# ROS - Quick Tutorial for Beginners

This guide is a short version of ROS Tutorial for Beginner which lists useful terms, packages, and commands for quickly understand about the basic of ROS.

#ros

---

Last update: 2021-08-10 16:50:49

# Table of Content

A Gentle Introduction to ROS by *Jason M. O'Kane*

The official guide is at https://wiki.ros.org/ROS/Tutorials.

This guide was created using **ROS Melodic** on **Ubuntu 18.04 LTS**.

# 1. Install ROS

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. Recommend ones of the versions below:

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

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

**What is the difference between ROS Melodic model and Noetic model?**

There aren't many differences at the base level. The ROS Noetic is recommended for Ubuntu 20.04 whereas ROS Melodic for Ubuntu 18.04:

| Feature | ROS Noetic | ROS Melodic |
|---|---|---|
| Python | 3.8 | 2.7 |
| Gazebo | 11.x | 9.0 |
| OpenCV | 4.2 | 3.2 |

Detailed comparison is at repositories.ros.org.

Choose the ROS version based on the installed OS. Here, **Melodic** is used on **Ubuntu 18.04**.

## 1.1. Configure repositories

Configure the Ubuntu repositories to allow "restricted," "universe," and "multiverse" by following the Ubuntu guide.

Setup source list to get ROS packages:

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

Add keys:

```
sudo apt install -y curl && \
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo
apt-key add -
```

Then pull the package list:

```
sudo apt update
```

Finally, install a desktop-full package as recommended to start learning:

```
sudo apt install -y ros-melodic-desktop-full && \
sudo apt install -y ros-melodic-rqt && \
sudo apt install -y ros-melodic-rqt-common-plugins
```

It's convenient if the ROS environment variables are automatically added to a bash session every time a new shell is launched:

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

A good way to check the installation is to ensure that environment variables like `ROS_ROOT` and `ROS_PACKAGE_PATH` are set:

```
printenv | grep ROS
```

```
ROS_ETC_DIR=/opt/ros/melodic/etc/ros
ROS_ROOT=/opt/ros/melodic/share/ros
ROS_MASTER_URI=http://localhost:11311
ROS_VERSION=1
ROS_PYTHON_VERSION=2
ROS_PACKAGE_PATH=/opt/ros/melodic/share
ROSLISP_PACKAGE_DIRECTORIES=
ROS_DISTRO=melodic
```

## 1.2. Build packages

Build packages are needed for code compilation.

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

Initialize the package `rosdep` to track package dependency:

```
sudo rosdep init && \
rosdep update
```

# 2. ROS Workspace

A catkin workspace is a folder where to modify, build, and install catkin packages.

The catkin_make command is a convenience tool for working with catkin workspaces. Running it the first time in a workspace, it will create a `CMakeLists.txt` link in the 'src' folder.

```
mkdir -p catkin_ws/src && \
cd catkin_ws && \
catkin_make
```

In the current directory, it should now have a `build` and `devel` folder. Inside the `devel` folder, there are now several `setup.*sh` files. Sourcing any of these files will overlay this workspace on top of current environment.

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

To make sure workspace is properly overlaid by the setup script, make sure `ROS_PACKAGE_PATH` environment variable includes the current workspace directory.

```
echo $ROS_PACKAGE_PATH
```

> /home/vqtrong/Work/catkin_ws/src:/opt/ros/melodic/share

This also helps ROS to find new packages.

## 2.1. The ROS File system

For this tutorial, to inspect a package in ros-tutorials, please install a prebuilt package using:

```
sudo apt install -y ros-melodic-ros-tutorials
```

Two main concepts of the File Systems:

- *Packages* are the software organization unit of ROS code. Each package can contain libraries, executables, scripts, or other artifacts.
- *Manifests (package.xml)* is a description of a package. It serves to define dependencies between packages and to capture meta information about the package like version, maintainer, license, etc…

> Code is spread across many ROS packages. Navigating with command-line tools such as `ls` and `cd` can be very tedious which is why ROS provides tools to help.

## 2.2. File system Tools

`rospack` allows getting information about packages.

```
rospack find roscpp
```

`roscd` allows changing directory ( `cd` ) directly to a package, and also is able to move to a subdirectory of a package

```
roscd roscpp        # go to cpp package
roscd roscpp/cmake  # go to cmake folder inside the cpp package
roscd log           # go to the log folder, available after run roscore
```

# 3. Create a ROS Package

Firstly, use the `catkin_create_pkg` script to create a new package called `beginner_tutorials` which depends on `std_msgs` , `roscpp` , and `rospy` :

catkin_ws/

```
cd src && \
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 out with the information given in the `catkin_create_pkg` command.

Build that new workspace again to see `beginner_tutorials` is added into `build` folder:

catkin_ws/src

```
cd .. && \
catkin_make
```

Add the workspace to the ROS environment:

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

To check the direct dependencies of a package:

```
rospack depends1 beginner_tutorials
```

To check the indirect dependencies of a package:

```
rospack depends beginner_tutorials
```

# 4. ROS Nodes

Quick overview of Graph Concepts:

- Nodes: A node is an executable that uses ROS to communicate with other nodes.
- Messages: ROS data type used when subscribing or publishing to a topic.
- Topics: Nodes can publish messages to a topic as well as subscribe to a topic to receive messages.
- Master: Name service for ROS (i.e. helps nodes find each other)
- Rosout: ROS equivalent of stdout/stderr
- Roscore: Master + Rosout + Ros parameter server (parameter server will be introduced later)

## 4.1. Roscore

`roscore` will start up a ROS Master, a ROS Parameter Server and a Rosout logging node

Options:

```
-h, --help           # show this help message and exit
-p, --port=          # master port. Only valid if master is launched
-v                   # verbose printing
-w, --numworkers=    # override number of worker threads
-t, --timeout=       # override the socket connection timeout (in seconds).
--master-logger-level= # set logger level
                       # ('debug', 'info', 'warn', 'error', 'fatal')
```

See more in http://wiki.ros.org/roscore.

## 4.2. Rosnode

`rosnode` is a command-line tool for printing information about ROS Nodes.

Commands:

```
rosnode ping    # test connectivity to node
rosnode list    # list active nodes
rosnode info    # print information about node
rosnode machine # list nodes running on a particular machine
rosnode kill    # kill a running node
rosnode cleanup # purge registration information of unreachable nodes
```

Type `rosnode <command> -h` for more detailed usage.

### 4.3. Rosrun

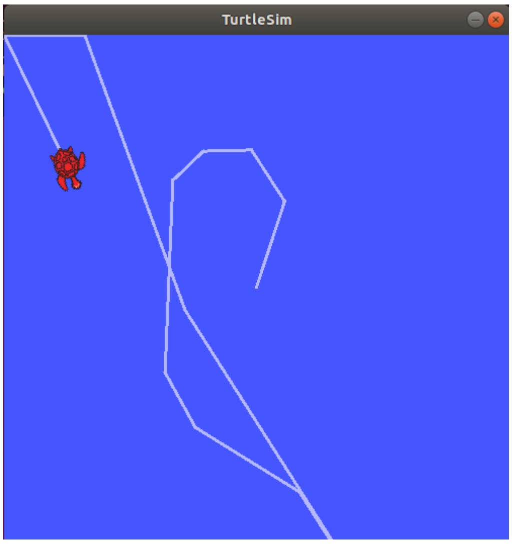
The syntax for `rosrun` is:

