Conceptos Básicos

Navegando por el sistema de archivos

roscd : Cambia a un directorio de ROS

– roscd : Cambia al workspace

– roscd [proyecto] : Cambia a un proyecto.

rosls : Muestra la lista de proyectos.

Crear Paquetes

roscreate-pkg [package\_name]

roscreate-pkg [package\_name] [depend1] [depend2] [depend3]

roscd

$ cd sandbox

Then create your package:

$ roscreate-pkg beginner\_tutorials std\_msgs rospy roscpp

### **First-order package dependencies**

When using roscreate-pkg earlier, a few package dependencies were provided. These **first-order** dependencies can now be reviewed with the rospack tool.

$ rospack depends1 beginner\_tutorials

As you can see, rospack lists the same dependencies that were used as arguments when running roscreate-pkg.These dependencies for a package are stored in the **manifest** file. Take a look at the manifest file.

$ roscd beginner\_tutorials

$ cat manifest.xml

### **Indirect package dependencies**

In many cases, a dependency will also have its own dependencies. For instance, rospy has other dependencies.

$ rospack depends1 rospy

A package can have quite a few indirect dependencies. Luckily rospack can recursively determine all nested dependencies.

$ rospack depends beginner\_tutorials

**Building a ROS Package**

#### **Using rosmake**

rosmake is just like the make command, but it does some special ROS magic. When you typerosmake beginner\_tutorials, it builds the beginner\_tutorials package, plus every package that it depends on, in the correct order. Since we listed rospy, roscpp, and std\_msgs as dependencies when creating our ROS package, these packages (and their dependiencies, and so on) will be built by rosmake as well.

Usage:

rosmake [package]

Try:

$ rosmake beginner\_tutorials

#### rosmake multiple packages

We can also use rosmake to build multiple packages at once.

Usage:

rosmake [package1] [package2] [package3]

## **Understanding ROS Nodes**

A node really isn't much more than an executable file within a ROS package. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a Topic. Nodes can also provide or use a Service.

### **Client Libraries**

ROS client libraries allow nodes written in different programming languages to communicate:

* rospy = python client library
* roscpp = c++ client library

### **roscore**

roscore is the first thing you should run when using ROS.

Please run:

$ roscore

**Using rosnode**

Open up a **new terminal**, and let's use **rosnode** to see what running roscore did...

rosnode displays information about the ROS nodes that are currently running. The rosnode list command lists these active nodes:

$ rosnode list

This showed us that there is only one node running: [rosout](http://wiki.ros.org/rosout). This is always running as it collects and logs nodes' debugging output.

The rosnode info command returns information about a specific node.

$ rosnode info /rosout

### **Using rosrun**

rosrun allows you to use the package name to directly run a node within a package (without having to know the package path).

Usage:

$ rosrun [package\_name] [node\_name]

So now we can run the turtlesim\_node in the turtlesim package.

Then, in a **new terminal**:

$ rosrun turtlesim turtlesim\_node

One powerful feature of ROS is that you can reassign Names from the command-line.

$ rosrun turtlesim turtlesim\_node \_\_name:=my\_turtle

We see our new /my\_turtle node. Let's use another rosnode command, ping, to test that it's up:

$ rosnode ping my\_turtle

## **Understanding ROS Topics**

#### **turtlesim**

For this tutorial we will also use turtlesim. Please run **in a new terminal**:

$ rosrun turtlesim turtlesim\_node

#### **turtle keyboard teleoperation**

We'll also need something to drive the turtle around with. Please run **in a new terminal**:

$ rosrun turtlesim turtle\_teleop\_key

Now you can use the arrow keys of the keyboard to drive the turtle around.

### **ROS Topics**

**In a new terminal**:

$ rosrun rqt\_graph rqt\_graph

**Using rostopic echo**

rostopic echo shows the data published on a topic.

Usage:

rostopic echo [topic]

Let's look at the command velocity data published by the turtle\_teleop\_key node.

For ROS Groovy and earlier, this data is published on the /turtle1/command\_velocity topic. **In a new terminal, run:**

$ rostopic echo /turtle1/command\_velocity

#### **Using rostopic list**

rostopic list returns a list of all topics currently subscribed to and published.

