University of Burgundy

Master of Science in Computer Vision – 2nd Year

Robotics Project Module

Technical Report of ROS Implementation

For Mapping and Localization using TurtleBot

Team Members

Gopikrishna Erabati

Dousai Nayee Muddin Khan

Mohit Kumar Ahuja

Under the supervision of

Dr. Ralph Seulin

Dr. Nathan Crombez

Dr. Raphael Duverne

Le2i, Le Creusot

CONTENTS

LIST OF FIGURES

1. Introduction

1.1 ROS

Robot Operating System (**ROS**) is robotics middleware (i.e. collection of software frameworks for robot software development). ROS is considered as meta open source operating system, which provides services designed for heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. Running sets of ROS-based processes are represented in a graph architecture where processing takes place in nodes that may receive, post and multiplex sensor, control, state, planning, actuator and other messages. Despite the importance of reactivity and low latency in robot control, ROS, itself, is *not* a real-time OS (RTOS), though it is possible to integrate ROS with real-time code. The lack of support for real-time systems is being addressed in the creation of ROS 2.0.

Software in the ROS Ecosystem can be separated into three groups:

- Language-and platform-independent tools used for building and distributing ROS-based software;
- ROS client library implementations such as roscpp, rospy, and roslisp;
- Packages containing application-related code which uses one or more ROS client libraries.

Both the language-independent tools and the main client libraries (C++, Python, and Lisp) are released under the terms of the BSD license, and as such is open source software and free for both commercial and research use. The majority of other packages are licensed under a variety of open source licenses. These other packages implement commonly used functionality and applications such as hardware drivers, robot models, datatypes, planning, perception, simultaneous localization and mapping, simulation tools, and other algorithms.

The main ROS client libraries (C++, Python, and Lisp) are geared toward a Unix-like system, primarily because of their dependence on large collections of open-source software dependencies. For these client libraries, Ubuntu Linux is listed as "Supported" while other variants such as Fedora Linux, macOS, and Microsoft Windows are designated "Experimental" and are supported by the community. The native Java ROS client library, rosjava, however, does not share these limitations and has enabled ROS-based software to be written for the Android OS. Rosjava has also enabled ROS to be integrated into an officially-supported MATLAB toolbox which can be used on Linux, Windows and MacOS. A JavaScript client library, roslibjs has also been developed which enables integration of software into a ROS system via any standards-compliant web browser.

1.2 TurtleBot

TurtleBot is a low-cost, personal robot kit with open-source software. TurtleBot was created at Willow Garage by Melonee Wise and Tully Foote in November 2010. Turtlebot is a well-advanced mobile robot which can be used in our daily life to move around the surroundings, Map them into 3D with very effective amount of battery life.

The TurtleBot kit consists of a mobile base basically called as Kobuki, 2D/3D distance sensor, laptop computer or SBC (Single Board Computer), and the TurtleBot mounting hardware kit. In addition to the TurtleBot kit, users can download the TurtleBot SDK from the ROS wiki. TurtleBot is designed to be easy to buy, build, and assemble, using off the shelf consumer products and parts that easily can be created from standard materials. As an entry-level mobile robotics platform, TurtleBot has many of the same capabilities of the company's larger robotics platforms, like PR2.

The TurtleBot we used in our project is "TurtleBot-2" with a kobuki base, Kinect-2 depth camera and an Asus think pad. The pictorial look of TurtleBot 2 is shown in below figure.

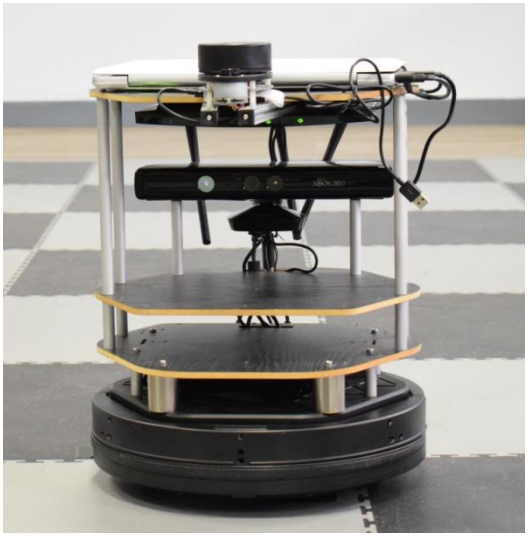

Figure 1. Turtlebot 2

1.3 Kinect Sensor

Kinect (codenamed Project Natal during development) was a line of motion sensing input devices by Microsoft for Xbox360 and Xbox One video game consoles and Microsoft Windows PCs. Based on a webcam-style add-on peripheral, it enables users to control and interact with their console/computer without the need for a game controller, through a natural user interface using gestures and spoken commands. The application can use Kinect's tracking functionality and Kinect sensor's motorized pivot to keep users in the frame even as they move around.

Using Kinect v1 you can track as many as six people and 25 skeletal joints per person—including new joints for hand tips, thumbs, and shoulder center—and improved understanding of the soft connective tissue and body positioning, you get more anatomically correct positions for crisp interactions, more accurate avateering, and avatars that are more lifelike. The enhanced fidelity of the depth camera, combined with improvements in the software, has led to a number of body tracking developments.

