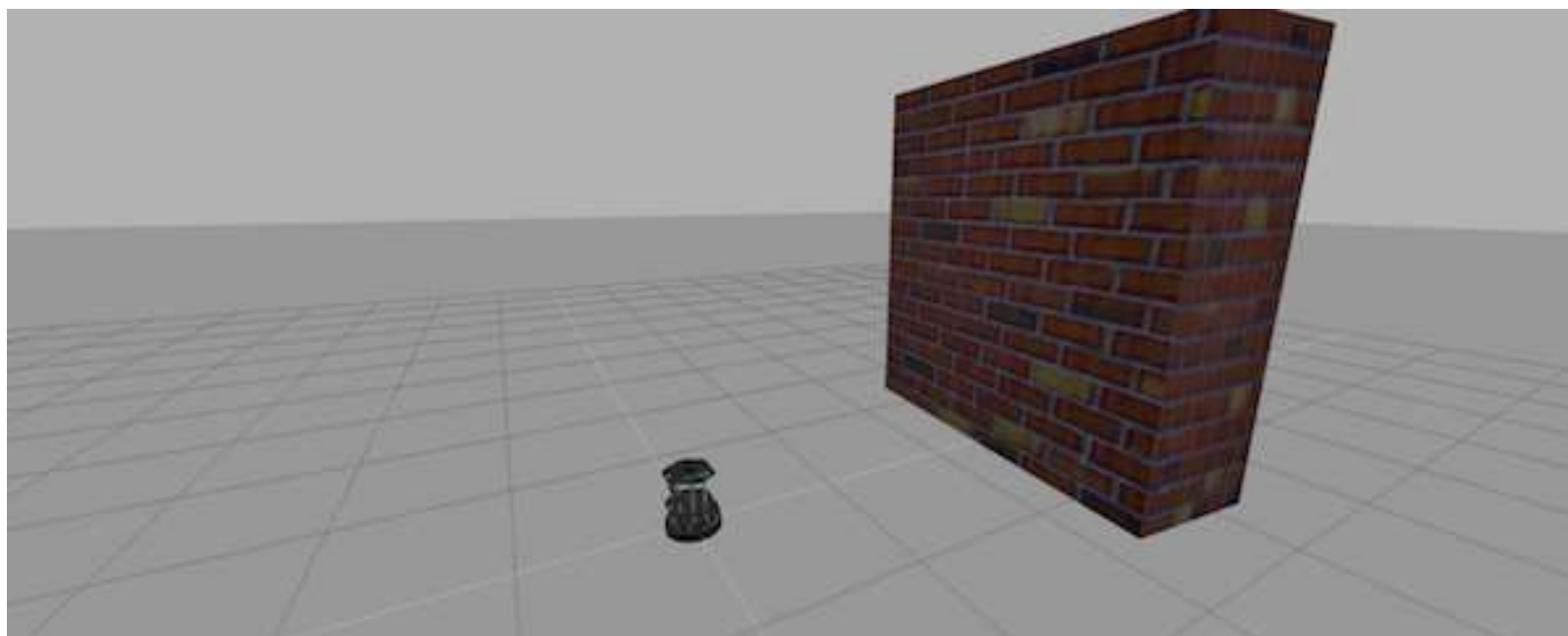

ROS Basics in 5 days

Unit 2 ROS Basics



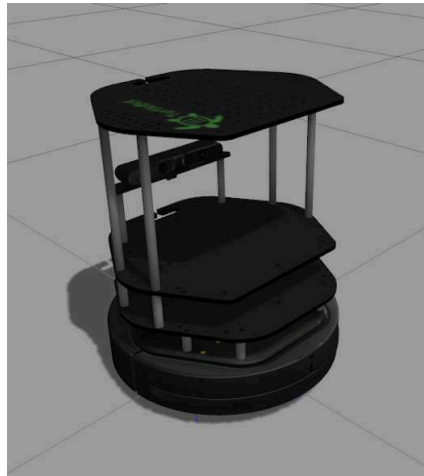
- Summary -

Estimated time to completion: 1'5 hours

Simulated robot: Turtlebot

What will you learn with this unit?

