ROS Concepts 1

ECE 495/595 Lecture Slides

Winter 2017

Instructor: Micho Radovnikovich

Summary and Quick Links

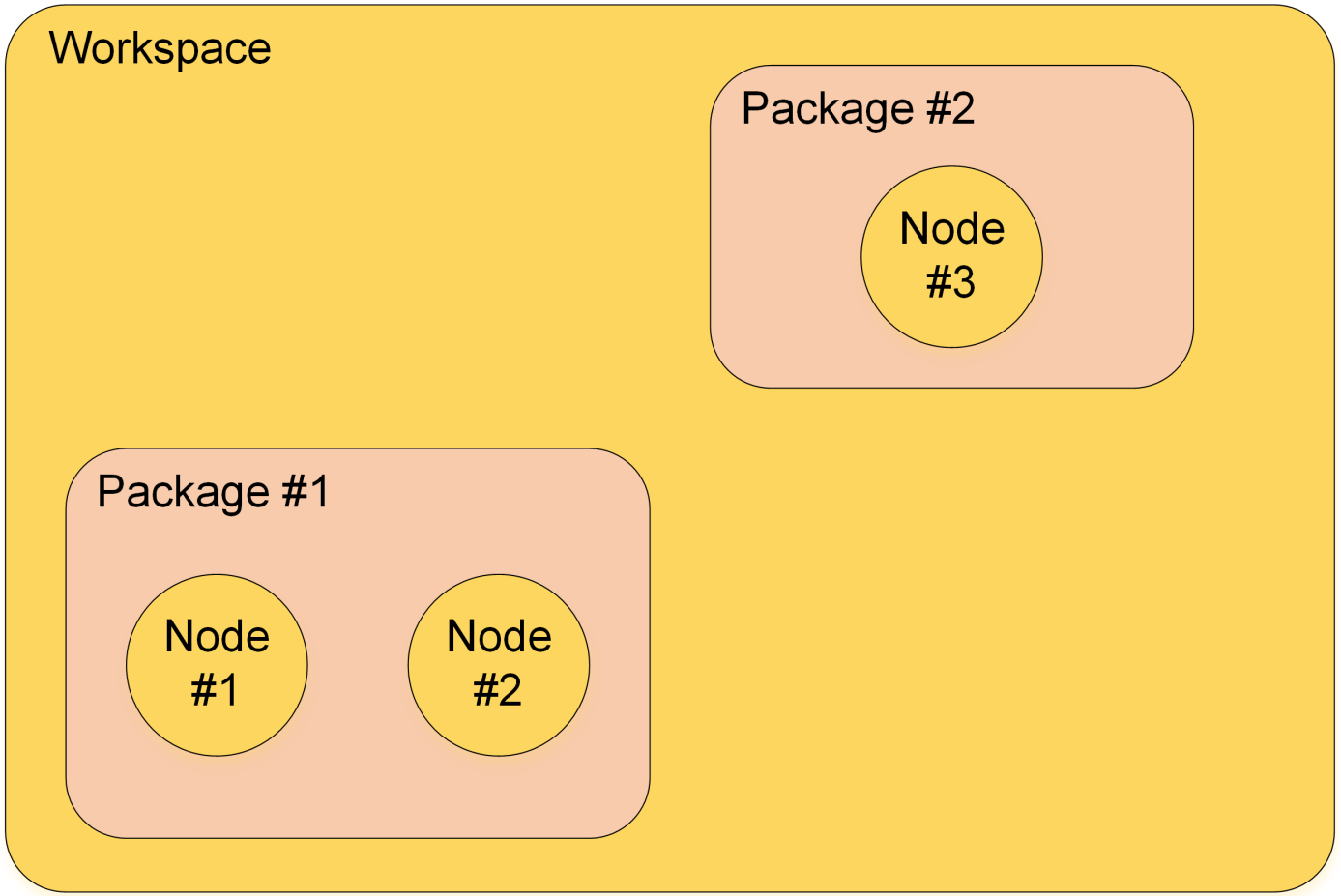
These slides contain the following concepts:

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.