For rostopic list use the **verbose** option:

$ rostopic list -v

**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.

#### **Using rostopic type**

rostopic type returns the message type of any topic being published.

Usage:

rostopic type [topic]

For ROS Groovy and earlier,

$ rostopic type /turtle1/command\_velocity

We can look at the details of the message using rosmsg:

$ rosmsg show turtlesim/Velocity

Now that we know what type of message turtlesim expects, we can publish commands to our turtle.

### **rostopic continued**

Now that we have learned about ROS **messages**, let's use rostopic with messages.

#### **Using rostopic pub**

rostopic pub publishes data on to a topic currently advertised.

Usage:

rostopic pub [topic] [msg\_type] [args]

For ROS Groovy and earlier, example:

$ rostopic pub -1 /turtle1/command\_velocity turtlesim/Velocity -- 2.0 1.8

For ROS Groovy and earlier,

* This command will publish messages to a given topic:

rostopic pub

* This option (dash-one) causes rostopic to only publish one message then exit:

-1

* This is the name of the topic to publish to:

/turtle1/command\_velocity

* This is the message type to use when publishing the topic:

turtlesim/Velocity

* This option (double-dash) tells the option parser that none of the following arguments is an option. This is required in cases where your arguments have a leading dash -, like negative numbers.

--

* As noted before, a turtlesim/Velocity msg has two floating point elements : linear and angular. In this case, 2.0becomes the linear value, and 1.8 is the angular value. These arguments are actually in YAML syntax, which is described more in the [YAML command line documentation](http://wiki.ros.org/ROS/YAMLCommandLine).

2.0 1.8

You may have noticed that the turtle has stopped moving; this is because the turtle requires a steady stream of commands at 1 Hz to keep moving. We can publish a steady stream of commands using rostopic pub -r command:

For ROS Groovy and earlier,

* $ rostopic pub /turtle1/command\_velocity turtlesim/Velocity -r 1 -- 2.0 -1.8

This publishes the velocity commands at a rate of 1 Hz on the velocity topic.

#### **Using rostopic hz**

rostopic hz reports the rate at which data is published.

Usage:

rostopic hz [topic]

Let's see how fast the turtlesim\_node is publishing /turtle1/pose:

$ rostopic hz /turtle1/pose

Now we can tell that the turtlesim is publishing data about our turtle at the rate of 60 Hz. We can also use rostopic typein conjunction with rosmsg show to get in depth information about a topic:

For ROS Groovy and earlier,

* $ rostopic type /turtle1/command\_velocity | rosmsg show

Now that we've examined the topics using rostopic let's use another tool to look at the data published by our turtlesim:

## **Understanding ROS Services and Parameters**

**Description:** This tutorial introduces ROS services, and parameters as well as using the [rosservice](http://wiki.ros.org/rosservice) and [rosparam](http://wiki.ros.org/rosparam)commandline tools

Assuming your turtlesim\_node is still running from the last tutorial, let's look at what services the turtlesim provides:

### **ROS Services**

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

### **Using rosservice**

rosservice can easily attach to ROS's client/service framework with services. rosservice has many commands that can be used on topics, as shown below:

Usage:

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

#### **rosservice list**

$ rosservice list

The list command shows us that the turtlesim node provides nine services: reset, clear, spawn, kill,turtle1/set\_pen, /turtle1/teleport\_absolute, /turtle1/teleport\_relative, turtlesim/get\_loggers, and turtlesim/set\_logger\_level. There are also two services related to the separate rosout node:/rosout/get\_loggers and /rosout/set\_logger\_level.

Let's look more closely at the clear service using rosservice type:

#### **rosservice type**

Usage:

rosservice type [service]

Let's find out what type the clear service is:

$ rosservice type clear

This service is empty, this means when the service call is made it takes no arguments (i.e. it sends no data when making a**request** and receives no data when receiving a **response**). Let's call this service using rosservice call:

#### **rosservice call**

Usage:

rosservice call [service] [args]

Here we'll call with no arguments because the service is of type empty:

$ rosservice call /clear

This does what we expect, it clears the background of the turtlesim\_node.

Let's look at the case where the service has arguments by looking at the information for the service spawn:

$ rosservice type spawn| rossrv show

This service lets us spawn a new turtle at a given location and orientation. The name field is optional, so let's not give our new turtle a name and let turtlesim create one for us.

