**Università degli Studi di Verona**

DIPARTIMENTO DI DI INFORMATICA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Tesi di Laurea Magistrale

# Multi-Robot Task Allocation for logistic applications

Candidato:
**Davide Zorzi**
**Matricola vr414572**

Relatore:
**Alessandro Farinelli**

Correlatore:
**Riccardo Muradore**

**Anno Accademico 2017-2018**

# Contents

**Abstract**

Robotics technology has recently matured sufficiently to deploy autonomous robotic systems for daily use in several applications: from disaster response to environmental monitoring and logistics. In this project present and evaluate the principal difference of central and distributed allocator task coordinator. In these applications we address off-line coordination, by casting the Multi-Robot logistics problem as a task assignment problem and proposing three solution techniques: Greedy Strategy Single robot Single task (GS1:1), which is a baseline greedy approach, Set Partition Strategy Single robot Multiple task (SPS1:N) and Coalition Formation Strategy Single robot Multiple task (CFS1:N), which are based on merging task for improve the spend time.
We evaluate the performance of our system in a realistic simulation enviroment (build with ROS and stage). In particular, in the simulated enviroment we compare our task assignment approaches with previously methods mentioned.

**Keywords:** Multi-Robot Task Allocator, logistic applications, Multi-Robot systems, coordination, task assignment

# Chapter 1

# Introduction

One of the fundamental areas in Robotics is multi-robot systems. More particularly, this thesis addresses the cooperation of a team of mobile robots in logistic missions. the main aspects studied herein are strategies for effective logistic performance, agent's coordination, scalability and applicability in real-life situations.
This introductory chapter presents the context of the research in order to clarify the motivation and significance of the problem. In addition, some guidelines about Multi-Robot systems in general and, more specifically, agents in logistic missions are herein introduced to lay the groundwork to approach the problem in hands. Finally, an overview of the document is given.

## 1.1 Context and Motivation

In recent years, robotics has been one of the scientific fields with the most substantial advances. Within the diverse areas that it embraces, mobile robotics has had great focus in the last decades from roboticists (i.e., researchers on robotics) around the world. In particular, issues like autonomous navigation, path planning, self-localization, coordination of robots, cooperative dynamics, mapping, exploration and coverage have become popular and have benefited from the progress of artificial intelligence, control theory, real-time systems, sensors' development, electronics, communication systems and systems integration [Parker, 2008].

Nowadays, we expect to see robots with many different shapes operating in different environments as on land, underwater, in the air, suspended on wires, climbing and so on. This evident growth is extremely motivating for the development and contribution of new developments by the community.
Security applications are a fundamental task with unquestionable impact on society. Combining this fact with the technological evolution observed in the last decades, it becomes clear that robot assistance can be a valuable resource by taking advantage of robots' expendability. In particular, multi-robot allocator task for logistic applications has high utility and is considered as a contemporary area with some relevant work presented in the last decade, especially in terms of strategies for coordinating teams of robots. However, many of the studies in the literature present unrealistic simplifications, strong limitations or questionable applicability as illustrated later on. Therefore, there is an eminent potential to explore in this context.
Moreover, the allocator task for logistic applications problem is very challenging in the context of Multi-Robot systems, because agents must navigate autonomously, coordinate their actions in a distributed or centralized way and acquire information about the surrounding space, possibly with communication constraints and independently of the number of robots in the team and the enviroment's dimension. All of these features lead to an excellent case study in mobile robotics and conclusions drawn from such studies may support the development of future approaches not only in the logistic domain but also in multi-robot systems, in general.

## 1.2 Multi-Robot Systems

In many applications, an autonomous mobile robot equipped with different sensors may adequately complete a given assignment. However, in several situations, it proves to be more expensive, less efficient and less robust than using a multi-robot system. In some cases, due to the need of combining different tasks and the dynamics of the environment.

Some characteristics of multi-robot systems include distributed control, autonomy, communicative agents and greater fault-tolerance. A single robot may be vulnerable to hostile environments or attackers, for example, in mil-

itary actions. In such scenarios, agents would greatly benefit from the assistance of nearby agents during emergencies, failures or malfunctions.

One of the main difficulties when approaching these systems is to coordinate many robots to perform a complex, global task in an efficient manner, maximizing group performance under a wide range of conditions, with the flexibility to take advantage of the resources available, embrace the requirements and constraints imposed and resolve issues like action selection, coherence, conflict resolution and communication. This cannot be done by just increasing the number of robots assigned to a task. A coordination mechanism must exist to establish relationships between agents so that they can accomplish the mission effectively.

## 1.3 The Multi-Robot system for logistic applications

Logistic apllication an infrastructure with multiple robots is no different than other multi-robot assignments, in the sense that it incorporates all the previously mentioned characteristics of Multi-Robot system. To understand this problem, it is important to firstly introduce the definition of logistic application.

**Definition 1.** *Industrial Logistics, the set of operations related to the procurement, destination and storage of materials and products of large industry; the coordination and provisioning of people or things for the purpose of higher production efficiency.*

Many real-world applications of Multi-Robot systems require agents to operate in known common environments. The agents are constantly engaged with new tasks and have to navigate between locations where the tasks need to be executed. On the other hand, the Multi-Robot system for logistic applications, given a set of agents attend to stream of incoming pickup-and-delivery tasks.

# Chapter 2

# Background and Related Works

In this section, we detail the main issues for Multi-Robot system coordination in industrial domains, then we provide a detailed discussion on coordination approaches, highlighting challenges and main solution techniques.

## 2.1 Multi-Robot system for Industrial Applications

In this thesis I also focus on industrial scenarios where robots have a high degree of autonomy and operate in a dynamic environment. In this work, I consider a similar setting where a set or robots are involved in trasportations tasks for logistics. However, I focus on the specific problem of task assignment ...

