10 Feb 2014 by Jonathan Bohren <me at jbohren.com>

ROS C++ Hello World (The Simplest ROS Tutorial)

ros tutorial c++

Introduction

There are a lot of ROS tutorials out there. This is the simplest one.

This tutorial demonstrates how to build a ROS "Hello World" executable written in C++ without getting into the details of ROS packages, workspaces, launchfiles, or other best practices. This tutorial is meant to demonstrate the **bare minimum** of what is required to interact with a ROS system. It is **not** meant to be an example of good ROS development practices, but rather is meant to be useful for someone new to ROS and software development in general.

Unlike most tutorials, this tutorial intends to teach you about ROS instead of just getting you to run some pre-defined programs. As such, very little will be explained preemptively to avoid errors. This is because you will always encounter errors. Not just when using ROS, but with any system. Instead, this tutorial welcomes errors because they present opportunities to explain how to *recover* from errors.

> ***NOTE:*** *This tutorial was written for the ROS Hydro Distribution. Assuming the commands are still accurate, if you wish to follow this tutorial with a different distribution of ROS, any time* `hydro` *is mentioned, simply replace it with the shortname for that distribution.*

Pre-Requisites

- A computer running a recent Ubuntu Linix [1] LTS (long-term support) installation
- Minimal experience with the Linux and the command-line interface
- Minimal experience with C++

Tools Used

- Ubuntu Linux [1]
- The bash shell [2]
- C++ [3]
- The GNU Compiler Collection (GCC) [4]
- Any plain-text editor (I like vim [5]).

ROS Packages Used

- roscpp
- rosconsole

Number of Windows Needed

- Browser for these instructions
- Window for your text editor
- Terminal to run `roscore`
- Terminal to run `hello_world_node`
- Terminal to run introspection commands like `rosnode` and `rostopic`

Contents

Installing ROS (if it hasn't already been installed)

For Ubuntu Linux, you can follow the following instructions, for other Linux platforms, see the main ROS installation instructions. As of the writing of this tutorial, ROS packages are only built with the Debian package management system [6]. This makes it easy to install on *debian-based* Linux distributions like Ubuntu.

Add the ROS Binary Package Repository

First, add the binary package repository hosted on ros.org to your sysmtem. This will allow you to locate pre-compiled ROS packages, and only needs to be done once, but is idempotent:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -cs) main" > /etc/apt/so
```

Next, get the ros.org PGP public key. This also only needs to be done once and is also idempotent.This will let you verify that your ROS packages are actually coming from ros.org and not some malicious middle-man. This is done automatically whenever you install a package from ros.org.

```
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

Install the Base ROS Packages

First, update the binary package index. This should be done whenever you want to make sure your system knows about the latest versions of binary packages available:

```
sudo apt-get update
```

Finally, install the base ROS packages from the ROS “Hydromedusa” distribution:

```
sudo apt-get install ros-hydro-ros-base
```

There are lots of other ROS packages available to install, but for this tutorial you only need a few of the “core” packages. To see the list of currently available binary packags, their versions, and build status, you can see the ROS debian package build status page.

Create THE SIMPLEST ROS (C++) PROGRAM

After you've installed the ***pre-requisites*** listed above, you can create a new directory for this tutorial. You can call this directory anything you want, but in this case, we'll call it `hello_world_tutorial` and make it anywhere. Then, enter the directory:

```
mkdir hello_world_tutorial
cd hello_world_tutorial
```

The first step is writing the simplest C++ program that can interact with ROS in a meaningful way. All it does is announce itself to the ROS Master as a ROS node called `hello_world_node` , then broadcast a Hello-world message over the standard `/rosout` topic, and then wait for a `SIGINT` or `ctrl-c` .

This program will be built from single file named `hello_world_node.cpp` with the following contents:

`hello_world_node.cpp`

```
// Include the ROS C++ APIs
#include <ros/ros.h>

// Standard C++ entry point
int main(int argc, char** argv) {
  // Announce this program to the ROS master as a "node" called "hello_world_node"
  ros::init(argc, argv, "hello_world_node");
  // Start the node resource managers (communication, time, etc)
  ros::start();
  // Broadcast a simple log message
  ROS_INFO_STREAM("Hello, world!");
  // Process ROS callbacks until receiving a SIGINT (ctrl-c)
  ros::spin();
  // Stop the node's resources
  ros::shutdown();
  // Exit tranquilly
  return 0;
}
```

