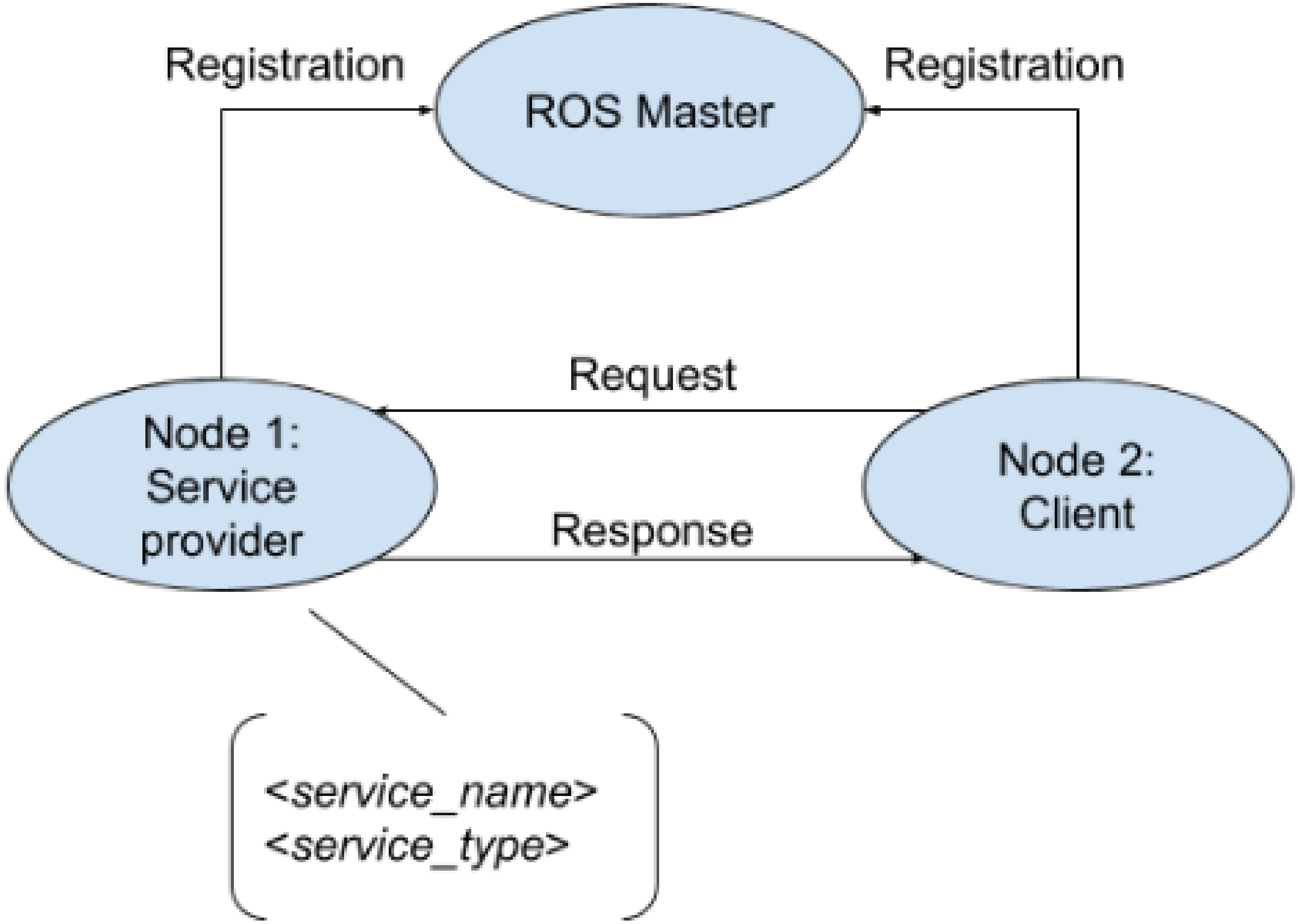
LEONARDO TECHNICAL TRAINING

Robotic Software
Lezione 3

Robot Operating System (ROS)

ROS Communication (Service)



Example 1.3

- Create a ROS package called ros_service with two nodes, a service server and a service client
  - Goal: client node sends a string to the server.
  - The server replies with a string

- Use the command line to inspect the active services
- Call the service using the command line

Example 1.3

- ROS services work on custom messages
- We can not rely on the implemented standard messages
- Some dependencies are needed to create new messages
  - message_generation
  - Message dependencies
    - $ roscd
    - $ cd ../src
    - $ catkin_create_pkg ros_service roscpp std_msgs messabge_generation
- Start creating our service message
  - The service messages MUST be put in the *srv* directory of the package, otherwise it will not be compiled

