Actionlib

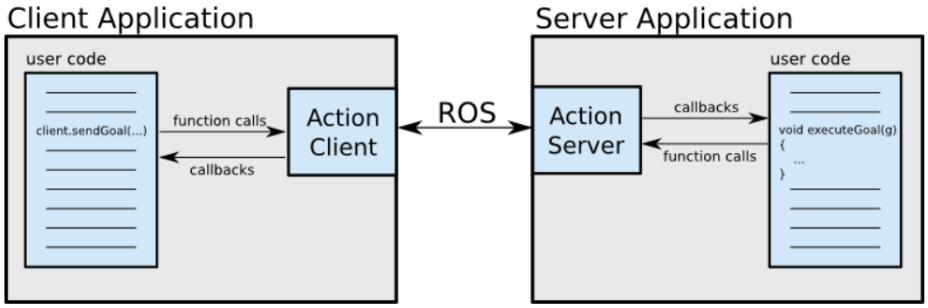
- In any large ROS based system, there are cases when someone would like to send a request to a node to perform some task, and also receive a reply to the request. This can currently be achieved via ROS services.

- 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.

→ This is achived by actionlib.

Client-Server Interaction

- The ActionClient and ActionServer communicate via a "ROS Action Protocol", which is built on top of ROS messages.



Action Specification: Goal, Feedback, & Result

- In order for the client and server to communicate, we need to define a few messages on which they communicate.

action specification

Goal

Feedback

Result

Feedback provides server implementers a way to tell an ActionClient about the incremental progress of a goal.

A result is sent from the ActionServer to the ActionClient upon completion of the goal.

★ .action file

- The action specification is defined using a .action file.
- The .action file has the goal definition, followed by the result definition, followed by the feedback definition, with each section separated by 3 hyphens (---).

Example

./action/DoDishes.action

```
# Define the goal
uint32 dishwasher_id  # Specify which dishwasher we want to use
---
# Define the result
uint32 total_dishes_cleaned
---
# Define a feedback message
float32 percent_complete
```

★ Build a package by Catkin

5.1.1 Build a package that contains .action file

Add the following to your CMakeLists.txt file *before* `catkin_package()`.

```
find_package(catkin REQUIRED genmsg actionlib_msgs)
add_action_files(DIRECTORY action FILES DoDishes.action)
generate_messages(DEPENDENCIES actionlib_msgs)
```

Additionally, the `package.xml` of the package that includes `.action` files must include the following dependencies:

```
<build_depend>actionlib_msgs</build_depend>
<exec_depend>actionlib_msgs</exec_depend>
```

Alternatively format 2 of package.xml onward, you can use `depend` tag:

```
<depend>actionlib</depend>
<depend>actionlib_msgs</depend>
```

5.1.2 Build a package that depends on actionlib API

Package that depends on actionlib API to implement an action server or use an action client needs another dependency on `actionlib`.

`CMakeLists.txt`

```
find_package(catkin REQUIRED genmsg actionlib_msgs actionlib)
add_action_files(DIRECTORY action FILES DoDishes.action)
generate_messages(DEPENDENCIES actionlib_msgs)
```

`package.xml`

```
<build_depend>actionlib</build_depend>
<build_depend>actionlib_msgs</build_depend>
<exec_depend>actionlib</exec_depend>
<exec_depend>actionlib_msgs</exec_depend>
```

★ Results

5.2 Results

For the `DoDishes.action`, the following messages are generated by `genaction.py`:

- `DoDishesAction.msg`
- `DoDishesActionGoal.msg`
- `DoDishesActionResult.msg`
- `DoDishesActionFeedback.msg`
- `DoDishesGoal.msg`
- `DoDishesResult.msg`
- `DoDishesFeedback.msg`

These messages are then used internally by actionlib to communicate between the ActionClient and ActionServer.

C++ Simple action Client

```cpp
#include <chores/DoDishesAction.h> // Note: "Action" is appended
#include <actionlib/client/simple_action_client.h>

typedef actionlib::SimpleActionClient<chores::DoDishesAction> Client;

int main(int argc, char** argv)
{
  ros::init(argc, argv, "do_dishes_client");
  Client client("do_dishes", true); // true -> don't need ros::spin()
  client.waitForServer();
  chores::DoDishesGoal goal;
  // Fill in goal here
  client.sendGoal(goal);
  client.waitForResult(ros::Duration(5.0));
  if (client.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
    printf("Yay! The dishes are now clean");
  printf("Current State: %s\n", client.getState().toString().c_str());
  return 0;
}
```

Note: For the C++ SimpleActionClient, the waitForServer method will only work if a separate thread is servicing the client's callback queue. This requires passing in true for the spin_thread option of the client's constructor, running with a multi-threaded spinner, or using your own thread to service ROS callback queues.

C++ Simple action Server

```cpp
#include <chores/DoDishesAction.h>  // Note: "Action" is appended
#include <actionlib/server/simple_action_server.h>

typedef actionlib::SimpleActionServer<chores::DoDishesAction> Server;

void execute(const chores::DoDishesGoalConstPtr& goal, Server* as)  // Note: "Action" is n
ot appended to DoDishes here
{
  // Do lots of awesome groundbreaking robot stuff here
  as->setSucceeded();
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "do_dishes_server");
  ros::NodeHandle n;
  Server server(n, "do_dishes", boost::bind(&execute, _1, &server), false);
  server.start();
  ros::spin();
  return 0;
}
```