Introduction to action servers and clients: designing your own action server

Wyatt Newman
October, 2014

Introduction: An earlier document, "Introduction to action servers and clients: the Joint Trajectory Behavior", and the accompanying code in …/examples/example_traj_client, an action client was illustrated. This action client communicated with an existing action server, "joint_traj_behavior" (in a package of the same name). In the present document, together with the example code in …/examples/example_action_server, it will be shown how to design your own action server and corresponding action client. Further details can be found on-line at: http://wiki.ros.org/actionlib. There are many variations, and only simple examples are treated here.

To date, our primary mode of communications within ROS has been through publishing/subscribing. Publish/subscribe does not enforce reliable communications. The subscriber may not yet be listening to the publisher (or may be otherwise occupied), and messages can get dropped. Often, dropping occasional messages is not a problem, e.g. when performing filtering, which depends on the cumulative properties of a potentially large number of samples. However, in other cases we may require absolute confirmation that two nodes have succeeded in communicating—including sending a message back to the requester. ROS "services" are a means to communicate from between two nodes with reliable transmission. However, ROS services are intended for very brief interactions. Services are inappropriate in many instances where the service to be provided may take a relatively long time to fulfill.

From http://wiki.ros.org/actionlib:

> "In any large ROS based system, there are cases when someone would like to send a request to a node to perform some task, and also receive a reply to the request. This can currently be achieved via ROS services.
>
> In some cases, however, if the service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing. The `actionlib` package provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server."

The joint trajectory behavior fits this description. Servicing a motion request may require seconds or tens of seconds to complete. Further, it may be important to interrupt this action in progress, e.g. if a dangerous condition is detected (such as an impending collision).

In the following, we will introduce how to specify messages for communication between an action-client and action-server, then how to design the action-server, and finally an example action-client. This description will reference the example code in …/examples/example_action_server of the class repository.

Preparing the action server package: A new action-server package may be navigating to …/catkin/src (or some subdirectory from here) and invoking:

```
hku_create_pkg example_action_server roscpp actionlib
```

This will create a new package called “example_action_server” that will be written in C++ and will utilize the “actionlib” library. (This is the package name chosen for the example code).

Invoking “hku_create_pkg” does much of the preparatory work for us, including establishing a package.xml file, a CmakeLists.txt file and subdirectories “include” and “src.” However, we will need an additional subdirectory. Navigate to within the new package directory and invoke:

```
mkdir action
```

We will use this directory to define our communications message format between our new server and its future clients. To get the new message pre-processed by hku_make, it is also necessary to edit the package.xml file and uncomment the line:

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

This will result in auto-generating various header files to define message types for client/server communications, as described further next.

Defining action messages:

Within the “action” subdirectory, create a new file with the suffix “.action”. In our example, this is “demo.action,” which contains the following text:

```
#goal definition

int32 input

---

#result definition

int32 output

int32 goal_stamp

---

#feedback

int32 fdbk
```

In the above, we have prescribed three fields: goal, result and feedback. The goal definition, in this simple case, contains only a single component, called “input”, which is of type int32. Note that the “#”sign is a comment delimiter. The labels for goal, result and feedback are just reminders. Message generation will ignore these comments and will assume that there are three fields—in this fixed order (goal, result, feedback), separated by three dashes.

Following three dashes, the “result” message is defined. In this example, the “result” definition consists of two components: “output” and “goal_stamp”, both of which are int32's.

The final definition, “feedback”, is also separated by three dashes, and the example contains only one field, called “fdbk”, also of type int32.

An action message must be written in the above format. The dashes and the order are important; the comments are optional, but helpful.

The components defined in these fields can be comprised of any existing message definitions, provided the corresponding message packages have been named in the "package.xml" file and the corresponding headers are included in the source code file (to be composed, as described below).

Because our package.xml file has "message_generation" enabled, once we perform "hku_make" on our new package, the build system will create multiple new *.h and files, although this is not obvious. These files will be located in …/catkin/devel/include/actionlib_example (for a package called "actionlib_example"). Within this directory, there will be 7 *.h files created, each with a name that starts with "demo" (the name we chose for our "action" specification). All of these are referenced in the header "demoAction.h", allowing one to refer subsequently to messages such as "demoGoal" or "demoResult."

For both our client and server code, we need to refer to these new messages, which is done by referencing the package and the auto-generated message names, such as demoResult, e.g. using:

```
example_action_server::demoResult result_; // put results here
```

The above line of C++ code instantiates a variable called "goal_" that is of type "demoGoal" as defined in the package "example_action_server." Subsequently, we may refer to components of result_, such as in the line:

```
result_.output = g_count;
```

When the server returns the goal message to its client, the value assigned to result_.output will be received by the client (as well as other fields of the result that are populated by the server).