Example 1.3

- ROS services work on custom messages
- We can not rely on the implemented standard messages
- Some dependencies are needed to create new messages
  - message_generation
  - Message dependencies
    - $ roscd
    - $ cd ../src
    - $ catkin_create_pkg ros_service roscpp std_msgs messabge_generation
- Start creating our service message
  - The service messages MUST be put in the *srv* directory of the package, otherwise it will not be compiled
  - $ rospack profile
  - $ roscd ros_service
  - $ mkdir srv && cd srv
  - $ touch service.srv

Example 1.3

- Start creating our service message
  - The service messages MUST be put in the *srv* directory of the package, otherwise it will not be compiled
  - $ rospack profile
  - $ roscd ros_service
  - $ mkdir srv && cd srv
  - $ touch service.srv
- The content of the service file is:
  - string in
  - ---
  - string out
- Before the --- sign, there are the input arguments
- After the --- sign, there are the output arguments

Example 1.3

- Modify the CMakeLists.txt
  - add_service_files(
  -   FILES
  -   service.srv
  -)

  - generate_messages(
  -     DEPENDENCIES
  -     std_msgs
  -)

- Compile the package
  - $ roscd
  - $ cd ..
  - $ catkin_make
- Check the message
  - $ rossrv show ros_service/service

Example 1.3

- Now you can create the two source (two nodes)

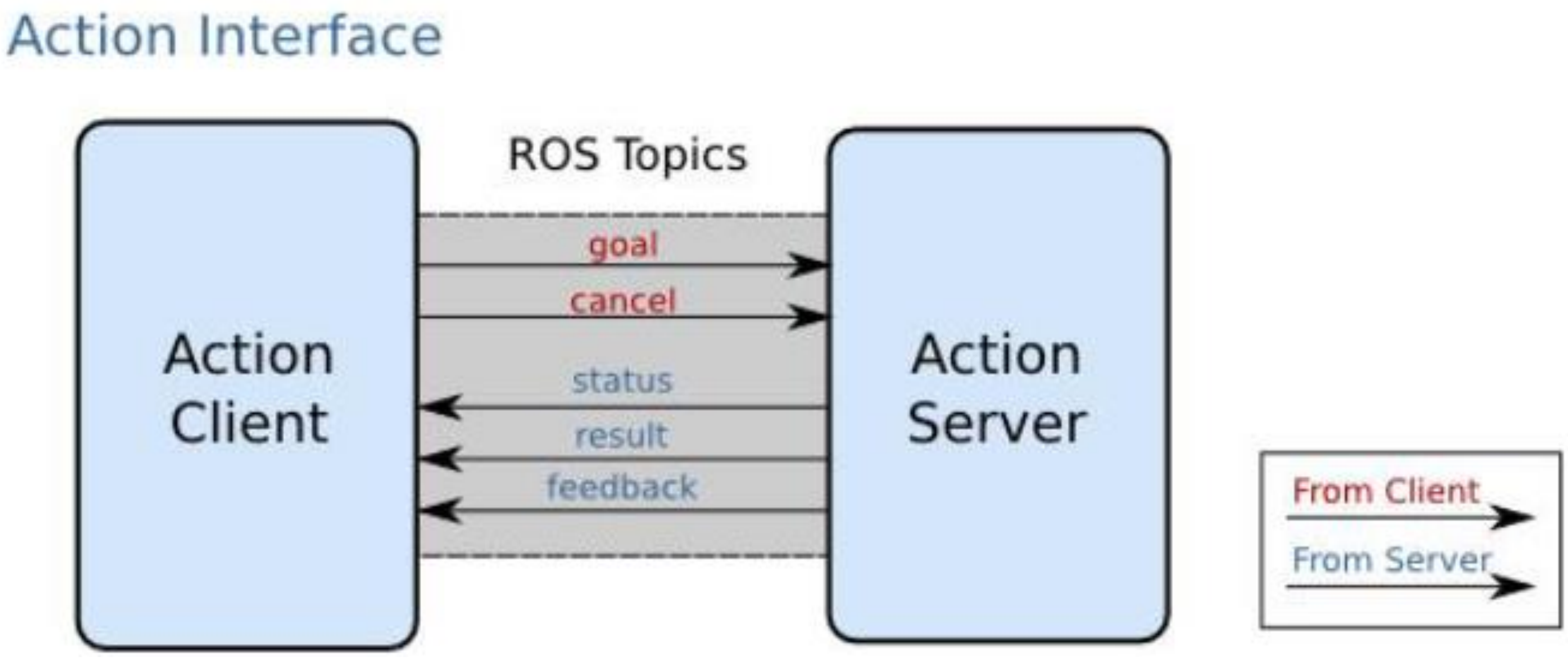
Example 2.3

- Replicate Example 1.3 in python
- Create a ROS package called ros_service with two nodes, a service server and a service client
  - Goal: client node sends a string to the server.
  - The server replies with a string

