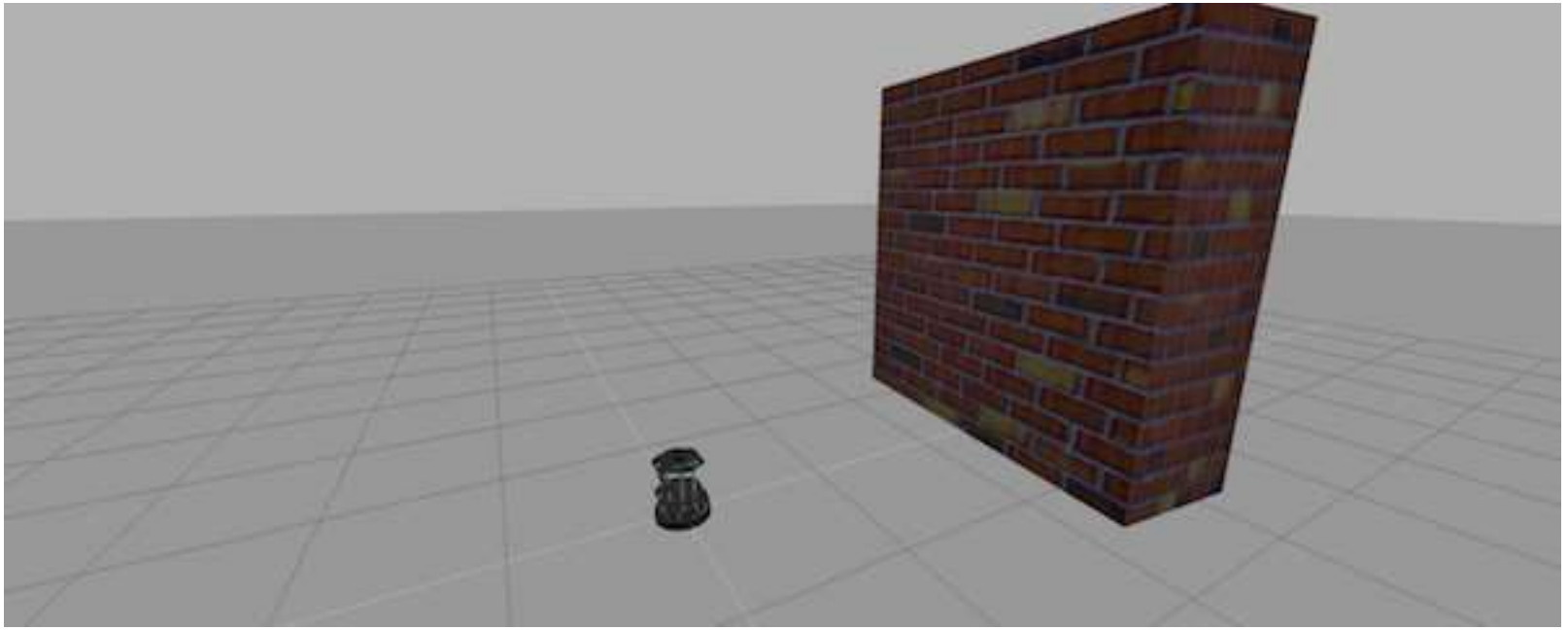
ROS Basics in 5 days

Unit 3 Understanding ROS Topics: Publishers



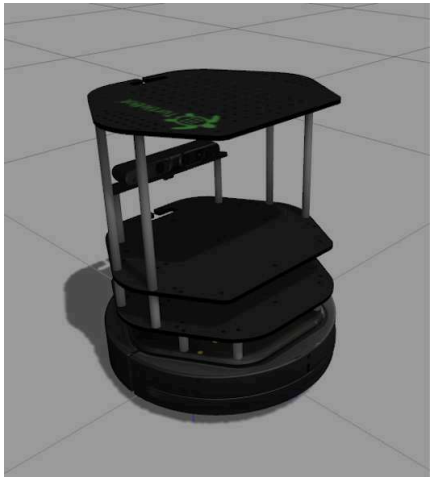
- Summary -

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

- Example 3.1 -

For this and following example, you should create a ROS package named **my_publisher_example_pkg**. Within this ROS package you will place all the files that you will create in this unit to keep an organized structure.

