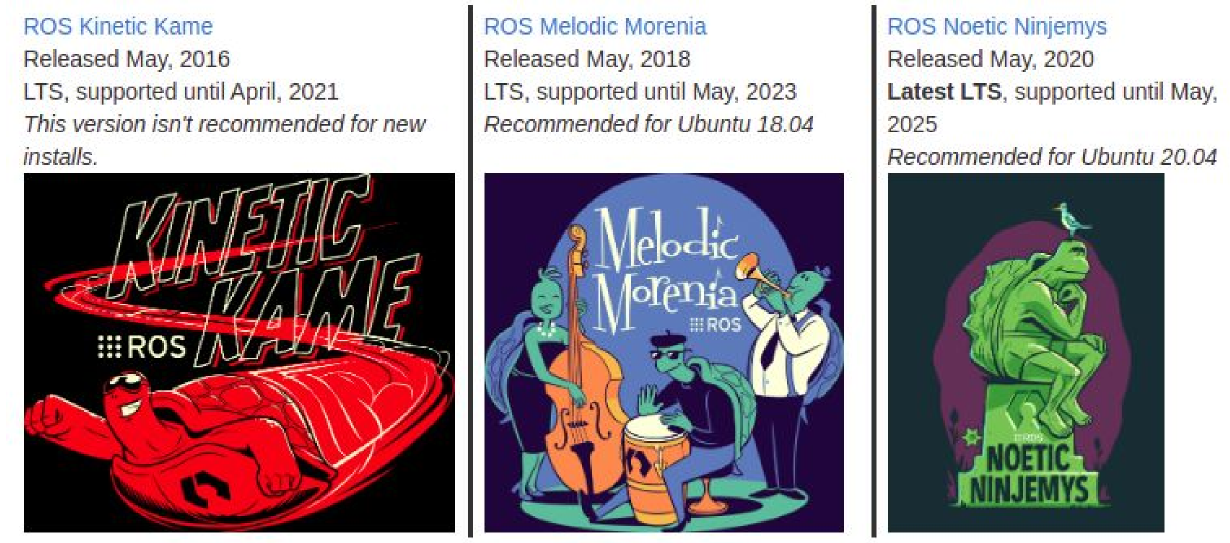
Robotics Lab

ROS(Robot Operating System)

Introduction

Robot Operating System (ROS or ros) is robotics middleware (i.e. collection of software frameworks for robot software development). Although ROS is not an operating system, it provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. Running sets of ROS-based processes are represented in a graph architecture where processing takes place in nodes that may receive, post and multiplex sensor data, control, state, planning, actuator, and other messages. Despite the importance of reactivity and low latency in robot control, ROS itself is not a real-time OS (RTOS). It is possible, however, to integrate ROS with real-time code.

ROS Installation Options



ROS Kinetic Kame
Released May, 2016
LTS, supported until April, 2021
This version isn't recommended for new installs.

ROS Melodic Morenia
Released May, 2018
LTS, supported until May, 2023
Recommended for Ubuntu 18.04

ROS Noetic Ninjemys
Released May, 2020
Latest LTS, supported until May, 2025
Recommended for Ubuntu 20.04

There is more than one ROS distribution supported at a time. Some are older releases with long term support, making them more stable, while others are newer with shorter support life times, but with binaries for more recent platforms and more recent versions of the ROS packages that make them up.

We will use ROS Melodic Morenia in this session.

Installation

1. Setup sources,list

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

2. Setup your Keys

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

3. Update Debian Package

```
$ sudo apt update
```

4. Install Desktop full version of ROS

ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators and 2D/3D perception

```
$ sudo apt install ros-melodic-desktop-full
```

5. Environment Setup

To add ROS environment variables automatically to your bash session every time a new shell is launched:

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

6. Dependencies for building packages

To create and manage your own ROS workspaces, there are various tools and requirements that are distributed separately. For example, rosinstall is a frequently used command-line tool that enables you to easily download many source trees for ROS packages with one command.

To install this tool and other dependencies for building ROS packages, run:

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

7. Initialize rosdep

Before you can use many ROS tools, you will need to initialize rosdep. rosdep enables you to easily install system dependencies for source you want to compile and is required to run some core components in ROS. If you have not yet installed rosdep, do so as follows.

```
$ sudo apt install python-rosdep
$ sudo rosdep init
$ rosdep update
```

Load the environment file

This command imports the environment setting file. Environment variables such as ROS_ROOT, ROS_PACKAGE_PATH, etc. are defined.

```
$ source /opt/ros/melodic/setup.bash
```

WORKSPACE

- **ROS Workspace** is a folder to organize ROS project files
- **Catkin**: it is a build tool that compiles source files to binaries
- **Catkin workspace**: It is a ROS Workspace where catkin is used as the build tool.

Creating and building a catkin workspace

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

Once you have finished building without errors, run the ‘**ls**’ command. In addition to the ‘**src**’ folder created by the user, ‘**build**’ and ‘**devel**’ folders have been created. The build related files for the catkin build system are saved in the ‘**build**’ folder, and the execution related files are saved in the ‘**devel**’ folder.

```
$ ls
```

```
build
devel
src
```

Lastly, import the setting file associated with the catkin build system.

```
$ source devel/setup.bash
```

Testing

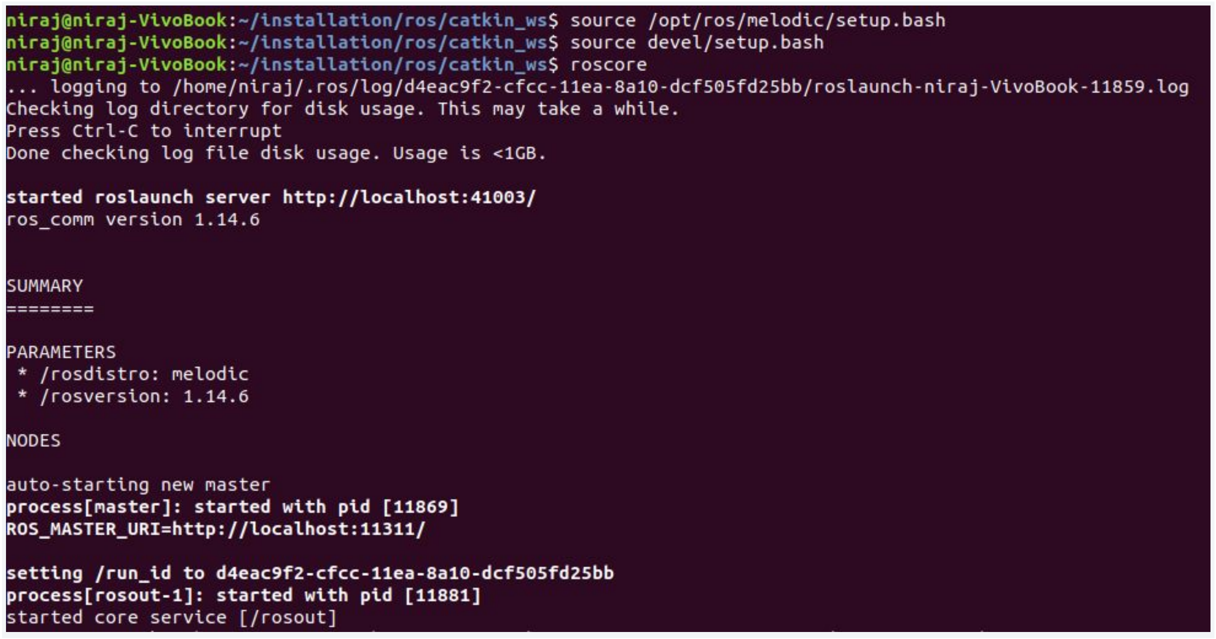
To test the installation of ROS, close all the terminal windows and open a new terminal window and enter the following commands

```
$ source /opt/ros/melodic/setup.bash
$ source ~/catkin_ws/devel/setup.bash
```

These two commands need to be executed every time we open a new terminal window in order to apply the settings to the current environment.

