

Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Research Project

iRobot Roomba Swarm, Navigation and Scheduling

Sarthak S Bapat.
(Matriculation: 3441556)

Course of Study: Information Technology, Embedded Systems

Examiner: Prof. Dr. Marco Aiello

Supervisor: Mr. Isaac Henderson Johnson Jeyakumar
(Fraunhofer IAO)

Commenced: December 1, 2020

Completed: June 10, 2021

Abstract

The focus of this work is to present a robotic swarm system using ROS (Robot Operating System) with a single ROS master. The cleaning of an indoor environment such as an office space often takes time and effort and thus could be automated using low-cost vacuum cleaning Robots. The aim is to provide the necessary software fundamentals and networking with computer systems for navigating low cost robots in a Smart Office environment. We demonstrate a swarm of two iRobot Roomba 650 robots as a robotic swarm system running with a single ROS master. For the visualization of swarm a linux PC with ROS installed and RVIZ (ROS Visualization tool) is used. Raspberry Pi 4B with Ubuntu 16.04 and ROS Kinetic is used for the processing on both the robots. The project doesn't focus on the optimization of the navigation stack parameters but only on its configuration. Both the robots are visualized and controlled in a single map of the office space. The work also presents the scheduling of both the robots based on the calendar that contains the meeting entries for a given day. Depending on the meeting schedule, the robots have the capability to navigate to the target location given to them. Using this system, we present a novel approach to automate the cleaning task in an indoor Smart Office environment.

Contents

1	Introduction	5
2	Background Information	7
2.1	Simultaneous Localization and Mapping	7
2.2	ROS Navigation Stack	7
2.3	ROS Visualisation Tool	9
3	State of the Art	10
3.1	Software Architecture for Robotic Swarm Systems	10
3.2	Map Sharing between multiple robots	11
3.3	Global and Local Planners in ROS	11
3.4	Scheduling of the Roomba	12
4	Design of Solution for Navigation and Scheduling of the Robots	13
4.1	Mapping the environment	13
4.2	Swarming of two Roomba robots	13
4.3	Navigation of the robots	14
4.4	Scheduling the robots	14
4.5	Algorithm for our system	14
5	Realization of Swarm and Navigation	16
5.1	Mapping of the office environment	16
5.2	Creating a swarm and Map sharing	17
5.3	Navigation to goal using ROS	21
5.4	Scheduling the Roomba	22
6	Discussion of the Results	23
6.1	Environment map	23
6.2	Swarm of two Robots	24
7	Conclusion and Future Scope	27
8	References	28

List of Figures

5.1	Slam Parameters	17
5.2	Create_2 launch file	18
5.3	Lidar launch file	18
5.4	Robot transform tree without namespace	19
5.5	Move base launch file	19
5.6	Robot1 bashrc file (ROS Master)	20
5.7	Robot2 bashrc file	20
5.8	Linux PC bashrc file	20
5.9	Robot Swarm Transform tree	21
6.1	Integrated home office environment	23
6.2	Integrated home office environment	24
6.3	Rosgraph before swarm	24
6.4	Rosgraph after swarm	25
6.5	Roomba Swarm	26

1 Introduction

Usability of the vacuum cleaning robots in a smart office environment with a large floor area is feasible but not cost and time efficient. The sensors on the robot are not sufficient for its good spatial orientation over large areas. Roomba is a vacuum cleaning robot manufactured by iRobot and has evolved over time getting equipped with new features. The series 900 of Roomba comes with VSLAM technology that has an optical camera enabling the robot to map the area it cleans [1]. This allows for a better spatial orientation but this comes at a monetary cost. The Roomba 600 series does not support the VSLAM technology but is equipped with 'Dirt sensors' that aid the cleaning process [1]. The robot can be extended to achieve the VSLAM functionality available in the 900 series using LiDAR sensor with ROS framework to map the environment beforehand and use the generated map to localize itself and navigate in the environment. The objective of this work is to provide an approach to clean office areas using a low-cost vacuum cleaning Roomba 600 series robots. The idea is to present a robotic swarm system using Roomba vacuum cleaning robots so that the cleaning process can start simultaneously.

The research consists of a method to create a swarm of two Roomba robots using ROS. There are two possible ways to establishing information exchange between the two robots in a multi-robot system, exchange the messages between robots using different ROS sub-systems or have a single ROS network that runs a single ROS master [2]. The first approach consists of multiple ROS sub-systems running multiple ROS masters and exchanging the messages between different sub-systems over a common network [2]. The other considers a single ROS sub-system running a single ROS master and communicates using ROS features such as topics, services or actions [2]. Perhaps, the biggest drawback of ROS is the publisher/subscriber management using a centralized roscore [11]. ROS was not designed to work in a distributed environment and ROS2 was then designed to overcome this and a few other drawbacks of ROS [11]. Since we have only two robots and one PC in our system, we focus on the later approach to address the robot swarm method.

With the technique discussed above, the goal is to swarm two Roombas in a single map of the office environment. One Roomba will run the ROS master, the other Roomba and the linux PC will be a part of the sub-system. Using RVIZ tool it is then possible to set the initial positions for each robot to have a better localization in the environment. The project also presents the scheduling of the robots to navigate to their respective goal locations. The scheduling would be done on the basis of meeting entries and the idea is to make sure the robots navigate to their goal locations a few minutes before the meeting time. This would allow for an automatic scheduling of the robot following the meeting time.

The report is further divided into multiple chapters that discuss the necessary details. The chapter Background Information consists an overview of the technical details required to understand the solution followed by the current state of the art discussed in chapter 3. In chapter 4, we present the

design of our system. Chapter 5 discusses the realisation of the solution and the discussion of the important results is done in chapter 6. The report ends with Conclusions and a discussion about possible future extensions in the project explained in chapter 7.

2 Background Information

We present the important concepts and the terminologies that are necessary to understand the details that follow. The concepts such as the navigation stack in ROS and Simultaneous Localization and Mapping form the base of this work and are therefore presented in this section.

2.1 Simultaneous Localization and Mapping

SLAM is an approach that allows the robot to plan a path by avoiding the obstacles around it. It is basically mapping of the environment at the same time when the robot is moving. The information from the sensors is gathered while the robot moves in an environment and a map is generated from the sensor data. ROS offers a localization service called gmapping which works on the SLAM technique [4]. So it is possible to generate the map of the environment using gmapping and then use the same map statically with AMCL, another localization method in ROS for the navigation of the robot.

2.2 ROS Navigation Stack

The main functionality of the Navigation stack is to get the robot's odometry information and the sensor streams and output the velocity commands to the robot's mobile base. The pre-requisites for the navigation stack to work are running of ROS on the robot, establish the transform tree and publish the sensor data using the ROS message types[3]. Let's have a look at the important items required to use the navigation stack.

2.2.1 Transforms

The robot sensors measure the environment data with respect to themselves and not in relation with the robot. So there is a need for a geometric conversion of the received data to make the sensor readings available in relation to the robot's different frames. For this conversion, ROS offers the TF tool which adjusts the sensor positions in relation to the robot and the converted readings are utilised for the navigation. Thus, the navigation stack understands the location of the sensors with respect to the robot's centre (base_link) [4].

