ROS2 Basics in 5 Days (Python)

Unit 5 Managing Complex Nodes & Multithreading

- Summary -

What will you learn in this unit?

- How to manage more complex Nodes
- Executors
- Callback Groups

- End of Summary -

5.1 Managing Complex Nodes

You have been working with programs that are relatively easy to manage. However, some issues may appear as the programs you create get more complex.

Start by creating a more complex program.

- Example 5.1 -

1. Create a new package named **unit5_pkg**.

Execute in Shell

In [ ]:

```
cd ~/ros2_ws/src
```

In [ ]:

```
ros2 pkg create --build-type ament_python unit5_pkg --dependencies rclpy std_msgs sensor_msgs geometry_msgs nav_msgs
```

2. Create a new file named **exercise51.py** inside the **unit5_pkg** folder in the package you have just created.

exercise51.py

In [ ]:

```
import rclpy
from rclpy.node import Node
import time
import numpy as np
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from rclpy.qos import ReliabilityPolicy, QoSProfile


class ControlClass(Node):

    def __init__(self, seconds_sleeping=10):
        super().__init__('sub_node')
        self._seconds_sleeping = seconds_sleeping
        # Define a Publisher for the /cmd_vel topic
        self.vel_pub = self.create_publisher(Twist, 'cmd_vel', 10)
        self.cmd = Twist()
        # Define a Subscriber for the /odom topic
        self.odom_sub = self.create_subscription(
            Odometry, 'odom', self.odom_callback, 10)
        # Define a Subscriber for the /scan topic
        self.scan_sub = self.create_subscription(LaserScan, 'scan', self.scan_callback, QoSProfile(
            depth=10, reliability=ReliabilityPolicy.BEST_EFFORT))
        # Define a timer object
        self.timer = self.create_timer(0.5, self.timer_callback)
        self.laser_msg = LaserScan()
        self.roll = 0.0
        self.pitch = 0.0
        self.yaw = 0.0

    # Callback function for the /odom Subscriber
    def odom_callback(self, msg):
        self.get_logger().info("Odom CallBack")
        orientation_q = msg.pose.pose.orientation
        orientation_list = [orientation_q.x,
                            orientation_q.y, orientation_q.z, orientation_q.w]
        (self.roll, self.pitch, self.yaw) = self.euler_from_quaternion(orientation_list)

    # Callback function for the /scan Subscriber
    def scan_callback(self, msg):
        self.get_logger().info("Scan CallBack")
        self.laser_msg = msg

    # Get the value of the front laser
    def get_front_laser(self):
        return self.laser_msg.ranges[360]

    # Get the yaw value
    def get_yaw(self):
        return self.yaw

    # Convert a quaternion to Euler angles
    def euler_from_quaternion(self, quaternion):
        """
        Converts quaternion (w in last place) to Euler roll, pitch, yaw
        quaternion = [x, y, z, w]
        Below should be replaced when porting for ROS2 Python tf_conversions is done.
        """
        x = quaternion[0]
        y = quaternion[1]
        z = quaternion[2]
        w = quaternion[3]

        sinr_cosp = 2 * (w * x + y * z)
        cosr_cosp = 1 - 2 * (x * x + y * y)
        roll = np.arctan2(sinr_cosp, cosr_cosp)

        sinp = 2 * (w * y - z * x)
        pitch = np.arcsin(sinp)

        siny_cosp = 2 * (w * z + x * y)
        cosy_cosp = 1 - 2 * (y * y + z * z)
        yaw = np.arctan2(siny_cosp, cosy_cosp)

        return roll, pitch, yaw

    # Send velocities to stop the robot
    def stop_robot(self):
        self.cmd.linear.x = 0.0
        self.cmd.angular.z = 0.0
        self.vel_pub.publish(self.cmd)

    # Send velocities to move the robot forward
    def move_straight(self):
        self.cmd.linear.x = 0.08
        self.cmd.angular.z = 0.0
        self.vel_pub.publish(self.cmd)

    # Send velocities to rotate the robot
    def rotate(self):
        self.cmd.angular.z = -0.2
        self.cmd.linear.x = 0.0
        self.vel_pub.publish(self.cmd)

        self.get_logger().info("Rotating for "+str(self._seconds_sleeping)+" seconds")
        # Keep rotating the robot for self._seconds_sleeping seconds
        for i in range(self._seconds_sleeping):
            self.get_logger().info("SLEEPING=="+str(i)+" seconds")
            time.sleep(1)

        self.stop_robot()

    # Callback for the Timer object
    def timer_callback(self):
        self.get_logger().info("Timer CallBack")
        try:
            self.get_logger().warning(">>>>>>>>>>>>>>>RANGES Value=" +
                                      str(self.laser_msg.ranges[360]))
            if not self.laser_msg.ranges[360] < 0.5:
                self.get_logger().info("MOVE STRAIGHT")
                self.move_straight()
            else:
                self.get_logger().info("STOP ROTATE")
                self.stop_robot()
                self.rotate()
        except:
            pass


def main(args=None):
    rclpy.init(args=args)
    control_node = ControlClass()
    try:
        rclpy.spin(control_node)
    finally:
        control_node.destroy_node()
```

```
    rclpy.shutdown()


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

- Notes -

Review the code to understand it. The logic of the Node is pretty simple:

- The robot will move forward until it detects an obstacle (the sphere) in front of the robot closer than 0.5 meters.
- When the sphere is closer than 0.5 meters, the robot stops (`self.stop_robot()`) and rotates (`self.rotate()`)
- The robot will rotate for the number of seconds specified in `self._seconds_sleeping` . In this case, it is 10 seconds.

- End of Notes -

3. Create a launch file named **exercise51.launch.py** to launch the Node you created.

Execute in Shell

In [ ]:

```
cd ~/ros2_ws/src/unit5_pkg
mkdir launch
```

Execute in Shell

In [ ]:

```
cd ~/ros2_ws/src/unit5_pkg/launch
touch exercise51.launch.py
chmod +x exercise51.launch.py
```

Add the following code to the file you created.

exercise51.launch.py

In [ ]:

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='unit5_pkg',
            executable='exercise51',
            output='screen',
            emulate_tty=True),
    ])
```

4. Modify the setup.py to add the launch file you created, and the entry points to the executable for the **exercise31.py** script.

setup.py

In [ ]:

```
from setuptools import setup
import os
from glob import glob

package_name = 'unit5_pkg'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'), glob('launch/*.launch.py'))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='somebody very awesome',
    maintainer_email='user@user.com',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'exercise51 = unit5_pkg.exercise51:main'
        ],
    },
)
```

5. Compile your package.

Execute in Shell

In [ ]:

```
cd ~/ros2_ws
colcon build --packages-select unit5_pkg
source ~/ros2_ws/install/setup.bash
```

6. Finally, launch the Node in your shell.

Execute in Shell

In [ ]:

```
ros2 launch unit5_pkg exercise51.launch.py
```

- End of Example 5.1 -

If you pay attention to the Node's behavior, you should see something strange happening.

- 1) As soon as the robot enters the ROTATE function, the **Odom and Scan CallBacks stop being executed**.
- 2) The Scan and Odom Callbacks get blocked by the Timer Callback.
- 3) In summary, when the robot is rotating, no other Callback is executed.

