**ROS Melodic Installation**

Now we are going to setup ROS Melodic in *Ubuntu* 18.04.4 LTS (Bionic Beaver) system. Each ROS version supports to a specific Ubuntu version. ROS Melodic for Ubuntu 18.04 and ROS Kinetic for Ubuntu 16.04.6 LTS (Xenial Xerus) vice versa. So you need select Ubuntu version to setup on our system based on which ROS version you need to work with.

In the Video I’m setting up ROS in Jetpack SDK for Nvidia Jetson Nano which comes with *Ubuntu* 18. Steps will be same for your PC or Raspberry Pi also.

Open a new terminal by pressing **Ctrl + Alt + t** or executing the “**Terminal**” application using the Ubuntu 18 launch system.

First step is to Setup our system to accept software from *packages.ros.org*:

$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb\_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

Now add a new apt key,

sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654

Next Update the Debian packages index.

sudo apt update

At this point, We are ready to install ROS now. There are 3 options when we install ROS.

* **Desktop-Full** **Install** – Comes with ROS, [rqt](http://wiki.ros.org/rqt), [rviz](http://wiki.ros.org/rviz), robot-generic libraries, 2D/3D simulators and large number of other packages. This is recomanded for PC, but for our Arm based single board computers it is hard to handle simulators. it is not recommended for an embedded platform.

sudo apt install ros-melodic-desktop-full

* **Desktop Install** – This option has ROS, [rqt](http://wiki.ros.org/rqt), [rviz](http://wiki.ros.org/rviz), and robot-generic libraries. For this installation in Jetson Nano, I will use this method.

sudo apt install ros-melodic-desktop

* **ROS-Base (Bare Bones)** – This installs only ROS package, build, and communication libraries. No GUI tools. We can later install packages as we need.

sudo apt install ros-melodic-ros-base

It will take some time to download and installation. You can sit back and relax for a while.

After finishing installation, next we will add ROS environment variables to run with every bash session when new terminal is opened. I will explain this later in a next post.

echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc

source ~/.bashrc

Also we need to install following dependencies for building packages.

sudo apt install python-rosdep python-rosinstall python-rosinstall-generator python-wstool build-essential

Next we will need to initialize rosdep. It enables us to easily install system dependencies.

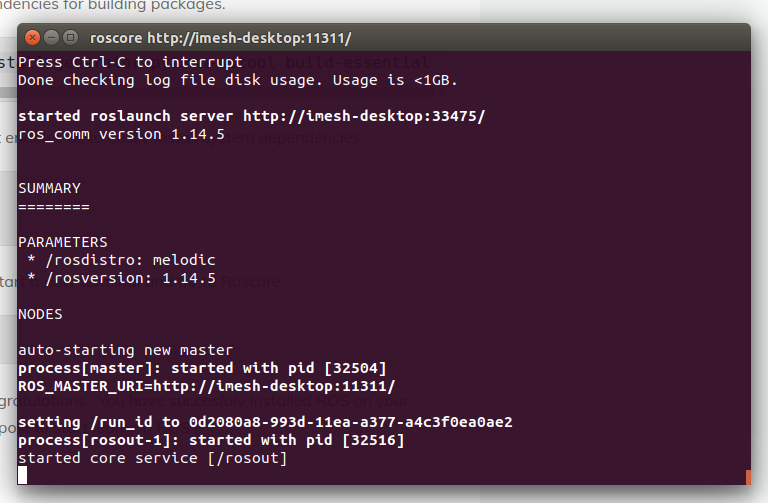
sudo rosdep init

rosdep update

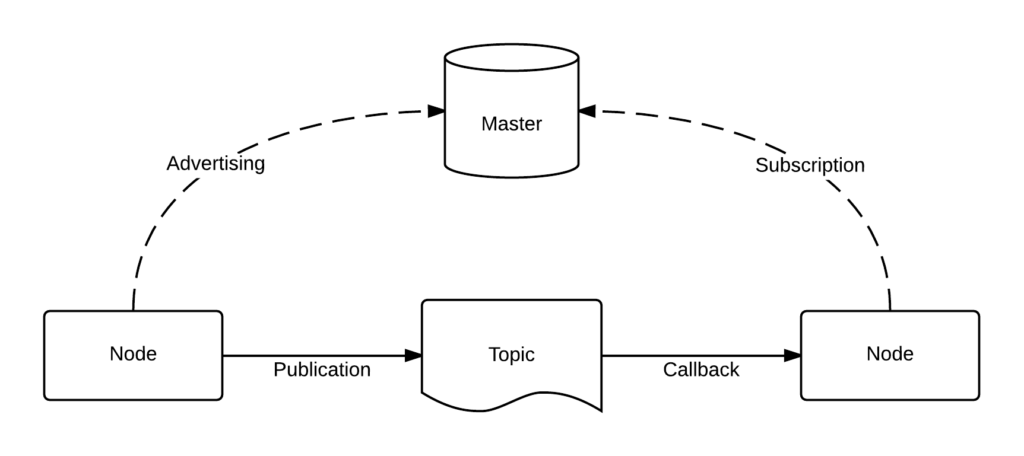
Now if everything went well, you can start a new terminal and try to Roscore.

