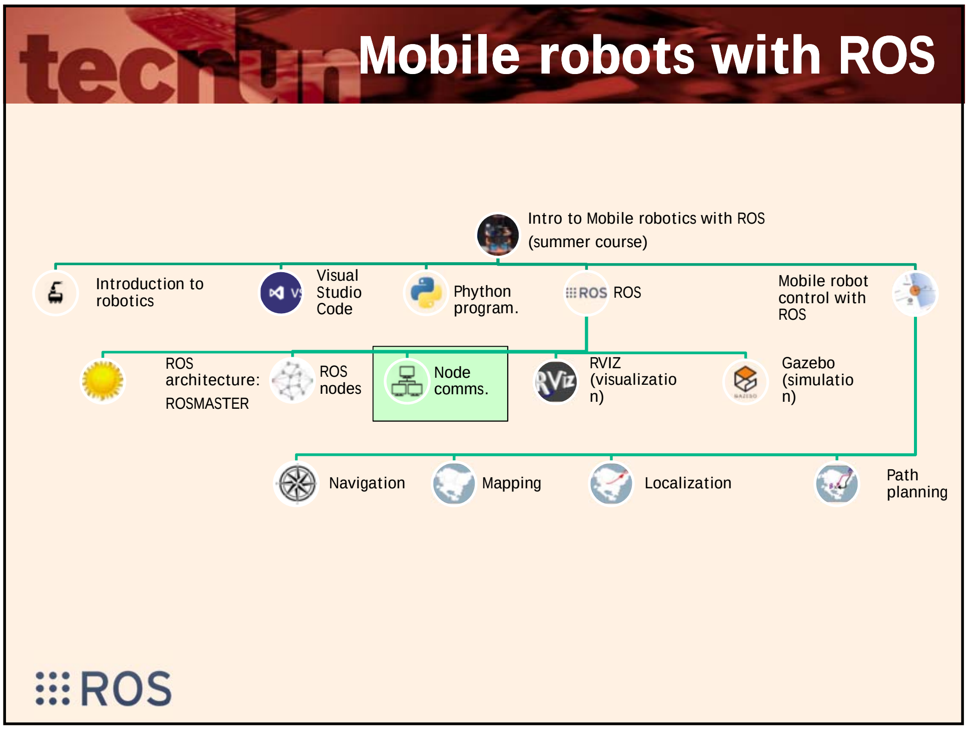
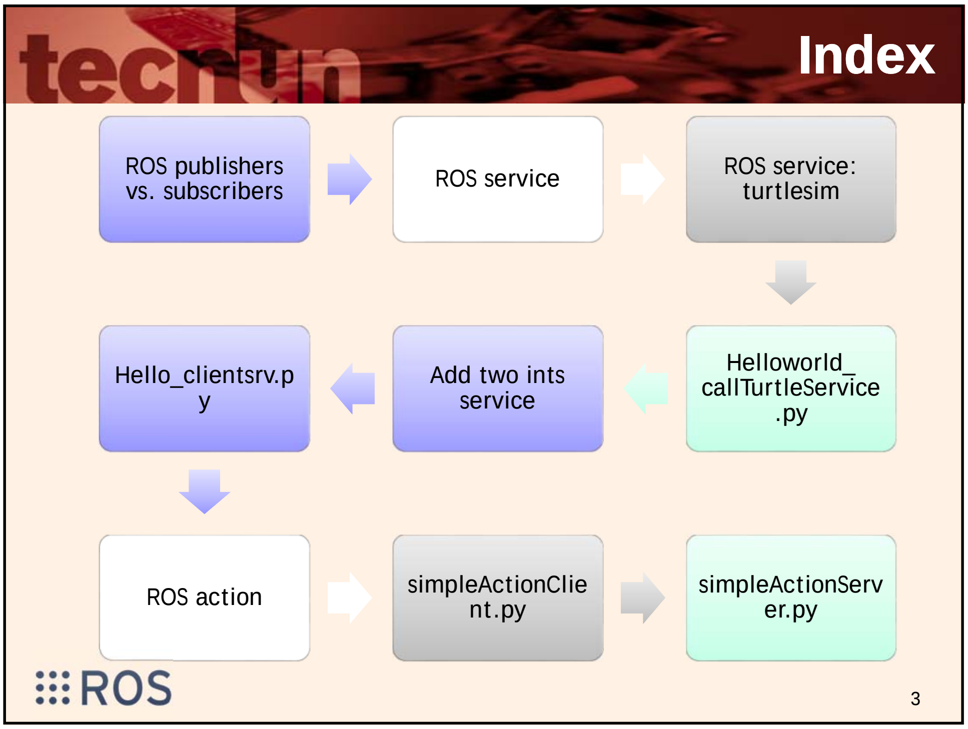
ROS service, ROS clients, ROS actions

Dr. Emilio Sánchez Tapia



Mobile robots with ROS



Index

ROS publishers vs. subscribers

ROS service

ROS service: turtlesim

Helloworld_callTurtleService.py

Add two ints service

Hello_clientsrv.py

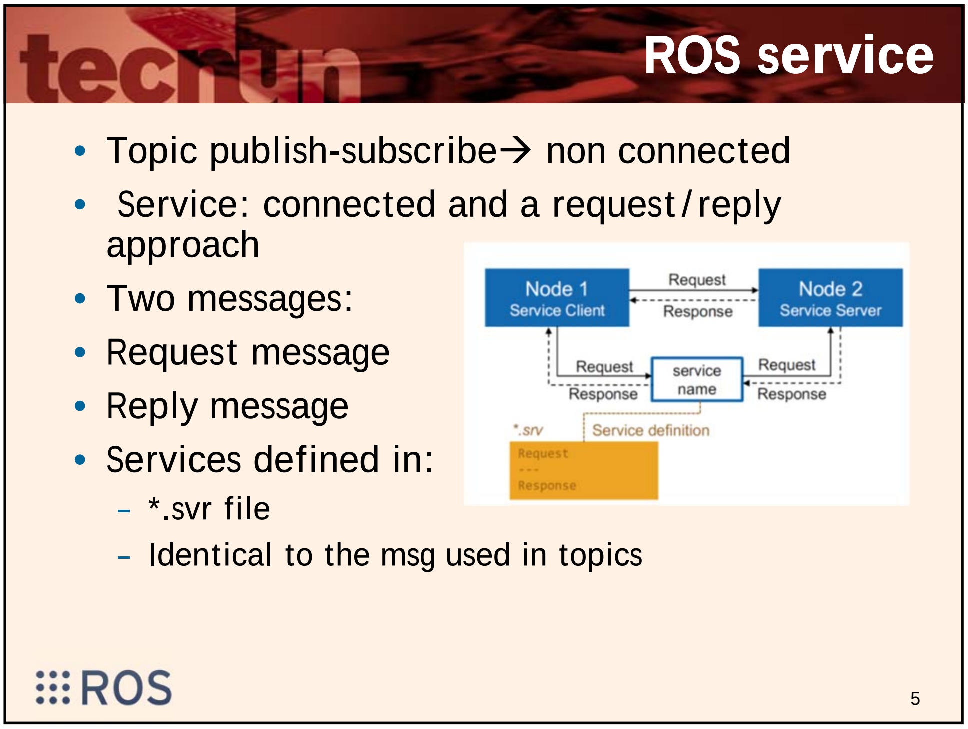
ROS action

simpleActionClient.py

simpleActionServer.py

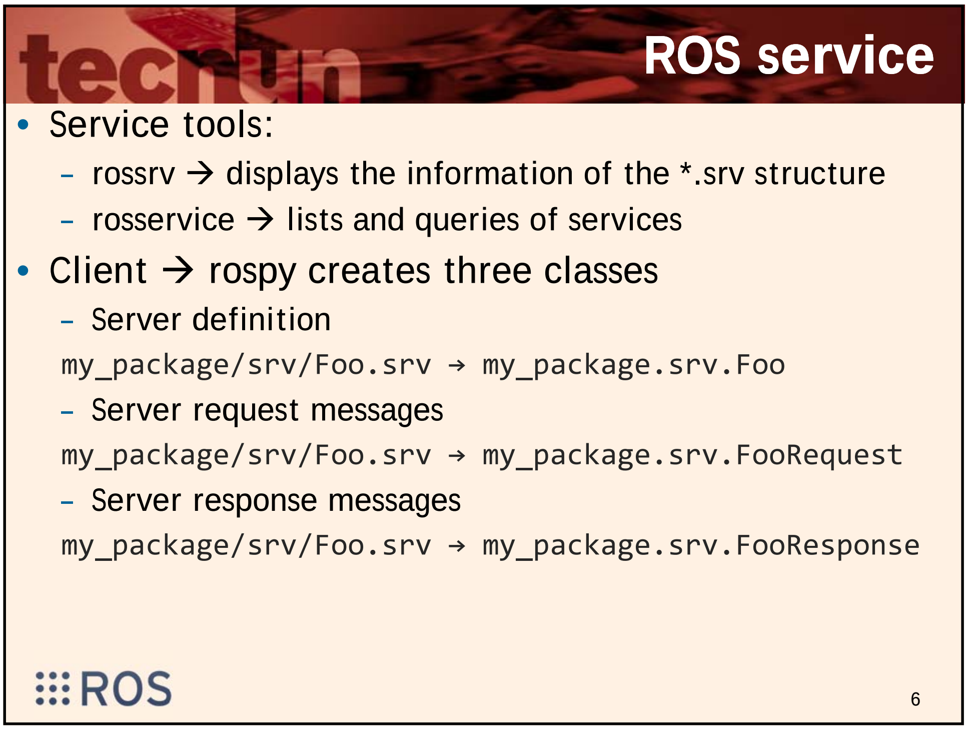
3

ROS service



ROS service

- Topic publish-subscribe→ non connected
- Service: connected and a request/reply approach
- Two messages:
- Request message
- Reply message
- Services defined in:
  - *.svr file
  - Identical to the msg used in topics