```
rosrun [--prefix cmd] [--debug] PACKAGE EXECUTABLE [ARGS]
```

The tool `rosrun` will locate `PACKAGE` and try to find an executable named `EXECUTABLE` in the PACKAGE tree. If it finds it, it will run it with `ARGS` .

Start with this guide, run the `turtlesim_node` in the `turtlesim` package:

```
rosrun turtlesim turtlesim_node
```




*Turtle sim*

## 5. ROS Topics

Run turtle keyboard `turtle_teleop_key` node in a new terminal:

```
rosrun turtlesim turtle_teleop_key
```

After that, the turtle can be moved by using keyboard arrow keys.

The `turtlesim_node` and the `turtle_teleop_key` node are communicating with each other over a ROS Topic. `turtle_teleop_key` is *publishing* the keystrokes on a topic, while `turtlesim` *subscribes* to the same topic to receive the keystrokes.

## 5.1. Rqt_graph

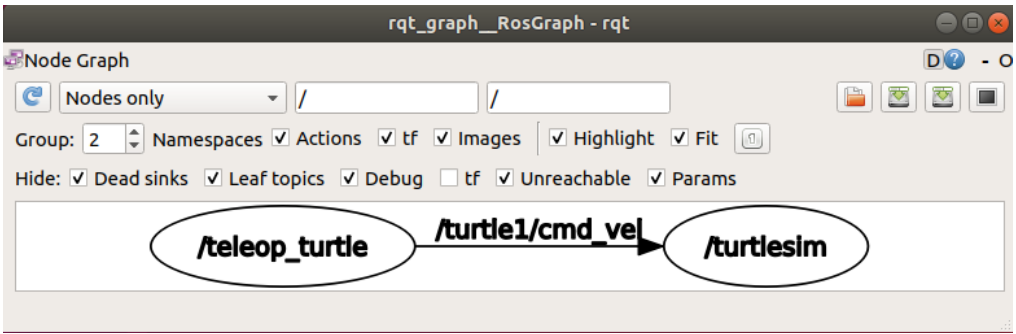
The `rqt_graph` tool creates a dynamic graph of what's going on in the system. `rqt_graph` is part of the `rqt` package.

```
sudo apt install -y ros-melodic-rqt && \
sudo apt install -y ros-melodic-rqt-common-plugins
```

Run `rqt_graph` in a new terminal:

```
rosrun rqt_graph rqt_graph
```

The `turtlesim_node` and the `turtle_teleop_key` nodes are communicating on the topic named `/turtle1/command_vel`.




*RosGraph*

## 5.2. Rostopic

The `rostopic` tool allows getting information about ROS topics.

```
rostopic bw     # display bandwidth used by topic
rostopic echo   # print messages to screen
rostopic hz     # display publishing rate of topic
rostopic list   # print information about active topics
rostopic pub    # publish data to topic
rostopic type   # print topic type
```

# 6. ROS Messages

Communication on topics happens by sending ROS messages between nodes. For the publisher (`turtle_teleop_key`) and subscriber (`turtlesim_node`) to communicate, the publisher and subscriber must send and receive the same type of message. This means that a topic type is defined by the message type published on it. The type of the message sent on a topic can be determined using `rostopic` type.

```
rostopic type /turtle1/cmd_vel
```

> geometry_msgs/Twist

The command `rosmsg show` prints out the details of the message:

```
rosmsg show geometry_msgs/Twist
```

```
geometry_msgs/Vector3 linear
float64 x
float64 y
float64 z
geometry_msgs/Vector3 angular
float64 x
float64 y
float64 z
```

## 6.1. Publish a message

A message can be published through command line:

```
rostopic pub [topic] [msg_type] [args]
```

For example, to send one message on the topic `/turtle1/cmd_vel` using message type `geometry_msgs/Twist` and its required `--` parameters `'[2.0, 0.0, 0.0]'` `'[0.0, 0.0, 1.8]'`, enter the below command:

```
rostopic pub -1 \
    /turtle1/cmd_vel \
    geometry_msgs/Twist \
    -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

To continuously send message at the rate of 1Hz, use `-r 1` options:

```
rostopic pub -r 1 \
    /turtle1/cmd_vel \
    geometry_msgs/Twist \
    -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
```

The rate of message can be inspected by using the command `rostopic hz`:

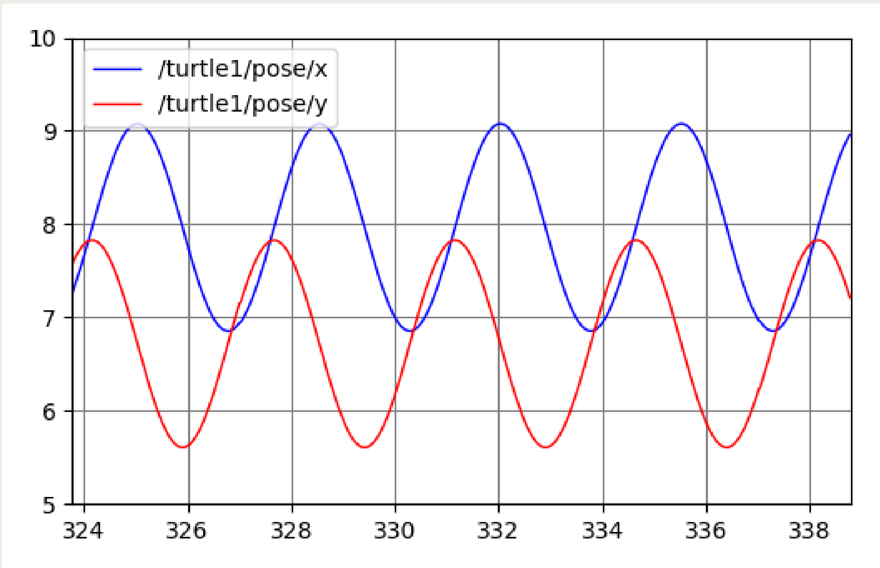
```
rostopic hz /turtle1/pose
```

## 6.2. Rqt_plot

The `rqt_plot` tool displays a scrolling time plot of the data published on topics:

```
rosrun rqt_plot rqt_plot
```

In the new window that should pop up, a text box in the upper left corner gives the ability to add any topic to the plot. Typing `/turtle1/pose/x` will highlight the plus button, previously disabled. Press it and repeat the same procedure with the topic `/turtle1/pose/y`. Now the turtle's x-y location is plotted in the graph.




