Robot Operating System – Quick-Start Guide

The Ohio State University

The Underwater Robotics Team

Date Revised: 2-4-19

Table of Contents

ABOUT THIS DOCUMENT **3**

INTRODUCTION **4**

WHAT IS ROS? 4

HIGH-LEVEL STRUCTURE 4

CATKIN, WORKSPACES, AND ROS PACKAGES **5**

WORKSPACE STRUCTURE 5

COMPILING CODE 5

SAMPLE CMAKELISTS.TXT 6

SAMPLE PACKAGE.XML 7

CREATING A CATKIN WORKSPACE 7

CREATING A CATKIN PACKAGE 8

EXERCISE 1 8

THE ~/.BASHRC FILE 8

EXERCISE 2 9

ROS: BEGINNER CONCEPTS **10**

HOW TO WRITE A SIMPLE PUBLISHER/SUBSCRIBER NODE IN C++ 10

The Header File *11*

The C++ File *12*

Building your Node *14*

The Launch File *15*

Launching a Node *15*

REFERENCES **17**

About This Document

This document is mean to be a quick-start guide for beginners getting started with Robot Operating System (ROS). This guide is by no means a replacement for the actual ROS wiki tutorials – this guide is solely meant to give beginners an introduction to the main concepts and implementation of ROS to help them follow the wiki tutorials with greater comfort. Not all concepts / features will be covered in detail, as some features may only be introduced on a high-level to inform the reader of ROS' capabilities. It is up to the reader to read through the ROS wiki tutorials for more in-depth explanations on what is covered in this quick-start guide.

By the end of this tutorial, you will understand how to write basic C++ code interfaced with ROS and be capable of using some of the key features ROS has to offer. While this tutorial contains many blocks of sample code which are easy to copy and paste into your code editor, the formatting may get messed up and it may not compile. Since the code is not too hard to write, it is recommended you write it yourself instead of using copy and paste. This way you get real experience writing your own code. It is always more valuable to do something yourself than to have someone else do it for you.

Introduction

What is ROS?

From the ROS wiki: "ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers" [1].

It is also worth mentioning that ROS is designed for parallel processing – this a major advantage for controlling robots. Robots typically have many sensors, data processing algorithms, and control algorithms that need to run simultaneously and as fast as possible. ROS' framework allows for such capability as information is quickly passed through from one node to the next.

High-Level Structure

On a high-level, ROS is composed of a server, in which the master handles all communications across nodes (the individual software components that make up the system) and keeps the system running until it's terminated. Nodes can pass data to each other on data streams called topics. Topics can only contain a *single* type of data packet, referred to as a message. Messages can be thought of as structs and can contain as many basic data fields as desired.

In general, sensors should have their own node so new data can be reported as soon as it arrives. There might be multiple nodes to handle data processing (one node per sensor). There might be several controller nodes in which each node is just a part of the larger control system (ex. PID). **Figure 1**, below, is a representation of how the ROS node structure will look like on a graph.

