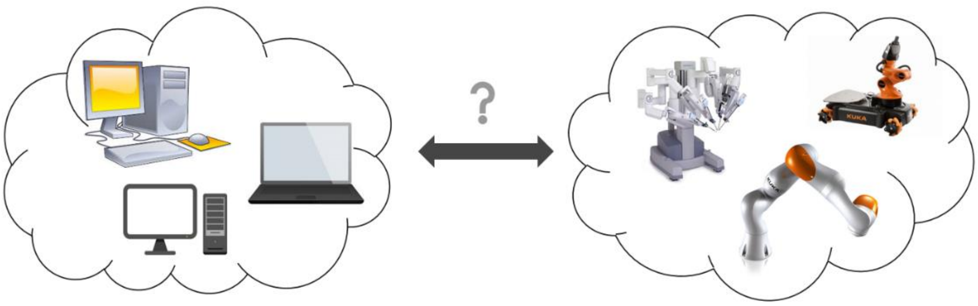
Contents

1 Introduction to ROS

1.1 What is ROS?

ROS stands for 'Robot Operating System' and its official wiki description is 'ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.'



In a few words, ROS is an open-source framework for robotics, whose main goal is to support code reuse and promote collaboration in robotics research and development. It's not an actual operating system, but more of a 'middleware', an abstraction layer that resides between the operating system and software applications.

1.2 Why is ROS useful?

ROS can be broken down to its four main characteristics:

- Plumbing: ROS provides publish-subscribe messaging infrastructure designed to support quick and easy construction of distributed computing systems
- Tools: ROS provides an extensive set of tools for configuring, starting, introspecting, debugging, visualizing, logging, testing, and stopping distributed computing systems.
- Capabilities: ROS provides a broad collection of libraries that implement useful robot functionality, with a focus on mobility, manipulation and perception.
- Ecosystem: ROS is supported and improved by a large community, with a strong focus on integration and documentation.

To sum up, using ROS helps into tackling some of the many issues that one often finds in developing robots such as cooperation between hardware and software, architectural differences in robotic systems and software modularity or reusability.

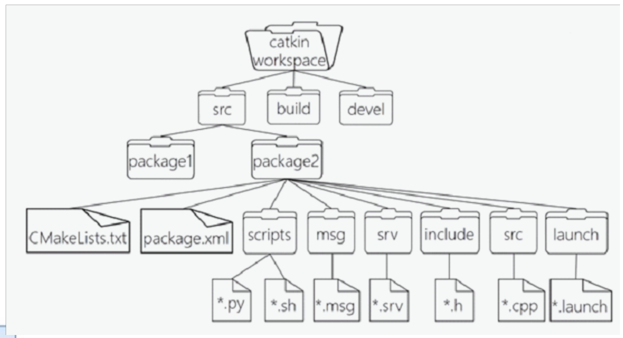
1.3 ROS Pros and Cons

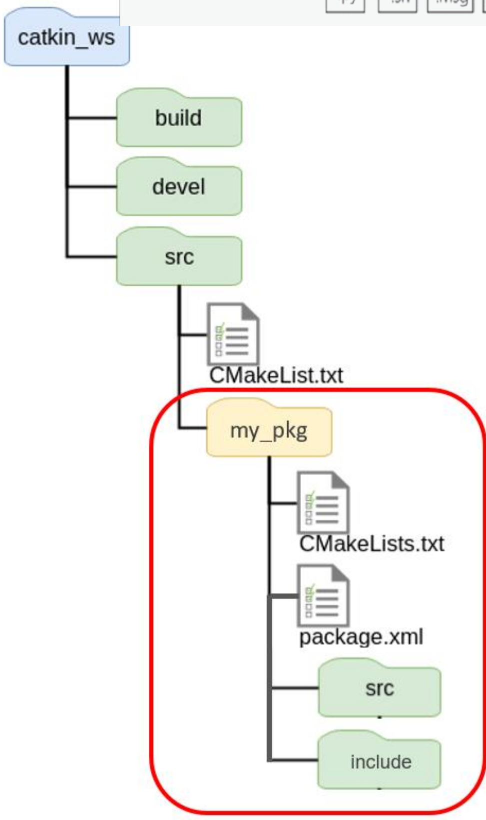
- ✓ Takes care of lots of low-level issues (communication, drivers, etc)
- ✓ Modular design and software reusability
- ✓ Open-source and great community support
- ✓ Large number of ready-to-use packages available

- ✗ It is not real-time
- ✗ Modular design may be inefficient compared to integrated design
- ✗ Running ROS code requires additional devices capable of running ROS

1.4 ROS components

1.4.1 Packages





A ROS package is a coherent collection of files that serves a specific purpose and all ROS software is organized into packages. A single package contains a source folder, an include folder, a CMakeList.txt file and a package.xml file.