ROS service

- Service tools:
  - rossrv → displays the information of the *.srv structure
  - rosservice → lists and queries of services
- Client → rospy creates three classes
  - Server definition

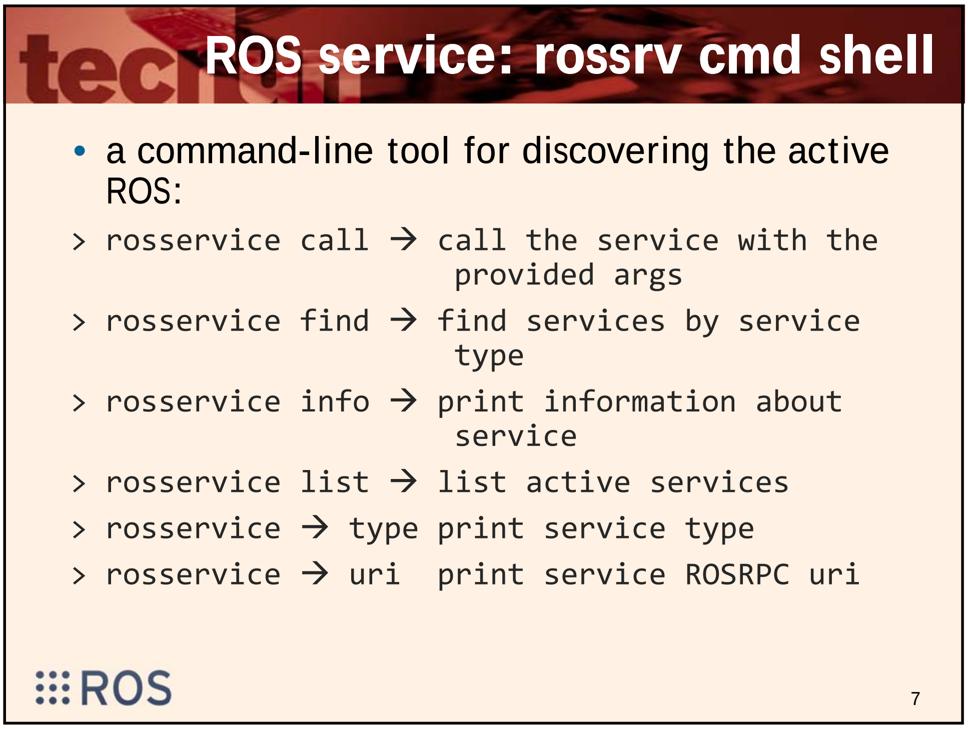
```
my_package/srv/Foo.srv → my_package.srv.Foo
```

  - Server request messages

```
my_package/srv/Foo.srv → my_package.srv.FooRequest
```

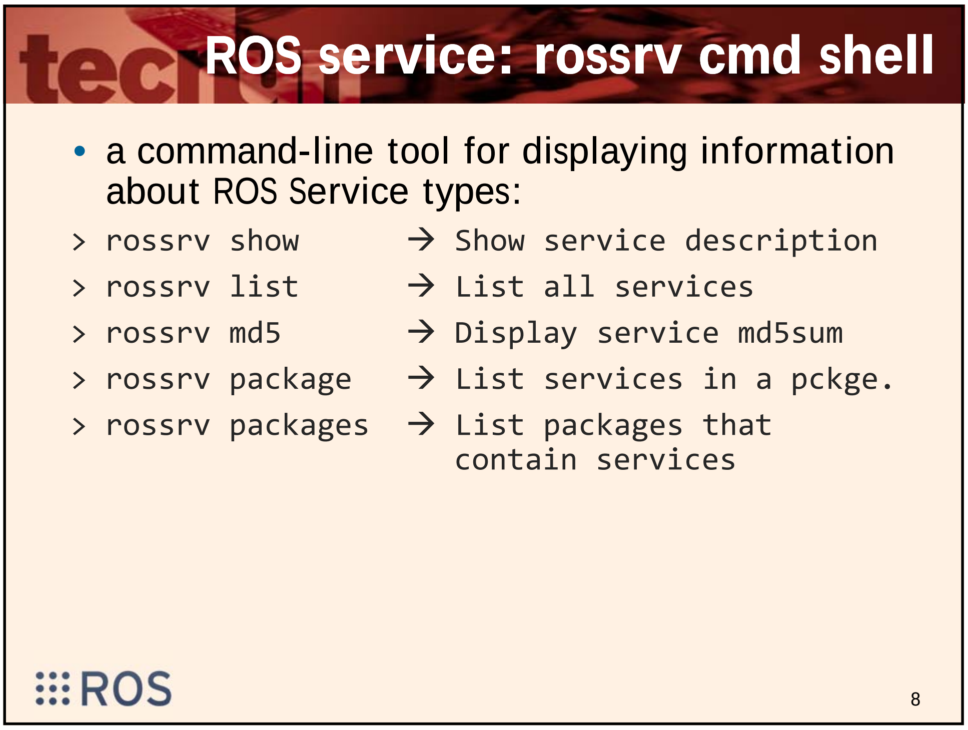
  - Server response messages

```
my_package/srv/Foo.srv → my_package.srv.FooResponse
```



ROS service: rossrv cmd shell

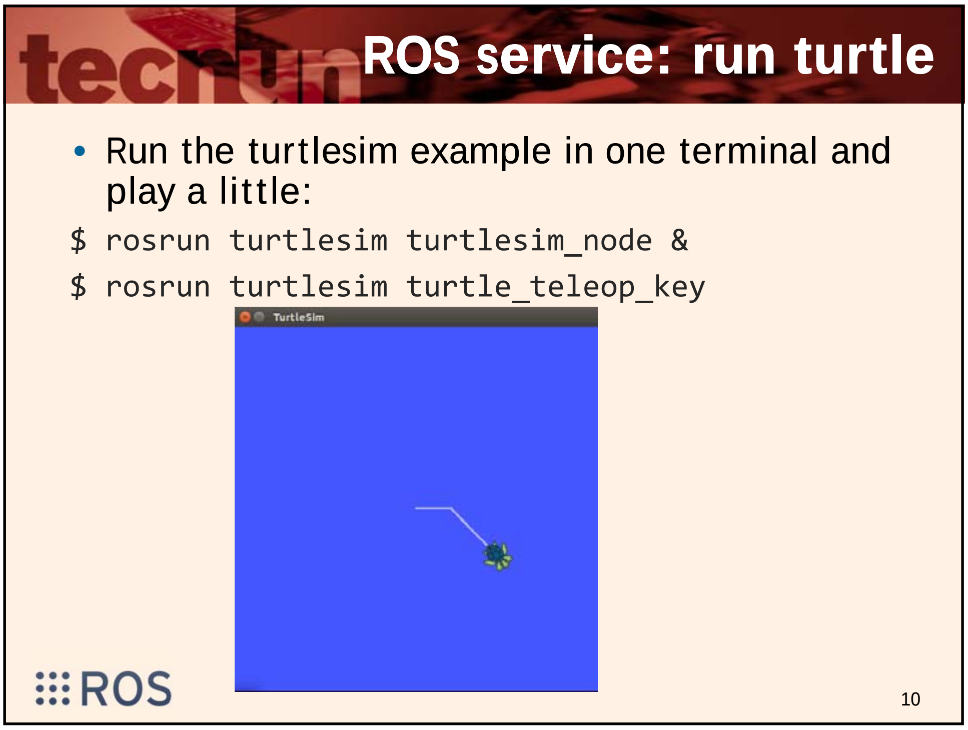
- a command-line tool for discovering the active ROS:
  - > rosservice call → call the service with the provided args
  - > rosservice find → find services by service type
  - > rosservice info → print information about service
  - > rosservice list → list active services
  - > rosservice → type print service type
  - > rosservice → uri print service ROSRPC uri



ROS service: rossrv cmd shell

- a command-line tool for displaying information about ROS Service types:
  - > rossrv show → Show service description
  - > rossrv list → List all services
  - > rossrv md5 → Display service md5sum
  - > rossrv package → List services in a pckge.
  - > rossrv packages → List packages that contain services

ROS service: turtlesim example

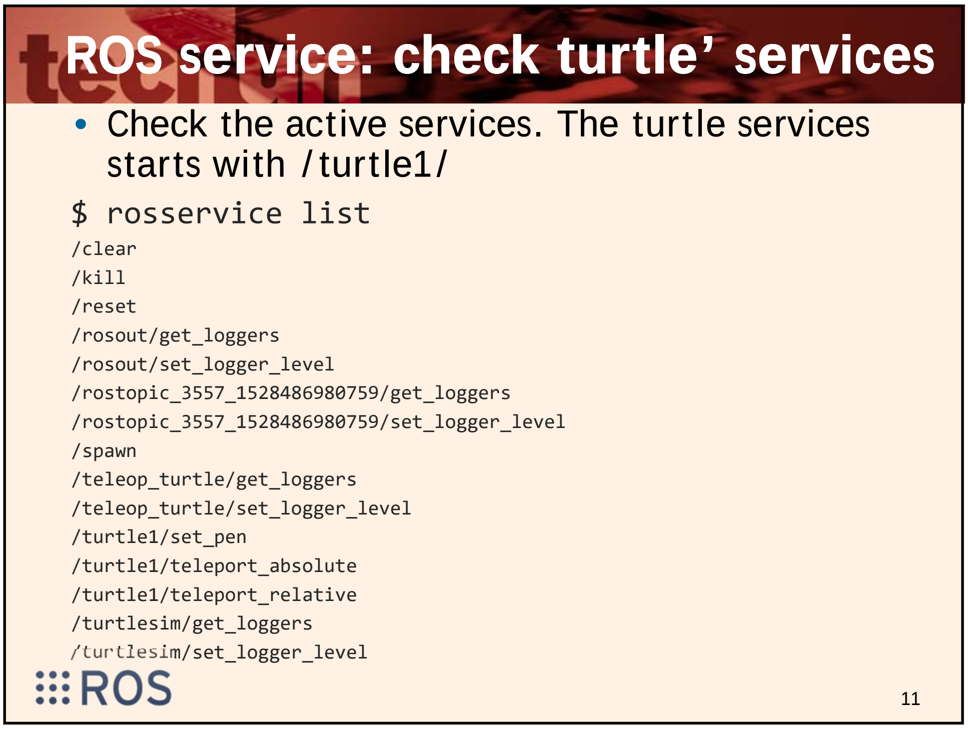


ROS service: run turtle

- Run the turtlesim example in one terminal and play a little:

```
$ rosrun turtlesim turtlesim_node &
$ rosrun turtlesim turtle_teleop_key
```