- How to structure and launch ROS programs (packages and launch files)
- How to create basic ROS programs (Python based)
- Basic ROS concepts: Nodes, Parameter Server, Environment Variables, Roscore



- End of Summary -

2.1 What is ROS?

This is probably the question that has brought you all here. Well, let me tell you that you are still not prepared to understand the answer to this question, so... let's get some work done first.

2.1.1 Move a Robot with ROS

On the right corner of the screen, you have your first simulated robot: the Turtlebot 2 robot against a large wall.

The screenshot displays a ROS simulation environment. On the left, a sidebar contains the following information:

- ROS IN 5 DAYS**
- Unit 2: Basic Concepts**
- Estimated time to completion:** 1'5 hours
- Simulated robot:** Turtlebot
- What will you learn with this unit?**
 - How to structure and launch ROS programs (packages and launch files)
 - How to create basic ROS programs (Python based)
 - Basic ROS concepts: Nodes, Parameter Server, Environment Variables, Roscore

The main area is divided into three panels:

- File Explorer:** Shows a directory structure for 'catkin_ws' with subdirectories 'build', 'devel', and 'src'.
- Terminal:** Displays the prompt 'user:~\$'.
- 3D View:** Shows a Turtlebot 2 robot in a simulated environment with a brick wall and a tiled floor.

The bottom status bar indicates the current page is '2 - ROS Basics' and the title is 'ROS Basics in 5 Days (Python) Noetic'. The page number '15' is also visible.

Let's move that robot!

How can you move the Turtlebot?

The easiest way is by executing an existing ROS program to control the robot. A ROS program is executed by using some special files called **launch files**.

Since a previously-made ROS program already exists that allows you to move the robot using the keyboard, let's *launch* that ROS program to teleoperate the robot.

- Example 2.1 -

Execute the following command in Terminal number #1.

► Execute in terminal #1

```
In [ ]: roslaunch turtlebot_teleop keyboard_teleop.launch
```



📄 terminal #1 Output

In []:

Control Your Turtlebot!



Moving around:

u i o
j k l
m , .

q/z : increase/decrease max speeds by 10%

w/x : increase/decrease only linear speed by 10%

e/c : increase/decrease only angular speed by 10%

space key, k : force stop

anything else : stop smoothly

CTRL-C to quit

Now, you can use the keys indicated in the Terminal Output in order to move the robot around. The basic keys are the following:

i	Move forward
,	Move backward
j	Turn left
l	Turn right
k	Stop
q	Increase / Decrease Speed
z	

Try it!! When you're done, you can press **Ctrl+C** to stop the execution of the program.

roslaunch is the command used to launch a ROS program. Its structure goes as follows:

```
In [ ]: roslaunch <package_name> <launch_file>
```



As you can see, that command has two parameters: the first one is **the name of the package** that contains the launch file, and the second one is **the name of the launch file** itself (which is stored inside the package).

- End Example 2.1 -

2.2 Now... what's a package?

ROS uses **packages** to organize its programs. You can think of a package as **all the files that a specific ROS program contains**; all its cpp files, python files, configuration files, compilation files, launch files, and parameters files.

All those files in the package are organized with the following structure:

- **launch** folder: Contains launch files
- **src** folder: Source files (cpp, python)
- **CMakeLists.txt**: List of cmake rules for compilation
- **package.xml**: Package information and dependencies

To go to any ROS package, ROS gives you a command named **roscd**. When typing:

```
In [ ]: roscd <package_name>
```



It will take you to the path where the package *package_name* is located.

- Example 2.2 -

Go to Terminal #1, navigate to the `turtlebot_teleop` package, and check that it has that structure.

► Execute in terminal #1

In []: `roscd turtlebot_teleop`

In []: `ls`

```
user:~$ roscd turtlebot_teleop/  
user:/home/simulations/public_sim_ws/src/all/turtlebot/turtlebot_teleop$ ls  
CHANGELOG.rst  CMakeLists.txt  launch  package.xml  param  README.md  src
```

Every ROS program that you want to execute is organized in a package.