- Use the command line to inspect the active services
- Call the service using the command line

actionlib

- Request/reply interaction between two nodes Client/Server architecture



actionlib

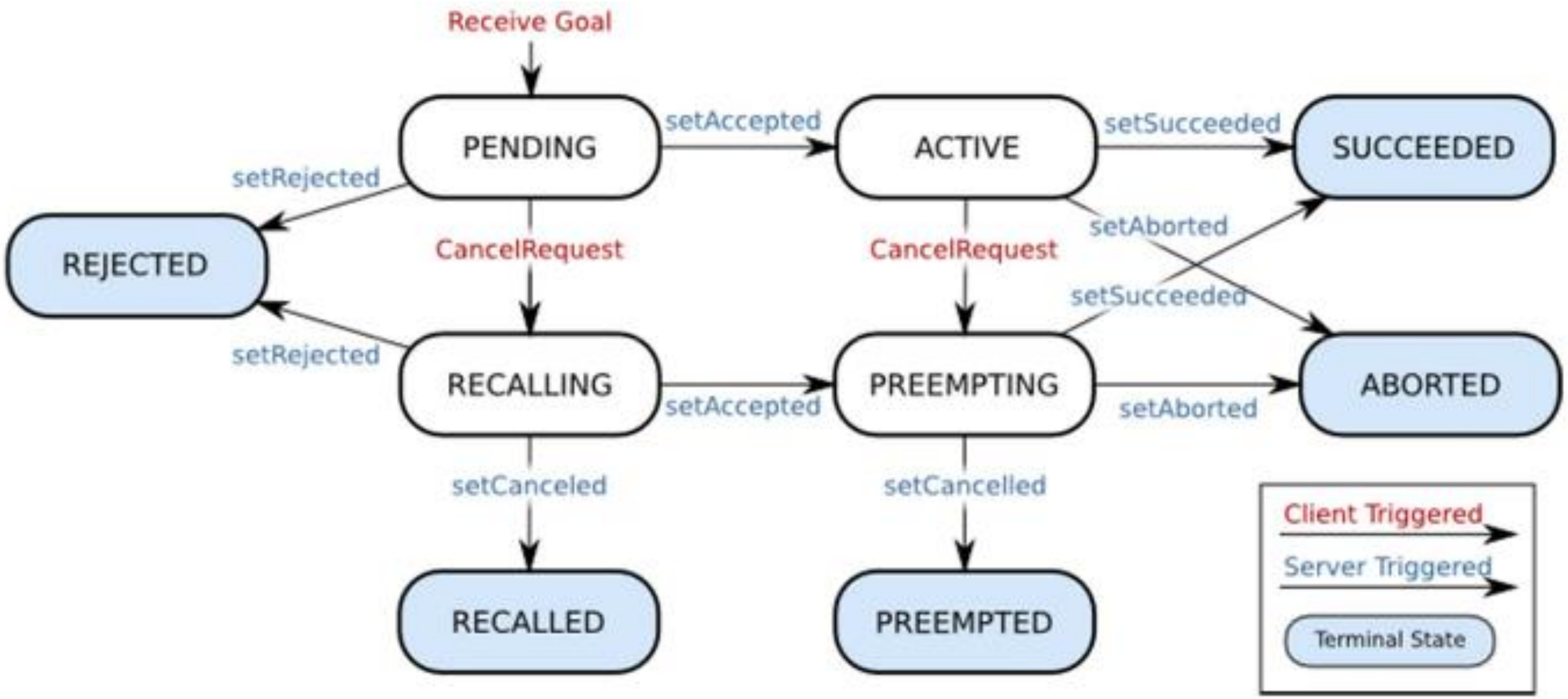
- Like a service?
  - If the server takes too much time we have to wait until it completes!
  - This could block the main application while waiting for the termination of the requested action
  - The calling client could be implemented to monitor the execution of the remote process
  - ROS actionlib implements a protocol in which we can preempt the running request and start sending another one if the request is not finished on time as we expected.
  - We have to specify the data type to request the action.

actionlib

- Action specification are stored inside .action file.
  - This file must be kept inside the action folder of the package
  - Goal: The goal to send and must be executed by the action server.
  - Feedback : Feedback is simply giving the progress of the current operation inside the callback function.
  - Result: The final result if sent to the client when the action is finished: it can be the computational result or an acknowledgment.

actionlib

- Action server implements state machine to manage the execution of the process



actionlib

- The action server can be in the following states:
  - Pending: The goal has yet to be processed by the action server.
  - Active: The goal is currently being processed by the action server.
  - Preempting: The goal is being processed, and a cancel request has been received from the action client, but the action server has not confirmed the goal is canceled
  - Rejected: The goal was rejected by the action server without being processed and without a request from the action client to cancel.
  - Succeeded: The goal was achieved successfully by the action server.
  - Aborted: The goal was terminated by the action server without an external request from the action client to cancel.
  - Preempted: Processing of the goal was canceled by either another goal, or a cancel request sent to the action server.

