Publishing to a Topic

Example 3-1 shows the basic code for advertising a topic and publishing messages on it. This node publishes consecutive integers on the topic `counter` at a rate of 2Hz.

Example 3-1. topic_publisher.py

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

import roslib; roslib.load_manifest('basics')
import rospy

from std_msgs.msg import Int32


rospy.init_node('topic_publisher')

pub = rospy.Publisher('counter', Int32)

rate = rospy.Rate(2)

count = 0
while not rospy.is_shutdown():
    pub.publish(count)
    count += 1
    rate.sleep()
```

The first three lines

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

import roslib; roslib.load_manifest('basics')
import rospy
```

should be familiar from (to come). The next line

```
from std_msgs.msg import Int32
```

imports the definition of the message that we're going to send over the topic. In this case, we're going to use a 32-bit integer, defined in the ROS standard message package, `std_msgs`. For the import to work as expected, we need to import from `<package name>.msg`, since this is where the package definitions are stored (more on this later). Since we're using a message from another package, we have to tell the ROS build system about this by adding a *dependency* to our `manifest.xml` file.

```
<depend package="std_msgs" />
```

Without this dependency, ROS will not know where to find the message definition and the node will not be able to run.

After initializing the node, we advertise it with a `Publisher`.

```
pub = rospy.Publisher('counter', Int32)
```

This gives the topic a name (`counter`), and specifies the type of message that will be sent over it (`Int32`). Behind the scenes, the publisher also sets up a connection to `roscore`, and sends some information to it. When another node tries to subscribe to the `counter` topic, `roscore` will share its list of publishers and subscribers, which the nodes will then use to create direct connections between all publishers and all subscribers of each topic.

At this point, the topic is advertised and is available for other nodes to subscribe to. Now we can go about actually publishing messages over the topic.

```
rate = rospy.Rate(2)

count = 0
while not rospy.is_shutdown():
    pub.publish(count)
    count += 1
    rate.sleep()
```

First, we set the rate, in Hz, at which we want to publish. For this example, we're going to publish twice a second. The `is_shutdown()` function will return `True` if the node is ready to be shutdown, and `False` otherwise, so we can use this to determine if it is time to exit the `while` loop.

Inside the `while` loop, we publish the current value of the counter, increment its value by 1, and then sleep for a while. The call to `rate.sleep()` will sleep for long enough to make sure that we run the body of the `while` loop at approximately 2Hz.

And that's it. We now have a minimalist ROS node that advertises the `counter` topic, and publishes integers on it.

Checking that Everything Works as Expected

Now that we have a working node, let's verify that it works. We can use the `rostopic` command to dig into the currently available topics. Open a new terminal, and start up `roscore`. Once it's running, you can see what topics are available by running in another terminal.

```
user@hostname$ rostopic list
/rosout
/rosout_agg
```

These topics are used by ROS for logging and debugging; don't worry about them. Now, run the node we've just looked at in (yet another) terminal.

```
user@hostname$ rosrun basics topic_publisher.py
```

Remember that the `basics` directory has to be in your `ROS_PACKAGE_PATH`, and that, if you typed in the code for the node yourself, the file will need to have its execute permissions set using `chmod`. Once the node is running, you can verify that the `counter` topic is advertised by running `rostopic list` again.

```
user@hostname$ rostopic list
/counter
/rosout
/rosout_agg
```

Even better, you can see the messages being published to the topic by running `rostopic echo`.

```
user@hostname$ rostopic echo counter -n 5
data: 681
---
data: 682
---
data: 683
---
data: 684
---
data: 685
---
```

The `-n 5` flag tells `rostopic` to only print out 5 messages. Without it, it will happily go on printing messages forever, until you stop it with a `ctrl-c`. We can also use `rostopic` to verify that we're publishing at the rate we think we are.

```
user@hostname$ rostopic hz counter
subscribed to [/counter]
average rate: 2.000
        min: 0.500s max: 0.500s std dev: 0.00000s window: 2
average rate: 2.000
        min: 0.500s max: 0.500s std dev: 0.00004s window: 4
average rate: 2.000
        min: 0.500s max: 0.500s std dev: 0.00006s window: 6
average rate: 2.000
        min: 0.500s max: 0.500s std dev: 0.00005s window: 7
```

`rostopic hz` has to be stopped with a `ctrl-c`. Similarly, `rostopic bw` will give information about the bandwidth being used for by the topic.

