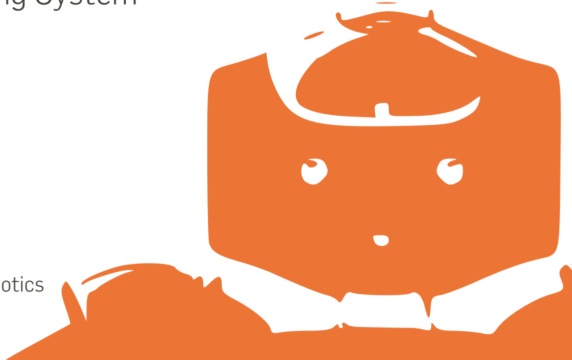
ROBOTICS WITH PLYMOUTH UNIVERSITY

ROCO318
Mobile and Humanoid Robots
ROS, the Robot Operating System

Séverin Lemaignan

Centre for Neural Systems and Robotics
Plymouth University



COURSEWORK ASSESSMENT

- Coursework: 30% of final mark; sensor project: 60% of coursework mark
- Complete report submitted **Thursday 17:00, 11th January 2018**
- One submission per group, single PDF file, max 10 pages (without images/code)

COURSEWORK ASSESSMENT

- Coursework: 30% of final mark; sensor project: 60% of coursework mark
- Complete report submitted **Thursday 17:00, 11th January 2018**
- One submission per group, single PDF file, max 10 pages (without images/code)

Check the coursework description on the DLE for the details of the marking scheme.

ROS IS NOT AN OPERATING SYSTEM

INSTEAD, ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind

INSTEAD, ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)

INSTEAD, ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules

INSTEAD, ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules
- **A middleware?**

INSTEAD, ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules

- A set of conventions to write and package robotic softwares

INSTEAD, ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules

- A set of conventions to write and package robotic softwares
- Deep integration of a few key open-source libraries (OpenCV, PCL, tf)

INSTEAD, ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules

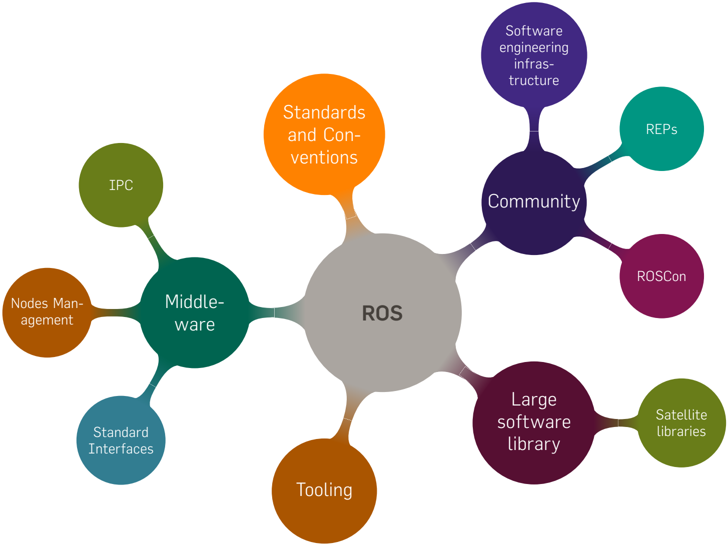
- A set of conventions to write and package robotic softwares
- Deep integration of a few key open-source libraries (OpenCV, PCL, tf)
- A set of tools to run and monitor the nodes

INSTEAD, ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules

- A set of conventions to write and package robotic softwares
- Deep integration of a few key open-source libraries (OpenCV, PCL, tf)
- A set of tools to run and monitor the nodes
- Engagement of a large academic community, leading to a library of thousands of nodes

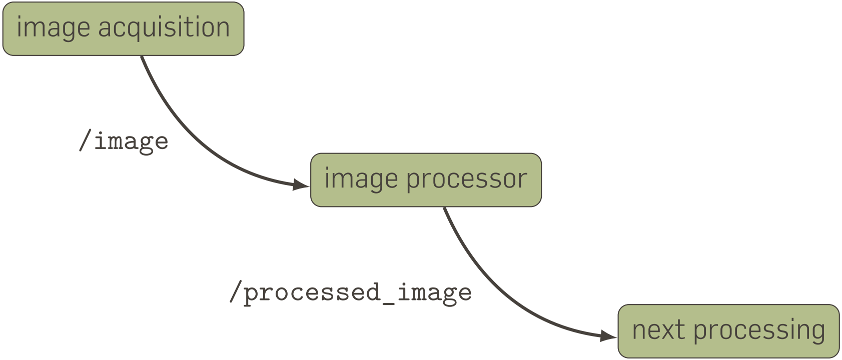
ROS ECOSYSTEM



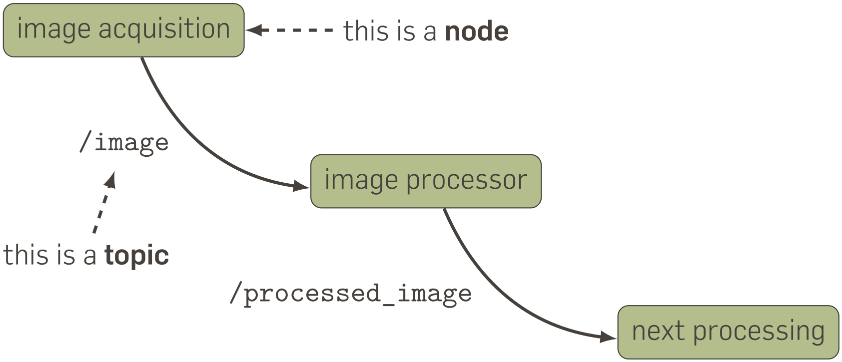
[we will revisit these slides at the end of the lecture]

A FIRST EXAMPLE

A SIMPLE IMAGE PROCESSING PIPELINE



A SIMPLE IMAGE PROCESSING PIPELINE



```python
import sys, cv2, rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge

def on_image(image):
    cv_image = bridge.imgmsg_to_cv2(image, "bgr8")
    rows, cols, channels = cv_image.shape
    cv2.circle(cv_image, (cols/2, rows/2), 50, (0,0,255), -1)
    image_pub.publish(bridge.cv2_to_imgmsg(cv_image, "bgr8"))

rospy.init_node('image_processor')
bridge = CvBridge()
image_sub = rospy.Subscriber("image",Image, on_image)
image_pub = rospy.Publisher("processed_image",Image)

while not rospy.is_shutdown():
    rospy.spin()
```

HOW TO USE THIS CODE?

First, we need to write data onto the `/image` topic, for instance from a webcam:

```
> rosrun usb_cam usb_cam_node
```

HOW TO USE THIS CODE?

First, we need to write data onto the `/image` topic, for instance from a webcam:

```
> rosrun usb_cam usb_cam_node
```

Then, we run our code:

```
> python image_processor.py
```

HOW TO USE THIS CODE?

First, we need to write data onto the `/image` topic, for instance from a webcam:

```
> rosrun usb_cam usb_cam_node
```

Then, we run our code:

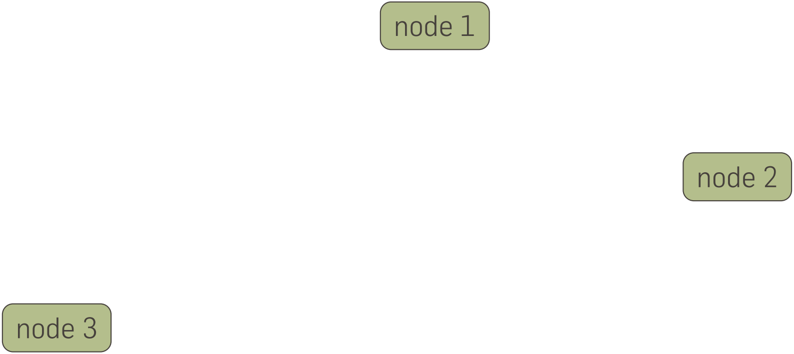
```
> python image_processor.py
```

Finally, we run a 3rd node to display the image:

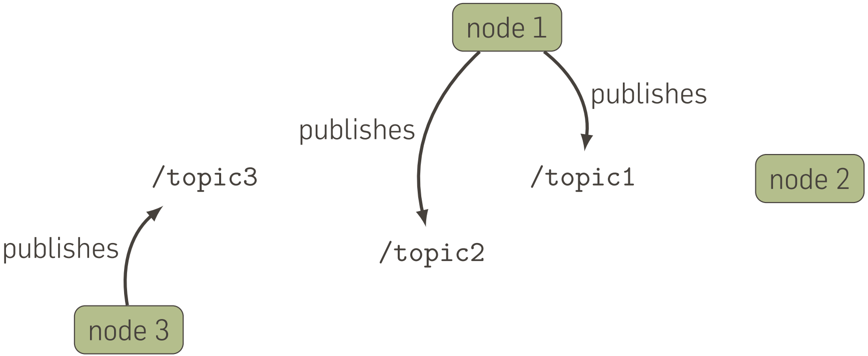
```
> rosrun image_view image_view image:=/processed_image
```

