Robotics Lab - Lecture 3

Starting with ROS programming (Part 1)

Jonathan Cacace

March 17, 2020

Contents

Preface

This manuscript contains the lecture notes for the *Robotics Lab* class taken at University of Naples Federico II for the students of automation engineering master degree. The aim of this course is to give an overview of the fundamental tools and techniques used to program advanced robotics systems (both industrial and mobile). After a brief introduction of the technologies commonly used to program robots (e.g. Linux, c++, git), the Robot Operating System (ROS) framework will be introduced and deeply studied. Simulation software will help the course attenders to test state-of-art robotic algorithms and their own robot control software.

In this lesson, we discuss two ROS packages implementing the communication protocols available in ROS: the publish-subscribe ad the service. For each package, we also show some useful command line tool used in ROS to handle the execution of ROS nodes.

1

Starting with ROS programming (Part 1)

1.1 Environment configuration

You need to install ROS in your system before to start programming with it. In our lessons we use ROS `Melodic`, however similar steps can be follow to install other ROS distributions.

The first step is the setup your computer to accept software from `packages.ros.org`.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

$ sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80'
--recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Then you should update your APT repository:

```
$ sudo apt-get update
```

Finally, you are ready to install ROS. There are many different libraries and tools in ROS. The full version of ROS includes all the packages commonly used in robotic programming and can be installed with the following command:

```
$ sudo apt-get install ros-melodic-desktop-full
```

This step will take some time. Moreover, the latter command doesn't install all the packages available in ROS. To find additional packages, use:

```
$ apt-cache search ros-melodic
```

Now ROS is installed in your system. However, you are not able to used ROS command yet. In fact, at this point if you try to run a ROS command like `roscd` in your linux shell, you will get the following error:

```
$ Command 'roscd' not found, did you mean:

$ command 'rosco' from deb python-rosinstall

$ Try: sudo apt install <deb name>
```

This happen because the ROS environment is not correctly configured. By default, ROS in installed in the following directory:

```
$ /opt/ros/${ROS_VERSION}
```

To properly load the environment you have to `source` the setup file included in the installation directory of ROS.

```
$ source /opt/ros/melodic/setup.bash
```

Now you should be able to use the ROS commands. However, with this setup the user workspace is set to a directory owned by the super user (i.e. `/opt/ros/melodic/share`). So, before to continue, you should create your own workspace in the user space:

```
$ cd ~
$ mkdir -p ros_ws/src
$ cd ros_ws/src
$ catkin_init_workspace
$ cd ..
$ catkin_make
```

The `catkin_make` is the compilation command. Now you have created a ROS workspace called `ros_ws`. If you check the content of this directory, it contains three folders: `build`, `devel` and `src`. In the `src` directory you must create or download new ROS packages. If a package is not placed there, it will not be compiled. The `build` directory instead contains the compilation file, while the `devel` folder contains the compiled libraries.

To set `ros_ws` workspace as the default workspace you need to source the `setup.bash` file contained in the `devel` folder. To source this file automatically when a new linux shell is opened, you could be put the source command `bashrc` file:

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

To test if everything is properly configure, you could try to enter the `roscd` command in a linux shell. If everything is right, this command will move you in the `devel` folder of your ROS workspace.

1.2 Create a ROS package

As already stated, all ROS packages, either created from scratch or downloaded from other code repositories, must be placed in the src folder of the ROS workspace, otherwise they can not be recognized by the ROS system and compiled.

To create a ROS package, switch to the catkin workspace `src` folder and create the package, using the following command:

```
$ catkin_create_pkg package_name [dependency1] [dependency2]
```

Try to create a simple node implementing the publish/subscribe communication protocol. Call this package: `ros_topic`:

```
$ catkin_create_pkg ros_topic roscpp std_msgs
```

As dependencies, we specified the following:

- `roscpp`: This is the C++ implementation of ROS. It is a ROS client library which provides APIs to C++ developers to make ROS nodes with ROS topics, services, parameters, and so on. We are including this dependency because we are going to write a ROS C++ node. Any ROS package which uses the C++ node must add this dependency.
- `std_msgs`: This package contains basic ROS primitive data types, such as integer, float, string, array, and so on. We can directly use these data types in our nodes without defining a new ROS message.