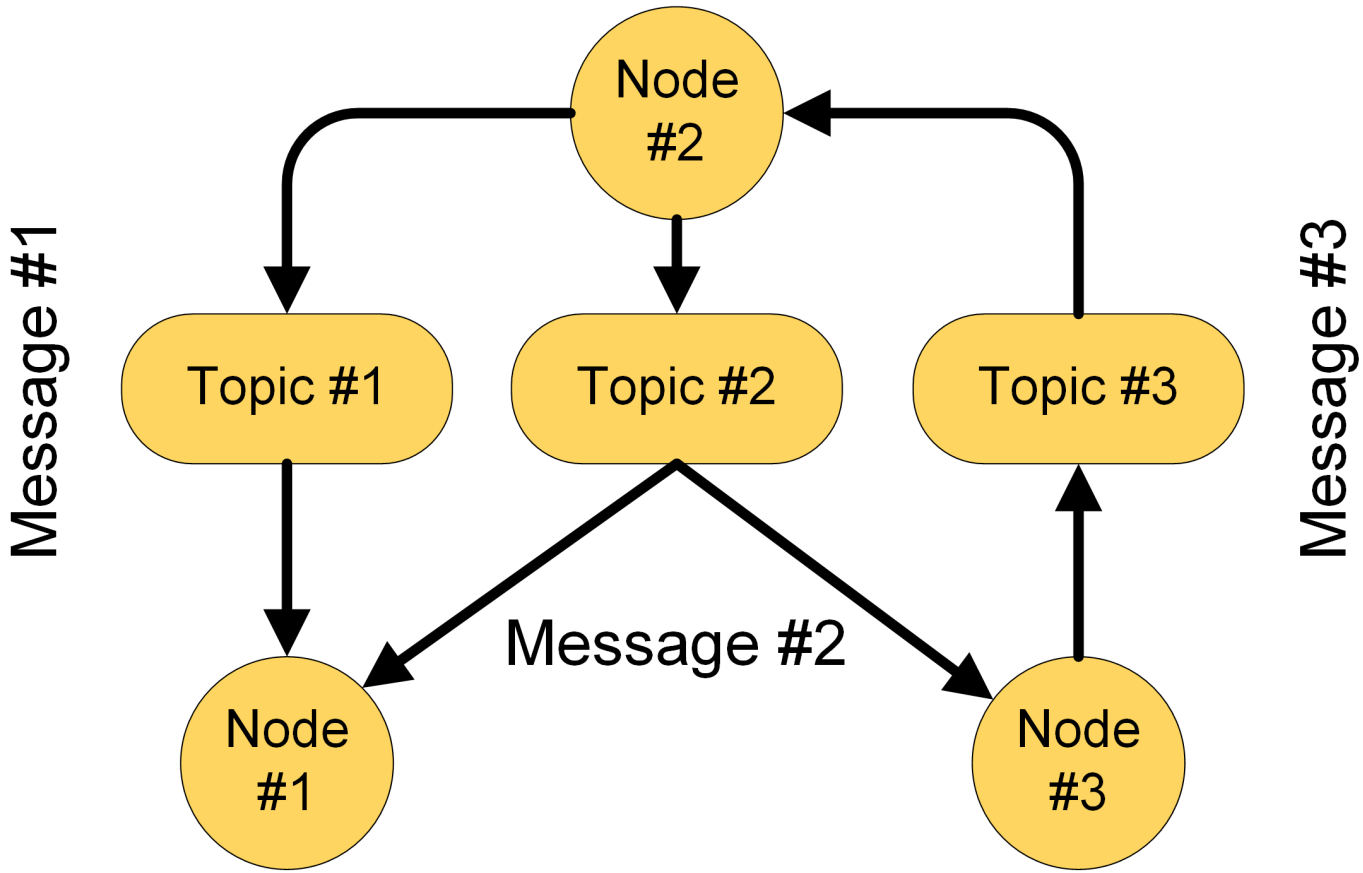
ROS Terminology



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.