The source folder consists of the source files whereas the include folder of the header files. The other two files are a bit more complicated.

The file CMakeLists.txt is the input to the CMake build system for building software packages. Any CMake-compliant package contains one or more CMakeLists.txt file that describe how to build the code and where to install it to. The overall structure of a CMakeLists.txt file is:

1. Required CMake Version (cmake_minimum_required)
2. Package Name (project())
3. Find other CMake/Catkin packages needed for build (find_package())
4. Enable Python module support (catkin_python_setup())
5. Message/Service/Action Generators(add_message_files(), add_service_files(), add_action_files())
6. Invoke message/service/action generation (generate_messages())
7. Specify package build info export (catkin_package())
8. Libraries/Executables to build (add_library()/add_executable()/target_link_libraries())
9. Tests to build (catkin_add_gtest())
10. Install rules (install())

The package manifest is an XML file called package.xml that must be included with any catkin-compliant package's root folder. This file defines properties about the package such as the package name, version numbers, authors, maintainers, and dependencies on other catkin packages. An example of a package.xml file for a ‘foo_core’ package is the following:

```
<package format="2">

    <name>foo_core</name>
    <version>1.2.4</version>
    <description>
    This package provides foo capability.
    </description>
    <maintainer email="email@email.org">Name Surname</maintainer>
    <license>BSD</license>

</package>
```

Some useful console ROS package commands are:

`sudo apt-get install ros-kinetic-package`	-> to install a single package
`catkin_create_pkg pkg_name dep1 dep2`	-> to create a single package in a catkin_ws
`rospack help [options] package_name`	-> help menu
`rospack list`	-> display list of all packages
`rospack find package_name`	-> return the absolute path to a package
`rospack depends package_name`	-> display a list of all the pkg dependencies

1.4.2 Catkin Workspace

A catkin workspace is a folder where you modify, build and install catkin packages. It has four or five sub-folders: source, build, devel and install.

The source space contains the source code of catkin packages. This is where you can extract/checkout/clone/source code for the packages you want to build. Each folder within the source space contains one or more catkin packages. The build space is where CMake is invoked to build the catkin packages in the source space. CMake and catkin keep their cache information and other intermediate files here. The development space (or devel space) is where built targets are placed prior to being installed. The way targets are organized in the devel space is the same as their layout when they are installed. Once targets are built, they can be installed into the install space by invoking the install target.

To create a Catkin Workspace type in a console:

```
source /opt/ros/kinetic/setup.bash
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make
source devel/setup.bash
```

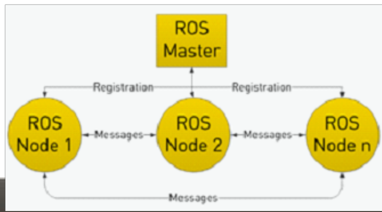
‘catkin_make’ is a convenience tool for building code in a catkin workspace.

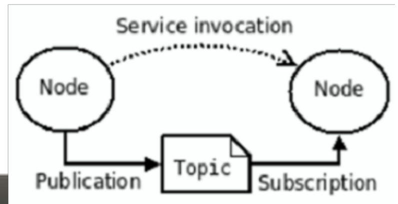
‘source devel/setup.bash’ is used to source your setup.bash file. By sourcing this file, you are adding several environment variables that ROS needs in order to work.