Every ROS program that you create will have to be organized in a package.

Packages are the main organization system of ROS programs.

- End Example 2.2 -

2.3 And... what's a launch file?

We've seen that ROS uses launch files in order to execute programs. But... how do they work? Let's have a look.

- Example 2.3 -

Open the **launch** folder inside the **turtlebot_teleop** package and check the **keyboard_teleop.launch** file.

► Execute in terminal #1

In []: `roscd turtlebot_teleop`

```
In [ ]: cd launch
```

```
In [ ]: cat keyboard_teleop.launch
```

terminal #1 Output

```
In [ ]: <launch>
  <!-- turtlebot_teleop_key already has its own built in velocity smoother -->
  <node pkg="turtlebot_teleop" type="turtlebot_teleop_key.py" name="turtlebot_teleop_keyboard" output="screen"
    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="1.5" type="double"/>
    <remap from="turtlebot_teleop_keyboard/cmd_vel" to="/cmd_vel"/> <!-- cmd_vel_mux/input/teleop/-->
  </node>
</launch>
```

In the launch file, there are some extra tags for setting parameters and remaps. The `<param>` tag defines a parameter in the Parameter Server, where nodes get parameters from. A lot of nodes use parameters to avoid modifying the source code, and above you can see how they are added.

The `<remap>` tag redirects messages from a topic to another. In this case, the teleop node publishes to `turtlebot_teleop_keyboard/cmd_vel` by default. We want it to publish on `cmd_vel`, so the tag is added.

All launch files are contained within a **<launch>** tag. Inside that tag, you can see a **<node>** tag, where we specify the following parameters:

1. **pkg="package_name"** # Name of the package that contains the code of the ROS program to execute
2. **type="python_file_name.py"** # Name of the program file that we want to execute
3. **name="node_name"** # Name of the ROS node that will launch our Python file
4. **output="type_of_output"** # Through which channel you will print the output of the Python file

- End Example 2.3 -

2.4 Create a ROS package

Until now we've been checking the structure of an already-built package... but now, let's create one ourselves.

When we want to create packages, we need to work in a very specific ROS workspace, which is known as ***the catkin workspace***. The catkin workspace is the directory in your hard disk where your own **ROS packages must reside** in order to be usable by ROS. Usually, the ***catkin workspace*** directory is called ***catkin_ws***.

- Example 2.4 -

Go to the catkin_ws in your Terminal.

In order to do this, type ***roscd*** in the *terminal*. You'll see that you are thrown to a ***catkin_ws/devel*** directory. Since you want to go to the workspace, just type ***cd ..*** to move up 1 directory. You must end up here in the ***/home/user/catkin_ws***.

► Execute in terminal #1

```
In [ ]: roscd
```



```
In [ ]: cd ..
```



```
In [ ]: pwd
```



📄 terminal #1 Output

```
In [ ]: /home/user/catkin_ws
```



Inside this workspace, there is a directory called ***src***. This folder will contain all the packages created. Every time you want to create a new package, you have to be in this directory (***catkin_ws/src***). Type in your Terminal ***cd src*** in order to move to the source directory.

► Execute in terminal #1

```
In [ ]: cd src
```



Now we are ready to create our first package! In order to create a package, type in your Terminal:

► Execute in terminal #1

```
In [ ]: catkin_create_pkg my_package rospy
```



This will create inside our **src** directory a new package with some files in it. We'll check this later. Now, let's see how this command is built:

```
In [ ]: catkin_create_pkg <package_name> <package_dependencies>
```



The **package_name** is the name of the package you want to create, and the **package_dependencies** are the names of other ROS packages that your package depends on.

