ROS 2 workshop

Ragesh Ramachandran

Feb 09, 2022

CONTENTS

CHAPTER

ONE

SESSION 1 - ROS 2 CONCEPTS AND FUNDAMENTALS

1.1 ROS 2 File System

1.1.1 1. Introduction

During this tutorial, you will learn how to navigate through your ROS 2 system. In addition, you will start your first ROS 2 nodes and create your own ROS workspace for further tutorials. You can use the given links in the documentation for further information.

- Lines beginning with $ are terminal commands.
  - To open a new terminal → use the shortcut ctrl+alt+t.
  - To open a new tab inside an existing terminal → use the shortcut ctrl+shift+t.
  - To kill a process in a terminal → use the shortcut crtl+c.
- Lines beginning with # indicate the syntax of the commands.

1.1.2 2. ROS File System

Before starting make sure that your system is aware of the latest ROS 2 packages:

```
$ sudo apt update
$ rosdep update
```

If you just installed **rosdep**, please run this commands first:

```
$ sudo rosdep init
```

The ROS 2 File System consists of ROS 2 packages – the smallest build part in ROS 2.

Usually, a ROS File System consists of hundreds of different packages. To navigate efficiently through your ROS system, ROS provides different management tools. The most common tools are described in the example of the ROS package turtlesim.

First of all, you have to source the ROS installation and its install workspace!

Hint: This has to be done for every new terminal window.

```
We are using **foxy** as ROS Distro

$ source /opt/ros/foxy/setup.bash
```

or

in your workspace

```
$ source install/setup.bash
```

1.1.3 3. Workspace

Colcon is a build tool for ROS 2. A workspace is a folder, where you can modify, build, and install packages. It is the place to create your packages and nodes or modify existing ones to fit your application. In the further tutorials, you will work in your workspace.

- Create workspace:

```
$ cd ~
$ mkdir -p ~/dev_ws/src
```

1.1.4 4. ROS packages

Two types of ROS 2 packages exist: binary packages and build-from-source packages.

ROS binary packages are in Ubuntu provided as debian packages, which can be managed via apt commands.

- Install a binary package:

```
$ sudo apt install ros-foxy-rosbag2*
```

```
# sudo apt install ros-<distribution>-<package-name>
```

ros-foxy-rosbag2* means all packages which start from "ros-foxy-rosbag2"

Locate a ROS package:

```
$  ros2 pkg prefix rosbag2
```

```
# ros2 pkg prefix <package_name>
```

List executables:

```
ros2 pkg executables action_tutorials_py
```

```
# ros2 pkg executables <package_name>
```

Build-from-source packages can be divided into packages provided by ROS 2 and your developments. Both have to be placed into the src folder of a ROS 2 workspace.

- Install an available build-from-source package:

  Hint: Ensure you're still in the dev_ws/src directory before you clone.

```
$ cd ~/dev_ws/src
$ git clone https://github.com/ros/ros_tutorials.git -b foxy-devel
```

```
# git clone -b <branch> <address>
```

Notice the "Branch" drop-down list to the left above the directories list. When you clone this repo, add the -b argument followed by the branch that corresponds with your ROS 2 distro.

To see the packages inside ros_tutorials, enter the command:

```
$ ls ros_tutorials
```

You will find you have four packages: `roscpp_tutorials rospy_tutorials ros_tutorials turtlesim`

Only **turtlesim** is ROS 2 package

- Resolve dependencies:

  Before building the workspace, you need to resolve package dependencies. You may have all the dependencies already, but best practice is to check for dependencies every time you clone. You wouldn't want a build to fail after a long wait because of missing dependencies.

  Hint: Ensure you're in the workspace root (~/dev_ws) directory before you run rosdep.

  If it is your first time to run `rosdep`, you need to run `rosdep init` first.

```
$ cd ..
$ rosdep install --from-paths src --ignore-src -r -y
```

  This command magically installs all the packages that the packages in your workspace depend upon but are missing on your computer. http://wiki.ros.org/rosdep

- Build the workspace with colcon

  Hint: Don't forget to source the ROS installation before build and make sure you are in the root of workspace(~/dev_ws).

```
$ source /opt/ros/foxy/setup.bash

$ colcon build --symlink-install
```

- Source the overlay When you build this workspace, your main ROS 2 environment is the "underlay". Now you can source overlay "on the top of" "underlay".

  Hint: If you open a new terminal, set up ROS 2 environment first.

```
you can start a new terminal window by
ctl + alt +t
```

```
$ cd dev_ws
$ source install/setup.bash
```

1.1.5 5. ROS nodes

A ROS node is an executable in the ROS environment. A node is always a part of a ROS package. How to start ROS nodes and how to display runtime information is explained in the example of the turtlesim package.

- Start a ROS node:

  Hint: You have to start each process in a separate terminal. Don't forget to source the ROS installation.

  ```
  $ ros2 run turtlesim turtlesim_node
  ```

  ```
  # ros2 run <package_name> <executable_node_name>
  ```

  A new window should pop up, displaying a turtle.

- Start another node to control the turtle:

  ```
  $ ros2 run turtlesim turtle_teleop_key
  ```

  ```
  # ros2 run <package_name> <executable_node_name>
  ```

  You can control the turtle with the arrow keys of the keyboard, if the current terminal running the turtle_teleop_node is selected.

The two nodes turtlesim_node and turtle_teleop_key are currently active on your ROS 2 system. Since systems that are more complex will consist of many nodes, ROS offers introspection tools to display information about active nodes.

- Display all active nodes:

  ```
  $ ros2 node list
  ```

- Display information about a node:

  ```
  $ ros2 node info /turtlesim
  ```

  ```
  # ros2 node info <active_node_name>
  ```

The given information comprises topics that are subscribed and published by this node.

1.2 Understanding ROS 2 nodes with a simple Publisher - Subscriber pair

Based on the ROS 2 Tutorials

1.2.1 Introduction

As we understood from the lectures, nodes are the fundamental units in ROS 2 which are usually written to perform a specific task. They can be created in a few different ways such as-

1. As simple in-line code in a script,
2. As local functions, and
3. As class objects… among others

We will be using the 3rd method, though it is the more complex, so as to get better used to this concept.

We start by writing two separate simple nodes, one that includes only publisher and another that includes only a subscriber. Finally, we will write a third node that includes both within the same program and are managed through an executor.

The first step is to create a python package to house all our nodes. You can do so using the command

```
$ ros2 pkg create --build-type ament_python <package_name>
```

(Make sure first that ROS 2 is sourced in every new terminal)

Make sure you run this command in the *src* directory of your workspace. You can use any package name you want, but for reference in this document, we call it `wshop_nodes`.

1.2.2 1. Publisher Node

The publisher and subscriber nodes used here are in fact the example code that ROS 2 provides.

We first present the code completely, and then discuss the interesting parts:

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String


class MinimalPublisher(Node):

    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1


def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
```

(continues on next page)

(continued from previous page)

```
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

Inside the python package you created above, there should be another folder with the same name. Create a python file inside that folder and paste this code in. You can name the file anything you want, but for reference in this document we assign the name `minimal_publisher.py` to it.

1.1 Explanation

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String
```

`rclpy` is the *ROS 2 Client Library* that provides the API for invoking ROS 2 through Python.`Node` is the main class which will be inherited here to instantiate our own node.`std_msgs.msg` is the library for standard messages that includes the `String` message type which we use in this node. This has to be declared as a dependency in `package.xml`, which we do next.

```
class MinimalPublisher(Node):

    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0
```

As explained above, we create a subclass of type `MinimalPublisher` using the base class `Node`.In the constructor `__init__()`, we pass the name of the node that we ish to assign to the constructer of the parent class using `super()`. The parent class `Node` takes care of actually assigning this string as a name.`self.publisher_ = self.create_publisher(String, 'topic', 10)` This line actually creates a publisher, using the message type `String` that we imported, with the name `topic` that we choose and having a queue size of `10`. Queue size is the size of the output buffer. The commands used till now are typical when creating a subscriber. What follows next is only logic that is relevant to this node, and you may implement this in any way depending on your requirements.

```
timer_period = 0.5  # seconds
self.timer = self.create_timer(timer_period, self.timer_callback)
self.i = 0
```

This creates a timer that ticks every 0.5s (2Hz), and calls the function `timer_callback` at every tick.

```
def timer_callback(self):
    msg = String()
    msg.data = 'Hello World: %d' % self.i
    self.publisher_.publish(msg)
    self.get_logger().info('Publishing: "%s"' % msg.data)
    self.i += 1
```

In the callback, we create an object `msg` of the type of the message we wish to publish, i.e `String`.We then populate the message with information we wish to publish. Looking at the description of the msg type using `ros2 interface show std_msgs/msg/String`, we see that it has only one field, which is `string data`. So we add a string into this field. Depending on the type of message used, you can populate it with relevant data.Once the data msg object is done, we simply publish it using the `publish()` method of the `publisher_` object.We also display this same message on the console for our verification using the `get_logger().info()` method of our `Node` class object.Publishing from within this timer callback ensures we have a consistent publishing rate of 2Hz. You could publish this in any way you want, using the proper message type and publish call.

```
def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
```

In the main method, we first declare that this Python script uses the rclpy library by invoking `init()` and passing any command line arguments provided (in this case none).We instantiate an object of the class we just created. Since the contructor already spawns the timer which publishes messages, no further action is needed to setup our node.The `spin()` method ensures that all the items of work, such as callbacks, are continuously executed until a `shutdown()` is called. This is quintessential to ensure that your node actually does its job!Finally, we destroy the node and manually call shutdown.

1.2 Add dependencies

In the base root folder of this package, you will find the `package.xml` which is important for declaring all dependancies of the package. We will now edit this file to ensure our code runs properly.

The `description`, `maintainer` and `license` tage should be appropriately filled out. For license, use any valid open source license like `Apache License 2.0`.

The buildtool we use by default is `ament_python` and you can see that this has already been assigned when we used the `ros2 pkg create` command. Below this, add the following two lines :

```
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

We already know from section **1.1** what these dependancies are. We just need to declare that these two libraries need to be included during execution time. (Buildtime dependencies are not required for Python)

1.3 Declaring the executable

Now that we have our code written and dependencies setup, we need to tell our build system that the script we created should be treated as an executable. We do this in `setup.py`.

Here, edit the `maintainer`, `maintainer_email`, `description` and `license` fields to assign exactly the same values as you did in `package.xml`.

