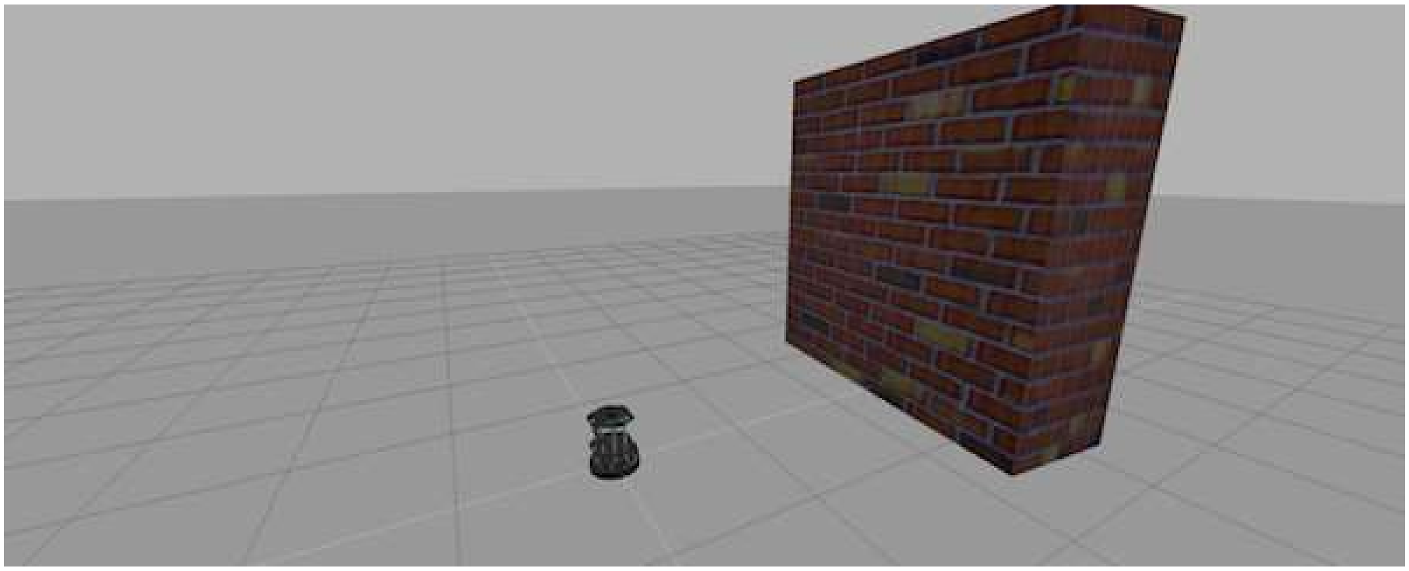
ROS Basics in 5 days (C++)

Unit 2 ROS Basics



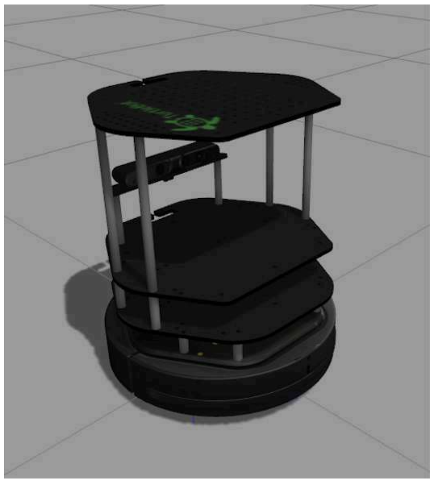
- Summary -

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 (C++ based)
- Basic ROS concepts: Nodes, Parameter Server, Environment Variables, Roscore