## 2.2 Coordination in Multi-Robot system

Coordination for Multi-Robot system (MRS) has been investigated from several diverse perspectives and nowadays, there is a wide range of techniques that can be used to orchestrate the actions and movements of robots operationg in the same enviroment. Specifically, the ability to effectively coordinate the actions of a MRS is a key requirement in several applications domains that range from disaster response to environmental monitoring, militaty op-

erations, manufacturing and logistics. In all such domains, coordination has been addressed using various frameworks and techniques and there are several survery papers dedicated to categorize such different approaches and identifying most prominent issues when developing MRS.
Given my focus on logistic scenarios, here I restrict my attention to coordination approaches based on optimization and specifically on task assignment as this the most common framework for my reference application domain.

# Chapter 3

# Problem

In this section I detail my reference scenario for MRS coordination and formalization problem.

## 3.1 Description

My reference scenario is based on a warehouse that stores items of various types. Such items must bo composed together to satisfy orders that arrive based on customers' demand. The items of various types are stored in particular section of the building (*loading bay*) and must be trasported to a set of *unloading bays* where such items are then packed together by human operators. The set of items to be trasported and where they should go depends on the orders. In my domain a set of robots is responsible for for transporting items from the loading bays to the unloading bays and the system goal is to maximize the throughput of the orders, i.e., to maximize the number of orders completed in the unit of time. Now, robots involved in transportation tasks move around the warehouse and are likely to interfere when they move in close proximity, and this can become a major source of inefficiency (e.g., robots must slow down and they might even collide causing serious delays in the system). Hence, a crucial aspect to maintain highly efficient and safe operations is to minimize the possible spatial interferences between robots. Specifically, here we propose to take this interferences into account in the task assignment process and assign tasks to robots so to reduce the possible

interferences among the transportation robots.

## 3.2 Fomalization

In this section I formalize the MRS coordination problem described above as a task allocation problem where the robots must be allocated to transportation tasks. In my formalization if transportation tasks are more than the available robots at each time step only a subset of tasks will be allocated. However, since the task allocation process is repeated over time robots effectively serve a sequence of tasks. In more detail, my model considers a set of items of different types $E = \{e_1, ..., e_N\}$, stored in a specific loading bay ($L$). The warehouse must serve a set of orders $O = \{o_1, ..., o_M\}$. Orders are processed in one or more than one of the unloading bays ($U_i$). Each order is defined by a vector of demand for each item type (the number of required items to close the order). Hence, $o_j =< d_{1,j}, ..., d_{N,j} >$, where $d_{i,j}$ is the demand for order $j$ of items of type $i$. When an order is finished a new one arrives, and we assume to have no knowledge on future orders. The orders induce a set of $N \times M$ trasportation tasks $T = t_{i,j}$, with $t_{i,j} =< d_{i,j}, dst_{i,j}, P_{i,j} >$, where $t_{i,j}$ defines the task of transporting $d_{i,j}$ items of type $i$ for order $o_j$ (hence to unloading bay $U_j$). Each task has a destination bay for centralized coordination the $t_{i,j}$ has a set of edges $P_{i,j}$ which respects the strategy used. I have a set of robot $R = \{r_1, ..., r_K\}$ that can execute transportation tasks, where each robot has a defined load capacity for each item type $C_k =< c_{1,k}, ..., c_{N,k} >$, hence $c_{i,k}$ is the load capacity of robot $k$ for items of type $i$.

# Chapter 4

# Solution

In this section I propose the approaches with centralized coordination. The first strategy, mentioned above, is the CGS1:1 which consider only one task allocated for one robot. The second strategy extends the first, the main concept of this strategy is merging the tasks for optimaze the capacity of robot.

## 4.1 Greedy Strategy Single robot : Single task (GS1:1)

## 4.2 Set Partition Strategy Single robot : Multiple task (SPS1:N)

## 4.3 Coalition Formation Strategy Single robot : Multiple task (CFS1:N)

# Chapter 5

# Empirical Setting

Writing software for robots is difficult, particularly as the scale and scope of robotics continues to grow. Different types of robots can have wildly varying hardware, making code reuse non trivial. On top of this, the magnitude of the required code can be daunting, as it must contain a deep stack starting from driver-level software and continuing up through perception, abstract reasoning, and beyond.

## 5.1 Robot Operating System (ROS)

Our choice fell on ROS (Robot Operating System) which is a widespread open-source, meta-operating system for a robot. It provides several services that are commonly offered by an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It is worth noting that the full source code of ROS is publicly available, ROS is distributed under the terms of the BSD license, which allows the development of both non-commercial and commercial projects.

### 5.1.1 Nomencalture and Architecture

In this section we simply outline the terminology adopted in the ROS community to allow an easy comprehension of the following discussion.
The fundamental concepts of the ROS implementation are *nodes*, *messages*,

*topics*, and *services*. In ROS a system is typically comprised of many nodes. In this context, the term *"node"* is interchangeable with *"software module"*. The use of term *"node"* arises from visualization of ROS-based systems at runtime: when many nodes are running, it is convenient to render the peer-to-peer communications as a graph, called the *computation graph*, with process as graph nodes and the peer-to-peer links as arcs.
Nodes communicate with each other by passing *messages*. A message is a a strictly typed data structure. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types and constants. Messages can be composed of other messages, and arrays of other messages, nested arbitrarily deep. Messages descriptions are usually stored in `my_package/msg/MyMessageType.msg` and define the data structures for messages sent in ROS, called custom message.
Here is a simple example of a `*.msg` file that uses a header, some integer primitive, arrays of integer and array of other `*.msg` files. The message is specified in a language neutral interface definition language (IDL) which uses very short text files to describe its fields and allow an easy composition of complex messages:

```
1  Header header
2  bool take
3  bool go_home
4  uint32 ID_ROBOT
5  uint32 item
6  uint32 order
7  uint32 demand
8  uint32 dst
9  uint32 path_distance
10 uint32[] route
```

The custom message above rappresent a `Task.msg` which contains the basic information to define a task in the system. Instead, the custom message below, rappresent a `Mission.msg` which is composed of task messages addressed to a specific robot.

```
1  Header header
2  uint32 ID_ROBOT
3  uint32 capacity
4  Task[] Mission
```

