ENPM809Y Final Project

December 16, 2022

Instructor:

Prof. Zeid Kootbally

Student:

Ji Liu

Yi-Chung Chen

Weili Su

Table of Contents

Introduction

The goal of this project is to navigate the Turtlebot to reach the final destination by retrieving the relevant information from an Aruco marker in a simulated environment (Gazebo). To complete this task, two ROS2 packages were developed using C++. The first package ("odom_updater") establishes the correct coordinate system transformation between the robot's odom frame and the robot's base footprint frame, which is required for the robot controller to correctly direct the robot to desired locations in the simulated environment. The second package ("target_reacher") is responsible for first directing the robot to reach goal #1, where it could scan the Aruco marker to retrieve the location of goal #2, i.e., the final destination. The package is then responsible for directing the robot to reach the final destination.

These two packages were successfully implemented and the Turtlebot could be reliably directed to scan the Aruco marker and reach the correct final destination. Additional tests were also performed where changes were made such as switching the Aruco marker loaded in the Gazebo, changing the relative coordinate of the final destination location, etc., and our robot can behave accordingly to the changes, suggesting the robustness of our packages. Below the approaches are explained in detail.

Approach

Establishing correct coordinate frames

To direct the robot to the final destination, the logic as shown in the flow chart was followed (Figure 1). This flow chart is mainly implemented by the "target_reacher" package. However, for this package to run properly, it is necessary to first establish the correct tree of coordinate system transformations, which is maintained by the TF2 package. For the relationship between coordinate frames to be properly defined, the TF2 package maintains a tree structure of coordinate frames, where each frame has a unique parent frame, except for the root frame. Upon starting the original simulated environment in Gazebo, there is a disconnect between the "robot1/odom" frame and "robot1/base_footprint" frame (our Turtlebot is referred to as "robot1"). The "robot1/odom" frame is the fixed reference frame where the robot's location and orientation are defined. The "robot1/base_footprint" frame is a frame representing the 2D projection of the robot's location onto the XY plane. Driving the robot to the final destination requires that the correct link between these two frames is established. To address this, the ROS2 package "odom_updater" was implemented. This package implements a class called "OdomUpdater". The member attributes of this class include a subscriber as well as a TF2 broadcaster.

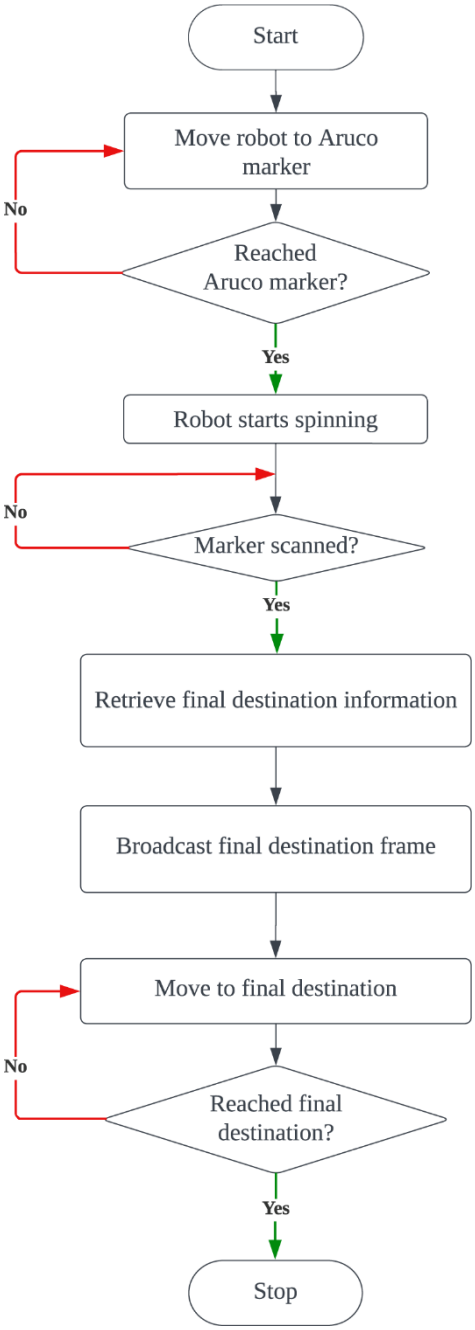

Figure 1 The flow chart representing the logic of our implemented ROS2 packages.

