Note on ROS:

I-/ Intro:

1)
Execute the following command in **WebShell number #1** in order to start moving the Kobuki robot.

WebShells are like Linux shells, but on the web. They are located, by default, at the bottom right section of the screen. You can type Linux commands there.

roslaunch publisher_example move.launch

Whenever you want to stop moving the robot, and in order to stop the program, just press Ctrl + C in the WebShell.
NOTE: You will notice that, even after stopping the program, the robot will still keep moving. In order to stop it, you will have to execute another command.
Now select **WebShell #2** and execute the following command in order to stop the Kobuki robot.

```
roslaunch publisher_example stop.launch
```

Note 1: Even after terminating the command with Ctrl+C, the robot will still keep moving. This is because the robot will keep listening to the last message that you published on the topic. You will understand what this means later.

2)

Execute the following command in WebShell number #1 in order to start the service.

```
roslaunch service_demo service_launch.launch
```

Execute the following command in WebShell number #2 in order to call the service.

```
rosservice call /service_demo "{}"
```

Note 1: The service must be up and running before you can call it. So make sure that you have launched the service before calling it.
Note 2: Bear in mind that your robot will start moving from the point you stopped it in the previous example. So, it may not coincide with the gif shown.

3)

Execute the following command in WebShell number #1 in order to start the action.

```
roslaunch action_demo action_launch.launch
```

Execute the following command in WebShell number #2 in order to "call" the action.

```
roslaunch action_demo_client client_launch.launch
```

Note 1: Make sure you've started the Action Server by executing the first command. Otherwise, you won't be able to call it.
Note 2: Bear in mind that your robot will start moving from the point you stopped it at in the previous example. So, it may not coincide with the gif shown.

II-/ Basics:

1)

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

```
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
```

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

```
roslaunch <package_name> <launch_file>
```

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:

```
roscd <package_name>
```

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.

And... what's a launch file?

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

roscd turtlebot_teleop
cd launch
cat keyboard_teleop.launch

```
<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>
```

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

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***.

```
roscd
cd ..
pwd

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 new package, you have to be in this directory (**catkin_ws/src**). Type in your WebShell ***cd src*** in order to move to the source directory.

```
cd src
```

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

```
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:

```
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.

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:

```
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*.

My first ROS program

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. Finally, press **Ctrl-S** to save your file with the changes. The Tab in the IDE will go from *Green* to *no color* (see pictures below).

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

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

```
roscd my_package
mkdir launch
```

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

```
touch launch/my_package_launch_file.launch
```

4- Fill this launch file as we've previously seen in this course {Example 2.3}.

```
<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 WebShell in order to launch your program.

```
roslaunch my_package my_package_launch_file.launch
```

Expected result:

```
user catkin_ws $ roslaunch my_package my_package_launch_file.launch
... 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
 * /rosdistro: 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
```

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:

```
rospack profile
```

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:

This error usually appears to users when they create a Python script from the WebShell. It happens because when created from the shell, 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:

```
ls -la
```

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:
chmod +x name_of_file.py

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:

```
rosnode 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:

```
#! /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 Ctrl + C
```

```
# in the Shell
```

```
rosnode list
```

expected output:

```
user ~ $ rosnode list
/ObiWan
/cmd_vel_mux
/gazebo
/mobile_base_nodelet_manager
/robot_state_publisher
/rosou
```

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

rosnode info /ObiWan

```
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
```

Compile a package

When you create a package, you will usually need to compile it in order to make it work. The command used by ROS to compile is the next one:

```
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.cd

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

```
roscd; cd ..
catkin_make
```

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:

```
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.

```
catkin_make --only-pkg-with-deps my_package
```

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:

```
rosparam list
```

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

```
rosparam get <parameter_name>
```

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

```
rosparam set <parameter_name> <value>
```

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

```
rosparam get /camera/imager_rate
rosparam set /camera/imager_rate 4.0
rosparam get /camera/imager_rate
```

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:

```
roscore
```

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.

Environment Variables

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

```
export | grep ROS
```

```
ser ~ $ export | grep ROS
declare -x
ROSLISP_PACKAGE_DIRECTORIES="/home/user/catkin_ws/devel/share/common-lisp"
declare -x ROS_DISTRO="kinetic"
declare -x ROS_ETC_DIR="/opt/ros/kinetic/etc/ros"
declare -x ROS_MASTER_URI="http://localhost:11311"
declare -x
ROS_PACKAGE_PATH="/home/user/catkin_ws/src:/opt/ros/kinetic/share:/opt/ros/k
inetic/stacks"
declare -x ROS_ROOT="/opt/ros/kinetic/share/ros"
```

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

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.

III-/ Topics - Publishers:

execute:

```
#! /usr/bin/env python

import rospy                                    # Import the Python library
for ROS
from std_msgs.msg import Int32                  # Import the Int32 message
from the std_msgs package

rospy.init_node('topic_publisher')              # Initiate a Node named
'topic_publisher'
pub = rospy.Publisher('/counter', Int32, queue_size=1)
                                                # Create a Publisher object,
that will publish on the /counter topic
                                                # messages of type Int32

rate = rospy.Rate(2)                            # Set a publish rate of 2 Hz
count = Int32()                                 # Create a var of type Int32
count.data = 0                                  # Initialize 'count' variable

while not rospy.is_shutdown():                  # Create a loop that will go
until someone stops the program execution
  pub.publish(count)                            # Publish the message within
the 'count' variable
  count.data += 1                               # Increment 'count' variable
  rate.sleep()                                  # Make sure the publish rate
maintains at 2 Hz
```

You have just created a topic named **/counter**, and published through it as an integer that increases indefinitely. Let's check some things.

A topic is like a pipe. **Nodes use topics to publish information for other nodes** so that they can communicate.

You can find out, at any time, the number of topics in the system by doing a ***rostopic list***. You can also check for a specific topic.

On your webshell, type ***rostopic list*** and check for a topic named '/counter'.

```
rostopic list | grep  '/counter'
```

```
user ~ $ rostopic list | grep '/counter'
/counter
```

Here, you have just listed all of the topics running right now and filtered with the **grep** command the ones that contain the word */counter*. If it appears, then the topic is running as it should.

You can request information about a topic by doing ***rostopic info <name_of_topic>***.

```
rostopic info /counter
```

```
user ~ $ rostopic info /counter
Type: std_msgs/Int32

Publishers:
 * /topic_publisher (http://ip-172-31-16-133:47971/)

Subscribers: None
```

The output indicates the type of information published (**std_msgs/Int32**), the node that is publishing this information (**/topic_publisher**), and if there is a node listening to that info (None in this case).
Now, type ***rostopic echo /counter*** and check the output of the topic in realtime.

```
rostopic echo /counter
```

So basically, what this code does is to **initiate a node and create a publisher that keeps publishing into the '/counter' topic a sequence of consecutive integers**. Summarizing:

A publisher is a node that keeps publishing a message into a topic. So now... what's a topic?

A topic is a channel that acts as a pipe, where other ROS nodes can either publish or read information. Let's now see some commands related to topics (some of them you've already used).

To **get a list of available topics** in a ROS system, you have to use the next command:

```
rostopic list
```

To **read the information that is being published in a topic,** use the next command:

```
rostopic echo <topic_name>
```

This command will start printing all of the information that is being published into the topic, which sometimes (ie: when there's a massive amount of information, or when messages have a very large structure) can be annoying. In this case, you can **read just the last message published into a topic** with the next command:

```
rostopic echo <topic_name> -n1
```

To **get information about a certain topic,** use the next command:

```
rostopic info <topic_name>
```

Finally, you can check the different options that *rostopic* command has by using the next command:

```
rostopic -h
```

Messages

As you may have noticed, topics handle information through messages. There are many different types of messages.

In the case of the code you executed before, the message type was an **std_msgs/Int32**, but ROS provides a lot of different messages. You can even create your own messages, but it is recommended to use ROS default messages when its possible.

Messages are defined in **.msg** files, which are located inside a **msg** directory of a package.
To **get information about a message,** you use the next command:

```
rosmsg show <message>
```

- Create a package with a launch file that launches the code simple_topic_publisher.py.
- Modify the code you used previously to publish data to the /cmd_vel topic.
- Launch the program and check that the robot moves.

Data for Excercice 2.1

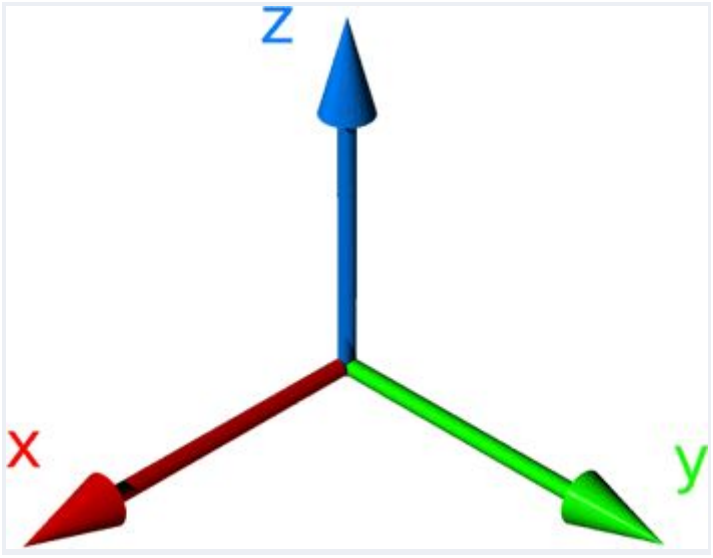
1.- The /cmd_vel topic is the topic used to move the robot. Do a ***rostopic info /cmd_vel*** in order to get information about this topic, and identify the message it uses. You have to modify the code to use that message.

2.- In order to fill the Twist message, you need to create an instance of the message. In Python, this is done like this: **var = Twist()**

3.- In order to know the structure of the Twist messages, you need to use the ***rosmsg show***

command, with the type of the message used by the topic **/cmd_vel**.

4.- In this case, the robot uses a differential drive plugin to move. That is, the robot can only move linearly in the **x** axis, or rotationaly in the angular **z** axis. This means that the only values that you need to fill in the Twist message are the linear **x** and the angular **z**.



5.- The magnitudes of the Twist message are in m/s, so it is recommended to use values between 0 and 1. For example, 0'5 m/s.

IV TOPIC - Subscriber:

You've learned that a topic is a channel where nodes can either write or read information. You've also seen that you can write into a topic using a publisher, so you may be thinking that there should also be some kind of similar tool to read information from a topic. And you're right! That's called a subscriber. **A subscriber is a node that reads information from a topic**.

execute:

```
#! /usr/bin/env python

import rospy
from std_msgs.msg import Int32

def callback(msg):                                      # Define a function
called 'callback' that receives a parameter
                                                        # named 'msg'

    print msg.data                                      # Print the value
'data' inside the 'msg' parameter
```

```
rospy.init_node('topic_subscriber')                    # Initiate a Node
called 'topic_subscriber'

sub = rospy.Subscriber('/counter', Int32, callback)    # Create a Subscriber
object that will listen to the /counter
                                                       # topic and will cal
the 'callback' function each time it reads
                                                       # something from the
topic
rospy.spin()                                           # Create a loop that
will keep the program in execution
```

```
rostopic echo /counter
```

```
user ~ $ rostopic echo /counter
WARNING: no messages received and simulated time is active.
Is /clock being published?
```

This means that **nobody is publishing into the /counter topic**, so there's no information to be read. Let's then publish something into the topic and see what happens. For that, let's introduce a new command:

```
rostopic pub <topic_name> <message_type> <value>
```

This command will publish the message you specify with the value you specify, in the topic you specify.

```
rostopic pub /counter std_msgs/Int32 5
```

```
user ~ $ rostopic echo /counter
WARNING: no messages received and simulated time is active.
Is /clock being published?
data:
5
---
```

This means that the value you published has been received by your subscriber program (which prints the value on the screen).

You've basically created a subscriber node that listens to the /counter topic, and each time it reads something, it calls a function that does a print of the msg. Initially, nothing happened since nobody was publishing into the /counter topic, but when you executed the *rostopic pub* command, you published a message into the /counter topic, so the function has printed the

number in the IPython's output and you could also see that message in the *rostopic echo* output.

How to Prepare CMakeLists.txt and package.xml for Custom Topic Message Compilation

Now you may be wondering... in case I need to publish some data that is not an Int32, which type of message should I use? You can use all ROS defined (***rosmsg list***) messages. But, in case none fit your needs, you can create a new one.
In order to create a new message, you will need to do the following steps:

1. Create a directory named 'msg' inside your package
2. Inside this directory, create a file named Name_of_your_message.msg (more information down)
3. Modify CMakeLists.txt file (more information down)
4. Modify package.xml file (more information down)
5. Compile
6. Use in code

1) Create a directory msg in your package.

```
Entrée [ ]:
roscd <package_name>
mkdir msg
```

2) The **Age.msg** file must contain this:

```
Entrée [ ]:
float32 years
float32 months
float32 days
```

3) **In CMakeLists.txt**
You will have to edit four functions inside CMakeLists.txt:

- find_package()
- add_message_files()
- generate_messages()
- catkin_package()

I. find_package()

This is where all the packages required to COMPILE the messages of the topics, services, and actions go. In package.xml, you have to state them as **build_depend**.
HINT 1: If you open the CMakeLists.txt file in your IDE, you'll see that almost all of the file is commented. This includes some of the lines you will have to modify. Instead of copying and pasting the lines below, find the equivalents in the file and uncomment them, and then add the parts that are missing.

```
find_package(catkin REQUIRED COMPONENTS
      rospy
      std_msgs
      message_generation   # Add message_generation here, after the other
```

```
packages
)
```

II. add_message_files()

This function includes all of the messages of this package (in the msg folder) to be compiled. The file should look like this.

```
add_message_files(
     FILES
     Age.msg
   ) # Dont Forget to UNCOMENT the parenthesis and add_message_files TOO
```

III. generate_messages()

Here is where the packages needed for the messages compilation are imported.

```
generate_messages(
     DEPENDENCIES
     std_msgs
) # Dont Forget to uncoment here TOO
```

IV. catkin_package()

State here all of the packages that will be needed by someone that executes something from your package. All of the packages stated here must be in the package.xml as **exec_depend**.

```
catkin_package(
     CATKIN_DEPENDS rospy message_runtime   # This will NOT be the only
thing here
)
```

Summuarize the minimum expression:

```
cmake_minimum_required(VERSION 2.8.3)
project(topic_ex)


find_package(catkin REQUIRED COMPONENTS
  std_msgs
  message_generation
)

add_message_files(
  FILES
  Age.msg
)
```

```
generate_messages(
  DEPENDENCIES
  std_msgs
)


catkin_package(
  CATKIN_DEPENDS rospy message_runtime
)

include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

4) **Modify package.xml**
Just add these 3 lines to the package.xml file.

```
<build_depend>message_generation</build_depend>

<build_export_depend>message_runtime</build_export_depend>
<exec_depend>message_runtime</exec_depend>
```

summarize minimum expression:

```
<?xml version="1.0"?>
<package format="2">
  <name>topic_ex</name>
  <version>0.0.0</version>
  <description>The topic_ex package</description>


  <maintainer email="user@todo.todo">user</maintainer>

  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>message_generation</build_depend>
  <build_export_depend>rospy</build_export_depend>
  <exec_depend>rospy</exec_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <exec_depend>std_msgs</exec_depend>
  <build_export_depend>message_runtime</build_export_depend>
  <exec_depend>message_runtime</exec_depend>
  <export>
```

```
  </export>
</package>
```

execute:

```
roscd; cd ..
catkin_make
source devel/setup.bash
```

VERY IMPORTANT: When you compile new messages, there is still an extra step before you can use the messages. You have to type in the Webshell, in the **catkin_ws** directory, the following command: ***source devel/setup.bash***.

This executes this bash file that sets, among other things, the newly generated messages created through the catkin_make.

If you don't do this, it might give you a python import error, saying it doesn't find the message generated.

HINT 2: To verify that your message has been created successfully, type in your webshell *rosmsg show Age*. If the structure of the Age message appears, it will mean that your message has been created successfully and it's ready to be used in your ROS programs.