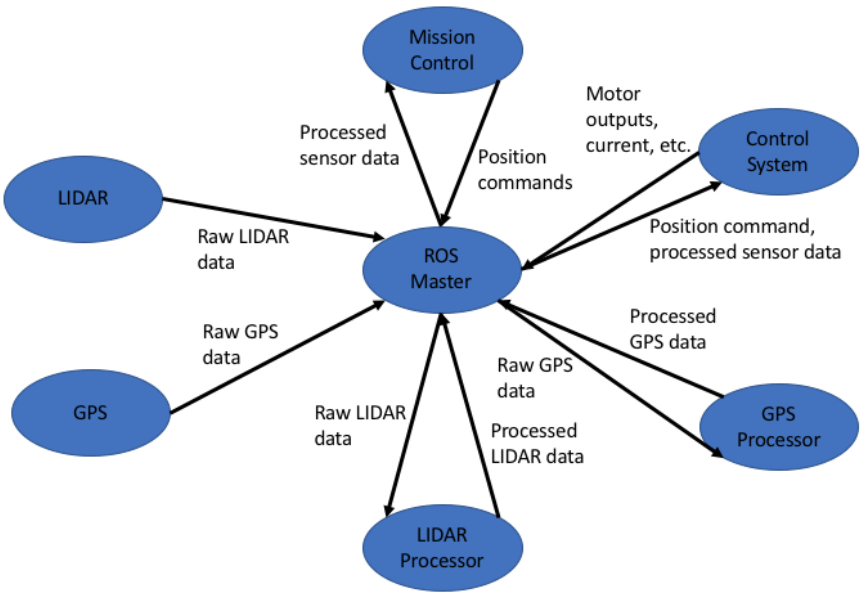

Figure 1: Sample ROS Node Graph

Catkin, Workspaces, and ROS Packages

ROS uses catkin (a cmake build system), which is a build system that allows for code to be compiled or installed in packages (groups of code). All ROS packages for a given project will be contained within a catkin workspace, which is the highest level where all the code resides. Each package should (generally) contain similarly purposed code. For instance, your system may have a hardware package to handle all-things sensor data, a controller package to handle all forms of control (PID, LQR, etc.), a vision package for computer vision / machine learning, etc.

Please read the ROS wiki tutorials regarding catkin for more information.

Workspace Structure

Each project should have its own workspace. The workspace will have a source folder, called "src" for all of the project code. Inside this "src" folder will be all of the ROS packages you create. Inside each ROS package will be another "src" folder for the package's source code. In general, you will have additional folders within each package. Here are some common folders and how they are typically used:

- src: contains .c and .cpp files
- include/<package_name>: contains header files for the .c and .cpp files
- scripts: contains python scripts
- launch: contains all launch files to run the package's nodes
- cfg: contains configuration files with parameters your nodes may require for execution
- msg: contains custom ROS messages

Compiling Code

Catkin has three steps when it comes to compiling. The first is a CMakeLists.txt file. This file indicates to the compiler all source files to compile within the package, and any dependencies each source file has. The main workspace also has a single CMakeLists.txt file, although this file is generally left untouched because all the work is taken care of in each package's CMakeLists.txt file.

The second file is the package.xml. This file indicates all dependencies that the package requires for building and running. This file usually isn't used much during build-time, but it is more readable by others to see what dependencies the given package requires.

Finally, if the above two files are up-to-date, then to compile your workspace's code enter the main directory of your workspace and type `catkin_make`.

Please see the following pages for samples of how these two files should look.

Sample CMakeLists.txt

When you first create a package, the auto-generated CmakeLists.txt file will contain all possible features with its own comments. At the minimum, please read the comments within that auto-generated file pertaining to the commands briefly described below (find_package, catkin_package, etc.). Please read the ROS wiki for further details.

```
cmake_minimum_required(VERSION 2.8.3)
project(beginner_nodes)
add_compile_options(-std=c+11)

# find_package() finds other catkin packages that are needed for building
# this package.
find_package(catkin REQUIRED
      COMPONENTS
      roscpp
      std_msgs
      geometry_msgs
)

# catkin_package() is responsible for ROS-specific configuration.
# The include directory is the most common part used. The line below
# indicates that exported header files go inside an include folder
catkin_package(INCLUDE_DIRS include)

# This part sets up any include paths for header files or library paths.
# The path for the package's include folder is needed.
# We also need to include all of the library paths for all packages
# we declared above in the find_package() command.
include_directories(include ${catkin_INCLUDE_DIRS})

# NOTE: Including some packages, such as OpenCV, require a very specific
# ordering. BEFORE the include_directories() line, add this line:
# find_package(OpenCV REQUIRED)
# Then, within the include_directories() command, add the following:
# ${OpenCV_INCLUDE_DIRS}

# Below are the three lines required for compiling a ROS node
# Replace "executable_name" with the name of your choice. Common
# convention is the same name as the node file.
add_executable(executable_name src/my_node.cpp)
target_link_libraries(executable_name ${catkin_LIBRARIES})
add_dependnecies(executable_name ${catkin_EXPORTED_TARGETS})
```