The subscriber subscribes to the topic "/robot1/odom" where the robot's location in reference to the "robot1/odom" frame is being actively published. The subscriber uses the following callback function ("**sub_callback**"), shown here using pseudo-code, to allow the TF2 broadcaster to broadcast the

transformation between “robot1/odom” and “robot1/base_footprint” frame, thereby establishing the correct link between the two in the TF2 tree (Box 1):

Box 1

Function sub_callback

copy robot’s location and orientation from message

build new geometry message with copied information

broadcast the new message using the TF2 broadcaster

End Function

With “odom_updater” running, the correct link can be established between the “robot1/odom” frame and the “robot1/base_footprint” (Figure 2).

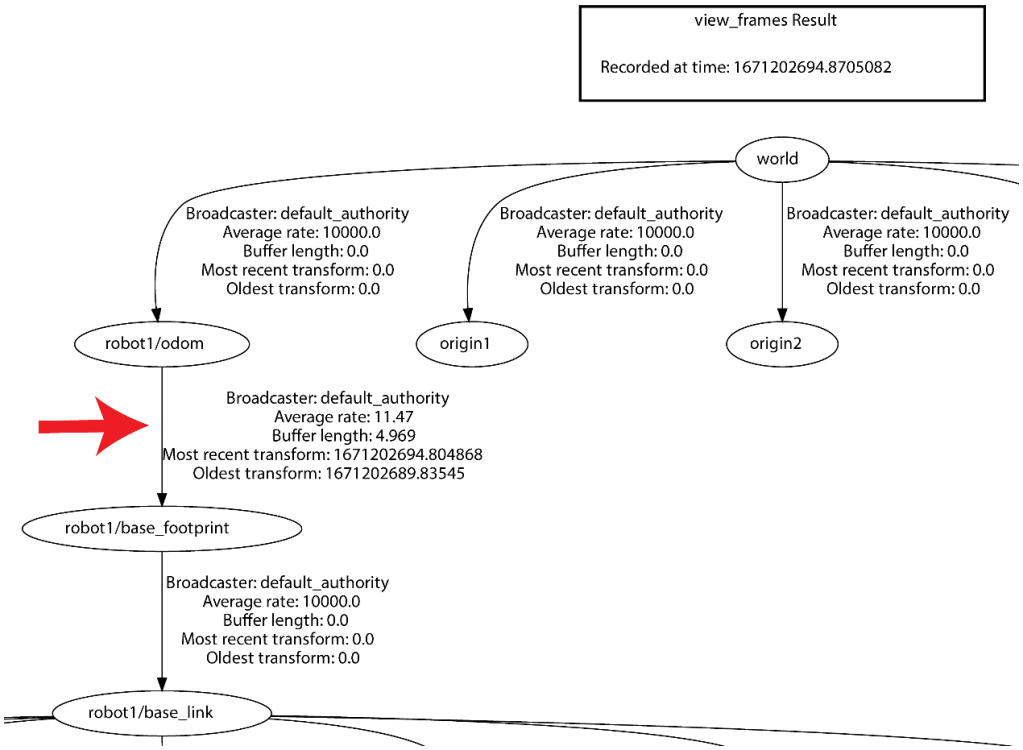

Figure 2 TF tree after the correct link (pointed by the red arrow) is established between “robot1/odom” frame and “robot1/base_footprint” frame. Only part of the tree is shown for clarity.

Directing the robot to the marker

Next, the "target_reacher" package was implemented to realize the logic shown in Figure 1. The package implements the "TargetReacher" class. The key member attributes include the following:

Member attribute	**Functionality**
m_bot_controller	Direct Turtlebot to goal locations
m_vel_pub	Command robot velocity
m_marker_sub	Subscribe to "/aruco_markers" topic
m_goal_reach_sub	Subscribe to "/goal_reached" topic
m_currloc_tf_listener	Look up the current location of the robot
m_stat_broadcaster	Broadcast final destination frame
m_des_tf_listener	Look up the final destination frame

These member attributes carry out key functions for directing the robot to the desired locations.

First, the robot needs to reach goal #1, where the Aruco marker can be scanned. To move the robot to the Aruco marker, a timer callback function ("**timer_move_bot_callback**")is used that runs every 0.5 seconds. The pseudo-code for the function is shown in Box 2.

The first part of this function is responsible for driving the robot to goal #1. The callback first determines whether the robot has reached the Aruco marker's location by reading a flag variable ("m_marker_reached") which takes a value of 0 if the robot has not reached the marker, and takes 1 otherwise. If "m_marker_reached" reads 0, the target reacher object will load the marker location from the node parameters and compare it with the current goal location of the robot stored as member attributes. If the two mismatches, the target reacher object updates the current goal with the marker location and commands the robot towards the marker via "m_bot_controller". The purpose of storing the current goal of the robot is to prevent having to command the bot controller with the same goal repeatedly.

