ROS Tutorials {Beginner Level}

1. Core ROS Tutorials

1.1) Beginner Level

① Installing and Configuring your ROS Environment

source /opt/ros/melodic/setup.bash

⇒ You will need to run this command on every new shell you open to have access to the ROS commands, Unless you add this line to your .bashrc.

* Create a ROS Workspace

① mkdir -p ~/catkin_ws/src

② cd ~/catkin_ws/

③ catkin_make {for creating the workspace}

⇒ inside catkin_ws there will be a 'devel' folder which contain setup.bash.

↓

{Sourcing this file will overlay this workspace on top of your environment}

② Navigating the ROS Filesystem

rospack find [Package_Name]

{This will return location of find package}

roscd [Package_Name]

{Changes the directory to find package}

roscd

{change the directory to the current workspace}

rosls [Pakage_Name]

{this lets you ls directly in a package by name rather than absolute path}

⇒ Pressing tab gives the valid option list.

③ Creating a Package #1 {31}

⇒ For a package to be considered a catkin Package it must meet few requirements:

1) The package must contain a Catkin Compliant package.xml file.
   → {Provides meta info about Package}

2) The package must contain a CMakeLists.txt which uses catkin.

3) Each package must have its own folder.

★ Customizing Your Package

ⓐ description tag

<description> Description of the Package </description>

ⓑ Maintainer tag

<maintainer email="Your@yourdomain.tld">
Yourname </maintainer>

ⓒ License tags

<license> BSD </license>

ⓓ dependencies tags

⇒ The dependencies are split into:

- buildtool_depend {catkin}
- build_depend
- build_export_depend
- exec_depend

④ Building a ROS Package

Catkin_make

{this Command is used to build the package}

⑤ Understanding ROS Nodes

⇒ When you open a new terminal your environment is reset and your ~/.bashrc file is sourced.

rosrun [Packagename] [node_name]

{To run a node within a package}

⇒ To change the name of node

rosrun [Package_name] [Old_Node_name] **_name:** [New_node_Name]

⑥ Understanding ROS Topic

* Using rqt_graph

⇒ rqt_graph creates a dynamic graph of what's going on in the system.

→ rqt_graph is part of the rqt package.

rosrun rqt_graph rqt_graph

{To run rqt_graph}

[ros command] **-h**

{To get details about the ROS command}

rostopic echo [topic_name]

{To display message being passed through the topic}

rostopic type [topic_name]

{returns the message type of any topic being published}

* Using rostopic pub

↳ rostopic pub publishes data on to a topic currently advertised.

rostopic pub [topic] [msg-type] [args]

-r ⇒ To publish steady stream of command

-1 ⇒ to only publish one message then exit

rostopic hz [topic_name]

{Reports the rate at which data is being published}

⇒ rqt_Plot displays a scrolling time plot of the data published on topic.

rosrun rqt_plot rqt_plot

{To run rqt_plot}

{In this GUI you can enter the topic variable to plot}

⑦ Understanding ROS Services & Parameters

rosservice list

{lists all the Services}

rosservice type [Service name]

{return type of Service}

rosservice call [Service] [args]

{To call a Service}

eg ⇒ rosservice call /clear

{Clears the background of turtle sim_node}

Example

{**rosservice type** /spawn | **rossrv show**}

* rosparam

⇒ Allows you to store and manipulate data on ROS parameter server.

⇒ rosparam uses YAML markup language for Syntax.

rosparam get /

{To show/list the contents of the entire parameter Server}

(8) Using rqt_Console and roslaunch

rqt_Console

↳ It attaches to ROS's logging framework to display output from nodes.

rqt_logger_level

↳ Allows us to change the verbosity level (DEBUG, WARN, INFO and ERROR) of nodes as they run

```
rosrun rqt_console rqt_console
```
{To run rqt_Console}

```
rosrun rqt_logger_level rqt_logger_level
```
{To run rqt_logger_level}

⇒ Logging level are prioritized in the following order:-

Fatal → Error → Warn → Info → Debug

⇒ By setting the logger level, you will get all messages of that priority level or higher.

