Robotics and Mechatronics

Mini Project Three

Mohammad Montazeri
School of Mechanical Engineering
College of Engineering, University of Tehran
Tehran, Iran; 810699269
mohammadmontazeri@ut.ac.ir

***Abstract*—Robot Operating System (ROS) is a flexible framework for writing robot software, offering tools, libraries, and conventions to simplify the creation of complex and robust robot behavior. This project leverages `turtlesim`, a popular educational simulator within ROS, to develop practical expertise in ROS. The objective is to control, manipulate, and modify the `turtlesim` environment. Through these activities, the project aims to enhance users' proficiency in ROS, preparing them for more advanced robotic programming and development tasks.**
***Index Terms*—ROS, turtlesim, command, error, service**

I. Introduction

The Robot Operating System (ROS) is a powerful and versatile framework designed to simplify the development of robot software. It provides a rich set of tools, libraries, and conventions that enable the creation of complex and scalable robotic applications. One of the key educational tools within ROS is `turtlesim`, a simple yet effective simulator that allows users to control a virtual turtle within a graphical environment. By interacting with `turtlesim`, users can learn and practice essential ROS concepts such as topics, services, and parameters. This project aims to utilize `turtlesim` to gain hands-on experience in controlling and manipulating the simulation environment, thereby building a strong foundation in ROS that will be beneficial for more advanced robotic programming and development endeavors. The main objective is to draw the assigned characters shown in Figure 1 using turtles in `turtlesim` environment and then rotating them on their own coordinates. For a bonus point, both of these characters will be ready to move by the client's will, using keyboard arrow keys.

(a) Greek "pi" symbol

(b) English number "five" symbol

Fig. 1. First (left [5]) and second (right) assigned characters for this project

Services Available for `turtlesim` in ROS

In the Robot Operating System (ROS), the `turtlesim` package is a popular tool used to teach and demonstrate various ROS concepts (Figure 2). First of all, we need to gain a better understanding of `turtlesim` and standard services available for it. Here is a thorough list of services you can see by executing `turtlesim_node` and running `rosservice list`:

A. */clear*

- **Service Type**: `std_srvs/Empty`
- **Description**: This service clears the turtlesim background, essentially removing any drawings made by the turtle. It resets the simulation background to the initial state without moving the turtle.

B. */kill*

- **Service Type**: `turtlesim/Kill`
- **Description**: This service deletes a turtle from the simulation. You need to provide the name of the turtle you want to remove.
- **Usage**: You call this service with the name of the turtle as a parameter.

C. */reset*

- **Service Type**: `std_srvs/Empty`
- **Description**: This service resets the turtlesim to its original state, which includes clearing the background and resetting the position and orientation of the default turtle to its starting location.

D. */spawn*

- **Service Type**: `turtlesim/Spawn`
- **Description**: This service spawns a new turtle at a specified location and orientation. You need to provide the coordinates (x, y), the orientation (theta), and optionally the name for the new turtle.
- **Usage**: This is used when you want to add new turtles to the simulation.

E. /turtle1/set_pen

- **Service Type**: `turtlesim/SetPen`
- **Description**: This service changes the properties of the turtle's pen, such as color (r, g, b), width, and whether the pen is up or down (drawing or not drawing).
- **Usage**: Useful for changing the drawing style of the turtle.

F. /turtle1/teleport_absolute

- **Service Type**: `turtlesim/TeleportAbsolute`
- **Description**: This service teleports the turtle to an absolute position in the simulation with a specified orientation.
- **Usage**: This can be used to move the turtle instantly to any position within the simulation space.

G. /turtle1/teleport_relative

- **Service Type**: `turtlesim/TeleportRelative`
- **Description**: This service teleports the turtle relative to its current position and orientation.
- **Usage**: You can move the turtle forward by a specified distance and rotate it by a specified angle.

H. /turtle1

- **Service Type**: `std_srvs/Empty`
- **Description**: This is not actually a standard service but a node namespace prefix. By default, turtlesim starts with one turtle named "turtle1". Services related to this turtle will typically use this namespace.

