Understanding ROS 2 Topics

Estimated time to completion: **35 minutes**

3.5 Topic Subscriber

To get started, it's important to understand what a subscriber is. Based on what you've learned so far, you can think of a **subscriber** as a **node that reads data from a specific Topic**.

To put this concept into practice, we'll create an obstacle detection program.

The goal is to build an obstacle detector that subscribes to the `laser_scan` topic in order to detect obstacles around the robot.

In the file `subscriber_obstacle_detector.py`, which we created in Exercise 3.1, we will update the code to turn it into a subscriber.

We'll walk through the process step by step.

Step1: Create a subscription to a `topic` inside the ROS2 node class:

- We want to subscribe to the `laser_scan` `topic`.
- For that we will use the method `self.create_subscription()`. Its `self` because its making reference to the `Node` class, so its the `Node` class that gives us the capability of starting a `SUBSCRIBER`.
- We have to add the following code:

In [ ]:

```
# import the LaserScan module from sensor_msgs interface
from sensor_msgs.msg import LaserScan
# import Quality of Service library, to set the correct p
from rclpy.qos import ReliabilityPolicy, QoSProfile

class ObstacleDetectorNode(Node):
    def __init__(self, node_name="obstacle_detector_node"
        self._node_name = node_name
        super().__init__(self._node_name)

            # create the subscriber object
        # in this case, the subscriptor will be subscribe
        # use the LaserScan module for /scan topic
        # send the received info to the listener_callback
        self.subscriber = self.create_subscription(
            LaserScan,
            '/laser_scan',
            self.laserscan_callback,
            QoSProfile(depth=10, reliability=ReliabilityP
```

- The `create_subscription()` has the following arguments:
  - `interface` **type of the** `topic`: In this case its `LaserScan`
  - **name of the** `topic`: `/laser_scan`.
  - `Callback` **function name**: `laserscan_callback`, which we will define after inside the `ObstacleDetectorNode`.
  - **Quality of Service configuration**: This define the security and reliability policy of the `topic`. This is set by the `PUBLISHER`, so when we `subscribe` we have to research what the `publisher` policy is so our `subscriber` works. If not set properly, **THE** `SUBSCRIBER` **WON'T RECEIVE MESSAGES**.

- Notes -

Quality of Service. You may have encountered this concept before. You might have come across related information if you have explored and experimented more with the `ros2 topic echo` command.

Execute in Terminal #1

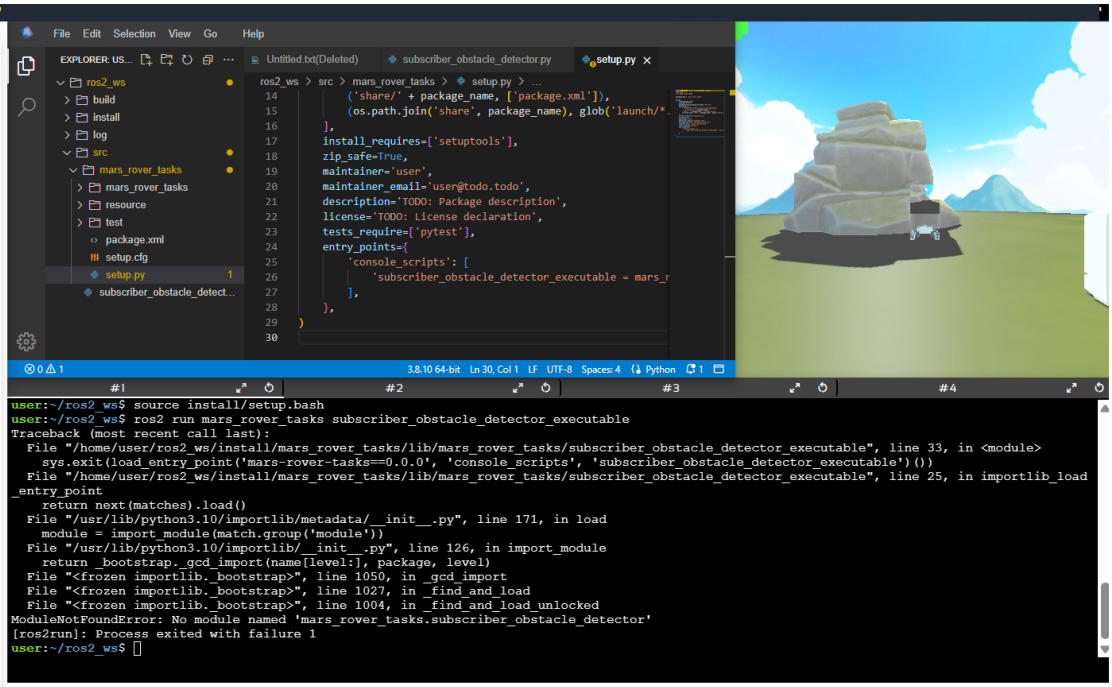
In [ ]:

```
ros2 topic echo -h
```

All these options, including Quality of Service, are referred to as `QoS` (Quality of Service).

