Basic ROS Action Client/Server for Gripper Control

Michaloski, John L.
6/20/2016 9:44:00 AM
X:\src\ROS\GripperMalady\BasicRosActionServerGripperControl.docx

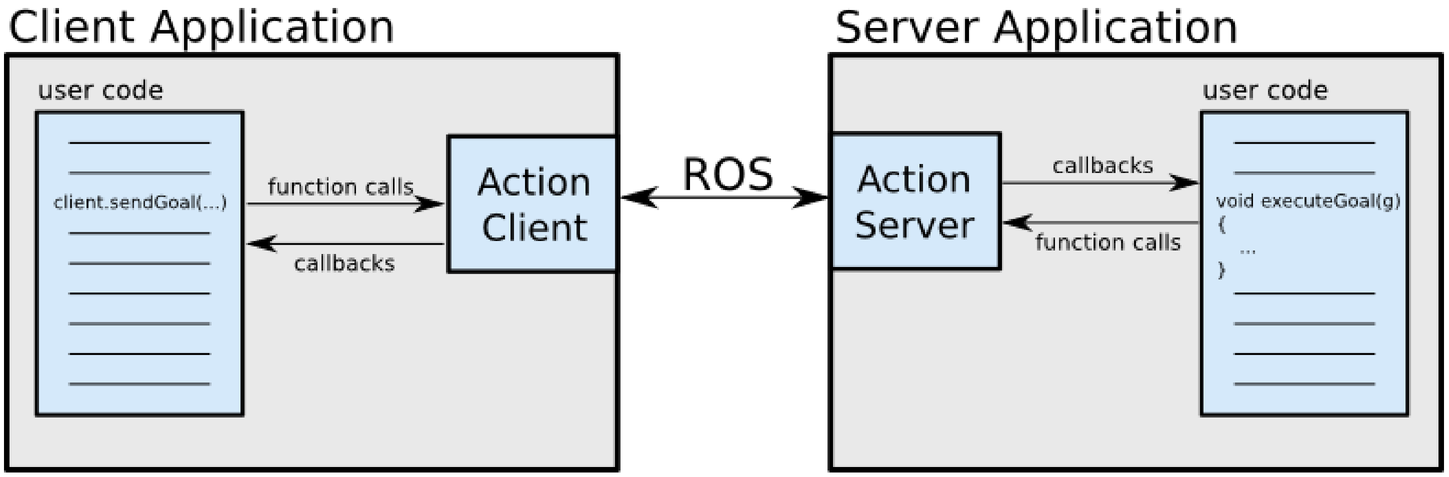
This document describes a minimal amount of C++ code to achieve a simple gripper actionlib client/server. This is the software paradigm that numerous ROS implementers have chosen, so we have boiled it down to the basics. This document does not describe the .action setup, but rather uses established gripper messages. WHope it helps.

Notation

ROS Robot Operating System

ROS Actionlib

ROS provides the actionlib stack which is a standardized interface for dealing with preemptable tasks. Readers are referred to the ROS http://wiki.ros.org/actionlib web site for a tutorial on actionlib. The ActionClient and ActionServer communicate via a "ROS Action Protocol", which is built on top of ROS messages. The client and server then provide a simple API for users to request goals (on the client side) or to execute goals (on the server side) via function calls and callbacks. The client and server communicate through a TCP port



In order for the client and server to communicate, ROS defines a few messages on which they communicate. This is with an action specification. This defines the Goal, Feedback, and Result messages with which clients and servers communicate:

Goal

To accomplish tasks using actions, we introduce the notion of a goal that can be sent to an ActionServer by an ActionClient. In the case of moving the base, the goal would be a PoseStamped message that contains information about where the robot should move to in the world. For controlling the tilting laser scanner, the goal would contain the scan parameters (min angle, max angle, speed, etc).

Feedback

Feedback provides server implementers a way to tell an ActionClient about the incremental progress of a goal. For moving the base, this might be the robot's current pose along the path. For controlling the tilting laser scanner, this might be the time left until the scan completes.