```
rosmsg show Age
```

```
user ~ $ rosmsg show Age
[topic_ex/Age]:
float32 years
float32 months
float32 days
```

There is an issue in ROS that could give you problems when importing msgs from the msg directory. If your package has the same name as the Python file that does the import of the msg, this will give an error saying that it doesn't find the msg element. This is due to the way Python works. Therefore, you have to be careful to not name the Python file exactly the same as its parent package.

Example:
Package name = "my_package"
Python file name = "my_package.py"

This will give an import error because it will try to import the message from the my_package.py file, from a directory .msg that doesn't exists.

IV Service - Clients:

Topics - Services - Actions

To understand what services are and when to use them, you have to compare them with topics and actions.

Imagine you have your own personal BB-8 robot. It has a laser sensor, a face-recognition system, and a navigation system. The laser will use a **Topic** to publish all of the laser readings at 20hz. We use a topic because we need to have that information available all the time for other ROS systems, such as the navigation system.

The Face-recognition system will provide a **Service**. Your ROS program will call that service and **WAIT** until it gives you the name of the person BB-8 has in front of it.

The navigation system will provide an **Action**. Your ROS program will call the action to move the robot somewhere, and **WHILE** it's performing that task, your program will perform other tasks, such as complain about how tiring C-3PO is. And that action will give you **Feedback** (for example: distance left to the desired coordinates) along the process of moving to the coordinates.
So... What's the difference between a **Service** and an **Action**?

Services are **Synchronous**. When your ROS program calls a service, your program can't continue until it receives a result from the service.

Actions are **Asynchronous**. It's like launching a new thread. When your ROS program calls an action, your program can perform other tasks while the action is being performed in another thread.
Conclusion: Use services when your program can't continue until it receives the result from the service.

```
roslaunch iri_wam_aff_demo start_demo.launch
```

What did you do just now?

The launch file has launched two nodes (Yes! You can launch more than one node with a single launch file):

- /iri_wam_reproduce_trajectory
- /iri_wam_aff_demo

The first node provides the ***/execute_trajectory*** service. This is the node that contains the **service**. The second node, performs calls to that service. When the service is called, the robot will execute a given trajectory.

You have to leave the *start_demo.launch* running, otherwise the services won't be there to see them.
Execute the following command in a different shell from the one that has the roslaunch

start_demo.launch running:

```
rosservice list
```

```
user ~ $ rosservice list
/camera/rgb/image_raw/compressed/set_parameters
/camera/rgb/image_raw/compressedDepth/set_parameters
/camera/rgb/image_raw/theora/set_parameters
/camera/set_camera_info
/camera/set_parameters
/execute_trajectory
/gazebo/apply_body_wrench
...
```

Execute the following command to know more about the service ***/execute_trajectory***.

```
rosservice info /execute_trajectory
```

```
user ~ $ rosservice info /execute_trajectory
Node: /iri_wam_reproduce_trajectory
URI: rosrpc://ip-172-31-17-169:35175
Type: iri_wam_reproduce_trajectory/ExecTraj
Args: file
```

Here you have two relevant parts of data.

- **Node**: It states the node that provides (has created) that service.
- **Type**: It refers to the kind of message used by this service. It has the same structure as topics do. It's always made of ***package_where_the_service_message_is_defined / Name_of_the_File_where_Service_message_is_defined***. In this case, the package is ***iri_wam_reproduce_trajectory***, and the file where the Service Message is defined is called ***ExecTraj***.
- **Args**: Here you can find the arguments that this service takes when called. In this case, it only takes a ***trajectory file path*** stored in the variable called ***file***.

Do you remember how to go directly to a package and where to find the launch files?

```
roscd iri_wam_aff_demo
cd launch/
cat start_demo.launch
```

```
<launch>

  <include file="$(find
iri_wam_reproduce_trajectory)/launch/start_service.launch"/>
```

```
  <node pkg ="iri_wam_aff_demo"
        type="iri_wam_aff_demo_node"
        name="iri_wam_aff_demo"
        output="screen">
  </node>

</launch>
```

1) The first part of the launch file calls another launch file called ***start_service.launch***.

That launch file starts the node that provides the */execute_trajectory* service. Note that it's using a special ROS launch file function to find the path of the package given.

```
<include file="$(find
package_where_launch_is)/launch/my_launch_file.launch"/>
```

2) The second part launches a node just as you learned in the **ROS Basics Unit**. That node is the one that will call the */execute_trajectory* service in order to make the robot move.
3) The second node is not a Python node, but a cpp compiled (binary) one. You can build ROS programs either in Cpp or Python. This course focuses on Python.

```
<node pkg ="package_where_cpp_is"
      type="name_of_binary_after_compiling_cpp"
      name="name_of_the_node_initialised_in_cpp"
      output="screen">
</node>
```

How to call a service

You can call a service manually from the console. This is very useful for testing and having a basic idea of how the service works.

```
rosservice call /the_service_name TAB-TAB
```

Let's call the service with the name **/trajectory_by_name** by issuing the following command. But before being able to call this Service, you will have to launch it. For doing so you can execute the following command:

```
roslaunch trajectory_by_name start_service.launch
```

```
rosservice call /trajectory_by_name [TAB]+[TAB]
```

When you [TAB]+[TAB], an extra element appears. ROS autocompletes with the structure needed to input/request the service.
In this case, it gives the following structure:

"traj_name: 'the_name_of_the_trajectory_you_want_to_execute'"
The **/trajectory_by_name** Service is a service provided by the Robot Ignite Academy as an example, that allows you **to execute a specified trajectory** with the robotic arm.
Use that service to execute one trajectory with the WAM Arm. The trajectories that are available are the following ones: **init_pos**, **get_food** and **release_food**.

```
rosservice call /trajectory_by_name "traj_name: 'get_food'"
```

You can now try to call the service providing different trajectory names (from the ones indicated above), and see how the robot executes each one of them.

But how do you interact with a service programatically?

Execute:

```
#! /usr/bin/env python

import rospy
# Import the service message used by the service /trajectory_by_name
from trajectory_by_name_srv.srv import TrajByName, TrajByNameRequest
import sys

# Initialise a ROS node with the name service_client
rospy.init_node('service_client')
# Wait for the service client /trajectory_by_name to be running
rospy.wait_for_service('/trajectory_by_name')
# Create the connection to the service
traj_by_name_service = rospy.ServiceProxy('/trajectory_by_name', TrajByName)
# Create an object of type TrajByNameRequest
traj_by_name_object = TrajByNameRequest()
# Fill the variable traj_name of this object with the desired value
traj_by_name_object.traj_name = "release_food"
# Send through the connection the name of the trajectory to be executed by
the robot
result = traj_by_name_service(traj_by_name_object)
# Print the result given by the service called
print result
```

How to know the structure of the service message used by the service?

```
rosservice info /name_of_the_service
```

This will return you the name_of_the_package/Name_of_Service_message
Then, you can explore the structure of that service message with the following command:

```
rossrv show name_of_the_package/Name_of_Service_message
```

1- Service message files have the extension *.srv*. Remember that Topic message files have the extension *.msg*

2- Service message files are defined inside a *srv* directory. Remember that Topic message files are defined inside a *msg* directory

3- Service messages have TWO parts:
REQUEST

RESPONSE
In the case of the TrajByName service, **REQUEST** contains a string called ***traj_name*** and **RESPONSE** is composed of a boolean named ***success***, and a string named ***status_message***.

The Number of elements on each part of the service message can vary depending on the service needs. You can even put none if you find that it is irrelevant for your service. The important part of the message is the three dashes **---**, because they define the file as a Service Message. Summarizing:

The **REQUEST** is the part of the service message that defines **HOW you will do a call** to your service. This means, what variables you will have to pass to the Service Server so that it is able to complete its task.

The **RESPONSE** is the part of the service message that defines **HOW your service will respond** after completing its functionality. If, for instance, it will return an string with a certaing message saying that everything went well, or if it will return nothing, etc...

4- How to use Service messages In your code:
Whenever a Service message is compiled, 3 message objects are created. Let's say have a Service message named **MyServiceMessage**. If we compile this message, 3 objects will be generated:

- **MyServiceMessage**: This is the Service message itself. It's used for creating a connection to the service server, as you saw in Python Program {4.5}:

```
traj_by_name_service = rospy.ServiceProxy('/trajectory_by_name', TrajByName)
```

MyServiceMessageRequest: This is the object used for creating a request to send to the server. Just like your web browser (a client) connects to a webserver to request web pages, using *HttpRequest* objects. So, this object is used for sending a request to the service server, as you saw in Python Program {4.5}:

```
traj_by_name_object = TrajByNameRequest()
# Fill the variable traj_name of this object with the desired value
traj_by_name_object.traj_name = "release_food"
# Send through the connection the name of the trajectory to be executed by
the robot
result = traj_by_name_service(traj_by_name_object)
```

MyServiceMessageResponse: This is the object used for sending a response from the server back to the client, whenever the service ends. You will learn more details about this object in the following Chapter, where you will learn more about Service Servers.

V Service - Server:

Execute:

```
#! /usr/bin/env python

import rospy
from std_srvs.srv import Empty, EmptyResponse # you import the service
message python classes generated from Empty.srv.


def my_callback(request):
    print "My_callback has been called"
    return EmptyResponse() # the service Response class, in this case
EmptyResponse
    #return MyServiceResponse(len(request.words.split()))

rospy.init_node('service_server')
my_service = rospy.Service('/my_service', Empty , my_callback) # create
the Service called my_service with the defined callback
rospy.spin() # maintain the service open.
```

You have just created and started the Service Server. So basically, you have made this service available for anyone to call it.

```
rosservice list
```

```
/base_controller/command_select
/bb8/camera1/image_raw/compressed/set_parameters
/bb8/camera1/image_raw/compressedDepth/set_parameters
/bb8/camera1/image_raw/theora/set_parameters
...
/my_service
...
```

Now, you have to actually **CALL** it. So, call the **/my_service** service manually. Remember the calling structure discussed in the previous chapter and don't forget to TAB-TAB to autocomplete the structure of the Service message.

```
rosservice call /my_service [TAB]+[TAB]
```

You should've seen the message **'My callback has been called'** printed at the output of the cell with the Python code.

How to create your own service message

So, what if none of the service messages that are available in ROS fit your needs? Then, you create your own message, as you did with the Topic messages.

In order to create a service message, you will have to follow the next steps:

1) Create a package like this:

```
roscd;cd ..;cd src
catkin_create_pkg my_custom_srv_msg_pkg rospy
```

2) Create your own Service message with the following structure. You can put as many variables as you need, of any type supported by ROS: ROS Message Types. Create a ***srv*** folder inside your package , as you did with the topics ***msg*** folder. Then, inside this ***srv*** folder, create a file called ***MyCustomServiceMessage.srv***. You can create with the IDE or the WebShell, as you wish.

```
roscd my_custom_srv_msg_pkg/
mkdir srv
vim srv/MyCustomServiceMessage.srv
```

The **MyCustomServiceMessage.srv** could be something like this:

```
int32 duration    # The time (in seconds) during which BB-8 will keep moving
in circles
---
```

```
bool success      # Did it achieve it?
```

How to Prepare CMakeLists.txt and package.xml for Custom Service Compilation

You have to edit two files in the package similarly to how we explained for Topics:

- CMakeLists.txt
- package.xml

Modification of CMakeLists.txt

You will have to edit four functions inside CMakeLists.txt:

- find_package()
- add_service_files()
- generate_messages()
- catkin_package()

I. find_package()

All the packages needed to COMPILE the messages of topics, services, and actions go here. It's only getting its paths, and not really importing them to be used in the compilation.
The same packages you write here will go in **package.xml,** stating them as **build_depend**.

```
find_package(catkin REQUIRED COMPONENTS
  std_msgs
  message_generation
)
```

II. add_service_files()

This function contains a list of all of the service messages defined in this package (defined in the srv folder).
For our example:

```
add_service_files(
  FILES
  MyCustomServiceMessage.srv
)
```

III. generate_messages()

Here is where the packages needed for the service messages compilation are imported.

```
generate_messages(
```

```
  DEPENDENCIES
  std_msgs
)
```

IV. catkin_package()

State here all of the packages that will be needed by someone that executes something from your package. All of the packages stated here must be in the **package.xml** file as **<exec_depend>**.

```
catkin_package(
      CATKIN_DEPENDS
      rospy
)
```

Modification of package.xml:

1. Add all of the packages needed to compile the messages.
   In this case, you only need to add the ***message_generation***.
   You will have to import those packages as **<build_depend>**.

2. On the other hand, if you need a package for the execution of the programs inside your package, you will have to import those packages as **<exec_depend>**.

In this case, you will only need to add these 3 lines to your **package.xml** file:

```
<build_depend>message_generation</build_depend>

<build_export_depend>message_runtime</build_export_depend>
<exec_depend>message_runtime</exec_depend>
```

Once you're done, compile your package and source the newly generated messages:

```
roscd;cd ..
catkin_make
source devel/setup.bash
```

Important!! When you compile new messages through catkin_make, there is an extra step that needs to be done. You have to type in the WebShell, in the **catkin_ws** directory, the following command: ***source devel/setup.bash***.

This command executes the bash file that sets, among other things, the newly generated messages created with ***catkin_make***.
If you don't do this, it might give you a Python import error, saying that it doesn't find the message generated.

You should see among all the messages something similar to:

Generating Python code from SRV my_custom_srv_msg_pkg/MyCustomServiceMessage

To check that you have the new service message in your system, and ready to be used, type the following:

```
rossrv list | grep MyCustomServiceMessage
```

```
user ~ $ rossrv list | grep MyCustomServiceMessage
my_custom_srv_msg_pkg/MyCustomServiceMessage
```

Now, create a Service Server that uses this type of message.

```
#! /usr/bin/env python

import rospy
from my_custom_srv_msg_pkg.srv import MyCustomServiceMessage,
MyCustomServiceMessageResponse # you import the service message python
classes

# generated from MyCustomServiceMessage.srv.


def my_callback(request):

    print "Request Data==> duration="+str(request.duration)
    my_response = MyCustomServiceMessageResponse()
    if request.duration > 5.0:
        my_response.success = True
    else:
        my_response.success = False
    return  my_response # the service Response class, in this case
MyCustomServiceMessageResponse

rospy.init_node('service_client')
my_service = rospy.Service('/my_service', MyCustomServiceMessage ,
my_callback) # create the Service called my_service with the defined
callback
rospy.spin() # maintain the service open.
```

Summary

Let's do a quick summary of the most important parts of **ROS Services**, just to try to put everything in place.

A **ROS Service** provides a certain functionality of your robot. A ROS Service is composed of 2 parts:

- **Service Server**: This is what **PROVIDES** the functionality. Whatever you want your Service to do, you have to place it in the Service Server.
- **Service Client**: This is what **CALLS** the functionality provided by the Service Server. That is, it CALLS the Service Server.

ROS Services use a special service message, which is composed of 2 parts:

- **Request**: The request is the part of the message that is used to CALL the Service. Therefore, it is sent by the Service Client to the Service Server.
- **Response**: The response is the part of the message that is returned by the Service Server to the Service Client, once the Service has finished.

ROS Services are synchronous. This means that whenever you CALL a Service Server, you have to wait until the Service has finished (and returns a response) before you can do other stuff with your robot.

Object-Oriented Programming

In all the programs we've wrote until now, we have designed our programs around functions, which manipulate data. This is called the procedure-oriented way of programming. But let me tell you that there is another way of programming! This one is based on combining data and functionality, and wrapping them inside "things" known as objects. This way of programming is known as object-oriented programming, or OOP. You may be asking... why are you telling me about this? And why now? And those are great questions!

Most of the time, you will still be able to use procedural programming, but when writing larger and more complex programs, this method can become a pain. In these cases, it is much better to use OOP, since your code will be better organized and it will be much easier to understand (and debug).
Anyway, let me remind you that the goal of this unit is not to learn what OOP is, or what a Python class is, but how to apply these concepts to your ROS code. This a ROS course, isn't it? So, we'll try to summarize these concepts in the easiest and fastest way possible, just to put everything in context.
So, as I've already said before, OOP is based on the concept of "objects." These objects are usually defined by two things:

- Attributes: This is the data associated with the object, defined in fields.
- Methods: These are the procedures or functions associated with the object.

The most important feature of objects is that the methods associated with them can access and modify the data fields of the object with which they are associated.

What are Python classes?

So, everything sounds super cool and interesting, but I haven't read a word yet about Python classes. What the heck are Python classes? Let's see!

A Python classes is, basically, a code template for creating an object. An object is created using the constructor of the class. This object will then be called the instance of the class. Well, what about seeing an example of a Python class, so that we can understand everything better? Let's

see then!

VI Python classes in ROS:

```
class Jedi:
    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print('Hello, my name is', self.name)

j = Jedi('ObiWan')
j.say_hi()
```

This is the **__init__** method of the class. It is also known as the constructor because it will be called as soon as an object of a class is created. It is usually used for the initialization of the attributes. In this case, we are initializing an attribute of the class, which is the name of the Jedi.

This is another method of the class. In this case, it just contains a simple print inside. So, when this method is called, the print will be executed.

Note that we are using the **self** keyword. This keyword refers to the object itself. Each time we define a new method of the class, we will need to pass this keyword as the first argument. Also, when accessing the attributes of the class, we will use this keyword, as you can see.

Using Python classes in ROS

Below you can have a look at a class that will control the movement of the BB-8 robot.