You can also find out about an advertised topic with `rostopic info`.

```
user@hostname$ rostopic info counter
Type: std_msgs/Int32

Publishers:
 * /topic_publisher (http://hostname:39964/)

Subscribers: None
```

This reveals that `counter` carries messages of type `std_msgs/Int32`, that it is currently being advertised by `topic_publisher`, and than no-one is currently subscribing to it. Since it's possible for more than one node to publish to the same topic, and for more than one node to be subscribed to a topic, this command can help you make sure things are connected in the way that you think they are. The publisher `topic_publisher` is running on the computer `hostname` and is communicating over TCP port 39964+.[1] `rostopic type` works similarly, but only returns the message type for a given topic.

Finally, you can find all of the topics that publish a certain message type using `rostopic find`.

```
user@hostname$ rostopic find std_msgs/Int32
/counter
```

Note that you have to give both the package name (`std_msgs`) and the message type (`Int32`) for this to work.

So, now we have a node that's happily publishing consecutive integers, and we can verify that everything is as it should be. Now, let's turn our attention to a node that subscribes to this topic and uses the messages it is receiving.

Subscribing to a Topic

Example 3-2 shows a minimalist node that subscribes to the `counter` topic, and prints out the values in the messages as they arrive.

Example 3-2. topic_subscriber.py

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

import roslib; roslib.load_manifest('basics')

import rospy
from std_msgs.msg import Int32


def callback(msg):
    print msg.data


rospy.init_node('topic_subscriber')

sub = rospy.Subscriber('counter', Int32, callback)

rospy.spin()
```

The first interesting part of this code is the *callback* that handles the messages as they come in.

1. Don't worry if you don't know what a TCP port is. ROS will generally take care of this for you without you having to think about it.

```
def callback(msg):
    print msg.data
```

ROS is an event-driven system, and it uses callback functions heavily. Once a node has subscribed to a topic, every time a message arrives on it, the associated callback function is called, with the message as its parameter. In this case, the function simply prints out the data contained in the message (see "Defining Your Own Message Types" on page 25 for more details about messages and what they contain).

After initializing the node, as before, we subscribe to the `counter` topic.

```
sub = rospy.Subscriber('counter', Int32, callback)
```

We give the name of the topic, the message type on the topic, and the name of the callback function. Behind the scenes, the subsciber passes this information on to `roscore`, and tries to make a direct connection with the publishers of this topic. If the topic does not exist, or if the type is wrong, there are no error messages: the node will simply have nothing to do until messages start being published on the topic.

Once the subscription is made, we give control over to ROS by running `rospy.spin()`. This function will only return when the node is ready to shut down. This is just a useful shortcut to avoid having to define a top-level `while` loop like we did in Example 3-1; ROS does not necessarily need to "take over" the main thread of execution.

Checking that Everything Works as Expected

First, make sure that the publisher node is still running, and that it is still publishing messages on the `counter` topic. Now, in another terminal, start up the subscriber node.

```
user@hostname$ rosrun basics topic_subscriber
355
356
357
358
359
360
```

It should start to print out integers published to the `counter` topic by the publisher node. Congratulations! You're now running your first ROS system: Example 3-1 is sending messages to Example 3-2. You can visualize this system by typing `rqt_graph`, which will attempt to draw the publishers and subscribers in a logical manner.

We can also publish messages to a topic from the command line using `rostopic pub`. Run the following command, and watch the output of the subscriber node.

```
user@hostname$ rostopic pub counter std_msgs/Int32 1000000
```

We can use `rostopic info` again to make sure things are the way we expect them to be.

```
user@hostname$ rostopic info counter
Type: std_msgs/Int32

Publishers:
 * /topic_publisher (http://hostname:46674/)

