Robot Operating System (ROS2)

Vilma Muço

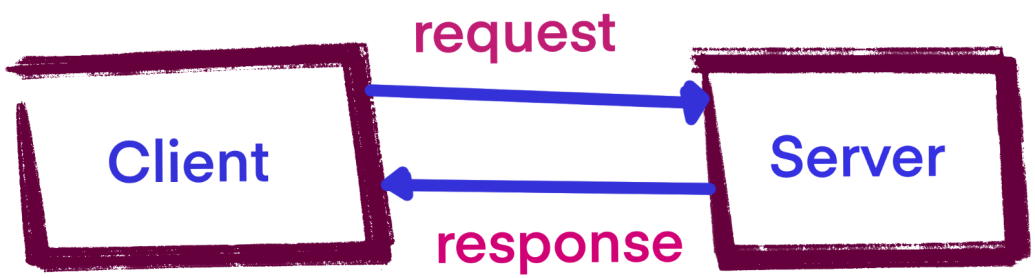
Université de Toulon, Master Ingénierie des Systèmes Complexes (ISC) -Erasmus Mundus MIR

November, 2024

Table of Contents

Services

- The node that sends a request is called the client node
- The node that provides the response is the server node
- The structure of the request and the response is defined in a .srv file



Turtlesim

```
$ros2 service list
$ros2 service type  /turtle1/set_pen
```

```
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 service list
/clear
/kill
/reset
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
/turtlesim/set_parameters
/turtlesim/set_parameters_atomically
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 service type  /turtle1/set_pen
turtlesim/srv/SetPen
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 interface show turtlesim/srv/SetPen
uint8 r
uint8 g
uint8 b
uint8 width
uint8 off
---
vilma@vilma-Precision-3571:~/ros2_ws$
```

Ros Help command

```
$ros2 -h
$ros2 service -h
```

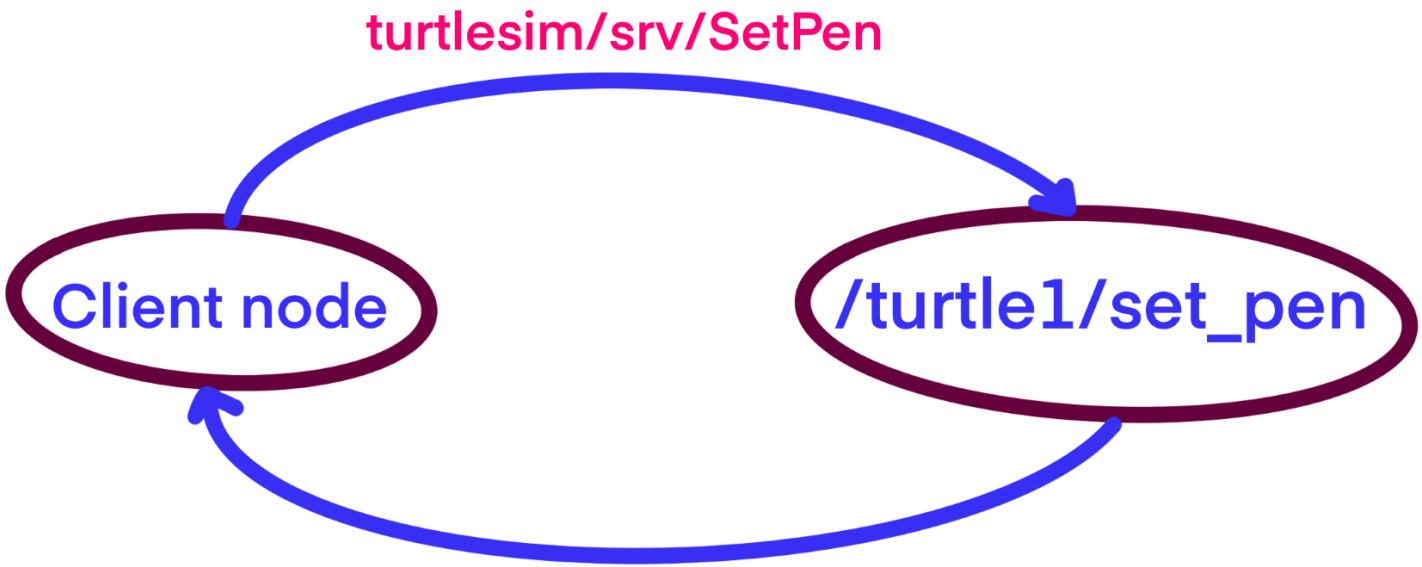
```
vilma@:~$ ros2 service -h
usage: ros2 service [-h] [--include-hidden-services] Call `ros2 service <command> -h` for more detai

Various service related sub-commands

options:
  -h, --help            show this help message and exit
  --include-hidden-services
                        Consider hidden services as well

Commands:
  call  Call a service
  find  Output a list of available services of a given type
  list  Output a list of available services
  type  Output a service's type

  Call `ros2 service <command> -h` for more detailed usage.
vilma@:~$
```


turtlesim/srv/SetPen
Client node
/turtle1/set_pen

Creating a new package

```
ros2 pkg create --build-type ament_python --dependencies rclpy  p_client_server
```

- Modify package.xml (add exec_depend for turtlesim) and setup.py
- Create the clientNode.py file
- Add an entry point for the client node

p_client_server > setup.py > ...

```python
from setuptools import find_packages, setup

package_name = 'p_client_server'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='vilma',
    maintainer_email='vilma.muco@exail.com',
    description='TODO: Package description',
    license='Apache License 2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'client_node = p_client_server.clientNode:main',
        ],
    },
)

```

Client Program

```python
import rclpy
from rclpy.node import Node
from turtlesim.srv import SetPen
import sys
import time

class ClientNode(Node):

    def __init__(self):
        super().__init__('client_node')
        self.cli = self.create_client(SetPen, '/test_namespace/turtle1/set_pen')
        while not self.cli.wait_for_service(timeout_sec=3.0):
            self.get_logger().info('service not available, waiting again...')
        self.req = SetPen.Request()

    def send_request(self, red, green, blue):
        self.req.r = red
        self.req.g = green
        self.req.b = blue
        self.req.width = 5
        self.req.off = 0
        self.future = self.cli.call_async(self.req)
        rclpy.spin_until_future_complete(self, self.future)

def main():
    rclpy.init()

    red = 255 # pen rgb colors
    green = 0
    blue = 0
    client_node = ClientNode()

    while True:
        # switch between red and green
        if red == 255:
            red = 0
        else:
            red = 255

        if green == 255:
            green = 0
        else:
            green = 255

        response = client_node.send_request(red, green, blue)
        time.sleep(0.2)

    client_node.destroy_node()
    rclpy.shutdown()

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

Code Analysis

