

**Note:** This tutorial assumes that you have completed the previous tutorials: understanding ROS services and parameters (/ROS/Tutorials/UnderstandingServicesParams).

💡 Please ask about problems and questions regarding this tutorial on [answers.ros.org](http://answers.ros.org) (<http://answers.ros.org>). Don't forget to include in your question the link to this page, the versions of your OS & ROS, and also add appropriate tags.

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

**Description:** This tutorial covers how to write a publisher and subscriber node in C++.

**Tutorial Level:** BEGINNER

**Next Tutorial:** Examining the simple publisher and subscriber (/ROS/Tutorials/ExaminingPublisherSubscriber)

catkin

roscpp

## Contents

1. Writing the Publisher Node
  1. The Code
  2. The Code Explained
2. Writing the Subscriber Node
  1. The Code
  2. The Code Explained
3. Building your nodes
4. Building your nodes
5. Additional Resources
  1. Video Tutorial

## 1. Writing the Publisher Node

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

Change directories to your beginner\_tutorials package you created in your catkin workspace previous tutorials:

```
roscd beginner_tutorials
```

### 1.1 The Code

Create a src directory in the beginner\_tutorials package directory:

```
mkdir -p src
```

This directory will contain any source files for our beginner\_tutorials package.

Create the src/talker.cpp file within the beginner\_tutorials package and paste the following inside it:

[https://raw.githubusercontent.com/ros/ros\\_tutorials/kinetic-devel/roscpp\\_tutorials/talker/talker.cpp](https://raw.githubusercontent.com/ros/ros_tutorials/kinetic-devel/roscpp_tutorials/talker/talker.cpp)  
([https://raw.githubusercontent.com/ros/ros\\_tutorials/kinetic-devel/roscpp\\_tutorials/talker/talker.cpp](https://raw.githubusercontent.com/ros/ros_tutorials/kinetic-devel/roscpp_tutorials/talker/talker.cpp))

Toggle line numbers

```

27 #include "ros/ros.h"
28 #include "std_msgs/String.h"
29
30 #include <sstream>
31
32 /**
33  * This tutorial demonstrates simple sending of messages over the ROS system.
34  */
35 int main(int argc, char **argv)
36 {
37     /**
38      * The ros::init() function needs to see argc and argv so that it can perform
39      * any ROS arguments and name remapping that were provided at the command line.
40      * For programmatic remappings you can use a different version of init() which ta
kes
41      * remappings directly, but for most command-line programs, passing argc and argv
is
42      * the easiest way to do it. The third argument to init() is the name of the nod
e.
43      *
44      * You must call one of the versions of ros::init() before using any other
45      * part of the ROS system.
46      */
47     ros::init(argc, argv, "talker");
48
49     /**
50      * NodeHandle is the main access point to communications with the ROS system.
51      * The first NodeHandle constructed will fully initialize this node, and the last
52      * NodeHandle destructed will close down the node.
53      */
54     ros::NodeHandle n;
55
56     /**
57      * The advertise() function is how you tell ROS that you want to
58      * publish on a given topic name. This invokes a call to the ROS
59      * master node, which keeps a registry of who is publishing and who
60      * is subscribing. After this advertise() call is made, the master
61      * node will notify anyone who is trying to subscribe to this topic name,
62      * and they will in turn negotiate a peer-to-peer connection with this
63      * node. advertise() returns a Publisher object which allows you to
64      * publish messages on that topic through a call to publish(). Once
65      * all copies of the returned Publisher object are destroyed, the topic
66      * will be automatically unadvertised.
67      *
68      * The second parameter to advertise() is the size of the message queue
69      * used for publishing messages. If messages are published more quickly
70      * than we can send them, the number here specifies how many messages to
71      * buffer up before throwing some away.
72      */
73     ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
74
75     ros::Rate loop_rate(10);
76
77     /**
78      * A count of how many messages we have sent. This is used to create

```

```

79  * a unique string for each message.
80  */
81  int count = 0;
82  while (ros::ok())
83  {
84      /**
85       * This is a message object. You stuff it with data, and then publish it.
86       */
87      std_msgs::String msg;
88
89      std::stringstream ss;
90      ss << "hello world " << count;
91      msg.data = ss.str();
92
93      ROS_INFO("%s", msg.data.c_str());
94
95      /**
96       * The publish() function is how you send messages. The parameter
97       * is the message object. The type of this object must agree with the type
98       * given as a template parameter to the advertise<>() call, as was done
99       * in the constructor above.
100     */
101     chatter_pub.publish(msg);
102
103     ros::spinOnce();
104
105     loop_rate.sleep();
106     ++count;
107 }
108
109
110 return 0;
111 }

```

## 1.2 The Code Explained

Now, let's break the code down.

Toggle line numbers

```

27 #include "ros/ros.h"
28

```

`ros/ros.h` is a convenience include that includes all the headers necessary to use the most common public pieces of the ROS system.