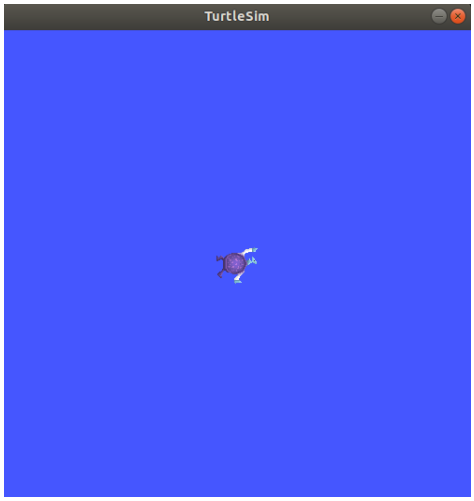


Fig. 2. A simple overview of turtlesim environment. [3].

These services allow us to manipulate the `turtlesim` environment in various ways, providing a comprehensive toolset for learning and demonstrating ROS functionalities. We will be using some of these features towards our own targets later.

II. Workflow of the Program

Here's the main sequence a client should traverse to get the desired response from the provided program.

1) copy the *catkin_ws* appended to this report to your *catkin* directory.
2) source the files to acquaint the *roscore* with the *my_pkg* package; this must be done using this bash command: `source /catkin_ws/devel/setup.bash`
3) specify which of the two characters shown in Figure **??** you want to be sketched on *turtlesim* environment as you launch the files using this bash command: `roslaunch my_pkg my_launch.launch character:=1` [1]
4) wait to see the turtles emerging on the screen gradually.
5) enjoy the scenery for 1.2 seconds until the rotation of turtles start.
6) click on the terminal and use arrow keys to move the turtles wherever you want; note that it's recommended to avoid colliding with walls which causes yellow warnings on tour terminal; it must also be noted that working with arrow-keys is a little bit tricky here, since there might be lags, unwanted movements, untracked key-presses and delays based on the processing capabilities of the your system's resources.[2]
7) exit the program using ctrl+c in the running terminal window.

III. Investigation of Codes

In this section, we'd have a brief explanation of the codes written for this project. Although all codes have sufficient comments which make them realizable, this section is suggested to be read for further information.

A. Overview

This report comes along a directory containing the *catkin workspace* which includes the *build*, *devel* and *src* folders. The *build* directory contains files compiled with `catkin_make` command; *devel* directory also contains project files which are beyond the subject of this assignment. The *src* folder, however, contains the base files of this project, under the package-name of *my_pkg*. Here's where one can see the script files (`drawer.py`) in *src* folder and launch files (`my_launch.launch`) in *launch* folder.

B. Launch File

Within `my_launch.launch`, one can see that an `arg` tag is used to define a parameter used inside the script file later, to specify which of the 2 assigned characters must be initialized. Then the main `turtlesim_node` is defined and a `/clear` service is called on it to erase unwanted drawing on the *turtlesim* environment. The script file is then added along with its parameter which would be equal to '1' by default, if not specified when calling the launch file.

[1] leaving the parameter empty will show the first character by default.

[2] don't forget to select the terminal window before working with arrow-keys, and remember to hold the keys as long as it takes the turtles to respond.