$ rosservice call spawn 2 2 0.2 ""

The service call returns with the name of the newly created turtle

Now our turtlesim should look like this:

### **Using rosparam**

rosparam allows you to store and manipulate data on the ROS [Parameter Server](http://wiki.ros.org/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:

Usage:

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

Let's look at what parameters are currently on the param server:

#### **rosparam list**

$ rosparam list

Here we can see that the turtlesim node has three parameters on the param server for background color:

Let's change one of the parameter values using rosparam set:

#### **rosparam set and rosparam get**

Usage:

rosparam set [param\_name]

rosparam get [param\_name]

Here will change the red channel of the background color:

$ rosparam set background\_r 150

This changes the parameter value, now we have to call the clear service for the parameter change to take effect:

$ rosservice call clear

Now our turtlesim looks like this:

Now let's look at the values of other parameters on the param server. Let's get the value of the green background channel:

$ rosparam get background\_g

We can also use rosparam get / to show us the contents of the entire Parameter Server.

$ rosparam get /

You may wish to store this in a file so that you can reload it at another time. This is easy using rosparam:

#### **rosparam dump and rosparam load**

Usage:

rosparam dump [file\_name] [namespace]

rosparam load [file\_name] [namespace]

Here we write all the parameters to the file params.yaml

$ rosparam dump params.yaml

You can even load these yaml files into new namespaces, e.g. copy:

$ rosparam load params.yaml copy

$ rosparam get copy/background\_b

## **Using rqt\_console and roslaunch**

**Description:** This tutorial introduces ROS using [rqt\_console](http://wiki.ros.org/rqt_console) and [rqt\_logger\_level](http://wiki.ros.org/rqt_logger_level) for debugging and [roslaunch](http://wiki.ros.org/roslaunch) for starting many nodes at once. If you use ROS fuerte or ealier distros where [rqt](http://wiki.ros.org/rqt) isn't fully available, please see this page with [this page](http://wiki.ros.org/ROS/Tutorials/UsingRxconsoleRoslaunch) that uses old rx based tools.  
  
**Tutorial Level:** BEGINNER  
  
**Next Tutorial:** [Using rosed](http://wiki.ros.org/ROS/Tutorials/UsingRosEd)

**Tabla de Contenidos**

1. [Prerequisites rqt and turtlesim package](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "Prerequisites_rqt_and_turtlesim_package)
2. [Using rqt\_console and rqt\_logger\_level](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "Using_rqt_console_and_rqt_logger_level)
   1. [Quick Note about logger levels](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "Quick_Note_about_logger_levels)
   2. [Using roslaunch](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "Using_roslaunch)
   3. [The Launch File](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "The_Launch_File)
   4. [The Launch File Explained](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "The_Launch_File_Explained)
   5. [roslaunching](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "roslaunching)

### **Prerequisites rqt and turtlesim package**

The tutorial uses both the rqt and turtlesim packages. To do this tutorial, please install both packages, if you have not yet done so.

$ sudo apt-get install ros-<distro>-rqt ros-<distro>-rqt-common-plugins ros-<distro>-turtlesim

Replace <distro> with the name of your ROS distribution (e.g. hydro, groovy, electric, fuerte).

**NOTE:** you may have already built rqt and turtlesim for one of the previous tutorials. If you are not sure, installing them again will not hurt anything.

### **Using rqt\_console and rqt\_logger\_level**

rqt\_console 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.

Now let's look at the turtlesim output in rqt\_console and switch logger levels in rqt\_logger\_level as we use turtlesim. Before we start the turtlesim, **in two new terminals** start rqt\_console and rqt\_logger\_level:

$ rosrun rqt\_console rqt\_console

$ rosrun rqt\_logger\_level rqt\_logger\_level

Now let's start turtlesim in a **new terminal**:

$ rosrun turtlesim turtlesim\_node

Since the default logger level is INFO you will see any info that the turtlesim publishes when it starts up, which should look like:

Now let's change the logger level to Warn by refreshing the nodes in the rqt\_logger\_level window and selecting Warn as shown below:

Now let's run our turtle into the wall and see what is displayed in our rqt\_console:

For ROS Hydro and later,

* rostopic pub /turtle1/cmd\_vel geometry\_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 0.0]'

For ROS Groovy and earlier,

* rostopic pub /turtle1/command\_velocity turtlesim/Velocity -r 1 -- 2.0 0.0

#### **Quick Note about logger levels**

Logging levels are prioritized in the following order:

Fatal

Error

Warn

Info

Debug

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

Let's Ctrl-C our turtlesim and let's use roslaunch to bring up multiple turtlesim nodes and a mimicking node to cause one turtlesim to mimic another:

### **Introduction to msg and srv**

* [msg](http://wiki.ros.org/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 languages.
* [srv](http://wiki.ros.org/srv): an srv file describes a service. It is composed of two parts: a request and a response.

msg files are stored in the msg directory of a package, and srv files are stored in the srv directory.

msgs are just simple text files with a field type and field name per line. The field types you can use are:

* int8, int16, int32, int64 (plus uint\*)
* float32, float64
* string
* time, duration
* other msg files
* variable-length array[] and fixed-length array[C]

There is also a special type in ROS: Header, the header contains a timestamp and coordinate frame information that are commonly used in ROS. You will frequently see the first line in a msg file have Header header.

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

Header header

string child\_frame\_id

geometry\_msgs/PoseWithCovariance pose

geometry\_msgs/TwistWithCovariance twist

srv files are just like msg files, except they contain two parts: a request and a response. The two parts are separated by a '---' line. Here is an example of a srv file:

int64 A

int64 B

---

int64

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

## **Using msg**

### **Creating a msg**

Let's define a new msg in the package that was created in the previous tutorial.

$ roscd beginner\_tutorials

$ mkdir msg

$ echo "int64 num" > msg/Num.msg

The example above is the simplest, where the .msg file contains only 1 line. You can, of course, create more complex files by adding multiple elements per line like this:

string first\_name

string last\_name

uint8 age

uint32 score

There's one more step, though. We need to make sure that the msg files are turned into source code for C++, Python, and other languages. Open CMakeLists.txt in your favorite text editor ([rosed](http://wiki.ros.org/ROS/Tutorials/UsingRosEd) from the previous tutorial is a good option) and remove # to uncomment the following line:

# rosbuild\_genmsg()

### **Using rosmsg**

That's all you need to do to create a msg. Let's make sure that ROS can see it using the rosmsg show command.

Usage:

$ rosmsg show [message type]

Example:

$ rosmsg show beginner\_tutorials/Num

In the previous example, the message type consists of two parts:

* beginner\_tutorials -- the package where the message is defined
* Num -- The name of the msg Num.

If you can't remember which Package a msg is in, you can leave out the package name. Try:

$ rosmsg show Num

## **Using srv**

### **Creating a srv**

Let's use the package we just created to create a srv:

$ roscd beginner\_tutorials

$ mkdir srv

Instead of creating a new srv definition by hand, we will copy an existing one from another package.

For that, roscp is a useful commandline tool for copying files from one package to another.

Usage:

$ roscp [package\_name] [file\_to\_copy\_path] [copy\_path]

Now we can copy a service from the [rospy\_tutorials](http://wiki.ros.org/rospy_tutorials) package:

$ roscp rospy\_tutorials AddTwoInts.srv srv/AddTwoInts.srv

There's one more step, though. We need to make sure that the srv files are turned into source code for C++, Python, and other languages.

Once again, open CMakeLists.txt and remove # to uncomment the following line:

# rosbuild\_gensrv()

### **Using rossrv**

That's all you need to do to create a srv. Let's make sure that ROS can see it using the rossrv show command.

Usage:

$ rossrv show <service type>

Example:

$ rossrv show beginner\_tutorials/AddTwoInts

Similar to rosmsg, you can find service files like this without specifying package name:

$ rossrv show AddTwoInts

## **Common step for msg and srv**

Now that we have made some new messages we need to make our package again:

$ rosmake beginner\_tutorials

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/.

The full specification for the message format is available at the [Message Description Language](http://wiki.ros.org/ROS/Message_Description_Language) page.

## **Getting Help**

We've seen quite a few ROS tools already. It can be difficult to keep track of what arguments each command requires. Luckily, most ROS tools provide their own help.