*RosPlot*

# 7. ROS Services

Services are another way that nodes can communicate with each other. Services allow nodes to send a *request* and receive a *response*.

A `rosservice` can easily be attached to ROS's client/service framework with services. Commands that can be used on services are:

```
rosservice list          # print information about active services
rosservice call          # call the service with the provided args
rosservice type          # print service type
rosservice find          # find services by service type
rosservice uri           # print service ROSRPC uri
```

Let's see current services:

```
rosservice list
```

```
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

Check the parameter of a service:

```
rosservice type /clear
```

```
std_srvs/Empty
```

The `/clear` service shows an Empty parameter, but the `/spawn` has 4 parameter and returns string:

```
rosservice type /spawn
```

```
turtlesim/Spawn
```

```
rosservice type /spawn | rossrv show
```

```
float32 x
float32 y
float32 theta
string name
---
string name
```

The input parameter are `x`, `y`, `theta` and `name`, and the output is `name`. Ok, let's clear the background and spawn a new turtle:

```
rosservice call /clear && \
rosservice call /spawn 2 2 0.2 ""
```

## 7.1. Rosparam

The `rosparam` tool allows storing and manipulate data on the ROS Parameter Server. The Parameter Server can store integers, floats, boolean, dictionaries, and lists. `rosparam` uses the YAML markup language for syntax.

> In simple cases, YAML looks very natural:

- `1` is an integer,
- `1.0` is a float,
- `one` is a string,
- `true` is a boolean,
- `[1, 2, 3]` is a list of integers, and
- `{a: b, c: d}` is a dictionary.

`rosparam` has many commands that can be used on parameters, as shown below:

```
rosparam set            # set parameter
rosparam get            # get parameter
rosparam load           # load parameters from file
rosparam dump           # dump parameters to file
rosparam delete         # delete parameter
rosparam list           # list parameter names
```

See the list of parameters:

```
rosparam list
```

```
/rosdistro
/roslaunch/uris/host_ubuntu18__43509
/rosversion
/run_id
/turtlesim/background_b
/turtlesim/background_g
/turtlesim/background_r
```

Here will change the red channel of the background color:

```
rosparam set /turtlesim/background_r 150
```

Use get command to see the parameters:

```
rosparam get / && \
rosparam get /turtlesim/background_g
```

`dump` and `load` option are also available:

```
rosparam dump [file_name] [namespace]
rosparam load [file_name] [namespace]
```

For example:

```
rosparam dump params.yaml
```

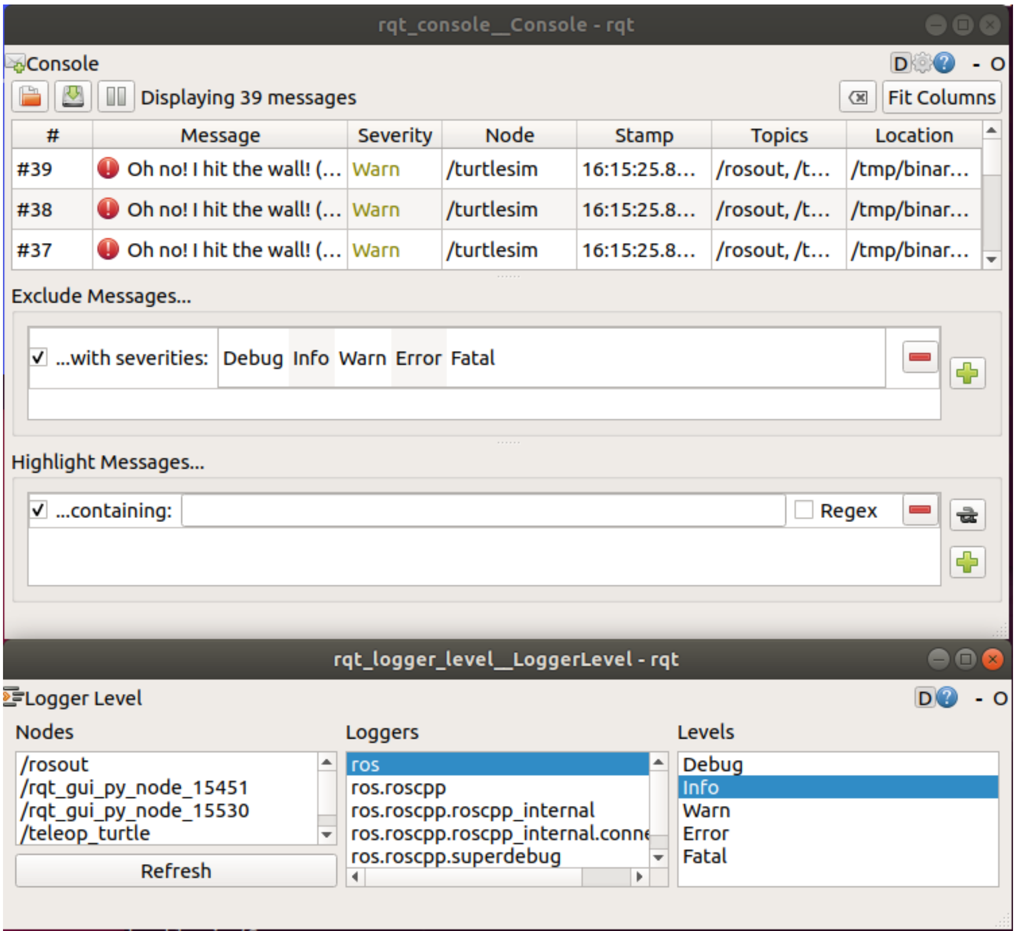
will create a file `params.yaml` with the content similar to:

```
rosdistro: "melodic

    "
roslaunch:
    uris: { host_ubuntu18__43509: "http://ubuntu18:43509/" }
rosversion: "1.14.11

    "
run_id: 409b84fc-e2ff-11eb-be5c-080027b61567
turtlesim: { background_b: 255, background_g: 86, background_r: 150 }
```

# 8. ROS Console

The command `rqt_console` attaches to ROS's logging framework to display output from nodes. `rqt_logger_level` allows to change the verbosity level ( `DEBUG` , `WARN` , `INFO` , and `ERROR` ) of nodes as they run.

```
rosrun rqt_console rqt_console && \
rosrun rqt_logger_level rqt_logger_level
```




*ROS Console and Logger Level*

Logging levels are prioritized in the following order:

1. Fatal
2. Error
3. Warn
4. Info
5. Debug

The `Fatal` level has the highest priority and `Debug` level has the lowest. By setting the logger level, logger will show all messages of that priority level or higher. For example, by setting the level to `Warn`, it will get all `Warn`, `Error`, and `Fatal` logging messages.

# 9. ROS Launch

The `roslaunch` command starts nodes as defined in a launch file.

