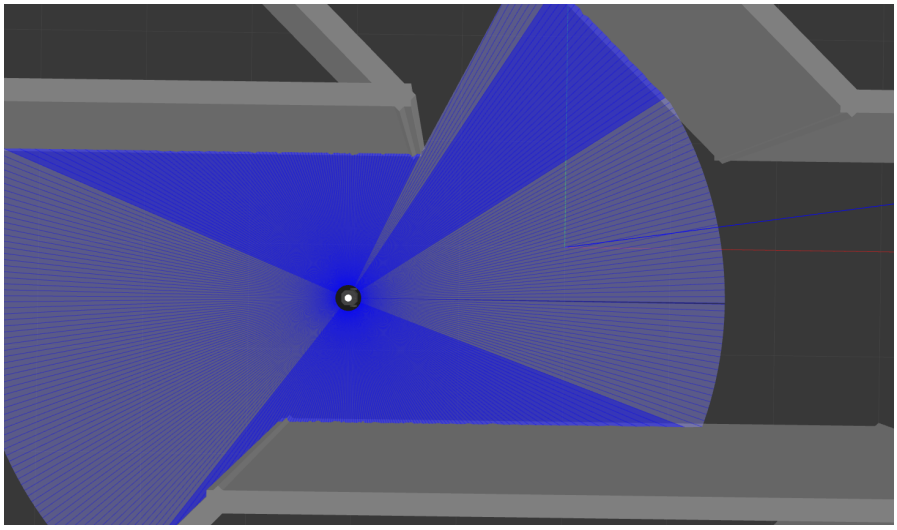

Figure 4.5: Mapping process using Cartographer

Once the entire environment was mapped, it was possible to save the image produced through the client *map_saver_cli*, that produces a *.pgm* output and a *.yaml* output describing scale of the image and its reference system.
Anyway, the map contained in said file had still some imperfections, such as obstacles that were not properly detected by the LiDAR or just partially identified. Some corners of the environment indeed, due to the geometry of the obstacles and the features of the sensor were quite difficult to scan completely.
In order to fix this problem the image editor Gimp was used to correct the imperfections occurred during the map generation.
As it can be noticed from Figure 4.6, some walls and corners of the environment were not properly identifies, and then later corrected producing the resulting image shown in Figure 4.7.
Lastly, the output produced by Gimp has the *.xcf* extension, that was converted in a *.png* image to be utilized later on with OpenRMF and OpenTCS.

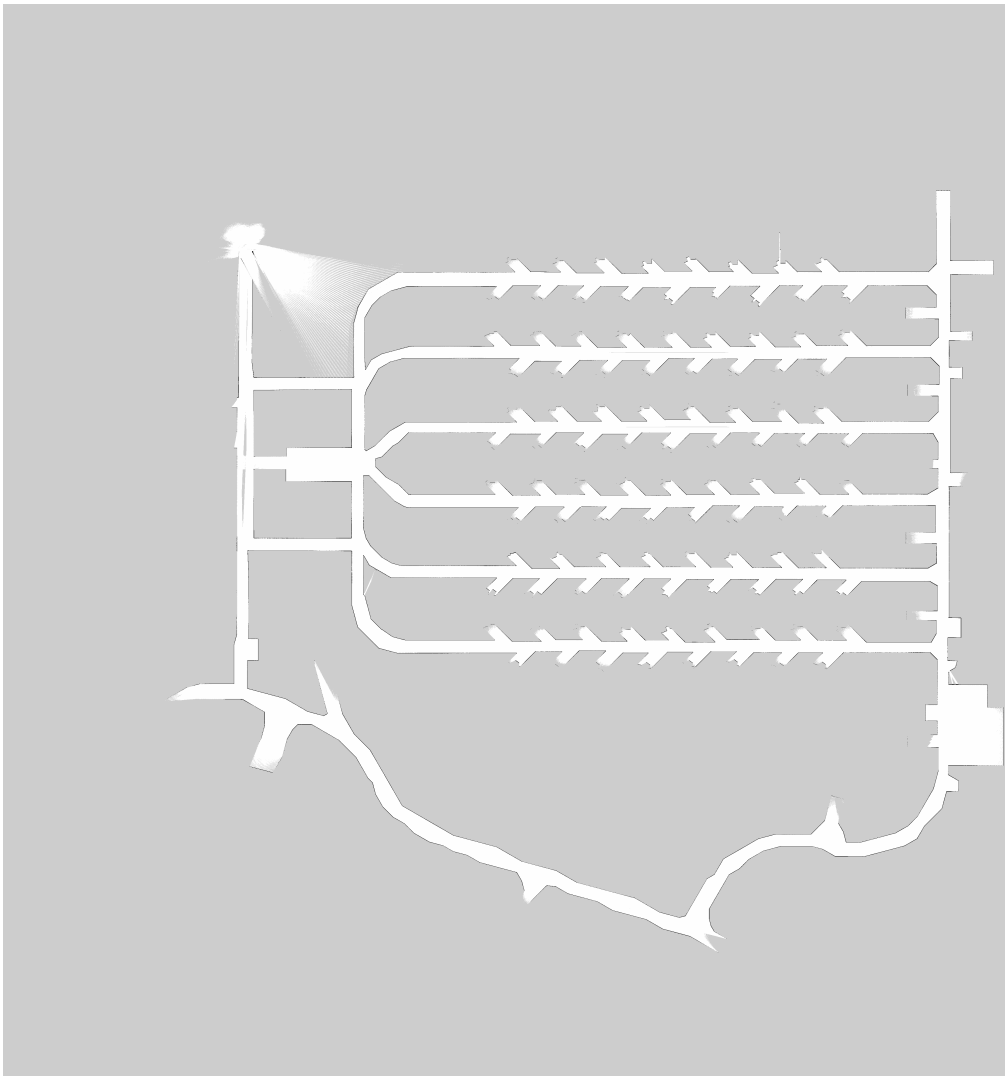

Figure 4.6: Map generated using Cartographer

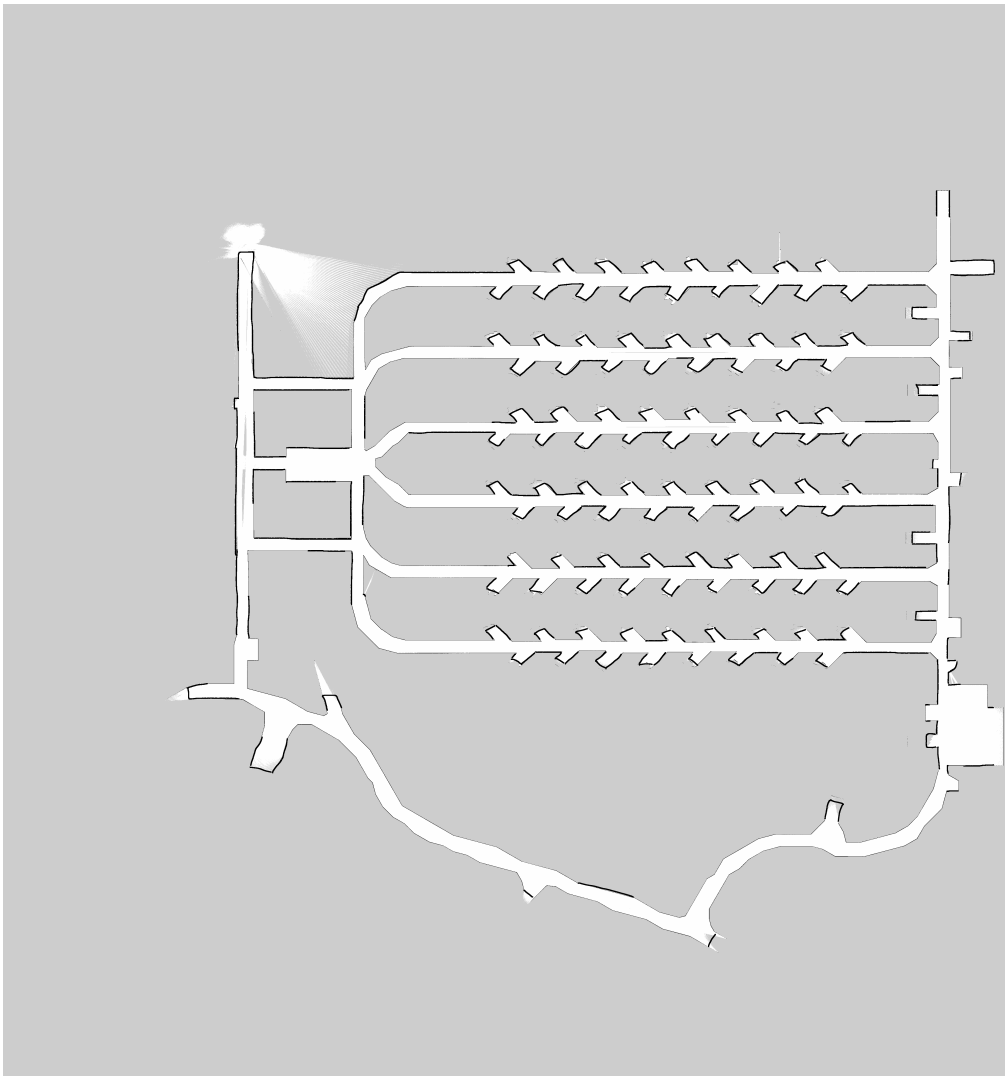

Figure 4.7: Corrected map

4.3 Robot spawn

Once the map of the simulated environment has been generated, it is possible to allocate robots into the environment, so to allow them to perform tasks and start simulations. This process of allocation of robots will be called from now on as *robot spawn*.
In order to be coherent with the layout design of the mine, the robots will be spawned in the workshop area shown in Figure 2.1, in the bottom-right part of the layout. The robots indeed are assumed to perform their initial task starting from

the workshop, and then come back to the same point where they were spawned at the end of the task or sequence of tasks.
As a matter of fact, when a robot finishes its task and has no other tasks to process, it begins being just an obstacle for the other robots if it occupies the lanes of the layout, where the majority of the tasks are executed. For this reason they are ordered to come back to the workshop, where they can safely become idle waiting for a new task to process.

4.3.1 Gazebo-Rviz coordinate transformation

As just said, the robots can be spawned in the workshop area, but, to be more precise, ten points have been chosen to be the spawning points, and they are shown in Figure 4.8, from p_1 to p_{10}.

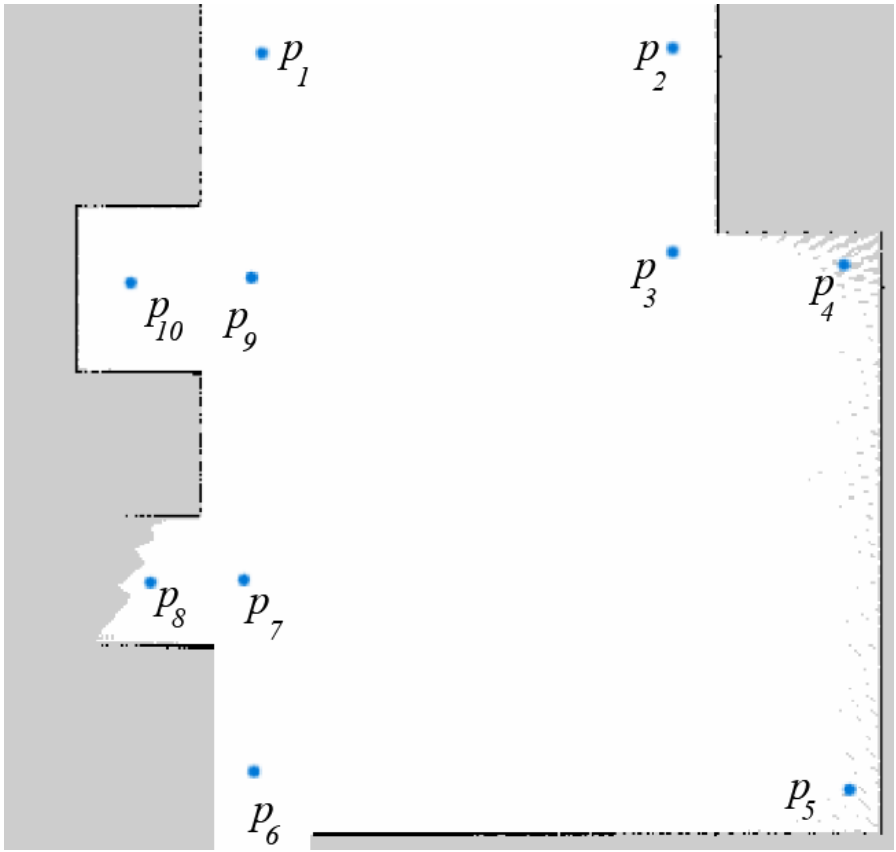

Figure 4.8: Permitted spawning points

In order to allow spawning the robots in the said points, it is necessary to know their x and y coordinates (the z coordinate is not relevant since they all lay on

the same level). In the *.world* file containing the simulated mining environment, the origin of the reference system lays in the middle of the fifth lane from the top, as shown in Figure 4.2. Through Gazebo 11 then it is possible to know the coordinates of the spawning points with respect to this reference system, that are:

$$p_1 : (61.70, -25.40)$$

$$p_2 : (70.30, -25.65)$$

$$p_3 : (70.15, -30.65)$$

$$p_4 : (73.75, -30.75)$$

$$p_5 : (73.75, -41.00)$$

$$p_6 : (61.90, -40.75)$$

$$p_7 : (61.50, -36.90)$$

$$p_8 : (55.40, -37.00)$$

$$p_9 : (61.50, -30.75)$$

$$p_{10} : (59.00, -30.70)$$

While Gazebo 11 allows spawning the robots in the simulation environment, Rviz has been used for visualization. Some behaviours indeed cannot be visualized through Gazebo 11, such as the path that the robots have to follow to get to the next navigation goal, the way the reference system of the robot moves and rotates with it, its range of detection of the obstacles along the path, and so on. This is the reason why, after spawning the robots in Gazebo 11, it was necessary to correctly make them appear in Rviz as well.
Anyway, an important thing that needs to be considered is that the map in Gazebo 11 and Rviz does not refer to the same reference system. While indeed the reference system of the first one is manually set at the moment in which the environment is created (so manually set to the fifth lane), the second is based on the *.png* map generated through Cartographer, that has the origin of its reference system at the upper-left corner of the image, with the positive x direction poiting to the right, and the positive y direction pointing up.
To be precise, the transformations to switch from the coordinates in Gazebo 11 to Rviz are:

$$x_{Rviz} = x_{Gazebo} + 137.0$$

$$y_{Rviz} = y_{Gazebo} - 110.5$$

Making the new coordinates in Rviz be:

$$p_1 : (198.70, -135.45)$$
$$p_2 : (207.30, -135.70)$$
$$p_3 : (207.15, -140.70)$$
$$p_4 : (210.75, -140.80)$$
$$p_5 : (210.75, -151.05)$$
$$p_6 : (198.90, -150.80)$$
$$p_7 : (198.50, -146.95)$$
$$p_8 : (192.40, -147.05)$$
$$p_9 : (198.50, -140.80)$$
$$p_{10} : (196.00, -140.75)$$

4.3.2 Spawning nodes

All the ROS2 nodes in charge of spawning the robots in the mining environment are collected in the *.launch* file *robots_mina.launch.py*.
Specifically, two fleets of Turtlebot3 robots have been spawned, the burger fleet and the waffle fleet, respectively destined to perform delivery and cleaning tasks. Many of these nodes make use of *namespaces*: a namespace is a prefix preceded by a '/', that specifies which robot a specific topic is referring to. For instance, the topic specifying the initial pose of the robots named *tb1* and *tb2* will be, respectively, */tb1/initialpose* and */tb2/initialpose*. According to the task, the spawning nodes can be categorized in three main groups, as reported below.