- End Example 2.4 -

- Example 2.5 -

In order to check that our package has been created successfully, we can use some ROS commands related to packages. For example, let's type:

► Execute in terminal #1

```
In [ ]: rospack list
```



```
In [ ]: rospack list | grep my_package
```



```
In [ ]: roscd my_package
```



rospack list: Gives you a list with all of the packages in your ROS system.

rospack list | grep my_package: Filters, from all of the packages located in the ROS system, the package named `my_package` .

roscd my_package: Takes you to the location in the Hard Drive of the package, named `my_package` .

You can also see the package created and its contents by just opening it through the IDE (similar to [{Figure 1.1}](#))

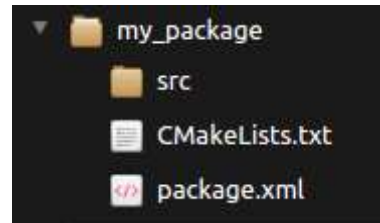


Fig.1.1 - IDE created package `my_package`

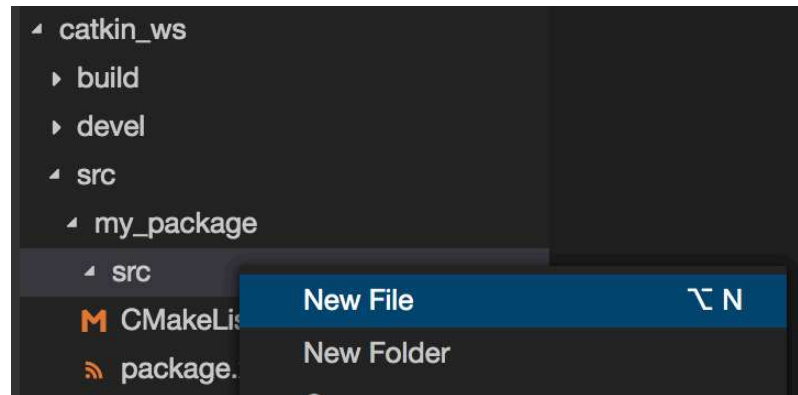
- End Example 2.5 -

2.5 My first ROS program

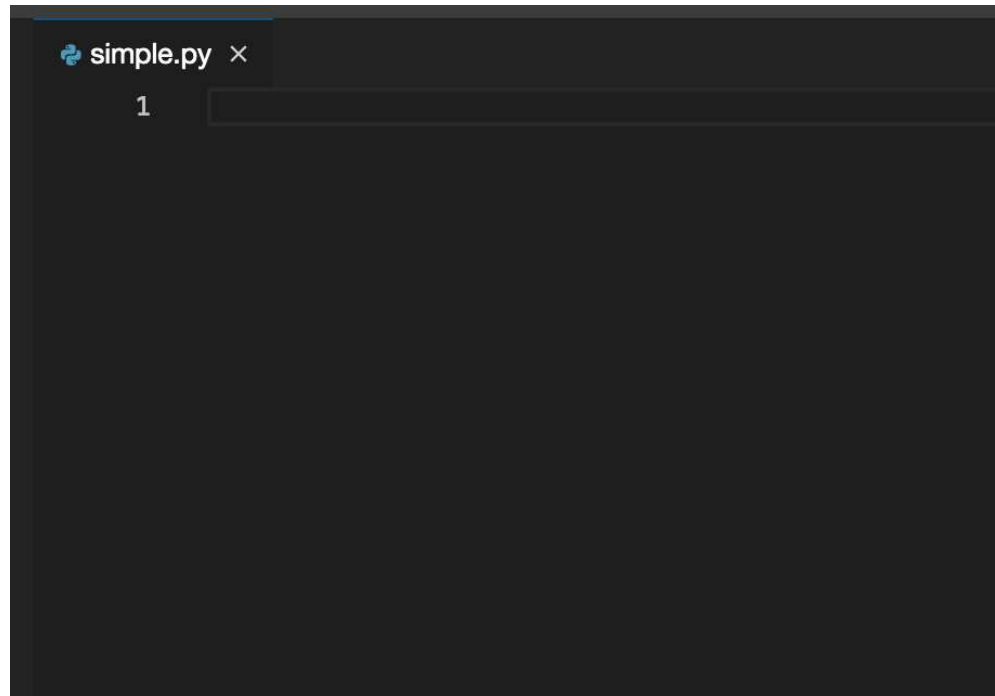
At this point, you should have your first package created... but now you need to do something with it! Let's do our first ROS program!

