POLITECNICO DI TORINO

Department of Control and Computer Engineering

Master’s degree in Mechatronic Engineering

2018/2019

Robotics – Prof. Rizzo

ROS project

TA

Stefano Primatesta

Students

Di Domenico Dario – 263409

Gianfagna Daniele – 260948

Malacarne Francesco – 260199

Martini Mauro – 260771

Summary

Project goals

The main purpose of this project is to succeed in accomplishing several tasks using the TurtleBot3 version of ROS, Robot Operating System, in order to become familiar with such a virtual environment and its main features and potential.
In particular, three main goals are decided to be studied and developed:

1. **Multiple navigation on a single machine:** launch a simulation exploiting Gazebo being able to spawn at least two robots in a common virtual world, asking for individual goals by means of the RViz graphic interface. Thanks to this first step, it is possible to get a better whole understanding of navigation main topics (amcl, move_base, odometry), as well as useful tools as the tf tree.
2. **Creation of a node able to send the closest robot to a certain point:** write a working node in python to command a global navigation goal to the two spawned robots in Gazebo. Such a result is chosen to be reached in order to go more into depth in ROS working structure, looking at the communication principles between different nodes and at the most common functions involved in a script of this type. A brief introduction to python is also necessary.
3. **Multiple navigation on multiple machines**: connect ROS between at least three separate machines, in order to test the possibility to assign to each of them a specific role in the simulation.

Approach

Firstly, some tests involving both RViz and Gazebo are run with a single robot on a single machine to check the successful installation of Turtlebot3 ROS packages. Then, after the comprehension of how the navigation works, the goal was to replicate the structure of a single robot simulation but rearranging it for two robots. Therefore, the Gazebo simulation launch files and the navigation ones are modified in order to allow the desired tasks. Once executed the navigation simulation of two robots on a single machine, the python node is created.

Theory

Map

The first essential feature for navigation is the map. The map is a fundamental requirement to guarantee a correct navigation system. SLAM (Simultaneous Localization And Mapping) is developed to let the robot create a map with or without a help of human being. This is a method of creating a map while the robot explores the unknown space and detects its surroundings and estimates its current location as well as creating a map. [1]

The map_server node reads a map saved on the disk and offers it as a ROS service. The map file is an image *.pgm with associated a *.yaml file with some parameters (resolution, origin position, etc.).

Pose of the Turtlebot3

The second element to guarantee a reliable navigation is the robot pose: it defines the position and the orientation in the map. ROS defines pose as the combination of the robot's position (x, y, z) and orientation (x, y, z, w). The orientation is described by x, y, z and w in quaternion form, whereas position is described by three vectors, such as x, y and z. [1]

Distance Measuring Sensor

For navigation, robot should have sensors such as LDS (Laser Distance Sensor), LRF (Laser Range Finder) or LiDAR that can measure distance to the obstacle on the XY plane. In other words, it is necessary to mount a sensor capable of measuring the distance on the XY plane. In Turtlebot3, a laser scanner is used. Its data are published on a /scan topic of type sensor_msgs/LaserScan. [1]

Path calculation and driving

The last essential feature for navigation is to calculate and travel the optimal route to the destination. This is called path search and planning, and there are many algorithms that perform this, such as potential field, particle filter, and RRT (Rapidly-exploring Random Tree). In Turtlebot3, a combination of a particle filter and the Dijkstra's algorithm is used. [1]

Odometry

Odometry informs about the robot motion and is the feedback of the motion control. Odometry data are useful to reconstruct the robot pose. In fact, the traveled distance should be measured by means of the amount of rotation of the driving wheel. Such information is provided by the wheel encoders. [1]

Since odometry is generally subjected to uncertainties and errors (for instance due to slipping condition), such information can be supported/corrected by range and vision sensors. Moreover, odometry defines the transformation odom → /base_link in the tf_tree, providing the transformation between the odom frame, fixed in the spawn position of the robot, and the base_link frame which follows the robot along the map. [2]

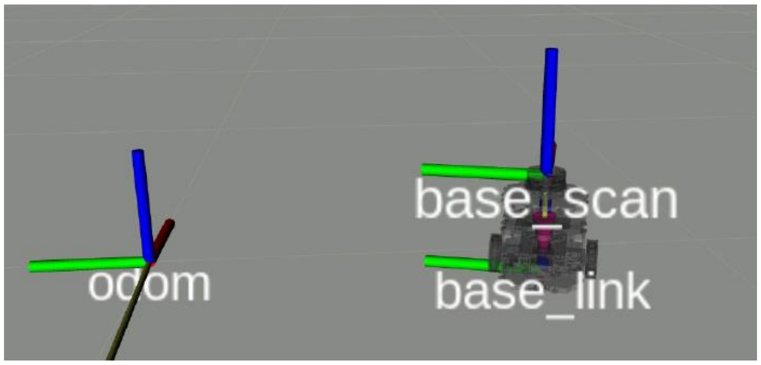


Figure 2.1: odom and base_link frames for Turtlebot3 burger.

Localization and pose estimation

On the map, the robot updates its odometry information with the encoder and the inertial sensor (IMU sensor) and measures the distance from the pose of the sensor to the obstacle (wall, object, furniture, etc.). The encoder calculates the approximate pose of the robot with dead reckoning, which measures the amount of rotation of the driving wheel. This process comes with quite an amount of estimation error and the inertial information measured by the inertial sensor compensates for the error of the calculated pose.

Furthermore, this estimated pose can be corrected once again with the surrounding environmental information obtained through the distance sensor or the camera used when creating the map. This pose estimation methodology includes Kalman filter, Markov localization, Monte Carlo localization using particle filter, and so on. For Turtlebot3 pose estimation, particle filter localization method and Adaptive Monte Carlo Localization (AMCL) is used. [1]

Navigation stack

Navigation stack is a collection of ROS packages to move autonomously the robot from one location to the specified destination in a given environment. For this purpose, a map that contains geometry information of furniture, objects and walls of the given environment is required. Given a goal pose, it uses sensors and odometry data to provide safe velocity commands to the mobile robot. The basic structure of the navigation stack is shown in figure 2.2:

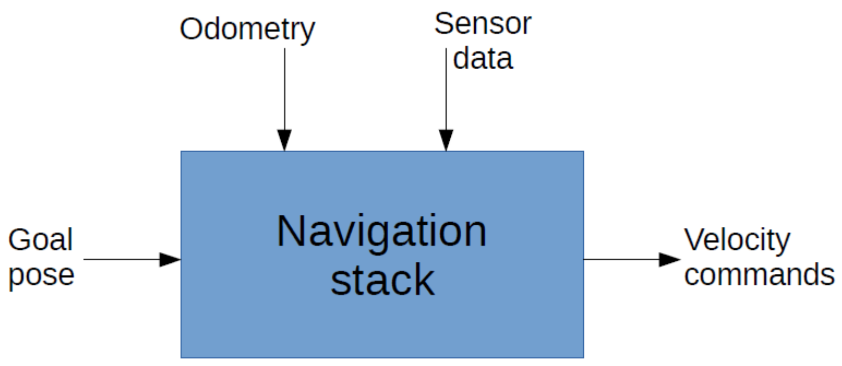


Figure 2.2: basic structure of the navigation stack. [2]

In order to work properly, the navigation stack requires three main ROS nodes:

- amcl: performs the localization of the robot in the map;

- move_base: compute the velocity commands to reach the goal pose;
- map_server: provides the reference map.

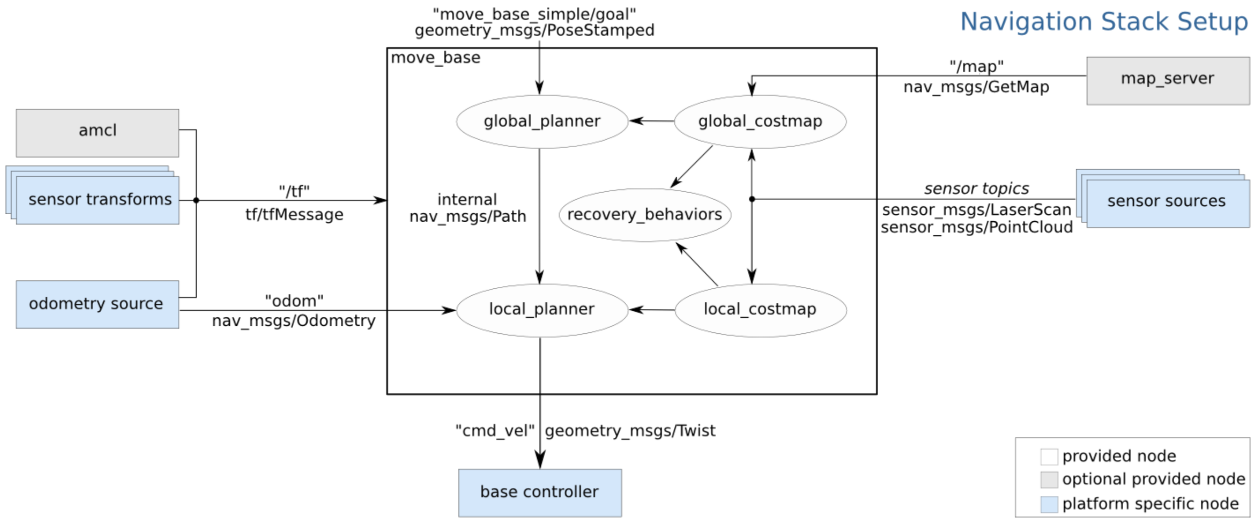


Figure 2.3: main topics of the navigation stack. [2]

Moreover, the robot navigation workflow is displayed in figure 2.4.

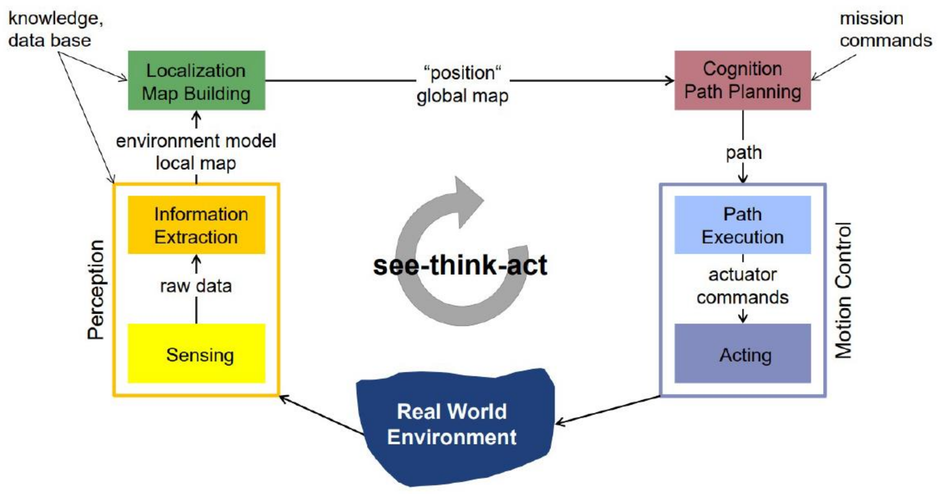


Figure 2.4: navigation process flow behind the autonomous navigation.

AMCL node (Adaptive Monte Carlo Localization)

Estimates robot position using a particle filter and exploits a standard probabilistic localization algorithm.

It is subscriber of the topics:

- /scan (type sensor_msgs/LaserScan): laser scanner data.
- /initial_pose (type geometry_msgs/PoseWithCovarianceStamped.msg): the initial pose of the robot on the map.
- /map (type nav_msgs/OccupancyGrid): the map published by the map_server node.

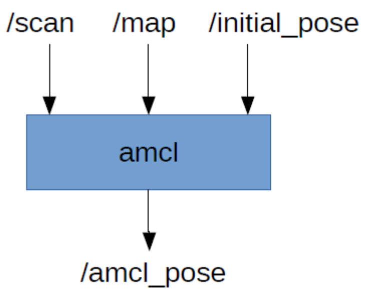


Figure 2.5: amcl inputs and output.

Once obtained such information, it provides the estimated robot pose inside /amcl_pose.

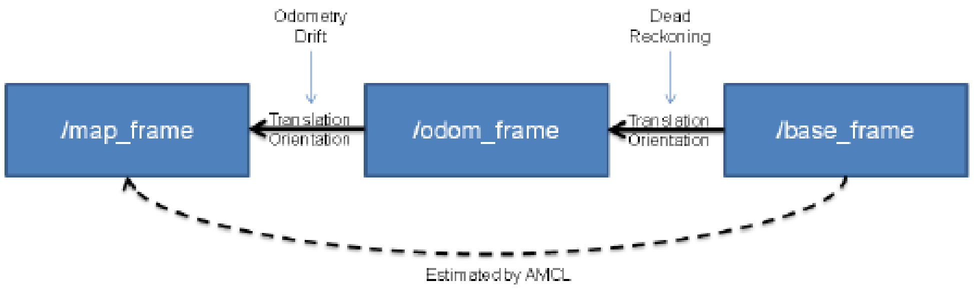


Figure 2.6: amcl node working principle.

The working principle behind the amcl theory is a probabilistic approach: the robot is located in the most probable region depending on the input data (sensor, odometry and map). Furthermore, a particle filter is used to estimate the pose.

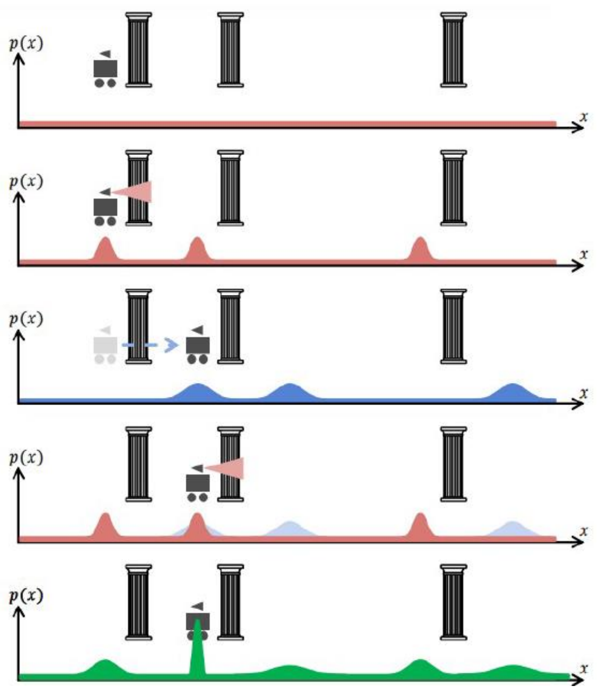


Figure 2.7: probabilistic working principle on which amcl is based on.

move_base node

move_base provides a safe motion to reach the desired goal pose and it requires:

- The actual pose of the robot (amcl);
- Odometry data;
- Sensors topic.

