ROS Made Easy
1: Workspaces, Packages and Rospy

Aniruddh K Budhgavi
Enigma, IIIT-B

June 18, 2020

1 Important Note

This tutorial was created for **ROS1 Melodic Morenia** on **Ubuntu 18.04 Bionic Beaver**, in **June 2020**. I expect them to become rapidly out of date. It is my hope that Team Enigma will continually maintain and update these tutorials.

This tutorial assumes that you are running Ubuntu, and have at least an elementary grasp of Python 2.7 and C/C++ .

All the code for this tutorial is available at `https://github.com/aniruddhkb/enigmatutorials`.

The aim of this tutorial is to make you *functional* in ROS, not to make you a master. For that, look elsewhere.

2 Creating a Catkin workspace

1. ROS uses a package building system called Catkin. Catkin is built on top of CMake, which enables us to include files from many different directories in a simple way.
2. Run the following commands

```
:~$ mkdir workspaces
:~$ cd workspaces
:~/workspaces$ mkdir intro2ros
:~/workspaces/intro2ros$ cd intro2ros
:~/workspaces/intro2ros$ mkdir src
:~/workspaces/intro2ros$ catkin_make
```

   You should be fine if the last line of the output is something like:

```
-- Build files have been written to:
<path>/intro2ros/workspaces/intro2ros/build
####
#### Running command: "make -j8 -l8" in
"<path>/intro2ros/workspaces/intro2ros/build"
####
```

   This sequence of commands:

   (a) Creates a folder for our workspaces.
   (b) Creates a workspace `intro2ros`
   (c) Builds the workspace using the `catkin_make` command.

3. A **workspace** is a folder where Catkin packages can be built, modified and installed. For some projects, a single workspace is fine, but I like to segment related packages into separate workspaces. Hence the need for a separate folder for them.

4. The `catkin_make` command is used to build all the packages present in that workspace. `catkin_make` **must always be run only from the workspace folder**. The command may fail if there is no `src` folder in the workspace.

5. To make sure that ROS recognises packages in your workspace, add the following line to your `.bashrc`:

```
source <path-to-workspace>/intro2ros/devel/setup.bash
```

   Close and restart the terminal. Note: If you have mistyped the above command, you may see something like this when you reopen the terminal:

```
bash: <path>/intro2ros/devel/setup.bash:
No such file or directory
```

6. The source code for the packages in the workspace should be present in the `src` folder of the workspace.

7. For more information, see `http://wiki.ros.org/catkin/workspaces` and `http://wiki.ros.org/catkin/Tutorials/create_a_workspace`.

3 Creating a package

1. Run:

```
:~$ roscd
:.../workspaces/intro2ros/devel$ cd ../src
:.../workspaces/intro2ros/src$ catkin_create_pkg intro2rospy rospy std_msgs
        geometry_msgs turtlesim
```

   Output:

```
Created file intro2rospy/CMakeLists.txt
Created file intro2rospy/package.xml
Created folder intro2rospy/src
Successfully created files in <path>/intro2ros/src/intro2rospy.
Please adjust the values in package.xml.
```

   The `catkin_create_pkg` command is used to create packages and **must be run in the `src` folder** of your workspace.

2. To build all the packages in a workspace, run `catkin_make` from the workspace directory.

3. Go to the package folder and see what files are there.

```
:.../intro2rospy$ ls
```

   Output:

```
CMakeLists.txt  package.xml  src
```

4. `src` is a folder for C++ source files (of which we have none for this package). `package.xml` is a manifest file which tells us and ROS about the package and its dependencies. `CMakeLists.txt` is a configuration file for compilation using CMake.

