ROS Basics in 5 days

Unit 8 ROS Actions: Clients

- Summary -

Estimated time to completion: 3 hours

What will you learn with this unit?

- What is a ROS action
- How to manage the actions of a robot
- How to call an action server

- End of Summary -

1) Did you understand the previous sections about topics and services?
2) Are they clear to you?
3) Did you have a good breakfast today?

If your answers to all of those questions were yes, then you are ready to learn about ROS actions. Otherwise, go back and do not come back until all of those answers are a big YES. You are going to need it...

Quadrotor simulation

Before starting with ROS actions learning, let's have some fun with the quadrotor simulation.
Make the quadrotor take off and control it with the keyboard.
How would you do that?

By issuing the following commands:

First, you need to take off.

▶ **Execute in Shell #1**

In [ ]:
```
rostopic pub /drone/takeoff std_msgs/Empty "{}"
```

Hit **CTRL+C** to stop it and to be able to type more commands. In this case, the commands to move the drone with the keyboard.

▶ Execute in Shell #1

In [ ]:
```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

To land the drone, just publish into the **/drone/land** topic:

▶ Execute in Shell #1

In [ ]:
```
rostopic pub /drone/land std_msgs/Empty "{}"
```

Code Explanation #1

`rosrun` : ROS command that allows you to run a ROS program without having to create a launch file to launch it (it is a different way from what we've been doing here).

`teleop_twist_keyboard` : Name of the package where the ROS program is. In this case, where the python executable is.

`teleop_twist_keyboard.py` : Python executable that will be run. In this case, it's an executable that allows you to input movement commands through the keyboard. When executed, it displays the instructions to move the robot.

End Code Explanation #1

Fig.8.1 - AR.Drone moved with teleop_twist_keyboard.py.

- Exercise 8.1 -

Try to takeoff, move, and land the drone using the keyboard, as shown in the gif in {Fig: 8.1}

- End of Exercise 8.1 -

8.1 What are actions?

Actions are like asynchronous calls to services

Actions are very similar to services. When you call an action, you are calling a functionality that another node is providing. Just the same as with services. The difference is that when your node calls a service, it must wait until the service finishes. **When your node calls an action, it doesn't necessarily have to wait for the action to complete.**

Hence, an action is an asynchronous call to another node's functionality.

- The node that provides the functionality has to contain an **action server**. The *action server* allows other nodes to call that action functionality.
- The node that calls to the functionality has to contain an **action client**. The *action client* allows a node to connect to the *action server* of another node.

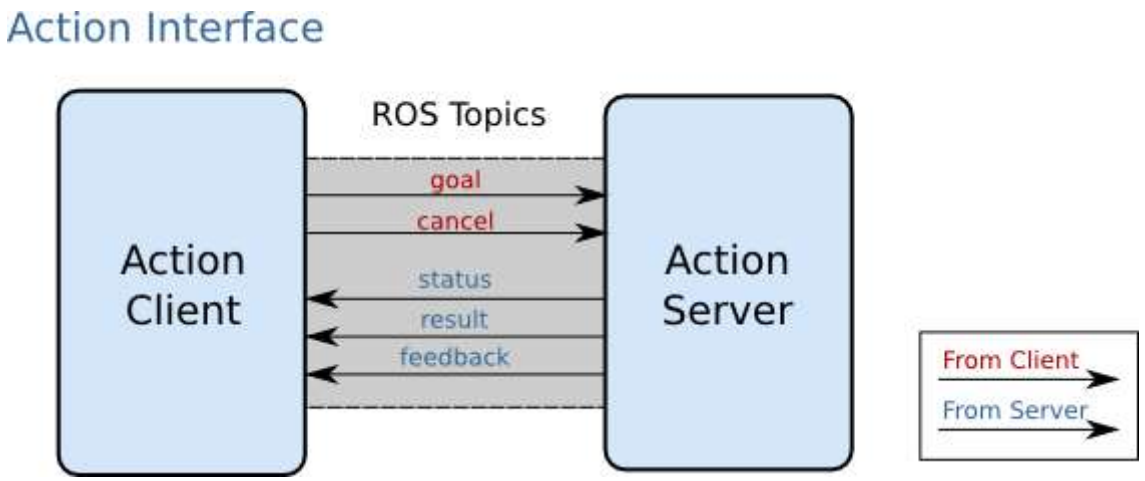


Fig. 8.2 - Action Interface Diagram showing the under-the-hood communication system

Now let's see an action in action (I'm so funny!).

- Exercise 8.2 -

Go to a shell and launch the *ardrone action server* with the following command:

Important!! Keep this program running for the rest of the tutorial, since it is the one that provides the action server you are going to use.

▶ Execute in Shell #1

In [ ]:
```
roslaunch ardrone_as action_server.launch
```

Leave it running.

Questions:

- How do you know the whole list of topics available?

▶ Execute in Shell #2

In [ ]:
```
rostopic list
```

- How do you know the whole list of services available?

▶ Execute in Shell #2

In [ ]:
```
rosservice list
```

- How do you know the whole list of action servers available?

▶ Execute in Shell #2

In [ ]:
```
rosaction list
```

WRONG GUESS!!!

Fig. 8.3 - Didn't Say The Magic Word

In order to find which actions are available on a robot, you must do a ***rostopic list.***

▶ Execute in Shell #2

In [ ]:
```
rostopic list
```

Shell #2 Output