An intent of the action message is to define how a client should specify to the server to what "goal" should be pursued. Sometimes the goal is a physical action (as in the case of our joint trajectory behavior server), but sometimes the goal is to perform a computation. For the latter case, we need to define a "result" message, which should contain the fields appropriate to contain the expected result format. The "feedback" message can be useful for informing clients of incremental progress (including validating that the server is still alive and making progress towards the current goal).

Action messages can be simpler (e.g., if no result or feedback messages are needed), or considerably more complex (e.g., returning a multi-step plan or a processed point cloud).

Designing an action server: The source file "example_action_server.cpp", in the package of the same name, shows a minimal action server. This node uses the "actionlib" library, and it must correspondingly include the actionlib header file:

```
#include <actionlib/server/simple_action_server.h>
```

In addition, we include a header that references our new action message with the line:

```
#include<example_action_server/demoAction.h>
```

Note that we did not create the header "demoAction.h." Rather, this header gets auto-generated by the build system, based on the template we defined in: example_action_server/action/demo.action. Our chosen name “demo.action” is used by the build system to create the header “demoAction.h”.

In the example source file “example_action_server.cpp”, a class is defined, arbitrarily called “exampleActionServer”. This class owns a ros::NodeHandle, instances of actions messages declared as:

```
    example_action_server::demoGoal goal_; // goal message, received from client
    example_action_server::demoResult result_; // put results here
    example_action_server::demoFeedback feedback_; // not used in this example;
```

which use messages types auto-generated from our package's demo.action file.

The class also owns an object “as_”, of type:

```
actionlib::SimpleActionServer<example_action_server::demoAction>
```

The above syntax may be confusing. “actionlib” refers to the actionlib library, which contains a templated class “SimpleActionServer.” We can re-use this class with our new message types by specifying the new messages via: `example_action_server::demoAction.` The first field is our server's package name, and the second field is the “mangled” message name, starting with our chosen base name (demo), appended with the key word “Action”.

Our new class also has a public method, declared as:

void executeCB(const actionlib::SimpleActionServer<example_action_server::demoAction>::GoalConstPtr& goal);

The method name “executeCB” is our own, arbitrary choice. The argument of this function is a poniter to a goal message. Declaration of the goal message is long-winded, referring to the actionlib library, the templated class SimpleActionServer, and our own action message, referred to by:

```
  example_action_server::demoAction
```

which further has an auto-generated type GoalConstPtr.

The constructor of our class also has a cryptic syntax:

```
exampleActionServer::exampleActionServer() :
   as_(nh_, "example_action",
boost::bind(&exampleActionServer::executeCB, this, _1),false)
```

This syntax is required to pass variables to the constructor of the inner object, as_ (an action-server object). We desire to refer to the (private) node handle, nh_, and to instantiate our new action-server object with our chosen name “example_action”. Further, we specify that the new action server should utilize a function of our own design within its behavior. This function we wish for the action-server to use is called “executeCB” (our own name choice). We specify that it is defined within the namespace off class “exampleActionServer”, and the keyword “this” indicates that executeCB is a member of the current object. The argument “_1” is used by boost::bind to tell the simple action server object that our defined function takes one argument. Finally, the initialization specifies “false” to decline to start the server running. This is to avoid a race condition on start-up. Instead, we manually induce start-up withing the constructor at the next line:

```
 as_.start(); //start the server running
```

Within the main program, we instantiate an object of our new class, exampleActionServer, then go into a “spin”. The real work of this node is subsequently performed by callbacks of the action server.

So far, the above details are merely required format. The example code can be copied/pasted, and changes may be made for new names.   The new package name should be replaced for each instance of “example_action_server.” A more mnemonic class name should be chosen and substituted for each occurrence of “exampleActionServer.” A mnemonic action file should be created, and the base name of this action file should be substituted everywhere for the example name “demo” (e.g., in ::demoAction). The variable names goal_, result_ and feedback_ can be changed, but these names should be valid to stay as-is. The callback function name may be changed, if desired, but the example name “executeCB” could be retained without confusion. Importantly, the server name (currently “example_action”) should be changed to a name that is meaningful for the new server. The node name in “main” (“demo_action_server_node”) should also be changed to something relevant.

All of this complexity largely can be ignored, provided the renaming is consistent throughout. The heart of the server is contained within the functionality of “executeCB”. Your new server should install meaningful code here, which performs a useful service.

