ROS Concepts 1

ECE 4900/5900 Lecture Slides

Fall 2019

Instructor: Wing-Yue Geoffrey Louie

Summary and Quick Links

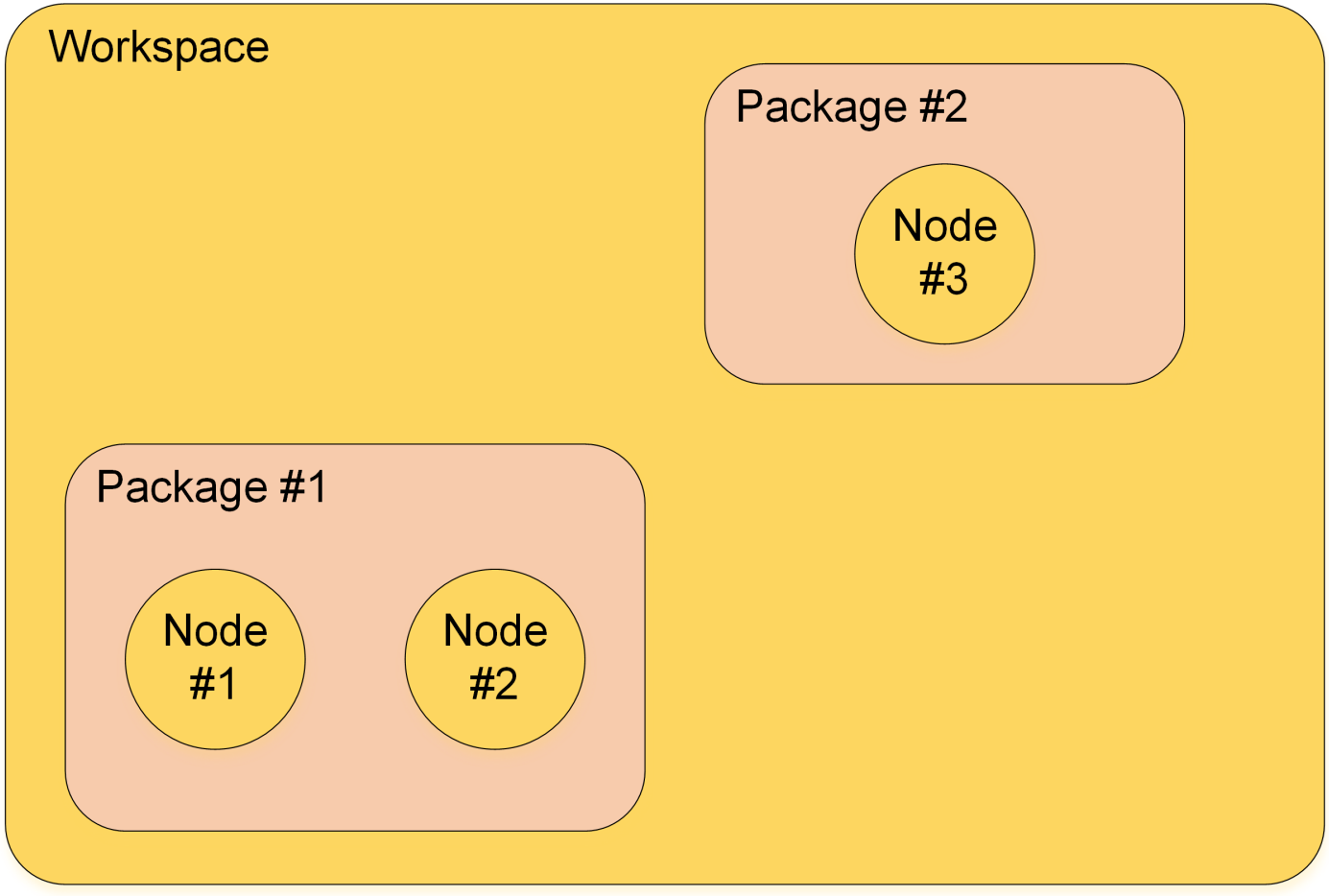
These slides contain the following concepts:

ROS Terminology

These will be discussed in more detail, but here is a quick overview of common terms in ROS:

- **Workspace** – Top-level entity where all components of a ROS system are contained.
- **Package** – A modular collection of ROS programs and libraries.
- **Node** – An independent program that executes code and interacts with the rest of the ROS environment.