```
...
...
/ardrone_action_server/cancel
/ardrone_action_server/feedback
/ardrone_action_server/goal
/ardrone_action_server/result
/ardrone_action_server/status
...
...
```

- End of Exercise 8.2 -

When a robot provides an action, you will see that in the topics list. There are 5 topics with the same base name, and with the subtopics ***cancel***, ***feedback***, ***goal***, ***result***, and ***status***.

For example, in the previous rostopic list, the following topics were listed:

```
/ardrone_action_server/cancel
/ardrone_action_server/feedback
/ardrone_action_server/goal
/ardrone_action_server/result
/ardrone_action_server/status
```

This is because you previously launched the **ardrone_action_server** with the command **roslaunch ardrone_as action_server.launch** (Exercise 8.2).

Every action server creates those 5 topics, so you can always tell that an action server is there because you identified those 5 topics.

Therefore, in the example above:

- `ardrone_action_server` : Is the name of the Action Server.

- `cancel, feedback, goal, result and status` : Are the messages used to comunicate with the Action Server.

8.2 Calling an action server

The ***ardrone_action_server*** action server is an action that you can call. If you call it, it will start taking pictures with the front camera, one picture every second, for the amount of seconds specified in the calling message (it is a parameter that you specify in the call).

Calling an action server means sending a message to it. In the same way as with *topics* and *services*, it all works by passing messages around.

- The message of a topic is composed of a single part: the information the topic provides.
- The message of a service has two parts: the request and the response.
- **The message of an action server is divided into three parts: the goal, the result, and the feedback.**

All of the action messages used are defined in the ***action*** directory of their package.

You can go to the ***ardrone_as*** package and see that it contains a directory called ***action***. Inside that *action* directory, there is a file called ***Ardrone.action***. That is the file that specifies the type of the message that the action uses.

Type in a shell the following commands to see the message structure:

- Exercise 8.3 -

Type in a shell the following commands to see the message structure:

▶ Execute in Shell #2

In [ ]:
```
roscd ardrone_as/action
```

In [ ]:
```
cat Ardrone.action
```

Shell #2 Output

```
#goal for the drone
int32 nseconds  # the number of seconds the drone will be taking pictures
---
#result
sensor_msgs/CompressedImage[] allPictures # an array containing all the pictures taken along the nseconds
---
#feedback
sensor_msgs/CompressedImage lastImage  # the last image taken
```

- End of Exercise 8.3 -

You can see in the previous step how the message is composed of three parts:

goal: Consists of a variable called ***nseconds*** of type ***Int32***. This Int32 type is a standard ROS message, therefore, it can be found in the std_msgs package (http://wiki.ros.org/std_msgs). Because it's a standard package of ROS, it's not needed to indicate the package where the *Int32* can be found.
result: Consists of a variable called ***allPictures***, which is an array of type ***CompressedImage[]***, found in the sensor_msgs package (http://wiki.ros.org/sensor_msgs).
feedback: Consists of a variable called ***lastImage*** of type ***CompressedImage[]***, found in the sensor_msgs package (http://wiki.ros.org/sensor_msgs).

You will learn in the second part of this chapter about how to create your own action messages. For now, you must only understand that every time you call an action, the message implied contains three parts, and that each part can contain **more than one** variable.

8.2.1 Actions provide feedback

Due to the fact that calling an action server does not interrupt your thread, action servers provide a message called ***the feedback***. The feedback is a message that the action server generates every once in a while to indicate how the action is going (informing the caller of the status of the requested action). It is generated while the action is in progress.

8.2.2 How to call an action server

The way you call an action server is by implementing an ***action client***.

The following is a self-explanatory example of how to implement an action client that calls the ardrone_action_server and makes it take pictures for 10 seconds.

- Exercise 8.4 -

First, create the package **my_action_client_example_pkg**, executing the following comands in the Web Shell:

▶ Execute in Shell #1

In [ ]:
```
cd ~/catkin_ws/src
```

In [ ]:
```
catkin_create_pkg my_action_client_example_pkg rospy std_msgs
```

Now create the **ardrone_action_client.py** script inside it:

▶ Execute in Shell #1

In [ ]:
```
cd ~/catkin_ws/src/my_action_client_example_pkg
```

In [ ]:
```
mkdir scripts
```

In [ ]:
```
cd scripts
```

In [ ]:
```
touch ardrone_action_client.py
```

In [ ]:
```
chmod +x ardrone_action_client.py
```

- End of Exercise 8.4 -

Python Program {8.4a}: ardrone_action_client.py

In [ ]:

```
#! /usr/bin/env python
import rospy
import time
import actionlib
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult, ArdroneFeedback

nImage = 1

# definition of the feedback callback. This will be called when feedback
# is received from the action server
# it just prints a message indicating a new message has been received
def feedback_callback(feedback):
    global nImage
    print('[Feedback] image n.%d received'%nImage)
    nImage += 1

# initializes the action client node
rospy.init_node('drone_action_client')

# create the connection to the action server
client = actionlib.SimpleActionClient('/ardrone_action_server', ArdroneAction)
# waits until the action server is up and running
client.wait_for_server()

# creates a goal to send to the action server
goal = ArdroneGoal()
goal.nseconds = 10 # indicates, take pictures along 10 seconds

# sends the goal to the action server, specifying which feedback function
# to call when feedback received
client.send_goal(goal, feedback_cb=feedback_callback)