Terminal #1 Output

```
usage: ros2 topic echo [-h] [--spin-time SPIN_TIME] [
...
  --qos-profile {unknown,system_default,sensor_data,s
                        Quality of service preset pro
                        sensor_data)
  --qos-depth N         Queue size setting to subscri
                        profile option)
  --qos-history {system_default,keep_last,keep_all,unk
                        History of samples setting to
                        of --qos-profile option, defa
  --qos-reliability {system_default,reliable,best_effo
                        Quality of service reliabilit
                        reliability value of --qos-pr
                        existing publishers )
  --qos-durability {system_default,transient_local,vo
```



```
                      Quality of service durability
                      durability value of --qos-prof
                      existing publishers )
...
```

As you can see, you can choose from different profiles such as `sensor_data`, `services_default`, `parameters`, and others.

For now, all you need to know is that:

The `QoS` settings of the `topic` `Publisher` and the `Subscriber` node need to be compatible.

If they are not compatible, the communication between them won't work.

Due to the fact that it is the `Publisher` that sets the QoS of the `topic`, the `subscribers` are depend on that configuration.

To get information about the QoS setting of a specific `topic`, you can use the following command:

Execute in Terminal #1

In [ ]:

```
ros2 topic info /laser_scan -v
```

Terminal #1 Output

```
Type: sensor_msgs/msg/LaserScan

Publisher count: 1

Node name: laser
Node namespace: /
Topic type: sensor_msgs/msg/LaserScan
Endpoint type: PUBLISHER
GID: 01.0f.c4.30.39.01.c7.1e.01.00.00.00.00.00.51.0
3.00.00.00.00.00.00.00.00
QoS profile:
  Reliability: RELIABLE
  History (Depth): UNKNOWN
  Durability: VOLATILE
  Lifespan: Infinite
  Deadline: Infinite
  Liveliness: AUTOMATIC
  Liveliness lease duration: Infinite

Subscription count: 0
```

- For instance, you can see from the output that the `Reliability` setting is `RELIABLE`.
- This is why you should set the same in your `subscriber` node:

In [ ]:

```
QoSProfile(depth=10, reliability=ReliabilityPolicy.RELIAB
```

In any case, do not worry too much about these `QoS` settings right now, as they are a somewhat more advanced concept. If you want to learn more about this `topic`, you can check out the **Understanding** `QoS` unit from the Intermediate ROS2 course.

- End of Notes -

Step2: Create a subscription `callback`:

- We define the `callback` `laserscan_callback`.
- All `topic` `callbacks` have as input two elements:
  - `self`: contains everything related to the class.
  - `msg`: This is where the latest `topic` message will be stored.
- This `msg` object will have the structure defined in the `LaserScan` message:

Execute in Terminal #1

In [ ]:

```
ros2 interface show sensor_msgs/msg/LaserScan
```

Terminal #1 Output

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different
behavior (e.g. a sonar
# array), please find or create a different message,
since applications
# will make fairly laser-specific assumptions about
this data

std_msgs/Header header # timestamp in the header is
the acquisition time of
        builtin_interfaces/Time stamp
                int32 sec
                uint32 nanosec
        string frame_id
                        # the first ray in the
scan.
                        #
                        # in frame frame_id, an
gles are measured around
                        # the positive Z axis
(counterclockwise, if Z is up)
                        # with zero angle being
forward along the x axis

float32 angle_min            # start angle of the sc
an [rad]
float32 angle_max            # end angle of the scan
[rad]
float32 angle_increment      # angular distance betw
een measurements [rad]

float32 time_increment       # time between measurem
ents [seconds] - if your scanner
                        # is moving, this will
be used in interpolating position
                        # of 3d points
float32 scan_time            # time between scans [s
econds]

float32 range_min            # minimum range value
[m]
float32 range_max            # maximum range value
[m]

float32[] ranges             # range data [m]
```

```
                                # (Note: values < range
_min or > range_max should be discarded)
float32[] intensities       # intensity data [devic
e-specific units].  If your
                                # device does not provi
de intensities, please leave
                                # the array empty.
```

- In Python, we will access these variables in the same way.
- For example, if we want to access the `ranges`, we would use `msg.ranges`.
- Inside the `laserscan_callback` we want to access for the moment only to :
  - `ranges` : To know the distance of the objects detected. In this case we will look for the closest object (minimum distance).

In [ ]:

```
def laserscan_callback(self, msg):
    # Find the minimum distance in the ranges array
    min_distance = min(msg.ranges)

    # Log the minimum distance value
    self.get_logger().info(f'Minimum distance: {min_dista
```

Step3: Add the `spin()`

- We need to add the `spin()` method to make the node keep executing indefinitely, and process the `callbacks`.
- `spin()` has the main function of processing the incoming `callbacks`, without it won't work.

- At the end, your code should look something like this:

subscriber_obstacle_detector.py

In [ ]:

```
#!/usr/bin/env python

import rclpy
from rclpy.node import Node
# import the LaserScan module from sensor_msgs interface
from sensor_msgs.msg import LaserScan
# import Quality of Service library, to set the correct p
from rclpy.qos import ReliabilityPolicy, QoSProfile

class ObstacleDetectorNode(Node):
    def __init__(self, node_name="obstacle_detector_node"
        self._node_name = node_name
        super().__init__(self._node_name)

                # create the subscriber object
        # in this case, the subscriptor will be subscribe
        # use the LaserScan module for /scan topic
        # send the received info to the laserscan_callbac
        self.subscriber = self.create_subscription(
            LaserScan,
            '/laser_scan',
            self.laserscan_callback,
            QoSProfile(depth=10, reliability=ReliabilityP

        self.get_logger().info(self._node_name +" Ready..

    def laserscan_callback(self, msg):
        # Find the minimum distance in the ranges array
        min_distance = min(msg.ranges)

        # Log the minimum distance value
        self.get_logger().info(f'Minimum distance: {min_d

def main(args=None):
    rclpy.init(args=args)
    node = ObstacleDetectorNode()
    rclpy.spin(node)
    rclpy.shutdown()

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

Step4: Start the ROS2 node

Execute in Terminal #1

- Compile to apply the latest changes:

In [ ]:

```
cd ~/ros2_ws/
source install/setup.bash
colcon build --packages-select mars_rover_tasks
source install/setup.bash
```

Execute in Terminal #1

In [ ]:

```
cd ~/ros2_ws/
source install/setup.bash
ros2 run mars_rover_tasks subscriber_obstacle_detector_ex
```

You should see output in your terminal similar to the following:

Terminal #1 Output

```
[INFO] [1724778650.814256725] [obstacle_detector_nod
e]: Minimum distance: 1.94 meters
[INFO] [1724778650.919816621] [obstacle_detector_nod
e]: Minimum distance: 1.94 meters
[INFO] [1724778651.017848573] [obstacle_detector_nod
e]: Minimum distance: 1.94 meters
[INFO] [1724778651.115014803] [obstacle_detector_nod
e]: Minimum distance: 1.94 meters
[INFO] [1724778651.238732992] [obstacle_detector_nod
e]: Minimum distance: 1.94 meters
[INFO] [1724778651.347471638] [obstacle_detector_nod
```

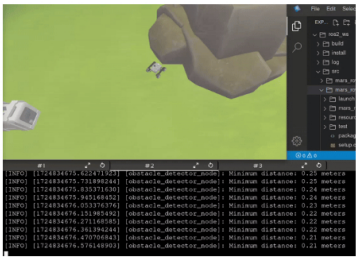
```
e]: Minimum distance: 1.94 meters
```

- Move the robot around and observe how the minimum distance changes:

▶ Execute in Terminal #2

In [ ]:

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

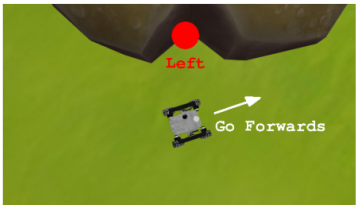


- Exercise 3.2 -

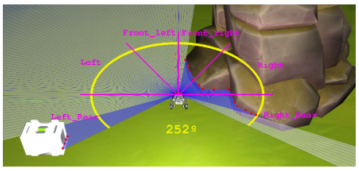
Create a better obstacle detector and action suggestion system

- The `subscriber_obstacle_detector.py` from the example is very basic, and doesn't give us enough data to be able to decide what to do.
- We need to add the capability of detecting if the obstacles are in the front, left or right sides simultaneously.
- We also need to provide suggested movements based on the detection.
- Let's look at some examples:

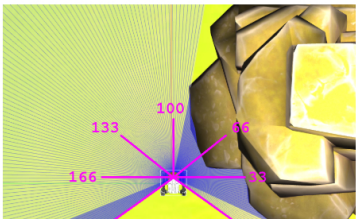
- The script must be able to detect various obstacle scenarios.
- Based on these scenarios, it should also suggest actions to perform.
- For example, if it detects an obstacle on the left, it might suggest **Go Forwards** or **Go Forwards turning Right** depending on the obstacle's location.



