Final Project

Introductory Robot Programming

December 15, 2021

Students:
Aditya Varadaraj (UID 117054859)

Saurabh Palande (UID 118133959)

Akhilrajan Vethirajan (UID 117431773)

Instructors:
Z. Kootbally

Group:
8

Semester:
Fall 2021

Course code:
ENPM809Y

Contents

Figure 1: (a) Packbot for collapsed buildings [2] ,(b) Aerial robot and (c) USV for shipwreck rescue [3]

1 Introduction

The problem statement for the 809Y Final Project is inspired by a Urban Search & Rescue application. Robots capable of travelling through rough terrains and extreme conditions are used in such applications. After a disaster occurs, a robot is sent to explore the unknown environment (For instance, a partially collapsed building (ground robots (Fig. 1(a))), a shipwreck (aerial + aquatic) (Fig. 1(b), 1(c)) or a flood-trapped area (aquatic + ground or amphibious robots)) and locates trapped or unconcious human victims of the disaster. The robot builds a map of the collapsed building as it explores, and places markers in the map of where the victims are located. This map is then given to trained First Responders who use it to go into the building and rescue the victims.

- The map was already provided in maps folder of the catkin package
- Instead of simulated victims, aruco markers (squared fiducial markers) are used.
- We use a turtlebot (called ***explorer***) to use the map of the building/arena to find the Aruco markers (victims). We use another turtlebot (called ***follower***) to fetch the victims (go to these locations and come back home).
- The problem statement is to create a ROS Program in C++ which can:
  1. Find Victims: Task the ***explorer*** to go through four different locations. At each location, the robot should rotate until it detects the Aruco Marker and then store the position and marker fiducial ID in the program using a broadcaster-listener framework. Once all 4 marker locations have been visited, robot should go back to its start position.
  2. Rescue Victims: Task the ***follower*** to visit each Aruco marker (based on location found by ***explorer***) in increasing order of Aruco Fiducial ID (First visit the marker with ID 0, then ID 1, and so on). Once all 4 marker locations have been visited, the ***follower*** should go back to its start position.
  3. Program should be exited using $ros :: shutdown()$

2 Approach

2.1 Explorer task

- When the executable is launched it initializes the Move Base server and client for both the explorer and the follower. A publisher to publish the angular velocity to the cmd_vel topic and a subscriber to subscribe to fudicial_transforms topic are initialized. The subscriber constantly subscribes to the afore mentioned topic. Any new message in the subscribed topic will trigger the callback function but the callback function discards these messages. Only when the explorer reaches the target and starts scanning for the Aruco marker the fiducial messages are processed. This functionality is achieved by a flag named **m_broadcast_to_call** which is set to false by default.

- A Ros node handle is created and is used to obtain the target locations from the ROS param server. The four targets that the explorer has to visit are stored in a 2D array. The first target location is sent to the explorer's Move Base client. The Move Base client internally computes the trajectory for the explorer to follow and steers the robot to the target. Once the explorer reaches the target location the Move Base client sets the succeeded flag to true.

- When the succeeded flag is true i.e., when the robot has reached the target, the flag **m_broadcast_to_call** is set to true and the rotate function is called. The rotate function consists of a variable of type geometry_msgs/Twist used to set linear and angular velocity of the robot. The angular velocity is set to 0.4 and the linear velocity is set to 0 so that the explorer spins in place. This geometry_msgs is published using the publisher.

- Now that the **m_broadcast_to_call** flag is set, the subscriber can process the messages it receives. When the Aruco marker comes into the camera's field of view the fiducial_transforms have transformation and translation values. The callback method calls the TF Broadcaster.

- The TF Broadcaster assigns a TF frame to the Aruco marker with reference to the camera frame. Now, the **m_broadcast_to_call** flag is reset, TF Listener and the function to stop the explorer spin (**m_stop**) are called.

- The TF listener is a TF package that transforms a TF frame from one frame of reference to another TF frame of reference. Here we call the TF listener to change the Aruco marker frame assigned by the Broadcaster from explorer camera's frame of reference to the map frame. In other words, the detected Aruco marker's position in the map coordinate frame is obtained by the TF Listener.