Toggle line numbers

```

28 #include "std_msgs/String.h"
29

```

This includes the `std_msgs/String` ([http://docs.ros.org/api/std\\_msgs/html/msg/String.html](http://docs.ros.org/api/std_msgs/html/msg/String.html)) message, which resides in the `std_msgs` (`/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` (`/msg`) page.

Toggle line numbers

```
47  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.

The name used here must be a base name (/Names#Graph), ie. it cannot have a / in it.

Toggle line numbers

```
54  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.

Toggle line numbers

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

Tell the master that we are going to be publishing a message of type `std_msgs/String` ([http://docs.ros.org/api/std\\_msgs/html/msg/String.html](http://docs.ros.org/api/std_msgs/html/msg/String.html)) on the topic `chatter`. This lets the master tell any nodes listening on `chatter` that we are going to publish data on that topic. The second argument is the size of our publishing queue. In this case if we are publishing too quickly it will buffer up a maximum of 1000 messages before beginning to throw away old ones.

`NodeHandle::advertise()` returns a `ros::Publisher` object, which serves two purposes: 1) it contains a `publish()` method that lets you publish messages onto the topic it was created with, and 2) when it goes out of scope, it will automatically `unadvertise`.

Toggle line numbers

```
75  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.

In this case we tell it we want to run at 10Hz.

Toggle line numbers

```
81  int count = 0;
82  while (ros::ok())
83  {
```

By default `roscpp` will install a `SIGINT` handler which provides `Ctrl-C` handling which will cause `ros::ok()` to return false if that happens.

`ros::ok()` will return false if:

- a `SIGINT` is received (`Ctrl-C`)
- we have been kicked off the network by another node with the same name
- `ros::shutdown()` has been called by another part of the application.
- all `ros::NodeHandles` (`/NodeHandles`) have been destroyed

Once `ros::ok()` returns false, all ROS calls will fail.

Toggle line numbers

```

87     std_msgs::String msg;
88
89     std::stringstream ss;
90     ss << "hello world " << count;
91     msg.data = ss.str();

```

We broadcast a message on ROS using a message-adapted class, generally generated from a msg file (/msg). More complicated datatypes are possible, but for now we're going to use the standard `String` message, which has one member: "data".

Toggle line numbers

```

101     chatter_pub.publish(msg);

```

Now we actually broadcast the message to anyone who is connected.

Toggle line numbers

```

93     ROS_INFO("%s", msg.data.c_str());

```

`ROS_INFO` and friends are our replacement for `printf/cout`. See the `roscpp` documentation (/roscpp) for more information.

Toggle line numbers

```

103     ros::spinOnce();

```

Calling `ros::spinOnce()` here is not necessary for this simple program, because we are not receiving any callbacks. However, if you were to add a subscription into this application, and did not have `ros::spinOnce()` here, your callbacks would never get called. So, add it for good measure.

Toggle line numbers

```

105     loop_rate.sleep();

```

Now we use the `ros::Rate` object to sleep for the time remaining to let us hit our 10Hz publish rate.

Here's the condensed version of what's going on:

- Initialize the ROS system
- Advertise that we are going to be publishing `std_msgs/String` ([http://docs.ros.org/api/std\\_msgs/html/msg/String.html](http://docs.ros.org/api/std_msgs/html/msg/String.html)) messages on the `chatter` topic to the master
- Loop while publishing messages to `chatter` 10 times a second

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

## 2. Writing the Subscriber Node

### 2.1 The Code

Create the `src/listener.cpp` file within the `beginner_tutorials` package and paste the following inside it:

[https://raw.githubusercontent.com/ros/ros\\_tutorials/kinetic-devel/roscpp\\_tutorials/listener/listener.cpp](https://raw.githubusercontent.com/ros/ros_tutorials/kinetic-devel/roscpp_tutorials/listener/listener.cpp)  
 ([https://raw.githubusercontent.com/ros/ros\\_tutorials/kinetic-devel/roscpp\\_tutorials/listener/listener.cpp](https://raw.githubusercontent.com/ros/ros_tutorials/kinetic-devel/roscpp_tutorials/listener/listener.cpp))

Toggle line numbers