```
1  import rclpy
2  from rclpy.node import Node
3  from turtlesim.srv import SetPen
4  import sys
5  import time
6
7  class ClientNode(Node):
8
9      def __init__(self):
10         super().__init__('client_node')
11         self.cli = self.create_client(SetPen, '/test_namespace/turtle1/set_pen')
12         while not self.cli.wait_for_service(timeout_sec=3.0):
13             self.get_logger().info('service not available, waiting again...')
14         self.req = SetPen.Request()
15
```

Code Analysis

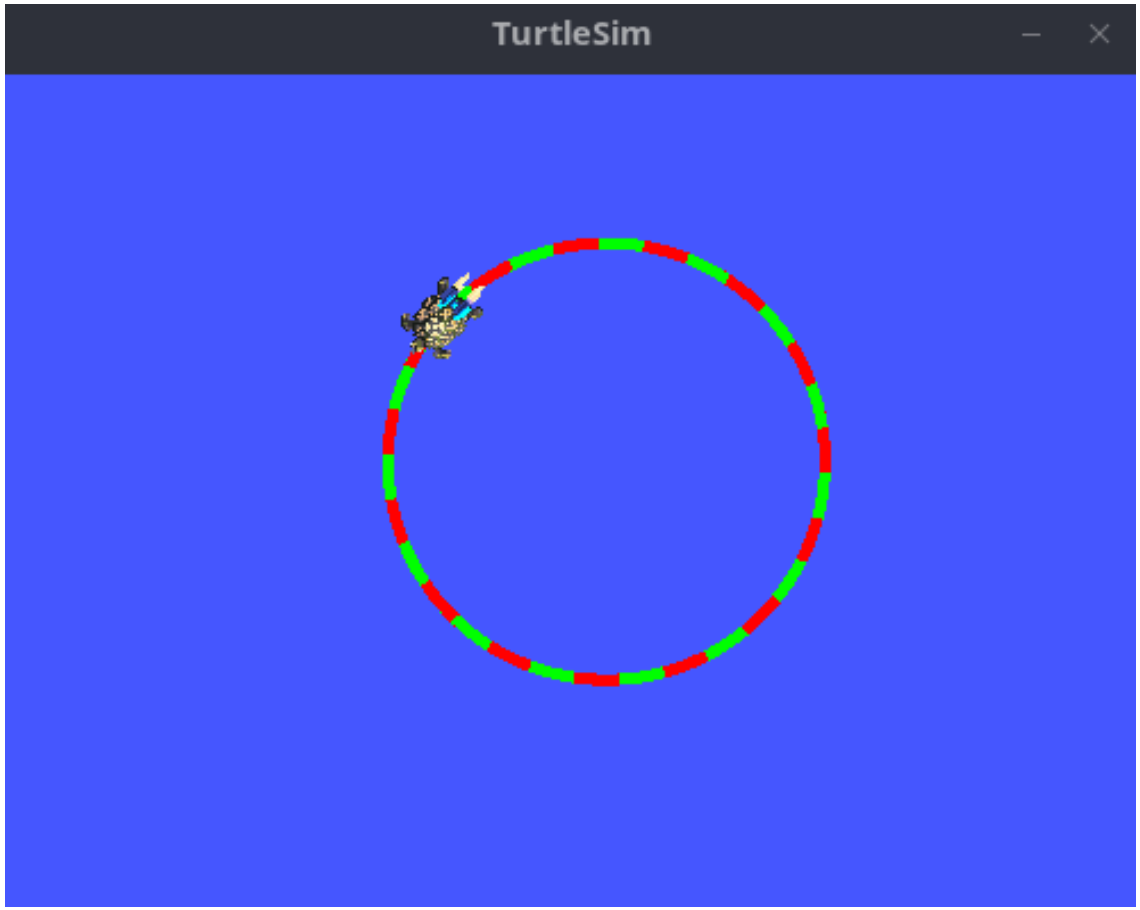
```
1       def send_request(self, red, green, blue):
2           self.req.r = red
3           self.req.g = green
4           self.req.b = blue
5           self.req.width = 5
6           self.req.off = 0
7           self.future = self.cli.call_async(self.req)
8           rclpy.spin_until_future_complete(self, self.future)
9
10      def main():
11          rclpy.init()
12          red = 255 # pen rgb colors
13          green = 0
14          blue = 0
15          client_node = ClientNode()
16
17              # switch between red and green
18          while True:
19              red, green = green, red
20              response = client_node.send_request(red, green, blue)
21              time.sleep(0.2)
22
23          client_node.destroy_node()
24          rclpy.shutdown()
25
26      if __name__ == '__main__':
27          main()
```

Run the program

- Launch the circle_path_launch.py file
- Build and run the client node

```
colcon build --packages-select p_client_server
ros2 run p_client_server client_node
```

Run the program



Custom message and service types

- Good practice to use predefined interface definitions
- For the moment there is no way of generating a .msg and a .srv file in a pure Python package
- We can only define new interfaces in a CMake package
- CMake is an open source, cross-platform family of tools designed to build, test, and package software.
- CMake gives you control of the software compilation process using simple independent configuration files.
- We can use ament_cmake_python to build interfaces and python code in the same package

- It is good practice to keep .msg and .srv files in their own directories within a package

```
$ros2 pkg create --build-type ament_cmake p_interfaces
$cd p_interfaces
$mkdir msg
$mkdir srv
```

```
vilma@vilma-Precision-3571:~/ros2_ws/p_interfaces$ mkdir srv
vilma@vilma-Precision-3571:~/ros2_ws/p_interfaces$ mkdir msg
vilma@vilma-Precision-3571:~/ros2_ws/p_interfaces$ ll
total 32
drwxrwxr-x  6 vilma vilma 4096 nov.   8 11:29 ./
drwxrwxr-x 10 vilma vilma 4096 nov.   8 11:25 ../
-rw-rw-r--  1 vilma vilma  905 nov.   8 11:25 CMakeLists.txt
drwxrwxr-x  3 vilma vilma 4096 nov.   8 11:25 include/
drwxrwxr-x  2 vilma vilma 4096 nov.   8 11:29 msg/
-rw-rw-r--  1 vilma vilma  603 nov.   8 11:25 package.xml
drwxrwxr-x  2 vilma vilma 4096 nov.   8 11:25 src/
drwxrwxr-x  2 vilma vilma 4096 nov.   8 11:29 srv/
vilma@vilma-Precision-3571:~/ros2_ws/p_interfaces$
```

- Create a new file Num.msg that transfers a single 64-bit integer called num:

```
int64 num
```

- Create a new file ChangeRadius.srv with the following request and response structure:

```
int64 radius
---
bool radius_changed
geometry_msgs/Vector3 linear_velocity
```

- Modify the CMakeLists.txt:

```
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(\${PROJECT_NAME}
  "msg/Num.msg"
  "srv/ChangeRadius.srv"
  DEPENDENCIES geometry_msgs # Add packages that above msg and srv
                            # depend on, in this case geometry_msgs
)
```

Package.xml

- Because the interfaces rely on rosidl_default_generators for generating language-specific code, you need to declare a build tool dependency on it
- rosidl_default_runtime is a runtime or execution-stage dependency, needed to be able to use the interfaces later
- rosidl_interface_packages is the name of the dependency group that your package, should be associated with, using the member_of_group tag.

```
<depend>geometry_msgs</depend>
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Build the p_interfaces package

```
colcon build --packages-select p_interfaces
source install/setup.bash
ros2 interface show p_interfaces/msg/Num
ros2 interface show p_interfaces/srv/ChangeRadius
```

```
vilma@vilma-Precision-3571:~/ros2_ws$ colcon build --packages-select p_interfaces
Starting >>> p_interfaces
Finished <<< p_interfaces [1.12s]

Summary: 1 package finished [1.21s]
vilma@vilma-Precision-3571:~/ros2_ws$ source install/setup.bash
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 interface show p_interfaces/msg/Num
int64 num
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 interface show p_interfaces/srv/ChangeRadius
int64 radius
---
bool radius_changed
geometry_msgs/Vector3 linear_velocity
        float64 x
        float64 y
        float64 z
vilma@vilma-Precision-3571:~/ros2_ws$
```

Test the new Interface

In order to test the new Interface:

- Add a service to the circleController node to change the radius of the trajectory of the turtle.