```
#!/usr/bin/env python

import rospy
from geometry_msgs.msg import Twist

class MoveBB8():

    def __init__(self):
        self.bb8_vel_publisher = rospy.Publisher('/cmd_vel', Twist,
queue_size=1)
        self.cmd = Twist()
        self.ctrl_c = False
        self.rate = rospy.Rate(10) # 10hz
        rospy.on_shutdown(self.shutdownhook)
```

```
    def publish_once_in_cmd_vel(self):
        """
        This is because publishing in topics sometimes fails the first time
you publish.
        In continuous publishing systems, this is no big deal, but in
systems that publish only
        once, it IS very important.
        """
        while not self.ctrl_c:
            connections = self.bb8_vel_publisher.get_num_connections()
            if connections > 0:
                self.bb8_vel_publisher.publish(self.cmd)
                rospy.loginfo("Cmd Published")
                break
            else:
                self.rate.sleep()

    def shutdownhook(self):
        # works better than the rospy.is_shutdown()
        self.ctrl_c = True

    def move_bb8(self, linear_speed=0.2, angular_speed=0.2):

        self.cmd.linear.x = linear_speed
        self.cmd.angular.z = angular_speed

        rospy.loginfo("Moving BB8!")
        self.publish_once_in_cmd_vel()

if __name__ == '__main__':
    rospy.init_node('move_bb8_test', anonymous=True)
    movebb8_object = MoveBB8()
    try:
        movebb8_object.move_bb8()
    except rospy.ROSInterruptException:
        pass
```

```
INIT:
```

This is the constructor of the class. Here, we are defining several attributes:

- **bb8_vel_publisher**: Creates a Publisher for the **/cmd_vel** topic
- **cmd**: Contains a Twist message
- **ctrl_c**: Contains a boolean indicating if we have pressed Ctrl+C in order to stop the program
- **rate**: Contains the rate frequency

Also, as you can see in the last line of the **__init__** method, we are setting that when our ROS

program is closed (**rospy.on_shutdown**), the class method **self.shutdownhook** will be triggered.

publish_once_in_cmd_vel :

This class method is used in order to make sure that the first message we publish into a topic is successfully received, as you can read on the commented code. Don't pay too much attention to the code inside it, since it's not important right now. Just keep in mind that you can use this class method for situations where you will publish one single command into a topic.

shutdownhook :

This method will be called when our ROS program is closed, as you saw on the constructor of the class. Therefore, we are setting the **ctrl_c** attribute to True.

move_bb8 :

This method is used to move the robot. As you can see, we just fill in the **cmd** attribute with the values we want, and we then call the **publish_once_in_cmd_vel()** method.

Main:

Finally, in the main function, we are doing three things:

- We create a ROS node named **move_bb8_test**
- We create an object of the MoveBB8 class, which is stored into a variable named **movebb8_object**.
- We call the **move_bb8()** method of the class in order to start moving the BB-8 robot.

Note that when calling the **move_bb8()** method, we are using a **try/except** structure. This is because if the call to the method fails, the error would be caught by the **ROSInterruptException**, avoiding error messages.

Create a ROS package in which to place the class.

```
catkin_create_pkg my_python_class rospy
```

Now, add the bb8_move_circle_class.py file to your package and execute it by running the following command:

```
rosrun my_python_class bb8_move_circle_class.py
```

```
#! /usr/bin/env python

import rospy
from std_srvs.srv import Empty, EmptyResponse
from bb8_move_circle_class import MoveBB8
```

```
def my_callback(request):
    rospy.loginfo("The Service move_bb8_in_circle has been called")
    movebb8_object = MoveBB8()
    movebb8_object.move_bb8()
    rospy.loginfo("Finished service move_bb8_in_circle")
    return EmptyResponse()

rospy.init_node('service_move_bb8_in_circle_server')
my_service = rospy.Service('/move_bb8_in_circle', Empty , my_callback)
rospy.loginfo("Service /move_bb8_in_circle Ready")
rospy.spin() # mantain the service open.
```

```
from move_bb8_circle_class import MoveBB8
```

This line of code imports the **MoveBB8** class we've created from the Python file **move_bb8_circle_class.py**.

```
movebb8_object = MoveBB8()
movebb8_object.move_bb8()
```

Then here, we are creating an object of the **MoveBB8**, and we are calling the **move_bb8()** method in order to start moving the robot.

First, add the bb8_move_circle_service_server.py file to your package. Add also a launch file that starts the Service Server, like the one below:

```
<launch>
    <!-- Start Service Server for move_bb8_in_circle service -->
    <node pkg="my_python_class" type="bb8_move_circle_service_server.py"
name="service_move_bb8_in_circle_server"  output="screen">
    </node>
</launch>
```

That is, that the ROS service **/move_bb8_in_circle** is available, and that when you call it, the BB-8 robot starts moving in circles.

VII Actions - Clients:

"rosrun": ROS command that allows you to run a ROS program without having to create a launch file to launch it (it is a different way from what we've been doing here).

"teleop_twist_keyboard": Name of the package where the ROS program is. In this case, where the python executable is.

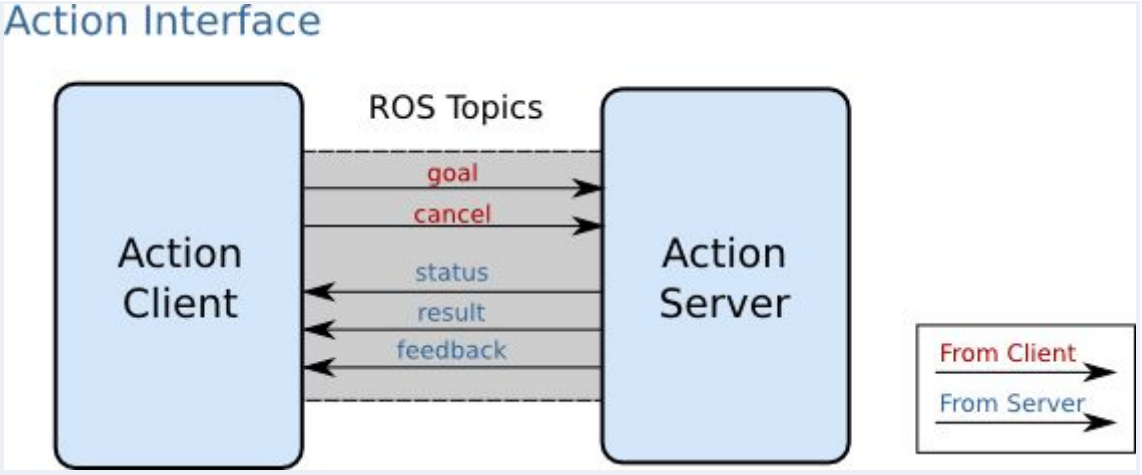
"teleop_twist_keyboard.py": Python executable that will be run. In this case, it's an executable that allows you to input movement commands through the keyboard. When executed, it displays the instructions to move the robot.

What are actions?

Actions are like asynchronous calls to services
Actions are very similar to services. When you call an action, you are calling a functionality that another node is providing. Just the same as with services. The difference is that when your node calls a service, it must wait until the service finishes. **When your node calls an action, it doesn't necessarily have to wait for the action to complete.**

Hence, an action is an asynchronous call to another node's functionality.

- The node that provides the functionality has to contain an **action server**. The *action server* allows other nodes to call that action functionality.
- The node that calls to the functionality has to contain an **action client**. The *action client* allows a node to connect to the *action server* of another node.



In order to find which actions are available on a robot, you must do a ***rostopic list.***

```
rostopic list
```

```
user ~ $ rostopic list
...
...
/ardrone_action_server/cancel
/ardrone_action_server/feedback
/ardrone_action_server/goal
/ardrone_action_server/result
```

```
/ardrone_action_server/status
...
...
```

When a robot provides an action, you will see that in the topics list. There are 5 topics with the same base name, and with the subtopics ***cancel***, ***feedback***, ***goal***, ***result***, and ***status***.

This is because you previously launched the **ardrone_action_server** with the command **roslaunch ardrone_as action_server.launch**

Every action server creates those 5 topics, so you can always tell that an action server is there because you identified those 5 topics.

Therefore, in the example above:

- **ardrone_action_server: Is the name of the Action Server.**
- **cancel, feedback, goal, result and status: Are the messages used to comunicate with the Action Server.**

Calling an action server

The *ardrone_action_server* action server is an action that you can call. If you call it, it will start taking pictures with the front camera, one picture every second, for the amount of seconds specified in the calling message (it is a parameter that you specify in the call).

Calling an action server means sending a message to it. In the same way as with *topics* and *services*, it all works by passing messages around.

- **The message of a topic is composed of a single part: the information the topic provides.**
- **The message of a service has two parts: the request and the response.**
- **The message of an action server is divided into three parts: the goal, the result, and the feedback.**

All of the action messages used are defined in the *action* directory of their package.

You can go to the *ardrone_as* package and see that it contains a directory called *action*. Inside that *action* directory, there is a file called *Ardrone.action*. That is the file that specifies the type of the message that the action uses.

Type in a shell the following commands to see the message structure:

```
roscd ardrone_as/action; cat Ardrone.action
```

```
user ~ $ roscd ardrone_as/action; cat Ardrone.action
#goal for the drone
int32 nseconds  # the number of seconds the drone will be taking pictures
```

```
---
#result
sensor_msgs/CompressedImage[] allPictures # an array containing all the
pictures taken along the nseconds
---
#feedback
sensor_msgs/CompressedImage lastImage  # the last image taken
```

You can see in the previous step how the message is composed of three parts:

goal: Consists of a variable called *nseconds* of type *Int32*. This Int32 type is a standard ROS message, therefore, it can be found in the std_msgs package. Because it's a standard package of ROS, it's not needed to indicate the package where the *Int32* can be found.
result: Consists of a variable called *allPictures*, which is an array of type *CompressedImage[]*, found in the sensor_msgs package.
feedback: Consists of a variable called *lastImage* of type *CompressedImage[]*, found in the sensor_msgs package.

You will learn in the second part of this chapter about how to create your own action messages. For now, you must only understand that every time you call an action, the message implied contains three parts, and that each part can contain more than one variable.

Actions provide feedback

Due to the fact that calling an action server does not interrupt your thread, action servers provide a message called *the feedback*. The feedback is a message that the action server generates every once in a while to indicate how the action is going (informing the caller of the status of the requested action). It is generated while the action is in progress.

How to call an action server

The way you call an action server is by implementing an *action client*.

The following is a self-explanatory example of how to implement an action client that calls the ardrone_action_server and makes it take pictures for 10 seconds.

```
#! /usr/bin/env python
import rospy
import time
import actionlib
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult,
ArdroneFeedback

nImage = 1

# definition of the feedback callback. This will be called when feedback
# is received from the action server
# it just prints a message indicating a new message has been received
```

```
def feedback_callback(feedback):
    global nImage
    print('[Feedback] image n.%d received'%nImage)
    nImage += 1

# initializes the action client node
rospy.init_node('drone_action_client')

# create the connection to the action server
client = actionlib.SimpleActionClient('/ardrone_action_server',
ArdroneAction)
# waits until the action server is up and running
client.wait_for_server()

# creates a goal to send to the action server
goal = ArdroneGoal()
goal.nseconds = 10 # indicates, take pictures along 10 seconds

# sends the goal to the action server, specifying which feedback function
# to call when feedback received
client.send_goal(goal, feedback_cb=feedback_callback)

# Uncomment these lines to test goal preemption:
#time.sleep(3.0)
#client.cancel_goal()  # would cancel the goal 3 seconds after starting

# wait until the result is obtained
# you can do other stuff here instead of waiting
# and check for status from time to time
# status = client.get_state()
# check the client API link below for more info

client.wait_for_result()

print('[Result] State: %d'%(client.get_state()))
```

The code to call an action server is very simple:

- **First, you create a client connected to the action server you want:**

  client = actionlib.SimpleActionClient('/ardrone_action_server', ArdroneAction)
  client = actionlib.SimpleActionClient('/the_action_server_name', the_action_server_message_python_object)

  - **First parameter is the name of the action server you want to connecto to.**
  - **Second parameter is the type of action message that it uses. The**

convention goes as follows:

If your action message file was called *Ardrone.action*, then the type of action message you must specify is *ArdroneAction*. The same rule applies to any other type (*R2Action*, for an *R2.action* file or *LukeAction* for a *Luke.action* file). In our exercise it is:

client = actionlib.SimpleAction('/ardrone_action_server', ArdroneAction)

- **Then you create a goal:**

  goal = ArdroneGoal()

  Again, the convention goes as follows:

  If your action message file was called *****Ardrone.action***, then the type of goal message you must specify is *****ArdroneGoal()*****. The same rule applies to any other type (*****R2Goal()***** for an *****R2.action***** file or *****LukeGoal()***** for a *****Luke.action***** file).**
  Because the goal message requires to provide the number of seconds taking pictures (in the *****nseconds variable***), you must set that parameter in the goal class instance:**

  goal.nseconds = 10

- **Next, you send the goal to the action server:**

  client.send_goal(goal, feedback_cb=feedback_callback)

  That sentence calls the action. In order to call it, you must specify 2 things:
  1. **The goal parameters**
  2. **A feedback function to be called from time to time to know the status of the action.**

     At this point, the action server has received the goal and started to execute it (taking pictures for 10 seconds). Also, feedback messages are being received. Every time a feedback message is received, the *****feedback_callback*** function is executed.**

- **Finally, you wait for the result:**

  client.wait_for_result()

How to perform other tasks while the Action is in progress

You know how to call an action and wait for the result but... That's exactly what a service does! Then why are you learning actions?
Good point!

So, the SimpleActionClient objects have two functions that can be used for knowing if the action that is being performed has finished, and how:

1) wait_for_result(): This function is very simple. When called, it will wait until the action has finished and returns a true value. As you can see, it's useless if you want to perform other tasks in parallel because the program will stop there until the action is finished.

2) get_state(): This function is much more interesting. When called, it returns an integer that indicates in which state is the action that the SimpleActionClient object is connected to.

0 ==> PENDING

1 ==> ACTIVE

2 ==> DONE

3 ==> WARN

4 ==> ERROR

This allows you to create a while loop that checks if the value returned by get_state() is 2 or higher. If it is not, it means that the action is still in progress, so you can keep doing other things.

Execute the following Python codes {5.5a: wait_for_result_test.py} and {5.5b: no_wait_for_result_test.py} by clicking on them, one at a time, and then clicking on the play button on the top righ-hand corner of the IPython notebook.

Observe the difference between them (the code is printed below) and think about why this is.

IMPORTANT!! Remember that you need to have the *roslaunch ardrone_as action_server.launch* running (probably in WebShell #1), otherwise this won't work because there will be NO Action Server to be connected to.