Next, look for the section that starts with `entry_points={`. We edit this part to declare the executable and its entry point to look like this:

```
entry_points={
        'console_scripts': [
                'talker = wshop_nodes.minimal_publisher:main',
        ],
},
```

In this case, `talker` is the name we assign to the executable, `wshop_nodes` is the package, `minimal_publisher` is the name of the python file and `main` is the entry point to this executable (i.e. main function). Replace with the names you chose accordingly.

You can use the same prototype to declare executables in all ROS 2 python packages.

1.4 Setup.cfg

The final configuration file is `setup.cfg`, which, fortunately for us, is already configured properly and needs no more changes! These settings indicate to ROS 2 where the executable shall be put for discovery after building the package.

1.2.3 2 Subscriber Node

Similar to the publisher, we will now create a subscriber. Next to the publisher file, create another python file and paste in the code below, which will be explained next.

This document refers to this file as `minimal_subscriber.py`.

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String


class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)
```

(continues on next page)

(continued from previous page)

```
def main(args=None):
    rclpy.init(args=args)

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

2.1 Explanation

The subscriber code has many similarities to the publisher code, and in this section we review what differs.

The part that is immediately relevant is creating the subscriber

```
self.subscription = self.create_subscription(
    String,
    'topic',
    self.listener_callback,
    10)
```

The first parameter to pass to the function is the msg type, the second is the name of the topic - this should be the same as declared in the publisher, the third is the callback function for the subscriber and the last is the message buffer size.

The nxt part to understand is the callback function.

```
def listener_callback(self, msg):
    self.get_logger().info('I heard: "%s"' % msg.data)
```

The parameter that is automatically passed to this dunction is the incoming message. In this case, it is simply printed to console.

2.2 Declaring the executable

We need to declare a new executable for this subscriber node. We add another line to `setup.py` similarly as before and it would look like this:

```
entry_points={
        'console_scripts': [
                'talker = wshop_nodes.minimal_publisher:main',
                'listener = wshop_nodes.minimal_subscriber:main'
```

(continues on next page)

(continued from previous page)

```
        ],
},
```

1.2.4 3 Build and run

Before building, it is always good to check if all dependencies have been installed. We execute the following from the **base workspace folder** (i.e. just above the src folder of your **workspace**):

`rosdep install --from-paths src --ignore-src -r --rosdistro <distro> -y`Substitute with the current version of ROS 2 you are running on. Ex: `foxy`

From the same location, build the workspace:

`colcon build --symlink-install`

Now we need to source this workspace in order to be able to discover the executable that we just built:

`source install/local_setup.bash`

Finally, we are ready to run an executable. Recalling from section **1.3**, the name we assigned to the executable with the publisher is `talker`. So we run this:

`ros2 run wshop_nodes talker`

Open another terminal and similarly source ROS 2 and this workspace to run the subscriber executable:

`ros2 run wshop_nodes listener`

1.2.5 4 Composed nodes

We will now create a third executable, that demonstrates the node composition feature using executors. It will be a single Python script that implements two nodes - the same publisher and subscriber as above, but composes them with a single executor.

Create yet another python file, which for reference here is named as `composed_nodes.py` and paste the following:

```
import rclpy

from wshop_nodes.minimal_publisher import MinimalPublisher
from wshop_nodes.minimal_subscriber import MinimalSubscriber

from rclpy.executors import SingleThreadedExecutor


def main(args=None):
    rclpy.init(args=args)
    try:
        minimal_publisher = MinimalPublisher()
        minimal_subscriber = MinimalSubscriber()

        executor = SingleThreadedExecutor()
        executor.add_node(minimal_publisher)
        executor.add_node(minimal_subscriber)

        try:
```

(continues on next page)

(continued from previous page)

```
            executor.spin()
        finally:
            executor.shutdown()
            minimal_publisher.destroy_node()
            minimal_subscriber.destroy_node()

    finally:
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

4.1 Explanation

```
from wshop_nodes.minimal_publisher import MinimalPublisher
from wshop_nodes.minimal_subscriber import MinimalSubscriber

from rclpy.executors import SingleThreadedExecutor
```

We import the previous two node classes that we created into this executable.We also import the single threaded executor that we will be using to compose the nodes.

```
minimal_publisher = MinimalPublisher()
minimal_subscriber = MinimalSubscriber()
```

Similarly as before, we create node objects from the two node classes.

```
executor = SingleThreadedExecutor()
executor.add_node(minimal_publisher)
executor.add_node(minimal_subscriber)
```

This is the new part, where we create an executor object, and add our two nodes into it.

```
try:
    executor.spin()
finally:
    executor.shutdown()
    minimal_publisher.destroy_node()
    minimal_subscriber.destroy_node()
```

Instead of spinning the main executable directly, we instead spin the executor that contains our two nodes.

In this case, we have added a publisher and a subscriber within the same executor to simply demonstrate how multiple nodes can be added. In a practical scenario, this may not make much sense, and requires separate executors. This, however, is a more advanced topic and out of scope for this workshop.

4.2 Declaring the executable

Similarly as the previous two cases, we need to declare a third executor that points to the code we just created, and the result would look like this:

```
    entry_points={
        'console_scripts': [
                'talker = wshop_nodes.minimal_publisher:main',
                'listener = wshop_nodes.minimal_subscriber:main',
        'composed = wshop_nodes.composed_nodes:main'
        ],
    },
```

4.3 Build and run

Build the workspace and run this new executable, whose name in this case is `composed`.You will observe that indeed both the publisher and subscriber are running from within the same executable.

1.3 Understanding ROS 2 with Turtlesim

1.3.1 1. Introduction

Turtlesim is the Flagship example application for ROS and ROS 2. It demonstrates in simple but effective ways the basic concepts.

This workshop encourages you to refer to the cheat sheet for the syntax and type the commands on your own in order to learn by trial and error. Make use of the `--help` option for commands as well. However, solutions are also provided in the end. Feel free to approach this any way as you wish.

1.3.2 2. Starting the Turtle simulator

You can start the main application by simply executing two of its nodes. Refer to the cheat sheet for the syntax to execute a node.

The package name you need in this case is `turtlesim` and the nodes you need to start are `turtlesim_node` and `turtle_teleop_key`. Make sure to source ROS 2 and run these nodes in two separate terminals.

> Start the 2 nodes of the application

Once you successfully start these nodes, you should see a window popup with a blue background and a random turtle in the middle (this is an RQT pane, if you are interested in knowing!). This little guy is going to help us understand ROS 2. We should treat it as if it is an AGV (Automated Guided Vehicle), and we observe this scene from a top down perspective.

Before moving the turtle, you might find it useful to right click on the title bar of the simulator screen and select *Always on top* (undo this when you no longer need it).

> Keep the simulator window on top

Next, make sure your currently active window is the terminal where you started the teleop node by clicking once on it.

> Activate the teleop terminal

Now, as the instructions on the terminal say, you can move the turtle around by pressing the arrow keys on the keyboard, or use the other keys listed to set absolute orientations.

Move the turtle with the arrow keys on the keyboard

Once you are satisfied with playing with the turtle, you can go ahead to the next step.

1.3.3 3. Observing

You can start introspecting at this point and already see many interesting things. Please spend a few minutes trying to list the nodes, parameters, topics, services and actions. Refer to the cheat sheet for the CLI syntax for the list/info/show/echo commands.

List all available entities

Try to get information about each of them as well as the associated types (msg and srv).

Get more information on these entities

You can find out which of the topics are publishers, and listen in on what these topics are currently publishing.

Echo the topics

Perform all of the above introspection activities using RQT wherever applicable. Refer to the cheat sheet to see what plugins are available, or simply explore the RQT `Plugins` menu!

Use RQT for introspection

NOTE

You might notice some extra topics with the word `action` in the name when you use RQT Node Graph, which you do not see when you list the topics in CLI. You can safely ignore these for now, as they will be addressed in a later lecture.

3.1 Description of topics

The topic `/turtle1/color_sensor` tells you the RGB values of the color of the trail left behind when the turtle moves.The topic `/turtle1/cmd_vel` is used to instruct the turtle to move. Echo this topic in a separate terminal and then use the teleop node to move the turtle to observe how velocity commands are given to it.The topic `/turtle1/pose` publishes the current pose of the turtle. Echo this topic to see how the pose of the turtle changes as you move it.

3.2 Description of services

The services `/kill` and `/spawn` are used to kill and spawn turtles respectively.The service `/clear` clears the background of trail lines and `/reset` resets the position of the turtle.The service `/turtle1/set_pen` sets the color and thickness of the trail line.The services `/turtle1/teleport_xx` move the turtle instantly.

You can ignore the services with the word `parameter` in the names for now.

1.3.4 4. Interacting

In this step you will practice interacting with the topics, services, actions and parameters provided by the turtlesim.

4.1 Services

Call the service `/turtle1/set_pen` from the CLI first by referring to the cheat sheet for the syntax. For selecting the right type, use double tab after typing in the name of the service to retrieve a list of options for the type and use the most appropriate. (Hint: Service types DO NOT have – in their names, and have their names in/hierarchical/format).

In order to find out the right dictionary format to use, inspect the service type from another terminal (Hint: `ros2 interface show` and `ros2 interface proto`). The key values take an integer in the range 0-255. Once you call the service correctly, move the turtle with teleop to observe the change in trail color.

> Set the pen color of the turtle, and visually verify the change

You can echo `/turtle1/color_sensor` to verify if this is the same RGB value that you set.

> Check the topic to cross verify

You can try to call the same service from RQT as well. Refer to the cheat sheet to select the right plugin.

> Use RQT to perform the same service call

You can similarly call the other services as well and observe what changes they make.

> Invoke all the other services as well using CLI and/or RQT.

4.2 Parameters

The parameter commands are fairly straightforward. Try to list the params available and see what values they have. Change the background color of the sim by setting one or more of the relevant params. The allowed range of values for this is 0 - 255.

> Change background color of the same by setting a parameter

4.3 Actions

Next, you can invoke the action provided by this sim. List the actions available, find out the interface type, and then invoke the action with meaningful values.

In this case, the action server can be invoked by a command line client using the `ros2 action send_goal <goal>` command. The syntax is similar to calling a service. The input parameter is *theta* which is an angle measured in radians, i.e. The valid range is -3.14 < *theta* < 3.14.

Using the `--feedback` option with the command prints the feedback to the console. The result of the action is always displayed.

> Invoke an action call from the terminal

4.4 Topics

Finally, you can publish a velocity command on `/turtle1/cmd_vel` from CLI and RQT in a similar manner as calling a service, but of course, using the right commands for topics/messages instead. Make sure the teleop node is shut down before attempting this.

(Hint: The msg type is composite in this case, using *Vector3*. Each *Vector3* type has 3 fields: `x, y, y`. The sub-fields can be accessed with `:` Ex- `linear:x:0.5`. Play around with spaces until the command works. The solution is provided in the end.)

> Control the turtle from a terminal publisher

1.3.5 5. Using ros2bags

In a separate terminal start recording a ros2bag file with the selected topic `/turtle1/cmd_vel`. Back in your keyboard teleop terminal, give some velocity commands to make the turtle move. Go back to the ros2bag terminal and hit *ctrl+c* to kill it and stop recording.

Replay this ros2bag file, and you will notice the turtle moving in the same way as you recorded.

1.3.6 6. Advanced - Remapping and other options

Every ROS 2 command and sub-command has a list of options that you can use to modify its behavior. The list of options can be seen with `-h` and included as desired. You can experiment with these as well and see how the commands you have already executed so far change.

For example, when you publish a velocity command from CLI, it publishes this message continuously and the turtle keeps moving until you kill the publisher. You could instead give a `-r 0.5` option to make it publish at 0.5 Hz. meaning there is a slight pause before every movement.

You could try adding the same rate option command to the spawn service (make sure to remove the name field as this would otherwise cause a name clash error). You will observe that turtles keep spawning until you kill the service caller.

1.3.7 7. Solutions

7.1 Starting the Turtle simulator

