Ros Tutorial

Mohammadali Varfan

May 15, 2017

This is a document for the important tutorials of ROS.

1 Basics

1.1 Concepts

- **Node:**
  Nodes are the process that perform computation. Each ROS node is written using ROS client libraries such as roscpp and rospy. Using client library APIs, we can implement different types of communication methods in ROS nodes. In a robot, there will be many nodes to perform different kinds of tasks. Using the ROS communication methods, it can communicate with each other and exchange data. One of the aims of ROS nodes is to build simple processes rather than a large process with all functionality.

- **Master:**
  The ROS Master provides name registration and lookup to the rest of the nodes. Nodes will not be able to find each other, exchange messages, or invoke services without a ROS Master. In a distributed system, we should run the master on one computer, and other remote nodes can find each other by communicating with this master.

- **Topics:**
  Each message in ROS is transported using named buses called topics.
  When a node sends a message through a topic, then we can say the node is publishing a topic. When a node receives a message through a topic, then we can say that the node is subscribing to a topic. The publishing node and subscribing node are not aware of each other's existence. We can even subscribe a topic that might not have any publisher. In short, the production of information and consumption of it are decoupled. Each topic has a unique name, and any node can access this topic and send data through it as long as they have the right message type.

- **Service:**
  In some robot applications, a publish/subscribe model will not be enough if it needs a request/response interaction. The publish/subscribe model

is a kind of one-way transport system and when we work with a distributed system, we might need a request/response kind of interaction. ROS Services are used in these case. We can define a service definition that contains two parts; one is for requests and the other is for responses. Using ROS Services, we can write a server node and client node. The server node provides the service under a name, and when the client node sends a request message to this server, it will respond and send the result to the client. The client might need to wait until the server

1.2 Important Commands

- **rosmsg**
  The "rosmsg" command tool can be used to inspect the message header and the field types. The following command helps to view the message header of a particular message:

  ```
  $ rosmsg show std_msgs/Header
  ```

- **rosrun**
  This command is used to run an executable inside a package.

- **rqt_graph**
  The topics are mentioned in a rectangle and nodes are represented in ellipses. The messages and parameters are not included in this graph.

- **rosnode info [node_name]**
  This will print the information about that node.

- **rosnode list**
  This will list all the running nodes.

- **rostopic bw [topic]**
  This command will display the bandwidth used by the given topic

- **rostopic echo [topic]**
  This command will print the content of the given topic

- **rostopic find [message_type]**
  This command will find topics using the given message type

- **rostopic hz [topic]**
  This command will display the publishing rate of the given topic

- **rostopic info [topic]**
  This command will print information about an active topic

- **rostopic list**
  This command will list all active topics in the ROS system

- **rostopic pub [topic] [message_type] [args]**
  This command can be used to publish a value to a topic with a message type
- **rostopic type [topic]**
  This will display the message type of the given topic.

1.3 Creating ROS Workspace

- **Creating Workspace**
  Build a workspace folder in the home directory and create a src folder inside the workspace folder:
  mkdir home/catkin_ws/src
  Switch to the source folder. The packages are created inside this package:
  cd home/catkin_ws/src
  Initialize a new catkin workspace:
  catkin_init_workspace
  We can build the workspace even if there are no packages. We can use the following command to switch to the workspace folder:
  cd home/catkin_ws
  The catkin_make command will build the following workspace:
  catkin_make

- **Set Environment Variables**
  After building the empty workspace, we should set the environment of the current workspace to be visible by the ROS system. This process is called overlaying a workspace. We should add the package environment using the following command:
  echo "source ~/catkin_ws/devel/setup.bash" >>~/.bashrc
  source ~/.bashrc

1.4 Creating ROS Package

Switch to the catkin workspace src folder and create the package using the following command:
catkin_create_pkg [package_name] [dependency1] [dependency2]

For building package we use "**catkin_make**" command in the root directory of ROS ("~/catkin_ws")

1.5 ROS Services

The ROS services are a type request/response communication between ROS nodes. One node will send a request and wait until it gets a response from the other. The request/response communication is also using the ROS message description.

Similar to the message definitions using the .msg file, we have to define the service definition in another file called .srv, which has to be kept inside the srv sub directory of the package. Similar to the message definition, a service description language is used to define the ROS service types. An example service description format is as follows:
#Request message type

string str
—
#Response message type
string str

1.6 ROS Nodes

page 28 / "read from other book because of python".
For example, the camera node on a robot could be named "camera", and it could output a message topic named "image" and read a parameter named "frame_rate" to know how fast to send images.

1.6.1 Namespaces and Remapping

what happens when a robot has two cameras? We wouldnt want to have to write a separate program for each camera, nor would we want the output of both cameras to be interleaved on the image topic, since that would require all subscribers to image to have logic that separates the image streams. ROS provides two mechanisms to handle these situations: namespaces and remapping.

In the previous example, a robot with two cameras could launch two camera drivers in separate namespaces, such as left and right, which would result in image streams named left/image and right/image.

This avoids a topic name collision, but how could we send these data streams to another program that was still expecting to receive messages on the topic image? One answer would be to launch this other program in the same namespace as the first, but perhaps this program needs to reach into more than one namespace. Enter remapping.

In ROS, any string in a program that defines a name can be remapped at runtime. As one example, there is a commonly used program in ROS called image_view that renders a live video window of images being sent on the image topic. At least, that is what is written in the source code of the image_view program. Using remapping, we can instead cause the image_view program to render the right/image topic, or the left/image topic, without having to modify the source code of image_view!

1.6.2 roslaunch

roslaunch is a command-line tool designed to automate the launching of collections of ROS nodes:
roslaunch [PACKAGE_NAME] [LAUNCH_FILE_NAME]

Launch files are XML files that describe a collection of nodes along with their topic remappings and parameters. By convention, these files have a suffix of .launch.

```
<launch>
        <node name="talker" pkg="rospy_tutorials"
                type="talker.py" output="screen" />
        <node name="listener" pkg="rospy_tutorials"
                type="listener.py" output="screen" />
</launch>
```

1.7

1.8