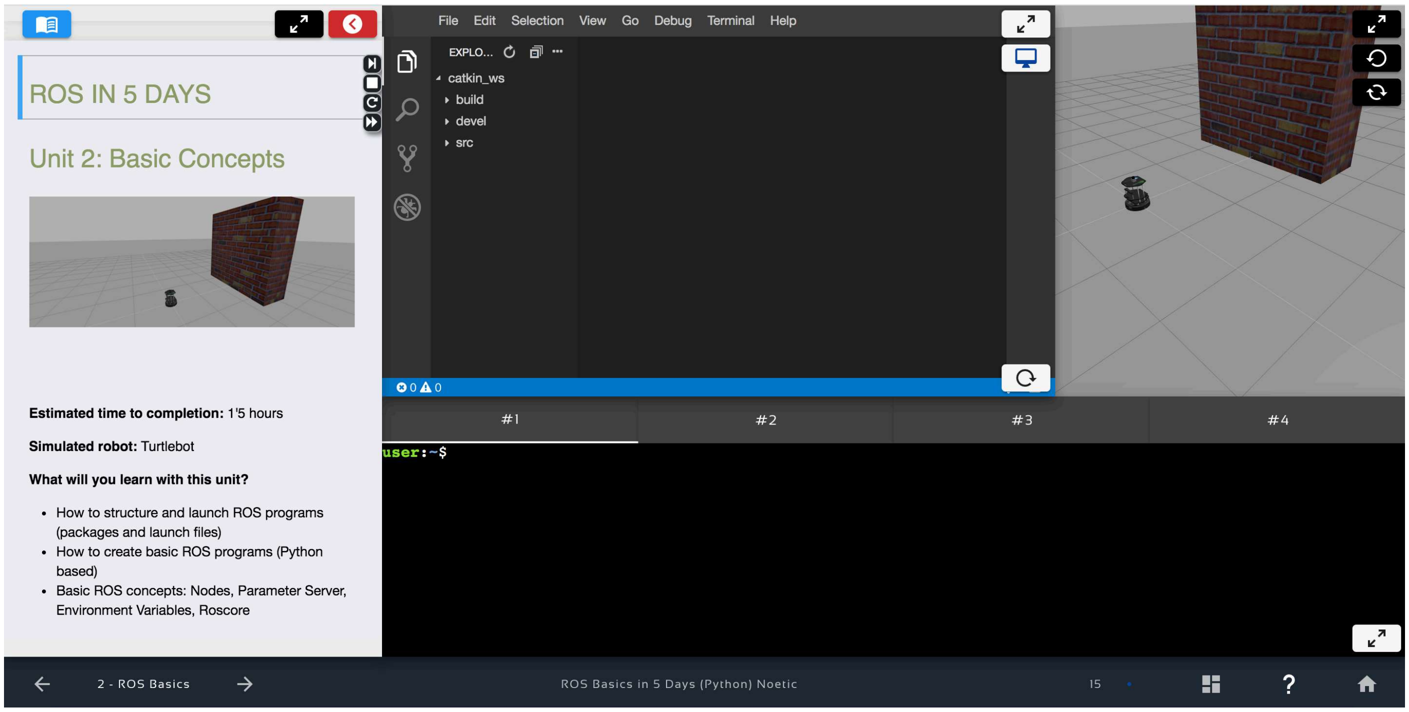
- End of Summary -

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.



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

```
user:~$
```

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 WebShell number #1:

▶ Execute in Shell #1

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

Shell #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 WebShell 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 z	**Increase / Decrease Speed**

Try it!! When you're done, you can **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 of 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>
```

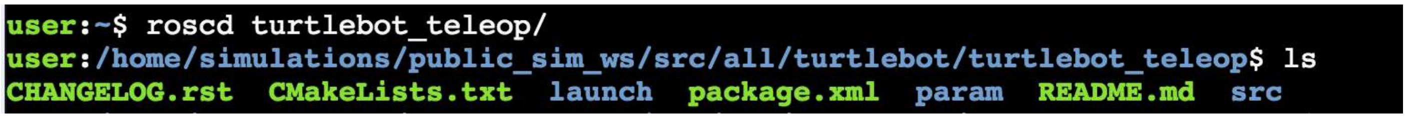
It will take you to the path where the package *package_name* is located.

- Example 2.2 -

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

▶ Execute in Shell #1

In [ ]:
```
roscd turtlebot_teleop
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 of 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 Shell #1

In [ ]:
```
roscd turtlebot_teleop
cd launch
cat keyboard_teleop_cpp.launch
```

Shell #1 Output

In [ ]:
```
<launch>
  <!-- turtlebot_teleop_key already has its own built in velocity smoother -->
  <node pkg="turtlebot_teleop" type="turtlebot_teleop_key" name="turtlebot_tel
    <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_mux/input/tele
  </node>
</launch>
```

In the launch file, you have some extra tags for setting parameters and remaps. For now, don't worry about those tags and focus on the node tag.