1. The first group is in charge of allocating the robots in Gazebo. The coordinates of their positions are collected in a 2 by n vector, where n is the number of robots to be allocated, and where the position in the vector indicates the robot number. For each robot, the executable *spawn_entity.py* is run: this executable allows visualizing the tridimensional model of the robot in Gazebo 11, taking some inputs arguments:
   - The name of the robot.
   - The coordinates of the spawning position with respect to the reference system in Gazebo.
   - The *.urdf* file containing the model, describing the shape, the physical features, and the usage of eventual sensors.
   - The namespace associated to this specific robot.

2. The second group is in charge of opening the Rviz windows for the visualization tasks. Specifically, it takes as input the *.yaml* file associated to mine map, and the information related to the robot spawned, such as name, namespace, and its coordinated with respect to the reference system in Rviz. This allows to visualize the map, with different colors to distinguish between obstacles and free space, the robots themselves, with their reference systems moving with them, and sets the default visualization configurations.
   Figure 4.9 shows how a robot is visualized in Rviz.

3. The third group is in charge of activating the navigation stack, all nodes belonging to the *Nav2* packages.

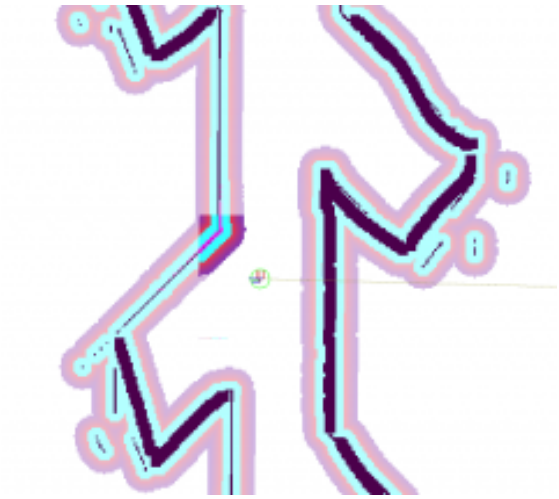

Figure 4.9: Rviz visualization

4.4 Employment of OpenRMF

After the generation of the simulated environment, the creation of the relative map, and the spawn of the robots within it, the OpenRMF framework has been employed for controlling the robot fleets in order to perform tasks.
The first step is to provide the robots with their navigation graphs, representing the paths they can occupy to move inside the mining environment.
The map itself indeed does not contain all the information related to how the robots can move inside the map, for instance, it does not tell which are the areas in which the robots cannot enter, the direction and the orientation the robot must have while following a path, the areas dedicated to the execution of tasks, and so on.

4.4.1 Navigation graph generation

The navigation graphs have been generated exploiting the functionalities provided by the Traffic Editor GUI. It allows indeed to draw the paths on a reference image, in this case the *.pgm* image containing the corrected map of the mining environment, so to allow a better precision while generating the navigation graphs.
The first step was the allocation of waypoints, locations that the robots can occupy and that can have different properties according to the requirements of the environment.
In particular, the distribution of the waypoints and their names where chosen following a precise criterion, in order to facilitate the task allocation process. The aforementioned criterion is the following:

- For each of the six horizontal lanes going from the Workshop area to the Crushing Station have been allocated 37 waypoints, named according to a specific pattern.
  N stands for the lane number, and it falls in the range *[1,...,6]*, so that $N = 1$ represents the uppermost lane, while $N = 6$ represents the lowest one.
  On the other hand, n stands for the number of the waypoint in the i_{th} lane, within the range *[1,...,20]*, so that $n = 1$ represents the leftmost waypoint in the lane, while $n = 20$ represents the rightmost one.
  Then:

  - If the waypoint falls precisely in the lane, according to its position, it will be identified by the name *L_<N>_<n>*.
    For example, the fith waypoint in the third lane is identified by the name *L_3_5*.
  - If the waypoint falls inside the drawpoint branch of a lane, it will be identified by the name of the closest waypoint followed by the suffix *_in*.
    For example, the waypoint that falls in the drawpoint branch closest to the *L_3_5* waypoint, is identified by the name *L_3_5_in*.

- The waypoints that allow the access to the Crushing Station are identified with the name *U_n*, where n is the number of the waypoint, and falls within the interval *[0,...3]*. Specifically, *U_0* is the lowest one and *U_3* is the uppermost one.

- The spawning waypoints inside the Workshop are identified with the name *E_n*, where n is the number of the waypoint, and falls within the interval *[1,...,10]*. They are numbered in clockwise order, starting from the uppermost and leftmost waypoint.

- The vertical path on the right of the map that connects the workshop with the horizontal lanes presents 27 waypoints, named according to the following pattern:
  - If the waypoint falls precisely within the vertical path, it is identified by the name *F_n*, where *n* represents the number of the waypoint, and falls within the interval *[1,...,17]*. Specifically, *F_1* represents the lowest waypoint and *F_17* the uppermost one.
  - If the waypoint falls in one of the horizontal branches, it takes the name of the closest waypoint followed by the suffix *_in*. For example, the waypoint that falls within the branch of the *F_8* waypoint, is identified by the name *F_8_in*.

Some examples are shown in Figure 4.10.

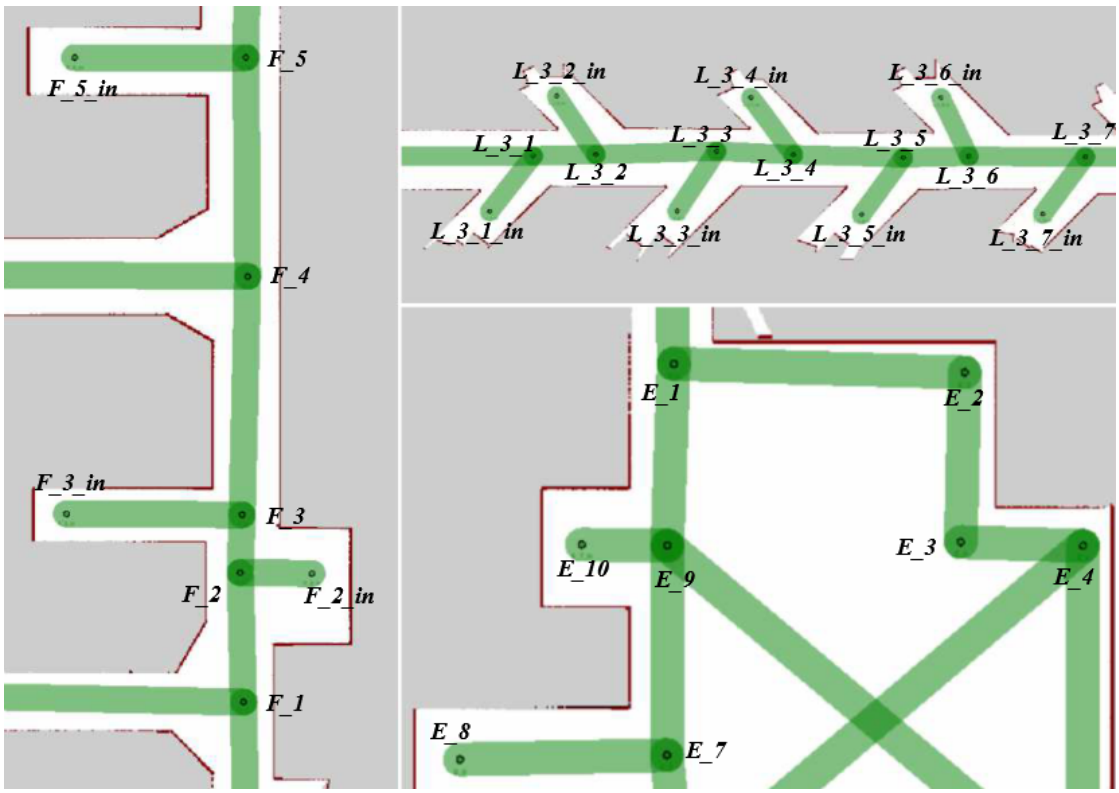

Figure 4.10: From the left, clockwise: vertical path connecting the workshop with the horizontal lanes; third lane; workshop.

Once all the waypoints were generated and properly named, the following step has been the definition of their properties. According to the location they occupy in the map, the waypoints characterized by special properties are the following:

- The *E_n* waypoints, that are the ones where it is allowed to spawn the robots, have the property *spawn_robot_name* set to the name of the robot that has to be generated at that specific position, have the flag *is_charger* and the flag *is_parking_spot* set to true, so that the robots can recognize this location as charging and parking station.

- The *L_N_n_in* waypoints, have the property *pickup_dispenser* set to the name of the plugin used to simulate the pickup of the object to be delivered, that is in this case the Teleport Dispenser plugin. These waypoints indeed, as it can be observed from the layout of Figure 2.1, coincide with the pickup locations to perform delivery tasks.

- The *U_n* waypoints have the property *dropoff_ingestor* set to the name of the plugin used to simulate the dropoff of the object to be delivered, that is in this case the Teleport Ingestor plugin. These waypoints indeed, as it can be observed from the layout of Figure 2.1, coincide with dropoff locations to perform delivery tasks.

- Some waypoints within the six horizontal lanes, the *L_N_n* ones, have the flag *is_cleaning_zone* set to true, and the property *dock_name* set to *zone_x*, where *x* represents the number associated to the cleaning zone. According to the simulation performed, the number of and the disposition of the cleaning zones can vary, reason why there are not fixed waypoints having set these properties, but it depends instead on the simulation that has to be done.

Finally, the last step to complete the characterization of the navigation graphs, is the generation of the paths interconnecting the waypoints. The waypoints have been connected as shown in Figure 4.11. Every path, has the following properties set: direction set to *bidirectional*, since all the paths can be covered in both directions; orientation unspecified, since all the path can be covered both forward and backward oriented.

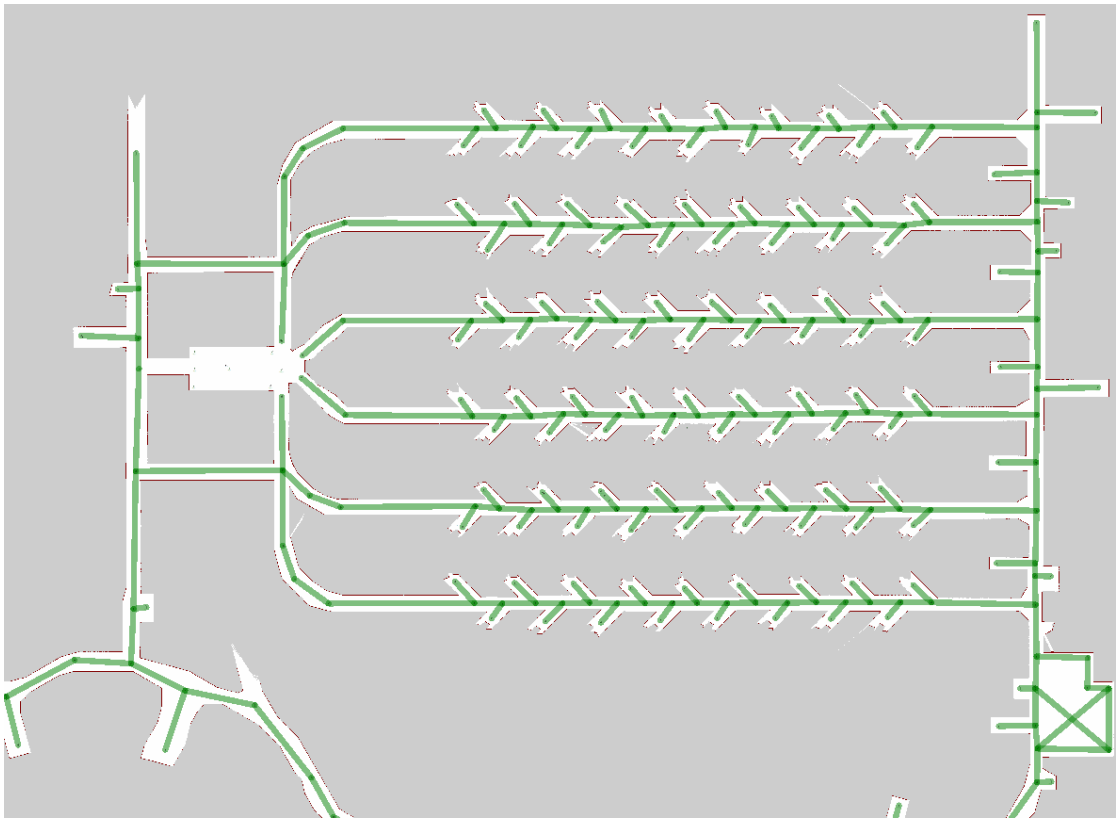

Figure 4.11: Complete navigation graph

Once completed, the navigation graph can be saved in Traffic Editor, that produces an *.building.yaml* output. Anyway, this file has to be converted to a *.yaml* extension in order to be properly interpreted by the fleet adapter. To solve this problem, the node *building_map_generator* of the package *rmf_building_map_tools* has been used, that accounts for this conversion.

4.4.2 Fleet adapter configuration

Each robot fleet of the system needs to be associated to a fleet adapter in order to allow the communication with the core of OpenRMF. In order to do this two *.launch.xml* files were developed activating all the essential nodes to implement the fleet adapters, they are *turtlebot3_burger_adapter.launch.xml*, for the burger fleet, and *turtlebot3_waffle_adapter.launch.xml*, for the waffle fleet.
These files, in order to correctly implement the functionalities they need to provide, take as inputs:

- *fleet_name*, respectively burger and waffle.
- *control_type*, that is set to full control, since the fleet adapter should be able to access to all the possible information the robots can transmit in order to properly implement the bidding process and the task dispatching process.
- *nav_graph*, indicating the file containing the navigation graph the fleet refers to for path planning. The navigation graph generated at the previous point was selected.
- *linear_velocity*, set to 30.0, as specified in the *.urdf* files.
- *angular_velocity*, set to 2.0, as specified in the *.urdf* files.
- *linear_acceleration*, set to 1.5, as specified in the *.urdf* files.
- *angular_acceleration*, set to 2.0, as specified in the *.urdf* files.
- *footprint_radius*, indicating the radius of the circle representing the physical limits of the robot, set to 0.5.
- *vicinity_radius*, indicating the radius of the circle representing the area around the robot that any other robot or obstacle cannot enter, set to 0.7.
- *perform_loop*, that enables the robot to perform patrol tasks, set to true for both the burger fleet adapter and the waffle fleet adapter.
- *perform_deliveries*, that enables the robot to perform delivery tasks, set to true only for the burger fleet adapter.
- *perform_cleaning*, that enables the robot to perform cleaning tasks, set to true only for the waffle fleet adapter.
- *discovery_timeout*, indicating the maximum time the fleet adapter can take to compute the paths the robots have to follow to perform a task, before giving up, used for te bidding process, set to 60.0.
- *battery_voltage*, set to 12.0.
- *battery_capacity*, set to 24.0.
- *battery_charging_current*, set to 5.0.
- *recharge_threshold*, indicating the battery level at which the robot ceases to operate, set to 0.1.