10



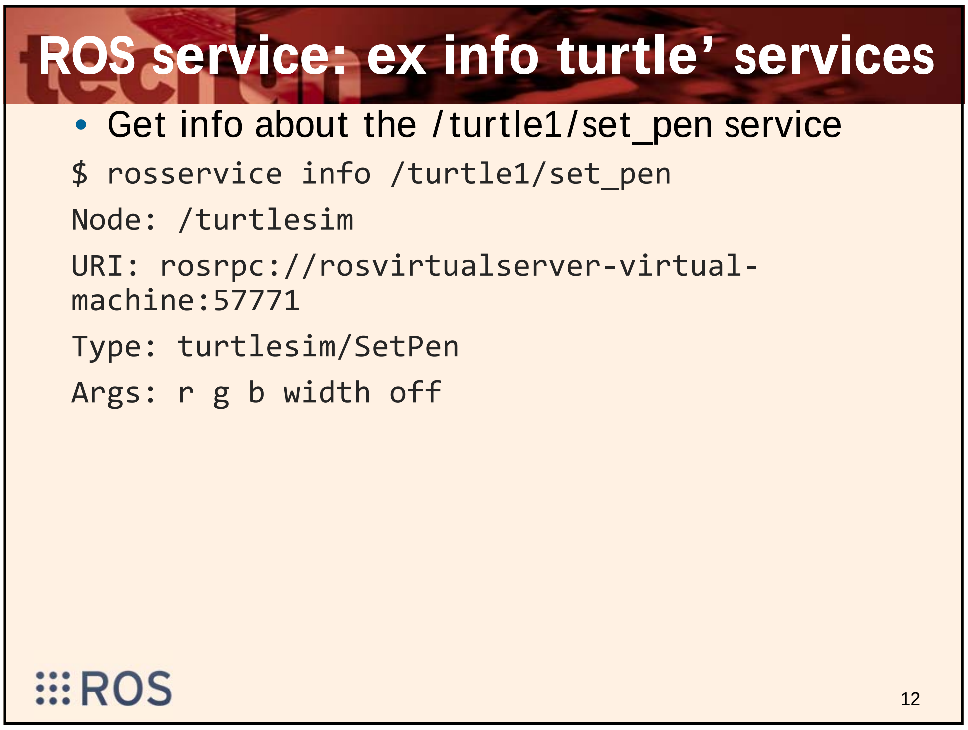
ROS service: check turtle' services

- Check the active services. The turtle services starts with /turtle1/

```
$ rosservice list
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/rostopic_3557_1528486980759/get_loggers
/rostopic_3557_1528486980759/set_logger_level
/spawn
/teleop_turtle/get_loggers
/teleop_turtle/set_logger_level
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

ROS

11



ROS service: ex info turtle' services

- Get info about the /turtle1/set_pen service

```
$ rosservice info /turtle1/set_pen
Node: /turtlesim
URI: rosrpc://rosvirtualserver-virtual-
machine:57771
Type: turtlesim/SetPen
Args: r g b width off
```

ROS

12

ROS service: type of msg of services

- Get info about the turtlesim/SetPen service type

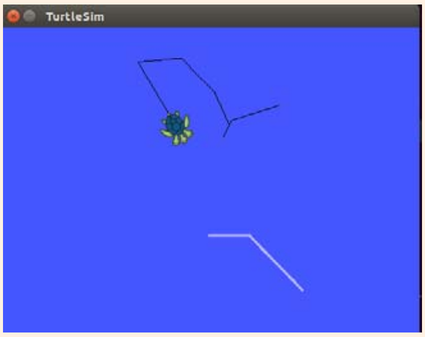
```
$ rossrv info turtlesim/SetPen
uint8 r
uint8 g
uint8 b
uint8 width
uint8 off
```

ROS service: use turtle’ services

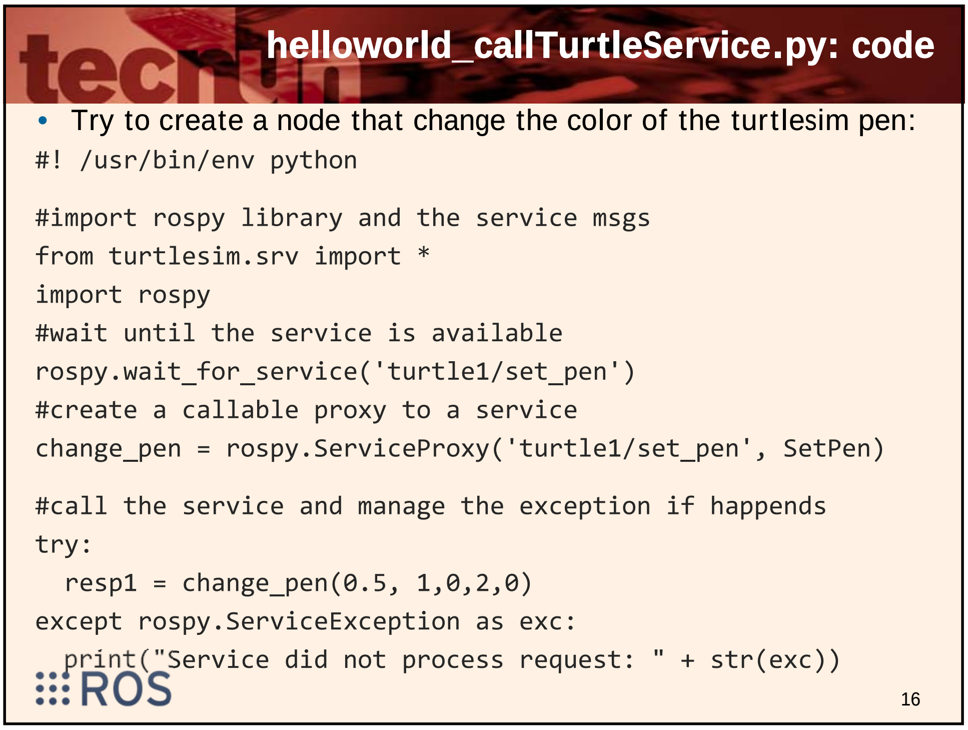
- The type of service of service /turtle1/Set_Pen is turtlesim/SetPen
- We can use this service to change the color and the width of the line drawn by the turtle!

```
$ rosservice call /turtle1/set_pen 1 0 0 1 0
```

- Play again with the turtle



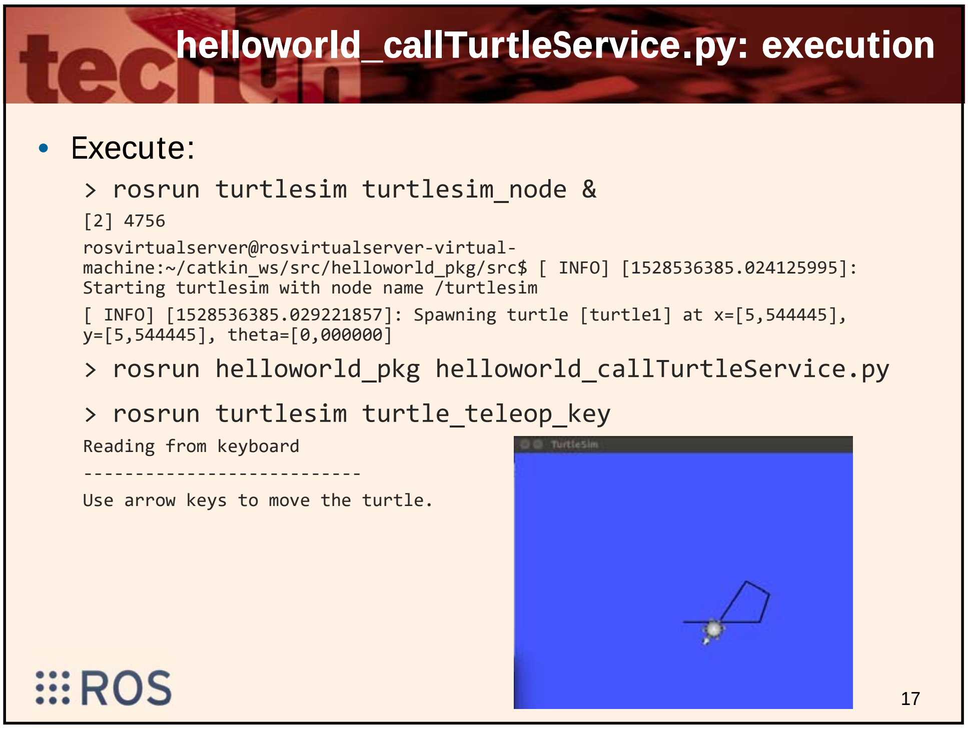
Helloworld_callTurtleService.py



helloworld_callTurtleService.py: code

- Try to create a node that change the color of the turtlesim pen:

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

#import rospy library and the service msgs
from turtlesim.srv import *
import rospy
#wait until the service is available
rospy.wait_for_service('turtle1/set_pen')
#create a callable proxy to a service
change_pen = rospy.ServiceProxy('turtle1/set_pen', SetPen)

#call the service and manage the exception if happends
try:
  resp1 = change_pen(0.5, 1,0,2,0)
except rospy.ServiceException as exc:
  print("Service did not process request: " + str(exc))
```