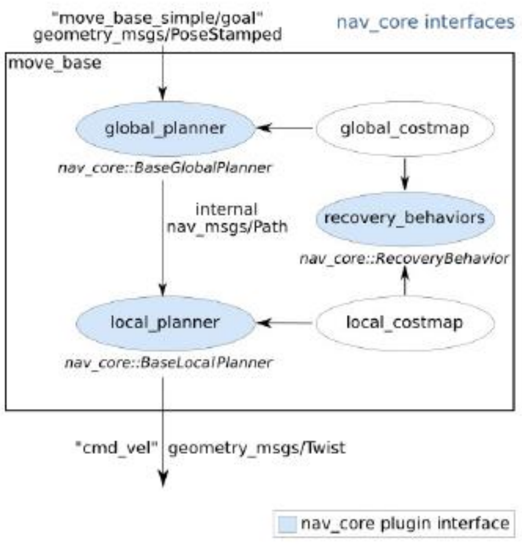
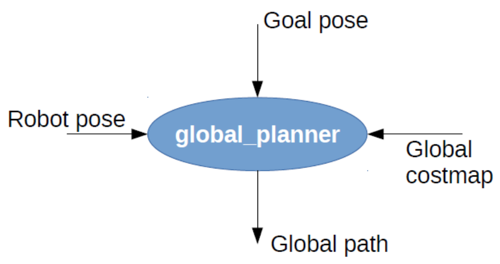

"move_base_simple/goal"
geometry_msgs/PoseStamped
nav_core interfaces
move_base
global_planner
global_costmap
nav_core::BaseGlobalPlanner
internal
nav_msgs/Path
recovery_behaviors
nav_core::RecoveryBehavior
local_planner
local_costmap
nav_core::BaseLocalPlanner
"cmd_vel" geometry_msgs/Twist
nav_core plugin interface

Figure 2.8: topic used to accomplish a navigation goal.

It consists of 5 elements:

- global planner: computes the global path from the current pose of the robot to the desired goal pose and it searches for an optimal path avoiding obstacles marked on the global costmap. It receives as inputs the goal pose, the robot pose and the global costmap. The output is the global path; anyway, before sending velocity commands, the path must be checked by the local planner, which must approve or reject the desired trajectory. The default algorithm is the Dijkstra's one.



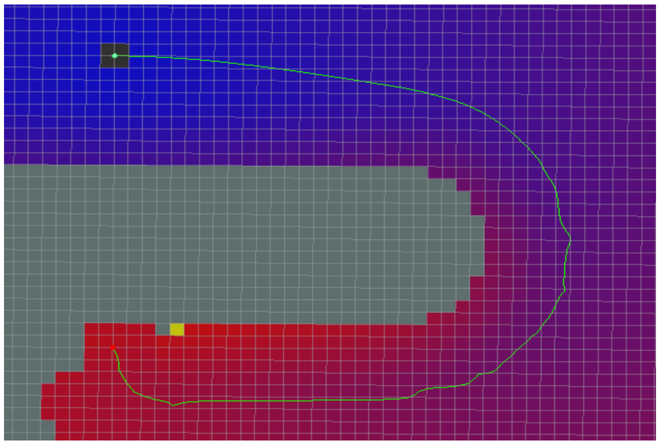


Figure 2.9: global planner using Dijkstra's algorithm.

- global costmap: is the reference map of the global planner and it is initialized using the static map provided by the map_server. It has the same dimension and resolution of the map of the map_server and is a dynamic map, updated with sensor data with a fixed rate (1 Hz). Each value of the costmap has a value in the range between 0 and 254 proportional with the probability of collision with obstacles.

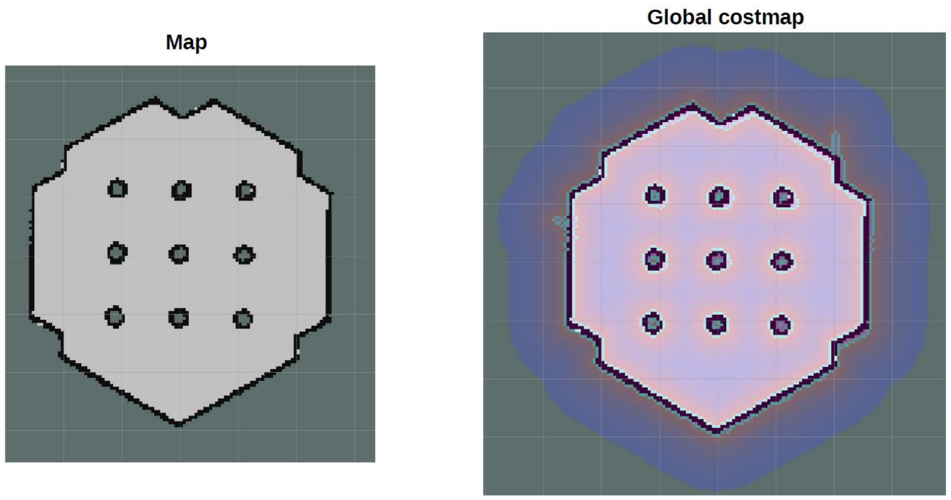
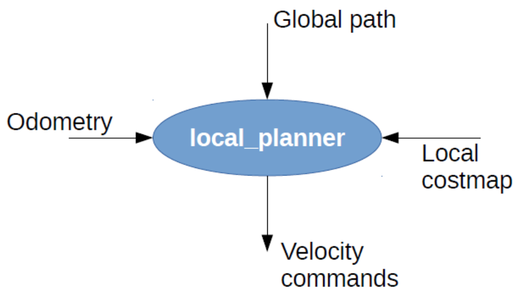


Figure 2.10: map and global costmap comparison.

- local planner: is the motion controller and follows the global path providing velocity commands to the mobile base of the robot. Avoidance of obstacle is performed by means of the local costmap. Uses odometry data as a feedback of the motion control. Publishes on a topic of type geometry_msgs/Twist. It receives the global path, the odometry and the local costmap to provide the actual velocity commands. In the presence of an obstacle within the global path, it provides a new trajectory able to avoid collisions.



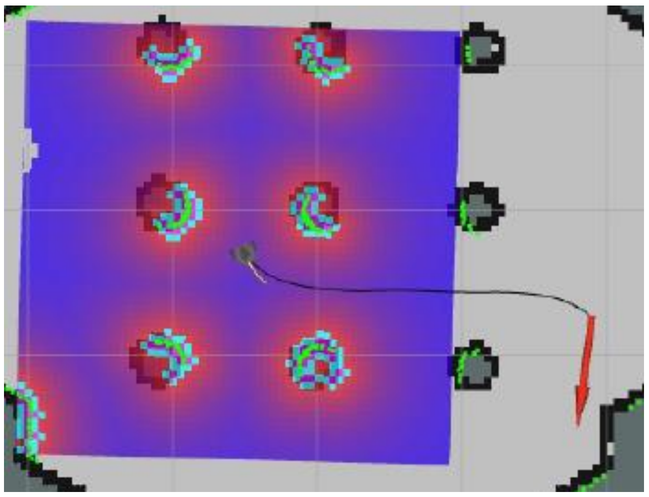


Figure 2.11: example of local and global planner.

- local costmap: it is initialized with the information from the global costmap and defined around the robot as a dynamic window. It is updated with sensor data and uses the same logic as the global costmap, but with a higher frequency (which implies more computational effort, i.e. mandatory lower dimensions).

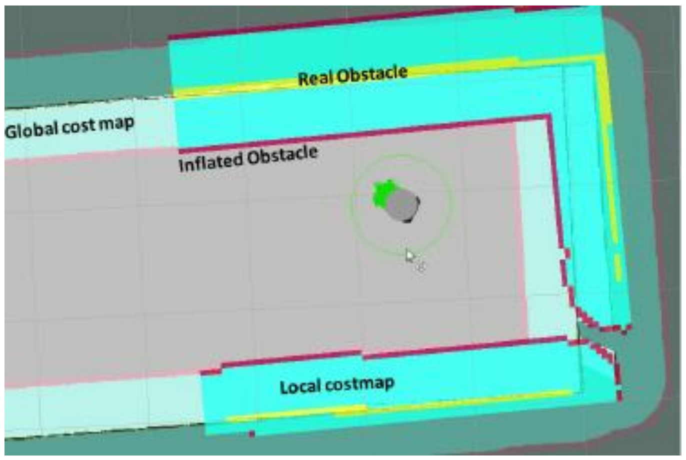


Figure 2.12: local costmap example.

- recovery behaviours: executed in case of robot stucking when it is not able to follow the global path. It performs a series of actions to update the local and global costmaps. If it is followed by another stucking point, it aborts the current navigation.

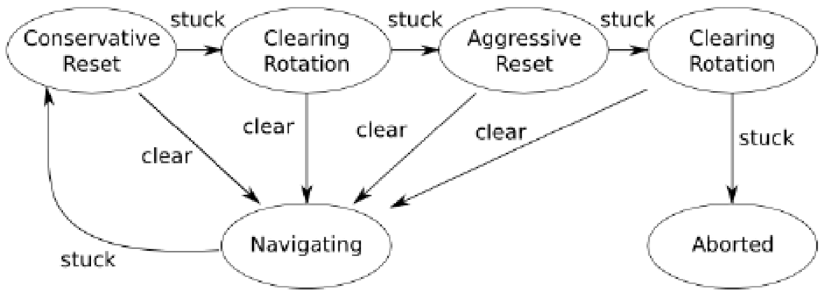


Figure 2.13: entire tasks performed by the robot behind the navigation goal. [2]

Navigation description

Launching the navigation simulation

To launch the navigation simulation it is necessary to launch two different files:

- turtlebot3_world.launch to launch the turtlebot3_world and to spawn the robot.

```
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

- turtlebot3_navigation.launch to launch all the nodes for the navigation stack.

```
$ roslaunch turtlebot3_navigation turtlebot3_navigation.launch
```

Once launched the simulation of 1 robot in Gazebo, the navigation stack allows to set a navigation goal that is reached by the robot after the computation of the fastest obstacle-free path.

Tf_tree analysis

The tf_tree graph is a graph that connect the involved reference frames and the topics that connect them. The analysis of the tf_tree is useful to understand which the relationships between the frames and the main topics are.

The map frame contains the map of the turtlebot3 world, and it is connected to the odom frame by means of the /amcl topic, responsible for the transformation of the reference frame of the robot along time. The transformation is a dynamic transformation since it evolves during time. Then, the odom frame is connected to the base_footprint frame which is responsible for the reference frame of the robot projected in the plane z=0 (viewed from the top). Such connection is necessary since the odom frame is fixed in the initial position of the robot, whereas base_footprint moves with the robot itself.

The last relevant connection is between the base_footprint and the base_link. The base_link frame is the frame of the base of the robot, defined as a consequence of its structure.

All the other frames are defined with respect to the base_link, which correspond to the main reference frame of the robot.

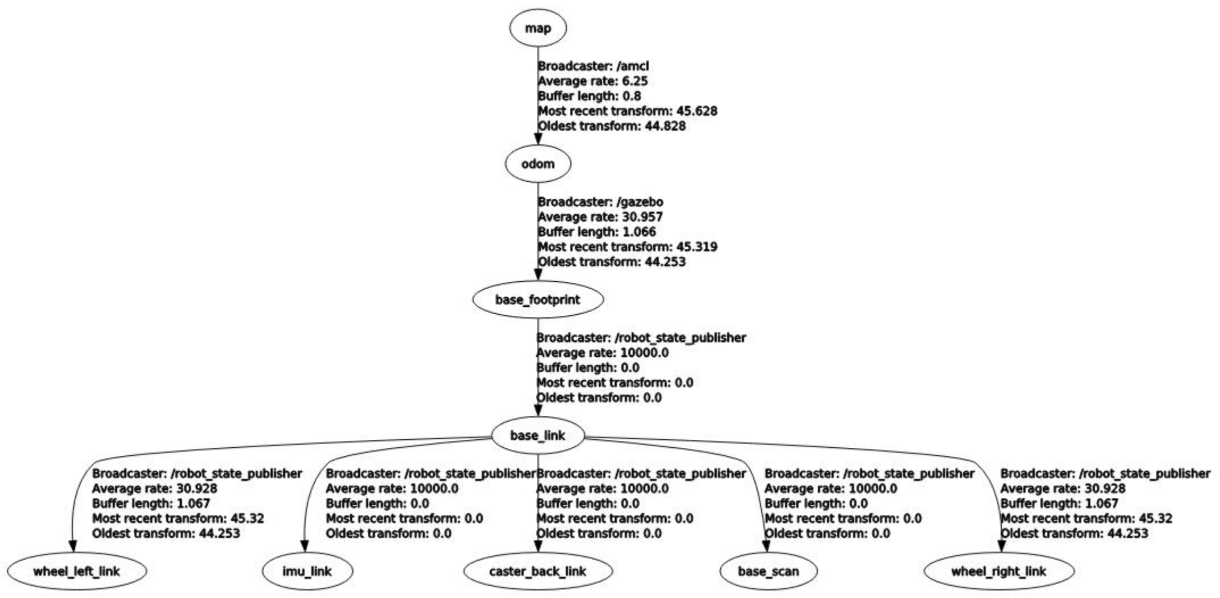


Figure 3.1: tf_tree of the navigation of a single robot on a single machine.

Relevant topics

The relevant topics for the navigation are mainly:

- /map: it provides the map of the world where the simulation takes place.
- /initialpose: responsible for the definition of the initial pose of the robot.
- /odom: necessary to correctly map the odom frame to the map one.
- /amcl: responsible for the probabilistic estimation of the robot position.
- /move_base_simple/goal: necessary to send the navigation goal.
- /cmd_vel: necessary to feed the robot with the correct velocity inputs.

Launch files modification for the multiple navigation

The primary performed step is the multi robot simulation and cooperation on the same computer. The purposes attainment passes through the modification of the Gazebo launch file and the navigation one. As a further step after the comprehension of the ROS working principle, the files are rearranged in order to have a multi robot simulation on more computers connected together (as explained in the following paragraphs). The launch files in ROS provide a convenient way to start up multiple node and a master, as well as other initialization requirements such as setting parameters.

Gazebo simulation launch file

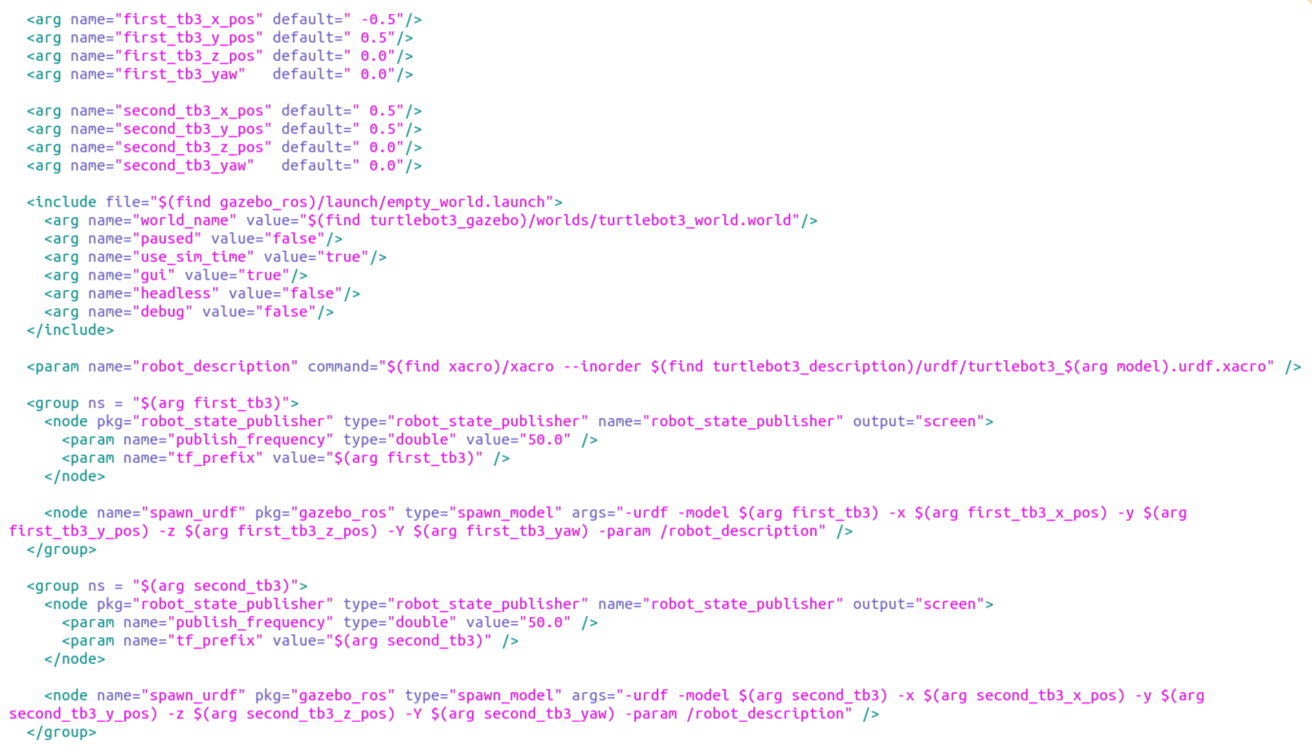
The launch files in ROS are in the format of .launch and they use an XML format. They can be placed anywhere within a package directory, but it is common to make a "Launch" directory inside the workspace directory. The modified file is called turtlebot3_world_multi.launch and the script is shown in figure below.

The script is organized as follow:

- some parameters are defined by using the <arg name="..." value="..." /> syntax, since it is necessary to use local variables in launch files;
- the *robot description* allows to define the used model for the simulation, and it must be declared outside from the namespaces, since both robots are considered as burger model;
- the two namespaces are constructed by means of the syntax:

  <group ns="...">
  </group>
  and in each one some topics are stated in order to guarantee the multi robot simulation;
- the *tf_prefix* is specified in the correspondent namespace. It keeps track of the transformation between the reference frame and the robot frame, so it must be reported for each robot;
- each robot is spawned in the desired position given by the starting declared arguments. The execution of the final step allows to create directly the robot into the Gazebo environment, by exploiting the syntax:

  <node name="spawn_urdf" />



