CENG483: Behavioural Robotics
Izmir Institute of Technology
Fall 2023
Instructor: Dr. Berat Alper Erol

Homework Set: 04

Ozan Gülbaş

This homework answers the problem set sequentially.

1. *Develop/write a controller for the turtle in a turtlesim simulator by teleoperating it, where the turtle demonstrates a command in a completely opposite way as asked by the user/operator, i.e., while the user requesting a movement towards the north, the turtle goes to the south, and so on.*

   To manipulate turtlesim source code, we need to create a workspace other than the ros system installed on root file system. To do this I have created a new directory first:

   ```
   mkdir -p ~/ros2_hw4_ws/src
   cd ~/ros2_hw4_ws/src
   ```

   After that, I have cloned the ros tutorial git repo to have a copy of turtlesim.

   ```
   git clone https://github.com/ros/ros_tutorials.git -b iron
   ```

   Then, I've run the following command in my workspace's root (~/**ros2_ws_hw4**/) to resolve dependencies before building the workspace.

   ```
   cd ..
   rosdep install -i --from-path src --rosdistro iron -y
   ```

   Then build the workspace with:

   ```
   colcon build
   ```

   After creating the workspace, I opened a new terminal and sourced the newly created workspace as an 'overlay' on top of ros system installed on my computer's root file system which became an 'underlay'.

   ```
   cd ~/ros2_hw4_ws
   source install/local_setup.bash
   ```

   Then I located the source file used for teleoperating the trutlesim turtle in
   ~/ros2_hw4_ws/src/ros_tutorials/turtlesim/tutorials/teleop_turtle_key.cpp
   and opened it with neovim text editor to edit.

```
cd src/ros_tutorials/turtlesim/tutorials/
nvim teleop_turtle_key.cpp
```

After examining the file, I found the necessary code block for operating the turtlesim turtle. Starting from **line 230**, the switch case compares the keyboard input with the keycodes coded at the beginning of the file and determines which operation should be published. Since our purpose is the revert the turtles movements with our keyboard inputs, I changed the cases with their respective inverses.

```
switch(c)
  {
  case ~~KEYCODE_LEFT~~ KEYCODE_RIGHT:
    RCLCPP_DEBUG(nh_->get_logger(), "LEFT");
    angular = 1.0;
    break;
  case ~~KEYCODE_RIGHT~~ KEYCODE_LEFT:
    RCLCPP_DEBUG(nh_->get_logger(), "RIGHT");
    angular = -1.0;
    break;
  case ~~KEYCODE_UP~~ KEYCODE_DOWN:
    RCLCPP_DEBUG(nh_->get_logger(), "UP");
    linear = 1.0;
    break;
  case ~~KEYCODE_DOWN~~ KEYCODE_UP:
    RCLCPP_DEBUG(nh_->get_logger(), "DOWN");
    linear = -1.0;
    break;
  case KEYCODE_G:
...
}
```

2. *Write code to control turtles in two turtlesim simulators using keyboard where, turtles run in exactly the opposite way to each other, i.e. the turtle in simulator 1 is moving forward while the turtle in simulator 2 is moving backwards. Let us say, the turtle in simulator 1 is turning 90 degrees in clockwise direction, turtle in simulator 2 must turn 90 degrees in counterclockwise direction*

   To achieve this goal, I first thought of a solution including a gateway node that subscribes to **turtle1/cmd_vel** topic and manipulates the received messages from **/teleop_turtle** publisher node and publishes them to related turtlesim turtle **/cmd_vel** topics **(Figure 1)** because I was daunted by the C++. I wrote the gateway node with Python however, I realized that changing the Twist message file without knowing for which command it was sent and for which value it should be invested is rather difficult.

