**Creating nodes**

You can run this tutorial on:

* [ROSbot 2.0](https://store.husarion.com/products/rosbot)
* [ROSbot 2.0 PRO](https://store.husarion.com/collections/dev-kits/products/rosbot-pro)
* [ROSbot 2.0 simulation model (Gazebo)](https://github.com/husarion/rosbot_description)

**Workspace setup**

To begin developing your own nodes, you need to do some workspace configuration first. Workspace is the place where all your source files, libraries and compiled nodes will be stored.

First you need to create a folder, where your workspace will be located. You can do this by typing in:

mkdir -p ~/ros\_workspace/src

**Copy**

This will create folder named ros\_workspace and folder src inside it. All of the source files for your nodes will be stored in folder src.

Then you can initialize your workspace with command catkin\_init\_workspace executed in src folder:

cd ~/ros\_workspace/src

catkin\_init\_workspace

**Copy**

Now you can move to your workspace main directory:

cd ~/ros\_workspace

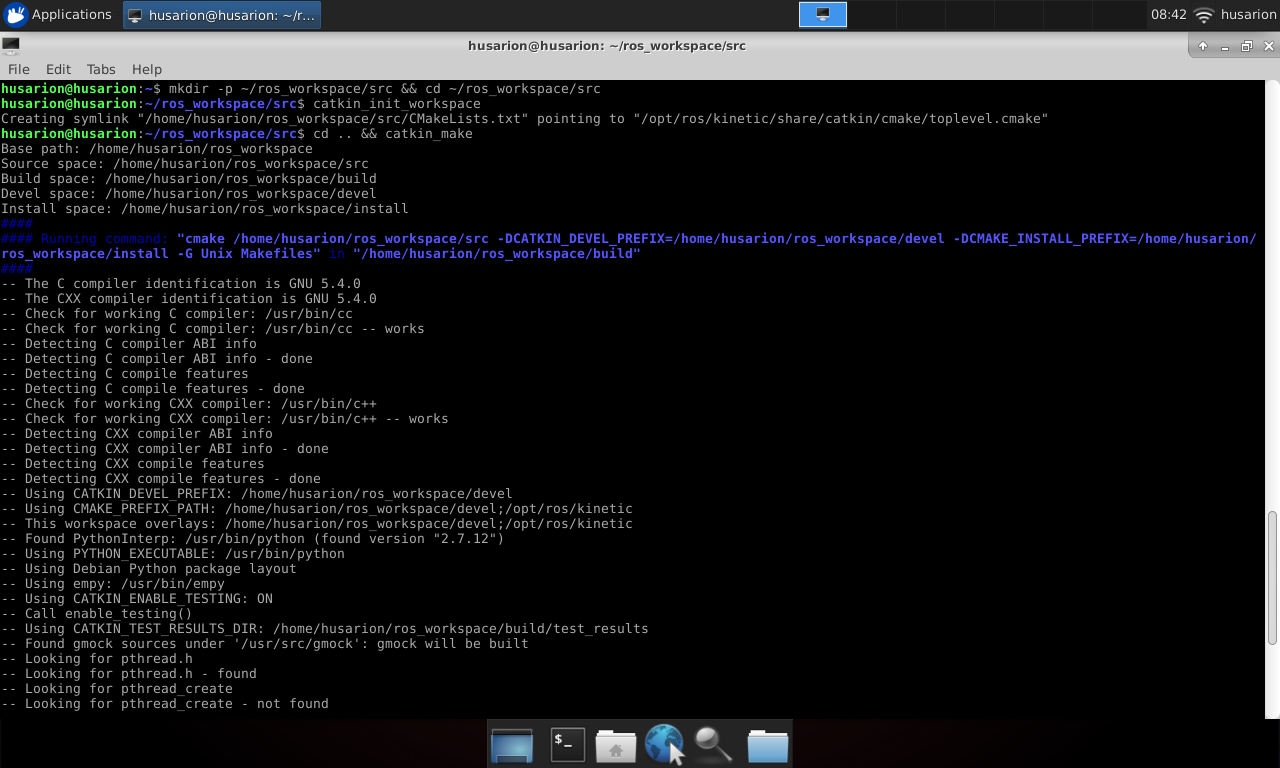
**Copy**

and compile it:

catkin\_make

**Copy**

After this command you should get output like this:



And it should end with:

####

#### Running command: "make -j4 -l4" **in** "/home/pi/ros\_workspace/build"

####

**Copy**

After this operation you should have two new folders in your workspace: build for storing files that are used during compilation and devel for storing output files.

Now your workspace is set up and ready for creating new nodes.

**Creating new package**

As you should already know, in ROS, nodes are distributed in packages, so in order to create a node you need to create a package. Packages are created with command catkin\_create\_pkg and it must be executed in srcfolder in your workspace.