- *recharge_soc*,indicating the battery level up to which the robot should charge before becoming operative again.
- *mass*, set to 20.0, as specified in the *.urdf* files.
- *inertia*, set to 10.0, as specified in the *.urdf* files.
- *friction_coefficient*, set to 0.22, as specified in the *.urdf* files.

Finally, for each fleet, they activate the node *rmf_fleet_adapter*, that is in charge of executing all the operations described in the full control part of the Table 3.1, and to perform robot election and path planning in order to participate to the bidding process.
Anyway, the fleet adapters, as explained more in detail in Chapter 3, are not the only players of the game: actually other nodes have to be activated to make the bidding process and the task dispatching process possible.
For this reason, another *.launch.xml* file has been developed, named *common.launch.xml*. Firstly, it activates the *rmf_schedule_node*, that implements the Traffic Schedule Database, needed to take track of the scheduled paths of the robots for tasks execution and able to notify the presence of eventual conflicts; secondly, it activates the *rmf_dispatcher_node*, that implements the Task Dispatcher, needed to select, through the bidding process, the best fleet to execute a specific task.
Lastly, it launches the *visualization.launch.xml*, that gets in charge of setting all the visualization configuration for Rviz. Specifically, this last file activates a series of nodes that are part of the *rmf_visualization* package, such as: the *schedule_visualizer_node*, that allows to display the programmed routes the robots have to follow, along with their footprint and vicinity radius; the *fleet-states_visualizer_node*, needed to visualize the current position of the robots as reported by their fleet management systems.

4.4.3 Server-client configuration

Finally, the fleet management system, that is in charge of translating the high level commands coming from the fleet adapter into commands that can be understood by the robots, has been implemented.
Since the Turtlebot3 fleets are not provided with any fleet management system, it was necessary to implement one. To be precise, the open source package *free_fleet* had been exploited for this task.
As explained in Chapter 3, the fleet management system that can be implemented through this package consists in a client-server structure, reason why it was necessary to develop four *.launch.xml* files: two for the client part and two for the server part, considering that the robot fleets are two, burger and waffle.

4.4.3.1 Server

The *.launch.xml* files implementing the server, respectively named *burger_server.launch.xml* and *waffle_server.launch.xml*, activate the node *free_fleet_server_ros2*, contained in the *free_fleet* package, that implements the server part of the fleet management system. It is indeed in charge of dispatching the commands coming from the fleet adapter to the correct robot of the fleet, relying on the *CycloneDDS* communication type.
This node, according to the parameters that takes as inputs, constitutes the burger server or the waffle server. Delving more in detail, the parameters are those listed below:

- The name of the fleet, set to burger or waffle, according to the fleet server that has been implemented.
- The names of the topics published by the fleet adapter the server will have access to, these are:
  - *fleet_state_topic*, set as *fleet_states*, and containing the information related to the pose and the location of the robot within the environment.
  - *mode_request_topic*, set as *robot_mode_requests*, and containing the information related to the operating mode of the robot (for instance idle, cleaning, docking, moving, and so on).
  - *path_request_topic*, set as *robot_path_requests*, and containing the information related to the path the robot has to follow in order to perform a task or to get to a point where to perform a specific operation.
  - *destination_request_topic*, set as *robot_destination_requests*, and containing the information related to the final destination the robot has to get to.
- The names the topics mentioned at the previous point will be published with to the client trough CycloneDDS, these names are, respectively:
  - *dds_robot_state_topic*
  - *dds_mode_request_topic*
  - *dds_path_request_topic*
  - *dds_destination_request_topic*
- The DDS domain that allows the connection between server and client. It has been set to 42 for the burger server and to 43 for the waffle server. The DDS domain specified by the client needs to be same in order to allow the communication.
- The frequency at which each topic will be published to the client, set to 2.0.

4.4.3.2 Client

While the server dispatches the commands coming from the fleet adapter, and returns to it the states and positions of the robots in the fleet, the client side of the fleet management system implemented through the *free_fleet* packages is in charge of translating this commands to the robots in such a way that they can be understood, in order to allow the robots to actually move and perform tasks.
The clients have been implemented developing two more *.launch.xml* files, *burger_client.launch.xml*, for the burger fleet, and *waffle_client.launch.xml*, for the waffle fleet.
They both rely on the node *free_fleet_client_node* of the *free_fleet* packages, that implements all the aforementioned functionalities and allows the physical control of the robot. As for the server case, according to the parameters specified in the *.launch.xml* file, this node can function as the burger client or the waffle client. An important thing that differentiates the client side from the server one, is that it is necessary to have one client for each of the robots in the fleets, and thus one node for each robot, even though it is possible to activate all of them in a single *.launch.xml* file.
Specifically, the parameters that tells the node which will be the robot in the fleet to be controlled are:

- The name of the fleet, set to burger or waffle.
- The DDS domain, that has to coincide with the one specified at the server side, thus set to 42 for the burger fleet and to 43 for the waffle fleet.
- The name of the robot associated to this client, that has to coincide to the name specified in the file exploited for the spawn of the very same robot.
- The *Nav2* server name, set to *navigate_to_pose*, since it makes use of this server for sending navigation goals.

4.4.4 Task definition

The last step before having the application ready for simulation consists in allocating the tasks the robots will have to execute.
The application developed involves three types of task, that are the patrol, clean and delivery task. The execution of these tasks relies on the executable files *Patrol.cpp*, *Clean.cpp* and *Delivery.cpp*, contained in the *rmf_task* package, and referenced by the fleet adapters of each fleet of the system.
These executables contain all the structures and functions to send commands to the fleet management server, and so, indirectly, to the various fleet management clients that will physically control the robots; but also contain the functions needed to estimate the time consumed by each robot of the fleet to complete said task, so to allow the fleet adapter to participate to the bidding process.
Anyway, before the execution of the task, it is needed to inform the fleet adapters about which task it is intended to be executed. The Task Dispatcher indeed, will send a notification to every fleet adapter of a new incoming task only after receiving a task request.
As described more in detail in Chapter 3, the Task Dispatcher only accepts task requests in a precise format in order to interpret it correctly.
To this aim, the *task_requester* node is invoked: using this node, it is only necessary to specify via terminal all the necessary parameters, and it will be in charge of serializing the information in the correct format and of properly sending it to the Task Dispatcher.
Then, three *.py* files have been developed, one for each of the three task types, each of them invoking the *task_requester* with the right parameters, that of course vary according to the task that has to be executed.
The files in question, that will be launched via terminal, and the parameters to be specified, are the following ones:

1. *dispatch_patrol.py*, launched with the following parameters:
    - -p, places to patrol through, can me more than one, compulsory argument.
    - -n, number of loops to be performed, optional. If the number of loops is not selected, it will be set to 1 by default.
    - -F, selected fleet, optional. If the fleet is not selected, it will be chosen through the bidding process.
    - -R, selected robot, optional. If the robot is not selected, it will be chosen through the bidding process.
    - -st, start time, optional. If not selected, it will be set to 0 by default.
    - -pt, priority, optional. If not selected, it will be set to 0 by default.
    - –use_sim_time, set to false by default

2. *dispatch_clean.py*, launched with the following parameters:
    - -cs, cleaning zone, compulsory argument.
    - -F, selected fleet, optional. If the fleet is not selected, it will be chosen through the bidding process.
    - -R, selected robot, optional. If the robot is not selected, it will be chosen through the bidding process.
    - -st, start time, optional. If not selected, it will be set to 0 by default.
    - -pt, priority, optional. If not selected, it will be set to 0 by default.
    - –use_sim_time, set to false by default.
3. *dispatch_delivery.py*, launched with the following parameters:
    - -p, pickup places, can me more than one, compulsory argument.
    - -d, dropoff places, can be more than one, compulsory argument.
    - -ph, pickup handler, indicating the instrument or plugin that is in charge of disposing the object on the robot, can be more than one, compulsory argument.
    - -dh, dropoff handler, indicating the instrument or plugin that is in charge of taking the object from the robot, can be more than one, compulsory argument.
    - -pp, pickup payload, indicating the object that has to be picked, can be more than one, compulsory argument.
    - -dp, dropoff payload, indicating the object that has to be dropped, can be more than one, compulsory argument.
    - -F, selected fleet, optional. If the fleet is not selected, it will be chosen through the bidding process.
    - -R, selected robot, optional. If the robot is not selected, it will be chosen through the bidding process.
    - -st, start time, optional. If not selected, it will be set to 0 by default.
    - -pt, priority, optional. If not selected, it will be set to 0 by default.
    - –use_sim_time, set to false by default.

Teleport plugins

For what concerns the delivery task, specifically the pickup and dropoff handlers, two already implemented plugins offered by *rmf_demos_assets* package have been used, they are *Teleport Dispenser* and *Teleport Ingestor*.

These plugins allow to simulate, respectively, the process of dispensing an object by positioning it on the top of the robot, and the dropoff of the same obect where it is required to be delivered.
Once the robot reached the pickup point or the dropoff point in the delivery point, it will notify the relative plugin by sending a message over the topics *dispenser_request* or *ingestor_request*: this activates the plugins that will teleport the object either on the robot or on the delivery point, by simply modifying its position. Once the pickup or dropoff action is completed, the plugin sends back a confirmation message over the topics *dispenser_result* or *ingestor_result*, communicating if the task was performed correctly.
In order to properly integrate these two plugins with the system, their associated *.urdf* models have to be added to the *.world* file containing the mining environment. Figure 4.12 shows how an object, simulated through the model of a coke can, is teleported on the delivery robot using the plugin Teleport Dispenser.

Figure 4.12: Teleport Dispenser plugin (the cube on the left of the image)

4.5 Employment of OpenTCS

The second application developed for this thesis work aims at the very same objective of the first one, the control of a multi-fleet system in a mining environment, but exploiting another framework, OpenTCS, that as it will be demonstrated in Chapter 5, has different characteristics, limitations, pro and cons with respect to the OpenRMF framework.
As for OpenRMF, the OpenTCS fleet management system needs to be provided, along with the simulated environment, with a sort of navigation graph, that in this framework takes the name of plant model, indicating the allowed paths that the robots can follow inside the mine.

4.5.1 Plant Model generation

Unfortunately, OpenTCS does not come with a GUI like Traffic Editor, that allows to exploit a map as a reference on which the plant model can be drawn: the GUI offered by OpenTCS indeed, the Plant Overview client, does not allow using any reference image, making the process way longer.

4.5.1.1 Points

In order to provide the system with a plant model without the possibility to use a reference image, it was needed to write the raw *.xml* code for the generation of the points of the map, representing the positions that the robots can occupy.
Of course, the plant model generated for OpenTCS, needs to be the best representation of the navigation graph generated for OpenRMF, in order to efficiently compare the two applications.
For this reason, the coordinates of the points that were inserted in the *.xml* file of the plant model where directly taken from the *.building.yaml* file containing the navigation graph. This way, after inserting the correct coordinates in the produced code, it was sure that all the points of the plant model were perfectly representing the very same waypoints of the navigation graph.
After running the *.xml* file containing the plant model using the Plant Overview client, it was now possible to use the GUI to finish editing the file.
For each point generated the property *halt point* was set to true: this is a necessary modification that has to be done, because in order to make a path routable this has to contain at least an halt point, so to allow the robot to eventually stop in case of conflict. Since it is not possible to forecast the routes that will be calculated by OpenTCS in advance, and since there was no specific restriction about this, all the points of the plant model were set to halt points.

Furthermore, in order to allow a decent visualization of the plant model, considering the huge dimensions in comparison with the dimensions of the screen of the GUI, also the layout coordinates (reference in Chapter 3) were changed to a value about ten times smaller.
Lastly, it was not needed to specify other properties relative to the points, as done for OpenRMF, since as explained in Chapter 3, the characterization of a specific position as a position where it is possible to execute a task, in OpenTCS, is defined trough locations, and it is not an intrinsic property of a point.
In order to imitate as much as possible the navigation graph, all the points of the plant model were named so to keep the same names of the respective waypoints in OpenRMF.

4.5.1.2 Paths and Locations

The points of the plant model were interconnected among them respecting the scheme provided by the navigation graph of the first application.
Anyway, in order to emulate the first example, all the paths connecting two consecutive points need to be bidirectional. Unfortunately, paths in OpenTCS can only be unidirectional, hence it was necessary to draw between two points a double path, consisting in two paths that are overlapped and opposite in direction. This aspect can be better seen in Figure 4.13 and Figure 4.14 noticing the direction to which the arrows of the paths point in the plant model.
In oder to allow task executions, location had to be added to the model, specifically:

- A location, named *CRUSHING_STATION*, was added at the corresponding area of the layout of *Figure 2.1*, representing the position at which it is possible to execute the dropoff the objects picked at the drawpoint branches.
  This location has been connected to the points *U_0*, *U_1*, *U_2* and *U_3*, since it is possible to have access to the crushing station from all of this four points. The location type was set to *UNLOAD*, since the type of task that can be executed at this location is dropping off an object previously loaded.

- A set of *LOAD* type locations have been added to the plant model in order to allow the vehicles to pick objects along the path before dropping them at the dropoff location.
  These locations were named *LOAD_L_N_n_in*, where *N* represents the number of the lane, within the interval *[1,...,6]*, and *n* represents the number of the point in said lane, within the interval *[1,...20]*, where it is possible to execute an pickup task.

- A set of *CLEAN* type locations have been added to the plant model, specifically six, one for each lane, named *CLEAN_ZONE_N*, where *N* represents the number of the lane where it is required to perform a cleaning task.

Done this modifications, the resulting plant model looks like the one shown in Figure 4.13, with some areas of interest better evidenced in Figure 4.14.

