DAT160 Semester Project - Report

Gunnar Fimreite Kleiven

Link to GitHub repository:
`https://github.com/gunnarkleiven/DAT160SemesterProject`

1 Introduction

1.1 Task

Our task is to implement a Search and Rescue mission with two Turtlebot3 robots in ROS. The focus will be on things we have learned in this course, from mobile robots, location and navigation, to robot teams and their architecture. To utilize these ideas and concepts, we're going to use the Robot Operating System, with its features and communication structure.
The robots are supposed to navigate in an unknown environment and locating "injured persons" and fires throughout the map. There needs to be a line of communication between the robots, both in sharing the generated map of the environment as well as coordination around bigger fires. Within a given time period, the robots ideally need to explore the whole map, in order to achieve the goal of locating as many targets as possible.
The robots only have a small set of sensors available, and they need to generate their own global map without any prior knowledge except for their starting position. They need to be able to dynamically adapt, as they will be tested in several different environments.

1.2 My approach

Initially when starting this project, my approach was to start following and implementing the suggested controller layout to the different parts of the system. This gave a good introduction to how we should structure the system, and which of the nodes would communicate with each other. Since we are working with two of the same kind of robot, I wanted to utilize the namespaces for the robots, so that we would not need to create two separate files for two nodes with the same functionality. I also wanted the leader node to work as a centralized communication controller, and have it do most of the planning.

2 Design Process

2.1 Iterations

When working with this project, my approach was to split up the tasks up in smaller parts, and solve them one by one. I also wanted to finish a feature or step of the progress before moving on to the next, because a lot of the sub-tasks in this project is depending on the previous. Also, since this is an introduction to be working in ROS, I was aware that I would probably not be able to create a "proper" ROS design and implementation. By that, I mean that I would most likely not use the ROS communication patters optimally, but rather focus on making them work in my case and this particular set of problems. This means, the design would have some "ad hoc" solutions, and might not be as extensible as good software often should be.

The overall design of the project went through a lot of different designs, as my approach to solving a problem in ROS often meant a testing with use of different communication patterns. Some of the iterations

went something like this: by going through trial and error, develop a simple, working solution to a sub-problem → do a lot of manual testing to check if I actually get the desired behavior from the robots → by doing so, realize there is a better way to solve it → refactor and implement the bettter way to do it → test to see that is still solves the given problem. Some of these design problems will be discussed further later in the report.

2.2 Division of work

I was the only one in my group, so everything in the project was done by me. However, I need to add that got a lot of good help from the group leaders, and some of the ideas and snippets of code was created after input from Laurenz. Also, some lines lines of code were retrieved from the official ROS documentation.

2.3 Encountered problems

One of the first and major challenges I faced, was how I should effectively use the different communication patters to utilize both the robots and ROS in general, in the best way possible. To find out which communication patterns to use, I often tried out multiple types, and went with the one that best solved my concrete problem. An example of this, is how the leader should should send locations to the robots. I tried a normal Publish-Subscribe pattern over a topic at first. The robots would send a message over their own topics, and the leader would subscribe to both of them. When the leader got the request, it responded back on each separate topic. After implementing this, I quickly realized that instead of sending one message forth and another one back on different channels, it was obvious that a ROS Service would work much better for the issue. It is however, a good example of how looking back on a problem often gives a clearer view of the solutions. I guess that's why they call it hindsight.

Another problem, which actually took a while to notice, was the fact that the leader found unexplored position on the map inside walls. A screenshot of this can be seen in Figure 1. The robots would just stand still when receiving a position like this, as they obviously have no way of navigating here.

3 Final implementation

3.1 How to run the system

There hasn't been made any changes to the launch files, so to run the system you need to run two launch files, in two separate terminals. An example of executing the code would be to open one terminal and use the command:

```
$ roslaunch multi_robot_challenge rescue_robots_w0.launch
```

And then opening another terminal and launching the controllers:

```
$ roslaunch multi_robot_challenge controller.launch
```

3.2 Topics overview

There are a great number of active topics when running the project. Most of them originate form the libraries used in the launch files, while some of them was added by me to use. It would not be useful to list every