C. Script File

- Within `drawer.py`, an interpreter is allocated to the program first, followed by importing the necessary libraries and packages.
- `key_mapping` dictionary is then introduced to map the pressed arrow keys to the directions to which they point, used later to move the turtles.
- `get_key` is used to read the keys pressed in the terminal while the *turtlesim* environment is running.
- `flush_input` is used to erase the previously entered keys from memory, in order to prevent lags in executing the key tasks.
- `Change_bg_color` function sets a new color (defined in RGB format) to the background of the *turtlesim* environment. By default it applies a yellow color but one can specify any desired color by giving its R-G-B factors to this function, later in the main part of the code. This function also deletes the default turtle which comes with the `turtlesim_node` by default.
- `Drawer` function gets the name of the character designated to be sketched and based on that, decides which function to be executed next. It also provides the name list of all turtles created. The *delay* parameter defined in this function controls how fast the turtles must be created.
- `Turtle_pixel_one` or `_two` are used to iteratively `/spawn` turtles, pixel by pixel on the *turtlesim* board. To do so, the *x, y, theta* and *name* of each turtle is specified based on the chosen character and passed to the `spawn_client` which was borrowed from `Drawer` function.
- `Move_turtles` function is used for linear and angular velocity of turtles, taken as *speed* and *angular.z* relatively. In this function, a list is defined as `publishers` which will contain the *velocity-message publisher* of each turtle. Another list called `monitor` is also defined which will contain an object of *Turtle_pose* class for each turtle, which will monitor its pose and do other stuff.[3] The variable *rate* controls the speed of motion signals and is set to 5 Hz. Then using `while not rospy.is_shutdown()` loop, the motion of turtles start, splitting a thread to listen to the keyboard and another one to publish the velocity message of each turtle. These velocity messages are of type Twist imported from `geometry_msgs.msg`. Within this vital loop, if the pressed keys read by `get_key` function were one of the pre-defined keys in `key_mapping` dictionary, the corresponding linear velocity would be assigned to all turtles, after their angular velocity being temporarily set to zero via `obj.set_orientation()` line; otherwise, turtles would keep their angular velocity specified by *angular.z* parameter.
- `Turtle_pose` class gets the name of a turtle and defines a subscriber of its *Pose*, which is a message type from `turtlesim.msg` publisher. Using `callback` method the *x, y* and *theta* of turtle is saved as `self.pose` and returned to any applicant via `get_theta` method. `set_orientation` method uses the same route as `Move_turtles` function to temporarily set the angular velocity of turtle to zero so that it can move in one of the *rightward, leftward, downward* and *upward* uninterruptedly. This method also uses `TeleportAbsolute` service to set the *theta* parameter of the turtle to zero, so it faces right, otherwise, its local *left, right, up* and *down* directions would differ from the global observer of the *turtlesim* board.
- In the main trunk of the program, its node would be initiated under the name `TurtleCharacter`.
- Using `rospy.get_param`, the argument of the program which is the *to-be-sketched character* is imported from the launch command.
- Then the main functions explained above get called one by one to start the sequence of the program. Note that a slight delay of 1.2 seconds is put before rotating (and/or moving) the turtles so that the observer can thoroughly witness the sketched character before it starts shaking! By this, the codes end.

[3]this class will be explained later on its own section

IV. Errors and Handlings

In this section, some of the errors occurred during the processes of working with ROS are mentioned and the solutions exerted to fix them are also briefly explained.

A. Starting Turtlesim

Error `[ERROR] [1717396146. 738976425]: [registerPublisher] Failed to contact master at [localhost: 11311]. Retrying ...` happened while trying to start up the turtlesim environment using `rosrun turtlesim turtlesim_node` command. Turns out I hadn't have initialized ROS using `roscore` command.

B. Compiling packages after creating message file

The following error happened while trying to compile the entire *catkin* workspace using `catkin_make` after creating a new message file and modifying relevant files. Turns out I deleted the `FILES` line from the *CMakeLists.txt* file. Re-adding that, solved the issue.

```
CMake Error at /opt/ros/noetic/share/genmsg/
cmake/genmsg-extras.cmake:77 (message):
add_message_files() called with unused
arguments: demo_msg.msg
Call Stack (most recent call first):
demo_pkg/CMakeLists.txt:50 (add_message_files
)
Configuring incomplete, errors occurred!
--
See also "/home/saeed/catkin_ws/build/
CMakeFiles/CMakeOutput. log".
See also "/home/saeed/catkin_ws/build/
CMakeFiles/CMakeError.log".
Invoking "cmake" failed
```

C. Using Enter instead of Tab

After creating a service of my own, consisting of a new '*.srv*' file, a server and a client, I tried to run the server alone to see if it worked correctly. After using `rosservice list` to see my custom service name, *meaner*, and advancing to give the arguments to `rosservice call /meaner`, I faced the error `Usage: rosservice call /service [args ... ] rosservice: error: Please specify service arguments`. Turns out I had to use *Tab* key after the mentioned command, instead of *Enter*, which was the key I used.