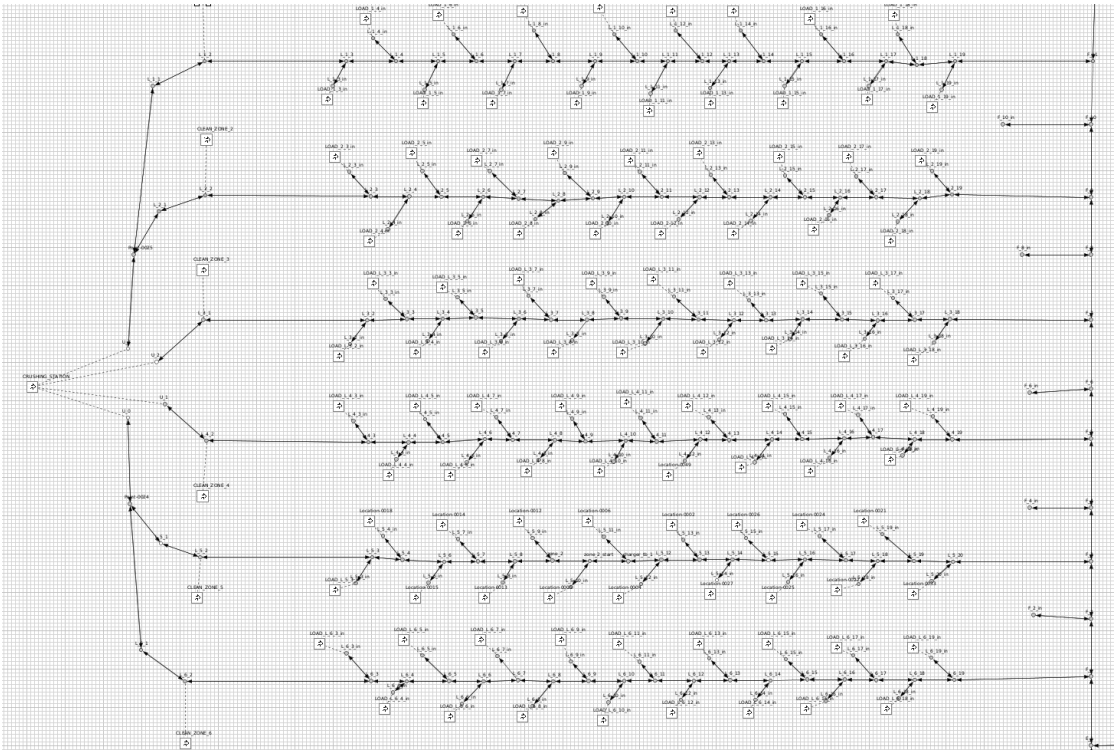

Figure 4.13: Resulting plant model

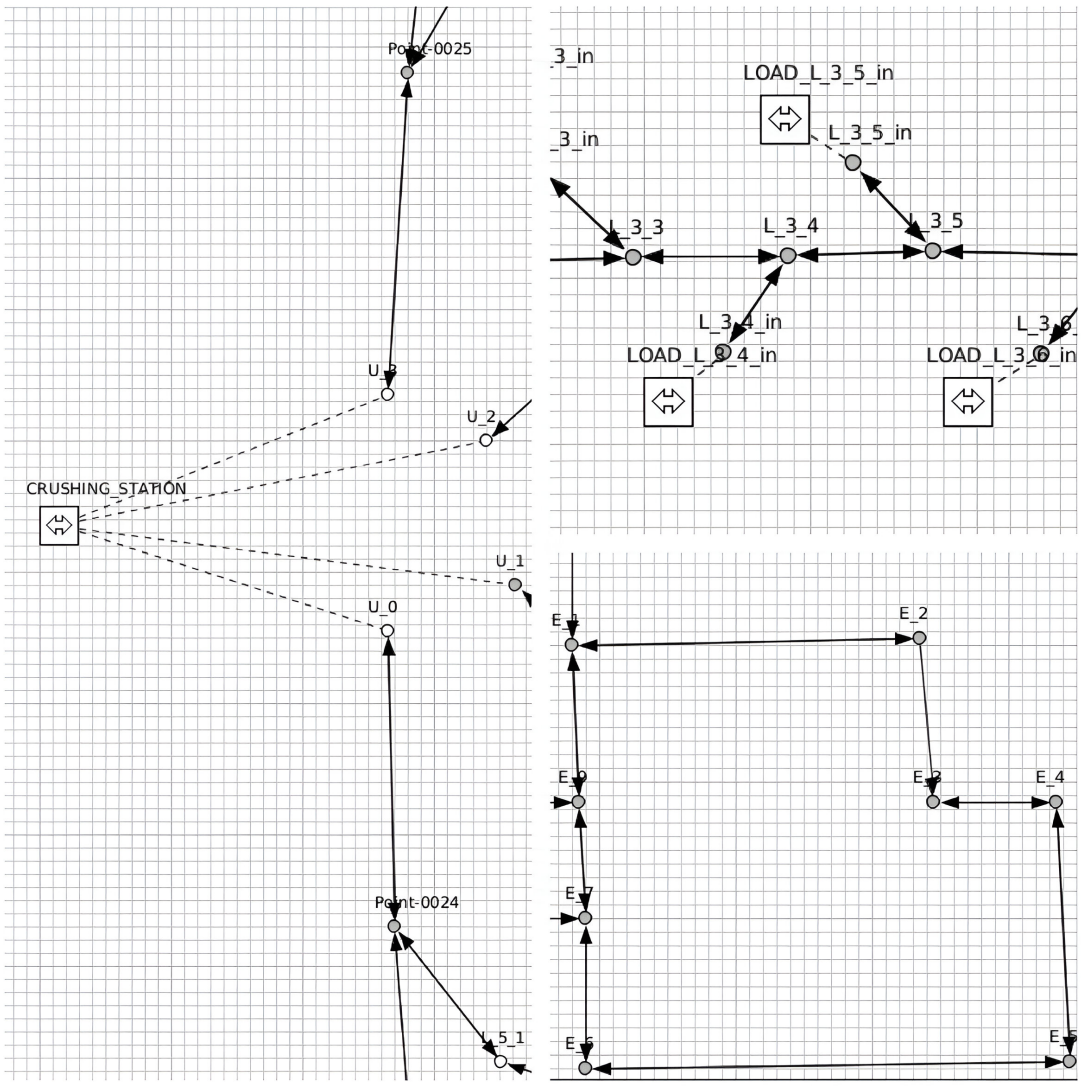

Figure 4.14: From the left, moving clockwise: the *CRUSHING_STATION* location with some of the cleaning locations; some of the *LOAD* type locations; the area of the map where the robots are spawned.

4.5.1.3 Vehicles

Finally, the vehicle models have been added to the plant model. In order to do that it was necessary to set the following parameters through the Plant Overview client:

- *Critical energy level*, set to 0.1.
- *Good energy level*, set to 1.0.

- *Fully recharged energy level*, set to 1.0, since in OpenRMF there is no distinction between this parameter and the previous one.

- *Sufficiently recharged energy level*, set to 1.0, since in OpenRMF there is no distinction between this parameter and the previous one.

- *Integration level*, set to *utilized*, so to allow the vehicles to process transport orders (and so to execute tasks) and to be considered by OpenTCS during the path planning process.

The names of the vehicles were chosen in such a way to be compliant with the names of the robot spawned trough the *.launch.py* file developed for the spawning process.
Once the plant model was finalized, it has been saved from the Plant Overview client, that automatically generates the final *.xml* file containing its code.

4.5.2 Operating Mode

Once the plant model has been correctly generated it is possible to abandon the *Modelling Mode* of the Plant Overview client, and switch to the *Operating Mode* to continue develop the application. From now on, it was possible to make the robot appear on the GUI of the Plant Overview client, by activating the *Kernel Control Panel*. This can be done via terminal, running the *startKernelControlCenter.sh* executable, showing a new GUI.
Through this panel it is also possible to control the vehicles and order them to execute tasks, but first it has been necessary to set up all the configurations of the Kernel Control Center.
As explained in Chapter 3, the following ones were the steps followed for the setup:

1. First of all the adapter has to be activated, so to allow the communication between the OpenTCS framework and ROS2.
   The adapter OpenTCS-NeNa has been chosen for this purpose, for all the vehicles spawned in the environment.

2. Once activated the adapter, it was needed to specify the namespace through which it is possible to identify the specific robot that it is nedeed to control. The adapter indeed will communicate the commands to the right robots by publishing topics whose name is preceded by the namespace associated to said robot. For instance, the namespace of the robot *tb1*, will consist in its name preceded by a '/', thus it will be */tb1*.
   All the topics and services exploited for the navigation of *tb1* will then be referenced by this namespace.

3. Then it has been proceeded to the spawn of the robots over the Plant Overview client, by setting their initial position through the Kernel Control Center, allowing them to be visualized in the plant model.
   By doing this, the topic */initial_pose* is updated with the actual position of the vehicle, that of course needs to coincide with the position specified in Rviz during the spawning process at the very initial stage.

4. Lastly, before having the vehicle ready to execute tasks, it was necessary to send a first navigation goal, done through the Kernel Control Center as well, to allow it to orient itself and to get a better estimation of the environment that surrounds it. While moving indeed, the robot refines the intial estimation it has of the surrounding obstacles.

Once set all these configurations, the robots are finally ready to execute tasks, or transport orders, as they are called in OpenTCS.
This can be done, differently from OpenRMF, where everything is done via terminal, through the Plant Overview client when in Operating Mode, exploiting the toolbar at the top of the GUI.
Selecting the option *Create Transport Order*, it is possible to command the vehicles to execute tasks by just selecting the location they have to get to. Then, the location type will automatically tell the robots which kind of operation they have to perform.
The delivery tasks, as performed in OpenRMF, for instance, is done by generating a Transport Order that consists in two operations, a *LOAD* one and an *UNLOAD* one.

4.6 Simulation

In order to evaluate the performance of the OpenRMF framework and OpenTCS for the management of a robot multi-fleet simulated mining environment, the applications developed for this thesis work have been tested several times through different simulations. Each simulation differs from the other ones for number of robots and for executed tasks, putting in evidence many aspects of the behaviours of OpenRMF and OpenTCS, that will be discussed in detail in Chapter 5.
Each of the following simulations have been repeated five times, to assess if the results produced are the same or can vary, and if so, how much they can vary.
For the sake of clarity, using OpenTCS it is not possible to control multiple robot fleets at the same time, since it is a framework designed specifically for single-fleet systems. Said that, while using OpenTCS as fleet management system, each robot fleet can operate only after the previous robot fleet is done executing all the tasks and all the robots have come back to the their initial positions.

The following subsections will specify more in detail how the simulations were performed.

simulations 1.1 to 1.5

- Number of robots: 3 robots.
- Tasks: 2 delivery tasks and 1 clean task.
  - First delivery task: *L_5_3_in* as pickup location and *U_0* as dropoff location.
  - Second delivery task: *L_6_3_in* as pickup location and *U_0* as dropoff location.
  - Clean task: lane 5 to be cleaned.
- Robots: *tb1*, *tb2*(burger) and *tw1*(waffle).
  - *tb1* executes the first delivery, spawned at *E_1*.
  - *tb2* executes the second delivery, spawned at *E_2*.
  - *tw1* executes the cleaning, spawned at *E_9*.

simulations 2.1 to 2.5

- Number of robots: 3 robots.
- Tasks: 4 delivery tasks and 1 clean task.
  - First delivery task: *L_5_3_in* as pickup location and *U_0* as dropoff location.
  - Second delivery task: *L_6_3_in* as pickup location and *U_0* as dropoff location.
  - Third delivery task: *L_5_4_in* as pickup location and *U_0* as dropoff location.
  - Fourth delivery task: *L_6_4_in* as pickup location and *U_0* as dropoff location.
  - Clean task: lane 5 to be cleaned.
- Robots: *tb1*, *tb2*(burger), and *tw1*(waffle).
  - *tb1* executes the first and the third delivery, spawned at *E_1*.
  - *tb2* executes the second and the fourth delivery, spawned at *E_2*.
  - *tw1* executes the cleaning, spawned at *E_9*.

simulations 3.1 to 3.5

- Number of robots: 4 robots.
- Tasks: 2 delivery tasks and 2 clean tasks.
  - First delivery task: *L_5_3_in* as pickup location and *U_0* as dropoff location.
  - Second delivery task: *L_6_3_in* as pickup location and *U_0* as dropoff location.
  - First clean task: lane 5 to be cleaned.
  - Second clean task: lane 6 to be cleaned.
- Robots: *tb1*, *tb2* (burger), *tw1* and *tw2*(waffle).
  - *tb1* executes the first delivery, spawned at *E_1*.
  - *tb2* executes the second delivery, spawned at *E_2*.
  - *tw1* executes the first cleaning, spawned at *E_9*.
  - *tw2* executes the second cleaning, spawned at *E_8*.

simulations 4.1 to 4.5

- Number of robots: 6 robots.
- Tasks: 4 delivery tasks and 2 clean tasks.
  - First delivery task: *L_5_3_in* as pickup location and *U_0* as dropoff location.
  - Second delivery task: *L_6_3_in* as pickup location and *U_0* as dropoff location.
  - Third delivery task: *L_2_3_in* as pickup location and *U_3* as dropoff location.
  - Fourth delivery task: *L_1_3_in* as pickup location and *U_3* as dropoff location.
  - First clean task: lane 5 to be cleaned.
  - Second clean task: lane 2 to be cleaned.
- Robots: *tb1*, *tb2*, *tb3*, *tb4* (burger), *tw1* and *tw2*(waffle).
  - *tb1* executes the first delivery, spawned at *E_1*.
  - *tb2* executes the second delivery, spawned at *E_2*.

  - *tb3* executes the third delivery, spawned at *E_9*.
  - *tb4* executes the fourth delivery, spawned at *E_8*.
  - *tw1* executes the first cleaning, spawned at *E_3*.
  - *tw2* executes the second cleaning, spawned at *E_4*.

The following simulations instead, were performed only using OpenRMF, to test its performances at solving complex conflicts, while the number of robots gradually increases. These simulations indeed aimed at putting in evidence that the behaviour of OpenRMF gets affected by the number of robots and the complexity of conflicts, a thing that instead does not affect the performances of OpenTCS. For these simulations, the original navigation graph has been slightly modified, creating a new waypoint *U*, at the Crushing Station, connecting the waypoints *U_0*, *U_1*, *U_2* and *U_3*. All the robots that perform deliveries have the *U* waypoint as dropoff point, that is also the waypoint to be cleaned.

simulations 5.1 to 5.5

- Number of robots: 2 robots.
- Tasks: 1 delivery task and 1 clean task.
- Robots: *tb1*,(burger) and *tw1*(waffle)
  - *tb1* executes the delivery, spawned at *L_5_10*; pickup point at *L_5_3_in*.
  - *tw1* executes the cleaning, spawned at *L_4_10*.

simulations 6.1 to 6.5

- Number of robots: 3 robots.
- Tasks: 2 delivery tasks and 1 clean task.
- Robots: *tb1*, *tb2*(burger) and *tw1*(waffle)
  - *tb1* executes the first delivery, spawned at *L_5_10*; pickup point at *L_5_3_in*.
  - *tb2* executes the second delivery, spawned at *L_3_10*; pickup point at *L_3_3_in*.
  - *tw1* executes the cleaning, spawned at *L_4_10*.

simulations 7.1 to 7.5

- Number of robots: 4 robots.
- Tasks: 3 delivery tasks and 1 clean task.
- Robots: *tb1*, *tb2*, *tb3*(burger) and *tw1*(waffle)
  - *tb1* executes the first delivery, spawned at *L_5_10*; pickup point at *L_5_3_in*.
  - *tb2* executes the second delivery, spawned at *L_3_10*; pickup point at *L_3_3_in*.
  - *tb3* executes the third delivery, spawned at *L_6_10*; pickup point at *L_6_3_in*.
  - *tw1* executes the cleaning, spawned at *L_4_10*.

simulations 8.1 to 8.5

- Number of robots: 5 robots.
- Tasks: 4 delivery tasks and 1 clean task.
- Robots: *tb1*, *tb2*, *tb3*, *tb4*(burger) and *tw1*(waffle)
  - *tb1* executes the first delivery, spawned at *L_5_10*; pickup point at *L_5_3_in*.
  - *tb2* executes the second delivery, spawned at *L_3_10*; pickup point at *L_3_3_in*.
  - *tb3* executes the third delivery, spawned at *L_6_10*; pickup point at *L_6_3_in*.
  - *tb2* executes the third delivery, spawned at *L_2_10*; pickup point at *L_2_3_in*.
  - *tw1* executes the cleaning, spawned at *L_4_10*.

Chapter 5

Results

In light of the results gotten from the simulations mentioned in Chapter 5, the way OpenRMF and OpenTCS address the problem of managing and controlling a multi-fleet robot system in a mining environment resulted to be remarkably different, making the two applications best applicable for different use cases and for different purposes.
This section will delve more in detail into the performance of the two applications, showing what are the pros and the cons of both the frameworks, if they can address more general issues and their implementative limitations.

