# ROS Basics in 5 days

## Unit 9 ROS Actions: Servers & Messages

- Summary -

**Estimated time to completion:** 2.5 hours

**What will you learn with this unit?**

- How to create an action server
- How to build your own action message

- End of Summary -

In the previous lesson, you learned how to **CALL** an action server creating an action client. In this lesson, you are going to learn how to **CREATE** your own action server.

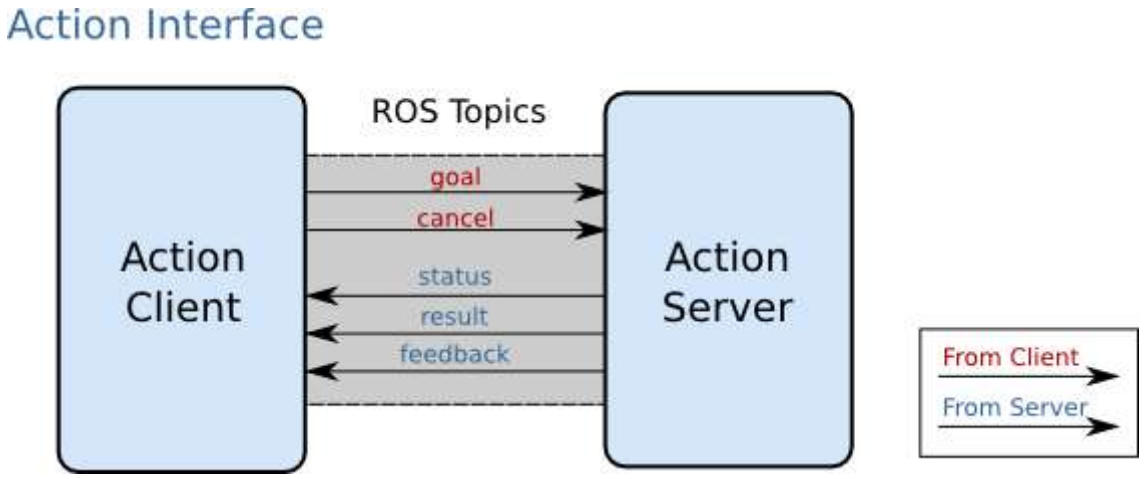


Fig.9.5 - Action Interface Diagram

## 9.1 Writing an action server

- Exercise 9.11 -

What follows is the code of an example of a ROS action server. When called, the action server will generate a Fibonacci sequence of a given order. The action server goal message must indicate the order of the sequence to be calculated, the feedback of the sequence as it is being computed, and the result of the final Fibonacci sequence.

First, create the package **my_action_server_example_pkg**, executing the following comands in the Web Terminal:

▶ Execute in Terminal #1

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

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

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

▶ Execute in Terminal #1

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

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

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

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

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

**Python Program {9.11a}: fibonacci_action_server.py**

In [1]:

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

import actionlib

from actionlib_tutorials.msg import FibonacciFeedback, FibonacciResult, FibonacciAction

class FibonacciClass(object):

  # create messages that are used to publish feedback/result
  _feedback = FibonacciFeedback()
  _result   = FibonacciResult()

  def __init__(self):
    # creates the action server
    self._as = actionlib.SimpleActionServer("fibonacci_as", FibonacciAction, self.goal_callback, False)
    self._as.start()

  def goal_callback(self, goal):
    # this callback is called when the action server is called.
    # this is the function that computes the Fibonacci sequence
    # and returns the sequence to the node that called the action server

    # helper variables
    r = rospy.Rate(1)
    success = True

    # append the seeds for the fibonacci sequence
    self._feedback.sequence = []
    self._feedback.sequence.append(0)
    self._feedback.sequence.append(1)

    # publish info to the console for the user
    rospy.loginfo('"fibonacci_as": Executing, creating fibonacci sequence of order %i with seeds %i, %i' % (
```