- The transformed frame is stored in an array according to its fiducial id. The next target location is retrieved from the target locations array and the above steps are repeated for the remaining target locations.

- When the explorer completes the intended functionality at the final target location, the coordinates of the home position are given as the new target. The **m_explorer_done** flag is set to true so that the follower can start execution after explorer reaches home position.

- The flowchart for the explorer and UML class diagram for the explorer class are as shown in Fig. 2. The Fig. 3 shows the fiducial ID and position of each marker being printed on the terminal when the explorer detects each marker.

In Fig. 4, we can see that the explorer reaches all 4 locations, detects the marker (as can be seen in image from turtlebot's camera) and prints corresponding ID and Position in map frame on the terminal.

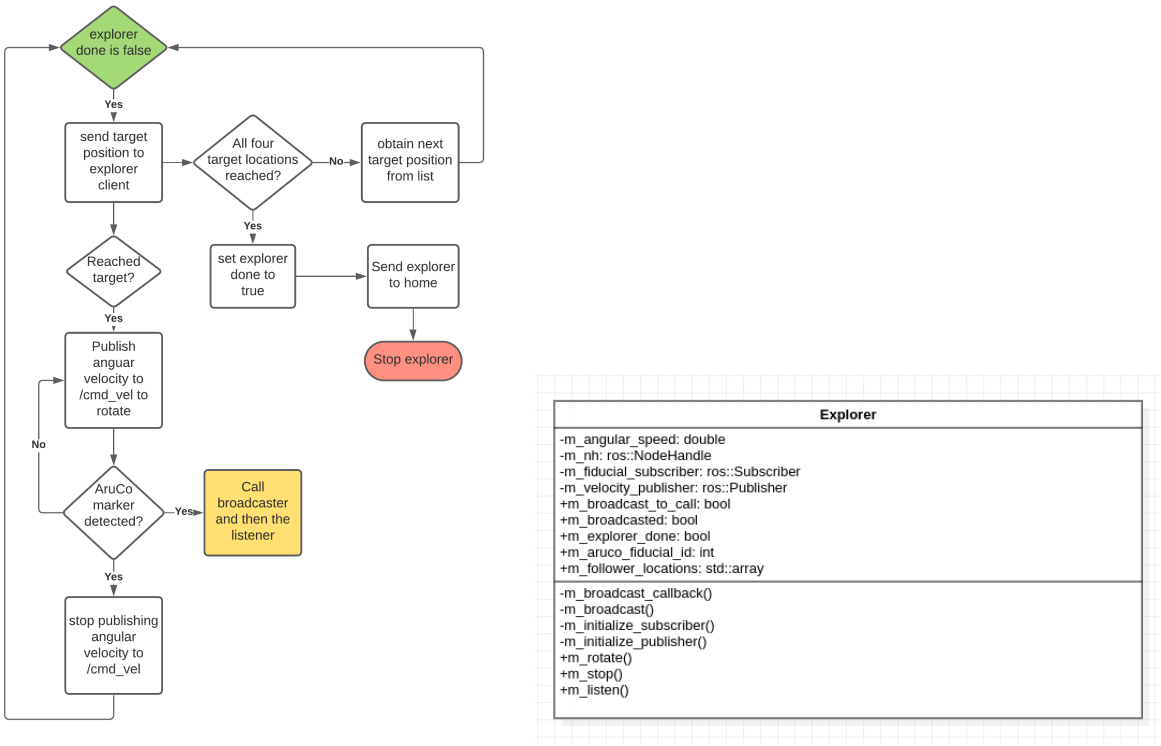

Figure 2: Flowchart for (a)Explorer Logic and (b)Explorer Class UML Diagram