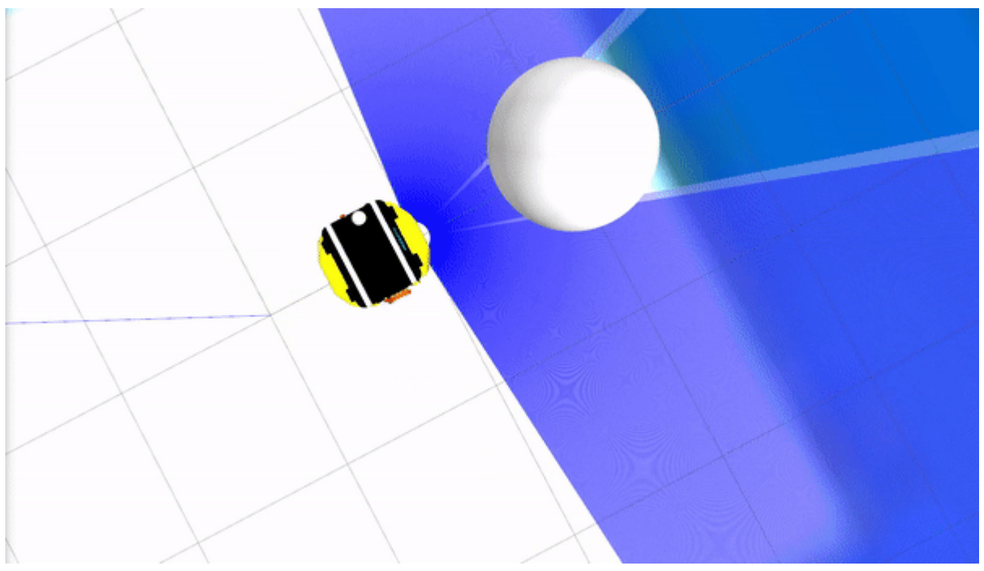
Output in Shell

```
[exercise31-1] [INFO] [1631462014.163838534] [sub_node]: Odom CallBack
[exercise31-1] [INFO] [1631462014.167065180] [sub_node]: Scan CallBack
[exercise31-1] [INFO] [1631462014.173721644] [sub_node]: Odom CallBack
[exercise31-1] [INFO] [1631462014.177145493] [sub_node]: Scan CallBack

# Here comes the timer and gets inside the STOP ROTATE
[exercise31-1] [INFO] [1631462014.183357178] [sub_node]: Timer CallBack
[exercise31-1] [WARN] [1631462014.183843557] [sub_node]: >>>>>>>>>>>>>>RANGES Value=0.43359941244125366
[exercise31-1] [INFO] [1631462014.184297364] [sub_node]: STOP ROTATE
[exercise31-1] [INFO] [1631462014.184905910] [sub_node]: Rotating for 10 seconds
[exercise31-1] [INFO] [1631462014.185361387] [sub_node]: SLEEPING==0 seconds
[exercise31-1] [INFO] [1631462015.187221747] [sub_node]: SLEEPING==1 seconds
[exercise31-1] [INFO] [1631462016.188986502] [sub_node]: SLEEPING==2 seconds
[exercise31-1] [INFO] [1631462017.190589988] [sub_node]: SLEEPING==3 seconds
[exercise31-1] [INFO] [1631462018.191754193] [sub_node]: SLEEPING==4 seconds
[exercise31-1] [INFO] [1631462019.192855738] [sub_node]: SLEEPING==5 seconds
[exercise31-1] [INFO] [1631462020.194047276] [sub_node]: SLEEPING==6 seconds
[exercise31-1] [INFO] [1631462021.195551992] [sub_node]: SLEEPING==7 seconds
[exercise31-1] [INFO] [1631462022.197470055] [sub_node]: SLEEPING==8 seconds
[exercise31-1] [INFO] [1631462023.199393248] [sub_node]: SLEEPING==9 seconds

# When it finishes, it executes again, because of stack priorities
# The sensor value is the same because the scan Callback has not been called yet.
[exercise31-1] [INFO] [1631462024.201929184] [sub_node]: Timer CallBack
[exercise31-1] [WARN] [1631462024.202541739] [sub_node]: >>>>>>>>>>>>>>RANGES Value=0.43359941244125366
[exercise31-1] [INFO] [1631462024.203118334] [sub_node]: STOP ROTATE
[exercise31-1] [INFO] [1631462024.203789991] [sub_node]: Rotating for 10 seconds
[exercise31-1] [INFO] [1631462024.204353594] [sub_node]: SLEEPING==0 seconds
[exercise31-1] [INFO] [1631462025.206108691] [sub_node]: SLEEPING==1 seconds
[exercise31-1] [INFO] [1631462026.207503346] [sub_node]: SLEEPING==2 seconds
[exercise31-1] [INFO] [1631462027.209666668] [sub_node]: SLEEPING==3 seconds
[exercise31-1] [INFO] [1631462028.211459854] [sub_node]: SLEEPING==4 seconds
[exercise31-1] [INFO] [1631462029.213224914] [sub_node]: SLEEPING==5 seconds
[exercise31-1] [INFO] [1631462030.215086652] [sub_node]: SLEEPING==6 seconds
[exercise31-1] [INFO] [1631462031.215881326] [sub_node]: SLEEPING==7 seconds
[exercise31-1] [INFO] [1631462032.217448376] [sub_node]: SLEEPING==8 seconds
[exercise31-1] [INFO] [1631462033.218958477] [sub_node]: SLEEPING==9 seconds

# And now that the scan finally was called, and the sensor value updated
# The Timer Callback can now move straight.
[exercise31-1] [INFO] [1631462034.222230681] [sub_node]: Odom CallBack
[exercise31-1] [INFO] [1631462034.224277957] [sub_node]: Scan CallBack
[exercise31-1] [INFO] [1631462034.226654888] [sub_node]: Timer CallBack
[exercise31-1] [WARN] [1631462034.228090777] [sub_node]: >>>>>>>>>>>>>>RANGES Value=13.314866065979004
[exercise31-1] [INFO] [1631462034.229608321] [sub_node]: MOVE STRAIGHT
[exercise31-1] [INFO] [1631462034.232116453] [sub_node]: Odom CallBack
[exercise31-1] [INFO] [1631462034.234230636] [sub_node]: Scan CallBack
```



There is a problem here.

- You need the sensor data to be updated. Therefore, you cannot allow the **timer_callback** to block the sensor callbacks. For instance, if this was a car, you could not wait for the control system to finish moving the car to get a new sensor reading.
- On the other hand, the control system has to be very regular because you cannot wait for a sensor processing callback to finish to take action. In real systems, if the sensor data is too old, the system is **STOPPED for SECURITY REASONS**.

What can you do? Use **multiple threads** to avoid this collision between Callbacks.

- Notes -

To stop the robot's movement from the shell, use the following command:

In [ ]:

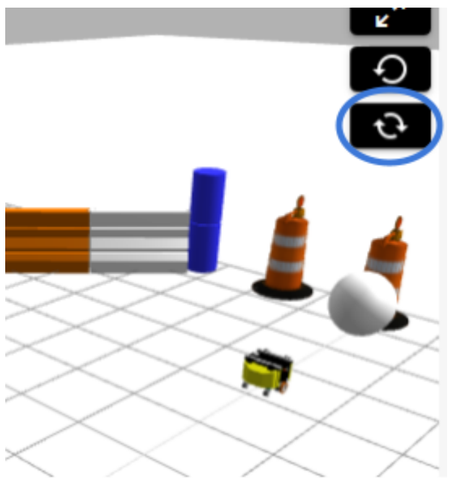
```
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist "linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
"
```

To reset the robot's initial position, use the following command:

In [ ]:

```
ros2 service call /reset_world std_srvs/srv/Empty "{}"
```

You can also restart the simulation by pressing the last icon in the simulation window:



- End of Notes -

5.2 Executors and Callback Groups

In the previous section, **the main thread blocked the call to the rest of the Callback functions**.

Two components control the execution of Callbacks: **executors** and **Callback groups**.

- Executors are responsible for the execution of the Callbacks.
- Callback Groups are used to control the execution of multiple Callbacks.

5.2.1 Executors

An executor controls the threading model used to process Callbacks. Callbacks are work units like Subscription Callbacks, Timer Callbacks, Service Callbacks, etc.

You will find two types of executors:

- **SingleThreadedExecutor**: This is a simple executor. It runs the Callbacks in the main thread (the one that calls `executor.spin()`).
- **MultiThreadedExecutor**: It allows multiple threads to run Callbacks in parallel.

5.2.2 Callback Groups

A Callback Group controls when Callbacks are allowed to be executed.

You can find two different Callback Groups:

- **ReentrantCallbackGroup**: Allows Callbacks to be executed in parallel without restriction.
- **MutuallyExclusiveCallbackGroup**: Allows only one Callback to be executed at a time.

Now, modify the Node created previously, adding an executor and using Callback Groups.

- Example 5.2 -

1. Create a new file named **exercise52.py** with the code shown below.

This Node works similarly to the Node created in Example 5.1. The only difference is that:

- You explicitly state that you are using a single thread executor.
- You create three different `MutuallyExclusiveCallbackGroup()` for the Odom, Scan, and Timer Callbacks.