Example 3.3

- Create a ROS package called ros_action with two nodes:
  - An action client and an action subscriber
  - The action client sends a number as the goal
  - The action server receives the goal and counts from 0 to the goal number with a step size of 1 and with a 1 second delay
  - If it completes before the given time, it will send the result; otherwise, the task will be preempted by the client The feedback here is the progress of counting

Example 3.3

- Like ROS services, a custom message must be created to use the action
- Additional dependencies must be added to our package
  - actionlib
  - actionlib_msgs
    - $ roscd
    - $ cd ../src
    - $ catkin_create_pkg ros_action roscpp actionlib actionlib_msgs std_msgs
- The action files must be put in the action subfolder of the package
  - $ rospack profile
  - $ roscd ros_action
  - $ mkdir action
  - $ touch demo.action

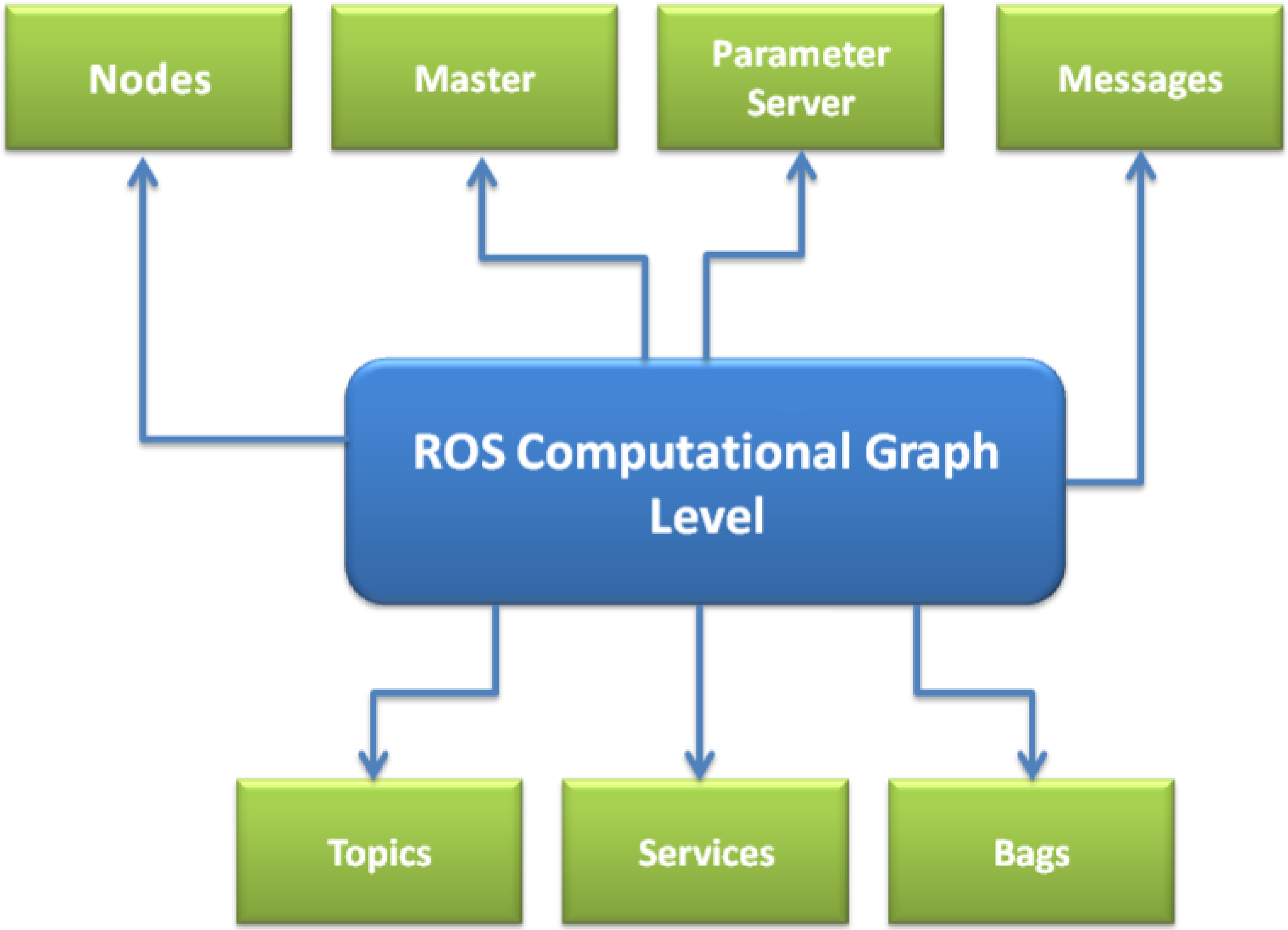
Example 3.3

- The content of the demo.action file
  - #goal definition
  - int32 count
  - ---
  - #result definition
  - int32 final_count
  - ---
  - #feedback
  - int32 current_number