5. Let's examine the package manifest. Removing the comments, it looks like:

```
<?xml version="1.0"?>
<package format="2">
    <name>intro2rospy</name>
    <version>0.0.0</version>
    <description>The intro2rospy package</description>

    <maintainer email="akb@todo.todo">akb</maintainer>

    <license>TODO</license>

    <buildtool_depend>catkin</buildtool_depend>
    <build_depend>geometry_msgs</build_depend>
    <build_depend>rospy</build_depend>
    <build_depend>std_msgs</build_depend>
    <build_depend>turtlesim</build_depend>
    <build_export_depend>geometry_msgs</build_export_depend>
    <build_export_depend>rospy</build_export_depend>
    <build_export_depend>std_msgs</build_export_depend>
    <build_export_depend>turtlesim</build_export_depend>
    <exec_depend>geometry_msgs</exec_depend>
    <exec_depend>rospy</exec_depend>
    <exec_depend>std_msgs</exec_depend>
    <exec_depend>turtlesim</exec_depend>

    <export>

    </export>
</package>
```

   (a) The `name`, `version`, `description`, `maintainer` and `license` tags should be obvious.

   (b) The `buildtool_depend` tag specifies those tools that the package needs in order to build itself. Typically, this is only `catkin`.

   (c) The `build_depend` tag specifies the packages needed to build this package.

   (d) The `exec_depend` tag specifies the packages needed to run code in this package.

   (e) The `build_export_depend` tag specifies the packages needed for this package to work in a header file for another package. Don't worry if you don't understand this – it is not relevant for this short course.

   (f) The `export` tag is used for metapackages, which are outside the scope of our discussion. Metapackages are packages whose only function is to combine multiple packages which typically work together.

6. For more information, see `http://wiki.ros.org/catkin/package.xml` and `http://wiki.ros.org/ROS/Tutorials/CreatingPackage`.

4 Our first ROSPy node

1. In `intro2rospy`, create a `scripts` folder. In this folder, create `hello_world.py`. The complete file:

```
#!/usr/bin/env python
import rospy
rospy.init_node('hello_world')
rospy.loginfo("Hello, world!")
```

   Let's examine it line by line.

(a) `#!/usr/bin/env python` tells the system that this executable should use the Python 2.7 interpreter.

(b) `rospy.init_node('hello_world')` creates a new node with the name `hello_world`.

(c) `rospy.loginfo("Hello, world!")` writes a log message, which is typically printed to `STDOUT`.For more information, see `http://wiki.ros.org/rospy/Overview/Logging`.

(d) One important point: You cannot define multiple nodes in the body of an executable. You can have multiple nodes from the same executable at the same time by calling the executable multiple times, but at a time, an executable can spawn only one node.

2. Let's edit `CMakeLists.txt` for `intro2rospy`. Note: This is the `CMakeLists.txt` inside the `intro2rospy` folder, not the one in the `intro2ros` folder. Removing the comments, the file is:

```
cmake_minimum_required(VERSION 3.0.2)
project(intro2rospy)

find_package(catkin REQUIRED COMPONENTS
    geometry_msgs
    rospy
    std_msgs
    turtlesim
)

catkin_package()

include_directories(
    include
    ${catkin_INCLUDE_DIRS}
)
```

3. Let's examine this line by line.

(a) `cmake_minimum_required` and `project` should be obvious.

(b) `find_package` is used to specify those CMake packages which we need to build this package.

(c) `catkin_package` specifies information specific to Catkin.

(d) `include_directories` specifies where the included header files can be found. This is relevant to ROSCpp executables we wish to build.

4. Add the following to the end of `CMakeLists.txt`:

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

`catkin_install_python` is used to specify the Python scripts we wish to install.

5. For more information, see `http://wiki.ros.org/catkin/CMakeLists.txt` and `http://docs.ros.org/api/catkin/html/howto/format2/installing_python.html`.