Syntax of catkin\_create\_pkg is:

catkin\_create\_pkg package\_name [required packages]

**Copy**

where package\_name is desired package name and argument required packages is optional and contain names of packages that are used by newly created packages.

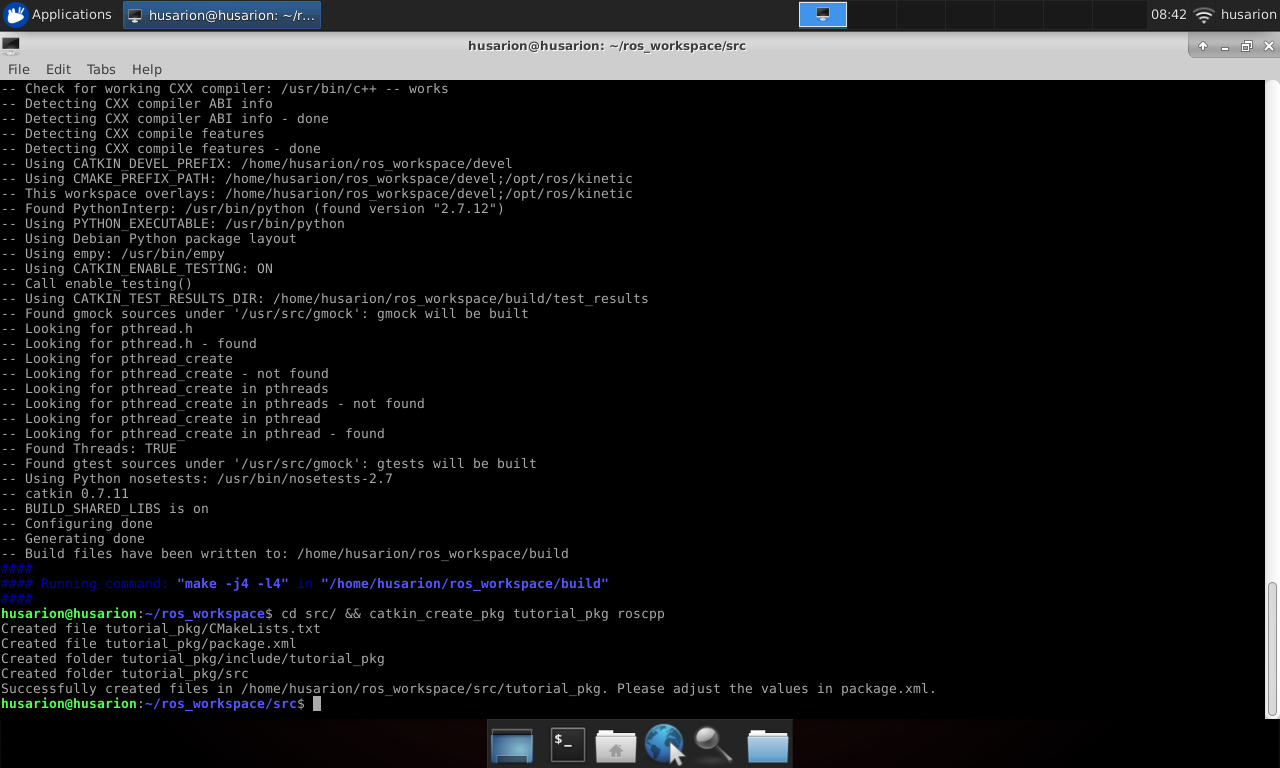
For our tutorial we will create package named tutorial\_pkg which depends on package roscpp. Package roscpp is a basic ROS library for C++.

cd ~/ros\_workspace/src

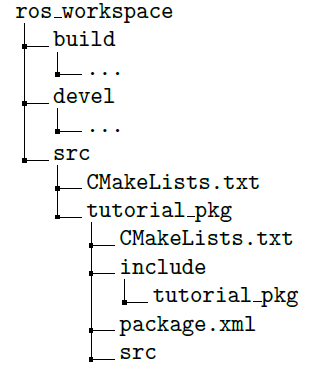
catkin\_create\_pkg tutorial\_pkg roscpp

**Copy**

After typing in this command you should get output like this:



This will create folder named tutorial\_pkg and some files in it. Your workspace file structure should now look like like below:



Created files are:

* CMakeLists.txt - these are build instructions for your nodes, you need to edit this file if you want to compile any node, we will do it later.
* package.xml - this file contains package metadata like author, description, version or required packages. Package can be built without changing it, but you should adjust this file if you want to publish your package to others.

**Write code for your first node**

Let’s create C++ file for your node, name it tutorial\_pkg\_node.cpp and place it in src folder under tutorial\_pkg:

touch ~/ros\_workspace/src/tutorial\_pkg/src/tutorial\_pkg\_node.cpp

**Copy**

Open file in your favourite text editor and paste:

#**include** <ros/ros.h>

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

{

ros::init(argc, argv, "example\_node");

ros::NodeHandle **n**("~");

ros::Rate **loop\_rate**(50);

**while** (ros::ok())

{

ros::spinOnce();

loop\_rate.sleep();

}

}

**Copy**

Code explanation line by line:

#**include** <ros/ros.h>

**Copy**

Add header files for basic ROS libraries.