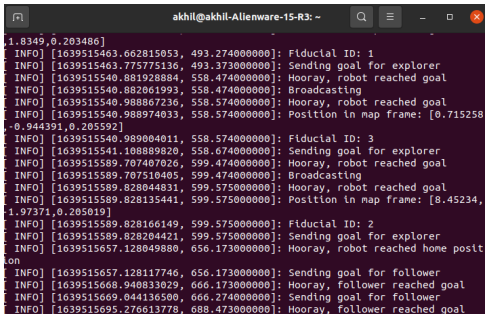

Figure 3: Detection of Aruco Marker

2.2 Aruco detection, Broadcaster-Listener framework and position data storage

The challenge of detecting Aruco marker and getting its location in the map frame and storing it so that the follower can reach them can be divided into 4 steps which are as follows:

- Rotating the explorer at the goal locations to detect the aruco marker.
- Creating and publishing a new frame called "***marker_frame***" on the topic **/tf** and broadcasting this frame once the marker is detected using a Broadcaster.
- Listening to the topic **/tf** after broadcasting and converting the Aruco marker's pose in the **/map** frame and storing this position in an array.

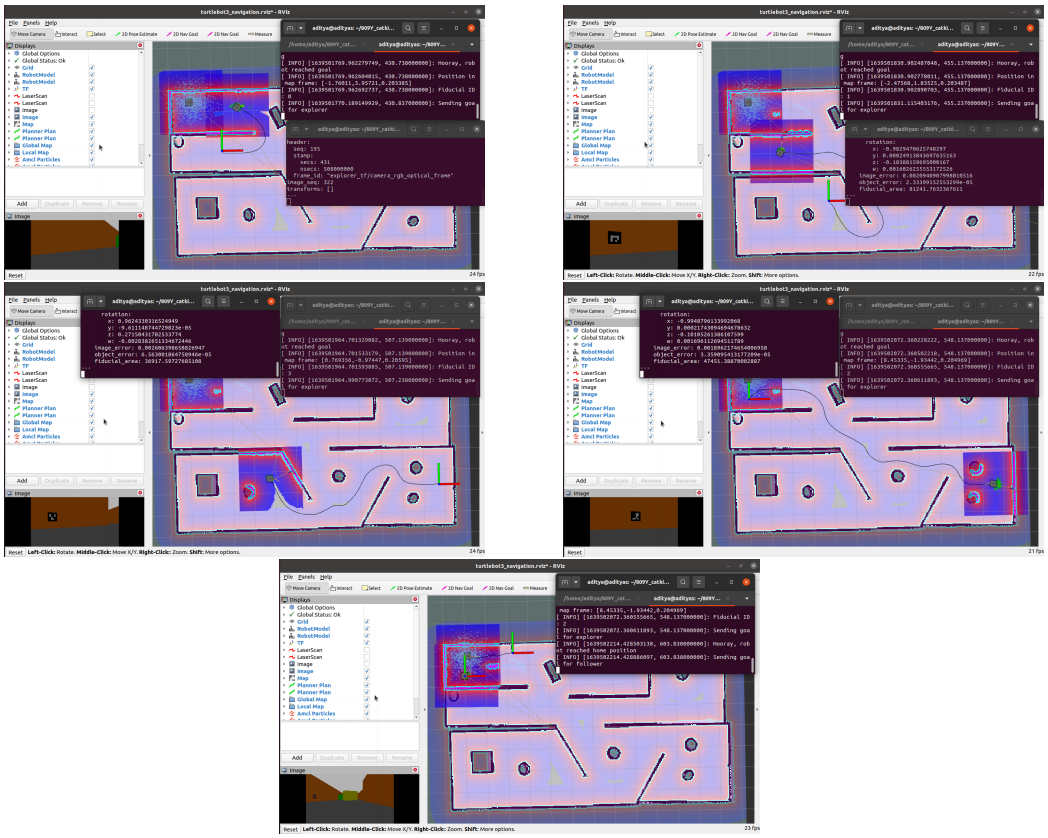

Figure 4: Explorer reached, reading Aruco marker and broadcasting at (a) location 1 (fiducial ID 0) ,(b) location 2 (fiducial ID 1), (c) location 3 (fiducial ID 3), (d) location 4 (fiducial ID 2) and (e) back home

- Stopping the explorer rotation by publishing zero angular velocity.