```
#! /usr/bin/env python

import rospy
import time
```

```
import actionlib
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult,
ArdroneFeedback

nImage = 1

# definition of the feedback callback. This will be called when feedback
# is received from the action server
# it just prints a message indicating a new message has been received
def feedback_callback(feedback):
    global nImage
    print('[Feedback] image n.%d received'%nImage)
    nImage += 1


# initializes the action client node
rospy.init_node('example_with_waitforresult_action_client_node')


action_server_name = '/ardrone_action_server'
client = actionlib.SimpleActionClient(action_server_name, ArdroneAction)

# waits until the action server is up and running
rospy.loginfo('Waiting for action Server '+action_server_name)
client.wait_for_server()
rospy.loginfo('Action Server Found...'+action_server_name)


# creates a goal to send to the action server
goal = ArdroneGoal()
goal.nseconds = 10 # indicates, take pictures along 10 seconds

client.send_goal(goal, feedback_cb=feedback_callback)
rate = rospy.Rate(1)

rospy.loginfo("Lets Start The Wait for the Action To finish Loop...")
while not client.wait_for_result():
    rospy.loginfo("Doing Stuff while waiting for the Server to give a
result....")
    rate.sleep()

rospy.loginfo("Example with WaitForResult Finished.")
```

```
#! /usr/bin/env python

import rospy
import time
import actionlib
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult,
ArdroneFeedback

"""
class SimpleGoalState:
    PENDING = 0
    ACTIVE = 1
    DONE = 2
    WARN = 3
    ERROR = 4

"""
# We create some constants with the corresponing vaules from the
SimpleGoalState class
PENDING = 0
ACTIVE = 1
DONE = 2
WARN = 3
ERROR = 4

nImage = 1

# definition of the feedback callback. This will be called when feedback
# is received from the action server
# it just prints a message indicating a new message has been received
def feedback_callback(feedback):
    """
    Error that might jump

    self._feedback.lastImage =
AttributeError: 'ArdroneAS' obj

    """
    global nImage
    print('[Feedback] image n.%d received'%nImage)
    nImage += 1

# initializes the action client node
rospy.init_node('example_no_waitforresult_action_client_node')

action_server_name = '/ardrone_action_server'
client = actionlib.SimpleActionClient(action_server_name, ArdroneAction)
```

```
# waits until the action server is up and running
rospy.loginfo('Waiting for action Server '+action_server_name)
client.wait_for_server()
rospy.loginfo('Action Server Found...'+action_server_name)

# creates a goal to send to the action server
goal = ArdroneGoal()
goal.nseconds = 10 # indicates, take pictures along 10 seconds

client.send_goal(goal, feedback_cb=feedback_callback)


# You can access the SimpleAction Variable "simple_state", that will be 1 if
active, and 2 when finished.
#Its a variable, better use a function like get_state.
#state = client.simple_state
# state_result will give the FINAL STATE. Will be 1 when Active, and 2 if NO
ERROR, 3 If Any Warning, and 3 if ERROR
state_result = client.get_state()

rate = rospy.Rate(1)

rospy.loginfo("state_result: "+str(state_result))

while state_result < DONE:
    rospy.loginfo("Doing Stuff while waiting for the Server to give a
result....")
    rate.sleep()
    state_result = client.get_state()
    rospy.loginfo("state_result: "+str(state_result))

rospy.loginfo("[Result] State: "+str(state_result))
if state_result == ERROR:
    rospy.logerr("Something went wrong in the Server Side")
if state_result == WARN:
    rospy.logwarn("There is a warning in the Server Side")

#rospy.loginfo("[Result] State: "+str(client.get_result()))
```

Essentially, the difference is that in the first program (5.5a), the log message *rospy.loginfo("Doing Stuff while waiting for the Server to give a result....")* will never be printed, while in the 5.5b it will.

This is because in 5.5a, the program starts the while loop to check if the value returned by *client.wait_for_result()* is True or False, but it will wait for a value that will only be returned when the Action has Finished. Therefore, it will never get inside the while loop because it will always return the value True.

On the other hand, in the 5.5b program, it checks if *state_result < DONE*. And because the function *get_state()* will return the current state of the action immediately, it allows other tasks to perform in parallel. In this case, printing the log message WHILE printing also the feedback of the the Action.

Preempting a goal

It happens that you can cancel a goal previously sent to an action server prior to its completion.

Cancelling a goal while it is being executed is called preempting a goal

You may need to preempt a goal for many reasons, like, for example, the robot went mad about your goal and it is safer to stop it prior to the robot doing some harm.

In order to preempt a goal, you send the cancel_goal to the server through the client connection.

```
client.cancel_goal()
```

See how the goal gets cancelled.

Important!! Remember that you have to have the *roslaunch ardrone_as action_server.launch* running (probably in WebShell #1), otherwise this won't work because there will be NO action server to be connected to.

```
#! /usr/bin/env python

import rospy
import time
import actionlib
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult,
ArdroneFeedback

# We create some constants with the corresponing vaules from the
SimpleGoalState class
PENDING = 0
ACTIVE = 1
```

```
DONE = 2
WARN = 3
ERROR = 4

nImage = 1

# definition of the feedback callback. This will be called when feedback
# is received from the action server
# it just prints a message indicating a new message has been received
def feedback_callback(feedback):
    """
    Error that might jump

    self._feedback.lastImage =
AttributeError: 'ArdroneAS' obj

    """
    global nImage
    print('[Feedback] image n.%d received'%nImage)
    nImage += 1

# initializes the action client node
rospy.init_node('example_no_waitforresult_action_client_node')

action_server_name = '/ardrone_action_server'
client = actionlib.SimpleActionClient(action_server_name, ArdroneAction)

# waits until the action server is up and running
rospy.loginfo('Waiting for action Server '+action_server_name)
client.wait_for_server()
rospy.loginfo('Action Server Found...'+action_server_name)

# creates a goal to send to the action server
goal = ArdroneGoal()
goal.nseconds = 10 # indicates, take pictures along 10 seconds

client.send_goal(goal, feedback_cb=feedback_callback)


# You can access the SimpleAction Variable "simple_state", that will be 1 if
active, and 2 when finished.
#Its a variable, better use a function like get_state.
#state = client.simple_state
# state_result will give the FINAL STATE. Will be 1 when Active, and 2 if NO
ERROR, 3 If Any Warning, and 3 if ERROR
state_result = client.get_state()

rate = rospy.Rate(1)
```

```
rospy.loginfo("state_result: "+str(state_result))
counter = 0
while state_result < DONE:
    rospy.loginfo("Doing Stuff while waiting for the Server to give a
result....")
    counter += 1
    rate.sleep()
    state_result = client.get_state()
    rospy.loginfo("state_result: "+str(state_result)+", counter
="+str(counter))
    if counter == 2:
        rospy.logwarn("Canceling Goal...")
        client.cancel_goal()
        rospy.logwarn("Goal Canceled")
        state_result = client.get_state()
        rospy.loginfo("Update state_result after Cancel :
"+str(state_result)+", counter ="+str(counter))
```

This program counts to 2, and then it cancels the goal. This triggers the server to finish the goal and, therefore, the function ****get_state()** returns the value DONE (2).**

There is a known ROS issue with Actions. It issues a warning when the connection is severed. It normally happens when you cancel a goal or you just terminate a program with a client object in it. The warning is given in the Server Side.

[WARN] Inbound TCP/IP connection failed: connection from sender terminated before handshake header received. 0 bytes were received. Please check sender for additional details.

Just don't panic, it has no effect on your program.

How does all that work?

You need to understand how the communication inside the actions works. It is not that you are going to use it for programming. As you have seen, programming an action client is very simple. However, it will happen that your code will have bugs and you will have to debug it. In order to do proper debugging, you need to understand how the communication between *action servers* and *action clients* works.
As you already know, an *action message* has three parts:

- the goal
- the result
- the feedback

Each one corresponds to a topic and to a type of message.
For example, in the case of the **ardrone_action_server**, the topics involved are the following:

- the goal topic: /ardrone_action_server/goal

- the result topic: /ardrone_action_server/result
- the feedback topic: /ardrone_action_server/feedback

Look again at the ActionClient+ActionServer communication diagram.

So, whenever an action server is called, the sequence of steps are as follows:

1. When an ***action client*** calls an ***action server*** from a node, what actually happens is that the ***action client*** sends to the ***action server*** the goal requested through the */ardrone_action_server/goal* topic.
2. When the ***action server*** starts to execute the goal, it sends to the ***action client*** the feedback through the */ardrone_action_server/feedback* topic.
3. Finally, when the ***action server*** has finished the goal, it sends to the ***action client*** the result through the */ardrone_action_server/result* topic.

Now, let's do the following exercise in order to see how all this ballet happens underneath your programs.

Due to the way actions work, you can actually call the *ardrone_action_server action server* publishing in the topics directly (emulating by hence, what the Python code action client is doing). It is important that you understand this because you will need this knowledge to debug your programs.

```
roslaunch ardrone_as action_server.launch
```

Let's activate the *ardrone_action_server* action server through topics with the following exercise.

Use the webshell to send a goal to the */ardrone_action_server* action server, and to observe what happens in the *result* and *feedback* topics.

```
rostopic pub /[name_of_action_server]/goal
/[type_of_the_message_used_by_the_topic] [TAB][TAB]
```

You should see the same result as in exercice 5.8, with the difference that the goal has been sent by hand, publishing directly into the goal topic, instead of publishing through a Python program.

- You don't have to type the message by hand. Remember to use TAB-TAB to make ROS autocomplete your commands (or give you options).
- Once you achieve that, and ROS autocompletes the message you must send, you will have to modify the parameter **nseconds**, because the default value is zero (remember that parameter indicates the number of seconds taking pictures). Move to the correct place of the message using the keyboard.

The axclient

Until now, you've learnt to send goals to an Action Server using these 2 methods:

- Publishing directly into the /goal topic of the Action Server
- Publishing the goal using Python code

But, let me tell you that there's still one more method you can use in order to send goals to an Action Server, which is much easier and faster than the 2 methods you've learnt: using the **axclient**.

The axclient is, basically, a GUI tool provided by the actionlib package, that allows you to interact with an Action Server in a very easy and visual way. The command used to launch the axclient is the following:

```
rosrun actionlib axclient.py /<name_of_action_server>
```

Before starting with this exercise, make sure that you have your action server running. If it is not running, the axclient won't work.

```
roslaunch ardrone_as action_server.launch
```

Now, let's launch the axclient in order to send goals to the Action Server.

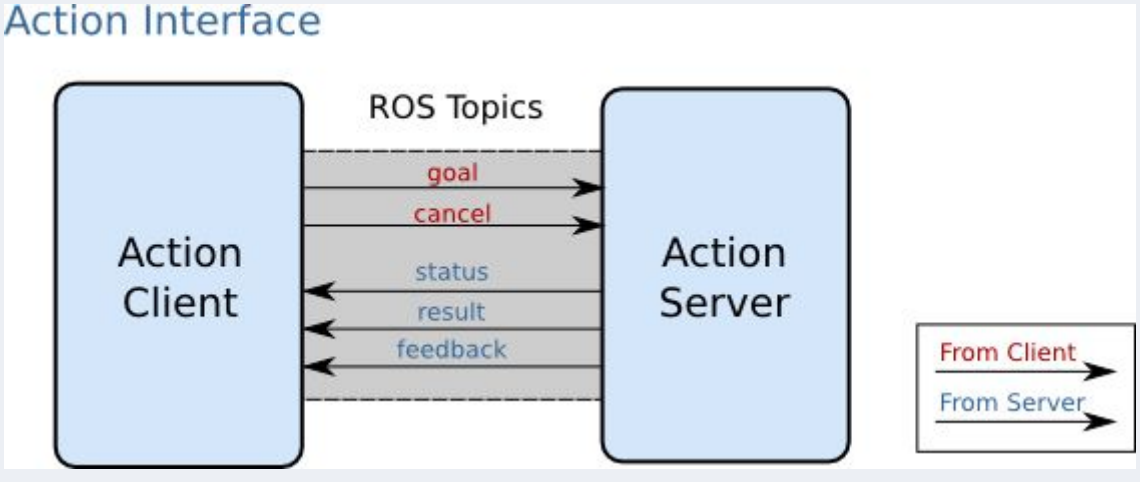
```
rosrun actionlib axclient.py /ardrone_action_server
```

Now everything is settled, and you can start playing with axclient! For instance, you could send goals to the Action Server and visualize the different topics that take part in an Action, which you have learnt in this Chapter.

Sometimes you may find that when you click the SEND GOAL button, or when you try to change the value of the goal you want to send, you can't interact with the axclient screen. If that's the case, just go to another tab and return again to the tab with axclient, and everything will work fine.

VIII Actions - Server:

In the previous lesson, you learned how to **CALL** an action server creating an action client. In this lesson, you are going to learn how to **CREATE** your own action server.



Writing an action server

What follows is the code of an example of a ROS action server. When called, the action server will generate a Fibonacci sequence of a given order. The action server goal message must indicate the order of the sequence to be calculated, the feedback of the sequence as it is being computed, and the result of the final Fibonacci sequence.

```
#! /usr/bin/env python
import rospy

import actionlib

from actionlib_tutorials.msg import FibonacciFeedback, FibonacciResult,
FibonacciAction

class FibonacciClass(object):

  # create messages that are used to publish feedback/result
  _feedback = FibonacciFeedback()
  _result   = FibonacciResult()

  def __init__(self):
    # creates the action server
    self._as = actionlib.SimpleActionServer("fibonacci_as", FibonacciAction,
self.goal_callback, False)
    self._as.start()

  def goal_callback(self, goal):
    # this callback is called when the action server is called.
    # this is the function that computes the Fibonacci sequence
    # and returns the sequence to the node that called the action server
```

```
    # helper variables
    r = rospy.Rate(1)
    success = True

    # append the seeds for the fibonacci sequence
    self._feedback.sequence = []
    self._feedback.sequence.append(0)
    self._feedback.sequence.append(1)

    # publish info to the console for the user
    rospy.loginfo('"fibonacci_as": Executing, creating fibonacci sequence of
order %i with seeds %i, %i' % ( goal.order, self._feedback.sequence[0],
self._feedback.sequence[1]))

    # starts calculating the Fibonacci sequence
    fibonacciOrder = goal.order
    for i in xrange(1, fibonacciOrder):

      # check that preempt (cancelation) has not been requested by the
action client
      if self._as.is_preempt_requested():
        rospy.loginfo('The goal has been cancelled/preempted')
        # the following line, sets the client in preempted state (goal
cancelled)
        self._as.set_preempted()
        success = False
        # we end the calculation of the Fibonacci sequence
        break

      # builds the next feedback msg to be sent
      self._feedback.sequence.append(self._feedback.sequence[i] +
self._feedback.sequence[i-1])
      # publish the feedback
      self._as.publish_feedback(self._feedback)
      # the sequence is computed at 1 Hz frequency
      r.sleep()

    # at this point, either the goal has been achieved (success==true)
    # or the client preempted the goal (success==false)
    # If success, then we publish the final result
    # If not success, we do not publish anything in the result
    if success:
      self._result.sequence = self._feedback.sequence
      rospy.loginfo('Succeeded calculating the Fibonacci of order %i' %
fibonacciOrder )
      self._as.set_succeeded(self._result)

if __name__ == '__main__':
```

```
  rospy.init_node('fibonacci')
  FibonacciClass()
  rospy.spin()
```

In this case, the action server is using an action message definition called ***Fibonacci.action***. That message has been created by ROS into its ***actionlib_tutorials*** package.

```
from actionlib_tutorials.msg import FibonacciFeedback, FibonacciResult,
FibonacciAction
```

Here we are importing the message objects generates by this ***Fibonacci.action*** file.

```
_feedback = FibonacciFeedback()
_result   = FibonacciResult()
```

Here, we are creating the message objects that will be used for publishing the **feedback** and the **result** of the action.