roscore

It you are seeing following output, Congratulations.. You have succesfuly installed ROS on your system. Now you can continue to next post to start learning ROS !!



**ROS Master**

(Image Source – wikimedia.org)

The role of the Master is to enable individual ROS nodes to locate one another. We have only one master in the system and Once these nodes have located each other they communicate with each other peer-to-peer. It manages the communication between nodes and every node registers at startup with the master.

We can start ROS Master with

roscore

**ROS Nodes**

A ROS node is basically one executable or we can say as piece of code that represents a part of your ROS application. We can break our application into separate pieces as nodes which performs different functions in our final application. And you will see that it is better to have many nodes that provide only a single functionality, rather than have a large node that makes everything in the system.

With ROS Node concept, we can have set of separate codes that runs parallely in our application. It makes our code easy to organize and reusable. In above figure you can see that there are two nodes connected to same master.

Nodes can be written in Python, C++ or other programming language that supports.

**Communication Between Nodes**

There are three forms of communication between nodes in ROS.

* Topics
* Services
* Actions

**ROS Topics**

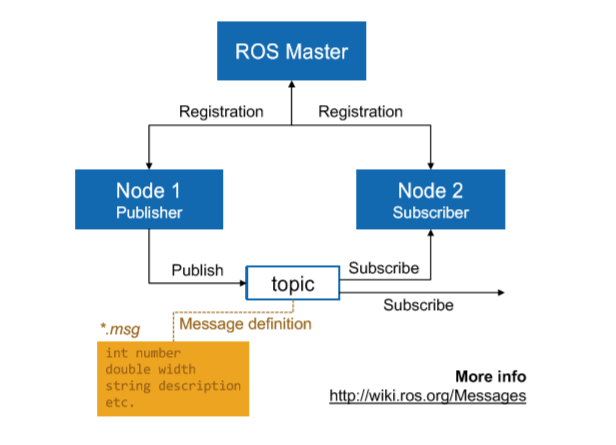
ROS Topics deliver data between ROS Nodes. In above figure you can see two ROS Nodes connected with a Topic. So one node can be a publisher and other can be the subscriber. Subscriber gets the data that publisher put into the topic. There can be any number of publisher and subscribers connected into same topic.

**ROS Messages**

ROS Messages define the structure of the data that flows through a Topic. Message descriptions are stored in .msg files.

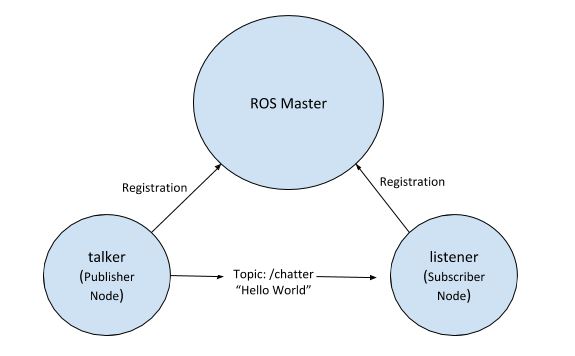
There are lots of predefined message types in ROS and also you can define a your own massage type and store it in the *msg/* sub directory of a ROS package.

Take a look at the system in below figure.



In this system, there are 2 nodes. One Publisher and one subscriber into a one topic. The data structure of the data passed through the topic defined in Message definition.

let’s look at a basic example.



In this example, we have two ROS nodes called talker and listener. Here our talker node will publish to the topic named /chatter and listener node subscribe to the topic and print the data into the terminal. We will see how to create a workspace, package, write ROS nodes, how to launch them individually, and with a launch file and how to inspect data on the topic.