```
  <arg name="first_tb3_x_pos" default=" -0.5"/>
  <arg name="first_tb3_y_pos" default=" 0.5"/>
  <arg name="first_tb3_z_pos" default=" 0.0"/>
  <arg name="first_tb3_yaw"   default=" 0.0"/>

  <arg name="second_tb3_x_pos" default=" 0.5"/>
  <arg name="second_tb3_y_pos" default=" 0.5"/>
  <arg name="second_tb3_z_pos" default=" 0.0"/>
  <arg name="second_tb3_yaw"   default=" 0.0"/>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find turtlebot3_gazebo)/worlds/turtlebot3_world.world"/>
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="debug" value="false"/>
  </include>

  <param name="robot_description" command="$(find xacro)/xacro --inorder $(find turtlebot3_description)/urdf/turtlebot3_$(arg model).urdf.xacro" />

  <group ns = "$(arg first_tb3)">
    <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher" output="screen">
      <param name="publish_frequency" type="double" value="50.0" />
      <param name="tf_prefix" value="$(arg first_tb3)" />
    </node>

    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-urdf -model $(arg first_tb3) -x $(arg first_tb3_x_pos) -y $(arg
first_tb3_y_pos) -z $(arg first_tb3_z_pos) -Y $(arg first_tb3_yaw) -param /robot_description" />
  </group>

  <group ns = "$(arg second_tb3)">
    <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher" output="screen">
      <param name="publish_frequency" type="double" value="50.0" />
      <param name="tf_prefix" value="$(arg second_tb3)" />
    </node>

    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-urdf -model $(arg second_tb3) -x $(arg second_tb3_x_pos) -y $(arg
second_tb3_y_pos) -z $(arg second_tb3_z_pos) -Y $(arg second_tb3_yaw) -param /robot_description" />
  </group>
```

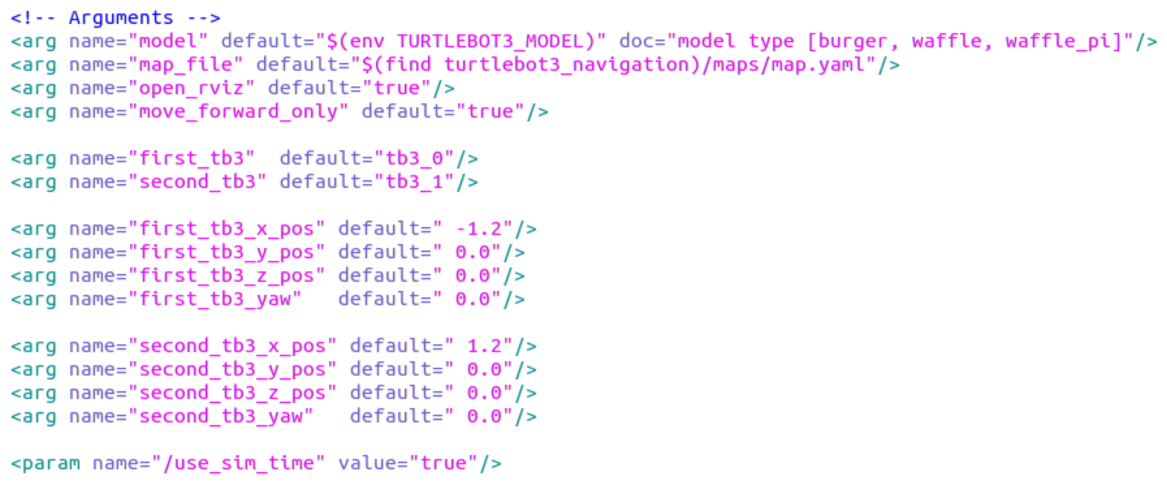
Figure 4.1: Gazebo launch file modification for the simulation of 2 robots on a single machine. The file is called turtlebot3_world_multi.launch.

Navigation launch file

The second launch file to be modified is the navigation one, in particular the file turtlebot3_navigation_multi.launch. The original path of the file is turtlebot3_navigation/launch. As shown in the text of the file, figure 4.2 and 4.3, the content of the modified file is organized in order to run correctly the navigation with two burger model robots, called “tb3_0” and “tb3_1”.

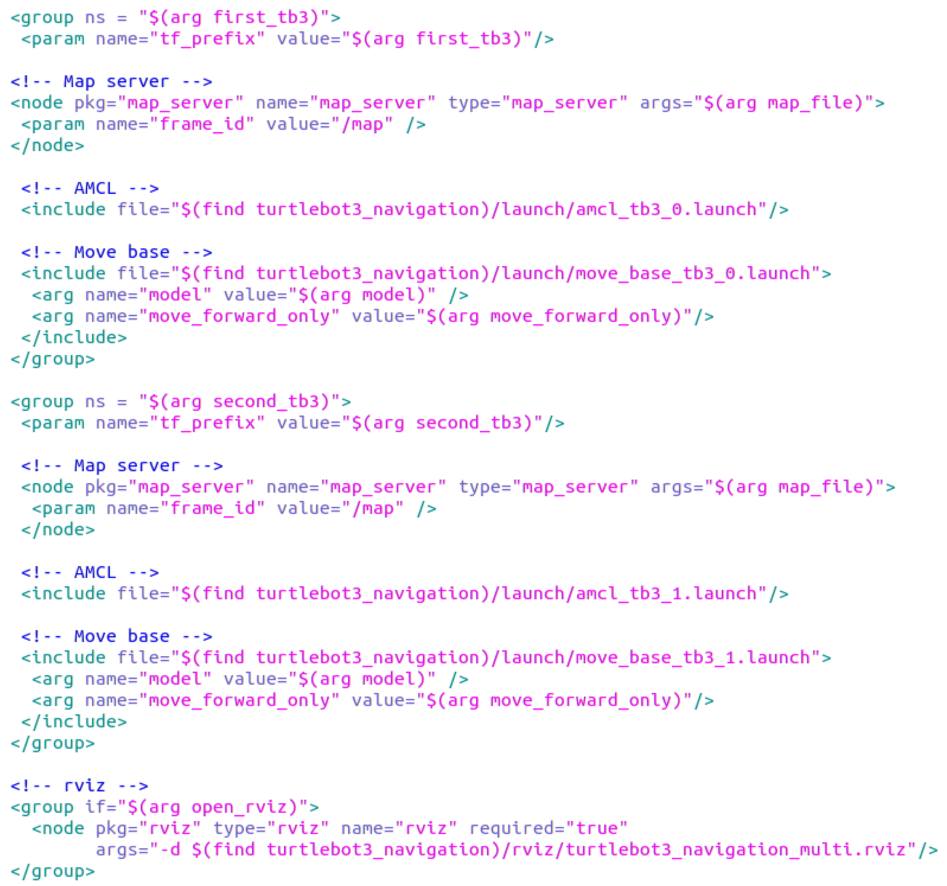
In the **Arguments** section:

“tb3_0” and “tb3_1” are assigned as default values, respectively to arguments “first_tb3” and “second_tb3”. This specification is a key step in order to map correctly everything related to the two different robots.
For each of the two robots the initial default pose is defined assigning to the x position different and opposite values, 1.2 for tb3_0 and -1.2 for tb3_1.
In order to synchronize the simulations running on both RViz and Gazebo, the parameter “/use_sim_time” is set to “true” value. In this way the two virtual environments are updated at the same time.



```
<!-- Arguments -->
<arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
<arg name="map_file" default="$(find turtlebot3_navigation)/maps/map.yaml"/>
<arg name="open_rviz" default="true"/>
<arg name="move_forward_only" default="true"/>

<arg name="first_tb3"  default="tb3_0"/>
<arg name="second_tb3" default="tb3_1"/>

<arg name="first_tb3_x_pos" default=" -1.2"/>
<arg name="first_tb3_y_pos" default=" 0.0"/>
<arg name="first_tb3_z_pos" default=" 0.0"/>
<arg name="first_tb3_yaw"   default=" 0.0"/>

<arg name="second_tb3_x_pos" default=" 1.2"/>
<arg name="second_tb3_y_pos" default=" 0.0"/>
<arg name="second_tb3_z_pos" default=" 0.0"/>
<arg name="second_tb3_yaw"   default=" 0.0"/>

<param name="/use_sim_time" value="true"/>
```

Figure 4.2: turtlebot3_navigation_multi.launch file, argument and parameters definition

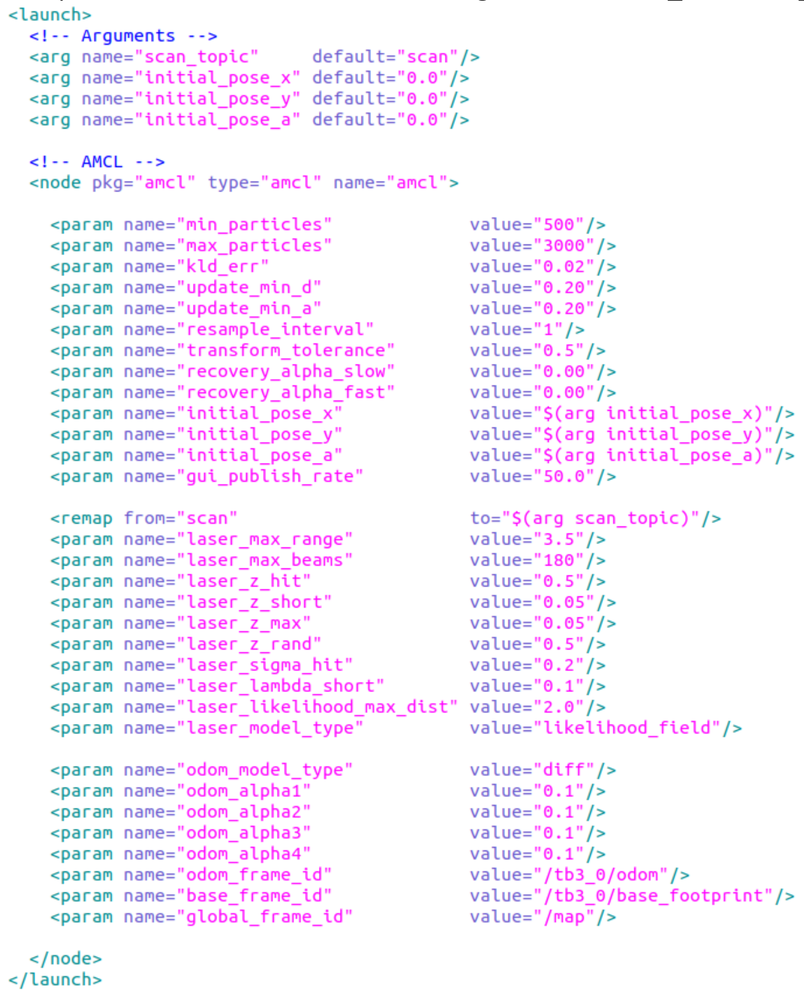


```
<group ns = "$(arg first_tb3)">
 <param name="tf_prefix" value="$(arg first_tb3)"/>

<!-- Map server -->
<node pkg="map_server" name="map_server" type="map_server" args="$(arg map_file)">
 <param name="frame_id" value="/map" />
</node>

 <!-- AMCL -->
 <include file="$(find turtlebot3_navigation)/launch/amcl_tb3_0.launch"/>

 <!-- Move base -->
 <include file="$(find turtlebot3_navigation)/launch/move_base_tb3_0.launch">
  <arg name="model" value="$(arg model)" />
  <arg name="move_forward_only" value="$(arg move_forward_only)"/>
 </include>
</group>

<group ns = "$(arg second_tb3)">
 <param name="tf_prefix" value="$(arg second_tb3)"/>

 <!-- Map server -->
 <node pkg="map_server" name="map_server" type="map_server" args="$(arg map_file)">
  <param name="frame_id" value="/map" />
 </node>

 <!-- AMCL -->
 <include file="$(find turtlebot3_navigation)/launch/amcl_tb3_1.launch"/>

 <!-- Move base -->
 <include file="$(find turtlebot3_navigation)/launch/move_base_tb3_1.launch">
  <arg name="model" value="$(arg model)" />
  <arg name="move_forward_only" value="$(arg move_forward_only)"/>
 </include>
</group>