2.2.2 AMCL and Map server

AMCL is a localization system that runs on a known map of an environment. It does not manage the map but uses an already built map to localize the robot. The localization approach is called 'Adaptive Monte Carlo Localization' which randomly distributes the particles in the known map representing the possible robot locations and then uses a particle filter to determine the actual pose of the robot [4].

Map Server package has two nodes, map server and the map saver. The map server node runs with a static map as a ROS service and the map saver saves a dynamically generated map to a file [4]. The map saved by the map saver node generates two files, a .pgm file representing the image and a .yaml file containing the static data for the map.

2.2.3 gmapping

Gmapping is a localization system but unlike AMCL it runs on an unknown environment and performs Simultaneous Localization and Mapping. A new 2D map of the environment is generated using the robot pose and the laser data [4].

2.2.4 Local and Global Costmaps

The local and global costmaps are the topics that contain the information representing obstacles in a 2D plane. They also contain information about security inflation radius which is an area around the obstacles that guarantees the robot will not collide with any objects irrespective of its orientation. Local costmap represents a window around the robot whereas the global costmap represents the entire environment [4].

2.2.5 Local and Global Planners

The global planner uses the robot's current location and the goal and calculates a low-cost trajectory towards the goal with respect to the global costmap. The local planner works over a local costmap and is able to detect more obstacles due to high definition of the local costmap. Thus, it creates the trajectory rollout over the global trajectory, that is able to return to the original trajectory with the fewer cost. ROS provides a package called move_base that uses the global and the local planner to make the robot navigate towards the goal location [4].

2.2.6 Sensors and Controllers

The odometry source for the robot is generated using the wheel encoder data and the base controller of the robot is responsible for taking velocity commands from the cmd_vel topic and generate the obtained velocities [4].

2.3 ROS Visualisation Tool

ROS provides a tool called RVIZ for the visualisation of the navigation behaviour of the robot. RVIZ is often run on a different machine than that operates the robot and this is achieved by using multiple machines that communicate over the same ROS topics. The user can add the topics of interest to RVIZ for visualisation [4].

With the concepts discussed above, we have the basic understanding of the robotics system and its main components and are good to dive into the further details.

3 State of the Art

Multi-robot systems are a group of robots coordinated in order to perform a particular task which may not be achieved by a single robot or can be optimized if performed in the group. The analysis of the task to perform should find a possible increase in the efficiency when using a multi-robot system. Two unit systems is a type of multi-robot system which consists of two robots to perform the tasks. The focus of this system is on the tasks that cannot be performed by a single robot alone or to achieve complementary abilities [5]. In our application, we need multi-robot ability for faster cleaning of the office environment and we use a two unit system as a robotic swarm for our task.

3.1 Software Architecture for Robotic Swarm Systems

The software architecture is one of the most important areas of any robotic system and is in itself an area of study for the swarm robotic systems. In the ROS framework, the ROS core is the most important component of the ROS computational graph. The ROS master enables individual ROS nodes to locate each other, provides the naming and registration services to the nodes in the system and tracks the publishers and the subscribers to topics as well as services [5].

There are two architectures possible for the swarm systems, one using a centralized ROS master and the other with multiple ROS masters (multi-master). Most ROS applications use a centralized ROS master because they run on a single robot or they need to have a common point for the processing of information. The drawback of centralized ROS master is that it is overloaded when used in a multi-robot system and is a single point of failure. Multi-master architecture allows multiple ROS masters running on different robots in the system thus making it distributed and modular. As a result of the distributed management, new connections can be established with the system and the ROS parameter server is available even though all the masters have not started up [5]. ROS has added the multi-master ability by introducing new packages that offer this functionality. The new packages come up with their pros and cons and their usage highly depends on the application requirements.

ROS package `ad hoc_communication` works on the principle of ad-hoc networks and ad-hoc routing protocols to establish the communication between different ROS masters. It allows for unicast, multicast and broadcast level information exchange. The package has critical limitations like the communication between different masters is possible using raw socket implementations, hence the node needs to be executed with super-user privileges which is undesired. Also, the package does not support all the ROS message types which is a major limitation [5].

The ROS package `multimaster_fkie` is a simple and fully compatible implementation for ROS topics and services. With this package it's possible to run the ROS architecture unmodified without adding any APIs or libraries to establish the multi-master communication. The package implements two additional nodes, `master_discovery` and `master_synchronization`. The `master_discovery` node continuously scans and advertises about its local master state and receives information from other

masters in the network. When `master_discovery` advertises the state change, `master_synchronization` node is triggered and the local master requests the information to the foreign node and synchronizes it into its own topic or service. Limitation of this package is that it does not provide namespace handling. So, in case of multiple nodes and topics with same names running on different ROS masters it becomes difficult to identify and separate the nodes from one another [5].

The `rocon_multimaster` is a multi-robot framework running on top of a centralized workspace for robotic solutions. The masters are organized in groups with a central hub where a gateway is located. This allows for a more secure architecture keeping large information trespassing inside the hub. The major drawback of this package is the complexity as it requires a lot of parameterization and configuration. This package is therefore intended to be used with large scale multi-robot projects [5].

3.2 Map Sharing between multiple robots

Multiple robots sharing a map of the environment they operate in is an important aspect of robot swarms or the multi-robot systems. The visualization of multiple robots in the map of the environment is possible using the ROS tool RVIZ. Using RVIZ its possible to visualize the exact location of the robots in the map, the path generated by the global and local planners to the destination and even setting up robot's initial pose and goal locations.

The principle behind the map sharing between two robots is that both operate in different namespace groups and have different transform prefix. Transform prefix allows to set the namespace for the robot's frames so as to differentiate between the frames of both the robots. The namespace group enables the ROS topics and nodes to be attached with the desired namespace. The two robots are pointed to the map frame which is hosted on the ROS map server and does not form a part of any robot namespace. This enables the robots to operate in different namespaces and share the same map with clear distinguishing ability.

To achieve the sharing of the map using the technique discussed, it is very important for the robots to establish a correct transform tree and point to the map frame. If a robots transform tree with namespace does not match the one without using a namespace, then the navigation stack does not work as expected. We will discuss further about the transform tree in the implementation details.

3.3 Global and Local Planners in ROS

Global planner in ROS is responsible for generating an obstacle free path from robot's initial position to the desired goal position. The `navfn` and the `global_planner` are widely used ROS global planners. The `navfn` planner produces a minimum cost plan from start to end by using A* or Dijkstra algorithm by taking the global costmap as an input. The main disadvantage of `navfn` is the lack of smoothness of the produced path. The `global_planner` is a successor of `navfn` and generates paths using A* or Dijkstra algorithms to reduce the computational load. The paths are not optimal in an 8-connected sense. There are many other global planners like `asr_navfn`, `sbpl_lattice_planner`, `lattice_planner` and all planners use alterations of the A* algorithm [7].

A local planner takes in the perceived sensor data and the global path generated by the global planner as inputs and generates the velocity commands for the robot to navigate to the goal location. The `dwa_local_planner` discretely samples the robot's control space performing a forward simulation for each sample, evaluates trajectories against a local costmap and discards illegal trajectories. The main drawback of `dwa_local_planner` is that it does not support dynamic obstacle avoidance. The `teb_local_planner` gets the global path and minimizes it during runtime with respect to the trajectory execution time keeping separation from obstacles. This planner supports obstacle avoidance for dynamic obstacles [7].