# Uncomment these lines to test goal preemption:
#time.sleep(3.0)
```

```
#client.cancel_goal()  # would cancel the goal 3 seconds after starting

# wait until the result is obtained
# you can do other stuff here instead of waiting
# and check for status from time to time
# status = client.get_state()
# check the client API link below for more info

client.wait_for_result()

print('[Result] State: %d'%(client.get_state()))
```

Code Explanation Python Program: {8.4a}

And now execute it in a webshell:

▶ Execute in Shell #2

In [ ]:
```
rosrun my_action_client_example_pkg ardrone_action_client.py
```

Important!! Remember that you have to have the `roslaunch ardrone_as action_server.launch` running (probably in WebShell #1), otherwise this won't work because there will be NO action server to be connected to.

▶ Execute in Shell #1

In [ ]:
```
roslaunch ardrone_as action_server.launch
```

The code to call an action server is very simple:

- First, you create a client connected to the action server you want:

client = actionlib.SimpleActionClient('/ardrone_action_server', ArdroneAction)
client = actionlib.SimpleActionClient('/the_action_server_name', the_action_server_message_python_object)

  * First parameter is the name of the action server you want to connecto to.
  * Second parameter is the type of action message that it uses. The convention goes as follows:

  If your **action message file** was called **Ardrone.action**, then the type of action message you must specify is **ArdroneAction**. The same rule applies to any other type (**R2Action**, for an **R2.action** file or **LukeAction** for a **Luke.action** file). In our exercise it is:

client = actionlib.SimpleAction('/ardrone_action_server', ArdroneAction)

- Then you create a goal:

goal = ArdroneGoal()

  Again, the convention goes as follows:

  If your **action message file** was called ***Ardrone.action***, then the type of goal message you must specify is ***ArdroneGoal()***. The same rule applies to any other type (***R2Goal()*** for an ***R2.action*** file or ***LukeGoal()*** for a ***Luke.action*** file).
  Because the goal message requires to provide the number of seconds taking pictures (in the ***nseconds variable***), you must set that parameter in the goal class instance:

goal.nseconds = 10

- Next, you send the goal to the action server:

client.send_goal(goal, feedback_cb=feedback_callback)

  That sentence calls the action. In order to call it, you must specify 2 things:

  1. The goal parameters

2. A feedback function to be called from time to time to know the status of the action.

   At this point, the action server has received the goal and started to execute it (taking pictures for 10 seconds). Also, feedback messages are being received. Every time a feedback message is received, the ***feedback_callback*** function is executed.

- Finally, you wait for the result:

client.wait_for_result()

End Code Explanation {8.4a}

8.3 How to perform other tasks while the Action is in progress

You know how to call an action and wait for the result but... That's exactly what a service does! Then why are you learning actions?
Good point!

So, the **SimpleActionClient** objects have two functions that can be used for knowing if the action that is being performed has finished, and how:

1) **wait_for_result()**: This function is very simple. When called, it will wait until the action has finished and returns a true value. As you can see, it's useless if you want to perform other tasks in parallel because the program will stop there until the action is finished.

2) **get_state()**: This function is much more interesting. When called, it returns an integer that indicates in which state is the action that the SimpleActionClient object is connected to.

0 ==> PENDING
1 ==> ACTIVE
2 ==> PREEMPTED
3 ==> SUCCEEDED
4 ==> ABORTED
5 ==> REJECTED
6 ==> PREEMPTING
7 ==> RECALLING
8 ==> RECALLED
9 ==> LOST

This allows you to create a while loop that checks if the value returned by get_state() has succeeded, is still processing, recalled, aborted, etc. This allows you to check the status of the action goal, while still being able to do other things in the meantime.

- Exercise 8.5 -

Execute the following Python codes {8.5a: wait_for_result_test.py} and {8.5b: no_wait_for_result_test.py}

Observe the difference between them (the code is printed below) and think about why this is.

Important!! Remember that you have to have the `roslaunch ardrone_as action_server.launch` running (probably in WebShell #1), otherwise this won't work because there will be NO action server to be connected to.

Create these scripts in the package **my_action_client_example_pkg** (created before):

▶ Execute in Shell #2

In [ ]:
```
roscd my_action_client_example_pkg/scripts
```

In [ ]:
```
touch wait_for_result_test.py
```

In [ ]:
```
chmod +x wait_for_result_test.py
```

In [ ]:
```
touch no_wait_for_result_test.py
```

In [ ]:
```
chmod +x no_wait_for_result_test.py
```

- End of Exercise 8.5 -

Python Program {8.5a}: wait_for_result_test.py

In [ ]:

```
#! /usr/bin/env python

import rospy
import time
import actionlib
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult, ArdroneFeedback

nImage = 1

# definition of the feedback callback. This will be called when feedback
# is received from the action server
# it just prints a message indicating a new message has been received
def feedback_callback(feedback):
    global nImage
    print('[Feedback] image n.%d received'%nImage)
    nImage += 1


# initializes the action client node
rospy.init_node('example_with_waitforresult_action_client_node')


action_server_name = '/ardrone_action_server'
client = actionlib.SimpleActionClient(action_server_name, ArdroneAction)

# waits until the action server is up and running
rospy.loginfo('Waiting for action Server '+action_server_name)
client.wait_for_server()
rospy.loginfo('Action Server Found...'+action_server_name)


# creates a goal to send to the action server
goal = ArdroneGoal()
goal.nseconds = 10 # indicates, take pictures along 10 seconds
```