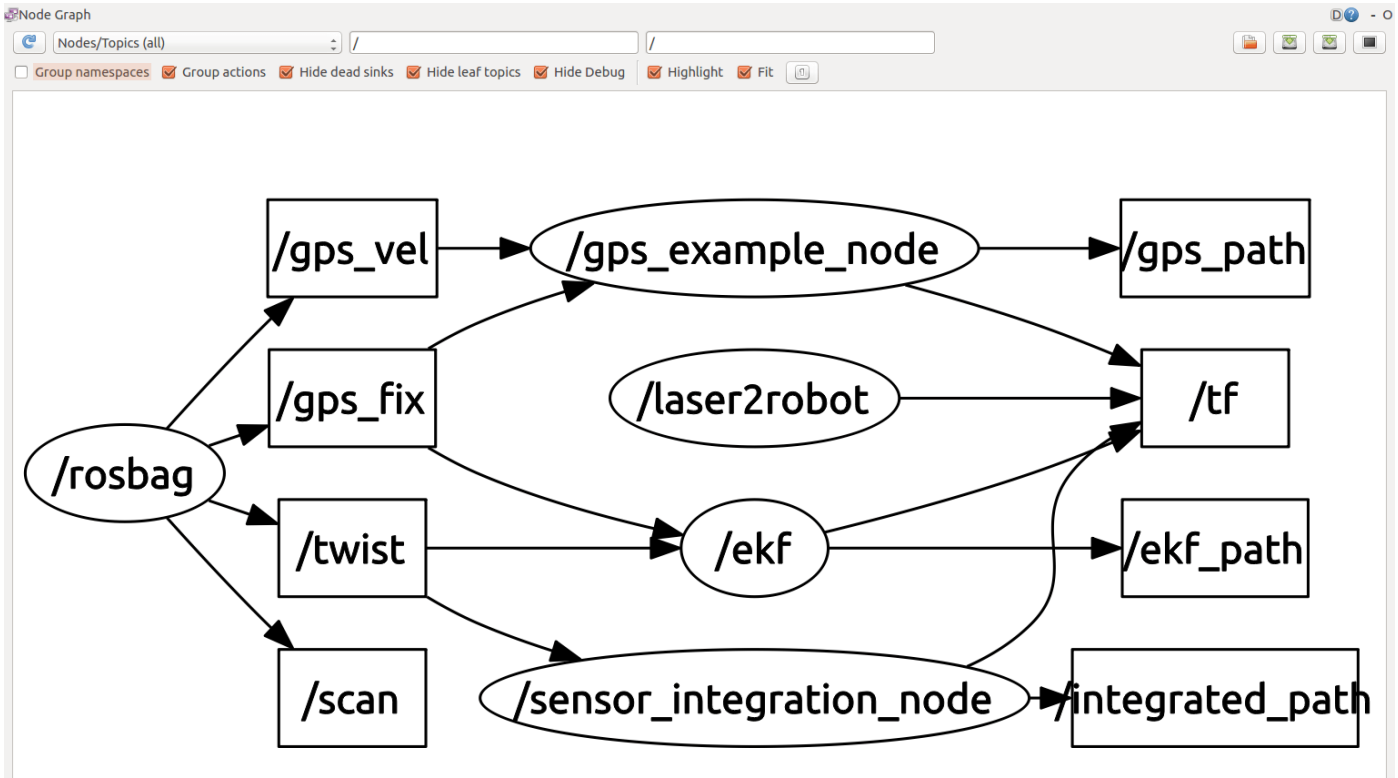
ROS Terminology



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>
  <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>
  <build_depend>geometry_msgs</build_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>tf</build_depend>
  <run_depend>geometry_msgs</run_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>tf</run_depend>

  <export>
  </export>
</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.
- *export* — This tag is used to provide capability for other packages to link to the locally compiled libraries.

