Robot Operating System (ROS2)

Vilma Muço

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

December, 2024

Table of Contents

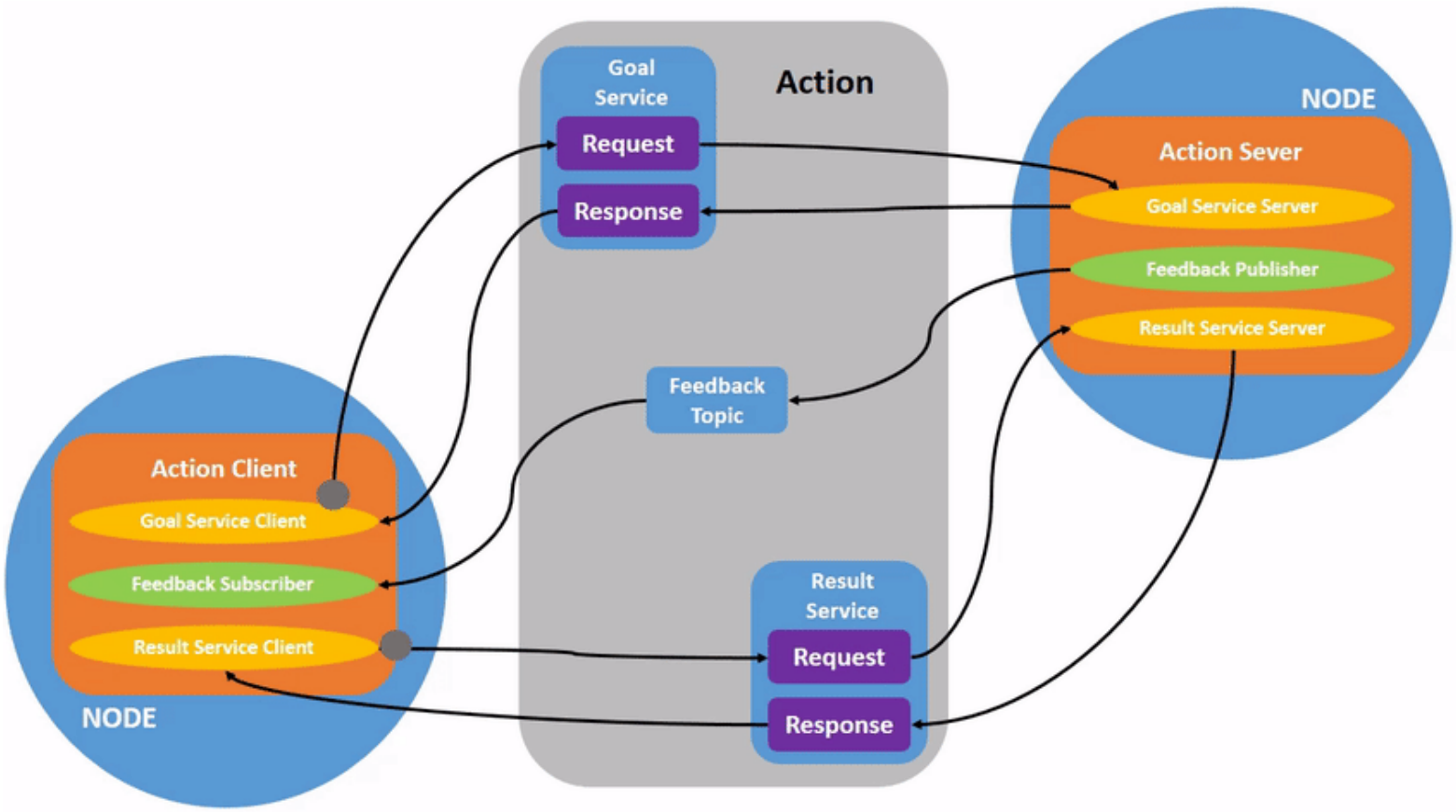


Image taken from *docs.ros.org*

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

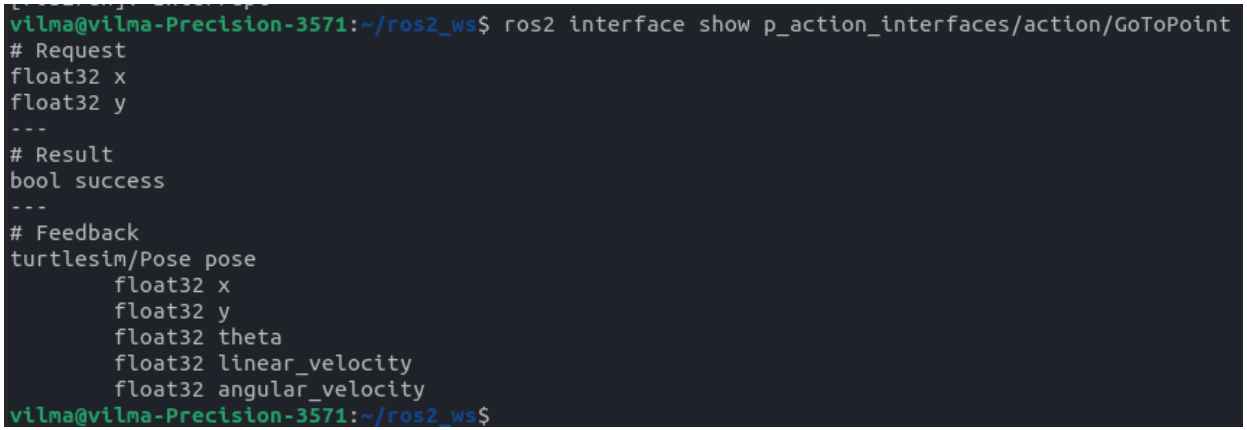
```
find_package(turtlesim REQUIRED)

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
```