We used Kinect first to create the map of both the rooms and then Kinect is used to sense and locate the robot autonomously in both the rooms by SLAM algorithm.

Figure 2. Kinect v1

1.4 Task

The motto of the project is to gain experience in the implementation of different robotic algorithms using ROS framework.

The hardware we are using is a Turtle Bot 2 with a Kobuki base as our robotic hardware platform with Kinect 1 which is an RGB-D sensor. We are using ROS as a software framework.

1. **The first step of task is to build a map of the environment and navigate to the desired location on the map.**

2. Next, **we have to sense the location of the marker (e.g. AR marker, colour markers etc) in the map, where there is pick and place task, and autonomously localise and navigate to the desired marker location**.

3. After reaching to the desired marker location, we have to precisely move towards the specified location based on visual servoing.

4. At this location, we have a robotic arm which picks an object (e.g a small cube) and places on our turtlebot (called as pick and place task).

5. **After, the pick and place task, again the robot needs to find another marker, which specifies the final target location, and autonomously localise and navigate to the desired marker location, which finishes the complete task of the project.**

   From the above mentioned points, Point 1,2 and 5 are the focused tasks in this report. A simple task diagram for the this points is labelled in the below figure.

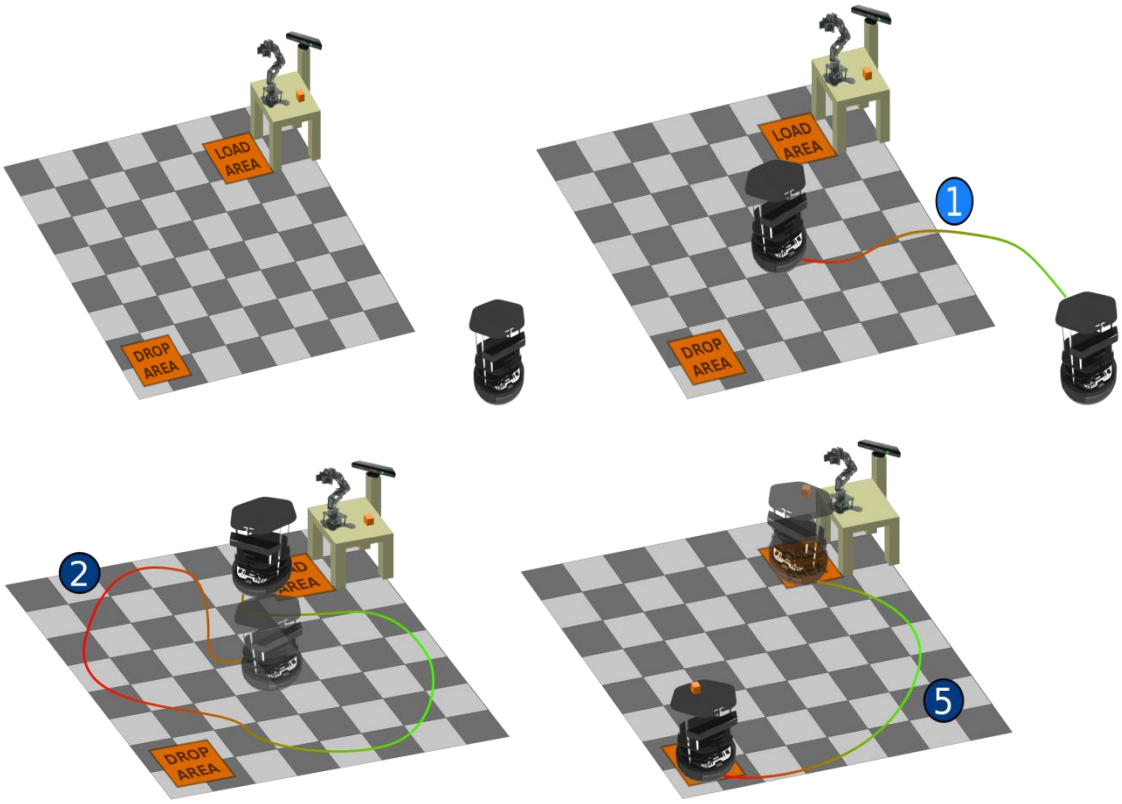

Figure 3. Tasks

2. Packages in Simulation and Real Time

For the development of our project, we have to choose and create our own environment in simulation by creating our world surroundings by using our preferred markers in the gazebo and also mapping the real-time environment by using Turtlebot with the help of some packages. For the robot localization in the environment, we have chosen ar tags (Augmented Reality) both in simulation and real-time. We will explain in the below sections why we have chosen this tags over the other ones.

