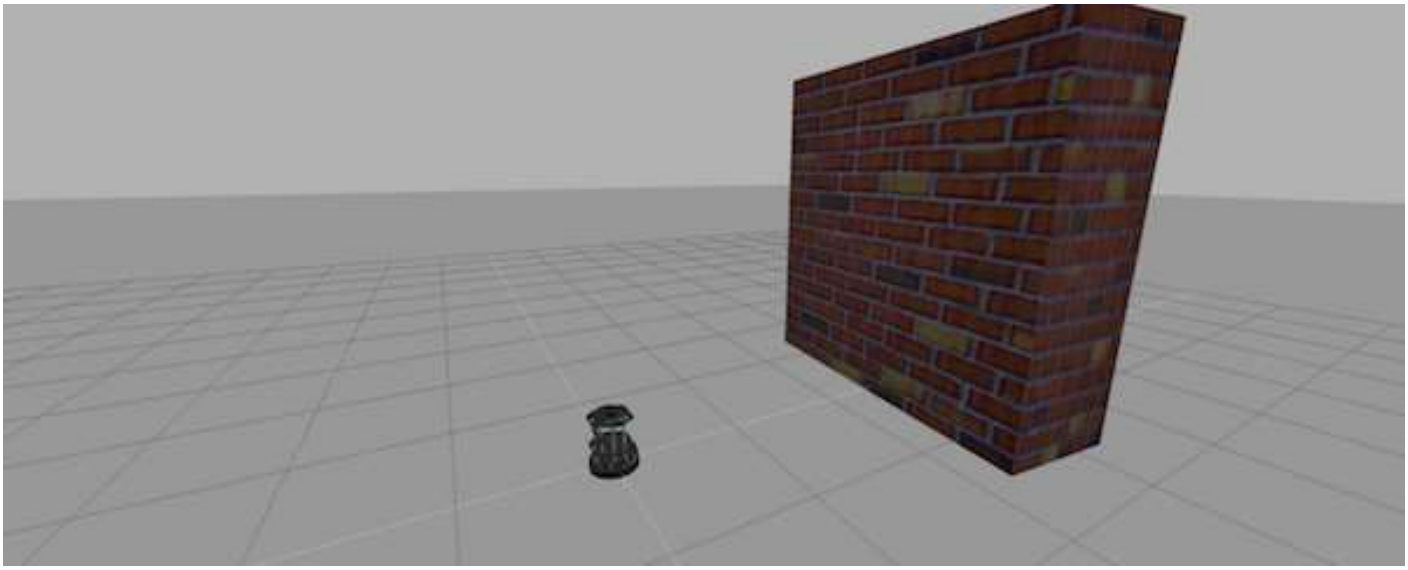
ROS Basics in 5 days (C++)

Unit 3 Understanding ROS Topics: Publishers



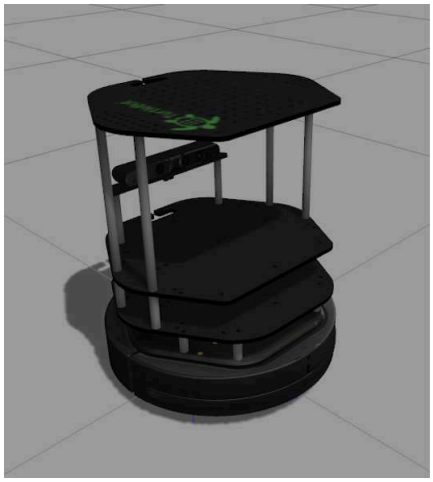
- Summary -

Estimated time to completion: 2.5 hours

Simulation: Turtlebot

What will you learn with this unit?

- What are ROS topics and how to manage them
- What is a publisher and how to create one
- What are topic messages and how they work



- End of Summary -

3.1 Topic Publisher

- Exercise 3.1 -

- Create a new package named **topic_publisher_pkg**. When creating the package, add as dependencies **roscpp** and **std_msgs**.
- Inside the src folder of the package, create a new file named **simple_topic_publisher.cpp**. Inside this file, copy the contents of simple_topic_publisher.cpp
- Create a launch file for launching this code.
- Do the necessary modifications to your **CMakeLists.txt** file, and compile the package.
- Execute the launch file to run your executable.