After ran this command, a new directory appears in your ROS workspace. A typical structure of an ROS package is shown in Fig. 1.1.

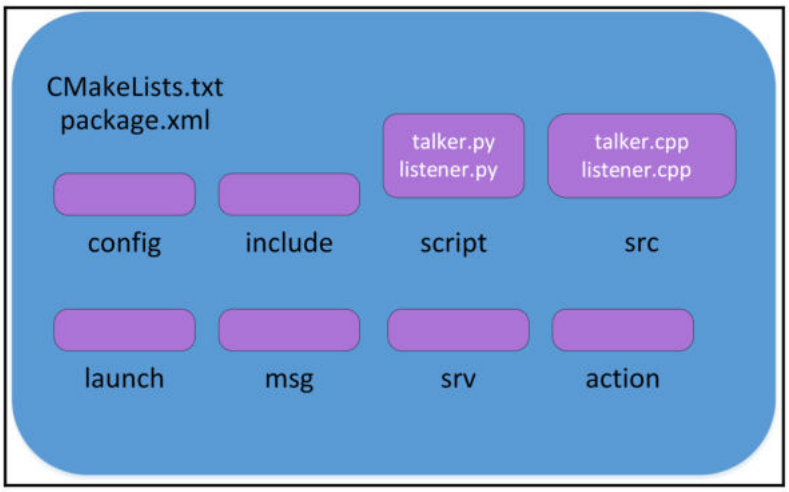

Figure 1.1: Structure of a typical ROS package

- `config`: All configuration files that are used in this ROS package are kept in this folder. This folder is created by the user and it is a common practice to name the folder config to keep the configuration files in it.
- `include/package_name`: This folder consists of headers and libraries that we need to use inside the package.
- `script`: This folder keeps executable Python scripts. In the block diagram, we can see two example scripts.
- `src`: This folder stores the C++ source codes.
- `launch`: This folder keeps the launch files that are used to launch one or more ROS nodes.
- `msg`: This folder contains custom message definitions.
- `srv`: This folder contains the services definitions.
- `action`: This folder contains the action files.
- `package.xml`: This is the package manifest file of this package. In particular, this file defines properties about the package such as the package name, version numbers, authors, maintainers, and dependencies on other catkin packages.
- `CMakeLists.txt`: This files contains the directives to compile the package.

After that the `ros_topic` has been created, you should be able to use the first ROS command introduced here, the `roscd`. In particular, `roscd` command is used to change the current directory using a package name or a special location. If we give the argument a package name, it will switch to that package folder (i.e. `~/ros_ws/devel`). Consider that after created or downloaded a new package in your workspace, you should inform the ROS system about it, updating the ROS filesystem using the command:

```
$ rospack profile
```

Now you can try to reach the ROS package folder using the following command:

```
$ roscd ros_topic
```

If everything is working properly, you will be move in the directory of the package:

```
$ ~/ros_ws/src/ros_topic
```

After that the ROS package has been successfully created, we can start adding nodes to it. A ROS package can contain multiple ROS nodes. We will create two nodes, one to publish a topic and one to subscribe to a topic. Let's start with the publisher one.

To create a new node in the `ros_topic` package, move in the `src` directory of the package and create an empty source file:

```
$ roscd ros_topic/src
$ touch ros_publisher.cpp
```

The aim of this node is to publish an integer value on a topic called `/numbers`.