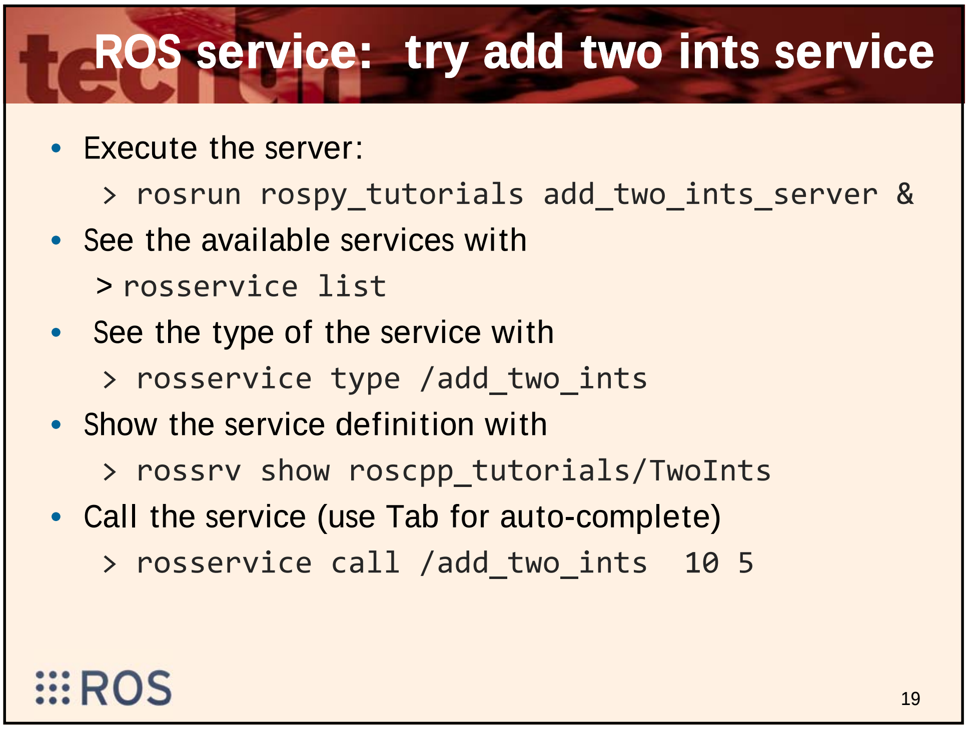
helloworld_callTurtleService.py: execution

- Execute:

```
> rosrun turtlesim turtlesim_node &
[2] 4756
rosvirtualserver@rosvirtualserver-virtual-
machine:~/catkin_ws/src/helloworld_pkg/src$ [ INFO] [1528536385.024125995]:
Starting turtlesim with node name /turtlesim
[ INFO] [1528536385.029221857]: Spawning turtle [turtle1] at x=[5,544445],
y=[5,544445], theta=[0,000000]
> rosrun helloworld_pkg helloworld_callTurtleService.py
> rosrun turtlesim turtle_teleop_key
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
```

17

Add two ints service



ROS service: try add two ints service

- Execute the server:

```
> rosrun rospy_tutorials add_two_ints_server &
```

- See the available services with

```
> rosservice list
```

- See the type of the service with

```
> rosservice type /add_two_ints
```

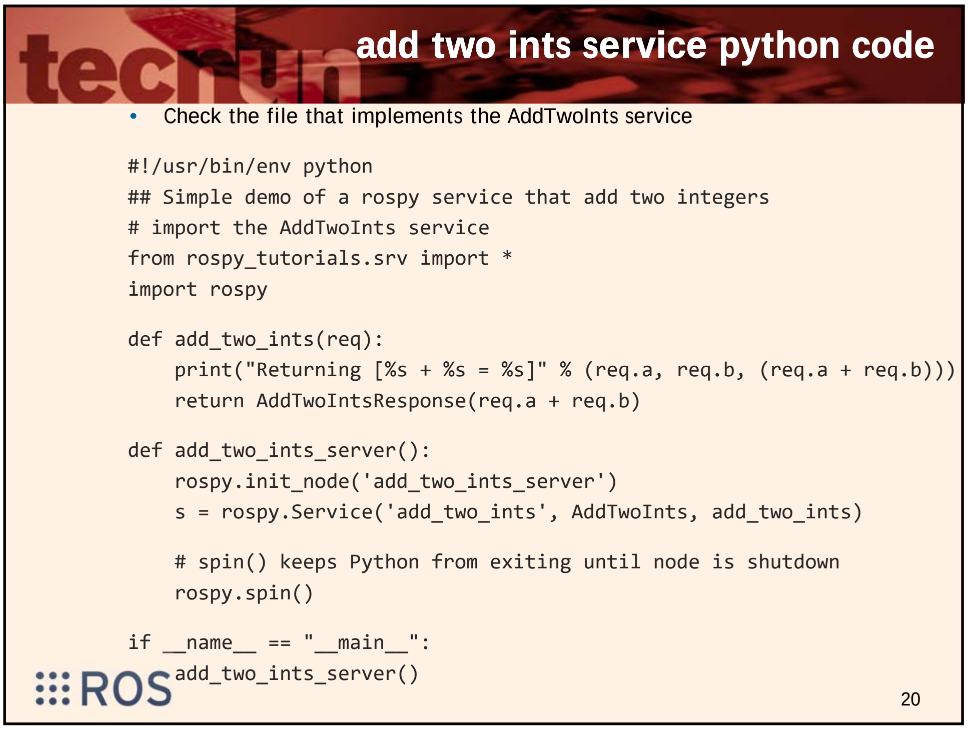
- Show the service definition with

```
> rossrv show roscpp_tutorials/TwoInts
```

- Call the service (use Tab for auto-complete)

```
> rosservice call /add_two_ints  10 5
```