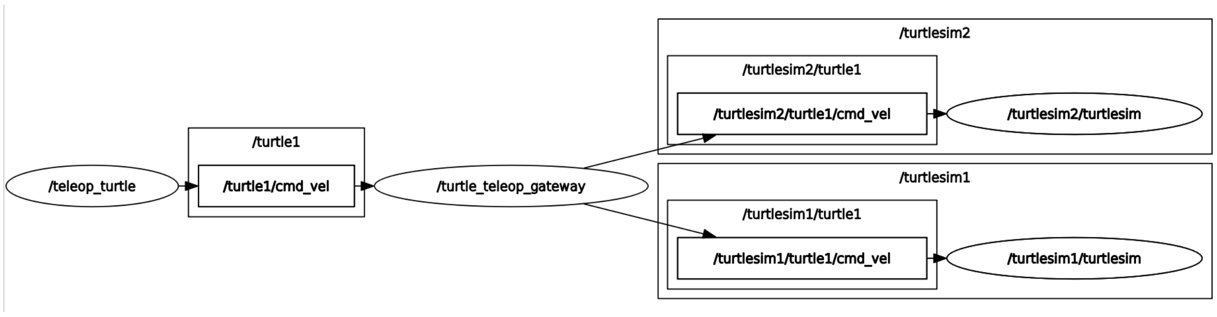

Figure 1: Planned roadmap with a gateway node

   So, I started to read through the *teleop_turtle_key.cpp* to find which part I should change to achieve this goal. The plan is like this: create two new publishers in **/teleop_turtle** node which will publish to the related turtlesim turtle **/cmd_vel** topics instead of creating a whole new node for the gateway.

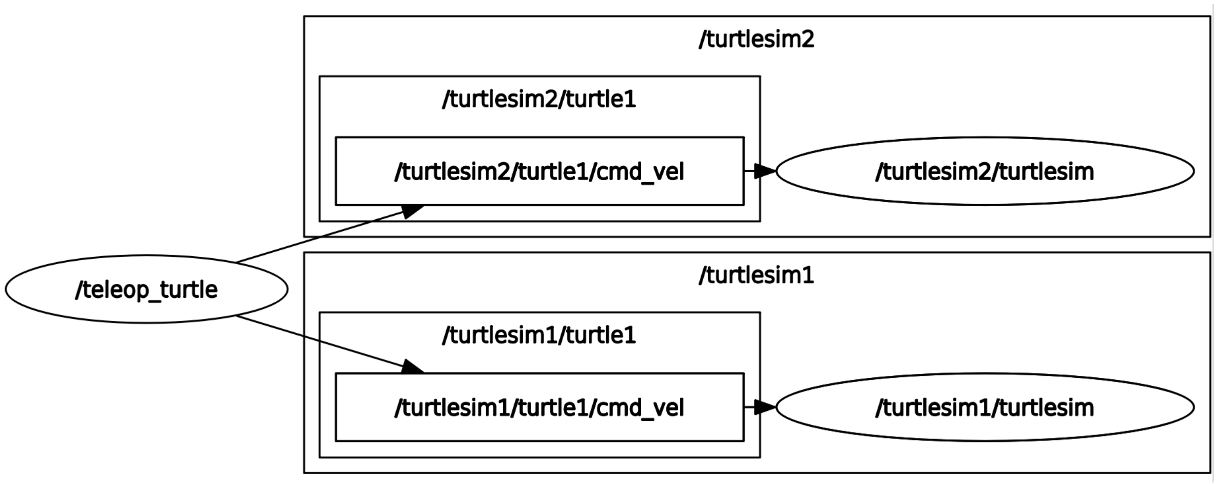

Figure 2: Practical approach that will be implemented without a gateway node

   The first manipulation is in line 196 inside the `TeleopTurtle()` function. Here the pre-written code creates a publisher that publishes to the **turtle1/cmd_vel** topic with a queue size of 10

```
twist_pub_ = nh_->create_publisher<geometry_msgs::msg::Twist>
    (
        "turtle1/cmd_vel",
        1
    );
```

I copied/pasted the same line twice to create two more publishers that publish **turtlesim1/turtle1/cmd_vel** and **turtlesim2/turtle1/cmd_vel** respectively.

```
twist_pub_slave_one_ =
    nh_->create_publisher<geometry_msgs::msg::Twist>
        (
            "turtlesim1/turtle1/cmd_vel",
            1
        );

twist_pub_slave_two_ =
    nh_->create_publisher<geometry_msgs::msg::Twist>
        (
            "turtlesim2/turtle1/cmd_vel",
            1
        );
```