```
roslaunch [package] [filename.launch]
```

Starting with the `beginner_tutorials` package:

```
cd catkin_ws && \
source devel/setup.bash && \
roscd beginner_tutorials
```

```
vqtrong@ubuntu18:~/Work/catkin_ws/src/beginner_tutorials$
```

Then let's make a `launch` directory:

```
mkdir launch && \
cd launch
```

Now let's create a launch file called `turtlemimic.launch` and paste the following:

```
<launch>
    <group ns="turtlesim1">
        <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
    </group>

    <group ns="turtlesim2">
        <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
    </group>

    <node pkg="turtlesim" name="mimic" type="mimic">
        <remap from="input" to="turtlesim1/turtle1"/>
        <remap from="output" to="turtlesim2/turtle1"/>
    </node>
</launch>
```

Notes:

- Two groups with a namespace tag of `turtlesim1` and `turtlesim2` are created from a `turtlesim` node with a name of `sim`. This allows to start two simulators without having name conflicts.
- The mimic node with the topics `input` and `output` remapped to `turtlesim1` and `turtlesim2`. This renaming will cause `turtlesim2` to mimic `turtlesim1`.

## 9.1. Roslaunch

Start the launch file:

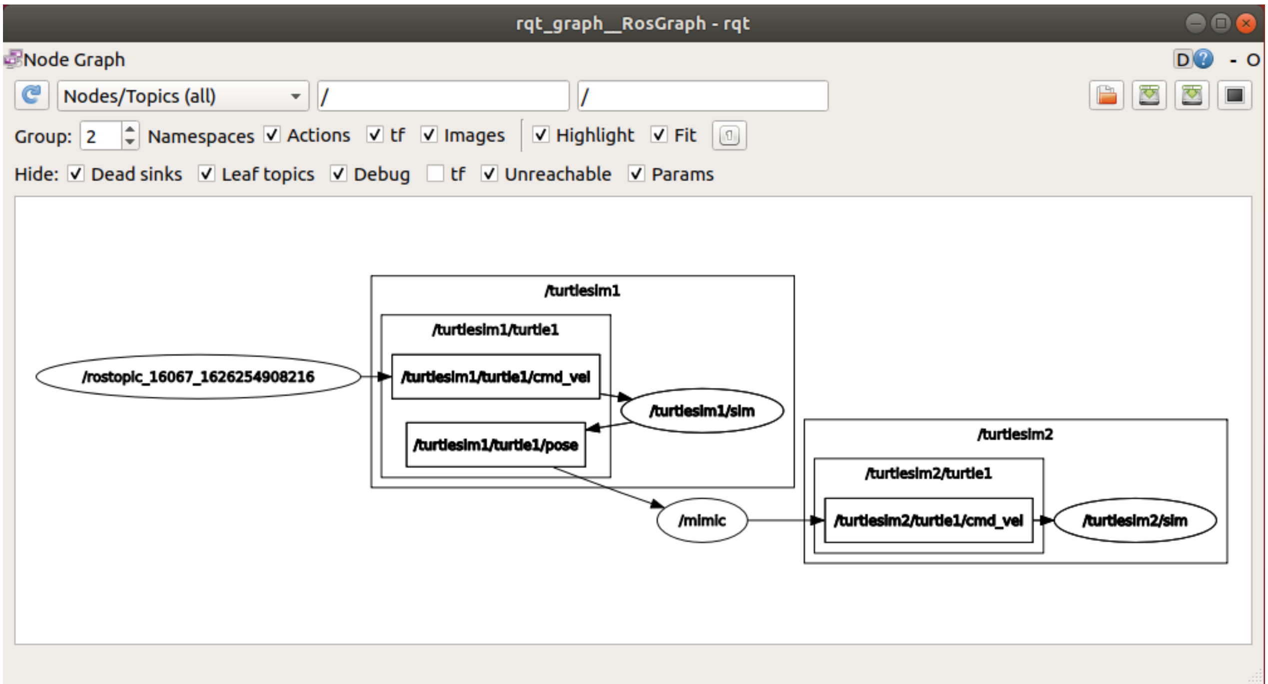
```
roslaunch beginner_tutorials turtlemimic.launch
```

And post messages to the first turtle:

```
rostopic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
```

Run the `rqt_graph` to see what is going on:

```
rqt_graph
```




*Turtle mimic nodes graph*

# 10. Create ROS `msg` and `srv`

The *msg* files are simple text files that describe the fields of a ROS message. They are used to generating source code for messages in different languages. The *msg* files are stored in the `msg` directory of a package

The field types include:

- `int8`, `int16`, `int32`, `int64`
- `float32`, `float64`
- `string`
- `time`, `duration`
- other `msg` files
- variable-length `array[]` and fixed-length `array[x]`

There is also a special type in ROS: `Header`, the header contains a timestamp and coordinate frame information that are commonly used in ROS.

Here is an example of a msg that uses a `Header`, a `string` primitive, and two other messages :

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

The *srv* file describes a service. It is composed of two parts: a request and a response, and *srv* files are stored in the `srv` directory. The two parts are separated by a '—' line.

Here is an example of a srv file:

```
int64 A
int64 B
---
int64 Sum
```

In the above example, A and B are the request, and Sum is the response.

## 10.1. Create a `msg`

Let's define a new msg in the `beginner_tutorials` package:

```
roscd beginner_tutorials && \
mkdir msg && \
echo "int64 num" > msg/Num.msg
```

Using `rosmsg` to see a message definition:

```
rosmsg show beginner_tutorials/Num
```

## 10.2. Creating a `srv`

Let’s define a new msg in the `beginner_tutorials` package:

```
roscd beginner_tutorials && \
mkdir srv && \
touch srv/AddTwoInts.srv
```

The file content:

AddTwoInts.srv

```
int64 a
int64 b
---
int64 sum
```

Using `rossrv` to see a service definition:

```
rossrv show beginner_tutorials/AddTwoInts
```

## 10.3. Generate `msg` and `srv`

To make sure that the msg files are turned into source code for C++, Python, and other languages, open `package.xml`, and make sure these two lines are in it:

package.xml

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

and then add these packages into the `CMakeLists.txt` file:

CMakeLists.txt

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

## Generate messages in the 'msg' folder
add_message_files(
   FILES
   Num.msg
)
```

```
## Generate services in the 'srv' folder
add_service_files(
  FILES
  AddTwoInts.srv
)

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

make the package again:

```
roscd beginner_tutorials && \
cd ../.. && \
catkin_make && \
cd -
```

Any `.msg` file in the msg directory will generate code for use in all supported languages:

- The C++ message header file will be generated in `catkin_ws/devel/include/beginner_tutorials/`.
- The Python script will be created in `catkin_ws/devel/lib/python2.7/dist-packages/beginner_tutorials/msg`.
- The lisp file appears in `catkin_ws/devel/share/common-lisp/ros/beginner_tutorials/msg/`.

Similarly, any `.srv` files in the srv directory will have generated code in supported languages.

- For C++, this will generate header files in the same directory as the messages.
- For Python and Lisp, there will be a `srv` folder beside the `msg` folders.

Here are the generated files:

devel/include/beginner_tutorials/Num.h
devel/include/beginner_tutorials/AddTwoInts.h

Let's see what was generated!

In the Num.h header file:

1. Create a namespace based on the package name

   ```
   namespace beginner_tutorials {}
   ```

2. Use a template of an Allocator to create a new Num type:

```
template <class ContainerAllocator>
struct Num_ {
    typedef Num_<ContainerAllocator> Type;

    // constructor
    Num_() : num(0) {

    }
    Num_(const ContainerAllocator& _alloc) : num(0)  {
        (void)_alloc;
    }

    // members
    typedef int64_t _num_type;
    _num_type num;

    // define new type of pointer
    typedef boost::shared_ptr<
        ::beginner_tutorials::Num_<ContainerAllocator> > Ptr;
    typedef boost::shared_ptr<
        ::beginner_tutorials::Num_<ContainerAllocator> const> ConstPtr;

}; // struct Num_
```

3. Create new pointer types of this new Num type

```
typedef ::beginner_tutorials::Num_< std::allocator<void> > Num;

typedef boost::shared_ptr< ::beginner_tutorials::Num > NumPtr;
typedef boost::shared_ptr< ::beginner_tutorials::Num const> NumConstPtr;
```

Noted that ROS uses `shared_ptr` to manage the memory, that will prevent any memory leak issue.

4. Create friendly operations

```
template<class ContainerAllocator>
struct Printer<
    ::beginner_tutorials::Num_<ContainerAllocator> >
{
    template<typename Stream> static void stream(
        Stream& s,
        const std::string& indent,
        const ::beginner_tutorials::Num_<ContainerAllocator>& v)
{
    s << indent << "num: ";
    Printer<int64_t>::stream(s, indent + "  ", v.num);
}
};

template<typename ContainerAllocator>
std::ostream& operator<<(
```

```
    std::ostream& s,
    const ::beginner_tutorials::Num_<ContainerAllocator> & v)
{
    ros::message_operations::Printer<
        ::beginner_tutorials::Num_<ContainerAllocator> >
            ::stream(s, "", v);
    return s;
}


template<typename ContainerAllocator1, typename ContainerAllocator2>
bool operator==(
    const ::beginner_tutorials::Num_<ContainerAllocator1> & lhs,
    const ::beginner_tutorials::Num_<ContainerAllocator2> & rhs)
{
    return lhs.num == rhs.num;
}

template<typename ContainerAllocator1, typename ContainerAllocator2>
bool operator!=(
    const ::beginner_tutorials::Num_<ContainerAllocator1> & lhs,
    const ::beginner_tutorials::Num_<ContainerAllocator2> & rhs)
{
    return !(lhs == rhs);
}
```

5. Metadata which will be used by ROS to show object's information

```
template <class ContainerAllocator>
struct IsFixedSize< ::beginner_tutorials::Num_<ContainerAllocator> >
: TrueType
{ };

template <class ContainerAllocator>
struct IsFixedSize< ::beginner_tutorials::Num_<ContainerAllocator> const>
: TrueType
{ };

template <class ContainerAllocator>
struct IsMessage< ::beginner_tutorials::Num_<ContainerAllocator> >
: TrueType
{ };

template <class ContainerAllocator>
struct IsMessage< ::beginner_tutorials::Num_<ContainerAllocator> const>
: TrueType
{ };

template <class ContainerAllocator>
struct HasHeader< ::beginner_tutorials::Num_<ContainerAllocator> >
: FalseType
{ };
```

```
template <class ContainerAllocator>
struct HasHeader< ::beginner_tutorials::Num_<ContainerAllocator> const>
: FalseType
{ };


template<class ContainerAllocator>
struct MD5Sum< ::beginner_tutorials::Num_<ContainerAllocator> > {
    static const char* value() {
        return "57d3c40ec3ac3754af76a83e6e73127a";
    }

    static const char* value(
        const ::beginner_tutorials::Num_<ContainerAllocator>&) {
            return value();
        }
    static const uint64_t static_value1 = 0x57d3c40ec3ac3754ULL;
    static const uint64_t static_value2 = 0xaf76a83e6e73127aULL;
};

template<class ContainerAllocator>
struct DataType< ::beginner_tutorials::Num_<ContainerAllocator> > {
    static const char* value() {
        return "beginner_tutorials/Num";
    }

    static const char* value(
        const ::beginner_tutorials::Num_<ContainerAllocator>&) {
            return value();
        }
};

template<class ContainerAllocator>
struct Definition< ::beginner_tutorials::Num_<ContainerAllocator> > {
    static const char* value() {
        return "int64 num\n";
    }

    static const char* value(
        const ::beginner_tutorials::Num_<ContainerAllocator>&) {
            return value();
        }
};
```

# 11. Publisher and Subscriber (C++)

Go to the source code folder of the `beginner_tutorials` package:

```
roscd beginner_tutorials && \
mkdir src && \
cd src
```

## 11.1. A Publisher Node

This tutorial demonstrates simple sending of messages over the ROS system.

```
nano talker.cpp
```

src/talker.cpp

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

int main(int argc, char **argv)
{
    // init with a name
    ros::init(argc, argv, "talker");

    // create a node
    ros::NodeHandle n;

    // publish on a the `chatter` topic, queue = 1000
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>(
        "chatter",
        1000
    );

    // publishing rate 1 Hz
    ros::Rate loop_rate(1);

    // main loop
    int count = 0;
    while (ros::ok()) {
        // message
        std_msgs::String msg;

        // content
        std::stringstream ss;
        ss << "hello world " << count++;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());

        // publish
        chatter_pub.publish(msg);

        // check status
        ros::spinOnce();

        // sleep
        loop_rate.sleep();
    }
```

```
    return 0;
}
```

## 11.2. A Subscriber Node

This tutorial demonstrates simple receipt of messages over the ROS system.

```
nano listener.cpp
```

src/listener.cpp

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
    // init with a name
    ros::init(argc, argv, "listener");

    // create a node
    ros::NodeHandle n;

    // subscribe on a the `chatter` topic, queue = 1000
    // execute chatterCallback on receive
    ros::Subscriber sub = n.subscribe(
        "chatter",
        1000,
        chatterCallback
    );

    // main loop
    ros::spin();

    return 0;
}
```

## 11.3. Building new nodes

Add the source code files which need to be compiled into the `CMakeLists.txt`. With all dependency packages listed above, add below lines also:

```
cd .. && \
nano CMakeLists.txt
```

CMakeLists.txt

```
## Declare a C++ executable

add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker beginner_tutorials_generate_messages_cpp)