Example 3.3

- Modify the CMakeLists.txt
  - ## Generate actions in the 'action' folder
  - add_action_files(
  -    FILES
  -    demo.action
  -)
  - ## Generate added messages and services with any dependencies listed here
  - generate_messages(
  -    DEPENDENCIES
  -    actionlib_msgs
  -      std_msgs
  -)
  - add_dependencies(action_client ros_action_generate_messages_cpp)

ROS Computation graph



Bagfile

- A bag is a file format used in ROS for storing ROS message data
- It represents the main logging system for ROS data and can be used to save and later work on a stream of topic data.
  - Es: if you are working with a camera sensor, you can just record the sensor output placed on its scene and work with the captured data without the hardware.
- To create a new bagfile you can use the following command:
  - $ rosbag record [TOPICS] [OPTIONS]
- You can choose the -a option to record all topics active in your system
- You can choose the -O option to specify the bagfile name.
- To reproduce a bagile:
  - $ rosbag play [BAGFILE]
- You can choose the -l option to play the bagfile in loop

Parameter server

- A shared server in which all ROS nodes can access parameters
- A node can read, write, modify, and delete parameter values from the parameter server
- Parameters can be stored in file and loaded them into the server
- The rosparam tool is used to get and set the ROS
- parameter from the command line.
- To set a value in the given parameter:
  - $ rosparam set [parameter_name] [value]
- To retrieve a value from the given parameter:
  - $ rosparam get [parameter_name]

Parameter server

- To get the value of a parameter from source code in C++:

    - `int my_num;`
    - `if (!nh.getParam("my_num", my_num)) {`
        - `my_num = 1;`
    - `}`

- This function accepts as arguments the name of the parameter, the variable to fill with its value and a default value
- The default value is used when the requested parameter is not present in the parameter server

Example 4.3

- Use parameter server to store a value and use it into a ROS node
- To set a param we can use different modes
  - $ rosparam set [PARAM_NAME] [VALUE]
  - Configuration file

roslaunch

- We only used rosrun command to start a node
- roslaunch command is used to start nodes using configuration (launch) files
- Launch files are a very useful feature for launching more than one node
- We can write all nodes inside an XML-based file called launch files parsing it with roslaunch
- This command will also automatically start the ROS Master and the parameter server (is this good?)
- Launch files can be also used to set ROS parameters

Custom messages compilation

- If your package uses custom messages compiled in the same context of your package could happen that the compiled tries to generate the executable before to generate the messages
- You will receive an error like:
  - impossible to find the .h of your messages
- Typically, a good way to work is to the define the custom messages of your software in a separate package, like:
  - `std_msgs`
  - `geometry_msgs`
  - `nav_msgs`
  - ...
- Specify the custom messages in the dependencies of your ROS package
  - add_dependencies(EXEC_NAME PKG_NAME_generate_messages_cpp)

Example 4.4

- Develop a ROS package with one node to publish a custom message

Exercise 1.4

- Develop a ROS package with three nodes.
  - One node streams random numbers from 0 to 20.
  - Another node receives this random number and if it is the correct one calls a service called *login* implemented on the third node.
  - This service accepts two input elements: a string in which is store the username, and a number to store the magic number.
  - The service server just prints as output the received number and the user, then terminates its execution.

- Time 20 minutes

Exercise 2.4

- Replicate exercise 1.4 in python, try also to make communicate C++ and python nodes in this application