The steps are explained in detail below:

- **Step 1:** Once the explorer reaches the target goal location as mentioned in the parameter server, we need to rotate the explorer to detect the Aruco marker. This is done by publishing an **angular velocity** of **0.4** in z direction by the publisher **m_velocity_publisher** on the topic explorer/cmd_vel. This task is accomplished by the **m_rotate()** function. After the explorer starts rotating, set the flag **m_broadcast_to_call** as **true**.
- **Step 2:** After the explorer starts rotating at the target location to detect the aruco marker, there are three conditions which **need to be true** before the broadcaster broadcasts the transform to the new frame.
  - Condition 1: **m_broadcast_to_call** which is set to true once the explorer starts rotating.
  - Condition 2: The marker should be detected which implies that the **transforms array should not be empty**.
  - Condition 3: Since there might be a case where the explorer camera detects far away aruco marker rather than the marker at the target location, we placed a condition on fiducial

area. The **fiducial area must be greater than 750** so that only the marker at the target location is broadcasted.

Only after all these conditions are true, we call the function **m_broadcast()** which creates a new frame called "**marker_frame**" and publishes it on the topic /tf. Once this is broadcasted, set the **m_broadcasted** to true.

- **Step 3:** To get the pose of the Aruco marker in the map frame we use the tf/listener which transforms the pose of aruco marker in the "**marker_frame**" as broadcasted by the broadcaster to the map frame so that the follower can reach the aruco markers. To call the **m_listener()**, the value of **m_broadcasted** needs to be true which is set to true after broadcasting the frame in the **m_broadcast()** function. We store these transformed poses in an **std::array** → **m_follower_locations** based on the fiducial id.
- **Step 4:** After the explorer detects the aruco marker and the broadcaster and listener task is complete we need to stop the rotation of the explorer so that it goes to the next target location. This is accomplished by publishing zero angular velocity in z direction to the topic explorer/cmd_vel.

Algorithm 1 Aruco detection and Position data storage

if *explorer reached goal location* **then**
  start rotating *the explorer to detect the aruco marker*
  m_broadcast_to_call ← **true**
  if m_broadcast_to_call *is* **true** *and* **transforms is not empty** *and* **fiducial_area** *is* **greater than 750 then**
    call the **broadcaster** *to* **broadcast**
    m_broadcasted ← **true**
    call the **listener** *to* **lookup** *for the transform*
    stop the rotation *of the explorer*
  end
end