1.4.3 Master

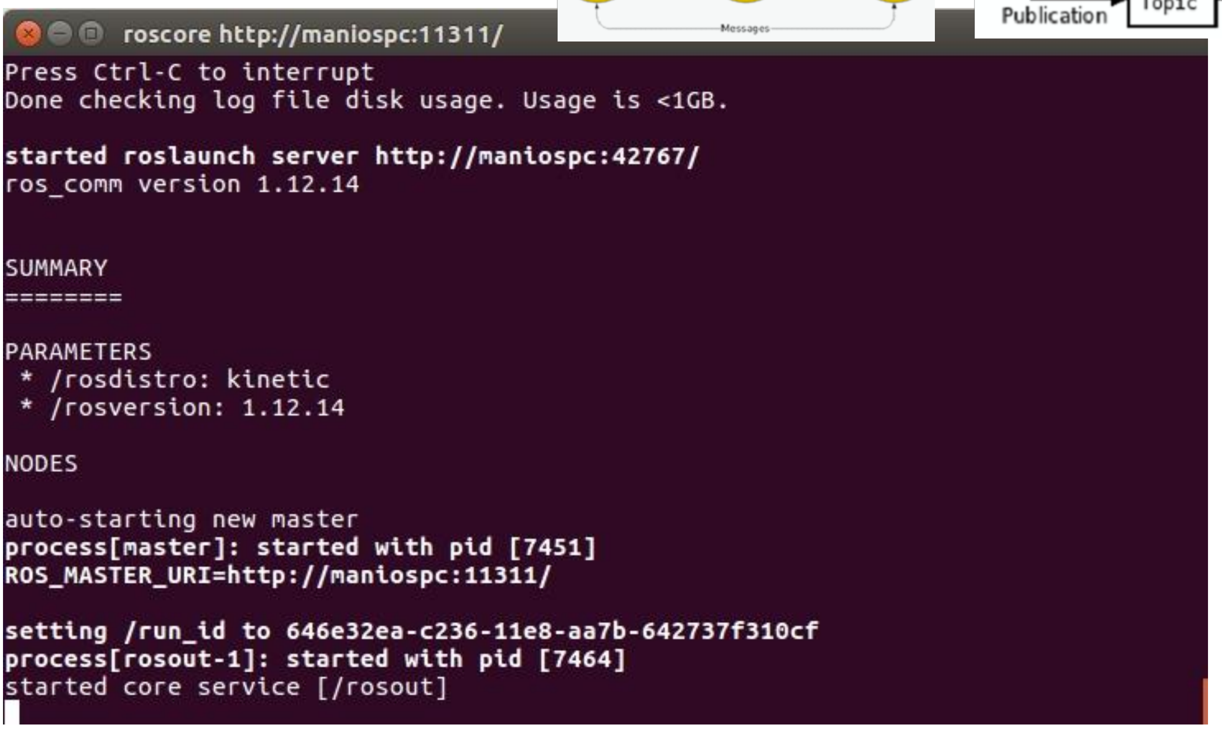
The ROS Master is part of the so-called ‘ROS core’, a set of three programs that are necessary for the ROS runtime. The master negotiates communication connections, provides name registration and lookup for ROS graph resources. The master should continue running for the entire time that ROS is being used. One reasonable workflow is to start the master in one terminal, then open other terminals for your work.

To start the master type in a console:






```
roscore
```



```
roscore http://maniospc:11311/
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://maniospc:42767/
ros_comm version 1.12.14


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [7451]
ROS_MASTER_URI=http://maniospc:11311/

setting /run_id to 646e32ea-c236-11e8-aa7b-642737f310cf
process[rosout-1]: started with pid [7464]
started core service [/rosout]
```

And to kill it just press Ctrl+C.

1.4.4 Nodes

ROS nodes are processes that perform computation to absolve some task (eg read sensors or control motors) and communicate with other nodes using the ROS infrastructure. Nodes are implemented using client libraries. Rospy is the Python library and roscpp is the C++ library.

To run a node type in a console:

```
rosrun package_name executable_name
```

Both those names are defined in the CMakeLists.txt file.

To get information about nodes that are already running:

```
rosnode <command>
```

such as rosnode list, or rosnode info node_name. For example, if we ask for information about the 'listener' node that we run in the first example of Lab02 then we get:

```
manios@maniospc: ~
manios@maniospc:~$ rosnode info /listener
--------------------------------------------------------------------------------
Node [/listener]
Publications:
 * /rosout [rosgraph_msgs/Log]

Subscriptions:
 * /chatter [std_msgs/String]

Services:
 * /listener/get_loggers
 * /listener/set_logger_level


contacting node http://maniospc:35657/ ...
Pid: 5866
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound
    * transport: TCPROS
 * topic: /chatter
    * to: /talker (http://maniospc:43881/)
    * direction: inbound
    * transport: TCPROS

manios@maniospc:~$
```

1.4.5 Topics

Topics are named channels for communication, where messages of the same kind are exchanged. Communication between nodes follows the publish/subscribe paradigm: a node sends out a message by publishing it to a topic, and another node reads it by subscribing to that same topic. Multiple nodes can publish messages on a topic and multiple nodes can also read messages from a topic. Having said that, it is important to mention that communication is asynchronous, meaning that it doesn't take into consideration the timing or the ordering of the messages.

Some useful console ROS package commands are:

rostopic <command> [options]	-> to get information about ROS topics
rostopic list	-> list of active topics

rostopic echo topic_name	-> print messages to screen
rostopic hz topic_name	-> display publishing rate of topic
rostopic type topic_name	-> display type about active topic
rorsopic info topic_name	-> print information about active topic
rostopic pub –r rate_hz topic_name message_type message_content	-> print information about active topic

1.4.6 Messages

P(x, y, z, v t)

Messages are used by nodes to communicate with each other. A message is a data structure, comprising typed fields (predefined ROS messages, custom messages). A message can include a timestamp, something especially useful when dealing with sensor data. An example of a message is the following:

std_msgs/Header.msg
uint32 seq
time stamp
string frame_id

std_msgs/String.msg
string.data

1.4.7 Services

The publish/subscribe model is a very flexible communication paradigm, but its many-to-many one-way transport is not appropriate for RPC request/reply interactions, which are often required in a distributed system. Request/reply is done via a service, which is defined by a pair of messages: one for the request and one for the reply. A providing ROS node offers a service under a string name, and a client calls the service by sending the request message and awaiting the reply. Client libraries usually present this interaction to the programmer as if it were a remote procedure call. Services are defined using srv files, which are compiled into source code by a ROS client library.