Now enter the following command to run roscore:

```
$ roscore
```



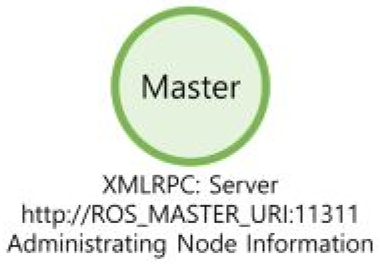
```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ source /opt/ros/melodic/setup.bash
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ source devel/setup.bash
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ roscore
... logging to /home/niraj/.ros/log/d4eac9f2-cfcc-11ea-8a10-dcf505fd25bb/roslaunch-niraj-VivoBook-11859.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://localhost:41003/
ros_comm version 1.14.6


SUMMARY
========

PARAMETERS
 * /rosdistro: melodic
 * /rosversion: 1.14.6

NODES

auto-starting new master
process[master]: started with pid [11869]
ROS_MASTER_URI=http://localhost:11311/

setting /run_id to d4eac9f2-cfcc-11ea-8a10-dcf505fd25bb
process[rosout-1]: started with pid [11881]
started core service [/rosout]
```

If it runs like the following without errors, the installation is completed. Terminate the process with [Ctrl+c].

Fundamental Concepts

Master

- Master acts as a name server for node to node connections and message communication.
- The command **roscore** is used to run the master.
- Once the master is run, one can register the name of each node and get information when needed.
- When ROS is executed, the master will be configured with the URI address and port configured in the ROS_MASTER_URI. By default, the URI address uses the IP address of local PC, and port number 11311, unless otherwise modified.

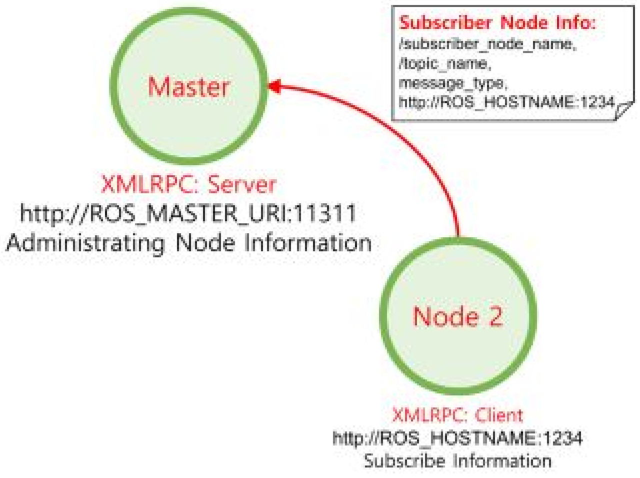


roscore

- roscore is the command that runs the ROS master.
- roscore can be run from another computer within the network if multiple computers are within the same network
- When the ROS master is running, the URI address and port number assigned for ROS_MASTER_URI environment variable are used. If the environment variables are not set, the local IP address is used as the URI address and port number 11311 is used.

ROS Node

- It refers to the smallest unit of processor running in ROS.
- It can be considered as one executable program.
- It is recommended to create a single node for each purpose. Eg. In case of mobile robots, the program to operate the robot is broken down into specialized functions such as sensor drive, sensor data conversion, obstacle recognition, motor drive, encoder input, and navigation.
- Upon start up, a node registers information such as name, message type, URI address and port number of the node.
- The registered node can act as a publisher, subscriber, service server or service client based on the registered information and nodes can exchange messages using topics and services.
- Node uses XMLRPC for communication with the master and uses XMLRPC or TCPROS of the TCP/IP protocols when communicating between nodes.



rosrun

- rosrun is the basic execution command of ROS.
- It is used to run a single node in the package.
- The node uses the ROS_HOSTNAME environment variable stored in the computer on which the node is running as the URI address, and the port is set to an arbitrary unique value.

```
$ rosrun [package_name] [node_name]
```

roslaunch

- roslaunch is the command to execute multiple nodes.
- roslaunch uses the launch file (*.launch) file(XML based) to define the nodes to be executed.

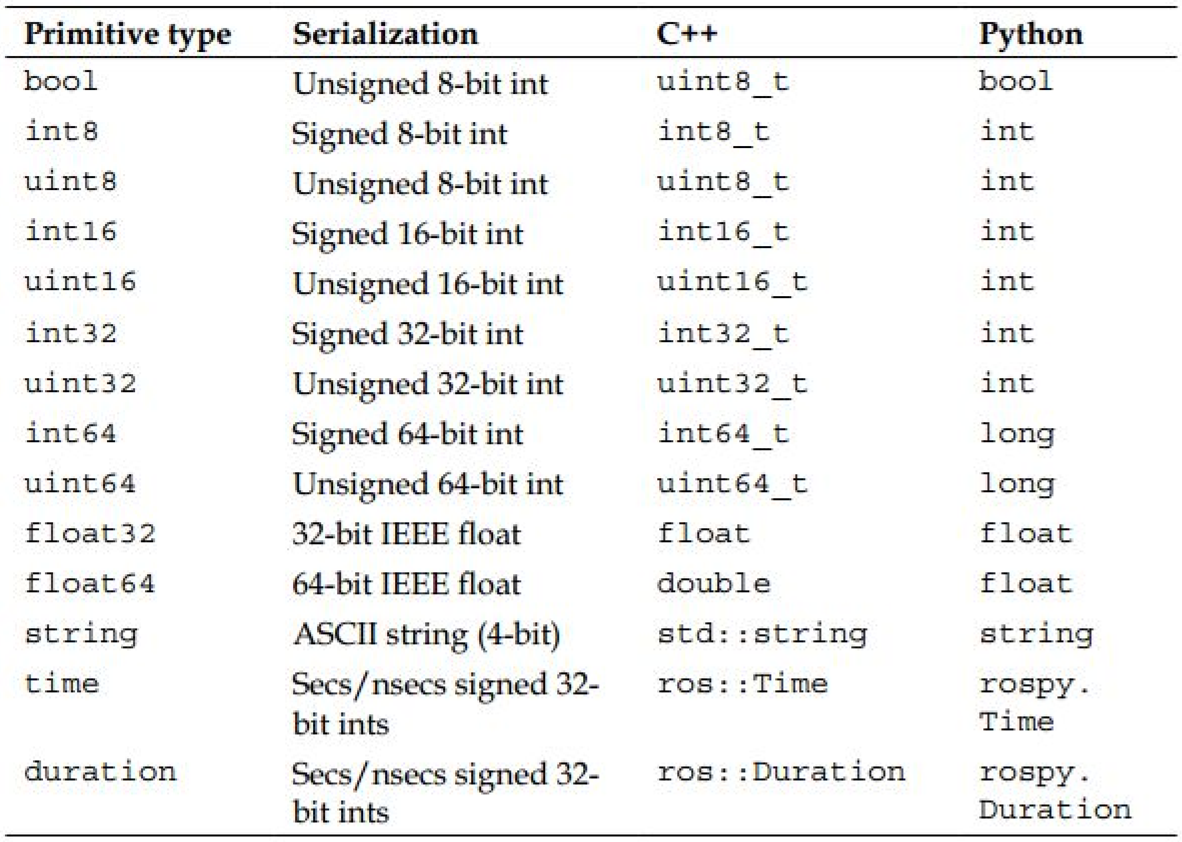
ROS parameter server

- A program that normally runs along with the ROS master
- The user can store various parameters or values on this server and all the nodes can access it
- Privacy can also be set for parameters; if it is private, only specific nodes can access it.

ROS Message

- A node sends or receives data between nodes via a message.
- Messages are variables such as integer, floating point and boolean.
- Nested message structure that contains other messages or an array of messages can be used in the message.
- A message must have two principal parts: fields and constants; where field means the type of data for example int32, float32, string etc and constants define the name of the fields.