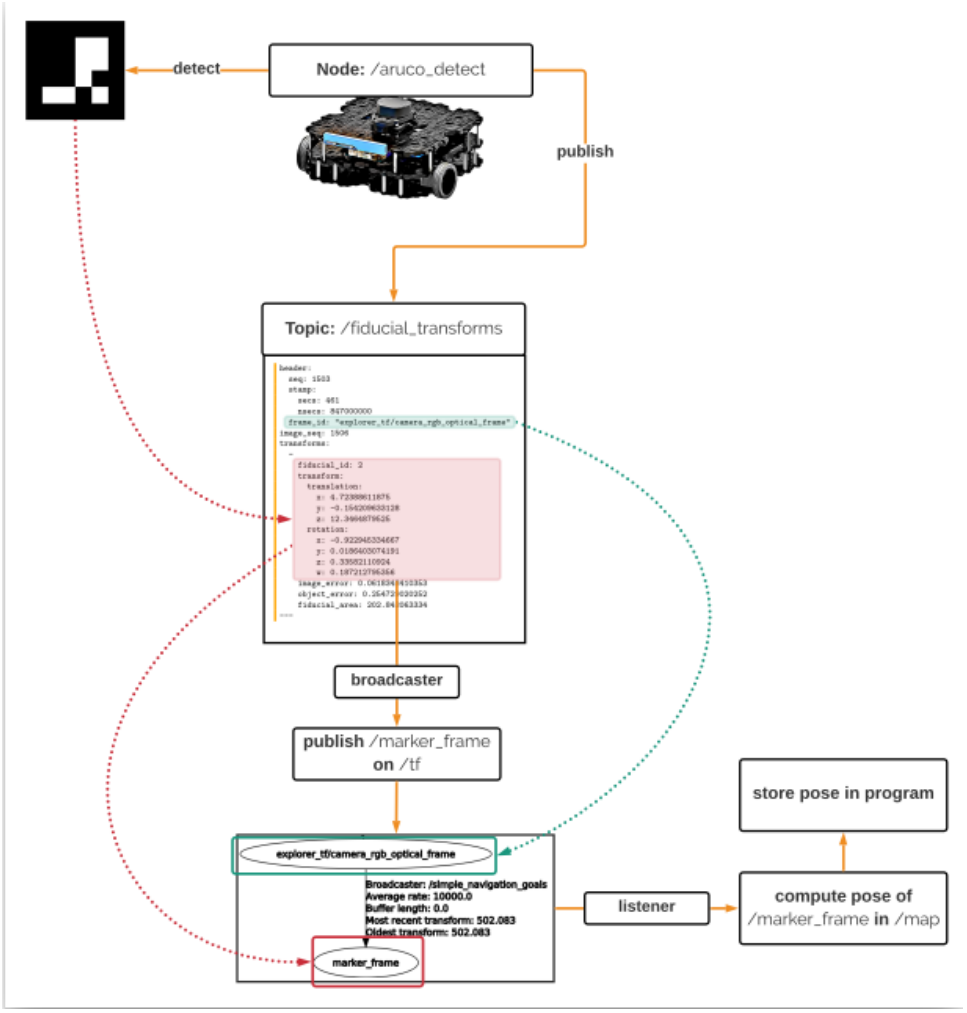

Figure 5: Broadcaster and Listener [1]

2.3 Follower task

The follower needs to go to the various marker locations using the position detected, broadcasted and stored by the explorer as explained in previous subsections. Also, a tolerance of 0.4mts. needs to be given. This is done as follows:

- The Listener **m_listen(tfBuffer)** looks up the transform broadcasted by broadcaster. This gives us position in x,y,z and orientation in quaternions.
- From this, we create a tf::Quaternion object with the extracted quaternion values. Then, we use tf::Matrix3x3 to convert this into transformation matrix. And then, we use the getRPY() method of tf::Matrix3x3 object to convert quaternions to Roll, Pitch and Yaw.
- The yaw is the robot's orientation about z axis in radians. So this is the angle that the frame attached to the marker and hence the wall makes w.r.t. global frame. So using $\theta = -\frac{\pi}{2} + yaw$, should give us the angle that the line perpendicular to the wall makes w.r.t. global frame.
- So using $x_{marker} + 0.4 * cos(\theta)$ and $y_{marker} + 0.4 * sin(\theta)$ should give us the desired goal x and y positions of the follower which is stored based on fiducial ID in a 2D array.

- Then, the location of 1st marker (fiducial ID 0) is sent to the MoveBase Client. When follower reaches the goal, MoveBase sets a boolean "succeeded" to true. Then, a counter keeping track of fiducial ID is incremented so that location of next marker. This continues until all 4 marker locations are reached.
- After that, the location of home position of follower is passed to the MoveBase client. Once the robot reaches the home position, ROS program is exited using **ros::shutdown()**