When the robot arrives at the marker, the flag variable "m_marker_reached" is set to 1 in a separate callback function ("**goal_callback**", Box 3).

Box 2

```
Function timer_move_bot_callback
    If m_marker_reached == 0
        Load the marker location from node parameter
        If current goal do not match maker location
            Set marker location as current goal
            Command robot controller with current goal
        End If
    Else
        If m_final_des_found == 1 and m_final_des_reached == 0
            Try
                Look up final destination frame in reference to robot1/odom frame
            Catch
                Issue warning to user
                Return
            Retrieve x,y coordinate of final destination under robot1/odom frame
            If current goal coordinate and final destination coordinate mismatch
                Set final destination coordinate as current goal coordinate
                Command robot to reach current goal
            End if
        End if
    End If
End Function
```

Reaching marker location

Once the robot reached the target location (goal #1 or goal #2), a message will be published on the topic "/goal_reached". The class member attribute "m_goal_reach_sub" is a subscriber to this topic, and its callback function ("**goal_callback**") determines the next action for the robot, the pseudo-code of which is shown in Box 3. The callback function first determines which goal location the robot has reached (goal #1 vs goal #2) by using "m_currloc_tf_listener", a TF2 listener, which looks up the transformation between the "robot1/odom" frame and the "robot1/base_footprint" frame. The listener extracts the XY coordinate of the robot and compares it with the marker location loaded from the node parameter to determine if the robot has reached the marker. If so, the target reacher sets the flag variable "m_reached_marker" to 1 and then commands the robot to start spinning to scan the Aruco marker by publishing a pure angular twist

message on the "/robot1/cmd_vel" topic via "m_vel_pub". This is only performed when the flag variable "m_final_des_found" equals 0, meaning the final destination information has not been retrieved. This flag variable is set to 1 after the Aruco marker is successfully scanned (Box 4). The second part of this callback function deals with the scenario when the robot reaches the final destination and will be discussed later.

Box 3

```
Function goal_callback
        Determine the robot's current location
        If robot is at Aruco marker
                Set m_marker_reached to 1 if previously 0
                If m_final_des_found == 0
                        Set angular velocity to 0.75
                End If
                return
        End If
        If robot is at final destination
                Set flag variable for reaching destination to 1 if previously 0
        End If
End Function
```

Retrieving final destination information

After the robot successfully scanned the marker, the marker information will be published on the topic "/aruco_markers". Class member attribute "m_marker_sub" subscribes to this topic, and the callback function is "**marker_callback**" (Box 4). This callback retrieves the marker ID from the newly published message and checks if the information about the final destination has been already found by checking the corresponding flag variable, "m_final_des_found". If it equals 0, suggesting that the final destination information has not been retrieved, the callback function proceeds to retrieve from the node parameter the final destination reference frame ("origin1", "origin2", "origin3", or "origin4") and the destination coordinate specific to the marker ID. Next, the callback function broadcasts the destination as a static frame under the reference frame via "m_stat_broadcaster". Before the callback exits, it also set the flag variable "m_final_des_found" to 1 to prevent the callback function from being executed multiple times.

Box 4

```
Function marker_callback
        Retrieve marker ID from new message published on "/aruco_markers"
        If m_final_des_found == 0
                Stop the robot momentarily
        Else
                return
        End If
        Retrieve the reference frame of final destination from node parameter
        Retrieve the final destination coordinates under the reference frame from node parameter
        according to marker ID
        Broadcast the final destination frame under the reference frame
        Set m_final_des_found to 1 if previously 0
End Function
```

Directing the robot to the final destination

Finally, the robot is directed to the final destination by the same timer callback that is used to drive the robot to goal #1 ("**timer_move_bot_callback**", Box 2). The second half of the callback function first asserts that the robot has found the final destination information ("m_final_des_found" equals 1) but has not reached the destination ("m_final_des_reached" equals 0). If these criteria are met, the callback retrieves the coordinate of the final destination via "m_des_tf_listener", a TF2 listener that looks up the transformation of the final destination frame under the "robot1/odom" frame. Next, the callback sets the coordinate of the final destination as the current goal of the robot through the bot controller. After the robot reaches the final destination, the second part of the "**goal_callback**" sets "m_final_des_reached" to 1 and thus prevents further execution of "**timer_move_bot_callback**".