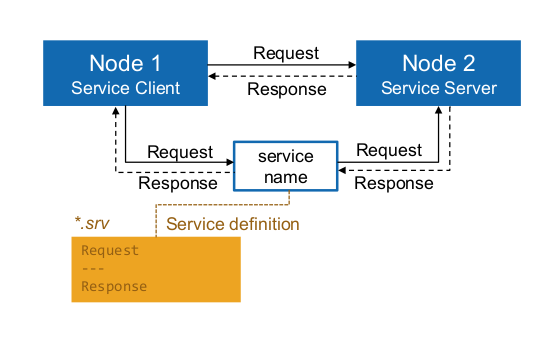
**ROS Services**

ROS Services implement a Request/response communication between nodes. When we publish to a topic, we send that data in a many-to-many fashion. Instead of that, with services we can make a request to a node and receive some data as a response only to that node.

In services we have,

* The service server which advertises the service.
* The service client that accesses this service.

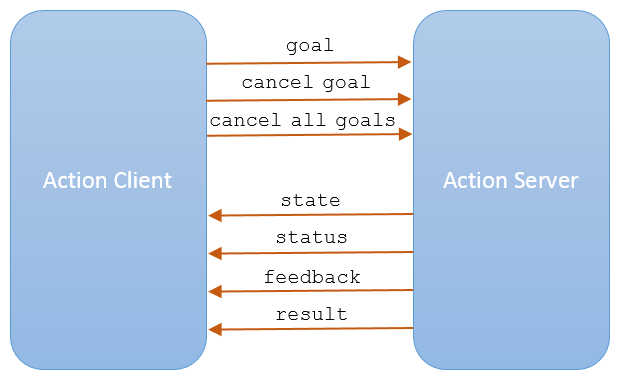
Same as in ROS messages, services are defined in \*.srv files in *srv/* sub directory of a *ROS* package.



**ROS Actions**

As we talked earlier, we saw that in services, Service clients send a request to a service server and get a response, but there is no information about the progress.

Similar to services, in actions we also have an action server and action client. action clients send a request to an action server in order to achieve some goal and will get a result. Unlike services, while the action is being performed an action server sends progress feedback to the client.



* **Action Server** – An action server provides an action. It advertises the action to other ROS entities and executes the action when a goal is received and accepted. While executing the action, it provides feedback about the progress of all executing actions and handles requests to cancel one or more actions. After completing the action, the action server will send the result including whether it succeeded, failed, or was canceled.
* **Action client** – An action client sends one or more goals (an action to be performed) and monitors their progress.

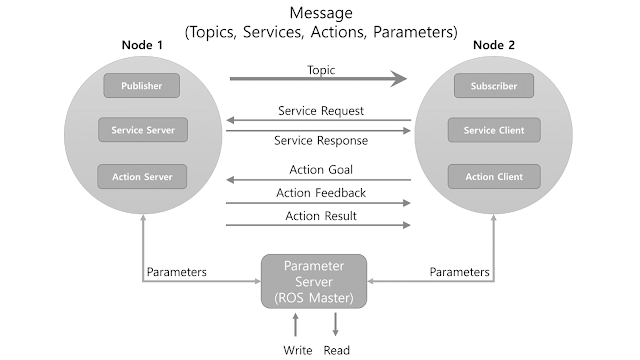
*For example* – Using an action interface to goto\_position , you send a goal, then move on to other tasks while the robot is driving. Along the way, you receive periodic progress updates (distance traveled, estimated time to goal, etc.), culminating in a result message (did the robot make it to the goal or was it forced to give up?). And if something more important comes up, you can at any time cancel the goal and send the robot somewhere else.

Actions require only a little more effort to define and use than do services, and they provide a lot more power and flexibility.

**Parameter Server**

Parameter Server gives us the possibility to store data in a central location. It is possible for any node to retrieve data from the parameter server and also nodes can store the parameters to the parameter server.

Users can load parameters into the parameter server with a launch file with \*.yaml files.



**ROS in action inside a Mobile Robot**

Now it’s time to see how actually these concepts implemented in a mobile robot. For live demonstration, please check the video so you can have better understanding.

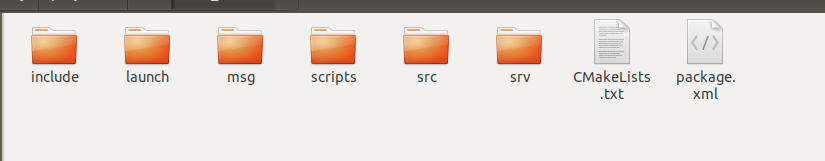
This is the mobile robot platform i developed while learning ROS. I really enjoy building things and there is nothing like hands on experience to learn something new for me.