* Using roslaunch

```
roslaunch [package_name] [filename.launch]
```
{To start a launch file}

(9) Using rosed to edit files in ROS

rosed Package_name filename
{To edit the file in the package}

⇒ Default editor of rosed is Vim.
If it is not then then,
export EDITOR = 'nano -w'

(10) Creating a ROS message (#2) (35) and Srv

* Msg ⇒ .msg files are simple text files that describe the fields of a ROS message.
→ They are used to generate source code for messages in different language.

* Srv ⇒ An .srv file describes a service. It is composed of two parts: a request & a response.

msg
↳ message_name.msg

⇒ edit Package.xml & CMakeLists.txt accordingly.

Srv
↳ Service_name.srv

⇒ edit Package.xml & CMakeLists.txt accordingly.

(11) Writing a Simple Publisher and Subscriber (C++)

★ Writing the Publisher Node

```
#include "ros/ros.h"
```
→ ros.h is in ros package

```
#include "std_msgs/String.h"
```
→ String.h in std_msgs package

```
#include <sstream>
```
→ StringStream

⇒ used to convert int into String and String into int.

```
StringStream sso;
    int a=55;
    sso << a; // Integer stored in String stream sso from a
    String b;
    sso >> b; // String stored in b from sso.
```

sso << → sso >>

⇒ A StringStream associates a string object with a stream allowing you to read from the string as if it were a stream (like cin)

Basic methods are:-

1. Clear() ⇒ to clear the stream
2. str() ⇒ to get and set string object whose content is present in stream.
3. << ⇒ add a string to the StringStream object.
4. >> ⇒ read something from the String Stream object

```
int main (int argc, char** argv)
{
```

argc → Number of things that we enter in Terminal while running our executable.

argv → It stores what we enter in the command line while running our executable.

argv[0] → first element

argv[1] → Second element (command)

⋮

and so on

```
ros::init(argc, argv, "talker");
```

ros::init → function init is in ros namespace

"talker" → Name of the node

The name used here must be a base name, i.e. it cannot have a / in it

```
ros::NodeHandle n;
```

NodeHandle is the main access point to communication with the ROS System

→ The first NodeHandle created will actually do the initialization of the node, and the last one destructed will cleanup any resources the node was using

```
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
```

⇒ The advertise( ) function is how you tell ROS that you want to publish on a given topic name.

⇒ This invokes a call to the ROS master node.

(which keeps a registry of who is publishing & who is subscribing)

⇒ After this master node will notify anyone who is trying to subscribe to this topic name, and they will in turn negotiate a peer-to-peer connection with this node.

⇒ advertise() returns a Publisher object which allows you to publish message on that topic through a call to publish().

⇒ Once all copies of the returned Publisher object are destroyed, the topic will be automatically unadvertised.

⇒ Std_msgs::String is msg type
⇒ Chatter is name of topic
⇒ 1000 ⇒ Size of buffer

ros::Rate loop_rate(10);

⇒ Rate object allows you to specify a frequency that you would like to loop at.
⇒ It will keep track of how long it has been since the last call to Rate::Sleep(), and sleep for the correct amount of time.

```
int count=0;
while (ros::ok())
{
```

⇒ ros::ok() always returns true except if :-
(1) Ctrl + C is pressed
(2) we have been kicked off the network by another node with the same name.
(3) ros::shutdown() has been called by another part of the application.
(4) all ros::NodeHandles has been distroyed.

Std_msgs::String msg

{message of name msg and type String in Std_msgs name space}

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

⇒ ROS_INFO is replacement for printf.