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>

int main(int argc, char **argv) {
  ros::init(argc, argv,"ros_topic_publisher");
  ros::NodeHandle nh;
  ros::Publisher topic_pub =
  nh.advertise<std_msgs::Int32>("/numbers",10);
  ros::Rate rate(10);
  int count = 0;
  while (ros::ok()) {
    std_msgs::Int32 msg;
    msg.data = count++;
    ROS_INFO("%d",msg.data);
    topic_pub.publish(msg);
    rate.sleep();
  }
  return 0;
}
```

In the above code, we firstly include the header files needed to use ROS api (`roscpp`): the `ros/ros.h` is the main header of ROS, while the `std_msgs/Int32.h` is the standard message definition of the integer datatype. In order to initialize a ROS node with a given name, we used the following code line:

```
  ros::init(argc, argv,"ros_topic_publisher");
```

This line is mandatory for all ROS nodes, otherwise will be impossible to use all ROS api functions. The name provided in the `init` function should be unique and will be used by the ROS master to handle it. Each ROS node typically has a `NodeHandle`, an object used to communicate with the whole ROS system. To declare it we use the following line of code:

```
ros::NodeHandle nh;
```

Then, we can create the object representing the topic publisher:

```
ros::Publisher topic_pub = nh.advertise<std_msgs::Int32>("/numbers", 10);
```

This will create a topic publisher and name the topic `/numbers` with a message type `std_msgs::Int32`. The second argument is the buffer size.

It indicates how many messages need to be put in a buffer before sending. It should be set to high if the data sending rate is high is used to set the frequency of sending data.

Another important feature of ROS is represented the `ros::Rate` object. This object is used to run loops at a desired frequency. Note that the `Rate` takes into account the elapsed time between the end of the previous loop and the new loop. When you create the `Rate` object you specify also the desired loop rate (in `Hetz`). To create an infinite while loop, we exploit the `ros::ok()` function, that returns zero when Ctrl+C is pressed. The message that we want to publish is a `std_msgs::Int32`, so, after declared it we fill its data field. How is composed a `std_msgs::Int32`? If you want to know how a message is composed, you can used the `rosmsg` command. The inbuilt tools called `rosmsg` is used to get information about ROS messages. Here are some parameters used along with `rosmsg`:

```
$ rosmsg show [message]: This shows the message description
$ rosmsg list: This lists all messages installed in your system
$ rosmsg md5 [message] : This displays md5sum of a message
$ rosmsg package [package_name] : This lists messages
in a package
```

So, considering our initial aim, to show how is composed the `std_msgs::Int32` message, we can use the `rosmsg show` command. So open a new terminal and insert the following command:

```
$ rosmsg show std_msgs/Int32

Output:
$ int32 data
```

So to fill the contents of `std_msgs::Int32` message, you need to refer to the data field.

To publish the message to the `/numbers` topic we use the method `publish`, who takes as input the message to broadcast on your topic.

Compile and run a ROS node

To compile a ROS package you need to edit its `CMakeLists.txt` file. In particular, we have to inform the building tool about what source file must be compiled and its dependencies. To compile the ROS publisher node, add the following lines at the end of the `CMakeLists.txt`:

```
1 #This will create executables of the nodes
2 add_executable(topic_publisher src/ros_publisher.cpp)
3
4 #This will link executables to the appropriate libraries
5 target_link_libraries(topic_publisher ${catkin_LIBRARIES})
```

At this point, you can use the `catkin_make` command to build the package. We can first switch to a workspace:

```
$ cd ~/ros_ws
```

Build `ros_topic` package as follows:

```
$ catkin_make
```

Consider that the `catkin_make` command compiles all the package in your workspace. Sometimes, you could have draft versions of other package that bring to compilation errors or unsatisfied dependencies. In this way the compilation could fail. To compile only one package and not the entire workspace you can use the `DCATKIN_WHITELIST_PACKAGES` argument. With this option, it is possible to set one or more packages enabled to be compiled.