# Let’s Make Robots 03 – Catkin Workspaces and ROS Packages

#### ****ROS Packages****

A ROS package can be set of nodes, or a library to use as a driver for sensor. Packages make it easy to manage and reuse. In following example, we will see how we can create a package add nodes and run it.

Inside ROS package we have a typical structure of files and folders. This structure looks as follows.

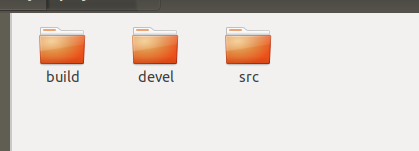


* include/package\_name/ : This directory includes the headers of the libraries that you would need.
* msg/ : If you develop nonstandard messages, put them here.
* scripts/ : These are executable scripts that can be in Bash, [Python](https://subscribe.packtpub.com/learn-python/), or any other scripting language.
* src/ : This is where the source files of your programs are present. You can create a folder for nodes and nodelets or organize it as you want.
* srv/ : This represents the service (srv) types.
* CMakeLists.txt : This is the CMake build file.
* package.xml : This is the package manifest.

#### ****Catkin Workspace****

A workspace is a folder where we can modify, build, and install ROS packages. We place our ROS Nodes inside packages. We can have several separate workspaces in our system.

Inside workspace. there are few folders which represent different spaces with different roles.



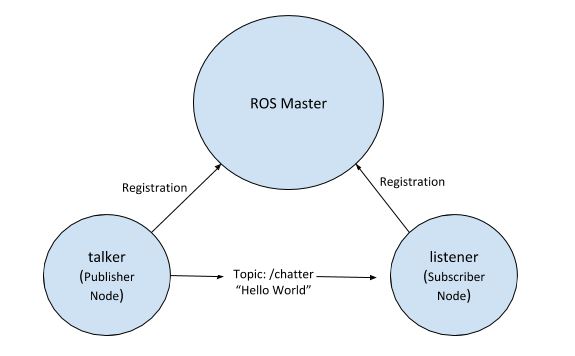
* **The source space**: In the source space (the src folder), you put your packages, projects, clone packages, and so on.
* **The build space**: In the build folder, cmake and catkin keep the cache information, configuration, and other intermediate files for our packages and projects.
* **Development (devel) space**: The devel folder is used to keep the compiled programs.

#### ****Launch files****

When we create a ROS based application, we will end up with several nodes and packages in our system and for our final application, we will need to run them all. To make this easy,we can write a launch file to start all the nodes and packages we need.

### ****Talker – Listener Example****

To understand these concepts better, let’s look at a basic example.

(Image Creator: Addison Sears-Collins)

In this basic example, we have two ROS nodes called talker and listener. Here our talker node will publish to the topic named /chatter and listener node subscribe to the topic and print the data into the terminal. With this example, we will see how to create a workspace, package, write ROS nodes, how to launch them individually and with a launch file and how to inspect data in topics.

First we are going to create a workspace to put our package. start a new terminal and run the following command. It will create a new folder called roboticslive\_ws in home directory.

mkdir roboticslive\_ws

Now navigate inside that folder and create a new folder named “src”. Then navigate into src folder.

cd roboticslive\_ws

mkdir src

cd src

following command will initiate the catkin workspace.

catkin\_init\_workspace

Now go back to the workspace folder and build the workspace.

cd ..

catkin\_make

if you are building ROS from source to achieve Python 3 compatibility, first catkin\_make command in a clean catkin workspace must be following.

catkin\_make -DPYTHON\_EXECUTABLE=/usr/bin/python3

There should be two new folders named “build” and “devel”. These folders contain automatically generated files during build process and we are not going to touch them. All the files that we will be working goes inside the src folder.

We have successfully created our workspace. Like this, we can have several separate workspaces as we need.

Now we are going to create a ROS package to put our two ROS nodes for our example. Navigate into the src folder and then we can create our ROS Package with following command.

cd src

catkin\_create\_pkg roboticslive\_ex01 std\_msgs rospy

Here, our package name is roboticslive\_ex01. std\_msgs and rospy are dependencies on which that package depends.

