**IE 410 – INTRODUCTION TO ROBOTICS**

**Lab-4 report**

**ROS cpp programming**

**Team M410:**

201901011 – HIMANSHU DUDHATRA

201901024 – DHAVALSINH RAJ

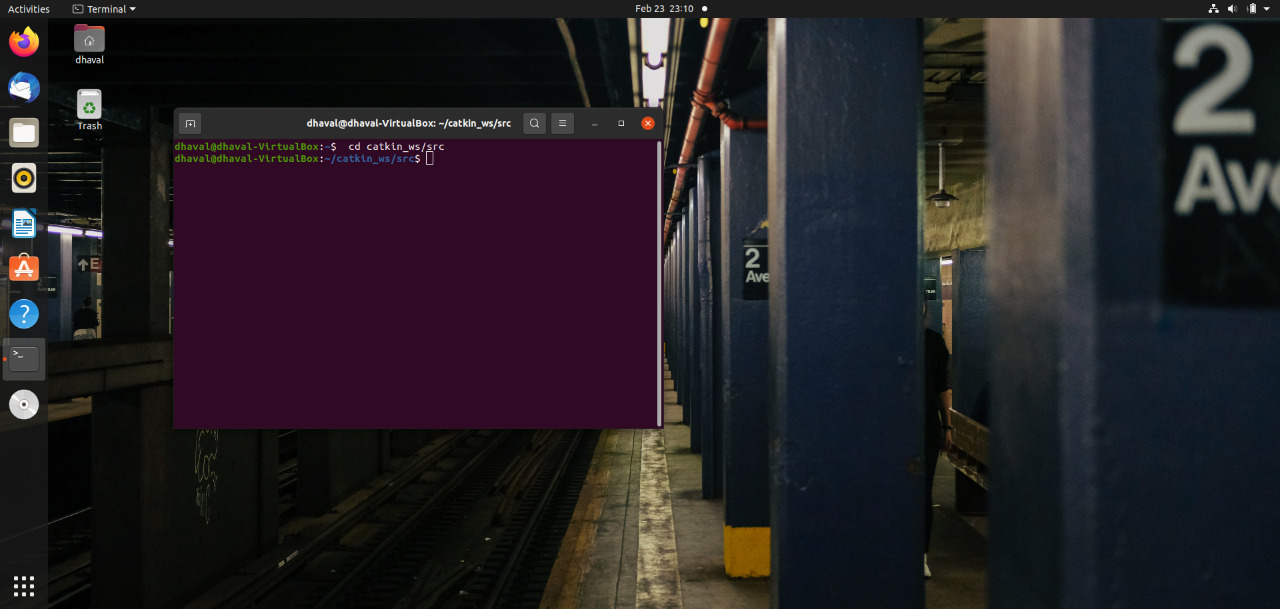
201901100 – SHUBHAM PATEL

201901145 – GARGEY PATEL

* **Creating catkin package**