  Eg.	int32 ID
  	string name
  	float32 distance
- Standard types to use in message are:



Primitive type	Serialization	C++	Python
bool	Unsigned 8-bit int	uint8_t	bool
int8	Signed 8-bit int	int8_t	int
uint8	Unsigned 8-bit int	uint8_t	int
int16	Signed 16-bit int	int16_t	int
uint16	Unsigned 16-bit int	uint16_t	int
int32	Signed 32-bit int	int32_t	int
uint32	Unsigned 32-bit int	uint32_t	int
int64	Signed 64-bit int	int64_t	long
uint64	Unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ASCII string (4-bit)	std::string	string
time	Secs/nsecs signed 32-bit ints	ros::Time	rospy.Time
duration	Secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration

ROS Topic

- A topic is literally like a topic in a conversation.
- A publisher node first registers its topic with the master and then starts publishing messages on a topic.
- Subscriber nodes that want to request the topic request information of the publisher node corresponding to the name of the topic registered in the master.
- Based on the information, the subscriber node directly connects to the publisher node to exchange messages as a topic.

Publish and Publisher

- The term ‘publish’ stands for the action of transmitting relative messages corresponding to the topic.
- The publisher node registers its own information and topic with the master, and sends a message to connected subscriber nodes that are interested in the same topic.

Subscribe and Subscriber

- The term ‘subscribe’ stands for the action of receiving relative messages corresponding to the topic.
- The subscriber node registers its own information and topic with the master, and receives publisher information that publishes relative topics from the master.
- Based on received publisher information, the subscriber node directly requests connection to the publisher node and receives messages from the connected publisher node.

ROS Demo (turtlesim)

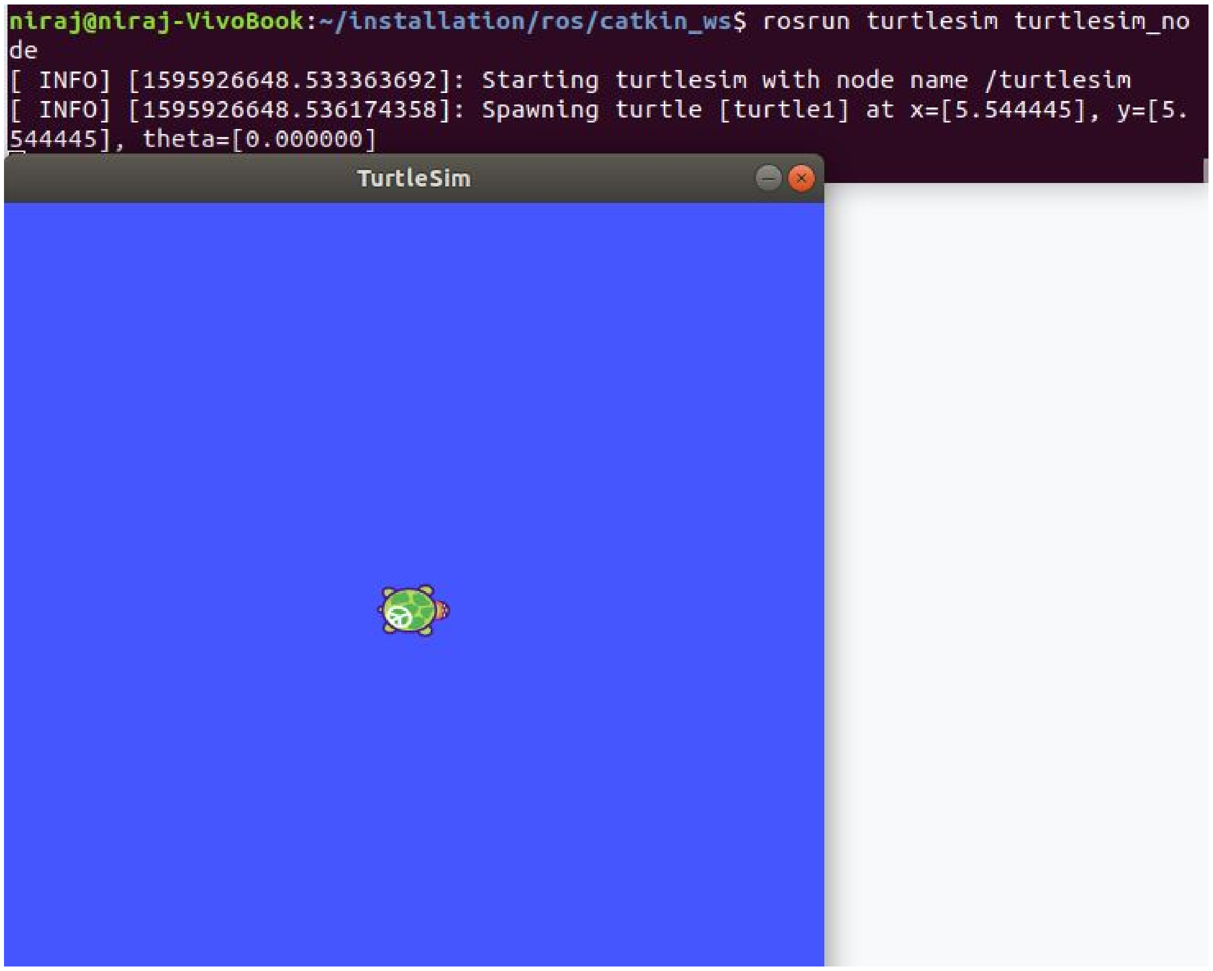
We will use an application called turtlesim for learning the ROS concept.
Start the application using the following commands(Each command on new terminal tab):

1. Start roscore

```
$ roscore
```

2. Start turtlesim node

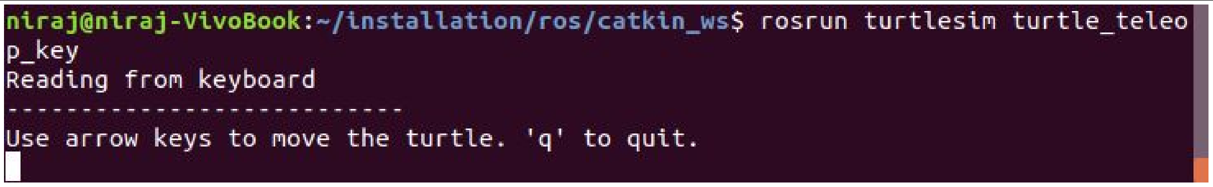
```
$ rosrun turtlesim turtlesim_node
```



```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rosrun turtlesim turtlesim_no
de
[ INFO] [1595926648.533363692]: Starting turtlesim with node name /turtlesim
[ INFO] [1595926648.536174358]: Spawning turtle [turtle1] at x=[5.544445], y=[5.
544445], theta=[0.000000]
```

3. Start teleop to move the turtle using keyboard

```
$ rosrun turtlesim turtle_teleop_key
```



```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rosrun turtlesim turtle_teleo
p_key
Reading from keyboard
---------------------------
Use arrow keys to move the turtle. 'q' to quit.
```

Now you will be able to move the turtle using arrow keys in your computer.

Note: To move the turtle using arrow keys, the terminal with teleop operation running should be selected.