5.1 Predictability

One of the first important aspects that emerged is related to the predictability of the results, that is, how much predictable are the results produced by either OpenRMF and OpenTCS, or in other words, how much the outcomes of the same simulation can differ among them. Specifically, the execution time recorded while using OpenRMF exposed a high variance, way bigger than the one gotten using OpenTCS as fleet management system.
Furthermore, the sequence of execution of tasks shows pretty good this behaviour: while OpenTCS always responds producing the same sequences, the ones produced by OpenRMF differ in every case, leading not only to a very variable total execution time, but also to even more variable finishing times for each task. This aspect can be observed in the graphs from Figure 5.1 to Figure 5.6, showing the outputs produced by OpenTCS for the simulations 1.1 to 1.3 and 2.1 to 2.3, and from the graphs from Figure 5.7 to Figure 5.12, showing the outputs produced by OpenRMF for the same simulations.

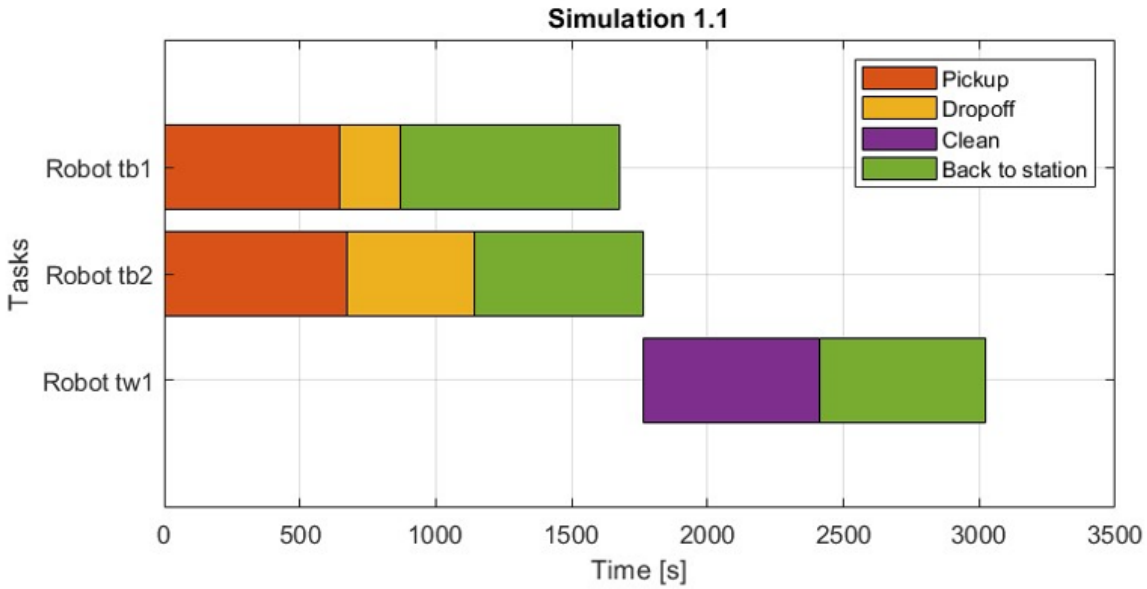

Figure 5.1: Simulation 1.1 with OpenTCS

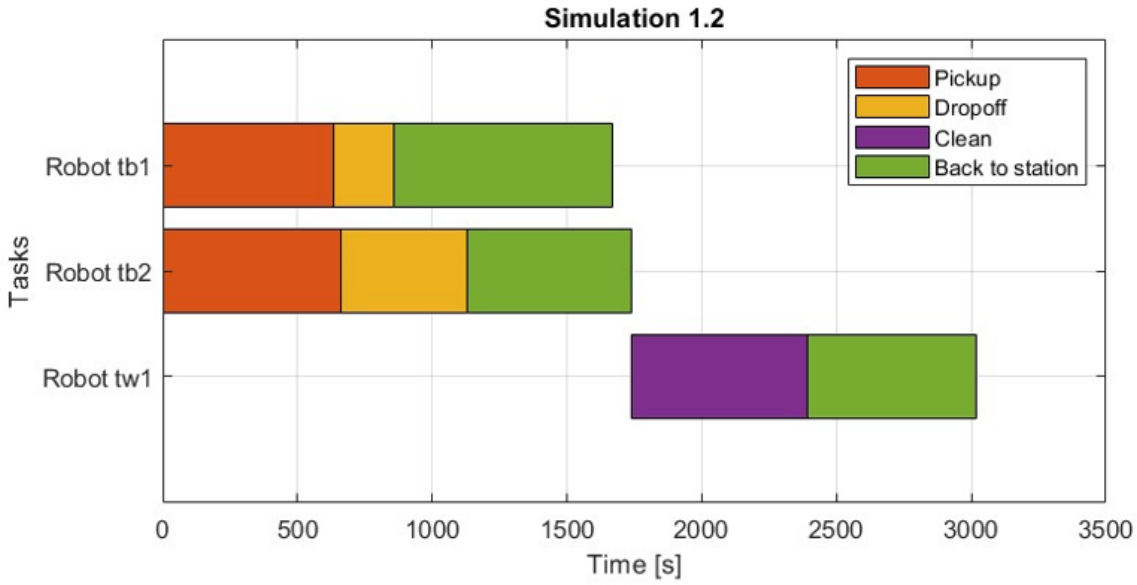

Figure 5.2: Simulation 1.2 with OpenTCS

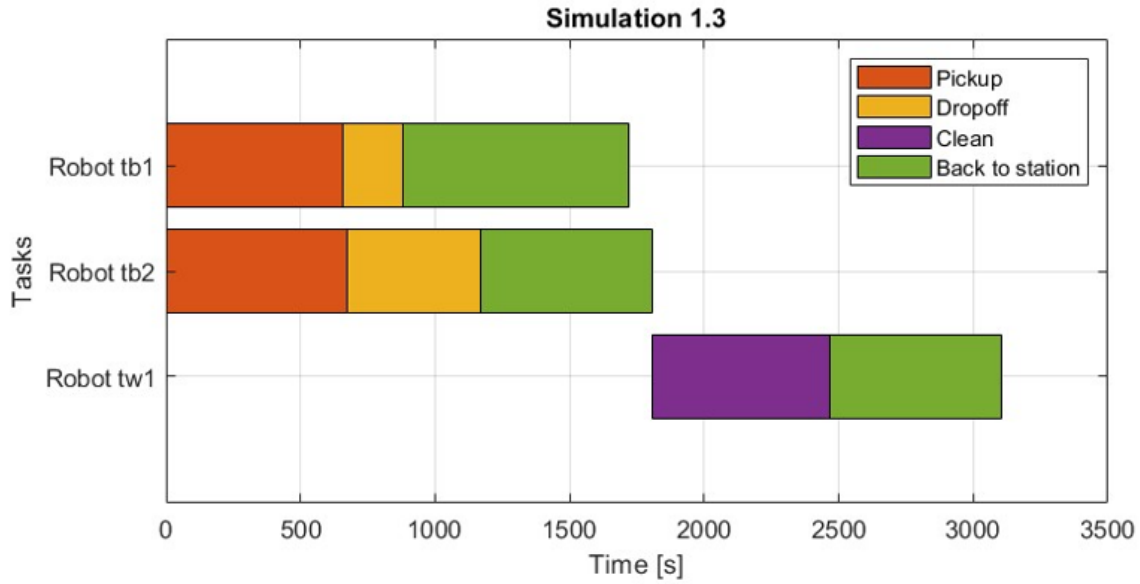

Figure 5.3: Simulation 1.3 with OpenTCS

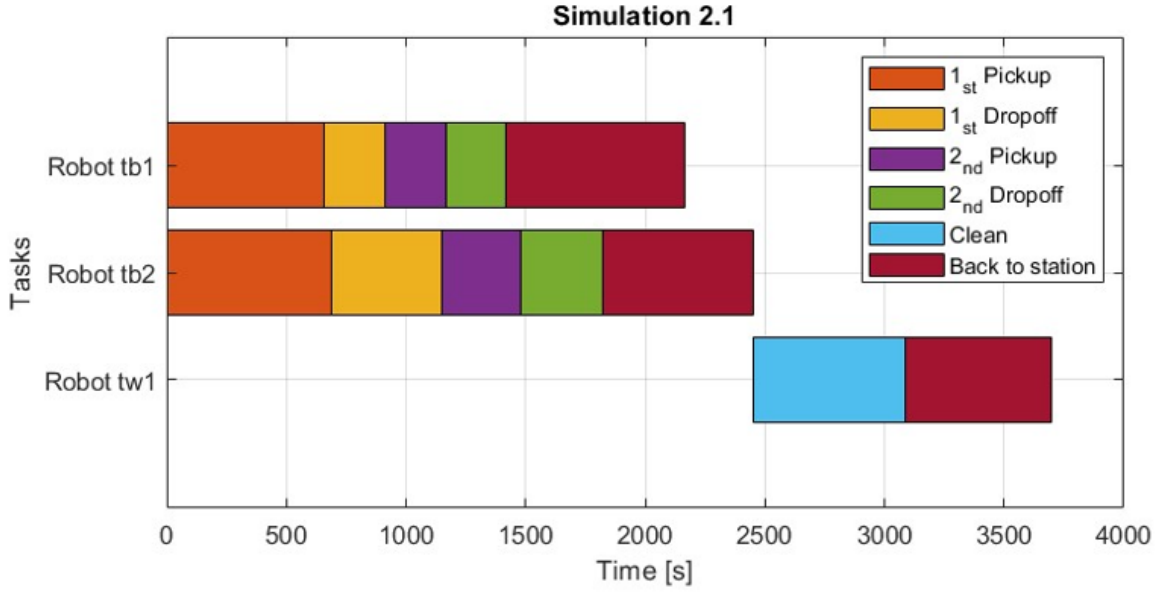

Figure 5.4: Simulation 3.1 with OpenTCS

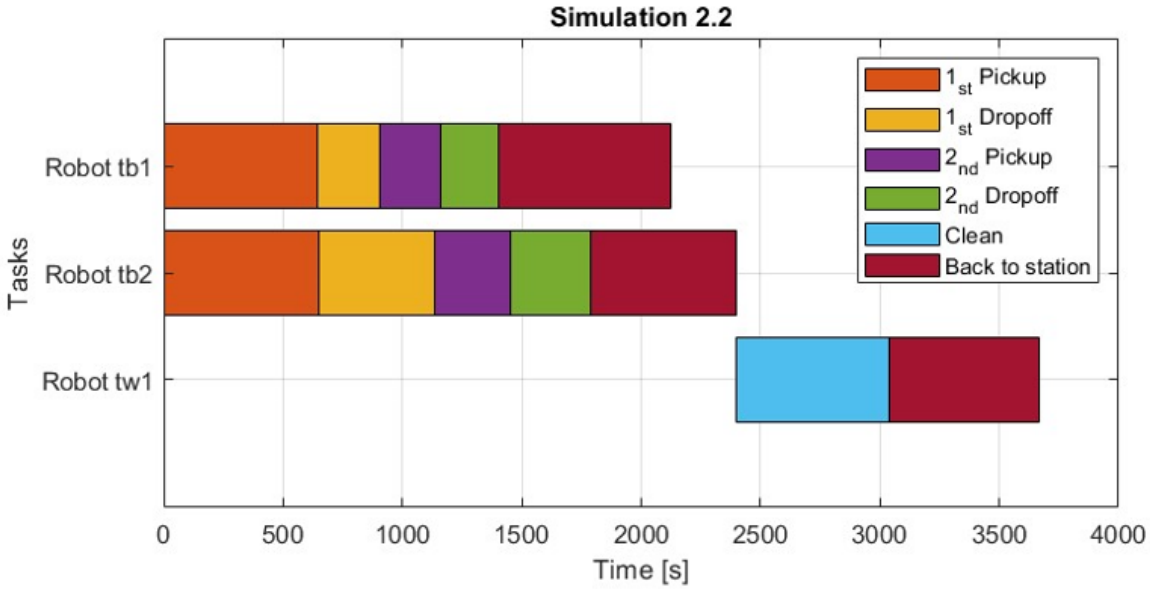

Figure 5.5: Simulation 3.2 with OpenTCS

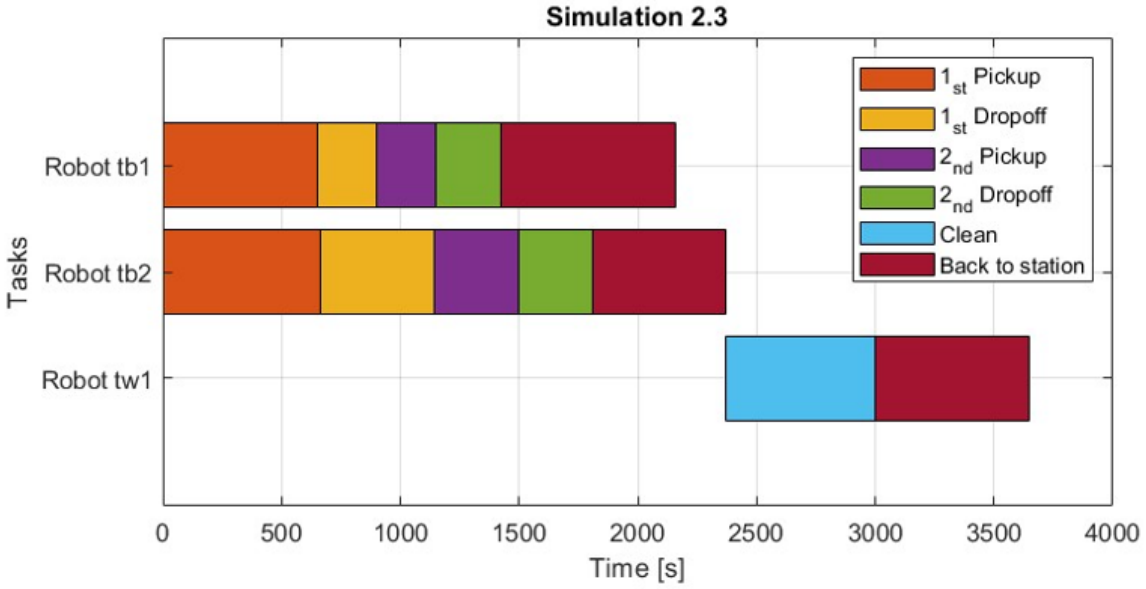

Figure 5.6: Simulation 3.3 with OpenTCS

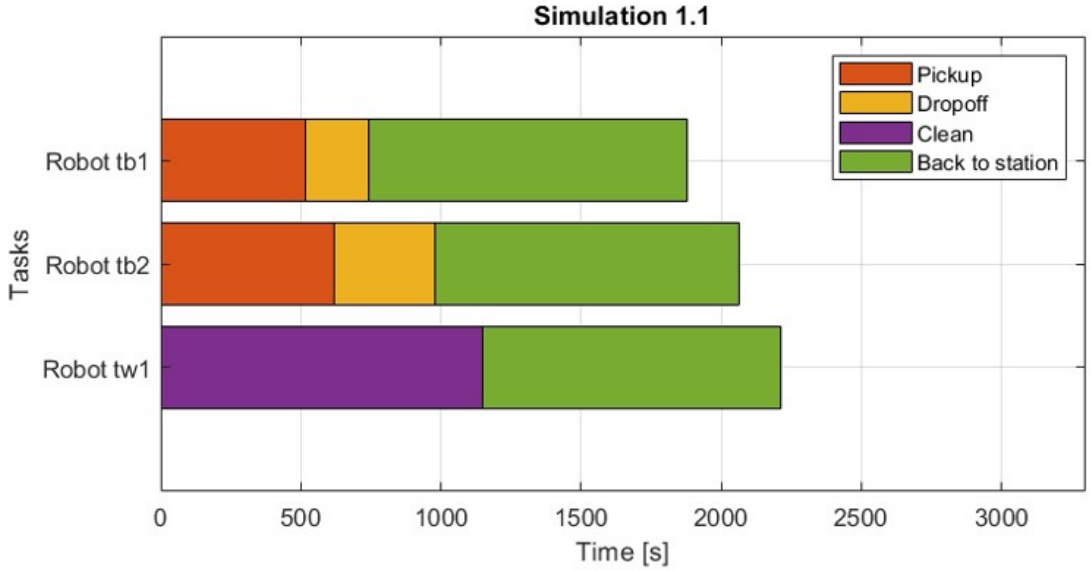

Figure 5.7: Simulation 1.1 with OpenRMF

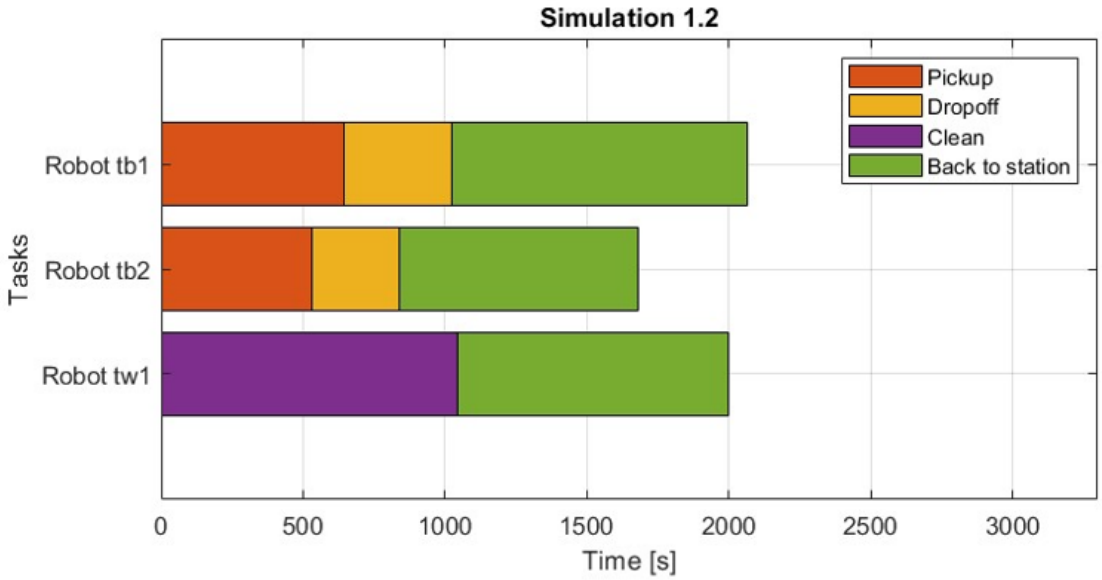

Figure 5.8: Simulation 1.2 with OpenRMF

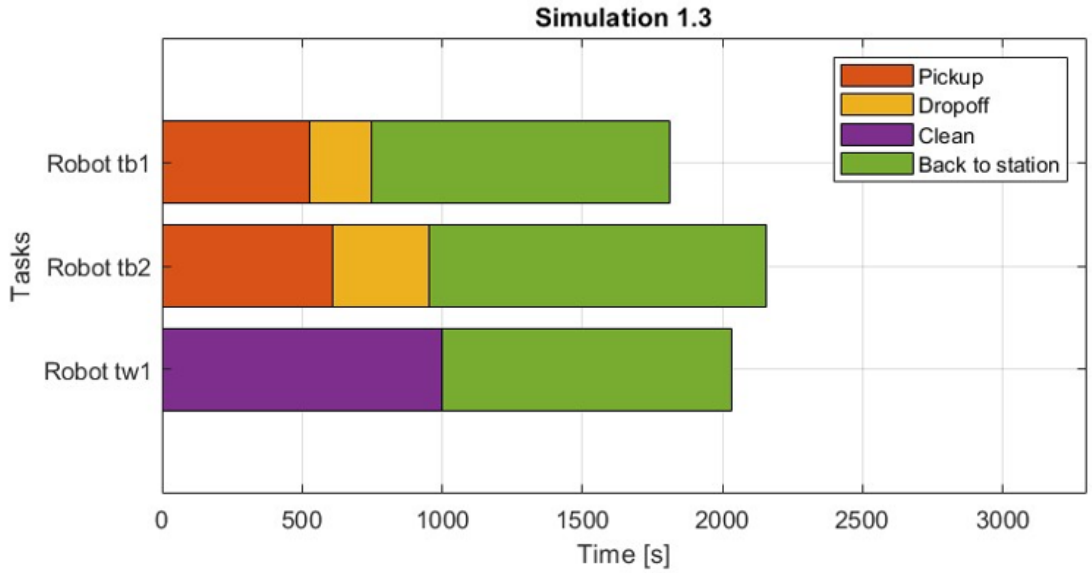

Figure 5.9: Simulation 1.3 with OpenRMF

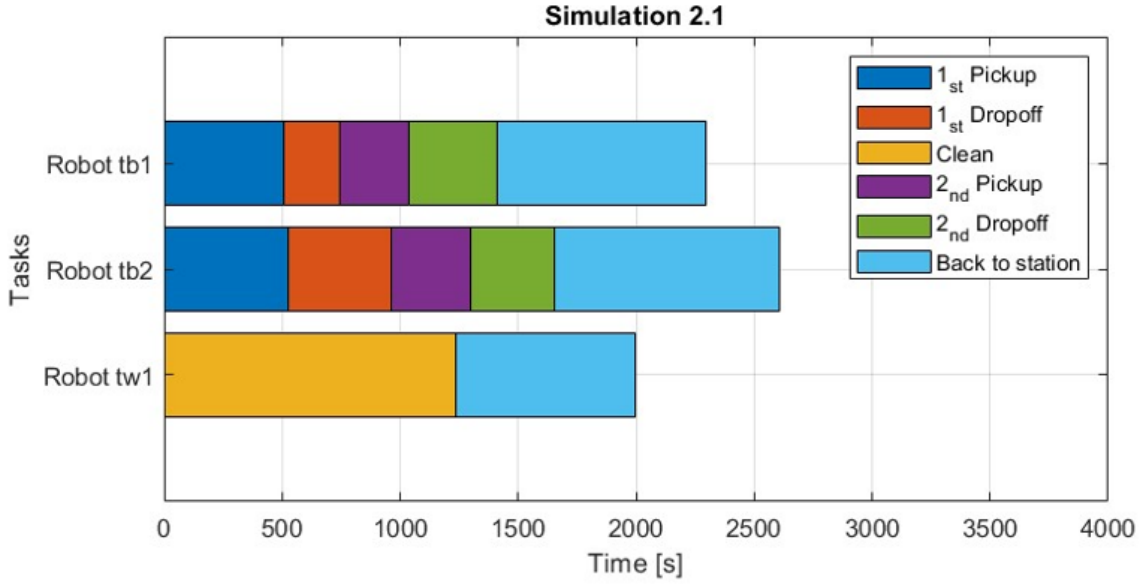

Figure 5.10: Simulation 3.1 with OpenRMF

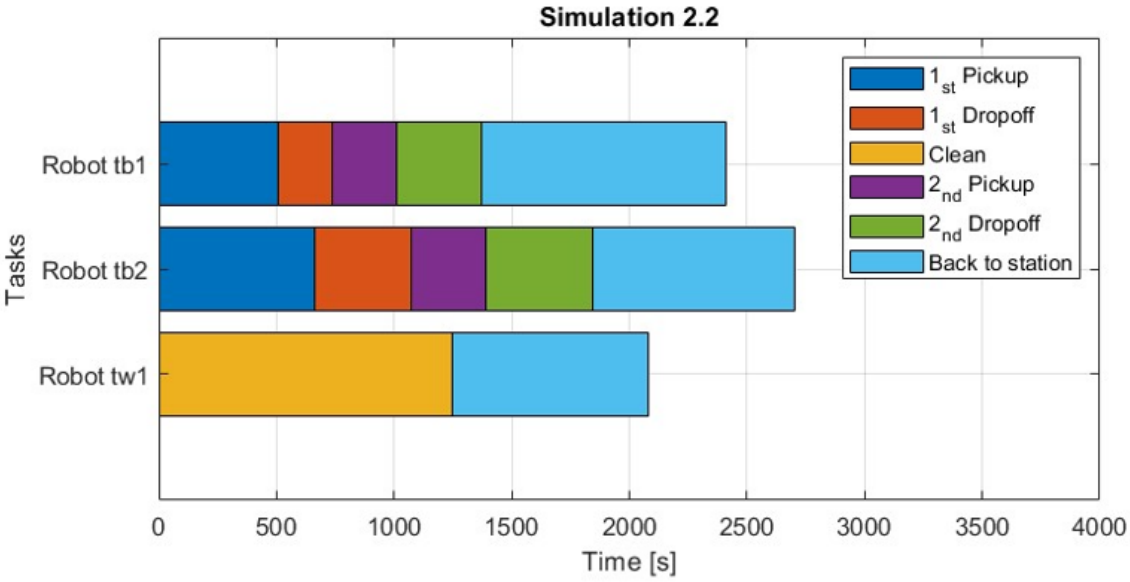

Figure 5.11: Simulation 3.2 with OpenRMF

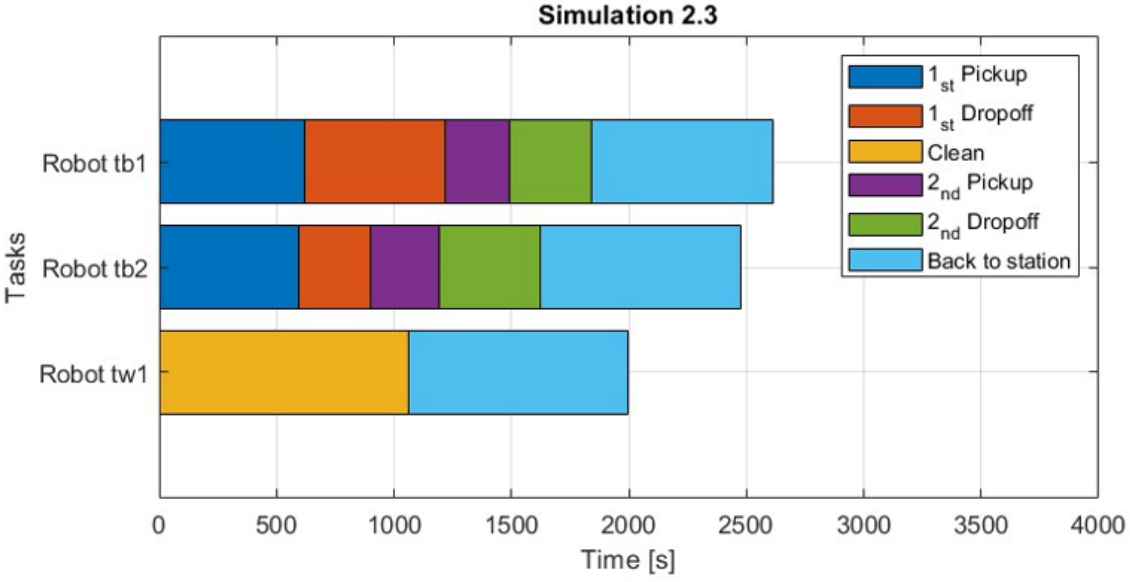

Figure 5.12: Simulation 3.3 with OpenRMF

As evident from the previous figures, the sequences of the task finishing times produced by OpenRMF are always different, proving that OpenTCS has a way more expectable behaviour in terms of planning the paths the robots will have to follow.
While OpenRMF indeed generates the routes of the robots by computing a first estimation and by correcting it in case of conflicts, OpenTCS directly generates the optimal paths: this is the reason why OpenRMF presents less predictable results, behaviour that is intrinsic to its same working principle and that cannot be adjusted.

5.2 Single task execution and total cycle time