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

**Copy**

Beginning of node main function.

ros::init(argc, argv, "example\_node");

**Copy**

Initialization of ROS node, this function contacts with ROS master and registers node in the system.

ros::NodeHandle **n**("~");

**Copy**

Get the handle for node, this handle is required for interactions with system e.g. subscribing to topic.

ros::Rate **loop\_rate**(50);

**Copy**

Define rate for repeatable operations.

**while** (ros::ok()) {

**Copy**

Check if ROS is working. E.g. if ROS master is stopped or there was sent signal to stop the system, ros::ok()will return false.

ros::spinOnce();

**Copy**

Process all incoming messages.

loop\_rate.sleep();

**Copy**

Wait until defined time passes.

You can save the C++ file.

**Building your node**

Before you build the node, you need to edit CMakeLists.txt from tutorial\_pkg directory. Open it in your favourite text editor.

Find line:

# add\_compile\_options(-std=c++11)

**Copy**

and uncomment it (remove # sign). This will allow to use C++11 standard of C++.

You should also find and uncomment line:

# add\_executable(${PROJECT\_NAME}\_node src/tutorial\_pkg\_node.cpp)

**Copy**

This will let the compiler know that it should create executable file from defined source. Created executable will be your node. Variable PROJECT\_NAME is defined by line project(tutorial\_pkg). This results in tutorial\_pkg\_node as the name of the executable. You can adjust it to your needs.

After that find and uncomment lines:

# target\_link\_libraries(${PROJECT\_NAME}\_node

# ${catkin\_LIBRARIES}

# )

**Copy**

This will cause compiler to link libraries required by your node. Save the changes and close editor.

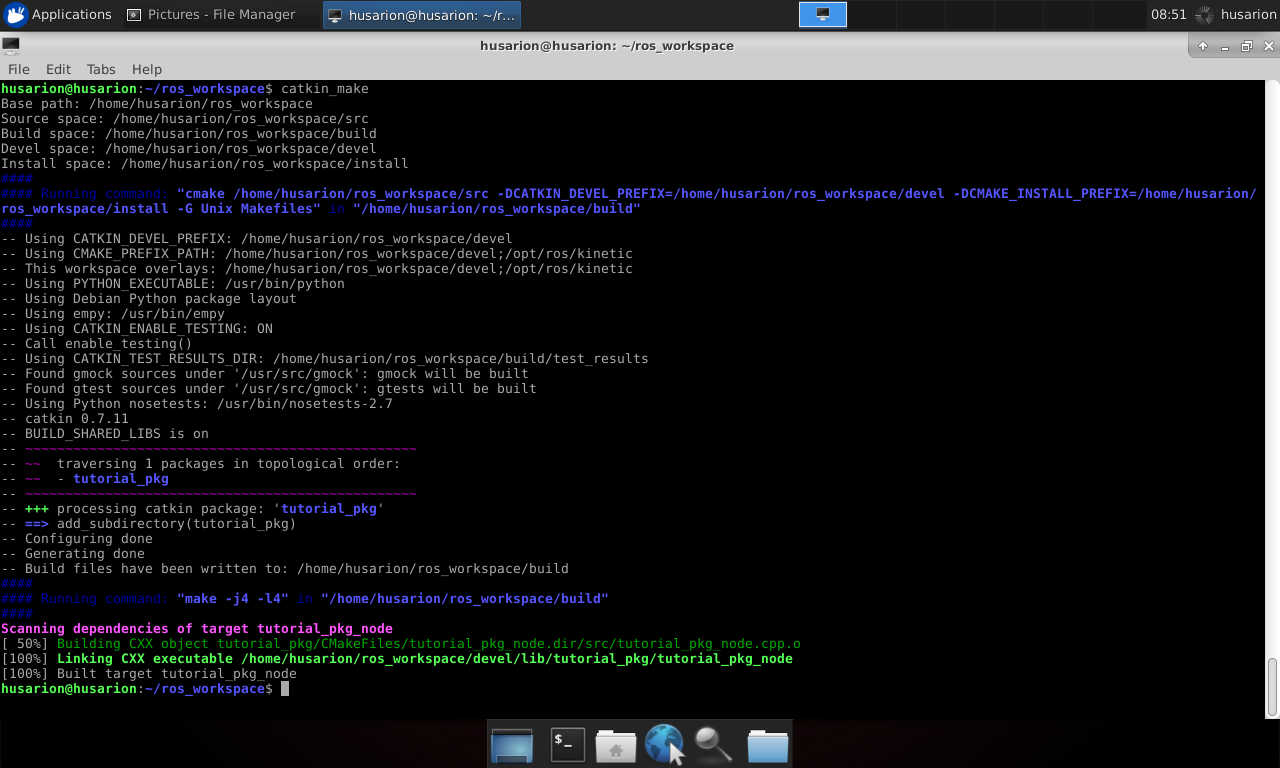
Open terminal, move to workspace main directory and build your project with command catkin\_make:

cd ~/ros\_workspace

catkin\_make

**Copy**

You should get output like this:



**Running your node**

Your node is built and ready for running, but before you run it, you need to load some environment variables:

source ~/ros\_workspace/devel/setup.sh

**Copy**

These environment variables allow you to run node regardless of directory you are working in. You have to load it every time you open new terminal or you can add line:

. ~/ros\_workspace/devel/setup.sh

**Copy**

to your .bashrc file.

To run your node you can use command line or .launch file as with any other node. Remember that package is tutorial\_pkg and node is tutorial\_pkg\_node.

**Task 1** Run your node with command line or .launch file. Then use rosnode and rqt\_graph tools to examine system and check if your node is visible in the system.

To remind, you can start ROS by typing in the name of the node, you can do this with the following command:

rosrun package\_name node\_type [options]

**Copy**

For the node you just created it will be:

rosrun tutorial\_pkg tutorial\_pkg\_node

**Copy**

If you want to use .launch files associated with your custom package you will have to create launchdirectory:

mkdir ~/ros\_workspace/src/tutorial\_pkg/launch

**Copy**

Place your .launch files there. This way you can start them by typing:

roslaunch tutorial\_pkg your\_launch\_file.launch

**Copy**

Example launch file for tutorial\_pkg\_node will be as follows:

<**launch**>

<**node** pkg="tutorial\_pkg" type="tutorial\_pkg\_node" name="tutorial\_pkg\_node" output="screen">

</**node**>

</**launch**>

**Copy**

Save it as tutorial\_pkg\_node.launch in ~/ros\_workspace/src/tutorial\_pkg/launch directory and launch it:

roslaunch tutorial\_pkg tutorial\_pkg\_node.launch

**Copy**

**Subscribing to topic**

You will modify your node to subscribe to topic /camera/rgb/image\_raw and calculate average brightness of image.

To process message received from the camera you need a header file with message type definition. You can include it with:

#**include** <sensor\_msgs/Image.h>

**Copy**

Image message is an object consisting of following fields:

* std\_msgs/Header header - header with message metedata
* uint32 height - image height in pixels
* uint32 width - image width in pixels
* string encoding - pixel encoding definition
* uint8 is\_bigendian - is data expressed in bigendian manner
* uint32 step - length of data for one row
* std::vector<uint8\_t> data - actual image data

Then you need a function for processing received message:

**void** **imageCallback**(**const** sensor\_msgs::ImageConstPtr &image)

{

**long** **long** sum = 0;

**for** (**int** value : image->data)

{

sum += value;

}

**int** avg = sum / image->data.size();

std::cout << "Brightness: " << avg << std::endl;

}

**Copy**

Code explanation line by line:

**void** **imageCallback**(**const** sensor\_msgs::ImageConstPtr &image)

**Copy**

Function definition, argument is pointer to incoming message.

**long** **long** sum = 0;

**Copy**

Variable for storing sum of all pixel values.

**for**( **int** value : image->data )

**Copy**

Iteration through every pixel and colour.

sum+=value;

**Copy**

Add current pixel value to sum.

**int** avg = sum/image->data.size();

**Copy**

Calculate average value.

std::cout << "Brightness: " << avg << std::endl;

**Copy**

Print brightness value to screen.

Last thing to do is defining topic to subscribe:

ros::Subscriber sub = n.subscribe("/camera/rgb/image\_raw", 10, imageCallback);

**Copy**

Here we use method subscribe of NodeHandle object. Arguments of method are:

* /camera/rgb/image\_raw - name of topic to subscribe.
* 10 - message queue size. Messages are processed in order they come in. In the case that node receives, in short time, more messages than this value, excessive messages will be dropped.
* imageCallback - function to process incoming messages.

Your final code should look like this:

#**include** <ros/ros.h>

#**include** <sensor\_msgs/Image.h>

**void** **imageCallback**(**const** sensor\_msgs::ImageConstPtr &image)

{

**long** **long** sum = 0;

**for** (**int** value : image->data)

{

sum += value;

}

**int** avg = sum / image->data.size();

std::cout << "Brightness: " << avg << std::endl;

}

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

{

ros::init(argc, argv, "example\_node");

ros::NodeHandle **n**("~");

ros::Subscriber sub = n.subscribe("/camera/rgb/image\_raw", 10, imageCallback);

ros::Rate **loop\_rate**(50);

**while** (ros::ok())

{

ros::spinOnce();

loop\_rate.sleep();

}

}

**Copy**

**Task 2** Build your node and run it along with astra.launch. Use rosnode, rostopic and rqt\_graph tools to examine system and check how data is passed between nodes.