D. Simultaneous launch despite system delay

I struggled so much on running a trial launch file, which called the `/clear` service after it started the `turtlesim_node`. It always got an error like `ERROR: Service [/clear] is not available. [clear-3] process has died [pid 3416, exit code 2, ...]`. That took a lot of my time to be debugged. I first thought that the problem came from my launch file and it needed to be compiled first. So I changed to the `/catkin_ws/` directory and tried to compile my package with `catkin_make`. That caused a new error which I managed to solve eventually[4], but didn't work out for the main error I used to get. So I kept manipulating my launch file, to get another error: Nodes with the same *name* attribute[5]. Fixing that and keeping the same procedure to alter my launch file, I started to suspect that the `/clear` service comes from `std_srvs`, not `rosservice`. Changing that in my launch file caused a new error which took me a lot of time and effort to fix[6]. So starting back from the first steps, I tried to replace `/clear` service with other services defined for `turtlesim_node` like `/kill`, `/reset` or `/spawn`. All of them confronted errors such as `ERROR: Service [/kill] is not available`. After that, sick of all my indecisive attempts, I took a long break to come back with a new idea from my web-searches: to execute my node as a bash-script. Although I was able to access my desired services (like `/clear`) in my python-scripts, but this idea was more mesmerizing. So I used the codes provided by AI [2] to create a node based on a new bash-script. That, too, got an error[7]. Fixing that, I came to a relief when I saw my files launch uninterruptedly. Finally, turns out the nodes introduced in my launch file used to get executed so fast, that their order was not observed. In other words, the error I was encountering `ERROR: service [/clear] is not available` occurred because the turtlesim_node might not have be fully initialized when the rosservice call was made. This could happen because the launch file started both nodes simultaneously without ensuring that the turtlesim_node was fully ready. One way to ensure proper synchronization was to use the rosservice command with a slight delay to allow the turtlesim_node to initialize properly. However, ROS launch files did not natively support delays directly. Instead, I used a small workaround by calling rosservice within a rosrun command with a shell script to introduce a delay. That's how this enormous and crucial error was resolved.

[4]This error is explained at section IV-E.
[5]This error is explained at section IV-F
[6]This error is explained at section IV-G
[7]This error is explained at section IV-H

E. Compile error

As explained above, when I tried to compile my package with `catkin_make`, I got `The build space at '/home/saeed/catkin_ws/build' was previously built by ''. Please remove the build space or pick a different build space`. To fix that, I manually deleted the *build* directory of my `/catkin_ws/` and retied the compile process.

F. Nodes with the same name attribute

Error `RLException: roslaunch file contains multiple nodes named [/TurtleCharacter]. Please check all <node> 'name' attributes to make sure they are unique.` appeared when I executed one of my trial launch files, which contained two nodes, including `name="TurtleCharacter"`. Turns out nodal names must be unique.

G. Incorrect pkg and type attributes

When I used a node like `<node pkg="std_srvs" type="Empty" name="turtle2" args="call /clear" />` within my launch file, I faced an error like `ERROR: cannot launch node of type [std_srvs/Empty]: Cannot locate node of type [Empty] in package [std_srvs]. Make sure file exists in package path and permission is set to executable (chmod +x)`. Turns out, unlike my temporary suspicion, the `/clear` command shouldn't be written as a launch node in this way; rather, it should be like: `<node pkg="rosservice" type="rosservice" name="clear" args="call /clear" />`.

H. Oblivion of permissions

I wrote my bash-script named `spawn_turtle.sh` in the `src/` directory of my package, containing `sleep 5 rosservice call /spawn 2 2 0.5 turtle2`. Then I added a node as `<node name="spawn_turtle" pkg="my_pkg" type="spawn_turtle.sh" output="screen" cwd="node"/>` to my launch file. Launching this file faced `ERROR: cannot launch node of type [my_pkg/spawn_turtle.sh]: Cannot locate node of type [spawn_turtle.sh] in package [my_pkg]. Make sure file exists in package path and permission is set to executable (chmod +x)`. Turns out I hadn't have given the execution permission to the bash-script before using it in my launch file.

I. Connection error in Linux

Once I tried to start Linux on my virtual machine, when I confronted `*ERROR* Failed to send host log message. [FAILED] Failed to start Network Name Resolution.` Although I knew having my VPN turned on in my host OS (Windows) interrupts connection on Linux, this error got me surprized since my VPN was off. After a few times of rebooting the virtual box, and getting the same error, I figured out that my DNS address doesn't cope with the guest OS (Linux). I remembered that I had modified my DNS long ago to be able to bypass the OpenAI [2] sanctions on Iran DNS addresses. Resetting my IP and DNS solved the problem in the end.