```
ros2 run turtlesim turtlesim_noderos2 run turtlesim turtle_teleop_key
```

7.2 Observing

```
ros2 node listros2 topic list -tros2 topic info /turtle1/cmd_velros2 interface show
turtlesim/msg/Poseros2 service listros2 interface show turtlesim/srv/Spawnros2 interface
proto turtlesim/srv/Spawn
```

7.3 Interacting

```
ros2 service call /spawn turtlesim/srv/Spawn "{x:  5,y:  5,theta:  0}"ros2 service call
/reset std_srvs/srv/Empty
```

```
ros2 topic pub /turtle1/cmd_vel geometry_msgs/msg/Twist "linear:
  x: 0.5
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.5"
```

```
ros2 topic pub -r 0.5 /turtle1/cmd_vel geometry_msgs/msg/Twist "linear:
  x: 0.5
  y: 0.0
  z: 0.0
```

(continues on next page)

(continued from previous page)

```
angular:
  x: 0.0
  y: 0.0
  z: 0.5"
```

```
ros2 service call -r 0.5 /spawn turtlesim/srv/Spawn "{x:  5,y:  5,theta:  0}"
```

```
ros2 service call /turtle1/set_pen turtlesim/srv/SetPen "r:  100 g:  0 b:  0 width:  0
'off':  0"
```

```
ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute {'theta:
-1.57'} --feedback
```

```
ros2 param set /turtlesim background_r 125
```

7.4 Using ros2bags

```
ros2 bag record /turtle1/pose -o velocities
```

CHAPTER

TWO

SESSION 2 - ROS 2 NAVIGATION

2.1 TF2 - The second generation of the transform library

2.1.1 Introduction

This tutorial will help you understand the flexibility of transforms with the TF2 library. It will introduce Dynamic Transform Broadcasters, Transform Listeners and Static Transform Broadcasters.

For this tutorial, we will use the Turtlesim again. This time, we will insert a second turtle into the simulation. As you drive the first turtle around with keyboard teleop, the second turte will follow it closely. In order to do this, the second turtle needs to know where the first one is, w.r.t. its own coordinate frames. This can be easily achieved using the TF2 library.

The example code is based on tf2_example and is tested with ROS 2 Foxy.

2.1.2 1. Dynamic TF Broadcaster

The word dynamic indicates that the transform that is being published is constantly changing. This is useful for tracking moving parts. In this case, we continuously broadcast the current position of the first turtle.

Create an ament python package with dependencies on tf2_ros (Use the "depend" tag).

```
ros2 pkg create --build-type ament_python <package_name> --dependencies tf2_ros rclpy
```

Create a new python script in this package (under the folder with the package name) and register it as an executable in `setup.py`. For the purpose of this document, the package name is assumed as `tf2_workshop` and executable name as `broadcaster`.

Make sure the scipy library is installed:

```
pip3 install scipy
```

Copy the code shown below into the new file, save it, and build it.

```
#! /usr/bin/env python3

import rclpy
import sys

from geometry_msgs.msg import TransformStamped
from rclpy.node import Node
from scipy.spatial.transform import Rotation as R
```

(continues on next page)

(continued from previous page)

```
from tf2_ros.transform_broadcaster import TransformBroadcaster
from turtlesim.msg import Pose

class DynamicBroadcaster(Node):

    def __init__(self, turtle_name):
        super().__init__('dynamic_broadcaster')
        self.name_ = turtle_name
        self.get_logger().info("Broadcasting pose of : {}".format(self.name_))
        self.tfb_ = TransformBroadcaster(self)
        self.sub_pose = self.create_subscription(Pose, "{}/pose".format(self.name_),␣
↪self.handle_pose, 10)

    def handle_pose(self, msg):

        tfs = TransformStamped()
        tfs.header.stamp = self.get_clock().now().to_msg()
        tfs.header.frame_id="world"
        tfs._child_frame_id = self.name_
        tfs.transform.translation.x = msg.x
        tfs.transform.translation.y = msg.y
        tfs.transform.translation.z = 0.0

        r = R.from_euler('xyz',[0,0,msg.theta])

        tfs.transform.rotation.x = r.as_quat()[0]
        tfs.transform.rotation.y = r.as_quat()[1]
        tfs.transform.rotation.z = r.as_quat()[2]
        tfs.transform.rotation.w = r.as_quat()[3]

        self.tfb_.sendTransform(tfs)

def main(argv=sys.argv[1]):
    rclpy.init(args=argv)
    node = DynamicBroadcaster(sys.argv[1])

    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass

    node.destroy_node()
    rclpy.shutdown()


if __name__ == "__main__":
    main()
```

1.1 Explanation

This follows the same basic structure for a ROS 2 node. The TF2 specific lines will be explained.

```
from geometry_msgs.msg import TransformStamped
from scipy.spatial.transform import Rotation as R
from tf2_ros.transform_broadcaster import TransformBroadcaster
```

Imports the modules required for TF2. Scipy is used to convert from Euler angles to quaternion since the `tf_conversions` package has not been ported over to ROS 2 yet.

```
self.tfb_ = TransformBroadcaster(self)
```

Initializes a dynamic transform broadcaster which sends the transforms.

```
self.name_ = turtle_name
```

The name of the turtle for which we are broadcasting comes from the CLI as an argument.

```
tfs = TransformStamped()
...
self.tfb_.sendTransform(tfs)
```

Create a transform datatype with a header and populate it with meaningful data. The parent frame is always "world" and the child frame (i.e. current frame being published) is the name chosen. Transmit this transformation from within the subscriber callback. This implies it updates the frame (nearly) as fast as it receives the pose.

- Header
  - Timestamp : (Determine the moment when this transform is happening. This is mainly rclpy.Time.time() when you want to send the actual transform. This means the transform can change over time to generate a dynamic motion.)
  - Frame_ID : (The frame ID of the origin frame)
- Child Frame ID: (Frame ID to which the transform is happening)
- Transform
  - Position : in meter (X, Y and Z)
  - Orientation : in Quaternion (You can use the TF Quaternion from Euler function to use the roll, pitch and yaw angles in rad instead)

1.2 Declaring the executable

Now that we have our code written and dependencies setup, we need to tell our build system that the script we created should be treated as an executable. We do this in `setup.py`. Look for the section that starts with `entry_points={`. We edit this part to declare the executable and its entry point to look like this:

```
entry_points={
        'console_scripts': [
                'broadcaster = tf2_workshop.broadcaster:main',
        ],
},
```

1.3 Testing the Broadcaster

First, start the turtlesim node :

```
ros2 run turtlesim turtlesim_node
```

Then start the broadcaster, with your chosen name for the turtle as the only argument. Here we assume `turtle1`:

```
ros2 run tf2_workshop broadcaster turtle1
```

If all works well, the broadcaster is now sending the TF data for turtle1. This can be verified with:

```
ros2 run tf2_ros tf2_echo turtle1 world
```

or by simply running:

```
ros2 run tf2_ros tf2_monitor
```

or also through rviz2 by adding the `TF` display module.

2.1.3 2. TF Listener

Once your robot system has a fully fleshed out TF tree with valid data at a good frequency, you can then make use of these transforms for your application through listeners, which actually solve inverse kinematics for you. This is the true power of TF2.

In the following example, we will create a TF listener, which listens to the TF tree to get the transformation between the first and the second turtle.

Create another new python file for the listener and add it as an executable. This document assumes the name `listener` for it.

Copy the following code into the file and save it:

```
#!/usr/bin/env python3

import sys
import math

from geometry_msgs.msg import Twist

import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile
from tf2_ros.transform_listener import TransformListener
from tf2_ros.buffer import Buffer
from tf2_ros import LookupException

class TfListener(Node):

    def __init__(self, first_turtle, second_turtle):
        super().__init__('tf_listener')
        self.first_name_ = first_turtle
        self.second_name_ = second_turtle
        self.get_logger().info("Transforming from {} to {}".format(self.second_name_,␣
↪self.first_name_))
```

(continues on next page)

(continued from previous page)

```
        self._tf_buffer = Buffer()
        self._tf_listener = TransformListener(self._tf_buffer, self)
        self.cmd_ = Twist ()
        self.publisher_ = self.create_publisher(Twist, "{}/cmd_vel".format(self.second_
↪name_),10)
        self.timer = self.create_timer(0.33, self.timer_callback) #30 Hz = 0.333s

    def timer_callback(self):
        try:
            trans = self._tf_buffer.lookup_transform(self.second_name_, self.first_name_,
↪ rclpy.time.Time())
            self.cmd_.linear.x = math.sqrt(trans.transform.translation.x ** 2 + trans.
↪transform.translation.y ** 2)
            self.cmd_.angular.z = 4 * math.atan2(trans.transform.translation.y , trans.
↪transform.translation.x)
            self.publisher_.publish(self.cmd_)

        except LookupException as e:
            self.get_logger().error('failed to get transform {} \n'.format(repr(e)))

def main(argv=sys.argv):
    rclpy.init(args=argv)
    node = TfListener(sys.argv[1], sys.argv[2])
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

if __name__ == "__main__":
    main()
```

2.1 Explanation

The new lines are explained here.

```
from tf2_ros.transform_listener import TransformListener
from tf2_ros.buffer import Buffer
from tf2_ros import LookupException
```

These modules are required for listeners. Recall that the buffer provides an abstracted API of the core TF2 package which is ROS agnostic.