```

28 #include "ros/ros.h"
29 #include "std_msgs/String.h"
30
31 /**
32  * This tutorial demonstrates simple receipt of messages over the ROS system.
33  */
34 void chatterCallback(const std_msgs::String::ConstPtr& msg)
35 {
36     ROS_INFO("I heard: [%s]", msg->data.c_str());
37 }
38
39 int main(int argc, char **argv)
40 {
41     /**
42      * The ros::init() function needs to see argc and argv so that it can perform
43      * any ROS arguments and name remapping that were provided at the command line.
44      * For programmatic remappings you can use a different version of init() which ta
kes
45      * remappings directly, but for most command-line programs, passing argc and argv
is
46      * the easiest way to do it. The third argument to init() is the name of the nod
e.
47      *
48      * You must call one of the versions of ros::init() before using any other
49      * part of the ROS system.
50      */
51     ros::init(argc, argv, "listener");
52
53     /**
54      * NodeHandle is the main access point to communications with the ROS system.
55      * The first NodeHandle constructed will fully initialize this node, and the last
56      * NodeHandle destructed will close down the node.
57      */
58     ros::NodeHandle n;
59
60     /**
61      * The subscribe() call is how you tell ROS that you want to receive messages
62      * on a given topic. This invokes a call to the ROS
63      * master node, which keeps a registry of who is publishing and who
64      * is subscribing. Messages are passed to a callback function, here
65      * called chatterCallback. subscribe() returns a Subscriber object that you
66      * must hold on to until you want to unsubscribe. When all copies of the Subscri
ber
67      * object go out of scope, this callback will automatically be unsubscribed from
68      * this topic.
69      *
70      * The second parameter to the subscribe() function is the size of the message
71      * queue. If messages are arriving faster than they are being processed, this
72      * is the number of messages that will be buffered up before beginning to throw
73      * away the oldest ones.
74      */
75     ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
76
77     /**
78      * ros::spin() will enter a loop, pumping callbacks. With this version, all

```

```

79  * callbacks will be called from within this thread (the main one).  ros::spin()
80  * will exit when Ctrl-C is pressed, or the node is shutdown by the master.
81  */
82  ros::spin();
83
84  return 0;
85 }

```

## 2.2 The Code Explained

Now, let's break it down piece by piece, ignoring some pieces that have already been explained above.

Toggle line numbers

```

34 void chatterCallback(const std_msgs::String::ConstPtr& msg)
35 {
36     ROS_INFO("I heard: [%s]", msg->data.c_str());
37 }

```

This is the callback function that will get called when a new message has arrived on the `chatter` topic. The message is passed in a [boost shared\\_ptr](http://www.boost.org/doc/libs/1_37_0/libs/smart_ptr/shared_ptr.htm) ([http://www.boost.org/doc/libs/1\\_37\\_0/libs/smart\\_ptr/shared\\_ptr.htm](http://www.boost.org/doc/libs/1_37_0/libs/smart_ptr/shared_ptr.htm)), which means you can store it off if you want, without worrying about it getting deleted underneath you, and without copying the underlying data.

Toggle line numbers

```

75  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.

`NodeHandle::subscribe()` returns a `ros::Subscriber` object, that you must hold on to until you want to unsubscribe. When the `Subscriber` object is destructed, it will automatically unsubscribe from the `chatter` topic.

There are versions of the `NodeHandle::subscribe()` function which allow you to specify a class member function, or even anything callable by a `Boost.Function` object. The [roscpp overview \(/roscpp/Overview\)](#) contains more information.

Toggle line numbers

```

82  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.

There are other ways of pumping callbacks, but we won't worry about those here. The [roscpp\\_tutorials \(/roscpp\\_tutorials\)](#) package has some demo applications which demonstrate this. The [roscpp overview \(/roscpp/Overview\)](#) also contains more information.

Again, here's a condensed version of what's going on:

- Initialize the ROS system
- Subscribe to the `chatter` topic
- Spin, waiting for messages to arrive



- When a message arrives, the `chatterCallback()` function is called

### 3. Building your nodes

You used `catkin_create_pkg` (`/catkin/commands/catkin_create_pkg`) in a previous tutorial which created a `package.xml` (`/catkin/package_manifest`) and a `CMakeLists.txt` (`/catkin/CMakeLists.txt`) file for you.

The generated `CMakeLists.txt` (`/catkin/CMakeLists.txt`) should look like this (with modifications from the Creating Msgs and Srvs (`/ROS/Tutorials/CreatingMsgAndSrv`) tutorial and unused comments and examples removed):



[https://raw.githubusercontent.com/ros/catkin\\_tutorials/master/create\\_package\\_modified/catkin\\_ws/src/beginner\\_tutorials/CMakeLists.txt](https://raw.githubusercontent.com/ros/catkin_tutorials/master/create_package_modified/catkin_ws/src/beginner_tutorials/CMakeLists.txt)  
([https://raw.githubusercontent.com/ros/catkin\\_tutorials/master/create\\_package\\_modified/catkin\\_ws/src/beginner\\_tutorials/CMakeLists.txt](https://raw.githubusercontent.com/ros/catkin_tutorials/master/create_package_modified/catkin_ws/src/beginner_tutorials/CMakeLists.txt))

Toggle line numbers