Sample package.xml

When you first create a new catkin package, it also auto-generates this file for you. Below is a *brief* outline of the more important information. Further details about the package.xml file can be found on the ROS wiki.

```
<?xml version="1.0"?>
<package format="2">
      <!-- NOTE: This is a comment in xml -->
      <!-- Stuff about name, maintainer email, license, blah blah -->

      <!-- This line is REQUIRED, since it specifies the build tool -->
      <buildtool_depend>catkin</buildtool_depend>

      <!-- Remember the find_package() line from CMakeLists.txt? -->
      <!-- Generally, all of thoese packages should be inside -->
      <!-- a build_depend line like below -->
      <build_depend>roscpp</build_depend>

      <!-- These are the packages OTHER catkin packages need if they -->
      <!-- include this package -->
      <build_export_depend>roscpp</build_export_depend>

      <!-- This is where you list any dependencies needed at run-time -->
      <exec_depend>roscpp</exec_depend>

</package>
```

Creating a Catkin Workspace

Follow the below commands in a terminal to create a catkin_workspace.

user@hostname$ `mkdir -p ~/<workspace_name>/src`
user@hostname$ `cd ~/<workspace_name>/src`
user@hostname:~<workspace_name>/src$ `catkin_init_workspace`
user@hostname:~<workspace_name>/src$ `cd ..`
user@hostname:~<workspace_name>$ `catkin_make`

Upon running the command `catkin_make`, two additional directories within our main workspace are will be created: "build" and "devel." These directories are always regenerated upon running `catkin_make` and contain crucial information used by other catkin packages and even the terminal.

Creating a Catkin Package

It is quite simple to create a new catkin package. Once you have your workspace created, all you need to know is the package name and some of the first-order dependencies that your package will require (replace <dep1>, <dep2> with roscpp, rospy, etc.). Note: some commands continue on the next line.

```
user@hostname$ cd ~/<workspace_name>/src
user@hostname:~<workspace_name>/src$ catkin_create_pkg <package_name>
<dep1> <dep2> …
```

Exercise 1

Follow the above commands and create a catkin_workspace and replace "<workspace_name>" with "ros_tutorials". Then, enter your workspace (go inside the "src" folder). Now create a package called "beginner_nodes" with the following dependencies in the command line: roscpp, std_msgs, and geometry_msgs.

Enter your new package and see what it contains. You will always have an auto-generated package.xml and CMakeLists.txt file, but you will have a *src* folder and an "include/beginner_turorials" folder. These "src" and "include" directories are always created when "roscpp" is declared as a first-order dependency. It is recommended that you try to read through some of the comments within both the CMakeLists.txt and package.xml files so you have a better understanding of what goes on inside them. It may not make a lot of sense right now, but hopefully it will make more sense as you start writing your own code.

Finally, inside the "beginner_nodes" package you just created, make a new sub-directory called "launch." You should now have three folders inside the package: src, include, and launch.

The ~/.bashrc File

The .bashrc file is a hidden file (indicated by the "." In front) and resides in your computer's home directory (hence the "~"). Bashrc stands for "Born Again Shell Run Commands." A terminal is also called a "shell," and each time you open a terminal this file is executed. There are ROS-specific terminal commands that we will use, but the terminal needs to know what these commands do and how to handle the arguments and pass data around. To view this file, simply type in a terminal:

```
$ nano ~/.bashrc
```

NOTE: nano is the built-in code editor for a terminal. You can only use keyboard keys and scrolling with a mouse – mouse clicks do not work here. To exit from the editor, press CTRL-X, type "y" or "n" to save/not save changes, then press ENTER.