The `teb_local_planner` is the most robust as it definitely reaches the goal location and that too faster than other available planners. The `dwa_local_planner` is slow and fails to navigate to the goal most of the times [7]. In our implementation, we have configured the `global_planner` and both the `dwa_local_planner` and the `teb_local_planner` for the navigation but we will stick to `teb_local_planner` due to its advantages.

3.4 Scheduling of the Roomba

The Roomba robots are rich in features and the newer versions allow the robot to be scheduled for cleaning by setting the time accordingly [1]. The scheduling though has to be done manually and adjusted as per the requirements, the meeting time in our case. To address this and simplify the scheduling task, we present an approach to schedule the cleaning of the environment by the robots before the meeting by monitoring the meeting entries in the calendar. The approach consists of monitoring all the meetings scheduled for the day and the state of the meeting room. With this we provide a solution that addresses the issue of scheduling the robot manually by being compliant with the meeting timings and the availability of the room.

4 Design of Solution for Navigation and Scheduling of the Robots

The cleaning of office space using a swarm of two Roombas requires them to share a map of the environment. We design the sequence of steps in the following manner to attain our objective of creating a swarm using two robots and make them navigate to their goal locations using the same map.

4.1 Mapping the environment

The map of the target space can be created by mapping out the area using multiple robots and then combining the map generated by each one. Also, it is possible to map out the entire area using a single robot. Since the area under consideration in our system is not too large, it is possible to map it using a single roomba using a LiDAR sensor mounted on it. This approach is simple to implement than the other one and possible with existing ROS packages. The process of mapping along with an image of the generated map is discussed in the later section.

4.2 Swarming of two Roomba robots

The swarm system we present consists of only two Roomba robots using the ROS framework on a Raspberry Pi 4 with Ubuntu 16.04. Since we have only two robots and implementation of multi-master system is challenging in ROS due to the cons of different packages we discussed, we propose a solution using a single ROS master system to create the swarm. One robot will run the ROS master and the other will use the same ROS master to communicate using its nodes and topics. The robots being similar and configured with almost similar software packages, it becomes important to have a distinguishing factor between the two when they are launched. When the robots are launched, the different nodes communicate with each other using ROS topics. To make the topic/node names unique to a robot we use the ROS namespaces.

Using the ROS namespaces it is possible to launch the robot's nodes, topics and services with a name prefix associated with it. The frames of the robot are also renamed using a naming prefix (transform prefix) to establish a correct and expected transform between all its frames. If the transform tree is not as expected, the navigation stack of the robot fails. With the help of namespaces, we can attach naming prefix to each robot and launch them together with one running the ROS master. This avoids any conflict between the names and all the nodes, topics, services and the parameters for each robot are recognisable.

To enable the map sharing between both Roombas, we run the map server node on only one robot and pass the map of the environment. Both the robots subscribe to the same map and therefore their transform trees points to the map frame which is shared. Using the RVIZ we can then visualize the footprint, global and local path planners, LiDAR laser scanners, pointclouds and a few other topics for both the robots. More regarding the implementation is discussed in the next section.

4.3 Navigation of the robots

The navigation of an individual robot is possible using the RVIZ tool. The RVIZ has two options, to set the initial pose of the robot on the map and to assign it a goal location on the map. When the robot's 'move_base' server is started and the necessary navigation stack files are launched then the RVIZ could easily be used to navigate the robot in the mapped environment. However, for two robots this is a bit tricky in RVIZ. Always the 2D initial pose and 2D navigate goal buttons have to be changed to the respective robot's topic and then assign it a goal location. Setting the initial pose of the robot has no other alternative, but navigation could be automated. Another important reason is to schedule the Roomba to navigate to its destination automatically. The automatic setting of goal location would also allow to check the meeting entries and the room state before sending the goal co-ordinates to the 'move_base' server. Otherwise, we cannot schedule the robots for navigation.

To address these issues, we present a navigation script using python and ROS client server model. For each individual robot, the goal co-ordinates are given to the script and the robots navigate to their goal locations using the ROS navigation stack by subscribing to the move_base server. The scripts have to be run on both the robots and they work with the individual move_base servers. The details regarding the implementation are discussed in the next section.

4.4 Scheduling the robots

The scheduling will have a control over the robot navigation depending on the calendar entries for the day. The idea is to check the meeting time and set the cleaning time 45 minutes prior to the meeting time. When the current time meets the cleaning time, then the robots should start navigating to the goal locations.

4.5 Algorithm for our system

Now, having discussed about the different tasks that are required, we will discuss the algorithm that binds all of them together. This is just the main functional algorithm and it does not contain the details of all the functions and internal data items.

Algorithm 4.1 Scheduling the Robot Navigation for cleaning

Initializations

broker \leftarrow 137.251.108.69*port* \leftarrow 80*topicState* \leftarrow "calendar/fhg.iao.vslab@gmail.com/state"*topicEntries* \leftarrow "calendar/fhg.iao.vslab@gmail.com/entriesToday"*messageReceivedFlag* \leftarrow False (Modified in on_message callback)*roomState* \leftarrow Set by MQTT message callbackConnect to server using *broker*, *port*Subscribe to *topicState*, *topicEntries***while forever do**

Call the on_message callback

if *messageReceivedFlag* **then** Get the *currentTime* Get the *currentDate* *timeDelta* = 45 mins *calendar_data* = Parsed meeting entries for today **while** *calendar_data* **do** Get *meetingTime* **if** *currentTime* < *meetingTime* **then**

break

end **end** *cleaningTime* = *meetingTime* - *timeDelta* **if** *roomState* == "ROOM_FREE" or *roomState* == "ROOM_FREE_LONGTERM" **then** **if** *currentTime* >= *meetingTime* **then**

Initialize ros node "navigation_goal"

Read goal coordinates from coordinates.yaml file

Instantiate the Navigation class with the coordinates

Call the method that navigates to goal

rospy.spin()

end **end** **end**

Wait for 3 seconds

end

With the above proposed algorithm, we can achieve the desired objective of automatic scheduling of the robots depending on the meeting times obtained from the calendar.

5 Realization of Swarm and Navigation

Let's dive into the implementation details in this chapter. In the previous chapter, we had an overview of what work packages the design of our system consists. Now, we will discuss all the technical details regarding the implementation of our proposed system.

5.1 Mapping of the office environment

As discussed in the earlier chapter, the mapping of the environment is done using a single Roomba. The robot is configured with a LiDAR sensor mounted on its top to perceive the surroundings in 2D. The LiDAR data is published on the `/scan` topic and is used for creating the map.