<!-- rviz -->
<group if="$(arg open_rviz)">
  <node pkg="rviz" type="rviz" name="rviz" required="true"
        args="-d $(find turtlebot3_navigation)/rviz/turtlebot3_navigation_multi.rviz"/>
</group>
```

Figure 4.3: group namespace for both tb3_0 and tb3_1 robots in turtlebot3_navigation_multi.launch file

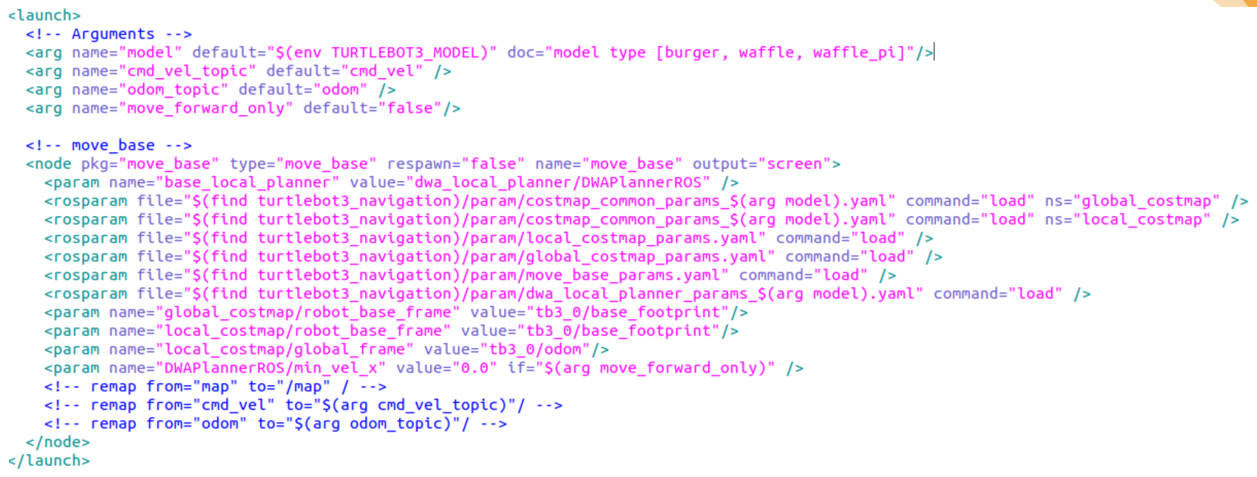
It is important to remark that for each robot is used a specific namespace, identified by the “tb3_0” and “tb3_1”. It is fundamental to define the parameter “**tf_prefix**” assigning it the value of the respective robot identification (“tb3_0” and “tb3_1”, to avoid conflicts). Inside the single namespace the necessary files and parameters are included or defined. In particular, these are mainly related to:

- **Map server:** define the parameter “frame_id” with the value “/map” in order to select the common map for the single robot frame. Here the node “map_server” is called taking the argument “map_file” from the “map.yaml” file. This last text file containing a list of parameters is discovered to be crucial for the working condition of the whole multiple navigation tool, as described later in the bug solution paragraph.
- **AMCL:** include the launch file of the single burger bot (amcl_tb3_0.launch and amcl_tb3_1.launch), reported below in figure 4.4. Notice that the last three parameters in this file are modified specifying the address /tb3_0/ or /tb3_1/ for both the odom and the base_footprint, and giving the value “/map” to the global_frame_id.
- **Move base:** the single robot move_base launch file is included (move_base_tb3_0.launch and move_base_tb3_0.launch) in the multiple navigation one, with the associated arguments definition. Also in this case, the value of some parameters of interest, global_costmap and local_costmap, is specified with the address of the singular robot /tb3_0/ or /tb3_1/.



```
<launch>
  <!-- Arguments -->
  <arg name="scan_topic"     default="scan"/>
  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>

  <!-- AMCL -->
  <node pkg="amcl" type="amcl" name="amcl">

    <param name="min_particles"             value="500"/>
    <param name="max_particles"             value="3000"/>
    <param name="kld_err"                   value="0.02"/>
    <param name="update_min_d"              value="0.20"/>
    <param name="update_min_a"              value="0.20"/>
    <param name="resample_interval"         value="1"/>
    <param name="transform_tolerance"       value="0.5"/>
    <param name="recovery_alpha_slow"       value="0.00"/>
    <param name="recovery_alpha_fast"       value="0.00"/>
    <param name="initial_pose_x"            value="$(arg initial_pose_x)"/>
    <param name="initial_pose_y"            value="$(arg initial_pose_y)"/>
    <param name="initial_pose_a"            value="$(arg initial_pose_a)"/>
    <param name="gui_publish_rate"          value="50.0"/>

    <remap from="scan"                      to="$(arg scan_topic)"/>
    <param name="laser_max_range"           value="3.5"/>
    <param name="laser_max_beams"           value="180"/>
    <param name="laser_z_hit"               value="0.5"/>
    <param name="laser_z_short"             value="0.05"/>
    <param name="laser_z_max"               value="0.05"/>
    <param name="laser_z_rand"              value="0.5"/>
    <param name="laser_sigma_hit"           value="0.2"/>
    <param name="laser_lambda_short"        value="0.1"/>
    <param name="laser_likelihood_max_dist" value="2.0"/>
    <param name="laser_model_type"          value="likelihood_field"/>

    <param name="odom_model_type"           value="diff"/>
    <param name="odom_alpha1"               value="0.1"/>
    <param name="odom_alpha2"               value="0.1"/>
    <param name="odom_alpha3"               value="0.1"/>
    <param name="odom_alpha4"               value="0.1"/>
    <param name="odom_frame_id"             value="/tb3_0/odom"/>
    <param name="base_frame_id"             value="/tb3_0/base_footprint"/>
    <param name="global_frame_id"           value="/map"/>

  </node>
</launch>
```

Figure 4.4: amcl_tb3_0 launch file.



```
<launch>
  <!-- Arguments -->
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="cmd_vel_topic" default="cmd_vel" />
  <arg name="odom_topic" default="odom" />
  <arg name="move_forward_only" default="false"/>

  <!-- move_base -->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
    <param name="base_local_planner" value="dwa_local_planner/DWAPlannerROS" />
    <rosparam file="$(find turtlebot3_navigation)/param/costmap_common_params_$(arg model).yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find turtlebot3_navigation)/param/costmap_common_params_$(arg model).yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find turtlebot3_navigation)/param/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find turtlebot3_navigation)/param/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find turtlebot3_navigation)/param/move_base_params.yaml" command="load" />
    <rosparam file="$(find turtlebot3_navigation)/param/dwa_local_planner_params_$(arg model).yaml" command="load" />
    <param name="global_costmap/robot_base_frame" value="tb3_0/base_footprint"/>
    <param name="local_costmap/robot_base_frame" value="tb3_0/base_footprint"/>
    <param name="local_costmap/global_frame" value="tb3_0/odom"/>
    <param name="DWAPlannerROS/min_vel_x" value="0.0" if="$(arg move_forward_only)" />
    <!-- remap from="map" to="/map" / -->
    <!-- remap from="cmd_vel" to="$(arg cmd_vel_topic)"/ -->
    <!-- remap from="odom" to="$(arg odom_topic)"/ -->
  </node>
</launch>
```

Figure 4.5: move_base_tb3_0 launch file

RViz configuration file

The RViz configuration file are modified for two main reasons:

1. The standard file considers only one robot;

2. All the topics have a wrong path, since in the single simulation there are no namespaces, for instance /odom instead of /tb3_0/odom.

The main modifications are:

- The addition of a 2D pose estimate botton for the second robot /tb3_1/initialpose (and a remapping of the first one /tb3_0/initialpose);
- The addition of a 2D navigation goal botton for the second robot /tb3_1/move_base_simple/goal (and a remapping of the first one /tb3_0/move_base_simple/goal), to be able to specify which robot needs to go to a target;
- The addition of a third 2D navigation goal button associated with a new topic /move_base_simple/goal where the navigation goal is saved for both robots.

Bug solution

The /odom topic associated with each robot is not connected to the global map topic (/map) by default. Forcing the connection with a static transformation is wrong, as the transformation need to be dynamic.

```
  <node pkg="tf" type="static_transform_publisher" name="amcl_0"  args="0 0 0 0 0 0 /map /$(arg first_tb3)/odom 1000"/>

  <node pkg="tf" type="static_transform_publisher" name="amcl_1"  args="0 0 0 0 0 0 /map /$(arg second_tb3)/odom 1000"/>
```

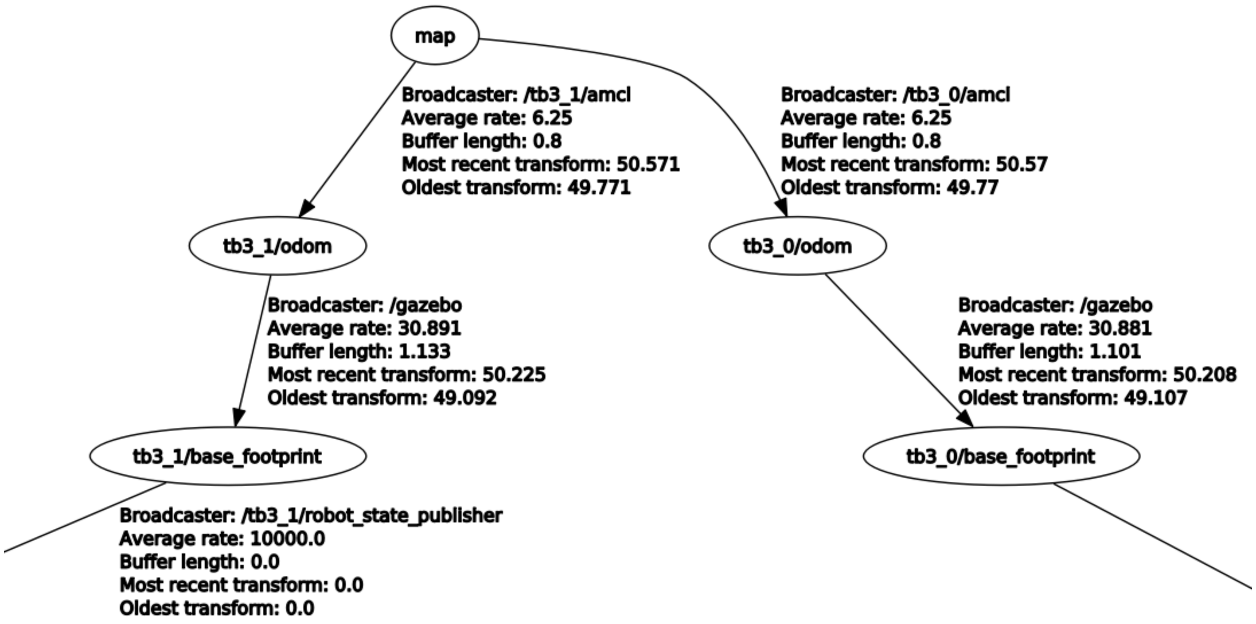


Figure 4.6: connection detail of the map frame and the odom one.

The error coming from the simulation is the following one:

```
[ WARN] [1556717553.387544930, 203.264000000]: Timed out waiting for transform from base_footprint to map to become available before running costmap, tf error: canTransform: target_frame map does not exist.. canTransform returned after 203.264 timeout was 0.1
```

By analyzing the tf_tree shape and the error message, the problem has to be related with the amcl or move_base launch files.

First of all, within the amcl file, a parameter is missing:

```
<param name="global_frame_id" value="/map"/>
```

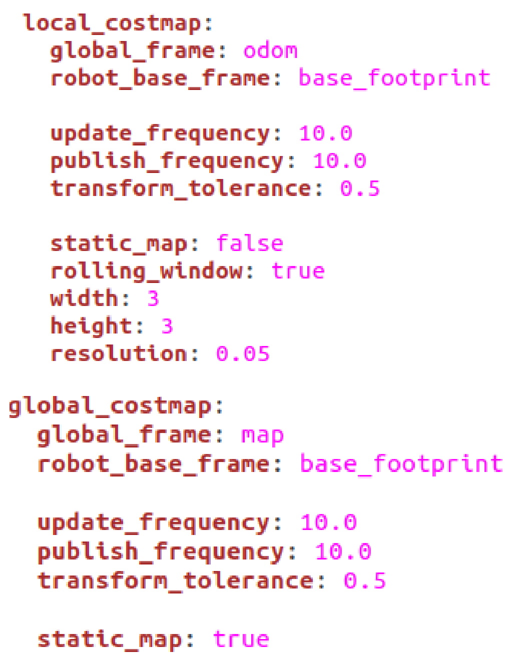
The *global_frame_id* parameter is necessary when dealing with multiple machines on the same map. In such conditions the map is one and they do not have to refer to their own map (tb3_0/map and tb3_1/map), but they must be related to the global one.

Nevertheless, this is not the main problem, since the error is still present. After several attempts, the solution is found in the definition of some parameters inside the two parameter files.

global_costmap_params.yaml

local_costmap_params.yaml

In such files, parameters are set in a wrong way by default.