```
    # starts calculating the Fibonacci sequence
    fibonacciOrder = goal.order
    for i in range(1, fibonacciOrder):

      # check that preempt (cancelation) has not been requested by the action client
      if self._as.is_preempt_requested():
        rospy.loginfo('The goal has been cancelled/preempted')
        # the following line, sets the client in preempted state (goal cancelled)
        self._as.set_preempted()
        success = False
        # we end the calculation of the Fibonacci sequence
        break

      # builds the next feedback msg to be sent
      self._feedback.sequence.append(self._feedback.sequence[i] + self._feedback.sequence[i-1])
      # publish the feedback
      self._as.publish_feedback(self._feedback)
      # the sequence is computed at 1 Hz frequency
      r.sleep()

    # at this point, either the goal has been achieved (success==true)
    # or the client preempted the goal (success==false)
    # If success, then we publish the final result
    # If not success, we do not publish anything in the result
    if success:
      self._result.sequence = self._feedback.sequence
      rospy.loginfo('Succeeded calculating the Fibonacci of order %i' % fibonacciOrder )
      self._as.set_succeeded(self._result)

if __name__ == '__main__':
  rospy.init_node('fibonacci')
  FibonacciClass()
  rospy.spin()
```

And run it in a Web Terminal:

▶ Execute in Terminal #1

In [ ]:
```
rosrun my_action_server_example_pkg fibonacci_action_server.py
```

- End of Exercise 9.11 -

**Code Explanation Python Program: {9.11a}**

In this case, the action server is using an action message definition called ***Fibonacci.action***. That message has been created by ROS into its ***actionlib_tutorials*** package.

In [ ]:
```
from actionlib_tutorials.msg import FibonacciFeedback, FibonacciResult, FibonacciAction
```

Here we are importing the message objects generates by this ***Fibonacci.action*** file.

In [ ]:
```
_feedback = FibonacciFeedback()
_result   = FibonacciResult()
```

Here, we are creating the message objects that will be used for publishing the **feedback** and the **result** of the action.

In [ ]:
```
def __init__(self):
    # creates the action server
    self._as = actionlib.SimpleActionServer("fibonacci_as", FibonacciAction, self.goal_callback, False)
    self._as.start()
```

This is the constructor of the class. Inside this constructor, we are creating an Action Server that will be called **"fibonacci_as"**, that will use the Action message **FibonacciAction**, and that will have a callback function called **goal_callback**, that will be activated each time a new goal is sent to the Action Server.

In [ ]:
```
def goal_callback(self, goal):

    r = rospy.Rate(1)
    success = True
```

Here we define the **goal callback** function. Each time a new goal is sento to the Action Server, this function will be called.

In [ ]:
```
self._feedback.sequence = []
self._feedback.sequence.append(0)
self._feedback.sequence.append(1)