THE KEY ROS CONCEPTS

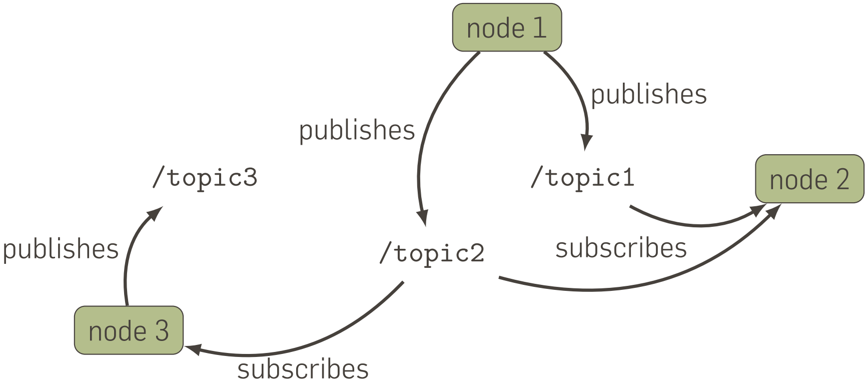
TALKING NODES



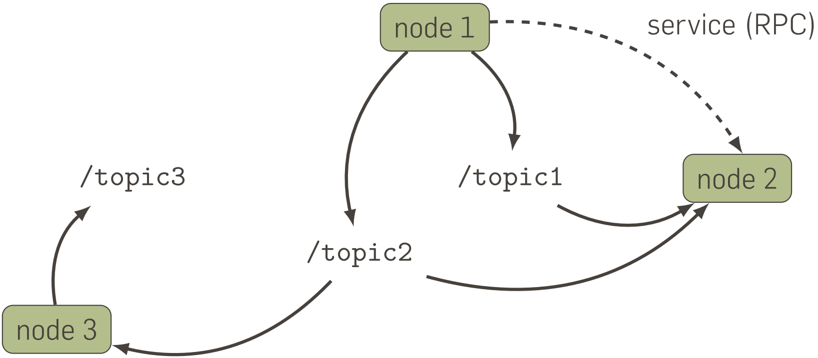
TALKING NODES



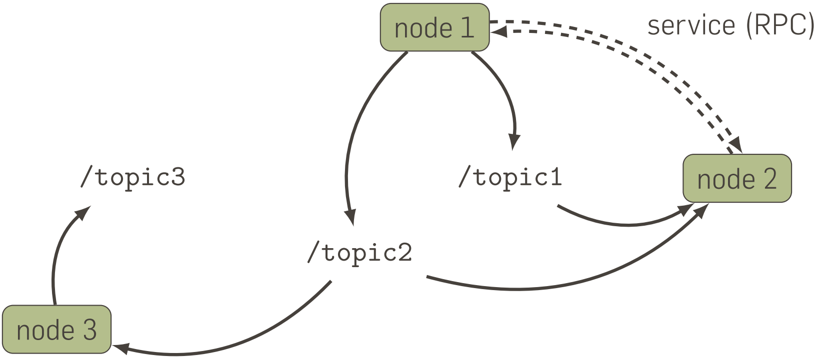
TALKING NODES



TALKING NODES

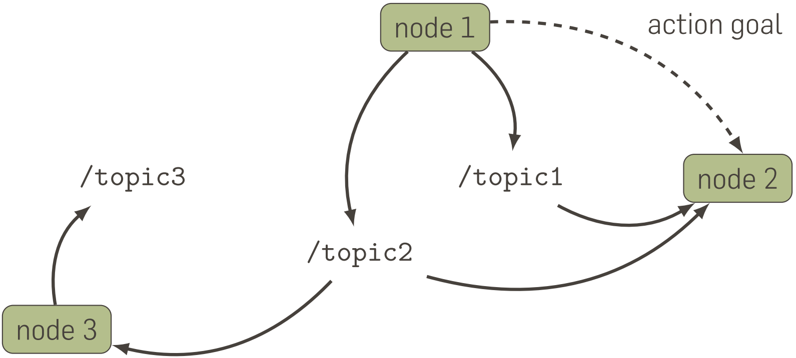


TALKING NODES

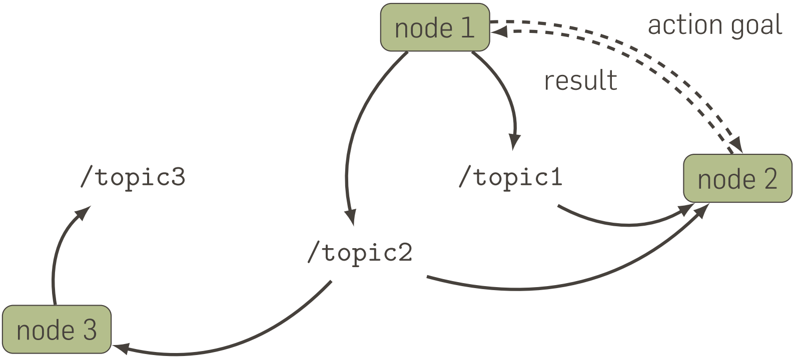


Services: **synchronous**: call is blocking, only suitable when very short processing (e.g. setting a parameter)

TALKING NODES

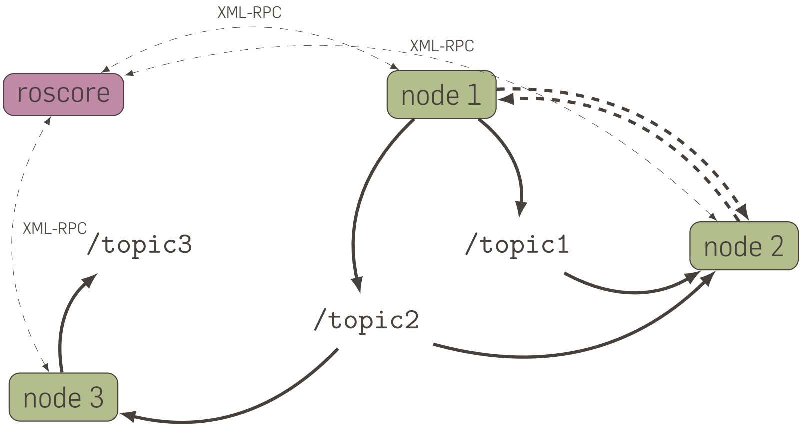


TALKING NODES



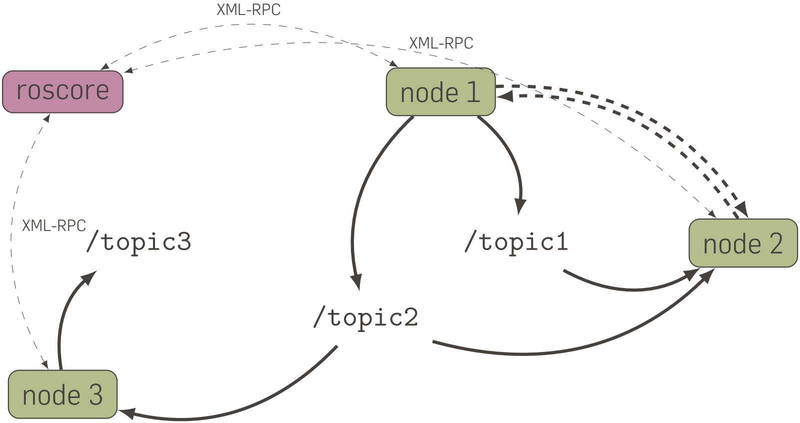
Actions: **asynchronous**: call is non-blocking, suitable for long processes (e.g. motion planning)

TALKING NODES



The `roscore` daemon act as yellow pages for the nodes to discover each others.

TALKING NODES



When nodes are distributed on different machines:
`ROS_MASTER_URI=http://<host>:<port>` to point to `roscore`

MESSAGES

Topics are TCP ports on which data is exchanged.

The data is **serialized** using a format specific to each type of data: ROS defines its **data interface** with *messages*.

MESSAGES

Topics are TCP ports on which data is exchanged.

The data is **serialized** using a format specific to each type of data: ROS defines its **data interface** with *messages*.

```
> rosmsg show geometry_msgs/Pose
geometry_msgs/Point position
  float64 x
  float64 y
  float64 z
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
```

Source: geometry_msgs::Pose definition

MESSAGE EXAMPLE: JOINT STATE

```
> rosmsg show sensor_msgs/JointState
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string[] name
float64[] position
float64[] velocity
float64[] effort
```