Next, I located the necessary code block that publishes the given message. Starting from line 309, I found the necessary code that creates a geometry message of Twist type, manipulates that message according to the keyboard input, and publishes the manipulated message.

```
geometry_msgs::msg::Twist twist;

twist.angular.z =
    nh_->get_parameter("scale_angular").as_double() * angular;
twist.linear.x =
    nh_->get_parameter("scale_linear").as_double() * linear;

twist_pub_->publish(twist);
```

To have the turtles on different nodes behave exactly opposite of each other, I changed the published message by creating a new Twist message and changing its necessary values with the negative values of the keyboard input values. Then I published the messages to related topics.

```
geometry_msgs::msg::Twist twist_inv;

twist_inv.angular.z =
    nh_->get_parameter("scale_angular").as_double() * -angular;
twist_inv.linear.x =
    nh_->get_parameter("scale_linear").as_double() * -linear;

twist_pub_slave_one_->publish(twist);
twist_pub_slave_two_->publish(twist_inv);
```

Then, I had to declare the two new publishers in the constructor **TeleopTurtle::TeleopTurtle()** to make the code work.

```
rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr twist_pub_slave_one_;
rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr twist_pub_slave_two_;
```

With these changes, the turtles on different nodes behave exactly opposite of each other however, when I tried to rotate absolute orientations, they were behaving the same. To make absolute orientation rotations behave oppositely, I have followed a similar way as before. First, I created two new action clients:

```
rotate_absolute_client_slave_one_ =
    rclcpp_action::create_client<turtlesim::action::RotateAbsolute>
        (
            nh_,
            "turtlesim1/turtle1/rotate_absolute"
        );

rotate_absolute_client_slave_two_ =
    rclcpp_action::create_client<turtlesim::action::RotateAbsolute>
        (
            nh_,
            "turtlesim2/turtle1/rotate_absolute"
        );
```

Then I created `sendGoalInv()` function that has the same code with the pre-written `sendGoal()` function but initializes the goal with the **negative theta** value.

```
void sendGoalInv(float theta){
    auto goal = turtlesim::action::RotateAbsolute::Goal();
    goal.theta = -theta;

    auto send_goal_options =
        rclcpp_action::Client
            <turtlesim::action::RotateAbsolute>
                ::SendGoalOptions();

    send_goal_options.goal_response_callback =
        [this](
            rclcpp_action::ClientGoalHandle
                <turtlesim::action::RotateAbsolute>
                    ::SharedPtr goal_handle
            )
        {
            RCLCPP_DEBUG
                (
                    nh_->get_logger(),
                    "Goal response received"
                );
            this->goal_handle_ = goal_handle;
        };
rotate_absolute_client_slave_one_->async_send_goal(goal, send_goal_options);
}
```

Then I added, the necessary line to `sendGoal()` function to send goal to **roatate_absolute_client_slave_two_**

```
    rotate_absolute_client_slave_two_->
        async_send_goal
            (
                goal,
                send_goal_options
            );
```

Next, I added `sendGoalInv()` function for every rotate absolute orientation case:

```
switch(c)
    {
    case KEYCODE_RIGHT:
        RCLCPP_DEBUG(nh_->get_logger(), "LEFT");
        angular = 1.0;
        break;
    case KEYCODE_LEFT:
        RCLCPP_DEBUG(nh_->get_logger(), "RIGHT");
        angular = -1.0;
        break;
    case KEYCODE_DOWN:
        ...
        ...
    case KEYCODE_G:
        RCLCPP_DEBUG(nh_->get_logger(), "G");
        sendGoal(0.0f);
        sendGoalInv(0.0f);
        break;
      case KEYCODE_T:
        RCLCPP_DEBUG(nh_->get_logger(), "T");
        sendGoal(0.7854f);
        sendGoalInv(0.7854f);
        break;
      case KEYCODE_R:
        ...
        ...
    case KEYCODE_B:
        RCLCPP_DEBUG(nh_->get_logger(), "B");
        sendGoal(-0.7854f);
        sendGoalInv(-0.7854f);
        break;
      case KEYCODE_F:
        RCLCPP_DEBUG(nh_->get_logger(), "F");
        cancelGoal();
        break;
      case KEYCODE_Q:
        RCLCPP_DEBUG(nh_->get_logger(), "quit");
        running = false;
        break;
      default:
        ...
        break;
      }
```