- End of Exercise 3.1 -

- Notes for Exercise 3.1 -

1.- In order to do this exercise, you can simply follow the same steps you made in the previous Chapter. It is almost the same.

2.- Remember, in order to create a package with **roscpp** and **std_msgs** as dependencies, you should use a command like the below one:

In [ ]:
```
catkin_create_pkg topic_publisher_pkg roscpp std_msgs
```

3.- The lines to add into the **CmakeLists.txt** file could be something like this:

In [ ]:
```
add_executable(simple_topic_publisher src/simple_topic_publisher.cpp)
add_dependencies(simple_topic_publisher ${simple_topic_publisher_EXPORTED_TARG
target_link_libraries(simple_topic_publisher
   ${catkin_LIBRARIES}
 )
```

- End of Notes -

C++ Program {3.1}: simple_topic_publisher.cpp

In [ ]:
```
#include <ros/ros.h>
#include <std_msgs/Int32.h>

int main(int argc, char** argv) {

    ros::init(argc, argv, "topic_publisher");
    ros::NodeHandle nh;

    ros::Publisher pub = nh.advertise<std_msgs::Int32>("counter", 1000);
    ros::Rate loop_rate(2);

    std_msgs::Int32 count;
    count.data = 0;

    while (ros::ok())
    {
        pub.publish(count);
        ros::spinOnce();
        loop_rate.sleep();
        ++count.data;
    }

    return 0;
}
```

END C++ Program {3.1}: simple_topic_publisher.cpp

Nothing happens? Well... that's not actually true! You have just created a topic named **/counter**, and published through it as an integer that increases indefinitely. Let's check some things.

A topic is like a pipe. **Nodes use topics to publish information for other nodes** so that they can communicate. You can find out, at any time, the number of topics in the system by doing a ***rostopic list***. You can also check for a specific topic.

On your webshell, type ***rostopic list*** and check for a topic named '/counter'. Remember that the node publishing the topic must be running.

▶ Execute in Shell #2

In [ ]:
```
rostopic list | grep  '/counter'
```

Shell #2 Output

In [ ]:
```
user ~ $ rostopic list | grep '/counter'
/counter
```

Here, you have just listed all of the topics running right now and filtered with the **grep** command the ones that contain the word */counter*. If it appears, then the topic is running as it should.

You can request information about a topic by doing ***rostopic info <name_of_topic>***.

Now, type ***rostopic info /counter***.

▶ Execute in Shell #2

In [ ]:
```
rostopic info /counter
```

Shell #2 Output

In [ ]:
```
user ~ $ rostopic info /counter
Type: std_msgs/Int32

Publishers:
 * /topic_publisher (http://ip-172-31-16-133:47971/)

Subscribers: None
```

The output indicates the type of information published (**std_msgs/Int32**), the node that is publishing (**/topic_publisher**), and if there is a node listening to that info (None in this case).

Now, type ***rostopic echo /counter*** and check the output of the topic in realtime.

▶ Execute in Shell #2

In [ ]:
```
rostopic echo /counter
```

You should see a succession of consecutive numbers, similar to the following:

Shell #2 Output

In [ ]:

```
rostopic echo /counter
data:
985
---
data:
986
---
data:
987
---
data:
988
---
```

Ok, so... what has just happened? Let's explain it in more detail. First, let's crumble the code we've executed. You can check the comments in the code below explaining what each line of the code does:

In [ ]:
```
#include <ros/ros.h>
#include <std_msgs/Int32.h>
// Import all the necessary ROS libraries and import the Int32 message from th

int main(int argc, char** argv) {

    ros::init(argc, argv, "topic_publisher"); // Initiate a Node named 'topic_
    ros::NodeHandle nh;

    ros::Publisher pub = nh.advertise<std_msgs::Int32>("counter", 1000); // Cr
                                                                         // of
    ros::Rate loop_rate(2); // Set a publish rate of 2 Hz

    std_msgs::Int32 count; // Create a variable of type Int32
    count.data = 0; // Initialize 'count' variable

    while (ros::ok()) // Create a loop that will go until someone stops the pr
    {
        pub.publish(count); // Publish the message within the 'count' variable
        ros::spinOnce();
        loop_rate.sleep(); // Make sure the publish rate maintains at 2 Hz
        ++count.data; // Increment 'count' variable
    }

    return 0;
}
```

So basically, what this code does is to **initiate a node and create a publisher that keeps publishing into the '/counter' topic a sequence of consecutive integers**. Summarizing:

A publisher is a node that keeps publishing a message into a topic. So now... what's a topic?