ROS



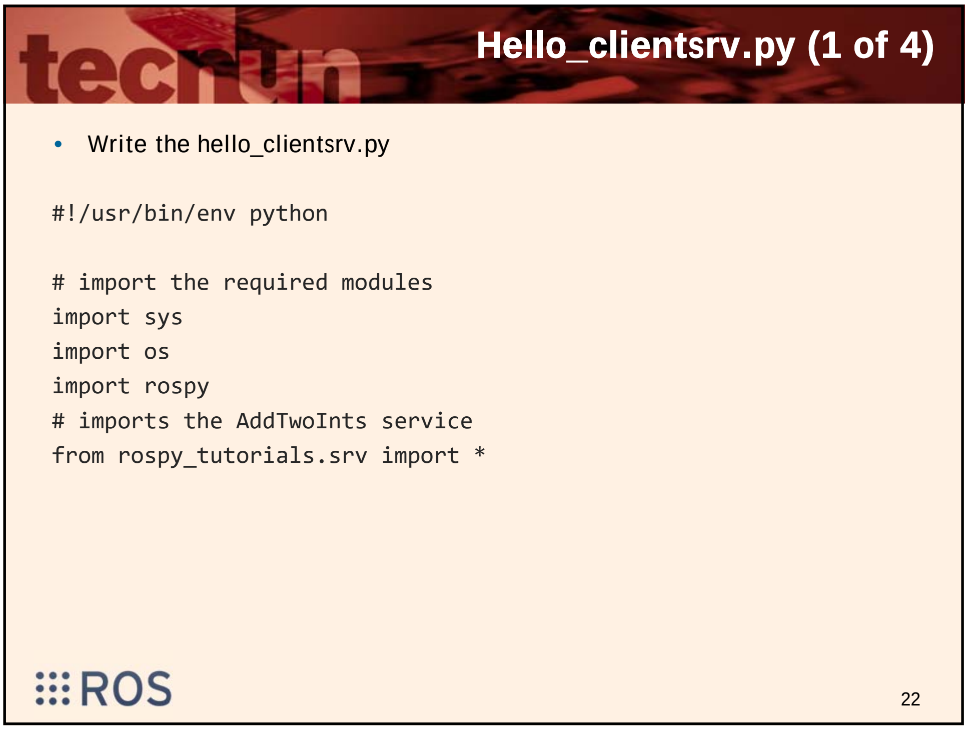
add two ints service python code

- Check the file that implements the AddTwoInts service

```
#!/usr/bin/env python
## Simple demo of a rospy service that add two integers
# import the AddTwoInts service
from rospy_tutorials.srv import *
import rospy

def add_two_ints(req):
    print("Returning [%s + %s = %s]" % (req.a, req.b, (req.a + req.b)))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, add_two_ints)

    # spin() keeps Python from exiting until node is shutdown
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

ROS

Hello_clientsrv.py

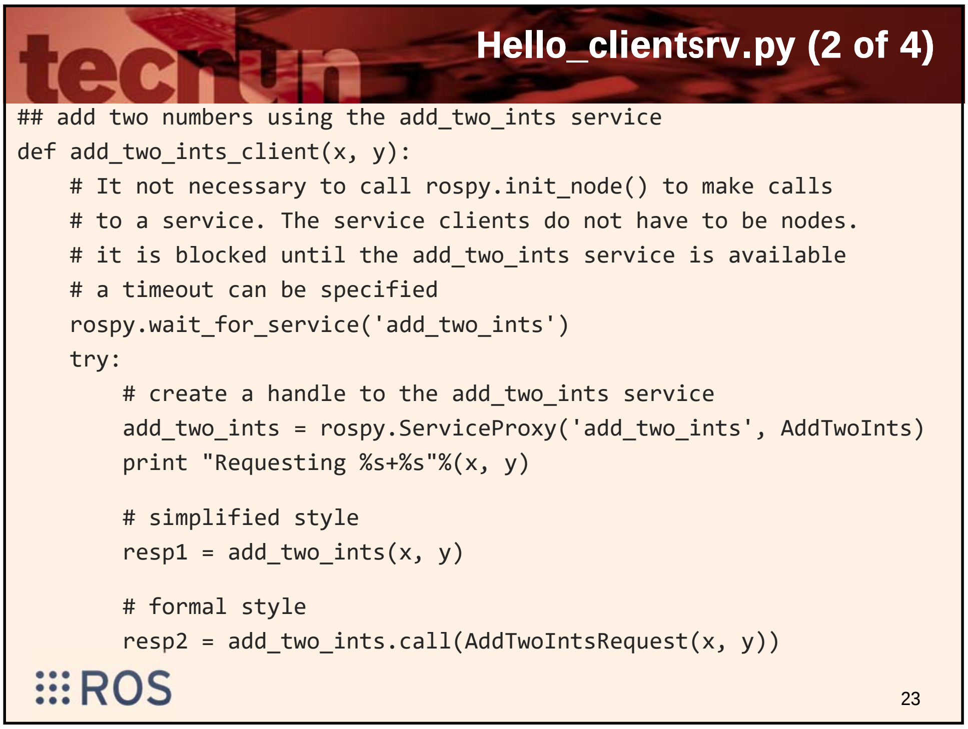


Hello_clientsrv.py (1 of 4)

- Write the hello_clientsrv.py

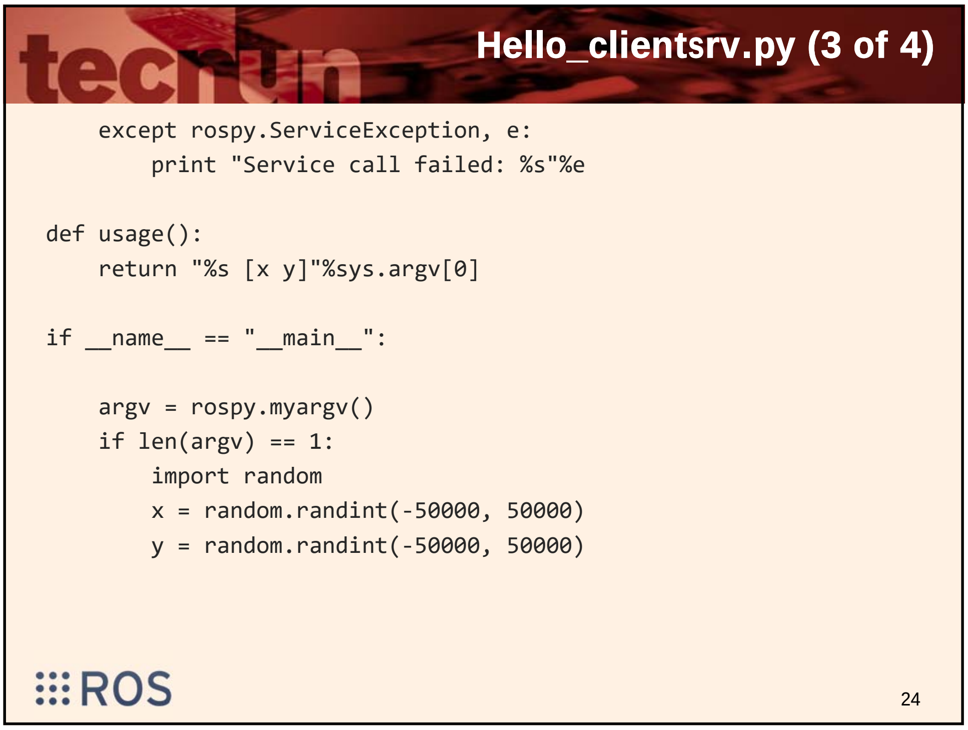
```
#!/usr/bin/env python

# import the required modules
import sys
import os
import rospy
# imports the AddTwoInts service
from rospy_tutorials.srv import *
```

tecnun
ROS
22



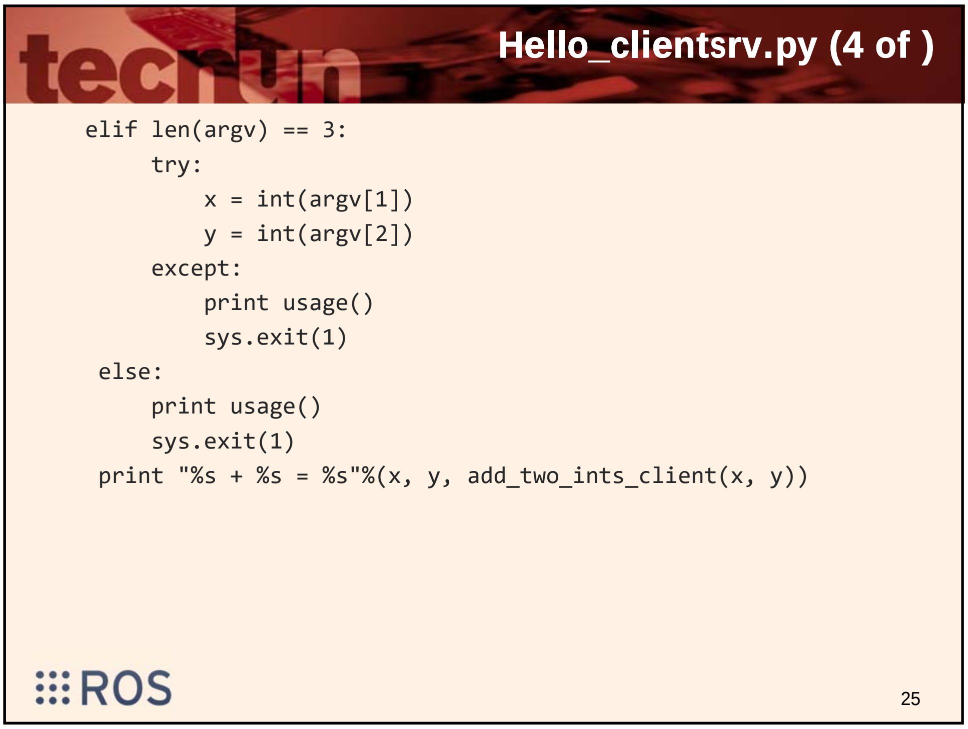
Hello_clientsrv.py (2 of 4)

```
## add two numbers using the add_two_ints service
def add_two_ints_client(x, y):
    # It not necessary to call rospy.init_node() to make calls
    # to a service. The service clients do not have to be nodes.
    # it is blocked until the add_two_ints service is available
    # a timeout can be specified
    rospy.wait_for_service('add_two_ints')
    try:
        # create a handle to the add_two_ints service
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        print "Requesting %s+%s"%(x, y)

        # simplified style
        resp1 = add_two_ints(x, y)

        # formal style
        resp2 = add_two_ints.call(AddTwoIntsRequest(x, y))
```



Hello_clientsrv.py (3 of 4)

```
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":

    argv = rospy.myargv()
    if len(argv) == 1:
        import random
        x = random.randint(-50000, 50000)
        y = random.randint(-50000, 50000)
```

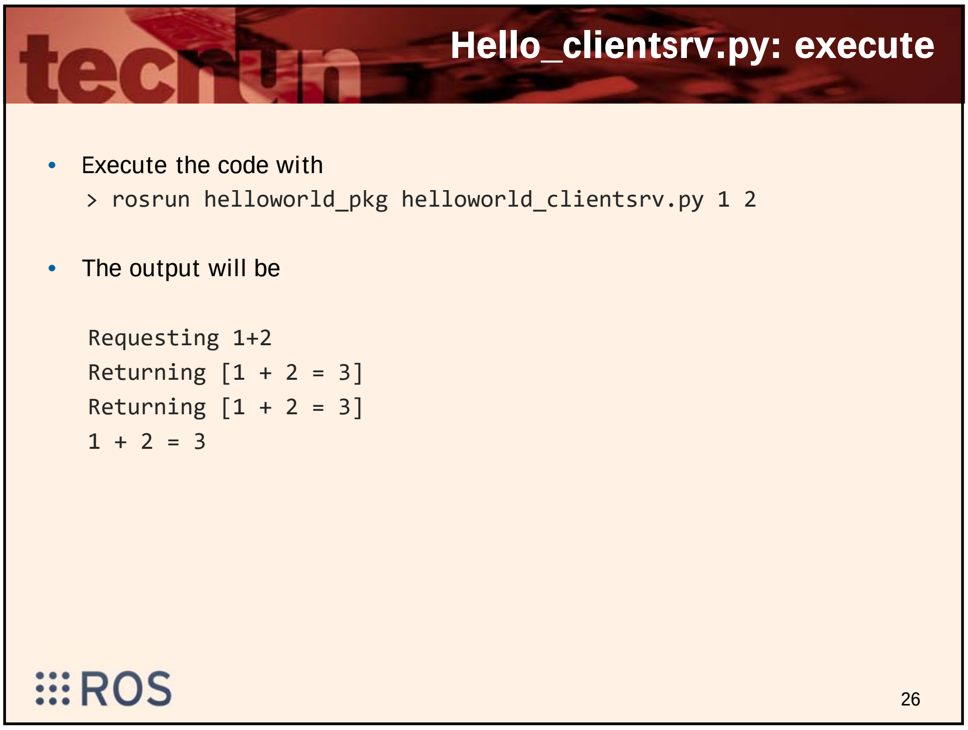


Hello_clientsrv.py (4 of)

```
elif len(argv) == 3:
     try:
         x = int(argv[1])
         y = int(argv[2])
     except:
         print usage()
         sys.exit(1)
 else:
     print usage()
     sys.exit(1)
 print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