Test the new Interface

```
1  import rclpy
2  from rclpy.node import Node
3  from geometry_msgs.msg import Twist
4
5  from p_interfaces.srv import ChangeRadius
6
7
8  class CircleController(Node):
9      def __init__(self):
10         super().__init__('circle_controller')
11         self.publisher_ = self.create_publisher(Twist, 'turtle1/cmd_vel', 10)
12         timer_period = 0.1  # seconds
13         self.timer = self.create_timer(timer_period, self.timer_callback)
14         self.srv = self.create_service(ChangeRadius, 'change_radius', self.change_radius_callback)
15         self.radius = 1.0
16
```

Test the new Interface

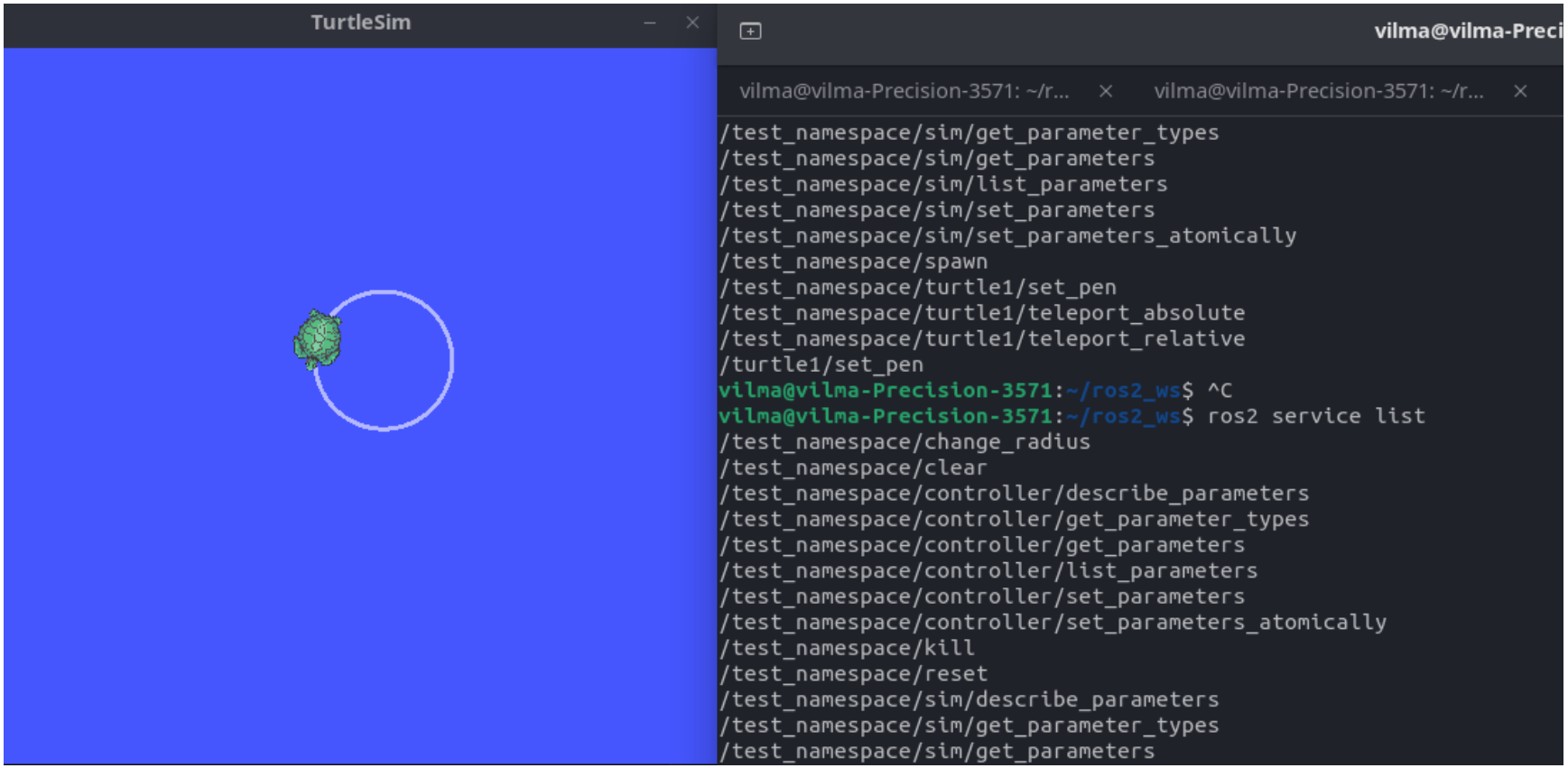
```
1     def timer_callback(self):
2         vel = Twist()
3         vel.linear.x = self.radius * 1.0  # linear velocity = radius * angular velocity
4         vel.angular.z = 1.0
5         self.publisher_.publish(vel)
6
7     def change_radius_callback(self, request, response):
8         self.radius = request.radius
9         response.linear_velocity.x = self.radius * 1.0
10        response.linear_velocity.y = 0.0
11        response.linear_velocity.z = 1.0
12        response.success = True
13        return response
14
```

Build the code

- Modify the package.xml
- Build the package

```
<exec_depend>p_interfaces</exec_depend>
colcon build --packages-select p_controller
source install/setup.bash
ros2 launch p_controller/p_controller/launch/circle_path_launch.py
```

Build the code



ros2 service call

- We can use the command line to request a service

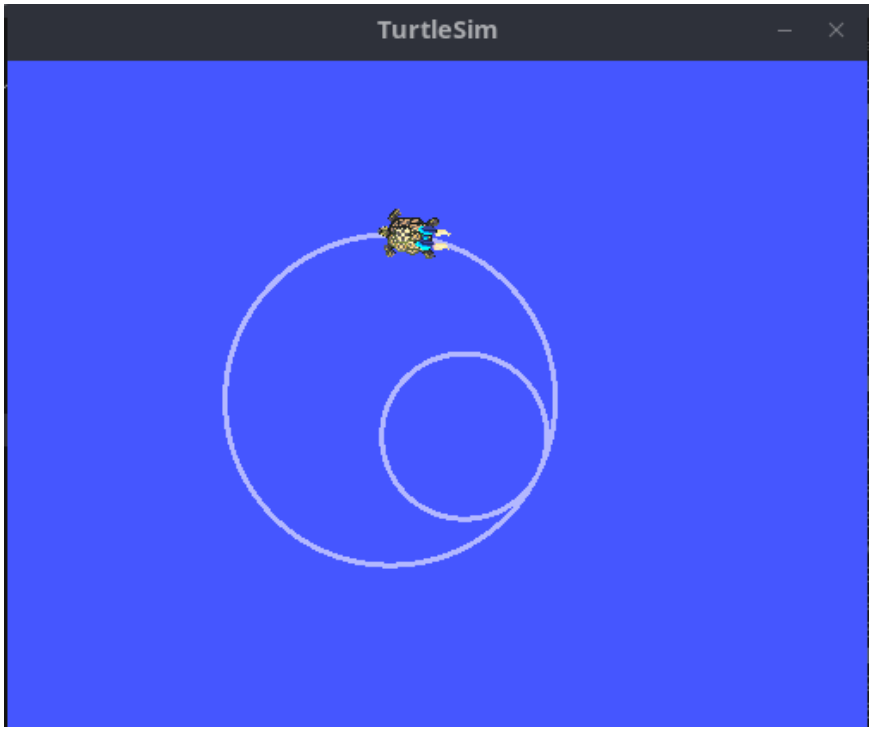
```
$ros2 service call <service_name> <service_type> <arguments>
$ros2 service call /test_namespace/change_radius p_interfaces/srv/ChangeRadius "{radius : 2}"
```

```
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 service call /test_namespace/change_radius p_interfaces/srv/ChangeRadius "{radius : 2
}"
waiting for service to become available...
requester: making request: p_interfaces.srv.ChangeRadius_Request(radius=2)

response:
p_interfaces.srv.ChangeRadius_Response(radius_changed=True, linear_velocity=geometry_msgs.msg.Vector3(x=2.0, y=0.0, z=1.0))

vilma@vilma-Precision-3571:~/ros2_ws$
```