When using launch files we can make use of prevoiously created files, this way we can make configuration easier and more readable.

Instead of configuring tutorial\_pkg\_node again, we will include file created in prevoius step:

<**launch**>

<**arg** name="use\_gazebo" default="false"/>

<**include** unless="$(arg use\_gazebo)" file="$(find astra\_launch)/launch/astra.launch"/>

<**include** if="$(arg use\_gazebo)" file="$(find rosbot\_description)/launch/rosbot.launch"/>

<**include** file="$(find tutorial\_pkg)/launch/tutorial\_pkg\_node.launch"/>

</**launch**>

**Copy**

Save above file as tutorial\_2.launch in ~/ros\_workspace/src/tutorial\_pkg/launch directory and launch it:

roslaunch tutorial\_pkg tutorial\_2.launch

**Copy**

or if you are using **Gazebo** simulator:

roslaunch tutorial\_pkg tutorial\_2.launch use\_gazebo:=true

**Copy**

**Receiving parameters**

Your node can receive parameters, they are used to customize behaviour of node e.g. subscribed topic name, device name or transmission speed for serial port.

You will modify a node to receive boolean parameter which defines if node should print image brightness to screen.

To receive the parameter you need a variable to store its value, in this example variable should have a global scope:

**bool** print\_b;

**Copy**

Then receive parameter value:

n.param<**bool**>("print\_brightness", print\_b, false);

**Copy**

Here we use method param of NodeHandle object. Arguments of method are:

* print\_brightness - name of parameter to receive.
* print\_b - variable to store parameter value.
* false - parameter default value.

Last thing is to print brightness dependant on parameter value:

**if** (print\_b)

{

std::cout << "Brightness: " << avg << std::endl;

}

**Copy**

Your final code should look like this:

#**include** <ros/ros.h>

#**include** <sensor\_msgs/Image.h>

**bool** print\_b;

**void** **imageCallback**(**const** sensor\_msgs::ImageConstPtr &image)

{

**long** **long** sum = 0;

**for** (**int** value : image->data)

{

sum += value;

}

**int** avg = sum / image->data.size();

**if** (print\_b)

{

std::cout << "Brightness: " << avg << std::endl;

}

}

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

{

ros::init(argc, argv, "example\_node");

ros::NodeHandle **n**("~");

ros::Subscriber sub = n.subscribe("/camera/rgb/image\_raw", 10, imageCallback);

n.param<**bool**>("print\_brightness", print\_b, false);

ros::Rate **loop\_rate**(50);

**while** (ros::ok())

{

ros::spinOnce();

loop\_rate.sleep();

}

}

**Copy**

**Task 3** Run your node with parameter print\_brightness set to true and again set to false. Observe how behaviour of node changes.

To add parameter for node, you will need to add <param> tag inside <node> tag in tutorial\_pkg\_node.launch file.

**Publishing to topic**

You will modify node to publish brightness value to a new topic with message of type std\_msgs/UInt8. Message std\_msgs/UInt8 is object with only one field data, which contain actual integer data.

Begin with including message header file:

#**include** <std\_msgs/UInt8.h>

**Copy**

Next define publisher object with global scope:

ros::Publisher brightness\_pub;

**Copy**

Then register in the system to publish to a specific topic:

brightness\_pub = n.advertise<std\_msgs::UInt8>("brightness" , 1);

**Copy**

Here we use method advertise of NodeHandle object. Arguments of method are:

* brightness - topic name.
* 1 - message queue size.

You also need to declare type of message which will be published, in this case it is std\_msgs::UInt8.

Last thing is to put some data into message and send it to topic with some frequency:

std\_msgs::UInt8 brightness\_value;

brightness\_value.data=avg;

brightness\_pub.publish(brightness\_value);

**Copy**

In our example it can be done while processing each message incoming from camera topic.

Your final code should look like this:

#**include** <ros/ros.h>

#**include** <sensor\_msgs/Image.h>

#**include** <std\_msgs/UInt8.h>

**bool** print\_b;

ros::Publisher brightness\_pub;

**void** **imageCallback**(**const** sensor\_msgs::ImageConstPtr &image)

{

**long** **long** sum = 0;

**for** (**int** value : image->data)

{

sum += value;

}

**int** avg = sum / image->data.size();

**if** (print\_b)

{

std::cout << "Brightness: " << avg << std::endl;

}

std\_msgs::UInt8 brightness\_value;

brightness\_value.data = avg;

brightness\_pub.publish(brightness\_value);

}

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

{

ros::init(argc, argv, "example\_node");

ros::NodeHandle **n**("~");

ros::Subscriber sub = n.subscribe("/camera/rgb/image\_raw", 10, imageCallback);

n.param<**bool**>("print\_brightness", print\_b, false);

brightness\_pub = n.advertise<std\_msgs::UInt8>("brightness", 1);

ros::Rate **loop\_rate**(50);

**while** (ros::ok())

{

ros::spinOnce();

loop\_rate.sleep();

}

}

**Copy**

**Task 4** Compile your node and run it along with astra.launch. Use rosnode, rostopic and rqt\_graph tools to examine the system, then use rostopic echo tool to read brightness of the image from the camera.