```
self._tf_buffer = Buffer()
self._tf_listener = TransformListener(self._tf_buffer, self)
```

Boilerplate code for setting up a new listener.

```
trans = self._tf_buffer.lookup_transform(self.second_name_, self.first_name_, rclpy.time.
↪Time())
```

The main workhorse of this node. The syntax accepts the target frame first and the source frame second. In other words, the pose of the origin of the source frame w.r.t the target frame. It calculates the transformation from the first turtle to the second turtle, which tells us how far away turtle1 is from turtle2.

```
self.cmd_.linear.x = math.sqrt(trans.transform.translation.x ** 2 + trans.transform.
↪translation.y ** 2)
self.cmd_.angular.z = 4 * math.atan2(trans.transform.translation.y , trans.transform.
↪translation.x)
self.publisher_.publish(self.cmd_)
```

This part is a very simply controller, that generates velocity commands based on the distance remaining. This part can be replaced by any other controller you wish to try for the following motion.

Make sure to add the listener as an executable as well (refer to the previous section) and then rebuild the package.

2.2 Testing the listener

The turtlesim simulation and teleop node need to be running first.

To test the application, you first need to spawn a second turtle, the name of which is assumed here to be `turtle2`:

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: "turtle2"}"
```

The parameters indicate the starting pose and name.

You also need to start up another TF broadcaster for this new turtle, similar to the previous one, but with the name of this turtle instead.

Finally, run the executable node for the listener by passing the first argument as the name of the first turtle and the second argument as the name of the second turtle:

```
ros2 run tf2_workshop listener turtle1 turtle2
```

Now when you move around the first turtle using the keyboard, the second turtle follows it.

You can once again check the TF tree to observe the new additions.

2.1.4 3. Static Transform Broadcaster

The dynamic transform broadcaster is used for moving objects. But for parts that do not move, the simpler static broadcaster does the job well. This could be used for instances like a camera fixed in a room, or a lidar mounted on a mobile robot.

The static publisher is already available as an executable node, and it can simply be started with the right command line arguments which are in order :

```
Translation.x, Translation.y, Translation.z, Rotation.yaw, Rotation.pitch, Rotation.roll,
↪ Parent frame, child frame
```

For example :

```
ros2 run tf2_ros static_transform_publisher 0.1 0 0 -1.57 0.0 0.0 turtle1 turtle_cam1
```

This will add the frame `turtle_cam1` related to the `turtle1` frame. The virtual camera is mounted +0.1 m in x-axis from view of the turtle base and rotated -1.57 rad in yaw. This might not make much real world sense, but hey it's just an example!

3.1 Testing the static broadcaster

The TF published is exactly of the same format as the dynamic publisher, and this can be easily verified with any of the aforementioned means.

2.2 TurtleBot in ROS 2

2.2.1 1. Introduction

- The goal for this tutorial:
  - Simulate TurtleBot in gazebo
  - Get ideas about how to control physical/simulated TurtleBot
  - Control Turtlebot from keyboard
- The packages that you need for this tutorial:
  - turtlebot3_gazebo
  - turtlebot3_teleop
  - turtlebot3_bringup(on TurtleBot)
- Lines beginning with `$` indicates the syntax of these commands. Commands are executed in a terminal:
  - Open a new terminal → use the shortcut ctrl+alt+t. Open a new tab inside an existing terminal → use the shortcut ctrl+shift+t.
- Info: The computer of the real robot will be accessed from your local computer remotely. For every further command, a tag will inform which computer has to be used. It can be either `[TurtleBot]` or `[Remote PC]`.

Please try simulation first and use keyboard to control it.

2.2.2 2. Preparation

2.1. Check packages

Before start, check if there are turtlrbot3* packages

```
Source ROS workspace first

$ source /opt/ros/foxy/setup.bash
```

```
$ ros2 pkg list | grep turtlebot3
```

If you don't have turtlebot3 packages, you can install debian packages or from source code.

A. Install debian packages

```
sudo apt install ros-foxy-turtlebot3*
```

B. Install from source code

- Step 1: Download turtlebot3.repos

```
First entering your workspace
(If you don't have workspace yet, you need to create one with an src folder in it)

$ wget https://raw.githubusercontent.com/ipa-rwu/\
turtlebot3/foxy-devel/turtlebot3.repos
```

- Step 2: Using vcstools get packages

  Make sure you have "src" folder in you workspace, then run this command to get source code for turtlebot3.

```
$ vcs import src<turtlebot3.repos
```

  If you didn't install vcstools, you can install it as followed:

```
sudo sh -c 'echo \
"deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main"\
> /etc/apt/sources.list.d/ros-latest.list'
```

```
sudo apt-key adv --keyserver hkp://pool.sks-keyservers.net \
--recv-key 0xAB17C654
```

```
sudo apt update && sudo apt install python3-vcstool
```

- Step 3: Using rosdep get dependencies

```
$ rosdep update
$ rosdep install --from-paths src --ignore-src  -y
```

- Step 4: Install packages

```
$ colcon build --symlink-install
```

- Step 5: Source your workspace

```
$ source install/setup.bash
```

2.2.3 3. Simulation

In this chapter you will learn how to simulate TurtleBot in gazebo

1. Open a terminal

   If you don't set up ROS Domain ID, then the default ROS_DOMAIN_ID=0.

   In this case, we only work with one turtlebot so we can use default ROS Domain ID.

   If you want to use different ROS Domain ID, you can perform:

```
$ export ROS_DOMAIN_ID=11
```

2. Set up ROS environment arguments

   If you use debian packages,

```
$ source  /opt/ros/foxy/setup.bash
```

If you use packages in your workspace:

```
First entering your workspace

$ source  install/setup.bash
```

3. Set up turtlebot model

```
$ export TURTLEBOT3_MODEL=burger
```

4. Set up Gazebo model path

```
$ export GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:`ros2 pkg \
prefix turtlebot3_gazebo \
`/share/turtlebot3_gazebo/models/
```

5. Launch Gazebo with simulation world

```
$ ros2 launch turtlebot3_gazebo empty_world.launch.py
```

You also can start different world by replacing `empty_world.launch.py` with `turtlebot3_house.launch.py`

You can check ros topics and ros graph.s

2.2.4 4. Control the robot

In this chapter you will learn how to use keyboard or joystick to control robot. In general we will start a ros node that will publish to topic **/cmd_vel**

4.1. Keyboard

1. Open a new terminal
   - Set up ROS environment arguments
   - (Set up ROS_DOMAIN_ID): Only if you set up ROS_DOMAIN_ID in chapter3
2. Set up turtlebot model

```
$ export TURTLEBOT3_MODEL=burger
```

3. Run a teleoperation node

```
$ ros2 run turtlebot3_teleop teleop_keyboard
```

If the program is successfully launched, the following output will appear in the terminal window and you can control the robot following the instruction.

```
Control Your TurtleBot3!
---------------------------
Moving around:
        w
   a    s    d
        x
```

(continues on next page)

(continued from previous page)

```
w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi : ~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi : ~ 1.82)

space key, s : force stop

CTRL-C to quit
```

4.2. PS3 Joystick

First of all, connect the given PS3 Joystick to the remote PC via the USB cable and install required packages for teleoperation using PS3 joystick.

```
$ sudo pip install ds4drv
```

```
$ sudo ds4drv
$ ros2 run joy joy_node
$ ros2 run teleop_twist_joy teleop_node
```

Button map PS3

Enable regular-speed movement button: L2 shoulder button

Enable high-speed movement: L1 shoulder button

You can use Joystick axis and joystick angular axis to control turtlbot.

2.2.5 5. Physical TurtleBot3

In this chapter you will learn how to use physical TurtleBot

5.1. Setting up to a turtlebot ROS 2 Network

Info: The computer of the real robot will be accessed from your local computer remotely. For every further command, a tag will inform which computer has to be used. It can be either `[TurtleBot]` or `[Remote PC]`.

As the robot you are using does not have any input devices or monitor, we have to start it in another way. Luckily we can work remotely from a local workstation using SSH. SSH provides a secure communication channel over an unsecured network in a client-server architecture. It connects an SSH client application with an SSH server.

To establish a connection to a computer remotely, the username and the IP address of the remote computer must be known. Additionally, both the computer must be connected to the same network. In our case, every TurtleBot has a <TB-user>, a <TB-password> and an <TB-IP> address that can be found at the robot. First of all, you should check if you can ping the computer of the TurtleBot. This ensures that both machines are in the same network.

`[Remote PC]`

```
$ ping <TB-IP>
```

You should see something like:

```
64 bytes from 192.168.1.75: icmp_seq=1 ttl=64 time=0.062 ms
64 bytes from 192.168.1.75: icmp_seq=2 ttl=64 time=0.056 ms
64 bytes from 192.168.1.75: icmp_seq=3 ttl=64 time=0.050 ms
64 bytes from 192.168.1.75: icmp_seq=4 ttl=64 time=0.050 ms
```

If this check has been successful one can connect remotely to the TurtleBot with the command:

`[Remote PC]`

```
$ ssh <TB-userName>@<TB-IP>
```

You will be asked to enter the password of the remote account, which is the one retrieved at the robot (TB-password). After the connection is established, the same terminal should show the following line:

```
$ <TB-userName>@<TB-IP>
```

This means you are working from the home directory of the TurtleBot. So you can start applications running on the processor of the robot. All commands that will have to be entered during this tutorial in this terminal, will be labelled with a [**TurtleBot**] tag.

Hint: You can disconnect the ssh connection by typing "exit" or "Ctrl + D" in the terminal.

5.2. Bring up the TurtleBot

Before running any ROS command, you need to source the workspace first. In this case, you will find environment variables are defined in the file $HOME/.bashrc. So you can run the command below to set up environment variables and source the workspace.

`[TurtleBot3]`

```
$ source .bashrc
```

Bring up the robot with the following launch-file:

[TurtleBot3]

```
$ ros2 launch turtlebot3_bringup robot.launch.py
```

Afterwards, you can again check the nodes within a terminal of the local machine to see which applications are running within the same ROS Network.

Hint: The ROS 2 DOMAIN ID of TurtleBot must be exported on every new terminal of the [Remote PC]!