2.1 Simulation

To build the simulation environment we will need the below prerequisite installed before on our machine:

- Turtlebot_Gazebo
- Ar Tags
- Turtlebot_navigation

Turtlebot_Gazebo is basically an indoor simulation where we can create our own world with respect to the pre available objects. This package comes with the gazebo world environment and also turtlebot. We can install this package on the below line in Ubuntu terminal:

Sudo apt-get install ros-indigo-turtlebot-gazebo

To launch the empty gazebo environment with just turtlebot we will use:

Roslaunch turtlebot_gazebo turtlebot_empty_world.launch

To build our own simulation world we will need few objects like walls and tables from gazebo but we will also need to arrange our ar tags in this environment as we are going to test on ar tags. So to get ar tags at first we have to install ar tags package for our ros machine. We will use the below line to install *ar_track_alvar* package

sudo apt-get install ros-indigo-ar-track-alvar

To create the Markers in gazebo we will copy the markers into the gazebo model directory as:

/generate_markers_model.py -i IMAGE_DIRECTORY -g GAZEBO_MODELS_DIRECTORY -s SIZE_IN_MILLIMETER

Where the parameters are described as :

-i or --input: directory where the marker images are stored (absolute path)

-g or --gazebodir: directory of the gazebo models (usually in ~/.gazebo/models)

-s or --size: size of the marker in millimeter

For our project, we are choosing ar tags because ar tags can easily be detected from long range distance even if they are in small dimensional size. Very easy to detect by using RGB-D cameras like Kinect. The differentiate of markers can easily be detected by their size and shape.

Once we build our own gazebo world with ar tags, we can save this world as .sdf file from the File menu and Save World As. To open our gazebo environment we will use:

gazebo labenvironment.sdf

As we have our gazebo built environment, we can use our own world to build the map and then we will proceed to check ar tags detection on the built map.

The steps to build the map in thesimulation are as follows:

- To open our built gazebo environment

roslaunch turtlebot_gazebo turtlebot_world.laumch world_file:= /path/of/file/labenvironment

- For mapping of the environment, we will use gmapping

roslaunch turtlebot_gazebo gmapping_demo.launch

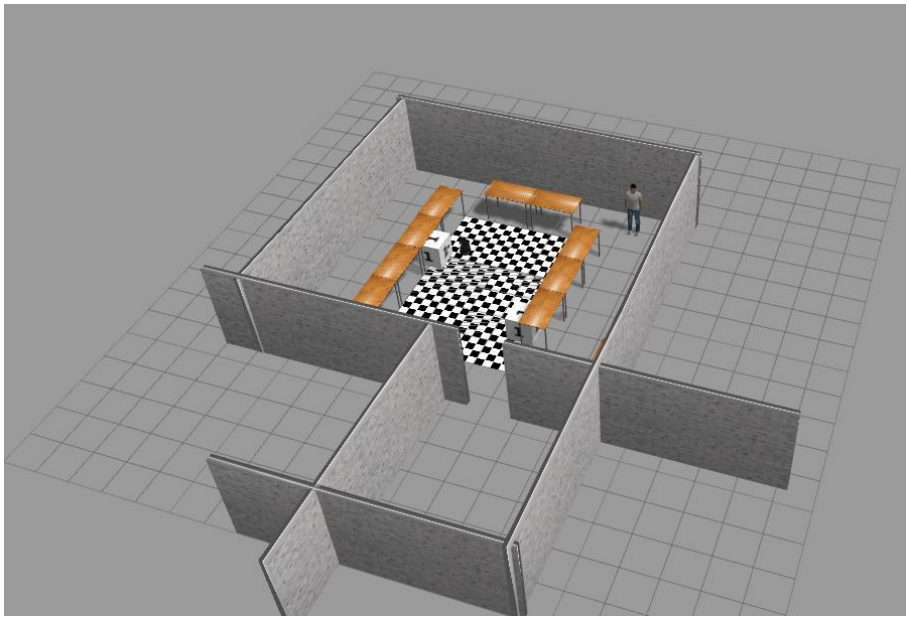

Figure 4. Gazebo world of our built Environment

- We will use keyboard teleop to move around the turtlebot in gazebo

  roslaunch turtlebot_teleop keyboard_teleop.launch

- Now we will move around the turtlebot with keyboard controls and build the map,we can even use rviz to know the precision of the mapping

  roslaunch turtlebot_rviz_launchers view_robot.launch

- As we move the turtlebot around the built environment, now we can save the map which will generate the .png and .yaml files

  rosrun map_saver map_saver –f ~/path/to/save/labenvironment_gazebo

- Now we can use .yaml file for navigating the turtlebot by using amcl_demo launch

  roslaunch turtlebot_gazebo amcl_demo.launch map_file:= /path/of/file/labenvironment_gazebo.yaml

- At the end we will launch ar_tracker.launch with our own created nodes for task 1 (Pickup Point) and task2 (Drop Point).