**Calling the service**

You will modify node to call to a service with message type std\_srvs/Empty, this type has no field and can not carry any data, it can be used only for invoking action in another node and getting reply when its done.

As a service provider we will use image\_saver node from image\_view package. Image\_saver have one service named save. every time it is called, one frame from subscribed image topic is saved to hard drive.

Desired node behaviour is to count incoming frames and call service once per given number of frames.

Begin with importing required header files:

#**include** <std\_srvs/Empty.h>

**Copy**

We need one variable for counting passed frames:

**int** frames\_passed = 0;

**Copy**

In imageCallback function increment counter with every incoming message:

frames\_passed++;

**Copy**

Create a client which will be caling to service:

ros::ServiceClient client = n.serviceClient<std\_srvs::Empty>("/image\_saver/save");

**Copy**

Here we use method serviceClient of NodeHandle object. Method has only one argument, it is the name of service. You also need to determine message type for service: std\_srvs::Empty.

Instantiate message object:

std\_srvs::Empty srv;

**Copy**

Check if required number of frames passed and reset counter:

**if** (frames\_passed > 100)

{

frames\_passed = 0;

**Copy**

Call the service:

client.call(srv);

**Copy**

Your final code should look like this:

#**include** <ros/ros.h>

#**include** <sensor\_msgs/Image.h>

#**include** <std\_msgs/UInt8.h>

#**include** <std\_srvs/Empty.h>

**bool** print\_b;

ros::Publisher brightness\_pub;

**int** frames\_passed = 0;

**void** **imageCallback**(**const** sensor\_msgs::ImageConstPtr &image)

{

**long** **long** sum = 0;

**for** (**int** value : image->data)

{

sum += value;

}

**int** avg = sum / image->data.size();

**if** (print\_b)

{

std::cout << "Brightness: " << avg << std::endl;

}

std\_msgs::UInt8 brightness\_value;

brightness\_value.data = avg;

brightness\_pub.publish(brightness\_value);

frames\_passed++;

}

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

{

ros::init(argc, argv, "example\_node");

ros::NodeHandle **n**("~");

ros::Subscriber sub = n.subscribe("/camera/rgb/image\_raw", 10, imageCallback);

n.param<**bool**>("print\_brightness", print\_b, false);

brightness\_pub = n.advertise<std\_msgs::UInt8>("brightness", 1);

ros::ServiceClient client = n.serviceClient<std\_srvs::Empty>("/image\_saver/save");

std\_srvs::Empty srv;

ros::Rate **loop\_rate**(50);

**while** (ros::ok())

{

ros::spinOnce();

**if** (frames\_passed > 100)

{

frames\_passed = 0;

client.call(srv);

}

loop\_rate.sleep();

}

}

**Copy**

**Task 5** Build your node and run it with astra.launch and image\_saver. Use rosnode, rostopic and rqt\_graph tools to examine the system and check how data is passed between nodes. Let the nodes work for a certain time. Observe as new frames are being saved to your workspace directory.

For image\_saver node you can create separate launch file:

<**launch**>

<**node** pkg="image\_view" type="image\_saver" name="image\_saver">

<**param** name="save\_all\_image" value="false" />

<**param** name="filename\_format" value="$(env HOME)/ros\_workspace/image%04d.%s"/>

<**remap** from="/image" to="/camera/rgb/image\_raw"/>

</**node**>

</**launch**>

**Copy**

Save it as image\_saver.launch in ~/ros\_workspace/src/tutorial\_pkg/launch directory and include it in tutorial\_2.launch:

<**launch**>

<**arg** name="use\_gazebo" default="false"/>

<**include** unless="$(arg use\_gazebo)" file="$(find astra\_launch)/launch/astra.launch"/>

<**include** if="$(arg use\_gazebo)" file="$(find rosbot\_description)/launch/rosbot.launch"/>

<**include** file="$(find tutorial\_pkg)/launch/tutorial\_pkg\_node.launch"/>

<**include** file="$(find tutorial\_pkg)/launch/image\_saver.launch"/>

</**launch**>

**Copy**

To delete image files created by this example run following command in your ros\_workspace directory:

rm $(find image\*)

**Copy**

**Providing a service**

You will modify node to provide a service, which returns information regarding how many images were saved. This service will have a message type std\_srvs/Trigger, it has no field for request and two fields for response: integer to indicate if service was triggered successfully or not and string for short summary of executed action.