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="cpp_executable_name"** # Name of the cpp executable file that we want to execute
3. **name="node_name"** # Name of the ROS node that will launch our C++ file
4. **output="type_of_output"** # Through which channel you will print the output of the program

- End of Example 2.3 -

2.4 Create a 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 webshell.

In order to do this, type ***roscd*** in the *shell*. 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 Shell #1

In [ ]:
```
roscd
cd ..
pwd
```

Shell #1 Output

In [ ]:
```
user ~ $ pwd
/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 package, you have to be in this directory (**catkin_ws/src**).Type in your web shell **cd src** in order to move to the source directory.

▶ Execute in Shell #1

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

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

▶ Execute in Shell #1

In [ ]:
```
catkin_create_pkg my_package roscpp
```

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_dependecies>
```

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 of 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 Shell #1

In [ ]:
```
rospack list
rospack list | grep my_package
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 2.1})

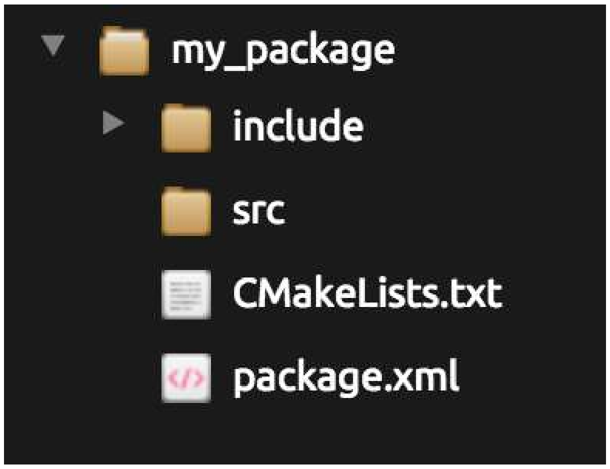


Fig. 2.1 - IDE created package my_package

- End of Example 2.5 -

2.5 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 2 most common ones:

2.5.1 catkin make

On of the most popular options used by ROS Developers to compile their packages is the next one:

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 2.6 -

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

▶ Execute in Shell #1

In [ ]:
```
roscd; cd ..
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 Shell #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 Shell #1

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

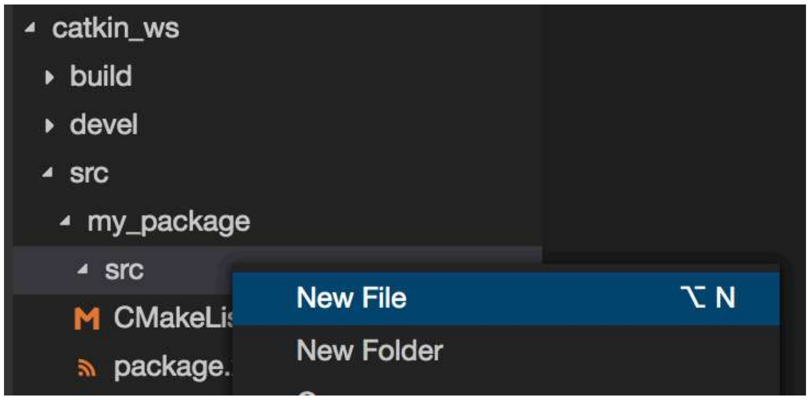
- End of Example 2.6 -

2.6 My first ROS program

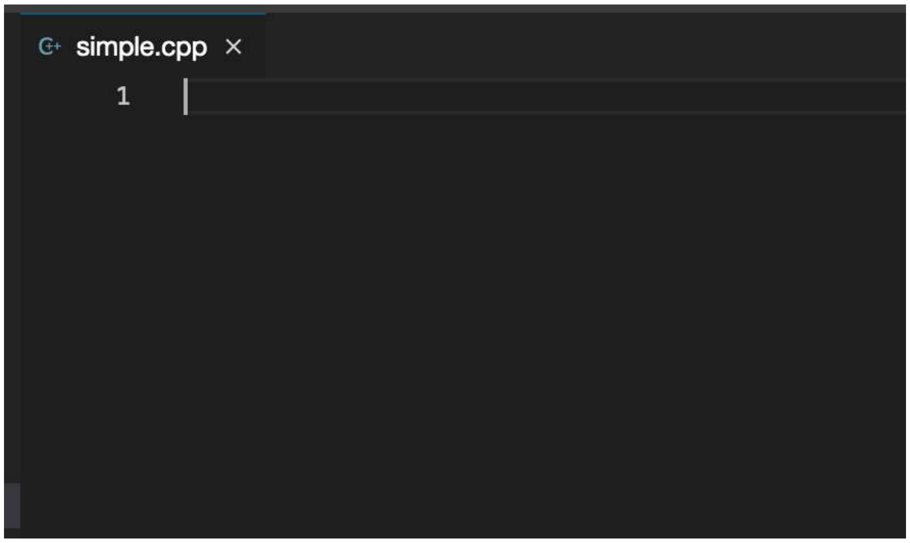
At this point, you should have your first package created... but now you need to do something with it! Let's do our first ROS program!