To allow the terminal to run these commands, we "source" a few setup.bash files. Sourcing essentially means we are setting up environment variables the terminal needs – these environment variables, for the most part, are paths to variables or directories containing crucial information regarding carrying out these commands. When we initially install ROS, one line should be automatically added to the bottom of the .bashrc file:

```
source /opt/ros/kinetic/setup.bash
```

This line allows the terminal to invoke ROS-specific commands. But some ROS commands also utilize our source code, as you will later on see. So how do we indicate to the terminal how to use our source code? Remember those "build" and "devel" folders that were generated from compiling our code? The "devel" folder contains a setup.bash file. All we need to do is add the following line below the ROS sourcing line:

```
source ~/<workspace_name>/devel/setup.bash
```

Exercise 2

Enter your computer's ~/.bashrc file and verify the ROS sourcing line exists near the bottom of the file – it need not be the *last* line, but it will likely be towards the end of the file. If the line does not exist, then please add it. Then add a line somewhere beneath it to source the code we will soon create in our beginner_nodes package. Remember to replace "<workspace_name>" with "ros_tutorials." The bottom of your .bashrc file should look similar (may not be an exact match) to this:

```
source /opt/ros/kinetic/setup.bash
source ~/ros_tutorials/devel/setup.bash
```

REMINDER: If "source /opt/ros/kinetic/setup.bash" already exists, DO NOT add it again.

Beginner Concepts

As discussed previously, the building block of a ROS framework is the node. Nodes are individual components meant to execute a process over and over again (ex. receive and transmit raw sensor data, process sensor data, perform computations for a control stack, etc.). Nodes are compiled independently from one another, but may rely on messages relayed from other nodes for it to execute its code (nodes can *subscribe* to topics).

Topics, or data streams, are the means by which nodes pass data to each other (nodes can *advertise or publish* messages on topics). Each topic can only relay data regarding a single data packet called a message. There are pre-existing packages for standard messages, such as std_msgs and geometry_msgs. You can also create your own custom message consisting of a series of existing messages.

How to Write a Simple Publisher/Subscriber Node in C++

The ROS tutorial that covers how to write a simple publisher/subscriber in C++ places all the code inside the file's main() function. Generally, a node should be abstracted as a class for better organization and functionality. Let's see what is involved in writing a simple publisher/subscriber node.

The Header File

Inside your "beginner_nodes" package, create a file called "test_pub_sub.h" within the "include/beginner_nodes" folder. Write the following lines of code inside:

```
#ifndef TEST_PUB_SUB_H // Standard C syntax for beginning a header file
#define TEST_PUB_SUB_H

#include "ros/ros.h" // Include the ROS header files
#include "std_msgs/Int8.h" // Include the header files for a Int8 message

Class TestPubSub
{
private:
      ros::NodeHandle nh; // Create a nodehandle object
      ros::Subscriber sub; // Create a ROS Subscriber object
      ros::Publisher pub; // Create a ROS Publisher object
public:
      TestPubSub(); // Declare class constructor
      void TestCB(const std_msgs::Int8::ConstPtr& msg); // Declare callback method
      void Loop(); // Declare a looping function
};

#endif // Standard C syntax for ending a header file
```

In the header file above, the most important thing to understand are what the private variables and public methods mean. For the private variables, the nodehandle is the actual object that the node uses to communicate to the server. The subscriber object is what listens for incoming messages on a user-defined topic. The publisher object is how a node can publish messages on a topic. Regarding the public methods, every class needs a constructor to initialize it. The callback function (TestCB) is the function that is executed every time a new message comes in that the subscriber listens to. And the Loop() function is what keeps the node alive until it is shutdown. You will see all this in action shortly.

