BAHRIA UNIVERSITY ISLAMABAD

ROBOTICS LAB

LAB 3: ROBOT OPERATING SYSTEM - 2

Name of Student: …….

Roll No.: …….

Date of Experiment: …….

Report submitted on: ……..

Marks obtained: ……

Remarks: ……

Instructor's Signature: ……..

Fall 2021

Goals

By the end of this lab, you should be able to:

• Set up simple publisher and subscriber nodes and visualize their working in terminal.

• Create a custom message.

• Run multiple nodes at once using roslaunch.

• Record your experiment in .bag file.

Writing the publisher Node

Change directories to your beginner_tutorials package you created in your catkin workspace previous tutorials:

```
roscd beginner_tutorials
```

First lets create a 'scripts' folder to store our Python scripts in:

```
$ mkdir scripts
$ cd scripts
```

Then download the example script talker.py to your new scripts directory and make it executable:

```
$ wget https://raw.github.com/ros/ros_tutorials/kinetic-devel/rospy_tutorials
/001_talker_listener/talker.py
$ chmod +x talker.py
```

Add the following to your CMakeLists.txt. This makes sure the python script gets installed properly, and uses the right python interpreter.

```
catkin_install_python(PROGRAMS scripts/talker.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

You must have now a talker.py file inside the scrips folder. Lets now understand this code in c++.

Talker.cpp Node

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv) {
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(1);
  int count = 0;

  while (ros::ok()) {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();
    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
    }
  return 0;
}
```

Explanation:

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
```

- Ros/ros.h is a convenience header that includes most of the pieces necessary to run a Ros System.
- Std_msgs/String.h is the message type that we need to pass in this exmole
    - You will have to include a different header if you want to use a different message type
- Sstream is responsible for some string manipulations in C++

```
ros::init(argc, argv, "talker");
ros::NodeHandle n;
```

• ros::init is responsible for collecting ROS specific information from arguments passed at the command line

- It also takes in the name of our node
- Remember that node names need to be unique in a running system
- We'll see an example of such an argument in the next example

- The creation of a ros::NodeHandle object does a lot of work

o It initializes the node to allow communication with other ROS nodes and the master in the ROS infrastructure

- Allows you to interact with the node associated with this process

```
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
ros::Rate loop_rate(1);
```

- NodeHandle::advertise is responsible for making the XML/RPC call to the ROS Master advertising std_msgs::straing on the topic named “chatter”
- Loop rate is used to maintain the frequency of publishing at 1 Hz(i.e 1message per second)

```
int count = 0;
while (ros::ok()) {
```

- count is used to keep track of the number of messages transmitted. Its value is attached to the string message that is published

- ros ok() ensures that everything is still alright in the ROS framework. If something is amiss, then it will return false effectively terminating the program. Examples of situations where it will return false:
  o You Ctrl+c the program (SIGINT)
  o You open up another node with the same name.
  o You call ros: shutdown() somewhere in your code

```
std_msgs::String msg;
std::stringstream ss;
ss << "hello world " << count;
msg.data = ss.str();
```

- These 4 lines do some string manipulation to put the count inside the string message.
  - The reason we do it this way is that c++ does not have a good equivalent toString function
- msg.data is a std::string

```
ROS_INFO("%s", msg.data.c_str());
chatter_pub.publish(msg);
```

- ROS INFO is a macro that publishes a information message in the ROS ecosystem. By default ROS INFO messages are also published to the screen.
  - There are debug tools in ROS that can read these messages
  - You can change what level of messages you want to be have published

- ros::Publisher::publish() sends the message to all subscribers.

```
ros::spinOnce();
loop_rate.sleep();
++count;
```

- ros:spin Once() is analogous to the main function of the ROS framework
  - Whenever you are subscribed to one or many topics, the callbacks for receiving messages on those topics are not called immediately.
  - Instead they are placed in a queue which is processed when you call ros spinOnce()
  - What would happen if we remove the spinOnce() call?
- ros::Rate :sleep() helps maintain a particular publishing frequency
- count is incremented to keep track of messages

Writing the subscriber Node

Download the listener.py file into your scripts directory:

```
$ roscd beginner_tutorials/scripts/

$ wget https://raw.github.com/ros/ros_tutorials/kinetic-devel/rospy_tutorials
/001_talker_listener/listener.py

$ chmod +x listener.py
```

Then, edit the catkin_install_python() call in your CMakeLists.txt so it looks like the following

```
catkin_install_python(PROGRAMS scripts/talker.py scripts/listener.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

You must have now a listener.py file inside the scrips folder. Lets now understand this code in c++.

Listerner.cpp Node:

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr msg) {
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub =
    n.subscribe<std_msgs::String>("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

Explanation:

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr msg) {
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

- Same headers as before.
- chatterCallback() is a function we have defined that gets called whenever we receive a message on the subscribed topic.
- It has a well typed argument.

```
int main(int argc, char **argv) {
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub =
    n.subscribe<std_msgs::String>("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

- ros::Node Handle::subscribe makes an XML/RPC call to the ROS master
  - It subscribes to the topic chatter
  - 1000 is the queue size. In case we are unable to process messages fast enough. This is only useful in case of irregular processing times of messages. Why?
  - The third argument is the callback function to call whenever we receive a message
- rosspin() a convenience function that loops around ros: spinOnce() while checking ros ok()

Output

Compile the code using “catkin_make” Run roscore and then enter command “rosrun beginner_tutorials talker.py” to run the talker node and “rosrun beginner_tutorials listener.py” to run listener node.

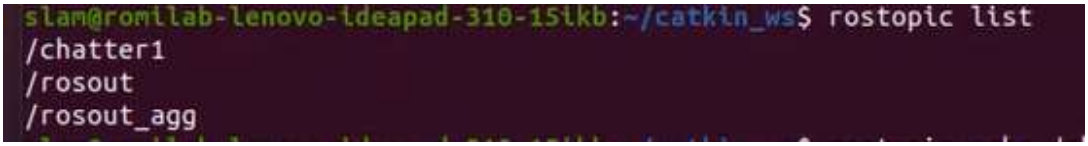
Output of Talker Node:

```
[ INFO] [1605701279.128505747]: hello world 0
[ INFO] [1605701279.228535038]: hello world 1
[ INFO] [1605701279.328599817]: hello world 2
[ INFO] [1605701279.428536082]: hello world 3
[ INFO] [1605701279.528542966]: hello world 4
[ INFO] [1605701279.628540471]: hello world 5
[ INFO] [1605701279.728567284]: hello world 6
[ INFO] [1605701279.828542563]: hello world 7
[ INFO] [1605701279.928542487]: hello world 8
[ INFO] [1605701280.028535132]: hello world 9
[ INFO] [1605701280.128556488]: hello world 10
[ INFO] [1605701280.228544117]: hello world 11
[ INFO] [1605701280.328555614]: hello world 12
[ INFO] [1605701280.428574961]: hello world 13
[ INFO] [1605701280.528526970]: hello world 14
[ INFO] [1605701280.628526945]: hello world 15
[ INFO] [1605701280.728544360]: hello world 16
[ INFO] [1605701280.828565403]: hello world 17
[ INFO] [1605701280.928551751]: hello world 18
[ INFO] [1605701281.028547905]: hello world 19
[ INFO] [1605701281.128533538]: hello world 20
[ INFO] [1605701281.228526779]: hello world 21
```

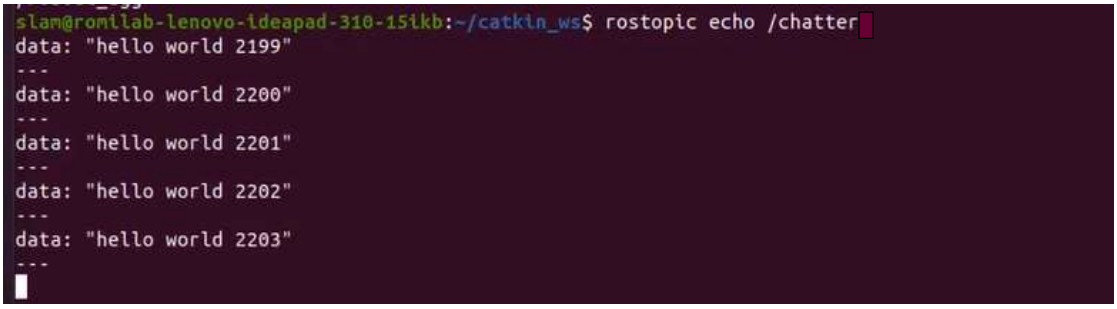
Output of Listener node:

Output of rosnode list

Output of rostopic list



Output of rostopic echo /chatter



Creating a Custom Message

- msg files are simple text files that describe the fields of a ROS message.
- Create a directory 'msg' within the package and inside the directory create a .msg file.
- Inside the .msg file, write what variables you want to include in your custom message along with their datatype.
- Build the package, make necessary changes in CMakeLists.txt and package.xml
- You can now use the msg file.
    - Rosmsg show [packegename]/msgname

Open package.xml, and make sure these two lines are in it and uncommented:

```
<build_depend>message_generation</build_depend>
  <exec_depend>message_runtime</exec_depend>
```

ROS Launch

- Roslaunch is used to run many nodes at once with one command.

First go to the beginner_tutorials package we created and built earlier:

```
$ roscd beginner_tutorials
```

If roscd says something similar to roscd: No such package/stack 'beginner_tutorials' , you will need to source the environment setup file like you did at the end of the create_a_workspace tutorial:

```
$ cd ~/catkin_ws
$ source devel/setup.bash
$ roscd beginner_tutorials
```

Then let's make a launch directory:

```
$ mkdir launch
$ cd launch
```

Create a .launch file and write this code in it.

```
<launch>

<node
        pkg = "beginner_tutorials"
        type = "talker"
        name = "talker"
/>

<node
        pkg = "beginner_tutorials"
        type = "listener"
        name = "listener"
        respawn="true"
/>

<node
        pkg = "beginner_tutorials"
        type = "customMessageCheck"
        name = "customMessageCheck"
        respawn="true"
/>

</launch>
```

Run the launch file using:

```
$ roslaunch [package] [filename.launch]
```

ROS Bag

- Ros bag is used for recording and playing back data.
- Commands
    - **rosbag record -a** : records the environment and makes .bag file in the directory
    - **rosbag info <bag file>** : tells information about the bag file like its size, duration etc.
    - **rosbag play <bag file>** : plays the the recorded bag file

Tasks

- Create a publisher and subscriber node, compile it using “catkin_make” and run the nodes. Attach the screenshots and describe publishing node is publishing which message to what topic and similarly subscribing node is subscribing to which topic.
- Create a custom message, publish it from one node and subscribe it from another node.
- Create launch a launch file and run both publisher/subscriber node using single command.
- Capture the environment using rosbag and run it and attach the screenshots.