```
client.send_goal(goal, feedback_cb=feedback_callback)
rate = rospy.Rate(1)

rospy.loginfo("Lets Start The Wait for the Action To finish Loop...")
while not client.wait_for_result():
    rospy.loginfo("Doing Stuff while waiting for the Server to give a result....")
    rate.sleep()

rospy.loginfo("Example with WaitForResult Finished.")
```

Python Program {8.5b}: no_wait_for_result_test.py

In [ ]:

```
#! /usr/bin/env python

import rospy
import time
import actionlib
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult, ArdroneFeedback

"""
class SimpleGoalState:
    PENDING = 0
    ACTIVE = 1
    DONE = 2
    WARN = 3
    ERROR = 4

"""
# We create some constants with the corresponing vaules from the SimpleGoalState class
PENDING = 0
ACTIVE = 1
DONE = 2
WARN = 3
ERROR = 4

nImage = 1

# definition of the feedback callback. This will be called when feedback
# is received from the action server
# it just prints a message indicating a new message has been received
def feedback_callback(feedback):
    """
    Error that might jump

    self._feedback.lastImage =
AttributeError: 'ArdroneAS' obj
```

```
    """
    global nImage
    print('[Feedback] image n.%d received'%nImage)
    nImage += 1

# initializes the action client node
rospy.init_node('example_no_waitforresult_action_client_node')

action_server_name = '/ardrone_action_server'
client = actionlib.SimpleActionClient(action_server_name, ArdroneAction)

# waits until the action server is up and running
rospy.loginfo('Waiting for action Server '+action_server_name)
client.wait_for_server()
rospy.loginfo('Action Server Found...'+action_server_name)

# creates a goal to send to the action server
goal = ArdroneGoal()
goal.nseconds = 10 # indicates, take pictures along 10 seconds

client.send_goal(goal, feedback_cb=feedback_callback)


# You can access the SimpleAction Variable "simple_state", that will be 1 if active, and 2 when finished.
#Its a variable, better use a function like get_state.
#state = client.simple_state
# state_result will give the FINAL STATE. Will be 1 when Active, and 2 if NO ERROR, 3 If Any Warning, and 3 i
state_result = client.get_state()

rate = rospy.Rate(1)

rospy.loginfo("state_result: "+str(state_result))

while state_result < DONE:
```

```
    rospy.loginfo("Doing Stuff while waiting for the Server to give a result....")
    rate.sleep()
    state_result = client.get_state()
    rospy.loginfo("state_result: "+str(state_result))

rospy.loginfo("[Result] State: "+str(state_result))
if state_result == ERROR:
    rospy.logerr("Something went wrong in the Server Side")
if state_result == WARN:
    rospy.logwarn("There is a warning in the Server Side")

#rospy.loginfo("[Result] State: "+str(client.get_result()))
```

Code Explanation Python Programs: {8.5a} and {8.5b}

And now execute it in a webshell:

▶ Execute in Shell #2

In [ ]:
```
rosrun my_action_client_example_pkg wait_for_result_test.py
```

And after:

In [ ]:
```
rosrun my_action_client_example_pkg no_wait_for_result_test.py
```

Important!! Remember that you have to have the `roslaunch ardrone_as action_server.launch` running (probably in WebShell #1), otherwise this won't work because there will be NO action server to be connected to.

▶ Execute in Shell #1

In [ ]:
```
roslaunch ardrone_as action_server.launch
```

Essentially, the difference is that in the first program (8.5a), the log message *rospy.loginfo("Doing Stuff while waiting for the Server to give a result....")* will never be printed, while in the 8.5b it will.

This is because in 8.5a, the program starts the while loop to check if the value returned by *client.wait_for_result()* is True or False, but it will wait for a value that will only be returned when the Action has Finished. Therefore, it will never get inside the while loop because it will always return the value True.

On the other hand, in the 8.5b program, it checks if *state_result < DONE*. And because the function ***get_state()*** will return the current state of the action immediately, it allows other tasks to perform in parallel. In this case, **printing the log message WHILE printing also the feedback of the the Action**.

End Code Explanation Python Programs: {8.5a} and {8.5b}

You can get more information about ROS action clients in python in this **client API** (http://docs.ros.org/jade/api/actionlib/html/classactionlib_1_1simple__action__client_1_1SimpleActionClient.html)

- Exercise 8.6 -

- Create a package that contains and launches the action client from Exercise {8.4a}: ardrone_action_client.py, from a launch file.
- Add some code that makes the quadcopter move around while the action server has been called (in order to take pictures while the robot is moving).
- Stop the movement of the quadcopter when the last picture has been taken (action server has finished).

- End of Exercise 8.6 -

- Notes for Exercise 8.6 -

1) You can send Twist commands to the quadcopter in order to move it. These commands have to be published in ***/cmd_vel*** topic. Remember the ***Topics Units***.

2) You must send movement commands while waiting until the result is received, creating a loop that sends commands at the same time that check for completion. In order to be able to send commands while the action is in progress, you need to use the SimpleActionClient function ***get_state()***.

3) Also, take into account that in some cases, the 1st message published into a topic may fail (because the topic connections are not ready yet). It's important to bear this in mind specially for **taking off/landing** the drone, since it's based in a single publication into the corresponding topics. Have a look at this post (https://get-help.robotigniteacademy.com/t/how-to-publish-once-only-one-message-into-a-topic-and-get-it-to-work/346) for more information about this.

- End of Notes -

8.4 Preempting a goal

It happens that you can cancel a goal previously sent to an action server prior to its completion.
Cancelling a goal while it is being executed is called **preempting a goal**

You may need to preempt a goal for many reasons, like, for example, the robot went mad about your goal and it is safer to stop it prior to the robot doing some harm.

In order to preempt a goal, you send the cancel_goal to the server through the client connection.

In [ ]:
```
client.cancel_goal()
```

- Exercise 8.7 -

Execute the following Python code {8.6a: cancel_goal_test.py}

See how the goal gets cancelled.

Important!! Remember that you have to have the `roslaunch ardrone_as action_server.launch` running (probably in WebShell #1), otherwise this won't work because there will be NO action server to be connected to.

Create this script in the package **my_action_client_example_pkg** (created before):

▶ **Execute in Shell #2**

In [ ]:
```
roscd my_action_client_example_pkg/scripts
```

In [ ]:
```
touch cancel_goal_test.py
```

In [ ]:
```
chmod +x cancel_goal_test.py
```

- End of Exercise 8.7 -

Python Program {8.6a}: cancel_goal_test.py

In [ ]:

```
#! /usr/bin/env python

import rospy
import time
import actionlib
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult, ArdroneFeedback

# We create some constants with the corresponing vaules from the SimpleGoalState class
PENDING = 0
ACTIVE = 1
DONE = 2
WARN = 3
ERROR = 4

nImage = 1

# definition of the feedback callback. This will be called when feedback
# is received from the action server
# it just prints a message indicating a new message has been received
def feedback_callback(feedback):
    """
    Error that might jump

    self._feedback.lastImage =
AttributeError: 'ArdroneAS' obj

    """
    global nImage
    print('[Feedback] image n.%d received'%nImage)
    nImage += 1

# initializes the action client node
rospy.init_node('example_no_waitforresult_action_client_node')
```

```
action_server_name = '/ardrone_action_server'
client = actionlib.SimpleActionClient(action_server_name, ArdroneAction)

# waits until the action server is up and running
rospy.loginfo('Waiting for action Server '+action_server_name)
client.wait_for_server()
rospy.loginfo('Action Server Found...'+action_server_name)

# creates a goal to send to the action server
goal = ArdroneGoal()
goal.nseconds = 10 # indicates, take pictures along 10 seconds

client.send_goal(goal, feedback_cb=feedback_callback)


# You can access the SimpleAction Variable "simple_state", that will be 1 if active, and 2 when finished.
#Its a variable, better use a function like get_state.
#state = client.simple_state
# state_result will give the FINAL STATE. Will be 1 when Active, and 2 if NO ERROR, 3 If Any Warning, and 3 i
state_result = client.get_state()

rate = rospy.Rate(1)

rospy.loginfo("state_result: "+str(state_result))
counter = 0
while state_result < DONE:
    rospy.loginfo("Doing Stuff while waiting for the Server to give a result....")
    counter += 1
    rate.sleep()
    state_result = client.get_state()
    rospy.loginfo("state_result: "+str(state_result)+", counter ="+str(counter))
    if counter == 2:
        rospy.logwarn("Canceling Goal...")
        client.cancel_goal()
        rospy.logwarn("Goal Canceled")
```