Now we have created roboticslive\_ex01 package. create “scripts” folder inside the roboticslive\_ex01 folder. This is where our python nodes will be located. You can place C++ nodes inside the src folder of the roboticslive\_ex01 package.

cd roboticslive\_ex01

mkdir scripts

Now download the files for the two ROS nodes [here](https://roboticslive.cc/wp-content/uploads/2020/05/listener_talker_example.zip) and place them inside the scripts folder. We will go through the codes for these ROS nodes later.

After placing python files for two nodes inside the scripts folder, run following commands to make these files executable.

cd scripts

chmod +x talker.py

chmod +x listener.py

Now it is time to build our workspace. Navigate back to top level directory of the workspace and run following command.

cd ~/rl\_ws/

catkin\_make

After finishing build process, we need to add the workspace to our ROS environment. For that we need to source the generated setup file. For every terminal we open, we need to run following command to make sure that the terminal knows about our current workspace and it’s packages inside.

source devel/setup.bash

It is time to run the nodes and test. First we need to start roscore.

roscore

Open a new terminal and let’s first start talker node.

source rl\_ws/devel/setup.bash

rosrun roboticslive\_ex01 talker.py

Here, **rosrun** is the command to start a ros node and **roboticslive\_ex01** is the package name. **talker.py** is the node name. We are simply asking from ros to run talker.py node from roboticslive\_ex01 package. Now you should see in terminal printing hello world with current time in float seconds as it is published into the **chatter** topic.

Same way we can start listener node also. Open a new terminal and,

source rl\_ws/devel/setup.bash

rosrun roboticslive\_ex01 listener.py

In new terminal, you should see “heard hello world” with current time in float seconds as it receives messages from talker node.

Now let’s examine what’s going on. Run following command in a new terminal.

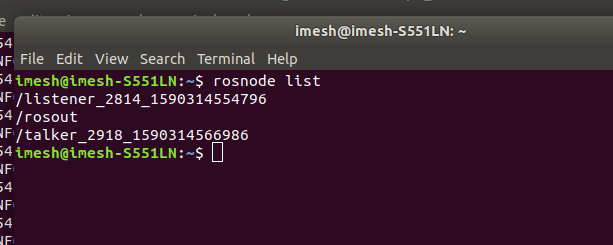
rqt\_graph is a GUI plugin for visualizing the ROS computation graph. With that we can vizualize how nodes and topics are connected. for our current example, you should see the following output.



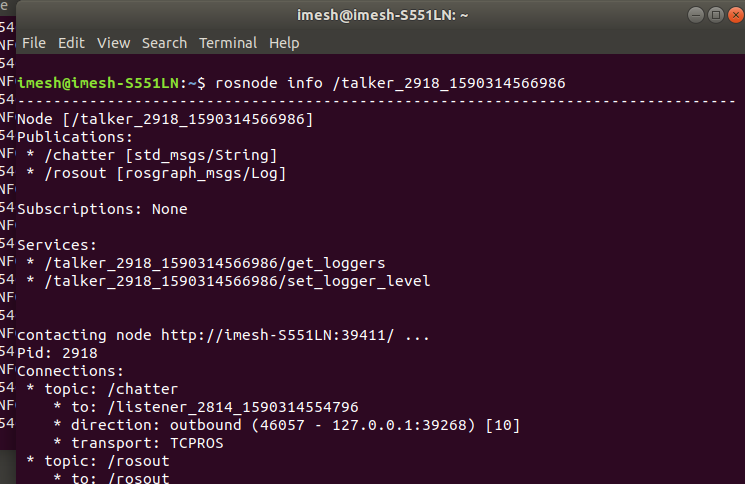
You can see here we have two ROS nodes connected to /chatter topic.

Also now you can try following commands to get more info about nodes and topics running. These commands are very useful when it comes to debugging a ROS application. Open a new terminal and run following commands one by one.

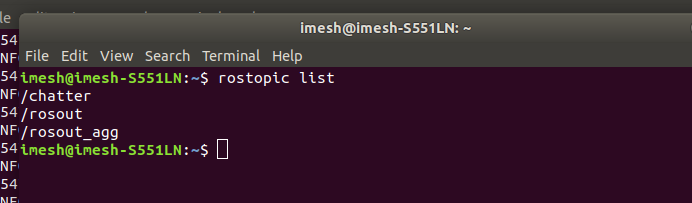
**rosnode list** – Gives a list of all currently running nodes.



