ROS Made Easy
2: Roscpp

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 I expect them to become rapidly out of date.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 Your first ROSCpp node

1. Create a package `intro2roscpp` in the `src` folder of the `intro2ros` workspace. To do this, run:

```
.../intro2ros/src$ catkin_create_pkg intro2roscpp roscpp std_msgs
```

2. In `intro2roscpp/src`, create `hello_world.cpp`. The file:

```cpp
#include <ros/ros.h>
int main(int argc, char** argv)
{
    ros::init(argc, argv, "hello_world");
    ros::NodeHandle nh;
    ROS_INFO_STREAM("Hello, world!");
    return 0;
}
```

   Line by line:

   (a) `#include <ros/ros.h>` is to include the header file for ROSCpp. VS Code may yell when you do this – the solution is to add `/opt/ros/melodic/include` to your C++ include path. VS Code should tell you how to do this.

   (b) `int main(int argc, char** argv)` is so that you can handle command-line arguments. This is necessary only when you wish to rename your node or change some initial parameters – but is present as a default for the next step.

   (c) `ros::init(argc, argv, "hello_world");` is to initialize the node, with the default name `hello_world` and to pass the command-line arguments so that any changes at runtime can be implemented. **You must do this before starting or using any node**.

(d) `ros::NodeHandle nh;` is to start the node. There is a subtle difference between initializing and starting the node. The ROS documentation says, "Initializing the node simply reads the command line arguments and environment to figure out things like the node name, namespace and remappings. It does not contact the master.".

(e) As with ROSPy, we can only start handle one node per executable file at a time. We can create multiple copies of the same node by running the executable multiple times, but we can't have the same instance of the executable create two nodes.

(f) The node is started when the first `NodeHandle` is created and shut down when the last `NodeHandle` is destroyed. For more information, see `http://wiki.ros.org/roscpp/Overview/Initialization%20and%20Shutdown`

(g) `ROS_INFO_STREAM("Hello, world!");` prints to the log level `info` for that node.

3. Let's modify `CMakeLists.txt`. The file should look like:

```
cmake_minimum_required(VERSION 3.0.2)
project(intro2roscpp)

find_package(catkin REQUIRED COMPONENTS
    roscpp
    std_msgs
)

catkin_package()

include_directories(
    include
    ${catkin_INCLUDE_DIRS}
)
```

4. Add the following lines to `CMakeLists.txt`:

```
add_executable(hello_world src/hello_world.cpp)
target_link_libraries(hello_world ${catkin_LIBRARIES})
```

The first line specifies that we wish to build `hello_world` from `src/hello_world.cpp`. The second line says that we wish to link the libraries of catkin to the executable we are building.

5. 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:
     "make cmake_check_build_system" 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 2 packages in topological order:
-- ~~  - intro2roscpp
-- ~~  - intro2rospy
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- +++ processing catkin package: 'intro2roscpp'
-- ==> add_subdirectory(intro2roscpp)
-- +++ 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"
####
Scanning dependencies of target hello_world
[ 50%] Building CXX object
    intro2roscpp/CMakeFiles/hello_world.dir/src/hello_world.cpp.o
[100%] Linking CXX executable
    <path>/intro2ros/devel/lib/intro2roscpp/hello_world
[100%] Built target hello_world
```

You may see a failure if the libraries are not linked properly, or if there is an error in the C++ file.

6. In two terminals, run:

```
:~$ roscore
:~$ rosrun intro2roscpp hello_world
```

Output:

```
[ INFO] [1592456814.700655274]: Hello, world!
```

Next, Let's make a simple publisher.

3 Making a publisher

1. Make `chatter_pub.cpp` in `intro2roscpp/src`. The file:

```cpp
#include <ros/ros.h>
#include <std_msgs/String.h>
#include <iostream>

int main(int argc, char** argv)
{
    ros::init(argc, argv, "chatter_pub");
    ros::NodeHandle nh;

    ros::Publisher pub = nh.advertise<std_msgs::String>("chatter", 1000);
    ros::Rate rate(2);

    while(ros::ok())
    {
        std::string str;
        getline(std::cin,str);

        std::string nodeName = ros::this_node::getName();

        std_msgs::String msg;
        msg.data = nodeName + ": " + str;

        pub.publish(msg);
        rate.sleep();
    }
    return 0;
}
```

   Line by line:

   (a) `#include <std_msgs/String.h>` – Like in Python, we need to import the messages we wish to use.

   (b) `nh.advertise<std_msgs::String>("chatter", 1000)` gives us a publisher of `std_msgs/String` messages to the topic `chatter` with a queue size of `1000`.

   (c) `ros::Rate rate(2)` gives us a 2 Hertz rate object.

   (d) `ros::ok()` checks whether the node has shut down.

   (e) `ros::this_node::getName()` gives the full name of this node.

   (f) The next two lines,

   ```
   std_msgs::String msg;
   msg.data = nodeName + ": " + str;
   ```

   Create a message of type `std_msgs/String` and assign it accordingly.

   (g) `pub.publish(msg)` publishes the message.

   (h) `rate.sleep()` ensures that the publishing rate never crosses 2 Hertz.

2. Add the following lines to `CMakeLists.txt`:

```
add_executable(chatter_pub src/chatter_pub.cpp)
target_link_libraries(chatter_pub ${catkin_LIBRARIES})
```

3. Run `catkin_make` in the workspace folder.

4. Now, in three terminals, run:

```
:~$ roscore
:~$ rosrun intro2roscpp chatter_pub
:~$ rostopic echo chatter
```

In the second terminal, any messages you enter through `STDIN` will be published to the topic `chatter`. We are monitoring this topic in the third terminal. Output at the third terminal is like this:

```
data: "/chatter_pub: Welcome to Enigma"
---
data: "/chatter_pub: The airspeed of an unladen swallow"
---
data: "/chatter_pub: An African or European swallow?"
---
```

Let us now make a subscriber object.

4 Making a subscriber

1. Make a file `chatter_sub.cpp`. The file:

```
#include <ros/ros.h>
#include <std_msgs/String.h>
#include <iostream>

void callback(const std_msgs::String::ConstPtr &msg)
{
    std::cout << (msg->data) << std::endl;
}

int main(int argc, char** argv)
{
    ros::init(argc, argv, "chatter_sub");
    ros::NodeHandle nh;

    ros::Subscriber sub = nh.subscribe("chatter", 1000, &callback);
    ros::spin();
    return 0;
}
```

Line by line:

   (a) `nh.subscribe("chatter", 1000, &callback)` creates a subscriber to the topic `chatter` with a queue size of `1000` which runs `callback` every time it receives a message.

   (b) `ros::spin()` hands over control of execution to ROS so that the callback function can be executed.

2. Make the changes in `CMakeLists.txt`, build the package, and run the publisher and subscriber nodes. The text entered at the publisher terminal should show up at the subscriber terminal like this:

```
/chatter_pub: Hello, there!
/chatter_pub: General Kenobi, you are a bold one!
```

5 Looking ahead

You can now build publisher and subscriber nodes in ROSCpp as well as ROSPy. Next time, we will look at using services and the parameter server.