Source: sensor_msgs::JointState definition

MESSAGE EXAMPLE: IMAGE

```
> rosmsg show sensor_msgs/Image
std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

Source: sensor_msgs::Image definition

MESSAGES CONTENT

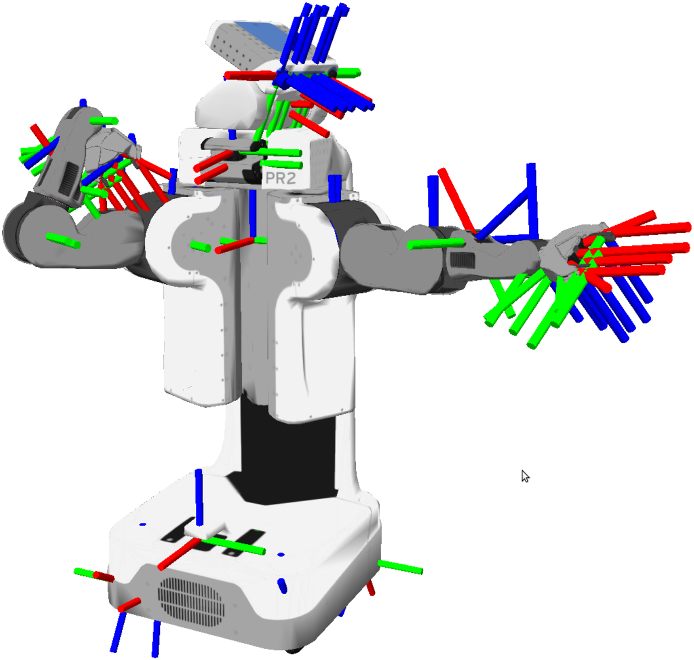
```
> rostopic echo /camera/image_raw
header:
    seq: 56
    stamp:
      secs: 1449243166
      nsecs: 415330019
    frame_id: /camera_frame
height: 720
width: 1280
encoding: rgb8
is_bigendian: 0
step: 3840
data: [32, 57, 51, 36, 61, 55, 41, 63, 60,...
```

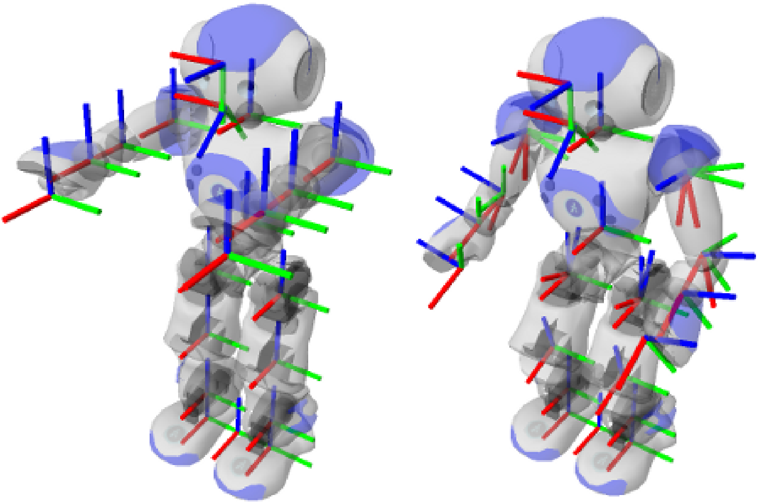
What are these ***frames***?

What are these *frames*?

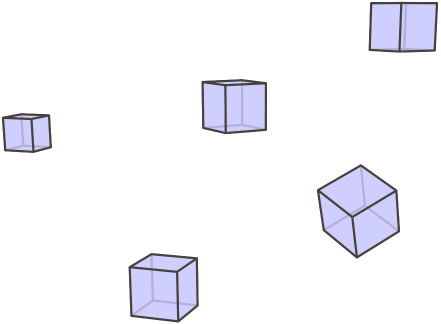
A **frame** is a labelled orthogonal basis with a convenient (6D) origin.

Each part of the robot has usually its own frame, sensors have their frames, objects in the environment have their frames, etc.

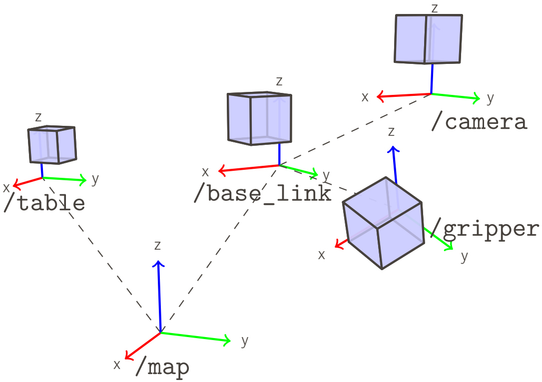

PR2



FRAMES

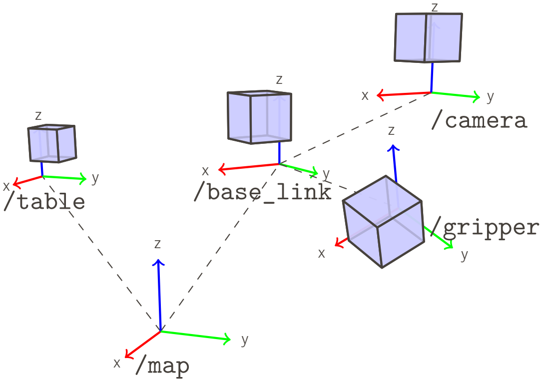


FRAMES



The **TF** library is responsible for maintaining the full transformation tree, and calculating the transformation between any two frames.

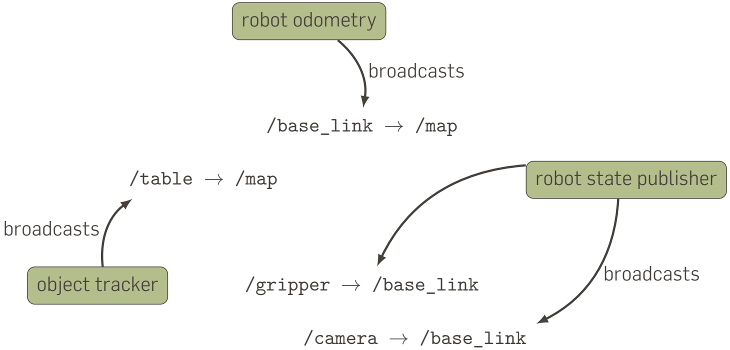
FRAMES



Attention: even though they look similar, frames and topics are unrelated.

"CREATING" FRAMES

Frames come to existence as soon as someone (a node) broadcast them.



HOW TO WRITE A TF BROADCASTER?

C++

```cpp
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>


int main(int argc, char** argv){

  float x=0.f,y=0.f,theta=0.f;

  ros::init(argc, argv, "my_tf_broadcaster");
  tf::TransformBroadcaster br;
  ros::Rate rate(10); // 10 hz

  while (ros::ok()) {
    tf::Transform transform(
                tf::Quaternion(0, 0, theta),
                tf::Vector3(x, y, 0.0));

    br.sendTransform(
         tf::StampedTransform(transform,
                       ros::Time::now(),
                       "my_robot", "map"));


    x++;
    rate.sleep();
  }
  return 0;
};
```

Python

```python
import rospy
import tf
from tf.transformations import quaternion_from_euler

if __name__ == '__main__':

   x = 0.; y = 0.; theta = 0.

   rospy.init_node('my_tf_broadcaster')
   br = tf.TransformBroadcaster()
   rate = rospy.Rate(10) # 10hz

   while not rospy.is_shutdown():

       br.sendTransform(
           (x, y, 0),
           quaternion_from_euler(0, 0, theta),
           rospy.Time.now(),
           "my_robot", "map")