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(listener beginner_tutorials_generate_messages_cpp)
```

This will create two executables, talker and listener, which by default will go into package directory in devel space, located by default at `catkin_ws/devel/lib/<package name>`.

Finally, make the package again:

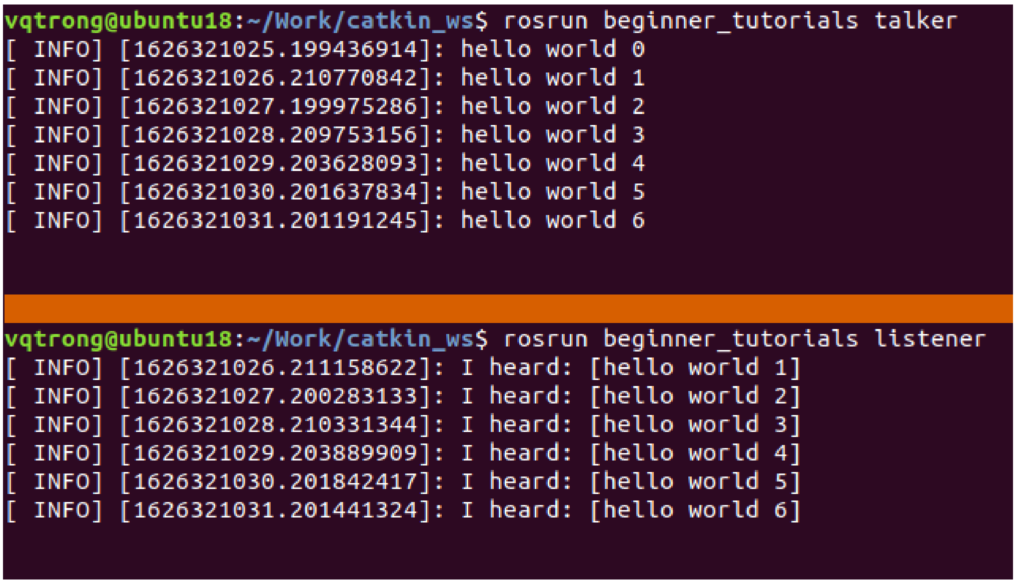
```
roscd beginner_tutorials && \
cd ../.. && \
catkin_make && \
cd -
```

## 11.4. Run new nodes

Run `roscore` first if it is not running. Then run 2 new nodes in two terminals:

```
rosrun beginner_tutorials talker
```

```
rosrun beginner_tutorials listener
```




*Talker and Listener*

# 12. Publisher and Subscriber (Python)

Go to the scripts' folder of the `beginner_tutorials` package:

```
roscd beginner_tutorials && \
mkdir scripts && \
cd scripts
```

## 12.1. A Publisher Node

This tutorial demonstrates simple sending of messages over the ROS system.

```
nano talker.py
```

Note that a node is created from a publisher, in contrast to C++ implementation, a publisher is created from a node.

In ROS, nodes are uniquely named. If two nodes with the same name are launched, the previous one is kicked off. The `anonymous=True` flag means that rospy will choose a unique name for a new `listener` node so that multiple listeners can run simultaneously.

scripts/talker.py

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

import rospy
from std_msgs.msg import String

def talker():
    # create a publisher, on topic `chatter`
    pub = rospy.Publisher('chatter', String, queue_size=10)

    # create a node
    rospy.init_node('talker', anonymous=True)

    # set the rate of publishing
    rate = rospy.Rate(1) # 1hz

    # main loop
    while not rospy.is_shutdown():
        # make content
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)

        # publish a message
        pub.publish(hello_str)

        rate.sleep()
```

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

## 12.2. A Subscriber Node

This tutorial demonstrates simple receipt of messages over the ROS system.

```
nano listener.py
```

scripts/listener.py

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

import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():
    # create a node
    rospy.init_node('listener', anonymous=True)

    # create a subcriber
    rospy.Subscriber("chatter", String, callback)

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

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

`rospy.spin()` simply keeps the node from exiting until the node has been shutdown. Unlike roscpp, `rospy.spin()` does not affect the subscriber callback functions, as those have their own threads.

## 12.3. Make script executable

Scripts need to get execution permission before they can run:

scripts

```
chmod +x *
```

### 12.4. Building new nodes

Add the source code files which need to be compiled into the `CMakeLists.txt`. With all dependency packages listed above, add below lines also:

```
cd .. && \
nano CMakeLists.txt
```

CMakeLists.txt

```
catkin_install_python(PROGRAMS
    scripts/talker.py
    scripts/listener.py
    DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

Finally, make the package again:

```
roscd beginner_tutorials && \
cd ../.. && \
catkin_make && \
cd -
```

### 12.5. Run new nodes

Run `roscore` first if it is not running. Then run 2 new nodes in two terminals:

```
rosrun beginner_tutorials talker.py
```

```
rosrun beginner_tutorials listener.py
```

⚠ **Python script execution**

If the error `/usr/bin/env: 'python\r': No such file or directory` shows up, it is because of the ending line characters. Unix use `LF` only while Windows use `CRLF`. Save python scripts in Unix ending character only.

## 13. Service and Client (C++)

Go to the source code folder of the `beginner_tutorials` package:

```
roscd beginner_tutorials && \
mkdir src && \
cd src
```

## 13.1. A Service Node

This guide will create the service `add_two_ints_server` node which will receive two ints and return the sum.

This service uses the `beginner_tutorials/AddTwoInts.h` header file generated from the srv file that is created earlier.

```
nano add_two_ints_server.cpp
```

src/add_two_ints_server.cpp

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"

// service fuction
bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
{
    res.sum = req.a + req.b;
    ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
    ROS_INFO("sending back response: [%ld]", (long int)res.sum);
    return true;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "add_two_ints_server");

    // create a node
    ros::NodeHandle n;

    // node will have a service
    ros::ServiceServer service = n.advertiseService("add_two_ints", add);
    ROS_INFO("Ready to add two ints.");

    // main loop
    ros::spin();

    return 0;
}
```

## 13.2. A Client Node

This guide will create the service `add_two_ints_client` node which will receive two ints and return the sum.

```
nano add_two_ints_client.cpp
```

src/add_two_ints_client.cpp

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>

int main(int argc, char **argv)
{
    // check args
    ros::init(argc, argv, "add_two_ints_client");
    if (argc != 3)
    {
        ROS_INFO("usage: add_two_ints_client X Y");
        return 1;
    }

    // create a node
    ros::NodeHandle n;

    // node will have a client
    ros::ServiceClient client = n.serviceClient<beginner_tutorials::AddTwoInts>
("add_two_ints");

    // create a service target
    beginner_tutorials::AddTwoInts srv;

    // add params
    srv.request.a = atoll(argv[1]);
    srv.request.b = atoll(argv[2]);

    // call to service
    if (client.call(srv)) {
        ROS_INFO("Sum: %ld", (long int)srv.response.sum);
    } else {
    ROS_ERROR("Failed to call service add_two_ints");
        return 1;
    }

    return 0;
}
```

## 13.3. Building new nodes

Add the source code files which need to be compiled into the `CMakeLists.txt`. With all dependency packages listed above, add below lines also:

```
cd .. && \
nano CMakeLists.txt
```

CMakeLists.txt

```
## Declare a C++ executable

add_executable(add_two_ints_server src/add_two_ints_server.cpp)
```

```
target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
add_dependencies(add_two_ints_server beginner_tutorials_gencpp)

add_executable(add_two_ints_client src/add_two_ints_client.cpp)
target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
add_dependencies(add_two_ints_client beginner_tutorials_gencpp)
```

This will create two executables, add_two_ints_server and add_two_ints_client, which by default will go into package directory in devel space, located by default at `catkin_ws/devel/lib/<package name>`.

Finally, make the package again:

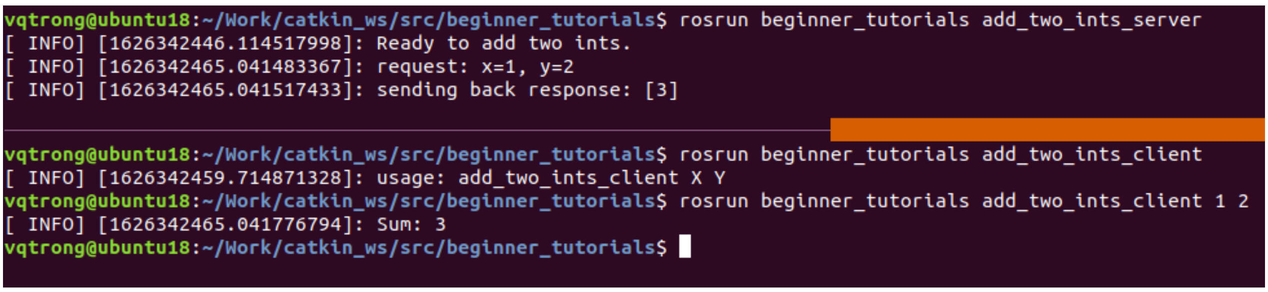
```
roscd beginner_tutorials && \
cd ../.. && \
catkin_make && \
cd -
```

## 13.4. Run new nodes

Run `roscore` first if it is not running. Then run 2 new nodes in two terminals:

```
rosrun beginner_tutorials add_two_ints_server
```

```
rosrun beginner_tutorials add_two_ints_client 1 2
```




*Service and Client*

# 14. Service and Client (Python)

Go to the source code folder of the `beginner_tutorials` package:

```
roscd beginner_tutorials && \
mkdir scripts && \
cd scripts && \
```

## 14.1. A Service Node

This guide will create the service `add_two_ints_server` node which will receive two ints and return the sum.

This service uses the `beginner_tutorials/AddTwoInts.h` header file generated from the srv file that is created earlier.

```
nano add_two_ints_server.py
```

scripts/add_two_ints_server.py

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

from __future__ import print_function

from beginner_tutorials.srv import AddTwoInts,AddTwoIntsResponse
import rospy

def handle_add_two_ints(req):
    print("Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b)))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print("Ready to add two ints.")
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

## 14.2. A Client Node

This guide will create the service `add_two_ints_client` node which will receive two ints and return the sum.

```
nano add_two_ints_client.py
```

scripts/add_two_ints_client.py

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

from __future__ import print_function

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
```

```
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException as e:
        print("Service call failed: %s"%e)

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print(usage())
        sys.exit(1)
    print("Requesting %s+%s"%(x, y))
    print("%s + %s = %s"%(x, y, add_two_ints_client(x, y)))
```

## 14.3. Make scripts executable

Scripts need to get execution permission before they can run:

```
chmod +x *
```

## 14.4. Building new nodes

Add the source code files which need to be compiled into the `CMakeLists.txt`. With all dependency packages listed above, add below lines also:

```
cd .. && \
nano CMakeLists.txt
```

CMakeLists.txt

```
catkin_install_python(PROGRAMS
    scripts/add_two_ints_server.py
    scripts/add_two_ints_client.py
    DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

This will create two executables, add_two_ints_server and add_two_ints_client, which by default will go into package directory in devel space, located by default at `catkin_ws/devel/lib/<package name>`.

Finally, make the package again:

```
roscd beginner_tutorials && \
cd ../.. && \
catkin_make && \
cd -
```

### 14.5. Run new nodes

Run `roscore` first if it is not running. Then run 2 new nodes in two terminals:

```
rosrun beginner_tutorials add_two_ints_server.py
```

```
rosrun beginner_tutorials add_two_ints_client.py 1 3
```

## 15. Playback data

This tutorial will teach how to record data from a running ROS system into a `.bag` file, and then to play back the data to produce similar behavior in a running system.

### 15.1. Record data

First, execute the following commands in separate terminals:

Terminal 1:

```
roscore
```

Terminal 2:

```
rosrun turtlesim turtlesim_node
```

Terminal 3:

```
rosrun turtlesim turtle_teleop_key
```

This will start two nodes — the `turtlesim` visualizer and a node that allows for the keyboard control of `turtlesim` using the arrows keys on the keyboard.

Let's examine the full list of topics that are currently being published in the running system. To do this, open a new terminal and execute the command:

```
rostopic list -v
```

```
Published topics:
 * /turtle1/color_sensor [turtlesim/Color] 1 publisher
 * /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher
 * /rosout [rosgraph_msgs/Log] 2 publishers
 * /rosout_agg [rosgraph_msgs/Log] 1 publisher
 * /turtle1/pose [turtlesim/Pose] 1 publisher

Subscribed topics:
 * /turtle1/cmd_vel [geometry_msgs/Twist] 1 subscriber
 * /rosout [rosgraph_msgs/Log] 1 subscriber
```

The list of published topics is the only message types that could potentially be recorded in the data log file, as only published messages are recorded:

- The topic `/turtle1/cmd_vel` is the command message published by the `teleop_turtle` node that is taken as input by the `turtlesim` process.
- The messages `/turtle1/color_sensor` and `/turtle1/pose` are output messages published by `turtlesim`.

Open a new terminal window. In this window run the following commands. Running `rosbag record` with the option `-a` indicates that all published topics should be accumulated in a bag file.

```
roscd beginner_tutorials && \
mkdir bagfiles && \
cd bagfiles && \
rosbag record -a
```

```
[ INFO] [1626862434.586239631]: Recording to '2021-07-21-17-13-54.bag'.
[ INFO] [1626862434.587320465]: Subscribing to /turtle1/color_sensor
[ INFO] [1626862434.589356574]: Subscribing to /turtle1/cmd_vel
[ INFO] [1626862434.591447646]: Subscribing to /rosout
[ INFO] [1626862434.593544025]: Subscribing to /rosout_agg
[ INFO] [1626862434.595557444]: Subscribing to /turtle1/pose
```

Move back to the terminal window with `turtle_teleop` and move the turtle around for 10 or so seconds.

## 15.2. Rosbag info

Run the command `rosbag info` to see the info of a *rosbag* file:

```
rosbag info <bagfile>
```

```
path:        xxx.bag
version:     2.0
duration:    2:21s (141s)
start:       xxx
end:         xxx
```