[Remote PC]

```
$ export ROS_DOMAIN_ID = <ROS_DOMAIN_ID of TurtleBot>
```

[Remote PC]

```
$ ros2 node list
```

You can check all existing topics of the system:

[Remote PC]

```
$ ros2 topic list
```

You can as well check all existing service of the system:

```
$ ros2 service list
```

You can as well check tf:

```
$ ros2 run tf2_ros tf2_monitor
```

Then you also can use keyboard or joystick as you control TurtleBot in simulation to control real TurtleBot

2.3 ROS 2 Cartographer

2.3.1 1. Introduction

- The goal of this tutorial is to
  - use Cartographer to create a map of environment
- The packages that will be used:
  - cartographer
  - cartographer-ros
  - turtlebot3_cartographer
  - turtlebot3_teleop
  - turtlebot3_gazebo

This tutorial explains how to use the Cartographer for mapping and localization.

- Lines beginning with $ indicates the syntax of these commands. Commands are executed in a terminal:

  - Open a new terminal → use the shortcut ctrl+alt+t. Open a new tab inside an existing terminal → use the shortcut ctrl+shift+t.

- Info: The computer of the real robot will be accessed from your local computer remotely. For every further command, a tag will inform which computer has to be used. It can be either `[TurtleBot]` or `[Remote PC]`.

2.3.2 2. General approach

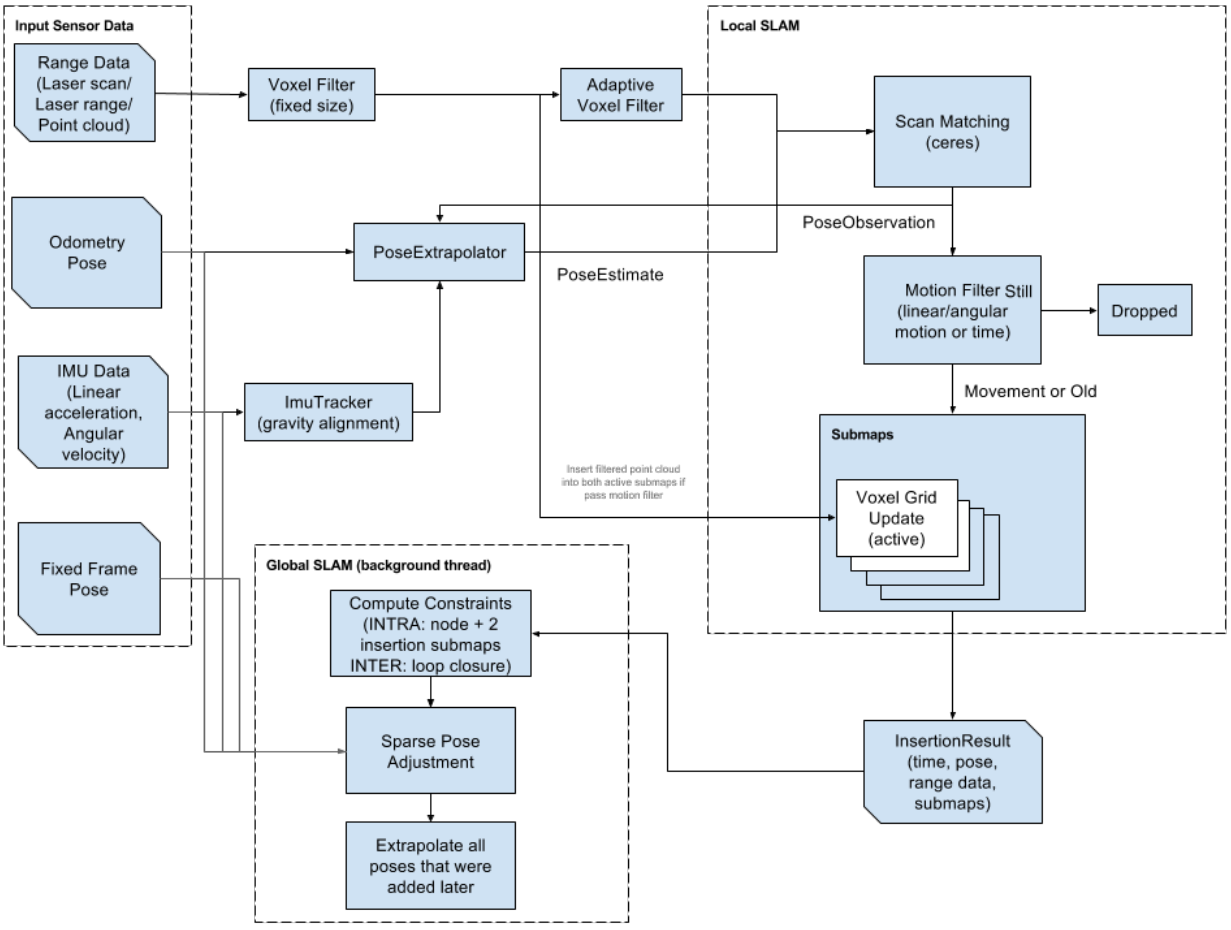
The main problem in mobile robotics is localization and mapping. To estimate the position of the robot in an environment, you need some kind of map from this environment to determine the actual position in this environment. On the other hand, you need the actual position of robots to create a map related to its position. Therefore you can use SLAM – Simultaneous Localization and Mapping. ROS provides different packages to solve this problem:

- **2D**: gmapping, hector_slam, cartographer, ohm_tsd_slam…
- **3D**: rgbdslam, ccny_rgbd, lsd_slam, rtabmap…

For ground-based robots, it is often sufficient to use 2D SLAM to navigate through the environment. In the following tutorial, cartographer will be used. Cartographer SLAM builds a map of the environment and simultaneously estimates the platform's 2D pose. The localization is based on a laser scan and the odometry of the robot.

2.3.3 3. Start Cartographer

3.1. Technical Overview



Technical Overview

Figure 1: Technical Overview

source: cartographer

3.2. Check packages

3.2.1. Check if there are cartographer packages

```
```bash
# source ROS 2
$ source /opt/ros/foxy/setup.bash

$ ros2 pkg list |grep cartographer

# You will get
# cartographer_ros
# cartographer_ros_msgs
```
```

If you don't have "cartographer_ros" and "cartographer_ros_msgs", you can install cartographer by performing the following:

Before installing package, you need to make sure which ROS distribution you are using.

```
$ sudo apt install ros-$ROS_DISTRO-cartographer
```

3.2.2. Check if there are turtlebot3* packages

```
$ ros2 pkg list | grep turtlebot3
```

If you don't have turtlebot3 packages, you can install debian packages or from source code.

A. Install debian packages

```
sudo apt install ros-foxy-turtlebot3*
```

B. Install from source code

First entering your workspace

```
$ wget https://raw.githubusercontent.com/ROBOTIS-GIT/turtlebot3/foxy-devel/turtlebot3.
↪repos
```

Make sure you have "src" folder, then run this command to get source code for turtlebot3

```
$ vcs import src<turtlebot3.repos
```

Source your ROS 2 installation workspace and install dependencies

```
$ source /opt/ros/foxy/setup.bash
$ rosdep update
$ rosdep install --from-paths src --ignore-src --rosdistro
$ROS_DISTRO -y
```

Compile codes

```
$ colcon build
```

Source your workspace

```
$ source install/setup.bash
```

3.3. Startup system of turtleBot and teleoperation

3.3.1. Simulation in gazebo

1. Set up turtlebot model

   ```
   $ export TURTLEBOT3_MODEL=burger
   ```

2. Set up Gazebo model path

```
$ export GAZEBO_MODEL_PATH=`ros2 pkg \
prefix turtlebot3_gazebo`/share/turtlebot3_gazebo/models/
```

3. Launch Gazebo with a simulation world

```
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

3.3.2. Physical robot

[TurtleBot3]

a. open a terminal and use `ssh` connect to Turtlebot3.

b. bring up basic packages to start its applications.

```
$ source .bashrc

$ cd turtlebot3_ws

#Set up ROS_DOMAIN_ID
$ export ROS_DOMAIN_ID="Your Number"
# e.g. export ROS_DOMAIN_ID=11

$ source install/setup.bash

$ ros2 launch turtlebot3_bringup robot.launch.py
```

3.3.3. Run teleoperation node

[Remote PC]

1. Open a new terminal
2. Set up ROS environment

```
$ source /opt/ros/foxy/setup.bash
```

3. (Set up ROS_DOMAIN_ID)

   If you set up ROS_DOMAIN_ID for running turtlebot simulation or physical turtlebot, then you need to set the same ROS_DOMAIN_ID here.

```
$ export ROS_DOMAIN_ID="Your Number"
# e.g. export ROS_DOMAIN_ID=11
```

4. Set up turtlebot model

```
$ export TURTLEBOT3_MODEL=burger
```

5. Run teleoperation node

```
$ ros2 run turtlebot3_teleop teleop_keyboard
```

3.4. Run SLAM nodes

1. open a new terminal on Remote PC
2. Set up ROS environment

```
$ source /opt/ros/foxy/setup.bash
```

3. (Set up ROS_DOMAIN_ID)

   If you set up ROS_DOMAIN_ID for running turtlebot simulation or physical turtlebot, then you need to set the same ROS_DOMAIN_ID here.

   [Remote PC]

```
$ export ROS_DOMAIN_ID="Your Number"
# e.g. export ROS_DOMAIN_ID=11
```

4. run the SLAM nodes

   If you are using simulation, you need to use simulation time. You can set `use_sim_time` to True.

   a. Simulation

   [Remote PC]

```
$ ros2 launch turtlebot3_cartographer \
cartographer.launch.py \
use_sim_time:=True
```

   b. For a real robot

   [Remote PC]

```
$ ros2 run cartographer_ros occupancy_grid_node -resolution 0.05\
-publish_period_sec 1.0