Some useful console ROS service commands are:

rosservice list	-> to get a list of services that are currently active
rosnode info node_name	-> to see all services offered by a particular node
rosservice node service_name	-> to find a node offering a specific service
rosservice info service_name	-> to determine the service data

1.4.8 Launchfiles

The launch file is a mechanism that ROS provides for starting the master and many nodes all at once. It includes information on the nodes and options to automatically respawn processes that have already died. The use of launch files is widespread throughout many ROS packages since those files are a very convenient way to specify and configure which nodes are used in a package with multiple nodes.

An example of a launch file that starts two nodes at once is the following:

```
<launch>
  <node
    pkg="turtlesim"
    type="turtlesim_node"
    name="turtlesim"
    respawn="true"
  />
  <node
    pkg="turtlesim"
    type="turtle_teleop_key"
    name="teleop_key"
    required="true"
    launch-prefix="xterm -e"
  />
</launch>
```

To run a launch file type in the console:

xml

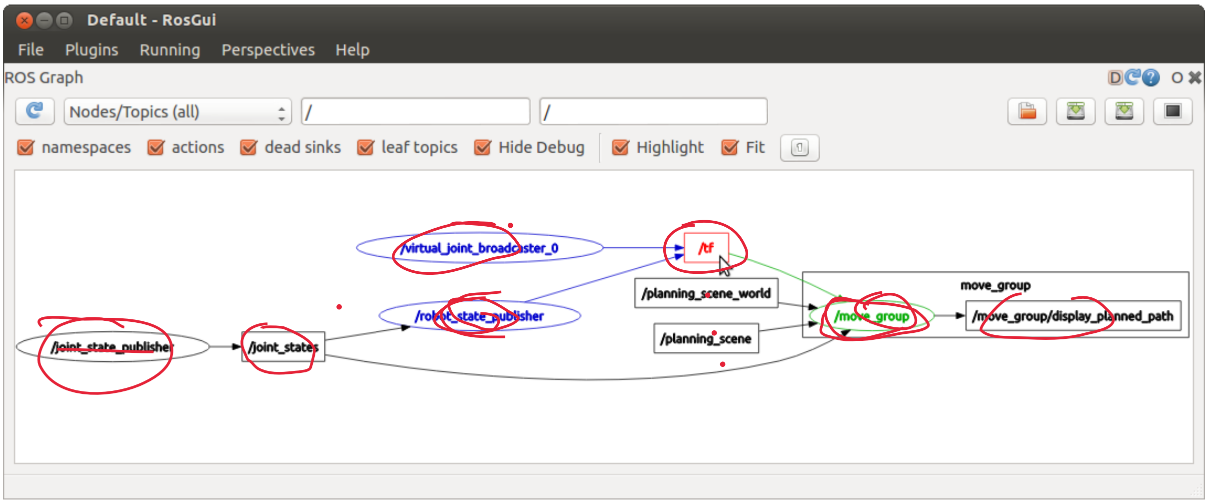
```
roslaunch package_name file.launch arg1:=input1 arg2:=input2
```

An efficient way to work with launch files is using the <arg> tag. The <arg> tag allows you to create more re-usable and configurable launch files by specifying values that are passed via the command-line, passing in via an <include>, or declared for higher-level files. Args are not global. An arg declaration is specific to a single launch file, much like a local parameter in a method and you must explicitly pass arg values to an included file.

Similar to the <arg> tag is the <param> tag. The <param> tag defines a parameter to be set on the Parameter Server. Instead of value, you can specify a textfile, binfile or command attribute to set the value of a parameter. The <param> tag can be put inside of a <node> tag, in which case the parameter is treated like a private parameter.

1.4.9 rqt_graph/rqt_plot

rqt_graph provides a GUI plugin for visualizing the ROS computation graph. Its components are made generic so that other packages where you want to achieve graph representation can depend upon this package. Very similarly, rqt_plot provides a GUI plugin visualizing numeric values in a 2D plot using different plotting backends.



To run rqt_graph simply type: rqt_graph into a console.

To run rqt_plot you must run:

```
rosdep install rqt_plot                    -> for the first time only
rqt_plot                                   -> for every other time
```

1.5 Resources

As mentioned in the start of this introduction, one of the main advantages of ROS is its open-source character and its wide online support.

https://wiki.ros.org - The ROS Wiki is a massive database with Documentation on a variety of resources, from install instructions, to tutorials and publications.

https://www.cse.sc.edu/~jokane/agitr/agitr-letter.pdf - ‘A gentle introduction to ROS’ by Jason M. O’Kane, University of South Carolina is an extensive look into getting started with ROS.