{ c_str returns a const char * and points to a null terminated string (i.e. C style string) }

```
Chatter_Pub.Publish(msg);
ros::spinOnce();
```

{ ⇒ Calling ros::spinOnce() here is not necessary.
⇒ If we were to add a subscription into this application, and did not have ros::spinOnce(), here your callbacks would never get called. }

```
loop_rate.Sleep();
 ++count;
}

return 0;
}
```

★ Writing the Subscriber Node

Const char * Ptr → Value it is Pointing to is ~~[illegible]~~ Constant but Pointer ~~[illegible]~~ can change address.

eg:
a=5
Const char * Ptr = &a;
*Ptr = 'b'; Not Valid
b = 6;
Ptr = &b; Valid

Char * Const Ptr

Value it is pointing to can change but pointer cannot change

#1 {19} Creating a Catkin Package

1) Go to Src folder of your work space.
   {or change current working directory to Src folder of your workspace}

2) Catkin_Create_Pkg <Package_Name> [Dependency 1] [Dependency 2] ....

Example
{std_msgs, rospy, roscpp}

building the package in Catkin workspace

1) Change working directory to Catkin_ws

2) Catkin_make

* Writing the Subscriber Node

```
#include "ros/ros.h"
#include "std_msgs/String.h"
Void chattercallback (const std_msgs::String::ConstPtr& msg)
{
  ROSINFO ("I heard [%s]", msg->data.c_str());
}

int main (int argc, char **argv)
{
ros::init (argc, argv, "listener");
ros::NodeHandle handle;
ros::Subscriber sub;
Sub = handle.subscribe ("chatter", 1000, chatter Callback);
ros::spin()
return 0
}
```

ros::spin() enters a loop, calling message callbacks as fast as possible.

add_exe
target_li
add_depe

⇒ Aft
and

* Building your Nodes

Add the following in the CMakeLists.txt

```
add_executable (talker src/talker.cpp)
target_link_libraries (talker ${catkin_LIBRARIES})
add_dependencies (talker test_package1_generat
                                _message_cpp)

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

⇒ After that change pwd to your catkin workspace and execute catkin_make command.

12 Examining the simple publisher and Subscriber

ros run PackageName node name

{To run the node}

⇒ Before running any node you need to run ros core.

13 Writing a Simple Service and Client

To create Service node which will receive two ints and return the sum.

Service Node

```
# include "ros/ros.h"
# include "test_package1/AddTwoInts.h"
```

{Header file generated from the srv file.}

```
bool add (test_package1::AddTwoInts:Request &req,
  test_package1::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);
```

(#2) (2.5)

Creating a ROS msg

* Creating a Srv

1) Make a directory srv in
2) Add AddTwoInts.srv to
   → int64
   int64
   ---
   int64
3) In Package.xml
   <build_depend> message_
   <exec_depend> message
4) In CMakeLists.txt
   → Add message_gene
   → Add AddTwoInts.s
   → Add std_msgs in
5) Catkin_make install

```
return true;
}
```

#2 [25]

Creating a ROS msg and Srv

* Creating a Srv

1) Make a directory srv in your package.

2) Add AddTwoInts.srv to it.

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

} request

} response

3) In Package.xml

<build_depend> message_generation </build_depend>

<exec_depend> message_runtime </exec_depend>

4) In CMakeLists.txt

- → Add message_generation in find package( )
- → Add AddTwoInts.srv in add_service_files()
- → Add Std_msgs in 'generate_messages()

5) Catkin_make install

```
return true;
}
```

```
int) req.a, (long int) req.b );
(long int) res.sum);
```

```
int main(int argc, char ** argv)
{
ros:: init (argc, argv, "add_two_ints_Server");
ros:: NodeHandle n

ros:: ServiceServer service;
service = n.advertiseService ("add_two_ints", add);
ROS_INFO (" Ready to add two ints");
ros:: spin();

return 0;
}
```

{Service name}

Writing the Client Node

```
#include "ros/ros.h"
#include "test_Package1/AddTwoInts.h"
# include <cstdlib>
```

{For Converting String to integer}

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

if (argc != 3)
{
ROS_INFO ("Usage: add_two_ints_client X Y");
return 1;
}
```

```
ros::NodeHandle n;
ros::ServiceClient client
client = n.ServiceClient<test_package1::AddTwoInts>
        ("add_two_ints");

testpackage1::AddTwoInts srv;
srv.request.a = atoll (argv[1]);
srv.request.b = atoll (argv[2]);
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;
}
```