These simple high-level message definitions is then parsed and processed by a code generator module, one for each support language (currentrly `C++`), which generates native implementations that “feel” like native objects, and

are automatically serialized and deserialized by ROS as messages are sent and received.
A node sends a message by publishing it to a given *topic*, which is simply a string such as `/topic` or `/pkg/topic`. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each other existence (decoupling). It is important to point out that because nodes connect to each other at runtime, the graph can be *dynamically* modified.
Although the topic-based publish-subscribe model is a flexible communications paradigm, its "broadcast" routing scheme is not appropriate forsynchronous transactions, which can simplify the design of some nodes. For this purpose ROS includes the concept of *services*, defined by a string name and a pair of strictly typed messages: one for the request and one for the response. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply.
As for the topic-based paradigm a high-level description of a service is then parsed and processed by a code generator module which generates the corresponding native implementation in a supported target language. Usually C++ messages are generated in `my_package/msg_gen/cpp/include/my_package`, while C++ services are generated in `my_package/srv_gen/cpp/include/my_package`.
To support collaborative development, the ROS software system is organized into *packages*. A ROS package is simply a directory which contains an XML file describing the package and stating any dependencies. A collection of ROS packages is a directory tree with ROS packages at the leaves: a ROS package repository may thus contain an arbitrarily complex scheme of subdirectories. This structure is primarily meant to partition the building of ROS-based software into small, manageable chunks of functionality.
In ROS, a *stack* of software is a cluster of nodes that does something coherent as a whole, as is illustrated in the simple *navigation* example reported in Figure. To allow for "packaged" functionality such as a navigation system, ROS provides a tool called `roslaunch`, which reads an XML-like description of a graph and instantiates the graph on the cluster, optionally on specific

hosts. Thus ROS is able to instantiate a set of nodes with a single command, once the nodes are described in a `launch` file, the simple usage is:

```
roslaunch [package] [filename.launch]
```

### 5.1.2 The Stage 2D Simulation

For visualization purposes we adopted the Stage 2D robot simulator which provides a virtual world populated by mobile robots and enriched with sensors, actuators and both approximate and exact localization. Stage is designed to be sufficiently simple to allow an easy set-up but at the same time it is intended to be just realistic enough to enable users to move controllers directly between Stage robots and real robots.
Stage is made available in ROS with the `stageros` node which wraps the simulator and exposes its functionality to the rest of the system. The following code reports how it is launched:

```
<?xml version="1.0" encoding="UTF-8" ?>
<launch>
    <arg name="map" default="grid" />
    <arg name="stage_pkg" default="stage_ros"/>
    <arg name="custom_stage" default="false" />
    <group unless="$(arg custom_stage)">
        <node name="stageros" pkg="$(arg stage_pkg)" type="stageros"
        args="$(find patrolling_sim)/maps/$(arg map)/$(arg map).world"
        output="screen" />
    </group>
    <group if="$(arg custom_stage)">
        <node name="stageros" pkg="$(arg stage_pkg)" type="stageros"
        args="$(find patrolling_sim)/maps/$(arg map)/$(arg map).world"
        output="screen">
            <param name="base_frame" value="base_link" />
            <param name="laser_topic" value="base_scan" />
            <param name="laser_frame" value="base_laser_link" />
        </node>
    </group>
</launch>
```

The `*.world` file specified tells Stage everything about the world, from obstacles (usually represented via a `*.pgm` image), to robots and other objects. In particular, after the definition of some parameters related to general camera and GUI options, we specify the static map on which the robot has to navigate (we will describe its characteristics shortly) and finally we include two specific files which aims defining the properties of respectively the laser sensor and the robot. The last instruction just throws the robot in the map by indicating it's $x$, $y$, $z$ and $\theta$ coordinates, this is summarized in:

```
include "../hokuyo.inc"
include "../crobot.inc"
include "../floorplan.inc"
include "../cpoint.inc"
window
( size [ 460 180 1 ]
  rotate [ 0.000 0.000 ]
  center [ 11.5 4.0 ]
  scale 20
  show_data 1)
floorplan
( size [23.0 8.0 1]
  pose [11.5 4.0 0 0]
  bitmap "model5.pgm")
include "robots.inc"
include "point.inc"
```

The first included file (hokuyo.inc) defines the physical and technical properties of the particular laser range finders support that we adopt: we define it to have a circular shape and to be mounted on top of the robot base which has the same circular shape. As for the sensor properties we specify the following parameters described in:

```
define hokuyo ranger
(
  sensor(
    range [ 0.0 5.0 ] # the max/min range reported by the scanner, in meters.
```

```
    fov 230 # the angular field of view of the scanner, in degrees.
    samples 1081 # the number of laser samples per scan.
    )
  # model properties
  color "orange"
  size [ 0.1 0.1 0.1 ]
  block( points 4
    point[0] [0 0]
    point[1] [0 1]
    point[2] [1 1]
    point[3] [1 0]
    z [0 1])
)
```

The second included file (crobot.inc) defines the physical properties of the robot, as mentioned above we define it to have a circular shape which is suffice for our purpose of having a mobile camera that moves around the world:

```
    define crobot position(
    size [0.3 0.3 0.2]
    origin [0 0 0 0]
    gui_nose 0
    drive "diff"

    # This block approximates a circular shape of a Robot
    block( points 16
        point[0] [ 0.225 0.000 ]
        point[1] [ 0.208 0.086 ]
        point[2] [ 0.159 0.159 ]
        point[3] [ 0.086 0.208 ]
        point[4] [ 0.000 0.225 ]
        point[5] [ -0.086 0.208 ]
        point[6] [ -0.159 0.159 ]
        point[7] [ -0.208 0.086 ]
        point[8] [ -0.225 0.000 ]
        point[9] [ -0.208 -0.086 ]
```

```
        point[10] [ -0.159 -0.159 ]
        point[11] [ -0.086 -0.208 ]
        point[12] [ -0.000 -0.225 ]
        point[13] [ 0.086 -0.208 ]
        point[14] [ 0.159 -0.159 ]
        point[15] [ 0.208 -0.086 ]
        z [0 1]
    )

    hokuyo( pose [0.15 0 -0.1 0] )

    # Report error-free position in world coordinates
    localization "gps"
    #localization_origin [ 0 0 0 0 ]

    # Some more realistic localization error
    localization "odom"
    odom_error [ 0.01 0.01 0.0 0.1 ]
)
```