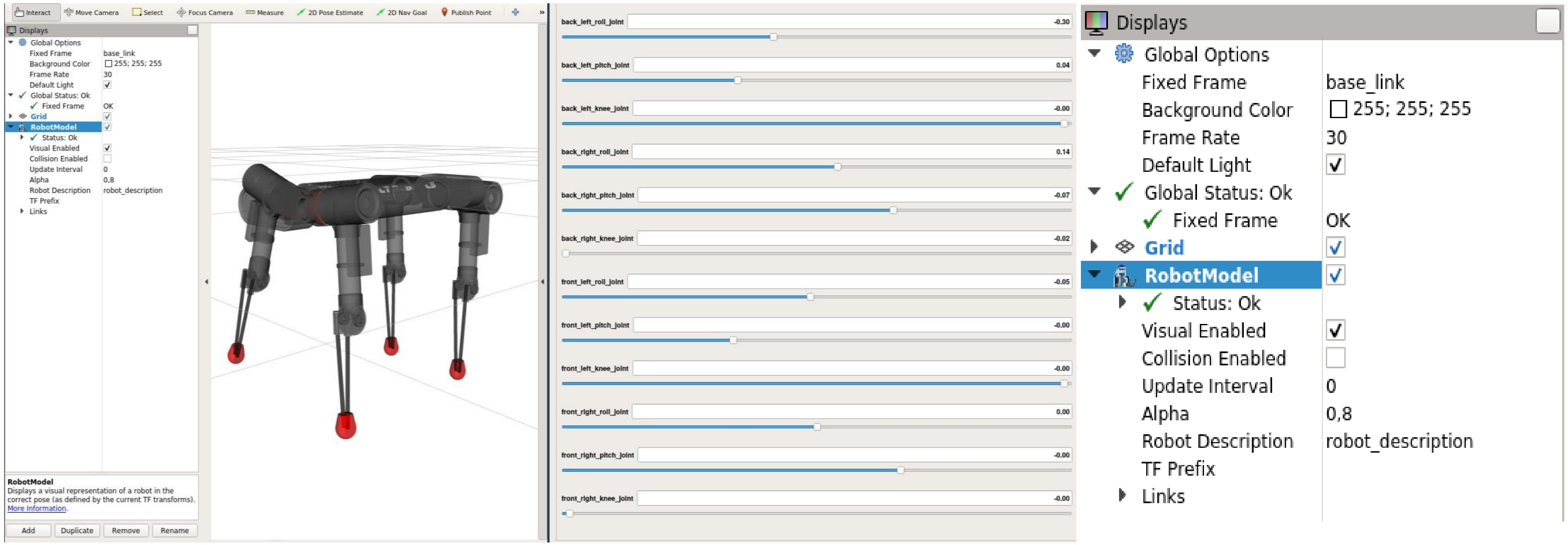
- Time 20 minutes

Modeling and visualization

- Learn how to model a robotic structure from scratch:
  - Create a configuration file specifying the kinematics and the dynamics of your robot
  - Define and connect joints and links Define the shape of robotic links
  - Learn how to use a robot model for visualization and control
- ROS is so much useful in robot programming when you need functionalities already implemented in other ROS packages.
  - Several robotic software need for the knowledge of robotic structure of the robot to work:
  - forward / inverse kinematics motion and path planning obstacle avoidance navigation
- Model a robot from scratch is not easy
  - Robots can be composed by several joints and links
  - Links can be complex to design
  - How to characterize the dynamics?
  - Generate the robot model from the CAD
- **URDF** (Unified Robot Description Format) is the most popular file to model a robot

Modeling and visualization

- RViz is used to visualize robot models. Start RViz node using this command:
  - $ rosrun rviz rviz
- RViz has multiple configuration panels. The most useful one is the Display panel.
- RViz has several plugins. Among them RobotModel is used to visualize the robot in the visualization environment.



Modeling and visualization

- XML documents must contain one root element that is the parent of all other elements:
  - <root>
  - <child>
    - < subchild > . . . . . < / subchild >
  - </child>
  - </root >A tag could also contain attributes:
  - <child attr1="value" attr2="value"> . . . </child>

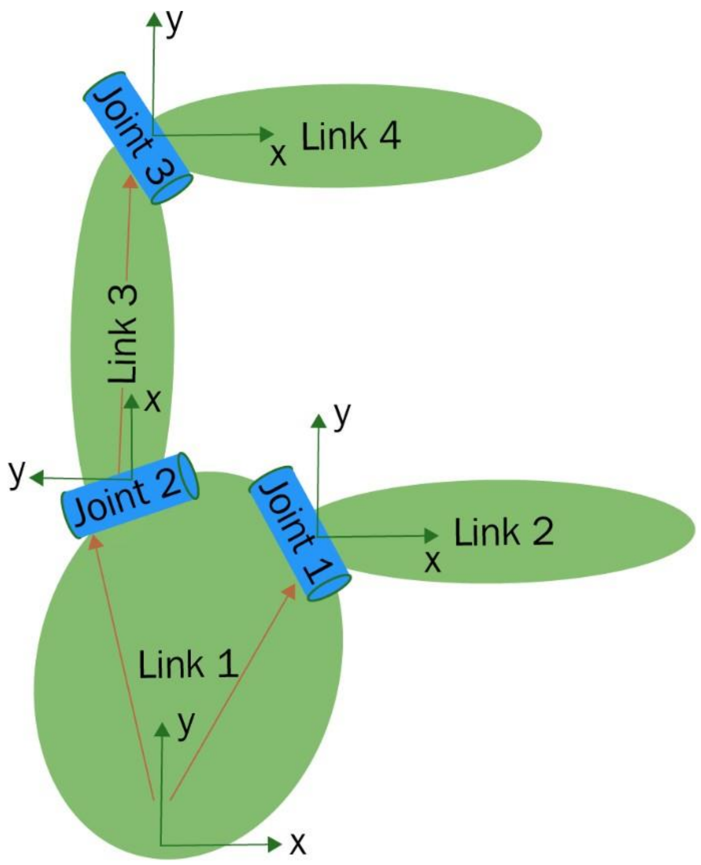
- A tag can also contain ONLY attributes
  - <child attr1="value" attr2="value"> . . . </child>