rospy.loginfo('"fibonacci_as": Executing, creating fibonacci sequence of order %i with seeds %i, %i' % ( goal
```

Here we are **initializing** the Fibonacci sequence, and setting up the first values (seeds) of it. Also, we print data for the user related to the Fibonacci sequence the Action Server is going to calculate.

In [ ]:
```
fibonacciOrder = goal.order
for i in range(1, fibonacciOrder):
```

Here, we start a loop that while go until the **goal.order** value is reached. This value is, obviously, the order of the Fibonacci sequence that the user has sent from the Action Client.

In [ ]:
```
if self._as.is_preempt_requested():
    rospy.loginfo('The goal has been cancelled/preempted')
    # the following line, sets the client in preempted state (goal cancelled)
    self._as.set_preempted()
    success = False
    # we end the calculation of the Fibonacci sequence
    break
```

We check if the goal has been cancelled (preempted). Remember you saw how to to preempt a goal in the previous Chapter.

In [ ]:
```
self._feedback.sequence.append(self._feedback.sequence[i] + self._feedback.sequence[i-1])
self._as.publish_feedback(self._feedback)
r.sleep()
```

Here, we are actually calculating the values of the Fibonacci sequence. You can check how a Fibonacci sequence is calculated here: Fibonacci sequence (https://en.wikipedia.org/wiki/Fibonacci_number). Also, we keep publishing **feedback** each time a new value of the sequence is calculated.

In [ ]:
```
if success:
  self._result.sequence = self._feedback.sequence
  rospy.loginfo('Succeeded calculating the Fibonacci of order %i' % fibonacciOrder )
  self._as.set_succeeded(self._result)
```

If everything went OK, we publish the **result**, which is the whole Fibonacci sequence, and we set the Action as succedeed using the **set_succeeded()** function.

**End Code Explanation Python Program: {9.11a}**

- Exercise 9.12a -

Check the structure of the Fibonacci.action message definition by visiting the ***action*** directory of the ***actionlib_tutorials*** package.

- End of Exercise 9.12a -

- Exercise 9.12b -

Launch again the python code above {9.11a} to have the Fibonacci server running.

Then, execute the following commands in their corresponding Terminals.

Echo the result topic:

▶ Execute in Terminal #2:

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

Echo the feedback topic

▶ Execute in Terminal #3:

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

Manually send the goal to your Fibonacci server, publishing directly to the topic (as you learned in the previous chapter).

▶ Execute in Terminal #4

In [ ]:
```
rostopic pub /fibonacci_as/goal actionlib_tutorials/FibonacciActionGoal [TAB][TAB]
```

- End of Exercise 9.12b -

- Expected Result for Exercise 9.12b -

After having called the action, the feedback topic should be publishing the feedback, and the result once the calculations are finished.

- End of Expected Result -

- Notes for Exercise 9.12b -

- You must be aware that the name of the messages (the class) used in the Python code are called `FibonacciGoal`, `FibonacciResult`, and `FibonacciFeedback`, while the name of the messages used in the topics are called `FibonacciActionGoal`, `FibonacciActionResult`, and `FibonacciActionFeedback`.

  Do not worry about that, just bear it in mind and use it accordingly.

- End of Notes -

- Exercise 9.13 -

a) Create a package with an action server that makes the drone move in a square when called.

b) Call the action server through the topics and observe the result and feedback.

c) Base your code in the previous Fibonacci example {9.11a} and the client you did in Exercice 8.6 that moved the drone while taking pictures.

- End of Exercise 9.13 -

- Expected Result for Exercise 9.13 -

The result must show the AR.Drone doing a square in the air when the action server is called, as shown in the animation beneath {Fig:9.6}

Fig.9.6 - AR.Drone moved through commands sent by a custom action server

- End of Expected Result -

- Notes for Exercise 9.13 -

- The size of the side of the square should be specified in the goal message as an integer.
- The feedback should publish the current side (as a number) the robot is at while doing the square.
- The result should publish the total number of seconds it took the drone to do the square
- Use the *Test.action* message for that action server. Use the Terminal command *find /opt/ros/noetic/ -name Test.action* to find where that message is defined. Then, analyze the different fields of the msg in order to learn how to use it in your action server. As you can see its in the package **actionlib**.

- End of Notes -

## 9.2 How to create your own action server message

**It is always recommended that you use the action messages already provided by ROS.** These can be found in the following ROS packages:

- actionlib
  - Test.action
  - TestRequest.action
  - TwoInts.action
- actionlib_tutorials
  - Fibonacci.action
  - Averaging.action

However, it may happen that you need to create your own type. Let's learn how to do it.

To create your own custom action message you have to:

1.- Create an **action** directory within your package.

2.- Create your **Name.action** action message file.

- The Name of the action message file will determine later the name of the classes to be used in the **action server** and/or **action client**. ROS convention indicates that the name has to be camel-case.
- Remember the Name.action file has to contain three parts, each part separated by three hyphens.

In [ ]:
```
#goal
package_where_message_is/message_type goal_var_name
---
#result
package_where_message_is/message_type result_var_name
---
#feedback
package_where_message_is/message_type feedback_var_name
```

- If you do not need one part of the message (for example, you don't need to provide feedback), then you can leave that part empty. But you **must always specify the hyphen separtors**.

3.- Modify the file CMakeLists.txt and the package.xml to include action message compilation. Read the detailed description below.

## 9.3 Prepare CMakeLists.txt and package.xml files for custom action messages compilation

You have to edit two files in the package, in the same way that we explained for topics and services:

- CMakeLists.txt
- package.xml

### 9.3.1 Modification of CMakeLists.txt

You will have to edit four functions inside CMakeLists.txt:

- find_package()
- add_action_files()
- generate_messages()
- catkin_package()

## I. find_package()

All of the packages needed to COMPILE the messages of topic, services, and actions go here.
In *package.xml*, you have to state them as built.

In [ ]:
```
find_package(catkin REQUIRED COMPONENTS
      # your packages are listed here
      actionlib_msgs
)
```

## II. add_action_files()

This function will contain all of the action messages from this package (which are stored in the **action** folder) that need to be compiled.
Place them beneath the FILES tag.

In [ ]:
```
add_action_files(
      FILES
      Name.action
)
```

## III. generate_messages()

The packages needed for the action messages compilation are imported here. Write the same here as you wrote in the find_package.

In [ ]:
```
generate_messages(
      DEPENDENCIES
      actionlib_msgs
      # Your packages go here
)
```

## IV. catkin_package()

State here all of the packages that will be needed by someone that executes something from your package.
All of the packages stated here must be in the **package.xml** file as **<exec_depend>**.

In [ ]:
```
catkin_package(
      CATKIN_DEPENDS
      rospy
      # Your package dependencies go here
)
```

Summarizing, You should end with a **CMakeLists.txt** file similar to this:

CMakeLists.txt

In [ ]:

```
cmake_minimum_required(VERSION 2.8.3)
project(my_custom_action_msg_pkg)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  std_msgs
  actionlib_msgs
)

## Generate actions in the 'action' folder
add_action_files(
   FILES
   Name.action
 )

## Generate added messages and services with any dependencies listed here
generate_messages(
   DEPENDENCIES
   std_msgs actionlib_msgs
 )

catkin_package(
 CATKIN_DEPENDS rospy
)

## Specify additional locations of header files
## Your package locations should be listed before other locations
# include_directories(include)
include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

### 9.3.2 Modification of package.xml

1.- Add all of the packages needed to compile the messages.

If, for example, one of your variables in the **.action** file uses a message defined outside the std_msgs package, let's say **nav_msgs/Odometry**, you will need to import it. To do so, you would have to add as **<build_depend>** the **nav_msgs** package, adding the following line:

In [ ]:
```
<build_depend>nav_msgs<build_depend>
```

2.- On the other hand, if you need a package for the execution of the programs inside your package, you will have to import those packages as **<exec_depend>**, adding the following line:

In [ ]:
```
<build_export_depend>nav_msgs<build_export_depend>
<exec_depend>nav_msgs<exec_depend>
```

When you compile custom action messages, it's **mandatory** to add the **actionlib_msgs** as build_dependency.

In [ ]:
```
<build_depend>actionlib_msgs</build_depend>
```

When you use Python, it's **mandatory** to add the **rospy** as **run_dependency**.

In [ ]:
```
<build_export_depend>rospy<build_export_depend>
<exec_depend>rospy<exec_depend>
```

This is due to the fact that the rospy python module is needed in order to run all of your python ROS code.