Start with including required header files:

#**include** <std\_srvs/Trigger.h>

**Copy**

Add variable for storing number of saved images:

**int** saved\_imgs = 0;

**Copy**

Next, you need a function to execute when service is called:

**bool** **saved\_img**(std\_srvs::Trigger::Request &req, std\_srvs::Trigger::Response &res)

{

res.success = 1;

std::string **str**("Saved images: ");

std::string num = std::to\_string(saved\_imgs);

str.append(num);

res.message = str;

**return** true;

}

**Copy**

Arguments for this function are pointers to request and response data. All services are called the same way, even if it does not carry any data, in that case these are pointer of void type.

Prepare string with response description:

std::string **str**("Saved images: ");

std::string num = std::to\_string(saved\_imgs);

str.append(num);

**Copy**

Fill string field with data:

res.message= str;

**Copy**

Fill integer field with data, this mean service was executed properly:

res.success=1;

**Copy**

Finish function, response will be sent to requesting node:

**return** true;

**Copy**

next thing to do is to increment image counter after saving frame:

saved\_imgs++;

**Copy**

Last thing to do is to register provided service in the system:

ros::ServiceServer service = n.advertiseService("saved\_images", saved\_img);

**Copy**

Here we use method advertiseService of NodeHandle object. Arguments of method are:

* saved\_images - service name.
* saved\_img - method to execute.

Your final code should look like this:

#**include** <ros/ros.h>

#**include** <sensor\_msgs/Image.h>

#**include** <std\_msgs/UInt8.h>

#**include** <std\_srvs/Empty.h>

#**include** <std\_srvs/Trigger.h>

**bool** print\_b;

ros::Publisher brightness\_pub;

**int** frames\_passed = 0;

**int** saved\_imgs = 0;

**void** **imageCallback**(**const** sensor\_msgs::ImageConstPtr &image)

{

**long** **long** sum = 0;

**for** (**int** value : image->data)

{

sum += value;

}

**int** avg = sum / image->data.size();

**if** (print\_b)

{

std::cout << "Brightness: " << avg << std::endl;

}

std\_msgs::UInt8 brightness\_value;

brightness\_value.data = avg;

brightness\_pub.publish(brightness\_value);

frames\_passed++;

}

**bool** **saved\_img**(std\_srvs::Trigger::Request &req, std\_srvs::Trigger::Response &res)

{

res.success = 1;

std::string **str**("Saved images: ");

std::string num = std::to\_string(saved\_imgs);

str.append(num);

res.message = str;

**return** true;

}

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

{

ros::init(argc, argv, "example\_node");

ros::NodeHandle **n**("~");

ros::Subscriber sub = n.subscribe("/camera/rgb/image\_raw", 10, imageCallback);

n.param<**bool**>("print\_brightness", print\_b, false);

brightness\_pub = n.advertise<std\_msgs::UInt8>("brightness", 1);

ros::ServiceClient client = n.serviceClient<std\_srvs::Empty>("/image\_saver/save");

std\_srvs::Empty srv;

ros::ServiceServer service = n.advertiseService("saved\_images", saved\_img);

ros::Rate **loop\_rate**(50);

**while** (ros::ok())

{

ros::spinOnce();

**if** (frames\_passed > 100)

{

frames\_passed = 0;

client.call(srv);

saved\_imgs++;

}

loop\_rate.sleep();