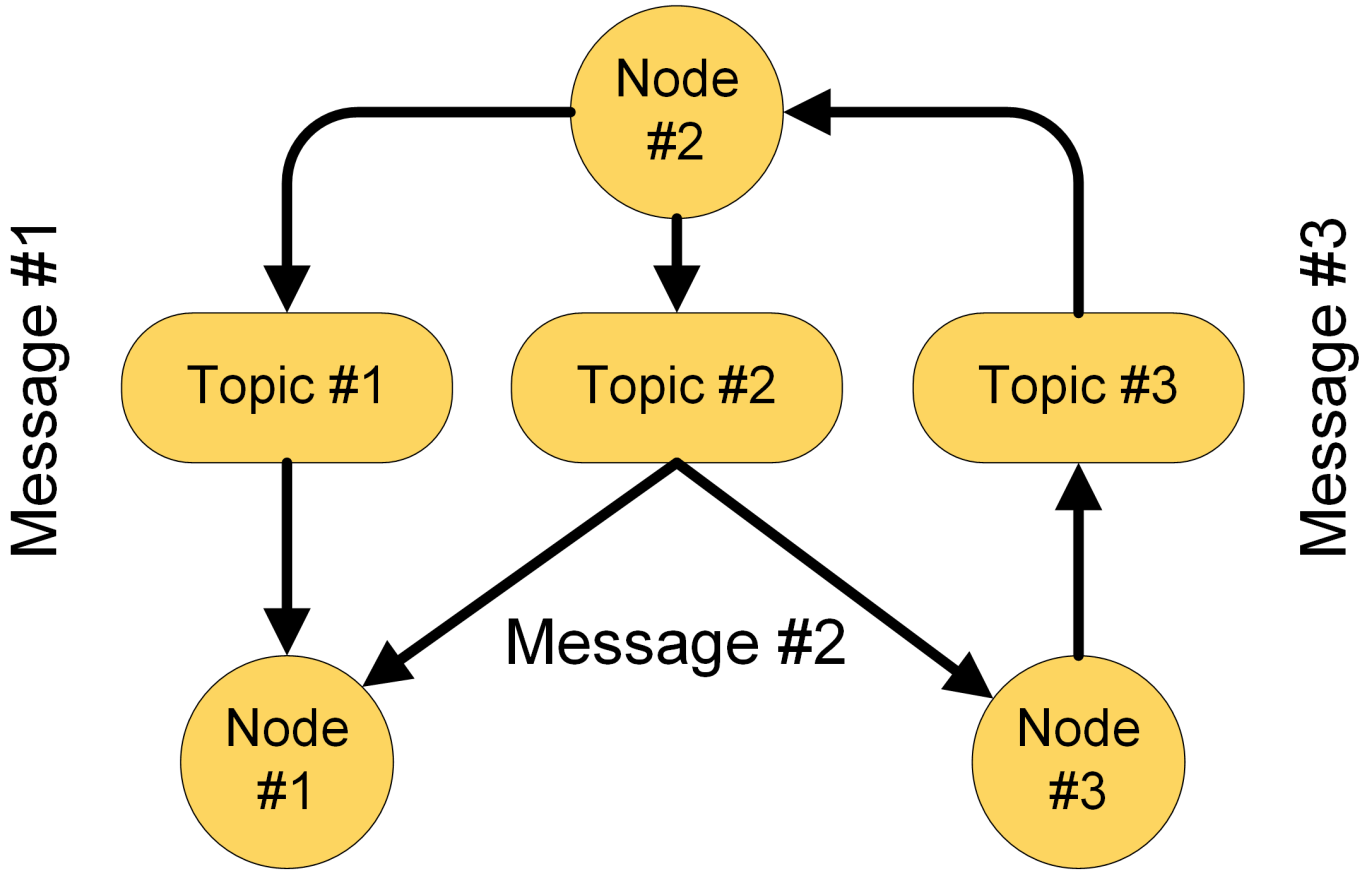
ROS Terminology



ROS Terminology

- **Topic** – A publisher/subscriber channel of communication between different ROS nodes.
- **Message** – Specific data that is transmitted over a topic.
- **Service** – A request/response channel of communication between different ROS nodes.