## 5.2 Localization and Navigation

Purpose of this section is to describing how we address the two main problems in the context of mobile robotics: *localization* and *navigation*. The former deals with tracking the pose of the robot during its motion allowing it to localize itself in the map, the latter deals with driving the robot from a starting position to a goal position trying to avoid potential obstacles. The complete code for the `launch` file in which this takes are solved is reported below.

```
    <?xml version="1.0" encoding="UTF-8" ?>
<launch>
<arg name="robotname" default="robot_0" />
<arg name="mapname" default="grid" />
<arg name="use_amcl" default="true" />
```

```
<arg name="use_move_base" default="true" />
<arg name="use_srrg_localizer" default="false" />
<arg name="use_spqrel_planner" default="false" />
<group ns="$(arg robotname)">

  <!-- Run the map server -->
  <node name="map_server" pkg="map_server" type="map_server"
  args="$(find patrolling_sim)/maps/$(arg mapname)/$(arg mapname).yaml" />
  <!-- Standard ROS navigation modules -->

  <group if="$(arg use_amcl)">
      <!--- AMCL -->
      <include file="$(find patrolling_sim)/params/amcl/amcl_diff.launch" />

      <!-- Override AMCL Frame Params to include prefix -->
      <param name="/$(arg robotname)/amcl/base_frame_id"
          value="$(arg robotname)/base_link"/>
      <param name="/$(arg robotname)/amcl/odom_frame_id"
          value="$(arg robotname)/odom"/>
      <param name="/$(arg robotname)/amcl/global_frame_id"
          value="map"/> <!--common map frame for all robots -->
  </group>

  <group if="$(arg use_move_base)">
      <node pkg="move_base" type="move_base" respawn="false" name="move_base"
          output="screen">
<rosparam file=
    "$(find patrolling_sim)/params/move_base/costmap_common_params.yaml"
      command="load" ns="global_costmap" />
<rosparam file=
    "$(find patrolling_sim)/params/move_base/costmap_common_params.yaml"
      command="load" ns="local_costmap" />
<rosparam file=
    "$(find patrolling_sim)/params/move_base/local_costmap_params.yaml"
      command="load" />
<rosparam file=
```

```
42     "$(find patrolling_sim)/params/move_base/global_costmap_params.yaml"
43       command="load" />
44 <rosparam file=
45     "$(find patrolling_sim)/params/move_base/base_local_planner_params.yaml"
46       command="load" />
47           <!-- remap from="cmd_vel" to="desired_cmd_vel" / -->
48 
49           <!-- Override MOVE_BASE Frame Params to include prefix -->
50           <param name="global_costmap/laser_scan_sensor/sensor_frame"
51              value="/$(arg robotname)/base_laser_link"/>
52           <param name="global_costmap/laser_scan_sensor/topic"
53              value="/$(arg robotname)/base_scan"/>
54           <param name="global_costmap/robot_base_frame"
55              value="/$(arg robotname)/base_link"/>
56           <param name="local_costmap/global_frame"
57              value="/$(arg robotname)/odom"/>
58           <param name="local_costmap/laser_scan_sensor/sensor_frame"
59              value="/$(arg robotname)/base_laser_link"/>
60           <param name="local_costmap/laser_scan_sensor/topic"
61              value="/$(arg robotname)/base_scan"/>
62           <param name="local_costmap/robot_base_frame"
63              value="/$(arg robotname)/base_link"/>
64       </node>
65   </group>
66 
67   <!--- srrg_localizer -->
68   <group if="$(arg use_srrg_localizer)">
69     <node pkg="tf" type="static_transform_publisher" name="link1_broadcaster"
70           args="0 0 0 0 0 0 1 /map /$(arg robotname)/map 100" />
71     <node pkg="spqrel_navigation" type="srrg_localizer2d_node" name="srrg_localizer"
72           output="screen">
73         <param name="global_frame_id" value="$(arg robotname)/map"/>
74         <param name="base_frame_id" value="$(arg robotname)/base_link"/>
75         <param name="odom_frame_id" value="$(arg robotname)/odom"/>
76         <param name="laser_topic" value="$(arg robotname)/base_scan"/>
77         <param name="use_gui" value="false"/>
```

```
78        </node>
79   </group>
80   <!--- spqrel_planner -->
81   <group if="$(arg use_spqrel_planner)">
82         <node pkg="spqrel_navigation" type="spqrel_planner_node"
83         name="spqrel_planner" output="screen">
84            <param name="max_range" value="10.0"/>
85            <param name="max_linear_vel" value="1.0"/>
86            <param name="max_angular_vel" value="1.0"/>
87            <param name="global_frame_id" value="$(arg robotname)/map"/>
88            <param name="base_frame_id" value="$(arg robotname)/base_link"/>
89            <param name="laser_topic" value="$(arg robotname)/base_scan"/>
90            <param name="command_vel_topic" value="$(arg robotname)/cmd_vel"/>
91            <param name="robot_radius" value="0.3"/> <!-- Raggio del robot -->
92            <param name="use_gui" value="false"/>
93         </node>
94   </group>
95   <!-- INITIAL POSES FOR LOCALIZER -->
96   <include file=
97     "$(find patrolling_sim)/params/amcl/$(arg robotname)_initial_pose.xml" />
98 </group>
99 </launch>
```