ROS offers a few packages for building the map such as `cartographer`, `hector_slam` and `gmapping`. In our case, we will use `slam_gmapping` node available in the `gmapping` package to build the map. We have to drive our Roomba in the environment and collect the laser scan data generated by the LiDAR sensor which is then consumed by the `slam_gmapping` node to build the map. To start the mapping process, we have to launch the robot's create autonomy using the command:

```
roslaunch ca_driver create_2.launch
```

After that the LiDAR sensor needs to be started and it is done with the command:

```
roslaunch ydlidar lidar.launch
```

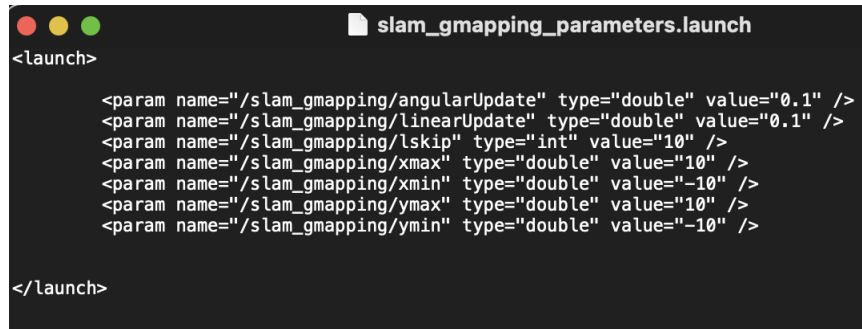
For driving the robot around, we need to control it using a keyboard. To enable the keyboard control of Roomba, we need to run the `teleop_twist_keyboard.py` script available in `teleop_twist_keyboard` package.

```
roslaunch teleop_twist_keyboard teleop_twist_keyboard.py
```

`slam_gmapping` uses laser data for robot's movement estimation and it's better if the laser scan data is available over a wide field of view [8]. We can improve the quality of mapping by setting a few `gmapping` parameters to different values. To do that, run the `slam_gmapping_parameters.launch` file before starting the mapping.

```
roslaunch gmapping slam_gmapping_parameters.launch
```


The `slam_gmapping_parameters.launch` file contains a few gmapping parameters that are adjusted to get a better quality of map.



```
<launch>
  <param name="/slam_gmapping/angularUpdate" type="double" value="0.1" />
  <param name="/slam_gmapping/linearUpdate" type="double" value="0.1" />
  <param name="/slam_gmapping/lskip" type="int" value="10" />
  <param name="/slam_gmapping/xmax" type="double" value="10" />
  <param name="/slam_gmapping/xmin" type="double" value="-10" />
  <param name="/slam_gmapping/ymax" type="double" value="10" />
  <param name="/slam_gmapping/ymin" type="double" value="-10" />
</launch>
```

Figure 5.1: Slam Parameters

The `xmax`, `xmin`, `ymax`, `ymin` parameters determine the extent of the map. The `lskip` determines the number of beams to skip when processing a laser scan message. The `linearUpdate` and `angularUpdate` decide the inclusion of a new scan after robot has moved a certain linear and angular distance [8]. Now that the parameters are set, we can start the `slam_gmapping` node and start driving the robot around. To start the gmapping enter the following command.

roslaunch gmapping slam_gmapping scan := /scan

The `slam_gmapping` node consumes the LiDAR sensor data available at `/scan` topic and generates the map. To get a good quality of map it is desirable to visit a location twice and drive the robot slowly. Also a good technique during turns it to stop the robot and then rotate it in the desired direction and start driving again. This helps to have the final map close to actual design of the environment. After driving around the environment for some time the map is generated and it has to be saved. To save the map run the `map_saver` available on the `map_server` without stopping the gmapping.

roslaunch map_server map_saver -f file_name_for_the_image

Passing the filename saves the map with that name and creates a `.pgm` and `.yaml` file. The `.pgm` file is the image of the map and `.yaml` file contains the map metadata. With the process we just discussed, we have the map of our environment ready.

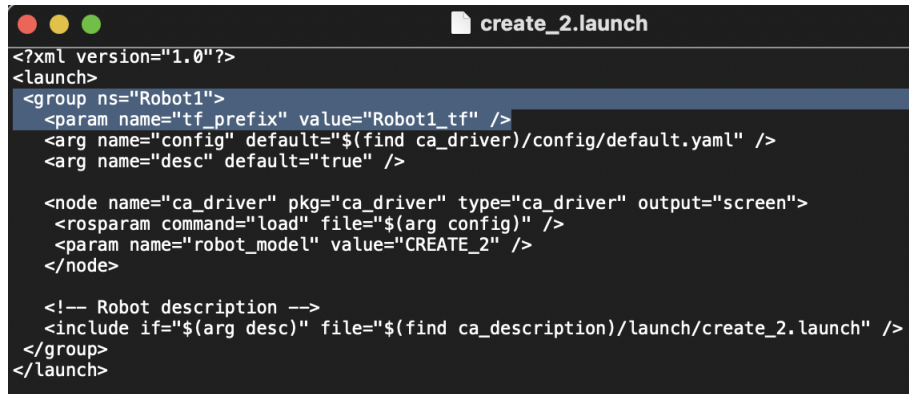
5.2 Creating a swarm and Map sharing

5.2.1 Namespace configuration

The most important requirement of our swarm system is presence of two robots and map sharing between them. The map of the environment is ready and now we look forward to configure the two Roombas in different namespaces using the ROS namespace functionality. Both the robots are configured with a LiDAR sensor, a Raspberry Pi 4 with Ubuntu 16.04 and ROS kinetic and a serial 8-DIN to USB cable to communicate between the Roomba and Raspberry pi.

The namespace configuration is required for all the nodes and topics related to the navigation or that are required for the functioning of the ROS navigation stack. Here, we will present the namespace configuration for one robot, the same can be followed for the second robot as well.

The `create_2.launch` and the `ydldidar.launch` files are modified to launch the first Roomba in the namespace ***Robot1***.

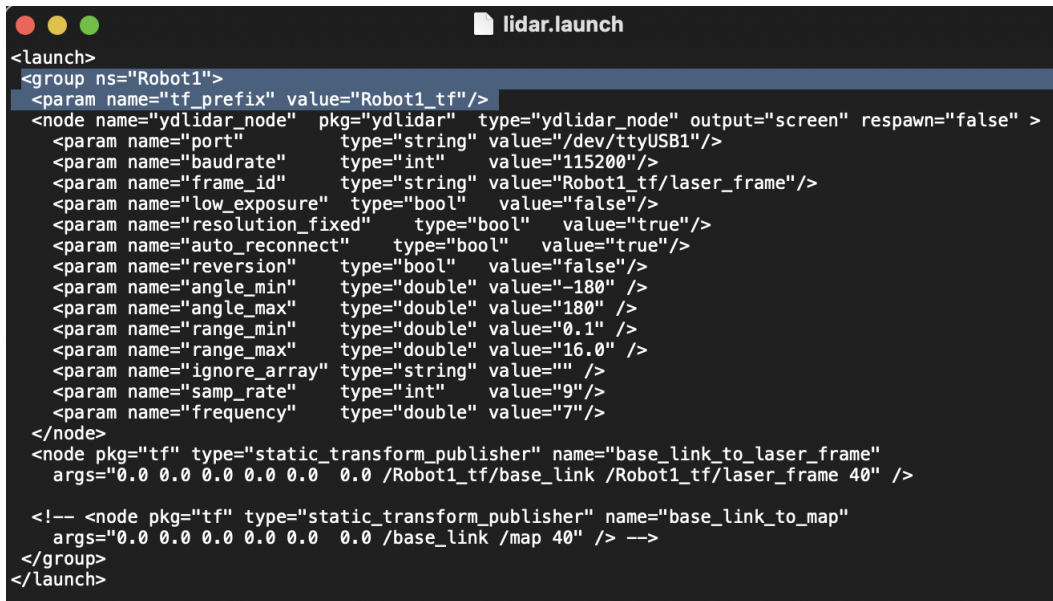


```
<?xml version="1.0"?>
<launch>
  <group ns="Robot1">
    <param name="tf_prefix" value="Robot1_tf" />
    <arg name="config" default="$(find ca_driver)/config/default.yaml" />
    <arg name="desc" default="true" />

    <node name="ca_driver" pkg="ca_driver" type="ca_driver" output="screen">
      <rosparam command="load" file="$(arg config)" />
      <param name="robot_model" value="CREATE_2" />
    </node>

    <!-- Robot description -->
    <include if="$(arg desc)" file="$(find ca_description)/launch/create_2.launch" />
  </group>
</launch>
```