2 Examples

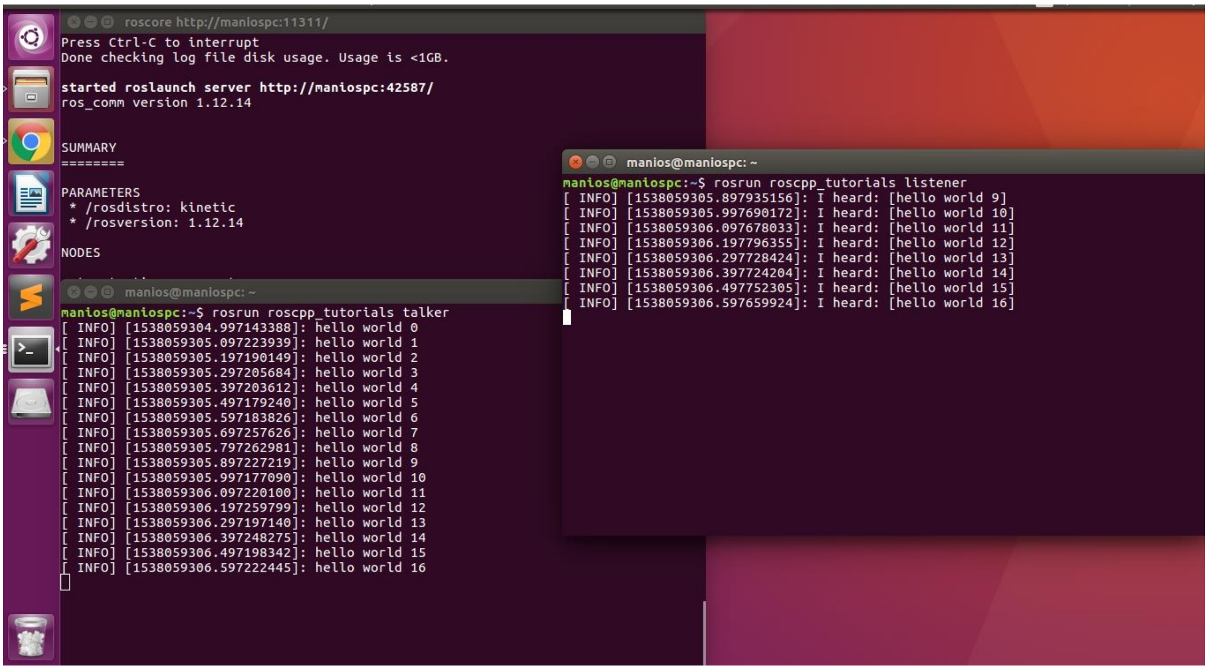
Now we are going to see some of the things above in action.

2.1 Talker – listener example

Run the three commands you see below, each one in a different terminal:

```
roscore

rosrun roscpp_tutorials talker

rosrun roscpp_tutorials listener
```

After we initiated the ROS master with ‘roscore’, we then ran two nodes of the same ‘roscpp_tutorials’ package. The ‘talker’ node is the publisher which publishes a simple ‘hello world’ string, and the ‘listener’ node, the subscriber, subscribes to it and receives the message.