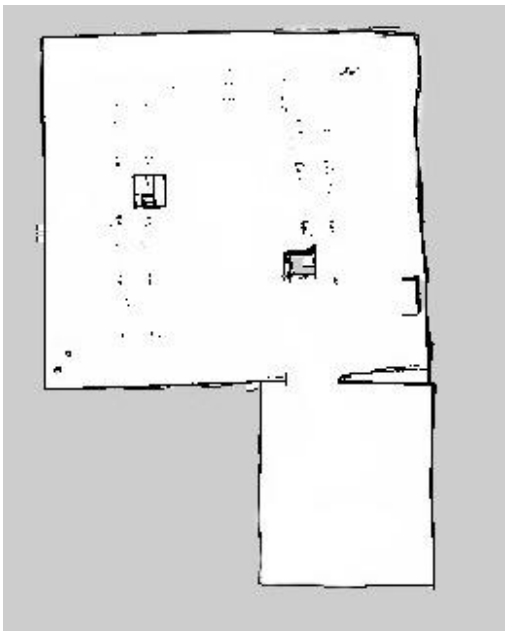

Figure 5 .Simulation map of world in Gazebo

2.2 Real-time

For the real implementation of the turtlebot, we will connect the turtlebot with our workstation and then we will run our packages from workstation to map the real environment. To connect turtlebot from the workstation we will use:

sshturtlebot@ipaddress

As we connect to the turtlebot by using workstation password we are ready to launch our own packages on both turtlebot space and also on the workstation. One of the important packages which we will talk here new than in simulation is *turtlebot_bringupminimal.launch*. This package is launched on turtlebot and then as the same procedure for simulation, we will launch gmapping and teleop operations to map the real environment and save the map as the same steps described above but on turtlebot. The real-time map should be always saved on the turtlebot.Once we have the map saved on turtlebot we can launch the minimal launch and turtlebot_navigation with the created real-time map for amcl_demo on turtlebot and the rest of the packages like ar_tags and our own created nodes on workstation.

- We will use the following launch file to run minimal launch

roslaunch turtlebot_bringup minimal.launch

- Now we will launch amcl demo with the saved map as shown in figure 6

roslaunch turtlebot_navigation amcl_demo.launch
map_file:=/path/of/file/labenvironment_real.yaml

- As the last launch file we will use ar tags and our nodes on workstation

Figure 6.Real time map environment

3. Strategy for the task

The strategy we followed to achieve the task specified is given as a flowchart, as in fig.7.

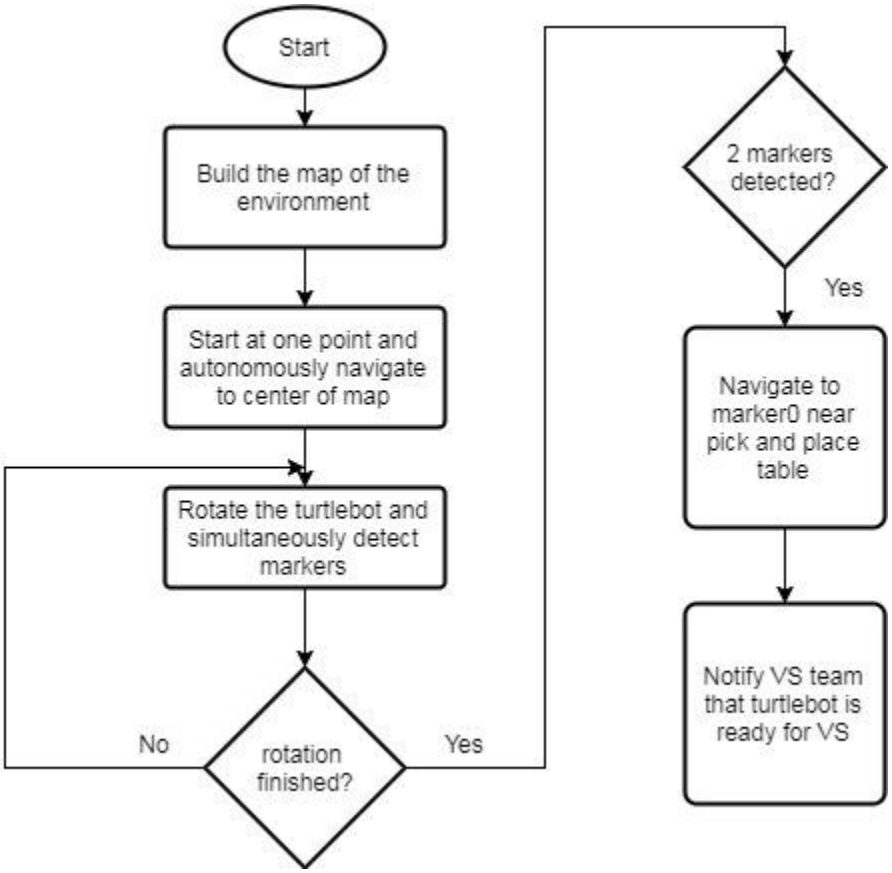

Figure 7. Flowchart of strategy used to achieve the task

In order to achieve the above strategy we developed nodes to do the different tasks and communication between nodes is achieved by flags.

Firstly we used 'gmapping' package to map and 'turtlebot_teleop' package to navigate the turtlebot, to build the map of environment and the map is used by the 'amcl' package for autonomous navigation of turtlebot in the map.

3.1 Nodes developed

3.1.1 goToPoint

The 'goToPoint' node which is a python script file, publishes the intial pose of turtlebot on the 'initialpose' topic which is of type 'geometry_msgs/PoseWithCovarianceStamped' and it also publishes the centre position of the map to 'move_base/goal' topic which is of type 'move_base_msgs/MoveBaseActionGoal'. After the turtlebot reaches the centre of the map, it publishes a flag on 'centerReached_flag' topic which is of type 'std_msgs/Bool'. The rqt graph of the node looks as in fig. 8.