Figure 5.2: Create_2 launch file



```
<launch>
  <group ns="Robot1">
    <param name="tf_prefix" value="Robot1_tf"/>
    <node name="ydlidar_node" pkg="ydlidar" type="ydlidar_node" output="screen" respawn="false" >
      <param name="port" type="string" value="/dev/ttyUSB1"/>
      <param name="baudrate" type="int" value="115200"/>
      <param name="frame_id" type="string" value="Robot1_tf/laser_frame"/>
      <param name="low_exposure" type="bool" value="false"/>
      <param name="resolution_fixed" type="bool" value="true"/>
      <param name="auto_reconnect" type="bool" value="true"/>
      <param name="reversion" type="bool" value="false"/>
      <param name="angle_min" type="double" value="-180" />
      <param name="angle_max" type="double" value="180" />
      <param name="range_min" type="double" value="0.1" />
      <param name="range_max" type="double" value="16.0" />
      <param name="ignore_array" type="string" value="" />
      <param name="samp_rate" type="int" value="9"/>
      <param name="frequency" type="double" value="7"/>
    </node>
    <node pkg="tf" type="static_transform_publisher" name="base_link_to_laser_frame"
      args="0.0 0.0 0.0 0.0 0.0 0.0 0.0 /Robot1_tf/base_link /Robot1_tf/laser_frame 40" />

    <!-- <node pkg="tf" type="static_transform_publisher" name="base_link_to_map"
      args="0.0 0.0 0.0 0.0 0.0 0.0 0.0 /base_link /map 40" /> -->
  </group>
</launch>
```

Figure 5.3: Lidar launch file

The tag `<group ns>` associates the name prefix to all the nodes and topics launched within the group. The parameter `tf_prefix` is added to update the transform tree to include the transform prefix. If this is not done, though the nodes and topics are launched with name prefix the robot frame transform is not updated and thus the namespaces does not work. This is how the transform between different frames of the robot looks like without configuring it with namespace.

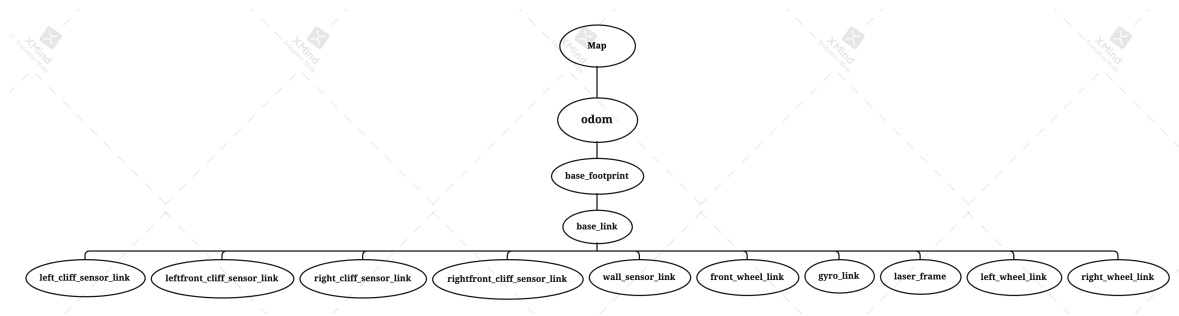


Figure 5.4: Robot transform tree without namespace

The entities in circles are the frames of the robot and the links show the relationship between them. Also the important information like the broadcaster and the frequency of broadcasting between various frames is recorded. We will see that after the namespace configuration, the frames will be prefixed with **Robot1_tf**.

Next, we launch the `move_base` server which is responsible for the ROS navigation stack. The `move_base.launch` file loads all the parameters for global and local costmap, global and local planners and the `amcl`.

```

move_base.launch
<launch>
  <master auto="start"/>
  <!-- Run the map server -->
  <node name="map_server" pkg="map_server" type="map_server" args="$(find roomba_2dnav)/vslabrun7.yaml">
  </node>

  <group ns="Robot1">
    <param name="tf_prefix" value="Robot1_tf" />
    <!-- Run AMCL -->
    <include file="$(find amcl)/examples/amcl_omni.launch" />

    <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
      <rosparam file="$(find roomba_2dnav)/costmap_common_params.yaml" command="load" ns="global_costmap" />
      <rosparam file="$(find roomba_2dnav)/costmap_common_params.yaml" command="load" ns="local_costmap" />
      <rosparam file="$(find roomba_2dnav)/local_costmap_params.yaml" command="load" />
      <rosparam file="$(find roomba_2dnav)/global_costmap_params.yaml" command="load" />
      <rosparam file="$(find roomba_2dnav)/base_local_planner_params.yaml" command="load" />
      <param name="base_local_planner" value="base_local_planner/TrajectoryPlannerROS" />
      <!-- <param name="base_local_planner" value="dwa_local_planner/DWAPLannerROS" /> -->
      <param name="base_global_planner" value="global_planner/GlobalPlanner" />
      <!-- <rosparam file="$(find roomba_2dnav)/dwa_local_planner_params.yaml" command="load" /> -->
      <rosparam file="$(find roomba_2dnav)/global_planner_params.yaml" command="load" />
    </node>
  </group>
</launch>

```

Figure 5.5: Move base launch file

We first launch the map server with the desired map. The map server is launched in only one robot as it is a common node and has to be shared between the two robots. Since, it is a common node, map server is independent of any namespace. After map server, the navigation stack parameter files and the `amcl` files are launched with the desired namespace depending on the robot. The modification and launch of the files mentioned is sufficient for the robots to operate in different

namespaces and share a common map. Further, we need to setup the ROS master configuration using the IP addresses to enable the ROS master on one robot and the other robot running on the same master.