```
def __init__(self):
    # creates the action server
    self._as = actionlib.SimpleActionServer("fibonacci_as", FibonacciAction,
self.goal_callback, False)
    self._as.start()
```

This is the constructor of the class. Inside this constructor, we are creating an Action Server that will be called **"fibonacci_as"**, that will use the Action message **FibonacciAction**, and that will have a callback function called **goal_callback**, that will be activated each time a new goal is sent to the Action Server.

```
def goal_callback(self, goal):

    r = rospy.Rate(1)
    success = True
```

Here we define the **goal callback** function. Each time a new goal is sento to the Action Server, this function will be called.

```
self._feedback.sequence = []
self._feedback.sequence.append(0)
self._feedback.sequence.append(1)

rospy.loginfo('"fibonacci_as": Executing, creating fibonacci sequence of
order %i with seeds %i, %i' % ( goal.order, self._feedback.sequence[0],
self._feedback.sequence[1]))
```

Here we are **initializing** the Fibonacci sequence, and setting up the first values (seeds) of it. Also, we print data for the user related to the Fibonacci sequence the Action Server is going to calculate.

```
fibonacciOrder = goal.order
for i in xrange(1, fibonacciOrder):
```

Here, we start a loop that while go until the **goal.order** value is reached. This value is, obviously, the order of the Fibonacci sequence that the user has sent from the Action Client.

```
if self._as.is_preempt_requested():
    rospy.loginfo('The goal has been cancelled/preempted')
    # the following line, sets the client in preempted state (goal
cancelled)
    self._as.set_preempted()
    success = False
    # we end the calculation of the Fibonacci sequence
    break
```

We check if the goal has been cancelled (preempted). Remember you saw how to to preempt a goal in the previous Chapter.

```
self._feedback.sequence.append(self._feedback.sequence[i] +
self._feedback.sequence[i-1])
self._as.publish_feedback(self._feedback)
r.sleep()
```

Here, we are actually calculating the values of the Fibonacci sequence. You can check how a Fibonacci sequence is calculated here: Fibonacci sequence. Also, we keep publishing **feedback** each time a new value of the sequence is calculated.

```
if success:
  self._result.sequence = self._feedback.sequence
  rospy.loginfo('Succeeded calculating the Fibonacci of order %i' %
fibonacciOrder )
  self._as.set_succeeded(self._result)
```

If everything went OK, we publish the **result**, which is the whole Fibonacci sequence, and we set the Action as succedeed using the **set_succeeded()** function.

Launch again the python code above {5.11a} to have the Fibonacci server running.

Then, execute the following commands in their corresponding WebShells.

```
rostopic echo /fibonacci_as/result
```

```
rostopic echo /fibonacci_as/feedback
```

```
rostopic pub /fibonacci_as/goal actionlib_tutorials/FibonacciActionGoal
[TAB][TAB]
```

After having called the action, the feedback topic should be publishing the feedback, and the result once the calculations are finished.

You must be aware that the name of the messages (the class) used in the Python code are called ***FibonacciGoal***, ***FibonacciResult***, and ***FibonacciFeedback***, while the name of the messages used in the topics are called ***FibonacciActionGoal***, ***FibonacciActionResult***, and ***FibonacciActionFeedback***.

Do not worry about that, just bear it in mind and use it accordingly.

How to create your own action server message

It is always recommended that you use the action messages already provided by ROS. These can be found in the following ROS packages:

- actionlib
  - Test.action
  - TestRequest.action
  - TwoInts.action
- actionlib_tutorials
  - Fibonacci.action
  - Averaging.action

However, it may happen that you need to create your own type. Let's learn how to do it.
To create your own custom action message you have to:
1.- Create an **action** directory within your package.

2.- Create your **Name.action** action message file.

- The Name of the action message file will determine later the name of the classes to be used in the **action server** and/or **action client**. ROS convention indicates that the name has to be camel-case.
- Remember the Name.action file has to contain three parts, each part separated by three hyphens.

```
#goal
package_where_message_is/message_type goal_var_name
---
#result
package_where_message_is/message_type result_var_name
```

```
---
#feedback
package_where_message_is/message_type feedback_var_name
```

- If you do not need one part of the message (for example, you don't need to provide feedback), then you can leave that part empty. But you **must always specify the hyphen separtors**.

3.- Modify the file CMakeLists.txt and the package.xml to include action message compilation. Read the detailed description below.

How to prepare CMakeLists.txt and package.xml files for custom action messages compilation

You have to edit two files in the package, in the same way that we explained for topics and services:

- CMakeLists.txt
- package.xml

Modification of CMakeLists.txt

You will have to edit four functions inside CMakeLists.txt:

- find_package()
- add_action_files()
- generate_messages()
- catkin_package()

I. find_package()

All of the packages needed to COMPILE the messages of topic, services, and actions go here. In *package.xml*, you have to state them as built.

```
find_package(catkin REQUIRED COMPONENTS
      # your packages are listed here
      actionlib_msgs
)
```

II. add_action_files()

This function will contain all of the action messages from this package (which are stored in the **action** folder) that need to be compiled.
Place them beneath
the FILES tag.

```
add_action_files(
      FILES
      Name.action
)
```

III. generate_messages()

The packages needed for the action messages compilation are imported here. Write the same here as you wrote in the find_package.

```
generate_messages(
      DEPENDENCIES
      actionlib_msgs
      # Your packages go here
)
```

IV. catkin_package()

State here all of the packages that will be needed by someone that executes something from your package.
All of the packages stated here must be in the **package.xml** file as **<exec_depend>**.

```
catkin_package(
      CATKIN_DEPENDS
      rospy
      # Your package dependencies go here
)
```

Modification of package.xml:

1.- Add all of the packages needed to compile the messages.

If, for example, one of your variables in the **.action** file uses a message defined outside the std_msgs package, let's say **nav_msgs/Odometry**, you will need to import it. To do so, you would have to add as **<build_depend>** the **nav_msgs** package, adding the following line:

```
<build_depend>nav_msgs<build_depend>
```

2.- On the other hand, if you need a package for the execution of the programs inside your package, you will have to import those packages as **<exec_depend>**, adding the following line:

```
<build_export_depend>nav_msgs<build_export_depend>
<exec_depend>nav_msgs<exec_depend>
```

When you compile custom action messages, it's **mandatory** to add the **actionlib_msgs** as build_dependency.

```
<build_depend>actionlib_msgs</build_depend>
```

When you use Python, it's **mandatory** to add the **rospy** as **run_dependency**.

```
<build_export_depend>rospy<build_export_depend>
<exec_depend>rospy<exec_depend>
```

This is due to the fact that the rospy python module is needed in order to run all of your python ROS code.

Summarizing

```
cmake_minimum_required(VERSION 2.8.3)
project(my_custom_action_msg_pkg)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  std_msgs
  actionlib_msgs
)

## Generate actions in the 'action' folder
add_action_files(
   FILES
   Name.action
 )

## Generate added messages and services with any dependencies listed here
generate_messages(
   DEPENDENCIES
   std_msgs actionlib_msgs
 )

catkin_package(
 CATKIN_DEPENDS rospy
)

## Specify additional locations of header files
## Your package locations should be listed before other locations
# include_directories(include)
include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

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

```
<package format="2">
  <name>my_custom_action_msg_pkg</name>
  <version>0.0.0</version>
  <description>The my_custom_action_msg_pkg package</description>
  <maintainer email="user@todo.todo">user</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>actionlib</build_depend>
  <build_depend>actionlib_msgs</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_export_depend>actionlib</build_export_depend>
  <build_export_depend>actionlib_msgs</build_export_depend>
  <build_export_depend>rospy</build_export_depend>
  <exec_depend>actionlib</exec_depend>
  <exec_depend>actionlib_msgs</exec_depend>
  <exec_depend>rospy</exec_depend>

  <export>
  </export>
</package>
```

Finally, when everything is correctly set up, you just have to compile:

```
roscd; cd ..
catkin_make
source devel/setup.bash
rosmsg list | grep Name
```

```
my_custom_action_msg_pkg/NameAction
my_custom_action_msg_pkg/NameActionFeedback
my_custom_action_msg_pkg/NameActionGoal
my_custom_action_msg_pkg/NameActionResult
my_custom_action_msg_pkg/NameFeedback
my_custom_action_msg_pkg/NameGoal
my_custom_action_msg_pkg/NameResult
```

Note that you haven't imported the **std_msgs** package anywhere. But you can use the messages declared there in your custom .actions. That's because this package forms part of the roscore file systems, so therefore, it's embedded in the compilation protocols, and no declaration of use is needed.

IX Debugging tools:

One of the most difficult, but important, parts of robotics is: **knowing how to turn your ideas and knowledge into real projects**. There is a constant in robotics projects: **nothing works as in theory**. Reality is much more complex and, therefore, you need tools to discover what is going on and find where the problem is. That's why debugging and visualization tools are essential in robotics, especially when working with complex data formats such as **images**, **laser-scans**, **pointclouds** or **kinematic data**.

```
roswtf
```

```
user ~ $ roswtf
the rosdep view is empty: call 'sudo rosdep init' and 'rosdep update'
No package or stack in context
================================================================================
Static checks summary:

Found 1 error(s).

ERROR ROS Dep database not initialized: Please initialize rosdep database
with sudo rosdep init.
================================================================================
Beginning tests of your ROS graph. These may take awhile...
analyzing graph...
... done analyzing graph
running graph rules...
... done running graph rules

Online checks summary:

Found 2 warning(s).
Warnings are things that may be just fine, but are sometimes at fault

WARNING The following node subscriptions are unconnected:
 * /gazebo:
   * /gazebo/set_model_state
   * /gazebo/set_link_state
   * /iri_wam/iri_wam_controller/follow_joint_trajectory/cancel
   * /iri_wam/iri_wam_controller/follow_joint_trajectory/goal
   * /iri_wam/e_stop
   * /iri_wam/iri_wam_controller/command
```

```
WARNING These nodes have died:
 * urdf_spawner-4
```

In this particular case, it tells you {webshell-out-6.1} that the package *rosdep* hasn't been initialised, so you may have issues installing new ROS packages from the Internet. In this case, there is no problem because the system you are using (Robot Ignite Academy system) is not meant for installing anything.

And this takes us to the question: **What does *roswtf* do?**

By default, it checks **two** ROS fields:

- **File-system issues:** It checks environmental variables, packages, and launch files, among other things. It looks for any inconsistencies that might be errors. You can use the command *roswtf* alone to get the system global status. But you can also use it to check particular *launch files* before using them.

```
roslaunch iri_wam_aff_demo false_start_demo.launch
```

```
user ~ $ roslaunch iri_wam_aff_demo false_start_demo.launch
... logging to
/home/user/.ros/log/fd58c97c-9068-11e6-9889-02c6d37ebbf9/roslaunch-ip-172-31
-20-234-12087.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-172-31-20-234:38217/

SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.11.20

NODES
  /
    iri_wam_aff_demo (iri_wam_reproduce_trajectory/iri_wam_aff_demo_node)
    iri_wam_reproduce_trajectory
(iri_wam_reproduce_trajectory/iri_wam_reproduce_trajectory_node)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
```

```
process[iri_wam_reproduce_trajectory-1]: started with pid [12111]
ERROR: cannot launch node of type
[iri_wam_reproduce_trajectory/iri_wam_aff_demo_node]: can't locate
node[iri_wam_aff_demo_node] in package [iri_wam_reproduce_trajectory]
```

Any clue? It just tells you that it can't locate your *iri_wam_aff_demo_node*. Now try **roswtf** on that launch file to get a bit more information on what might be the problem:

```
roscd iri_wam_aff_demo/launch
roswtf false_start_demo.launch
```

```
user launch $ roswtf false_start_demo.launch
the rosdep view is empty: call 'sudo rosdep init' and 'rosdep update'
[rospack] Error: the rosdep view is empty: call 'sudo rosdep init' and
'rosdep update'
================================================================================
Static checks summary:

Found 2 error(s).

ERROR ROS Dep database not initialized: Please initialize rosdep database
with sudo rosdep init.
ERROR Several nodes in your launch file could not be located. These are
either typed incorrectly or need to be built:
 * node [iri_wam_aff_demo_node] in package [iri_wam_reproduce_trajectory]
```

To make ***roswtf yourlaunchfile.launch*** work, you need to go to the path where the file is. That's why you had to use the ***roscd*** command.
The error shown above it is telling you that ***roswtf*** can't find the ***iri_wam_aff_demo_node***. It states that because it's a binary, it might be that you haven't compiled it yet or that you spelt something wrong. Essentially, it's telling you that there is no node ***iri_wam_aff_demo_node*** in the package ***iri_wam_reproduce_trajectory***.

- **Online/graph issues**: **roswtf** also checks for any inconsistencies in the connections between nodes, topics, actions, and so on. It warns you if something is not connected or it's connected where it shouldn't be. **These warnings aren't necessarily errors**. They are simply things that ROS finds odd. It's up to you to know if it's an error or if it's just the way your project is wired.

```
roswtf
```

```
user ~ $ roswtf
the rosdep view is empty: call 'sudo rosdep init' and 'rosdep update'
```

```
No package or stack in context
================================================================================
====
Static checks summary:

Found 1 error(s).

ERROR ROS Dep database not initialized: Please initialize rosdep database
with sudo rosdep init.
================================================================================
====
Beginning tests of your ROS graph. These may take awhile...
analyzing graph...
... done analyzing graph
running graph rules...
... done running graph rules

Online checks summary:

Found 2 warning(s).
Warnings are things that may be just fine, but are sometimes at fault

WARNING The following node subscriptions are unconnected:
 * /gazebo:
   * /gazebo/set_model_state
   * /gazebo/set_link_state
   * /iri_wam/iri_wam_controller/follow_joint_trajectory/cancel
   * /iri_wam/iri_wam_controller/follow_joint_trajectory/goal
   * /iri_wam/e_stop
   * /iri_wam/iri_wam_controller/command

WARNING These nodes have died:
 * urdf_spawner-4
```

You executed this command at the start, but you didn't pay attention to the lower part warnings. These warnings are *Graph issues*.

- It states that some subscribers are not connected to the topics that they are meant to be connected to: This is quite normal as they might be nodes that only connect at the start or in certain moments. No error here.
- The second warning states that a node has died: This also is quite normal as nodes, like in this case, that only run when they spawn objects, die after being used. But ROS is so kind that it lets you know, just in case it shouldn't work that way.

ROS Debugging Messages and Rqt-Console

You have used *print()* during this course to print information about how your programs are doing. **Prints** are the *Dark Side of the Force*, so from now on, you will use a more *Jedi* way of doing things. **LOGS** are the way. Logs allow you to print them on the screen, but also to store them in

the ROS framework, so you can classify, sort, filter, or else.
In logging systems, there are always levels of logging, as shown in {Fig-6.1}. In ROS logs case, there are **five** levels. Each level includes deeper levels. So, for example, if you use **Error** level, all the messages for **Error** and **Fatal** will be shown. If your level is **Warning**, then all the messages for levels **Warning**, **Error** and **Fatal** will be shown.