Writing an action server

```
1
2   class GoToGoal(Node):
3       def __init__(self):
4           super().__init__("go_to_goal")
5
6           self.publisher_ = self.create_publisher(Twist, "turtle1/cmd_vel", 10)
7
8           velocity_callback_group = MutuallyExclusiveCallbackGroup()
9           self.subscription = self.create_subscription(
10              Pose,
11              "turtle1/pose",
12              self.pose_listener_callback,
13              10,
14              callback_group=velocity_callback_group,
15          )
16
17          action_callback_group = MutuallyExclusiveCallbackGroup()
18          self.action_server = ActionServer(
19              self,
20              GoToPoint,
21              "go_to_point",
22              self.action_callback,
23              callback_group=action_callback_group,
24          )
25
26          self.current_pose = Pose()
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

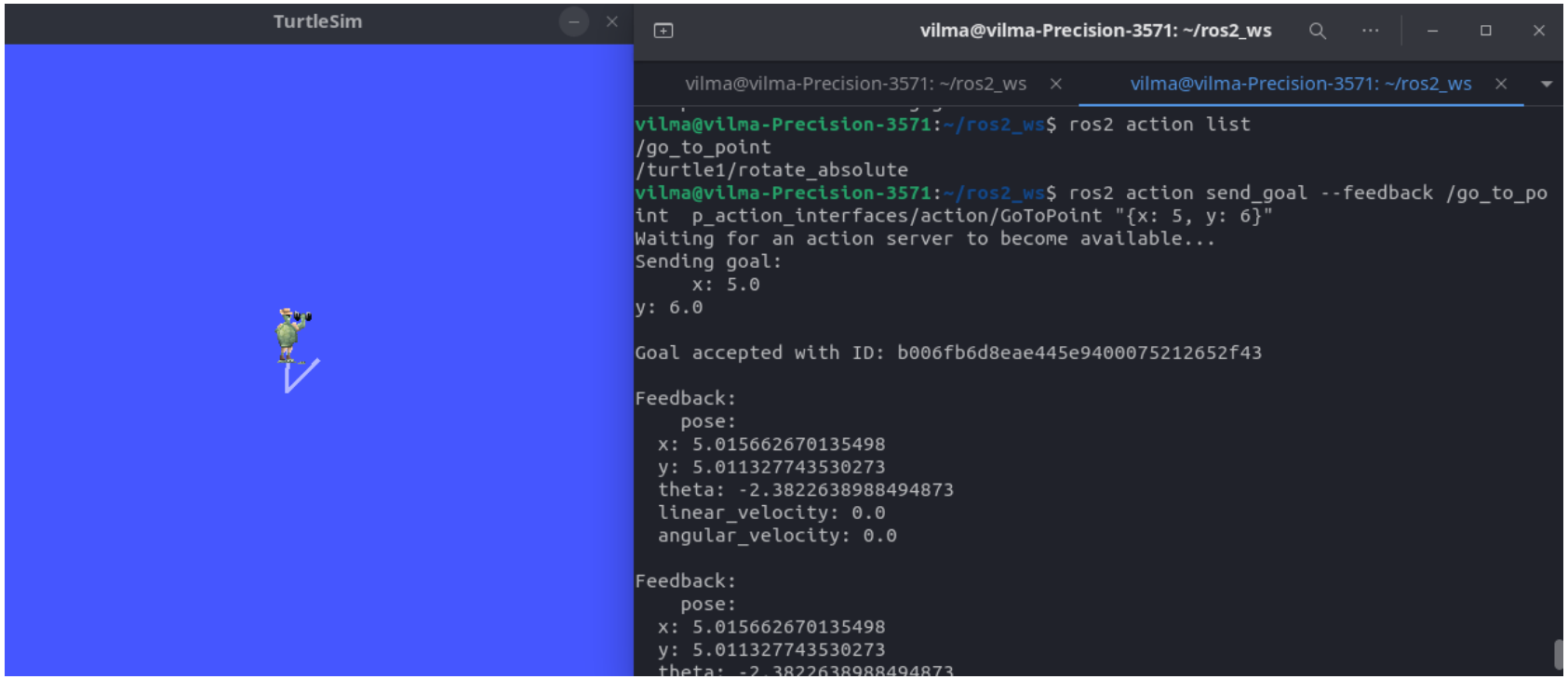
```
1  def main(args=None):
2      rclpy.init(args=args)
3
4      go_to_goal = GoToGoal()
5      executor = MultiThreadedExecutor()  # needed to run functions concurrently
6      executor.add_node(go_to_goal)
7
8      executor.spin()
9
10     go_to_goal.destroy_node()
11     rclpy.shutdown()
12
13
14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

```
1  import rclpy
2  from rclpy.action import ActionClient
3  from rclpy.node import Node
4  import time
5  from p_action_interfaces.action import GoToPoint
6
7  class DrawSquareActionClient(Node):
8      def __init__(self):
9          super().__init__('draw_square_action_client')
10         self._action_client = ActionClient(self, GoToPoint, 'go_to_point')
11         # create list with 4 edges of the square in tuple format starting from coordinate (5, 5)
12         # and with distance of a units between each edge going counter-clockwise
13         a = 3.0
14         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)]
15         self.current_step = 0 # the current edge of the square the turtle is on
16
17     def send_goal(self):
18         x, y = self.edges[self.current_step]
19         goal_msg = GoToPoint.Goal()
20         goal_msg.x = x
21         goal_msg.y = y
22
23         self._action_client.wait_for_server()
```