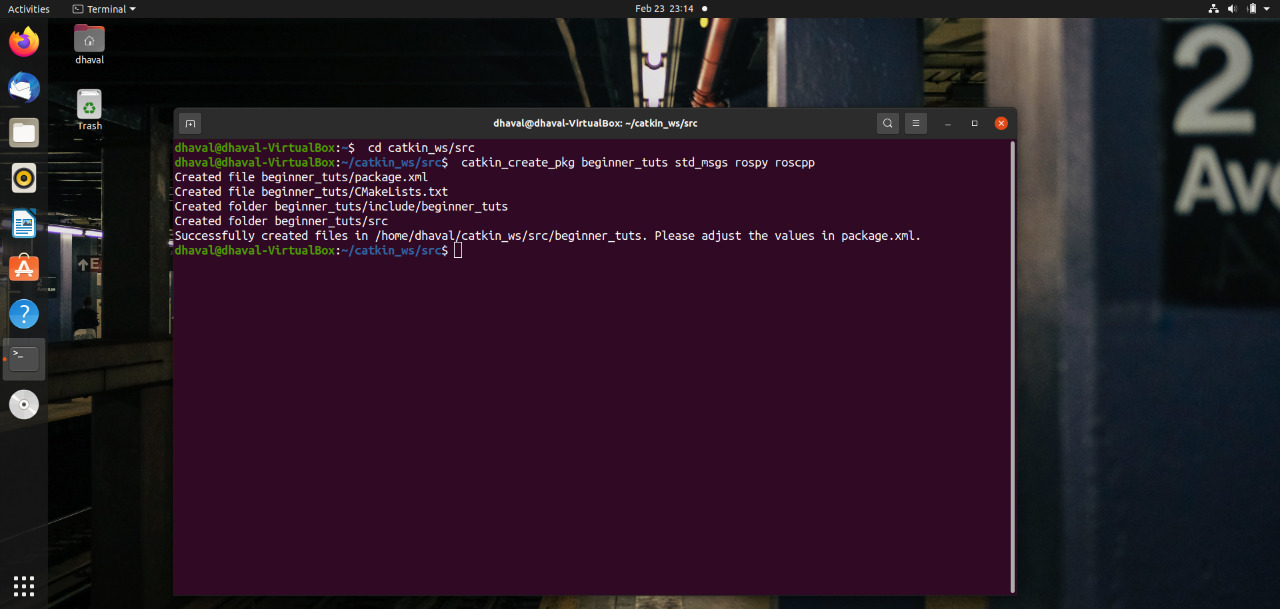
First change to the source space directory of the catkin workspace you created.

$ cd ~/catkin\_ws/src



Now use the catkin\_create\_pkg script to create a new package called 'beginner\_tuts' which depends on std\_msgs, roscpp, and rospy:

$ catkin\_create\_pkg beginner\_tuts std\_msgs rospy roscpp

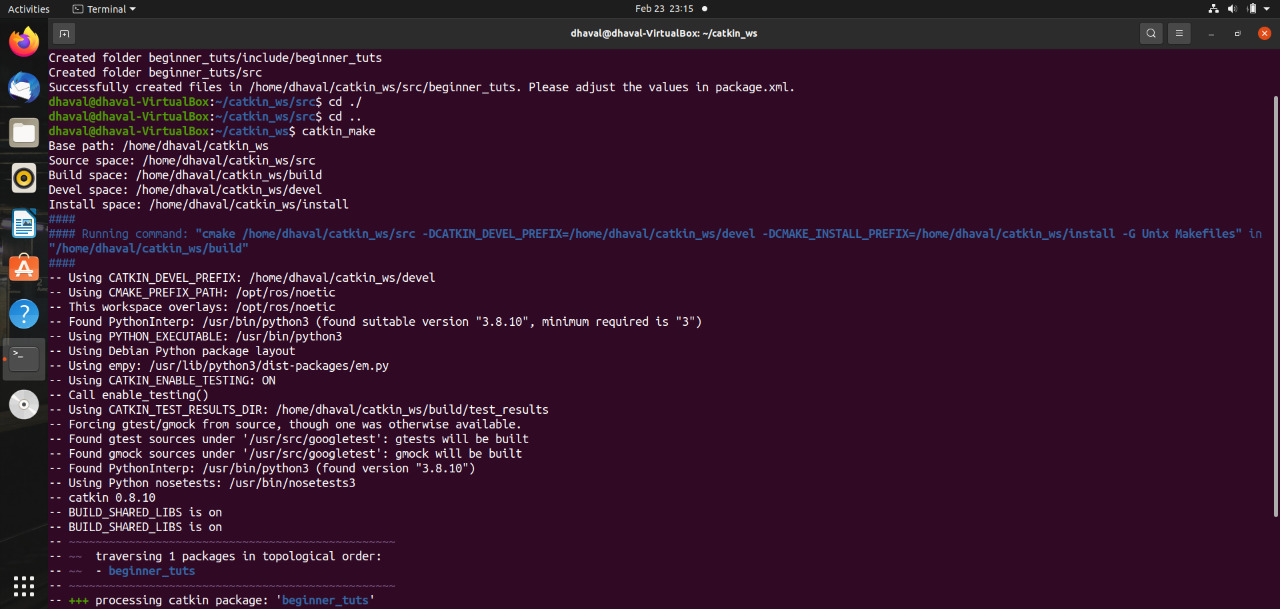


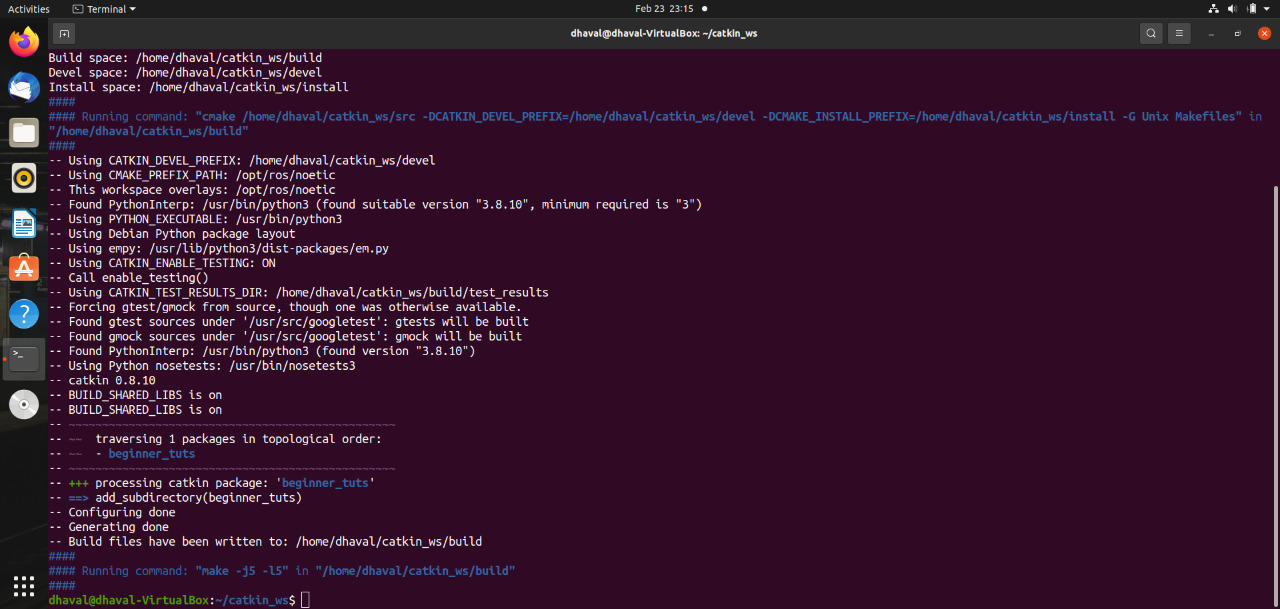
* **Building a catkin workspace and sourcing the setup file**

Now you need to build the packages in the catkin workspace:

$ cd ~/catkin\_ws

$ catkin\_make

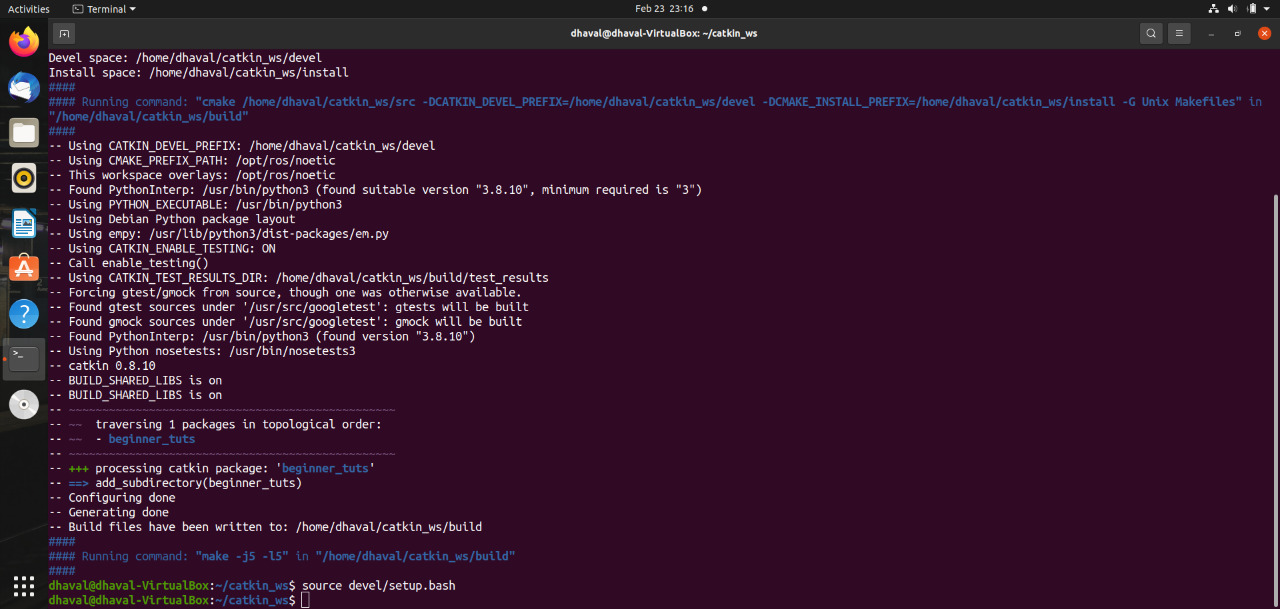




After the workspace has been built it has created a similar structure in the devel subfolder as you usually find under /opt/ros/$ROSDISTRO\_NAME.

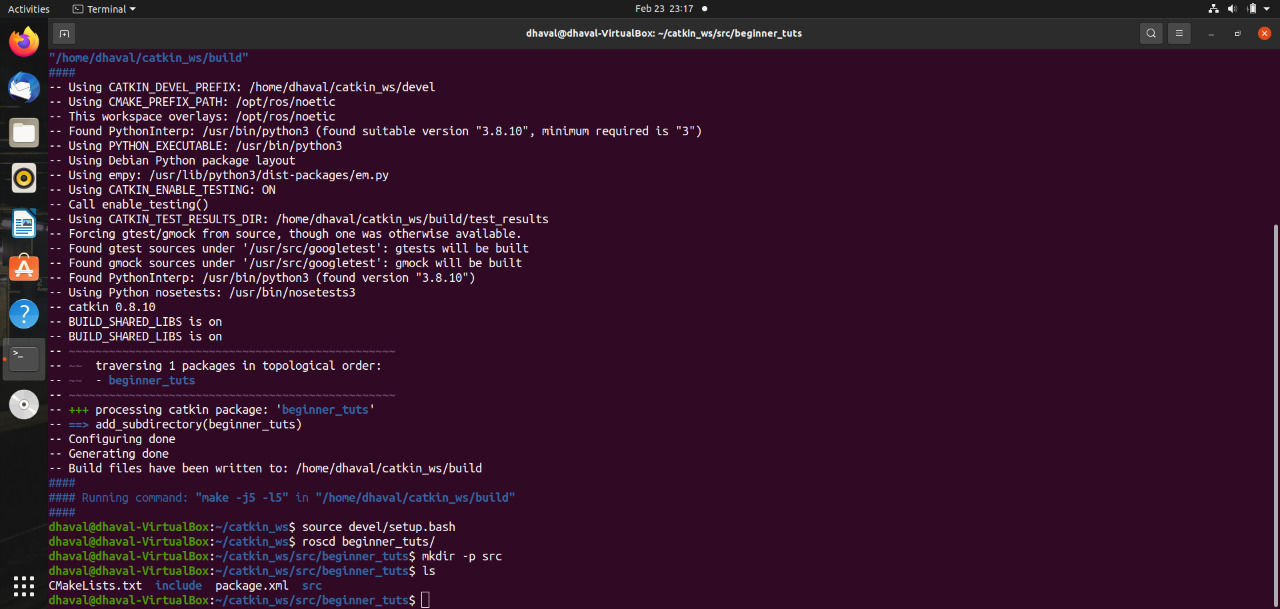
To add the workspace to your ROS environment you need to source the generated setup file:

$ . ~/catkin\_ws/devel/setup.bash

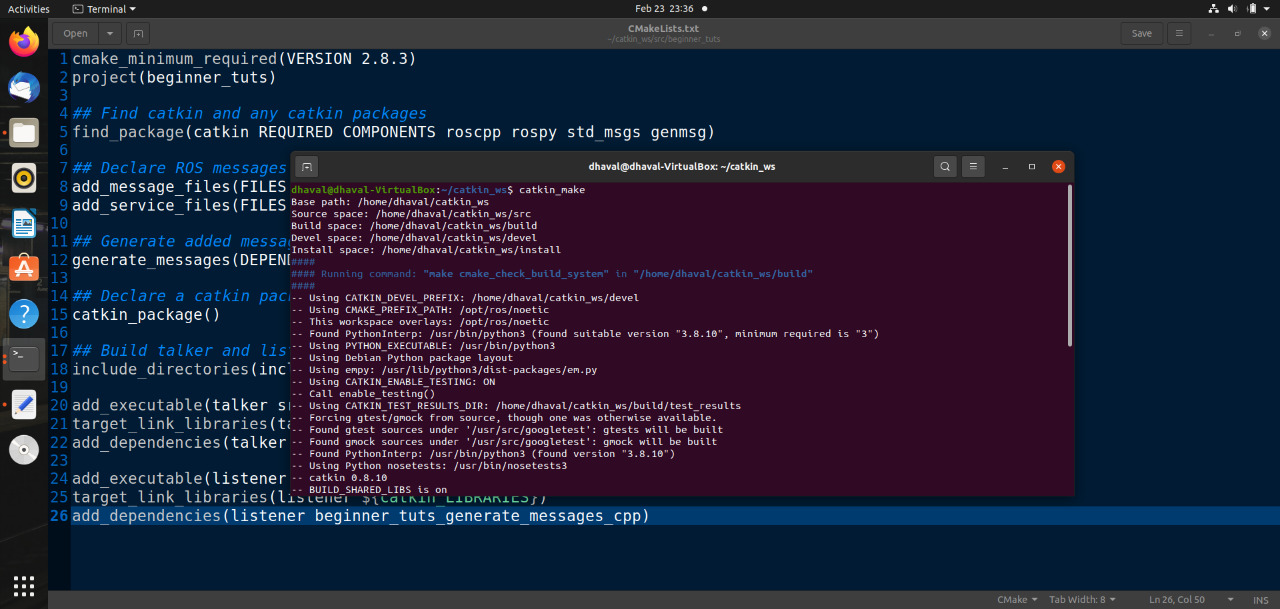


Now,

$ roscd beginner\_tuts/



$ catkin\_make



* **Talker.cpp**

// %Tag(FULLTEXT)%

// %Tag(ROS\_HEADER)%

#include "ros/ros.h"

// %EndTag(ROS\_HEADER)%

// %Tag(MSG\_HEADER)%

#include "std\_msgs/String.h"

// %EndTag(MSG\_HEADER)%

#include <sstream>

/\*\*

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

\*/

int main(int argc, char \*\*argv)

{

/\*\*

\* The ros::init() function needs to see argc and argv so that it can perform

\* any ROS arguments and name remapping that were provided at the command line.

\* For programmatic remappings you can use a different version of init() which takes

\* remappings directly, but for most command-line programs, passing argc and argv is

\* the easiest way to do it. The third argument to init() is the name of the node.

\*

\* You must call one of the versions of ros::init() before using any other

\* part of the ROS system.

\*/

// %Tag(INIT)%

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

// %EndTag(INIT)%

/\*\*

\* NodeHandle is the main access point to communications with the ROS system.