J. Linux fatal error

I have been working couple of days on this project, using Oracle VM to bring up my Linux. Sometimes I left my system on sleep for many hours so it would automatically hibernate with the virtual box being opened. This, I believe, harmed my system gradually until I once confronted a blank dark screen in my Linux which didn't have any functionality. I restarted the VM or my entire PC for many times, but it didn't work. I believe the lack of memory space had to do something with it too, so I tried entering the bios environment to solve the problem. Despite all of my attempts to repair the system, even with following all of the instructions taken from the web, I wasn't able to rehabilitate the system. I even couldn't restore my files, which took a lot of my time and power to reach to a noticeable level. Frustrated and hopeless, I uninstalled the entire virtual machine and migrated to the online simulator introduced in the classroom, *the Construct* [4]. Moving to this new environment, I had to start from scratch. That made me reconsider some of the previously mentioned error handlings. For example, I figured out that the resources this website allocates to me, in contrast to my own system, handles the ROS nodes better and faster. Thus the problem discussed in IV-D didn't ever happen in *the Construct*. That's why I haven't included the mentioned bash script file in my project.

V. Conclusion

In this project, we successfully demonstrated the ability to manipulate and control the turtlesim simulation environment in ROS using Python. Our key accomplishments included *Interactive Control* and *Practical Application of ROS Tools*. Overall, this project provided a hands-on opportunity to explore the capabilities of ROS in a simulated environment, demonstrating how Python scripts can be used to control and modify simulations in real-time, thereby laying a foundation for more complex robotic applications and system integrations.

VI. Results

Some videos from the workflow of this project are appended in this link along with supporting materials.

(a) Sketched character 2: "pi" symbol

(b) Sketched character 2: "five" symbol

Fig. 3. Results for first (left) and second (right) assigned characters for this project

References

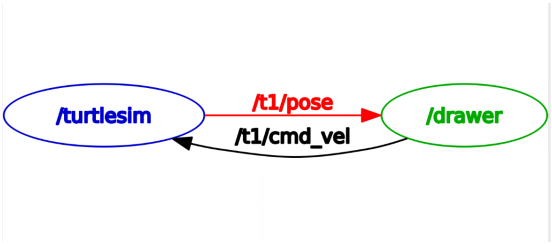
[1] Open Robotics, "Wiki ROS", https://wiki.ros.org/

[2] OpenAI, (2023), ChatGPT (Sep 25 version) [Large language model], https://chat.openai.com/chat

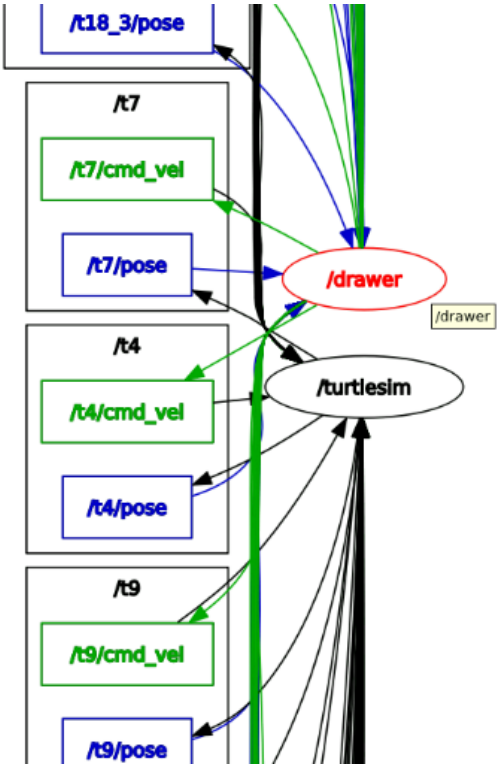
[3] ROS Documentation, https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Introducing-Turtlesim/Introducing-Turtlesim.html

[4] The Construct: Where Your Robotics Career Happens, 2024, https://app.theconstruct.ai/

[5] Image courtesy of https://codepoints.net/U+03D6?lang=en and https://graphemica.com/%CF%96



(a) Nodes only mode



(b) Nodes and Topics mode (partial view)

Fig. 4. Results from `rqt_graph` for active nodes and topics overview