ROS

25



Hello_clientsrv.py: execute

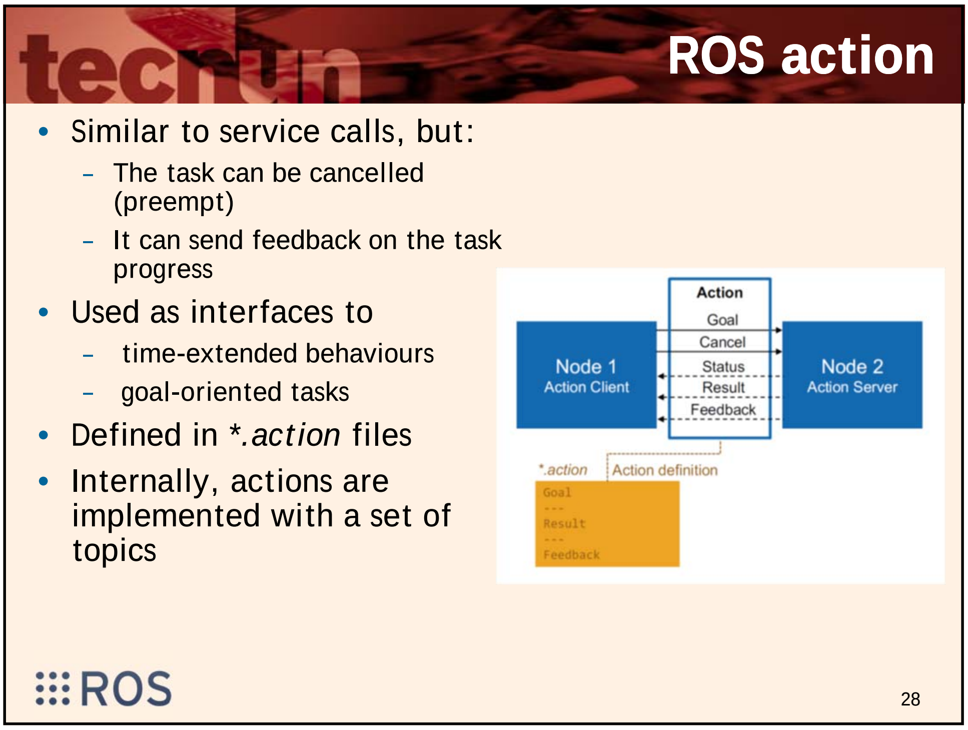
- Execute the code with
  `> rosrun helloworld_pkg helloworld_clientsrv.py 1 2`
- The output will be

```
Requesting 1+2
Returning [1 + 2 = 3]
Returning [1 + 2 = 3]
1 + 2 = 3
```

ROS

26

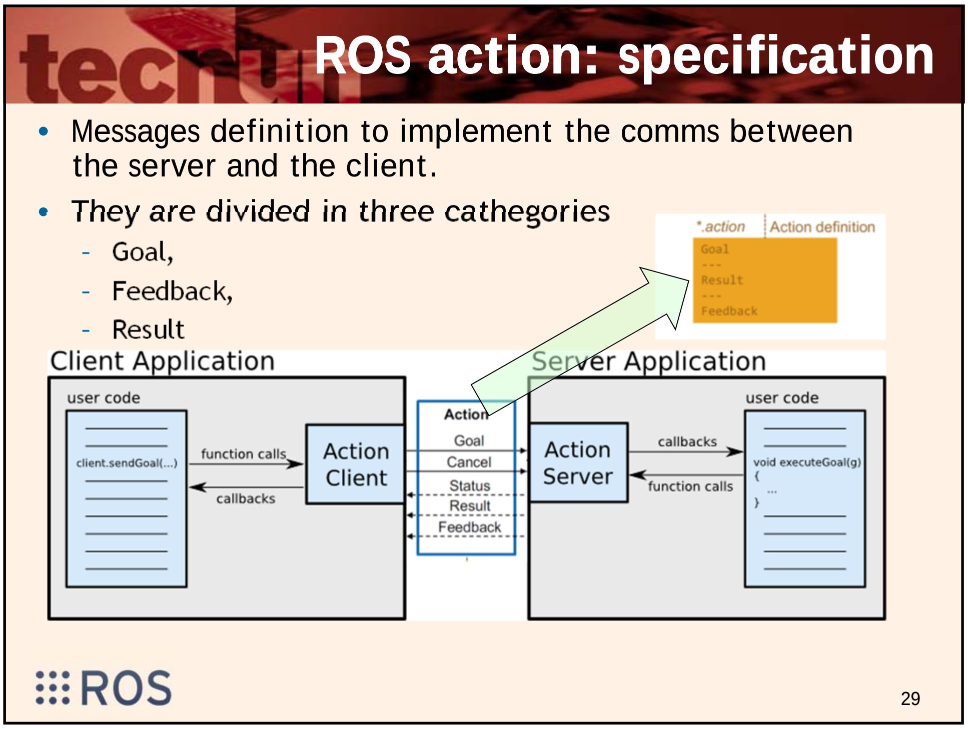
ROS action



ROS action

- Similar to service calls, but:
  - The task can be cancelled (preempt)
  - It can send feedback on the task progress
- Used as interfaces to
  - time-extended behaviours
  - goal-oriented tasks
- Defined in **.action* files
- Internally, actions are implemented with a set of topics

```
Goal
---
Result
---
Feedback
```



ROS action: specification

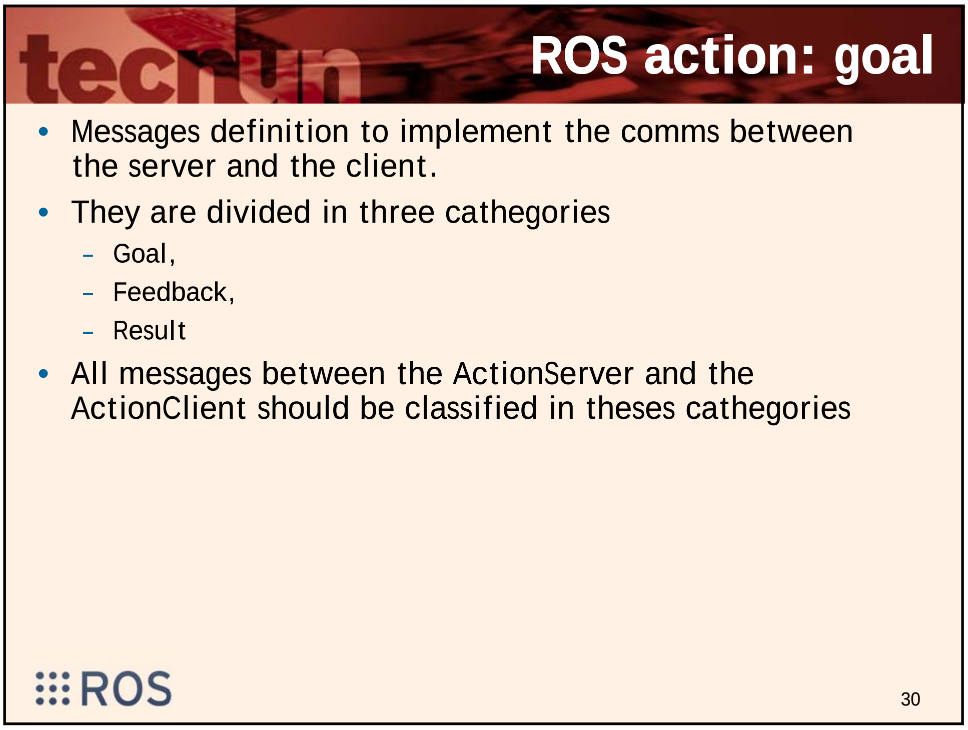
- Messages definition to implement the comms between the server and the client.
- *They are divided in three cathegories*
  - Goal,
  - Feedback,
  - Result

*.action
Action definition
Goal

Result

Feedback
Client Application
user code
client.sendGoal(...)
function calls
callbacks
Action Client
Action
Goal
Cancel
Status
Result
Feedback
Server Application
Action Server
callbacks
function calls
user code
void executeGoal(g)
{
...
}
:::ROS