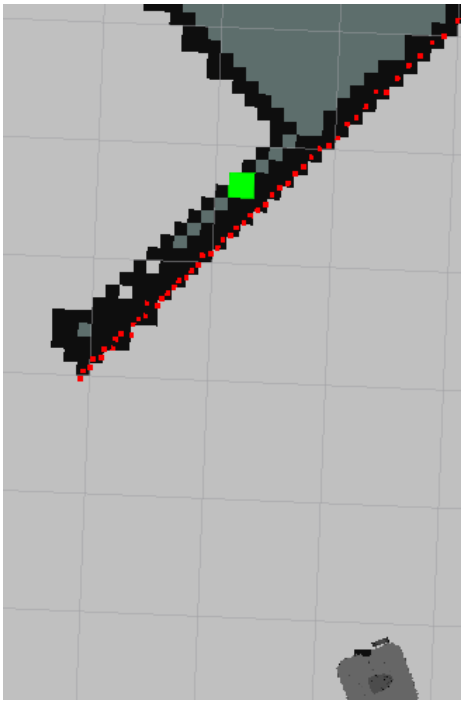

Figure 1: Position inside a wall

active rostopic, but I created an overview of the topics which are published and subscribed to by the leader node and the robot nodes. The overview can be found in Table 1.

Note that the different topics in the table has a deliberate difference with the use of "/". The reason for this is because of the namespaces in ROS. The robots are running on two different namespaces, namely "/tb3_0" and "/tb3_1". Because of this, it makes a big difference if we use "/" in the topic names or not. By omitting the "/" from the topic names, we are automatically including the namespaces, e.g., subscribing to **"odom"** from the "/tb3_0" robot would actually mean subscribing to **"/tb3_0/odom"**. On the other hand, subscribing to **"/odom"** would make the node subscribe to the topic without the namespaces.

3.3 Flow of the system

To help understand the flow of the system, here is a brief overview after both the launch files are executed:

1. The leader node and the two robot nodes initiate
2. All three nodes wait for the map merging to publish the map. This takes about 20 seconds in the simulation, which means the actual duration will depend upon the "real time factor" of which the simulation is running on. Both the leader and the robot nodes are subscribing to the **/map** topic.
3. When the first map is published, the leader is ready to receive requests. The robot nodes are manually set to start navigating a few meters straight forward. This is added to "kickstart" the map exploration.
4. After the initial navigation, the robot nodes begin the loop of requesting new positions. They send a request (as clients) over a service to the leader.
5. The leader receives the request, along with the position of the robot making that request. The leader then:

Leader	
Subscribes to: /map /unreachable_position Service server: /robot_service	Publishes to: /map_available /targets
Robots	
Subscribes to: leader_commands odom ar_pose_marker /map_available /map Service client: /robot_service move_base Action client: move_base	Publishes to: /cmd_vel /unreachable_positions

Table 1: Overview of the publishers and subscribers

(a) Calculates all the desirable, unexplored locations, with the algorithm explained in Section 3.4.

(b) Calculates all of the distances from the robot position to all the desirable positions.

(c) Sorts these distances.

(d) Picks the position based on the parameters discussed further in Section 3.4

6. Robots sets its state to "navigating", and sets its target to the received position. Starts navigating towards it.

7. When the robot reaches its destination, the navigation state is set to false.

   (a) After a certain number of navigation, the robot does a 360 degrees spin, to (hopefully) be able to locate any AR tags it might have missed.

8. The robot repeats from step 4, by requesting a new target position.

3.4 Exploration and navigation algorithm

One of the biggest tasks in this project, is to generate the robot's exploration of the unknown map. Fortunately, we could could use the built in "move_base" to move the robot to a target. However, I still had to implement an algorithm for doing the exploration of the unknown map.
The idea behind my solution, is to have the robots request new positions from the leader every time they are in a state of "not currently navigating". They do this over a ROS Service communication pattern, where the robots work as clients sending requests, and the leader works as the server sending responses. Before receiving the request, the leader is not aware of the positions of the robots, and it also does not take into account which of the robots sends the request. This information is sent with the request message by the robots.
Whenever the leader gets a request over the "/robot_service topic, it first calculates all the desired positions on the map. In the scope of this project, a desirable position is a position which will help us further explore

the map. More specifically, it is a (discovered) point on the Occupancy Grid which is adjacent to an unexplored area, as well as not being adjacent to a wall.
An extraction from the algorithm can be seen in Listing 1.

```
all_desired_positions = []

for x in range(len(self.map.data)):
    if self.map.data[x] == 0:
        if self.check_adjacent("undiscovered", x, w_count, width, self.map.data) and
            self.check_adjacent("wall", x, w_count, width, self.map.data) and
                self.mapToPosition(x) not in self.unreachable_positions:
                    all_desired_positions.append(x)
```