       x += 1
       rate.sleep()
```

```
> rostopic echo tf
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 1449488936
        nsecs: 480597909
      frame_id: map
    child_frame_id: my_robot
    transform:
      translation:
        x: 239.0
        y: 0.0
        z: 0.0
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 1.0
```

TO SUMMARIZE: KEY CONCEPTS

- Node
- Master
- Messages
- Topics
- Services
- Actions
- Transformations/frames

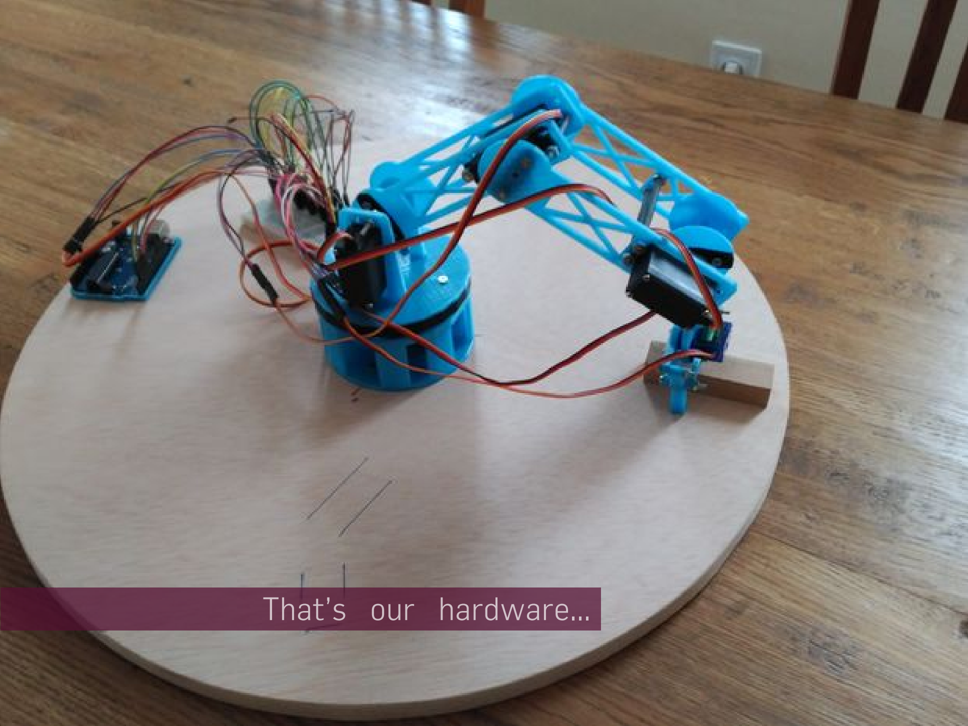
TO SUMMARIZE: KEY CONCEPTS

- Node
- Master
- Messages
- Topics
- Services
- Actions
- Transformations/frames

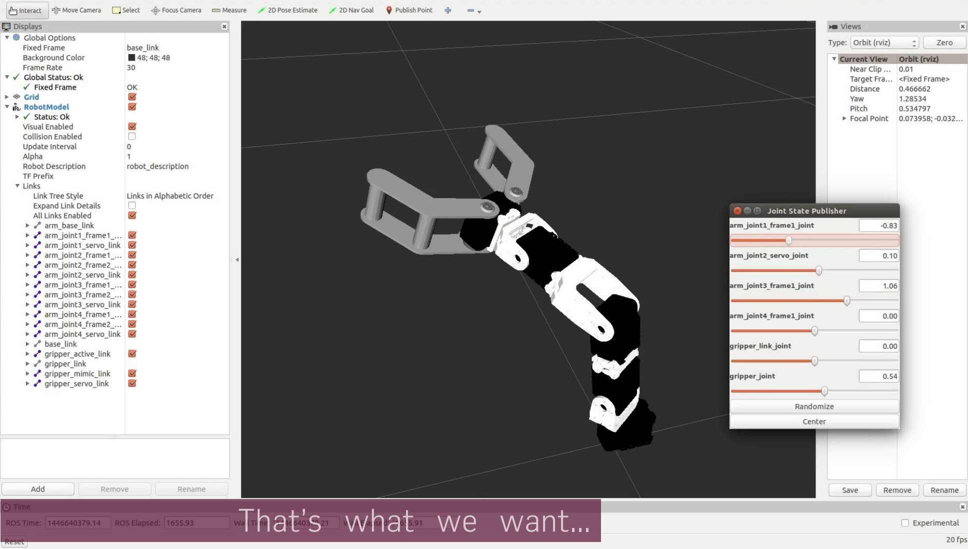
Some additional concepts that we will discuss later on:

- Package
- Launch file

EXAMPLE: A ROBOTIC ARM WITH ROS

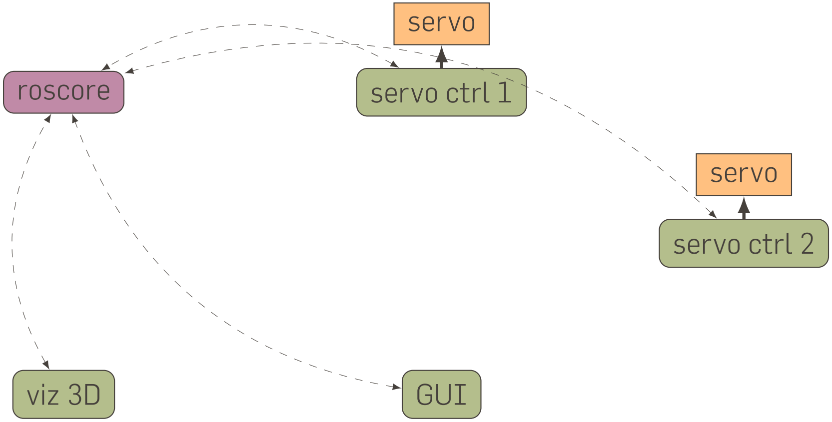


That’s our hardware...

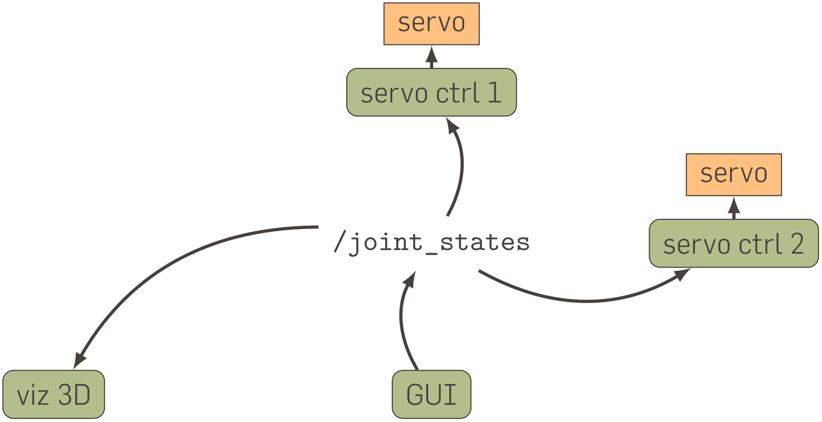


That's what we want...

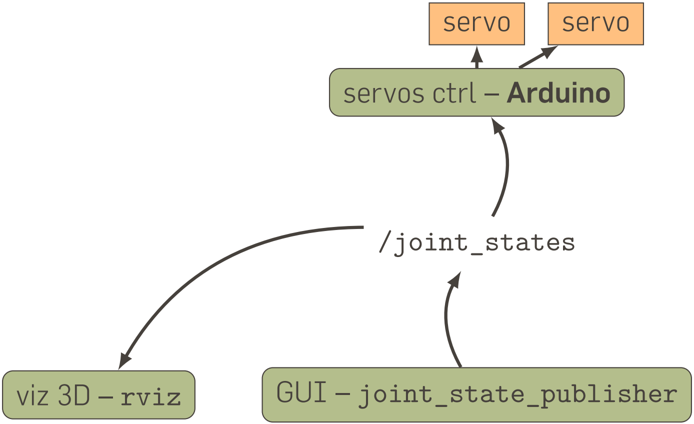
NODES



NODES

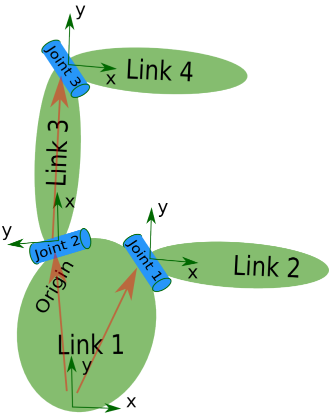


NODES


servo
servo
servos ctrl – **Arduino**
/joint_states
viz 3D – rviz
GUI – joint_state_publisher

URDF

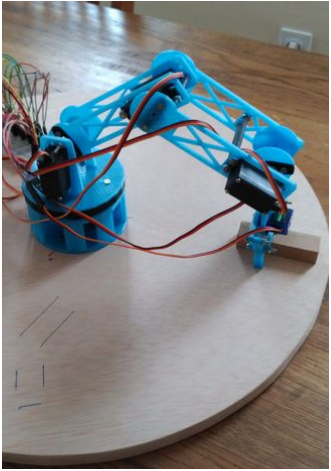
URDF (*Unified Robot Description Format*) is an XML-based language to describe a robot.



- Primitives (cylinders, cubes, spheres) to describe the geometry
- For complex geometries, any STL or Collada meshes can be used
- Only ***tree structures*** can be represented: no parallel robots
- Only ***rigid links*** can be represented: no soft robots

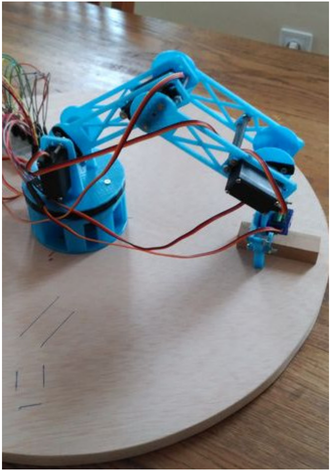
URDF: DESCRIBING THE KINEMATICS OF OUR ROBOT

```xml
<?xml version="1.0"?>
<robot name="roco_arm">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.06" radius="0.1"/>
      </geometry>
    </visual>
  </link>

</robot>
```



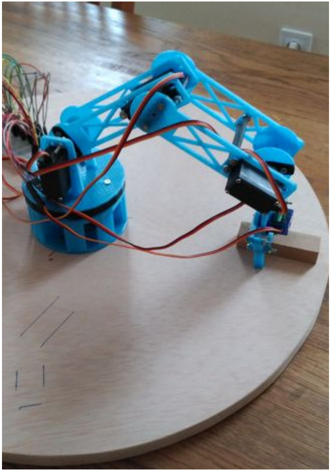
URDF: DESCRIBING THE KINEMATICS OF OUR ROBOT

```xml
<?xml version="1.0"?>
<robot name="roco_arm">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.06" radius="0.1"/>
      </geometry>
    </visual>
  </link>

  <link name="first_segment">
    <visual>
      <geometry>
        <box size="0.6 0.05 0.1"/>
      </geometry>
      <origin rpy="0 0 0" xyz="-0.3 0 0" />
    </visual>
  </link>




</robot>
```



URDF: DESCRIBING THE KINEMATICS OF OUR ROBOT

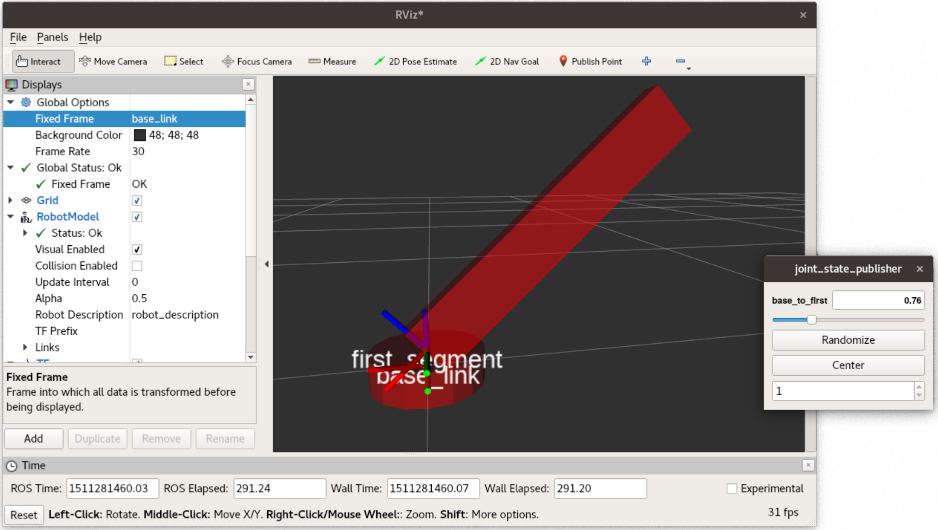
```xml
<?xml version="1.0"?>
<robot name="roco_arm">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.06" radius="0.1"/>
      </geometry>
    </visual>
  </link>

  <link name="first_segment">
    <visual>
      <geometry>
        <box size="0.6 0.05 0.1"/>
      </geometry>
      <origin rpy="0 0 0" xyz="-0.3 0 0" />
    </visual>
  </link>

