Introduction to ROS services

Wyatt Newman

February, 2015

Introduction: So far, our primary means of communications among nodes has been publish/subscribe. In this mode of communications, the publisher is unaware of its subscribers—and a subscriber knows only of a topic, not which node might publish to that topic. Messages are sent at unknown intervals, and subscribers might miss messages. This style of communications is appropriate for repetitive messages, such as publication of sensor values. For such cases, the sensor publisher should not need to know which or how many nodes subscribe to its output, nor will the publisher change its message in response to any requests from consumers. This form of communications is simple and flexible. Provided the preceding restrictions are not of concern, publish/subscribe is preferred.

Alternatively, it is sometimes necessary to establish bi-directional, one-to-one, reliable communication. In this case, the "client" sends a "request" to a "service," and the service sends back a "response" to the client. The question/answer interaction is on demand, and the client is aware of the name of the service provider.

It should be noted that ROS "services" are intended to be fast responses. When a client sends a request, the client is suspended until an answer returns. Your code should be tolerant of such delays. If the request involves extensive calculations or delays to respond, then you probably should use yet another alternative, the "action server/action client" mechanism (to be introduced in later notes).

The example below is contained in the package "example_ros_service" in the class repository. It is introduced by first describing what is a "service" message, then how to construct a service provider node, and finally how to construct a service client node.

Using Catkin-Simple: "Catkin-Simple" (https://github.com/catkin/catkin_simple) is used in this example package to illustrate a simplification of CMakeLists.txt. To use this, create a package, e.g.:

```
catkin_create_pkg my_new_package roscpp catkin_simple std_msgs
```

The above command will create a new package called "my_new_package", and the package.xml file will note dependencies on roscpp, std_msgs and catkin_simple. The use of catkin_simple simplifies requirements of the programmer in specifying details in CMakeLists.txt.

By referencing "catkin_simple" in package.xml, the CmakeLists.txt file simplifies to the following:

```
cmake_minimum_required(VERSION 2.8.3)
project(example_ros_service)
find_package(catkin_simple REQUIRED)
catkin_simple()
# Executables
cs_add_executable(example_ros_service src/example_ros_service.cpp)
cs_add_executable(example_ros_client src/example_ros_client.cpp)
cs_install()
cs_export()
```

Catkin-simple fills in the tedious, required details by inference from the package.xml file.

Service messages: Defining a custom service message requires describing the data types and field names for both a request and a response. This is done by a ROS template, called a "*.srv" file. Contents of this file auto-generate C++ headers that can be included in C++ files.

To create a new service message, within the desired package (in this example,"example_ros_service"), create a sub-directory called "srv." (This directory is at the same level as "src"). Inside this directory, create a text file named *.srv. In the current example, this text file has been named "example_server_msg.srv." The contents of the example server message is given below:

```
string name
---
bool on_the_list
bool good_guy
int32 age
string nickname
```

In the above, the request structure is defined by the lines *above* "---", and the response structure is defined by the lines *below* "---". The request (for this simple example) consists of a single component, referenced by the name "name", and the datatype contained in this field is a (ROS) string.

For the response part of this example, there are 4 fields, named on_the_list, good_guy, age and nickname. These have respective datatypes of bool, bool, int32 and string (all of which are defined as ROS message types). Although the service message is described very simply in a text file, the compiler will be instructed to parse this file and build a header file that can be included in a C++ program. To inform "catkin" to create this header file, the package.xml file for the package of interest must include the dependency:

```
<build_depend>message_generation</build_depend>
```

(this is already in the package.xml template, and it can simply be uncommented).

The CMakeLists.txt file is overly complex, but it can be replaced with the much simpler text given above under "Catkin-simple." (This version of CMakeLists.txt is already contained in the class repository). Note that catkin-simple uses: cs_add_executable() instead of add_executable().

Having defined the service message, one can test the package compilation even before any source code has been written. To do this, comment out the two cs_add_executable lines in CmakeLists.txt, e.g.:
#cs_add_executable(example_ros_service src/example_ros_service.cpp)
#cs_add_executable(example_ros_client src/example_ros_client.cpp)

The new package can be compiled by running:

```
catkin_make
```

from the ros_ws directory.