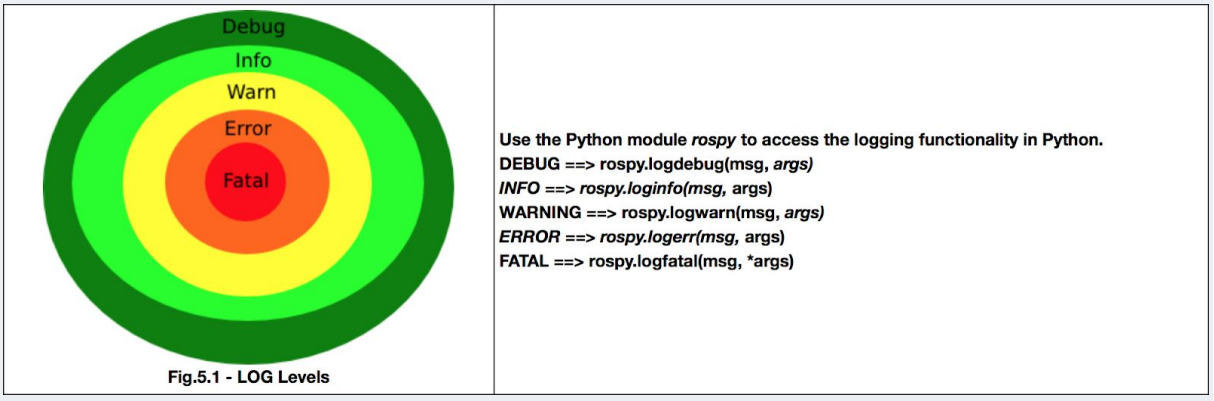


Fig.5.1 - LOG Levels

Execute:

```
#! /usr/bin/env python

import rospy
import random
import time

# Options: DEBUG, INFO, WARN, ERROR, FATAL
rospy.init_node('log_demo', log_level=rospy.DEBUG)
rate = rospy.Rate(0.5)

#rospy.loginfo_throttle(120, "DeathStars Minute info: "+str(time.time()))

while not rospy.is_shutdown():
    rospy.logdebug("There is a missing droid")
    rospy.loginfo("The Emperors Capuchino is done")
    rospy.logwarn("The Revels are coming time "+str(time.time()))
    exhaust_number = random.randint(1,100)
    port_number = random.randint(1,100)
    rospy.logerr(" The thermal exhaust port %s, right below the main port
%s", exhaust_number, port_number)
    rospy.logfatal("The DeathStar Is EXPLODING")
    rate.sleep()
    rospy.logfatal("END")
```

The best place to read all of the logs issued by all of the ROS Systems is: */rosout*

```
rostopic echo /rosout
```

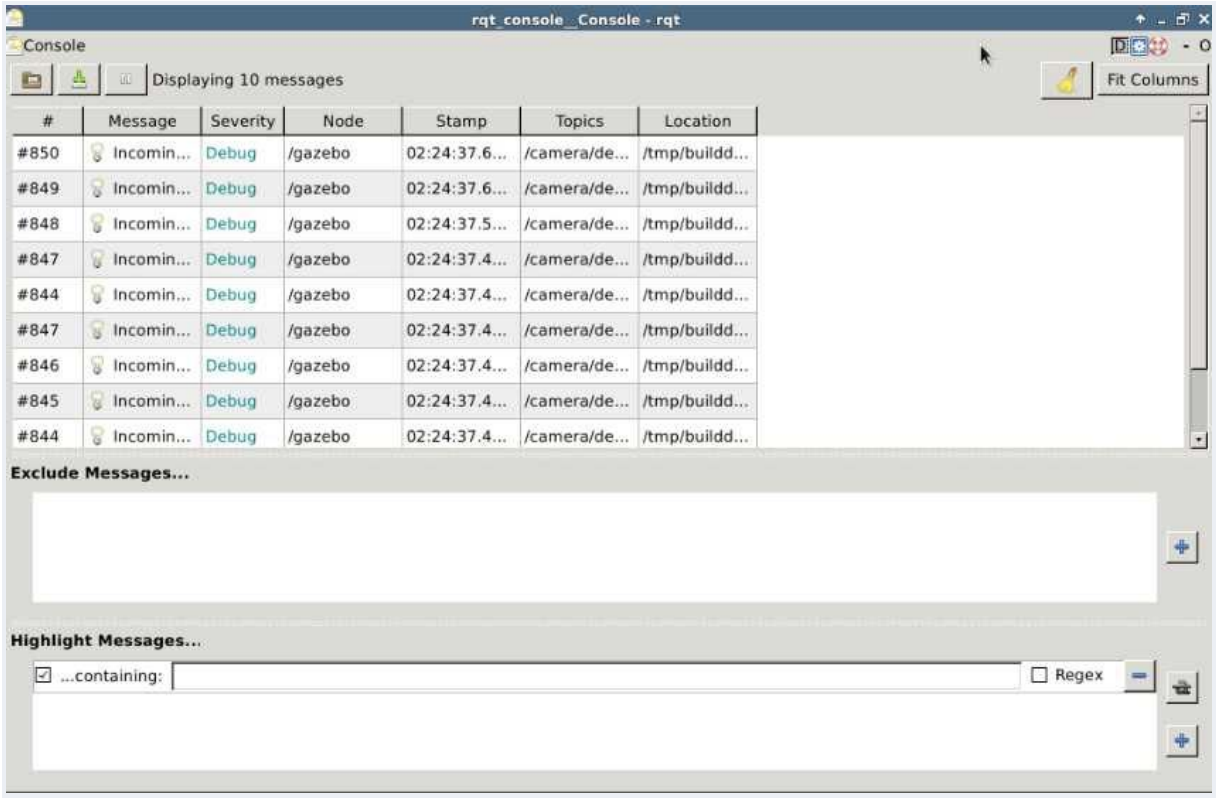
You should see all of the ROS logs in the current nodes, running in the system.

As you can see, with only one node publishing every 2 seconds, the amount of data is big. Now imagine **ten** nodes, publishing image data, laser data, using the actions, services, and publishing debug data of your DeepLearning node. It's really difficult to get the logging data that you want.

That's where **rqt_console** comes to the rescue.

```
roslaunch logger_example_pkg start_logger_example.launch
```

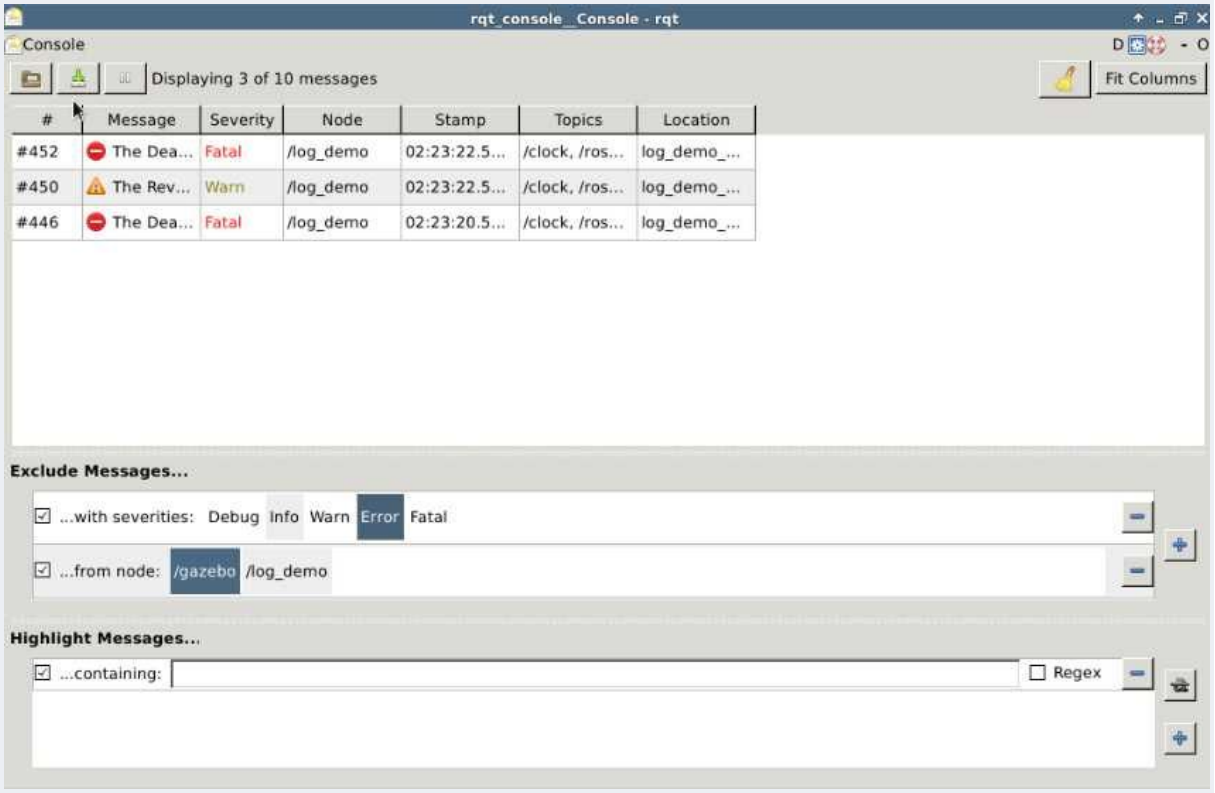
```
rqt_console
```



The rqt_console window is divided into three subpanels.

- The first panel outputs the logs. It has data about the message, severity/level, the node generating that message, and other data. Is here where you will extract all your logs data.
- The second one allows you to filter the messages issued on the first panel, excluding them based on criteria such as: node, severity level, or that it contains a certain word. To add a filter, just press the plus sign and select the desired one.
- The third panel allows you to highlight certain messages, while showing the other ones.

You have to also know that clicking on the tiny white gear on the right top corner, you can change the number of messages shown. Try to keep it as low as possible to avoid performance impact in your system.



Plot topic data and Rqt Plot

This is a very common need in any scientific discipline, but especially important in robotics. You need to know if your inclination is correct, your speed is the right one, the torque readings in an arm joint is above normal, or the laser is having anomalous readings. For all these types of data, you need a graphic tool that makes some sense of all the data you are receiving in a fast and real-time way. Here is where **rqt_plot** comes in handy.

To Continue you should have stopped the Exercice 6.1 node launched in the WebShell #1

- **Go to the WebShell and type the following command to start moving the robot arm:**

```
roslaunch iri_wam_aff_demo start_demo.launch
```

Go to another Webshell and type the following command to see the positions and the effort made by each joint of the robot arm:

```
rostopic echo /joint_states -n1
```

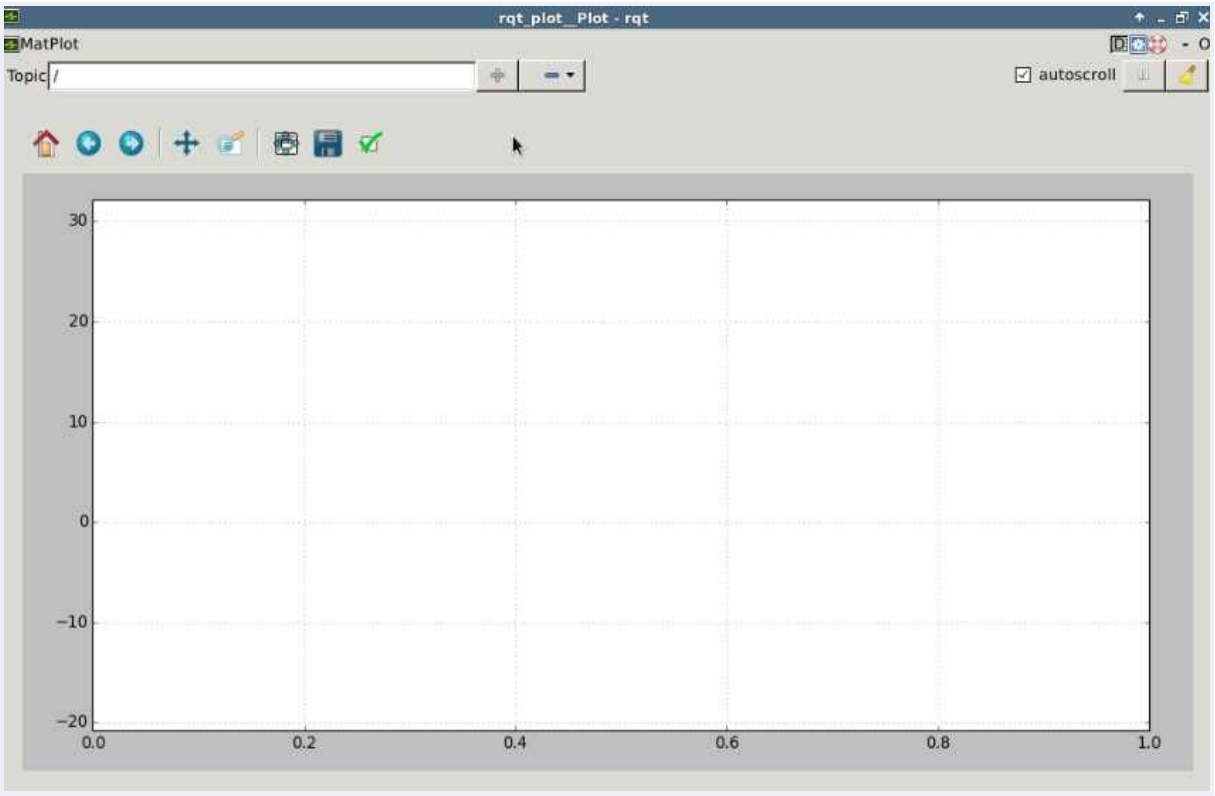
As you can probably see, knowing what's happening in the robots joints with only arrays of numbers is quite daunting.

So let's use the ***rqt_plot*** command to plot the robot joints array of data.

Go to the graphical interface and type in a terminal the following command to open the rqt_plot GUI:

Remember to hit [CTRL]+[C] to stop the rostopic echo

```
rqt_plot
```



In the *topic* input located in the top-left corner of the window, you have to write the topic structure that leads to the data that you want to plot. Bear in mind that in order to be plotted, the topic has to publish a number. Once written, you press the *PLUS SIGN* to start plotting the Topic.

In the case that we want to plot the robot joints, we need to plot the topic */joint_states*, which has the following structure (that you can already get by extracting the topic message type with *rostopic info* , and afterwards using *rosmsg show* command from **Unit 3**):

```
std_msgs/Header header
string[] name
float64[] position
float64[] velocity
float64[] effort
```

Then, to plot the velocity of the first joint of the robot, we would have to type */joint_states/velocity[0]*.

You can add as many plots as you want by pressing the "plus" button.

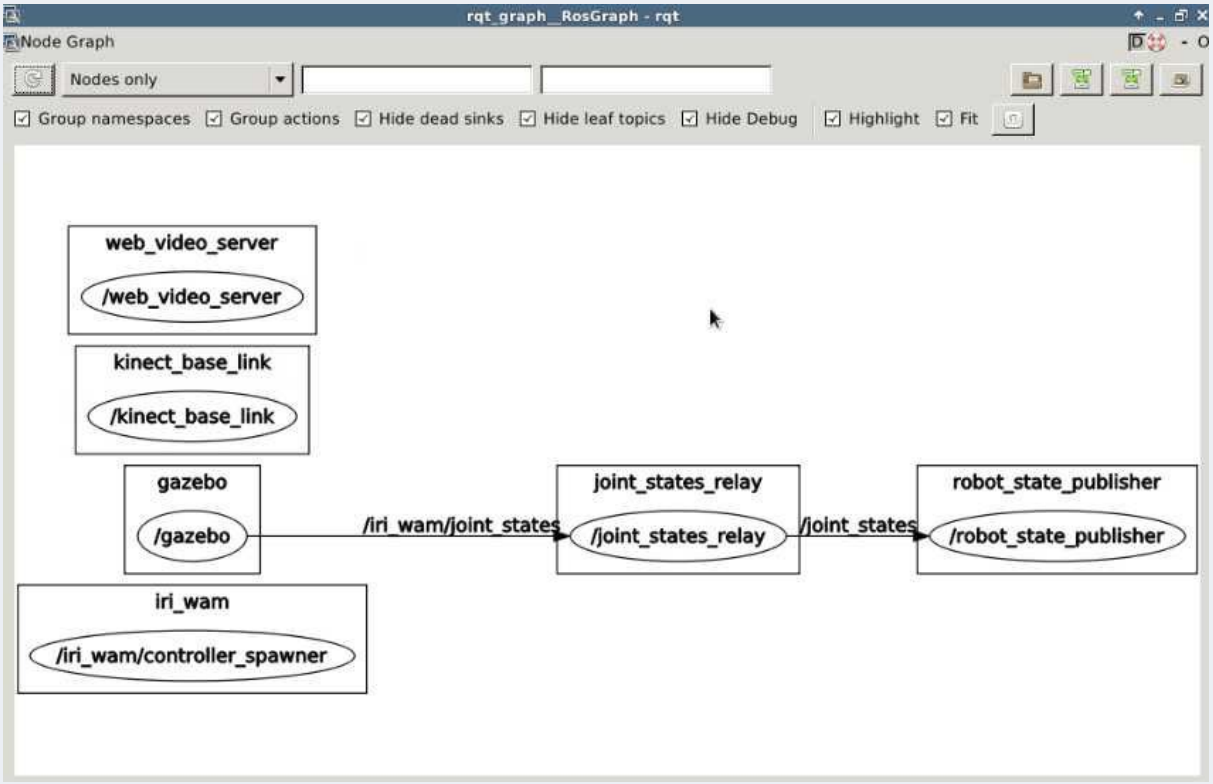
Node Connections and Rqt graph

Is your node connected to the right place? Why are you not receiving data from a topic? These questions are quite normal as you might have experienced already with ROS systems.