- Example 2.8 -

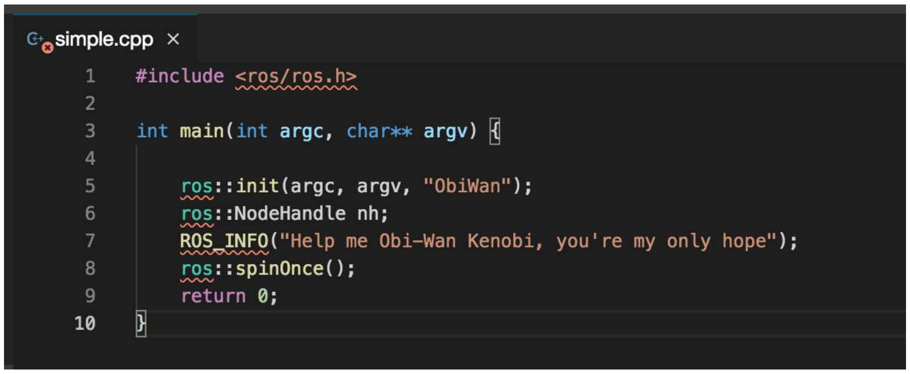
1- Create in the **src** directory in my_package a C++ file that will be executed. For this exercise, just copy this simple C++ code simple.cpp. You can create it directly by **RIGHT clicking** on the IDE on the src directory of your package, selecting New File, and writing the name of the file on the box that will appear.



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



Then, copy the content of simple.cpp into the new file.



```cpp
#include <ros/ros.h>

int main(int argc, char** argv) {

    ros::init(argc, argv, "ObiWan");
    ros::NodeHandle nh;
    ROS_INFO("Help me Obi-Wan Kenobi, you're my only hope");
    ros::spinOnce();
    return 0;
}
```

2- Create a ***launch*** directory inside the package named **my_package** {Example 2.4}.

▶ Execute in Shell #1

In [ ]:
```
roscd my_package
mkdir launch
```

You can also create it through the IDE.

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

▶ Execute in Shell #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.</font>

