# ROS BASICS 1

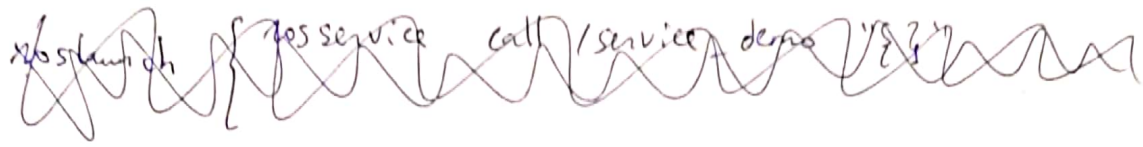
Services ≠ Actions

- Services → when called, the robot can be doing something else meanwhile
- Actions → when called, the service in execution must end beforehand

1° START SERVER

roslaunch { service_demo  service_launch.launch
action_demo  action_launch.launch

2° CALL



{ rosservice  call /service_demo "{}"
roslaunch  action_demo_client  client_launch.launch

Debugger → rosrun rviz rviz

Interface
0° Fixed frame → base_link
1° Add RobotModel
2° Add LaserScan → Topic /kobuki/laser/scan

---

Creation
- Topic
  - Publishers
  - Subscribers
  - } Move
  - } Read → IMU, odometry } data
- Service → Read IMU → Tells which way to go
- Action → Records odometry → Tells which distance has moved
- Global program → make the robot get out of the maze

roslaunch <package_name> <launch_file>

Package
- launch folder
- src folder
- CMakeLists.txt
- package.xml

Creation of packages

.pwd

rosed —> /home/user/catkin_ws/devel

cd .. —> /home/user/catkin_ws

ls —> build devel src

cd src —> /home/user/catkin_ws/src this directory

ls —> list items

vi —> visualize items

catkin_create_pkg {name} my_package {dependencies} rospy

Compilation of packages

/home/user/catkin_ws this directory

catkin_make or specifically catkin_make --only-pkg-with-deps my_package

launch file

PKG = "package_name"
↳ name of package
name = "node_name"
↳ node that will launch our python file

type="python_file_name.py"
↳ name of the file program
output = "type_of_output"
↳ printing channel

:1 at the end for exit

Create simple_file.py inside the src of your new package

rosed my_package

mkdir launch —> create a launch folder

simple_file.py

#! /usr/bin/env python

import rospy

rospy.init_node('Obi Wan') —> initialization of the node

rate = rospy.Rate(2) Hz

while not rospy.is_shutdown():
    print "Help me ..."
    rate.sleep()

—> rosnode list —> it won't appear unless the code has a loop

Launch program

roslaunch my_package my_package_launch.launch

Parameter server → dictionary → static data (configuration, ...)

rosparam list
" get <parameter_name>
set " <value>

export | grep ROS → get parameters of ROS

rostopic -h → obtain what you can do with that tool

Creation of a topic

```
#! /usr/bin/env python
import rospy
from std_msgs.msg import Int32

rospy.init_node('topic_publisher')
pub = rospy.Publisher('/counter', Int32, queue_size=1)
rate = rospy.Rate(2)
                  #2
count = Int32()
count.data = 0
while not rospy.is_shutdown():
    pub.publish(count)
    count.data += 1
    rate.sleep(2)
```

rostopic info /counter

Publisher is a node that keeps publishing a message into a topic

Topic is a channel that acts as a pipe where other nodes can either publish or read information

rostopic echo /counter -n 1

Structure of a message

ros msg show /cmd_vel

Error cannot launch node...

↳ chmod +x name_of_file.py in its directory

ROS → framework provides background to manage processes and communications

Encapsulate topic subscribers and publishers inside classes

## Debugging

to see debugging

While there are nodes running → rostopic echo /rosout

If there is flooding of information → rqt_console

```
rostopic info /joint_states

  Type: sensor_msgs/JointState
ros msg show JointState
    string[] name
    float64[] position
      "      velocity
      "      effort
```

rqt_plot to see graphs

rqt_graph → display of nodes running in ROS and their topic connections

# Record experimental Data and Rosbags



You test a robot in real conditions and you need to record every detail → rosbag → records all of the data passed through ROS topics system

↳ allows you to replay the test in a simulation

1° Go to src catkin_ws directory

2° rosbag record -O name_bag_file.bag name_topic_to_record 1
..
..
N

After a considered while...