```
size:        1.3 MB
messages:    18150
compression: none [2/2 chunks]
types:       geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
             rosgraph_msgs/Log   [acffd30cd6b6de30f120938c17c593fb]
             turtlesim/Color     [353891e354491c51aabe32df673fb446]
             turtlesim/Pose      [863b248d5016ca62ea2e895ae5265cf9]
topics:      /rosout                    217 msgs    : rosgraph_msgs/Log   (2
connections)
             /rosout_agg                214 msgs    : rosgraph_msgs/Log
             /turtle1/cmd_vel           286 msgs    : geometry_msgs/Twist
             /turtle1/color_sensor     8716 msgs    : turtlesim/Color
             /turtle1/pose             8717 msgs    : turtlesim/Pose
```

## 15.3. Rosbag play

The next step in this tutorial is to replay the bag file to reproduce behavior in the running system. First kill the `tele-operator` program that may be still running from the previous section.

Leave `turtlesim` running. In a terminal window run the following command:

```
rosbag play <bagfile>
```

In its default mode `rosbag play` will wait for a certain period (.2 seconds) after advertising each message before it actually begins publishing the contents of the bag file. Waiting for some duration allows any subscriber of a message to be alerted that the message has been advertised and that messages may follow. If `rosbag play` publishes messages immediately upon advertising, subscribers may not receive the first several published messages. The waiting period can be specified with the `-d` option.

## 15.4. Recording a subset

When running a complicated system, such as the pr2 software suite, there may be hundreds of topics being published, with some topics, like camera image streams, potentially publishing huge amounts of data. In such a system it is often impractical to write log files consisting of all topics to disk in a single bag file. The `rosbag record` command supports logging only particular topics to a bag file, allowing users to only record the topics of interest to them.

```
rosbag record -O subset /turtle1/cmd_vel /turtle1/pose
```

The `-O` argument tells `rosbag record` to log to a file named `subset.bag`, and the topic arguments cause `rosbag` record to only subscribe to these two topics.

⚠ **The limitations of rosbag record/play**

Different start condition can cause different results even the events are the same. The rate of recorded events is not guaranteed to be the same as the real actions.

## 16. Read message from Rosbag

The script `ros_readbagfile` will read rosbag file and extract all messages of selected topics:

```
ros_readbagfile <mybagfile.bag> [info] [N] [topic1] [topic2] [...]
```

Download and install `ros_readbag.py` using below command:

```
cd ~ && \
wget https://raw.githubusercontent.com/vuquangtrong/\
ros_readbagfile/main/ros_readbagfile
```

**Edit shebang to use python 2 if needed.**

Change `#!/usr/bin/python3` to `#!/usr/bin/python`.

Make it executable:

```
chmod +x ros_readbagfile
```

The `~/bin` directory for personal binaries:

```
mkdir -p ~/bin
```

Add this folder to the `PATH`:

```
echo "PATH=\"$PATH:~/bin\"" >> ~/.bashrc
```

Move this executable script into that directory as `ros_readbagfile`, so that it will be available as that command:

```
mv ros_readbagfile ~/bin/ros_readbagfile
```

**Usage**

1. See the information of the input bag file:

   ```
   ros_readbagfile mybagfile.bag info
   ```

2. Print all messages of all topics in the bag file to the screen:

```
ros_readbagfile mybagfile.bag
```

3. Print all messages of the topic /test in the bag file to the screen:

```
ros_readbagfile mybagfile.bag /test
```

4. Print at most N first messages of all topics in the bag file to the screen:

```
ros_readbagfile mybagfile.bag N
```

5. Print at most N first messages of the topic /test in the bag file to the screen:

```
ros_readbagfile mybagfile.bag N /test
```

6. To save the output to a file, use redirection syntax:

```
ros_readbagfile mybagfile.bag N /test > output.txt
```

Determine the exact topic names you'd like to read from the bag file, by using `rosbag info` as mentioned above, or use `ros_readbagfile info` command:

Use `ros_readbagfile` from terminal as below:

```
rosbag info 2021-07-21-17-13-54.bag
```

```
path:        2021-07-21-17-13-54.bag
version:     2.0
duration:    30.8s
start:       Jul 21 2021 17:13:54.60 (1626862434.60)
end:         Jul 21 2021 17:14:25.40 (1626862465.40)
size:        280.8 KB
messages:    3895
compression: none [1/1 chunks]
types:       geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
            rosgraph_msgs/Log   [acffd30cd6b6de30f120938c17c593fb]
            turtlesim/Color     [353891e354491c51aabe32df673fb446]
            turtlesim/Pose      [863b248d5016ca62ea2e895ae5265cf9]
topics:      /rosout                    4 msgs    : rosgraph_msgs/Log   (2
connections)
            /turtle1/cmd_vel          93 msgs    : geometry_msgs/Twist
            /turtle1/color_sensor   1899 msgs    : turtlesim/Color
            /turtle1/pose           1899 msgs    : turtlesim/Pose
```

Now, get all messages of the topic `/turtle1/pose` and save to the file:

```
ros_readbagfile 2021-07-21-17-13-54.bag /turtle1/pose > turtle1_pose.yaml
```

```
# =======================================
# topic:           /turtle1/pose
# msg_count:       1899
# timestamp (sec): 1626862465.397807837 # - - -
x: 10.0430784225
y: 6.37491464615
theta: -2.62400007248
linear_velocity: 0.0
angular_velocity: 0.0

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Total messages found:            1899
#
#    /turtle1/pose:                1899
#
# DONE.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

# 17. Reference

There is an interesting book named A Gentle Introduction to ROS by Jason M. O'Kane published on https://cse.sc.edu/~jokane/agitr/. This book supplements ROS's own documentation, explaining how to interact with existing ROS systems and how to create new ROS programs using C++, with special attention to common mistakes and misunderstandings.

An excerpt from the book:

> **Giving ROS control**
>
> The final complication is that ROS will only execute our callback function when we give it explicit permission to do so. There are actually two slightly different ways to accomplish this. One version looks like this:
>
> ```
> ros::spinOnce();
> ```
>
> This code asks ROS to execute all of the pending callbacks from all of the node's subscriptions, and then return control back to us. The other option looks like this:
>
> ```
> ros::spin();
> ```
>
> This alternative to `ros::spinOnce()` asks ROS to wait for and execute callbacks until the node shuts down. In other words, `ros::spin()` is roughly equivalent to this loop:
>
> ```
> while(ros::ok()) {
>     ros::spinOnce();
> }
> ```
>
> The question of whether to use `ros::spinOnce()` or `ros::spin()` comes down to this:

*Does your program have any repetitive work to do, other than responding to callbacks?*

- If the answer is *No*, then use `ros::spin()`.
- If the answer is *Yes*, then a reasonable option is to write a loop that does that other work and calls `ros::spinOnce()` periodically to process callbacks.

A common error in subscriber programs is to mistakenly omit both `ros::spinOnce()` and `ros::spin()`. In this case, ROS never has an opportunity to execute your callback function.

- Omitting `ros::spin()` will likely cause your program to exit shortly after it starts.
- Omitting `ros::spinOnce()` might make it appear as though no messages are being received.