```
local_costmap:
  global_frame: odom
  robot_base_frame: base_footprint

  update_frequency: 10.0
  publish_frequency: 10.0
  transform_tolerance: 0.5

  static_map: false
  rolling_window: true
  width: 3
  height: 3
  resolution: 0.05

global_costmap:
  global_frame: map
  robot_base_frame: base_footprint

  update_frequency: 10.0
  publish_frequency: 10.0
  transform_tolerance: 0.5

  static_map: true
```

Figure 4.7: default parameters in *global_costmap_params.yaml* and *local_costmap_params.yaml*.

By default, the topic odom and base_footprint are /odom and /base_footprint. Anyhow, they are not present in our tf_free, as we defined two namespaces for the two robots. As a consequence, it is necessary to remap such values for each robot.

Our choice is to create two amcl and move_base files, one for each robot, so that in each of them the right topic is addressed. Obviously, as mentioned before, the map topic in global_costmap does not need to be modified since the map is only one and it is necessary to have /map as topic.

For instance, in the move_base_tb3_0 file we added the following lines to perform the correct remap:

```
<param name="global_costmap/robot_base_frame" value="tb3_0/base_footprint"/>

<param name="local_costmap/robot_base_frame" value="tb3_0/base_footprint"/>

<param name="local_costmap/global_frame" value="tb3_0/odom"/>
```

In such way all the global and local costmaps have the right topics and can work properly.

Hence, the correct tf_tree is shown in the following figure:

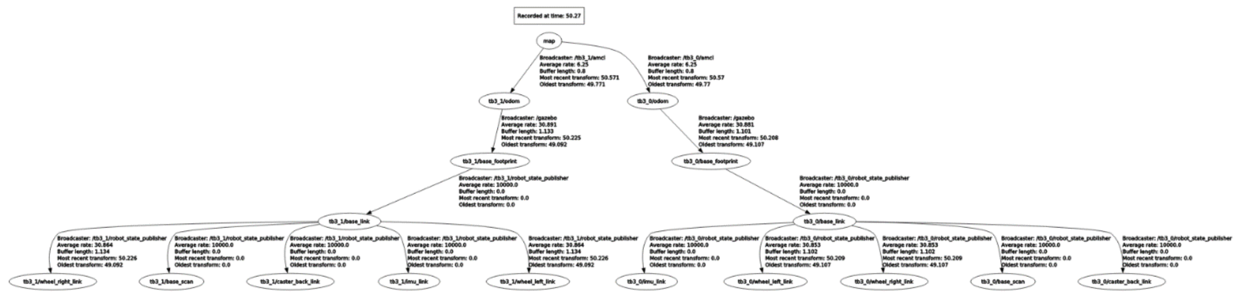

Recorded at time: 50.27
map
Broadcaster: /tb3_1/amcl Average rate: 6.25 Buffer length: 0.8 Most recent transform: 50.571 Oldest transform: 49.771
Broadcaster: /tb3_0/amcl Average rate: 6.25 Buffer length: 0.8 Most recent transform: 50.57 Oldest transform: 49.77
tb3_1/odom
tb3_0/odom
Broadcaster: /gazebo Average rate: 30.891 Buffer length: 1.133 Most recent transform: 50.225 Oldest transform: 49.092
Broadcaster: /gazebo Average rate: 30.881 Buffer length: 1.101 Most recent transform: 50.208 Oldest transform: 49.107
tb3_1/base_footprint
tb3_0/base_footprint
Broadcaster: /tb3_1/robot_state_publisher Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: /tb3_0/robot_state_publisher Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
tb3_1/base_link
tb3_0/base_link
Broadcaster: /tb3_1/robot_state_publisher Average rate: 30.864 Buffer length: 1.134 Most recent transform: 50.226 Oldest transform: 49.092
Broadcaster: /tb3_1/robot_state_publisher Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: /tb3_1/robot_state_publisher Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: /tb3_1/robot_state_publisher Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: /tb3_1/robot_state_publisher Average rate: 30.864 Buffer length: 1.134 Most recent transform: 50.226 Oldest transform: 49.092
Broadcaster: /tb3_0/robot_state_publisher Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: /tb3_0/robot_state_publisher Average rate: 30.853 Buffer length: 1.102 Most recent transform: 50.209 Oldest transform: 49.107
Broadcaster: /tb3_0/robot_state_publisher Average rate: 30.853 Buffer length: 1.102 Most recent transform: 50.209 Oldest transform: 49.107
Broadcaster: /tb3_0/robot_state_publisher Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: /tb3_0/robot_state_publisher Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
tb3_1/wheel_right_link
tb3_1/base_scan
tb3_1/caster_back_link
tb3_1/imu_link
tb3_1/wheel_left_link
tb3_0/imu_link
tb3_0/wheel_left_link
tb3_0/wheel_right_link
tb3_0/base_scan
tb3_0/caster_back_link

Figure 4.8: correct tf_tree of the multiple robot's navigation.

RViz file modification to solve the bug

1. Topics related to the navigation are all set for a single robot, therefore the namespace before the topic is missing, for instance the laser scan is /scan, but in our simulation, there are only /tb3_0/scan and /tb3_0/scan. Each topic has to be remapped accordingly to our tf_tree.

2. It is necessary to add all the parameters for the second robot, which are not present by default. They have been added into the RViz configuration file inside the correct place, copying the structure from the first one and rearranging it in terms of topic path.

3. Lastly the 2D pose estimate and 2D navigation goal has to be remapped: 2D pose estimate with tb3_0/InitialPose and tb3_1/InitialPose, whereas 2D navigation goal with tb3_0/move_base_simple/goal and tb3_1/move_base_simple/goal.

What is possible to do? On the same map there are two robots that can be launched with a navigation goal concurrently (after having set the initial pose for both).

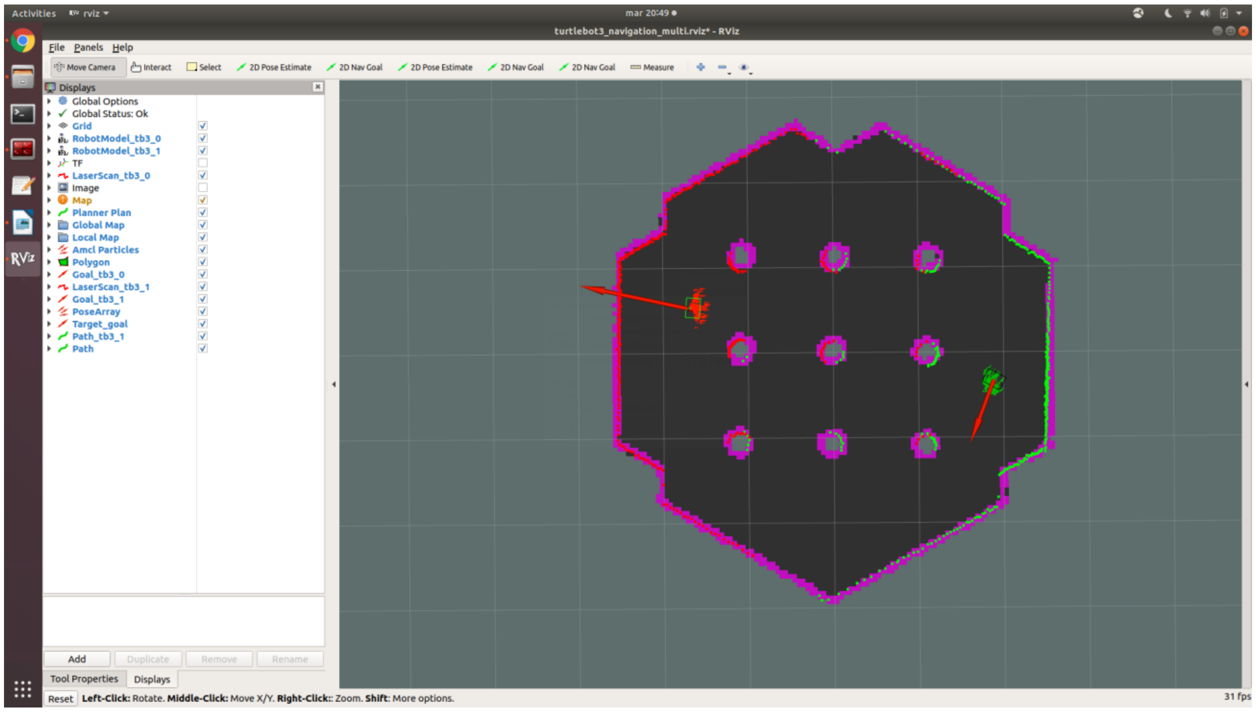


Figure 4.9: RViz UI modifications.

Python node

Node objective

The main purpose of the node is to read a navigation goal defined in RViz, understand which the closest robot is and send it to that goal. Such node could be developed either in Python, either in C++. For brevity, Python is chosen.

The idea behind the script

The working principle of the node is rather simple: the navigation goal is read by means of a function, as well as the two poses of the robots, then, by computing the difference in terms of ‘x’ and ‘y’ coordinates, the actual navigation goal is published only on the topic related to the closest robot.

Node description

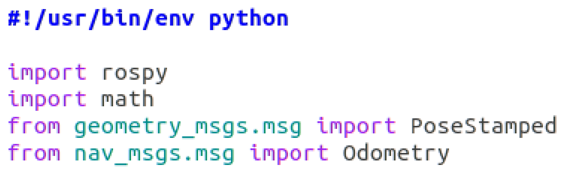
The python script is placed within the directory:

```
~/catkin_ws/src/navigation_package/scripts
```

Where navigation_package is a new package created from scratch for this project.

Libraries

The first part of the script is necessary to import all the libraries to allow the code to work. *Rospy* is necessary to develop a ROS node in Python, whereas *math* is necessary to perform some operations to understand which the closest robot to the navigation goal is. *PoseStamped* includes all the parameters to define a navigation goal, whereas *Odometry* contains all the parameters to define the pose of a robot.



```python
#!/usr/bin/env python

import rospy
import math
from geometry_msgs.msg import PoseStamped
from nav_msgs.msg import Odometry
```

Figure 5.1: code explanation: libraries import.

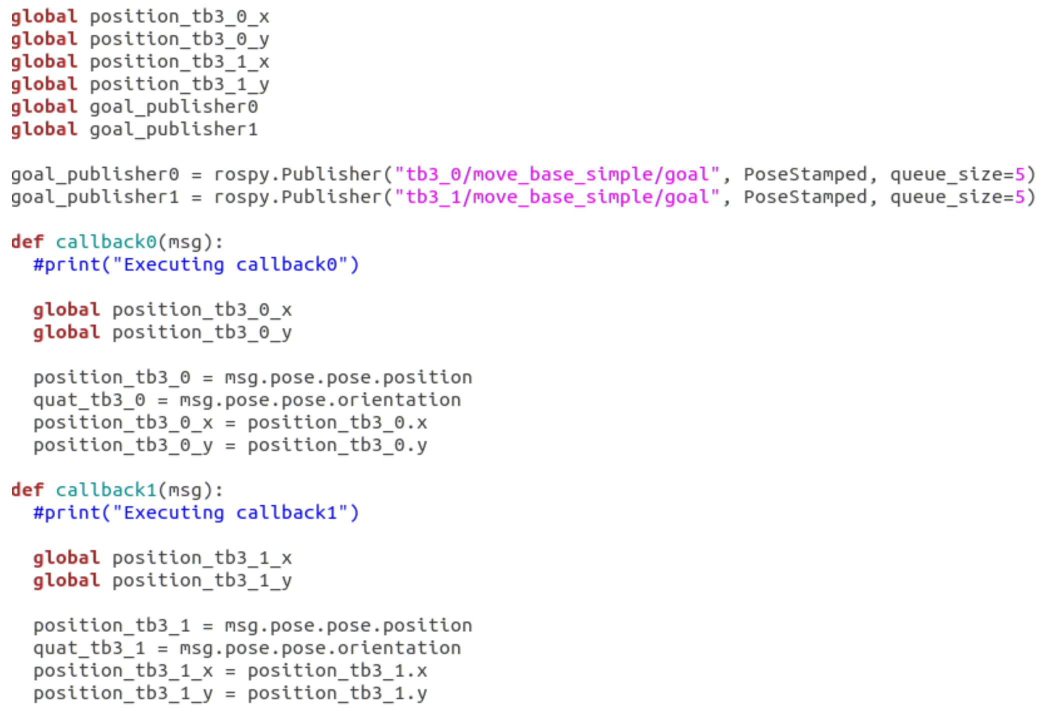
Functions to read the robot pose

Once included all the necessary libraries, it is possible to define the functions to read the robot pose.

Such functions are called *“callback0”* and *“callback1”* and have as main goal to read the pose of the robot which is present in the /tb3_0/odom and /tb3_1/odom topics, depending on the considered robot.

Both functions receive as argument a message (msg), which is an *Odometry* type taken from the robot’s odom topic, as explained later. In this message there are all the needed information to find the pose of the robots. The ‘x’ and ‘y’ coordinates of each robot are saved in a suitable global variable.

In order to be able to use the ‘x’ and ‘y’ position of the pose outside from the function, it is necessary to define such variables as global. Moreover, also the publisher variables must be declared as global variable to guarantee a sufficient amount of time before the creation of the goal and its publication (see “Second code version” for the detailed explanation).



```
global position_tb3_0_x
global position_tb3_0_y
global position_tb3_1_x
global position_tb3_1_y
global goal_publisher0
global goal_publisher1

goal_publisher0 = rospy.Publisher("tb3_0/move_base_simple/goal", PoseStamped, queue_size=5)
goal_publisher1 = rospy.Publisher("tb3_1/move_base_simple/goal", PoseStamped, queue_size=5)

def callback0(msg):
  #print("Executing callback0")

  global position_tb3_0_x
  global position_tb3_0_y

  position_tb3_0 = msg.pose.pose.position
  quat_tb3_0 = msg.pose.pose.orientation
  position_tb3_0_x = position_tb3_0.x
  position_tb3_0_y = position_tb3_0.y

def callback1(msg):
  #print("Executing callback1")

  global position_tb3_1_x
  global position_tb3_1_y

  position_tb3_1 = msg.pose.pose.position
  quat_tb3_1 = msg.pose.pose.orientation
  position_tb3_1_x = position_tb3_1.x
  position_tb3_1_y = position_tb3_1.y
```

Figure 5.2: code explanation: callback0 and callback1 functions.

Function to send a goal

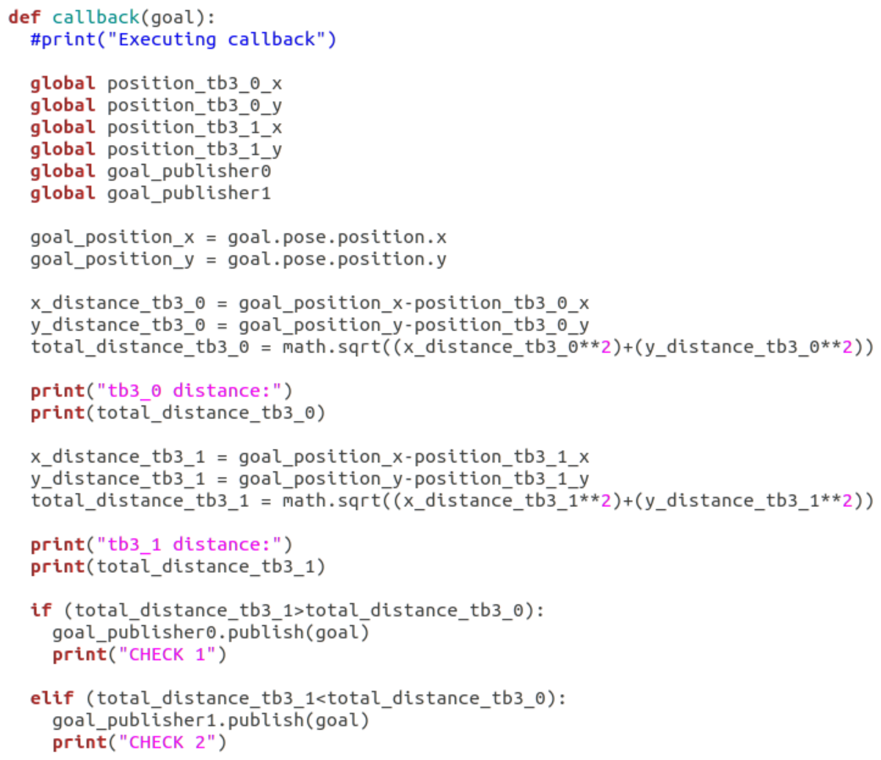
Once got the two poses, it is possible to read the navigation goal and find which is the closest robot. The navigation goal is set manually in RViz by means of a new button created from the scratch for this purpose. Such navigation goal is set to the topic /move_base_simple/goal, which is associated with no robot, since they are in a specific namespace:

/tb3_0/move_base_simple/goal

/tb3_1/move_base_simple/goal

The *callback* function receives a message as input (goal) which is a *PoseStamped* type taken from the topic /move_base_simple/goal. Inside the goal message there are the ‘x’ and ‘y’ coordinates of the navigation goal. Such coordinates are compared with the two poses, then the closest one is found by means of the Pythagorean theorem.

To actually send the goal to one of the two robots, it is necessary to publish the goal to the correct topic: if the robot tb3_0 is the nearest, then the goal is published to /tb3_0/move_base_simple/goal; by contrast, if the robot tb3_1 is the closest, then the goal is published to /tb3_1/move_base_simple/goal.