(To convert string to long long int)

(This actually calls the service with argument srv)

Building the node

1) Declare the node in CMakeList.txt.
2) Catkin_make

Example

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 test_package1_gencpp)

(17) Recording and playing data back

① Make a directory named bagfiles.

② rosbag record -a

This will record all the Published data in a bag file.

{Bag file name will begin with year, date and time and the suffix .bag}

③ rosbag info bagfile name

→ This will display all the information about the bag file.

④ rosbag play bagfile_name.

→ This will play the bag file Publishing all the data in sequence

⑤ rosbag record -O subset topic 1 topic 2

→ This will log to a file named subset.bag the data published by topic 1 and topic 2

<u>Note</u>

⇒ rosbag is limited in its ability to exactly duplicate the behavior of a running system in terms of when messages are recorded and processed.

↳ Very sensitive to small changes in timing in the system.

(18) Getting Started wit roswtf tool

⇒ roswtf examines your System to try and find problems.

(19) Navigating the ROS wiki

- → ROS Wiki Landing Page
- → ROS Package pages
- → ROS Stack Pages

(20) Where Next?

ROS
- → (1) Robot middle-ware
- → (2) Standard mechanism for developers around the world to share their code.

{To main reason to use ROS}

1) Launching a Simulator
2) Exploring RViz
3) Understanding TF

⇒ The TF package transforms between different coordinate frames used by your robot and keeps track of these transforms over time.

⇒ Constructing a URDF model of your robot.

4) Going Deeper

→ actionlib

{This package provides a standardized interface for interacting with Preemptible task.}

→ Navigation

{2D navigation : map-building and path planning}

→ MoveIt

{To Control the arms of your robot}

Parameter Server

① getParam()

bool getParam (Const std::String& key
, Parameter_type& output_value) const

{Parameter name}

{Place to put retrieved data}

⇒ It returns bool, which provides the ability to check if retrieving the parameter succeeded or not.

② param()

NodeHandle.param<std::string> ("my_Param", S, "default")

③ Setting Parameters

n.SetParam ("my_Param", "hello there");

④ Deleting Parameters

n.deleteParam ("my_Param");

⑤ Searching for Parameters

N.SearchParam ("b", Param_name);

small

Ros Message

① Creating a msg

① PackageName
└→ msg
└→ messagename.msg

String first_name
String last_name
Uint8 age
Uint32 Score

{ Example }

② In package.xml file

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

③ In CMakeLists.txt file

⇒ Add message_generation in find package.

⇒ Add CATKIN_DEPENDS message_runtime in catkin_package.

⇒ Add messageName.msg in add_message_files

⇒ Add std_msgs in generate_messages.

Launch file

① Using roslaunch

↓

{It starts nodes as defined in a launch file}

roslaunch [Package] [filename.launch]

⇒ launch files are kept in launch directory inside the package.

② Launch file

→ It is a xml file.

```
<launch>

    <group ns="name space">
        <node pkg="Package name" name="node name given"
              type="Original node name"/>
    </group>

        or

    <node pkg="Package name" name="node name given"
          type="Original node name"/>

        ⋮

</launch>
```

ROS Tutorial {Intermediate Level}

① Roslaunch tips for large projects

⇒ Large applications on a robot typically involve several interconnected nodes, each of which have many parameters.

② Running ROS across multiple machines

⇒ ROS is designed with distributed computing in mind.

↳ Driver node that communicates with a piece of hardware must run on the machine to which the hardware is physically connected.