6. Now, run `catkin_make` from the workspace folder. Output:

```
Base path: <path>intro2ros
Source space: <path>intro2ros/src
Build space: <path>intro2ros/build
Devel space: <path>intro2ros/devel
Install space: <path>intro2ros/install
####
```

```
#### Running command: "cmake <path>intro2ros/src
-DCATKIN_DEVEL_PREFIX=<path>intro2ros/devel
-DCMAKE_INSTALL_PREFIX=<path>intro2ros/install
-G Unix Makefiles" in "<path>intro2ros/build"
####
-- Using CATKIN_DEVEL_PREFIX: <path>intro2ros/devel
-- Using CMAKE_PREFIX_PATH: <path>intro2ros/devel;/opt/ros/melodic
-- This workspace overlays: <path>intro2ros/devel;/opt/ros/melodic
-- Found PythonInterp: /usr/bin/python2 (found suitable version "2.7.17"
, minimum required is "2")
-- Using PYTHON_EXECUTABLE: /usr/bin/python2
-- Using Debian Python package layout
-- Using empy: /usr/bin/empy
-- Using CATKIN_ENABLE_TESTING: ON
-- Call enable_testing()
-- Using CATKIN_TEST_RESULTS_DIR: <path>intro2ros/build/test_results
-- Found gtest sources under '/usr/src/googletest': gtests will be built
-- Found gmock sources under '/usr/src/googletest': gmock will be built
-- Found PythonInterp: /usr/bin/python2 (found version "2.7.17")
-- Using Python nosetests: /usr/bin/nosetests-2.7
-- catkin 0.7.23
-- BUILD_SHARED_LIBS is on
-- BUILD_SHARED_LIBS is on
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- ~~  traversing 1 packages in topological order:
-- ~~  - intro2rospy
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- +++ processing catkin package: 'intro2rospy'
-- ==> add_subdirectory(intro2rospy)
-- Configuring done
-- Generating done
-- Build files have been written to: <path>intro2ros/build
####
#### Running command: "make -j8 -l8" in "<path>intro2ros/build"
####
```

Ideally, your output should look something like that above. You may see "failed" along with some error messages if you've made a mistake in one of the steps. Usually, the error message is descriptive enough to get a hint. If not, Google is your friend.

7. There is one last step – navigate to the `scripts` folder and run:

```
:...scripts$ chmod 777 hello_world.py
```

8. This gives permission to execute the script as a file.

9. Now, in two terminals, run:

```
:~$ roscore
:~$ rosrun intro2rospy hello_world.py
```

Output:

```
[INFO] [1592385468.996500]: Hello, world!
```

Congratulations! You have just written your first ROS node, although it doesn't do very much. Next, let's create a simple publisher node.

5 Publishing to Turtlesim

1. In `scripts`, create a new script `random_publisher.py`. The file:

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
import random
def random_publisher():
    rospy.init_node('random_publisher', anonymous=True)
    pub = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=1000)
    rate = rospy.Rate(2)

    while not rospy.is_shutdown():
        to_publish = Twist()
        to_publish.linear.x =random.uniform(0, 1)
        to_publish.angular.z = random.uniform(-1, 1)
        rospy.loginfo(str(to_publish))
        pub.publish(to_publish)
        rate.sleep()

if __name__ == "__main__":
    random_publisher()
```

Let's examine this line-by-line.

(a) `from geometry_msgs.msg import Twist` – We need to import the classes for the messages we wish to use.

(b) `import random` – for random number generation.

(c) `anonymous=True` – This ensures that even if we run the same executable multiple times in the same session, the nodes all have unique names. All nodes are uniquely identified by their full name, so this is important.

(d) `rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=1000)` – creates a publisher for the topic `/turtle1/cmd_vel`, publishing `geometry_msgs/Twist` messages, with a queue size of 1000. Basically, if the messages are generated faster than the publisher can publish them, then the publisher must store the messages in a queue. The size of this queue is specified here. If more messages are waiting than the queue size, the new messages which are causing the queue to overflow are dropped.

(e) `rate = rospy.Rate(2)` sets up a rate object of 2 Hertz. We'll see more about this in a moment.

(f) `rospy.is_shutdown()` checks whether the node has received a shutdown request.

(g) The next section,

