

# ROS Exercises

Moldir Zabirova, Valeriya Kostyukova, Daryn Kenzhebek

## I. INTRODUCTION AND REAL-WORLD APPLICATIONS

**R**OS (Robot Operating System) is a powerful framework used in robotics to simplify the development of complex robot systems. It's not an actual operating system like Windows or Linux, but it's a collection of tools, libraries, that are used for various aspects of building and controlling manipulators and robotic systems. ROS offers standardized approach for regular tasks of any robot, such as sensing, data processing, communication between robot system parts, and motion/control. ROS makes all these complicated programming tasks integrated in one system, serving as a robot development template. Furthermore, ROS is open-source framework, which makes it collaborative ecosystem that accelerates robotics field development. ROS communities are designed for knowledge sharing and anyone interested in robotics can get help or contribute to innovative solutions.

ROS is widespread robotics framework around the world. One of the robotics companies using ROS as a fundamental tool is Clearpath. It is a Canadian company founded in 2009 by graduates of University of Waterloo. The number of robots that it produces in the fields of unmanned ground/surface(water) vehicles, and industrial vehicles is amazing. The company's robots are based on ROS and can be programmed with ROS. That is why these robots are used in the creation of third-party applications for mining, survey, inspection, agriculture, and material handling.

## II. EXERCISE 0

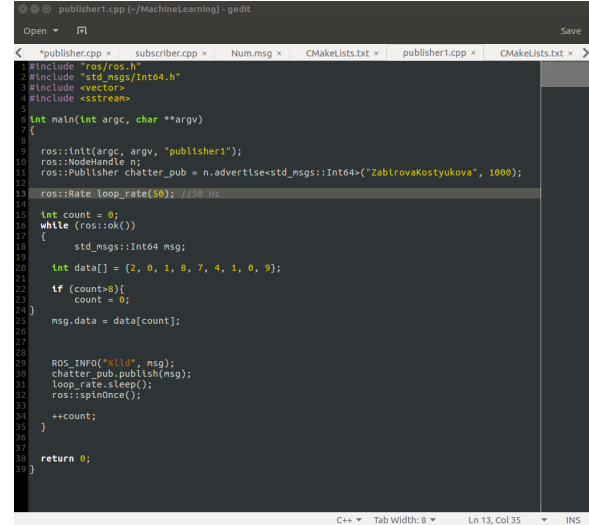
**I**N this task we wrote a simple publisher (Fig. 1) and subscriber (Fig. 2). The primary purpose of this publisher is to send a sequence of integer values (a student ID number) repeatedly to a topic named "ZabirovaKostyukova". To control the publishing rate we use a `ros::Rate` object. For this task, the rate should be 1 Hz and 50 Hz.

The subscriber is set to listen to the "ZabirovaKostyukova" topic, where it anticipates receiving messages in a form of ID number one by one. A callback function named `chatterCallback` is defined to handle incoming messages. When a message arrives on the subscribed topic, this function is automatically triggered to process the message's content and print the result in terminal.

## III. EXERCISE 1

**T**HE second task's aim was to make a communication run between nodes in the ROS network. The subscriber and publisher nodes were already written in Exercise 0 (Fig. 3). If we run two nodes, we can see that they are talking to each other. The subscriber code's output prints out the data from the publisher node.

## IV. EXERCISE 2



```

1 #include "ros/ros.h"
2 #include "std_msgs/Int64.h"
3 #include <vector>
4 #include <sstream>
5
6 int main(int argc, char **argv)
7 {
8     ros::init(argc, argv, "publisher");
9     ros::NodeHandle n;
10    ros::Publisher chatter_pub = n.advertise<std_msgs::Int64>("ZabirovaKostyukova", 1000);
11
12    ros::Rate loop_rate(50); // 50 Hz
13
14    int count = 0;
15    while (ros::ok())
16    {
17        std_msgs::Int64 msg;
18
19        int data[] = {2, 0, 1, 0, 7, 4, 1, 0, 9};
20
21        if (count % 8)
22            count = 0;
23
24        msg.data = data[count];
25
26        ROS_INFO("Willd", msg);
27        chatter_pub.publish(msg);
28        loop_rate.sleep();
29        ros::spinOnce();
30        ++count;
31    }
32    return 0;
33 }