  <joint name="base_to_first" type="revolute">
    <axis xyz="0 1 0" />
    <limit effort="1000" lower="0"
                upper="3.14" velocity="0.5" />
    <parent link="base_link"/>
    <child link="first_segment"/>
    <origin xyz="0 0 0.03" />
  </joint>
</robot>
```

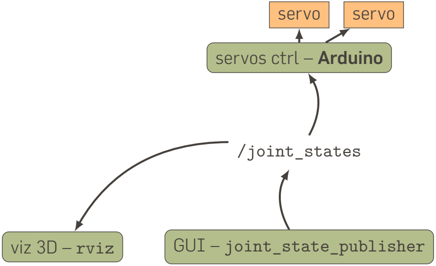


DISPLAY THE MODEL

To display and interact with the URDF model:

```
> rosparam set robot_description -t code/robot-arm.urdf
> rosrun robot_state_publisher robot_state_publisher
> rosrun joint_state_publisher joint_state_publisher _use_gui:=true
> rosrun rviz rviz
```

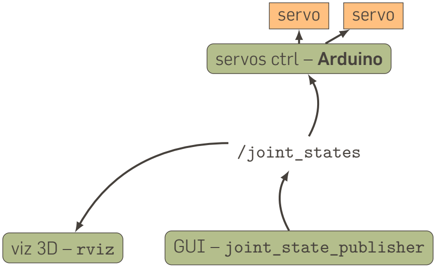

RViz*
File
Panels
Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Fixed Frame
base_link
Background Color
48; 48; 48
Frame Rate
30
Global Status: Ok
Fixed Frame
OK
Grid
RobotModel
Status: Ok
Visual Enabled
Collision Enabled
Update Interval
0
Alpha
0.5
Robot Description
robot_description
TF Prefix
Links
Fixed Frame
Frame into which all data is transformed before being displayed.
Add
Duplicate
Remove
Rename
first_segment
base_link
joint_state_publisher
base_to_first
0.76
Randomize
Center
1
Time
ROS Time:
1511281460.03
ROS Elapsed:
291.24
Wall Time:
1511281460.07
Wall Elapsed:
291.20
Experimental
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
31 fps



Why do we need `robot_state_publisher`?

`rviz` needs the transformations between each geometry. Going from a **joint state** (i.e. the angles for each joint) to transformations (i.e. **frames**) requires **forward kinematics**.

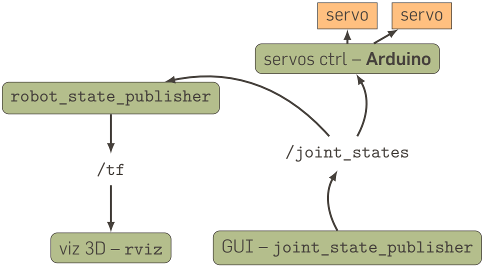
robot_state_publisher ***subscribes*** to the joint state topic `/joint_states` and ***broadcasts*** the corresponding TF frames.



Why do we need `robot_state_publisher`?

`rviz` needs the transformations between each geometry. Going from a **joint state** (i.e. the angles for each joint) to transformations (i.e. **frames**) requires **forward kinematics**.

robot_state_publisher ***subscribes*** to the joint state topic `/joint_states` and ***broadcasts*** the corresponding TF frames.



Why do we need `robot_state_publisher`?

`rviz` needs the transformations between each geometry. Going from a **joint state** (i.e. the angles for each joint) to transformations (i.e. **frames**) requires **forward kinematics**.

`robot_state_publisher` *subscribes* to the joint state topic `/joint_states` and *broadcasts* the corresponding TF frames.

JOINT STATE

```
> rosmsg show sensor_msgs/JointState
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string[] name
float64[] position
float64[] velocity
float64[] effort
```

Source: sensor_msgs::JointState definition

READING THE JOINT STATE ON AN ARDUINO

```
#include <Servo.h>
#include <ros.h>
#include <sensor_msgs/JointState.h>

using namespace ros;

NodeHandle  nh;
Servo servo;

void cb( const sensor_msgs::JointState& msg){
  int angle = (int) (msg.position[0] * 180/3.14);
  servo.write(angle); // 0-180
}

Subscriber<sensor_msgs::JointState> sub("joint_states", cb);

void setup(){
  nh.initNode();
  nh.subscribe(sub);

  servo.attach(9); //attach it to pin 9
}

void loop(){
  nh.spinOnce();
  delay(1);
}
```

CODE MANAGEMENT

PACKAGES

The source code of ROS nodes is usually organised into a ***package***.

```
> cd my_package
> ls
CMakeLists.txt  # describes the build process
cfg/  # node specific configurations
include/  # C/C++ public headers
launch/  # ROS launch files
msgs/  # custom datatypes (messages)
scripts/  # executable scripts, typically Python ROS nodes
package.xml  # package manifest
src/  # source code of your node
> rosrun my_package my_node
```

One package often contains more than one node.

Besides the source code, packages may contain as well specific messages, configuration files, launch files, etc.

PACKAGES

The source code of ROS nodes is usually organised into a ***package***.

```
> cd my_package
> ls
CMakeLists.txt  # describes the build process
cfg/  # node specific configurations
include/  # C/C++ public headers
launch/  # ROS launch files
msgs/  # custom datatypes (messages)
scripts/  # executable scripts, typically Python ROS nodes
package.xml  # package manifest
src/  # source code of your node
> rosrun my_package my_node
```

Packages must contain a ***manifest***, `package.xml`. It contains the package name, version, authors, as well as the dependencies.

PACKAGES

The source code of ROS nodes is usually organised into a ***package***.

```
> cd my_package
> ls
CMakeLists.txt  # describes the build process
cfg/  # node specific configurations
include/  # C/C++ public headers
launch/  # ROS launch files
msgs/  # custom datatypes (messages)
scripts/  # executable scripts, typically Python ROS nodes
package.xml  # package manifest
src/  # source code of your node
> rosrun my_package my_node
```

A new package template can be created easily:

```
> catkin_create_pkg my_package <ROS dependencies>
```

CREATING A ROS PACKAGE, STEP BY STEP

Let's turn the initial 'image processing' example into a proper ROS package.

```
> cd $HOME
> mkdir src && cd src
> catkin_create_pkg imgproc rospy
```

CREATING A ROS PACKAGE, STEP BY STEP

Let's turn the initial 'image processing' example into a proper ROS package.

```
> cd $HOME
> mkdir src && cd src
> catkin_create_pkg imgproc rospy
```

```
> ls imgproc
CMakeLists.txt  package.xml  src
```

FIRST, ADD SOME CODE

```
> cd imgproc   # we now are in $HOME/src/imgproc
> mkdir -p src/imgproc
> cd src/imgproc # we now are in $HOME/src/imgproc/src/imgproc
> touch __init__.py # required to create a Python module
> gedit processing.py
```

FIRST, ADD SOME CODE

```
> cd imgproc   # we now are in $HOME/src/imgproc
> mkdir -p src/imgproc
> cd src/imgproc # we now are in $HOME/src/imgproc/src/imgproc
> touch __init__.py # required to create a Python module
> gedit processing.py
```

Initially, we simply create a stub of our Python module for image processing:

```python
def process(image):
    print('Image to be processed: ' + image)
```

FIRST, ADD SOME INITIAL CODE

We create as well an executable (our future ROS node) in `scripts/`:

```
> cd ../..  # we now are back to $HOME/src/imgproc
> mkdir -p scripts && cd scripts
> gedit process_node
```

FIRST, ADD SOME INITIAL CODE

We create as well an executable (our future ROS node) in `scripts/`:

```
> cd ../..   # we now are back to $HOME/src/imgproc
> mkdir -p scripts && cd scripts
> gedit process_node
```

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

import imgproc.processing

imgproc.processing.process("my_image")
```

FIRST, ADD SOME INITIAL CODE

We create as well an executable (our future ROS node) in `scripts/`:

```
> cd ../..   # we now are back to $HOME/src/imgproc
> mkdir -p scripts && cd scripts
> gedit process_node
```

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

import imgproc.processing

imgproc.processing.process("my_image")
```

Finally, we make our script executable:

```
> chmod +x process_node
```

CONFIGURE THE PYTHON 'BUILD'

Because our node is written in Python, our `CMakeLists.txt` is simple:

```
cmake_minimum_required(VERSION 2.8.3)
project(imgproc)

find_package(catkin REQUIRED COMPONENTS
  rospy
)

catkin_python_setup()
catkin_package()

install(PROGRAMS
   scripts/process_node
   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

CONFIGURE THE PYTHON 'BUILD'

However, we need a `setup.py` (standard Python `distutils`-based packaging):

```
from distutils.core import setup
from catkin_pkg.python_setup import generate_distutils_setup

# fetch values from package.xml
setup_args = generate_distutils_setup(
                    packages=['imgproc'],
                    package_dir={'': 'src'},
                )

setup(**setup_args)
```

INSTALL THE NODE

We can now install our node:

```
> cd $HOME/src/imgproc  # back to the root of our ROS pkg
> mkdir -p build && cd build
> cmake -DCMAKE_INSTALL_PREFIX=<install prefix> ..
> make install
```

INSTALL THE NODE

We can now install our node:

```
> cd $HOME/src/imgproc  # back to the root of our ROS pkg
> mkdir -p build && cd build
> cmake -DCMAKE_INSTALL_PREFIX=<install prefix> ..
> make install
```

Assuming ROS is correctly installed, we can run our node:

```
> export ROS_PACKAGE_PATH=<prefix>/share:$ROS_PACKAGE_PATH
> rosrun imgproc process_node
Dataset: my_image
```

Add the `export ROS_PACKAGE_PATH...` line to the end of your `~/.bashrc` file, so that you do not have to type it everytime.