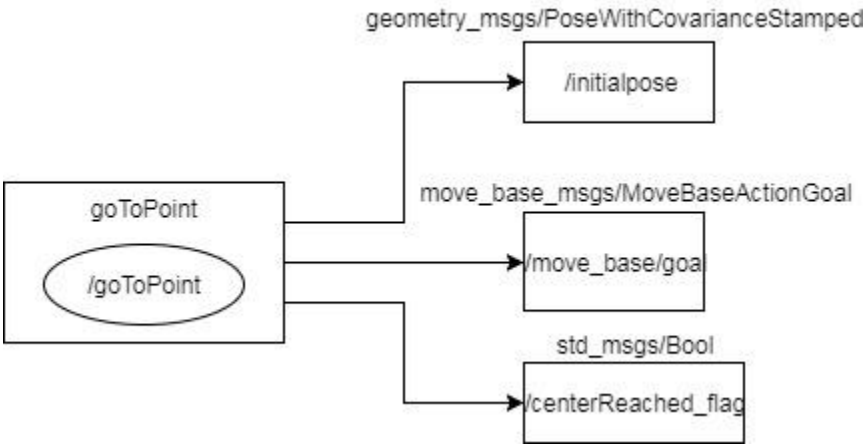

Figure. 8 rqt graph of 'goToPoint' node.

3.1.2 Localization

The localization node is used to get the localized pose of two markers in the map. The task for detecting two markers and calculating the pose of markers with respect to map and publishing flag on 'goToOn_flag' topic is as described in flowchart as in fig.9.

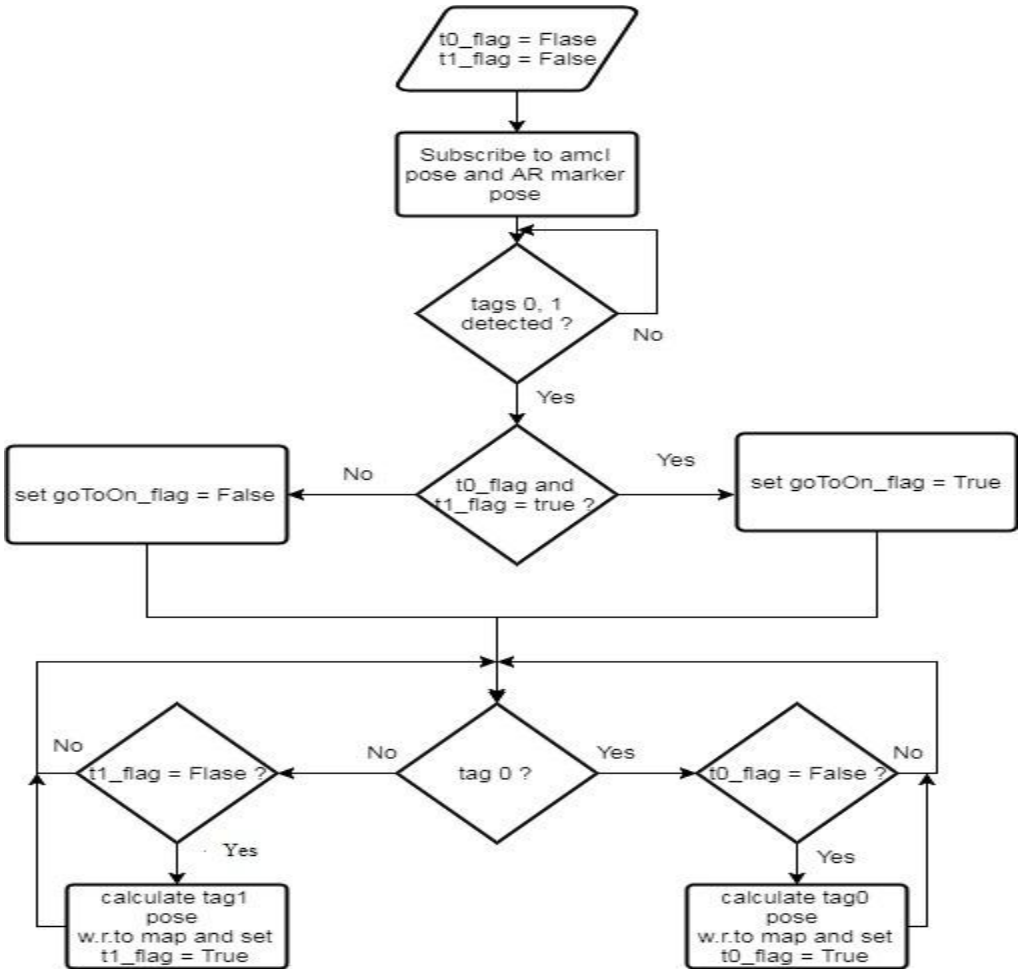

Fig.ure 9 Flowchart for localization node

This node subscribes to 'amcl_pose' topic and 'ar_marker_pose' topic to get the pose of turtlebot with respect to map and pose of marker with respect to base link of turtlebot respectively. When the turtlebot detects markers 0 and 1 as shown in figure 10, the 't0_flag' and 't1_flag' is checked which is initially set to 'False'. If both flags are true a flag is published on 'goToOn_flag' topic of type 'std_msgs/Bool' which tells that two markers are detected.