```
$ catkin_make -DCATKIN_WHITELIST_PACKAGES="pkg1,pkg2,..."
```

Note that is necessary to revert this configuration to compile other packages not specified in the `WHITELIST`. In fact, after set the `WHITELIST` it will remain saved in your system configuration, and you can directly execute the `catkin_make` command to compile only the packages specified in the `WHITELIST`. Differently, to bring again the list to the initial configuration you can use the following command:

```
$ catkin_make -DCATKIN_WHITELIST_PACKAGES=""
```

Now you are ready to run the publisher node. First of all, to execute ROS nodes you must activate a `roscore` in your system. To do this, on a linux terminal, run the following command:

```
$ roscore
```

This command runs the ROS master node on your local machine. Consider that this command locks your terminal, so to run other commands you need to open other linux shells.

To run the publisher node, you can use the `rosrun` command:

```
$ rosrun ros_topic topic_publisher
```

The output on the linux shell shows the `INFO` about the integer that are going to be published on `/numbers` topic. We can use two additionally commands to debug and understand the working of the nodes: `rosnode` and `rostopic`.

```
$ rosnode info [node_name]: This will print the
information about the node
$ rosnode kill [node_name]: This will kill a
running node
$ rosnode list: This will list the running nodes
```

```
$ rosnode machine [machine_name] : This will list
the nodes running on a particular
machine or a list of machines
$ rosnode ping: This will check the connectivity of a node
$ rosnode cleanup: This will purge
the registration of unreachable nodes
```

In particular, the output of `rosnode info /ros_topic_publisher` will provide information about the published and subscribed topics:

```
Node [/ros_topic_publisher]
Publications:
* /numbers [std_msgs/Int32]
* /rosout [rosgraph_msgs/Log]

Subscriptions: None

Services:
* /ros_topic_publisher/get_loggers
* /ros_topic_publisher/set_logger_level

contacting node http://jcacace-Inspiron-7570:43001/ ...
Pid: 19478
Connections:
* topic: /rosout
* to: /rosout
* direction: outbound
* transport: TCPROS
```

This is useful to understand the input and output of a node. As for the `rostopic` command, it can be used to get information about ROS topics. Here is the syntax of this command:

```
$ rostopic bw /topic: This command will display
the bandwidth used by the given topic.
$ rostopic echo /topic: This command will
print the content of the given topic in a human
readable format. Users can use the "-p"
option to print data in a csv format.
$ rostopic find /message_type: This command will
find topics using the given message type.
$ rostopic hz /topic: This command will display
the publishing rate of the given topic.
$ rostopic info /topic: This command will print
information about an active topic.
```

```
$ rostopic list: This command will list all active
topics in the ROS system.
$ rostopic pub /topic message_type args: This
command can be used to publish a value to a
topic with a message type.
$ rostopic type /topic: This will display the
message type of the given topic.
```

We can use these commands on the output of our publisher node. In particular, check which kind of topics are published by the node:

```
$ rostopic list
```

```
Output:
/numbers
/rosout
/rosout_agg
```

And check its content:

```
$ rostopic echo /numbers
```

```
Output:
data: 609
---
data: 610
---
data: 611
```

Before to discuss the subscriber node, let's introduce additional ROS graphical tools. In the first versions of ROS only few graphical tools were considered to plot data or display the connections between ROS nodes. In 2012, the first version of ROS including the `rqt` graphical interface has been released.

`rqt` is a software framework that implements the various GUI tools in the form of plugins. One can run all the existing GUI tools as dockable windows within rqt. The tools can still run in a traditional standalone method, but rqt makes it easier to manage all the various windows on the screen at one moment. To start `rqt` just type this command in your linux shell:

```
$ rqt
```

This command will open a new windows, as depicted in Fig. 1.2. In the `rqt` start window you can load any desired plugin present in your system. You can also add custom plugins. Try to use the `Topic Monitor` plugin to inspect the data published on the `/numbers` topic. Open the plugin,

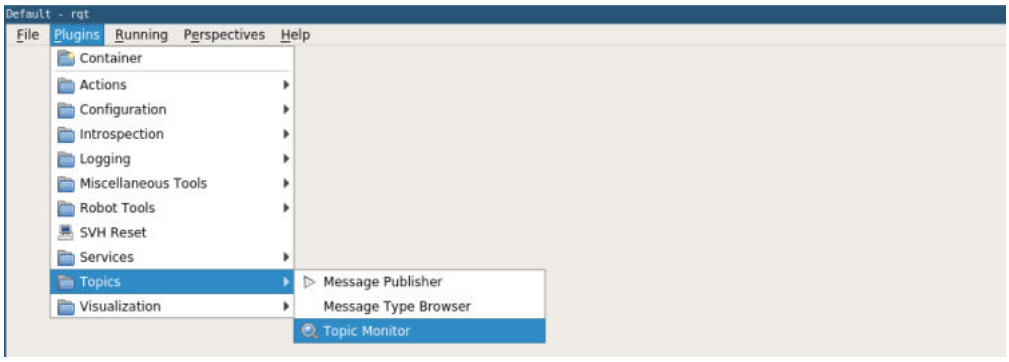

Figure 1.2: rqt window.

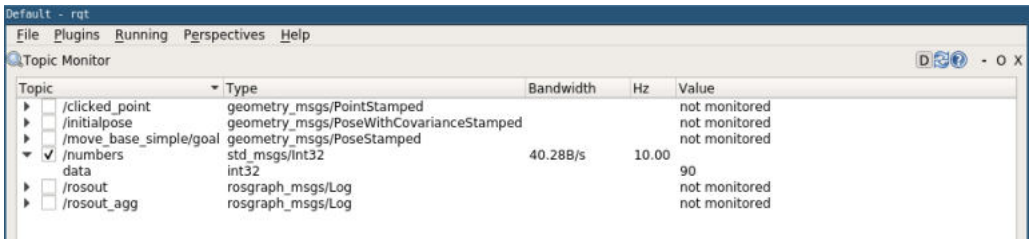

Figure 1.3: Topic monitor plugin.

and check the checkbox on this topic, as shown in Fig 1.3. As you can see, we obtained the same result of `rostopic echo` command, but in a simpler/another way.

ROS Subscriber

After ran the publisher node, you can create a subscriber node that use the data published on `/numbers` topic.

Let's now create a new source code file called `ros_subscriber.cpp`.

```
$ roscd ros_topic/src
$ touch ros_subscriber.cpp
```

A sample code to read the `std_msgs::Int32` data is here reported.

```
1 #include "ros/ros.h"
2 #include "std_msgs/Int32.h"
3 #include <iostream>
4
5 class ROS_SUB {
6
7   public:
8     ROS_SUB();
9     void topic_cb( std_msgs::Int32ConstPtr data);
10  private:
11    ros::NodeHandle _nh;
```

```
12      ros::Subscriber _topic_sub;
13 };
14
15 ROS_SUB::ROS_SUB() {
16   _topic_sub = _nh.subscribe
17   ("/numbers", 0, &ROS_SUB::topic_cb, this);
18
19 }
20
21 void ROS_SUB::topic_cb( std_msgs::Int32ConstPtr data) {
22   ROS_INFO("Listener: %d", data.data);
23
24 }
25
26 int main( int argc, char** argv ) {
27   ros::init(argc, argv, "ros_subscriber");
28   ROS_SUB rs;
29   ros::spin();
30   return 0;
31 }
```

Differently from the publisher node, in this example we use a class called `ROS_SUB`. In the constructor of the class, we declare a subscriber for the `std_msgs::Int32` data:

```
1 _topic_sub = _nh.subscribe ("/numbers", 0, &ROS_SUB::topic_cb, this);
```

In this line of code, we have to specify the name of the topic to read, the buffer and the callback function that receives the data. When you use class methods as subscribers, you must specify the class in were the method belong (`&ROS_SUB::topic_cb`) but also its context. In this case, we used `this`, which means that the subscriber will refer to the class it is part of. To update ROS topics the `ros::spin()` function is used. In particular, we have two functions that let all the callbacks get called for your subscriber: `ros::spin()` and `ros::spinOnce()`. The main differences between these two functions is that the first one is blocking function. The code after the `spin()` will never be executed. In addition, it will implement an infinite loop that makes your program alive over time.

As shown in the previous example, you can now modify the `CMakeLists.txt` file to add this new node (executable) and compile it with the `catkin_make` command. Now, you can launch both the nodes, the publisher and subscriber. To do this, type the following commands on three different linux shells:

```
$ roscore
$ rosrun ros_topic topic_publisher
$ rosrun ros_topic topic_subsciber
```

Now, you can use also the `rostopic` command to have more information about the connection between the publisher and subscriber.

```
rostopic info /numbers

Output:
Type: std_msgs/Int32

Publishers:
* /ros_topic_publisher (http://jcacace-Inspiron-7570:41703/)

Subscribers:
* /ros_subscriber (http://jcacace-Inspiron-7570:34901/)
```

This command provide information about the type of the message (`std_msgs::Int32`), but also the publisher (`ros_topic_publisher`) and the subscribers (`ros_subscriber`) active in the your ROS system.

Sometimes you just need to publish some data used to test a subscriber node (or send desired data like commanded velocity of similar). Of course, in such context implement a ROS node from scratch could be a waste of time. For this reason, we can directly publish a data using the command line tool:

```
$ rostopic pub /numbers std_msgs::Int32 "data: 13" -r 10
```

This command publishes the number 13 on the topic `numbers` with a publishing rate of 10 Hz.

Again, try to use the graphical tools of ROS to obtain the same results of `rostopic pub` command. We can use the `rqt_publisher` plugin of the `rqt` interface. In this case, try to directly load this plugin from the command line.

```
$ rosrun rqt_publisher rqt_publisher
```

This command will open the `rqt` interface with the `rqt_publisher` plugin. As shown in Figs. 1.4 and 1.5, from this window you can select and add a topic to publish, specify its value and publishing rate and finally publish the desired value.

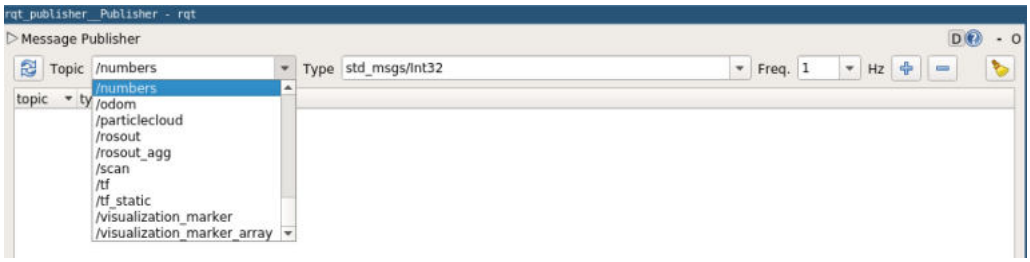

Figure 1.4: rqt publisher plugin.

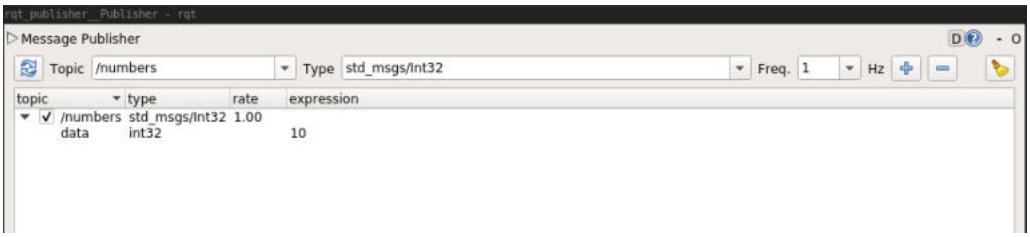

Figure 1.5: Publish on /numbers topic using rqt_publisher plugin.

1.3 ROS Service

In this section, we are going to create a new ROS package to implement ROS service protocol. The service nodes we are going to create can send a string message as a request to the server and the server node will send another message as a response. Differently from the example of previous section, in which the ROS publisher used a standard message already present in the installation of ROS (the `std_msgs::Int32`), in this case we have to define the service message exchanged between the client and the server. Let's start creating a new ROS package called `ros_service`