```
        state_result = client.get_state()
        rospy.loginfo("Update state_result after Cancel : "+str(state_result)+", counter ="+str(counter))
```

Execute in the terminal:

▶ Execute in Shell #2

In [ ]:
```
rosrun my_action_client_example_pkg cancel_goal_test.py
```

Important!! Remember that you have to have the `roslaunch ardrone_as action_server.launch` running (probably in WebShell #1), otherwise this won't work because there will be NO action server to be connected to.

▶ Execute in Shell #1

In [ ]:
```
roslaunch ardrone_as action_server.launch
```

Code Explanation Python Program: {8.6a}

It's exactly the same code as the {8.5b}, except for the use of the `cancel_goal()` function.

This program counts to 2, and then it cancels the goal. This triggers the server to finish the goal and, therefore, the function `get_state()` returns the value DONE (2).

End Code Explanation Python Program: {8.6a}

There is a known ROS issue with Actions. It issues a warning when the connection is severed. It normally happens when you cancel a goal or you just terminate a program with a client object in it. The warning is given in the Server Side.

[WARN] Inbound TCP/IP connection failed: connection from sender terminated before handshake header received. 0 bytes were received. Please check sender for additional details.

Just don't panic, it has no effect on your program.

8.5 How does all that work?

You need to understand how the communication inside the actions works. It is not that you are going to use it for programming. As you have seen, programming an action client is very simple. However, it will happen that your code will have bugs and you will have to debug it. In order to do proper debugging, you need to understand how the communication between *action servers* and *action clients* works.

As you already know, an *action message* has three parts:

- the goal
- the result
- the feedback

Each one corresponds to a topic and to a type of message.
For example, in the case of the **ardrone_action_server**, the topics involved are the following:

- the goal topic: **/ardrone_action_server/goal**
- the result topic: **/ardrone_action_server/result**
- the feedback topic: **/ardrone_action_server/feedback**

Look again at the ActionClient+ActionServer communication diagram.

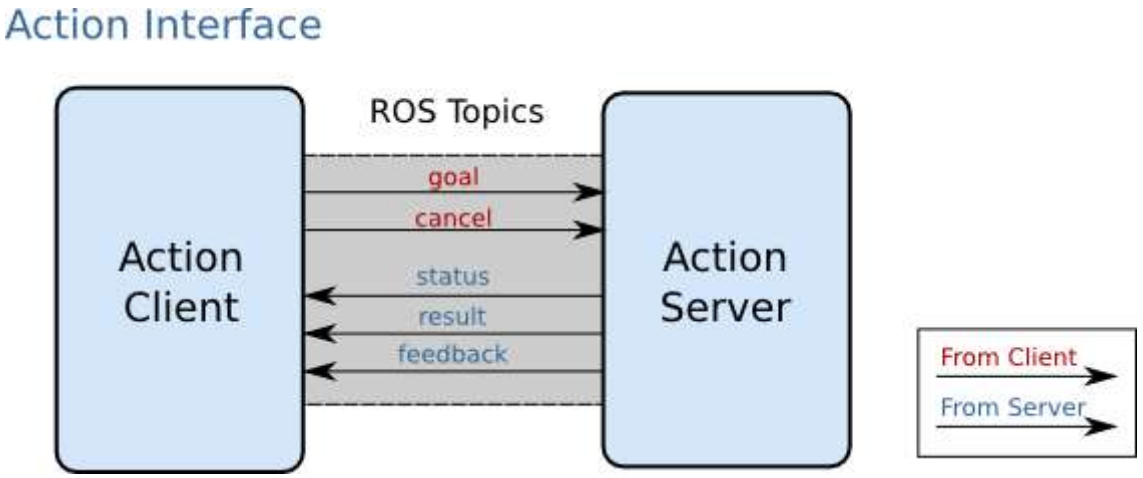


Fig. 8.4 - Action Interface Diagram Copy

So, whenever an action server is called, the sequence of steps are as follows:

1. When an ***action client*** calls an ***action server*** from a node, what actually happens is that the ***action client*** sends to the ***action server*** the goal requested through the */ardrone_action_server/goal* topic.
2. When the ***action server*** starts to execute the goal, it sends to the ***action client*** the feedback through the */ardrone_action_server/feedback* topic.
3. Finally, when the ***action server*** has finished the goal, it sends to the ***action client*** the result through the */ardrone_action_server/result* topic.

Now, let's do the following exercise in order to see how all this ballet happens underneath your programs.

- Exercise 8.8 -

For this exercise, you will need to execute 5 commands. However, as you already know, there are only 4 shells available. To overcome this, you will launch the Action Server in the background.

- Note -

In Linux, you can execute a command in the background using the `&` key. However, when a program is running in the background, you can't kill with **Ctrl + C** as usual. Instead, in order to kill the program, you can run the command `rosnode kill` to kill directly the ROS node:

- End of Note -

The following command launches the action server in the background and then executes a `rostopic echo` command for the topic `/ardrone_action_server/goal` :

▶ Execute in Shell #1

In [ ]:
```
roslaunch ardrone_as action_server.launch & rostopic echo /ardrone_action_server/goal
```

Echo the feedback topic:

▶ Execute in Shell #2:

In [ ]:
```
rostopic echo /ardrone_action_server/feedback
```

Echo the result topic

▶ Execute in Shell #3:

In [ ]:
```
rostopic echo /ardrone_action_server/result
```

Execute the launch you created in Exercise 8.6, so that the drone starts to take pictures and move around.

Launch the action server client.

▶ Execute in Shell #4:

In [ ]:
```
roslaunch <pkg_name> <your_launch_file.launch>
```

Here, the `<pkg_name>` is the name of your package and `<your_launch_file.launch>` is the name of the launch file with the `.launch` extension.

Now do the following:

- Quickly visit the terminal that contained the goal echo (Shell #1). A message should have appeared indicating the goal you sent (with 10 seconds of taking pictures).
- Quickly visit the feedback terminal containing the feedback echo (Shell #2). A new message should be appearing every second. This is the feedback sent by the **action server** to the **action client**.
- Quickly visit the result terminal (Shell #3). If 10 seconds have not yet passed since you launched the **action client**, then there should be no message. If you wait there until the 10 seconds pass, you will see the result message sent by the **action server** to the **action client** appear.

- End of Exercise 8.8 -

- Notes for Exercise 8.8 -

Each one of those three topics have their own type of message. The type of the message is built automatically by ROS from the **.action** file.

For example, in the case of the **ardrone_action_server** the **action file** is called **Ardrone.action**.

When you compile the package (with **catkin_make**), ROS will generate the following types of messages from the **Ardrone.action** file:

- ArdroneActionGoal
- ArdroneActionFeedback
- ArdroneActionResult

Each topic of the action server uses its associated type of message accordingly.

- End of Notes -

- Exercise 8.9 -

Do a *rostopic info* of any of the action topics and check that the types really correspond with the ones indicated above.

- End of Exercise 8.9 -

- Exercise 8.10 -

Due to the way actions work, you can actually call the *ardrone_action_server action server* publishing in the topics directly (emulating by hence, what the Python code action client is doing). It is important that you understand this because you will need this knowledge to debug your programs.

Press **[CTRL]+[C]** to kill the ardrone_as action_server.launch if you had it still running. If you had it demonized, use the command: *rosnode kill /ardrone_as*

Run the action server:

▶ **Execute in Shell #1:**

In [ ]:
```
roslaunch ardrone_as action_server.launch
```

Let's activate the *ardrone_action_server* action server through topics with the following exercise.

Use the webshell to send a goal to the */ardrone_action_server* action server, and to observe what happens in the *result* and *feedback* topics.

Send goal to the action server:

▶ **Execute in Shell #2:**

In [ ]:
```
rostopic pub /[name_of_action_server]/goal /[type_of_the_message_used_by_the_topic] [TAB][TAB]
```

- End of Exercise 8.10 -