Finally, I declared the necessary action clients to make the code work.

```
rclcpp_action
    ::Client<turtlesim::action::RotateAbsolute>
    ::SharedPtr rotate_absolute_client_slave_one_;

rclcpp_action
    ::Client<turtlesim::action::RotateAbsolute>
    ::SharedPtr rotate_absolute_client_slave_two_;
```

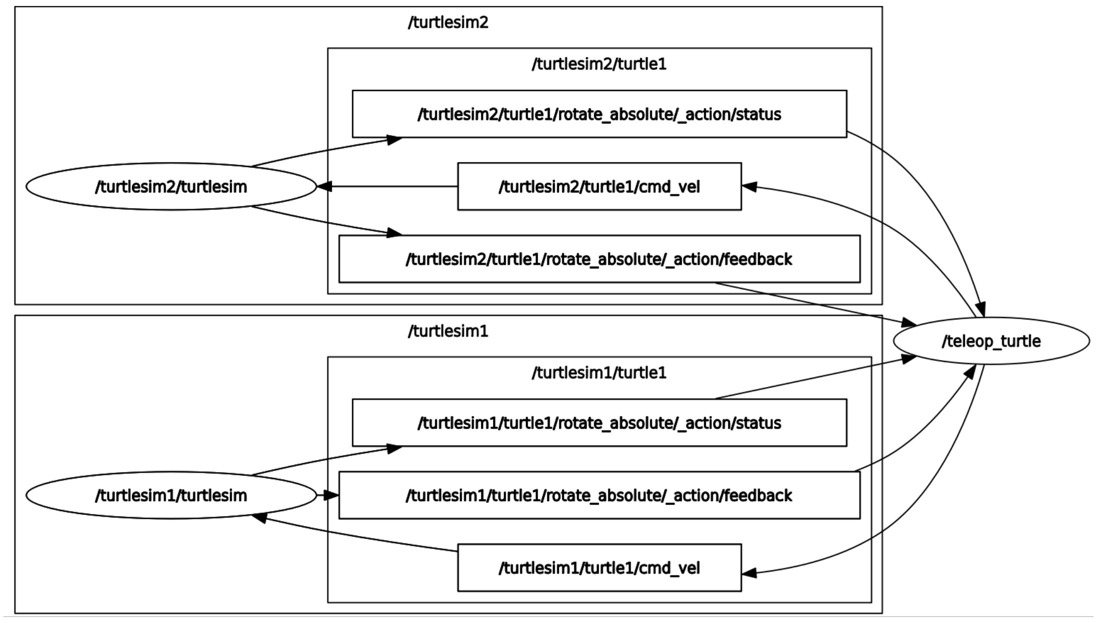
/turtlesim2
/turtlesim2/turtle1
/turtlesim2/turtle1/rotate_absolute/_action/status
/turtlesim2/turtlesim
/turtlesim2/turtle1/cmd_vel
/turtlesim2/turtle1/rotate_absolute/_action/feedback
/teleop_turtle
/turtlesim1
/turtlesim1/turtle1
/turtlesim1/turtle1/rotate_absolute/_action/status
/turtlesim1/turtlesim
/turtlesim1/turtle1/rotate_absolute/_action/feedback
/turtlesim1/turtle1/cmd_vel

Figure 3: Final rqt graph

1 References

Ros 2 documentation. ROS 2 Documentation - ROS 2 Documentation: Iron documentation. (n.d.). https://docs.ros.org/en/iron/index.html
Cem Tolga Münyas, Boğaçhan Ali Düşgül, Emre Sengir, Uğurcan Güngör,