Another important parameter took into account to evaluate the performances of the two applications is the finishing time resulting from the execution of a task or a sequence of tasks. Specifically, it has been analyzed the total cycle time of a complete simulation, thus considering the time spent to perform tasks combined with the time wasted for solving conflicts, and the execution time of a single task, to observe more in depth how OpenRMF and OpenTCS can manage it.
For what concerns the execution of a single task, surprisingly, it has been noticed that the finishing time provided by OpenRMF is way lower than the one provided by OpenTCS.
In order to assess this, each task has been executed singularly various times and the different finishing times have been collected, so to establish a mean value for the two frameworks. From this observations, OpenTCS resulted to provide execution times about 22.86% longer than the ones provided by OpenRMF, even though the paths, the robots and the tasks are exactly the same. This is a huge amount of time wasted, especially if applied to situations in which the execution of a task is a very long process. This behaviour is better put in evidence in Figure 5.13, where it is shown the mean value of the execution of a single task using both the frameworks: specifically this example is based on the execution of the pickup task performed by the robot *tb1*.
This behaviour cannot be caused by how the two frameworks manage conflicts, since the single task executions have been performed by ensuring that the robot executing the task was the only one in the environment, thus making not possible to have any kind of conflict in this scenario.
It appears to be caused instead by how OpenTCS and OpenRMF control the robots. Using the OpenTCS fleet management system, when the vehicles reach a new point in the map, the framework wastes a bit of time computing the next move and sending the next navigation goal to the robots. The amount of time wasted at a point might appear small, but the problem comes from the accumulation of

delay when the number of waypoints included in the path, from the robots position to the position where the task has to be executed, is big. Considering a simple pickup task in the example of Figure 5.13, for instance, the number of waypoints the robot has to pass through is 26; anyway this number gets way lager (more than the double) for a complete delivery task (pickup and dropoff) and considering the path to come back to the Workshop, making the delay issue even more evident. This issue does not affect instead the performance when using OpenRMF, since, even though the navigation goals are sent waypoint by waypoint as well, they are all computed at the beginning, allowing to save a lot of time.