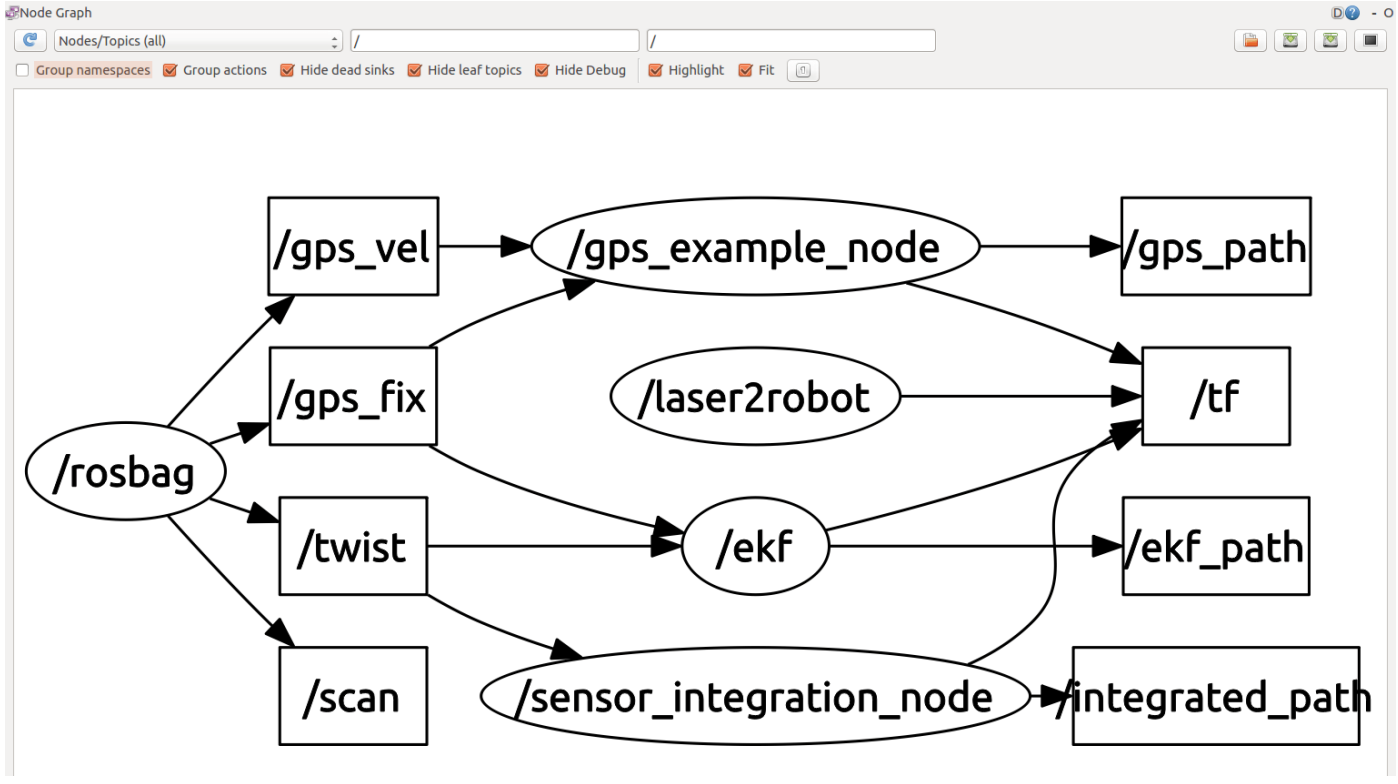
ROS Terminology



Node/Topic Visualization

- A handy tool to visualize the current state of ROS nodes and topics is the **rqt_graph**. On the command line, just type:

```
rqt_graph
```



Package Components

Every ROS package contains a **package.xml** file and a **CMakeLists.txt** file:

- **package.xml** — Describes the package by specifying which packages it depends on, among other things.
- **CMakeLists.txt** — Specifies input commands to CMake when it compiles the ROS workspace.