In executeCB(), you may refer to the information contained within the goal message. This is what is transmitted by a client to your server. Your code may result in information to be returned to the client, in which case you should fill in the expected fields of “result_.” (This example does not illustrate how to send feedback messages to the client, but this extension can be emulated from the on-line tutorials).

Once the important work within executeCB has been performed, this function must conclude by invoking either `as_.setAborted(result_)` or `as_.setSucceeded(result_)`.   In either case, providing the argument “result_” causes these member methods to transmit the message “result_” back to the client. Additionally, the client is informed whether the server concluded successfully or aborted.

In the simple example provided, the server merely does a simple communications diagnostic. The server merely copies the “input” field of the goal message into the “goal_stamp” field of the result message. The server also keeps track of how many times it has serviced goals, and it copies this value into the “output” field of the result message. For this simple, diagnostic server, if the server's record of number of goals achieved differs from the client's input, the server outputs an error message and shuts down.

Designing and action client: A compatible client of our new action server also resides in the same example package, “example_action_server”, with source file called “example_action_client.cpp.” This client also requires the header referencing the actionlib library:

```
#include <actionlib/client/simple_action_client.h>
```

as well as the header referring to the server's action message specifications:

```
#include<example_action_server/demoAction.h>
```

Often, different client programs may use the same server, and these clients might be defined in separate packages. In that case, it is necessary to reference the server package in the “package.xml” file in the new client package (e.g., in our case, <build_depend>example_action_server</build_depend> and <run_depend>example_action_server </run_depend> should be lines in the package.xml file).

The main program creates a goal object of the type prescribed by our action file:

```
example_action_server::demoGoal goal;
```

An object called “action_client” is instantiated from the (templated) class “actionlib::SimpleActionClient”. We specialize this object to use our defined action messages with the template specification: <example_action_server::demoAction>.

The new action client is constructed with the argument “example_action”. This is the name chosen in the design of our action server, as was specified in the class constructor:

exampleActionServer::exampleActionServer() :

as_(nh_, "example_action", boost::bind(&exampleActionServer::executeCB, this, _1),false)

Your new clients will need to know the name of their respective server in order to connect.

One of the member functions of an action-client is “waitForServer()”. The line:

```
bool server_exists =
action_client.waitForServer(ros::Duration(5.0)); // wait up to 5 sec
```

causes the new action client to attempt to connect to the named server. The format allows for waiting only up to some time limit, or waiting indefinitely (if the duration argument is omitted). This method returns “true” if successful connection is made.

In the example program, a goal message has its “input” field filled with the current iteration count (number of attempts to get served). The server is requested to perform its service, as specified in the “goal” message, by invoking the member function:

action_client.sendGoal(goal,&doneCb);

This form of “sendGoal” includes reference to a callback function (arbitrarily) named “doneCb”. The client then invokes:

```
bool finished_before_timeout =
action_client.waitForResult(ros::Duration(5.0));
```

which causes the client to suspend, waiting on the server, but with a specified time-out limit.

If the server returns within the specified time limit, the goalCB function will be triggered. This function receives the “result” message provided by the server. In the example code, the callback function compares the number of goals serviced by the server so far to the number of goals this client has requested. It prints out the difference between the two. If *only* this client requests goals from the server, and if the client and server are both started in the same session, then we should expect that the value of “diff” is always zero, since the number of goals served by the server and the number of goals requested by this client should be identical. However, if either the client or server are stopped and restarted (or if a second client is started), there will be a count mismatch.

For making your own client, you should refer to your respect server's package name and its corresponding action file, in place of all instances of “example_action_server” and occurrences of action file name “demo.” The new client's node name should also be changed to something mnemonic (and unique). In instantiating the action client, the name of your desired server should be substituted for “example_action.”

Most importantly, the “doneCb()” function (which may or may not be renamed) should contain some useful application code that accomplishes your objectives.

Running the example code:

To run the example code, first make sure there is a "roscore" (or gazebo) node running. The server and client can then be started in either order:

```
rosrun example_action_server example_action_server
```

```
rosrun example_action_server example_action_client
```

with these nodes running, there will be 5 new topics under /example_action: cancel, feedback, goal, result, and status. You can watch the nodes communicate by running:

```
 rostopic echo example_action/goal
```

or

```
 rostopic echo example_action/result
```

You can try killing and restarting these nodes to observe the behaviors to timeouts or goal-count mismatches. Reconnection (or new connection) from client to server appears to be somewhat unreliable. Your code might include automatic re-tries for some max count to achieve more reliable connection.