On another terminal type

```
rostopic list
```

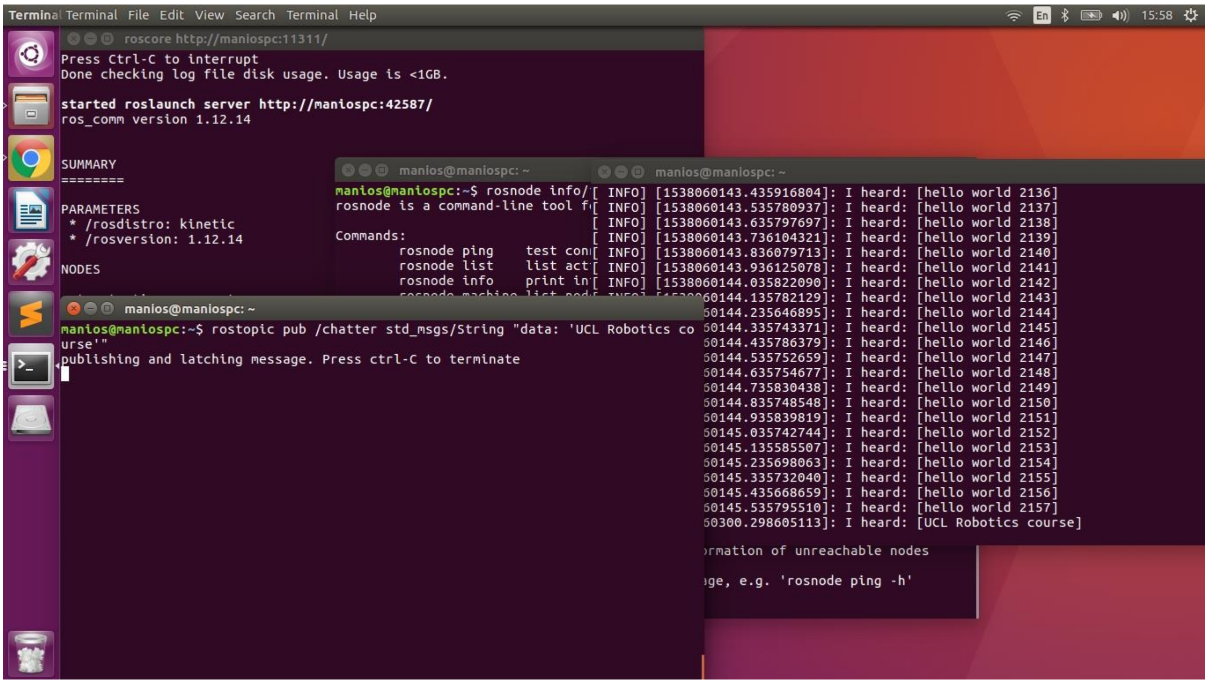
We can see three topics appearing. The one we are interested in is ‘/chatter’ because that is the one that the two nodes use.

We can also get information on the two nodes by typing:

```
rosnode info /talker

rosnode info /listener
```

Now kill the ‘talker’ node by pressing ‘Ctrl+C’ because will publish a message on the subscriber node ourselves. Type:

```
rostopic pub /chatter std_msgs/String ‘data: UCL robotics course’
```



Terminate every console, including the master, by pressing ‘Ctrl+C’.

2.2 TurtleBot example

TurtleBot is an open source hardware platform and mobile base. When powered by ROS, TurtleBot can handle vision, localization, communication and mobility. In this example we will run the simulation of the TurtleBot robot in Gazebo, and teleoperate it with the keyboard.

Gazebo is a 3D rigid body simulator for robots. A set of packages (gazebo_ros_pkgs) is a set of ROS packages that provide the necessary interfaces to simulate a robot in the Gazebo 3D environment. It integrates with ROS using ROS messages, services and dynamic reconfigure. There is no need for a standalone installation of Gazebo since it’s part of the full ROS installation.

First, we need to download and ‘catkin_make’ the necessary packages to run the TurtleBot simulation. To do this, we are going to use the git control system. Without going into much detail, Git is a version-control system for tracking changes in computer files and coordinating work on those files among multiple people. It is primarily used for source-code management in software development, but it can be used to keep track of changes in any set of files.

Instead of manually downloading and saving a specific set of files (repository) from their github.com page, you can directly 'clone' this repository into your desired location.

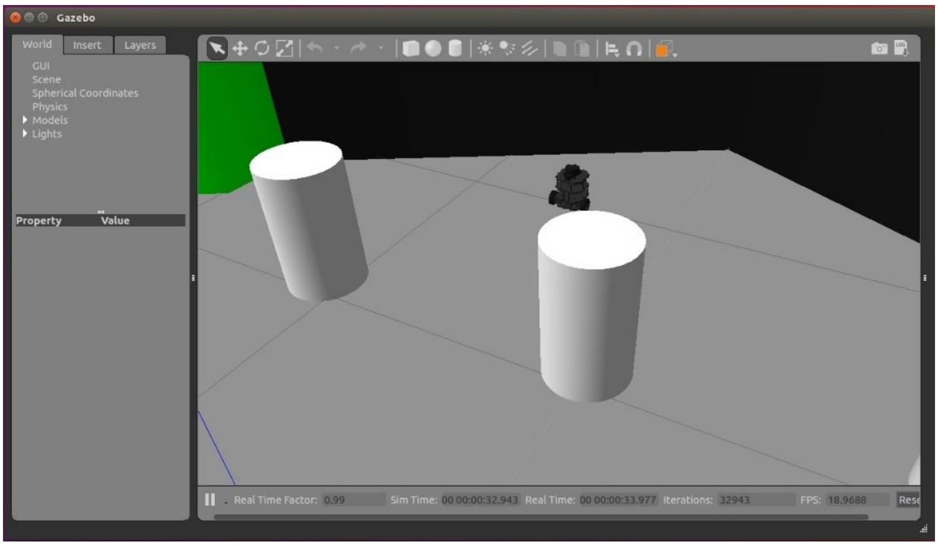
Navigate into the source folder of your workspace and clone the two repositories, 'turtlebot3' and 'turtlebot3_simulations'. Then, navigate back to your main workspace folder and 'catkin_make' the code and source the setup.bash file.

```
First create a 'catkin workspace' be following the instructions in 1.4.2

cd ~/catkin_ws/src

git clone https://github.com/ROBOTIS-GIT/turtlebot3.git

git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git

git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git

cd ..

catkin_make

source devel/setup.bash
```

Now we have set up everything needed to run the TurtleBot simulator. To actually run it, we have to choose which type of robot we want to operate ('burger' or 'waffle') and then launch all the appropriate nodes using a .launch file.