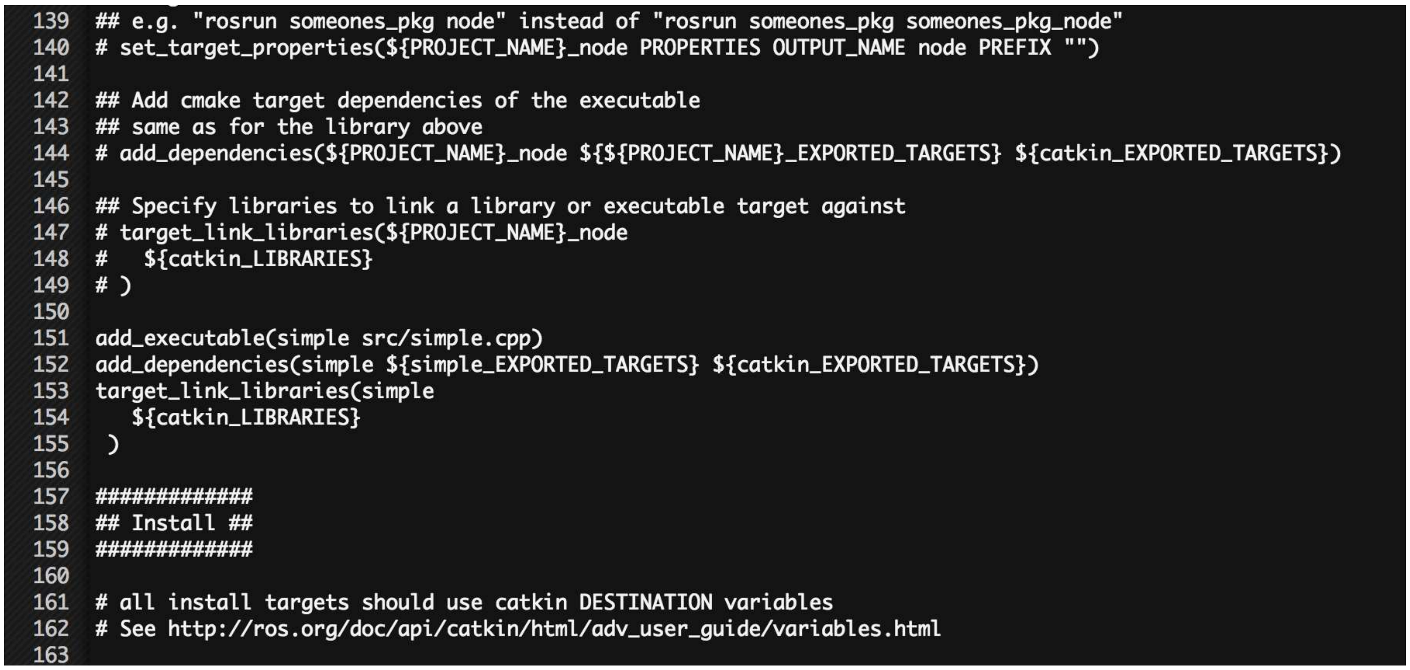
The final launch should be something similar to this: my_package_launch_file.launch

5- Modify the **CMakeLists.txt** file in order to generate an executable from the C++ file you have just created.

Note: This is something that is **required when working in ROS with C++**. When you finish this Exercise, you'll learn more about this subject. For now, just follow the instructions below.

In the **Build** section of your **CMakeLists.txt** file, add the following lines:

In [ ]:
```
add_executable(simple src/simple.cpp)
add_dependencies(simple ${simple_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(simple
   ${catkin_LIBRARIES}
 )
```



```
## e.g. "rosrun someones_pkg node" instead of "rosrun someones_pkg someones_pkg_node"
# set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")

## Add cmake target dependencies of the executable
## same as for the library above
# add_dependencies(${PROJECT_NAME}_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})

## Specify libraries to link a library or executable target against
# target_link_libraries(${PROJECT_NAME}_node
#   ${catkin_LIBRARIES}
# )

add_executable(simple src/simple.cpp)
add_dependencies(simple ${simple_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(simple
   ${catkin_LIBRARIES}
 )

############
## Install ##
############

# all install targets should use catkin DESTINATION variables
# See http://ros.org/doc/api/catkin/html/adv_user_guide/variables.html
```