- Example 2.6 -

1- Create in the **src** directory in `my_package` a Python file that will be executed. For this exercise, just copy this simple python code [simple.py](#). You can create it directly by **RIGHT clicking** on the IDE on the `src` directory of your package, selecting **New File**, and writing the name of the file on the box that will appear.



A new Tab should have appeared on the IDE with empty content.



Then, copy the content of [simple.py](#) into the new file.

```
simple.py x
1  #!/usr/bin/env python
2
3  import rospy
4
5  rospy.init_node('ObiWan')
6  print("Help me Obi-Wan Kenobi, you're my only hope")
```

****Python Program: simple.py****

```
In [ ]: #!/usr/bin/env python

import rospy

rospy.init_node('ObiWan')
print("Help me Obi-Wan Kenobi, you're my only hope")
```

****NOTE**:** If you create your Python file from the terminal, it may happen that it's created without execution permissions. If this happens, ROS won't be able to find it. If this is your case, you can give execution permissions to the file by typing the next command: *****chmod +x name_of_the_file.py*****

****END Python Program: simple.py****

2- Create a `launch` directory inside the package named **my_package**.

► Execute in terminal #1

```
In [ ]: roscd my_package
```

```
In [ ]: mkdir launch
```



You can also create it through the IDE.

3- Create a new launch file inside the launch directory.

► Execute in terminal #1

```
In [ ]: touch launch/my_package_launch_file.launch
```



You can also create it through the IDE.

4- Fill this launch file as we've previously seen in this course ([Example 2.3](#)).

HINT: You can copy from the `turtlebot_teleop` package, the `keyboard_teleop.launch` file and modify it. If you do so, remove the param and remap tags and leave only the node tag, because you don't need those parameters.

The final launch file should be something similar to this [my_package_launch_file.launch](#).

 `my_package_launch_file.launch`

You should have something similar to this in your `my_package_launch_file.launch`:

NOTE: Keep in mind that in the example below, the Python file in the attribute **type** is named **simple.py**. So, if you have named your Python file with a different name, this will be different.

In []:

```
<launch>
  <!-- My Package launch file -->
  <node pkg="my_package" type="simple.py" name="ObiWan" output="screen">
  </node>
</launch>
```



5- Finally, execute the roslaunch command in the Terminal in order to launch your program.

► Execute in terminal #1

In []:

```
roslaunch my_package my_package_launch_file.launch
```



- End Example 2.6 -

- Expected Result for Example 2.6 -

You should see Leia's quote among the output of the roslaunch command.

📄 terminal #1 Output

In []:

```
... logging to /home/user/.ros/log/d29014ac-911c-11e6-b306-02f9ff83faab/roslaunch-ip-172-31-30-5-28204.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ip-172-31-30-5:40504/

SUMMARY
=====

PARAMETERS
* /rostdistro: kinetic
* /rosversion: 1.11.20

NODES
/
  ObiWan (my_package/simple.py)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[ObiWan-1]: started with pid [28228]
Help me Obi-Wan Kenobi, you're my only hope
[ObiWan-1] process has finished cleanly
log file: /home/user/.ros/log/d29014ac-911c-11e6-b306-02f9ff83faab/ObiWan-1*.log
all processes on machine have died, roslaunch will exit
shutting down processing monitor...
... shutting down processing monitor complete
done
```

- End Expected Result -

Sometimes ROS won't detect a new package when you have just created it, so you won't be able to do a roslaunch. In this case, you can force ROS to do a refresh of its package list with the command:

► Execute in terminal #1

In []: rospack profile



Code Explanation: simple.py

Although it is a very simple Python script, let's explain it line by line, to avoid any confusion:

In []:

```
#!/usr/bin/env python
# This line will ensure the interpreter used is the first one on your environment's $PATH. Every Python file
# to start with this line at the top.

import rospy # Import the rospy, which is a Python Library for ROS.

rospy.init_node('ObiWan') # Initiate a node called ObiWan

print("Help me Obi-Wan Kenobi, you're my only hope") # A simple Python print
```



END Code Explanation: simple.py

2.6 Common Issues

From our experience, we've seen that it is a common issue when working with Python scripts in this Course, that users get an error similar to this one:

```
core service [/rosout] found
ERROR: cannot launch node of type [test_pkg/test.py]: can't locate node [test.py] in package [test_pkg]
No processes to monitor
shutting down processing monitor
```

This error usually appears to users when they create a Python script from the Terminal. It happens because when created from the terminal, the Python scripts don't have execution permissions. You can check the permissions of a file using the following command, inside the directory where the file is located at:

► Execute in terminal #1

In []: `ls -la`



```
user:~/catkin_ws/src/test_pkg/src$ ls -la
total 12
drwxrwxr-x 2 user user 4096 Jan 15 20:13 .
drwxrwxr-x 4 user user 4096 Jan 15 20:15 ..
-rw-rw-r-- 1 user user   82 Jan 15 20:14 test.py
```

The first row in the left indicates the permissions of this file. In this case, we have **-rw-rw-r-**. So, you only have **read(r)** and **write(w)** permissions on this file, but not execution permissions (which are represented with an **x**).

To add execution permissions to a file, you can use the following command:

► Execute in terminal #1

In []: `chmod +x name_of_file.py`



Using this command, you will see that execution permissions are added to the file. Also, the file will appear now in green color.

```
user:~/catkin_ws/src/test_pkg/src$ ls -la
total 12
drwxrwxr-x 2 user user 4096 Jan 15 20:13 .
drwxrwxr-x 4 user user 4096 Jan 15 20:15 ..
-rwxrwxr-x 1 user user   82 Jan 15 20:14 test.py
```

Doing this, the error shown above will disappear.

2.7 ROS Nodes

You've initiated a node in the previous code but... what's a node? ROS nodes are basically programs made in ROS. The ROS command to see what nodes are actually running in a computer is:

```
In [ ]: rosnod list
```



- Example 2.7 -

Type this command in a new terminal and look for the node you've just initiated (ObiWan).

► Execute in terminal #1

```
In [ ]: rosnod list
```



You can't find it? I know you can't. That's because the node is killed when the Python program ends. Let's change that.

Update your Python file [simple.py](#) with the following code:


```
**Python Program: simple_loop.py**
```

```
In [ ]: #!/usr/bin/env python

import rospy

rospy.init_node("ObiWan")
rate = rospy.Rate(2)           # We create a Rate object of 2Hz
while not rospy.is_shutdown(): # Endless loop until Ctrl + C
    print("Help me Obi-Wan Kenobi, you're my only hope")
    rate.sleep()               # We sleep the needed time to maintain the Rate fixed above

# This program creates an endless loop that repeats itself 2 times per second (2Hz) until somebody presses Ct
# in the terminal
```



****END Python Program: simple_loop.py****

Launch your program again using the roslaunch command.

► Execute in terminal #1

```
In [ ]: roslaunch my_package my_package_launch_file.launch
```

Now try again in another Web terminal:

► Execute in terminal #2

```
In [ ]: rosnod list
```

Can you now see your node?

 terminal #2 Output

```
In [ ]: /ObiWan  
        /cmd_vel_mux  
        /gazebo  
        /mobile_base_nodelet_manager  
        /robot_state_publisher  
        /rosout
```



In order to see information about our node, we can use the next command:

```
In [ ]: rosnode info <node_name>
```



This command will show us information about all the connections that our Node has.