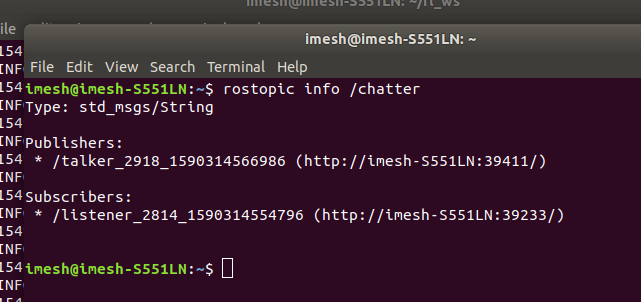
**rosnode info [node name]** – To get information about a node.



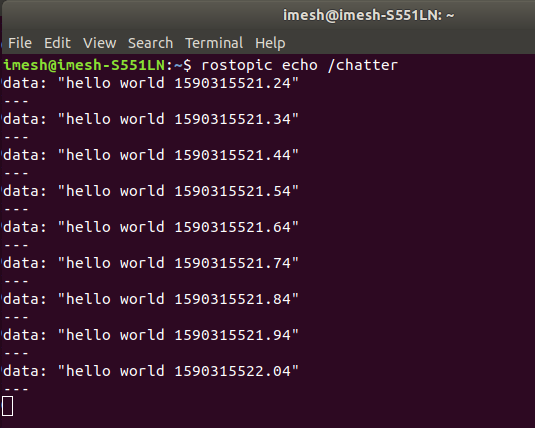
**rostopic list** – Gives a list of all currently running topics.



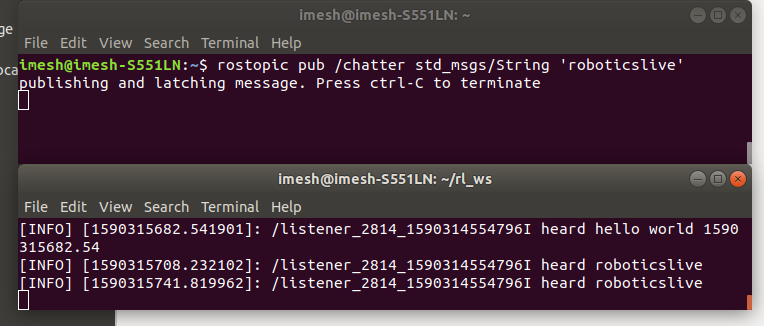
**rostopic info [topic name]** – To get information about a topic.



**rostopic echo [topic name]**– Check data passing inside the topic.



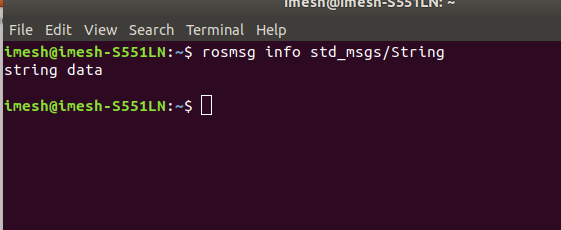
**rostopic pub [topic name] [msg type] [data]**– Publish data into a topic through terminal



In above figure we have two terminals. First one publishing data into the topic, here the string “roboticslive” and other terminal running the listener node which displays the data received in the topic.

Here the terminal acts as the publisher to the topic and listener node is the subscriber for that topic.

**rosmsg info [topic name]**– Display more inforamtion about message type.



This is only a few commands that is useful when debugging ROS applications. You can find more information about this at [ROS official website](https://www.ros.org/).

In this example, we started roscore and two nodes manually in seperate terminals. But when the application gets larger there can be large number of nodes to launch for our application. Starting all these nodes manually is just not practical and this is where **launch files** comes to rescue.

Now let’s write a simple launch file to start our example once.

Open a new terminal and navigate to roboticslive\_ex01 package directory. Then create a new folder named “launch” where our launch file will be located.

cd rl\_ws/src/roboticslive\_ex01/

mkdir launch

touch ex01.launch

Now open the files and copy the code below.

<?xml version="1.0" encoding="UTF-8"?>

<launch>

<!-- talker node -->

<node name="talker" pkg="roboticslive\_ex01" type="talker.py">

</node>

<!-- listener node -->

<node name="listener" pkg="roboticslive\_ex01" type="listener.py" output="screen">

</node>

</launch>

Here,

* pkg – Package name
* type – Node file you want to launch
* name – Unique identifier for your node

Now to start launch file, open a new terminal and,

cd rl\_ws/

source devel/setup.bash

roslaunch roboticslive\_ex01 ex01.launch

Now our example should start. Notice that with roslaunch, if there is no roscore running, it will start the roscore automatically.