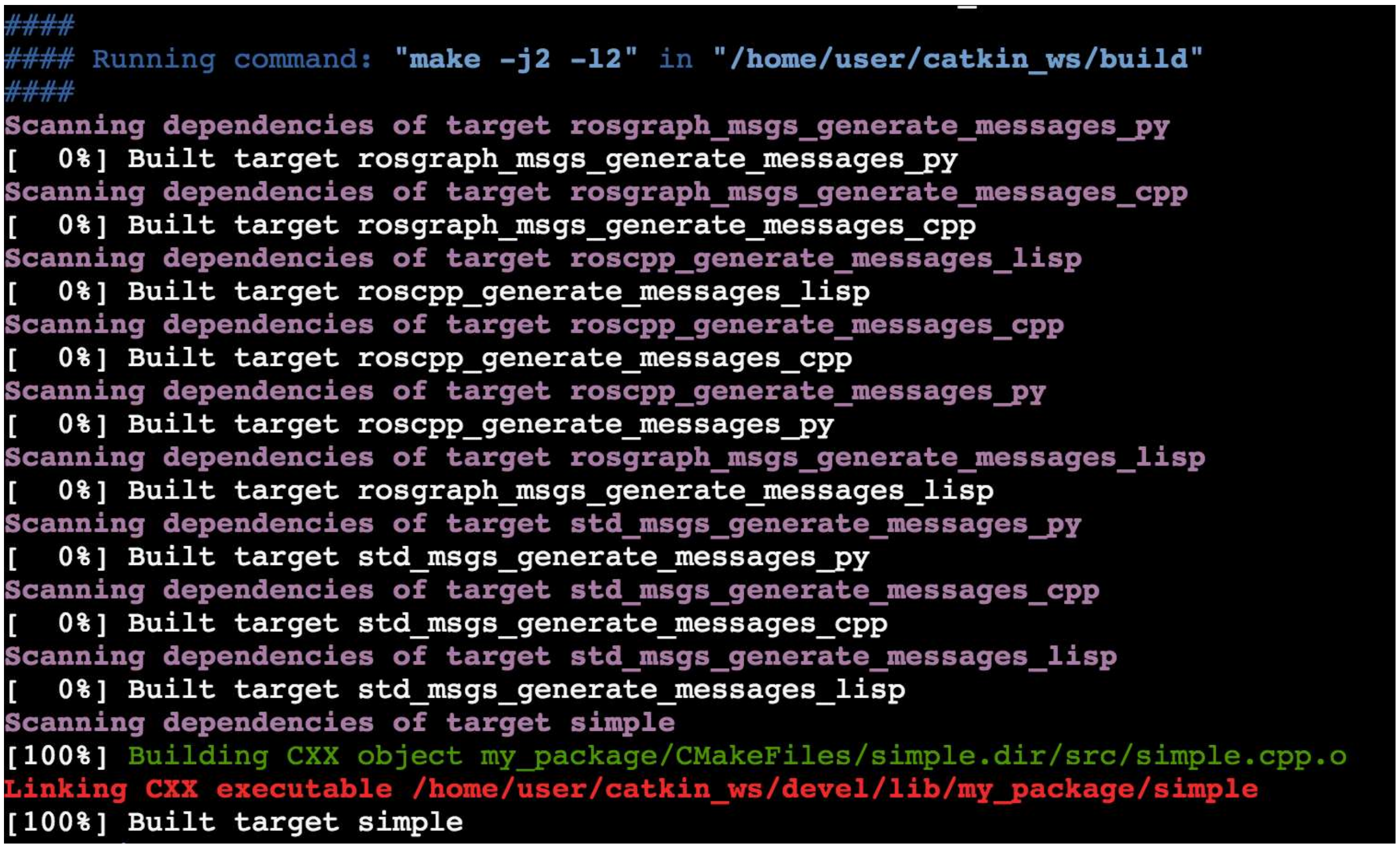
HINT: If you have a look at the file, you'll see that those lines are already in the file, but they are commented. If you feel so, you can uncomment those lines and modify them like the ones we provide above, instead of simply adding them to the end of the section.</font>

6- Compile your package as explained previously.

▶ Execute in Shell #1

In [ ]:
```
roscd;
cd ..;
catkin_make
```

If everything goes fine, you should get something like this as output:



```
####
#### Running command: "make -j2 -l2" in "/home/user/catkin_ws/build"
####
Scanning dependencies of target rosgraph_msgs_generate_messages_py
[  0%] Built target rosgraph_msgs_generate_messages_py
Scanning dependencies of target rosgraph_msgs_generate_messages_cpp
[  0%] Built target rosgraph_msgs_generate_messages_cpp
Scanning dependencies of target roscpp_generate_messages_lisp
[  0%] Built target roscpp_generate_messages_lisp
Scanning dependencies of target roscpp_generate_messages_cpp
[  0%] Built target roscpp_generate_messages_cpp
Scanning dependencies of target roscpp_generate_messages_py
[  0%] Built target roscpp_generate_messages_py
Scanning dependencies of target rosgraph_msgs_generate_messages_lisp
[  0%] Built target rosgraph_msgs_generate_messages_lisp
Scanning dependencies of target std_msgs_generate_messages_py
[  0%] Built target std_msgs_generate_messages_py
Scanning dependencies of target std_msgs_generate_messages_cpp
[  0%] Built target std_msgs_generate_messages_cpp
Scanning dependencies of target std_msgs_generate_messages_lisp
[  0%] Built target std_msgs_generate_messages_lisp
Scanning dependencies of target simple
[100%] Building CXX object my_package/CMakeFiles/simple.dir/src/simple.cpp.o
Linking CXX executable /home/user/catkin_ws/devel/lib/my_package/simple
[100%] Built target simple
```

7- Finally, execute the roslaunch command in the WebShell in order to launch your program.