### 5.2.1 Mapping

The robot navigation system can be initialized with or without an a priori, *static map*. When initialized without a map, the robot only knows about obstacles that it has seen, and will make optimistic global plans through areas that it has not yet visitied which may traverse unknown space, potentially intersecting unseen obstacles. As the robot receives more information about the world, it replans accordingly to avoid obstacles. Initialized with a static map, the robot will make informed plans about distant parts of the environment, using the map as prior obstacle information. In our case we provide ROS with a static map of real laboratory in University of Verona. Doing so requires us to set the *map_server* node that reads a map from disk and offers

it via a ROS service:

```
<!-- Run the map server -->
<node name="map_server" pkg="map_server" type="map_server"
args="$(find patrolling_sim)/maps/$(arg mapname)/$(arg mapname).yaml" />
```

The current implementation of *map_server* convert color values in the map image data into ternary occupancy values: *free*(0), *occupied*(100), and *unknown*(-1).
Maps manipulated by the tools in the *map_server* package are stored in a pair of file: the `*.yaml` file describes the map meta-data and names the image file while the image file encodes the occupancy data.
The `*.yaml` file given as an argument to the *map_server* node is reported below with a brief description of it's parameters.

```
# path to the image file containing the occupancy data.
image: src/patrolling_sim/maps/model5/model5.pgm
# resolution of the map, meters/pixel.
resolution: 0.014100
# the 2-D pose of the lower-left pixel in the map, as (x, y, yaw).
origin: [0.000000, 0.000000, 0.000000]
# whether the white/black free/occupied semantics should be reversed.
negate: 0
# pixels with occupancy probability greater than this are considered completely occupied.
occupied_thresh: 0.65
# pixels with occupancy probability less than this are considered completely free.
free_thresh: 0.196
```

### 5.2.2 Localization

Both in the case of a given static map or without a given static map, we require that the robot's pose be tracked in a consistent global coordinate frame. When not using a map, the robot's pose is usually estimated by integrating wheel odometry, possibly fused with data from an inertial measurement unit (IMU). When using a map, as in our case, the robot is usually localized using a probabilistic technique, in particular we adopt the Adaptive Monte Carlo Localization (`amcl`) system:

```
<group if="$(arg use_amcl)">
    <!--- AMCL -->
    <include file="$(find patrolling_sim)/params/amcl/amcl_diff.launch" />

    <!-- Override AMCL Frame Params to include prefix -->
    <param name="/$(arg robotname)/amcl/base_frame_id"
        value="$(arg robotname)/base_link"/>
    <param name="/$(arg robotname)/amcl/odom_frame_id"
        value="$(arg robotname)/odom"/>

    <!--common map frame for all robots -->
    <param name="/$(arg robotname)/amcl/global_frame_id" value="map"/>
</group>
```

`amcl` is a probabilistic localization system for a robot moving in 2D. It implements the Adaptive (or KLD-sampling) Monte Carlo Localization approach which uses a particle filter to track the pose of a robot against a known map. The map of the environment where the robot has to localize itself must be given to the robot beforehand. In our case the map is provided by tha *map_server* node. This map is the so called *occupacy map* like before mentioned: it contains a value for every location which indicates the probability that this location is occupied by an object such as a wall. `amcl` takes in a laser-based map, laser scans and outputs pose estimates. On startup, `amcl` initializes its particle filter according to the parameters provided. In this respect we remark that the `amcl` node is launched only after having setting up three categories of ROS parameters that we use to configure its behaviour: *overall filter*, *laser model* and *odometry model*. The complete configuration file is reported below.

```
<?xml version="1.0" encoding="UTF-8" ?>
<launch>
<arg name="robotname" default="robot_0" />
<node pkg="amcl" type="amcl" name="amcl" args="scan:=base_scan">
  <!-- Publish scans from best pose at a max of 10 Hz -->
  <param name="odom_model_type" value="diff"/>
  <param name="odom_alpha5" value="0.1"/>
```

```
8   <param name="transform_tolerance" value="0.2" />
9   <param name="gui_publish_rate" value="-10.0"/>
10  <param name="laser_max_beams" value="30"/>
11  <param name="min_particles" value="100"/>
12  <param name="max_particles" value="500"/>
13  <param name="kld_err" value="0.05"/>
14  <param name="kld_z" value="0.99"/>
15  <param name="odom_alpha1" value="0.2"/>
16  <param name="odom_alpha2" value="0.2"/>
17  <!-- translation std dev, m -->
18  <param name="odom_alpha3" value="0.8"/>
19  <param name="odom_alpha4" value="0.2"/>
20  <param name="laser_z_hit" value="0.5"/>
21  <param name="laser_z_short" value="0.05"/>
22  <param name="laser_z_max" value="0.05"/>
23  <param name="laser_z_rand" value="0.5"/>
24  <param name="laser_sigma_hit" value="0.2"/>
25  <param name="laser_lambda_short" value="0.1"/>
26  <param name="laser_lambda_short" value="0.1"/>
27  <param name="laser_model_type" value="likelihood_field"/>
28  <!-- <param name="laser_model_type" value="beam"/> -->
29  <param name="laser_likelihood_max_dist" value="2.0"/>
30  <param name="update_min_d" value="0.2"/>
31  <param name="update_min_a" value="0.5"/>
32  <param name="odom_frame_id" value="odom"/>
33  <param name="base_frame_id" value="base_link"/>
34  <param name="resample_interval" value="1"/>
35 </node>
36 </launch>
```

### 5.2.3 Navigation

At high level the navigation system is quite simple. It takes in data from sensors, odometry, and a navigation goal, and outputs velocity commands that are sent to a mobile base. The low-level architecture of system, however,

is complex and consist of many components that interacts together.
For navigation purposes we rely on the *move_base* package which provides an interface with the entire ROS navigation stack. The *move_base* package provides an implementation of an *action* that, given a goal in the world, will attempt to reach it with a mobile base. It links together a *global* and *local* planner to accomplish its navigation task, it also maintains two costmaps, one for the global planner and one for the local planner. An architecture view of the node its interaction with other components is show in Figure below.

The pre-requisites of navigation stack, along with a brief description of it's main components, are provided in the section below.