► Execute in terminal #2

```
In [ ]: rosnode info /ObiWan
```



📄 terminal #2 Output

In []:



```
-----  
Node [/ObiWan]  
Publications:  
  * /rosout [rosgraph_msgs/Log]  
  
Subscriptions:  
  * /clock [rosgraph_msgs/Clock]  
  
Services:  
  * /ObiWan/set_logger_level  
  * /ObiWan/get_loggers  
  
contacting node http://ip-172-31-30-5:58680/ ...  
Pid: 1215  
Connections:  
  * topic: /rosout  
    * to: /rosout  
    * direction: outbound  
    * transport: TCPROS  
  * topic: /clock  
    * to: /gazebo (http://ip-172-31-30-5:46415/)  
    * direction: inbound  
    * transport: TCPROS
```

For now, don't worry about the output of the command. You will understand more while going through the next chapters.

- End Example 2.7 -

2.8 Compile a package

When you create a package, you will usually need to compile it in order to make it work. There are different methods that can be used to compile your ROS packages. For this course we will present you the most common one.

2.8.1 catkin make

One of the most popular options used by ROS Developers to compile their packages is:

```
In [ ]: catkin_make
```



This command will compile your whole **src** directory, and **it needs to be issued in your *catkin_ws* directory in order to work. This is MANDATORY.** If you try to compile from another directory, it won't work.

- Example 2.8 -

Go to your *catkin_ws* directory and compile your source folder. You can do this by typing:

► Execute in terminal #1

```
In [ ]: cd ~/catkin_ws
```



```
In [ ]: catkin_make
```



After compiling, it's also very important to **source** your workspace. This will make sure that ROS will always get the latest changes done in your workspace.

► Execute in terminal #1

```
In [ ]: source devel/setup.bash
```



Sometimes (for example, in large projects) you will not want to compile all of your packages, but just the one(s) where you've made changes. You can do this with the following command:

```
In [ ]: catkin_make --only-pkg-with-deps <package_name>
```



This command will only compile the packages specified and its dependencies.

Try to compile your package named `my_package` with this command.

► Execute in terminal #1

```
In [ ]: catkin_make --only-pkg-with-deps my_package
```



```
In [ ]: source devel/setup.bash
```



- End Example 2.8 -

2.9 Parameter Server

A Parameter Server is a **dictionary** that ROS uses to store parameters. These parameters can be used by nodes at runtime and are normally used for static data, such as configuration parameters.

- End of Notes -

To get a list of these parameters, you can type:

► Execute in terminal #1


```
In [ ]: rosparam list
```



To get a value of a particular parameter, you can type:

```
In [ ]: rosparam get <parameter_name>
```



And to set a value to a parameter, you can type:

```
In [ ]: rosparam set <parameter_name> <value>
```



- Example 2.10 -

To get the value of the '/camera/imager_rate' parameter, and change it to '4.0,' you will have to do the following:

► Execute in terminal #1

```
In [ ]: rosparam get /camera/imager_rate
```



```
In [ ]: rosparam set /camera/imager_rate 4.0
```



```
In [ ]: rosparam get /camera/imager_rate
```



- End Example 2.10 -

You can create and delete new parameters for your own use, but do not worry about this right now. You will learn more about this in more advanced tutorials

2.10 Roscore

In order to have all of this working, we need to have a roscore running. The roscore is the **main process** that manages all of the ROS system. You always need to have a roscore running in order to work with ROS. The command that launches a roscore is:

In []: `roscore`

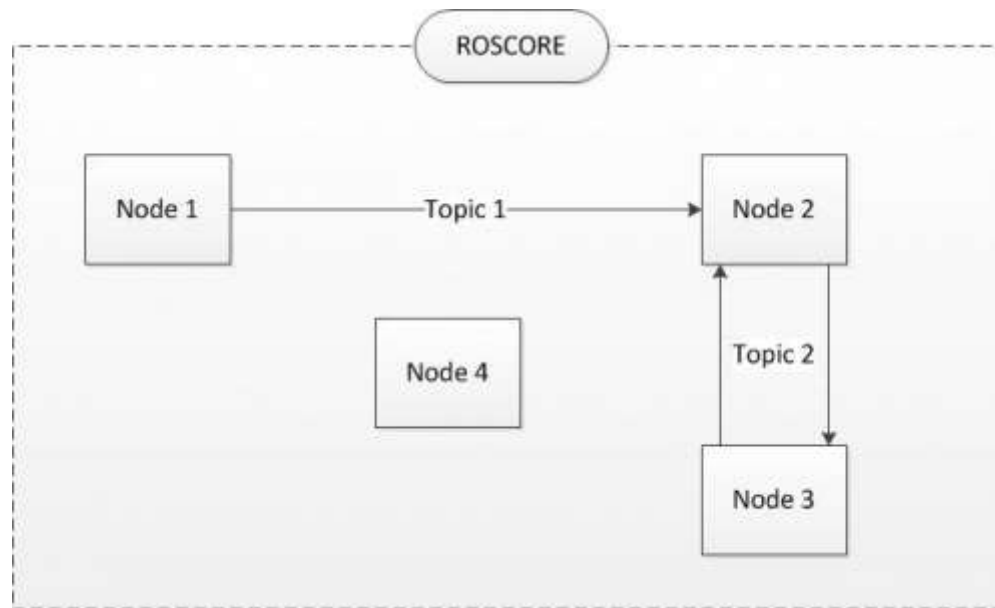


Fig.1.2 - ROS Core Diagram

****NOTE**:** The platform you are using for this course automatically launches a roscore for you when you enter a course, so you don't need to launch one.

2.11 Environment Variables

ROS uses a set of Linux system environment variables in order to work properly. You can check these variables by typing:

In []: export | grep ROS



****NOTE 1****: Depending on your computer, it could happen that you can't type the ****|**** symbol directly in your Terminal. If that's the case, just ****copy/paste**** the command by ****RIGHT-CLICKING**** on the Terminal and select ****Paste from Browser****. This feature will allow you to write anything on your Terminal, no matter what your computer configuration is.

In []: declare -x ROSLISP_PACKAGE_DIRECTORIES="/home/user/catkin_ws/devel/share/common-lisp"
declare -x ROS_DISTRO="noetic"
declare -x ROS_ETC_DIR="/opt/ros/noetic/etc/ros"
declare -x ROS_MASTER_URI="http://localhost:11311"
declare -x ROS_PACKAGE_PATH="/home/user/catkin_ws/src:/opt/ros/noetic/share:/opt/ros/noetic/stacks"
declare -x ROS_ROOT="/opt/ros/noetic/share/ros"



The most important variables are the **ROS_MASTER_URI** and the **ROS_PACKAGE_PATH**.

In []: ROS_MASTER_URI -> Contains the url where the ROS Core **is** being executed. Usually, your own computer (localhost)
ROS_PACKAGE_PATH -> Contains the paths **in** your Hard Drive where ROS has packages **in** it.



****NOTE 2****: At the platform you are using for this course, we have created an alias to display the environment variables of ROS. This alias is ****rosv****. By typing this on your terminal, you'll get a list of ROS environment variables. It is important that you know that this is ****not an official ROS command****, so you can only use it while working on this platform.

2.12 So now... what is ROS?

ROS is basically the framework that allows us to do all that we showed along this chapter. It provides the background to manage all these processes and communications between them... and much, much more!! In this tutorial you've just scratched the surface of ROS, the basic concepts. ROS is an extremely powerful tool. If you dive into our courses you'll learn much more about ROS and you'll find yourself able to do almost anything with your robots!

2.13 Additional material to learn more:

ROS Packages: <http://wiki.ros.org/Packages> (<http://wiki.ros.org/Packages>)

Ros Nodes: <http://wiki.ros.org/Nodes> (<http://wiki.ros.org/Nodes>)

Parameter Server: <http://wiki.ros.org/Parameter%20Server> (<http://wiki.ros.org/Parameter%20Server>)

Roscore: <http://wiki.ros.org/roscore> (<http://wiki.ros.org/roscore>)

ROS Environment Variables: <http://wiki.ros.org/ROS/EnvironmentVariables> (<http://wiki.ros.org/ROS/EnvironmentVariables>)



English
proofread