▶ Execute in Shell #1

In [ ]:

```
roslaunch my_package my_package_launch_file.launch
```

- End of Example 2.8 -

- Expected Result for Example 2.8 -

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

Shell #1 Output

In [ ]:
```
user catkin_ws $ roslaunch my_package my_package_launch_file.launch
... logging to /home/user/.ros/log/d29014ac-911c-11e6-b306-02f9ff83faab/roslau
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
 * /rosdistro: indigo
 * /rosversion: 1.11.20

NODES
  /
    ObiWan (my_package/simple)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[ObiWan-1]: started with pid [28228]
[ INFO] [1515028076.948011193]: 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*.l
all processes on machine have died, roslaunch will exit
shutting down processing monitor...
... shutting down processing monitor complete
done
```

- End of 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 Shell #1

In [ ]:
```
rospack profile
```

C++ Program {2.1a-cpp}: simple.cpp

In [ ]:
```
#include <ros/ros.h>

int main(int argc, char** argv) {

    ros::init(argc, argv, "ObiWan");
    ros::NodeHandle nh;
    ROS_INFO("Help me Obi-Wan Kenobi, you're my only hope");
    ros::spinOnce();
    return 0;
}
```

You may be wondering what this whole code means, right? Well, let's explain it line by line:

In [ ]:
```
#include <ros/ros.h>
// Here we are including all the headers necessary to use the most common publ
// Always we create a new C++ file, we will need to add this include.

int main(int argc, char** argv) { // We start the main C++ program

    ros::init(argc, argv, "ObiWan"); // We initiate a ROS node called ObiWan
    ros::NodeHandle nh; // We create a handler for the node. This handler will
    ROS_INFO("Help me Obi-Wan Kenobi, you're my only hope"); // This is the sa
    ros::spinOnce(); // Calling ros::spinOnce() here is not necessary for this
                     // measure.
    return 0; // We end our program
}
```

NOTE: If you create your C++ file from the shell, 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.cpp***

END C++ Program {2.1-cpp}: simple.py

Launch File {2.1-l}: 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 C++ executable name in the attribute **type** is named **simple**. So, if you have named your C++ executable with a different name, this will be different.

In [ ]:
```
<launch>
    <!-- My Package launch file -->
    <node pkg="my_package" type="simple" name="ObiWan"  output="screen">
    </node>
</launch>
```

END Launch File {2.1-l}: my_package_launch_file.launch

2.7 Modifying the CMakeLists.txt file

When coding with C++, it will be necessary to create binaries(executables) of your programs in order to be able to execute them. For that, you will need to modify the **CMakeLists.txt** file of your package, in order to indicate that you want to create an executable of your C++ file.

To do this, you need to add some lines into your **CMakeLists.txt** file. In fact, these lines are already in the file, but they are commented. You can also find them, and uncomment them. Whatever you want.

In the previous Exercise, you had the following lines:

In [ ]:
```
add_executable(simple src/simple.cpp)
add_dependencies(simple ${simple_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(simple
   ${catkin_LIBRARIES}
 )
```

But... what do this lines of code exactly do? Well, basically they do the following:

In [ ]:
```
add_executable(simple src/simple.cpp)
```

This line **generates an executable from the simple.cpp file**, which is in the src folder of your package. This executable **will be placed by default into the package directory of your devel space**, which is located by default at ~/catkin_ws/devel/lib/.

In [ ]:
```
add_dependencies(simple ${simple_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
```

This line adds all the cmake target dependencies of the executable. It's used basically to allow CMake to correctly compile a package, making sure all the dependencies are in place.

In [ ]:
```
target_link_libraries(simple
   ${catkin_LIBRARIES}
 )
```

This line specifies the libraries to use when linking a given target. For this case, it indicates to use the catkin libraries when linking to the executable you have created.