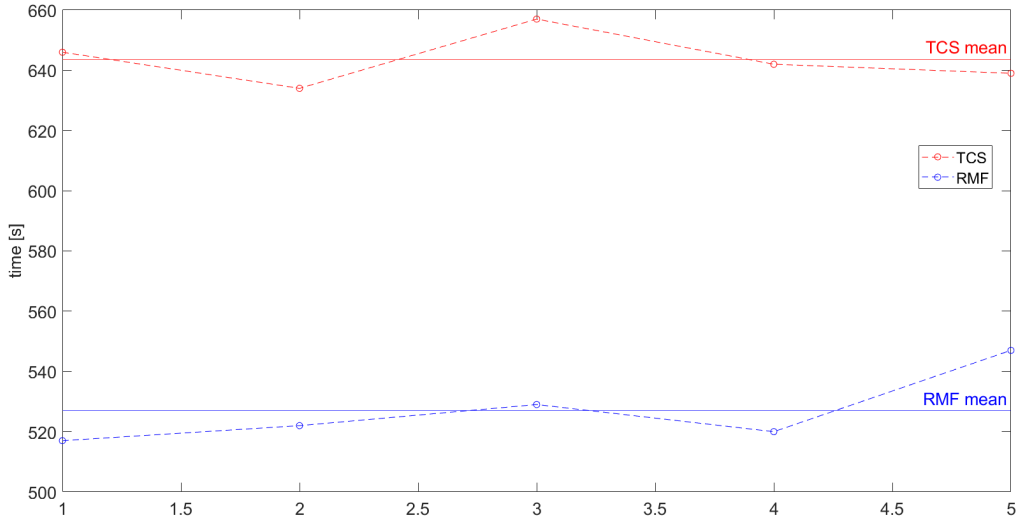

Figure 5.13: OpenRMF vs OpenTCS, single task execution time

For what concerns instead the total cycle time related to execution of a complete simulation, given by a sequence of tasks performed by multiple robots, OpenRMF stands again as the winner. Allowing to perform more tasks at the same time implies the presence of more robots in the environment, that most of the times need to pass through the same paths, contributing to the traffic in the lanes, to the generation of complex conflicts, and increasing the amount of time the robots have to spend waiting for the lanes to be free. Anyway, OpenRMF appears to manage the issue sufficiently well, at least if the number of robots does not increase too much, effectively reducing the total execution time, making the solution provided by this framework definitely better with respect to the one provided by OpenTCS.

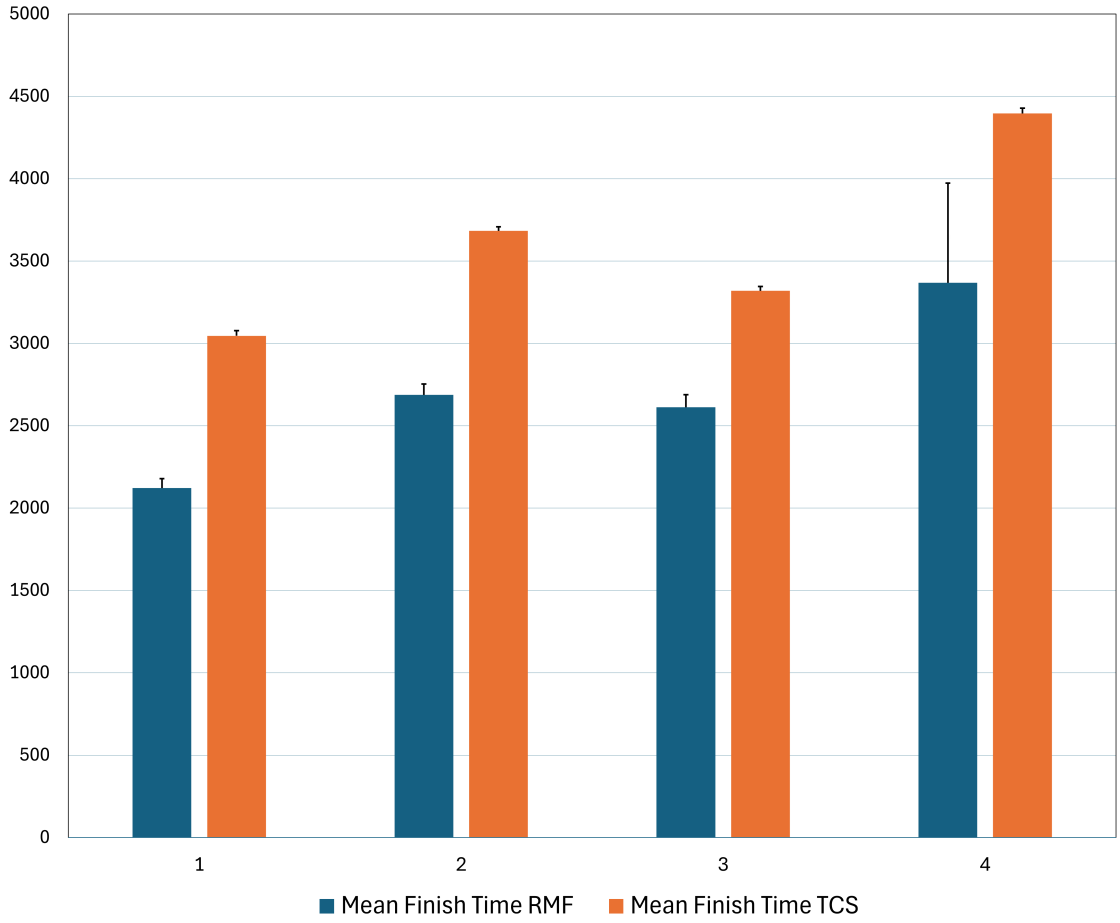

Figure 5.14: Average total cyle times provided by OpenRMF and OpenTCS for the simulations of the groups 1, 2, 3 and 4.

Figure 5.14 shows the average total cycle times needed by the two applications to complete the entire simulation, for the simulations of the groups 1, 2, 3 and 4. Furthermore, puts in evidence the variances, confirming what said in the previous subsection: the finishing times provided by OpenRMF show a definitely higher variance if compared to OpenTCS.
Unfortunately, as it will be explained in the following subsection, OpenRMF appears to be unable to manage cases in which the number of robots present in the environment is bigger than five, thing that did not allow to verify if the performance are still better with respect to OpenTCS in case of more intense traffic conditions and more complex conflicts. This is particularly evident observing the left hand side of the fourth column in Figure 5.14: each of the simulations of the group 4, performed with six robots, ended with a crash at very different times, explaining the high variance in the figure. On the other hand, OpenTCS did not

get affected by the number of robots, yielding stable results both in case of few or many machines in the environment.

5.3 Conflict resolution

From the simulations performed, it resulted evident that both OpenTCS and OpenRMF have limitations reagarding conflict resolutions: they address the problem in two opposite ways, showing to be more suitable for some use cases rather than others. The following paragraphs explain better in detail this issue.

5.3.1 OpenTCS

The results provided by OpenTCS confirm its predictability in the resolution of conflicts: each conflict indeed is solved always the same way, and always in the optimal manner, that is saying that all the conflicts are addressed with the minimum number of movements and making the robots follow the shortest path to free the lanes they are occupying, if other robots need to enter the same space.
This is because OpenTCS calculates all the routes before the execution of a task, ensuring they are the optimal ones; on the other side, OpenRMF calculates just an initial estimation, eventually replaced with a new one in case of conflict, making the route computed, with a non negligible probability, not optimal.
Anyway, in the context of conflict resolution, OpenTCS shows a heavy limitation: it is not always able to solve every conflict, thing that OpenRMF, even if not optimally, is able to do. Specifically, this might happen in case the conflict in question is generated at an intersection of the type shown in Figure 5.15, consisting in a graph with a pending leaf, if the tasks to be executed are allocated at different times.

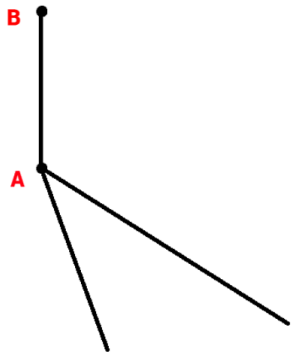

Figure 5.15: Intersection 1

This happens because OpenTCS, differently from OpenRMF, is not provided

with a dynamic planning feature, that is the ability to reschedule the paths, and while computing the routes it just considers the ready tasks and not the queued ones as well.
The following example better explain the situation in which this problem can arise:

1. $Task_1$ and $Task_2$ start to be executed, respectively by $Machine_1$ and $Machine_2$ and all the paths are generated optimally so not to have any conflict. They both head to point A in Figure 5.15.

2. $Task_1$ finishes its execution at point A, and immediately after, $Task_3$, starts to be executed by $Machine_1$ at point A, while $Machine_2$ reaches point B.

3. $Task_3$ generates a conflict with $Task_2$, that has not finished its execution yet.

4. The only way to solve this conflict is by changing the route for $Task_2$, that as previously said cannot be modified: there is a task abort.

This issue can be solved by slightly changing the paths of the environment, leading to the solution depicted in Figure 5.16: this way it is avoided that a robot gets stuck in a point without any free path to occupy to solve the conflict.
Unfortunately, even if this solution can be applied for this specific mine layout it is not generally valid and cannot be applied to every case.

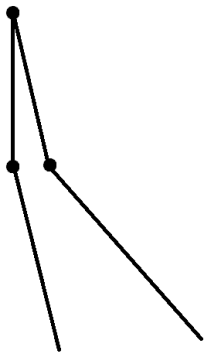

Figure 5.16: Intersection 2

5.3.2 OpenRMF

OpenRMF instead, given its ability to perform dynamic planning and to modify the routes previously computed, it is always able to solve any kind of conflict, given

any kind of navigation graph, even though not necessarily through the minimum number of movements or computing the shortest paths.
Anyway, this framework instead appears to be affected by another issue, that is the number of robots present in the environment, letting the performance gradually degrade as this number increases.
As the number of robots grows, OpenRMF calculates the routes with more and more difficulty and delay, making the resolution of a conflict, even if simple, gradually more and more different from the optimal one. This situation can worsen till provoking a crash between the robots, a situation which it is proven that can happen in case of six robots or more.
This is due to the fact that OpenRMF recognizes two types of position associated to a robot, that are the real position and the estimated one, represented in Figure 5.17 through, respectively, the pink marker and the green marker.

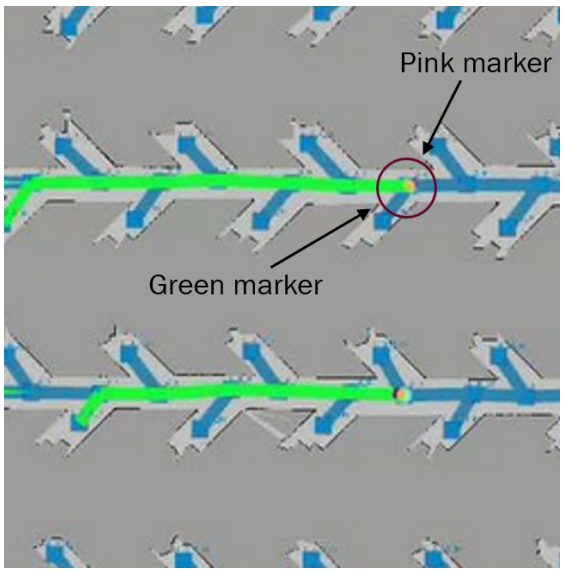

Figure 5.17: Overlap of the pink and green marker (pink marker: real position; green marker: estimated position; green path: the route each robot is following)

Said that, the two markers should be constantly approximately overlapped, meaning that the estimations OpenRMF computes represent the real positions.
Anyway, when the number of robots increases, OpenRMF appears to compute the estimations with more and more delay, degrading the quality of the calculations and making it not correspond to the reality. The consequences of this behaviour are shown in Figure 5.18, where is possible to notice how the distance betweem the pink marker and the green marker gets bigger and bigger as the number of robots in the environment increases.
Furthermore, when taking decisions about the path to compute and the necessity

to solve eventual conflicts, the core systems takes these decisions performing all the needed calculations on the estimated position of the robots instead of the real one, that is saying that OpenRMF decides based on the wrong coordinates, thinking they are the correct ones instead.

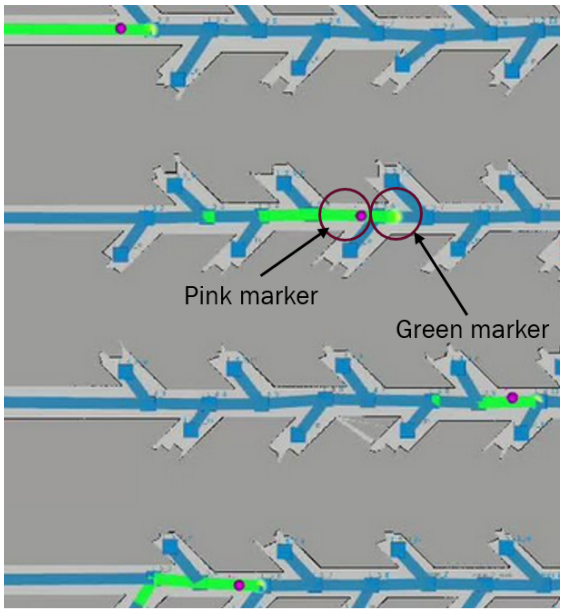

Figure 5.18: Difference between the position of the pink marker and the green one

From the simulation it has been proven that if the difference between the two marker is lower than the distance between two consecutive waypoints, even though with some delay, the core system is still able to solve properly any conflict and calculate correctly the paths the robots have to follow.
Once instead this difference gets bigger than the distance between two consecutive waypoints, OpenRMF still considers the robot as if it was occupying the wrong waypoint, computing the paths to solve eventual conflicts on the wrong data. A situation like this one is shown in Figure 5.19, where it is shown a crash between two robots.
This important issue constrains to prefer OpenTCS as control system in case many robots have to be controlled in the same environment, in order to avoid eventual crashes due to the increasing computational complexity.

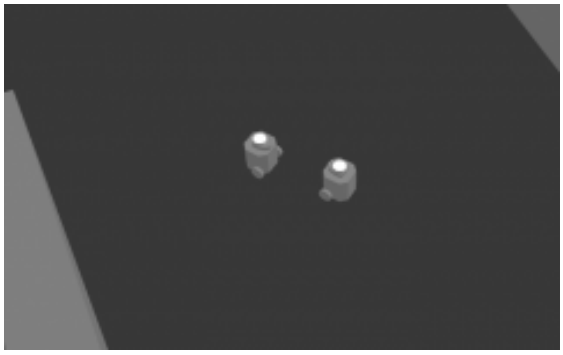

Figure 5.19: Crash between burger tb1 and burger tb3.

5.4 Time wasted in conflict resolution

The last aspect to be analyzed in this thesis work is how the two frameworks address the problem of conflict resolution in terms of time needed.
The number of robots indeed has been proven to not only affect the abilty of OpenRMF to resolve conflicts, but also the time it needs. Considering a simulation with OpenRMF, the amount of time wasted in the resolution of a conflict grows more and more with the increment of the number of robots with respect to the total time of the entire simulation. This behaviour is shown in Figure 5.21, where the portion of time needed to solve a conflict is shown in a cake diagram, while Figure 5.22 shows the increment trend. In particular, the time for conflict resolution appears to grow following a quasi-linear behaviour: anyway, since the maximum amount of robots tolerated before having a crash was five, it was not possible to perform any simulation having six or more robots in the environment.
Specifically, the simulations from 5.1 to 8.5 have been exploited to prove this behaviour, making all the robots pass through the very same cross at the same time, provoking the generation of many and very complex conflicts.
OpenTCS, on the other hand, resulted to be insensitive to how many machines are spawned in the simulated environment, keeping the performance and the timing for conflict resolution always the same.