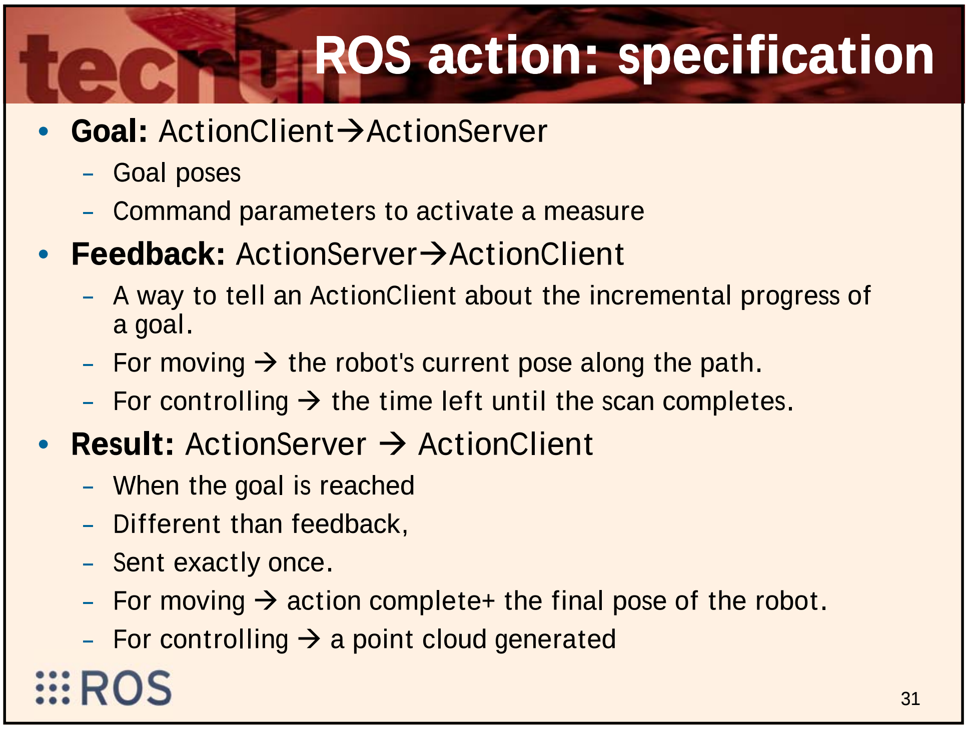
29



ROS action: goal

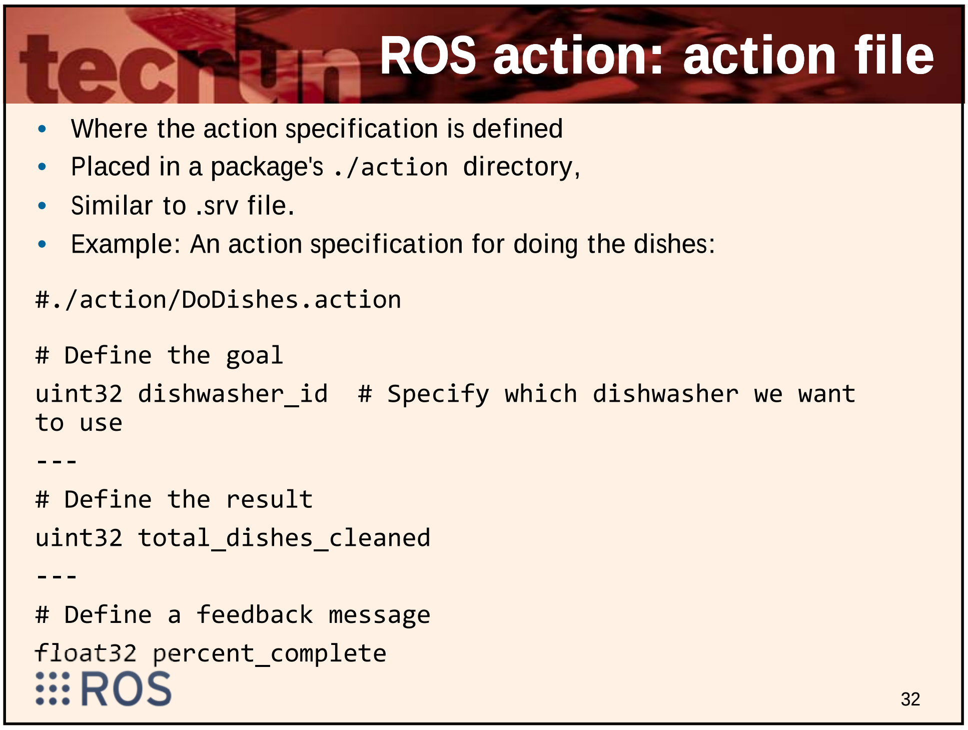
- Messages definition to implement the comms between the server and the client.
- They are divided in three cathegories
  - Goal,
  - Feedback,
  - Result
- All messages between the ActionServer and the ActionClient should be classified in theses cathegories

30



ROS action: specification

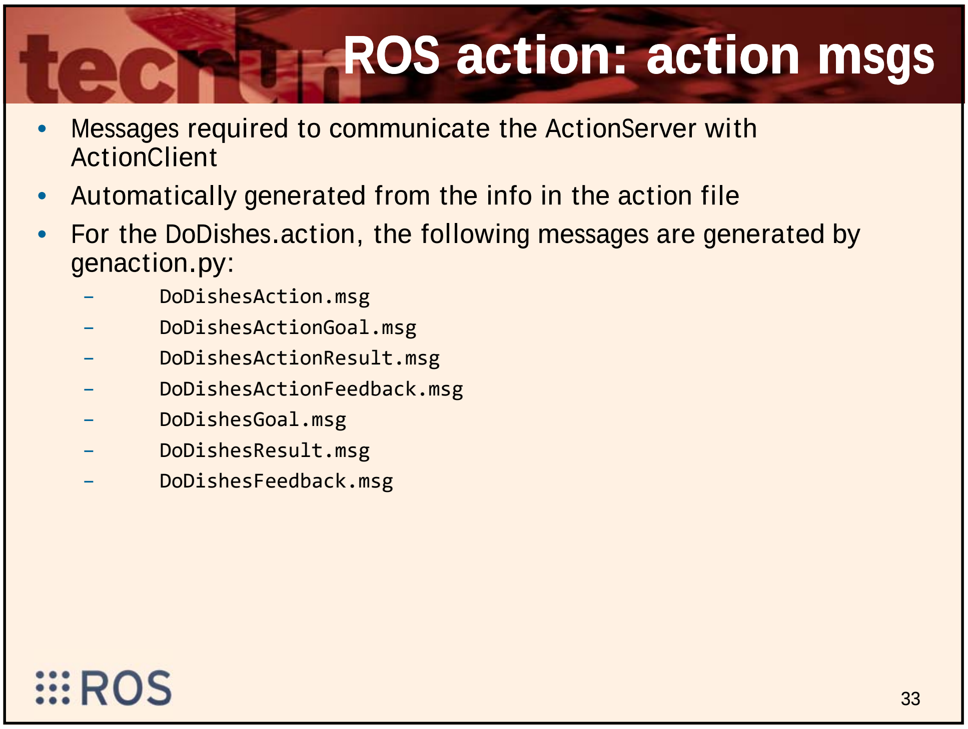
- **Goal:** ActionClient→ActionServer
  - Goal poses
  - Command parameters to activate a measure
- **Feedback:** ActionServer→ActionClient
  - A way to tell an ActionClient about the incremental progress of a goal.
  - For moving → the robot's current pose along the path.
  - For controlling → the time left until the scan completes.
- **Result:** ActionServer → ActionClient
  - When the goal is reached
  - Different than feedback,
  - Sent exactly once.
  - For moving → action complete+ the final pose of the robot.
  - For controlling → a point cloud generated



ROS action: action file

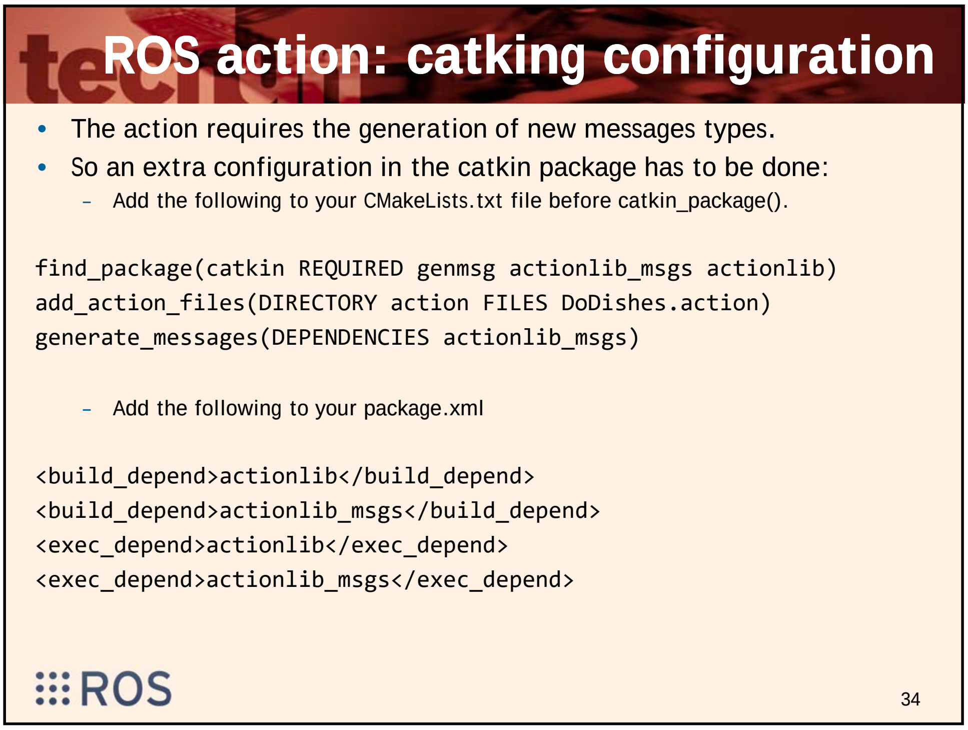
- Where the action specification is defined
- Placed in a package's `./action` directory,
- Similar to .srv file.
- Example: An action specification for doing the dishes:

```
#./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
```



ROS action: action msgs

- Messages required to communicate the ActionServer with ActionClient
- Automatically generated from the info in the action file
- 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`



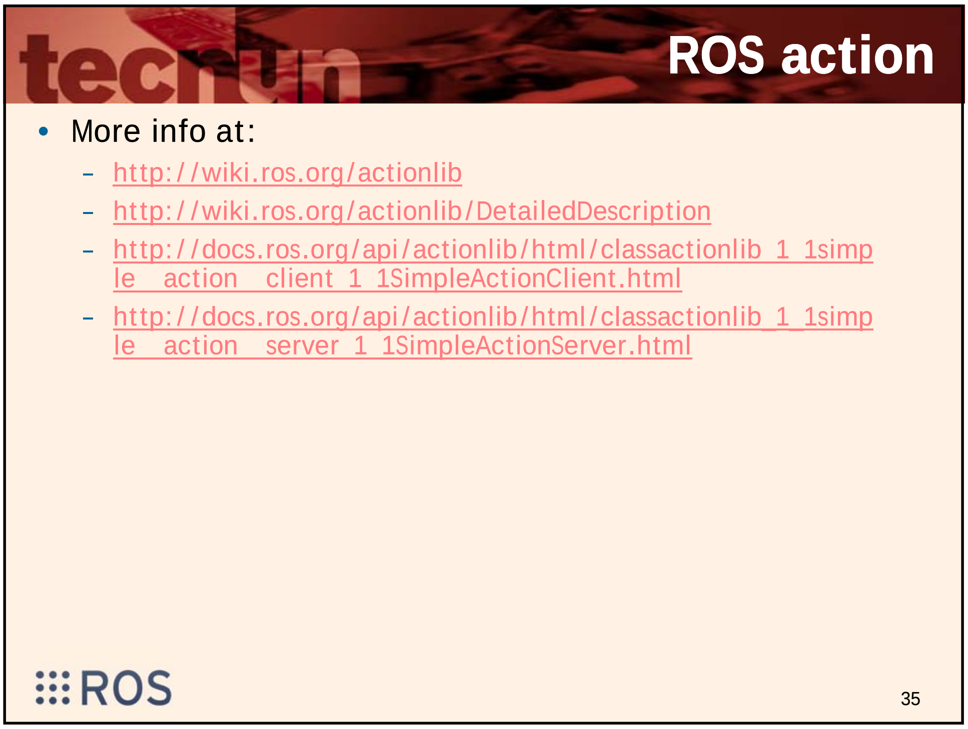
ROS action: catking configuration

- The action requires the generation of new messages types.
- So an extra configuration in the catkin package has to be done:
  - Add the following to your CMakeLists.txt file before catkin_package().