}

}

**Copy**

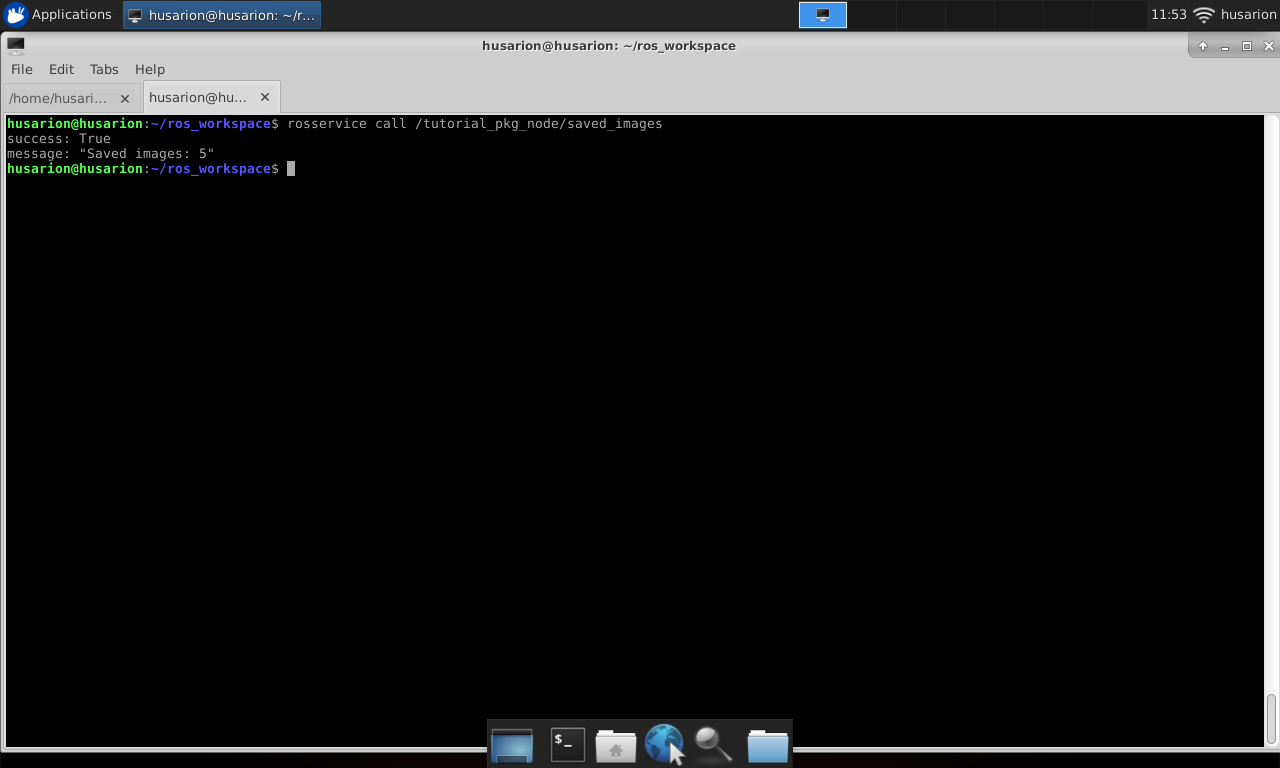
**Task 6** Build your node and run it as in previous task. Use rosnode, rostopic and rqt\_graph tools to examine the system.

Use rosservice call tool to call service provided by your node. Usage of rosservice is analogical to rostopic. To call service type:

rosservice call /tutorial\_pkg\_node/saved\_images

**Copy**

As a response you should get something like this:



**Message types**

In ROS there are many message types defined, they are grouped in packages accordingly to their application:

* [std\_msgs](http://docs.ros.org/kinetic/api/std_msgs/html/index-msg.html)
  + messages with basic data types like integer or float
* [sensor\_msgs](http://docs.ros.org/jade/api/sensor_msgs/html/index-msg.html)
  + for handling data from sensors
* [nav\_msgs](http://docs.ros.org/jade/api/nav_msgs/html/index-msg.html)
  + for handling maps or robot localization data
* [geometry\_msgs](http://docs.ros.org/jade/api/geometry_msgs/html/index-msg.html)
  + for handling information regarding object position, orientation, velocity or acceleration, also for defining points or polygons

While developing nodes for mobile robots you will probably want to use some of this messages:

* [sensor\_msgs/JointState](http://docs.ros.org/jade/api/sensor_msgs/html/msg/JointState.html)
  + it has fields for determining position , velocity and effort for a set of joints.
* [sensor\_msgs/LaserScan](http://docs.ros.org/jade/api/sensor_msgs/html/msg/LaserScan.html)
  + message for handling scans of planar laser scanners like RpLidar or Hokuyo.
* [sensor\_msgs/Range](http://docs.ros.org/jade/api/sensor_msgs/html/msg/Range.html)
  + message for one dimensional distance measurement
* [nav\_msgs/OccupancyGrid](http://docs.ros.org/api/nav_msgs/html/msg/OccupancyGrid.html)
  + message for handling flat occupancy grid based maps.
* [nav\_msgs/Odometry](http://docs.ros.org/api/nav_msgs/html/msg/Odometry.html)
  + message for defining robot position and orientation based on odometry measurements.
* [nav\_msgs/Path](http://docs.ros.org/api/nav_msgs/html/msg/Path.html)
  + message for defining robot path as a set of positions
* [geometry\_msgs/Twist](http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html)
  + message for defining linear and angular object velocity. Could be also used for setting desired velocities.

It is also possible to implement user defined messages if existing ones are not sufficient.

**Summary**

After completing this tutorial you should be able to write code for your own node, build and run it in the same way as it can be done with nodes provided with ROS.

Your node may be configured with parameters and it can:

* subscribe topics published by other nodes
* publish new topic
* provide service
* call service provided by other nodes

*by Łukasz Mitka, Husarion*

*Do you need any support with completing this tutorial or have any difficulties with software or hardware? Feel free to describe your thoughts on our community forum:*[*https://community.husarion.com/*](https://community.husarion.com/)*or to contact with our support:*[*support@husarion.com*](mailto:support@husarion.com)

[← 1. ROS INTRODUCTION](https://husarion.com/tutorials/ros-tutorials/1-ros-introduction)[3. SIMPLE KINEMATICS FOR MOBILE ROBOT →](https://husarion.com/tutorials/ros-tutorials/3-simple-kinematics-for-mobile-robot)