Listing 1: Extract from calculate_all_next_positions()

A short, step wise explanation of the algorithm:

- Loop through all the indexes of the map grid
- Check if the value on that point is 0, which means it's discovered and has no obstacle. If it is, then
  - Check if the position is adjacent to an undiscovered point,
  - Check if the position is not adjacent to a wall, and
  - Check if the position is not in the list of unreachable positions.
  - If all of these are true, append the index to the list
- The function to check adjacent positions imagines the grid to be a 2D grid, and checks above, below, left, and right, as well as checking for out of bounds
- In the end of this function, we map all the index positions in our list to ROS Points, using the given utility function mapToFunction()

The next important part of the algorithm is to find the closest desired position to the robot, which is done using the function find_closest_desired_point() in the file `scripts/leader.py`. At first, it iterates through all the desired positions, and calculates the distance to the robot's position. It then sorts the result list. After this, it points at the first positions, and iterates through it until it finds the closest positions which is also a certain distance from the robot, as well as not being in the list of unreachable positions.
Finally, the leader returns this position to the robot the robot, over the "currently open service request".

3.5 Movement parameters - Making the robots a tiny bit smarter

As with all kinds of simulation (and also with software in general), there are multiple ways that it can go wrong, even tho you feel certain that the solution you have should be working. Making these two turtlebots do what you intend them to do very rarely goes the right way in the beginning. This was especially relevant in the map exploration part of the project, where the leader node would constantly choose horrible positions, and the robot nodes would often refuse to go where they were sent. To fix some of these issues, I added some constraints and parameters, which I found very helpful to improve the exploration algorithm.

The first thing I quickly noticed, was that that the robots were often assigned positions very close to them, which led to short movements and a lot of small steps. With every step, the robot's orientation would also be slightly off, and the movements became more inaccurate. To solve this, I did not just have the leader give the robot its closest position, but the closest position which is also a certain distance away (as also mentioned in Section 3.4).

Another thing I added, which had probably the most impact, was that I added timers for the robot movements. This was created using the ROS class Timer [1]. The problem I wanted to solve, was when

the robots spent too much time moving forward, e.i., when their move_base navigation wanted to send them to an unreachable position and they were just standing still. So, whenever a robot started the "move_base action" i started a timer countdown. If they did not move a certain distance before that timer went out, it would cancel the action. On the other hand, if they did move over a distance, the timer would reset. This really helped a lot, because there were a lot of times where the robots would just stand still and turn around because they could not reach the given position. In most of the cases, this was because the location was too close to the wall. By using the timers, this would stop the robots from forever attempting to go where they could not.

Using the timers forces the robots to actually complete their navigations. But what if they just got reassigned the unreachable position from the leader every time? This question was the obvious follow up to the previous paragraph, and I solved it by simply having the robots publish over a topic to the leader whenever they could not reach a position. The leader would then store these positions in a list, and make sure it did not send the robots a position from the list of unreachable positions (which can be seen in code line 7 in Listing 3.4).

3.6 Implemented features

I did not manage to complete all of the features to successfully execute a search and rescue mission. I did however, manage to do most of it. Most of the time was spent on implementing the exploring functionality, and the end result here was really good. To see an example of a completed map exploration, see the YouTube links in Section 6. The robots are also capable of checking for AR tags as well, however this feature did for some reason often give false positives. So, the robots do look for AR tags along with the exploration. Unfortunately, they do not store the positions in the parameters, as suggested in the semester project description. It would probably not be too hard to implement, but I chose not to focus much on it, as the competitions were cancelled anyways. The final missing part is the coordination between the robots when discovering a big fire, where my potential solution is discussed a little further in Section 5.2.

4 Experiments

4.1 Manual testing

Most of the testing I did was manual testing, i.e., running the project and observing the robots' behavior. This is a very time consuming way to test the application, because it requires you to both wait for the whole project to initiate, as well as often wait for the robots to reach a certain point of execution. I assume that there is a better way to test functionality in ROS, but if so, I did not use it.

For the algorithm described in Listing 1, I created some test data inputs to validate that the checks done actually returned the proper positions. They were obviously tiny data-sets compared to the input from the actual map.