If you are interested, you can read more about the **CMakeLists.txt** file here:
http://wiki.ros.org/catkin/CMakeLists.txt (http://wiki.ros.org/catkin/CMakeLists.txt)

2.8 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 [ ]:
```
rosnode list
```

- Example 2.9 -

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

▶ Execute in Shell #1

In [ ]:
```
rosnode list
```

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

Update your C++ file simple.cpp with the following code:

C++ Program {2.1b-cpp}: simple_loop.cpp

In [ ]:
```
#include <ros/ros.h>

int main(int argc, char** argv) {

    ros::init(argc, argv, "ObiWan");
    ros::NodeHandle nh;
    ros::Rate loop_rate(2); // We create a Rate object of 2Hz

    while (ros::ok()) // Endless loop until Ctrl + C
    {
        ROS_INFO("Help me Obi-Wan Kenobi, you're my only hope");
        ros::spinOnce();
        loop_rate.sleep(); // We sleep the needed time to maintain the Rate fi
    }

    return 0;
}

// This program creates an endless loop that repeats itself 2 times per seconc
// in the Shell
```

END C++ Program {2.1b-cpp}: simple_loop.cpp

Launch your program again using the roslaunch command.

▶ **Execute in Shell #1**

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

Now try again in another Web Shell:

▶ **Execute in Shell #2**

In [ ]:
```
rosnode list
```

Can you now see your node?

Shell #2 Output

In [ ]:
```
user ~ $ rosnode list
/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 /ObiWan
```

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

▶ **Execute in Shell #2**

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

Shell #2 Output

In [ ]:
```
user ~ $ rosnode info /ObiWan
--------------------------------------------------------------------------------
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 tutorial.

- End of Example 2.9 -

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.

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

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 Shell #1

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

- End of 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
```

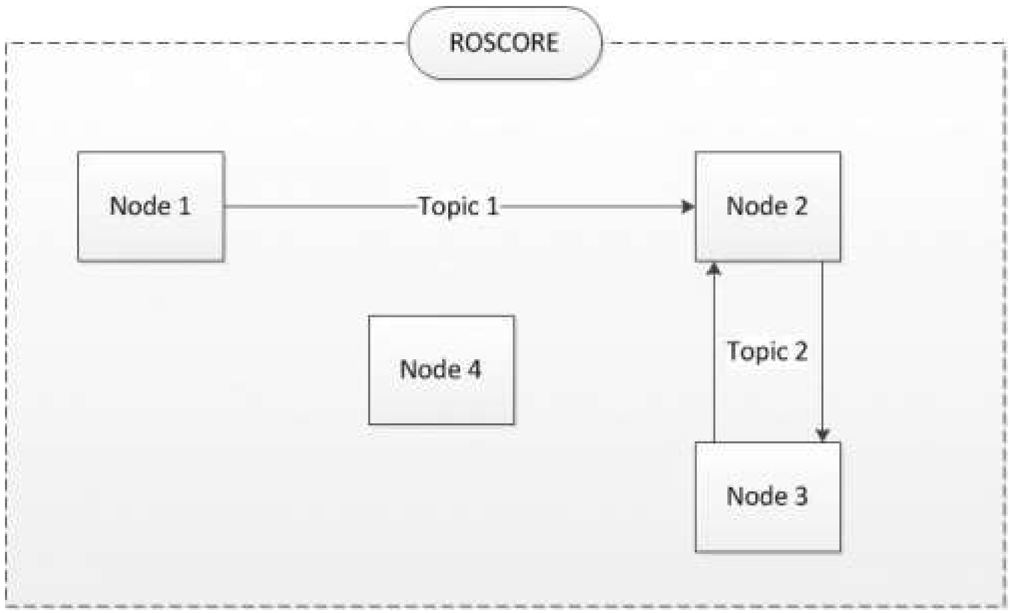


Fig. 2.2 - ROS Core Diagram

NOTE: At the platform you are using for this course, when you enter a course it automatically launches a roscore for you, 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 webshell. If that's the case, just **copy/paste** the command by **RIGHT-CLICKING** on the WebShell and select ***Paste from Browser***. This feature will allow you to write anything on your webshell, no matter what your computer configuration is.

In [ ]:
```
user ~ $ export | grep ROS
declare -x ROSLISP_PACKAGE_DIRECTORIES="/home/user/catkin_ws/devel/share/commc
declare -x ROS_DISTRO="indigo"
declare -x ROS_ETC_DIR="/opt/ros/indigo/etc/ros"
declare -x ROS_MASTER_URI="http://localhost:11311"
declare -x ROS_PACKAGE_PATH="/home/user/catkin_ws/src:/opt/ros/indigo/share:/c
declare -x ROS_ROOT="/opt/ros/indigo/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. Usual
ROS_PACKAGE_PATH -> Contains the paths in your Hard Drive where ROS has packag
```

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 **rosenv**. By typing this on your shell, 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