Compile the simplest ROS (C++) program

Since we're building a program with C++, we need to compile it into an executable that we can actually run. In this *very* simple case, this can be done by invoking `g++` directly with the following build command (make sure to get all of it):

```
g++ hello_world_node.cpp -o hello_world_node -I/opt/ros/hydro/include -L/opt/ros/hydro/lib -Wl,
```

If you are unfamilar with command-line usage of g++, the arguments passed to `g++` have the following meanings:

- **`hello_world_node.cpp`** The source file(s) to compile
- **`-o hello_world_node`** The name of the *output* file (the executable, in this case)
- **`-I/opt/ros/hydro/include`** An instruction to look for C++ header files in `/opt/ros/hydro/include`
- **`-L/opt/ros/hydro/lib`** An instruction to look for static libraries in `/opt/ros/hydro/lib`
- **`-Wl,-rpath,/opt/ros/hydro/lib`** An instruction to look for shared libraries in `/opt/ros/hydro/lib`
- **`-lroscpp`** Link against the library `libroscpp.so` (ROS C++ bindings)
- **`-lrosconsole`** Link against the library `librosconsole.so` (ROS distributed logging)
- **`-lrostime`** Link against the library `librostime.so` (ROS time measurement)

Run the simplest ROS (C++) program (and fail)

After successfully compiling the program, you can run it right away.

```
./hello_world_node
```

If you've been following this tutorial verbatim, then you'll get an error message of some sort. While you might only get one of these errors, you should read through all of them because it will help you recognize common problems in the future.

ERROR: ROS_MASTER_URI is not defined (Bad Environment)

```
[FATAL] [1392021564.231775029]: ROS_MASTER_URI is not defined in the
environment. Either type the following or (preferrably) add this to your
~/.bashrc file in order set up your local machine as a ROS master:

export ROS_MASTER_URI=http://localhost:11311
```

```
then, type 'roscore' in another shell to actually launch the master program.
```

This error means that you haven't *sourced* one of the ROS *setup files*, but this is a common mistake among novices. Specifically, you're missing the `$ROS_MASTER_URI` environment variable, which is set by one of these setup files.

Running ROS programs requires them to agree on a few specific things. One of these things is on which machine and which port the ROS Master is running. The ROS master is what enables ROS nodes to find each-other and communicate. As such, any ROS nodes which are meant to talk to each-other need to announce themselves to the **same** ROS master.

As the helpful error message describes, declaring with which ROS master any ROS programs you run should talk is done by exporting the `ROS_MASTER_URI` environment variable. The suggestion given by the error message also happens to be the default if you properly source the ROS *setup files*.

Instead of doing what the error message tells you (why should you do what an error message says?), you can *source* one of the standard ROS setup files to add `ROS_MASTER_URI` and other important variables to your currently running shell:

```
source /opt/ros/hydro/setup.sh
```

If you want this to be available in each new shell you create, you should add the above line to the bottom of your shell's runcom file like `.bashrc` if you're using `bash`. The `.bashrc` file is located in your home directory and is executed each time you open a new shell.

At this point, you can try running `hello_world_node` again:

```
./hello_world_node
```

ERROR: Failed to contact master (Good Environment, but no ROS Master)

```
[ERROR] [1392014787.460431497]: [registerPublisher] Failed to contact master at [localhost:1131
```

This error is reported if you've properly set up your ROS environment variables but you haven't run roscore. The `roscore` serves a few functions including running the ROS Master which, as was explained above, is what enables ROS nodes to find each-other and communicate. In this case, `hello_world_node` is looking for a ROS Master on the local machine, `localhost`, and on the default port, `11311`, and no ROS Master is replying.

This is an easily solved problem. All that you need to do, is open a *new* shell **with the appropriate environment**, and run `roscore`.

```
roscore
```

ERROR: command not found: roscore (Fool Me Twice…)

If you open a new shell and get an error like `command not found: roscore`, this is because you haven't sourced the ROS *setup files* in the new shell, as is described in the previous section. In addition to setting `ROS_MASTER_URI` and other `ROS_*` environment variables, the ROS setup files also extend some standard UNIX environment variables such as `PATH` and `PYTHONPATH`.

Unlike most programs installed to your system (which are put in `/usr/bin` or similar), ROS binaries are installed to `/opt/ros/$ROS_DISTRO/bin`. This allows installation of more than one ROS *distribution* on a single machine without causing conflicts.