```
def callback(goal):
  #print("Executing callback")

  global position_tb3_0_x
  global position_tb3_0_y
  global position_tb3_1_x
  global position_tb3_1_y
  global goal_publisher0
  global goal_publisher1

  goal_position_x = goal.pose.position.x
  goal_position_y = goal.pose.position.y

  x_distance_tb3_0 = goal_position_x-position_tb3_0_x
  y_distance_tb3_0 = goal_position_y-position_tb3_0_y
  total_distance_tb3_0 = math.sqrt((x_distance_tb3_0**2)+(y_distance_tb3_0**2))

  print("tb3_0 distance:")
  print(total_distance_tb3_0)

  x_distance_tb3_1 = goal_position_x-position_tb3_1_x
  y_distance_tb3_1 = goal_position_y-position_tb3_1_y
  total_distance_tb3_1 = math.sqrt((x_distance_tb3_1**2)+(y_distance_tb3_1**2))

  print("tb3_1 distance:")
  print(total_distance_tb3_1)

  if (total_distance_tb3_1>total_distance_tb3_0):
    goal_publisher0.publish(goal)
    print("CHECK 1")

  elif (total_distance_tb3_1<total_distance_tb3_0):
    goal_publisher1.publish(goal)
    print("CHECK 2")
```

Figure 5.3: code explanation: callback function.

Main

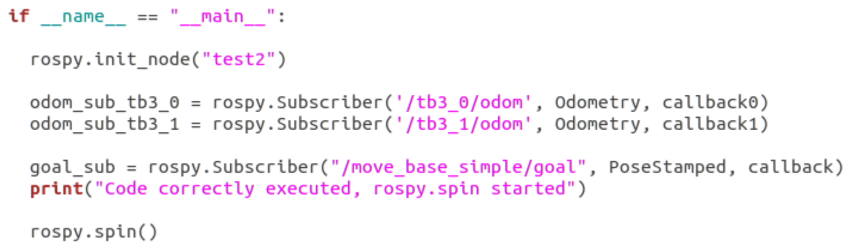
The main contains the node initialization and the subscription function to the interested topics.

The code line `odom_sub_tb3_0 = rospy.Subscriber('/tb3_0/odom', Odometry, callback0)` subscribes to the /tb3_0/odom topic, which contains an Odometry message that is sent to the function *callback0* (pose of the robot tb3_0).

The code line `odom_sub_tb3_1 = rospy.Subscriber('/tb3_1/odom', Odometry, callback0)` subscribes to the /tb3_1/odom topic, which contains an Odometry message that is sent to the function *callback1* (pose of the robot tb3_1).

The code line `goal_sub = rospy.Subscriber("/move_base_simple/goal", PoseStamped, callback)` subscribes to the topic which contains a PoseStamped message sent to the function *callback* (navigation goal).

In the end, *rospy.spin* allows to keep the node running until explicitly killed. Such condition allows to keep publishing target while running RViz.



```
if __name__ == "__main__":

  rospy.init_node("test2")

  odom_sub_tb3_0 = rospy.Subscriber('/tb3_0/odom', Odometry, callback0)
  odom_sub_tb3_1 = rospy.Subscriber('/tb3_1/odom', Odometry, callback1)

  goal_sub = rospy.Subscriber("/move_base_simple/goal", PoseStamped, callback)
  print("Code correctly executed, rospy.spin started")

  rospy.spin()
```

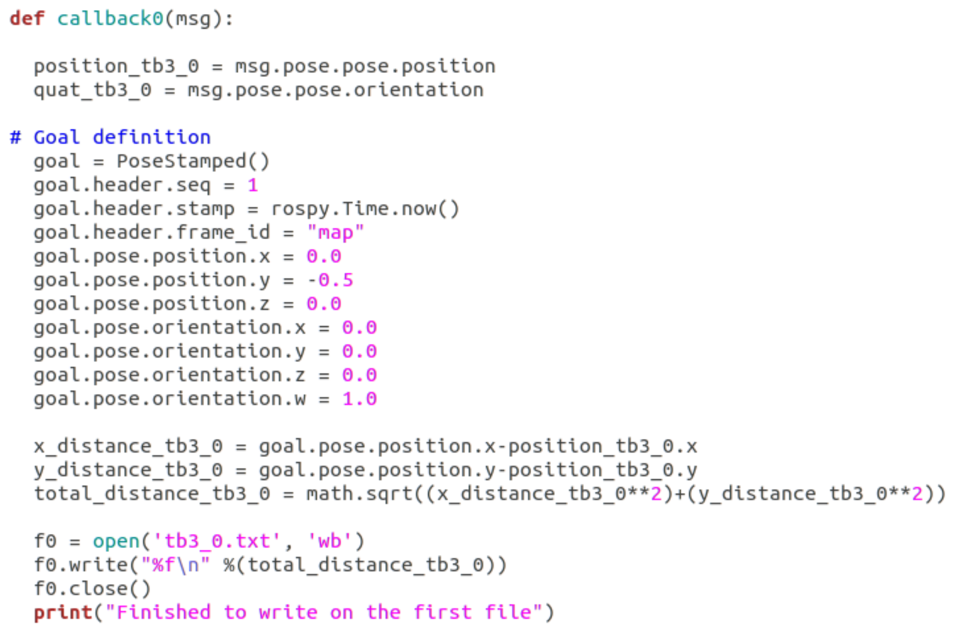
Figure 5.4: code explanation: main.

First code version

The first code version is badly implemented due to two main reasons:

1. Incapability of using *global variables*;
2. Goal publication performed manually inside the code.

The first problem is solved using a smart but slow system: the “global” variables, obtained from the desired *callback* function, are written on a text file. Afterward such values are taken again in order to exploit them into the computation of the robots distance from the goal position.



```python
def callback0(msg):

  position_tb3_0 = msg.pose.pose.position
  quat_tb3_0 = msg.pose.pose.orientation

# Goal definition
  goal = PoseStamped()
  goal.header.seq = 1
  goal.header.stamp = rospy.Time.now()
  goal.header.frame_id = "map"
  goal.pose.position.x = 0.0
  goal.pose.position.y = -0.5
  goal.pose.position.z = 0.0
  goal.pose.orientation.x = 0.0
  goal.pose.orientation.y = 0.0
  goal.pose.orientation.z = 0.0
  goal.pose.orientation.w = 1.0

  x_distance_tb3_0 = goal.pose.position.x-position_tb3_0.x
  y_distance_tb3_0 = goal.pose.position.y-position_tb3_0.y
  total_distance_tb3_0 = math.sqrt((x_distance_tb3_0**2)+(y_distance_tb3_0**2))

  f0 = open('tb3_0.txt', 'wb')
  f0.write("%f\n" %(total_distance_tb3_0))
  f0.close()
  print("Finished to write on the first file")
```

Figure 5.5: code explanation: first attempt of the callback0 function.

The second problem is lack of ability to exploit the global variables: it is necessary to define three times the goal target (one for each function *callback0* and *callback1*, plus one for the main). Moreover, the presence of the goal within the code do not allow to edit a goal during the node execution; it is necessary to stop the simulation any time the goal has needed to be changed.

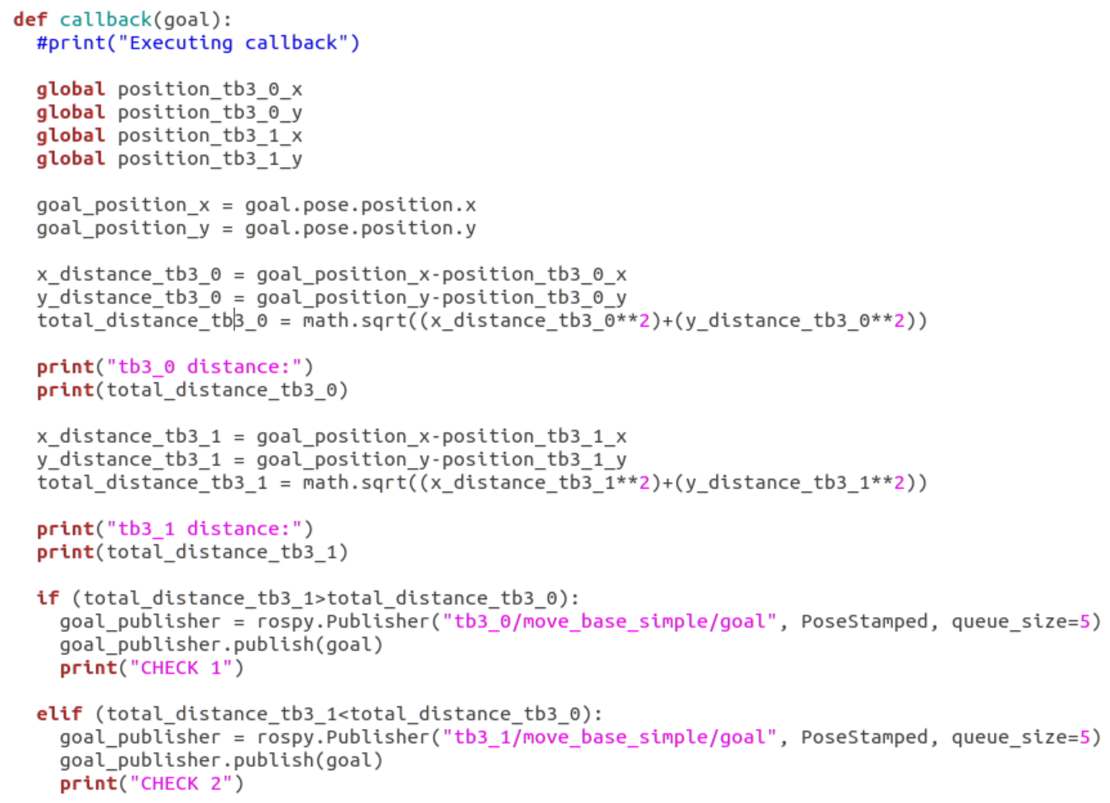
The updated version corrects both problems. The usage of global variables, defined in the main and in each function, allows to not use the external text files. The usage of a third function able to read the navigation goal in RViz and send it to the right robot allows to have a more efficient code, and to have a continuous simulation, namely allowing to set successive goals without killing the node each time.

Second code version

The second code version was very similar to the first one, but it had a boring bug: before being able to correctly run the node, it was necessary to publish a 2D navigation goal manually for the two robots. This fact was due to the insufficient time between the creation of the goal and its publication. The possible solutions were mainly 2:

1. Insert a sleep function of 0.25 seconds between the creation of the goal and the publication.

2. Define the goal as a global variable, so that it is created at the beginning of the code and there is a sufficient amount of time before the execution of the publishing line code.



```
def callback(goal):
  #print("Executing callback")

  global position_tb3_0_x
  global position_tb3_0_y
  global position_tb3_1_x
  global position_tb3_1_y

  goal_position_x = goal.pose.position.x
  goal_position_y = goal.pose.position.y

  x_distance_tb3_0 = goal_position_x-position_tb3_0_x
  y_distance_tb3_0 = goal_position_y-position_tb3_0_y
  total_distance_tb3_0 = math.sqrt((x_distance_tb3_0**2)+(y_distance_tb3_0**2))

  print("tb3_0 distance:")
  print(total_distance_tb3_0)

  x_distance_tb3_1 = goal_position_x-position_tb3_1_x
  y_distance_tb3_1 = goal_position_y-position_tb3_1_y
  total_distance_tb3_1 = math.sqrt((x_distance_tb3_1**2)+(y_distance_tb3_1**2))

  print("tb3_1 distance:")
  print(total_distance_tb3_1)

  if (total_distance_tb3_1>total_distance_tb3_0):
    goal_publisher = rospy.Publisher("tb3_0/move_base_simple/goal", PoseStamped, queue_size=5)
    goal_publisher.publish(goal)
    print("CHECK 1")

  elif (total_distance_tb3_1<total_distance_tb3_0):
    goal_publisher = rospy.Publisher("tb3_1/move_base_simple/goal", PoseStamped, queue_size=5)
    goal_publisher.publish(goal)
    print("CHECK 2")
```

Figure 5.6: second version of the callback function where the goal creation and publication were too close.

Multiple machines connection and simulation

The aim of this last step consists in running a simulation of the two turtlebot3 across three different machines. The first machine is the *master*, namely the machine on which the Gazebo and RViz GUIs are opened and where the node is executed. The second machine is responsible for the simulation of the first turtlebot3 (tb3_0), whereas the second one is responsible for the simulation of the second turtlebot3 (tb3_1). The overall scheme is shown in the following figure:

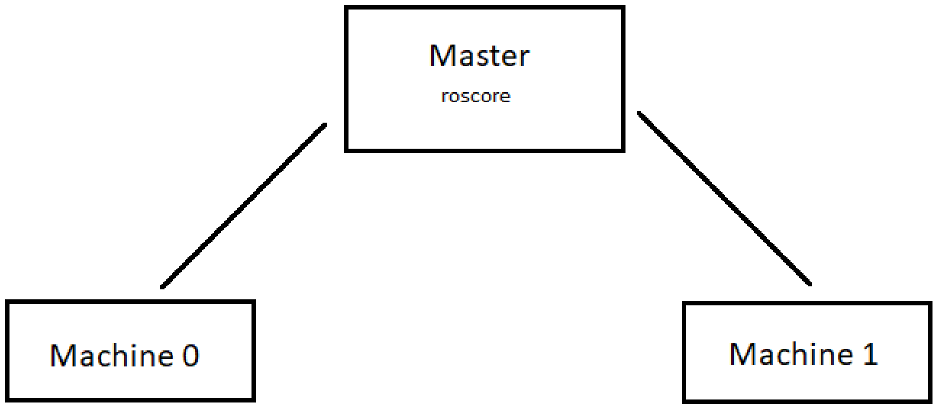


Figure 6.1: network scheme.

The basic concept behind the connection of multiple machines with ROS is that only one roscore is executed, then all the other platforms are configured to have the roscore associated with the IP address of the *master machine* (and not the local one as default).

Luckily, ROS is designed to allow multiple machines cooperation, therefore the procedure to configure the ROS master is very simple.

Before starting it is necessary to know the IP address of all the components that will be part of the net, by means of the following command:

```
$ hostname -I
```

Then, on the master machine it is necessary to export its IP address:

```
$ export ROS_IP=192.168.x.y (ip of the master machine)
```

On the *machine 0* it is necessary to export its IP address and set the ROS MASTER URI to the *master machine*:

```
$ export ROS_MASTER_URI=http://192.168.x.y:11311 (ip of the master machine)

$ export ROS_IP=192.168.x.y0 (ip of the machine 0)
```

Analogously, on the *machine 1* it is necessary to export its IP address and set the ROS MASTER URI to the *master machine*:

```
$ export ROS_MASTER_URI=http://192.168.x.y:11311 (ip of the master machine)