$ ros2 run cartographer_ros cartographer_node\
-configuration_directory]\
install/turtlebot3_cartographer/share/turtlebot3_cartographer\
/config -configuration_basename turtlebot3_lds_2d.lua
```

5. create a map

   Hint: Make sure that the Fixed Frame (in Global Options) in RViz is set to "map".

   In this way the map is fixed and the robot will move relative to it. The scanner of the Turtlebot3 covers 360 degrees of its surroundings. Thus, if objects are close by to the robot it will start to generate the map.

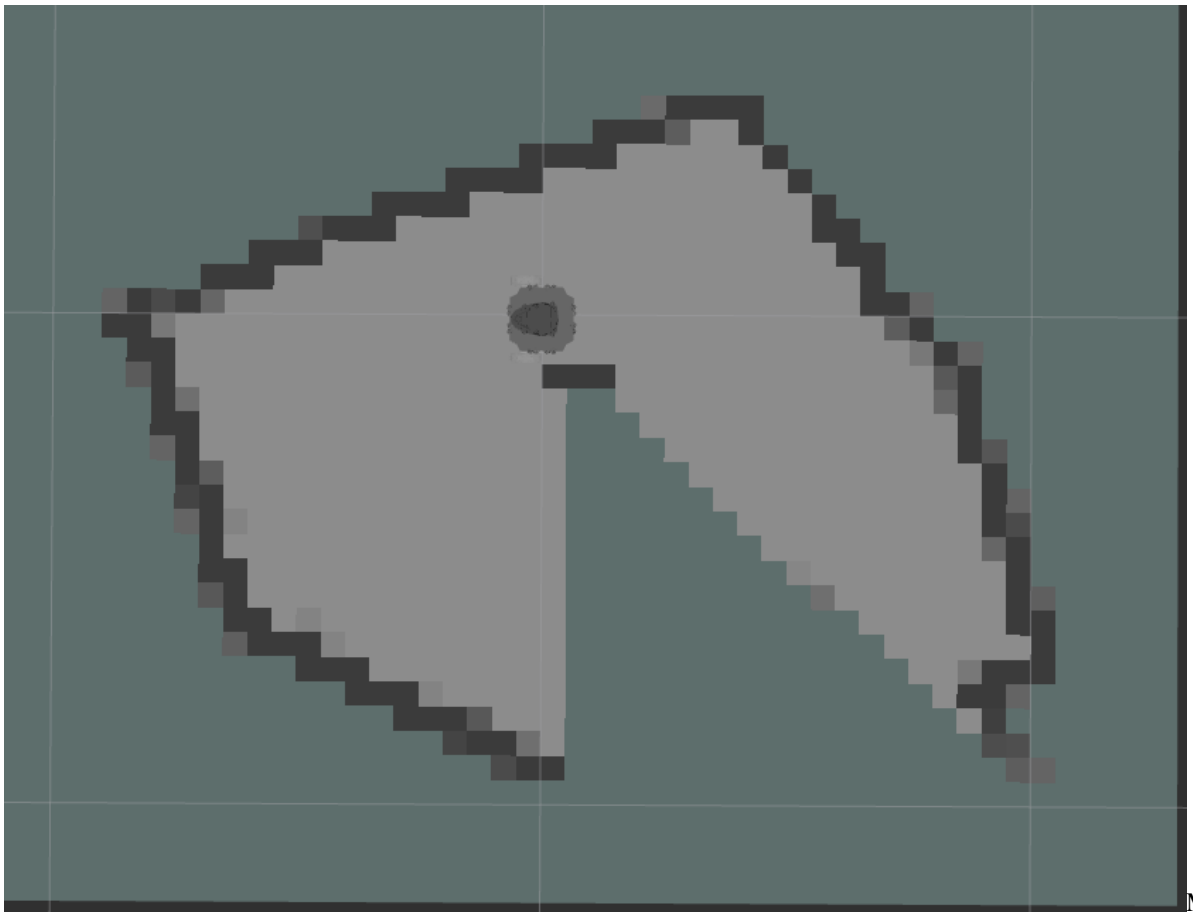
   Teleoperate the robot through the physical world until the enclosed environment is completely covered in the virtual map.

   The following hints help you to create **a nice map**:

```
* Try to drive as slow as possible
* Avoid to drive linear and rotate at the same time
* Do not drive too close to the obstacles
```

   In the left menu of RViz you can see several display modules. There is e.g. the RobotModel which is virtual visualization of the robot.

Furthermore, you can visualize the transforms of the available frames by checking the box of tf. Make yourself familiar with the available modules.



Map at startup Figure 2: A incomplete map at beginning in the real work setup

 Final Map Figure 3: A complete map at beginning in the real work setup

2.3.4 4. Save the Map

If you are satisfied with your map you can store it. You can save the map.

[Remote PC]

1. Open a new terminal and enter your workspace
2. (Set up ROS Domain ID) Only if you did set up ROS Domain ID before, you need to set up ROS Domain ID here.
3. run the map saver node.

```
$ ros2 run nav2_map_server map_saver_cli
```

You also can define a name of the map by

```
ros2 run nav2_map_server map_saver_cli -f my_map
```

If the terminal's path is "your workspace" they can be found in "your workspace" directory. An example of the map.pgm image is given in the following.

 Map

The node will create a map.pgm and a map.yaml files in the current directory, which is your workspace directory in this case.

Hint: The signs "~/" is a direct path to the home directory which works from every relative path.

2.4 ROS 2 Navigation

2.4.1 1. Introduction

- The goal of this tutorial is
  - to use the ROS 2 navigation capabilities to move the robot autonomously.
- The packages you will use:
  - workshop_ros2_navigation

Lines beginning with # indicates the syntax of these commands.

Commands are executed in a terminal:

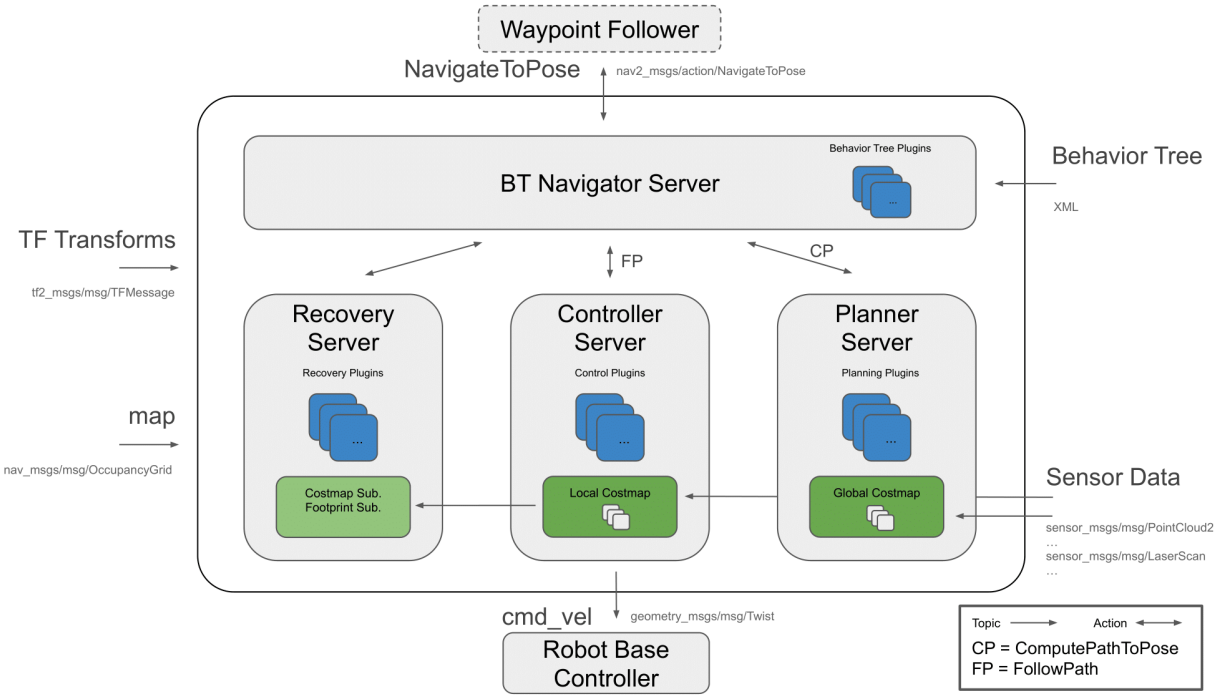
- Open a new terminal → use the shortcut ctrl+alt+t.
- Open a new tab inside an existing terminal → use the shortcut ctrl+shift+t.
- Lines beginning with $ indicates the syntax of these commands.
- The computer of the real robot will be accessed from your local computer remotely. For every further command, a tag will inform which computer has to be used. It can be either `[TurtleBot]` or `[Remote PC]`.

You can use the given links for further information.

2.4.2 2. General Approach

The ROS 2 Navigation System is the control system that enables a robot to autonomously reach a goal state, such as a specific position and orientation relative to a specific map. Given a current pose, a map, and a goal, such as a destination pose, the navigation system generates a plan to reach the goal, and outputs commands to autonomously drive the robot, respecting any safety constraints and avoiding obstacles encountered along the way.

It consists of several ROS components. An overview of its interactions is depicted in the following picture:



navigation_overvi

Figure 1: Navigation2 Architecture

source: navigataion

2.4.3 3. Launch the navigation stack

3.1. Check the map

1. Check the location of your map

   Once you create a map, you will have two files: “name of your map”.pgm and “name of your map”.yaml

   For example, a workspace has the following layout:

```
nav_ws/
  maps/
    my_map.yaml
    my_map.pgm
  src/
  build/
  install/
  log/
```

2. Check if the location of “name of your map”.pgm in “name of your map”.yaml is right

   For example, the context of “my_map.yaml” is as followed:

```
  # my_map.yaml
  image: ./my_map.pgm
  resolution: 0.050000
  origin: [-10.000000, -10.000000, 0.000000]
  negate: 0
```

(continues on next page)

(continued from previous page)

```
occupied_thresh: 0.65
free_thresh: 0.196
```

Hint: make sure that there is the right path of *my_map.pgm* in *my_map.yaml*. The path of "my_map.png" in "my_map.yaml" is relative. So if "my_map.yaml" and "my_map.png" are in the same folder, the parameter of "image" should be "image: ./my_map.pgm"

3.2. Start the simulated robot

1. Open a terminal
2. Set ROS environment variables
   - First you need to go into your workspace and source your workspace:

     ```
     $ source install/setup.bash
     ```

   - Set up Gazebo model path:

     ```
     $ export GAZEBO_MODEL_PATH=`ros2 pkg \
     prefix turtlebot3_gazebo`/share/turtlebot3_gazebo/models/
     ```

   - set up the robot model that you will use:

     ```
     $ export TURTLEBOT3_MODEL=burger
     ```

3. Bring up Turtlebot in simulation

   ```
   # in the same terminal, run