```
to_publish = Twist()
to_publish.linear.x =random.uniform(0, 1)
to_publish.angular.z = random.uniform(-1, 1)
```

Creates a `geometry_msgs/Twist` message and sets the linear and angular parts accordingly. The linear part is given a random number between 0 and 1, and the angular between -1 and 1.

(h) `pub.publish(to_publish)` publishes the message.

(i) `rate.sleep()` is used to enforce the frequency at which we wish to publish the messages. It measures the time taken by the other parts of the loop and pauses execution such that the rate of message publishing matches the frequency we passed to the rate object earlier.

(j) `if __name__ == "__main__":` – if we are directly executing this file and not importing it elsewhere, run the function.

2. Make the following change to the `CMakeLists.txt`:

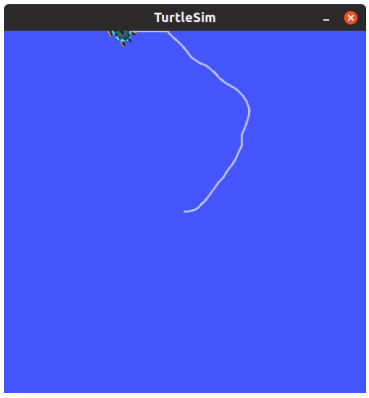
```
catkin_install_python(PROGRAMS
    scripts/hello_world.py
    scriptS/random_publisher.py
    DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

3. Don't forget to do the `chmod` step, and run `catkin_make`.

4. Now, in three terminals, run:

```
:~$ roscore
:~$ rosrun turtlesim turtlesim_node
:~$ rosrun intro2rospy random_publisher.py
```

5. You should see the turtle move with a random velocity. Like this:



Next, let's write a subscriber node.

6 Subscribing to turtlesim

1. The subscriber program, `pose_subscriber.py` is:

```
#!/usr/bin/env python
import rospy
from turtlesim.msg import Pose

def callback(msg):
    rospy.loginfo(str(rospy.get_name()) + " heard " + str(msg))

def pose_subscriber():
    rospy.init_node("pose_subscriber", anonymous=True)
    rospy.Subscriber("/turtle1/pose", Pose, callback, queue_size=1000)
    rospy.spin()

if __name__ == "__main__":
    pose_subscriber()
```

   Let's break this down line-by-line.

   (a) `rospy.Subscriber("/turtle1/pose", Pose, callback, queue_size=1000)` – this sets up a subscriber to `/turtle1/pose` with message type `turtlesim/Pose`, using `callback`, the function defined above, as the callback function. Whenever the subscriber receives a message on that topic, the callback function is executed, with the message being the argument. Lastly, we have the queue size. Incoming messages are placed in a queue, and messages can only be processed as fast as the execution of the callback function (among other things).

   (b) `rospy.spin()` hands over the execution of the program to ROS. The execution of the callback can only happen if `rospy.spin()` or `rospy.spinOnce()` is called. For more information, see `http://wiki.ros.org/roscpp/Overview/Callbacks%20and%20Spinning`.

   (c) `rospy.loginfo(str(rospy.get_name()) + " heard " + str(msg))` – `rospy.get_name()` gives us the full name of the node.

2. Make the necessary changes to CMakeLists and the permissions. Now, run:

```
:~$ roscore
:~$ rosrun turtlesim turtlesim_node
:~$ rosrun turtlesim turtle_teleop_key
:~$ rosrun intro2rospy pose_subscriber.py
```

   Control the turtle and see the change in the pose. Output:

```
...
[INFO] [<time>]: /pose_subscriber_11334_1592393854263 heard
x: 7.41065740585
y: 8.16551113129
theta: 0.639999985695
linear_velocity: 0.0
angular_velocity: 0.0
...
```

3. For more information, see `http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29`.

7 Looking ahead

You have written subscriber and publisher nodes, key features of ROS. With this knowledge, you should be able to make many non-trivial programs using ROSPy, though you may not yet be able to control robots using ROS. Next time, we will see how to write subscribers and publishers using ROSCpp.