exercise52.py

In [ ]:

```
import rclpy
from rclpy.node import Node
import time
import numpy as np
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from rclpy.qos import ReliabilityPolicy, QoSProfile
# Import the libraries to use executors and callback groups
from rclpy.callback_groups import ReentrantCallbackGroup, MutuallyExclusiveCallbackGroup
from rclpy.executors import MultiThreadedExecutor, SingleThreadedExecutor


class ControlClass(Node):

    def __init__(self, seconds_sleeping=10):
        super().__init__('sub_node')
        self._seconds_sleeping = seconds_sleeping
        # Define a Publisher for the /cmd_vel topic
        self.vel_pub = self.create_publisher(Twist, 'cmd_vel', 10)
        self.cmd = Twist()
        # Create a MutuallyExclusiveCallbackGroup
        self.group = MutuallyExclusiveCallbackGroup()
        # Define a Subscriber for the /odom topic and add it to the callback group
        self.odom_sub = self.create_subscription(
            Odometry, 'odom', self.odom_callback, 10,  callback_group=self.group)
        # Define a Subscriber for the /scan topic and add it to the callback group
        self.scan_sub = self.create_subscription(LaserScan, 'scan', self.scan_callback, QoSProfile(
            depth=10, reliability=ReliabilityPolicy.BEST_EFFORT), callback_group=self.group)
        # Define a timer object and add it to the callback group
        self.timer = self.create_timer(
            0.5, self.timer_callback, callback_group=self.group)
        self.laser_msg = LaserScan()
        self.roll = 0.0
        self.pitch = 0.0
        self.yaw = 0.0

    # Callback function for the /odom Subscriber
    def odom_callback(self, msg):
        self.get_logger().info("Odom CallBack")
        orientation_q = msg.pose.pose.orientation
        orientation_list = [orientation_q.x,
                            orientation_q.y, orientation_q.z, orientation_q.w]
        (self.roll, self.pitch, self.yaw) = self.euler_from_quaternion(orientation_list)

    # Callback function for the /scan Subscriber
    def scan_callback(self, msg):
        self.get_logger().info("Scan CallBack")
        self.laser_msg = msg

    # Get the value of the front laser
    def get_front_laser(self):
        return self.laser_msg.ranges[360]

    # Get the yaw value
    def get_yaw(self):
        return self.yaw

    # Convert a quaternion to Euler angles
    def euler_from_quaternion(self, quaternion):
        """
        Converts quaternion (w in last place) to Euler roll, pitch, yaw
        quaternion = [x, y, z, w]
        Below should be replaced when porting for ROS2 Python tf_conversions is done.
        """
        x = quaternion[0]
        y = quaternion[1]
        z = quaternion[2]
        w = quaternion[3]

        sinr_cosp = 2 * (w * x + y * z)
        cosr_cosp = 1 - 2 * (x * x + y * y)
        roll = np.arctan2(sinr_cosp, cosr_cosp)

        sinp = 2 * (w * y - z * x)
        pitch = np.arcsin(sinp)

        siny_cosp = 2 * (w * z + x * y)
        cosy_cosp = 1 - 2 * (y * y + z * z)
        yaw = np.arctan2(siny_cosp, cosy_cosp)

        return roll, pitch, yaw

    # Send velocities to stop the robot
    def stop_robot(self):
        self.cmd.linear.x = 0.0
        self.cmd.angular.z = 0.0
        self.vel_pub.publish(self.cmd)

    # Send velocities to move the robot forward
    def move_straight(self):
        self.cmd.linear.x = 0.08
        self.cmd.angular.z = 0.0
        self.vel_pub.publish(self.cmd)

    # Send velocities to rotate the robot
    def rotate(self):
        self.cmd.angular.z = -0.2
        self.cmd.linear.x = 0.0
        self.vel_pub.publish(self.cmd)

        self.get_logger().info("Rotating for "+str(self._seconds_sleeping)+" seconds")
        # Keep rotating the robot for self._seconds_sleeping seconds
        for i in range(self._seconds_sleeping):
            self.get_logger().info("SLEEPING=="+str(i)+" seconds")
            time.sleep(1)

        self.stop_robot()

    # Callback for the Timer object
    def timer_callback(self):
        self.get_logger().info("Timer CallBack")
        try:
            self.get_logger().warning(">>>>>>>>>>>>>>>RANGES Value=" +
                                      str(self.laser_msg.ranges[360]))
            if not self.laser_msg.ranges[360] < 0.5:
                self.get_logger().info("MOVE STRAIGHT")
                self.move_straight()
            else:
                self.get_logger().info("STOP ROTATE")
                self.stop_robot()
                self.rotate()
        except:
            pass


def main(args=None):
```

```
    rclpy.init(args=args)
    control_node = ControlClass()
    # Create a SingleThreadedExecutor
    executor = SingleThreadedExecutor()
    # Add the node to the executor
    executor.add_node(control_node)
    try:
        # Spin the executor
        executor.spin()
    finally:
        # Shutdown the executor
        executor.shutdown()
        control_node.destroy_node()
    rclpy.shutdown()


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

3. Create a launch file named **exercise52.launch.py** to launch the Node you created.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws/src/unit5_pkg/launch
touch exercise52.launch.py
chmod +x exercise52.launch.py
```

Add the following code to the file you created.

exercise52.launch.py

In [ ]:

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='unit5_pkg',
            executable='exercise52',
            output='screen',
            emulate_tty=True),
    ])
```

4. Modify the setup.py to add the launch file you created, and the entry points to the executable for the **exercise32.py** script.

setup.py

In [ ]:

```
from setuptools import setup
import os
from glob import glob

package_name = 'unit5_pkg'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'), glob('launch/*.launch.py'))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='somebody very awesome',
    maintainer_email='user@user.com',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'exercise51 = unit5_pkg.exercise51:main',
            'exercise52 = unit5_pkg.exercise52:main',
        ],
    },
)
```

5. Compile your package.

Execute in Shell

In [ ]:

```
cd ~/ros2_ws
colcon build --packages-select unit5_pkg
source ~/ros2_ws/install/setup.bash
```

6. Finally, relaunch your program.

Execute in Shell

In [ ]:

```
ros2 launch unit5_pkg exercise52.launch.py
```

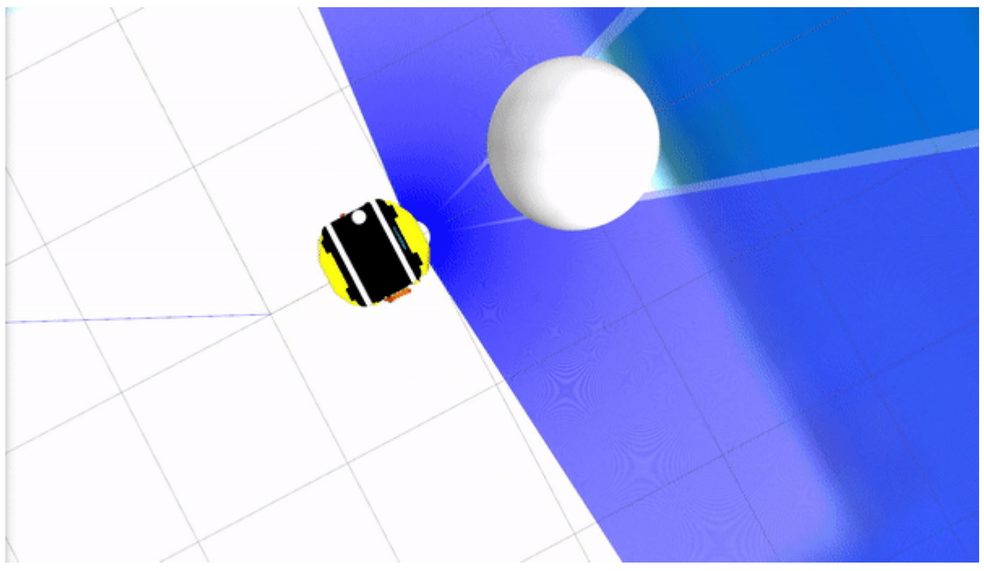
Output in Shell

```
[exercise31-1] [INFO] [1631462014.163838534] [sub_node]: Odom CallBack
[exercise31-1] [INFO] [1631462014.167065180] [sub_node]: Scan CallBack
[exercise31-1] [INFO] [1631462014.173721644] [sub_node]: Odom CallBack
[exercise31-1] [INFO] [1631462014.177145493] [sub_node]: Scan CallBack

# Here comes the timer and gets inside the STOP ROTATE
[exercise31-1] [INFO] [1631462014.183357178] [sub_node]: Timer CallBack
[exercise31-1] [WARN] [1631462014.183843557] [sub_node]: >>>>>>>>>>>>>>RANGES Value=0.43359941244125366
[exercise31-1] [INFO] [1631462014.184297364] [sub_node]: STOP ROTATE
[exercise31-1] [INFO] [1631462014.184905910] [sub_node]: Rotating for 10 seconds
[exercise31-1] [INFO] [1631462014.185361387] [sub_node]: SLEEPING==0 seconds
[exercise31-1] [INFO] [1631462015.187221747] [sub_node]: SLEEPING==1 seconds
[exercise31-1] [INFO] [1631462016.188986502] [sub_node]: SLEEPING==2 seconds
[exercise31-1] [INFO] [1631462017.190589988] [sub_node]: SLEEPING==3 seconds
[exercise31-1] [INFO] [1631462018.191754193] [sub_node]: SLEEPING==4 seconds
[exercise31-1] [INFO] [1631462019.192855738] [sub_node]: SLEEPING==5 seconds
[exercise31-1] [INFO] [1631462020.194047276] [sub_node]: SLEEPING==6 seconds
[exercise31-1] [INFO] [1631462021.195551992] [sub_node]: SLEEPING==7 seconds
[exercise31-1] [INFO] [1631462022.197470055] [sub_node]: SLEEPING==8 seconds
[exercise31-1] [INFO] [1631462023.199393248] [sub_node]: SLEEPING==9 seconds