The C++ File

Now, create a test_pub_sub.cpp file in the "src" folder. Write the following lines of code inside (you need not copy the comments, they are only there to explain what the code does):

```
// Include the contents from the header file we just made
#include "beginner_nodes/test_pub_sub.h"

// Define the main() function
int main(int argc, char** argv[])
{
      // Initialize a node called "test_pub_sub
      ros::init(argc, argv, "test_pub_sub";

      // Create a TestPubSub object and invoke its Loop() function
      TestPubSub tps;
      tps.Loop(); // Replace with ros::spin() if you don't need specific rate
}

// Define the constructor
TestPubSub::TestPubSub() : nh("test_pub_sub")
{
      // Initialize the subscriber and publisher.
      // REQUIRED: Must specify the type of message inside <> brackets
      // Subscriber has four parameters:
      //    1. Topic name
      //    2. Queue size. Just set to "1" for now.
      //    3. Address to the callback method
      //    4. Address to the current object linking the subscriber
      // Publisher has two parameters (same as the first two for a subscriber)
      sub = nh.subscribe<std_msgs::Int8>("test, 1, &TestPubSub::TestCB, this);
      pub = nh.advertise<std_msgs::Int8>("test", 1);
}

// Define the TestCB() callback method
Void TestPubSub::TestCB(const std_msgs::Int8:ConstPtr& msg)
{
      // Print a statement to the console/terminal
      // NOTE: because "msg" is a pointer to a struct, we must use the pointer
      // operator "->" to access its contents
      ROS_INFO("My sensor data: %i", msg->data);
}

// Define the Loop() method
Void TestPubSub::Loop()
{
      ros::Rate rate(1); // Set the rate at which this node executes in [Hz]
      while(!ros::ok()) // While the node is still okay (has not terminated)
      {
            std_msgs::Int8 sensor_data; // Create an Int8 message
            sensor_data.data = 6; // Populate its only data field
            pub.publish(sensor_data); // Publish the message on a topic

            ros::spinOnce(); // Contact the server for any new information
            rate.sleep(); // Sleep for remaining time so node runs at 1 Hz
      }
}
```

The comments within the above file should explain a fair amount of what a majority of the code means. But there are a few main points to address.

The `ros::spinOnce()` function is what keeps the node in contact with the server. The while-loop always executes as long as the node has not been terminated (ex. CTRL-C). However, the only way the node can publish new messages or be informed of any new messages coming in is by invoking the `ros::spin()` or `ros::spinOnce()` function. When a node "spins," it will *retrieve* any new information from the server (subsribe to topics) and/or *transmit* any new data on the server (advertise, or publish on topics). The `ros::spinOnce()` function should only be used inside the while loop because it tells the node to "spin" a single time. Using the `ros::spin()` function simply replaces the entire while loop itself if you do not need to do anything inside it or want the node to "spin" at a specified rate (see below for more details).

The `ros::Rate` feature allows you to force a node to "spin" at a specified rate. For example, if you want data to be output at a set frequency, such as 100 Hz, you would setup a `Loop()` function similar to the one above and use a rate of 100. When you use the `ros::Rate` feature, you MUST create the while loop and place BOTH the `ros::spinOnce()` and `rate.sleep()` commands inside – the `rate.sleep()` call is added because this forces the node to "sleep" for the remaining time so it executes each cycle at the given rate. If you do not need the node to run at a certain frequency, then simply replace the `Loop()` method call with `ros::spin()` as noted in the comment.

The callback method (commonly abbreviated as "CB" within the function name) is the method the node executes upon receiving an incoming message from a specific topics. Upon "spinning," if the server sees that a node is subscribed to a topic in which a new message has just arrived, it will pass the message, as a pointer variable to a struct-like object, to the node's appropriate callback method to process the data. The only code involved in setting up the subscriber object is initializing it in the constructor (explained in the code comments) and creating the callback method (in which the user controls the data processing).

Last, when you want to publish a message on a topic, all that is required is to initialize ALL fields within a type of message (could be from a standard ROS message package or from a custom message), and call the `publish()` function for the appropriate publisher. Publishing a null message will immediately terminate the node upon startup.

As a reminder, a publisher and subscriber can only publish or subscribe to a *single* topic, which in turn carries a *single* message type. The topic name and message type MUST be specified when initializing publisher and subscriber objects. Additional publishers and subscribers will have to be added if you want to increase the data being passed to and from the node.

NOTE: In the sample code above, we publish a static message in the while-loop. Normally you would not place much inside the while-loop since most of the work takes place inside a callback. For the purposes of this example, we publish message inside the while-loop so that we have "sensor" data to work with.