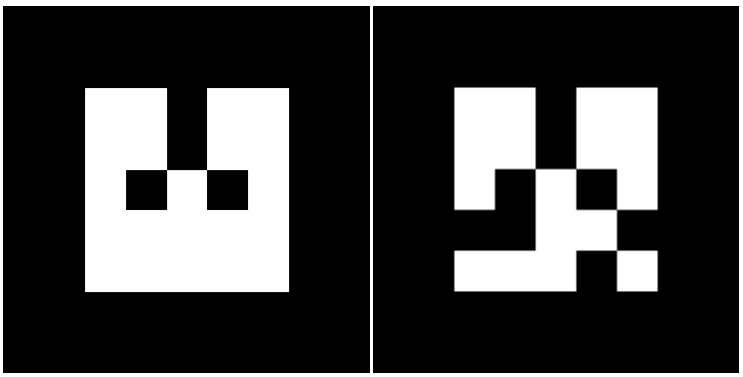

Figure 10 . AR Markers, Left: Marker 0, Right: Marker 1

When a marker is detected, the pose of turtlebot with respect to map is subscribed and the quaternion angles of rotation of frames of map and turtlebot is converted to rotation angle 'yaw' of turtlebot with respect to map according to formula,

$$\psi = \tan^{-1}\left(\frac{2(q_0 q_3 + q_1 q_2)}{1 - 2(q_2^2 + q_3^2)}\right)$$

where quaternion $q = q_0 + iq_1 + j\,q_2 + k\,q_3$ and ψ is the yaw angle of turtlebot.

If the pose of turtlebot is $(x_T^M, y_T^M, z_T^M, \psi)$and the marker is $(x_{Ma}^T, y_{Ma}^T, z_{Ma}^T)$ then the pose of marker with respect to map is given as,

$$\begin{bmatrix} x_{Ma}^M \\ y_{Ma}^M \\ z_{Ma}^M \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\psi & -\sin\psi & 0 & x_T^M \\ \sin\psi & \cos\psi & 0 & y_T^M \\ 0 & 0 & 0 & z_T^M \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{Ma}^T \\ y_{Ma}^T \\ z_{Ma}^T \\ 1 \end{bmatrix}$$

by the above affine transformations we get the pose of marker with respect to map and then 'to_flag' or/and 't1_flag' are set to 'True' and then the poses of two markers is published on 'target_pose0' and 'target_pose1' topics of type 'geometry_msgs/PoseStamped'.

The rqt graph of this node looks as in fig. 11.

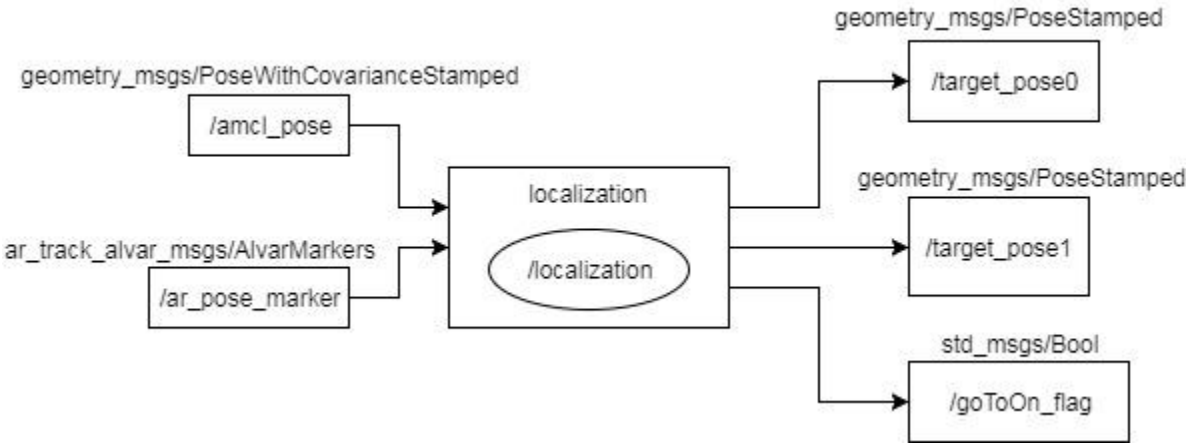

Figure 11. rqt graph of 'localization' node.

3.1.3 rotate

This node is used to rotate the turtlebot around its center of axis at center position of map for detecting the markers. When the 'centerReached_flag' is 'True', the turtlebot starts rotating, it rotates 360 degrees and then it publishes a flag on 'rotate_flag' topic of type 'std_msgs/Bool'.

The rqt graph of this node looks as in fig. 12.