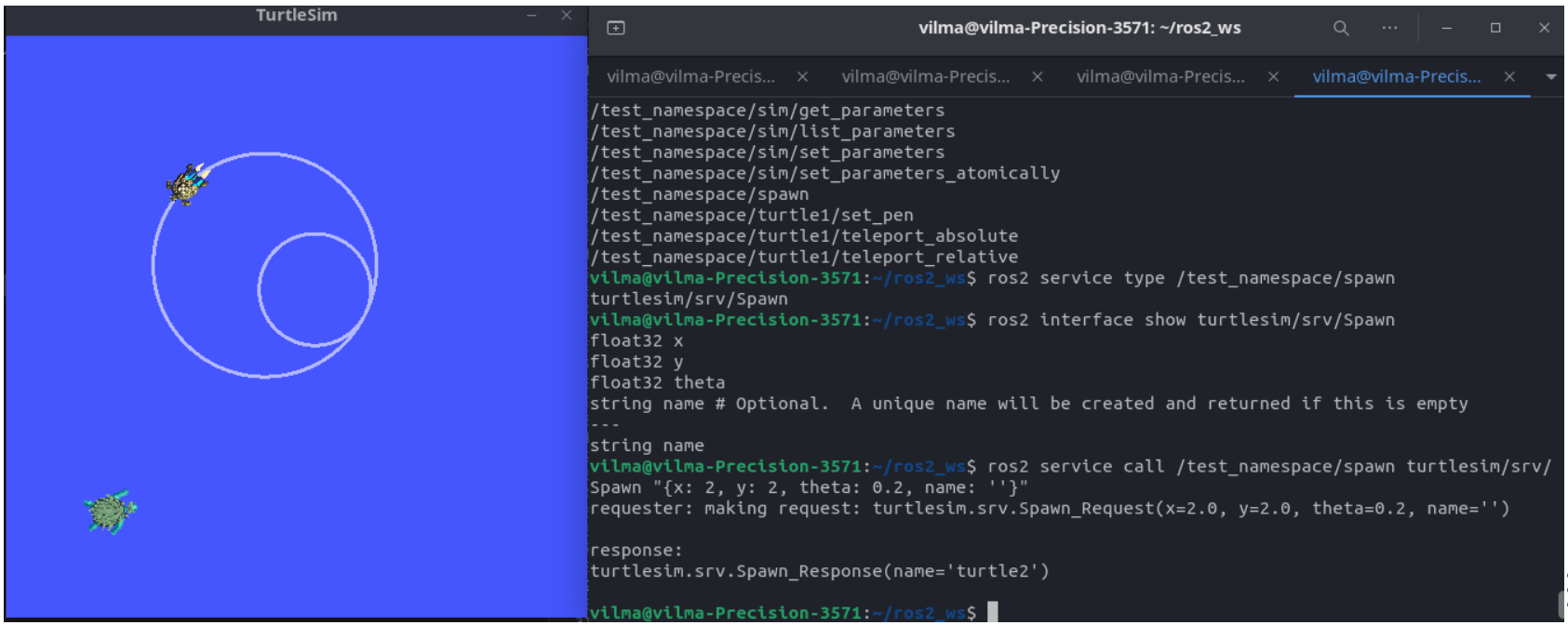
ros2 service call



ros2 service call

- Lets spawn another turtle

```
1  $ros2 service list
2  $ros2 service type /test_namespace/spawn
3  $ros2 interface show turtlesim/srv/Spawn
4  $ros2 service call /test_namespace/spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: ''}"
```

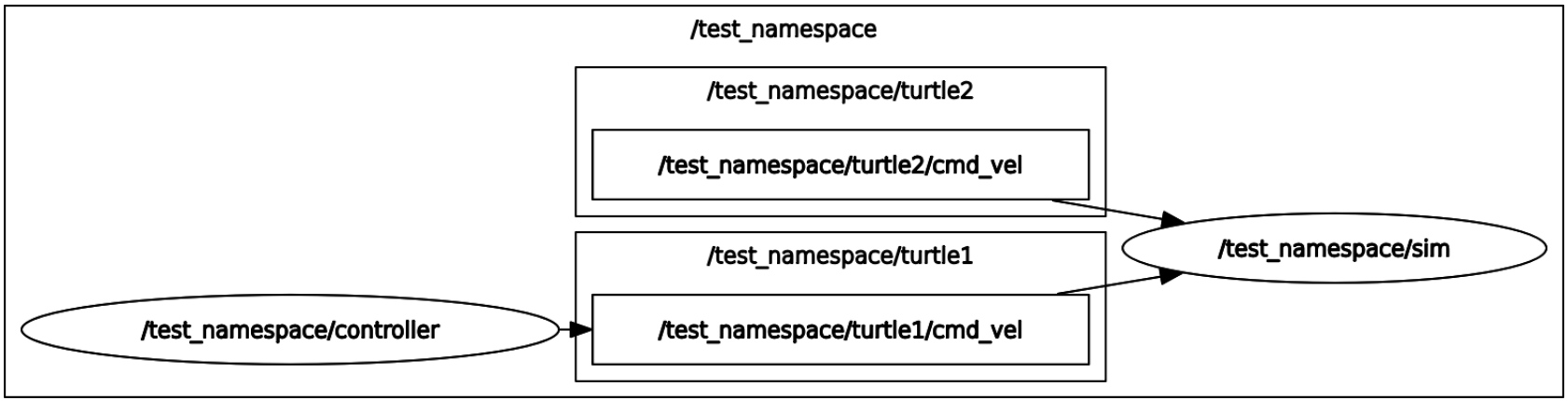


ros2 service call

- Pay attention to the namaspaces

```
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 service list
/test_namespace/change_radius
/test_namespace/clear
/test_namespace/controller/describe_parameters
/test_namespace/controller/get_parameter_types
/test_namespace/controller/get_parameters
/test_namespace/controller/list_parameters
/test_namespace/controller/set_parameters
/test_namespace/controller/set_parameters_atomically
/test_namespace/kill
/test_namespace/reset
/test_namespace/sim/describe_parameters
/test_namespace/sim/get_parameter_types
/test_namespace/sim/get_parameters
/test_namespace/sim/list_parameters
/test_namespace/sim/set_parameters
/test_namespace/sim/set_parameters_atomically
/test_namespace/spawn
/test_namespace/turtle1/set_pen
/test_namespace/turtle1/teleport_absolute
/test_namespace/turtle1/teleport_relative
/test_namespace/turtle2/set_pen
/test_namespace/turtle2/teleport_absolute
/test_namespace/turtle2/teleport_relative
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 topic list
/parameter_events
/rosout
/test_namespace/turtle1/cmd_vel
/test_namespace/turtle1/color_sensor
/test_namespace/turtle1/pose
/test_namespace/turtle2/cmd_vel
/test_namespace/turtle2/color_sensor
/test_namespace/turtle2/pose
vilma@vilma-Precision-3571:~/ros2_ws$
```

ros2 service call



Remapping

- Remapping allows you to reassign default node properties, like node name, topic names, service names, etc., to custom values.
- Remapping through the command line tool

```
ros2 run my_package node_executable --ros-args ...
ros2 run turtlesim turtle_teleop_key --ros-args --remap turtle1/cmd_vel:=turtle2/cmd_vel
```

