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 I expect them to become rapidly out of date.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. First, ensure you have Python 2.7 installed.
2. 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.
3. Follow the complete installation process from beginning to end. Do not skip any steps.
4. Close all active terminals and open a fresh terminal.
5. 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"
```

6. 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. To see the contents of a package, use `rosls <package-name>`:

```
:~$ rosls turtlesim
```

Output:

```
cmake  images  msg  package.xml  srv
```

7. To go to a package directory, use `roscd <package-name>`:

```
:~$ roscd turtlesim
```

8. We will later see how to create and build our own packages. For more information, see `http://wiki.ros.org/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. The same executable could be used again and again to create many running copies of the same node (but with different node names – node names are unique for a particular ROS instance).
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*.
6. For more information, see `http://wiki.ros.org/Nodes`

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`. For more information, see `http://wiki.ros.org/Messages` and `http://wiki.ros.org/Topics`.

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. For more information, see `http://wiki.ros.org/Services`.

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. For more information, see `http://wiki.ros.org/Parameter%20Server`.

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.

3. For more information, see `http://wiki.ros.org/Master`.

5 Turtlesim

1. Run the following three commands, **each in a separate terminal, in the same order**:

```
:~$ roscore
:~$ rosrun turtlesim turtlesim_node
:~$ rosrun turtlesim turtle_teleop_key
```

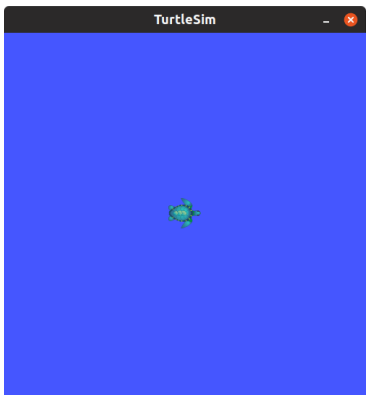
Output from first terminal should be something like:

```
... logging to /home/akb/.ros/log/9ad6f8d6-afcd-11ea-9d13-907841080e03
/roslaunch-akb-G3-3579-16205.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://akb-G3-3579:32923/
ros_comm version 1.14.5


SUMMARY
========

PARAMETERS
    * /rosdistro: melodic
    * /rosversion: 1.14.5

NODES

auto-starting new master
process[master]: started with pid [16353]
ROS_MASTER_URI=http://akb-G3-3579:11311/

setting /run_id to 9ad6f8d6-afcd-11ea-9d13-907841080e03
process[rosout-1]: started with pid [16364]
started core service [/rosout]
```

Output from the second terminal:

```
[ INFO] [1592310832.800133368]: Starting turtlesim with node name /turtlesim
[ INFO] [1592310832.814790457]: Spawning turtle [turtle1] at x=[5.544445],
 y=[5.544445], theta=[0.000000]
```

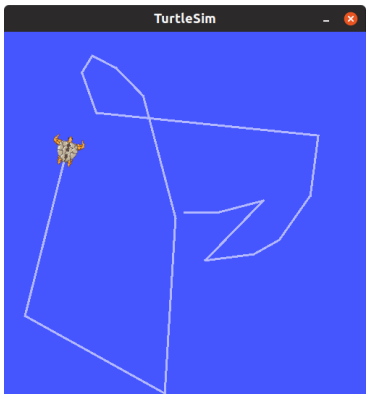
A window like this should open up:



Output from the third terminal:

```
Reading from keyboard
---------------------------
Use arrow keys to move the turtle. 'q' to quit.
```

2. Keep the third terminal window and the turtlesim window open. With the third terminal window active, you should be able to control the turtle using the arrow keys. You should see the turtle move and leave a trail behind it. Like this:



3. Let's now analyze what we did. First, we started a ROS master using the `roscore` command. Next, we started two nodes, each using the `rosrun` command. The format for rosrun is:

```
:~$ rosrun <pkg> <type>
```

4. There's nothing magical about rosrun. It is equivalent to going to the folder of turtlesim, finding the right executable, and running it directly, like a `./a.out` from C.

5. To view the currently running nodes, in a new terminal, run:

```
:~$ rosnode list
```

Output:

```
/rosout
/teleop_turtle
/turtlesim
```

So we have three nodes: `/rosout`, which is like STDOUT but for ROS; `/turtlesim`, which is the turtle window; and `/teleop_turtle`, which accepts the arrow key inputs.

6. There must be some way by which `/teleop_turtle` and `/turtlesim` are communicating. Let's investigate this. Run:

```
:~$ rosnode info /turtlesim
```

Output (by the way, use the TAB key for autocomplete):

```
--------------------------------------------------------------------------------
Node [/turtlesim]
Publications:
    * /rosout [rosgraph_msgs/Log]
    * /turtle1/color_sensor [turtlesim/Color]
    * /turtle1/pose [turtlesim/Pose]

Subscriptions:
    * /turtle1/cmd_vel [geometry_msgs/Twist]

Services:
    * /clear
    * /kill
    * /reset
    * /spawn
    * /turtle1/set_pen
    * /turtle1/teleport_absolute
    * /turtle1/teleport_relative
    * /turtlesim/get_loggers
    * /turtlesim/set_logger_level


contacting node http://akb-G3-3579:36561/ ...
Pid: 21017
Connections:
    * topic: /rosout
    * to: /rosout
    * direction: outbound (54721 - 127.0.0.1:38478) [53]
    * transport: TCPROS
    * topic: /turtle1/cmd_vel
    * to: /teleop_turtle (http://akb-G3-3579:40119/)
    * direction: inbound (33248 - akb-G3-3579:53883) [56]
    * transport: TCPROS
```

And run:

```
:~$ rosnode info /teleop_turtle
```

Output:

```
--------------------------------------------------------------------------------
Node [/teleop_turtle]
Publications:
    * /rosout [rosgraph_msgs/Log]
    * /turtle1/cmd_vel [geometry_msgs/Twist]

Subscriptions: None

Services:
    * /teleop_turtle/get_loggers
    * /teleop_turtle/set_logger_level


contacting node http://akb-G3-3579:40119/ ...
Pid: 21190
Connections:
```

```
        * topic: /rosout
        * to: /rosout
        * direction: outbound (53883 - 127.0.0.1:33246) [22]
        * transport: TCPROS
        * topic: /turtle1/cmd_vel
        * to: /turtlesim
        * direction: outbound (53883 - 127.0.0.1:33248) [20]
        * transport: TCPROS
```

You can ignore most of the output. Pay attention to the topics being published and subscribed to. Specifically, observe that `/teleop_turtle` publishes to `/turtle1/cmd_vel` while `/turtlesim` is subscribed to it. This could be the mechanism by which the instructions are being communicated.

7. Let's dig even deeper. Run:

```
:~$ rostopic list
```

Output:

```
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

Which is a list of the currently active topics. Now run (use TAB!):

```
:~$ rostopic echo /turtle1/cmd_vel
```

Do this in a separate window, and observe what happens when you command the turtle using `/teleop_turtle`. Output:

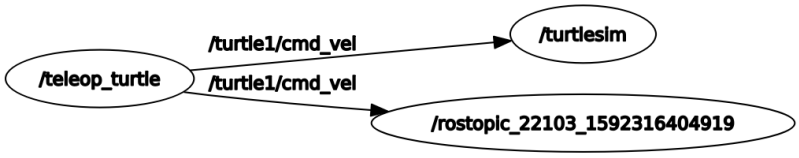
```
linear:
x: 2.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 0.0
---
linear:
x: 0.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 2.0
---
...
```

It would seem that this is indeed the way the commands are transmitted between the nodes.