Although no source code has yet been written, the catkin build system will recognize that there is a new "srv" file, and it will auto-generate a corresponding C++ compatible header file. This header file will share the same name as the srv file. In the present example, the srv file "example_server_msg.srv"

leads to creation of a header file called “example_server_msg.h”. This header file is located in the directory:

catkin/devel/include/example_ros_service

and this directory is created automatically as part of the creation of the service header file.

Having generated header files for the new service message, future source code can reference a dependency on our new package “example_ros_service” and include the header file “example_server_msg.h”, and the source code can then refer to the message fields of this new message.

For the present case, we will design an example server and a corresponding client that both reside in the same package. However, a complementary server and client could be designed in separate packages, as long as each package is instructed (via package.xml) where to find the required service message(s).

A ROS service node: Next, an example ROS service node is defined in the package “example_ROS_service”, in the subdirectory “src”, named “example_ros_service.cpp.” The CMakeLists.txt file is edited to instruct the compiler to compile this code by adding the line:

cs_add_executable(example_ros_service src/example_ros_service.cpp)

The source code of example_ros_service.cpp follows:

```
//example ROS service:
// run this as: rosrun example_ROS_service example_ROS_service
// in another window, tickle it manually with (e.g.):
//    rosservice call lookup_by_name 'Ted'


#include <ros/ros.h>
#include <example_ros_service/example_server_msg.h>
#include <iostream>
#include <string>
using namespace std;

bool callback(example_ros_service::example_server_msgRequest& request,
example_ros_service::example_server_msgResponse& response)
{
   ROS_INFO("callback activated");
   string in_name(request.name); //let's convert this to a C++-class string, so can use member funcs
   //cout<<"in_name:"<<in_name<<endl;
   response.on_the_list=false;

   // here is a dumb way to access a stupid database...
   // hey: this example is about services, not databases!
   if (in_name.compare("Bob")==0)
 {
       ROS_INFO("asked about Bob");
       response.age = 32;
       response.good_guy=false;
       response.on_the_list=true;
```

```
      response.nickname="BobTheTerrible";
    }
    if (in_name.compare("Ted")==0)
 {
      ROS_INFO("asked about Ted");
      response.age = 21;
      response.good_guy=true;
      response.on_the_list=true;
      response.nickname="Ted the Benevolent";
    }

  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "example_ros_service");
  ros::NodeHandle n;

  ros::ServiceServer service = n.advertiseService("lookup_by_name", callback);
  ROS_INFO("Ready to look up names.");
  ros::spin();

  return 0;
}
```

In the above, note the inclusion of the new header file:

```
#include <example_ROS_service/example_server_msg.h>
```

This file that was auto-generated from the service message definition.

Within the body of “main()”, the line:

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

is similar to creating a publisher in ROS. In this case, a service is created, and it will be known by the name “lookup_by_name.” When a request to this service arrives, the named callback function will be invoked. The service does not have a timed loop. Rather, it sleeps until a request comes in, and incoming requests are serviced by the callback function.

A less obvious attribute of the service node construction is the type declarations of the arguments of the service callback. The argument:

```
example_ros_service::example_server_msgRequest& request
```

declares that the object “request” is reference pointer of type

```
example_ros_service::example_server_msgRequest.
```

This may seem strange, since we did not define a datatype called example_server_msgRequest within our package example_ros_service. The build system created this datatype as part of auto-generating the message header file. The name example_server_msgRequest is created by appending your service

message name (example_server_msg) to the generic name “Request” (and similarly for “Response”). When you define a new service message, you can assume that the system will create these two new datatypes for you.

When the service callback function is invoked, the callback can examine the contents of the incoming request. In the present example, the request has only one field in its definition, called “name.” The line:

```
string in_name(request.name);
```

creates a C++-style “string” object from the characters contained in “request.name.” With this string object, we can invoke the member function “compare()”, e.g. to test if the name is identical to “Bob.”

```
if (in_name.compare("Bob")==0) ...
```

Within the callback routine, fields are filled in for the “response.” When the callback returns, this response message is transmitted back to the client who invoked the request. The mechanism for performing this communication is hidden from the programmer; it is performed as part of the ROS service paradigm (e.g., you do not have to invoke an action similar to “publish(response)”).

Once the ROS service example is compiled, it can be run with:

```
rosrun example_ROS_service example_ROS_service
```

(note: this assumes, as always, that “roscore” is running).