```
$ roscd && cd ..
$ cd src
$ catkin_create_pkg ros_service roscpp std_msgs
message_generation message_runtime
```

We add two additionally dependencies for this package, the `message_generation` and `message_runtime`, packages used to handle the building and run-time usage of custom messages.

Before to create the source code of the ROS nodes, let's add a custom `service` message. Create a new folder called `srv` in the package directory and add a `srv` file called `service.srv`. The definition of this file is as follows:

```
string in
---
string out
```

In this case, both the Request and Response filed of the service are strings. To use this service, we need to compile it. For this reason, you have to uncomment the following lines of the `CMakeLists.txt` file as shown here:

```
## Generate services in the 'srv' folder
add_service_files(
FILES
service.srv
)
```

and

```
generate_messages(
DEPENDENCIES
std_msgs
)
```

After making these changes, we can build the package using `catkin_make` and using the following command, we can verify the procedure:

```
$ rossrv show ros_service/service
```

If we see the same content as we defined in the file, we can confirm it's working.

Now, let's create the service server and client. Move in the `src` folder of the `ros_service` package and create a new source file:

```
$ roscd ros_service/src
$ touch service_server.cpp
```

The content of the server is listed below:

```
#include "ros/ros.h"
#include "ros_service/service.h"
#include <iostream>
#include <sstream>

using namespace std;

bool service_callback
(ros_service::service::Request &req, ros_service::service::Response &res) {
  std::stringstream ss;
  ss << "Received Here";
  res.out = ss.str();
  ROS_INFO("From Client [%s], Server says
  [%s]",req.in.c_str(),res.out.c_str());
  return true;
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "service_server");
  ros::NodeHandle n;
  ros::ServiceServer service = n.advertiseService("service", service_callback);
  ROS_INFO("Ready to receive from client.");
  ros::spin();
  return 0;
}
```

The `ros_service/service.h` header is a generated header, which contains our service definition and we can use this in our code. The server callback function is executed when a request is received on the server. The server can receive the request from clients with a message type of

`ros_service::service::Request` and sends the response in the `ros_service::service::Response` type. Finally, in order to offer the service, we need to include this line of code:

```
ros::ServiceServer service =
n.advertiseService("service", service_callback);
```

This code line instantiates a service called `service`, while the callback for it is a function called `service_callback`.

Like in the previous example, start this service and try to handle it using the commands provided by the ROS framework. Just add the compilation directives in the `CMakeLists.txt` file:

```
1 add_executable(service_server  src/service_server.cpp)
2 target_link_libraries(service_server ${catkin_LIBRARIES} )
```

Then, compile with `catkin_make` command:

```
$ roscd
$ cd ..
$ catkin_make
```

Finally, after ran a `roscore`, start the service:

```
$ roscore
$ rosrun ros_service service_server
```

At this point, we can check that the service instantiated in this node is active. Use the following command to inspect the service active in your ROS system:

```
$ rosservice list

Output:
/rosout/get_loggers
/rosout/set_logger_level
/service
/service_server/get_loggers
/service_server/set_logger_level
```

You can also call this service using the following command:

```
$ rosservice call /service "in_: 'Call'"

out: "Received Here"
```

The command `rosservice call` takes into account the name of the service to call and the list of arguments. In this case, it takes a string.

Let's now create the service client node.