- Launch files

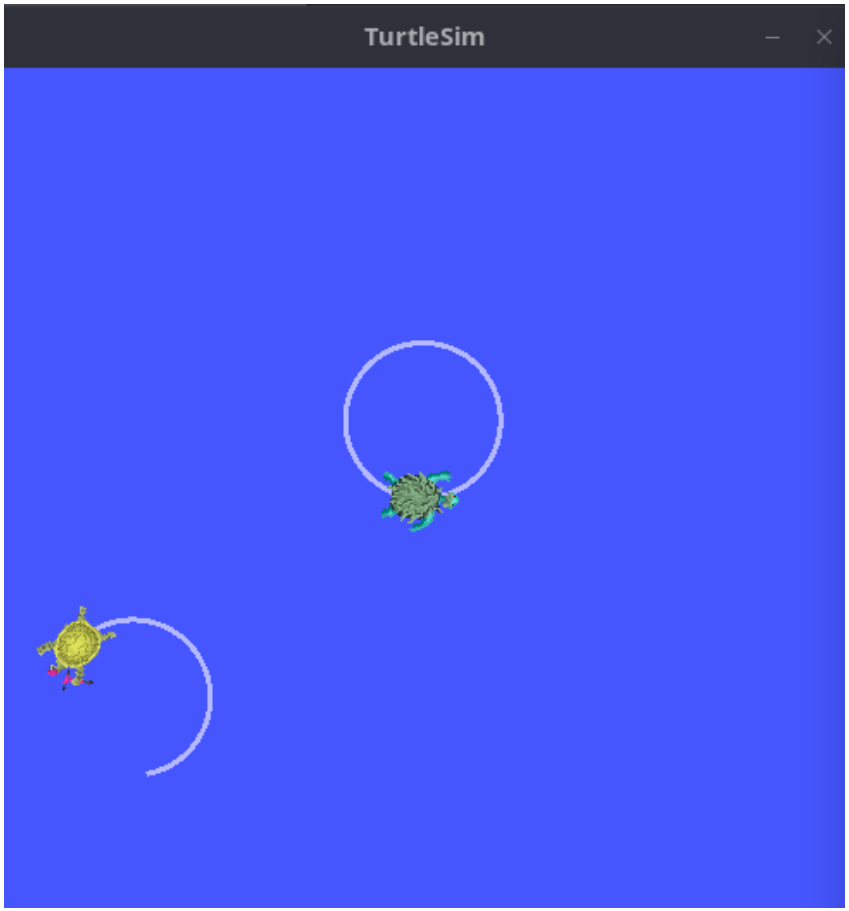
Remapping

- Modify the circle_path_launch.py

```
1  from launch import LaunchDescription
2  from launch_ros.actions import Node
3
4  def generate_launch_description():
5      return LaunchDescription([
6          Node(
7              package='turtlesim',
8              namespace='test_namespace',
9              executable='turtlesim_node',
10             name='sim',
11             remappings=[
12                 ('/test_namespace/turtle2/cmd_vel', '/test_namespace/turtle1/cmd_vel'),
13             ]
14         ),
15         Node(
16             package='p_controller',
17             namespace='test_namespace',
18             executable='circle_controller',
19             name='controller',
20         )
21     ])
```

Remapping

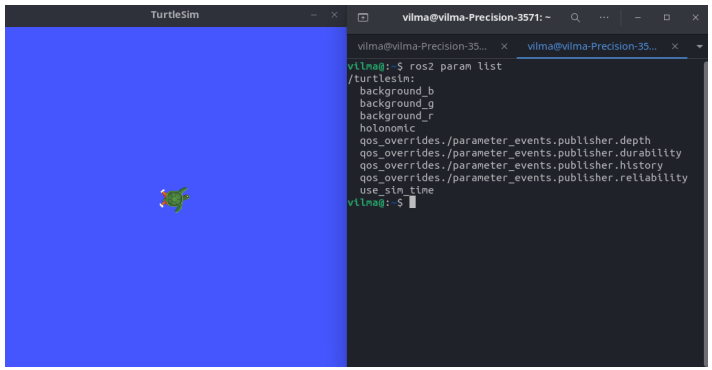
- Launch the nodes and spawn another turtle



Using parameters

- Adding parameters that can be configured during runtime
- Ex. ip address and port number for a Gps Receiver
- We can modify the value of the parameter :
  - via the console
  - from a launch file

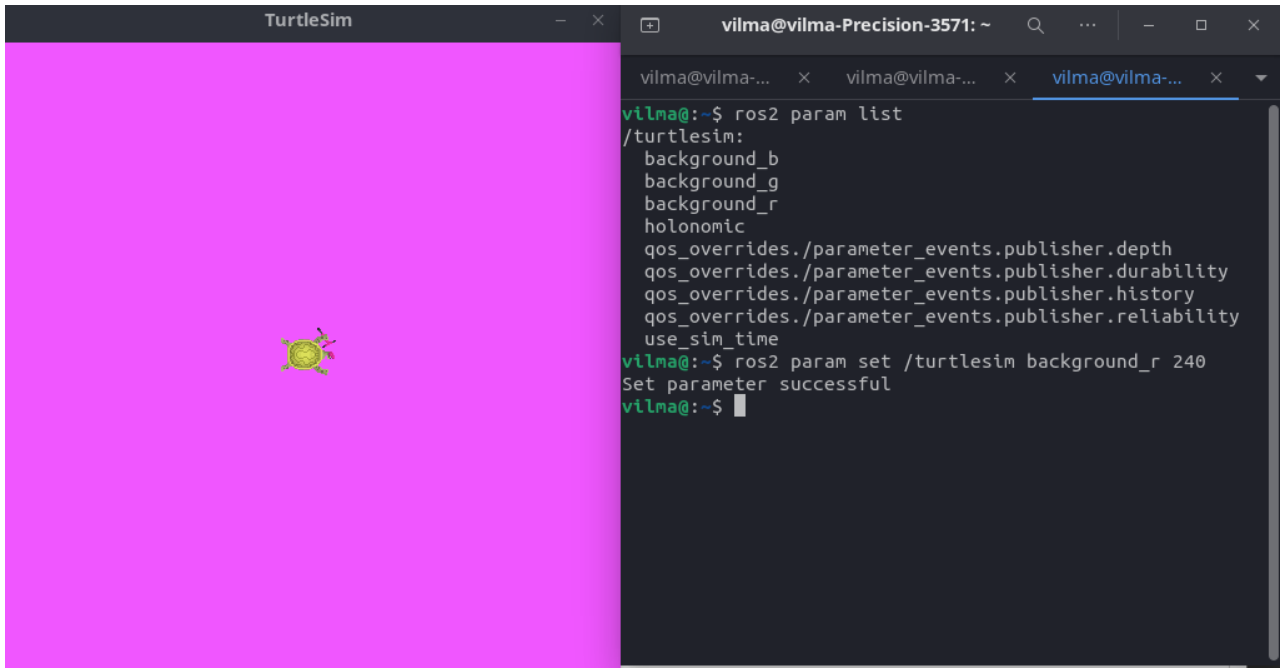
```
$ros2 param list
```



Using parameters

- Ex. change the background color of turtlesim

```
$ros2 param set /turtlesim background_r 240
```



