An Absolute Beginner’s Introduction to Robot Operating System (ROS™)

- Author: Tejaswi Digumarti (tejaswi.digumarti@sydney.edu.au)

- Last Updated: 5th March 2020

Foreword

This document provides a very brief introduction to Robot Operating System (ROS) and some of its components. This document assumes that the reader is an absolute beginner to ROS and has minimal understanding of Linux build systems. The primary audience is the students of the MTRX5700 - Experminetal Robotics course at the University of Sydney. This means that quite a few techincal details are skipped in favour of providing a high level understanding. Wherever possible, links to technical content are provided. As a result, while this document provides the reader with sufficient knowledge to start working with robots, it is in no way a comprehensive guide on ROS.

1. Introduction

What is ROS?

ROS is a framework for writing software for robots. Even though the name contains the term *Operating System*, it is not a complete operating system. It is rather a collection of software tools and libraries that enables programming complex robot behaviours across a wide variety of robot platforms. (see [1 (https://www.ros.org/about-ros/)], [2 (http://wiki.ros.org/)]) The tools that ROS offers can be categorized as shown in Table 1 and illustrated in Figure 1.

Table 1 - Summary of some key components in ROS (from [3 (https://www.researchgate.net/publication/314101187_Programming_for_Robotics_-_Introduction_to_ROS)]).

Plumbing	Tools	Capabilities	Ecosystem
Process Management	Visualization	Control	Package Organization
Inter-process Communication	Simulation	Planning	Software Distribution
Device Drivers	GUI	Perception	Documentation
	Data Logging	Mapping	Tutorials
	Hardware Abstraction	Manipulation	Forums

Figure 1 - Illustration of the components of ROS (from [1 (https://www.ros.org/about-ros/)].

Why use ROS?

1. ROS is modular meaning one can choose to use only specific components of ROS in their frameworks.
2. ROS programs can be distributed across multiple computers.
3. ROS modules can be written in multiple languages, as long as a client library exists (e.g. C++, Python, Matlab, Java).
4. ROS can be used as a generic tool to interact with multiple robot platforms, by abstracting away the hardware.
5. ROS has a large community of users which enables transfer of knowledge and code across industry and academia which helps in collaborative research and product development.
6. ROS has a permissive license (three-clause BSD) enabling reuse in both open source and closed source products. Code built using ROS can also be commercialized.

History

ROS was originally developed in 2000s at Stanford University and further developed by Willow Garage from 2007-2013. The first official release of ROS was ROS Box Turtle which was released on 2nd March, 2010. Since 2013 the Open Source Robotics Foundation has been managing and maintaining ROS.

Check this (http://wiki.ros.org/Distributions) link for all the cool ROS turtles.

Versions and Operating Systems

There are two main versions of ROS; ROS 1 and ROS 2. ROS 2 is still in its early stages, at the time of this document, and has not been fully adopted by the community. Hence this document will only cover topics related to ROS 1. Throughout this document when we use ROS, we refer to ROS 1. The latest long term stable distribution of ROS 1 at the time of this document is ROS Melodic Morenia (http://wiki.ros.org/melodic).

Figure 2 - The logo of ROS Melodic Morenia

While it is possible to install ROS in Linux, MacOS and Windows, it is easiest to install in Linux. We shall assume that the operating system of the user is Linux (more specifically Ubuntu 18.04).

2. The Workspace and Build system

Software developed using ROS is generally written in a ROS workspace - more commonly referred to as a *catkin workspace*. In simple terms a catkin workspace is just a folder with some special settings in it. These settings make sure that software packages written by a user are built and maintained in a clean way, with all the dependencies being found correctly, and let ROS be aware of these packages. The name of the folder can be whatever you like. Most online tutorials

call this *catkin_ws* and so we will use the same name to keep things consistent. One may create as many catkin workspaces as desired, however it is a good practice to group packages with a common end application into one workspace.

A catkin workspace can be created by first creating a normal folder and invoking the `catkin_init_workspace` or the `catkin init` command (from python-catkin-tools (https://catkin-tools.readthedocs.io/en/latest/installing.html)).

```
mkdir -p catkin_ws/src
cd catkin_ws/src
catkin_init_workspace
```

or

```
mkdir -p catkin_ws/src
cd catkin_ws
catkin init
```

Notes:
1. `catkin init` is slightly better than `catkin_init_workspace` and is recommended.
2. `catkin init` is run from the workspace folder while `catkin_init_workspace` is run from the src folder.
3. Using `catkin_init` creates a hidden folder named *.catkin_tools* inside the workspace folder where the special settings are stored.
4. Using `catkin_init_workspace` creats a file called *CMakeLists.txt* in the src folder. This is referred to as the top-level cmake file and is a symbolic link to the file */opt/ros/<version>/share/catkin/cmake/toplevel.cmake*.

2.1 The workspace structure

The workspace consists of 4 main folders which are as follows.
1. **src** - This is where your source code lies, organized into packages.
2. **build** - This is the build space for your source code. Intermediate build files are written to this folder.
3. **devel** - This is where the *targets* (executables, libraries, scripts, shared C++ headers) that are built are located. This folder also contains files that define the context of a ROS environment.
4. **logs** - This is where logs lie. 5. **install** - [Optional] - This is where the executables and libraries are installed to.

One would only work and modify files in the **src** folder and not touch the **build** and **devel** folders.

So far, we only created the **src** folder and the other folders do not exist yet. They appear once you build your code. (Build is also referred to as *compile* or *make*). This can be done by doing one of the following.
Note: Please use only one of the following and once you choose one, stick to it for the rest of the workflow. We recommend using `catkin build`.

```
catkin_make
```

or

```
catkin build
```

You should now see the other folders.

2.2 catkin build system

What happened when you invoked the above command?
ROS uses a custom **build system** called **catkin**. This build system compiled the code present in the **src** folder in the **build** folder and put the built executables and libraries in the **devel** folder. At this point, as the **src** folder is empty, the **build** and **devel** folders only have some standard packages and setup files.

So, what is a build system?
A build system takes source code and generates targets (executables, libraries, headers) from them. Let us take the example of building a C++ code written in one header file *header.h* located in the folder *include_folder* and one source file *code.cpp* and dependant on one library *libdep.so*.

Building it using the **GNU C++** compiler would be as follows.

```
g++ -Iinclude_folder code.cpp -ldep -o output
```

This command looks quite simple for this example. However, if you have a lot of source files, headers and multiple libraries that the code depends, using the above command can get quite cumbersome.

Luckily this can be automated with a build system. We configure the build system to let it know where the compiler is, where the source code, include files and libraries are, what libraries to link, what *targets* need to be built and where they have to be installed to.

One of the popular build systems is the CMake build system [4 (https://cmake.org/)]. In CMake, the configuration is specified in a file called *CMakeLists.txt*. **catkin** is a wrapper (another layer above) the CMake build

system. (python-catkin-tools are slightly more advanced, however the concept is similar.)

3. ROS Architecture

ROS Master

ROS Nodes

ROS Topics

ROS Messages

4. ROS Packages

Debugging

Visualization