```

Fig. 1. publisher.cpp code



```

1 #include "ros/ros.h"
2 #include "std_msgs/Int64.h"
3
4 void chatterCallback(const std_msgs::Int64::ConstPtr& msg)
5 {
6     ROS_INFO("I heard: [%ld]", msg->data);
7 }
8
9 int main(int argc, char **argv)
10 {
11     ros::init(argc, argv, "subscriber");
12     ros::NodeHandle n;
13     ros::Subscriber sub = n.subscribe("ZabirovaKostyukova", 1000, chatterCallback);
14     ros::spin();
15
16     return 0;
17 }

```

Fig. 2. subscriber.cpp code

**T**HIS exercise helps us to understand the subscriber's function more deeply. We used the `Turtlesim` library to simulate the turtle GUI. Then, we wrote the "turtle listener" node that prints out the position and orientation of the turtle (Fig. 4).

## V. EXERCISE 3

**N**EXT we need to make the turtle moving. In the list of rostopics, we choose command velocities to control the turtle. For a 2D turtle, we can use three variables: `linear.x`, `linear.y`, and `angular.z`.

In .cpp file, we publish the velocities as messages and use the callback function to see the current pose of the turtle (Fig. 5).

If we set the `linear-x` and `angular-z` velocities to 1.0, the turtle will cover a circular path (Fig. 6).

## VI. EXERCISE 4

```

danissa@yerkubulan-HP-Z640-Workstation: ~/MachineLearning
[INFO] [1693304526.125713114]: 0
[INFO] [1693304526.135684120]: 9
[INFO] [1693304526.145651969]: 2
[INFO] [1693304526.155636457]: 0
[INFO] [1693304526.165668216]: 1
[INFO] [1693304526.175638264]: 8
[INFO] [1693304526.185620020]: 7
[INFO] [1693304526.195656479]: 4
[INFO] [1693304526.205635543]: 1
[INFO] [1693304526.215645876]: 0
[INFO] [1693304526.225646373]: 9
[INFO] [1693304526.235601689]: 2
[INFO] [1693304526.245653892]: 0
[INFO] [1693304526.255669394]: 1
[INFO] [1693304526.265647604]: 8
[INFO] [1693304526.275663270]: 7
[INFO] [1693304526.285693915]: 4
[INFO] [1693304526.295608139]: 1
[INFO] [1693304526.305634263]: 0
[INFO] [1693304526.315635144]: 0
[INFO] [1693304526.325625208]: 2
[INFO] [1693304526.335650456]: 0
[INFO] [1693304526.345732275]: 1
[INFO] [1693304525.266005395]: I heard: [1]
[INFO] [1693304525.276042965]: I heard: [8]
[INFO] [1693304525.285892630]: I heard: [7]
[INFO] [1693304525.296022573]: I heard: [4]
[INFO] [1693304525.305882740]: I heard: [1]
[INFO] [1693304525.315842637]: I heard: [9]
[INFO] [1693304525.326029683]: I heard: [9]
[INFO] [1693304525.335873833]: I heard: [2]
[INFO] [1693304525.345939960]: I heard: [9]
[INFO] [1693304525.356023101]: I heard: [1]
[INFO] [1693304525.365856169]: I heard: [8]
[INFO] [1693304525.375896234]: I heard: [7]
[INFO] [1693304525.385902595]: I heard: [4]
[INFO] [1693304525.395834001]: I heard: [1]
[INFO] [1693304525.405937091]: I heard: [0]
[INFO] [1693304525.415992270]: I heard: [9]
[INFO] [1693304525.425867837]: I heard: [2]
[INFO] [1693304525.435937810]: I heard: [0]
[INFO] [1693304525.446012565]: I heard: [1]
[INFO] [1693304525.455904182]: I heard: [8]
[INFO] [1693304525.465970471]: I heard: [7]
[INFO] [1693304525.475816712]: I heard: [4]
[INFO] [1693304525.486010507]: I heard: [1]
danissa@yerkubulan-HP-Z640-Workstation: $