Using parameters

- Add the angular z velocity of the turtle as a parameter in the circle controller example

Modify the python code to use parameters

```python
def __init__(self):
    super().__init__('circle_controller')
    self.publisher_ = self.create_publisher(Twist, 'turtle1/cmd_vel', 10)
    timer_period = 0.1  # seconds
    self.timer = self.create_timer(timer_period, self.timer_callback)
    self.srv = self.create_service(ChangeRadius, 'change_radius', self.change_radius_callback)
    self.radius = 1.0
    self.declare_parameter('angular_z_velocity', 1.0)

def timer_callback(self):
    # if the angular velocity param is bigger than 1.0, reset it to 1.0
    if self.get_parameter('angular_z_velocity').value > 1.0:
        new_param = rclpy.parameter.Parameter('angular_z_velocity', rclpy.Parameter.Type.DOUBLE, 1.0)
        self.set_parameters([new_param])

    vel = Twist()
    vel.linear.x = self.radius * 1.0  # linear velocity = radius * angular velocity
    vel.angular.z = self.get_parameter('angular_z_velocity').value

    self.publisher_.publish(vel)
```

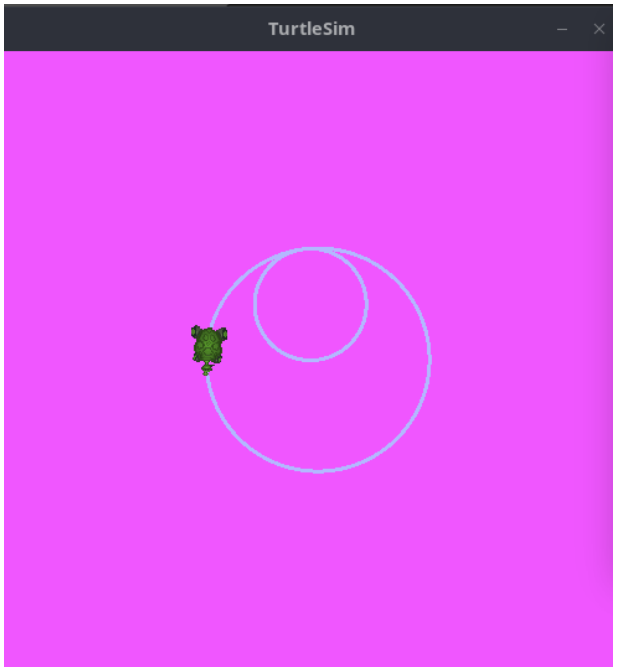
Build and Run

```
1  colcon build --packages-select p_controller
2  source install/setup.bash
3  $ros2 launch p_controller/p_controller/launch/circle_path_launch.py
4  $ros2 param list
```

```
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 param list
/test_namespace/controller:
  angular_z_velocity
  use_sim_time
/test_namespace/sim:
  background_b
  background_g
  background_r
  holonomic
  qos_overrides./parameter_events.publisher.depth
  qos_overrides./parameter_events.publisher.durability
  qos_overrides./parameter_events.publisher.history
  qos_overrides./parameter_events.publisher.reliability
  use_sim_time
vilma@vilma-Precision-3571:~/ros2_ws$ ^C
vilma@vilma-Precision-3571:~/ros2_ws$
```

Build and Run

```
ros2 param set /test_namespace/controller  angular_z_velocity 0.5
```

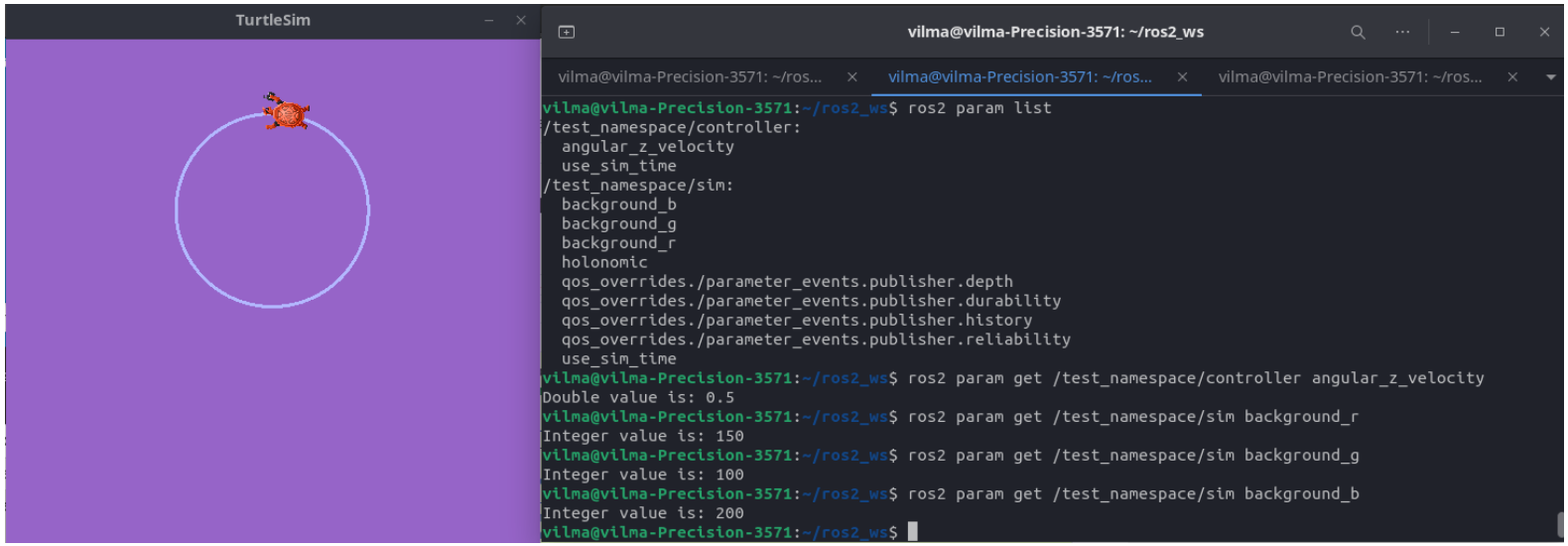


Param change via Launch files

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            namespace='test_namespace',
            executable='turtlesim_node',
            name='sim',
            remappings=[('/test_namespace/turtle2/cmd_vel', '/test_namespace/turtle1/cmd_vel')],
            parameters=[
                {"background_b": 200},
                {"background_g": 100},
                {"background_r": 150}
            ]
        ),
        Node(
            package='p_controller',
            namespace='test_namespace',
            executable='circle_controller',
            name='controller',
            parameters=[
                {"angular_z_velocity": 0.5}
            ]
        )
    ])
```

Run the Launch file



- Modify the service example code and use the service $/turtle1/set\_pen$ to draw two circles with 2 different colors

Custom Actions

- We can custom-define actions in our packages
- Actions are defined in .action files
- A request message is sent from an action client to an action server initiating a new goal.
- A result message is sent from an action server to an action client when a goal is done.
- Feedback messages are periodically sent from an action server to an action client with updates about a goal.