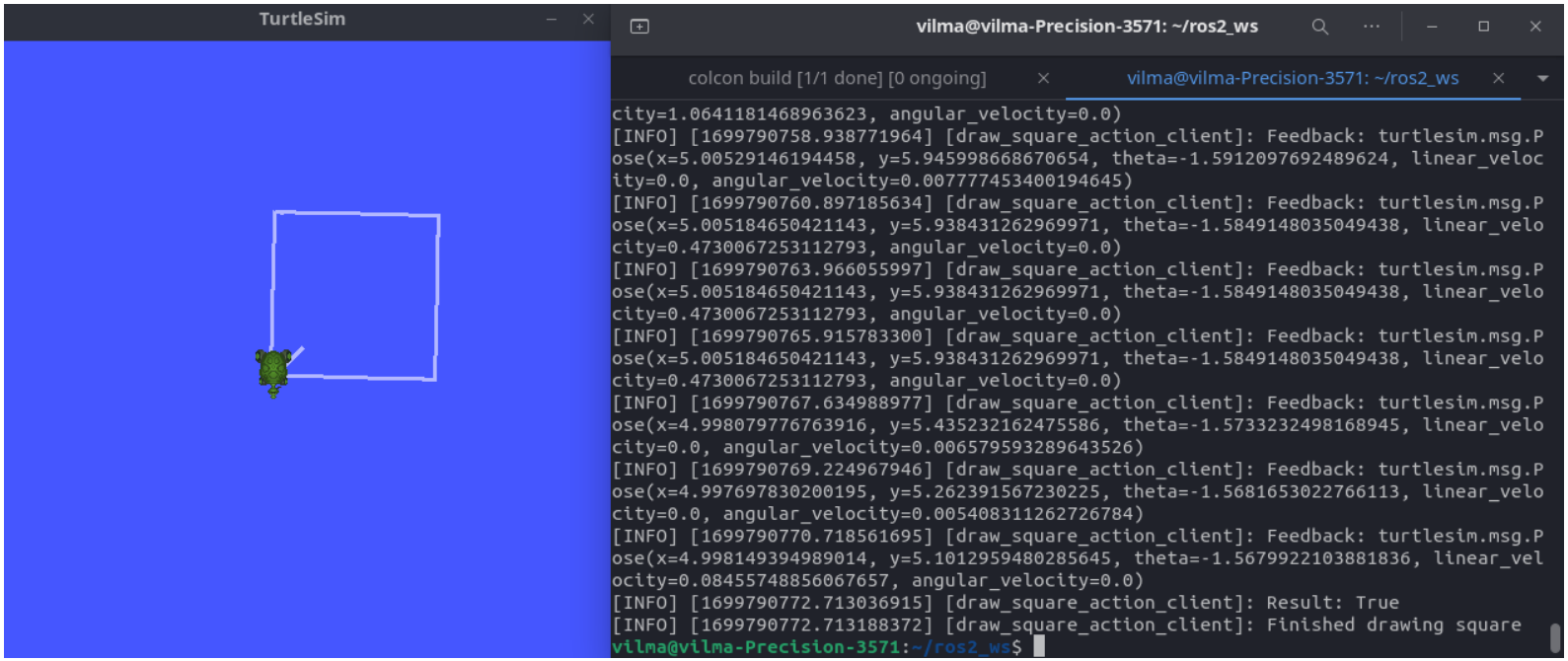
Action Client

```
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



- Modify the service example code and use the service */turtle1/set_pen* to draw two circles with 2 different colors
- Follow the instructions on https://github.com/vilmamuco/bluerov2_exercise

Modify the action example in order to draw four little squares to form a bigger square with twice the length given in the request. Send an email (muco.vilma@gmail.com) with the title [ROS2-COURSE-HOMEWORK-1] containing:

- a short video/recording of your screen with:
  - your explanation of the main lines of the code
  - buiding the package
  - the turtle creating the big square
- a zip folder with your code

Thank you!!!
Questions?