   $ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
   ```

4. Start navigation stack

   Open another terminal, source your workspace, set up the robot model that you will use, then

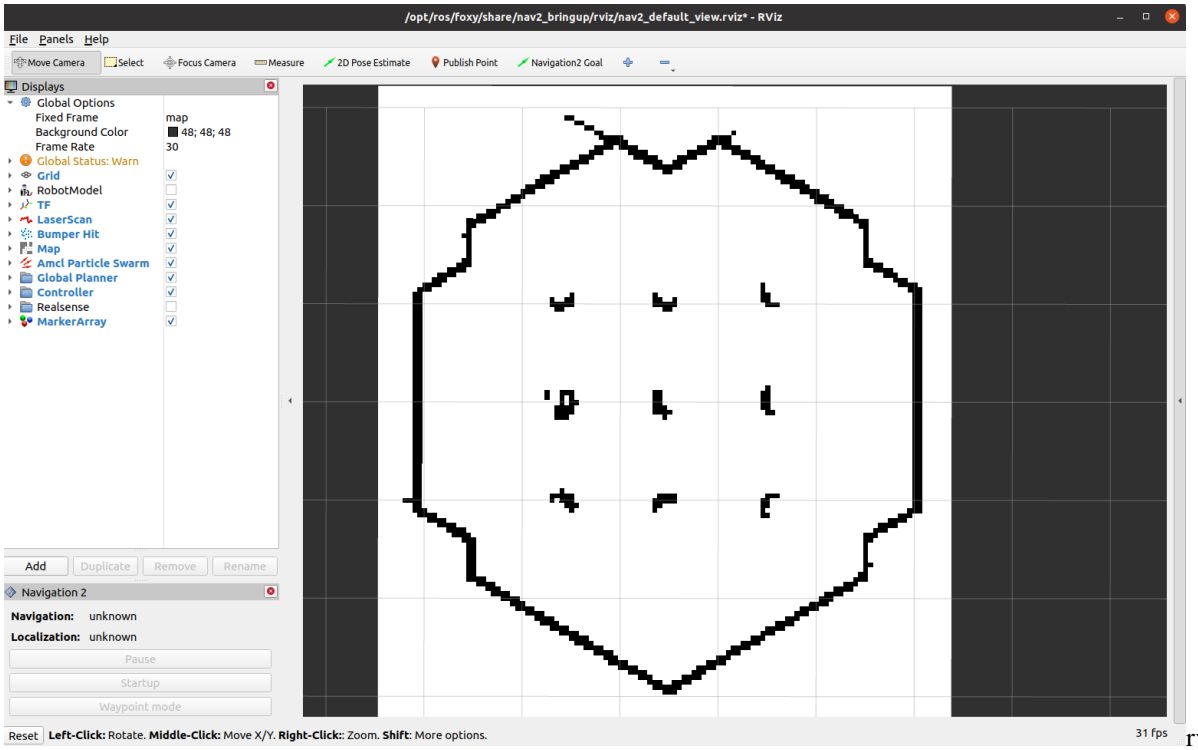
   ```
   $ ros2 launch turtlebot3_navigation2 \
   navigation2.launch.py \
   use_sim_time:=true map:=maps/"you map name".yaml
   ```

   The path of the map is relative to the place where you will run this command.

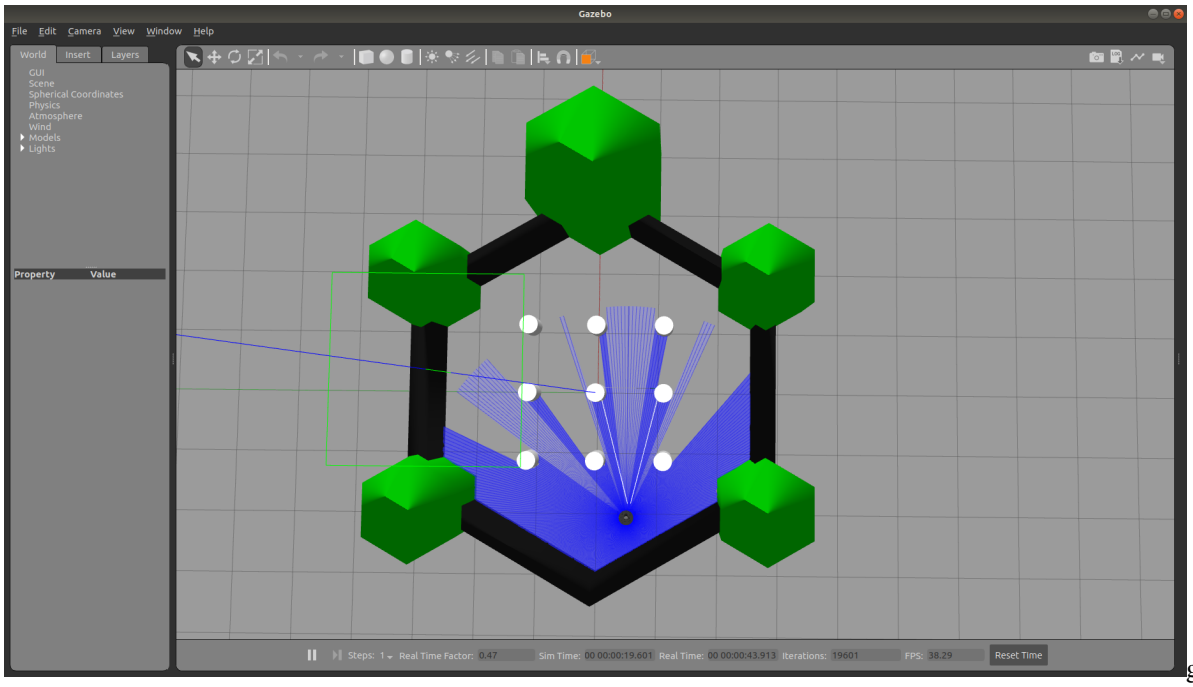
   You also can use this command to check which parameter that you can define:

   ```
   ros2 launch turtlebot3_navigation2 navigation2.launch.py \
   --show-args
   ```

   If everything has started correctly, you will see the RViz and Gazebo GUIs like this.



rviz_navigation_1



gazebo_navigation

3.3. Start with a physicals robot

Open a terminal on TurtleBot3.

Bring up basic packages to start TurtleBot3 applications.

[TurtleBot3]

```
$ source .bashrc
$ ros2 launch turtlebot3_bringup robot.launch.py
```

[Remote PC]