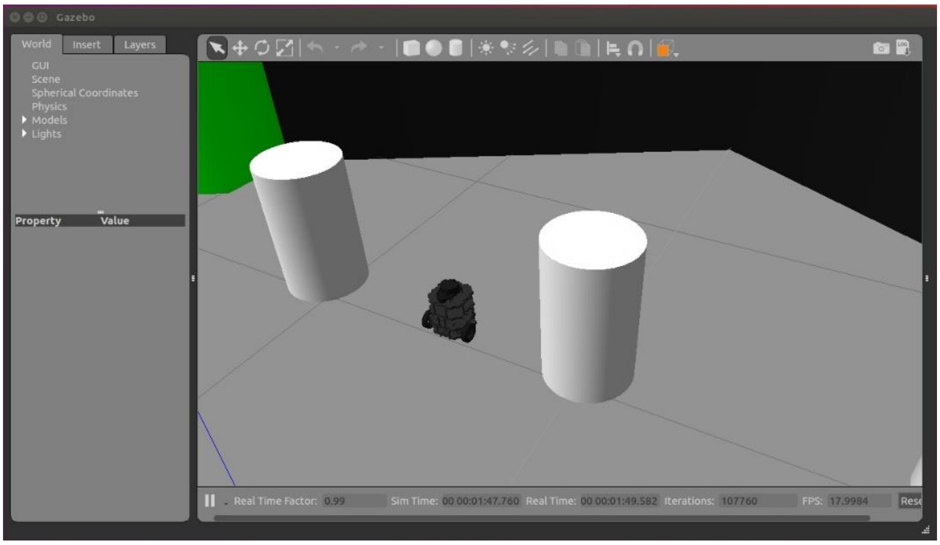
```
export TURTLEBOT3_MODEL="burger"

roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

During the first run you might need to wait for half a minute. If you are greeted with a blank screen, kill the launch file (Ctrl+C) and rerun it. If everything has worked fine, you will be able to see a basic environment with obstacles and the robot somewhere near them. You can manipulate the camera however you want (zoom in/out, change the orientation of the screen etc)



In a second terminal type the following and use your keyboard to move the robot.

```
cd ~/catkin_ws

source devel/setup.bash

export TURTLEBOT3_MODEL="burger"

roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```



2.3 Youbot example

This example shows the simulation of KUKA Youbot that we will be using throughout the lab and the courseworkσ. The arm has 5 revolute joints, thus 5 degrees of freedom.

To run the simulation, first we need to 'clone' some packages from the lab repository.

```
cd ~/catkin_ws/src

git clone https://github.com/LimingGao/ROS_Learning_Xiaoyun.git

cd ..

catkin_make

source devel/setup.bash
```

Before running this, you have to make sure that you install all the dependencies. You can run this command from any directory, but make sure you have the internet connection.

```
sudo apt install ros-kinetic-controller-manager ros-kinetic-joint-state-controller ros-kinetic-effort-controllers ros-kinetic-gazebo-ros-control ros-kinetic-joint-trajectory-controller ros-kinetic-velocity-controllers ros-kinetic-ros-controllers ros-kinetic-ros-control
```