Try:

$ rosmsg -h

* You should see a list of different rosmsg subcommands.

You can also get help for subcommands

$ rosmsg show -h

## **Review**

Lets just list some of the commands we've used so far:

* rospack = ros+pack(age) : provides information related to ROS packages
* roscd = ros+cd : **c**hanges **d**irectory to a ROS package or stack
* rosls = ros+ls : **l**ist**s** files in a ROS package
* roscp = ros+cp : **c**o**p**ies files from/to a ROS package
* rosmsg = ros+msg : provides information related to ROS message definitions
* rossrv = ros+srv : provides information related to ROS service definitions
* catkin\_make : makes (compiles) a ROS package
  + rosmake = ros+make : makes (compiles) a ROS package (if you're not using a catkin workspace)

## **Writing a Simple Publisher and Subscriber (Python)**

### **Writing the Publisher Node**

"Node" is the ROS term for an executable that is connected to the ROS network. Here we'll create the publisher ("talker") node which will continually broadcast a message.

Change directory into the beginner\_tutorials package, you created in the earlier tutorial, [creating a package](http://wiki.ros.org/ROS/Tutorials/CreatingPackage):

$ roscd beginner\_tutorials

#### **The Code**

First lets create a 'scripts' folder to store our Python scripts in:

$ mkdir scripts

$ cd scripts

Then download the example script [talker.py](https://raw.github.com/ros/ros_tutorials/indigo-devel/rospy_tutorials/001_talker_listener/talker.py) to your new scripts directory and make it executable:

$ wget https://raw.github.com/ros/ros\_tutorials/indigo-devel/rospy\_tutorials/001\_talker\_listener/talker.py

$ chmod +x talker.py

You can view and edit the file with  $ rosed beginner\_tutorials talker.py  or just look below.

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python))

#!/usr/bin/env python

# license removed for brevity

import rospy

from std\_msgs.msg import String

def talker():

pub = rospy.Publisher('chatter', String, queue\_size=10)

rospy.init\_node('talker', anonymous=True)

rate = rospy.Rate(10) # 10hz

while not rospy.is\_shutdown():

hello\_str = "hello world %s" % rospy.get\_time()

rospy.loginfo(hello\_str)

pub.publish(hello\_str)

rate.sleep()

if \_\_name\_\_ == '\_\_main\_\_':

try:

talker()

except rospy.ROSInterruptException:

pass

#### **The Code Explained**

Now, let's break the code down.

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python))

#!/usr/bin/env python

Every Python ROS [Node](http://wiki.ros.org/Nodes) will have this declaration at the top. The first line makes sure your script is executed as a Python script.

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python))

import rospy

from std\_msgs.msg import String

You need to import rospy if you are writing a ROS [Node](http://wiki.ros.org/Nodes). The std\_msgs.msg import is so that we can reuse thestd\_msgs/String message type (a simple string container) for publishing.

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python))

pub = rospy.Publisher('chatter', String, queue\_size=10)

rospy.init\_node('talker', anonymous=True)

This section of code defines the talker's interface to the rest of ROS.pub = rospy.Publisher("chatter", String, queue\_size=10) declares that your node is publishing to thechatter topic using the message type String. String here is actually the class std\_msgs.msg.String. Thequeue\_size argument is **New in ROS hydro** and limits the amount of queued messages if any subscriber is not receiving the them fast enough. In older ROS distributions just omit the argument.

The next line, rospy.init\_node(NAME), is very important as it tells rospy the name of your node -- until rospy has this information, it cannot start communicating with the ROS [Master](http://wiki.ros.org/Master). In this case, your node will take on the name talker. NOTE: the name must be a [base name](http://wiki.ros.org/Names), i.e. it cannot contain any slashes "/".

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python))

rate = rospy.Rate(10) # 10hz

This line creates a Rate object r. With the help of its method sleep(), it offers a convenient way for looping at the desired rate. With its argument of 10, we should expect to go through the loop 10 times per second (as long as our processing time does not exceed 1/10th of a second!)