```
$ cd "your workspace"
$ source install/setup.bash
$ export ROS_DOMAIN_ID =
 "same as ROS DOMAIN ID of the turtlebot you are using"
$ ros2 launch turtlebot3_navigation2 navigation2.launch.py\
 map:=maps/map.yaml
```

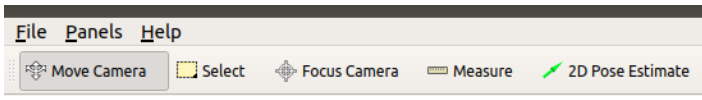
2.4.4 4. Navigate the robot via rviz

- **Step 1: Tell the robot where it is**

  after starting, the robot initially has no idea where it is. By default, Navigation 2 waits for you to give it an approximate starting position.

  It has to manually update the initial location and orientation of the TurtleBot3. This information is applied to the AMCL algorithm.

  This can be done graphically with RViz by the instruction below:

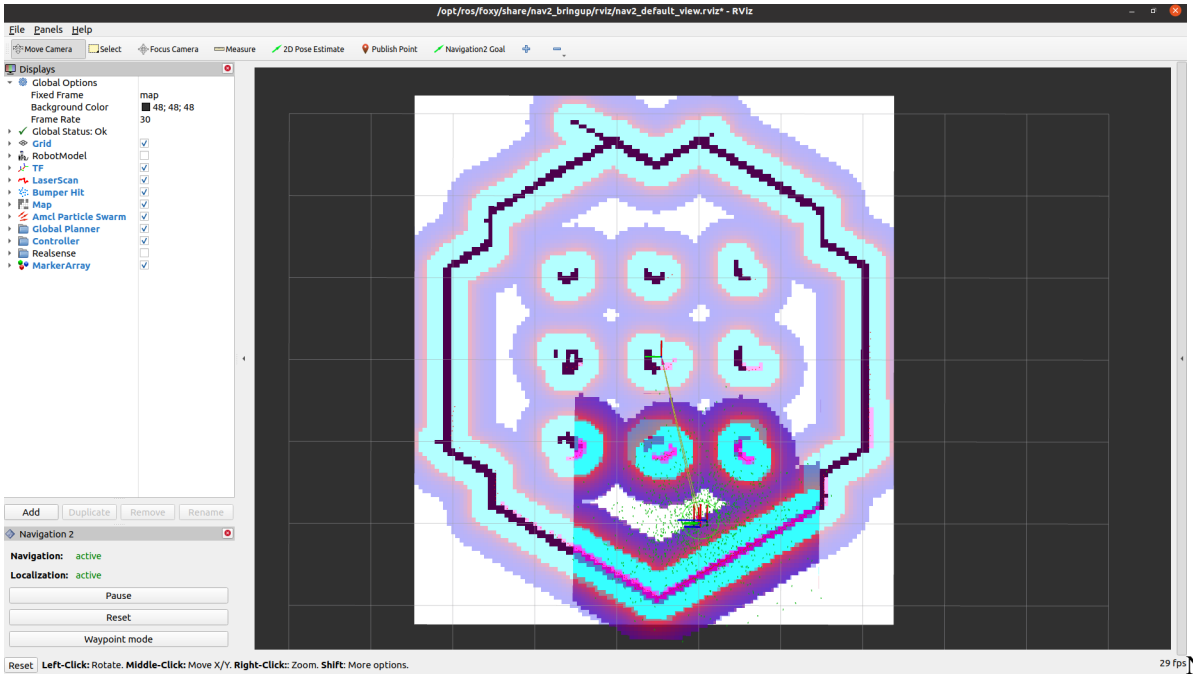


  - Click “2D Pose Estimate” button (in the top menu; see the picture)
  - Click on the approximate point in the map where the TurtleBot3 is located and drag the cursor to indicate the direction where TurtleBot3 faces.

  If you don’t get the location exactly right, that’s fine. Navigation 2 will refine the position as it navigates. You can also, click the “2D Pose Estimate” button and try again, if you prefer.

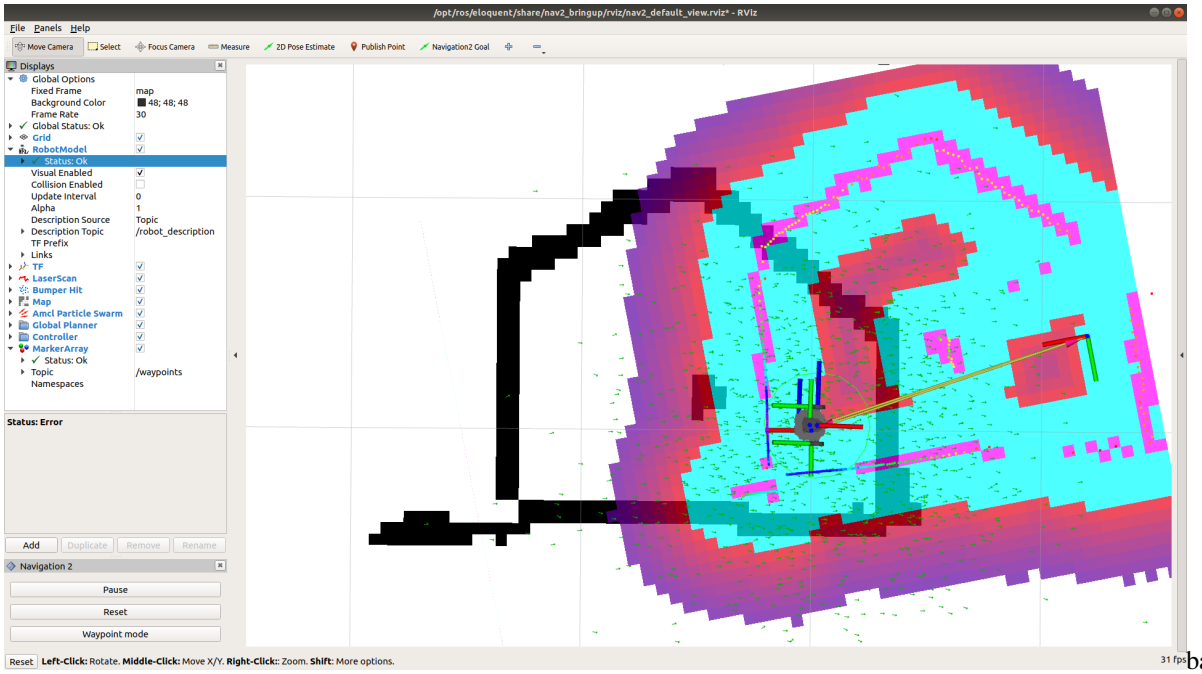
  Once you’ve set the initial pose, the tf tree will be complete and Navigation 2 is fully active and ready to go.

  If you are using simulation with turtlrbot_world map, it will show as below:

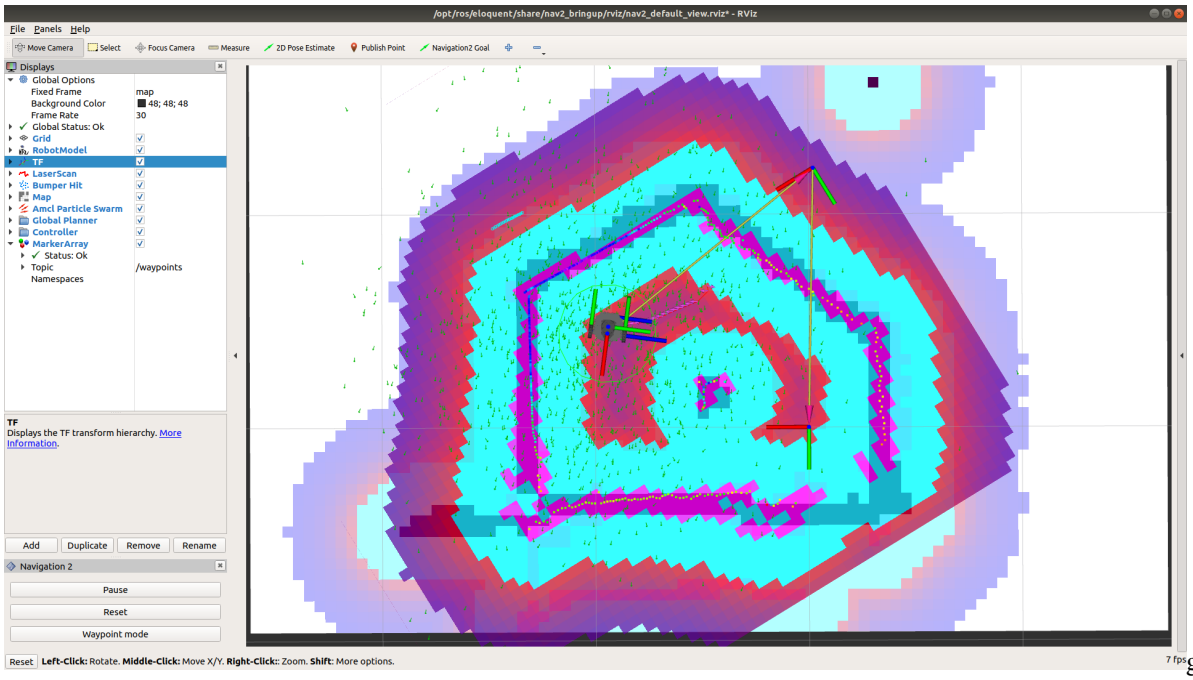


Navigation_is_rea

As soon as the 2D Pose Estimation arrow is drawn, the pose (transformation from the map to the robot) will update. As a result the centre of the laser scan has changed, too. Check if the visualization of the live laser scan matches the contours of the virtual map (Illustrated in the following two pictures! The left one is the wrong robot pose and the right one is right robot pose) to confirm that the new starting pose is accurate.



bad



good

- **Step 2: Give a goal**

  if the TurtleBot3 is localized, it can automatically create a path from the current position to any target reachable position on the map. In order to set a goal position, follow the instruction below:

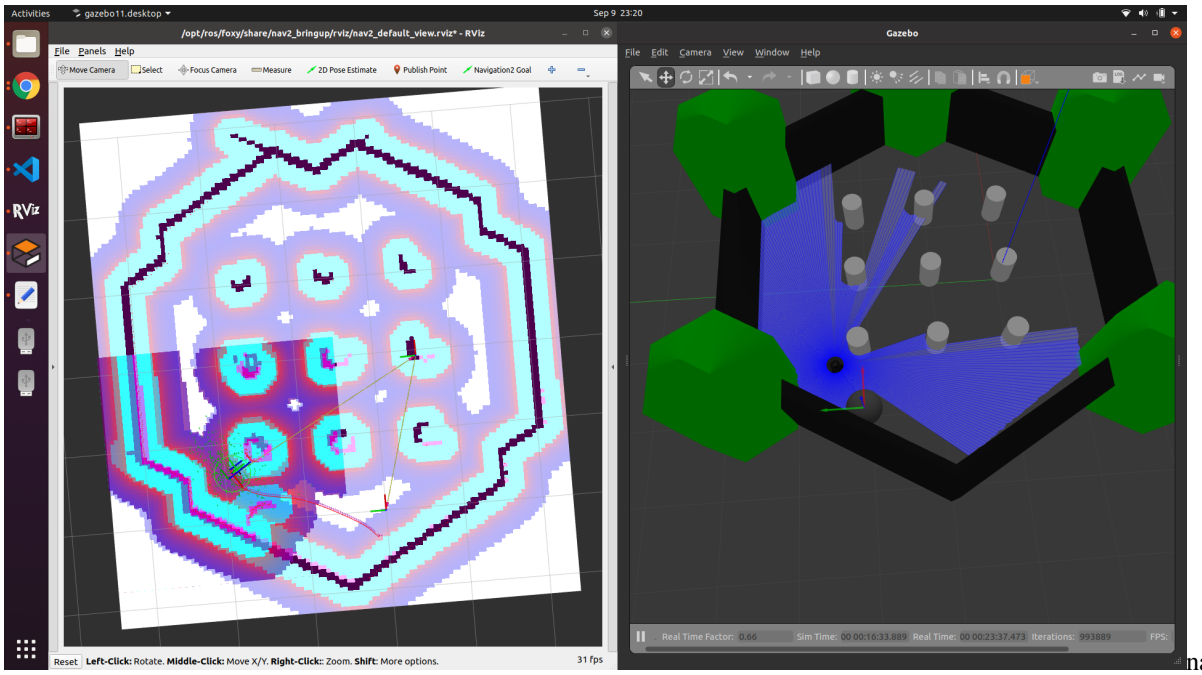
  – Click the 2D Nav Goal button (also in the top menu)

  – Click on a specific point in the map to set a goal position and drag the cursor to the direction where TurtleBot should be facing at the end

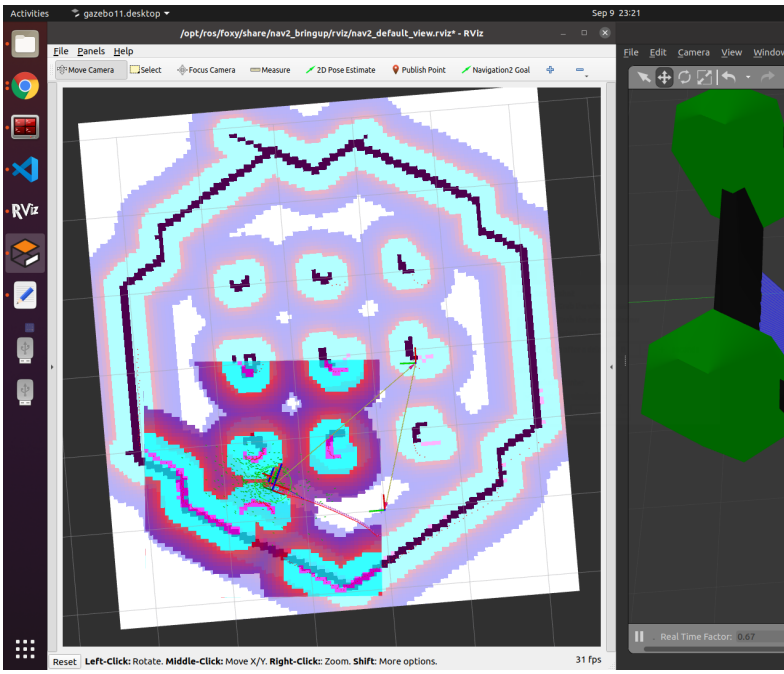
  Hint: If you wish to stop the robot before it reaches to the goal position, set the current position of TurtleBot3 as a goal position.

  The swarm of the green small arrows is the visualization of the **adaptive Monte Carlo localization (AMCL)**. Every green arrow stands for a possible position and orientation of the TurtleBot3. Notice that in the beginning its distribution is spread over the whole map. As soon as the robot moves the arrows get updated because the algorithm incorporates new measurements. During the movement the distribution of arrows becomes less chaotic and settles more and more to the robot's location, which finally means that the algorithm becomes more and more certain about the pose of the robot in the map.

- **Step 3: Play around**When the robot is on the way to the goal, you can put an obstacle in Gazebo:



nav_goal_ob_1



You can see how the robot reacts to this kind of situation:

It depends on how you design the behavior tree structure. The one we are using is "navigate_w_replanning_and_recovery.xml" You can find in this link

CHAPTER

THREE

SESSION 3 - ROS 2 MANIPULATION

3.1 Under development