\* The first NodeHandle constructed will fully initialize this node, and the last

\* NodeHandle destructed will close down the node.

\*/

// %Tag(NODEHANDLE)%

ros::NodeHandle n;

// %EndTag(NODEHANDLE)%

/\*\*

\* 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 and who

\* is subscribing. After this advertise() call is made, the 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 messages on that topic through a call to publish(). Once

\* all copies of the returned Publisher object are destroyed, the topic

\* will be automatically unadvertised.

\*

\* The second parameter to advertise() is the size of the message queue

\* used for publishing messages. If messages are published more quickly

\* than we can send them, the number here specifies how many messages to

\* buffer up before throwing some away.

\*/

// %Tag(PUBLISHER)%

ros::Publisher chatter\_pub = n.advertise<std\_msgs::String>("chatter", 1000);

// %EndTag(PUBLISHER)%

// %Tag(LOOP\_RATE)%

ros::Rate loop\_rate(10);

// %EndTag(LOOP\_RATE)%

/\*\*

\* A count of how many messages we have sent. This is used to create

\* a unique string for each message.

\*/

// %Tag(ROS\_OK)%

int count = 0;

while (ros::ok())

{

// %EndTag(ROS\_OK)%

/\*\*

\* This is a message object. You stuff it with data, and then publish it.

\*/

// %Tag(FILL\_MESSAGE)%

std\_msgs::String msg;

std::stringstream ss;

ss << "hello world " << count;

msg.data = ss.str();

// %EndTag(FILL\_MESSAGE)%

// %Tag(ROSCONSOLE)%

ROS\_INFO("%s", msg.data.c\_str());

// %EndTag(ROSCONSOLE)%

/\*\*

\* The publish() function is how you send messages. The parameter

\* is the message object. The type of this object must agree with the type

\* given as a template parameter to the advertise<>() call, as was done

\* in the constructor above.

\*/

// %Tag(PUBLISH)%

chatter\_pub.publish(msg);

// %EndTag(PUBLISH)%

// %Tag(SPINONCE)%

ros::spinOnce();

// %EndTag(SPINONCE)%

// %Tag(RATE\_SLEEP)%

loop\_rate.sleep();

// %EndTag(RATE\_SLEEP)%

++count;

}

return 0;

}

// %EndTag(FULLTEXT)%

cout<<ans<<endl;

return 0;

* **Explanation:**
* **#include "ros/ros.h"** → ros/ros.h is a convenience include that includes all the headers necessary to use the most common public pieces of the ROS system
* **#include "std\_msgs/String.h"** → This includes the std\_msgs/String message, which resides in the std\_msgs package. This is a header generated automatically from the String.msg file in that package. For more information on message definitions, see the msg page.
* **ros::init(argc, argv, "talker");** → Initialize ROS. This allows ROS to do name remapping through the command line -- not important for now. This is also where we specify the name of our node. Node names must be unique in a running system.
* **ros::NodeHandle n;** → Create a handle to this process' node. 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("chatter", 1000);** → Tell the master that we are going to be publishing a message of type std\_msgs/String on the topic chatter.
* **ros::Rate loop\_rate(10);** → A ros::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.
* **ROS\_INFO("%s", msg.data.c\_str());** → ROS\_INFO and friends are our replacement for printf/cout. See the rosconsole documentation for more information.
* **ros::spinOnce();** → Calling ros::spinOnce() here is not necessary for this simple program, because we are not receiving any callbacks. • loop\_rate.sleep(); → Now we use the ros::Rate object to sleep for the time remaining to let us hit our 10Hz publish rate.
* **Listner.cpp**

/\*

\* Copyright (C) 2008, Morgan Quigley and Willow Garage, Inc.

\*

\* Redistribution and use in source and binary forms, with or without

\* modification, are permitted provided that the following conditions are met:

\* \* Redistributions of source code must retain the above copyright notice,

\* this list of conditions and the following disclaimer.