## Trasformations

Robotics systems often need to track spartial relationships between *frames* for a variety of reasons: between a mobile robot and some fixed frame of reference for localizzation or, as in our case, between the frame related to the mobile base and the one related to the laser sensor. To simplify the treatment of spartial frames, a trasformation system has been written for ROS, called `tf`. The `tf` system constructs a dynamic *trasformation tree* which realtes all frames of reference in the system. At an abstract level, a trasformation tree define *offsets* in terms of both translation and rotation between different coordinate frames.
Referring to our case of a simple robot consisting of a circular mobile base with a single laser support mounted on top of it we define two coordinate frames: one corresponding to the center point of the base of the robot and one for the center point of the laser support that is mounted on top of the base. We've called the coordinate frame attached to the modile base `base_link` and the coordinate frame attached to the laser support `base_laser_link`. Once this configuration in defined, every time that we have some data from the laser in the form of distances from the laser's center point and we want to take this data and use it to help the mobile base avoid obastacles in the world we invoke the `tf` system which in the general case transforms the information towards the `base_link` coordiante frame. Below we reported the Figure which rappresent the configuration of `base_link` and the `base_laser_link` frames.

Our robot can thus use this information to reason about laser scans in the `base_link` frame and safely plan around obstacles in its environment.

## Sensor Information

The navigation stack uses information from sensors to avoid obstacles in the world, it assumes that these sensors are publishing either `sensor_msgs/LaserScan` or `sensor_msgs/PointCloud` message over ROS. Publishing data correctly from sensors over ROS is important for the navigation stack to operate safely. If the navigation stack receives no information from a robot's sensors, then of course it will drive blindly and, most likely, hit things. Our `sensor_msgs/LaserScan` message type, like many other messages sent over ROS, contain `tf` frame and time dependent information. To standardize how this information is sent, the `Header` message type is used as field in all such messages. The three field in the `Header` type are show below.

```
# Standard metadata for higher-level flow data types
uint32 seq
time stamp
string frame_id
```

The `seq` field corresponds to an identifier that automatically increases as messages are sent from a given publisher. The `stamp` field stores time information that should be associated with data in a message. The `frame_id` field stores `tf` frame information that should be associated with data in a message. The body of the `sensor_msgs/LaserScan` message holds informations about any given scan and contains the associated geometric informations in the following form:

```
# Laser scans angles are measured counter clockwise, with 0 facing forward
Header header
float32 angle_min # start angle of the scan [rad]
float32 angle_max # end angle of the scan [rad]
float32 angle_increment # angular distance between measurements [rad]
float32 time_increment # time between measurements [seconds]
float32 scan_time # time between scans [seconds]
float32 range_min # minimum range value [m]
```

```
float32 range_max # maximum range value [m]
float32[] ranges # range data [m] (Note: values < range_min or > range_max
should be discarded)
float32[] intensities # intensity data [device-specific units]
```

## Odometry Informations

The navigation stack requires that odometry information be published using `tf` and the `nav_msgs/Odometry` message. The last message in required because `tf` alone does not provide any information about the velocity of the robot, it is thus required that any *odometry source* publishes both this kind of information to *move_base*.
The `nav_msgs/Odometry` message stores an estimate of the position and velocity of a robot in free space:

```
# The pose in this message should be specified in the coordinate frame given
by header.frame_id.
# The twist in this message should be specified in the coordinate frame
given by the child_frame_id
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

The *pose* in this message corresponds to the estimated position of the robot in the odometric frame of reference along with an optional covariance for the certainty of that pose estimate. The *twist* in this message corresponds to the robot's velocity in the child frame, normally the coordinate frame of the mobile base, along with an optional covariance for certainty of that velocity estimate.
In our case since the `stageros` node represents the odometric source for the system,it is going to publish all the relevant trasformations between the coordinate frames it manages towards the `/tf` topic which maintains the current trasformation tree for our system, and a `nav_msgs/Odometry` message to the `/odom` topic so that the navigation stack can retrieve velocity information from it.

### Base Controller

As previously specified the navigation stack assumes that it can send velocity commands using a `geometry_msgs/Twist` message assumed to be in the base coordinate frame of the robot on the `/cmd_vel` topic. This means there must be a node subscribing to the `/cmd_vel` topic that is capable of talking $(v_x, v_y, v_\theta) \longleftrightarrow$ (`cmd_vel.linear.x`, `cmd_vel.linear.y`, `cmd_vel.angular.z`) veloccities and converting them into motor commands to send a mobile base. In this work the `stageros` node subscribes to the `/cmd_vel` topic simulating the movement of a real robot in a real environment.

## 5.2.4 Global and Local planner algorithms

The `base_local_planner` is responsible for computing velicty commands to send to the mobile base of the robot given a high-level plan. In particular, this package adheres to the `BaseLocalPlanner` interface specified in the `nav_core` package which provides common interfaces for both *global* and *local* action planning.
As we shall see, the local planner is seeded with the high-level plan produced by the global planner and the purpose of the `nav_core` package is just to provide common interfaces for navigation.

### Global planner

The global planner is given the obstacle and cost information contained in the global costmap, information from the robot's localization system, and a goal in the world. From these, it creates a high-level plan for the robot to follow to reach the goal location. In more detail, the global planner will create a series of way-points for the local planner to achieve, these way-points assumes that the robot is circular in shape, and may in fact be infeasible for a more general case. Also, the global planner doesn't take the dynamics of the robot into account so it can produce plans that are dynamically infeasible as well. While this ensures that the global planner returns in a small amount of time, it also means that the planner is *optimistic* in the plans that it creates. However our robot is not much affected by this problems because it is circular

in shape and does not have a particular dynamics to take into account.
The global planner used for this nagivation system is `ROS navigation` stack This planner, as started before, assumes a circular robot and operates on a costmap to compute a navigation function that can later be used to find a *minimum cost plan* from a start point to an end point in a grind.
As outlined before, the global planner may produce a path for the robot that is infeasible, such as a plan that turns through a narrow way too tightly, causing the corners of the robot to hit the surrounding obstacles. Because of its shortcomings, the global planner is used only as a high-level guide for navigation in an environment and we also need a *local planner* for navigation.

## Local planner

The local planner is responsible for generating velocity commands for the mobile base that will safely move the robot towards a goal. The local planner is seeded with the plan produced by the global planner, and attempts to follow it as closely as possible while taking into account the kinematics and dynamics of the robot as well as the obstacle information stored in the costmap.
The local planner used for this navigation system is `base_local_planner`. This package supports any robot footprint that can be represented as a convex polygon or circle, so it is fine for our circular-shaped robot. The `base_local_planner` package provides a *controller* that drives a mobile base in the plane. This controller serves to connect the path planner to the robot. Using a map, the planner creates a trajectory for the robot to move from a start to a goal location. Along the way, the planner creates, at least locally around the robot, a value function, represented as a grid map. This value function encodes the costs of traversing through the grid cells based on their occupancy status. The controller uses this value function to determine $v_x$, $v_y$ and $v_\theta$ velocities to send to the robot.
In order to generate safe velocity commands to send to the mobile base, the local planner we adopt makes use of a technique known as the *Dynamic Window Approach* (DWA) to forward simulate and select among potential commands based on a cost function. The basic idea is as follows:

- First it discretely samples the robot's control space (which means samples from the set of achievable velocities $v_x$, $v_y$ and $v_\theta$ for just one simulation step given the acceleration limits of the robot);

- Then for each sampled velocity, it performs forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time;

- Next it evaluates (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as *proximity to obstacles, proximity to the goal, proximity to the path produced by the global planner and speed* and it discards illegal trajectories (those that collide with obstacles);

- Finally it picks the highest-scoring trajectory and sends the associated velocity to the mobile base;

- Rinse and repeat.

In order to score trajectories efficiently, a map grid is used. For each control cycle, a grid is created around the robot (the size of the local costmap), and the global path is mapped onto this area. This means some of the grid cells will be marked with distance 0 to a path point, and distance 0 to the goal. A propagation algorithm then efficiently marks all other cells with their *Manhattan distance* to the closest of the points marked with zero. The goal of the global path may often lie outside the small area covered by the map grid, so when scoring trajectories for proximity to goal, what is considered is the "local goal", meaning the first path point which is inside the area having a consecutive point on the global path outside the area.
We report here our configuration file for the local planner we adopt, it is worth noting that through setting the weights on each component of the cost function differently it is possible to change drastically the behaviour of the robot: for example it is possible to force it to stay close to the projection of the global path into the local costmap, or to stay quite far from obstacles changing consequently it's trajectory across the map. The values below works well for our case since they take into account both the shape of the robot and

the needs for computation. For a detailed description parameters see code below.