# When it finishes, it executes again, because of stack priorities
# The sensor value is the same because the Scan Callback has not been called yet.
[exercise31-1] [INFO] [1631462024.201929184] [sub_node]: Timer CallBack
[exercise31-1] [WARN] [1631462024.202541739] [sub_node]: >>>>>>>>>>>>>>RANGES Value=0.43359941244125366
[exercise31-1] [INFO] [1631462024.203118334] [sub_node]: STOP ROTATE
[exercise31-1] [INFO] [1631462024.203789991] [sub_node]: Rotating for 10 seconds
[exercise31-1] [INFO] [1631462024.204353594] [sub_node]: SLEEPING==0 seconds
[exercise31-1] [INFO] [1631462025.206108691] [sub_node]: SLEEPING==1 seconds
[exercise31-1] [INFO] [1631462026.207503346] [sub_node]: SLEEPING==2 seconds
[exercise31-1] [INFO] [1631462027.209666668] [sub_node]: SLEEPING==3 seconds
[exercise31-1] [INFO] [1631462028.211459854] [sub_node]: SLEEPING==4 seconds
[exercise31-1] [INFO] [1631462029.213224914] [sub_node]: SLEEPING==5 seconds
[exercise31-1] [INFO] [1631462030.215086652] [sub_node]: SLEEPING==6 seconds
[exercise31-1] [INFO] [1631462031.215881326] [sub_node]: SLEEPING==7 seconds
[exercise31-1] [INFO] [1631462032.217448376] [sub_node]: SLEEPING==8 seconds
[exercise31-1] [INFO] [1631462033.218958477] [sub_node]: SLEEPING==9 seconds

# And now that the scan finally was called, and the sensor value updated
# The Timer Callback can now move straight.
[exercise31-1] [INFO] [1631462034.222230681] [sub_node]: Odom CallBack
[exercise31-1] [INFO] [1631462034.224277957] [sub_node]: Scan CallBack
[exercise31-1] [INFO] [1631462034.226654888] [sub_node]: Timer CallBack
[exercise31-1] [WARN] [1631462034.228090777] [sub_node]: >>>>>>>>>>>>>>RANGES Value=13.314866065979004
[exercise31-1] [INFO] [1631462034.229608321] [sub_node]: MOVE STRAIGHT
[exercise31-1] [INFO] [1631462034.232116453] [sub_node]: Odom CallBack
[exercise31-1] [INFO] [1631462034.234230636] [sub_node]: Scan CallBack
```



As you can see, it **happens the same as before, as it should**.

This is because you have changed nothing. You are still using a **Single Thread** executor, with all the Callbacks in the same group. So nothing changed in the Node.

Now fix this issue in the next exercise.

- End of Example 5.2 -

- Exercise 5.3 -

- Create a new Python program called `exercise53.py` and copy the code of `exercise52.py`.
- The new program **creates three different MutuallyExclusiveCallbackGroups**, one for each Callback.
- Also, change the executor to a **MultiThreadedExecutor**.
- Create a launch file named `exercise53.launch.py` to launch it.
- Remember to add an entry point in the `setup.py`.
- Compile and run your code and see what happens now with the Callbacks.

- End of Exercise 5.3 -

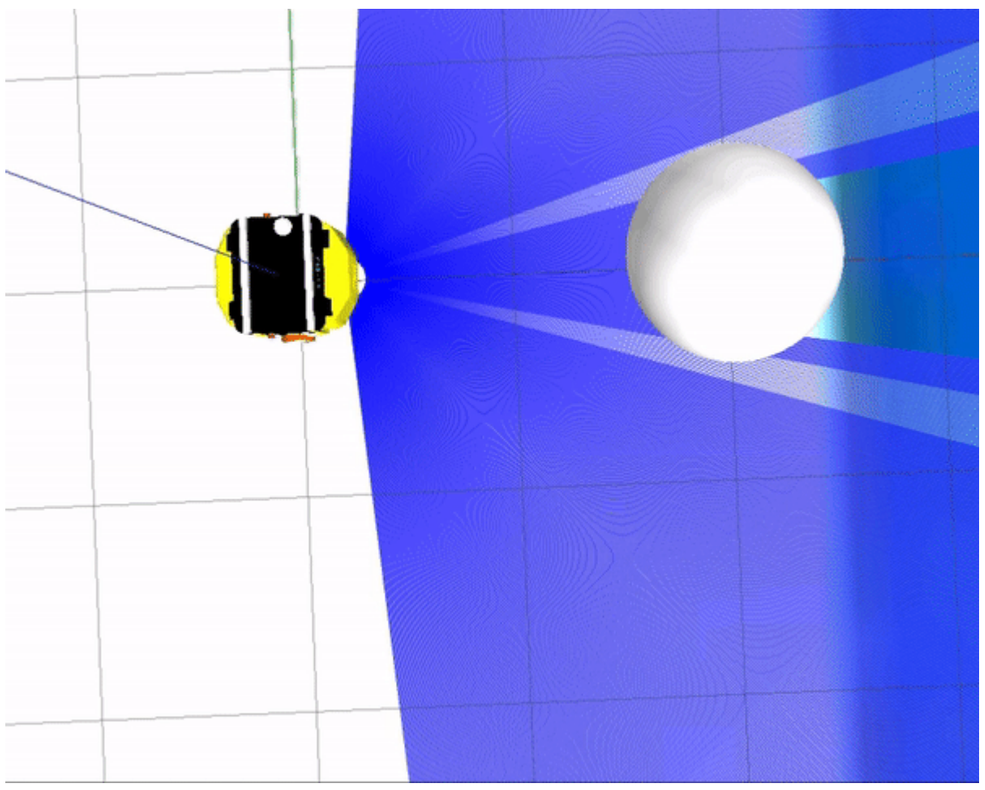
- Expected Behavior -

This time, the Sensor Callbacks do not get blocked by the main thread:

Output in Shell

```
[exercise31-1] [INFO] [1631462014.163838534] [sub_node]: Odom CallBack
[exercise31-1] [INFO] [1631462014.167065180] [sub_node]: Scan CallBack
[exercise31-1] [INFO] [1631462014.173721644] [sub_node]: Odom CallBack
[exercise31-1] [INFO] [1631462014.177145493] [sub_node]: Scan CallBack

# Here comes the timer and gets inside the STOP ROTATE
[exercise31-1] [INFO] [1631462014.183357178] [sub_node]: Timer CallBack
[exercise31-1] [WARN] [1631462014.183843557] [sub_node]: >>>>>>>>>>>>>>RANGES Value=0.43359941244125366
[exercise31-1] [INFO] [1631462014.184297364] [sub_node]: STOP ROTATE
[exercise31-1] [INFO] [1631462014.184905910] [sub_node]: Rotating for 10 seconds
[exercise31-1] [INFO] [1631462014.185361387] [sub_node]: SLEEPING==0 seconds
[exercise53-1] [DEBUG] [1673892811.163895369] [sub_node]: Odom CallBack
[exercise53-1] [DEBUG] [1673892811.164758000] [sub_node]: Scan CallBack
[exercise53-1] [DEBUG] [1673892811.167495877] [sub_node]: Scan CallBack
[exercise53-1] [DEBUG] [1673892811.169094602] [sub_node]: Odom CallBack
...
[exercise31-1] [INFO] [1631462015.187221747] [sub_node]: SLEEPING==1 seconds
[exercise53-1] [DEBUG] [1673892811.163895369] [sub_node]: Odom CallBack
[exercise53-1] [DEBUG] [1673892811.164758000] [sub_node]: Scan CallBack
[exercise53-1] [DEBUG] [1673892811.167495877] [sub_node]: Scan CallBack
[exercise53-1] [DEBUG] [1673892811.169094602] [sub_node]: Odom CallBack
...
[exercise31-1] [INFO] [1631462015.187221747] [sub_node]: SLEEPING==2 seconds
[exercise53-1] [DEBUG] [1673892811.163895369] [sub_node]: Odom CallBack
[exercise53-1] [DEBUG] [1673892811.164758000] [sub_node]: Scan CallBack
[exercise53-1] [DEBUG] [1673892811.167495877] [sub_node]: Scan CallBack
[exercise53-1] [DEBUG] [1673892811.169094602] [sub_node]: Odom CallBack
...
```



- End of Expected Behavior -

- Notes -

In [ ]:

```
executor = MultiThreadedExecutor(num_threads=4)
```

This will set the executor to a MultiThreadedExecutor. You can state how many threads it will have. If none are stated, `multiprocessing.cpu_count()` will extract from the system how many are available.

- You should contain these threads to avoid depleting the resources from your system unless you need them.

Here you have an example that you can try yourself:

Execute in Shell

In [ ]:

```
python3
```

The Python3 interpreter will open. Write the following lines of code:

In [ ]:

```
>>> import multiprocessing
>>> multiprocessing.cpu_count()
```

You will get the following output:

Output in Shell

```
Python 3.10.6 (main, Aug 10 2022, 11:40:04) [GCC 11.3.0] onlinuxType "help", "copyright", "credits" or "license" for mo
re information.
>>> import multiprocessing
>>> multiprocessing.cpu_count()
8
```

To exit the Python interpreter, press **Ctrl+D**.

This is how you create groups for Callbacks.

- Place a Callback in each group, depending on your needs.
- More groups mean more threads are needed and, therefore, more resources.

In [ ]:

```
self.group1 = MutuallyExclusiveCallbackGroup()
self.group2 = MutuallyExclusiveCallbackGroup()
self.group3 = MutuallyExclusiveCallbackGroup()
```

Here you have the documentation to get an idea of the parameters you will need:

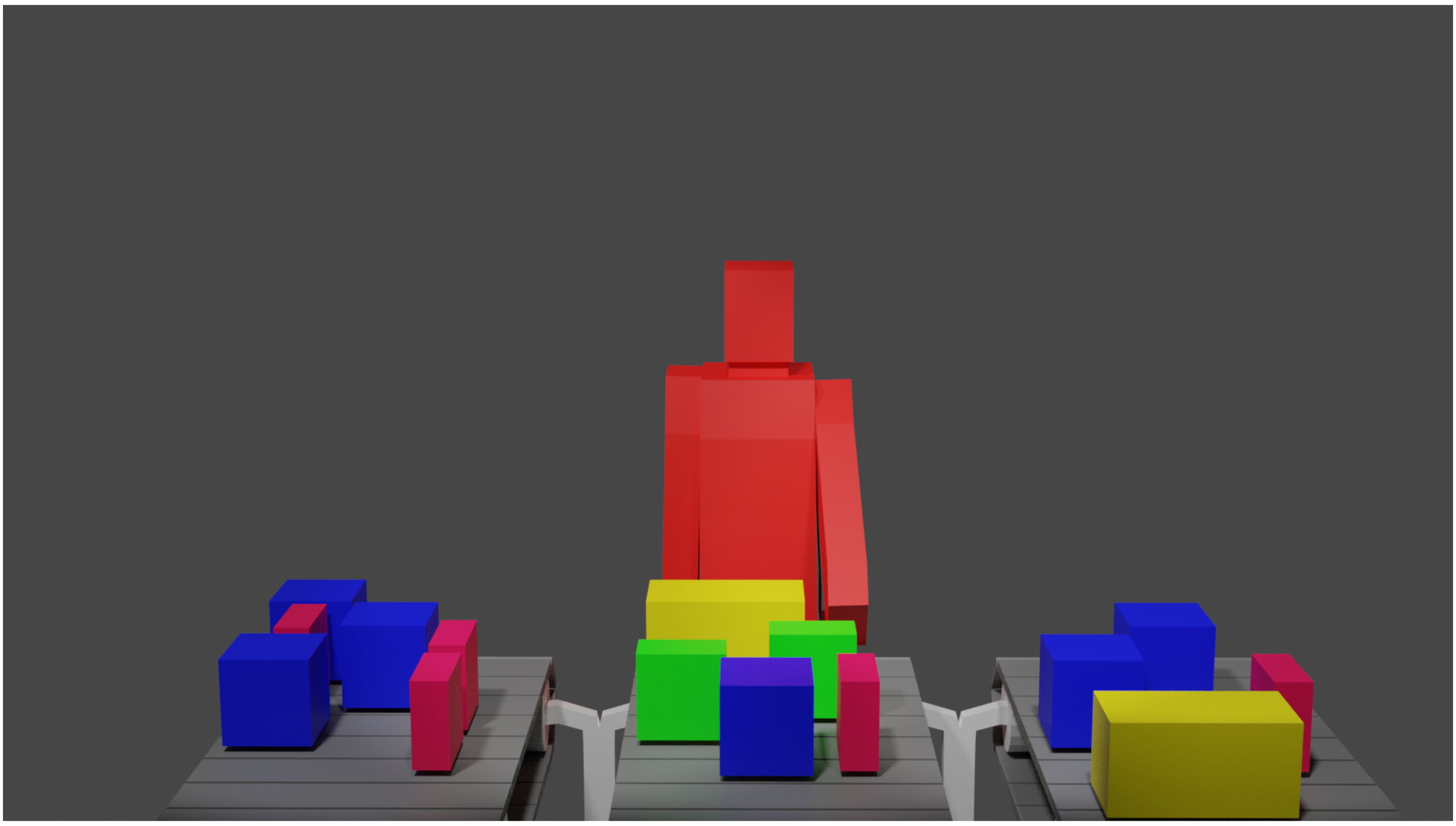
RCLPY Documentation

Why do you need a **MultithreadedExecutor** and **multiple Callback Groups**?

Suppose you set **num_threads=3**. Then, you will have three workers in the multi-threaded executor. And the **conveyor belts** depicted here are the **Callback Groups**. So, in this case, you also consider that you have **three**.

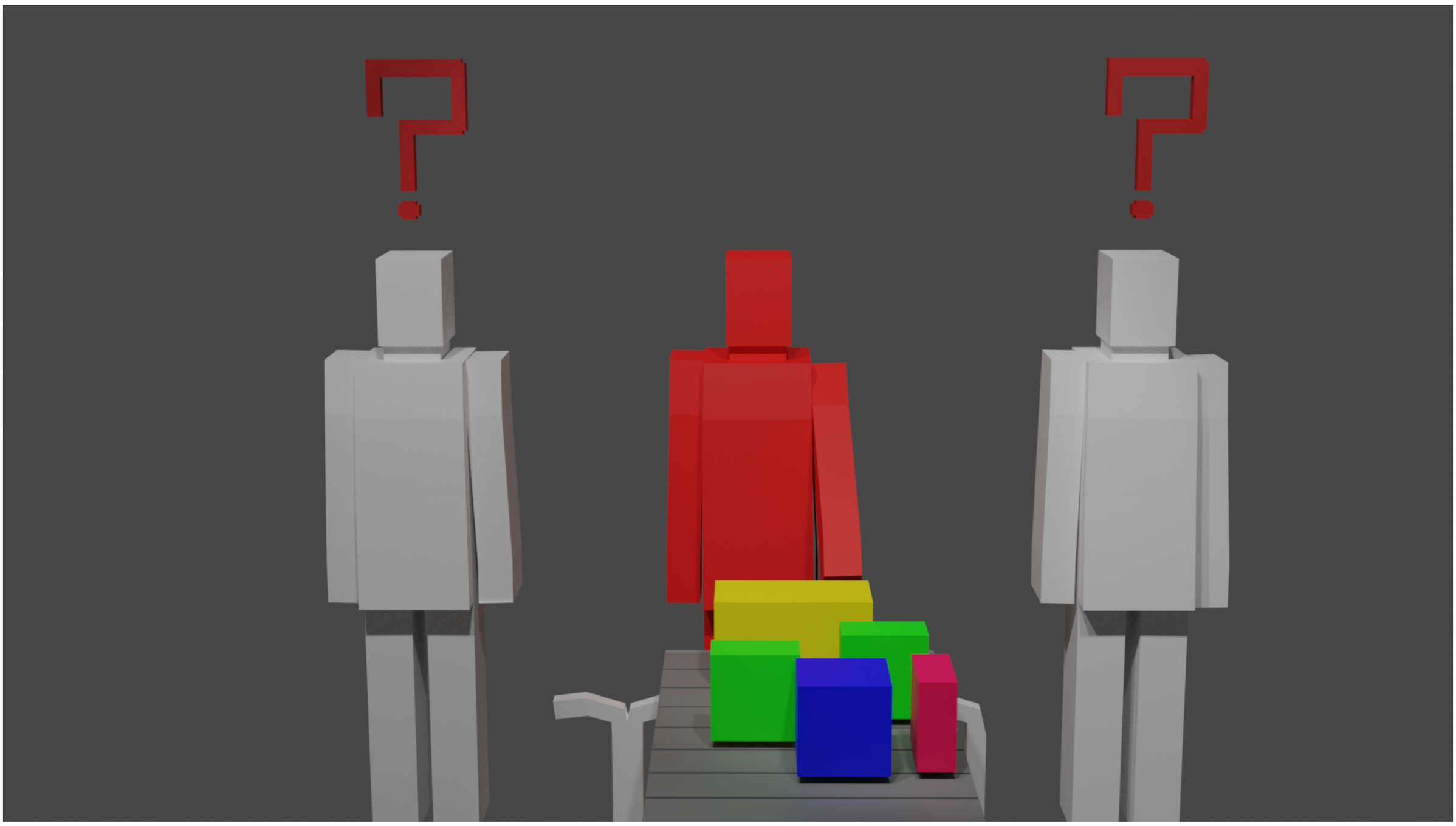
If you have separate Callback Groups, each process is set onto different queues, but only one executor will be available to process the methods from multiple queues.

It is like having **ONE worker** work in **three different areas of the factory**. It can only work in one area at a time.

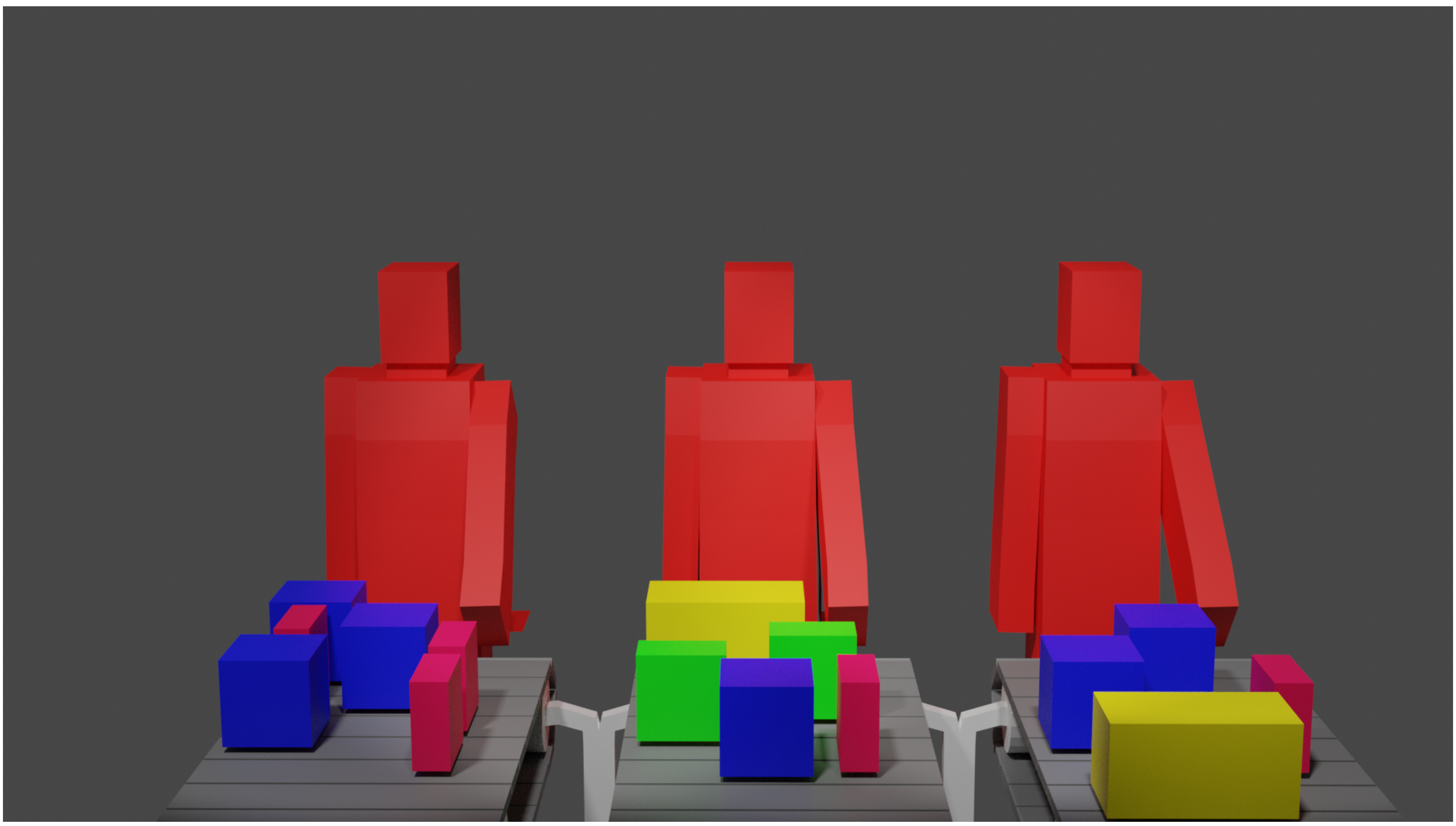


On the other hand, if you have a multi-threaded executor but only one Callback Group, there will be only one queue where the process gets put in.

It is like having **MULTIPLE workers** but only one area to work in with one chair. They will compete to see who can sit and work.



You need **MULTIPLE workers** and **MULTIPLE Areas** to use your resources efficiently.



Depending on the morphology of your system, you must decide how many **num_threads** (workers) you will need for the number of **queues or Callback Groups** you set. The best advice is to use the minimum amount of threads you need to have concurrent data input, and not lose your data or slow the algorithm down.

3D models created by: https://skfb.ly/onJvu, https://skfb.ly/6ZQXX, and https://skfb.ly/6SMCp.

- End of Notes -

- Solution for Exercise 5.3 -

exercise53.py

In [ ]:

```
import rclpy
from rclpy.node import Node
import time
import numpy as np
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from rclpy.qos import ReliabilityPolicy, QoSProfile
# Import the libraries to use executors and callback groups
from rclpy.callback_groups import ReentrantCallbackGroup, MutuallyExclusiveCallbackGroup
from rclpy.executors import MultiThreadedExecutor, SingleThreadedExecutor


class ControlClass(Node):

    def __init__(self, seconds_sleeping=10):
        super().__init__('sub_node')
        self._seconds_sleeping = seconds_sleeping
        # Define a Publisher for the /cmd_vel topic
        self.vel_pub = self.create_publisher(Twist, 'cmd_vel', 10)
        self.cmd = Twist()
        # Create 3 MutuallyExclusiveCallbackGroup
        self.group1 = MutuallyExclusiveCallbackGroup()
        self.group2 = MutuallyExclusiveCallbackGroup()
        self.group3 = MutuallyExclusiveCallbackGroup()
        # Define a Subscriber for the /odom topic and add it to the callback group
        self.odom_sub = self.create_subscription(
            Odometry, 'odom', self.odom_callback, 10,  callback_group=self.group1)
        # Define a Subscriber for the /scan topic and add it to the callback group
        self.scan_sub = self.create_subscription(LaserScan, 'scan', self.scan_callback, QoSProfile(
            depth=10, reliability=ReliabilityPolicy.BEST_EFFORT), callback_group=self.group2)
        # Define a timer object and add it to the callback group
        self.timer = self.create_timer(
            0.5, self.timer_callback, callback_group=self.group3)
        self.laser_msg = LaserScan()
        self.roll = 0.0
        self.pitch = 0.0
        self.yaw = 0.0

    # Callback function for the /odom Subscriber
    def odom_callback(self, msg):
        self.get_logger().info("Odom CallBack")
        orientation_q = msg.pose.pose.orientation
        orientation_list = [orientation_q.x,
                            orientation_q.y, orientation_q.z, orientation_q.w]
        (self.roll, self.pitch, self.yaw) = self.euler_from_quaternion(orientation_list)

    # Callback function for the /scan Subscriber
    def scan_callback(self, msg):
        self.get_logger().info("Scan CallBack")
        self.laser_msg = msg

    # Get the value of the front laser
    def get_front_laser(self):
        return self.laser_msg.ranges[360]

    # Get the yaw value
    def get_yaw(self):
        return self.yaw

    # Convert a quaternion to Euler angles
    def euler_from_quaternion(self, quaternion):
        """
        Converts quaternion (w in last place) to Euler roll, pitch, yaw
        quaternion = [x, y, z, w]
        Below should be replaced when porting for ROS2 Python tf_conversions is done.
        """
        x = quaternion[0]
        y = quaternion[1]
        z = quaternion[2]
        w = quaternion[3]

        sinr_cosp = 2 * (w * x + y * z)
        cosr_cosp = 1 - 2 * (x * x + y * y)
        roll = np.arctan2(sinr_cosp, cosr_cosp)

        sinp = 2 * (w * y - z * x)
        pitch = np.arcsin(sinp)

        siny_cosp = 2 * (w * z + x * y)
        cosy_cosp = 1 - 2 * (y * y + z * z)
        yaw = np.arctan2(siny_cosp, cosy_cosp)

        return roll, pitch, yaw

    # Send velocities to stop the robot
    def stop_robot(self):
        self.cmd.linear.x = 0.0
        self.cmd.angular.z = 0.0
        self.vel_pub.publish(self.cmd)

    # Send velocities to move the robot forward
    def move_straight(self):
        self.cmd.linear.x = 0.08
        self.cmd.angular.z = 0.0
        self.vel_pub.publish(self.cmd)

    # Send velocities to rotate the robot
    def rotate(self):
        self.cmd.angular.z = -0.2
        self.cmd.linear.x = 0.0
        self.vel_pub.publish(self.cmd)

        self.get_logger().info("Rotating for "+str(self._seconds_sleeping)+" seconds")
        # Keep rotating the robot for self._seconds_sleeping seconds
        for i in range(self._seconds_sleeping):
            self.get_logger().info("SLEEPING=="+str(i)+" seconds")
            time.sleep(1)

        self.stop_robot()

    # Callback for the Timer object
    def timer_callback(self):
        self.get_logger().info("Timer CallBack")
        try:
            self.get_logger().warning(">>>>>>>>>>>>>>>RANGES Value=" +
                                      str(self.laser_msg.ranges[360]))
            if not self.laser_msg.ranges[360] < 0.5:
                self.get_logger().info("MOVE STRAIGHT")
                self.move_straight()
            else:
                self.get_logger().info("STOP ROTATE")
                self.stop_robot()
                self.rotate()
        except:
            pass
```

```
def main(args=None):
    rclpy.init(args=args)
    control_node = ControlClass()
    # Create a MultiThreadedExecutor with 4 threads
    executor = MultiThreadedExecutor(num_threads=4)
    # Add the node to the executor
    executor.add_node(control_node)
    try:
        # Spin the executor
        executor.spin()
    finally:
        # Shutdown the executor
        executor.shutdown()
        control_node.destroy_node()
    rclpy.shutdown()


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

exercise53.launch.py

In [ ]:

```
from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description():
    return LaunchDescription([
        Node(
            package='unit5_pkg',
            executable='exercise53',
            output='screen',
            emulate_tty=True),
    ])
```

- End of Solution -

5.3 CallBack Groups Detail

You have used only **MutuallyExclusiveCallbackGroup** Callback Groups. This is because they are the default type in ROS2. However, there is another type called **ReentrantCallbackGroup**.