Together through these callback functions, the robot could successfully move through the steps as shown in the flow chart (Figure 1) and reached the final destination. Figure 3 shows the simulation with the "odom_updater" and "target_reacher" packages running where the robot correctly completed the steps required for this task (Figure 3A-D). After the marker is scanned, the final destination frame is also correctly set up to guide the robot to the final goal location (Figure 3E, F). Screen recordings of the same simulation can be found here for Gazebo and here for Rviz. Here Aruco marker 2 was loaded in the .world file and "frame_id" of the final destination was set to "origin2" in the .yaml file.

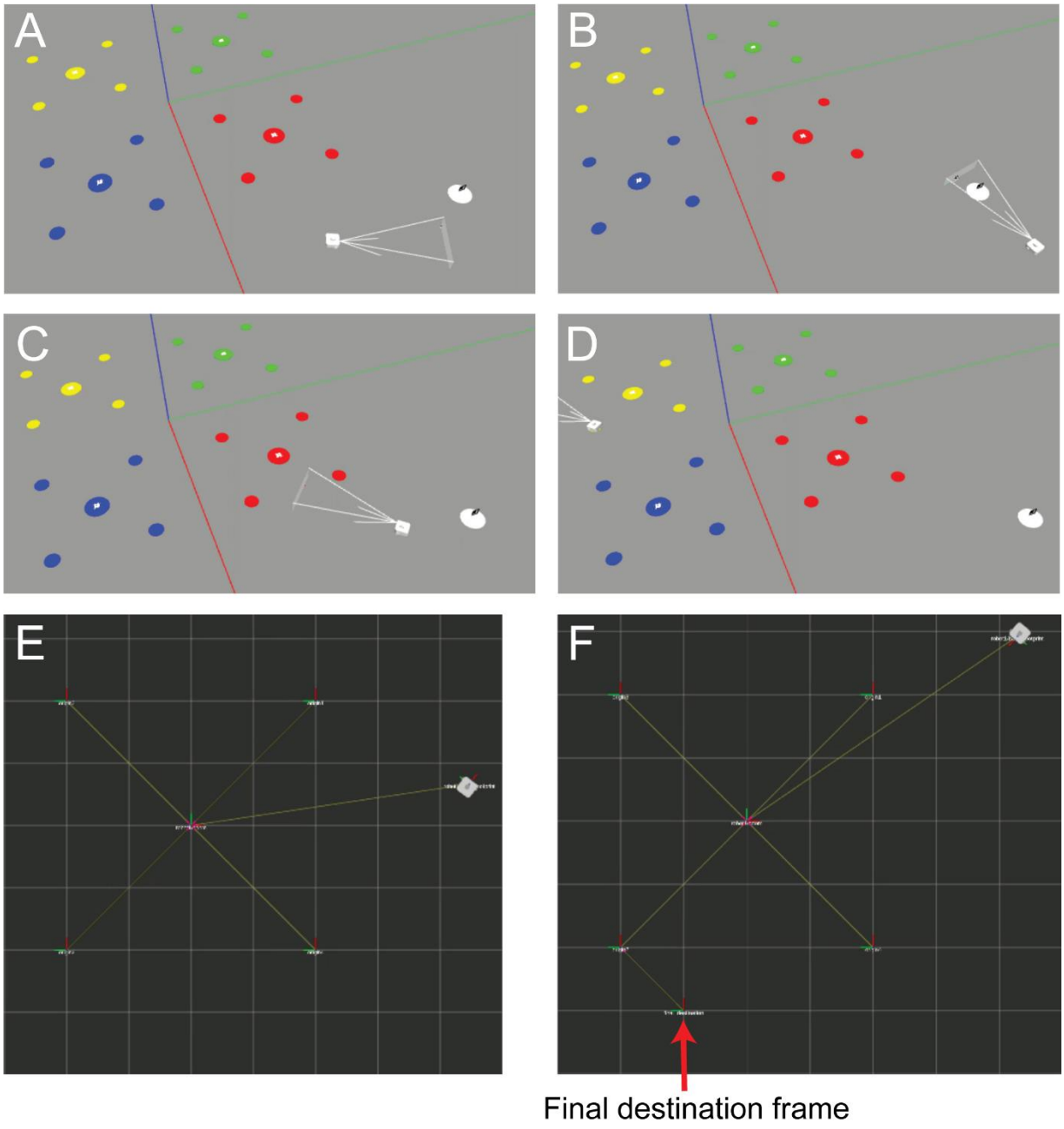

Figure 3 Simulation of Turtlebot in Gazebo while running the developed ROS2 packages. (A)-(D): The Gazebo environment showing the robot going through different steps in the flow chart (Figure 1). (A) The robot moves towards the Aruco marker. (B) The robot spins and scans the marker. (C) The robot is moving towards the final destination after the location information has been retrieved. (D) The robot arrived at the final destination. (E) and (F): The coordinate frames as shown in the Rviz environment. (E) The final destination frame is not broadcasted at the beginning of the simulation. (F) The final destination frame is correctly established under origin 2 (red arrow) after the marker is scanned. In this simulation, Aruco marker 2 was used and "frame_id" of final destination was set to "origin2" in the .yaml file.