Example package.xml File

```
<?xml version="1.0"?>
<package format="2">
  <name>odom_pub</name>
  <version>0.0.0</version>
  <description>The odom_pub package</description>

  <maintainer email="micho@todo.todo">micho</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>
  <depend>geometry_msgs</depend>
  <depend>roscpp</depend>
  <depend>tf</depend>

</package>
```

package.xml

package.xml files typically have the following tags:

- **name** — Name of the package in the ROS system.
- **version**, **description**, **maintainer** and **license** — These tags are used for when the package is released into the ROS community, but do not affect the operation of the node.

package.xml

- **build_depend** — Specifies another ROS package that this package depends on during build time (header and library files).

- **run_depend** — Specifies a ROS package dependency whose files are accessed at runtime. Typically, each package dependency will have both a build dependency and a run dependency.

CMakeLists.txt

The contents of the **CMakeLists.txt** file depend on what is desired to be compiled, along with some mandatory components. The mandatory components are:

- Catkin version and project name declaration:

```
cmake_minimum_required(VERSION 2.8.3)
project(example_package)
```

- Find catkin dependencies:

```
find_package(catkin REQUIRED COMPONENTS
  other_package1
  other_package2
)
```

CMakeLists.txt

Mandatory components, cont.

- Include core ROS library directories in compilation process:

```
include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

- Catkin package declaration:

```
catkin_package()
```

CMakeLists.txt

- To compile a node for execution:

```
add_executable(node_name
  src/file1.cpp
  src/file2.cpp
)
```

- Link node to core ROS libraries:

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

Example CMakeLists.txt File

```
cmake_minimum_required(VERSION 2.8.3)
project(odom_pub)

# List other catkin package dependencies
find_package(catkin REQUIRED COMPONENTS
  geometry_msgs
  roscpp
  tf
)

# Include core ROS library directories
include_directories(${catkin_INCLUDE_DIRS})

# Declare catkin package
catkin_package()

# Compile an executable node
add_executable(odom_pub src/odom_pub_node.cpp)
target_link_libraries(odom_pub ${catkin_LIBRARIES})
```

Setting up a ROS Workspace

- In the home directory, create a new folder called **ros** with a subfolder called **src** inside it:

```
cd ~
mkdir -p ros/src
```

- Change to the **ros** folder and run **catkin_make**:

```
cd ~/ros
catkin_make
```

- Set up **.bashrc** to run the generated **setup.bash** script when opening a terminal:

```
echo "source ~/ros/devel/setup.bash" >> ~/.bashrc
```

Creating a New Package

- Change to the **src** folder within the ROS workspace:

```
cd ~/ros/src
```

- Create a directory for a new package:

```
mkdir example_package
```

- Inside the **example_package** folder, create **CMakeLists.txt** and **package.xml** with the contents on the next slides.

Creating a New Package

- **CMakeLists.txt**:

```
cmake_minimum_required(VERSION 2.8.3)
project(example_package)

find_package(catkin REQUIRED COMPONENTS
  roscpp
)

include_directories(
  ${catkin_INCLUDE_DIRS}
)

catkin_package()
```

Creating a New Package

- **package.xml**:

```
<?xml version="1.0"?>
<package format="2">
  <name>example_package</name>
  <version>0.0.0</version>
  <description></description>

  <maintainer email="abc@xyz.com">ABC</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>
  <depend>roscpp</depend>

</package>
```

Writing a Topic Publisher

The following slides illustrate how to write a node that does the following:

- Subscribe to a string topic.
- Concatenate "_123" onto the string.
- Publish the new string on a different topic.

Writing a Topic Publisher

- Create a **scripts** folder in the root of the **example_package** package and create a Python script file called **topic_publisher.py** in the folder.

- Every Python ROS Node will have this declaration at the top. The first line makes sure your script is executed as a Python script.

