https://wiki.bu.edu/roboclass/index.php?title=ROS_Fundamentals

10 min read

ROS Fundamentals - Introduction to Robotics

A "module" of the ROS system, which provides a set of nodes, message definitions (topics), and a directory with at least the files "package.xml" and "CMakeLists.txt".

★★ "package.xml" contains general information about the package, such as a description, the contact information about the maintainer, and the dependencies of this package on other packages. The "CMakeLists.txt" file contains information about compiling the code for the scripts, nodes and messages/services. This is mainly required only for C/C++ code and messages/services; if you have only Python code, this is not required.

Packages commonly contain the subdirectories:

- "src": with the source code for the nodes,
- "script": with programs that are typically commands that do not create nodes,
- "launch": with launch files (more on this later)

See [1] for additional common structures. At a high level, packages can be system packages (those that come with a ROS distro and are typically installed under /opt/...) and user packages (those that *you* are developing).

A special directory containing all the packages that are under development (i.e., that are not system packages).

In our ROSBot platform and on the VM, the ROS workspace is the directory `ros_ws`.

TODO: insert screenshot of the content of the ros_ws directory, and list the packages that are there.

★★ "Making" or "building" a workspace means to compile all the code and message definitions in the packages in workspace. This can be done with the command `catkin make` (run from the workspace directory). This command uses the information contained in the CMakeLists.txt files inside each package to know what code and message to define. This command was part of the first in-class activity; see also the page Update the me416 labs package.

A node is a process that runs is intended to run continuously on your robot, and communicates with other nodes typically using *ROS messages* exchanged on *ROS topics*. Each node has a name. You can see what nodes are running under what names by using the command "rosnode list".

★★ Typically, the name of the node is the same as the one of the executable, although it could be set to be different. As a consequence, the same executable can be launched multiple times under different names, creating different nodes.

Nodes can be started manually using the command "rosrun <packagename> <nodename>". Note that if your workspace is configured correctly, you can call this command from any directory, and the executable <nodename> can be in any subdirectory of the package (e.g., it does not matter if it is under /src or /scripts). Note that you can use tab completion on <packagename> and <nodename>.

Note that each node, since it is supposed to run continuously, essentially "occupies" a terminal. You will need to use different terminals for different nodes.

You can launch a node from a package using `rosrun`, e.g.: `rosrun me416_lab talker.py`.

The master node needs to be run first.

Master node[edit]

The master node is a special node that is started with the command "roscore". This node needs to be started first, as it is necessary to launch other nodes (you will get an error if the master is not running), and handles the registrations necessary to publish and subscribe to topics.

The master node can be run from anywhere. In other words, it does not matter which directory you are in when running "roscore."

The output from the master node to the console provides information on which nodes are being run, and which have stopped running (with a semi-intelligible error message).

★★★ You can configure a machine to use a roscore running on another machine on the same network. In this way, you can send topics.

The most common way to communicate between nodes is through messages exchanged over topics. Topics are "channels" on which nodes can either *publish* (send messages) or *subscribe* (receive messages). You can see a list of topics using the command "rostopic list". Each topic is assigned a message type at its definition. These types are defined in ROS packages (either system or user-defined), and are essentially structures where the fields can be some primitive type (e.g., string, integer or floating point numbers) or other message types. Topics can transfer messages only of the type with which they have been defined. You can find information about the message type of a topic by using

```
rostopic info <topicname>
```

Note that tab completion works with rostopic.

While topics are "channels", messages are the "packets" exchanged on these "channels". You can see what messages are exchanged over topics. Loosely speaking, you will have "input" nodes that only publish information (images from a camera, commands from a keyboard), "processing" nodes that subscribe to a topic and publish the data on another after transforming it, and "output" nodes, that simply visualize the results.

TODO: insert screenshot with example

★★★ Message definitions Messages are defined under the `/msg/` subdirectory, following the format described at [2]

TODO: how to import a message in Python

A structure containing common meta-information about a package. See [[3][webpage]] for details.

TODO: add more details

Note 1: the field seq is automatically populated when a message is published. However, the fields frame_id and stamp need to be manually populated if they are used.