Summarizing, you should end with a package.xml file similar to this:

package.xml

In [ ]:

```
<?xml version="1.0"?>
<package format="2">
  <name>my_custom_action_msg_pkg</name>
  <version>0.0.0</version>
  <description>The my_custom_action_msg_pkg package</description>
  <maintainer email="user@todo.todo">user</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>actionlib</build_depend>
  <build_depend>actionlib_msgs</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_export_depend>actionlib</build_export_depend>
  <build_export_depend>actionlib_msgs</build_export_depend>
  <build_export_depend>rospy</build_export_depend>
  <exec_depend>actionlib</exec_depend>
  <exec_depend>actionlib_msgs</exec_depend>
  <exec_depend>rospy</exec_depend>

  <export>
  </export>
</package>
```

Finally, when everything is correctly set up, you just have to compile:

▶ Execute in Terminal #1

In [ ]:

```
roscd; cd ..
```

In [ ]:

```
catkin_make
```

In [ ]:
```
source devel/setup.bash
```

In [ ]:
```
rosmsg list | grep Name
```

You will get an output to the last command, similar to this:

Output in Terminal #1

```
my_custom_action_msg_pkg/NameAction
my_custom_action_msg_pkg/NameActionFeedback
my_custom_action_msg_pkg/NameActionGoal
my_custom_action_msg_pkg/NameActionResult
my_custom_action_msg_pkg/NameFeedback
my_custom_action_msg_pkg/NameGoal
my_custom_action_msg_pkg/NameResult
```

- Notes -

Note that you haven't imported the **std_msgs** package anywhere. But you can use the messages declared there in your custom .actions. That's because this package forms part of the roscore file systems, so therefore, it's embedded in the compilation protocols, and no declaration of use is needed.

- End of Notes -

# 9.4 Actions Quiz


For evaluating this Quiz, we will ask you to perform different tasks. For each task, very **specific instructions** will be provided: name of the package, names of the launch files and Python scripts, topic names to use, etc.

It is **VERY IMPORTANT** that you strictly follow these instructions, since they will allow our automated correction system to properly test your Quiz, and assign a score to it. If the names you use are different from the ones specified in the exam instructions, your exercise will be marked as **FAILED**, even though it works correctly.

## Create a Package with an action server with custom action message to move ardone

- The new action server will receive two words as a goal: TAKEOFF or LAND.
- When the action server receives the TAKEOFF word, the drone will take off. **The drone must remain in the take off position** until another command is sent to land it.
- When the action server receives the LAND word, the drone will land.
- As a feedback, it publishes once a second what action is taking place (taking off or landing).
- When the action finishes, the result will return nothing.

**Useful Data for the Quiz:**

- You need to create a new action message with the following structure:

In [ ]:
```
string goal
---
---
string feedback
```

Specifications

- The name of the package where you'll place all the code related to the Quiz will be **actions_quiz**.
- The name of the launch file that will start your Action Server will be **action_custom_msg.launch**.
- The name of the action will be **/action_custom_msg_as**.
- The name of your Action message file will be **CustomActionMsg.action**.
- Before correcting your Quiz, make sure that all your Python scripts are executalbe. They need to have full execution permissions in order to be executed by our autocorrection system. You can give them full execution permissions with the following command:

▶ **Execute in Terminal #1**

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

- Before correcting your Quiz, make sure you have terminated all the programs in your Web Terminals.

- Before correcting your Quiz, make sure that the drone is landed (not flying).

Grading Guide

The following will be checked, in order. *If a step fails, the steps following are skipped.*

1. Does the package exist?
2. Did the package compile successfully?
3. Did the action custom message compile successfully?
4. Was the action server started with the launch file?
5. Are all the expected action topics present (the five topics)?
6. Did the action server take the drone off, when the `TAKEOFF` goal is specified? Did it land the drone when the `LAND` goal was specified?

The grader will provide as much feedback on any failed step so you can make corrections where necessary.

## Quiz Correction

When you have finished the Quiz, you can correct it in order to get a Mark. For that, just click on the following button at the top of this Notebook.

Final Mark

In case you fail the Quiz, or you don't get the desired mark, do not get frustrated! You will have the chance to resend the Quiz in order to improve your score.

# You should finish PART III of the rosject before attempting next unit of this course!

English proofread