```
# Request
---
# Result
---
# Feedback
```

Custom Actions

- Create a new package

```
cd ~/ros2_ws/src
ros2 pkg create --license Apache-2.0 p_action_interfaces
cd p_action_interfaces
mkdir action
vi GoToPoint.action
```

- Create an action with goal to reach a point and stop

```
1  # Request
2  float32 x
3  float32 y
4  ---
5  # Result
6  bool success
7  ---
8  # Feedback
9  turtlesim/Pose pose
```

Build the custom action

- Modify the CMakeLists.txt

```
find_package(turtlesim REQUIRED)

find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "action/GoToPoint.action"
   DEPENDENCIES turtlesim
)
ament_package()
```

- Add the required dependencies to our package.xml

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
<exec_depend>turtlesim</exec_depend>
```

Build the custom action

```
colcon build --packages-select p_action_interfaces
source install/setup.bash
ros2 interface show p_action_interfaces/action/GoToPoint
```

```
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 interface show p_action_interfaces/action/GoToPoint
# Request
float32 x
float32 y
---
# Result
bool success
---
# Feedback
turtlesim/Pose pose
        float32 x
        float32 y
        float32 theta
        float32 linear_velocity
        float32 angular_velocity
vilma@vilma-Precision-3571:~/ros2_ws$
```

Writing an action server

```
1
2   class GoToGoal(Node):
3       def __init__(self):
4           super().__init__("go_to_goal")
5           self.publisher_ = self.create_publisher(Twist, "turtle1/cmd_vel", 10)
6
7           velocity_callback_group = MutuallyExclusiveCallbackGroup()
8           self.subscription = self.create_subscription(
9               Pose,
10              "turtle1/pose",
11              self.pose_listener_callback,
12              10,
13              callback_group=velocity_callback_group,
14          )
15
16          action_callback_group = MutuallyExclusiveCallbackGroup()
17          self.action_server = ActionServer(
18              self,
19              GoToPoint,
20              "go_to_point",
21              self.action_callback,
22              callback_group=action_callback_group,
23          )
24
25          self.current_pose = Pose()
26
27          # in the beginning, the turtle is happy where it is until it receives new instructions
28          self.goal_reached = True
29          self.velocity = Twist()  # 0 by default
30          self.goal_x = 0.0
31          self.goal_y = 0.0
32          self.EPSILON_ERROR = 0.1
```

Writing an action server

```
1     def action_callback(self, goal_handle):
2         self.get_logger().info('Executing goal...')
3         if not self.goal_reached:
4             print('got new goal before old one reached')
5         # set the goal
6         self.goal_x = goal_handle.request.x
7         self.goal_y = goal_handle.request.y
8         feedback_msg = GoToPoint.Feedback()
9         feedback_msg.pose = self.current_pose
10
11        # while the goal is not reached, send feedback to the client
12        diff_x = abs(self.current_pose.x - self.goal_x)
13        diff_y = abs(self.current_pose.y - self.goal_y)
14        while diff_x > self.EPSILON_ERROR or diff_y > self.EPSILON_ERROR:
15            print('sending feedback')
16            goal_handle.publish_feedback(feedback_msg)
17            feedback_msg.pose = self.current_pose
18            diff_x = abs(self.current_pose.x - self.goal_x)
19            diff_y = abs(self.current_pose.y - self.goal_y)
20            # put only this thread to sleep for 1 seconds
21            time.sleep(1)
22
23        goal_handle.succeed()
24        result = GoToPoint.Result()
25        result.success = True
26        self.goal_reached = True
27        return result
```

Writing an action server

```
1
2   def pose_listener_callback(self, msg):
3         self.current_pose = msg
4         angle = 0.0
5         delta = 0.0
6
7         diff_x = abs(self.current_pose.x - self.goal_x)
8         diff_y = abs(self.current_pose.y - self.goal_y)
9         # calculate the angle between the current pose and the goal pose
10        # and rotate the robot towards the goal
11        angle = math.atan2(self.goal_y - self.current_pose.y, self.goal_x - self.current_pose.x)
12
13        delta = angle - self.current_pose.theta
14        self.velocity.linear.x = 0.0
15
16        distance = math.sqrt(diff_x * diff_x + diff_y * diff_y)
17        if diff_x < self.EPSILON_ERROR and diff_y < self.EPSILON_ERROR:
18            self.velocity.angular.z = 0.0
19        elif delta < -0.02 or delta > 0.02:
20            self.velocity.angular.z =  delta * 0.5
21        else:
22            self.velocity.angular.z = 0.0
23            self.velocity.linear.x = distance * 0.5
24
25        # once we have updated what we want to do, publish the velocity
26        if self.velocity.linear.x != 0.0 or self.velocity.angular.z != 0.0:
27            self.publisher_.publish(self.velocity)
```

Writing an action server

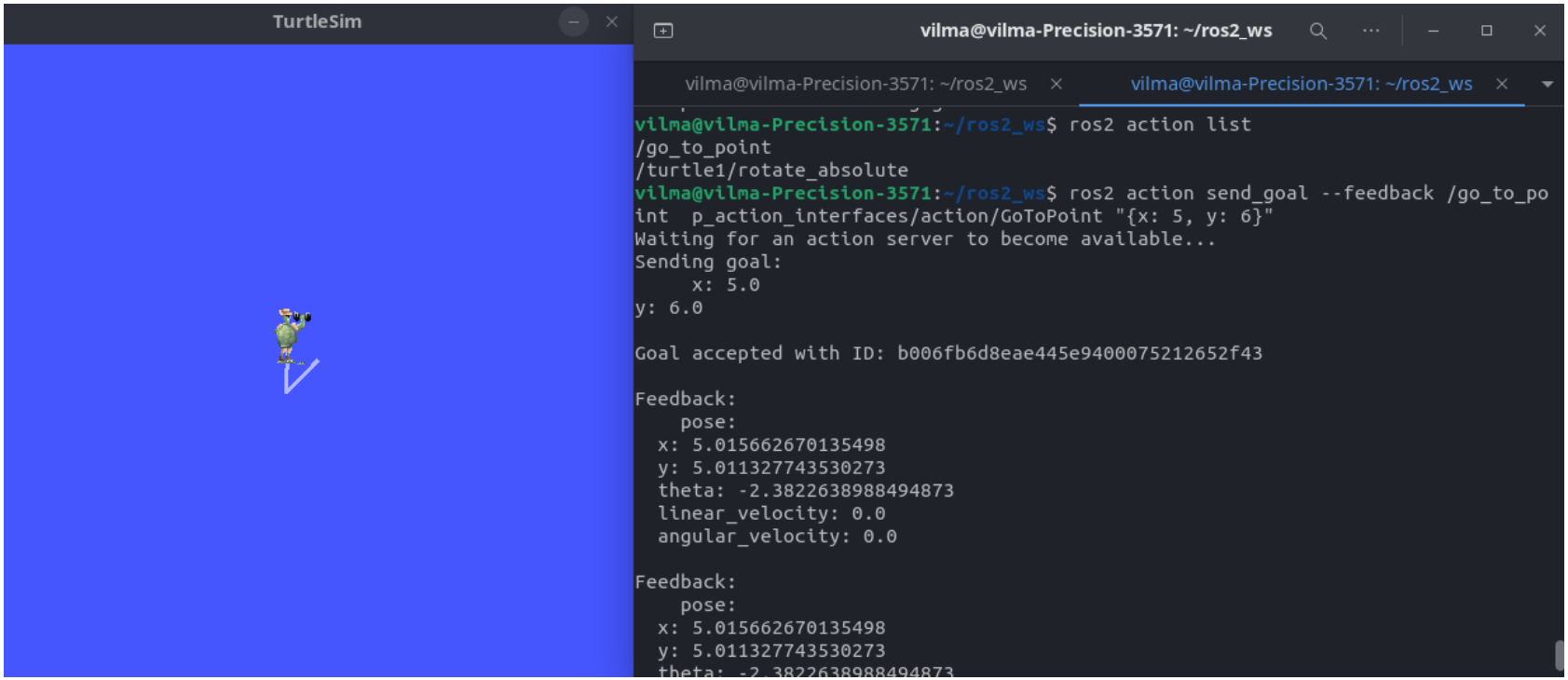
```
1  def main(args=None):
2      rclpy.init(args=args)
3
4      go_to_goal = GoToGoal()
5      executor = MultiThreadedExecutor()  # needed to run functions concurrently
6      executor.add_node(go_to_goal)
7
8      executor.spin()
9
10     go_to_goal.destroy_node()
11     rclpy.shutdown()
12
13
14 if __name__ == '__main__':
15     main()
```

