ROS Made Easy
0: Introduction and Installation

Aniruddh K Budhgavi
Enigma, IIIT-B

June 16, 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 I expect them to become rapidly out of date.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++ .

2 Motivation

In any field of software engineering, instead of repeatedly reinventing the wheel, it is better to reuse code. This is accomplished through packages and libraries. This same principle of code reuse is true for robotics.

ROS, or the **Robot Operating System** provides this functionality. ROS allows you to separate the problem of low-level hardware control from things like motion planning and perception. It provides a standardised mechanism for communication between multiple processes – both within a system and across a network. It provides tools for robot visualization and simulation and for common tasks like inverse kinematics. ROS can be thought of as an operating system because it provides all the functions that are expected of one. For more details, see `http://wiki.ros.org/ROS/Introduction` or the first chapter of *A gentle introduction to ROS by Jason M. O Kane*, provided here.

3 Installation

1. Follow the installation instructions at `http://wiki.ros.org/melodic/Installation/Ubuntu` and do a **full installation**. This is very important, because it provides many add-ons, including the **Gazebo simulator**. Getting them later on and integrating them with ROS may be difficult.

2. Follow the complete installation process from beginning to end. Do not skip any steps.

3. Close all active terminals and open a fresh terminal.

4. Run the following command to ensure that all is well:

   ```
   export | grep ROS
   ```

   which should display a bunch of environment variables, like this:

   ```
   declare -x CMAKE_PREFIX_PATH="/home/akb/Documents/dd_robot/devel
       :/home/akb/Documents/four_dof_arm/devel
       :/home/akb/Documents/localROS2/devel
       :/home/akb/Documents/localROS/devel
       :/opt/ros/melodic"
   declare -x GAZEBO_MODEL_PATH="/home/akb/Documents/localROS2/src
       :/usr/share/gazebo-9/models
   ```

```
    :/home/akb/Documents/four_dof_arm/src
    :/home/akb/Documents/dd_robot/src"
declare -x LD_LIBRARY_PATH="/home/akb/Documents/dd_robot/devel/lib
    :/home/akb/Documents/four_dof_arm/devel/lib
    :/home/akb/Documents/localROS2/devel/lib
    :/home/akb/Documents/localROS/devel/lib
    :/opt/ros/melodic/lib"
declare -x PKG_CONFIG_PATH="/home/akb/Documents/dd_robot/devel/lib/pkgconfig
    :/home/akb/Documents/four_dof_arm/devel/lib/pkgconfig
    :/home/akb/Documents/localROS2/devel/lib/pkgconfig
    :/home/akb/Documents/localROS/devel/lib/pkgconfig
    :/opt/ros/melodic/lib/pkgconfig"
declare -x ROSLISP_PACKAGE_DIRECTORIES="/home/akb/Documents/dd_robot/devel
/share/common-lisp
    :/home/akb/Documents/four_dof_arm/devel/share/common-lisp
    :/home/akb/Documents/localROS2/devel/share/common-lisp
    :/home/akb/Documents/localROS/devel/share/common-lisp"
declare -x ROS_DISTRO="melodic"
declare -x ROS_ETC_DIR="/opt/ros/melodic/etc/ros"
declare -x ROS_MASTER_URI="http://localhost:11311"
declare -x ROS_PACKAGE_PATH="/home/akb/Documents/dd_robot/src
    :/home/akb/Documents/four_dof_arm/src
    :/home/akb/Documents/localROS2/src
    :/home/akb/Documents/localROS/src
    :/opt/ros/melodic/share"
declare -x ROS_PYTHON_VERSION="2"
declare -x ROS_ROOT="/opt/ros/melodic/share/ros"
declare -x ROS_VERSION="1"
```

5. Also, try:

```
roscore
```

4 Core concepts

4.1 Packages

1. All ROS software is organized into packages. A package consists of:
   (a) A folder having the same name as the package name. The folder **is** the package.
   (b) A manifest file, `package.xml` which provides information about the package. Most importantly, it lists the packages that this package depends on.
   (c) A CMake configuration file, `CMakeLists.txt` which is used to compile the package.
   (d) The source code of the package executables.
   (e) Custom messages and services.
2. Example packages are `std_msgs`, `turtlesim`, `gazebo_ros` and `rviz`.
3. Use the `rospack` command to view information about packages. As a start, try:

   ```
   rospack list-names
   ```

   which should provide you with a list of the currently installed ROS packages.
4. Also try:

   ```
   rospack find turtlesim
   ```

   which should give you the path to the turtlesim package, `/opt/ros/melodic/share/turtlesim`.
5. To know more about rospack, try:

```
rospack help
```

6. We will later see how to create and build our own packages.

4.2 Nodes

1. A running instance of a ROS process is a node. The phrase ”running instance” is key – an executable file sitting idle on your hard disk is **not** a node.
2. A node can:
   - Compute something, just like any regular C++ or Python program.
   - Communicate with other nodes, or with the running instance of ROS.
3. There can be multiple nodes running on the same system, as well as multiple nodes across different systems.
4. The code for a node can be written in any language that supports the ROS client library, though official support is right now limited to C++ and Python.
5. Nodes can communicate through *messages*, *services* and the *parameter server*.

4.3 Messages and topics

1. The primary method of communication between ROS nodes is through messages on topics.
2. Communication through topics is many-to-many in nature. Nodes can *subscribe* to (receive messages from) a topic, and *publish* to (send messages to) a topic. When a message is published to a topic, all the nodes that have subscribed to that topic receive that message.
3. Topics, then, are like message boards or forums. It is not known which node published which message (unless the message itself contains this).
4. Every topic has a message type associated with it. Only messages of that type can be published to that topic. You can think of message type as being analogous to data type. The messages resemble structs in C.
5. To view the message types available, use the command:

   ```
   rosmsg list
   ```

6. To view the format of a message, use `rosmsg show <msg-type>`. For example,

   ```
   rosmsg show geometry_msgs/Twist
   ```

   Output:

   ```
   geometry_msgs/Vector3 linear
       float64 x
       float64 y
       float64 z
   geometry_msgs/Vector3 angular
       float64 x
       float64 y
       float64 z
   ```

7. An important point is that though the communication through topics is many-to-many, the messages themselves are directly transmitted between the publisher and subscribers – there is no ”server” that acts as an intermediary.
8. To know more about `rosmsg`, try `rosmsg help`.

4.4 Services

1. Services provide one-to-one communication between nodes.
2. A ROS node (service provider) can provide a service which can be *called* by another node (client).
3. When a service is called, a request message is sent from the client to the service provider, which can reply with a response message. These request/response messages are different from topic/messages.

4.5 The parameter server

1. The parameter server is a place where globally-accessible data can be stored.
2. Every instance of ROS has one parameter server, and every node on that instance can get and set parameters on that server.
3. These parameters are typically initialized when a ROS instance is started.

4.6 The master

1. The ROS master is a program which must be started before any nodes are activated.
2. The ROS master provides many functions, including:
   - Naming and registering nodes.
   - Establishing publisher-subscriber connections and tracking topics.
   - The parameter server.
   - Tracking services.
   - Debugging tools.

5 Looking forward

I understand if so far, it's been quite theoretical and dry. But this is necessary to get a good grasp of the fundamentals of ROS. Next time, we'll see how to start the ROS master, start nodes, and publish messages to topics.