Algorithm 2 Follower Pseudocode

In Explorer's Listener:
Extract quaternion from the transform and convert to roll-pitch-yaw angles
$\theta \leftarrow -\frac{\pi}{2} + yaw$
$x_{follower\_goal} \leftarrow x_{marker} + 0.4 * cos(\theta),\ where,\ 0.4\ is\ tolerance\ in\ mts.$
$y_{follower\_goal} \leftarrow y_{marker} + 0.4 * sin(\theta)$
m_follower_locations[**m_aruco_fiducial_id**][0] $\leftarrow x_{follower\_goal}$
m_follower_locations[**m_aruco_fiducial_id**][1] $\leftarrow y_{follower\_goal}$
In Main Code, initialize $f \leftarrow 0$ before while(ros::ok())
In oop_main.cpp (Main code) inside WHILE(ros::ok())
if explorer is done **then**
  Wait for explorer Result
  if *f is less than* 4 **then**
    Follower Target x $\leftarrow$ **m_follower_locations**[*f*][0]
    Follower Target y $\leftarrow$ **m_follower_locations**[*f*][1]
    if *follower goal is not sent* **then**
      send goal and set follower_goal_sent to true
    end
    if *follower has reached goal successfully* **then**
      increment f so that next location is given as goal
    end
  else
    Follower Target x $\leftarrow$ *home_x*
    Follower Target y $\leftarrow$ *home_y*
    if *follower goal is not sent* **then**
      send goal and set follower_goal_sent to true
    end
    if *follower has reached goal successfully* **then**
      Exit Program using ros :: shutdown()
    end
  end
end

As we can see in Fig. 6, we can see that the follower is able to reach the 4 locations in increasing order of fiducial ID and back home.

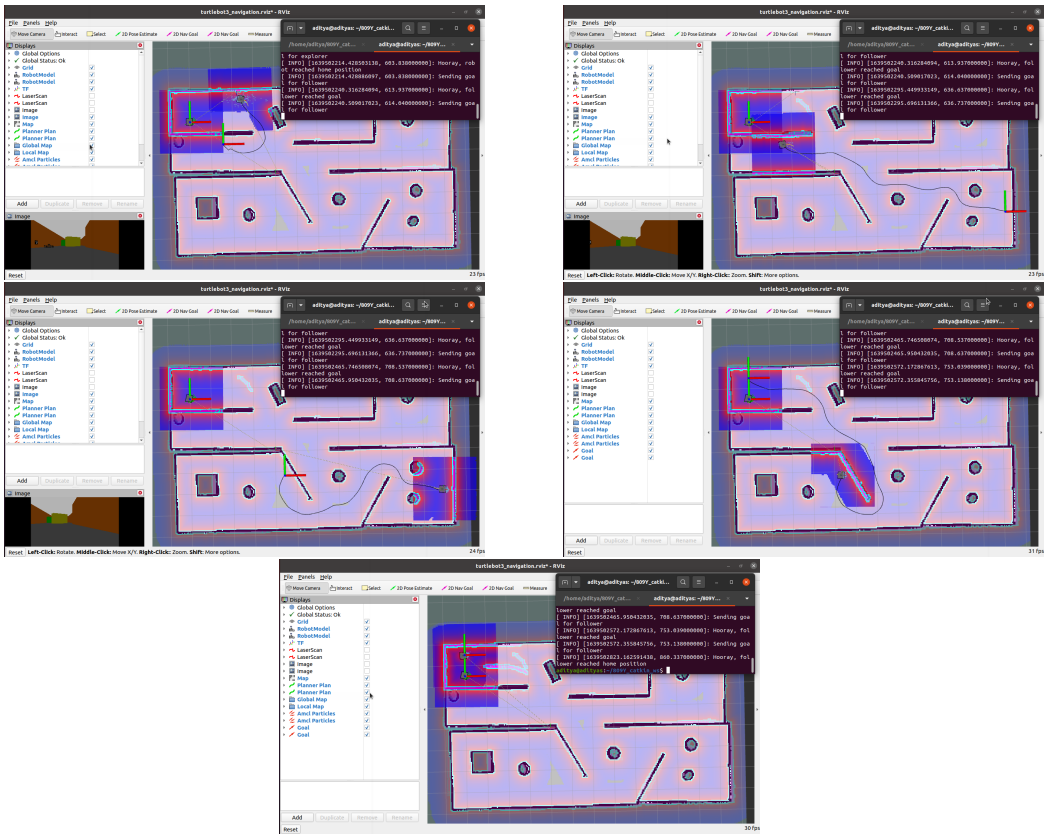

Figure 6: Follower reached (a) location 1 (fiducial ID 0) ,(b) location 2 (fiducial ID 1), (c) location 4 (fiducial ID 2), (d) location 3 (fiducial ID 3) and (e) back home