```

Fig. 3. Publisher and Subscriber

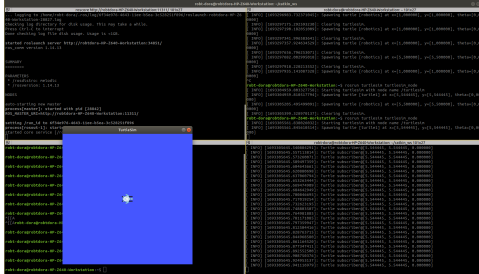


Fig. 4. A turtle in TurtleSim

FOR a Remote Procedure Call (RPC) we use ROS services. Two of the most used services in this exercise were `/clear` (to clear the path in the background) and `/kill` to remove a turtle.

We have also spawned a new turtle in the environment in a desired position (Fig. 7). For this, we created a client that sends a request to this service and defined the service message by setting the initial position of the spawned turtle (Fig. 8).

We can also publish desired velocities to the new turtle in order to control its movements and trajectories.

## VII. EXERCISE 5. FINAL EXERCISE.

THE spawned turtle can move along a specific trajectory. We used topics to control the turtle by changing its velocities in the terminal with this command:

```
rostopic pub /robotics/cmd_vel
geometry_msgs/Twist "linear:
```

Firstly, the turtle moves in a square. We changed only the x component of linear velocity to 10 coordinate units per second. Then we tuned the angular velocity to change in the z direction. Thus, the turtle turns to 90 degrees. The motion is repeated then (Fig. 9).

```

Open turtle_listener.cpp
~/catkin_ws/src/turtlebot_controller/src
Save

#include <ros/ros.h>
#include <turtlesim/Pose.h>

#include "geometry_msgs/Twist.h"

ros::Publisher pub;

void turtleCallback(const turtlesim::Pose::ConstPtr& msg)
{
    geometry_msgs::Twist my_vel;
    my_vel.linear.x = 1.0;
    my_vel.angular.z = 2.1;
    pub.publish(my_vel);

    ROS_INFO("Turtle subscriber[%f, %f, %f]",
    msg->x, msg->y, msg->theta);
}

int main (int argc, char **argv)
{
    ros::init(argc, argv, "turtlebot_subscriber");
    ros::NodeHandle nh;

    pub = nh.advertise<geometry_msgs::Twist>("robotics/cmd_vel", 1);

    ros::Subscriber sub = nh.subscribe("robotics/pose", 1, turtleCallback);
    ros::spin();
    return 0;
}

C++ Tab Width: 8 Ln 12, Col 32 INS

```

Fig. 5. C++ file to publish velocities

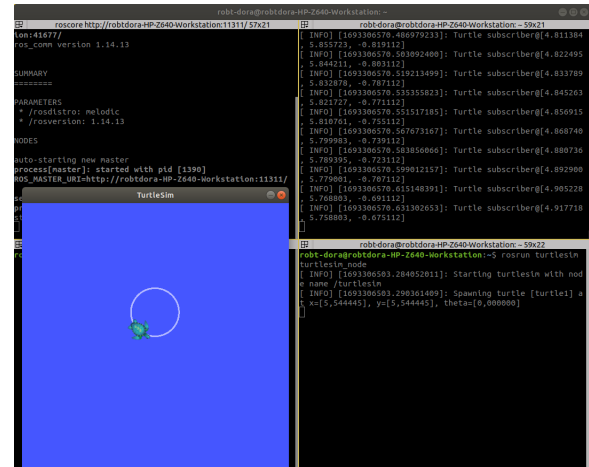


Fig. 6. A turtle moves in circular path

The second type of motion is the triangular path. The turtle spawns at [1, 1, 0] coordinate points. The pattern is similar to the square trajectory, except the angular velocity is increased to turn to a higher degree (Fig. 10).

## VIII. CONCLUSION

IN this laboratory, we delved into core concepts of the Robot Operating System (ROS). We grasped the fundamentals of message communication, mastering the publisher-subscriber model. Additionally, we explored ROS services, gaining insight into remote procedure calls. By simulating a 2D turtle and controlling its motion, we experienced practical robotic control. These exercises provided hands-on familiarity with ROS's power, setting the stage for more advanced applications in robotics and automation.