IMAGE PROCESSING

Let's update the node `reco` and the library (Python *module*) `processing.py` to perform simple image processing:

`processing.py`:

```
import cv2

def process(image):
    rows, cols, channels = image.shape
    cv2.circle(image, (cols/2, rows/2), 50, (0,0,255), -1)
```

IMAGE PROCESSING

process_node:

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

import sys, rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge

import imgproc.processing

def on_image(image):
    cv_image = bridge.imgmsg_to_cv2(image, "bgr8")
    imgproc.processing.process(cv_image)
    image_pub.publish(bridge.cv2_to_imgmsg(cv_image, "bgr8"))

if __name__ == '__main__':
    rospy.init_node('image_processor')
    bridge = CvBridge()
    image_sub = rospy.Subscriber("image",Image, on_image)
    image_pub = rospy.Publisher("processed_image",Image, queue_size=1)

    while not rospy.is_shutdown():
        rospy.spin()
```

TO USE THE NODE

```
> rosrun usb_cam usb_cam_node
```

```
> rosrun imgproc process_node image:=/usb_cam/image_raw
```

```
> rqt_image_view image:=/processed_image
```

Let’s try it!

LAUNCH FILES

Launch files allow to group together nodes and their configuration.

```
<launch>
  <arg name="model" />
  <arg name="gui" default="true" />

  <param name="robot_description" textfile="$(arg model)" />
  <param name="use_gui" value="$(arg gui)"/>

  <node name="joint_state_publisher" pkg="joint_state_publisher"
                              type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher"
                              type="state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" required="true" />

</launch>
```

LAUNCH FILES

```
<launch>
  <arg name="model" />
  <arg name="gui" default="true" />

  <param name="robot_description" textfile="$(arg model)" />
  <param name="use_gui" value="$(arg gui)"/>

  <node name="joint_state_publisher" pkg="joint_state_publisher"
                                     type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher"
                                     type="state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" required="true" />

</launch>
```

To use the launch file, call `roslaunch` instead of `rosrun`:

```
> roslaunch my_package interactive_arm.launch
```

LAUNCH FILES

```
<launch>
  <arg name="model" />
  <arg name="gui" default="true" />

  <param name="robot_description" textfile="$(arg model)" />
  <param name="use_gui" value="$(arg gui)"/>

  <node name="joint_state_publisher" pkg="joint_state_publisher"
                                    type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher"
                                    type="state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" required="true" />

</launch>
```

Arguments (`<arg>`) can be provided from the command-line:

```
> roslaunch my_package interactive_arm.launch
model:=my_arm.urdf
```

LAUNCH FILES

```
<launch>
  <arg name="model" />
  <arg name="gui" default="true" />

  <param name="robot_description" textfile="$(arg model)" />
  <param name="use_gui" value="$(arg gui)"/>

  <node name="joint_state_publisher" pkg="joint_state_publisher"
                                     type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher"
                                     type="state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" required="true" />

</launch>
```

Parameters (`<param>`) are loaded to the ROS *parameter server* and shared with all the ROS nodes.

LAUNCH FILES

```
<launch>
  <arg name="model" />
  <arg name="gui" default="true" />

  <param name="robot_description" textfile="$(arg model)" />
  <param name="use_gui" value="$(arg gui)"/>

  <node name="joint_state_publisher" pkg="joint_state_publisher"
                                    type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher"
                                    type="state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" required="true" />

</launch>
```

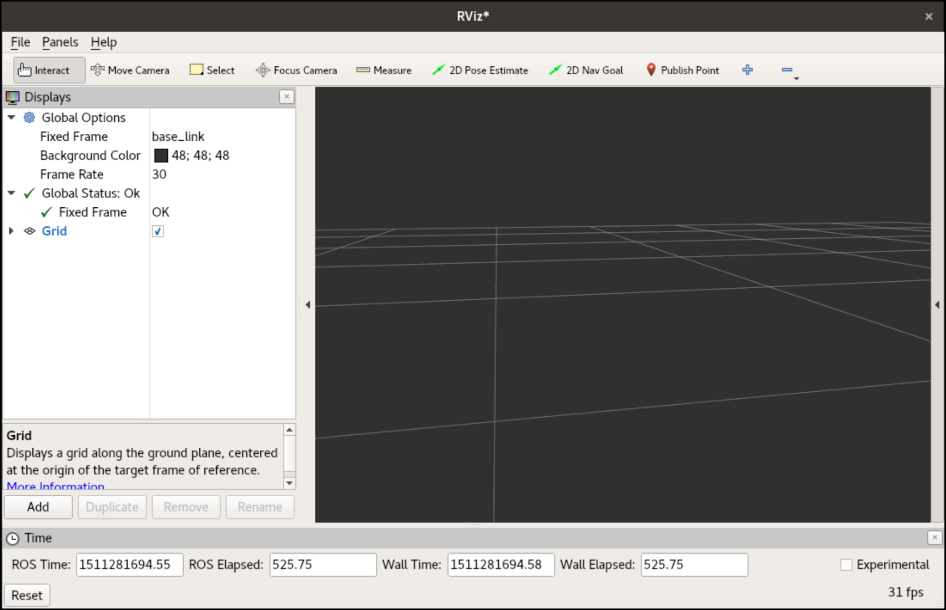
Some nodes may be marked as `required`: if they die (closed or crashed), all the other nodes in the launch file are killed as well.

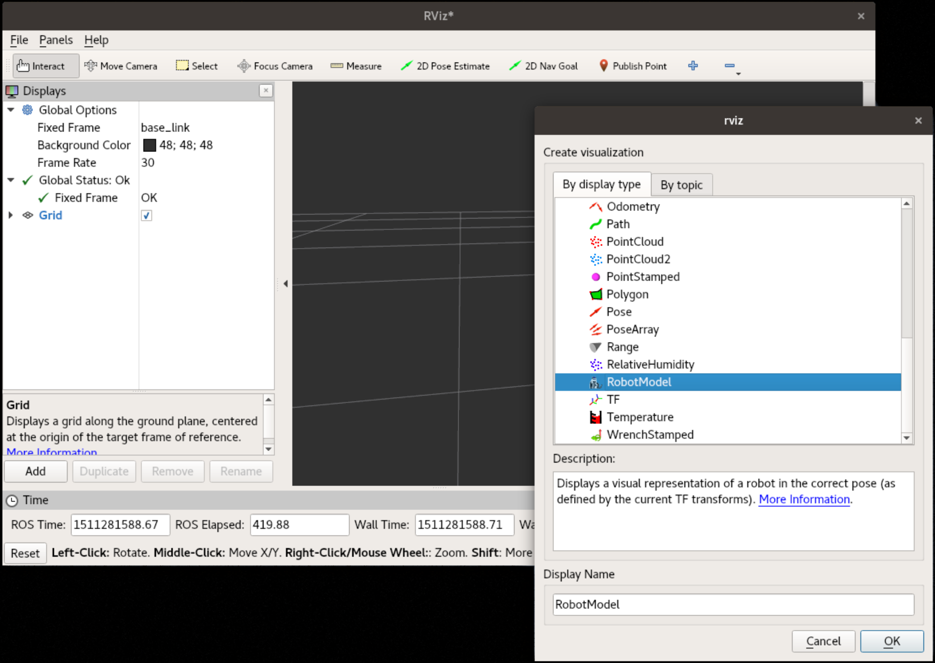
LAUNCH FILES

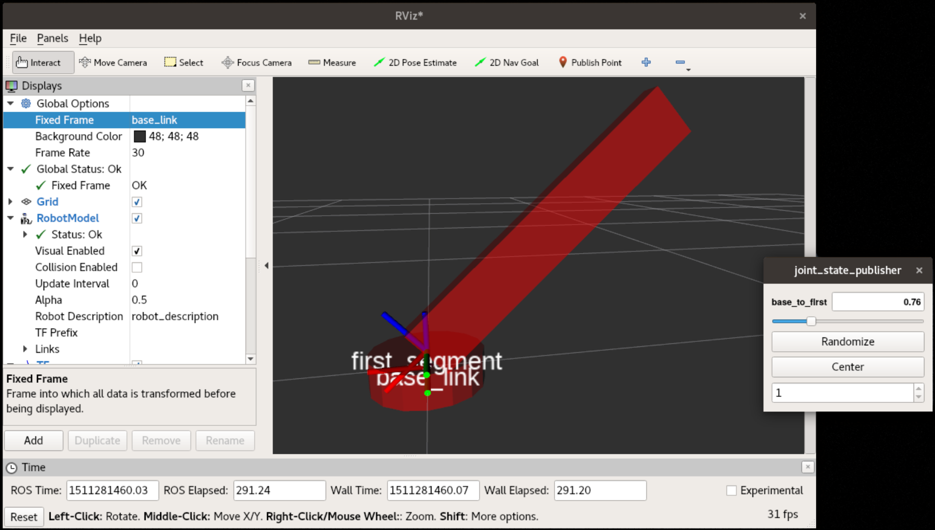
```
<launch>
  <arg name="model" />
  <arg name="gui" default="true" />

  <param name="robot_description" textfile="$(arg model)" />
  <param name="use_gui" value="$(arg gui)"/>

  <node name="joint_state_publisher" pkg="joint_state_publisher"
                              type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher"
                              type="state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" required="true" />