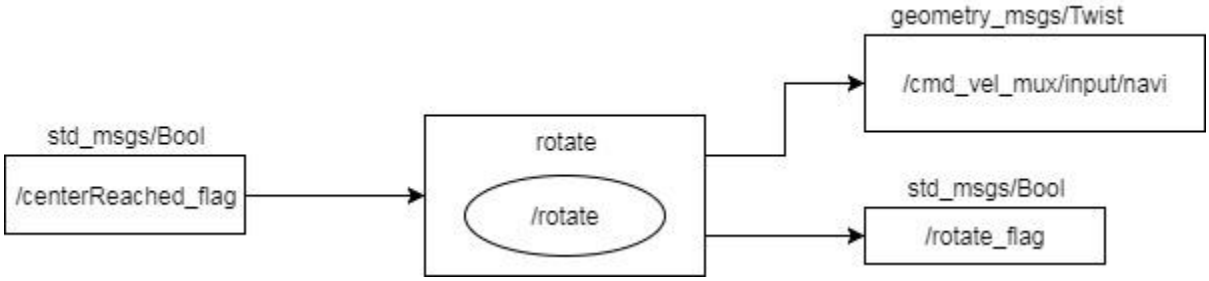

Figure 12. rqt graph of 'rotate' node.

3.1.4 goToTarget

When the two markers are detected, 'goToOn_flag' is set to 'True' and after 'roate_flag' is set to 'True' after rotation of turtlebot, the turtlebot moves to the pose computed for marker 0, which is the location of pickup of the object.

The rqt graph of this node looks like in fig. 13.

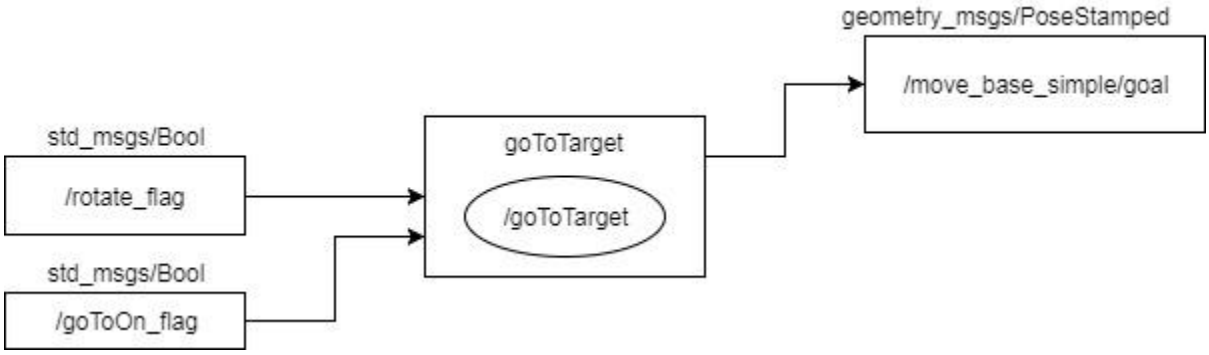

Figure 13. rqt graph of 'goToTarget' node.

3.1.5 goalStatus

After the turtlebot reaches the pickup location, we should notify the visual servoing team to precisely place the robot near the arm. This task of handshake with the visual servoing team is done by this node. It publishes on topic 'nav_status' of type 'std_msgs/String'. it subscribes to amcl pose and goal pose to check whether the turtlebot reached the goal or not.

The rqt graph of this node looks like in fig. 14.

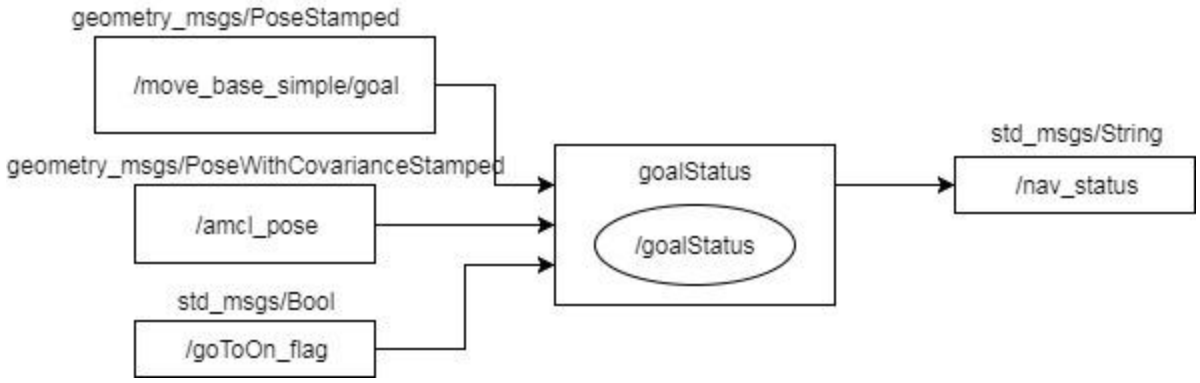

Figure 14. rqt graph of 'goalStatus' node.

3.1.6 goToTarget1

After the visual servoing team has done their job and robotic arm team picked block and placed on turtlebot, they handshake a flag on topic 'arm_status' of type 'std_msgs/String'. This node subscribes to this topic and it publishes goal of marker 1 pose on 'move_base_simple/goal'.