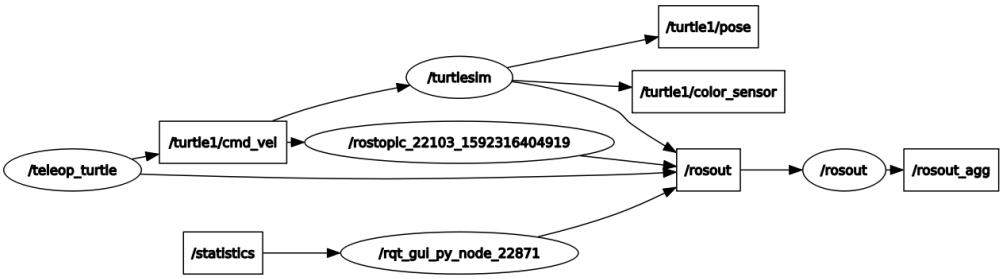
8. Another utility exists to observe the relation between multiple nodes. Try:

```
:~$ rqt_graph
```

Initially, the graph is like this:



You can see the relation between `/teleop_turtle` and `/turtlesim`. The other node is due to the `rostopic echo /turtle1/cmd_vel` command. Now, uncheck all the 'hide' options, and choose "Nodes/Topics(all)". The graph looks like:



`rqt_graph` is a valuable tool for visualizing and debugging.

9. A key point: Publishers don't know who the subscribers are, and subscribers don't know who the publishers are.

10. Now, let's look at the message format used in `/turtle1/cmd_vel`. Run:

```
:~$ rostopic type /turtle1/cmd_vel | rosmsg show
```

The `|` operator takes the output of the preceding command and gives it as an argument to the next command. Output:

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

This tells us that the message for `/turtle1/cmd_vel` is composed of two `geometry_msgs/Vector3`, one of which is `linear` and the other `angular`. The `geometry_msgs/Vector3` messages are themselves composed of three `float64`s – `x`, `y` and `z`.

11. It is possible to publish messages from the command line. With the three terminals from the beginning still running, open a fourth terminal and run:

```
:~$ rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist '[2,0,0]' '[0,0,1]'
```

And watch the turtle execute a circle.

12. Let's break down that last command. `rostopic pub` – we wish to publish; `-r 1` – at a fixed rate of 1 Hz; `/turtle1/cmd_vel` – topic; `geometry_msgs/Twist` – message type; `'[2, 0, 0]' '[0, 0, 1]'` – message contents.

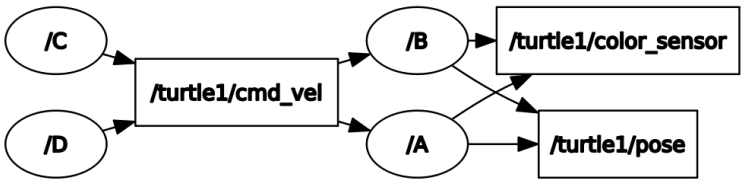
13. `/turtlesim` only uses `x` from `linear` and `z` from `angular`. It ignores the other two values.

14. As a final demonstration, close all terminals, reopen five new terminals and run (one in each terminal):

```
:~$ roscore
:~$ rosrun turtlesim turtlesim_node __name:=A
:~$ rosrun turtlesim turtlesim_node __name:=B
:~$ rosrun turtlesim turtle_teleop_key __name:=C
:~$ rosrun turtlesim turtle_teleop_key __name:=D
```

What do you think will happen? Will `C` control `A` and `D`, `B`? Or vice-versa? Experiment and check.

15. With the other terminals still open, run `rqt_graph` and see the result.



16. This should serve as an important lesson: Nodes are loosely coupled in nature. They are as independent as possible.

17. Lastly, try the `roswtf` command.

18. This section was adapted from `http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes`, `http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics` and *A Gentle Introduction to ROS, Chapter Two*

6 Looking ahead

Next time, we will be creating our first ROS package and writing our first ROS nodes.