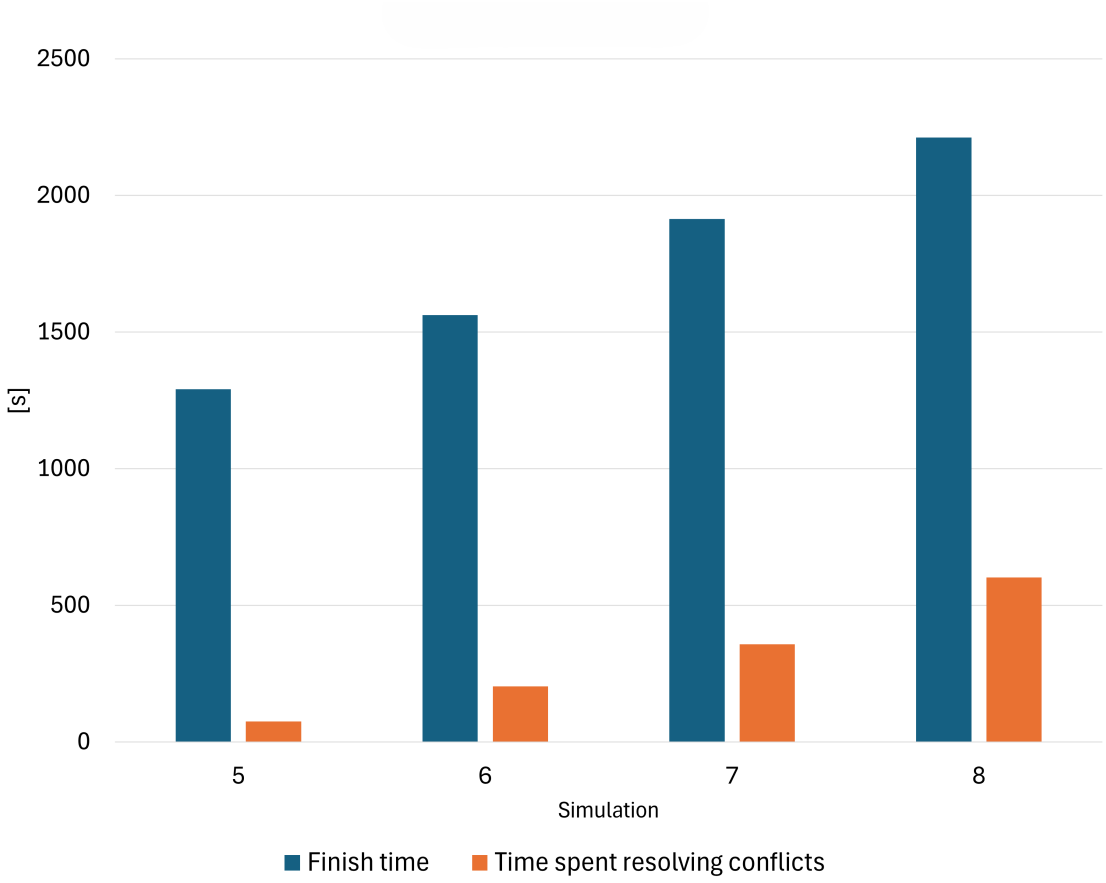

Figure 5.20: Mean total cycle time vs time needed to solve conflicts

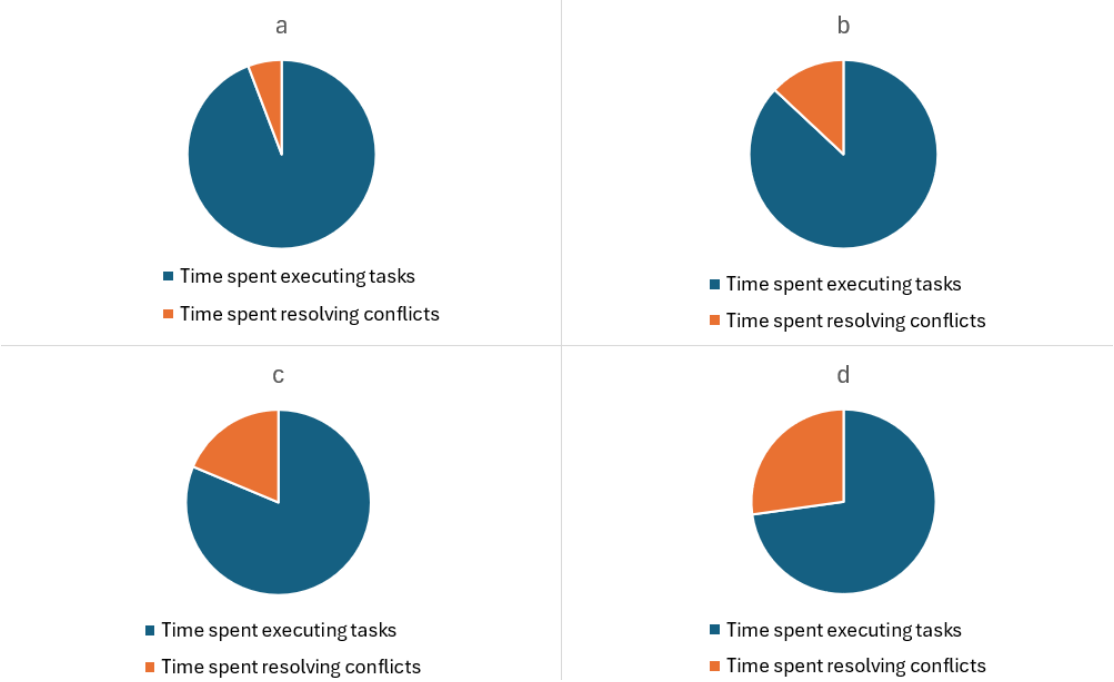

Figure 5.21: Time wasted solving conflicts with respect to the total time of the simulation (a: 2 robots, b: 3 robots, c: 4 robots, d:5 robots)

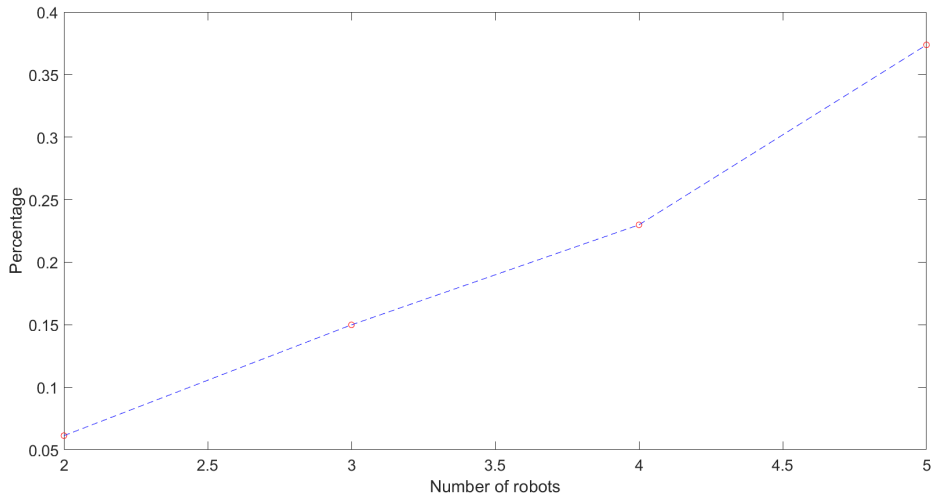

Figure 5.22: Percentage of time wasted resolving the conflicts over the entire execution

Chapter 6

Conclusions and future developments

Through the development of these two applications that respectively rely on OpenRMF and OpenTCS, and after evaluating them through simulations, it is possible to say that there is no absolute winner between the two frameworks when controlling multi fleet systems.
They both provide some advantages and limitations that make them applicable for different use cases.
The first thing immediately noticed in which the performances of the two applications differ, is for sure the predictability. OpenTCS, independently from the number of machines in the environment or the general computational complexity, in this case resulted way more expectable than OpenRMF, that instead has provided very different and unpredictable results.
Anyway, if the predictability is a good factor over which evaluate the performance of an application, for sure it is not the only one, and many other parameters have to be taken into account. The fact that OpenTCS could not solve conflicts of a specific type, for instance, is an even worse limitation, since it implies that the simulation could not be performed at all. This limitation becomes even bigger if there is no way to modify the paths over which OpenTCS calculates the routes, that could be for layout and scheduling reasons, so to make the eventuality of having a prohibited conflict disappear: if this is the case, OpenTCS cannot be chosen as fleet management system. Opposite behaviour instead is the one shown by the OpenRMF framework. It indeed resulted to be a noticed in which the performances of the two applications differ, is for sure the predictability. OpenTCS, independently from the number of machines in the environment or the general computational complexity, in this case resulted way more expectable than OpenRMF, that instead has provided very different and unpredictable results.

Anyway, if the predictability is a good factor over which evaluate the performance of an application, for sure it is not the only one, and many other parameters have to be taken into account.
The fact that OpenTCS could not solve conflicts of a specific type, for instance, is an even worse limitation, since it implies that the simulation could not be performed at all. This limitation becomes even bigger if there is no way to modify the paths over which OpenTCS calculates the routes, that could be for layout and scheduling reasons, so to make the eventuality of having a prohibited conflict disappear: if this is the case, OpenTCS cannot be chosen as fleet management system.
Opposite behaviour instead is the one shown by the OpenRMF framework. It indeed resulted to be able to solve any kind of conflict, independently from the type of conflict or its complexity: soon or later, it will find a way to solve it, even though not optimally as OpenTCS can.
Anyway, the number of robots in the environment plays a crucial role, since it wildly affects the results leading to a huge degradation of the performances, till provoking the robots to eventually crash. This suggests that OpenRMF might be preferred in case the number of robots in the environment is not relatively high, to guarantee that it is correctly functioning.
Lastly, the initial question that led to the development of this thesis work: is it possible to reduce the total cycle time by letting all the tasks be performed simultaneously instead of being executed in separate time windows?
The answer to this question is then yes, if the number of robots in the environment does not pass a threshold. It has been proven indeed, that despite the worsening of the traffic conditions in the mine and the augmented complexity of the conflicts that will be generated, OpenRMF is still able to provide always better results, in terms of time, with respect to OpenTCS. At least this is valid till the number of robots is greater or equal to six: in this case the computational complexity makes OpenRMF unable to follow the resolution and the generation of new conflicts, allowing the robots to crash. Hence, no meaningful results could be collected, and so it is not possible to say that this solution is better than OpenTCS in general.
Hence, by this analysis in few words it resulted that:

1. OpenRMF solves any kind of conflict while OpenTCS does not.
2. The cycle time with OpenRMF is lower than OpenTCS in case of single task execution and up to six robots in the environment.
3. OpenTCS yields results with a lower variance and characterized by always the same sequence of tasks to be executed with respect to the ones provided by OpenRMF, resulting more predictable.
4. OpenTCS, if able to solve a conflict, always solves it in the optimal way, hence through the minimum number of moves and computing the shortest paths.

This is not true for OpenRMF, especially if the number of robots increases.

It could be possible though to make some modifications to the core code of the frameworks, and to the libraries they exploit to see if there still is room for improvements, and if the limitations explained previously can be overcome.
First of all, an interesting study could be to let OpenRMF compute the routes the robots have to follow, in order to solve conflict, using the robots real positions instead of the estimated ones. This for example, could potentially be a solution to the problem given by the increasing difference between the estimated positions and the real ones that, as explained, causes the robots to crash.
Alternatively, since one of the key advantages of OpenTCS is its predictability and the fact that always decides optimally, an other alternative could be working on this framework and modifying some of its features that make it unusable in some cases.
For instance, it could be interesting to let OpenTCS schedule the paths not only considering the ready tasks, but also the queued ones, as OpenRMF does. This could help with the problem of conflict resolution: if OpenTCS considers the tasks that are not ready yet but are programmed to be executed in a second moment, it could compute the path knowing that potentially a new task could generate a prohibited conflict, making it not happen.
Finally, it could be possible to provide OpenTCS with an inter-fleet interface, making it applicable to control multi-fleet systems. Anyway, this modification is of way greater entity, but it could potentially allow OpenTCS to exploit the advantages OpenRMF has by controlling multiple fleets at the same time while keeping its reliability and optimality features.

Bibliography

[1] Cristofer Daniel Hernandez Larenas. «Sistema de Despacho para Cargadores Frontales de Bajo Perfil en Minerìa Subterranea». In: (2021) (cit. on pp. 4, 5).

[2] Carlos Tampier, Mauricio Mascaró, and Javier Ruiz-del-Solar. «Autonomous Loading System for Load-Haul-Dump (LHD) Machines Used in Underground Mining». In: (2021), Santiago de Chile (cit. on p. 6).

[3] Mauricio Mascaró, Isao Parra-Tsunekawa, Carlos Tampier, and Javier Ruiz-del-Solar. «Topological Navigation and Localization in Tunnels—Application to Autonomous Load-Haul-Dump Vehicles Operating in Underground Mines». In: (2021), Santiago de Chile (cit. on p. 6).

[4] Macenski, Steven, Martin, Francisco, White, Ruffin, Ginés Clavero, and Jonatan. «The Marathon 2: A Navigation System». In: (2020) (cit. on p. 7).

[5] Felipe Inostroza, Isao Parra-Tsunekawa, and Javier Ruiz-del-Solar. «Robust Localization for Underground Mining Vehicles: An Application in a Room and Pillar Mine». In: (2023), Santiago de Chile (cit. on p. 7).

[6] *ros2_documentation/source/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.rst at foxy · ros2/ros2_documentation · GitHub*. `https://github.com/ros2/ros2_documentation/blob/foxy/source/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.rst`. Accessed: 2024-09-10. 2024 (cit. on p. 11).

[7] *ros2_documentation/source/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.rst at foxy · ros2/ros2_documentation · GitHub*. `https://github.com/ros2/ros2_documentation/blob/foxy/source/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.rst`. Accessed: 2024-09-10. 2024 (cit. on p. 12).

[8] *ros2_documentation/source/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.rst at foxy · ros2/ros2_documentation · GitHub*. https://github.com/ros2/ros2_documentation/blob/foxy/source/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.rst. Accessed: 2024-09-10. 2024 (cit. on p. 13).

[9] *open-rmf · GitHub*. https://github.com/open-rmf. Accessed: 2024-09-10. 2024 (cit. on p. 14).

[10] *RMF Core Overview - Programming Multiple Robots with ROS 2*. https://osrf.github.io/ros2multirobotbook/rmf-core.html. Accessed: 2024-09-10. 2024 (cit. on pp. 17–19).

[11] *Tasks in RMF - Programming Multiple Robots with ROS 2*. https://osrf.github.io/ros2multirobotbook/task.html. Accessed: 2024-09-10. 2024 (cit. on pp. 21–23).

[12] *Traffic Editor - Programming Multiple Robots with ROS 2*. https://osrf.github.io/ros2multirobotbook/traffic-editor.html. Accessed: 2024-09-10. 2024 (cit. on pp. 25, 27).

[13] *GitHub - eclipse-cyclonedds/cyclonedds: Eclipse Cyclone DDS project*. https://github.com/eclipse-cyclonedds/cyclonedds. Accessed: 2024-09-10. 2024 (cit. on p. 29).

[14] *Free Fleet - Programming Multiple Robots with ROS 2*. https://osrf.github.io/ros2multirobotbook/integration_free-fleet.html. Accessed: 2024-09-10. 2024 (cit. on p. 29).

[15] The openTCS developers. *openTCS: User's Guide*. https://www.opentcs.org/docs/5.0/user/opentcs-users-guide.html. Accessed: 2024-09-10. 2024 (cit. on pp. 30, 31, 33, 35, 37–40).

[16] *GitHub - nielstiben/openTCS-NeNa: An open-source ROS 2 vehicle driver for OpenTCS*. https://github.com/nielstiben/openTCS-NeNa. Accessed: 2024-09-10. 2024 (cit. on p. 33).