- Expected Result for Exercise 8.10 -

You should see the same result as in exercise 8.8, with the difference that the goal has been sent by hand, publishing directly into the goal topic, instead of publishing through a Python program.

- End of Expected Result -

- Notes for Exercise 8.10 -

- You don't have to type the message by hand. Remember to use TAB-TAB to make ROS autocomplete your commands (or give you options).
- Once you achieve that, and ROS autocompletes the message you must send, you will have to modify the parameter **nseconds**, because the default value is zero (remember that parameter indicates the number of seconds taking pictures). Move to the correct place of the message using the keyboard.

- End of Notes -

8.6 The axclient

Until now, you've learnt to send goals to an Action Server using these 2 methods:

- Publishing directly into the /goal topic of the Action Server
- Publishing the goal using Python code

But, let me tell you that there's still one more method you can use in order to send goals to an Action Server, which is much easier and faster than the 2 methods you've learnt: using the **axclient**.

The axclient is, basically, a GUI tool provided by the actionlib package, that allows you to interact with an Action Server in a very easy and visual way. The command used to launch the axclient is the following:

In [ ]:
```
rosrun actionlib_tools axclient.py /<name_of_action_server>
```

Want do you think? Do you want to try it? Let's go then!

- Exercise 8.11 -

Before starting with this exercise, make sure that you have your action server running. If it is not running, the axclient won't work.

Press **[CTRL]+[C]** to kill the ardrone_as action_server.launch if you had it still running. If you had it demonized, use the command: *rosnode kill /ardrone_as*

Run the action server:

▶ **Execute in Shell #1:**

In [ ]:
```
roslaunch ardrone_as action_server.launch
```

Now, let's launch the axclient in order to send goals to the Action Server.

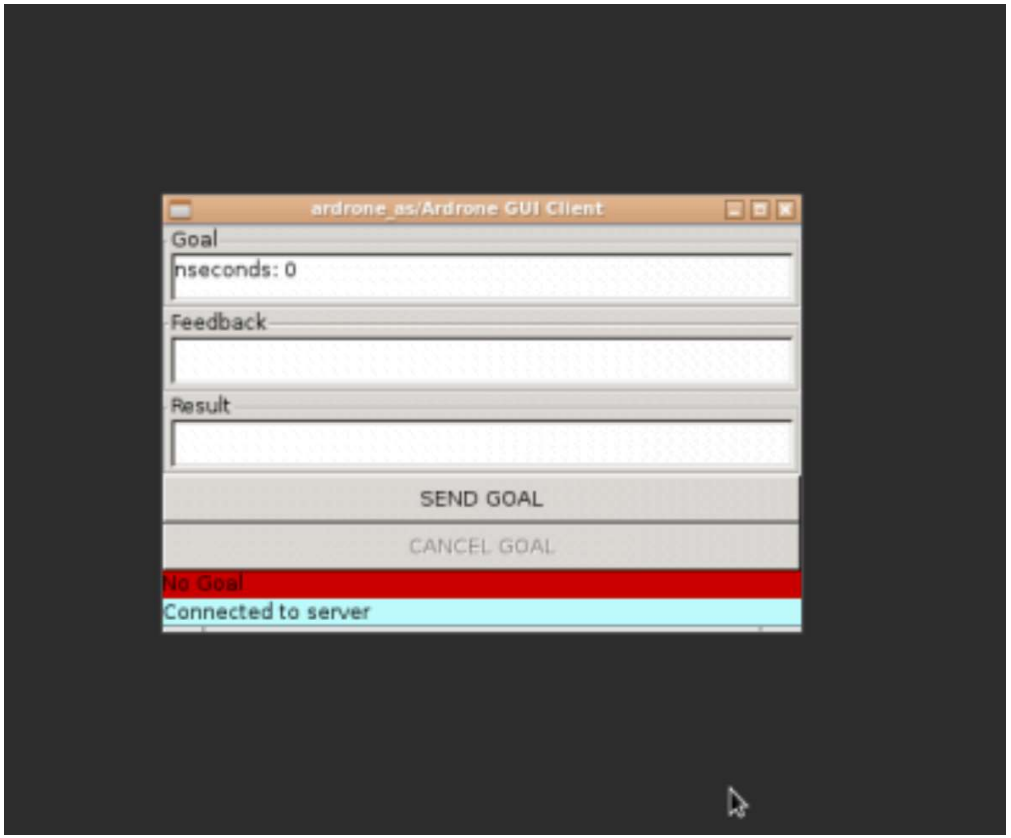
Launch axclient:

▶ **Execute in Shell #2:**

In [ ]:
```
rosrun actionlib_tools axclient.py /ardrone_action_server
```

In order to be able to visualize axclient, you will need to wait until the graphical interface window appears.

Afer a while, you should see in your screen something like this:



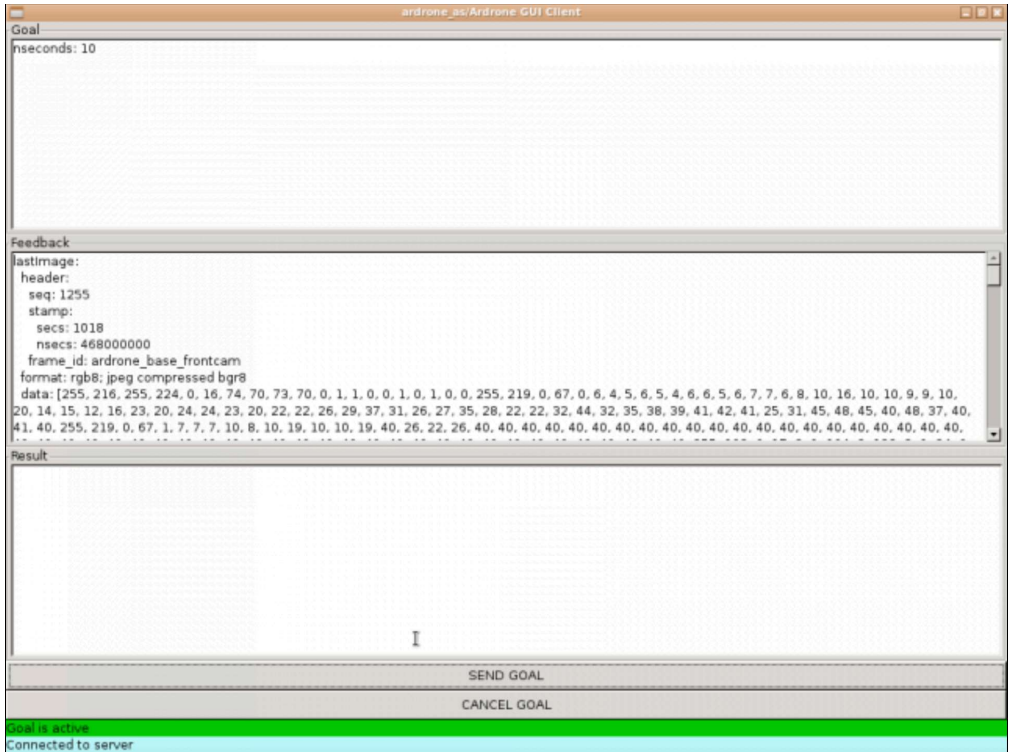
If you minimize the graphical interface window, just hit the icon with a screen and it will show again.

Now everything is settled, and you can start playing with axclient! For instance, you could send goals to the Action Server and visualize the different topics that take part in an Action, which you have learnt in this Chapter.

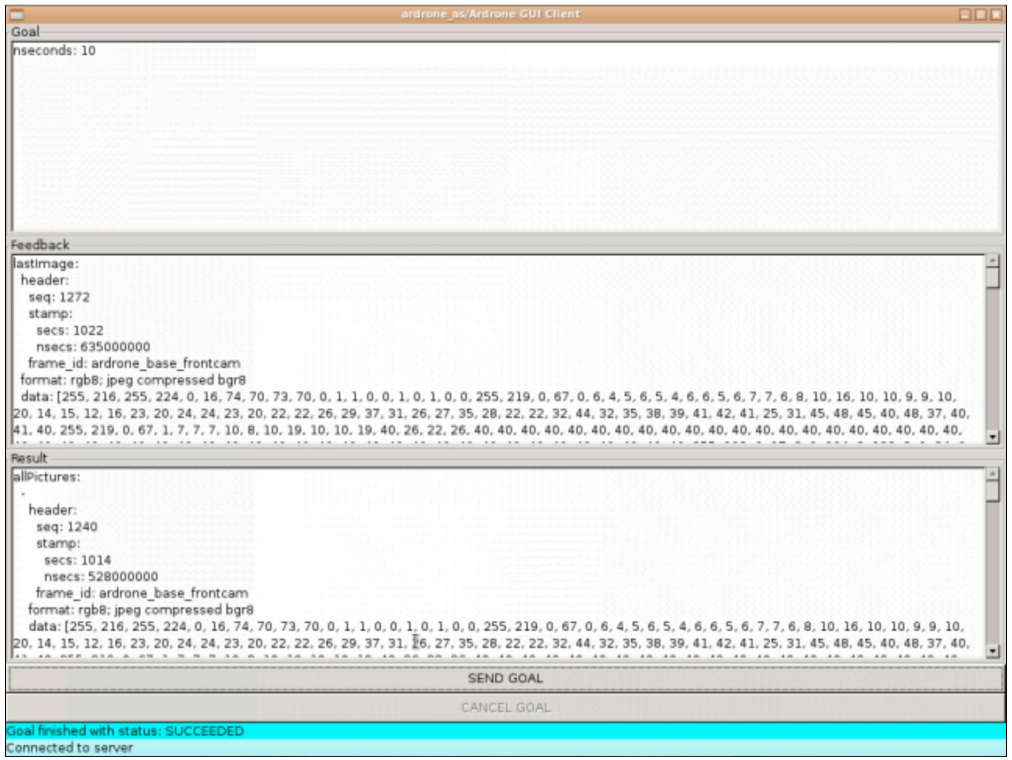
- End of Exercise 8.11 -

- Expected Result for Exercise 8.11 -

Action in process:



Action succeeded:



- End of Expected Result -

- Notes for Exercise 8.11 -

- Sometimes you may find that when you click the SEND GOAL button, or when you try to change the value of the goal you want to send, you can't interact with the axclient screen. If that's the case, just go to another tab and return again to the tab with axclient, and everything will work fine.

- End of Notes -

Maybe you're a little bit angry at us now, because we didn't show you this tool before? Don't be!!

The very simple reason why we didn't talk about this tool earlier is because we want you to learn how Actions really work inside. Once you have the knowledge, you are then ready to use the shortcuts.