Challenges

Challenge 1

The first challenge is how the flow chart can be realized (Figure 1) under the framework of callback functions. The flow chart has a well-defined sequence of actions. However, the callback functions are conceptually concurring, and thus their sequence of execution is not clear. To overcome this issue, flag variables are added as member attributes of the class "TargetReacher". The flag variables indicate whether the robot has reached goal #1 and goal #2 and whether the final destination information has been found. With the addition of these flag variables, it could be more precisely pinpointed which step within the flow chart the robot is at. Thus these flag variables serve as flow control. In addition, the flag variables can prevent unnecessary commands from being executed and thus save computational resources.

Challenge 2

To drive the robot to goal #1 and goal #2, a timer callback function is used that executes with a fixed frequency. Choosing a suitable frequency presents another challenge. If the frequency is low, then the robot might not be directed to new goals promptly. For example, the robot could stay at the Aruco marker for an extended amount of time before the timer function directs the robot to the new goal. However, if the frequency is too high, it might waste computational resources as the robot's state updates only sporadically. Given these considerations, the frequency of 2 Hz was arrived at through trial and error, and this frequency seems suitable and the robot can be promptly driven to the new goal location.

Challenge 3

The third challenge involves setting the correct include path for the project, especially since the project involves working with a custom ROS2 interface ("ros2_aruco_interfaces") and it requires finding the generated "aruco_markers.hpp" file after the interface package was built. Initially, it was thought that the generated file would be in the source folder of the corresponding package, but that was misguided and the generated files were actually located under the "install" folder of the ROS2 workspace. In addition, when ROS is running, the .yaml parameter file being referred to for the target reacher node is the one located in the workspace "install" folder rather than the original one in the package folder. Thus to modify parameters such as the final destination coordinate, one needs to modify the file in the "install" folder but not the one

in the package folder (unless one rebuilds the package). These now make sense because the “install” folder is the target installation location of all packages as specified in the CMakeLists.txt file.

In addition, close attention needs to be paid to the include path for using TF2 packages. For example, using the regular broadcaster one needs to include “tf2_ros/transform_broadcaster.h”. However, for the static broadcaster, one needs to include “tf2_ros/static_transform_broadcaster.h”. Therefore, to use the variety of functionalities provided by TF2, one needs to make sure that the correct packages are included.

Challenge 4

The fourth challenge is to correctly read the node parameter for example to retrieve the final destination coordinate. The method rclcpp::Node::get_paramter returns the type rclcpp::Parameter, and to properly retrieve the actual value of the parameter, this returned variable needs to be cast as double or string, using the method as_double() or as_string() respectively.

Challenge 5

The fifth challenge is to recognize the difference between TF broadcaster and static TF broadcaster. For example, the broadcaster to broadcast the final destination frame under one of the origin frames needs only to be static. The static broadcaster needs to broadcast the transformation only once, which is sufficient for that information to be retrieved by other callback functions at a later time. Therefore, using the static broadcaster is a more efficient way to achieve the desired functionality.

Contributions

	Code – “odom_updater”	Code – “target_reacher”	Report	Doxygen Documentation
Ji Liu	✓	✓	✓	
Yi-Chung Chen		✓	✓	✓
Weili Su			✓	

Resources

We used the following ROS2 tutorials for reference:

https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html

https://docs.ros.org/en/foxy/Tutorials/Intermediate/Tf2/Writing-A-Tf2-Listener-Cpp.html

https://docs.ros.org/en/galactic/Tutorials/Intermediate/Tf2/Writing-A-Tf2-Broadcaster-Cpp.html

Course feedback

Overall we found the course to be very informative, and the lecture and lecture notes are well structured. We indeed learned a lot about C++ and ROS2. One suggestion would be to have more hands-on practices with C++ and they can be in the form of smaller assignments rather than big group projects such that we could gain more practical experience with the programming language.