A topic is a channel that acts as a pipe, where other ROS nodes can either publish or read information. Let's now see some commands related to topics (some of them you've already used).

To **get a list of available topics** in a ROS system, you have to use the next command:

In [ ]:
```
rostopic list
```

To **read the information that is being published in a topic,** use the next command:

In [ ]:
```
rostopic echo <topic_name>
```

This command will start printing all of the information that is being published into the topic, which sometimes (ie: when there's a massive amount of information, or when messages have a very large structure) can be annoying. In this case, you can **read just the last message published into a topic** with the next command:

In [ ]:
```
rostopic echo <topic_name> -n1
```

To **get information about a certain topic,** use the next command:

In [ ]:
```
rostopic info <topic_name>
```

Finally, you can check the different options that *rostopic* command has by using the next command:

In [ ]:
```
rostopic -h
```

IMPORTANT NOTE

When you have finished with this section of the Notebook, make sure to **STOP** the previously executed code by selecting the cell with the code and clicking on the **Interrupt kernel** button at the top right corner of the Notebook. This is very important for doing the Next Unit properly.

IMPORTANT NOTE

3.2 Messages

As you may have noticed, topics handle information through messages. There are many different types of messages.

In the case of the code you executed before, the message type was an **std_msgs/Int32**, but ROS provides a lot of different messages. You can even create your own messages, but it is recommended to use ROS default messages when its possible.

Messages are defined in **.msg** files, which are located inside a **msg** directory of a package.

To **get information about a message,** you use the next command:

In [ ]:
```
rosmsg show <message>
```

- Example 3.1 -

For example, let's try to get information about the std_msgs/Int32 message. Type the following command and check the output.

▶ Execute in Shell #1

In [ ]:
```
rosmsg show std_msgs/Int32
```

Shell #1 Output

In [ ]:
```
user ~ $ rosmsg show std_msgs/Int32
[std_msgs/Int32]:
int32 data
```

In this case, the **Int32** message has only one variable named **data** of type **int32.** This Int32 message comes from the package std_msgs, and you can find it in its **msg** directory. If you want, you can have a look at the Int32.msg file by executing the following command:

In [ ]:
```
roscd std_msgs/msg/
```

- End of Example 3.1 -

Now you're ready to create your own publisher and make the robot move, so let's go for it!

- Exercise 3.2 -

- Modify the code you used previously so that now it publishes data to the /cmd_vel topic.
- Compile again your package.
- Launch the program and check that the robot moves.

- End of Exercise 3.2 -

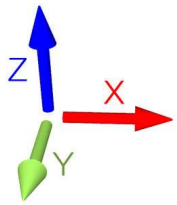
- Notes for Exercise 3.2 -

1.- The /cmd_vel topic is the topic used to move the robot. Do a ***rostopic info /cmd_vel*** in order to get information about this topic, and identify the message it uses. You have to modify the code to use that message.

2.- In order to fill the Twist message, you need to create an instance of the message. In C++, this is done like this: **geometry_msgs::Twist var;**

3.- In order to know the structure of the Twist messages, you need to use the **rosmsg show** command, with the type of the message used by the topic /cmd_vel.

4.- In this case, the robot uses a differential drive plugin to move. That is, the robot can only move linearly in the **x** axis, or rotationaly in the angular **z** axis. This means that the only values that you need to fill in the Twist message are the linear **x** and the angular **z**.



5.- The magnitudes of the Twist message are in m/s, so it is recommended to use values between 0 and 1. For example, 0'5 m/s.

- End of Notes -