```
controller_frequency: 5.0
TrajectoryPlannerROS:

# Robot Configuration Parameters
max_vel_x: 1.00
min_vel_x: 0.10
max_trans_vel: 1.00
min_trans_vel: 0.10
max_rot_vel: 1.0
min_in_place_rotational_vel: 0.1
acc_lim_th: 0.75
acc_lim_x: 0.50
acc_lim_y: 0.50
holonomic_robot: false

# Goal Tolerance Parameters
yaw_goal_tolerance: 6.28
xy_goal_tolerance: 0.40

# Controller Parameters
pdist_scale: 0.6
gdist_scale: 0.8
occdist_scale: 0.01
meter_scoring: true

# Forward Simulation Parameters
sim_time: 1.5
vx_samples: 6
vtheta_samples: 20

# Trajectory Scoring Parameters
heading_lookahead: 0.325
dwa: true

```

```
35 # Oscillation Prevention Parameters
36 oscillation_reset_dist: 0.05
```

## 5.3 Cost-maps configurations

The navigation stack uses two costmaps to store information about obstacles in the world. One costmap is used for *global planning*, meaning creating long-term plans over the entire environment, and the other is used for *local planning* and obstacle avoidance.
In particular, the ROS package `costmap_2D` provides an implementation of a 2D costmap that takes in sensor data from the world, build a 2D occupacy grid of the data and inflates costs in a 2D costmap based on the occupacy grid and a user specified *inflation radius*.
Because the robots we study are constrained to drive on flat ground, and cannot, for example, step or jump over obstructions, we assemble obstacle data into a planar costmap on which the planners operates. The costmap is initialized with our laboratory static map, but updates as new sensor data comes in to maintain an up-to-date view of the robot's local and global environment.
As previously specified, the laboratory image describes the occupacy state of each cell of the map in the color of the corresponding pixel. Cells with occupacy probability greater than the value stored in `occuped_thresh` are considered completely occupied and are assigned a lethal cost, meaning that no part of the robot's circular footprint is allowed to be inside of the corresponding two-dimensional cell. Then, inflation is performed in two dimension to propagate costs from obstacles out to user-specified *inflation radius*. Cells that are less than one inscribed radius of the robot away from an obstacle are assigned a uniformly high cost, after which an exponential decay function is applied that will cause the cost to decrease with the distance from the obstacles.

### 5.3.1 Common configuration

There are some configuration options that we want both global and local costmap to follow. In our case the global configuration settings for both the costmaps are as follows:

```
1 obstacle_range: 0.50
2 raytrace_range: 3.0
3 robot_radius: 0.33
4 inflation_radius: 0.33
5 observation_sources: laser_scan_sensor
6 laser_scan_sensor: {sensor_frame: base_laser_link,
7  data_type: LaserScan, topic: base_scan, marking: true, clearing: true}
```

The first two parameters set thresholds on obstacle information put into the costmap. The `obstacle_range` parameters determines the maximum range sensor reading that will result in an obstacle being put into the costmap. The `raytrace_range` parameter determines the range to which we will ray-trace freespace given a sensor reading.
Next we set the radius of the robot since it is circular and the inflation radius for the costmap. The inflation radius should be set to the maximum distance from obstacles at which a cost should be incurred.
The `observation_sources` parameter defines the sensor that is going to be passing information to the costmap, the last line sets it's parameters. The `sensor_frame` parameter is set to the name of the coordiante frame of the sensor, the `data_type` parameter is set to `LaserScan` since this is the type of message used by the topic, and the `topic` parameter is set to the name of the topic that the sensor publishes data on. The `marking` and `clearing` parameters determine whether the sensor will be used to add obstacle information to the costmap, clear obstacle information from the costmap, or do both.

### 5.3.2 Global configuration

Below are our configuration settings for the *global* costmap along with a description of their semantics:

```
global_costmap:
   global_frame: /map
   robot_base_frame: base_link
   update_frequency: 3.0
   publish_frequency: 0.0
   static_map: true
   inflation_radius: 0.66
```

The `global_frame` parameter defines what coordinate frame the costmap should run in, in this case, we'll choose the `map` frame. The `robot_base_frame` parameter defines the coordiante frame the costmap should reference for the base of the robot. The `update_frequency` parameter determines the frequency, in Hz, at which the costmap will run its update loop. The `static_map` parameter determines whether or not the costmap should initialize itself based on a map served by the *map_server*.

### 5.3.3 Local configuration

Below are our configuration settings for the *local* costmap along with a description of their semantics:

```
local_costmap:
   global_frame: odom
   robot_base_frame: base_link
   update_frequency: 3.0
   publish_frequency: 2.0
   static_map: false
   rolling_window: true
   width: 4.0
   height: 4.0
   resolution: 0.05
```

The `global_frame`, `robot_base_frame`, `update_frequency` and `static_map` parameters are the same as described previously. The `publish_frequency` parameter determines the rate, in Hz, at which the costmap will publish visualization information. Setting the `rolling_window` parameter to true means that the costmap will remain centered around the robot moves through the

world. The `width`, `height` and `resolution` parameters set the width [m], height [m] and resolution [m/cell] of the costmap.

### 5.3.4 Local and global coordinate frames

We now briefly discuss why it is important to distinguish between global and local coordinate frames when building a navigation system. A global coordinate frame, such as the one used by the *global planner* and specified through the `global_frame: /map` parameter, is advantageous in that it provides a globally consistent frame of reference, but it is flawed in that it is subject to discontinuous jumps in its estimation of the robot's position. For example, when the localization system is struggling to determine the robot's position, it is not uncommon for the robot to teleport, on a single update step, from one location to another that is one meter away.
A local coordinate frame, such as the one used by the *local planner* and specified through the `global_frame: /odom parameter`, has no such jumps, but presents its own flaws in that it is prone to drifting over time.
In theory, all planning and obstacle avoidance could be performed in the global frame, but this may lead to problems when discrete jumps in localization occur. To illustrate this, consider the case in which a robot navigates through a narrow way with limited clearance on either side. If the navigation system attempts to plan in a global frame, localization jumps may drastically affects the robot's obstacle information and a jump in the robot's position of just a few centimetres to either side, combined with new sensor data, may actually be enough for the robot to consider that way as unfeasible. If the robot instead operates in a local frame, nearby obstacles are not affected by jumps in localization, and the robot can traverse through the way independently of any difficulties with the localization system. Thus, we use the global coordinate frame to create high-level plans for the robot but also a local coordinate frame for local planning and obstacle avoidance. To relate the two frames, the plan produced by the global planner is mapped from the global coordinate frame into the local coordinate frame on every cycle.

## 5.4 Recovery behaviours

The navigation system as described until now works well most of the time, attempting to achieve a goal pose with its base to within a user-specified tolerance, but there are still situations where the robot can get stuck. One common cause of failure for the navigation system is *entrapment*, where the robot is surrounded by obstacles and cannot find a valid plan to its goal. When the robot finds itself in this situation, a number of increasingly aggressive recovery behaviours are executed to attempt to clear out space, Figure shows the relations among them. First, obstacles outside of a user settable region will be cleared from the robot's map. If this fails, the robot performs an in-place rotation to attempt to clear out space. If this too fails, a more aggressive map reset is attempted where the robot clears all obstacles that are outside of its circumscribed radius. After this, another in-place rotation is performed to clear space, and if this last step fails, the robot will abort on its goal which it now considers infeasible. After each of these behaviour completes, `move_base` will attempt to make a plan. If planning is successful, `move_base` will continue normal operation. Otherwise, the next recovery behaviour in the list will be executed.

## 5.5 Extension of Patrolling_sim package

In this thesis, the problem of allocation task for logistic applications a given environment with an arbitrary number of robots is studied. In this section, details are given on how agents obtain the representation of the environment.

### 5.5.1 Obtaining a Topological representation

In `patrolling_sim` package the topological map represent the area to travel by a graph $G = (V, E)$ with vertices $v_i \in V$ and edges $e_{i,j} \in E$, enabling robots to assess the topology of its surroundings. In this representation, vertices represent the load, unload bays and the travel locations, the edges represent the connectivity between those locations. The cost of an edge $|e_{i,j}|$ is defined by the metric distance between vertex $v_i$ and $v_j$. $|V|$ and $|E|$ repre-

sent the cardinality of the set $V$ and $E$, respectively. Seeing as undirected graphs are assumed, then: $|E| \leq \frac{|V| \cdot (|V|-1)}{2}$. A path $\pi$ is composed of an array of vertices in $V$.
Since the topological maps considered in this context represent real-world 2D environments, it is assumed that $G$ has the following properties:

- *Undirected*: where $|e_{i,j}| = |e_{j,i}|$ and the edge weights safety the triangle inequality.
- *Connected*: where $\forall v_h, v_i \in V, \exists x = \{v_h, \cdots, v_i\}$.
- *Simple*: where two neighbor vertices $v_i$ and $v_j$ are connected by a unique edge $e_{i,j}$ and no graph loops exist.
- *Planar*: where a pair of edges $e_{g,h}, e_{i,j} \in E$ never crosses each other.

As a consequence of there properties, $G$ is usually *non-complete*; for every pair $v_h, v_i \in V$ there may not exist an edge $|e_{h,i}|$ connecting each pair of vertices.
In this work, it noteworthy that any generic planar graph may be addressed, but for our logistic application in the laboratory map the topological graph $G$ is *complete*.

# Chapter 6

# Experiments

# Chapter 7

# Conclusions and Future Work

# Acknowledgements