```
$ roscd ros_service/src
$ touch service_client.cpp
```

The following code calls the service declared in the previous example:

```
#include "ros/ros.h"
#include <iostream>
#include "ros_service/service.h"
#include <iostream>
#include <sstream>

using namespace std;

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

  ros::init(argc, argv, "service_client");
  ros::NodeHandle n;
  ros::Rate loop_rate(10);
  ros::ServiceClient client =
  n.serviceClient<ros_service::service>("service");
  while (ros::ok()) {
    ros_service::service srv;
    std::stringstream ss;
    ss << "Sending from Here";
    srv.request.in = ss.str();
    if (client.call(srv)) {
      cout << "From Client:
      ["<<  srv.request.in << "],
      Server says [" <<
      srv.response.out << "]" << endl;
    }
    else {
      ROS_ERROR("Failed to call service");
      return 1;
    }
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

To create a service client of the `service` type, we can use the following code line:

```
ros::ServiceClient client =
n.serviceClient<ros_service::service>("service");
```

While, to send the service call to the server we can use the following line:

```
if (client.call(srv))
```

This line returns true if the service is successfully called, false otherwise.

Create custom messages

Similarly to the ROS service message (`srv`), you can create custom messages. ROS already provides a comprehensive set of messages to handle several situation of robotic programming (i.e. `sensor_msgs`, `geometry_msgs`, `nav_msgs`, etc...). However, in some situation could be useful to define your own ROS messages. The message definitions are stored in a .msg file the `msg` folder of your package. Let's create a custom message in the `ros_topic` package:

```
$ roscd ros_topic
$ mkdir msg && cd msg
$ touch demo.msg
```

In this message we want to group together a string and an integer:

```
1 string name
2 int32 data
```

When the `ros_topic` package was created, we hadn't planned to add custom message. For this reason, we have to manually add the `message_generation` dependency in the `CMakeLists.txt`. Open this file adding the`message_generation` in the `find_package` command:

```
1 find_package(catkin REQUIRED COMPONENTS
2   roscpp
3   std_msgs
4   message_generation
5 )
```

And decomment the following line and add the custom message file:

```
1 add_message_files(
2   FILES
3   demo.msg
4 )
5 generate_messages(
6   DEPENDENCIES
7   std_msgs
8 )
```

As usual, to use the added message, you have to compile the `ros_topic` package.

1.4 Exercise

1. *Parrot package*: develop a ROS package with a publisher and a subscriber. The publisher node accepts as input string or characters using the keyboard and publish such data on a ROS topic. The subscriber prints out the published data.

- Test publisher and subscriber also using `rqt` and command line ROS commands.

2. *Fibonacci package*: The Fibonacci Sequence is the series of numbers: `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ....` The next number is found by adding up the two numbers before it. Develop a ROS Service taking 2 numbers (`index`, `length`) as input and returning a portion of the fibonacci sequence:

   - The first argument of the service represents the index of an element along the fibonacci sequence:

     `f[0] = 0, f[1] = 1, f[2] = 1, f[3] = 2, ...`

   - The second argument represent the number of element to return
   - Starting from the `index` element of the sequence, return the next `length` element of the sequence.

     `Example:`

     `Input: ( 8, 3 ), Output: 21, 34, 55`
     `f[8] = 21, f[7] = 13 -> f[9] = 21+13 = 34`

3. *Adder package*: develop a ROS package that adds and publishes two random numbers.

   - Node1: in an infinite loop generates two random float numbers and publishes them on a Topic using a custom message
   - Node2: subscribes to the topic of Node1, sums the two numbers and republishes the result on another topic using custom message with three field:
     - Field 1: first random number
     - Field 2: second random number
     - Field 3: result of the sum