Manual interaction with a ROS service:

Once the example_ROS_service is running, we can see that a new service is available. From a command prompt, enter:

```
rosservice list
```

The response will show that there is a service by the name of /lookup_by_name (which we declared to be the service name of our node).

We can interact manually with the service from the command line, e.g. by typing:

```
rosservice call lookup_by_name 'Ted'
```

The response to this is:

```
on_the_list: True
good_guy: True
age: 21
nickname: Ted the Benevolent
```

We see that the service reacted appropriately to our request. More generally, service requests would be invoked from other ROS nodes.

An example ROS service client:

To interact with a ROS service programmatically, one composes a ROS “client.” An example is given below (example_ros_client.cpp in the package “example_ros_service”).

```
//example ROS client:
// first run: rosrun example_ros_service example_ros_service
// then start this node:  rosrun example_ros_service example_ros_client

#include <ros/ros.h>
#include <example_ros_service/example_server_msg.h> // srv message defined in this  package
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char **argv) {
   ros::init(argc, argv, "example_ros_client");
   ros::NodeHandle n;
   ros::ServiceClient client =
         n.serviceClient<example_ros_service::example_server_msg>("lookup_by_name");
   example_ros_service::example_server_msg srv;
   bool found_on_list = false;
   string in_name;
   while (ros::ok()) {
      cout<<endl;
      cout << "enter a name (x to quit): ";
      cin>>in_name;
      if (in_name.compare("x")==0)
         return 0;
      //cout<<"you entered "<<in_name<<endl;
      srv.request.name = in_name; //"Ted";
      if (client.call(srv)) {
         if (srv.response.on_the_list) {
            cout << srv.request.name << " is known as " << srv.response.nickname << endl;
            cout << "He is " << srv.response.age << " years old" << endl;
            if (srv.response.good_guy)
               cout << "He is reported to be a good guy" << endl;
            else
               cout << "Avoid him; he is not a good guy" << endl;
         } else {
            cout << srv.request.name << " is not in my database" << endl;
         }

      } else {
         ROS_ERROR("Failed to call service lookup_by_name");
         return 1;
      }
   }
   return 0;
}
```

In this program, we include the same message header as in the service node:

```
#include <example_ROS_service/example_server_msg.h>
```

Note that if this node were defined within another package, we would need to list the package dependency “example_ros_service” within the package.xml file.

There are two key lines in the client program. First,

```
ros::ServiceClient client =
    n.serviceClient<example_ROS_service::example_server_msg>("lookup_by_name");
```

creates a ROS “ServiceClient”. This service client expects to communicate requests and responses as defined in: example_ros_service::example_server_msg. Also, this service client expects to communicate with a named service, called "lookup_by_name". (This is the service name we had defined inside of our example service node).

Second, we instantiate an object of a consistent type for requests and responses with:

```
example_ros_service::example_server_msg srv;
```

The above type specifies the package name in which the service message is defined, and the name of the service message itself. In this case “srv” contains both a field for request and a field for response. To send out a service request, we first populate the fields of the request message (in this case, the request message has only a single component):

```
srv.request.name = in_name; //e.g., manually test with contents: "Ted";
```

We then perform the transaction with the named service through the following call:

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

This call will return a boolean to let us know if the call was successful or not. If the call is successful, then we may expect that the components of “srv.response” will be filled in (as provided by the ROS service). These components are examined and displayed by the example code. In this client example, the fields “on_the_list”, “name”, “age” and “good_guy” are evaluated. E.g.:

```
cout << "He is " << srv.response.age << " years old" << endl;
```

looks up and reports the “age” field of the service response message.

Conclusion: Using services can be appropriate for one-to-one, guaranteed communications. Although a sensor (e.g. a joint encoder) can simply keep publishing its current value, this would not be appropriate for a command such as “close_gripper.” In the latter case, we would want to send the command once, and be assured that the command was received and acted on. A client/server interaction would be appropriate for such needs.

Some extra work is required to define the service message and to make sure both the server and the client include the corresponding message headers. Services should be used only as quick request/response interactions. For interactions that can require long durations until a response is ready (e.g., “plan_path”), a more appropriate interface is “action-servers/action-clients”, which are described in subsequent notes.