5.2.2 ROS master configuration

To configure a single ROS master system for our swarm, we have to set a variable **"ROS_MASTER_URI"** with the appropriate IP address in the bashrc file on both the Roomba's as well as the linux PC. The IP address of the robot which is the master has to be set in **"ROS_MASTER_URI"** on the other robot and the linux system that runs the rviz tool.

```
export ROS_MASTER_URI=http://10.36.28.212:11311
export ROS_HOSTNAME=10.36.28.212
export ROS_IP=
```

Figure 5.6: Robot1 bashrc file (ROS Master)

As shown, the robot1 runs the ROS master and it's IP address is set correctly. If the host and the ROS master IP's are same then the robot runs the ROS master.

```
#ROS Config
export ROS_MASTER_URI=http://10.36.28.212:11311
export ROS_HOSTNAME=10.36.28.216
export ROS_IP=
```

Figure 5.7: Robot2 bashrc file

```
#ROS Config
export ROS_MASTER_URI=http://10.36.28.212:11311
export ROS_HOSTNAME=10.36.28.213
export ROS_IP=
```

Figure 5.8: Linux PC bashrc file

The settings for the robot2 and linux PC to work with the ROS master running on robot1 are shown above. The **'ROS_HOSTNAME'** contains the IP address of the host system. With these network settings and the namespace configurations for both the robots, we are able to establish a swarm and make the robots share the same map of the environment.

Following all the configuration as discussed in the last two sections, we get a new transform tree of the swarm system sharing a common map.

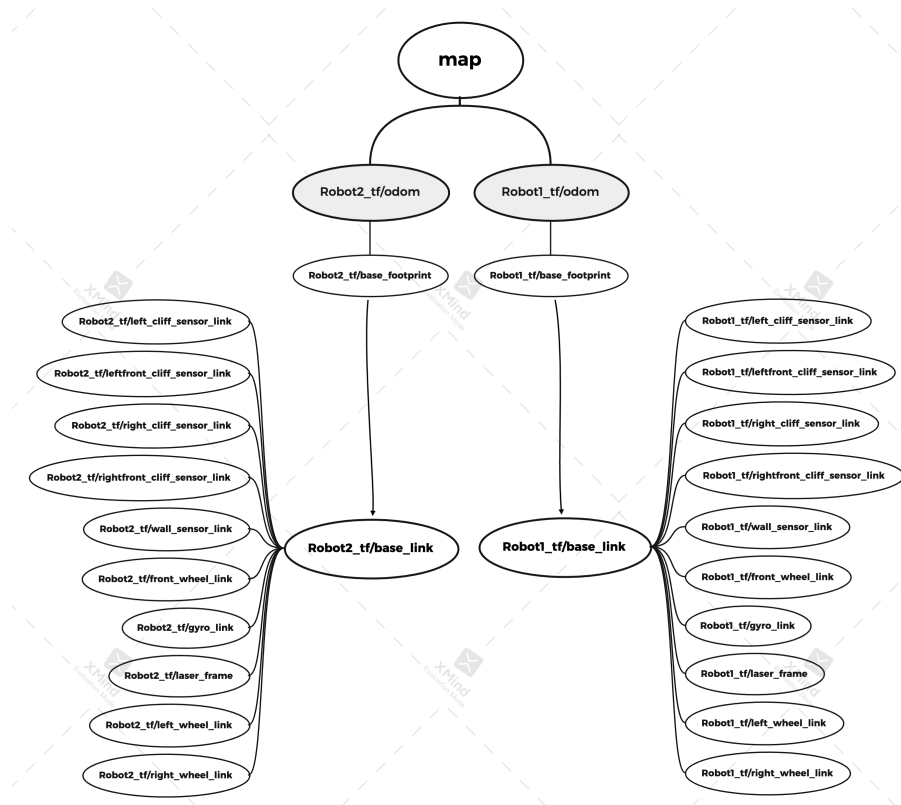


Figure 5.9: Robot Swarm Transform tree

5.3 Navigation to goal using ROS

In the RVIZ it is possible to set the initial pose of the robot on the map and then give it a goal location for navigation. When we have multiple robots sharing a map then it is difficult to set the goal location for all the robots one by one as we need to change the topic associated with the 2D Nav goal button. The initial position however is set using 2D Initial Pose button in RVIZ. Thus, we present a solution to navigate the two robots to their respective goal locations when the initial pose is set in RVIZ using the 2D Initial Pose button without using the RVIZ 2D Nav Goal button. Another reason to implement the navigation task is to be able to schedule the Roomba according to the meeting timings and room availability fetched from the calendar.

We will use the actionlib client server architecture available in ROS, typically called as ROS services. Using ROS services, it is possible to send request to a node to perform a task and receive a response. The ActionClient and ActionServer communicate via 'ROS Action Protocol' and provide users simple APIs to request goals [9]. The ROS move_base package is an implementation of actionlib server which attempts to reach a given goal with mobile base. It links the global and local planners together to provide the navigation and also maintains the local and the global costmaps. The move_base node takes in goals containing geometry_msgs/PoseStamped messages, it is basically an implementation of SimpleActionServer in ROS [10]. We present a SimpleActionClient in ROS that sends the desired goal co-ordinates to the move_base server to achieve the expected navigation results.

The SimpleActionClient is implemented in a python class named *NavigateToGoal* available in file *goal_navigate.py*. The class initializes itself with the goal co-ordinates given to it while creating its instance. The goal co-ordinates are given as position i.e x and y co-ordinates and quaternion for its orientation. The quaternion consists of four co-ordinates, the xOrientation, yOrientation, zOrientation and wOrientation. All the six co-ordinate locations are pre-stored in a *coordinates.yaml* file and read into the main script during initialization of NavigateToGoal class. The class contains a method *movebase_client* which is responsible for implementation of the SimpleActionClient to send the goal co-ordinates to the move_base server. It does the following in sequence:

1. Create a SimpleActionClient object and set the action as *MoveBaseAction*.
2. Create an instance of MoveBaseGoal.
3. Set the target frame id as 'map' and assign the goal co-ordinates to the MoveBaseGoal instance.
4. Send the goal location by passing the MoveBaseGoal instance to send_goal method of client object.
5. Wait for the result. Check the state and if 'SUCCEEDED' then the goal has been reached.

We have discussed the implementation of the ROS SimpleActionClient to navigate to the goal location. In the next section, we will discuss on how this is associated with the scheduling of the robots.

5.4 Scheduling the Roomba

The scheduling of the swarm system is done based on the calendar entries for a given day. The calendar entries consist of the information on the current state of the room, the meeting start and the end times. The messages from the calendar are sent to a server using the MQTT protocol and thus can be viewed in the MQTT Explorer application. We present an implementation using python and paho mqtt client to fetch the calendar entries from the server and use them to schedule our robots.

The python class named CalenderMQTT takes care of the connection to the server and mqtt related callbacks. During its instantiation, the class takes in the IP address of the server and the port number and creates a mqtt client object. The method connectionToBroker implements the connection task using the required credentials and once connected returns a client object. There are two static methods in the class. One is the mqtt on_connect callback which returns the status of the connection with the server. The other one is the mqtt on_message callback which processes the messages received by the client. We subscribe to two different topics to fetch the data for the state of the room and the meeting details. The corresponding messages are handled by the same on_message callback.

The two classes are integrated with the algorithm presented in section 4.4. Both the Roombas are scheduled to navigate to their respective destinations in accordance with the algorithm in section 4.4. The source code is available on the github repository.

6 Discussion of the Results

We will now look at a few results that we got on the implementation of our proposed solution. The first result is the map of the environment followed by the swarm system and finally the scheduling of both the robots as per the meeting times. Let's have a look at the results obtained.

6.1 Environment map

Here is how the home office environment looks like for which the proposed system is designed. The left side represents the office section and the right side is the home part. It is a single room with a partition planned in between to separate the home and the office sections.