Modeling and visualization

```xml
<?xml version="1.0"?>
<robot name="pan_tilt">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.01" radius="0.2"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <material name="yellow">
        <color rgba="1 1 0 1"/>
      </material>
    </visual>

    <collision>
      <geometry>
        <cylinder length="0.03" radius="0.2"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </collision>
    <inertial>
  <mass value="1"/>
  <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
    </inertial>
  </link>

  <joint name="pan_joint" type="revolute">
    <parent link="base_link"/>
    <child link="pan_link"/>
    <origin xyz="0 0 0.1"/>
    <axis xyz="0 0 1" />
    <limit effort="300" velocity="0.1" lower="-3.14" upper="3.14"/>
    <dynamics damping="50" friction="1"/>
```



Modeling and visualization

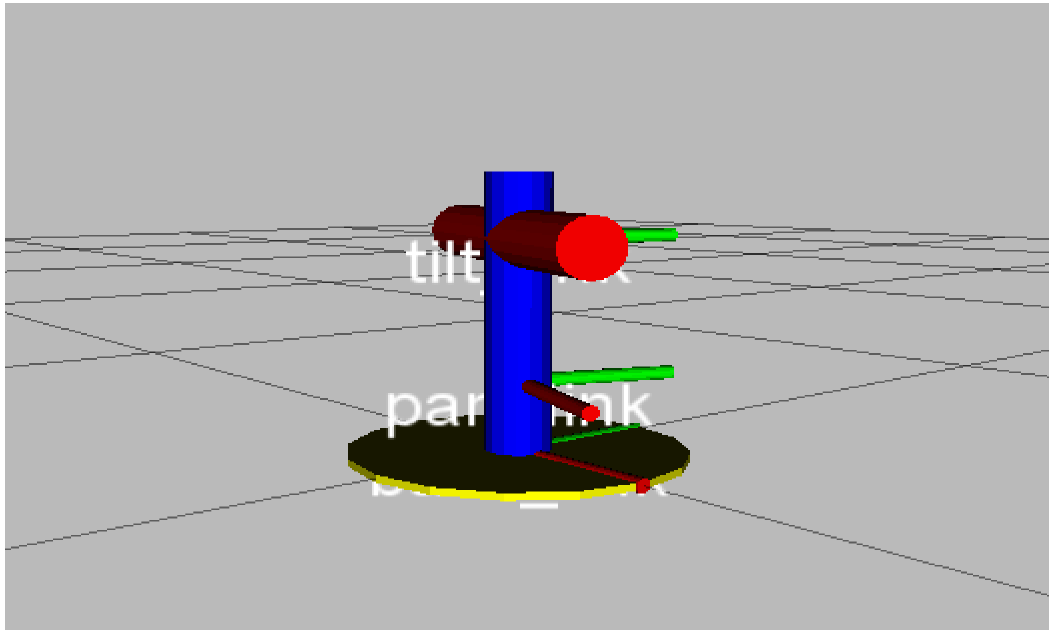
- robot : This tag encapsulates the entire robot model. Inside the robot tag, we can define the name of the robot, the links, and the joints of the robot
  - <robot name="<name of the robot>"
    - <link> . . . . . </link>
    - <joint> . . . . </joint>
  - </robot>
- link : The link tag represents a single link of a robot defining its size, shape, and color. We can also specify the dynamic properties of the link
  - <link name="<name of the link>">
    - <inertial> . . . . </inertial>
    - <visual> . . . . </visual>
    - <collision> . . . . </collision>
  - </link>
- The Visual section contains info about link shape
- The collision section describes the area surrounding the real link

Modeling and visualization

- joint: here we can specify the kinematics and the dynamics and the limits of the joint.
- Different type of joints are supported: revolute, continuous, prismatic, fixed, floating, and planar.
- <joint name=“name of the joint” />
  - <parent link=“name of the parent link” />
  - <child link=“name of the child link” />
  - <limit position, velocity, effort />
- </joint>
- A joint is formed between two links; the first is called the Parent link, and the second is called the Child link.