Result

A result is sent from the ActionServer to the ActionClient upon completion of the goal. This is different than feedback, since it is sent exactly once. This is extremely useful when the purpose of the action is to provide some sort of information. For move base, the result isn't very important, but it might contain the final pose of the robot. For controlling the tilting laser scanner, the result might contain a point cloud generated from the requested scan.

The values for the status of a goal are as follows:

- **PENDING** - The goal has yet to be processed by the action server
- **ACTIVE** - The goal is currently being processed by the action server
- **REJECTED** - The goal was rejected by the action server without being processed and without a request from the action client to cancel
- **SUCCEEDED** - The goal was achieved successfully by the action server
- **ABORTED** - The goal was aborted by the action server
- **PREEMPTING** - Processing of the goal was canceled by either another goal, or a cancel request sent to the action server
- **PREEMPTED** - The goal was preempted by either another goal, or a preempt message being sent to the action server
- **RECALLING** - The goal has not been processed and a cancel request has been received from the action client, but the action server has not confirmed the goal is canceled
- **RECALLED** - The goal was canceled by either another goal, or a cancel request before the action server began processing the goal
- **LOST** - The goal was sent by the ActionClient, but disappeared due to some communication error

Coding an Action Server in C++

The demo gripper action client demonstrates the use of a simple action server. Simple ActionServer implement a single goal policy on top of the ActionServer class. The specification of the policy is as follows:

- Only one goal can have an active status at a time
- New goals preempt previous goals based on the stamp in their GoalID field (later goals preempt earlier ones)
- An explicit preempt goal preempts all goals with timestamps that are less than or equal to the stamp associated with the preempt
- Accepting a new goal implies successful preemption of any old goal and the status of the old goal will be change automatically to reflect this

Calling acceptNewGoal accepts a new goal when one is available. The status of this goal is set to active upon acceptance, and the status of any previously active goal is set to preempted. Preempts received for the new goal between checking if isNewGoalAvailable or invocation of a goal callback and the acceptNewGoal call will not trigger a preempt callback. This means, isPreemptRequested should be called after accepting the goal even for callback-based implementations to make sure the new goal does not have a pending preempt request.

```
#include <ros/ros.h>
#include <actionlib/server/simple_action_server.h>
#include <control_msgs/GripperCommandAction.h>
#include <boost/bind.hpp>
#include <string>

class AGripperActionServer {
public:
    typedef actionlib::ActionServer<control_msgs::GripperCommandAction>  ActionServer;
    typedef boost::shared_ptr<ActionServer>                              ActionServerPtr;
    typedef ActionServer::GoalHandle                                     GoalHandle;
     typedef boost::shared_ptr<GoalHandle>                               GoalHandlePtr;
protected:
    ros::NodeHandle nh;
    std::string action_name;
    ActionServer * gripper_server;
public:
    AGripperActionServer (std::string name) ;
    void goalCB(GoalHandle gh) {
        double position_ = gh.getGoal()->command.position;
        double  max_effort_ = gh.getGoal()->command.max_effort;
        ROS_INFO("AGripperActionServer callback for gripper: %s Position=%f", action_name.c_str(),
position_, max_effort_);
    }
    void cancelCB(GoalHandle gh) {
        ROS_INFO("Cancel AGripperActionServer callback for gripper: %s", action_name.c_str());
    }
};

AGripperActionServer::AGripperActionServer (std::string name) : action_name(name)
{
    gripper_server=new ActionServer(nh, "gripper",
                                    boost::bind(&AGripperActionServer::goalCB,   this, _1),
                                    boost::bind(&AGripperActionServer::cancelCB,   this, _1),
                                    false);
                   gripper_server->start();
 }

int main (int argc, char **argv)
{
  ros::init(argc, argv, "test_gripper_action_server");

  ros::NodeHandle pnh("~");

  std::string gripper_name;
  pnh.param<std::string>("gripper_name", gripper_name, "gripper");

  std::string action_name = "gripper";

  // The name of the gripper -> this server communicates over name/inputs and name/outputs
    AGripperActionServer gripper (gripper_name);

  ROS_INFO("Sample action-server spinning for gripper: %s", gripper_name.c_str());
  ros::spin();
}
```

Coding an Action Client in C++

The demo gripper action client demonstrates the integration of an actionlib, gripper messages. This example leverages existing gripper action messages for goal, result, and feedback.