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

  - Add the following to your package.xml

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

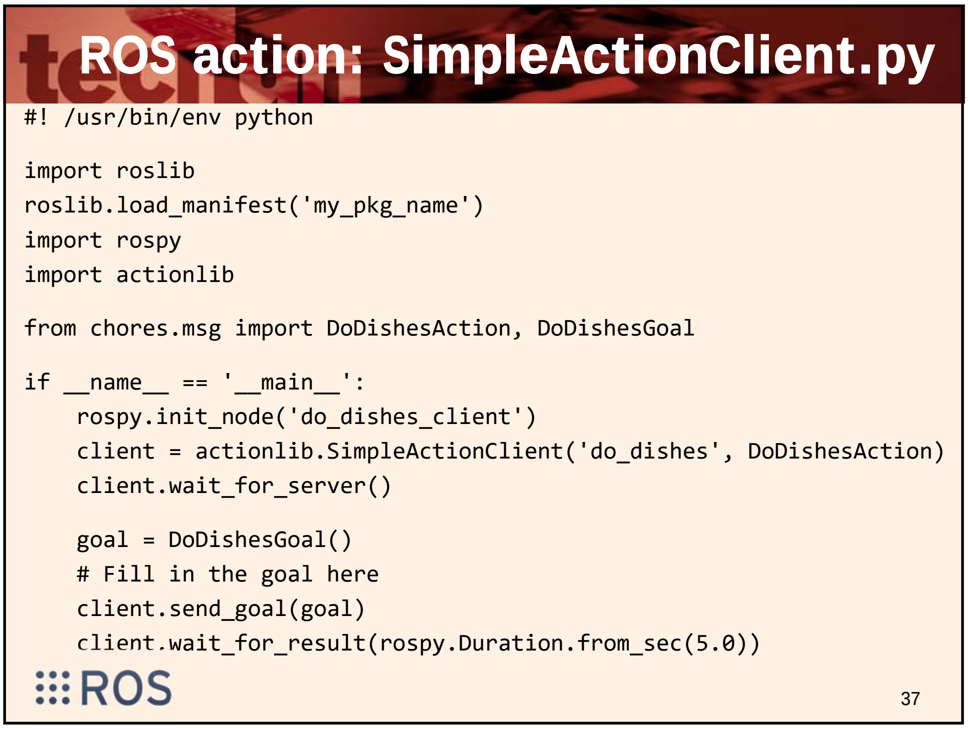


ROS action

- More info at:
  - http://wiki.ros.org/actionlib
  - http://wiki.ros.org/actionlib/DetailedDescription
  - http://docs.ros.org/api/actionlib/html/classactionlib_1_1simple__action__client_1_1SimpleActionClient.html
  - http://docs.ros.org/api/actionlib/html/classactionlib_1_1simple__action__server_1_1SimpleActionServer.html

35

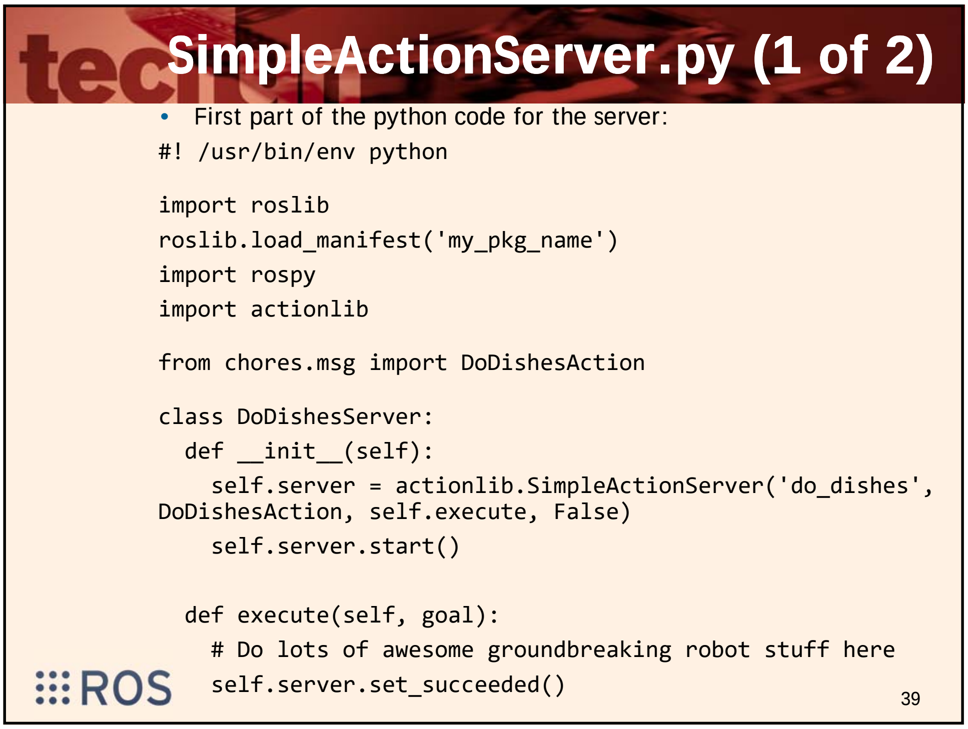
simpleActionClient.py



ROS action: SimpleActionClient.py

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

import roslib
roslib.load_manifest('my_pkg_name')
import rospy
import actionlib

from chores.msg import DoDishesAction, DoDishesGoal

if __name__ == '__main__':
    rospy.init_node('do_dishes_client')
    client = actionlib.SimpleActionClient('do_dishes', DoDishesAction)
    client.wait_for_server()

    goal = DoDishesGoal()
    # Fill in the goal here
    client.send_goal(goal)
    client.wait_for_result(rospy.Duration.from_sec(5.0))
```

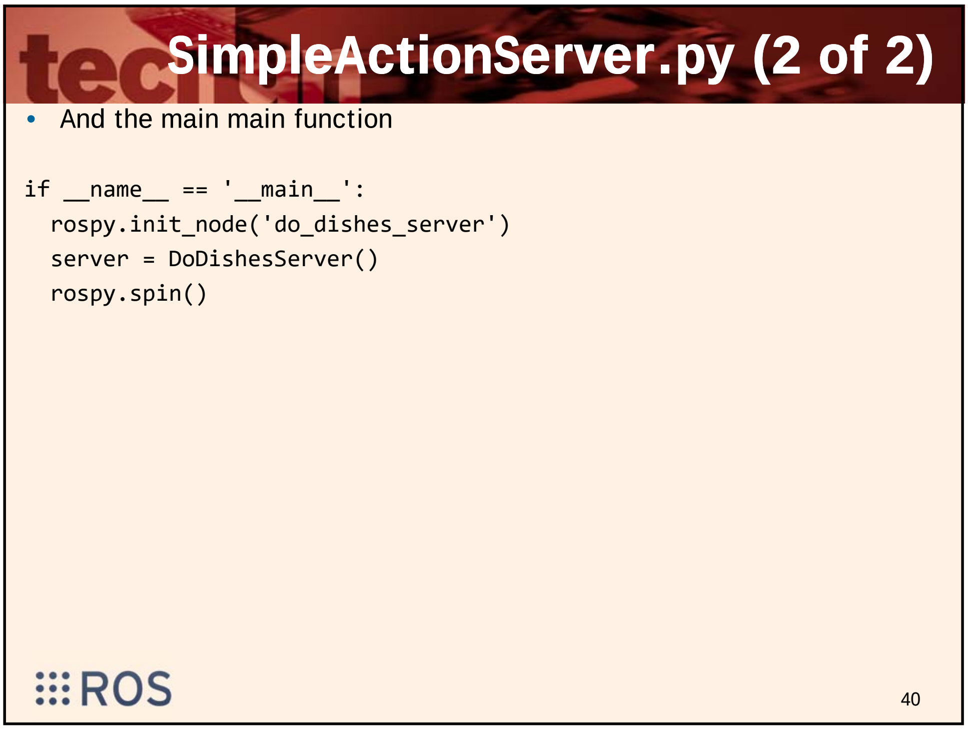
SimpleActionServer.py



SimpleActionServer.py (1 of 2)

- First part of the python code for the server:

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

import roslib
roslib.load_manifest('my_pkg_name')
import rospy
import actionlib

from chores.msg import DoDishesAction

class DoDishesServer:
  def __init__(self):
    self.server = actionlib.SimpleActionServer('do_dishes',
DoDishesAction, self.execute, False)
    self.server.start()

  def execute(self, goal):
    # Do lots of awesome groundbreaking robot stuff here
    self.server.set_succeeded()
```



SimpleActionServer.py (2 of 2)

- And the main main function

```
if __name__ == '__main__':
  rospy.init_node('do_dishes_server')
  server = DoDishesServer()
  rospy.spin()
```