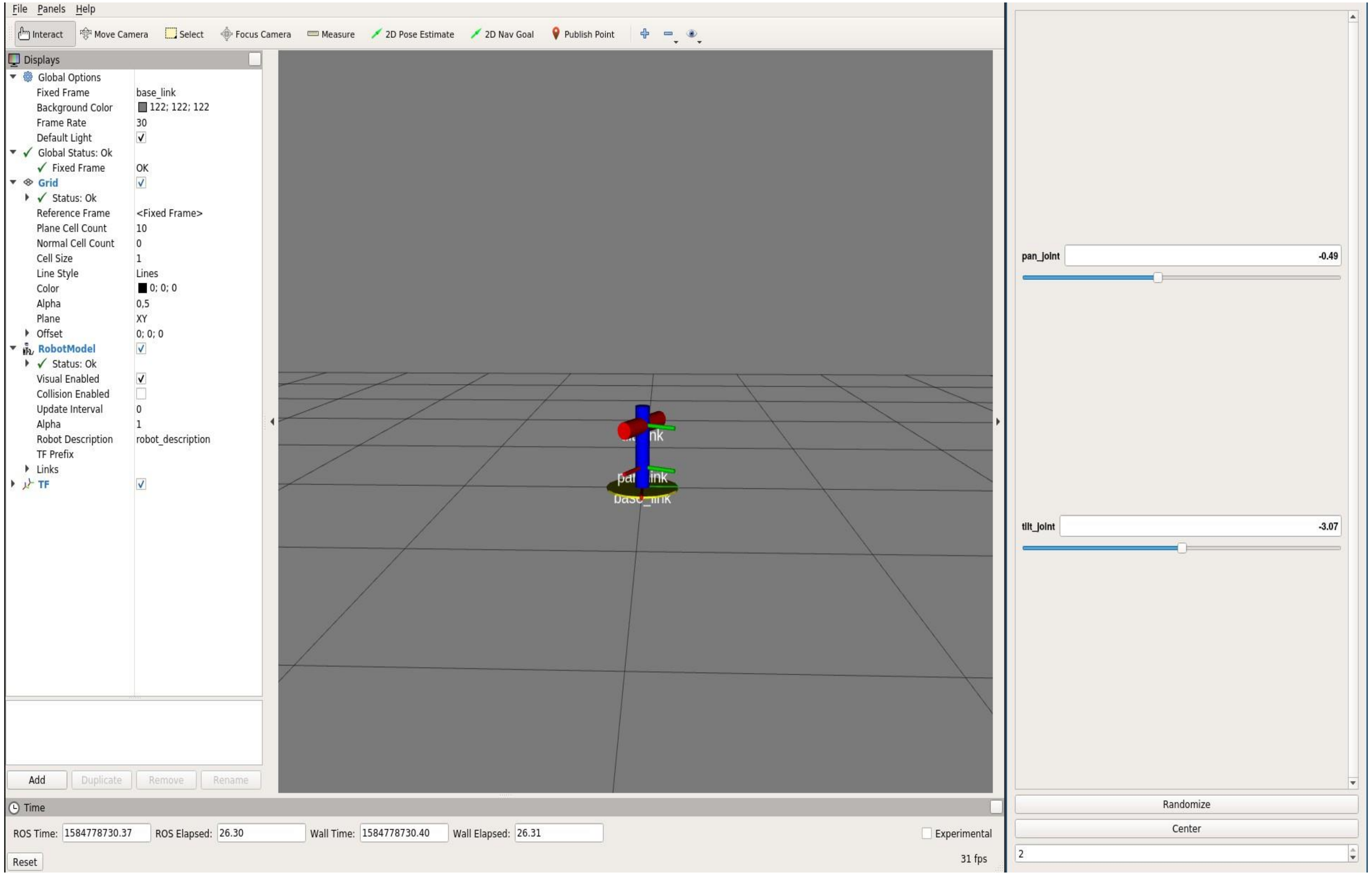
Example 1.3

- Robot models are included in dedicated packages in order to invoke the configuration and modelling files when needed
- Develop a ROS package containing the model of a 2 DOF robot
  - Goal: model a robot with two degree of freedom
  - A degree of freedom is a controllable joint of our robot
    - Pan: rotation left − > right
    - Tilt: rotation up − > down



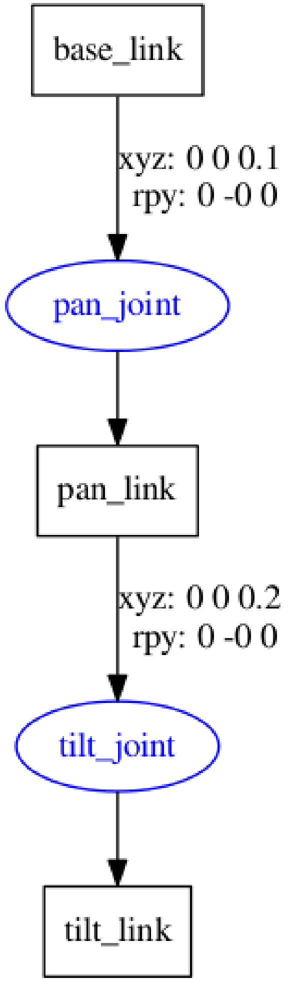
Example 1.3

- Visualize the URDF model of the robot in RVIz



check_urdf

- check_urdf accepts in input the file to check
  - $ roscd robot_description_pkg/urdf/
  - $ check_urdf pan_tilt.urdf
- We can visualize the structure of the robot graphically:
  - $ urdf_to_graphiz pan_tilt.urdf
- The robot model can be visualized using RViz RViz shows the shape of the robot
- Using RViz we can test the connections between the links RViz uses RobotModel plugin to display the robot
- The robot model is taken from ROS parameter server using the robot description parameter



Fine lezione 3