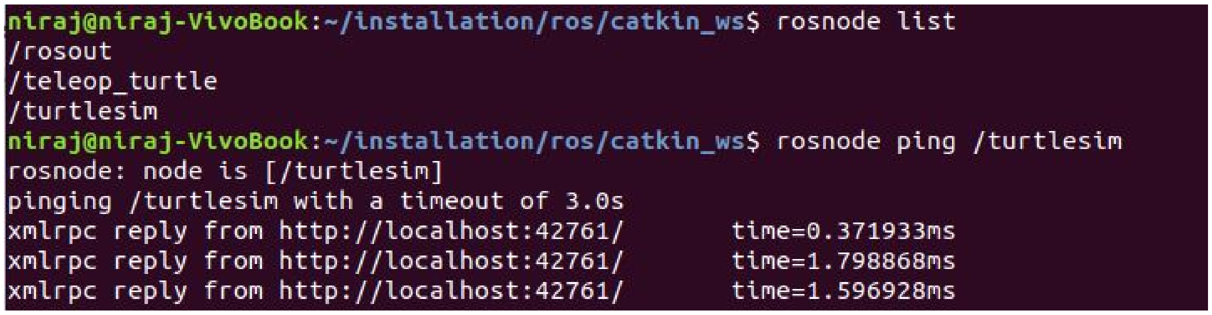
Inspection Commands

rosnode

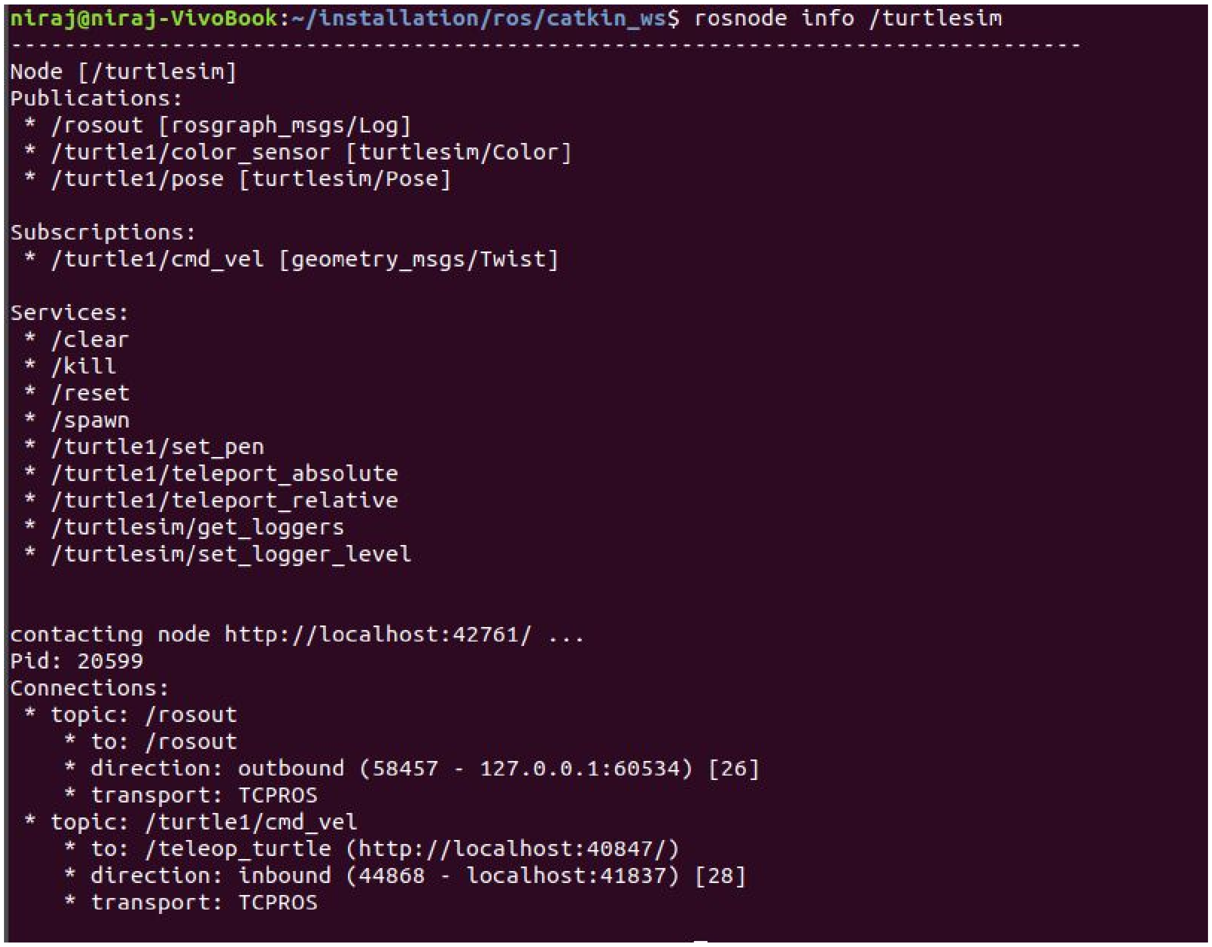
rosnode displays debugging information about ROS nodes including publications, subscriptions and connections

Commands:

`rosnode list`	List active nodes
`rosnode ping <node_name>`	Test connectivity to the node
`rosnode info <node_name>`	Print information about a node
`rosnode kill <node_name>`	Kills a running node



```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rosnode list
/rosout
/teleop_turtle
/turtlesim
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rosnode ping /turtlesim
rosnode: node is [/turtlesim]
pinging /turtlesim with a timeout of 3.0s
xmlrpc reply from http://localhost:42761/       time=0.371933ms
xmlrpc reply from http://localhost:42761/       time=1.798868ms
xmlrpc reply from http://localhost:42761/       time=1.596928ms
```



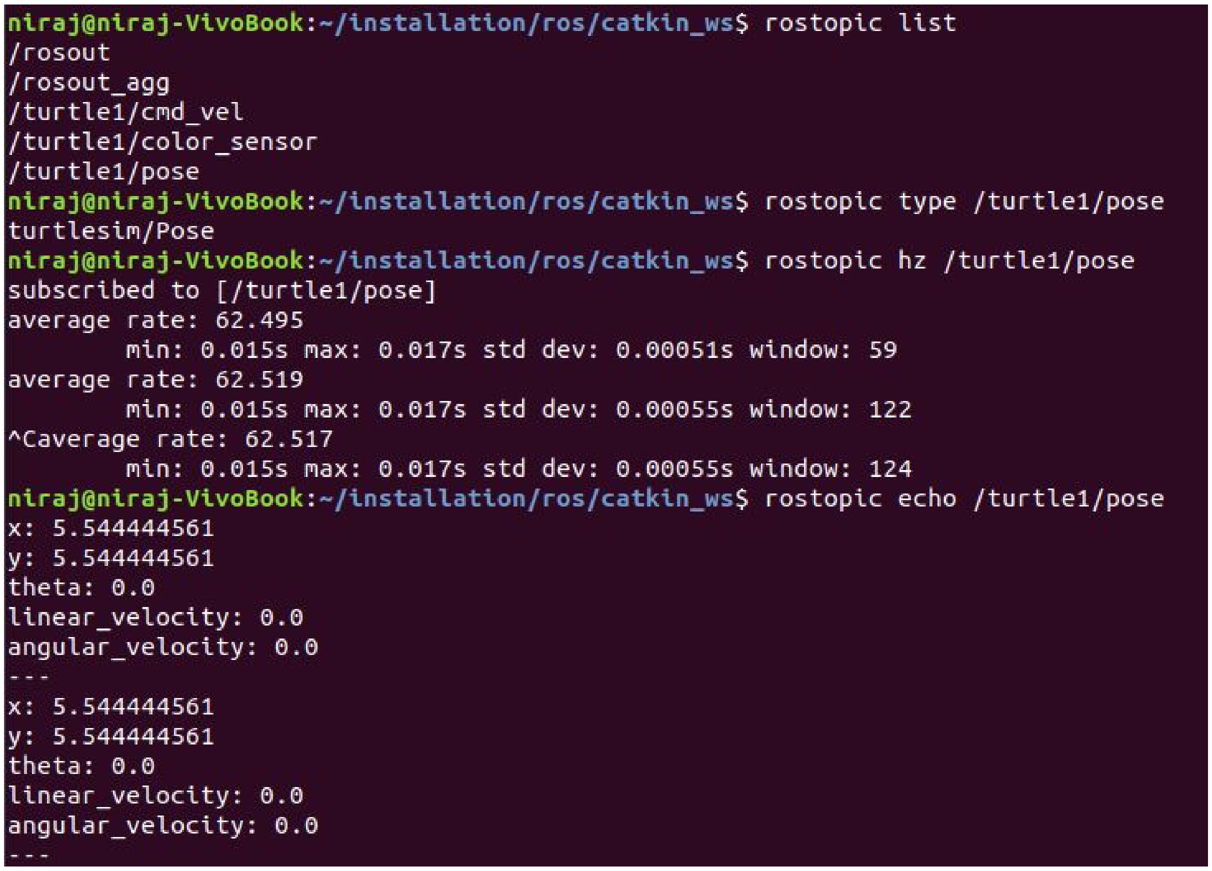
```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rosnode info /turtlesim
--------------------------------------------------------------------------------
Node [/turtlesim]
Publications:
 * /rosout [rosgraph_msgs/Log]
 * /turtle1/color_sensor [turtlesim/Color]
 * /turtle1/pose [turtlesim/Pose]

Subscriptions:
 * /turtle1/cmd_vel [geometry_msgs/Twist]

Services:
 * /clear
 * /kill
 * /reset
 * /spawn
 * /turtle1/set_pen
 * /turtle1/teleport_absolute
 * /turtle1/teleport_relative
 * /turtlesim/get_loggers
 * /turtlesim/set_logger_level


contacting node http://localhost:42761/ ...
Pid: 20599
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound (58457 - 127.0.0.1:60534) [26]
    * transport: TCPROS
 * topic: /turtle1/cmd_vel
    * to: /teleop_turtle (http://localhost:40847/)
    * direction: inbound (44868 - localhost:41837) [28]
    * transport: TCPROS
```