- We divide the `ranges` laser array into **SIX sectors** (252 degrees full range divided into 6, being 42º per sector) named:
  - `0 to 42 -> Left_Rear` , Action Suggested = Go Forwards
  - `43 to 84 -> Left` , Action Suggested = Go Forwards turning slightly right (proportional to the angle of detection)
  - `85 to 126 -> Front_Left` , Action Suggested = Turn Right
  - `126 to 168 -> Front_Right` , Action Suggested = Turn Left
  - `169 to 210 -> Right` , Action Suggested = Go Forwards turning slightly left (proportional to the angle of detection)
  - `211 to 252 -> Right_Rear` , Action Suggested = Go Forwards
  - If **NO DETECTION**, Go Forwards
  - And there are combinations of these detections, like detecting both `Fronf_Left` and `Fronf_Right` , which might be good idea to reverse turning to avoid getting stuck between contradictory actions (going forwards and going reverse). Its up to you to decide what to do in these cases and how to order these conditions, based on
- **A detection** is when the distance is lower than `0.8 meters` to be sure that we dont approach anything we dont intend to.
- For each sector we will pick the minimum distance.



- The `ranges` array indices go from right to left, with a total of 200 laser beams distributed across the 252º field of view.
- Use this information to define the index ranges for each of the sectors.



- This is your first serious robot application, so enjoy the process and have fun!

- End of Exercise 3.2 -

- Expected Result -

Execute in Terminal #1

In [ ]:

```
cd ~/ros2_ws
source install/setup.bash
ros2 run mars_rover_tasks subscriber_obstacle_detector_ex
```

Terminal #1 Output

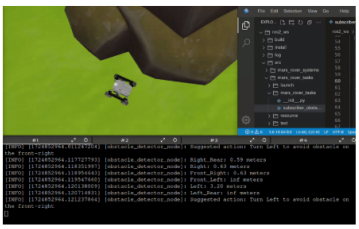
```
[INFO] [1724852976.464513897] [obstacle_detector_node
[INFO] [1724852976.465202067] [obstacle_detector_node
[INFO] [1724852976.465868065] [obstacle_detector_node
[INFO] [1724852976.466517859] [obstacle_detector_node
[INFO] [1724852976.467188508] [obstacle_detector_node
[INFO] [1724852976.472156784] [obstacle_detector_node
[INFO] [1724852976.472996279] [obstacle_detector_node
[INFO] [1724852976.576859960] [obstacle_detector_node
[INFO] [1724852976.577583679] [obstacle_detector_node
[INFO] [1724852976.578267697] [obstacle_detector_node
[INFO] [1724852976.578897339] [obstacle_detector_node
[INFO] [1724852976.579526391] [obstacle_detector_node
[INFO] [1724852976.580164494] [obstacle_detector_node
[INFO] [1724852976.580724798] [obstacle_detector_node
[INFO] [1724852976.702257608] [obstacle_detector_node
[INFO] [1724852976.702957455] [obstacle_detector_node
[INFO] [1724852976.703630842] [obstacle_detector_node
[INFO] [1724852976.704335964] [obstacle_detector_node
[INFO] [1724852976.704996277] [obstacle_detector_node
[INFO] [1724852976.705634819] [obstacle_detector_node
[INFO] [1724852976.706309071] [obstacle_detector_node
```

- Start the **teleoperation node** so you can move the Mars rover around and check if the suggested actions make sense.

Execute in Terminal #2

In [ ]:

```
cd ~/ros2_ws
source install/setup.bash
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```



- Expected Result -

- RECOMMENDATION -

PLEASE TRY TO COMPLETE THE EXERCISE BEFORE LOOKING AT THE SOLUTION

- END RECOMMENDATION -

- Solution for Exercise 3.2 -

subscriber_obstacle_detector.py

In [ ]:

```
#!/usr/bin/env python

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan
from rclpy.qos import ReliabilityPolicy, QoSProfile
import random

class ObstacleDetectorNode(Node):
    def __init__(self, node_name="obstacle_detector_node"
        self._node_name = node_name
```

```
        super().__init__(self._node_name)