(a) Office side



(b) Home side

Figure 6.1: Integrated home office environment

The idea is to have both the Roombas at their respective docking stations near the entrance or anywhere in the room and navigate to their fixed goal locations which will be assigned to them. Both the robots will ideally have different calendar data, so they will act according to the meeting information received and go to their locations and start the cleaning task.

We have built two maps of the same environment and we will now have a look at both of them. The one on the left does capture all the objects in the room but partially and not in great details. We did a remapping of the room again by moving the robot slower, especially during turns and moved it closer to the objects and got a better map which is seen on the right. As for our test purposes, we have used the one on the left during the implementation stages. The new map is also integrated and tested with the swarm system.

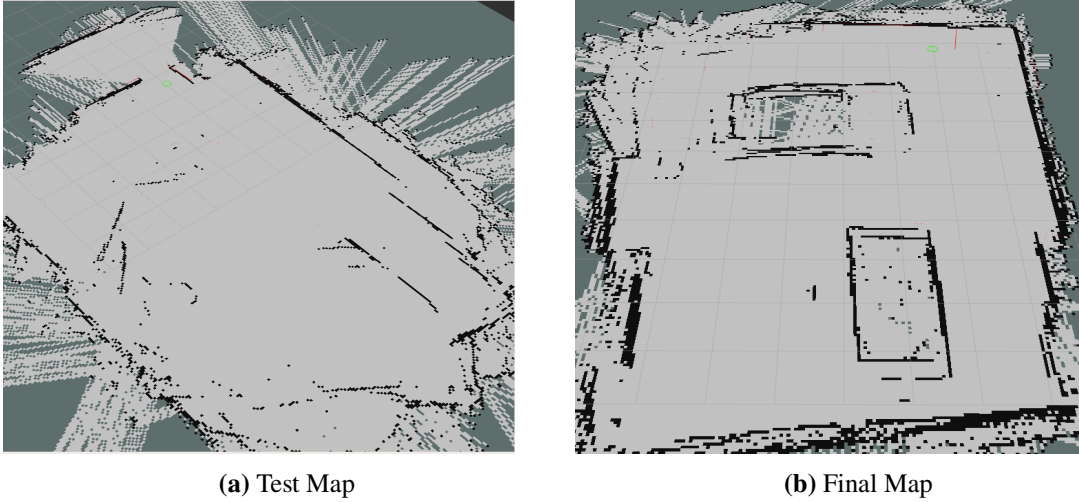


Figure 6.2: Integrated home office environment

6.2 Swarm of two Robots

The important results for swarm with two robots sharing the same map are the transform tree and the rosgraph which represents the nodes, topics and the connection between them. We have seen in the previous section how the transform tree is modified to have two robots in a single map. Here, we will analyze the results of the rosgraph before and after the swarm system. The rosgraph can be generated by the ROS command:

```
roslaunch rqt_graph rqt_graph
```

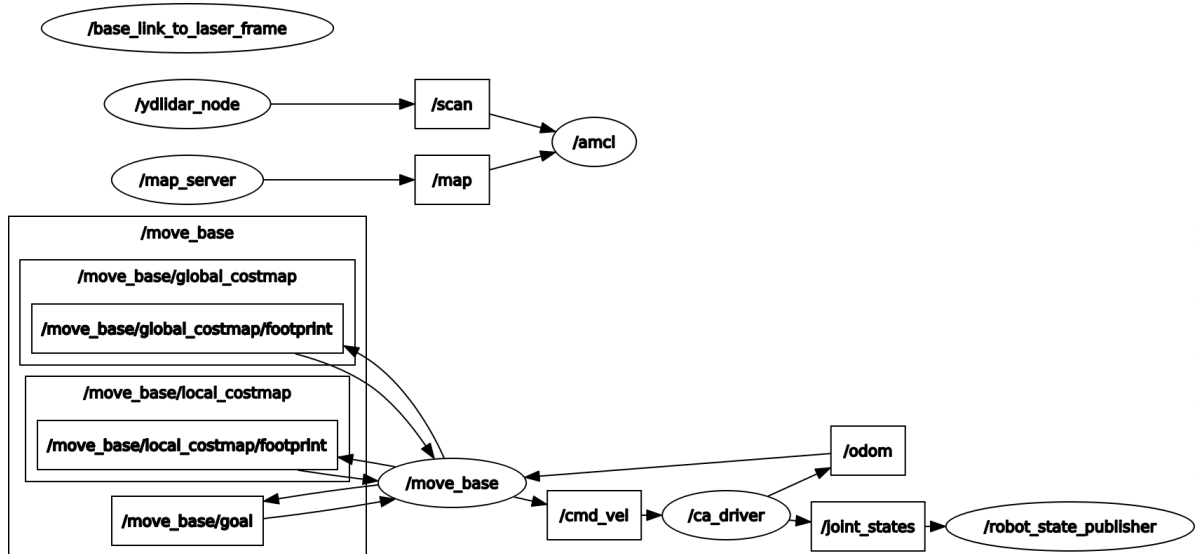


Figure 6.3: Rosgraph before swarm

In Figure 6.3, we see the rosgaph of the robot before the swarm. The entities in rectangles are the topics and the nodes are represented in oval. The arrows show the flow of data, outwards from a node means that it publishes on a certain topic and inwards means that the node subscribes to the topic where the arrow is coming from.