- First, create the package **my_publisher_example_pkg**, executing the following comands in the Web Terminal:

▶ Execute in Terminal #1

In [ ]:
```
cd ~/catkin_ws/src
```

In [ ]:
```
catkin_create_pkg my_publisher_example_pkg rospy std_msgs
```

- Next, create a new folder inside your ROS package named **scripts**:

▶ Execute in Terminal #1

In [ ]:
```
cd ~/catkin_ws/src/my_publisher_example_pkg
```

In [ ]:
```
mkdir scripts
```

- Inside the **scripts** folder, create the example Python script named **simple_topic_publisher.py**. Remember that you can also create it using the IDE.

▶ Execute in Terminal #1

In [ ]:
```
cd ~/catkin_ws/src/my_publisher_example_pkg/scripts
```

In [ ]:
```
touch simple_topic_publisher.py
```

In [ ]:
```
chmod +x simple_topic_publisher.py
```

- Paste the following code into the **simple_topics_publisher.py** script:

Python Program: simple_topic_publisher.py

In [ ]:
```
#! /usr/bin/env python

import rospy
from std_msgs.msg import Int32

rospy.init_node('topic_publisher')
pub = rospy.Publisher('/counter', Int32, queue_size=1)
rate = rospy.Rate(2)
count = Int32()
count.data = 0

while not rospy.is_shutdown():
  pub.publish(count)
  count.data += 1
  rate.sleep()
```

END Python Program: simple_topic_publisher.py

- And don't forget to compile and source your workspace!

▶ Execute in Terminal #1

In [ ]:
```
cd ~/catkin_ws
```

In [ ]:
```
catkin_make
```

In [ ]:
```
source devel/setup.bash
```

- Now you can execute the script with the command ***rosrun***:

▶ Execute in Terminal #1

In [ ]:
```
rosrun my_publisher_example_pkg simple_topic_publisher.py
```

- End Example 3.1 -

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 Terminal, type `rostopic list` and check for a topic named `/counter`.

▶ Execute in Terminal #2

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

Terminal #2 Output

```
/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 Terminal #2

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

Terminal #2 Output

```
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 this information (**/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 Terminal #2

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

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

Terminal #2 Output

```
data:
985
---
data:
986
---
data:
987
---
data:
988
---
```

Note: If you want, you can stop the message flow using **Ctrl+C**.

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 [ ]:

```
#! /usr/bin/env python

# Import the Python library for ROS
import rospy
# Import the Int32 message from the std_msgs package
from std_msgs.msg import Int32


# Initiate a Node named 'topic_publisher'
rospy.init_node('topic_publisher')

# Create a Publisher object, that will publish on the /counter topic
# messages of type Int32
pub = rospy.Publisher('/counter', Int32, queue_size=1)

# Set a publish rate of 2 Hz
rate = rospy.Rate(2)
# Create a variable of type Int32
count = Int32()
# Initialize 'count' variable
count.data = 0

# Create a loop that will go until someone stops the program execution
while not rospy.is_shutdown():
  # Publish the message within the 'count' variable
  pub.publish(count)
  # Increment 'count' variable
  count.data += 1
  # Make sure the publish rate maintains at 2 Hz
  rate.sleep()
```

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 selecting the Web Terminal #1 and pressing **Ctrl+C**. 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 3.2 -

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

▶ Execute in Terminal #1

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

Terminal #1 Output

```
[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 Example 3.2 -

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

- Exercise 3.1 -

- Create a package with a launch file that launches the code simple_topic_publisher.py.
- Modify the code you used previously to publish data to the **/cmd_vel** topic.
- Launch the program and check that the robot moves.

- End Exercise 3.1 -

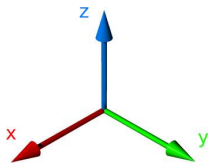
- Notes for Exercise 3.1 -

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 Python, this is done like this: **var = Twist()**

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.

6.- Refer back to the previous unit if you need a refresher on launch files.

- End Notes for Exercise 3.1 -