</launch>
```

Many more possibilities, see roslaunch documentation.

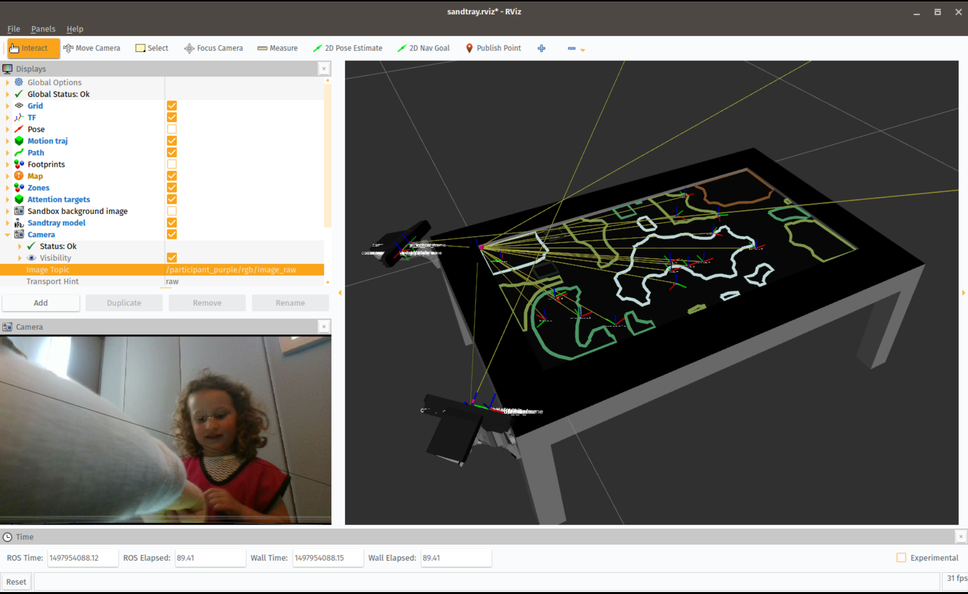
TOOLS

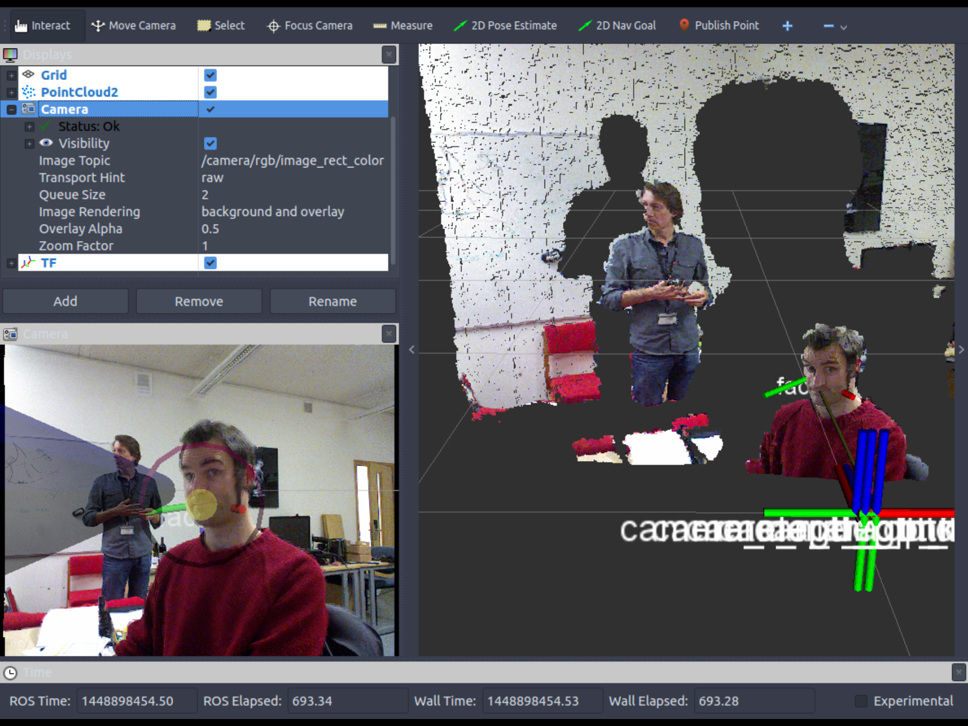

RViz*
File
Panels
Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Fixed Frame
base_link
Background Color
48; 48; 48
Frame Rate
30
Global Status: Ok
Fixed Frame
OK
Grid
Grid
Displays a grid along the ground plane, centered at the origin of the target frame of reference.
More Information
Add
Duplicate
Remove
Rename
Time
ROS Time:
1511281694.55
ROS Elapsed:
525.75
Wall Time:
1511281694.58
Wall Elapsed:
525.75
Experimental
Reset
31 fps

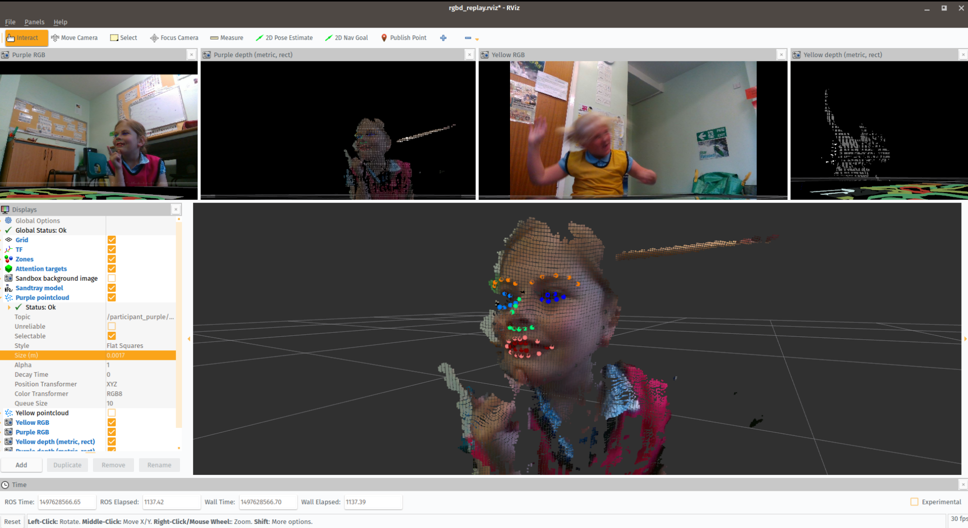

RViz*
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point
Displays
Global Options
Fixed Frame base_link
Background Color 48; 48; 48
Frame Rate 30
Global Status: Ok
Fixed Frame OK
Grid
Grid
Displays a grid along the ground plane, centered at the origin of the target frame of reference.
More Information
Add Duplicate Remove Rename
Time
ROS Time: 1511281588.67 ROS Elapsed: 419.88 Wall Time: 1511281588.71
Reset Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More
rviz
Create visualization
By display type By topic
Odometry
Path
PointCloud
PointCloud2
PointStamped
Polygon
Pose
PoseArray
Range
RelativeHumidity
RobotModel
TF
Temperature
WrenchStamped
Description:
Displays a visual representation of a robot in the correct pose (as defined by the current TF transforms). More Information.
Display Name
RobotModel
Cancel OK



RViz*
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point
Displays
Global Options
Fixed Frame base_link
Background Color 48; 48; 48
Frame Rate 30
Global Status: Ok
Fixed Frame OK
Grid
RobotModel
Status: Ok
Visual Enabled
Collision Enabled
Update Interval 0
Alpha 0.5
Robot Description robot_description
TF Prefix
Links
Fixed Frame
Frame into which all data is transformed before being displayed.
Add Duplicate Remove Rename
first_segment
base_link
joint_state_publisher
base_to_first 0.76
Randomize
Center
1
Time
ROS Time: 1511281460.03 ROS Elapsed: 291.24 Wall Time: 1511281460.07 Wall Elapsed: 291.20
Experimental
Reset Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
31 fps


sandtray.rviz* - RViz
File
Panels
Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Global Status: Ok
Grid
TF
Pose
Motion traj
Path
Footprints
Map
Zones
Attention targets
Sandbox background image
Sandtray model
Camera
Status: Ok
Visibility
Image Topic
/participant_purple/rgb/image_raw
Transport Hint
raw
Add
Duplicate
Remove
Rename
Camera
Time
ROS Time: 1497954088.12
ROS Elapsed: 89.41
Wall Time: 1497954088.15
Wall Elapsed: 89.41
Experimental
Reset
31 fps


Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Grid
PointCloud2
Camera
Status: Ok
Visibility
Image Topic
/camera/rgb/image_rect_color
Transport Hint
raw
Queue Size
2
Image Rendering
background and overlay
Overlay Alpha
0.5
Zoom Factor
1
TF
Add
Remove
Rename
Camera
Time
ROS Time:
1448898454.50
ROS Elapsed:
693.34
Wall Time:
1448898454.53
Wall Elapsed:
693.28
Experimental



rgbd_replay.rviz* - RViz
File
Panels
Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Purple RGB
Purple depth (metric, rect)
Yellow RGB
Yellow depth (metric, rect)
Displays
Global Options
Global Status: Ok
Grid
TF
Zones
Attention targets
Sandbox background image
Sandtray model
Purple pointcloud
Status: Ok
Topic
/participant_purple/...
Unreliable
Selectable
Style
Flat Squares
Size (m)
0.0017
Alpha
1
Decay Time
0
Position Transformer
XYZ
Color Transformer
RGB8
Queue Size
10
Yellow pointcloud
Yellow RGB
Purple RGB
Yellow depth (metric, rect)
Add
Duplicate
Remove
Rename
Time
ROS Time: 1497628566.65
ROS Elapsed: 1137.42
Wall Time: 1497628566.70
Wall Elapsed: 1137.39
Experimental
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
30 fps

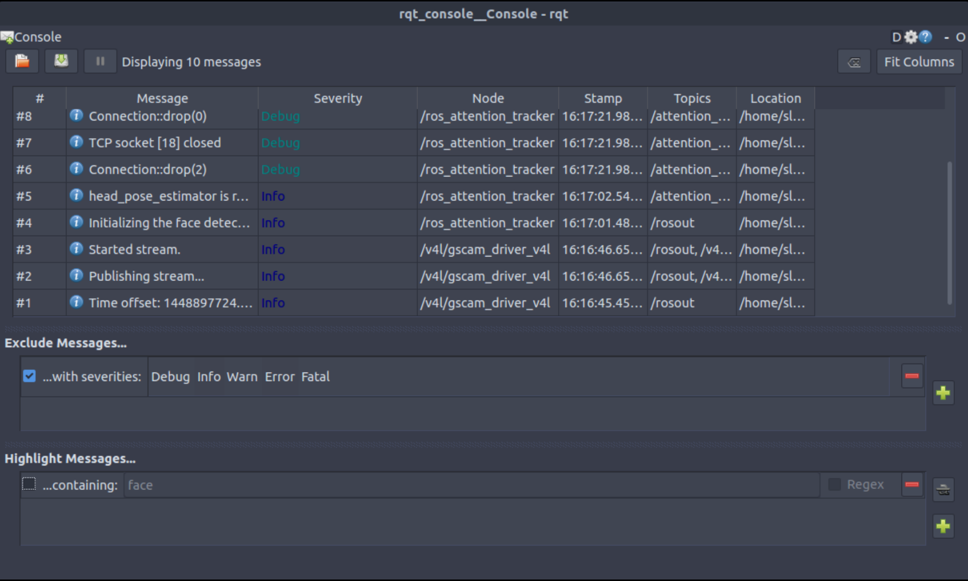
OTHER TOOLS

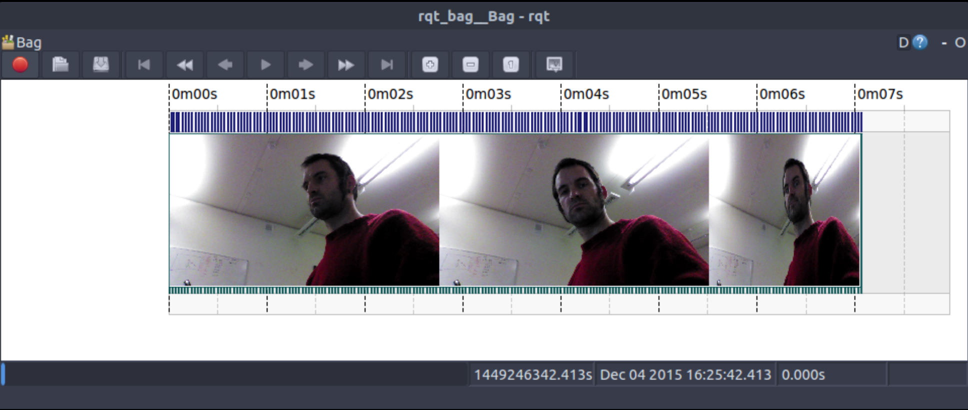
`rosnode`, `rostopic`,...	Print out/publish/call messages, services, nodes
`rosconsole`	Centralized logging
`rosbag`	Record and replay messages
`rqt_reconfigure`	Live configuration of nodes
`rqt_diagnostics`	Standardized diagnostics
`rosgraph`	plots the node network

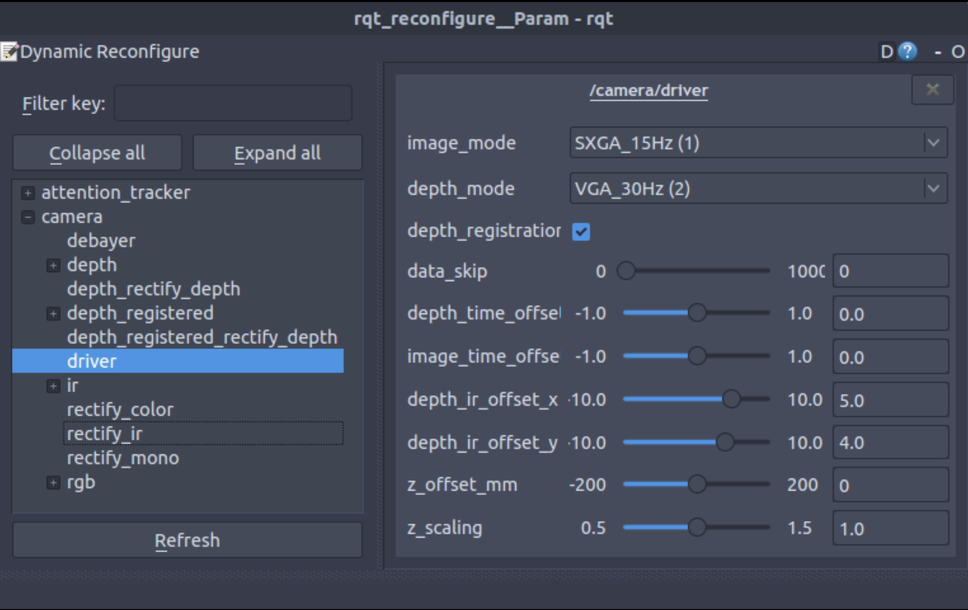
```
> rosnode list
/camera_base_link
/camera_base_link1
/camera_base_link2
/camera_base_link3
/ros_attention_tracker
/rosout
/camera/camera_nodelet_manager
/camera/debayer
/camera/depth_metric
/camera/depth_metric_rect
/camera/depth_points
/camera/depth_rectify_depth
/camera/depth_registered_hw_metric_rect
/camera/depth_registered_metric
/camera/depth_registered_rectify_depth
/camera/depth_registered_sw_metric_rect
/camera/disparity_depth
/camera/disparity_registered_hw
/camera/disparity_registered_sw
```