The main difference is:

- `ReentrantCallbackGroup` : Any Callback inside this group can be executed in parallel if there are enough threads. For example, if you add three Callbacks inside the same **ReentrantCallbackGroup** and have **two threads**, you can simultaneously execute **TWO** of the **THREE** Callbacks.
- `MutuallyExclusiveCallbackGroup` : All the Callbacks inside this group will only be executed one by one in that group, no matter how many threads you have. For example, if you have **three Callbacks inside this group** and **three threads**, only **ONE Callback at a time will be executed**.

- Example 5.3 -

Create a script that gives you an example of using these two different Callback Groups:

- In this script, you have **one service server** and **one timer Callback**.
- You can change some elements of the times, the number of threads, and the type of Callbacks through arguments.

1. Create a new file named **callback_groups_examples.py** with the code shown below.

callback_groups_examples.py

In [ ]:

```
# import the SetBool module from std_servs service interface
from pickle import TRUE
from std_srvs.srv import SetBool
import rclpy
from rclpy.node import Node
import time

from rclpy.callback_groups import ReentrantCallbackGroup, MutuallyExclusiveCallbackGroup
from rclpy.executors import MultiThreadedExecutor, SingleThreadedExecutor
from rclpy.qos import ReliabilityPolicy, QoSProfile

import argparse


class DummyServer(Node):

    def __init__(self, args, callback_group_type="reentrant"):

        self.timer_flag = True

        super().__init__('service_start_turn')

        # More info here: https://docs.python.org/3/library/argparse.html
        parser = argparse.ArgumentParser(
            description='Dummy Server to Learn about Callback Groups and Threads')

        # Add an argument for setting the service waiting time
        parser.add_argument('-service_wait_time',
                            type=float,
                            help='Time the service will be waiting',
                            required=True)

        # Add an argument for setting por of the timer callback
        parser.add_argument('-timer_period',
                            type=float,
                            nargs=1,
                            metavar='TIMEOUT',
                            default=1.0,
                            help="Time period of the Callback for the timer")

        # Add an argument for setting the callback group type
        parser.add_argument('-callback_group_type',
                            type=str,
                            default="reentrant",
                            help="Type of Callback Groups REENTRANT of EXCLUSIVE")

        # Add an argument for setting the number of threads
        parser.add_argument('-threads',
                            type=int,
                            default=1,
                            help="Number of threads to use in the executor")

        self.args = parser.parse_args(args[1:])

        parser.print_help()

        # <rclpy.callback_groups.MutuallyExclusiveCallbackGroup object at 0x7ff58fc9e8e0>
        # By default, the Callbacks are mutually exclusive. This means that in each group, only
        # one Callback can be done: https://docs.ros2.org/foxy/api/rclpy/api/node.html
        print("## DEFAULT Node Callback Group=" +
              str(self.default_callback_group))

        self.get_logger().warning("Setting "+self.args.callback_group_type+" Groups")
        if self.args.callback_group_type == "reentrant":
            # If you set the group reentrant, any Callback inside will be executed in parallel
            # If there are enough threads
            self.group1 = ReentrantCallbackGroup()
            self.get_logger().warning("ReentrantCallbackGroup Set")
            # Both the service and the timer use the same callback group
            self.srv = self.create_service(
                SetBool, '/dummy_server_srv', self.SetBool_callback, callback_group=self.group1)
            self.timer = self.create_timer(
                self.args.timer_period[0], self.timer_callback, callback_group=self.group1)

        elif self.args.callback_group_type == "exclusive":
            self.group1 = MutuallyExclusiveCallbackGroup()
            self.group2 = MutuallyExclusiveCallbackGroup()
            self.get_logger().warning("MutuallyExclusiveCallbackGroup Set")
            # Set one group for the service and another one for the timer
            self.srv = self.create_service(
                SetBool, '/dummy_server_srv', self.SetBool_callback, callback_group=self.group1)
            self.timer = self.create_timer(
                self.args.timer_period[0], self.timer_callback, callback_group=self.group2)

        else:
            # You do not set groups. Therefore, they will get the default group for the Node
            self.get_logger().error("NO GROUPS SET Set")
            self.srv = self.create_service(
                SetBool, '/dummy_server_srv', self.SetBool_callback)
            self.timer = self.create_timer(
                self.args.timer_period[0], self.timer_callback)

    def get_threads(self):
        return self.args.threads

    def SetBool_callback(self, request, response):
        self.get_logger().warning("Processing Server Message...")
        self.wait_for_sec(self.args.service_wait_time)
        self.get_logger().warning("Processing Server Message...DONE")
        response.message = 'TURNING'
        # return the response parameters
        return response

    def wait_for_sec(self, wait_sec, delta=1.0):
        i = 0
        while i < wait_sec:
            self.get_logger().info("..."+str(i)+"[WAITING...]")
            time.sleep(delta)
            i += delta

    def timer_callback(self):
        self.print_dummy_msgs()

    def print_dummy_msgs(self):
        if self.timer_flag:
            self.get_logger().info("TICK")
            self.timer_flag = False
        else:
            self.get_logger().info("TACK")
            self.timer_flag = True


def main(args=None):
    # To Use: ros2 service call /dummy_server_srv std_srvs/srv/SetBool data:\ false\
    # ros2 run unit5_pkg callback_groups_examples -service_wait_time 5.0 -timer_period 1.0
    # initialize the ROS communication
    rclpy.init(args=args)
```

```
    print("args==="+str(args))
    # Format the arguments given through ROS to use the arguments
    args_without_ros = rclpy.utilities.remove_ros_args(args)
    print("clean ROS args==="+str(args_without_ros))
    start_stop_service_node = DummyServer(args_without_ros)

    num_threads = start_stop_service_node.get_threads()
    start_stop_service_node.get_logger().info(
        'DummyServer Started with threads='+str(num_threads))

    executor = MultiThreadedExecutor(num_threads=num_threads)
    executor.add_node(start_stop_service_node)

    try:
        executor.spin()
    finally:
        executor.shutdown()
        start_stop_service_node.destroy_node()

    # shutdown the ROS communication
    rclpy.shutdown()


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

2. Modify the setup.py to add the launch file you created, and the entry points to the executable for the **callback_groups_examples.py** script.

In [ ]:

```
from setuptools import setup
import os
from glob import glob

package_name = 'unit5_pkg'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'), glob('launch/*.launch.py'))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='somebody very awesome',
    maintainer_email='user@user.com',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'exercise51 = unit5_pkg.exercise51:main',
            'exercise52 = unit5_pkg.exercise52:main',
            'exercise53 = unit5_pkg.exercise53:main',
            'callback_groups_examples = unit5_pkg.callback_groups_examples:main',
        ],
    },
)
```

3. Compile your package.

Execute in Shell

In [ ]:

```
cd ~/ros2_ws
colcon build --packages-select unit5_pkg
source ~/ros2_ws/install/setup.bash
```

5. Finally, launch the Node in your shell. Now, execute it like so and do some examples of execution and see the different results:

Test 1 Two Threads - Two MutuallyExclusiveCallbackGroups

Execute in Shell

In [ ]:

```
source ~/ros2_ws/install/setup.bash
# Different tests
ros2 run unit5_pkg callback_groups_examples -service_wait_time 5.0 -timer_period 1.0 -callback_group_type exclusive -threads 2
```

Output in Shell #1

```
args===None
clean ROS args===['/home/user/ros2_ws/install/example_3_4_pkg/lib/example_3_4_pkg/mutualexclusive_demo_fun', '-service_
wait_time', '5.0', '-timer_period', '1.0', '-callback_group_type', 'exclusive', '-threads', '2']
usage: mutualexclusive_demo_fun [-h] -service_wait_time SERVICE_WAIT_TIME [-timer_period TIMEOUT]
                                [-callback_group_type CALLBACK_GROUP_TYPE] [-threads THREADS]