The rqt graph of this node is as shown in fig. 15.

Figure 15. rqt graph of 'goToTarget1' node.

3.1.7 goalStatus1

After the turtlebot reaches the location to place the block, we should notify the visual servoing team to precisely place the robot near the arm. This task of handshake with the visual servoing team is done by this node. It publishes on topic 'nav_status' of type 'std_msgs/String'. it subscribes to 'target1_published' flag, amcl pose and goal pose to check whether the turtlebot reached the goal or not.

3.2 Communication between Nodes

The communication between nodes is achieved by using different flags. This process is is shown as a diagram as seen in fig. 16.

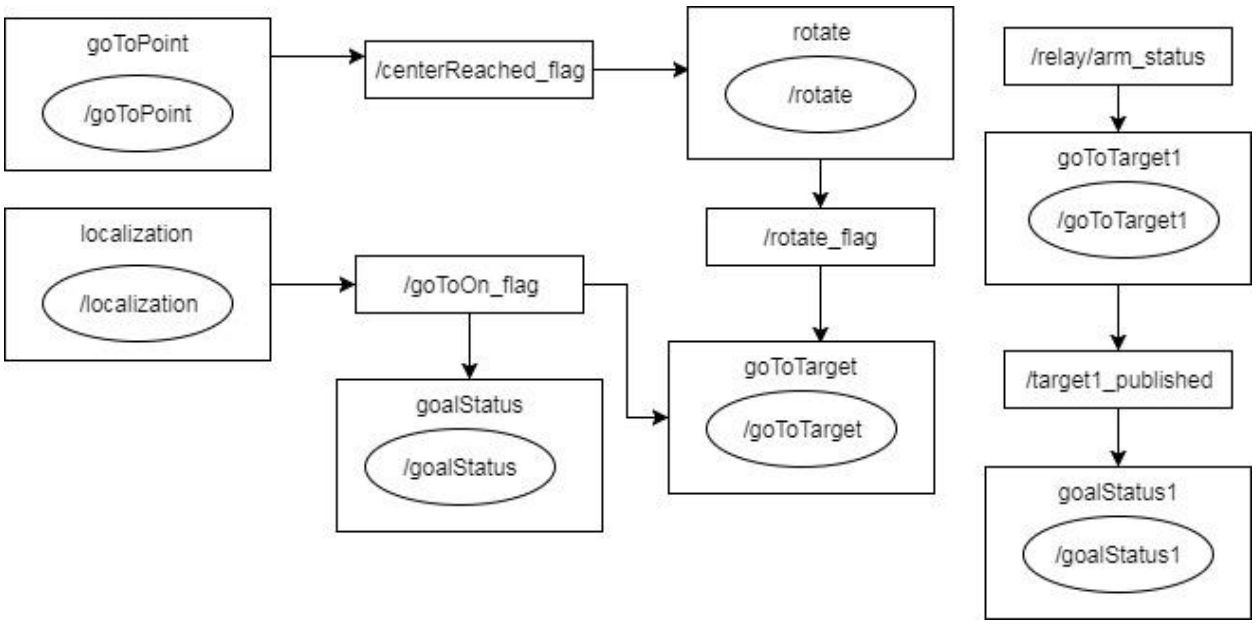

Figure 16. Flags with communication between nodes

3.3 Launch Files

1. For the real-time using turtlebot:

    We have developed two launch files:

    i) To launch minimal launch of turtlebot in 'turtlebot_bringup' package and launch 'amcl_demo' of 'turtlebot_navigation' package with the map file we created at first. (This launch file is launched on turtlebot)

    ii) To launch AR tracker launch file and all the nodes we developed at once. (This launch file is launched on workstation)

2. For the simulation in Gazebo using turtlebot:

    We have developed two launch files:

    i) To launch 'turtlebot_world' in 'turtlebot_gazebo' package using the world we created in Gazebo and launch 'amcl_demo' of 'turtlebot_navigation' package with the map file we created at first.

    ii) To launch AR tracker launch file and all the nodes we developed at once.

4. Results

Github link for code **:** CodeHere

4.1 Simulation in Gazebo

We have made our full simulations in Gazebo, before proceeding with real-world scenarios. When we are fully satisfied with our nodes and communication between them, we proceed to the real world.

The link for video of our work in Gazebo is here.

4.2 Real time with Turtlebot

After successful simulations in Gazebo, we have run our strategy in real time with Turtlebot and the link for the video of our work is here.

We also integrated our code with the visual servoing team, whose task is to precisely place the turtlebot near the pick and place table. The link for the video our both tasks is here.

5. Conclusion and Future Work

In this project, we had created a map of the environment using 'gmapping' package and used 'amcl' package to navigate autonomously in the map. We had developed various nodes to achieve our task of detection and localization of markers with respect to map and make the turtlebot to go to localized positions of markers to full fill the task of pick and place of objects.

As we know the map of the environment earlier, we have used a very basic strategy of finding and localizing the markers by just moving to center of the map and rotating the turtlebot around itself to find the markers. Instead of this strategy, we can use more complex exploration techniques to explore the environment to detect and localize the markers using 'rrt_exploration', 'frontier_exploration' packages.