```
#!/usr/bin/env python
```

Writing a Topic Publisher

- imports and global variables:

```
import rospy
from std_msgs.msg import String

pub = rospy.Publisher('/topic_out', String, queue_size=10)
```

Writing a Topic Publisher

- Main function:

```
if __name__ == '__main__':
  try:
    listener()
  except rospy.ROSInterruptException:
    pass
```

- listener function:

```
def listener():
  rospy.init_node('listener', anonymous=True)
  rospy.Subscriber("/topic_in", String, callback)

  # spin() keeps python from exiting until node is stopped
  rospy.spin()
```

Writing a Topic Publisher

- Topic receive callback:

```
def callback(data):
        new_string = String()
        new_string = data.data + "_123"
```

Code Details

```
#!/usr/bin/env python

import rospy
from std_msgs.msg import String

pub = rospy.Publisher('/topic_out', String, queue_size=10)
```

- Here, we import the core ROS Python library headers found in **rospy** and the string message type.

- The **pub** publisher object needs to be global because we may want to access it from different functions.

Code Details

```
rospy.init_node('listener', anonymous=True)
```

- This code initializes the node in the ROS system.
- 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.

Code Details

```
rospy.Subscriber("/topic_in", String, callback)
```

This code declares a ROS subscriber object that uses the node handle to subscribe to a topic. The three arguments to the subscribe function are:

- Name of the topic being subscribed to.
- The type of message subscribed to.
- Name of the callback function.

Code Details

```
pub = rospy.Publisher('/topic_out', String, queue_size=10)
```

This code initializes the global ROS publisher object. The two arguments to the advertise function are:

- Name of the topic.
- The message type to be published on the topic
- Number of messages to buffer.

Code Details

```
rospy.spin()
```

- This command causes the node to loop forever, or at least until the user stops the program.
- While looping, the node processes all the callbacks that were initialized. In this case, this is just the subscription to the topic **topic_in**.

Code Details

```
def callback(data):
        new_string = String()
        new_string = data.data + "_123"
        pub.publish(new_string)
```

- This function is called whenever a new message is published on the **topic_in** topic from some other node.
- The received string is passed to the **callback** function as the **data** argument.
- A new **String()** message is declared, and its data is set to the received string with "_123" concatenated to it. It is then published using the publisher object.

Compiling the Node

- For ROS python it is not required to compile the python script
- Instead the script file must be made executable by changing the permissions of the file (i.e. using chmod)
- Open a terminal and change to the folder where the script is located: **cd ~/ros/src/example_package/scripts**
- Modify the permissions of the script: **chmod +x topic_publisher.py**

Compiling the Node

- Although Python does not need to be compiled we use CMake as our build system to make sure that the autogenerated Python code for messages and services are created.
- Open a terminal and change to the workspace root directory: **cd ~/ros**
- Run **catkin_make** to compile.

Running the Node

- First, start a ROS core by opening a terminal and typing:

```
roscore
```

- A ROS core manages all the nodes and handles all the messaging between them. There must always be exactly one core running in order for the system to function; no more, no less.

- In another new terminal, run the node by typing:

```
rosrun example_package topic_publisher
```

Running the Node

- Publish a string on the **topic_in** topic at 1 Hz. Open a new terminal and type:

```
rostopic pub /topic_in std_msgs/String hello -r 1.0
```

- Check the topic being published by **topic_publisher**. Open another terminal and type:

```
rostopic echo /topic_out
```

- The string message should be "hello_123".

Writing a Service Advertiser

The following slides illustrate how to write a node that advertises a service. This service will:

- Take two double precision floats as input (request).
- Add the two values together and respond with the result (response).

Creating a "srv" File

First, a service definition file must be created to define a floating point request, and a floating point response:

- Create a folder called **srv** in the root of the package.
- Create an empty text document called **Adder.srv** and type:

```
float64 val1
float64 val2
---
float64 result
```

- This file will be used to automatically generate a header file that defines the custom service.

Writing a Service Advertiser

- Create a Python script file called **service_advertiser.py** in the **scripts** folder.

- Imports and global variables:

```
from service_example.srv import Adder,AdderResponse
import rospy
```

- The name of the package that creates the srv file is where we should be importing the service message from (i.e. service_example)
- The name of the imported file matches the name of the corresponding **srv** file, including the capitalization.

Writing a Service Advertiser

- Main function:

```
if __name__ == "__main__":
    add_two_ints_server()
```

- add_two_ints_server function:

```
def add_two_ints_server():
    rospy.init_node('service_advertiser')
    srv = rospy.Service('adder_service', Adder, srv_callback)
    print "Ready to add two ints."
    rospy.spin()
```

Writing a Service Advertiser

- Service callback function:

```
def srv_callback(req):
    return AdderResponse(req.val1 + req.val2)
```

Code Details

```
srv = rospy.Service('adder_service', Adder, srv_callback)
```

This code initializes the service server and advertises it on the ROS system . The three arguments to the Service function are:

- Name of the advertised service.
- The service type
- Name of the callback function.

Code Details

```
def srv_callback(req):
    return AdderResponse(req.val1 + req.val2)
```

- This function is called whenever another entity in the ROS system calls the **Adder** service that this node is advertising.
- The request object is passed to the callback as an arguments.
- A response object is then initialized, which has a single field of **result**. The field is populated with the sum of the two fields of the request object.
- The function then returns the response object

Compiling the Node

- In order to compile the service advertiser node, the **CMakeLists.txt** file must be modified to use the **Adder.srv** file to define the service.

- In **CMakeLists.txt**, add the following before the **catkin_package()** line to tell catkin about the **srv** file:

```
add_service_files(
  FILES
  Adder.srv
)

generate_messages()
```

Compiling the Node

- Finally, navigate to the root of the workspace in a terminal and run **catkin_make** to compile.

- This should auto-generate python libraries for the Adder service using the srv files you created. These python libraries are located in the following folder in your workspace:
  devel/lib/python2.7/dist-packages/package_name/srv

Running the Node

- Assuming the compilation was successful, run the service advertiser node by:

```
rosrun example_package service_advertiser
```

- Test the service by calling it from the command line in a new terminal:

```
rosservice call /adder_service '{val1: 40.0, val2: 30.0}'
```

- You should see the correct result of 70 after running the above command.

Writing a Service Client

The following slides illustrate how to write a service client node. This node will:

- Instantiate a service client object.
- Call the service advertised by the service advertiser example.
- Stop the program without spinning.

Writing a Service Client

- Create a new Python script file called **service_client.py** in the **scripts** folder.

- Includes and global variables:

```
import sys
import rospy
from service_example.srv import *
```

Writing a Service Client

- Main function:

```
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))
```

Writing a Service Client

- add_two_ints_client function:

```
def add_two_ints_client(x, y):
    rospy.wait_for_service('adder_service')
    try:
        add_two_ints = rospy.ServiceProxy('adder_service', Adder)
        resp1 = add_two_ints(x, y)
        return resp1.result
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e
```

Code Details

```
add_two_ints = rospy.ServiceProxy('adder_service', Adder)
```

- This line declares a service client object and ROS Python libraries to connect to the desired service.
- The name of the service is passed as a the first argument into the **ServiceProxy** method of the node
- The type of the service is passed as the second argument

Code Details

```
def add_two_ints_client(x, y):
    rospy.wait_for_service('adder_service')
    try:
        add_two_ints = rospy.ServiceProxy('adder_service', Adder)
        resp1 = add_two_ints(x, y)
        return resp1.result
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e
```

- Wait for the node advertising the service.
- Declare service
- Call the service using the declared service client object and populate the response in **resp1**
- Return the result

Code Details

```
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))
```

- Obtain input arguments from command-line
- Print the request that is being made
- Call the function which runs the service and print output

Compiling the Node

- Just as with the other nodes, open a terminal and change to the workspace root directory: **cd ~/ros**
- Run **catkin_make** to compile the service messages.
- Modify the permissions of your scripts by running the following in the scripts folder:

```
chmod +x service_advertiser.py
```

```
chmod +x service_client.py
```

Running your Nodes

- Test your service advertiser and client nodes by running the following commands in different terminals:

```
roscore
```

```
rosrun service_example service_advertiser.py
```

```
rosrun service_example service_client.py 5 4
```

- You should get a result of 5+4 = 9