4.2 Markers

The greatest tool I had for experimenting and testing, was to use the Marker tool in RViz[2]. Every time the robot received a new position and set it as its target, it also published a marker to RViz. This was a great way to visualize the steps the robot would take, as well as to always have an updated way-point too see where it was going. I manage to catch a lot of bugs with this tool, and it was very helpful when setting the parameters in Section 3.5. A screenshot of the markers in RViz can be seen in Figure 2

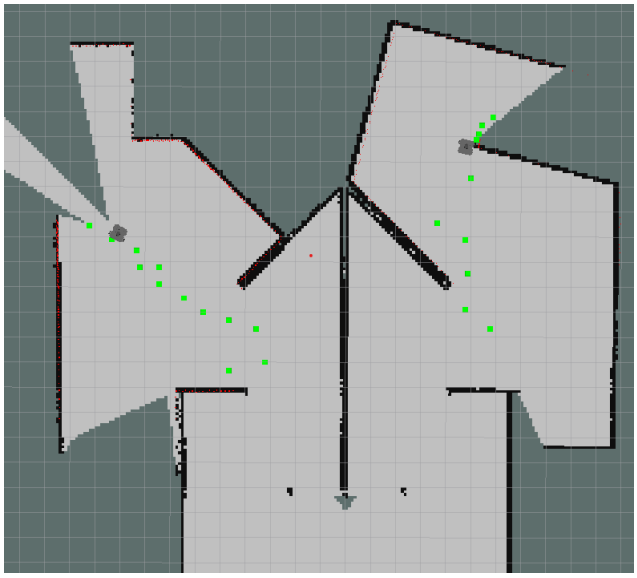

Figure 2: Markers in RViz

5 Conclusion

Our task was to implement a Search and Rescue mission, where the robots had to navigate in an unknown environment and look for "tags" around the world. In my opinion, I mostly succeeded in achieving these goals. The robots are (most of the times) fully able to explore the whole unknown map, as well as look for and register AR tags along the way. Furthermore, the design and flow of communication in ROS is well utilized, and the project should have good extensibility if I were to expand the functionalities, or add more robots.

5.1 Experiences

The semester project was a good way for us to utilize what we have learned about ROS in practise. Working on a specific task like this is a very good way to learn and get experience in ROS. After this project, and the previous assignment in ROS, I fell like I have learned a whole lot about hands on usage of ROS, and a lot of its features.
I don't regret being the only one on my group, but in hindsight it would have been much less work for me if I had been on a group. This is off course assuming there would be more people taking this course. The reason is that I have spent ***a lot*** of time on this project. Mostly because of the difficulties of working in ROS and with simulations; there are so many small errors that makes the robots do often weird and unpredictable things. It's often very hard to locate where the mistake originates from as well, since most of the times it's not due to the program terminating because of some errors in the code. Combined with the time it takes to start the launchers, it makes for a slow development process, at least in my experience. However, as I got better and achieved a better understanding of ROS and its parts, the development process became easier.

5.2 Potential improvements

There are a lot of improvements to be made to this project. The next step would be to implement the rest of the features from the problem description, for example fixing the AR tags recognition and saving the discovered tag positions to the parameters. Furthermore, it would not be difficult to add some more

coordination between the robots, when they discover the "big fires". A quick solution to this could be: Robot1 discovers a big fire → sends the position to the other robot → Robot2 saves it temporarily position and goes to Robot1 → after they met, Robot2 goes back to its saved temporarily position → both robots continue with the map exploration.

6 Links to YouTube videos

In this section you will find links to some YouTube videos showing some executions of the project.

- Showing RViz only, running the first world map. URL: `https://youtu.be/uIAQSVzaBh0`
  - Good for seeing the "steps"/positions, thanks to the markers in RViz
- Showing Gazebo only, running the first world map, URL: `https://youtu.be/pV4iH-cnYSA`
  - Does not grant too much informations, but it might be nice to see them just running in the simulator with physics
- Showing **both** RViz and Gazebo, while running the third map. URL: `https://youtu.be/maZjdJwa43w`
  - This is definitely the best video example, as it shows both RViz and Gazebo. Furthermore, it shows the robots succeeding in exploring a difficult map, including eventually going back to the beginning to investigate some left out unexplored regions.

References

[1] *rospy/Overview/Time*. `http://wiki.ros.org/rospy/Overview/Time`. Accessed: 2021-11-19.

[2] *rviz/DisplayTypes/Marker*. `http://wiki.ros.org/rviz/DisplayTypes/Marker`. Accessed: 2021-11-18.