3° Kill the terminal

4° rosbag info ~~laser.bag~~ name_bag_file.bag

5° rosbag play name_bag_file.bag
-l to loop indefinitely

6° rostopic echo /name_topic_to_record i

Make sure that the original data generator is NOT publishing

rosservice call /gazebo/pause_physics "{}"

rosbag record -a → record all the topics the robot is publishing

RViz → not a simulation, represents what the topics are publishing

rosrun rviz rviz

A robot has: laser sensor → Topic to publish readings

Face-recognition system → Service → call and wait

Navigation system → Action → call and meanwhile perform other tasks (feedback given)

Services are synchronous → When a service is called, the program can't continue until it receives a result

Actions are asynchronous

ros service list

ros service info /name_of_your_service

Type: <package>/<service_message_file>

| |
|---|
| vi <file> → edit |
| cat <file> → see |

ros service call /the_service_name TAB-TAB

In gazebo simulator

if you want to delete an object of the virtual world:

ros service call /gazebo/delete_model "model_name: 'cafe_table'"

to see the objects in the virtual world

ros topic echo /gazebo/model_states -n1

ros service info /gazebo/delete_model

ros srv show gazebo_msgs/DeleteModel

service message files → .srv → Request → string model_name

topic message files → .msg → Response → boolean success

→ string status_message

(guión = dash)

Services provide functionality to other nodes

If a node knows how to delete an object on the simulation, it can provide that functionality to other nodes through a service call, so they can call the service when they need to delete something

Services allow the specialization of nodes (each node specializes in one thing)

ROS Service Server → creates a service server ready to be called

" " Client → calls automatically that service server

rosrun → runs a ROS program without launch file

teleop_twist_keyboard → input movement commands through the keyboard

Actions are like asynchronous calls to services to another node's functionality

- The node that provides the functionality has to contain an action server
- The node that call to the functionality has to contain an action client

In order to get the actions available on a robot → rostopic list

Every action server creates 5 topics

- cancel
- feedback
- goal
- result
- status

} messages used to communicate with the action server

Message morphology of
- topic → information the topic provides
- service → goal + response
- action server → goal + result + feedback (every time you call an action)

If a message is standard type (can be found in std_msgs package), it's not needed to indicate its procedence

The code to call an action server is simple:

1° You create a client connected to the action server you want

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

2° You create a goal

```
goal = AndroneGoal()
```

Luke Action
2 luke.action

```
goal.nseconds = 10
```

3° You send the goal to the action server

client.send_goal(goal, feedback_cb = feedback_callback)

4° You wait for results

client.wait_for_result()

SimpleActionClient objects → 2 functions → Knowing if the action that is being performed has finished

- wait_for_result() → useless if you want to perform tasks in parallel
- get_state() → int32
  - 0 : Pending
  - 1 : Active
  - 2 : Done
  - 3 : Warn
  - 4 : Error

Cancel an already sent goal → preempting a goal

client.cancel_goal()

roslaunch exercise XX exercise XX.launch &

↳ to launch it in the background

↳ rosnode kill /exercise XX when you finish

It's possible to call an action server manually through topics:

- rostopic pub /[name_of_action_server]/goal /[type_of_the_message_used_by_the_topic] [TAB]x2
- send goal

Axclient → GUI → interact with action server

rosrun actionlib axclient.py /<name_of_action_server>

Some very helpful service structure is std_srvs/Trigger as you just ask for data without providing input

↳ you create a service that uses Trigger.srv

It's recommended to use action messages already provided by ROS which can be found in:

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

If you want to create your own type action message:

1 - Create an action directory within your package

2 - Create your Name.action (message) file

- Will determine the name of classes used in action server (Lo camel-case) " client
- Has to contain 3 parts separated by 3 hyphens regardless one part is empty

3 - Modify CMakeLists.txt and package.xml to include action message compilation

- CMakeLists.txt
  - find_package()
  - add_action_files()
  - generate_messages()
  - catkin_package()
- package.xml
  - Add packages required to compile messages (nav_msgs / Odometry)
    <build_depend> nav_msgs </build_depend>
  - If besides you need that package for the execution of the program
    <build_export_depend> nav_msgs </...>
    <exec_depend> nav_msgs </...>
  - When you compile custom action messages you must add
    <build_depend> actionlib_msgs </...>
  - When you use Python
    <build_export_depend> rospy </...>
    <exec_depend> rospy </...>

4 - Now you compile everything

roscd; cd ..; catkin_make; source devel/setup.bash; rosmsg list | grep Name