Run the program

- Run turtlesim node and the p_action_example package on two different terminals
- In a third terminal send a goal to the action via command line

```
colcon build --packages-select p_action_example  && ros2 run p_action_example go_to_goal
ros2 run turtlesim turtlesim_node
ros2 action list
ros2 action send_goal --feedback /go_to_point  p_action_interfaces/action/GoToPoint "{x: 5, y: 6}"
```

Run the program



Action Client

- Create an action client that moves the turtle in a square

```python
import rclpy
from rclpy.action import ActionClient
from rclpy.node import Node
import time
from p_action_interfaces.action import GoToPoint

class DrawSquareActionClient(Node):
    def __init__(self):
        super().__init__('draw_square_action_client')
        self._action_client = ActionClient(self, GoToPoint, 'go_to_point')
        # create list with 4 edges of the square in tuple format starting from coordinate (5, 5)
        # and with distance of a units between each edge going counter-clockwise
        a = 3.0
        self.edges = [(5.0, 5.0),(a + 5.0, 5.0), (a + 5.0, a + 5.0), (5.0, a + 5.0),(5.0, 5.0)]
        self.current_step = 0 # the current edge of the square the turtle is on

    def send_goal(self):
        x, y = self.edges[self.current_step]
        goal_msg = GoToPoint.Goal()
        goal_msg.x = x
        goal_msg.y = y

        self._action_client.wait_for_server()
```

Action Client

```
1     def send_goal(self):
2         x, y = self.edges[self.current_step]
3         goal_msg = GoToPoint.Goal()
4         goal_msg.x = x
5         goal_msg.y = y
6
7         self._action_client.wait_for_server()
8         self._send_goal_future = self._action_client.send_goal_async(goal_msg,
9         feedback_callback=self.feedback_callback)
10        self._send_goal_future.add_done_callback(self.goal_response_callback)
11
12    def goal_response_callback(self, future):
13        goal_handle = future.result()
14        if not goal_handle.accepted:
15            self.get_logger().info('Goal rejected :(')
16            return
17
18        self.get_logger().info('Goal accepted :)')
19        self._get_result_future = goal_handle.get_result_async()
20        self._get_result_future.add_done_callback(self.get_result_callback)
```

Action Client

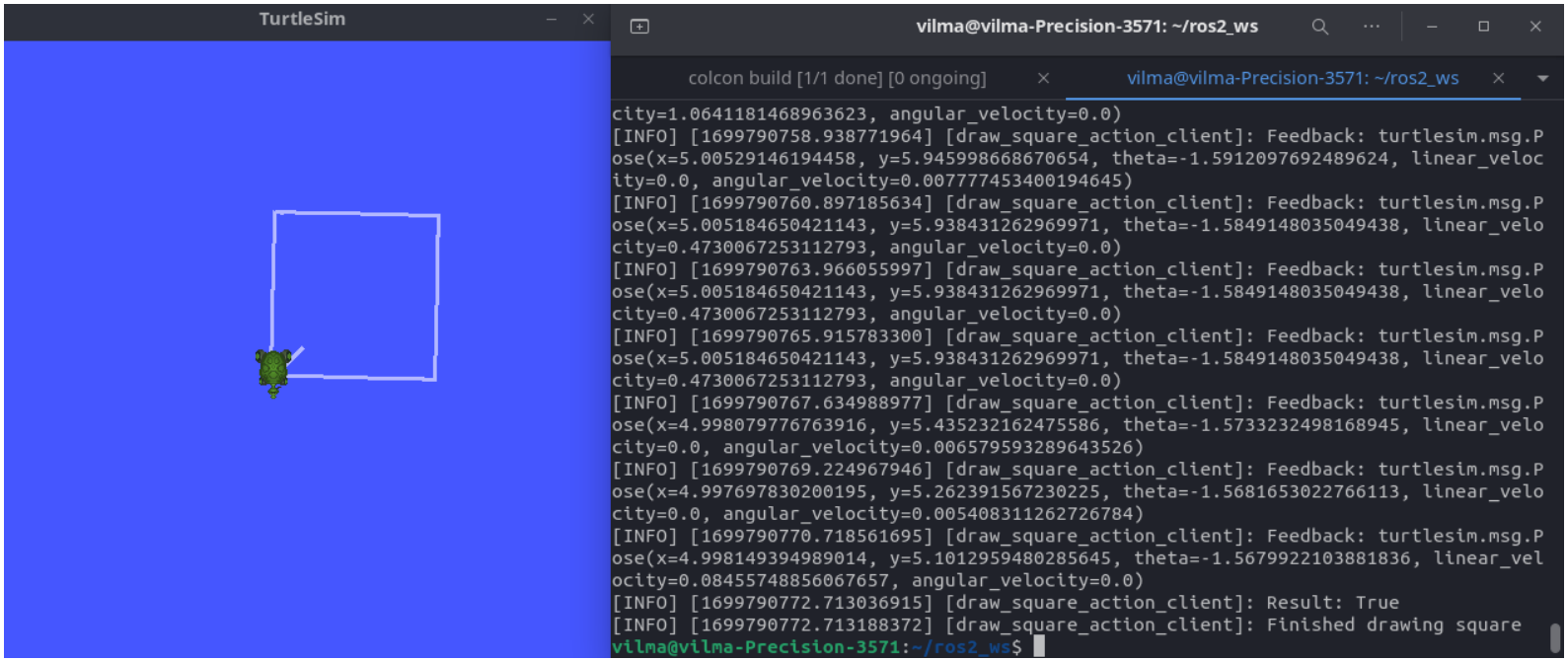
```
    def get_result_callback(self, future):
        result = future.result().result
        self.get_logger().info('Result: {0}'.format(result.success))
        # if result is True go to next step
        if result.success:
            self.current_step += 1
            if self.current_step < len(self.edges):
                self.send_goal()
            else:
                self.get_logger().info('Finished drawing square')
                rclpy.shutdown()

    def feedback_callback(self, feedback_msg):
        self.get_logger().info('Feedback: {0}'.format(feedback_msg.feedback.pose))

def main(args=None):
    rclpy.init(args=args)
    action_client = DrawSquareActionClient()
    action_client.send_goal()
    rclpy.spin(action_client)

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

Run the program



Thank you!!!
Questions?