Subscribers:
 * /topic_subscriber (http://hostname:53744/)
```

Now that you understand how basic topics work, we can talk about special types of topics, designed for nodes that only publish data infrequently, called *latched topics*.

Latched Topics

Messages in ROS are fleeting. If you're not subscribed to a topic when a message goes out on it, you'll miss it and will have to wait for the next one. This is fine if the publisher sends out messages frequently, since it won't be long until the next message comes along. However, there are cases where sending out frequent messages is a bad idea.

For example, the `map_server` node advertises a map (of type `nav_msgs/OccupancyGrid`) on the `map` topic. This represents a map of the world that the robot can use to determine where it is, such as the one shown in Figure 3-1. Often, this map never changes, and is published once when the `map_server` loads it from disk. However, if another node needs the map, but starts up after `map_server` publishes it, it will never get the message.

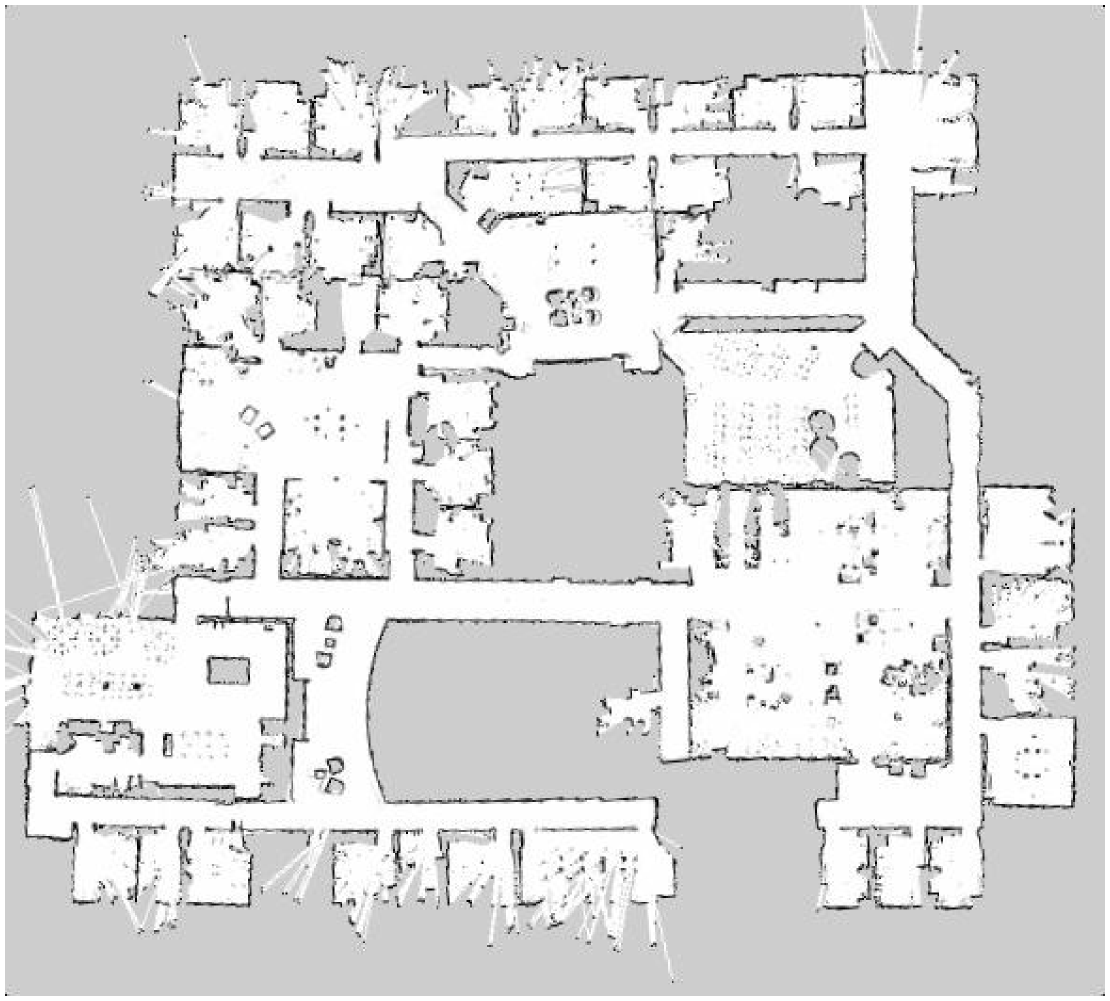

Figure 3-1. An example map.

We could periodically publish the map, but we don't want to publish the message more often than we have to, since it's typically huge. Even if we did decide to republish it, we would have to pick a suitable frequency, which might be tricky to get right.

Latched topics offer a simple solution to this problem. If a topic is marked as latched when it is advertised, subscribers automatically get **the last message sent** when they subscribe to the topic. In our example of `map_server`, this means that we only need to mark it as latched, and publish it once. Topics can be marked as latched with the optional `latch` argument.

```
pub = rospy.Publisher('map', nav_msgs/OccupancyGrid, latched=True)
```

Now that we know how to send messages over topics, it's time to think about what to do if we want to send a message that isn't already defined by ROS.