$ export ROS_IP=192.168.x.y1 (ip of the machine 1)
```

Launch files modification

Launch files must be slightly changed (by commenting or moving the previously defined parts) to perform the simulation across multiple machines:

1. The master computer will have to launch a file that initializes the gazebo GUI with the turtlebot3 world and the RViz GUI.
2. The machine 0 will have to spawn the robot named tb3_0, without launching the gazebo GUI, since it is already opened in the master machine. The same concept holds for the machine 1.
3. The machine 0 will start the navigation of the robot tb3_0, without opening RViz, since already done in the master pc. The same concept holds for the machine 1.

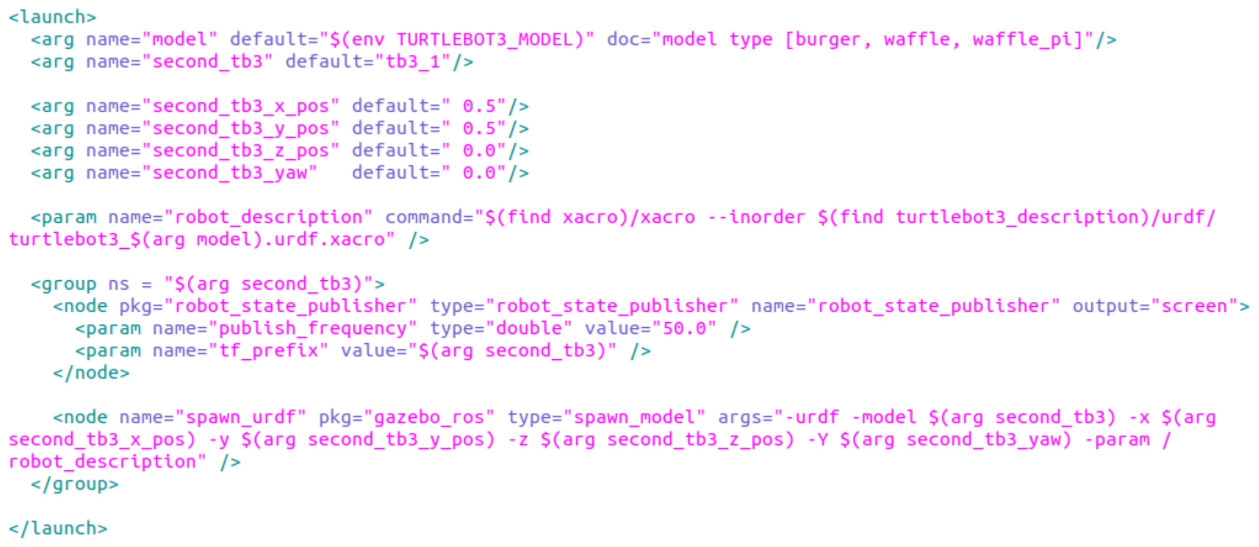
The following figures show all the modified files to run the simulation across multiple machines.

```
<launch>
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="open_rviz" default="true"/>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find turtlebot3_gazebo)/worlds/turtlebot3_world.world"/>
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="debug" value="false"/>
  </include>

  <!-- rviz -->
  <group if="$(arg open_rviz)">
    <node pkg="rviz" type="rviz" name="rviz" required="true"
          args="-d $(find turtlebot3_navigation)/rviz/turtlebot3_navigation_multi.rviz"/>
  </group>

</launch>
```

Figure 6.2: master launch file.

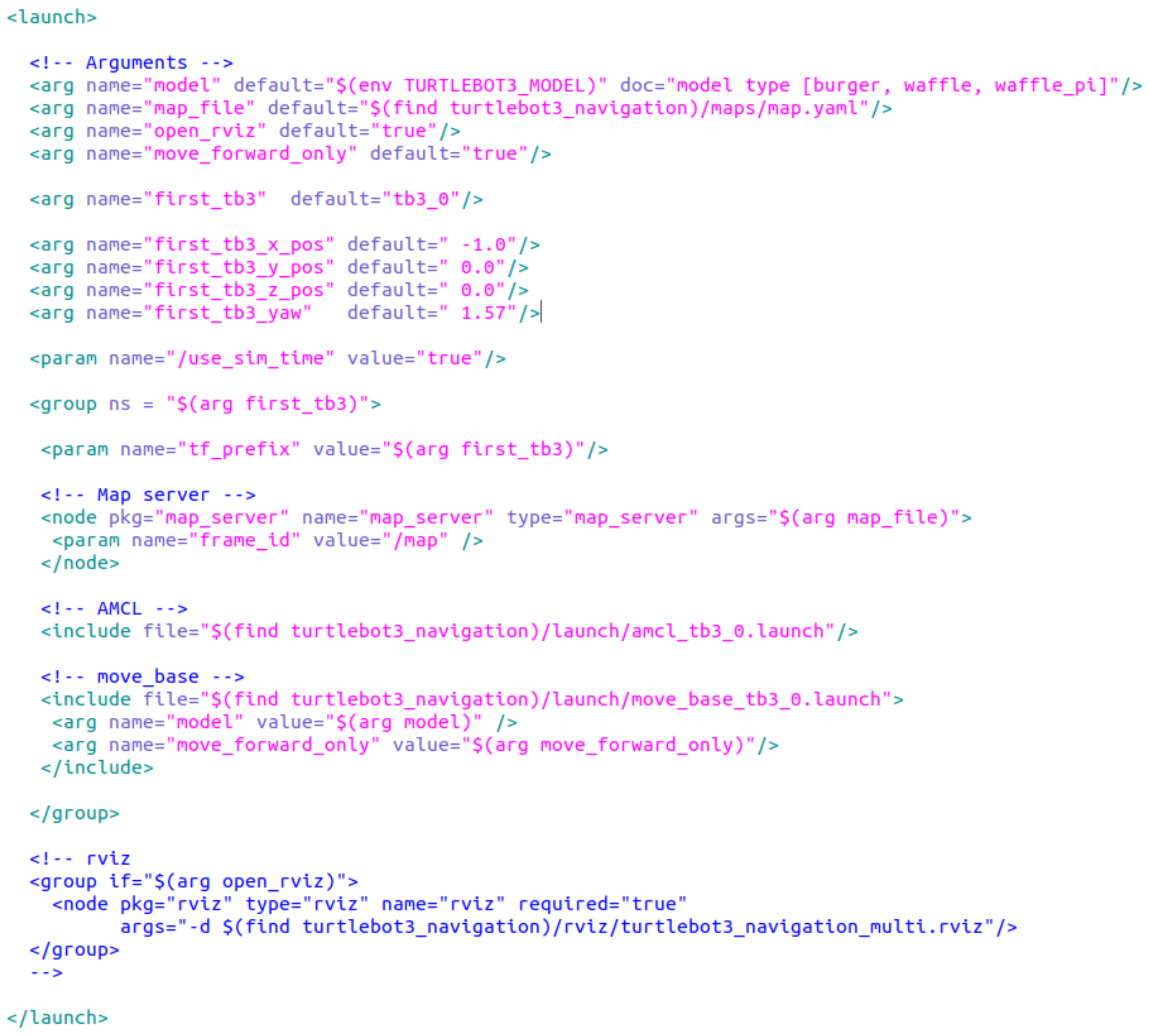
```
<launch>
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="first_tb3"  default="tb3_0"/>

  <arg name="first_tb3_x_pos" default=" -0.5"/>
  <arg name="first_tb3_y_pos" default=" 0.5"/>
  <arg name="first_tb3_z_pos" default=" 0.0"/>
  <arg name="first_tb3_yaw"   default=" 0.0"/>

  <param name="robot_description" command="$(find xacro)/xacro --inorder $(find turtlebot3_description)/urdf/
turtlebot3_$(arg model).urdf.xacro" />

  <group ns = "$(arg first_tb3)">
    <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher" output="screen">
      <param name="publish_frequency" type="double" value="50.0" />
      <param name="tf_prefix" value="$(arg first_tb3)" />
    </node>

    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-urdf -model $(arg first_tb3) -x $(arg
first_tb3_x_pos) -y $(arg first_tb3_y_pos) -z $(arg first_tb3_z_pos) -Y $(arg first_tb3_yaw) -param /
robot_description" />
  </group>

</launch>
```

Figure 6.3: tb3_0 launch file (machine 0).



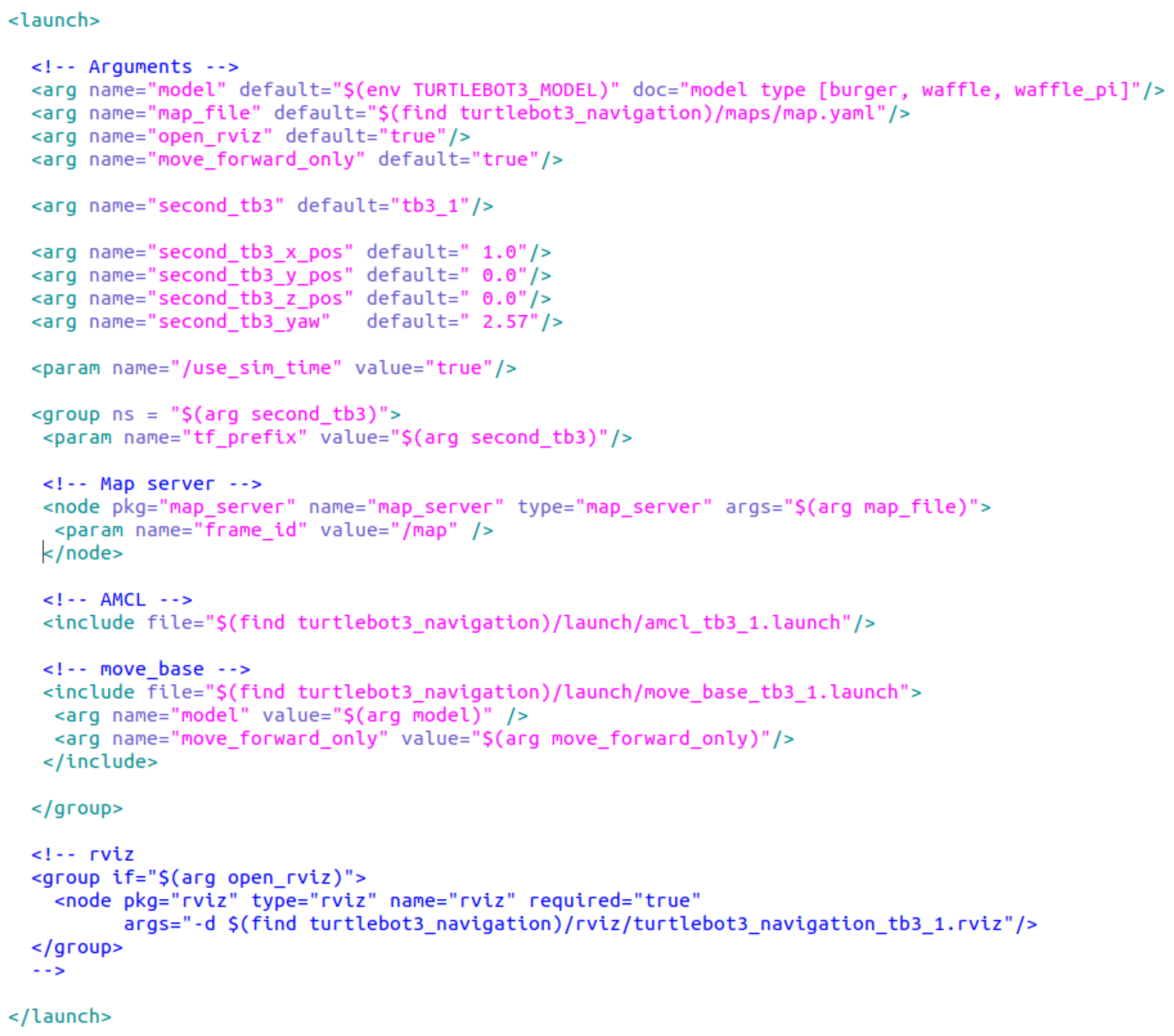
```
<launch>
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="second_tb3" default="tb3_1"/>

  <arg name="second_tb3_x_pos" default=" 0.5"/>
  <arg name="second_tb3_y_pos" default=" 0.5"/>
  <arg name="second_tb3_z_pos" default=" 0.0"/>
  <arg name="second_tb3_yaw"   default=" 0.0"/>

  <param name="robot_description" command="$(find xacro)/xacro --inorder $(find turtlebot3_description)/urdf/
turtlebot3_$(arg model).urdf.xacro" />

  <group ns = "$(arg second_tb3)">
    <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher" output="screen">
      <param name="publish_frequency" type="double" value="50.0" />
      <param name="tf_prefix" value="$(arg second_tb3)" />
    </node>

    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-urdf -model $(arg second_tb3) -x $(arg
second_tb3_x_pos) -y $(arg second_tb3_y_pos) -z $(arg second_tb3_z_pos) -Y $(arg second_tb3_yaw) -param /
robot_description" />
  </group>

</launch>
```

Figure 6.4: tb3_1 launch file (machine 1).



```
<launch>

  <!-- Arguments -->
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="map_file" default="$(find turtlebot3_navigation)/maps/map.yaml"/>
  <arg name="open_rviz" default="true"/>
  <arg name="move_forward_only" default="true"/>

  <arg name="first_tb3"  default="tb3_0"/>

  <arg name="first_tb3_x_pos" default=" -1.0"/>
  <arg name="first_tb3_y_pos" default=" 0.0"/>
  <arg name="first_tb3_z_pos" default=" 0.0"/>
  <arg name="first_tb3_yaw"   default=" 1.57"/>

  <param name="/use_sim_time" value="true"/>

  <group ns = "$(arg first_tb3)">

   <param name="tf_prefix" value="$(arg first_tb3)"/>

   <!-- Map server -->
   <node pkg="map_server" name="map_server" type="map_server" args="$(arg map_file)">
    <param name="frame_id" value="/map" />
   </node>

   <!-- AMCL -->
   <include file="$(find turtlebot3_navigation)/launch/amcl_tb3_0.launch"/>

   <!-- move_base -->
   <include file="$(find turtlebot3_navigation)/launch/move_base_tb3_0.launch">
    <arg name="model" value="$(arg model)" />
    <arg name="move_forward_only" value="$(arg move_forward_only)"/>
   </include>

  </group>

  <!-- rviz
  <group if="$(arg open_rviz)">
    <node pkg="rviz" type="rviz" name="rviz" required="true"
          args="-d $(find turtlebot3_navigation)/rviz/turtlebot3_navigation_multi.rviz"/>
  </group>
  -->

</launch>
```

Figure 6.5: tb3_0 navigation launch file (machine 0).