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python))

while not rospy.is\_shutdown():

hello\_str = "hello world %s" % rospy.get\_time()

rospy.loginfo(hello\_str)

pub.publish(hello\_str)

rate.sleep()

This loop is a fairly standard rospy construct: checking the rospy.is\_shutdown() flag and then doing work. You have to check is\_shutdown() to check if your program should exit (e.g. if there is a Ctrl-C or otherwise). In this case, the "work" is a call to pub.publish(String(str)) that publishes to our chatter topic using a newly created String message. The loop calls r.sleep(), which sleeps just long enough to maintain the desired rate through the loop.

(You may also run across rospy.sleep() which is similar to time.sleep() except that it works with simulated time as well (see [Clock](http://wiki.ros.org/Clock)).)

This loop also calls rospy.loginfo(str), which performs triple-duty: the messages get printed to screen, it gets written to the Node's log file, and it gets written to [rosout](http://wiki.ros.org/rosout). [rosout](http://wiki.ros.org/rosout) is a handy for debugging: you can pull up messages using[rqt\_console](http://wiki.ros.org/rqt_console) instead of having to find the console window with your Node's output.

std\_msgs.msg.String is a very simple message type, so you may be wondering what it looks like to publish more complicated types. The general rule of thumb is that constructor args are in the same order as in the .msg file. You can also pass in no arguments and initialize the fields directly, e.g.

msg = String()

msg.data = str

or you can initialize some of the fields and leave the rest with default values:

String(data=str)

You may be wondering about the last little bit:

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python))

try:

talker()

except rospy.ROSInterruptException:

pass

In addition to the standard Python \_\_main\_\_ check, this catches a rospy.ROSInterruptException exception, which can be thrown by rospy.sleep() and rospy.Rate.sleep() methods when Ctrl-C is pressed or your Node is otherwise shutdown. The reason this exception is raised is so that you don't accidentally continue executing code after thesleep().

Now we need to write a node to receive the messages.

### **Writing the Subscriber Node**

#### **The Code**

Download the [listener.py](https://raw.github.com/ros/ros_tutorials/indigo-devel/rospy_tutorials/001_talker_listener/listener.py) file into your scripts directory:

$ roscd beginner\_tutorials/scripts/

$ wget https://raw.github.com/ros/ros\_tutorials/indigo-devel/rospy\_tutorials/001\_talker\_listener/listener.py

The file contents look close to:

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python))

#!/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():

# In ROS, nodes are uniquely named. If two nodes with the same

# node are launched, the previous one is kicked off. The

# anonymous=True flag means that rospy will choose a unique

# name for our 'listener' node so that multiple listeners can

# run simultaneously.

rospy.init\_node('listener', anonymous=True)

rospy.Subscriber("chatter", String, callback)

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

rospy.spin()

if \_\_name\_\_ == '\_\_main\_\_':

listener()

Don't forget to make the node executable:

$ chmod +x listener.py

#### **The Code Explained**

The code for listener.py is similar to talker.py, except we've introduced a new callback-based mechanism for subscribing to messages.

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python))

rospy.init\_node('listener', anonymous=True)

rospy.Subscriber("chatter", String, callback)

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

rospy.spin()

This declares that your node subscribes to the chatter topic which is of type std\_msgs.msgs.String. When new messages are received, callback is invoked with the message as the first argument.

We also changed up the call to rospy.init\_node() somewhat. We've added the anonymous=True keyword argument. ROS requires that each node have a unique name. If a node with the same name comes up, it bumps the previous one. This is so that malfunctioning nodes can easily be kicked off the network. The anonymous=True flag tells rospy to generate a unique name for the node so that you can have multiple listener.py nodes run easily.

The final addition, rospy.spin() simply keeps your 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.

### **Building your nodes**

We use CMake as our build system and, yes, you have to use it even for Python nodes. This is to make sure that the autogenerated Python code for messages and services is created.

We also use a Makefile for a bit of convenience. roscreate-pkg automatically created a Makefile, so you don't have to edit it.

Now run make:

$ make

Now that you have written a simple publisher and subscriber, let's [examine the simple publisher and subscriber](http://wiki.ros.org/ROS/Tutorials/ExaminingPublisherSubscriber).