\* \* Redistributions in binary form must reproduce the above copyright

\* notice, this list of conditions and the following disclaimer in the

\* documentation and/or other materials provided with the distribution.

\* \* Neither the names of Stanford University or Willow Garage, Inc. nor the

names of its

\* contributors may be used to endorse or promote products derived from

\* this software without specific prior written permission.

\*

\* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"

\* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE

\* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE

\* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE

\* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR

\* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF

\* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS

\* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN

\* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)

\* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE

\* POSSIBILITY OF SUCH DAMAGE.

\*/

// %Tag(FULLTEXT)%

#include "ros/ros.h"

#include "std\_msgs/String.h"

/\*\*

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

\*/

// %Tag(CALLBACK)%

void chatterCallback(const std\_msgs::String::ConstPtr& msg)

{

ROS\_INFO("I heard: [%s]", msg->data.c\_str());

}

// %EndTag(CALLBACK)%

int main(int argc, char \*\*argv)

{

/\*\*

\* The ros::init() function needs to see argc and argv so that it can

perform

\* any ROS arguments and name remapping that were provided at the command

line.

\* For programmatic remappings you can use a different version of init()

which takes

\* remappings directly, but for most command-line programs, passing argc and

argv is

\* the easiest way to do it. The third argument to init() is the name of

the node.

\*

\* You must call one of the versions of ros::init() before using any other

\* part of the ROS system.

\*/

ros::init(argc, argv, "listener");

/\*\*

\* NodeHandle is the main access point to communications with the ROS

system.

\* The first NodeHandle constructed will fully initialize this node, and the

last

\* NodeHandle destructed will close down the node.

\*/

ros::NodeHandle n;

/\*\*

\* The subscribe() call is how you tell ROS that you want to receive

messages

\* on a given topic. This invokes a call to the ROS

\* master node, which keeps a registry of who is publishing and who

\* is subscribing. Messages are passed to a callback function, here

\* called chatterCallback. subscribe() returns a Subscriber object that you

\* must hold on to until you want to unsubscribe. When all copies of the

Subscriber

\* object go out of scope, this callback will automatically be unsubscribed

from

\* this topic.

\*

\* The second parameter to the subscribe() function is the size of the

message

\* queue. If messages are arriving faster than they are being processed,

this

\* is the number of messages that will be buffered up before beginning to

throw

\* away the oldest ones.

\*/

// %Tag(SUBSCRIBER)%

ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

// %EndTag(SUBSCRIBER)%

/\*\*

\* ros::spin() will enter a loop, pumping callbacks. With this version, all

\* callbacks will be called from within this thread (the main

one). ros::spin()

\* will exit when Ctrl-C is pressed, or the node is shutdown by the master.

\*/

// %Tag(SPIN)%

ros::spin();

// %EndTag(SPIN)%

return 0;

}

// %EndTag(FULLTEXT)%

* **Explanation:**
* **ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);** → Subscribe to the chatter topic with the master. ROS will call the chatterCallback() function whenever a new message arrives. The 2nd argument is the queue size, in case we are not able to process messages fast enough. In this case, if the queue reaches 1000 messages, we will start throwing away old messages as new ones arrive.
* **ros::spin();** → ros::spin() enters a loop, calling message callbacks as fast as possible. Don't worry though, if there's nothing for it to do it won't use much CPU. ros::spin() will exit once ros::ok() returns false, which means ros::shutdown() has been called, either by the default Ctrl-C handler, the master telling us to shutdown, or it being called manually.
* **Conclusion:**
* By performing this lab, we learned how to create and build a catkin package, how to write code for Publisher and Subscriber nodes in C++, Test Publisher and Subscriber nodes.
* We can do robot programming in traditional programming languages C++ or python.
* Ros node structure has 2 types of nodes: 1) master node 2) servant node and 2 types of servent node: publisher and subscriber.
* ros\_essential\_cpp contains definitions for talker and listener cpp files and also has an important package called std\_msg.
* Using C++ or python we can configure robot control without using inbuilt ros commands