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.

- Catkin package declaration:

```
catkin_package()
```

- 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
)

# 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
)

catkin_package()
```

Creating a New Package

- *package.xml*:

```
<?xml version="1.0"?>
<package>
  <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>
  <build_depend>roscpp</build_depend>
  <run_depend>roscpp</run_depend>

  <export>
  </export>
</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 C++ source file called *topic_publisher.cpp* in the *src* folder in the root of the *example_package* package.

- Includes and global variables:

```
#include <ros/ros.h>
#include <std_msgs/String.h>

ros::Publisher pub_string;
```

Writing a Topic Publisher

- Main function:

```
int main(int argc, char **argv) {
  ros::init(argc, argv, "topic_publisher");
  ros::NodeHandle node;

  ros::Subscriber sub_string = node.subscribe("/topic_in", 1,
                                              recvString);
  pub_string = node.advertise<std_msgs::String>("/topic_out", 1);

  ros::spin();
}
```

Writing a Topic Publisher

- Topic receive callback:

```
void recvString(const std_msgs::StringConstPtr& msg)
{
  std_msgs::String new_string;
  new_string.data = msg->data + "_123";
  pub_string.publish(new_string);
}
```

Code Details

```
#include <ros/ros.h>
#include <std_msgs/String.h>

ros::Publisher pub_string;
```

- Here, we include the core ROS library headers found in *ros/ros.h* and the string message header.

- The *pub_string* publisher object needs to be global because we access it from two different functions.

Code Details

```
ros::init(argc, argv, "topic_publisher");
ros::NodeHandle node;
```

- This code initializes the node in the ROS system and declares a NodeHandle object that is used to interact with the rest of the ROS system.

Code Details

```
ros::Subscriber sub_string = node.subscribe("/topic_in", 1,
                                            recvString);
```

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.
- Number of messages to buffer.
- Name of the callback function.

Code Details

```
pub_string = node.advertise<std_msgs::String>("/topic_out", 1);
```

This code initializes the global ROS publisher object which uses the node handle to advertise the output topic. The two arguments to the advertise function are:

- Name of the topic.
- Number of messages to buffer.

Code Details

```
ros::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

```
void recvString(const std_msgs::StringConstPtr& msg)
{
  std_msgs::String new_string;
  new_string.data = msg->data + "_123";
  pub_string.publish(new_string);
}
```

- This function is called whenever a new message is published on the *topic_in* topic from some other node.
- A pointer to the received string is passed to the *recvString* function as the *msg* argument.
- A new *std_msgs::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

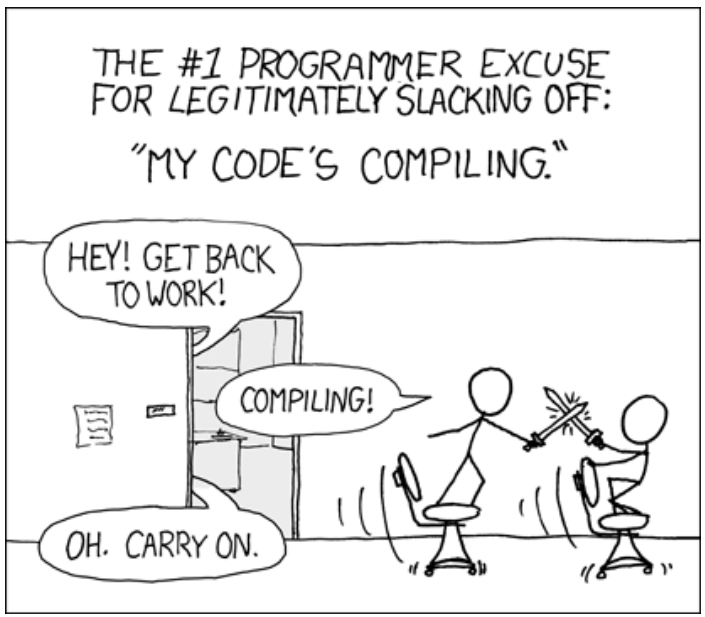
- In order to compile the node, the *CMakeLists.txt* file must be modified.
- Add the following lines to *CMakeLists* after the *catkin_package* line:

```
add_executable(topic_publisher src/topic_publisher.cpp)
target_link_libraries(topic_publisher ${catkin_LIBRARIES})
```

- The first line tells catkin to compile the program *simple_node* using the written source file.
- The second line links our program with the core libraries for the ROS components to work.

Compiling the Node

- After modifying CMakeLists, we can finally compile!
- Open a terminal and change to the workspace root directory: *cd ~/ros*
- Run *catkin_make* to compile.



http://xkcd.com/303/

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 simple_node
```

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 *simple_node*. 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 a double precision float as an input (request).
- Add 30 to the request 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 input
---
float64 result
```

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

Writing a Service Advertiser

- Create a C++ source file called *service_advertiser.cpp* in the *src* folder.

- Includes and global variables:

```
#include <ros/ros.h>
#include <example_package/Adder.h>
```

Writing a Service Advertiser

- Main function:

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "service_advertiser");
  ros::NodeHandle node;

  ros::ServiceServer srv = node.advertiseService("/adder_service",
                                                 srvCallback);

  ros::spin();
}
```

Writing a Service Advertiser

- Service callback function:

```
bool srvCallback(example_package::Adder::Request& req,
    example_package::Adder::Response& res)
{
  res.result = req.input + 30.0;
  return true;
}
```

Code Details

```
ros::ServiceServer srv = node.advertiseService("/adder_service",
                                               srvCallback);
```

This code initializes the service server and advertises it on the ROS system using the node handle. The two arguments to the advertiseService function are:

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

Code Details

```
bool srvCallback(example_package::Adder::Request& req,
    example_package::Adder::Response& res)
{
  res.result = req.input + 30.0;
  return true;
}
```

- This function is called whenever another entity in the ROS system calls the *adder* service that this node is advertising.
- The request and response objects are passed to the callback as arguments.
- The response object, which only consists of *result*, is populated with the input plus 30.
- The function then returns true to indicate successful completion.

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, as well as the *add_executable* line for the new node.

- 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(
  DEPENDENCIES
  std_msgs
)
```

Compiling the Node

- In *CMakeLists.txt*, add the following after the *catkin_package* line to generate the executable for the new service advertiser node:

```
add_executable(service_advertiser src/service_advertiser.cpp)
add_dependencies(service_advertiser ${PROJECT_NAME}_gencpp)
target_link_libraries(service_advertiser ${catkin_LIBRARIES})
```

- The new *add_dependencies* line is needed for the service advertiser node to link to the header file of the service.

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

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 '{input: {data: 40.0}}'
```

- 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.

Writing a Service Client

- Create a new C++ source file called *service_client.cpp* in the *src* folder.
- Includes and global variables:

```
#include <ros/ros.h>
#include <example_package/Adder.h>
```

Writing a Service Client

- Main function:

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "service_client");
  ros::NodeHandle node;

  ros::ServiceClient adder_srv =
      node.serviceClient<example_package::Adder>("adder_service");

  example_package::AdderRequest request;
  example_package::AdderResponse response;
  request.input = 5;

  bool success = add_numbers_srv.call(request, response);

  return 0;
}
```

Code Details

```
ros::ServiceClient adder_srv =
    node.serviceClient<example_package::Adder>("adder_service");
```

- This line declares a service client object and uses the node handle to connect to the desired service.
- The type of the service is inserted into the angle brackets.
- The name of the service is passed as a string argument into the *serviceClient* method of the node handle.

Code Details

```
example_package::AdderRequest request;
example_package::AdderResponse response;
request.input = 5;

bool success = add_numbers_srv.call(request, response);
```

- Declare request and response structures.
- Populate the request structure with the input data.
- Call the service using the declared service client object.
- The service call function returns a boolean indicating success or failure.