Dummy Server to learn about Callback Groups and Threads

optional arguments:
  -h, --help            show this help message and exit
  -service_wait_time SERVICE_WAIT_TIME
                        Time the service will be waiting
  -timer_period TIMEOUT
                        Time period of the Callback for the timer
  -callback_group_type CALLBACK_GROUP_TYPE
                        Type of Callback Groups REENTRANT of EXCLUSIVE
  -threads THREADS      Number of threads to use in the executor
## DEFAULT Node Callback Group=
[WARN] [1642597535.656961711] [service_start_turn]: Setting exclusive Groups
[WARN] [1642597535.657662490] [service_start_turn]: MutuallyExclusiveCallbackGroup Set
[INFO] [1642597535.659347743] [service_start_turn]: DummyServer Started with threads=2
[INFO] [1642597536.660330787] [service_start_turn]: TICK
[INFO] [1642597537.660840205] [service_start_turn]: TACK
[INFO] [1642597538.660295927] [service_start_turn]: TICK
[INFO] [1642597539.660024152] [service_start_turn]: TACK
```

See the following:

- The number of threads set, in this case, **two**.
- The type of Callback Group, in this case, **MutuallyExclusiveCallbackGroup**.

Call the service. This way, you check how it behaves, executing two Callbacks at the same time:

Execute in Shell

In [ ]:

```
ros2 service call /dummy_server_srv std_srvs/srv/SetBool data:\ false\
```

Output in Shell

As you can see, it executes both Callbacks simultaneously. This is because:

- Set **two** threads
- Set the **MutuallyExclusiveCallback** type, and each Callback is in a different group.

You can execute **BOTH Callbacks at the same time**.

Test 2 Two Threads - One ReentrantCallbackGroup

Execute in Shell

In [ ]:

```
ros2 run unit5_pkg callback_groups_examples -service_wait_time 5.0 -timer_period 1.0 -callback_group_type reentrant -threads 2
```

Output in Shell

```
args===None
clean ROS args===['/home/user/ros2_ws/install/example_3_4_pkg/lib/example_3_4_pkg/mutualexclusive_demo_fun', '-service_
wait_time', '5.0', '-timer_period', '1.0', '-callback_group_type', 'reentrant', '-threads', '2']
usage: mutualexclusive_demo_fun [-h] -service_wait_time SERVICE_WAIT_TIME [-timer_period TIMEOUT]
                                [-callback_group_type CALLBACK_GROUP_TYPE] [-threads THREADS]

Dummy Server to learn about Callback Groups and Threads

optional arguments:
  -h, --help            show this help message and exit
  -service_wait_time SERVICE_WAIT_TIME
                        Time the service will be waiting
  -timer_period TIMEOUT
                        Time period of the Callback for the timer
  -callback_group_type CALLBACK_GROUP_TYPE
                        Type of Callback Groups REENTRANT of EXCLUSIVE
  -threads THREADS      Number of threads to use in the executor
## DEFAULT Node Callback Group=
[WARN] [1642598228.281397084] [service_start_turn]: Setting reentrant Groups
[WARN] [1642598228.281990686] [service_start_turn]: ReentrantCallbackGroup Set
[INFO] [1642598228.283569796] [service_start_turn]: DummyServer Started with threads=2
[INFO] [1642598229.284564388] [service_start_turn]: TICK
[INFO] [1642597537.660840205] [service_start_turn]: TACK
[INFO] [1642597538.660295927] [service_start_turn]: TICK
[INFO] [1642597539.660024152] [service_start_turn]: TACK
```

Here you can see the following:

- The number of threads set, in this case, **two**.
- The type of Callback Group, in this case, **ReentrantCallbackGroup**.
- Set both Callbacks in the same group. You do not need multiple groups.

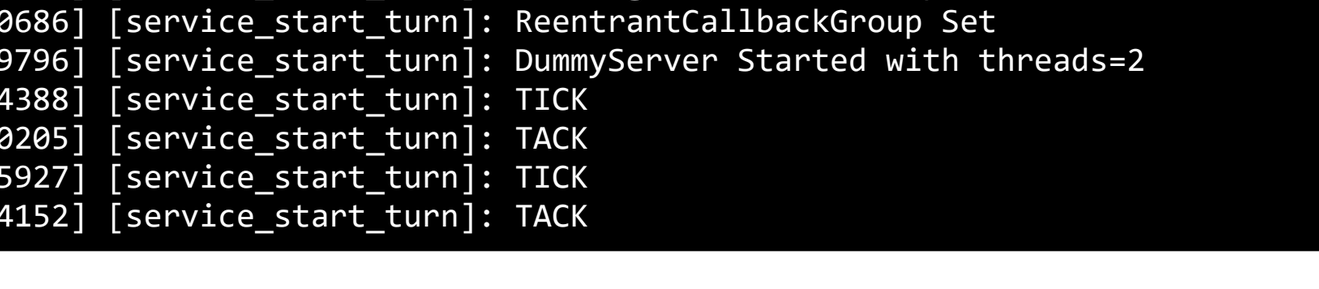
Call the service. This way, you check how it behaves executing two Callbacks at the same time:

Execute in Shell

In [ ]:

```
ros2 service call /dummy_server_srv std_srvs/srv/SetBool data:\ false\
```

Output in Shell



As you can see, both Callbacks simultaneously execute. This is because:

- Set **two** threads
- Set **ReentrantCallbackGroup** type, and both Callbacks are in this reentrant group, so they can be executed in parallel while you have enough threads.

You can execute **BOTH Callbacks at the same time**.

Test 3 One Thread - One ReentrantCallbackGroup

Execute in Shell

In [ ]:

```
ros2 run unit5_pkg callback_groups_examples -service_wait_time 5.0 -timer_period 1.0 -callback_group_type reentrant -threads 1
```

Output in Shell

```
args===None
clean ROS args===['/home/user/ros2_ws/install/example_3_4_pkg/lib/example_3_4_pkg/mutualexclusive_demo_fun', '-service_
wait_time', '5.0', '-timer_period', '1.0', '-callback_group_type', 'reentrant', '-threads', '1']
usage: mutualexclusive_demo_fun [-h] -service_wait_time SERVICE_WAIT_TIME [-timer_period TIMEOUT]
                                [-callback_group_type CALLBACK_GROUP_TYPE] [-threads THREADS]

Dummy Server to learn about Callback Groups and Threads

optional arguments:
  -h, --help            show this help message and exit
  -service_wait_time SERVICE_WAIT_TIME
                        Time the service will be waiting
  -timer_period TIMEOUT
                        Time period of the Callback for the timer
  -callback_group_type CALLBACK_GROUP_TYPE
                        Type of Callbacks Groups REENTRANT of EXCLUSIVE
  -threads THREADS      Number of threads to use in the executor
## DEFAULT Node Callback Group=
[WARN] [1642598548.233686096] [service_start_turn]: Setting reentrant Groups
[WARN] [1642598548.234280348] [service_start_turn]: ReentrantCallbackGroup Set
[INFO] [1642598548.235857734] [service_start_turn]: DummyServer Started with threads=1
[INFO] [1642598549.237222539] [service_start_turn]: TICK
[INFO] [1642597537.660840205] [service_start_turn]: TACK
[INFO] [1642597538.660295927] [service_start_turn]: TICK
[INFO] [1642597539.660024152] [service_start_turn]: TACK
```

See the following:

- The number of threads set, in this case, **one**.
- The type of Callback Group, in this case, **ReentrantCallbackGroup**.
- Set both Callbacks in the same group. You do not need multiple groups.

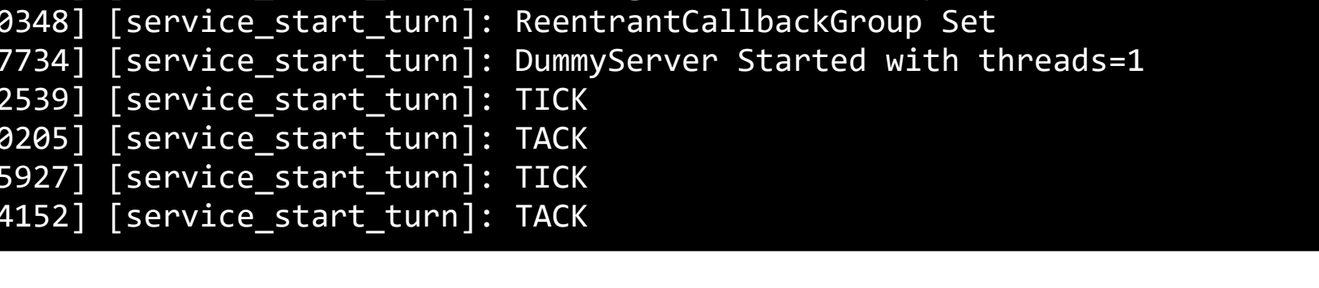
Call the service. This way, you check how it behaves executing two Callbacks at the same time:

Execute in Shell

In [ ]:

```
ros2 service call /dummy_server_srv std_srvs/srv/SetBool data:\ false\
```

Output in Shell



As you can see, now it only executes **ONE CALLBACK at a time**. This is because:

- Set **one** thread.
- Set **ReentrantCallbackGroup** type, and both Callbacks are in this reentrant group, so they can be executed in parallel while you have enough threads. However, you **DO NOT have enough threads**. You only have one, so the other Callback has to wait, even using **reentrant Callbacks**.

- End of Example 5.3 -