```
1 cmake_minimum_required(VERSION 2.8.3)
2 project(beginner_tutorials)
3
4 ## Find catkin and any catkin packages
5 find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs genmsg)
6
7 ## Declare ROS messages and services
8 add_message_files(DIRECTORY msg FILES Num.msg)
9 add_service_files(DIRECTORY srv FILES AddTwoInts.srv)
10
11 ## Generate added messages and services
12 generate_messages(DEPENDENCIES std_msgs)
13
14 ## Declare a catkin package
15 catkin_package()
```

Don't worry about modifying the commented (`#`) examples, simply add these few lines to the bottom of your `CMakeLists.txt` (`/catkin/CMakeLists.txt`):

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

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

Your resulting `CMakeLists.txt` (`/catkin/CMakeLists.txt`) file should look like this:



[https://raw.githubusercontent.com/ros/catkin\\_tutorials/master/create\\_package\\_pubsub/catkin\\_ws/src/beginner\\_tutorials/CMakeLists.txt](https://raw.githubusercontent.com/ros/catkin_tutorials/master/create_package_pubsub/catkin_ws/src/beginner_tutorials/CMakeLists.txt)  
([https://raw.githubusercontent.com/ros/catkin\\_tutorials/master/create\\_package\\_pubsub/catkin\\_ws/src/beginner\\_tutorials/CMakeLists.txt](https://raw.githubusercontent.com/ros/catkin_tutorials/master/create_package_pubsub/catkin_ws/src/beginner_tutorials/CMakeLists.txt))

Toggle line numbers

```

1 cmake_minimum_required(VERSION 2.8.3)
2 project(beginner_tutorials)
3
4 ## Find catkin and any catkin packages
5 find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs genmsg)
6
7 ## Declare ROS messages and services
8 add_message_files(FILES Num.msg)
9 add_service_files(FILES AddTwoInts.srv)
10
11 ## Generate added messages and services
12 generate_messages(DEPENDENCIES std_msgs)
13
14 ## Declare a catkin package
15 catkin_package()
16
17 ## Build talker and listener
18 include_directories(include ${catkin_INCLUDE_DIRS})
19
20 add_executable(talker src/talker.cpp)
21 target_link_libraries(talker ${catkin_LIBRARIES})
22 add_dependencies(talker beginner_tutorials_generate_messages_cpp)
23
24 add_executable(listener src/listener.cpp)
25 target_link_libraries(listener ${catkin_LIBRARIES})
26 add_dependencies(listener beginner_tutorials_generate_messages_cpp)

```

This will create two executables, `talker` and `listener`, which by default will go into package directory of your devel space (`/catkin/workspaces#Development_28Devel.29_Space`), located by default at `~/catkin_ws/devel/lib/<package name>`.

Note that you have to add dependencies for the executable targets to message generation targets:

```
add_dependencies(talker beginner_tutorials_generate_messages_cpp)
```

This makes sure message headers of this package are generated before being used. If you use messages from other packages inside your catkin workspace, you need to add dependencies to their respective generation targets as well, because catkin builds all projects in parallel. As of \*Groovy\* you can use the following variable to depend on all necessary targets:

```
target_link_libraries(talker ${catkin_LIBRARIES})
```

You can invoke executables directly or you can use `roslaunch` to invoke them. They are not placed in '`<prefix>/bin`' because that would pollute the `PATH` when installing your package to the system. If you wish for your executable to be on the `PATH` at installation time, you can setup an install target, see: `catkin/CMakeLists.txt` (`/catkin/CMakeLists.txt`)

For more detailed discription of the `CMakeLists.txt` (`/catkin/CMakeLists.txt`) file see: `catkin/CMakeLists.txt` (`/catkin/CMakeLists.txt`)

Now run `catkin_make`:

```

# In your catkin workspace
$ cd ~/catkin_ws
$ catkin_make

```

Note: Or if you're adding as new pkg, you may need to tell catkin to force making by `--force-cmake` option. See [catkin/Tutorials/using\\_a\\_workspace#With\\_catkin\\_make \(/catkin/Tutorials/using\\_a\\_workspace#With\\_catkin\\_make\)](#).

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

## 4. Additional Resources

Here are some additional resources contributed by the community:

### 4.1 Video Tutorial

The following video presents a small tutorial explaining how to write and test a publisher and subscriber in ROS with C++ and Python based on the talker/listener example above

[CS460] ROS Tutorial 3: ROS Publishers and Subscriber...



Except where otherwise noted, the ROS wiki is licensed under the Creative Commons Attribution 3.0

Wiki: [ROS/Tutorials/WritingPublisherSubscriber\(c++\)](#) (last edited 2016-09-28 13:36:00 by [yoyekw \(/yoyekw\)](#))

(<http://creativecommons.org/licenses/by/3.0/>) | Find us on Google+ (<https://plus.google.com/113789706402978299308>)

Brought to you by:  Open Source Robotics Foundation

(<http://www.osrfoundation.org>)