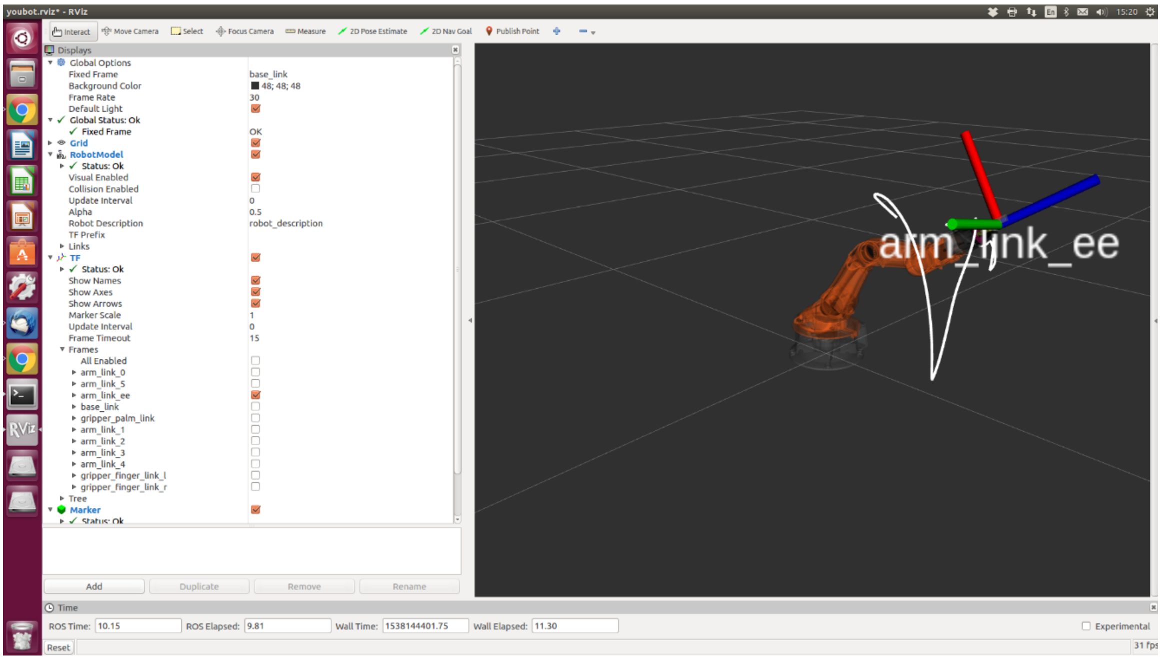
For some people who is using the ROS distribution other than kinetic, e.g. jade, melodic, etc. You have to change 'kinetic' to match your distribution, i.e.

```
sudo apt install ros-jade-controller-manager ros-jade-joint-state-controller ros-jade-effort-controllers ros-jade-gazebo-ros-control ros-jade-joint-trajectory-controller ros-jade-velocity-controllers ros-jade-ros-controllers ros-jade-ros-controls
```

For the first example, you can run the following command.

```
roslaunch youbot_traj_example youbot_traj.launch
```

The output on the screen should look like this,



The rviz (the simulator) shows the youbot arm moving around in the pre-defined motion. You can run how the message is passed around each node by running the following command in the second terminal.

```
rqt_graph
```

You can try playing around with the console in rviz to familiarise yourself with the environment, eg

- Check the checkbox "arm_link_xx" to show the coordinate frame.
- Adjust the value "alpha" under RobotModel to increase the opacity of the model.

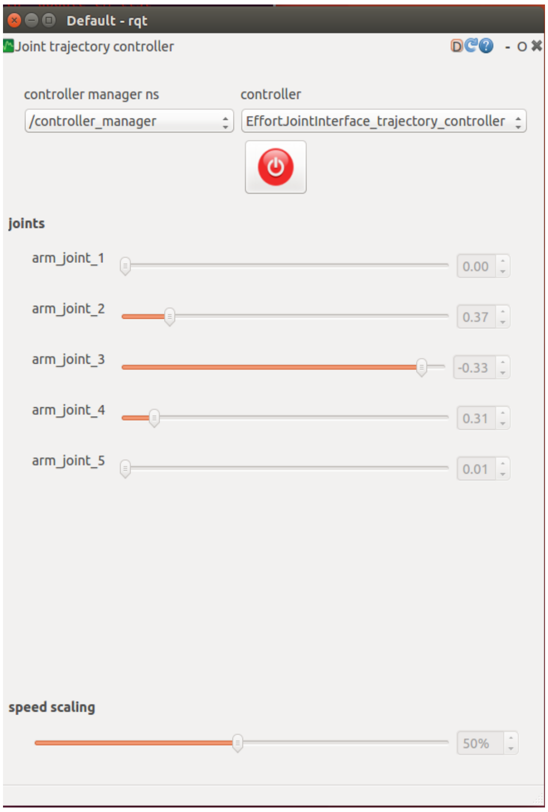
To run the second example, type the following command in the first terminal

```
roslaunch youbot_traj_example youbot_rqt_example.launch trajectory_interface:=true
```

The simulation will pop up like the first example, but the robot is not moving.

Then, run this in the second terminal.

```
cd ~/catkin_ws
source devel/setup.bash
rqt
```



A blank window will pop up. Look for the tabs *Plugins -> RobotTools -> Joint Trajectory Controllers*. You will get an interface like this. From here, you can adjust the sliders and observe the movement of the robot in rviz. This tool can be used to check whether your kinematic chain is implemented correctly.

Again, here you can run 'rqt_graph' to see how the messages are passing to each node through different topics.

SIDE NOTE:

In each example, you can read and trace how each package "depends" on and refer to each other. Every package in your workspace (catkin workspace) can build an executable or a library that constitute your robot simulation or a robot controller. Throughout the course, we will introduce more packages that controls joint data, kinematic, inverse kinematic, dynamics and many more because one robot application is a big collection of packages.