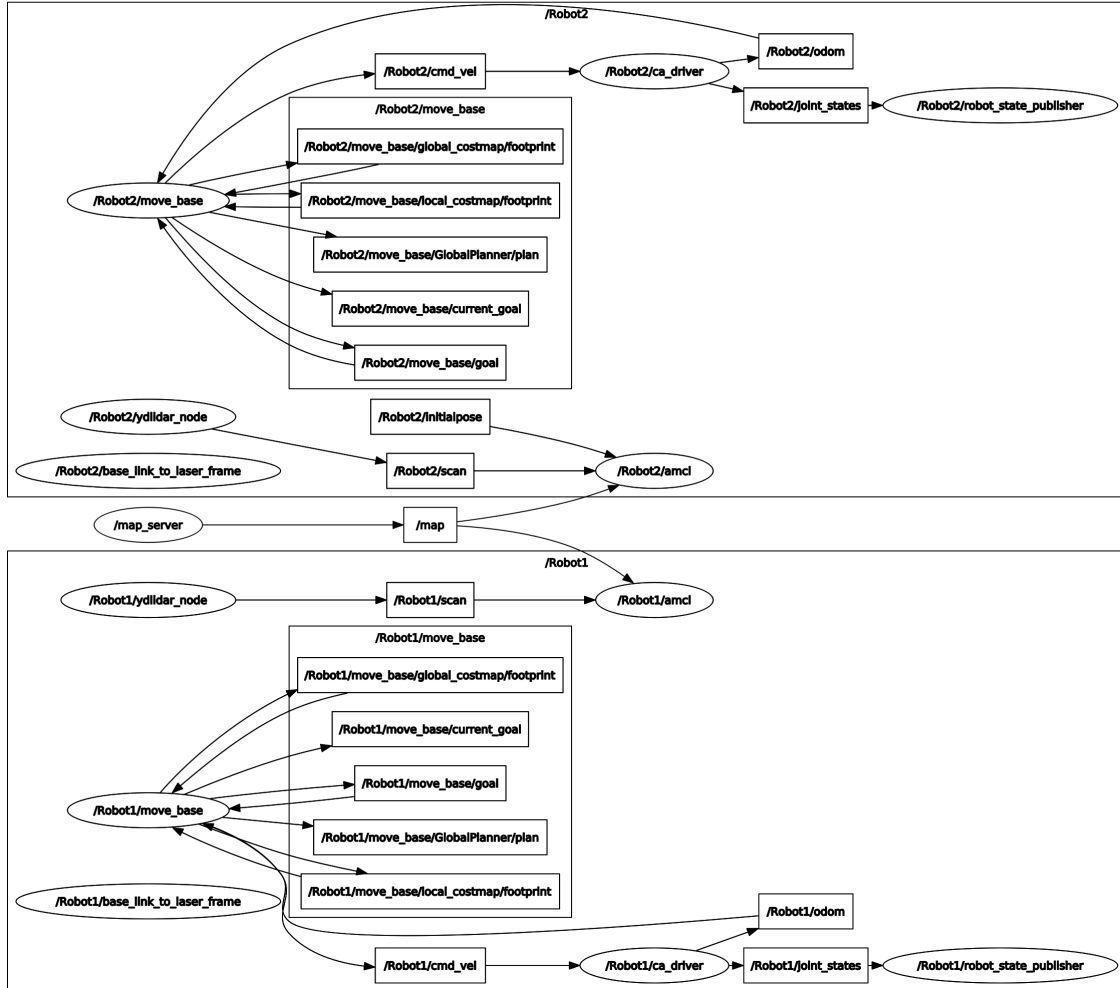


Figure 6.4: Rosgraph after swarm

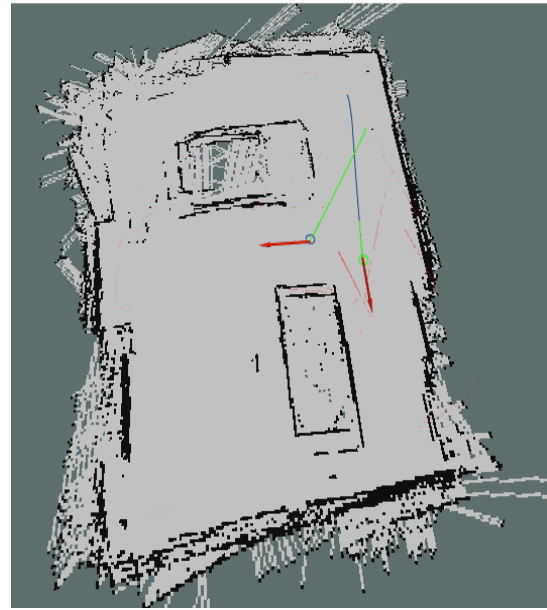
In the above figure, we can see that both the robots are launched in their individual namespaces and the nodes and topics are modified accordingly. Also, the map server is not in any namespace and the amcl node of both the robots has successfully subscribed to the map. This enables the swarm of the two robots into a single map.

The images below show both the robot footprints in the map of the environment. The figure on the left represent the two robots with their initial position set in the map. The one on the right is the final image taken after both the robots have attained their destination coordinates given to them.

We have discussed and seen the important results for our system. In the next chapter, we will discuss the possible further extensions that can be embedded into our system and the challenges that could arise while doing so.



(a) Robot Swarm - RVIZ



(b) Swarm Destination Reached

Figure 6.5: Roomba Swarm

7 Conclusion and Future Scope

In this project, we have presented a low cost solution to address the problem of cleaning in home office environment by creating a swarm system of two iRobot Roomba robots with the help of ROS framework running on a Raspberry Pi. The project discusses the technical aspects of robot's navigation, mapping and creation of a swarm system using a single ROS master and its visualization in RVIZ. We have also presented an algorithm for scheduling of the robots depending on the calendar information that they receive. We have not been able to actually start the cleaning process programatically as there is no python library that allows to send the 'clean' command to 'create_2' robot models. So, for the demonstration purpose we have shown the navigation of the robots to the goal location. The cleaning command will be integrated as soon as it is available for create_2 models in python.

Further, we would like to extend this project by creating a communication channel between the two robots. We plan to implement a case where both the robots would subscribe to each others battery levels and depending on the battery status decide the cleaning plan. The battery status is published on battery/charge topic and in our case on 'namespace'/battery/charge depending on the robot. The communication will be done using the MQTT protocol with the MQTT broker residing on the linux system. Both the robots will check the partner's battery and then their own. Depending on the level of charge available in both the robots, a decision will be taken to launch the robot with sufficient battery level for the cleaning task.

Another aspect is the return of the robot to its initial position or the place where the docking station is kept. For this the localization of the robot has to be near perfect in the environment. When the robot cleans the area around its goal location, its position in the map will change. To bring it back to its original position or the docking station, we think the initial position needs to be set again on the map or else the localization errors would add up and the robot will localize itself poorly, often missing its target location. So far, setting the initial position is done using RVIZ. In the future, if possible this can be made independent of RVIZ and then the return of the robot to its docking station would be possible from large distances.

8 References

- [1] “Roomba® vacuum robot | iRobot.” <https://www.irobot.de/roomba> (accessed May 05, 2021).
- [2] S. H. Juan, F. H. Cotarelo, and S. H. Juan, "Multi-master ros systems": 2015.
- [3] “navigation - ROS Wiki.” <http://wiki.ros.org/navigation> (accessed May 08, 2021).
- [4] Fabro, João Guimarães, Rodrigo Oliveira, André Becker, Thiago Brenner, Vinícius. ROS Navigation: Concepts and Tutorial (2016).
- [5] M. Garzón et al., “Using ROS in Multi-robot Systems: Experiences and Lessons Learned from Real-World Field Tests,” in Robot Operating System (ROS): The Complete Reference (Volume 2), A. Koubaa, Ed. Cham: Springer International Publishing, 2017.
- [6] “geometry/CoordinateFrameConventions - ROS Wiki.” <http://wiki.ros.org/geometry/Coordinate-FrameConventions> (accessed May 03, 2021).
- [7] A. Filotheou, E. Tsardoulas, A. Dimitriou, A. Symeonidis, and L. Petrou, “Quantitative and Qualitative Evaluation of ROS-Enabled Local and Global Planners in 2D Static Environments,” J Intell Robot Syst, vol. 98, no. 3–4, pp. 567–601, Jun. 2020.
- [8] “Programming Robots with ROS [Book].” <https://www.oreilly.com/library/view/programming-robots-with/9781449325480/> (accessed May 23, 2021).
- [9] “actionlib - ROS Wiki.” <http://wiki.ros.org/actionlib> (accessed May 30, 2021).
- [10] “move_base - ROS Wiki.” http://wiki.ros.org/move_base (accessed May 30, 2021).
- [11] H. Harms, J. Schmiemann, J. Schattenberg, and L. Frerichs, “Development of an Adaptable Communication Layer with QoS Capabilities for a Multi-Robot System,” in ROBOT 2017: Third Iberian Robotics Conference, Cham, 2018.