```
<launch>

  <!-- Arguments -->
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="map_file" default="$(find turtlebot3_navigation)/maps/map.yaml"/>
  <arg name="open_rviz" default="true"/>
  <arg name="move_forward_only" default="true"/>

  <arg name="second_tb3" default="tb3_1"/>

  <arg name="second_tb3_x_pos" default=" 1.0"/>
  <arg name="second_tb3_y_pos" default=" 0.0"/>
  <arg name="second_tb3_z_pos" default=" 0.0"/>
  <arg name="second_tb3_yaw"   default=" 2.57"/>

  <param name="/use_sim_time" value="true"/>

  <group ns = "$(arg second_tb3)">
   <param name="tf_prefix" value="$(arg second_tb3)"/>

   <!-- Map server -->
   <node pkg="map_server" name="map_server" type="map_server" args="$(arg map_file)">
    <param name="frame_id" value="/map" />
   </node>

   <!-- AMCL -->
   <include file="$(find turtlebot3_navigation)/launch/amcl_tb3_1.launch"/>

   <!-- move_base -->
   <include file="$(find turtlebot3_navigation)/launch/move_base_tb3_1.launch">
    <arg name="model" value="$(arg model)" />
    <arg name="move_forward_only" value="$(arg move_forward_only)"/>
   </include>

  </group>

  <!-- rviz
  <group if="$(arg open_rviz)">
    <node pkg="rviz" type="rviz" name="rviz" required="true"
          args="-d $(find turtlebot3_navigation)/rviz/turtlebot3_navigation_tb3_1.rviz"/>
  </group>
  -->

</launch>
```

Figure 6.7: tb3_1 navigation launch file (machine 1).

How to launch the simulation across multiple machines

The configuration is now completed. Any node (or launch file) launched on one of the two machines is displayed on the *master machine*, since the roscore is opened on that one.

As explained before, launch files are slightly different with respect to the ones used for the simulation on a single machine. In particular, the master machines launch the Gazebo GUI with the turtlebot3 world and additionally it also opens RViz GUI. Then, each machine spawns the robot in the *master machine* and each one starts also the navigation. Lastly, the *master machine* launches the node.

The entire procedure is the following one:

MASTER

First terminal

```
$ export ROS_IP=192.168.x.y (ip of the master machine)
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_gazebo turtlebot3_world_master.launch
```

MACHINE 0

First terminal (spawn)

```
$ export ROS_MASTER_URI=http://192.168.x.y:11311 (ip of the master machine)

$ export ROS_IP=192.168.x.y0 (ip of the machine 0)

$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_gazebo turtlebot3_world_tb3_0_machine_0.launch
```

Second terminal (navigation)

```
$ export ROS_MASTER_URI=http://192.168.x.y:11311 (ip of the master machine)

$ export ROS_IP=192.168.x.y0 (ip of the machine 0)

$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_navigation turtlebot3_navigation_tb3_0.launch
```

MACHINE 1

First terminal (spawn)

```
$ export ROS_MASTER_URI=http://192.168.x.y:11311 (ip of the master machine)

$ export ROS_IP=192.168.x.y1 (ip of the machine 1)

$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_gazebo turtlebot3_world_tb3_1_machine_1.launch
```

Second terminal (navigation)

```
$ export ROS_MASTER_URI=http://192.168.x.y:11311 (ip of the master machine)

$ export ROS_IP=192.168.x.y1 (ip of the machine 1)
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_navigation turtlebot3_navigation_tb3_1.launch
```

MASTER

Second terminal

```
$ export ROS_IP=192.168.x.y (ip of the master machine)

$ source ~/catkin_ws/devel/setup.bash

$ cd ~/catkin_ws/src/navigation_package/scripts

$ chmod +x test2.py

$ rosrun navigation_package test2.py
```

Guides and troubleshooting

Guides

How to add all our packages to avoid the manual procedure

To run our launch files and our node, there are two possibilities:

1. Follow the complete procedure that you can find here: “Complete manual procedure”.
2. Add manually the entire src folder, perform a catkin_make and start using our node and packages.

In this first part, the second possibility is explained.

1. Remove the already existing catkin workspace (or rename it to not lose it):

```
$ rm -r ~/catkin_ws
```

If you already have projects into the catkin_ws directory please do not remove the catkin_ws, only rename it.

2. Download from our GitHub page the catkin_ws directory with the source folder only:

https://drive.google.com/open?id=1G6ECFKyCK7Ljf4G8wbdKDaSupjOf6l3s

3. Copy it to the “Home” directory of Ubuntu.

4. Open a new terminal and launch:

```
$ source ~/catkin_ws/devel/setup.bash
```

5. Change the directory:

```
$ cd ~/catkin_ws
```

6. Do a catkin_make:

```
$ catkin_make
```

If the catkin_make didn’t report any trouble, you are ready to use all our packages. Therefore, you are able to use our files. To do this, go here “Running multiple robots with our modified launch files” and follow the steps.

How to map the world

The map can be implemented with the turtlebot3 SLAM. Follow the guide at 11.2.1.4.

http://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/#simulation

Launch Gazebo:

```
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

Launch SLAM:

```
$ source ~/catkin_ws/devel/setup.bash
$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

Launch the teleop key package to drive the robot. This procedure is necessary to create the map.

```
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Now you have to move around the map in order to create the full map. Once you finished (check it on Rviz), launch the following command.

```
$ rosrun map_server map_saver -f ~/map
```

How to add gmapping

Gmapping is not officially released for ROS melodic (you cannot download the package by means of sudo apt[…]). However, it is possible to load it in a different way. Follow this procedure.

Remark: start in the default directory: ~/

```
$ cd
$ sudo apt update
$ rosdep update
$ cd catkin_ws/
$ cd src
$ git clone https://github.com/ros-perception/openslam_gmapping src/openslam_gmapping
$ git clone https://github.com/ros-perception/slam_gmapping src/slam_gmapping
$ rosdep install --from-paths src/ -i
$ cd ..
$ catkin_make
```

The catkin_make process should end without errors.

The place where I found the solution was: https://answers.ros.org/question/300480/building-open_gmapping-from-source-on-melodicubuntu-1804/

To test it follow these instructions:

```
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

How to add the merge mapping (step necessary for the multi merging map)

```
$ cd
$ sudo apt update
$ rosdep update
$ cd catkin_ws/
$ cd src
$ git clone https://github.com/hrnr/m-explore.git

$ rosdep install --from-paths src/ -i
$ cd ..
$ catkin_make
```

The catkin_make process should end without errors.

Allow the execution of a node

In order to use a node, it is necessary to make it executable. Change the directory into the one that contains the node, then use the command chmode + “name”.py to execute it. In this case the node is named “talker.py” and it is inside the package “my_first_package” in the folder “scripts”.

```
$ cd ~/catkin_ws/src/my_first_package/scripts/
$ chmod +x talker.py
```

How to run a node

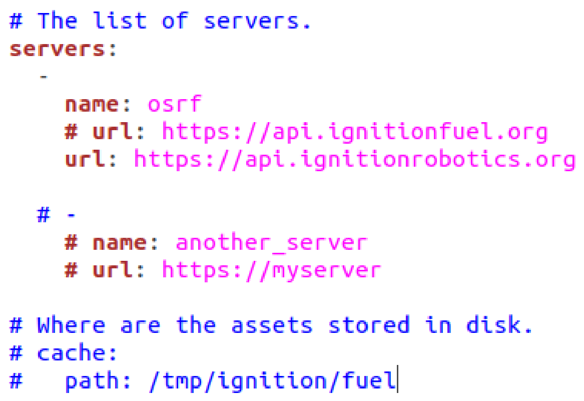
To run a node, firstly you have to allow its execution, then you have to use the command rosrun. Rosrun, then the name of the package and in the end the name of the node.

```
$ rosrun my_first_package talker.py
```

Troubleshooting

How to solve the error in Gazebo

Press Ctrl+H in the home folder to see the non-visible folders. Then go in .ignition, then in fuel, then open



```
# The list of servers.
servers:
  -
    name: osrf
    # url: https://api.ignitionfuel.org
    url: https://api.ignitionrobotics.org

  # -
    # name: another_server
    # url: https://myserver

# Where are the assets stored in disk.
# cache:
#   path: /tmp/ignition/fuel
```

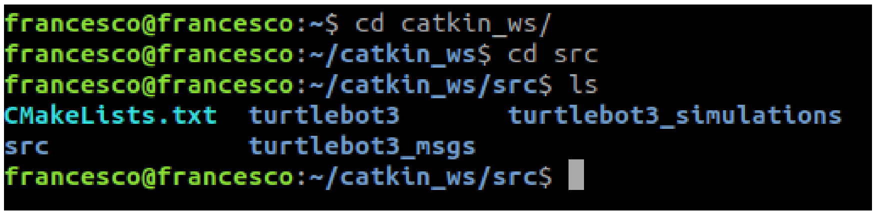
the config.yaml file. Comment the server address and change it with:

url: https://api.ignitionrobotics.org

Make sure that you have installed the turtlebot3 simulation package!

```
$ cd ~/catkin_ws/src/
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
$ cd ~/catkin_ws && catkin_make
```

You must have all the following packages installed to guarantee the simulation:



```
francesco@francesco:~$ cd catkin_ws/
francesco@francesco:~/catkin_ws$ cd src
francesco@francesco:~/catkin_ws/src$ ls
CMakeLists.txt  turtlebot3       turtlebot3_simulations
src             turtlebot3_msgs
francesco@francesco:~/catkin_ws/src$
```

Complete manual procedure

Installation of all the components to run turtlebot3

1. Remove the catkin_ws (if present).

   ```
   $ rm -r ~/catkin_ws
   $ echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
   $ source ~/.bashrc
   ```

2. Initialize a new catkin workspace with the command in the first page (from initializing procedure to building the workspace).

   ```
   $ mkdir -p ~/catkin_ws/src
   $ cd ~/catkin_ws/src
   $ catkin_init_workspace
   $ cd ..
   $ catkin_make
   ```

3. Start the installation process of turtlebot3, following the guide at http://emanual.robotis.com/docs/en/platform/turtlebot3/pc_setup/. Remember that you are using ROS melodic, not kinetic, then use the following lines, instead of the one present on the website. Moreover, the gmapping package is not present, therefore, you will have to add it manually (later).

   sudo apt-get install ros-melodic-joy ros-melodic-teleop-twist-joy ros-melodic-teleop-twist-keyboard ros-melodic-laser-proc ros-melodic-rgbd-launch ros-melodic-depthimage-to-laserscan ros-melodic-rosserial-arduino ros-melodic-rosserial-python ros-melodic-rosserial-server ros-melodic-rosserial-client ros-melodic-rosserial-msgs ros-melodic-amcl ros-melodic-map-server ros-melodic-move-base ros-melodic-urdf ros-melodic-xacro ros-melodic-compressed-image-transport ros-melodic-rqt-image-view ros-melodic-navigation ros-melodic-interactive-markers

4. Remember to follow all the instructions, after the previous command:

```
$ cd ~/catkin_ws/src/

$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git

$ git clone https://github.com/ROBOTIS-GIT/turtlebot3.git

$ cd ~/catkin_ws && catkin_make
```

5. Add gmapping manually following the guide “".

6. Install the simulation package:

```
$ cd ~/catkin_ws/src/

$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git

$ cd ~/catkin_ws && catkin_make
```

7. Solve the gazebo problem as explained in the above section: .

8. Once installed everything, launch the turtlebot3 simulation using the three commands

```
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

Slam and navigation

9. Map the world with SLAM:

Launch Gazebo:

```
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

Launch SLAM:

```
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

Launch the teleop key package to drive the robot. This procedure is necessary to create the map.

```
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Now you have to move around the map in order to create the full map. Once you finished (check it on Rviz: the map must be completely defined), launch the following command to save the map.

```
$ rosrun map_server map_saver -f ~/map
```

10. After having the map, start with the navigation:

Execute Gazebo:

```
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

Execute Navigation (remember that you **need the map**):

```
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_navigation turtlebot3_navigation.launch map_file:=$HOME/map.yaml
```

Add manually our packages

Once completed the following procedure you have set everything to allow a correct navigation of a single robot on a single machine. To be able to use our packages, you should download them from our repository and add them into the correct folders:

https://github.com/francimala/ROS_multiple_navigation

Find the folder turtlebot3_gazebo/launch and paste there the correct launch files.

Find the folder turtlebot3_navigation/launch and past there the correct files.

Find the folder turtlebot3_navigation/rviz and past there the correct files.

Go inside the catkin_ws/src folder and paste the navigation_package folder.

Do a catkin_make.

Running multiple robots with our modified launch files

If you want to use our launch files, you have to download them from our GitHub page putting them into the right directory:

- turtlebot3_world_multi.launch in the turtlebot3_gazebo/launch folder.
- turtlebot3_navigation_multi.launch in the turtlebot3_navigation/launch folder.
- turtlebot3_navigation_multi.rviz in turtlebot3_navigation/rviz folder.

Then it is possible to start with the procedure.

First terminal

```
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_gazebo turtlebot3_world_multi.launch
```

Second terminal

```
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_navigation turtlebot3_navigation_multi.launch
```

Running our node

The first thing to do consists in allowing the node to be used. The node is located inside a directory scripts into the navigation_package created by ourself. It is then necessary to move to the scripts directory inside the new package.

```
$ source ~/catkin_ws/devel/setup.bash

$ cd ~/catkin_ws/src/navigation_package/scripts

$ chmod +x test2.py
```

Once allowed the node, it is necessary to run it. To run a node it is sufficient to use the rosrun command followed by the package name and the python-script name.

```
$ source ~/catkin_ws/devel/setup.bash

$ rosrun navigation_package test2.py
```

Running the simulation across multiple machines

MASTER

First terminal

```
$ export ROS_IP=192.168.x.y (ip of the master machine)
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_gazebo turtlebot3_world_master.launch
```

MACHINE 0

First terminal (spawn)

```
$ export ROS_MASTER_URI=http://192.168.x.y:11311 (ip of the master machine)

$ export ROS_IP=192.168.x.y0 (ip of the machine 0)

$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_gazebo turtlebot3_world_tb3_0_machine_0.launch
```

Second terminal (navigation)

```
$ export ROS_MASTER_URI=http://192.168.x.y:11311 (ip of the master machine)

$ export ROS_IP=192.168.x.y0 (ip of the machine 0)

$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_navigation turtlebot3_navigation_tb3_0.launch
```

MACHINE 1

First terminal (spawn)

```
$ export ROS_MASTER_URI=http://192.168.x.y:11311 (ip of the master machine)

$ export ROS_IP=192.168.x.y1 (ip of the machine 1)

$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_gazebo turtlebot3_world_tb3_1_machine_1.launch
```

Second terminal (navigation)

```
$ export ROS_MASTER_URI=http://192.168.x.y:11311 (ip of the master machine)

$ export ROS_IP=192.168.x.y1 (ip of the machine 1)
$ source ~/catkin_ws/devel/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_navigation turtlebot3_navigation_tb3_1.launch
```

MASTER

Second terminal

```
$ export ROS_IP=192.168.x.y (ip of the master machine)

$ source ~/catkin_ws/devel/setup.bash

$ cd ~/catkin_ws/src/navigation_package/scripts

$ chmod +x test2.py

$ rosrun navigation_package test2.py
```

Goal reached.

Got new plan

Bibliografia

[1] H. C. R. J. T. L. YoonSeok Pyo, ROS Robot Programming, Seoul, Republic of Korea: ROBOTIS Co.,Ltd., 2017.

[2] “Wiki.ros.org,” [Online]. Available: http://wiki.ros.org/. [Accessed 12 05 2019].