Note 2: the field stamp is a time structure where seconds and nanoseconds are separated. The Python implementation has the method .to_sec() to obtain combine the two fields in a single number expressed in seconds.

As you build your system, you will find having to start multiple nodes that perform interrelated functions. For instance, a node that publishes images from a camera, another that performs some image processing, and other two that show the original and processed images. Instead of having to launch every node (including the master) manually, you can embed them into a launch file.

Launching launch files[edit]

You can run a launch file with the `roslaunch` command, which has the following syntax:

```
roslaunch <package_name> <file_name>.launch
```

Node that `roslaunch` will launch a master (i.e., `roscore`)

Launch files[edit]

Launch files are usually stored in the `launch` subdirectory of a ROS package. An example of a launch file (`manual_drive.launch`) is the following.

```
<?xml version="1.0"?>
<?ignore
 Script streams camera to a screen and loads the keyboard motor controls
?>
<launch>
  <include file="$(find me416_lab)/launch/camera.launch"/>
  <node pkg="me416_lab" name="motor_command" type="motor_command.py"/>
  <node pkg="me416_lab" name="key_op" type="key_op.py" output="screen"/>
</launch>
```

The launch file contains the following parts.

Header (mandatory)[edit]

The first line of every launch file is always the same.

```
 <?xml version="1.0"?>
```

Optional description[edit]

You can put a description of the launch file using the `?ignore` tag.

```
<?ignore
 <content of the description>
?>
```

Main tags (mandatory)[edit]

All the "instructions" in the launch file needs to be wrapped inside a `launch` tag:

```
<launch>
 [...]
</launch>
```

Nodes[edit]

To ask to start a node, you can use the command

```
  <node pkg="package_name" name="node_name" type="node_file_name"/>
```

There `node` tag can take the following attributes:

- The `type` attribute must match the name of the executable or script for a node (the same as the one you would use with `rosrun`.
- The `name` attribute changes the name of the node (the one listed by `rosnode list`); this name will override the one specified by the `rospy.init()` command in the script. This mechanism allows you to launch the same node multiple times with different names without having to use the `anonymous=True` argument in the code.
- Usually, the output from a node (produced with `rospy.loginfo()` or `print`) is suppressed. You can change this by adding the attribute `output="screen"`.

Inclusion of other launch files[edit]

You can launch other launch files by including them, using the following syntax:

```
<include file="$(find package_name)/launch/file_name.launch"/>
```

where you need to change `package_name` to specify in which package ROS should look for a launch file with file name `file_name<code>. This line is roughly equivalent to the command`

```
roslaunch package_name file_name
```

`TODO: parameters, topic remapping, service calls`

ROS bags are archives of messages and topics You can record/replay all topics, or only specific topics, and you can also "filter" bags (change topic names, remove topics, etc.) The ROS bags record also the timing of each message.

You use the command rosbag (on the command line, not Python) to manage ROS bags.

rosbag record[edit]

rosbag record --all (or rosbag record -a) records ALL messages and their timing from ALL the topics (even for nodes not started in the same terminal)

If, instead, you are interested in recording specific topics (e.g., the topic chatter), you can specify them instead of using the option --all, e.g., rosbag record chatter.

The rosbag record command will store all the information captured in a .bag file in the current directory (the filename is automatically derived from the current date and time).

The only way to stop the recording is by using the Ctrl+C key combination

rosbag play[edit]

The command rosbag play filename.bag will replay the content of the specified ROS bag file. All the messages from all the topics recorded are published again with the same relative timing as they were recorded. You can run the same or additional nodes while replaying a ROS bag; these node can subscribe or publish on the same topics from the bag.

A couple of important options for rosbag play:

```
--start=SEC Start playing from SEC seconds into the file
--duration=SEC Play for a duration of SEC seconds
```

See the official ROS Wiki page for additional options and uses of the rosbag command.

The ROS architecture makes it easy to be modular. For prototyping, it is best to keep each node as simple as possible, so that you can reuse them. Use nodes that are already available and tested, if possible. The only downside is that publishing/subscribing on nodes is slower than doing everything in the same program.

★★★★ You can use nodelets to increase performance.