Rqt_graph can help you figure that out in an easier way. It displays a visual graph of the nodes running in ROS and their topic connections. It's important to point out that it seems to have problems with connections that aren't topics.
Go to the graphical interface and type in a terminal the following command to open the rqt_graph GUI:

Remember to have in the WebShell #1 the roslaunch iri_wam_aff_demo start_demo.launch

```
rqt_graph
```

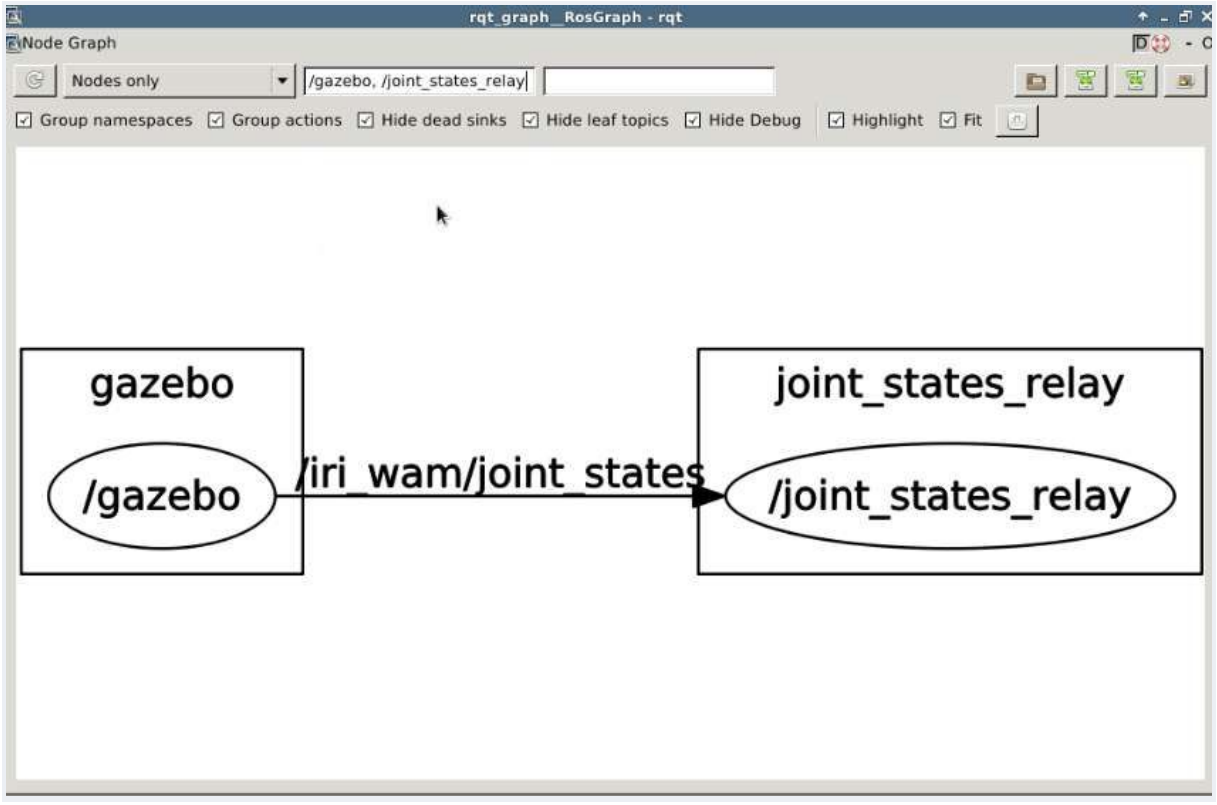


In the diagram {Fig-6.5}, you will be presented with all of the nodes currently running, connected by the topics they use to communicate with each other. There are two main elements that you need to know how to use:

- The **refresh button**: Which you have to press any time you have changed the nodes that are running:

- The **filtering options**: These are the three boxes just beside the refresh button. The first element lets you select between only nodes or topics. The second box allows you to filter by names of nodes.

Here is an example where you filter to just show the */gazebo* and the */joint_states_relay* {Fig-6.6}:



Record Experimental Data and Rosbags

One very common scenario in robotics is the following:

You have a very expensive real robot, let's say R2-D2, and you take it to a very difficult place to get, let's say The Death Star. You execute your mission there and you go back to the base. Now, you want to reproduce the same conditions to improve R2-D2s algorithms to open doors. But you don't have the DeathStar nor R2-D2. How can you get the same exact sensory readings to make your test? Well, you record them, of course! And this is what **rosbag** does with all the ROS topics generated. It records all of the data passed through the ROS topics system and allows you to replay it any time through a simple file.
The commands for playing with **rosbag** are:

- To **Record** data from the topics you want:

  rosbag record -O name_bag_file.bag name_topic_to_record1 name_topic_to_record2 ... name_topic_to_recordN

- To **Extract** general information about the recorded data:

  rosbag info name_bag_file.bag

- To **Replay** the recorded data:

  rosbag play name_bag_file.bag

> Replaying the data will make the rosbag publish the same topics with the same data, at the same time when the data was recorded.

1- Go to the WebShell and type the following command to make the robot start moving if you don't haven't already:

```
roslaunch iri_wam_aff_demo start_demo.launch
```

2- Go to another WebShell and go to the *src* directory. Type the following command to record the data from the */laser_scan*:

```
roscd; cd ../src
rosbag record -O laser.bag laser_scan
```

This last command will start the recording of data.

3- After 30 seconds or so, a lot of data have been recorded, press [CTRL]+[C] in the rosbag recording WebShell #2 to stop recording. Check that there has been a laser.bag file generated and it has the relevant information by typing:

```
rosbag info laser.bag
```

4- Once checked, CTRL+C on the start_demo.launch WebShell #1 to stop the robot.

5- Once the robot has stopped, type the following command to replay the *laser.bag* (the *-l* option is to loop the *rosbag* infinitely until you CTRL+C):

```
rosbag play -l laser.bag
```

6- Go to WebShell #3 and type the following command to read the ranges[100] of Topic */laser_scan* :

```
rostopic echo /laser_scan/ranges[100]
```

7- Type the following command in another WebShell , like WebShell #4. Then, go to the graphical interface and see how it plots the given data with rqt_plot:

```
rqt_plot /laser_scan/ranges[100]
```

Is it working? Did you find something odd?
There are various things that are wrong, but it's important that you memorize this.

1) The first thing you notice is that when you echo the topic ***/laser_scan*** topic, you get sudden changes in values.

Try getting some more information on who is publishing in the ***/laser_scan*** topic by writing in the WebShell:

Remember to hit [CTRL]+[C] if there was something running there first.

```
rostopic info /laser_scan
```

```
user ~ $ rostopic info /laser_scan
Type: sensor_msgs/LaserScan

Publishers:
 * /gazebo (http://ip-172-31-27-126:59384/)
 * /play_1476284237447256367 (http://ip-172-31-27-126:41011/)

Subscribers: None
```

As you can see, **TWO** nodes are publishing in the **/laser_scan** topic: **gazebo** (the simulation) and **play_x** (the rosbag play).

This means that not only is rosbag publishing data, but also the simulated robot.
So the first thing to do is to **PAUSE** the simulation, so that gazebo stops publishing laser readings.
For that, you have to execute the following command:

```
rosservice call /gazebo/pause_physics "{}"
```

And to **UnPAUSE** it again, just for you to know, just:

```
rosservice call /gazebo/unpause_physics "{}"
```

With this, you should have stopped all publishing from the simulated robot part and only left the rosbag to publish.
NOTE: When you execute the command to pause Gazebo physics, the node **/gazebo** will stop publishing into the **/laser_scan** topic. However, if you execute a *rostopic info* command and check the laser topic, you will still see the **/gazebo** node as a Publisher, which may cause confusion. This happens because ROS hasn't updated the data yet. But do not worry, because **/gazebo** will not be publishing into the laser topic.
Now you should be able to see a proper **/laser_scan** plot in the rqt_plot.
Still nothing?
2) Check the time you have in the rqt_plot. Do you see that, at a certain point, the time doesn't keep on going?
That's because you have stopped the simulation so that the time is not running anymore, apart from the tiny time frame in the rosbag that you are playing now.

Once rqt_plot reaches the maximum time, it stops. It doesn't return to the start, and therefore, if the values change outside the last time period shown, you won't see anything.
Take a look also at the rosbag play information of your currently running rosbag player

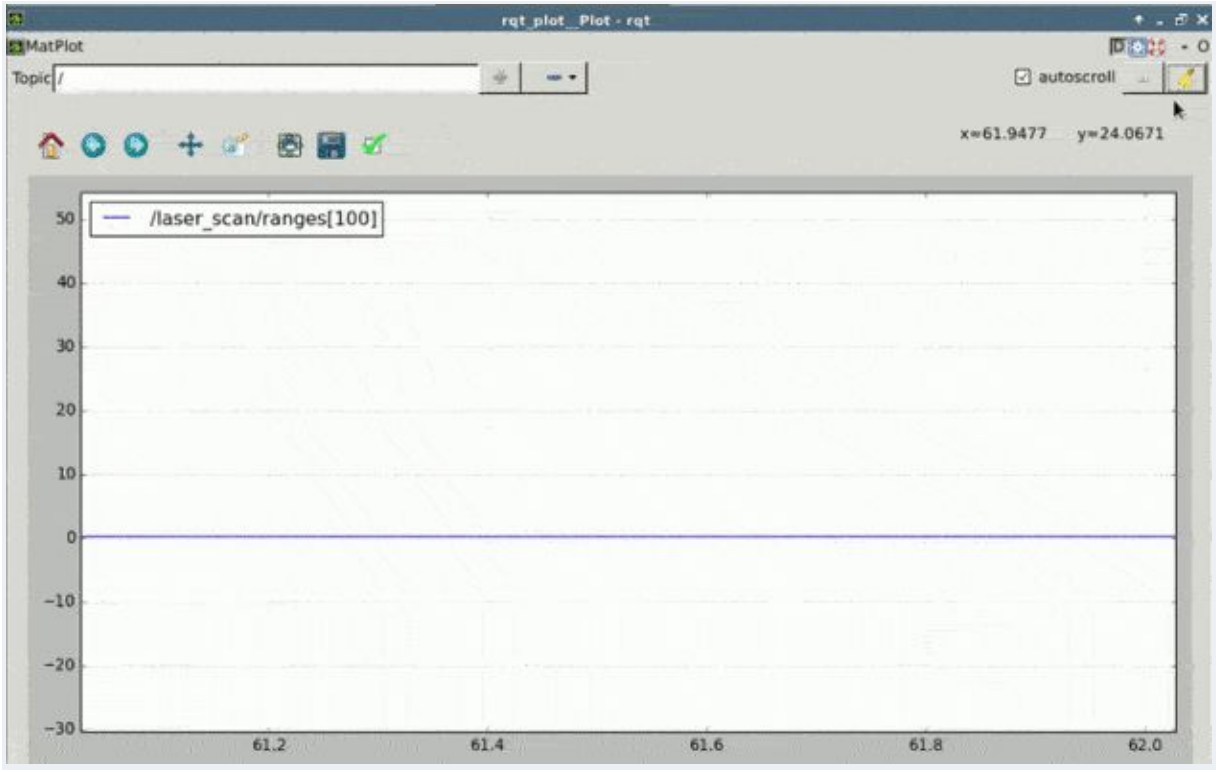
```
user ~ $ rosbag play -l laser.bag
[ INFO ] [1471001445.5575545086]: Opening laser.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to stop.
[RUNNING] Bag Time: 61.676140 Duration: 41.099140 / 41.452000
```

In this example, you can see that the current time is **41.099** of **41.452 seconds** recorded. And the rosbag time, therefore, will reach a maximum of around **62** seconds.

62 seconds, in this case, is the maximum time the rqt_plot will show, and will start around 20 seconds.

Therefore, you have to always **CLEAR** the plot area with the **clear button** in rqt_plot.

By doing a **CLEAR**, you should get something similar to {Fig-6.8}:



To summarize:

To use rosbag files, you have to make sure that the original data generator (real robot or simulation) is NOT publishing. Otherwise, you will get really weird data (the collision between the

original and the recorded data). You have to also keep in mind that if you are reading from a rosbag, time is finite and cyclical, and therefore, you have to clear the plot area to view all of the time period.
rosbag is especially useful when you don't have anything in your system (neither real nor simulated robot), and you run a bare **roscore**. In that situation, you would record all of the topics of the system when **you do have** either a simulated or real robot with the following command:

```
rosbag record -a
```

This command will record **ALL** the topics that the robot is publishing. Then, you can replay it in a bare roscore system and you will get all of the topics as if you had the robot.
Before continuing any further, please check that you've done the following:

- You have stopped the rosbag play by going to the WebShell #2 where it's executing and [CTRL]+[C]
- You have unpaused the simulation to have it working as normal:

```
rosservice call /gazebo/unpause_physics "{}"
```

Visualize Complex data and Rviz

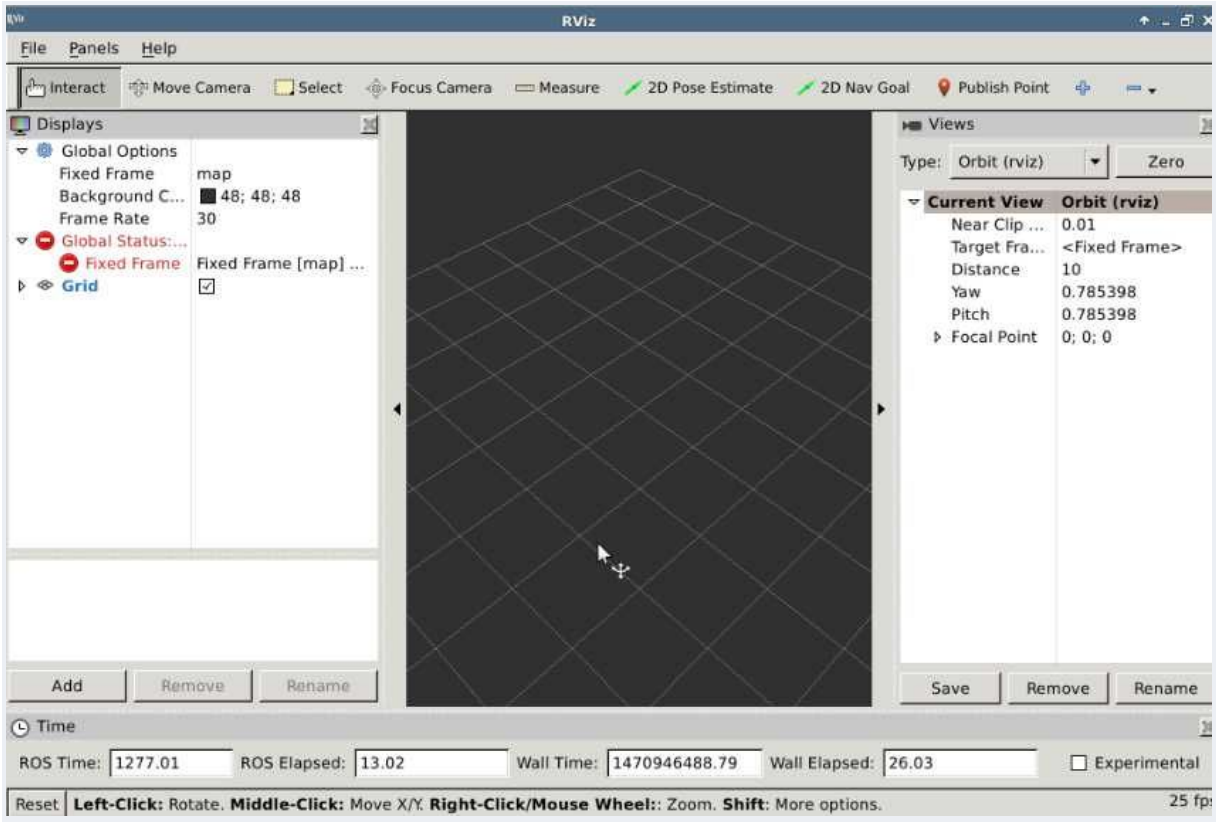
And here you have it. The **HollyMolly!** The Milenium Falcon! The most important tool for ROS debugging....**RVIZ**.