        # create the subscriber object
        self.subscriber = self.create_subscription(
            LaserScan,
            '/laser_scan',
            self.laserscan_callback,
            QoSProfile(depth=10, reliability=ReliabilityP
        )

        self.get_logger().info(self._node_name + " Ready.

    def laserscan_callback(self, msg):
        # Define the sectors
        sectors = {
            "Right_Rear": (0, 33),
            "Right": (34, 66),
            "Front_Right": (67, 100),
            "Front_Left": (101, 133),
            "Left": (134, 166),
            "Left_Rear": (167, 199)
        }

        # Initialize the minimum distances for each secto
        min_distances = {key: float('inf') for key in sec

        # Find the minimum distance in each sector
        for sector, (start_idx, end_idx) in sectors.items
            # Ensure the index range is within bounds and
            if start_idx < len(msg.ranges) and end_idx <
                sector_ranges = msg.ranges[start_idx:end_
                if sector_ranges:
                    min_distances[sector] = min(sector_ra

        # Log the minimum distances
        for sector, min_distance in min_distances.items()
            self.get_logger().info(f'{sector}: {min_dista

        # Define the threshold for obstacle detection
        obstacle_threshold = 0.8  # meters

        # Determine detected obstacles
        detections = {sector: min_distance < obstacle_thr

        # Determine suggested action based on detection a
        # Priority 1: Front detection, Priority 2: Side d
        # Priority 1
        if detections["Front_Left"] and detections["Front
            arbitrary_direction = "Right"
            action = "Selected Turn Arbitrary Direction "
        elif detections["Front_Left"] and not detections[
            action = "Turn Right to avoid obstacle on the
        elif detections["Front_Right"] and not detections
            action = "Turn Left to avoid obstacle on the
        # Priority 2
        elif detections["Left"]:
            action = "Go Forwards turning slightly right
        elif detections["Right"]:
            action = "Go Forwards turning slightly left t
        # Priority 3
        elif detections["Right_Rear"]:
            action = "Go Forwards, BUT DONT reverse Right
        elif detections["Left_Rear"]:
            action = "Go Forwards, BUT DONT reverse left"
        else:
            action = "Go Forwards"

        # Log the suggested action
        self.get_logger().info(f'Suggested action: {actio

def main(args=None):
    rclpy.init(args=args)
    node = ObstacleDetectorNode()
    rclpy.spin(node)
    rclpy.shutdown()

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

- Let's review it piece by piece:

In [ ]:

```
sectors = {
            "Right_Rear": (0, 33),
            "Right": (34, 66),
            "Front_Right": (67, 100),
            "Front_Left": (101, 133),
            "Left": (134, 166),
            "Left_Rear": (167, 199)
        }
```

- Using the indices, we define each of the **six sectors** inside a dictionary with the **name of the sector** and the **start and end indices** of its range.

In [ ]:

```
min_distances = {key: float('inf') for key in sectors.key
```

- The line `min_distances = {key: float('inf') for key in sectors.keys()}` creates a dictionary called min_distances with the keys corresponding to the sectors defined in the sectors dictionary, and it initializes the values for all these keys to float('inf').

In [ ]:

```
# Determine detected obstacles
detections = {sector: min_distance < obstacle_threshold f
```

- Here, we iterate over the minimum values for each sector and create a dictionary that has:
  - `key` : The name of the sector
  - `value` : A boolean indicating whether something was detected in that sector at a distance lower than 0.8 meters.

- This allows you to access these values like so: `detections["Front_Left"]`

In [ ]:

```
if detections["Front_Left"] and detections["Front_Right"]
    arbitrary_direction = "Right"
    action = "Selected Turn Arbitrary Direction "+arbitra
elif detections["Front_Left"] and not detections["Front_R
    action = "Turn Right to avoid obstacle on the front-l
elif detections["Front_Right"] and not detections["Front_
    action = "Turn Left to avoid obstacle on the front-ri
# Priority 2
elif detections["Left"]:
    action = "Go Forwards turning slightly right to avoid
elif detections["Right"]:
    action = "Go Forwards turning slightly left to avoid
# Priority 3
elif detections["Right_Rear"]:
    action = "Go Forwards, BUT DONT reverse Right"
elif detections["Left_Rear"]:
    action = "Go Forwards, BUT DONT reverse left"
else:
    action = "Go Forwards"
```

- You might have approached this differently. In this case, we prioritize checking high-priority detections first. If no high-priority detections are found, we then check the next set of detections.
- It is worth noting that when detecting obstacles in both `Front` sectors, we arbitrarily chose to turn **Right** , there is no specific reason why we chose right instead of left.

- End of Solution for Exercise 3.2 -

24/09/2024