As you might imagine, `roscore` is normally installed to `/opt/ros/$ROS_DISTRO/bin/roscore`, and when you source `/opt/ros/hydro/setup.sh`, it adds `/opt/ros/hydro/bin` to your `$PATH` environment variable.

Once your `roscore` is running, proceed to the next step.

Run the simplest ROS (C++) program (and succeed)

In the original shell in which you were trying to run `hello_world_node`, try running it again, now that the `roscore` is running in the other window:

```
./hello_world_node
```

If everything goes well, you should see this hopeful message:

```
[ INFO] [1392020655.967763511]: Hello, World!
```

Leave the node running, and proceed to the next step.

Inspecting the simplest ROS (C++) program

In a new shell **with a proper environment**, you can now inspect your node running in the first shell with standard ROS command-line tools.

One of the simplest, `rosnode` , is a command-line program for listing and querying information about ROS nodes. For example, the `info` subcommand will give you all the metadata ROS knows about a given node. You can get the info for `hello_world_node` like so:

```
rosnode info /hello_world_node
```

This will give you information similar to the following:

```
Node [/hello_world_node]
Publications:
 * /rosout [rosgraph_msgs/Log]

Subscriptions: None

Services:
 * /hello_world_node/set_logger_level
 * /hello_world_node/get_loggers

contacting node http://localhost:55332/ ...
Pid: 22598
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound
    * transport: TCPROS
```

Note that if you `ctrl-c` `hello_world_node` in the other window, and try to re-run this command, you will see something similar to the following:

```
Node [/hello_world_node]
Publications: None

Subscriptions: None

Services: None

cannot contact [/hello_world_node]: unknown node
```

Distributed Logging with rosconsole

Simple as it may be, `hello_world_node` is actually doing much more than announcing itself to the ROS master and then outputting “Hello, world!” to the command-line. This is actually being broadcast to *any other* ROS nodes which have subscribed to the standard ROS log message topic [7] called `/rosout` .

To see this, first make sure `hello_world_node` is still running. Then, run the following `rostopic` command in another window to display any messages on the `/rosout` topic:

```
rostopic echo /rosout
```

In the window running `rostopic` , you should now see the content of a single `rosgraph_msgs/Log` message beneath the `rostopic` command, with content similar to the following:

```
header:
  seq: 0
  stamp:
    secs: 1392025231
    nsecs: 354360393
  frame_id: ''
level: 2
name: /hello_world_node
msg: Hello, world!
file: hello_world_node.cpp
function: main
line: 11
topics: ['/rosout']
```

This is the ROS message generated by the `ROS_INFO_STREAM(...)` command in `hello_world_node.cpp` ! You've just transmitted your first ROS message from a publisher (`hello_world_node`) to a subscriber (`rostopic`).

You'll notice that this message contains far more than just the string passed to `ROS_INFO_STREAM(...)` , it also contains metadata about on which line and in which function the message was generated. This can be very useful both when debugging your own code, or when inspecting errors in code written by others.

What might be surpsrising is that the `rostopic echo` command received the log message *long after it was sent*. This might prompt questions about where this message was stored between the time `ROS_INFO_STREAM()` was called and when `rostopic echo` was started. The answer is that ROS topic publishers can optionally buffer the last message and send it out to any new subscribers once they establish the connection.

If you tried running these two programs in reverse (first `rostopic echo` and then `hello_world_node`), there's a chance that the "Hello, world!" message gets sent before `rostopic echo` has enough time to establish a connection to `hello_world_node` . This isn't a bug or problem with ROS, however, since ROS topics use a publish/subscribe communication pattern 8 , and such patterns aren't meant to be used for synchronous communcation.

Play Around

Now that you've been given some tools, play around with different messages and the `rosnode` and `rostopic` tools. Note that both of these tools have built-in documentation that you can read by passing the `--help` argument in the following way:

```
rostopic --help
rostopic echo --help
```

Conclusion

This tutorial has hopefully introduced you to some of the core ROS concepts like the ROS master, ROS nodes, and distributed logging with rosconsole and the `/rosout` topic. Additionally, it has hopefully introduced you to how ROS libraries are installed on a Linux system, which environment variables are needed to use those libraries, and how to set those environment variables with the provided ROS setup files.

references

1 The Ubuntu Linux Distribution ↩ ↩2
2 The Bourne Again Shell ↩
3 The C++ Programming Language ↩
4 The GNU Compiler Collection ↩
5 The VIM Text Editor ↩
6 The Debian Package Management System ↩
7 ROS Topics ↩

8. Publish-Subscribe Pattern ↩