## **Examining the Simple Publisher and Subscriber**

### **Running the Publisher**

Make sure that a roscore is up and running:

$ roscore

**catkin specific** If you are using catkin, make sure you have sourced your workspace's setup.sh file after callingcatkin\_make but before trying to use your applications:

# In your catkin workspace

$ cd ~/catkin\_ws

$ source ./devel/setup.bash

In the last tutorial we made a publisher called "talker". Let's run it:

$ rosrun beginner\_tutorials talker (C++)

$ rosrun beginner\_tutorials talker.py (Python)

The publisher node is up and running. Now we need a subscriber to receive messages from the publisher.

### **Running the Subscriber**

In the last tutorial we made a subscriber called "listener". Let's run it:

$ rosrun beginner\_tutorials listener (C++)

$ rosrun beginner\_tutorials listener.py (Python)

## **Writing a Simple Service and Client (Python)**

**Writing a Service Node**

Here we'll create the service ("add\_two\_ints\_server") node which will receive two ints and return the sum.

Change directory into the beginner\_tutorials package, you created in the earlier tutorial, [creating a package](http://wiki.ros.org/ROS/Tutorials/CreatingPackage):

$ roscd beginner\_tutorials

Please make sure you have followed the directions in the previous tutorial for creating the service needed in this tutorial,[creating the AddTwoInts.srv](http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv" \l "Creating_a_srv) (be sure to choose the right version of build tool you're using at the top of wiki page in the link).

#### **The Code**

Create the **scripts/add\_two\_ints\_server.py** file within the beginner\_tutorials package and paste the following inside it:

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(python))

#!/usr/bin/env python

import roslib; roslib.load\_manifest('beginner\_tutorials')

from beginner\_tutorials.srv import \*

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()

Don't forget to make the node executable:

* chmod +x scripts/add\_two\_ints\_server.py

#### **The Code Explained**

Now, let's break the code down.

There's very little to writing a service using [rospy](http://wiki.ros.org/rospy). We declare our node using init\_node() and then declare our service:

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(python))

s = rospy.Service('add\_two\_ints', AddTwoInts, handle\_add\_two\_ints)

This declares a new service named add\_two\_ints with the AddTwoInts service type. All requests are passed tohandle\_add\_two\_ints function. handle\_add\_two\_ints is called with instances of AddTwoIntsRequest and returns instances of AddTwoIntsResponse.

Just like with the subscriber example, rospy.spin() keeps your code from exiting until the service is shutdown.

### **Writing the Client Node**

#### **The Code**

Create the **scripts/add\_two\_ints\_client.py** file within the beginner\_tutorials package and paste the following inside it:

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(python))

#!/usr/bin/env python

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, 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))

Don't forget to make the node executable:

$ chmod +x scripts/add\_two\_ints\_client.py

#### **The Code Explained**

Now, let's break the code down.

The client code for calling services is also simple. For clients you don't have to call init\_node(). We first call:

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(python))

rospy.wait\_for\_service('add\_two\_ints')

This is a convenience method that blocks until the service named add\_two\_ints is available. Next we create a handle for calling the service:

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(python))

add\_two\_ints = rospy.ServiceProxy('add\_two\_ints', AddTwoInts)

We can use this handle just like a normal function and call it:

[[des]activar nros. de línea](http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(python))

resp1 = add\_two\_ints(x, y)

return resp1.sum

Because we've declared the type of the service to be AddTwoInts, it does the work of generating theAddTwoIntsRequest object for you (you're free to pass in your own instead). The return value is anAddTwoIntsResponse object. If the call fails, a rospy.ServiceException may be thrown, so you should setup the appropriate try/except block.

### **Building your nodes**

We use CMake as our build system and, yes, you have to use it even for Python nodes. This is to make sure that the [autogenerated Python code for messages and services](http://www.ros.org/wiki/ROS/Tutorials/CreatingMsgAndSrv" \l "Creating_a_srv) is created.

We also use a Makefile for a bit of convenience. roscreate-pkg automatically created a Makefile, so you don't have to edit it.

Now run make:

$ make

### **Try it out!**

In a **new terminal**, run

$ rosrun beginner\_tutorials add\_two\_ints\_server.py

In a **new terminal**, run

$ rosrun beginner\_tutorials add\_two\_ints\_client.py

Then run

$ rosrun beginner\_tutorials add\_two\_ints\_client.py 4 5