rostopic

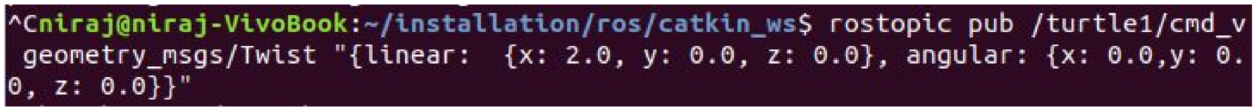
rostopic is a tool for displaying debug information about ROS topics, including publications, subscriptions, publishing rate and messages.

Commands:

`rostopic list`	List active topics
`rostopic echo`	Print message to screen
`rostopic hz`	Display publishing rate of topic
`rostopic pub`	Publish data to topic
`rostopic type`	Print topic type
`rostopic find`	Find topics by type



```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rostopic type /turtle1/pose
turtlesim/Pose
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rostopic hz /turtle1/pose
subscribed to [/turtle1/pose]
average rate: 62.495
        min: 0.015s max: 0.017s std dev: 0.00051s window: 59
average rate: 62.519
        min: 0.015s max: 0.017s std dev: 0.00055s window: 122
^Caverage rate: 62.517
        min: 0.015s max: 0.017s std dev: 0.00055s window: 124
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rostopic echo /turtle1/pose
x: 5.544444561
y: 5.544444561
theta: 0.0
linear_velocity: 0.0
angular_velocity: 0.0
---
x: 5.544444561
y: 5.544444561
theta: 0.0
linear_velocity: 0.0
angular_velocity: 0.0
---
```



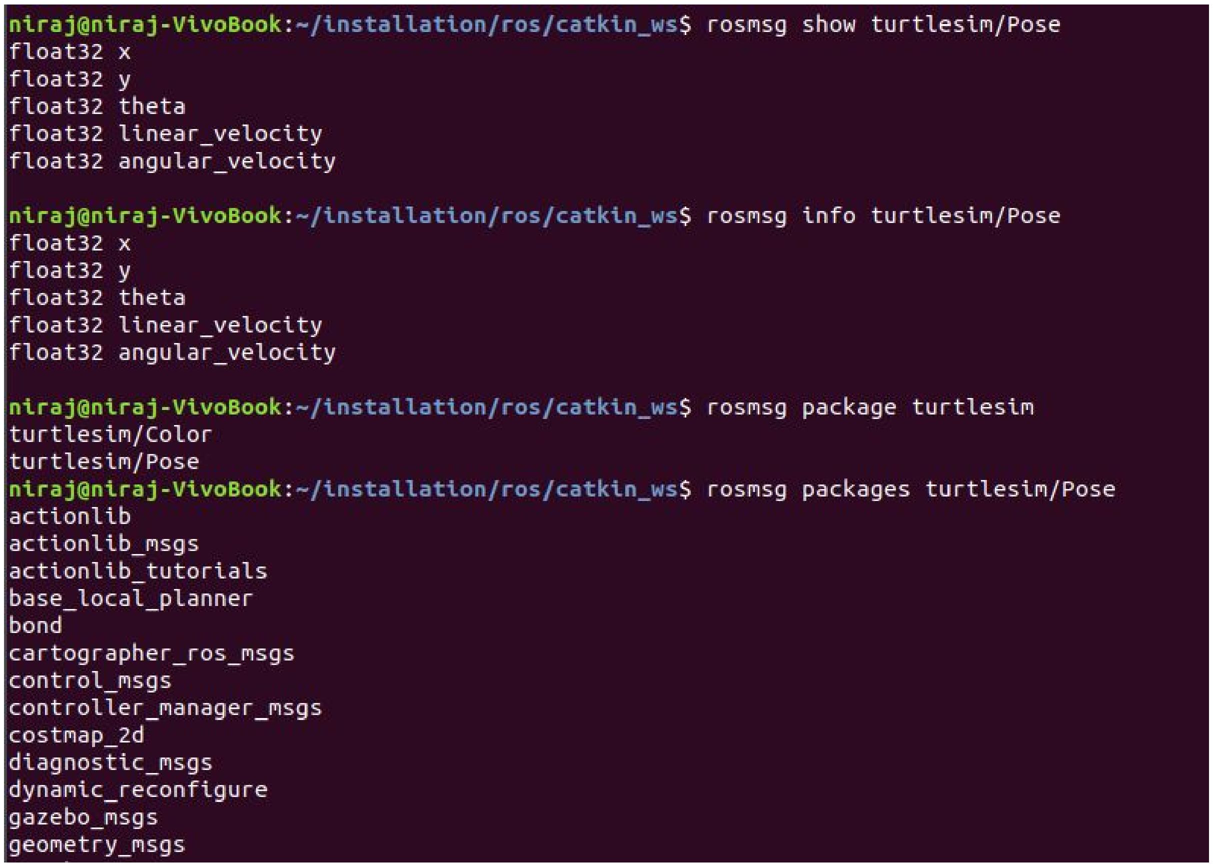
```
^Cniraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rostopic pub /turtle1/cmd_v
 geometry_msgs/Twist "{linear:  {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0,y: 0.
0, z: 0.0}}"
```

rosmsg

rosmsg is a command-line tool for displaying debug information about ROS Message types.