3 Challenges

- **Challenge 1:**
  - During initial stages of development, the broadcaster was used as the callback method of the subscriber. This caused the broadcaster to constantly broadcast whenever an Aruco marker was detected by the camera. Due to broadcaster failure the Listener did not transform the frames. This shouldn't be the case and a flag was created to control broadcasting operation. Even though the flag was set to true only after the explorer reaches goal it did not function as intended. The problem was the subscriber was reinitialized multiple times as its initialization was part of the iterative loop. This caused the subscriber to drop certain messages leading to unpredictable functioning of broadcaster.
  - This error was rectified by initializing the subscriber once and creating a separate callback method which will call the broadcaster only when the robot reaches target and starts scanning. When broadcasting operation is done, the listener is called to perform frame transformation.
- **Challenge 2:**
  - One of the challenges we faced was that of the Aruco marker getting detected and broadcasted before the explorer reached the goal location.

- We solved this problem by creating a flag **m_broadcast_to_call** and setting the value of this flag to true after the explorer reaches the goal location. Only after this flag becomes true, transforms are not empty and fiducial area is greater than 750, we call the broadcaster.

- **Challenge 3:**
  - Another challenge was that the Aruco marker location was getting stored as 0 0 for the marker with fiducial ID 1. With some guidance from professor, we found out that this problem was caused by marker with fiducial ID 2 being detected by camera while rotating at the second location although it was far away. So while rotating in place at 2^{nd} location, the turtlebot's camera was getting a small peek at the aruco marker located at the 4^{th} location (with ID 2) and was storing the position in ID 2 instead of ID 1.
  - We solved this problem by using an additional condition that fiducial area should be greater than 750 alongwith the rest 2 conditions being satisfied for the broadcaster to be called.

4 Project Contribution

- It was all a team effort overall.
- **Aditya Varadaraj** worked mainly on the follower part of the code, how to apply the tolerance to the marker positions to get follower goals, compiling everyone's portions of the report in a single LaTeX document.
- **Saurabh Palande** worked mainly on getting the broadcaster-listener part to work, UML Diagrams and rewriting code using OOP.
- **Akhilrajan Vethirajan** worked mainly on the explorer part of the code, flowchart for explorer logic and Doxygen Documentation.

5 Resources

- ROS
- Gazebo
- RviZ
- MoveBase
- Visual Studio Code
- `https://github.com/zeidk/final_project`
- `http://wiki.ros.org/roscpp/Overview/Parameter%20Server#CA-a03bfcab2d7595a784e24298b326fdc4c76f3a5`
- `https://gist.github.com/marcoarruda/f931232fe3490b7fa20dbb38da1195ac`
- `http://docs.ros.org/en/kinetic/api/aruco_detect/html/aruco__detect_8cpp_source.html`

6 Course Feedback

- The course was very good and the topics were taught very well by the instructor.
- Even though this is not primarily a ROS course, many ROS concepts were taught in the end of the course which is very helpful for us.
- The projects were all challenging and taught us a lot.
- The instructor could have talked a little more on GitHub and its Source Control features.

References

[1] Z. Kootbally, "ENPM 809Y Final Project: US& R Simplified Instructions", University of Maryland - College Park, December 2021.

[2] M. Norouzi and F. Bruijn and J.V. Miro, "Planning Stable Paths for Urban Search and Rescue Robots", Robocup 2011, Springer, vol. 7416, pp. 90-101, Jun. 2012, doi: 10.1007/978-3-642-32060-6-8.

[3] R. Mendonça et al., "A cooperative multi-robot team for the surveillance of shipwreck survivors at sea", OCEANS 2016 MTS/IEEE Monterey, 2016, pp. 1-6, doi: 10.1109/OCEANS.2016.7761074.