The demo gripper action client takes in goal messages of type control_msgs/GripperCommandAction. From the ROS documentation control_msgs/GripperCommandAction contains a single field, 'command,' of type control _msgs/GripperCommand. The GripperCommand command has two float64 fields, 'position' and 'max_effort'. The 'position' field specifies the desired gripper opening (the size of the space between the two fingertips) in meters: closed is 0.0, and fully open is approximately 0.09. The 'max_effort' field places a limit on the amount of effort (force in N) to apply while moving to that position. If 'max_effort' is negative, it is ignored.

```
#include <ros/ros.h>
#include <actionlib/client/simple_action_client.h>
#include <control_msgs/GripperCommandAction.h>

int main (int argc, char **argv)
{
  ros::init(argc, argv, "test_gripper_action_server");

  ros::NodeHandle pnh("~");

  std::string gripper_name;
  pnh.param<std::string>("gripper_name", gripper_name, "gripper");

  std::string action_name = "gripper";

  // Define the action client (true: we want to spin a thread)
  actionlib::SimpleActionClient< control_msgs::GripperCommandAction > ac(action_name , true);

  // Wait for the action server to come up
  while(!ac.waitForServer(ros::Duration(5.0))) {
    ROS_INFO("Waiting for turn action server to come up");
  }

  // Set the goal
  control_msgs::GripperCommandGoal goal;
  goal.command.position = 0.0;
  goal.command.max_effort = 100.0;

  // Send the goal
  ac.sendGoal(goal);
  return 0;
};
```

The type actionlib::SimpleActionClient<control_msgs::GripperCommandAction >   declares a simple client implementation of the ActionInterface which supports only one goal at a time. The actionlib::SimpleActionClient uses a action message set to construct a ActionClient, and exposes a limited set of easy-to-use hooks for the user. The ROS documentation points out that the concept of GoalHandles has been completely hidden from the user, and that usres must query the SimplyActionClient directly in order to monitor a goal.

Gripper ROS Workspace

First, a ROS catkin workspace was created for the gripper code.

```
  616  cd /usr/local/michalos/
  617  mkdir gripper_ws
  618  cd gripper_ws
  619  mkdir src
  620  catkin init
  Add in gripper server and client action package
  622  catkin build
```

The new catkin developer tool set was used to create and build the workspace.

Running Gripper actionlib Client/Server

After compiling the packages the shell commands will execute the client and server nodes (as well as bring up a roscore node).

```
> source devel/setup.bash
> roslaunch demogripperactionclient actiongripper.launch
```

To see output from demogripperactionserver, added output="screen" otherwise, only errors are sent to console.

```
 <node name="nist_gripper_action_server" pkg="demogripperactionserver"
    type="demogripperactionserver" output="screen">
    <param name="gripper_name" type="str" value="$(arg gripper_name)" />
  </node>
```

A snippet of the Client/server output in the roslaunch shell was:

```
setting /run_id to 271c897c-34b3-11e6-951c-ecf4bb31ca6d
process[rosout-1]: started with pid [7518]
started core service [/rosout]
process[nist_gripper_action_server-2]: started with pid [7535]
[ INFO] [1466185476.638244527]: Sample action-server spinning for gripper: gripper
process[nist_gripper_action_client-3]: started with pid [7566]
[   INFO]   [1466185477.143949150]:   AGripperActionServer   callback   for   gripper:   gripper
Position=0.000000
[nist_gripper_action_client-3] process has finished cleanly
```

The roslaunch file was straightforward:

```
<?xml version="1.0" ?>
<launch>
  <arg name="gripper_name" default="gripper" />

  <node name="nist_gripper_action_server" pkg="demogripperactionserver"
    type="demogripperactionserver" output="screen">
    <param name="gripper_name" type="str" value="$(arg gripper_name)" />
  </node>
  <node name="nist_gripper_action_client" pkg="demogripperactionclient"
    type="demogripperactionclient" output="screen">
    <param name="gripper_name" type="str" value="$(arg gripper_name)" />
  </node>
</launch>
```