Commands:

`rosmsg list`	List all messages
`rosmsg show`	show message description
`rosmsg info`	Alias for rosmsg show
`rosmsg md5`	Display message md5sum
`rosmsg package`	List messages in a package
`rosmsg packages`	List packages that contain messages



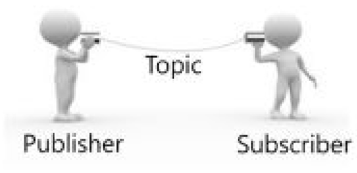
```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rosmsg show turtlesim/Pose
float32 x
float32 y
float32 theta
float32 linear_velocity
float32 angular_velocity

niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rosmsg info turtlesim/Pose
float32 x
float32 y
float32 theta
float32 linear_velocity
float32 angular_velocity

niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rosmsg package turtlesim
turtlesim/Color
turtlesim/Pose
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rosmsg packages turtlesim/Pose
actionlib
actionlib_msgs
actionlib_tutorials
base_local_planner
bond
cartographer_ros_msgs
control_msgs
controller_manager_msgs
costmap_2d
diagnostic_msgs
dynamic_reconfigure
gazebo_msgs
geometry_msgs
```

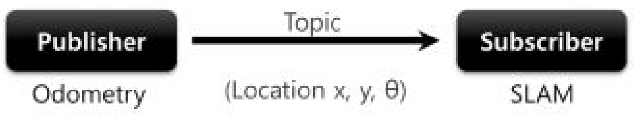
Exercise:

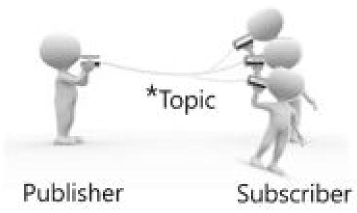
1. What are the active topics when you run **`$ rosrun turtlesim turtlesim_node`**?
2. What are the topics published by **`/teleop_turtle`** node [Note: **`$ rosrun turtlesim turtle_teleop_key`** should be running in terminal]?
3. What is the type of message passed in **`/turtle1/cmd_vel`** topic?
4. Show the information about the message **`geometry_msgs/Twist`** and **`turtlesim/Pose`** ?

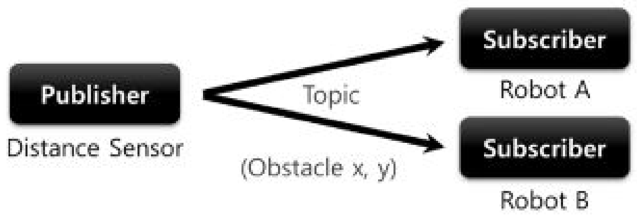
Fundamental ROS Nodes

Publisher and Subscriber





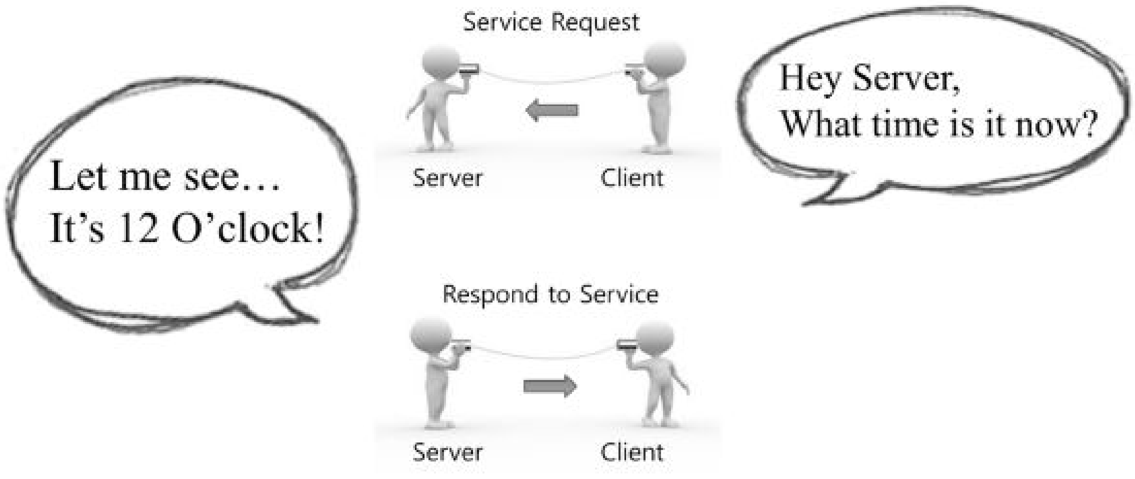




*Topic not only allows 1:1 Publisher and Subscriber communication, but also supports 1:N, N:1 and N:N depending on the purpose.

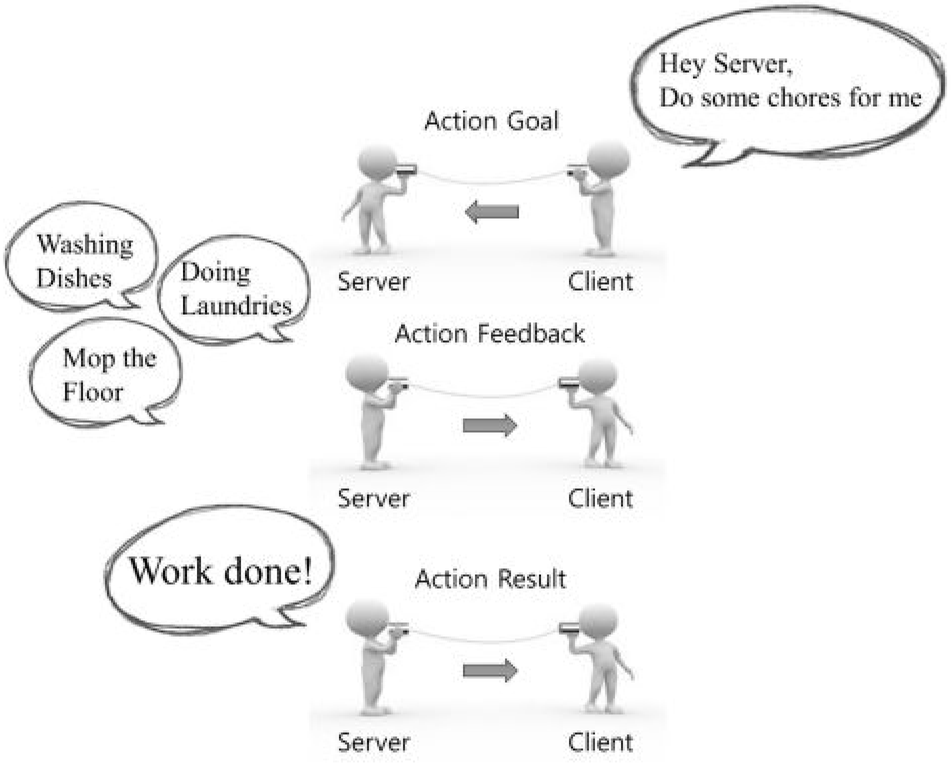
Services

- Request-Response communication
- Two message types: a request and a response
- ‘Event driven’ execution for ROS applications
- Blocking action(waits until execution is complete)