RVIZ is a tool that allows you to visualize *Images, PointClouds, Lasers, Kinematic Transformations, RobotModels*...The list is endless. You even can define your own markers. It's one of the reasons why ROS got such a great acceptance. Before RVIZ, it was really difficult to know what the Robot was perceiving. And that's the main concept:

RVIZ is **NOT** a simulation. I repeat: It's **NOT** a simulation.
RVIZ is a representation of what is being published in the topics, by the simulation or the real robot.
RVIZ is a really complex tool and it would take you a whole course just to master it. Here, you will get a glimpse of what it can give you.
Remember that you should have unpaused the simulations and stopped the rosbag as described in the rosbag section.
1- Type in WebShell #2 the following command:

```
rosrun rviz rviz
```

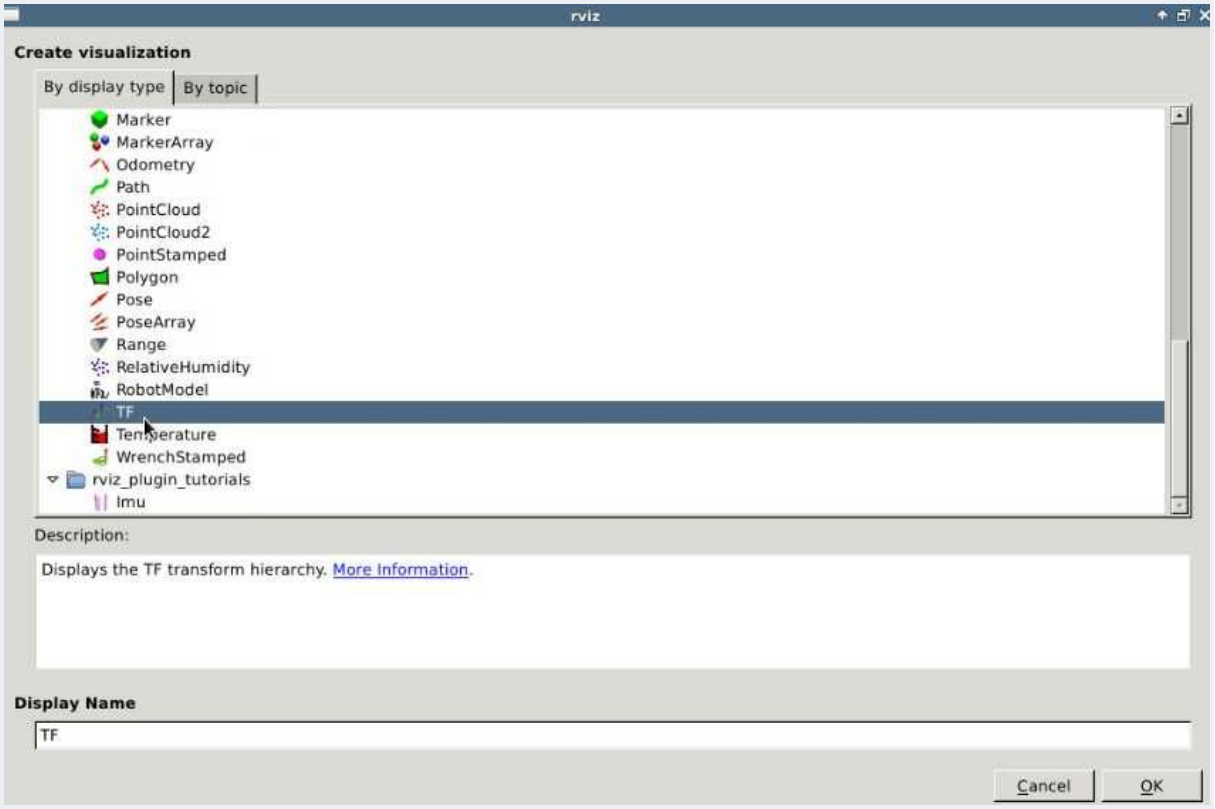
2- Then go to the graphical interface to see the RVIZ GUI:
You will be greeted by a window like {Fig-6.9}:



NOTE: In case you don't see the lower part of Rviz (the Add button, etc.), double-click at the top of the window to maximize it. Then you'll see it properly.
You need only to be concerned about a few elements to start enjoying RVIZ.

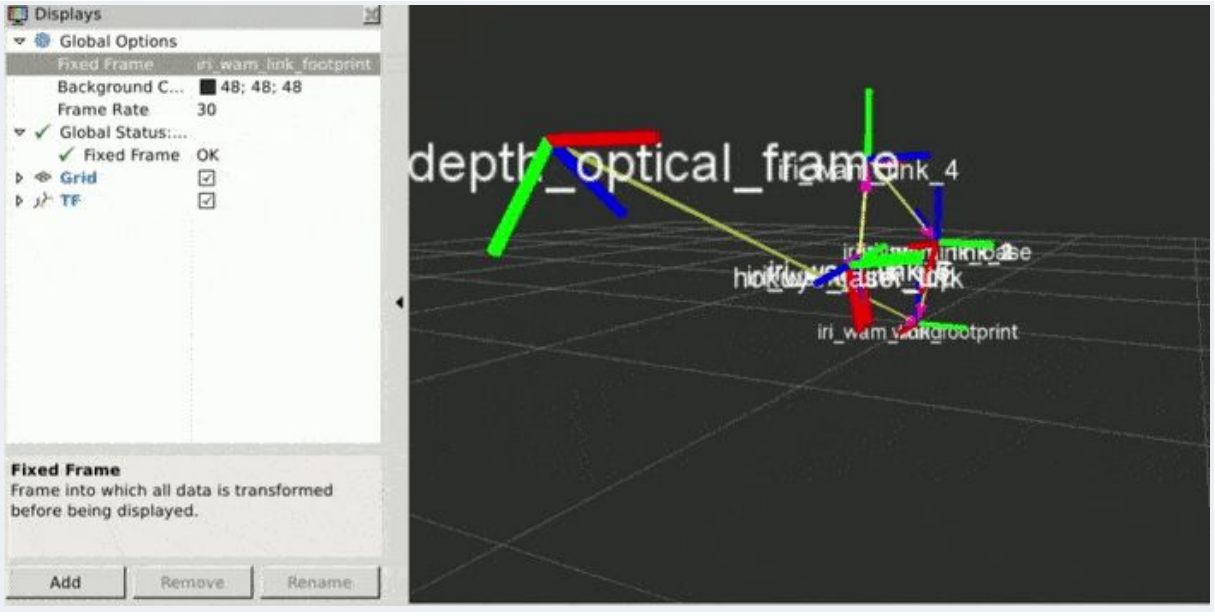
- **Central Panel**: Here is where all the magic happens. Here is where the data will be shown. It's a 3D space that you can rotate (LEFT-CLICK PRESSED), translate (CENTER MOUSE BUTTON PRESSED) and zoom in/out (LEFT-CLICK PRESSED).
- **Left Displays Panel**: Here is where you manage/configure all the elements that you wish to visualize in the central panel. You only need to use two elements:
- In **Global Options**, you have to select the **Fixed Frame** that suits you for the visualization of the data. It is the reference frame from which all the data will be referred to.
- The **Add button**. Clicking here you get all of the types of elements that can be represented in RVIZ.

Go to RVIZ in the graphical interface and add a TF element. For that, click "Add" and select the element TF in the list of elements provided, as shown in {Fig-6.10}.



- Go to the RVIZ Left panel, select as Fixed Frame the ***iri_wam_link_footprint*** and make sure that the **TF** element checkbox is checked. In a few moments, you should see all of the Robots Elements Axis represented in the CENTRAL Panel.
- Now, go to a WebShell #1 and enter the command to move the robot:

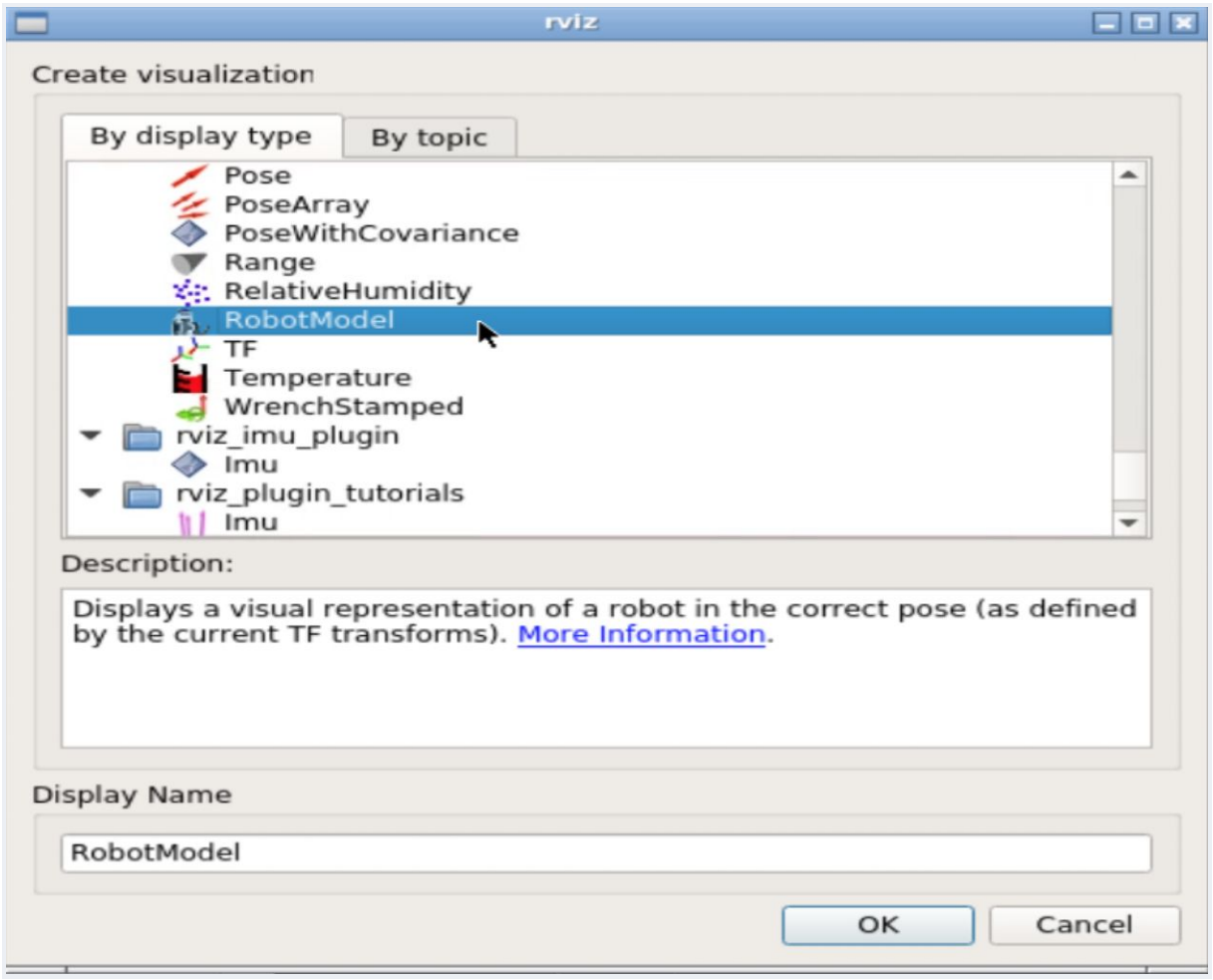
```
roslaunch iri_wam_aff_demo start_demo.launch
```



In {Fig-6.11}, you are seeing all of the transformations elements of the IRI Wam Simulation in

real-time. This allows you to see exactly what joint transformations are sent to the robot arm to check if it's working properly.

- Now press "Add" and select *RobotModel*, as shown in {Fig-6.12}



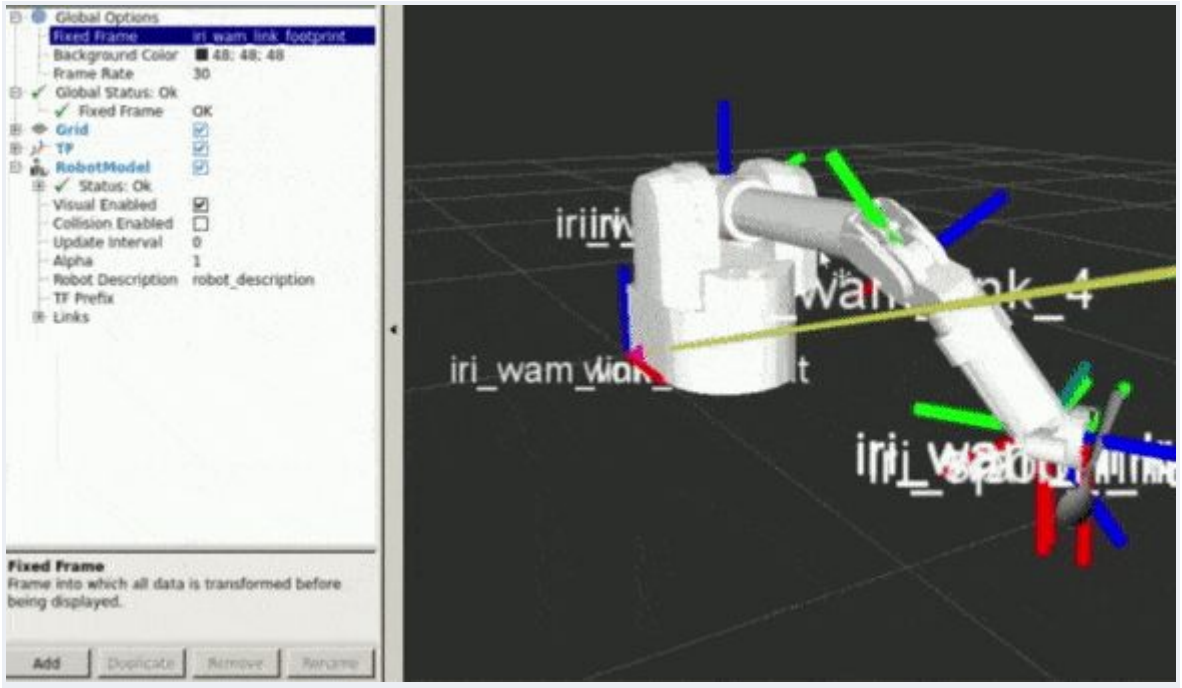
You should see now the 3D model of the robot, as shown in

Why can't you see the table? Or the bowl? Is there something wrong? Not at all!

Remember: RVIZ is **NOT** a simulation, **it represents what the TOPICS are publishing**. In this case the models that are represented are the ones that the *RobotStatePublisher node* is publishing in some ROS topics. There is NO node publishing about the bowl or the table.

Then how can you see the object around? Just like the robot does, through cameras, lasers, and other topic data.

Remember: RVIZ shows what your robot is perceiving, nothing else.


Global Options
Fixed Frame
iri_wam_link_footprint
Background Color
48; 48; 48
Frame Rate
30
Global Status: Ok
Fixed Frame
OK
Grid
TF
RobotModel
Status: Ok
Visual Enabled
Collision Enabled
Update Interval
0
Alpha
1
Robot Description
robot_description
TF Prefix
Links
Fixed Frame
Frame into which all data is transformed before being displayed.
Add
Duplicate
Remove
Rename