Building your Node

Now that you written the C++ and header files, we need to tell the compiler to build our node as an executable. To do this, we simply add three lines to the CMakeLists.txt file within our package. Because we have not touched this file yet, we will start by cleaning it up so we only have what we need. Please change your CMakeList.txt file to look like the one below. If you wish to keep everything else in the file for a future reference, this is up to you. Comments have been written for further clarification about what is going on under the hood.

```
# The first two lines below are REQUIRED and should always exist in this file
cmake_minimum_required(VERSION 2.8.3)
project(beginner_nodes)
# add_compile_options(-std=c++11)

# This command simply finds all packages that anything in here depends on
find_package(catkin REQUIRED COMPONENTS
      roscpp
      geometry_msgs
      std_msgs
)

# This is a catkin-specific command. Here we indicate the include folder is where
# exported headers go so that other catkin packages can use them.
# For example, this is what allowed us to include the "std_msgs/Int8.h" header
# file in our code.
# NOTE: Other parameters can be passed into this command. For the purposes of this
# exercise, we will only be using it for the above reason.
catkin_package(INCLUDE_DIRS include)

# This is a package-specific command used for locating any paths for all header
# files and other directories required for all code being built in this package.
# We currently only have two paths to pass in:
#     1. The include folder within our package
#     2. Any header files generated by catkin using the catkin_package() command
include_directories(include ${catkin_INCLUDE_DIRS})

# The following three commands are required for building a node with catkin.
# The first parameter in each commands is the name of the executable

# Create an executable and link to the appropriate .cpp file
add_executable(test_pub_sub src/test_pub_sub.cpp)

# Link any libraries required for building the code. All header files or code from
# any dependencies from the find_package() commands are lumped into a single
# parameter called: catkin_LIBRARIES
target_link_libraries(test_pub_sub ${catkin_LIBRARIES})

# Add any dependencies needed by the node such as messages, services, and actions
# All of these dependencies are lumped into a single parameter called:
# catkin_EXPORTED_TARGETS
add_dependencies(test_pub_sub ${catkin_EXPORTED_TARGETS})
```

Now you just need to run the `catkin_make` command at the top-level of your workspace.

The Launch File

One of the final steps in being able to run our node is to create a launch file. A launch file is an XML file that indicates a group of nodes we want to run at the same time. A launch file can be used to run as many nodes as we want. We can also pass in arguments and parameters to the nodes from the launch file – more on this later. For now, create a launch file called test_pub_sub.launch and place it inside the beginner_nodes' lauch folder. Place the following code inside.

```
<launch>
      <node pkg=”beginner_nodes” type=”test_pub_sub” name=”test_pub_sub”
            output=”screen” />
</launch
```

In your launch file, the "node" group is supposed to be on a single line – the above code simply did not fit on one line. This node group takes in a series of parameters, but we generally just use:

1. The package name
2. The type of executable (will need the executable name we made in the CMakeLists file)
3. The name of the node (the name from the ros::init() function call in the .cpp file)
4. Specify where output from the node goes. The ROS_INFO commands are basically print statements with a timestamp. But we have to specify that we want to see this output in the terminal, otherwise it will not appear.

This is the most plain and simple a launch file can get. Just be careful with syntax in these files.

Launching a Node

Nodes can be run via two possible commands from a terminal: `rosrun` or `roslaunch`.

`The rosrun` command only allows you to run a single node. The command structure is:

```
$ rosrun <package_name> <executable_name>
```

However, in practice running a single node at a time is impractical. This is why the concept of a launch file exists (which you just learned the basics of). The `roslaunch` command is used to run nodes as defined in a launch file and has a similar command structure:

```
$ roslaunch <package_name> <launch_file_name>
```

To execute our code, you will type: `roslaunch beginner_nodes test_pub_sub.launch`

Hopefully, your code built successfully from the previous step, and you were able to launch your node. If it works, you should see a new line printed to the terminal every second. Just press CTRL-C to terminate the node.

Exercise 3

Revise the sample code to make use of the following message type: a **Vector3** message from the **geometry_msgs** package. Look up the data fields contained inside the message to see how it is setup. Revise the code to do the same job as before, but with a different message:

- Publish a Vector3 message inside the while-loop (give each data field a different value)
- Inside the callback method, print all data fields to the terminal

References

[1] http://wiki.ros.org/ROS/Introduction