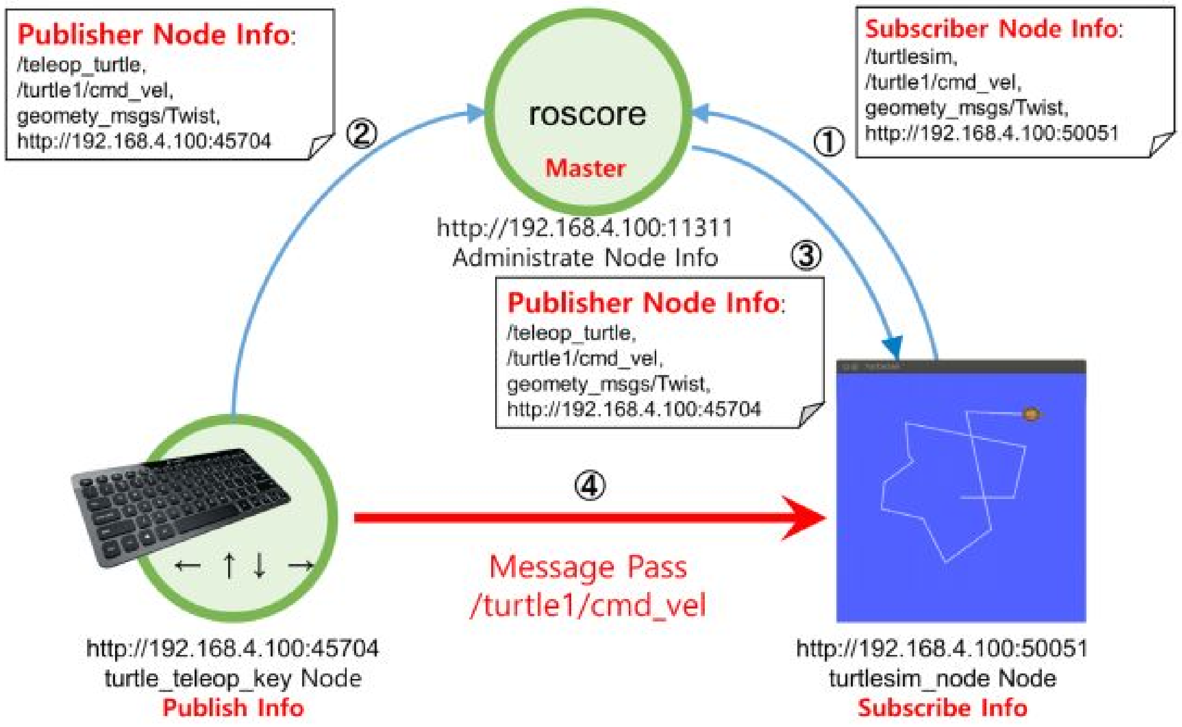

Service Request
Server
Client
Hey Server, What time is it now?
Let me see… It’s 12 O’clock!
Respond to Service
Server
Client

Actions

- Request-Response system
- The request field is called **goal(request),** response field is called **result(response)** and new field **feedback** provides continuous feedback related to processing the goal request.
- No waiting until an execution is complete(waiting is an option if required)



Overall message communication example



ROS Package

- Package is the basic unit of ROS.
- All source code, data files, dependencies and other files are organized in packages.

Creating a new package

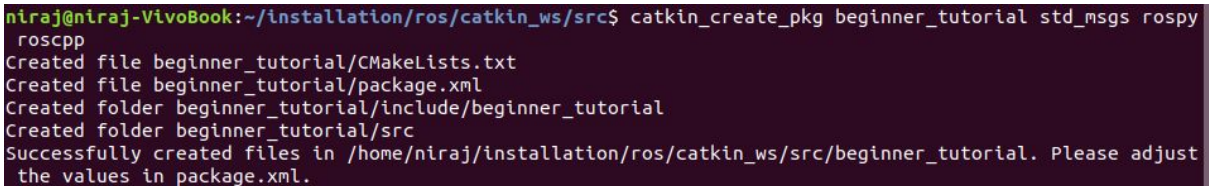
1. Change to source space directory of the workspace

```
$ cd ~/catkin_ws/src
```

2. Use **catkin_create_pkg** script to create a new package '**beginner_tutorials**' which depends on std_msgs, roscpp and rospy
   The syntax to create package is:
   `catkin_create_pkg <package_name> [depend1] [depend2] [depend3]...`

```
$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

This will create a **beginner_tutorials** folder which contains a **package.xml** and a **CMakeLists.txt** which have been partially filled.



```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws/src$ catkin_create_pkg beginner_tutorial std_msgs rospy
 roscpp
Created file beginner_tutorial/CMakeLists.txt
Created file beginner_tutorial/package.xml
Created folder beginner_tutorial/include/beginner_tutorial
Created folder beginner_tutorial/src
Successfully created files in /home/niraj/installation/ros/catkin_ws/src/beginner_tutorial. Please adjust
 the values in package.xml.
```

3. Build the catkin workspace.

```
$ cd ~/catkin_ws
$ catkin_make
```

4. Source the setup file

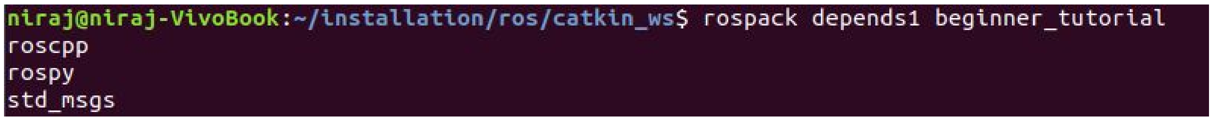
```
$ source ~/catkin_ws/devel/setup.bash
```

The package **beginner_tutorials** is created successfully.

We can check the dependencies of the package we created using following command:

```
$ rospack depends1 beginner_tutorials
```

We can use **depends1** to see the first order dependencies and depends only to see all the dependencies.



```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rospack depends1 beginner_tutorial
roscpp
rospy
std_msgs
```

Change directory to beginner_tutorials

```
$ roscd beginner_tutorials/
```

Check the package.xml

```
$ cat package.xml
```

You can fill in the information and also remove the comments which are not required. Example of final package.xml file

```
<?xml version="1.0"?>
<package format="2">
  <name>beginner_tutorials</name>
  <version>0.1.0</version>
  <description>The beginner_tutorials package</description>

  <maintainer email="you@yourdomain.com">Your Name</maintainer>
  <license>BSD</license>
  <url type="website">http://wiki.ros.org/beginner_tutorials</url>
  <author email="you@yourdomain.com">Jane Doe</author>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>

  <exec_depend>roscpp</exec_depend>
  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>

</package>
```

Creating a message type

1. Create a folder for message in package

```
$ roscd beginner_tutorials
$ mkdir msg
```

2. Create a .msg file with file name as name of the message in msg folder.

```
$ cd msg/
$ touch Demo_sensor.msg
```

3. Eg. We will use the following fields in the message. Add the following things in the Demo_sensor.msg file we created.

```
string obstacle_shape
float32 distance
```

4. We need to make sure that the msg files are turned into source code for C++, Python and other languages

   i. Open **package.xml** and make sure these two lines are in it and uncommented.

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

   Note that at build time, we need "message_generation", while at runtime, we only need "message_runtime".

   ii. Open **CMakeLists.txt**
   Add the message_generation dependency to the find_package call which already exists in CMakeLists.txt so that we can generate messages. We can do this by simply adding message_generation to the list of COMPONENTS such that it looks like this:

```
# Be careful we just need to modify the existing text to add
message_generation
find_package(catkin REQUIRED COMPONENTS
   roscpp
   rospy
   std_msgs
   message_generation
)
```

Also make sure you export the message runtime dependency

```
catkin_package(
  ...
  CATKIN_DEPENDS message_runtime ...
  ...)
```

Find the following code.

```
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

And uncomment it by removing # symbol and replace Message*.msg with our .msg file

```
add_message_files(
  FILES
  Demo_sensor.msg
)
```

Now we have to ensure the generate_messages() function is called.
Find the following code.

```
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
```

And uncomment it by removing # symbol and replace Message*.msg with our .msg file

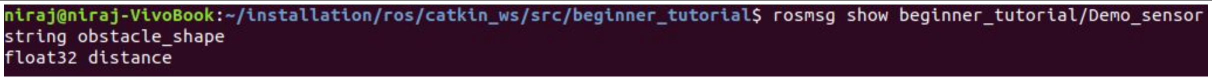
```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

We have successfully created a new message type.
To check the new message type we can use following command
Syntax

```
$ rosmsg show [message type]
```

Example

```
$ rosmsg show beginner_tutorials/Demo_sensor
```



```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws/src/beginner_tutorial$ rosmsg show beginner_tutorial/Demo_sensor
string obstacle_shape
float32 distance
```

Some helpful commands:

- `rospack` = ros+pack(age) : provides information related to ROS packages
- `roscd` = ros+cd : changes directory to a ROS package or stack
- `rosls` = ros+ls : lists files in a ROS package
- `roscp` = ros+cp : copies files from/to a ROS package
- `rosmsg` = ros+msg : provides information related to ROS message definitions
- `rossrv` = ros+srv : provides information related to ROS service definitions
- `catkin_make` : makes (compiles) a ROS package
    - `rosmake` = ros+make : makes (compiles) a ROS package (if you're not using a catkin workspace)

Writing a publisher node

Generally all the python scripts are stored in the 'scripts' folder in the package.
Let's make a scripts folder in our package:

```
$ roscd beginner_tutorials
$ mkdir scripts
$ cd scripts
```

Now let's create a python file inside the scripts folder and write the publisher code.

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

import rospy
from beginner_tutorial.msg import Demo_sensor

import random  # to randomise the message


def demo_pub():

   # create a publisher object Publisher(<topic_name>,  <message
type>, queue_size)
   publisher = rospy.Publisher('distance_info', Demo_sensor,
queue_size=10)

   # Initialize the ros node
   rospy.init_node('distance_publisher', anonymous=False)

   # Define the publisher rate (the unit is in Hz)
   rate = rospy.Rate(10)

   # Publish until not shutdown
   while not rospy.is_shutdown():

       # Create a message object
       message = Demo_sensor()

       # Update the message
       message.obstacle_shape = random.choice(["circle",
"rectangle"])
       message.distance = random.uniform(0.0, 5.0)

       # Log the message to console (optional)
       rospy.loginfo(message)
```

```
        # Publish the message
        publisher.publish(message)

        # wait until the rate defined
        rate.sleep()


if __name__ == '__main__':
    try:
        demo_pub()
    except rospy.ROSInterruptException:
        pass
```

Writing a subscriber node

Let's create a python file(demo_sub.py) inside the scripts folder and write the subscriber code which subscribes to the message published by the previous publisher node.

```
#!/usr/bin/env python
import rospy
from beginner_tutorial.msg import Demo_sensor


# This function is called when the subscriber gets the message from
the publisher
def callback(data):
    rospy.loginfo("the %s is %fm far from here",
                  data.obstacle_shape, data.distance)


def demo_sub():
    # initialize the subscriber node
    rospy.init_node('demo_subscriber', anonymous=False)

    # Declare a subscriber to subscribe to topic 'distance info'
which is of Demo_sensor
    # message type and will call the callback() function when it
receives the message
    rospy.Subscriber('distance_info', Demo_sensor, callback)

    # this command keeps python from exiting until the node is
stopped
    rospy.spin()


if __name__ == '__main__':
    demo_sub()
```

Note: We should always make the .py files executable using chmod command

```
$ chmod +x demo_pub.py
$ chmod +x demo_sub.py
```

Building the nodes

Go to the catkin workspace and run catkin_make

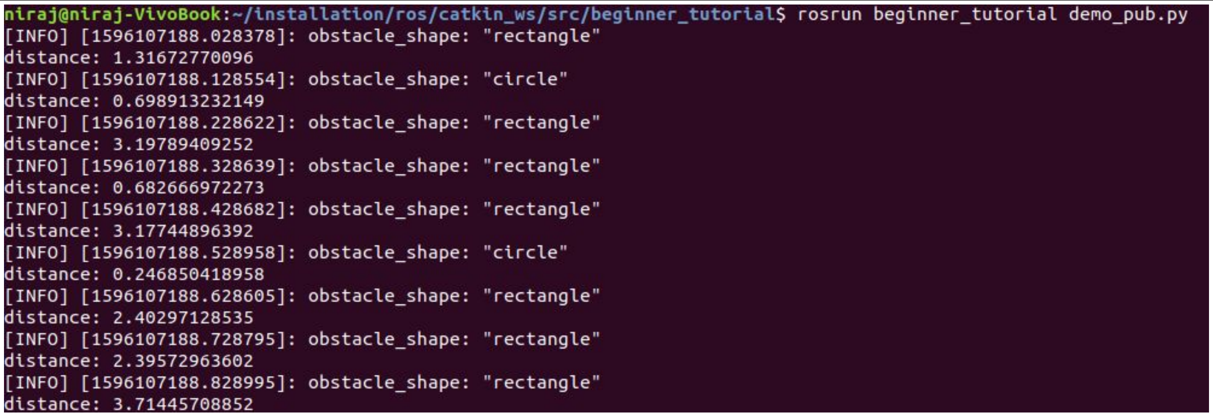
```
$ cd ~/catkin_ws
$ catkin_make
```

Now source the devel space of the workspace

```
$ source devel/setup.bash
```

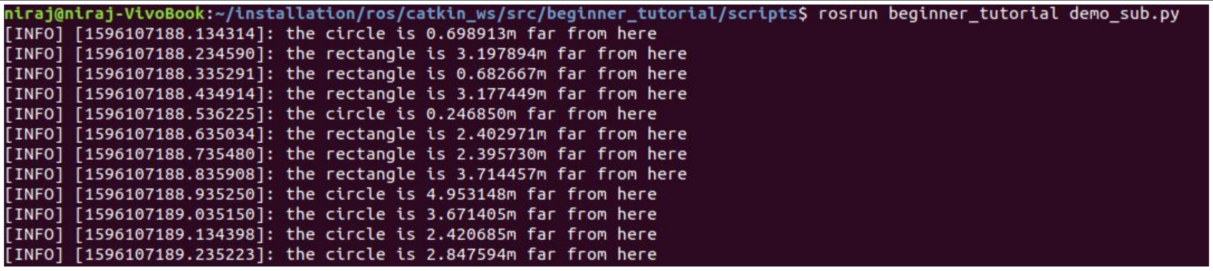
Now run the publisher and subscriber nodes in separate terminal tabs

```
$ rosrun beginner_tutorials demo_pub.py
```



```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws/src/beginner_tutorial$ rosrun beginner_tutorial demo_pub.py
[INFO] [1596107188.028378]: obstacle_shape: "rectangle"
distance: 1.31672770096
[INFO] [1596107188.128554]: obstacle_shape: "circle"
distance: 0.698913232149
[INFO] [1596107188.228622]: obstacle_shape: "rectangle"
distance: 3.19789409252
[INFO] [1596107188.328639]: obstacle_shape: "rectangle"
distance: 0.682666972273
[INFO] [1596107188.428682]: obstacle_shape: "rectangle"
distance: 3.17744896392
[INFO] [1596107188.528958]: obstacle_shape: "circle"
distance: 0.246850418958
[INFO] [1596107188.628605]: obstacle_shape: "rectangle"
distance: 2.40297128535
[INFO] [1596107188.728795]: obstacle_shape: "rectangle"
distance: 2.39572963602
[INFO] [1596107188.828995]: obstacle_shape: "rectangle"
distance: 3.71445708852
```

```
$ rosrun beginner_tutorials demo_sub.py
```



```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws/src/beginner_tutorial/scripts$ rosrun beginner_tutorial demo_sub.py
[INFO] [1596107188.134314]: the circle is 0.698913m far from here
[INFO] [1596107188.234590]: the rectangle is 3.197894m far from here
[INFO] [1596107188.335291]: the rectangle is 0.682667m far from here
[INFO] [1596107188.434914]: the rectangle is 3.177449m far from here
[INFO] [1596107188.536225]: the circle is 0.246850m far from here
[INFO] [1596107188.635034]: the rectangle is 2.402971m far from here
[INFO] [1596107188.735480]: the rectangle is 2.395730m far from here
[INFO] [1596107188.835908]: the rectangle is 3.714457m far from here
[INFO] [1596107188.935250]: the circle is 4.953148m far from here
[INFO] [1596107189.035150]: the circle is 3.671405m far from here
[INFO] [1596107189.134398]: the circle is 2.420685m far from here
[INFO] [1596107189.235223]: the circle is 2.847594m far from here
```

Exercise:

1. Create a new message type which takes your id and name. You can assume both id and name as string.
2. Create a publisher to publish your id and name as the message that you created in Q1.
3. Create a publisher to listen to the id and name published by the publisher in Q2. and print it out.