```
> rostopic list
/camera_info
/image
/nb_detected_faces
/rosout
/rosout_agg
/tf
/camera/depth/image_rect_raw
/camera/depth/image_rect_raw/compressed
/camera/depth/image_rect_raw/compressed/parameter_descriptions
/camera/depth/image_rect_raw/compressed/parameter_updates
/camera/depth/image_rect_raw/compressedDepth
/camera/depth/image_rect_raw/compressedDepth/parameter_descript
/camera/depth/image_rect_raw/compressedDepth/parameter_updates
/camera/depth/image_rect_raw/theora
/camera/depth/image_rect_raw/theora/parameter_descriptions
/camera/depth/image_rect_raw/theora/parameter_updates
/camera/depth_rectify_depth/parameter_descriptions
/camera/depth_rectify_depth/parameter_updates
```

```
> rostopic echo tf
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 1449222890
        nsecs: 396561780
      frame_id: /camera_link
    child_frame_id: /camera_rgb_frame
    transform:
      translation:
        x: 0.0
        y: -0.045
        z: 0.0
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 1.0
```


rqt_console__Console - rqt
Console
Displaying 10 messages
Fit Columns
| Message | Severity | Node | Stamp | Topics | Location
#8 | Connection::drop(0) | Debug | /ros_attention_tracker | 16:17:21.98... | /attention_... | /home/sl...
#7 | TCP socket [18] closed | Debug | /ros_attention_tracker | 16:17:21.98... | /attention_... | /home/sl...
#6 | Connection::drop(2) | Debug | /ros_attention_tracker | 16:17:21.98... | /attention_... | /home/sl...
#5 | head_pose_estimator is r... | Info | /ros_attention_tracker | 16:17:02.54... | /attention_... | /home/sl...
#4 | Initializing the face detec... | Info | /ros_attention_tracker | 16:17:01.48... | /rosout | /home/sl...
#3 | Started stream. | Info | /v4l/gscam_driver_v4l | 16:16:46.65... | /rosout, /v4... | /home/sl...
#2 | Publishing stream... | Info | /v4l/gscam_driver_v4l | 16:16:46.65... | /rosout, /v4... | /home/sl...
#1 | Time offset: 1448897724.... | Info | /v4l/gscam_driver_v4l | 16:16:45.45... | /rosout | /home/sl...
Exclude Messages...
...with severities: Debug Info Warn Error Fatal
Highlight Messages...
...containing: face
Regex


rqt_bag__Bag - rqt
Bag
0m00s
0m01s
0m02s
0m03s
0m04s
0m05s
0m06s
0m07s
1449246342.413s
Dec 04 2015 16:25:42.413
0.000s


rqt_reconfigure__Param - rqt
Dynamic Reconfigure
Filter key:
Collapse all
Expand all
attention_tracker
camera
debayer
depth
depth_rectify_depth
depth_registered
depth_registered_rectify_depth
driver
ir
rectify_color
rectify_ir
rectify_mono
rgb
Refresh
/camera/driver
image_mode
SXGA_15Hz (1)
depth_mode
VGA_30Hz (2)
depth_registratior
data_skip
0
100(
0
depth_time_offsel
-1.0
1.0
0.0
image_time_offse
-1.0
1.0
0.0
depth_ir_offset_x
-10.0
10.0
5.0
depth_ir_offset_y
-10.0
10.0
4.0
z_offset_mm
-200
200
0
z_scaling
0.5
1.5
1.0

ROS DOCUMENTATION

ROS DOCUMENTATION

Plenty!

Some examples:

- Tutorial: wiki.ros.org/ROS/Tutorials
- Supported robots: robots.ros.org
- Message definition example: sensor_msgs/LaserScan
- Node documentation example: wiki.ros.org/face_detector

SO, WHAT IS ROS?

ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind

ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)

ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules

ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules
- A set of conventions to write and package robotic softwares

ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules
- A set of conventions to write and package robotic softwares
- Deep integration of a few key open-source libraries (OpenCV, PCL, tf)

ROS IS...

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules
- A set of conventions to write and package robotic softwares
- Deep integration of a few key open-source libraries (OpenCV, PCL, tf)
- A set of tools to run and monitor the nodes

That's all, folks!

Questions:
Portland Square B316 or **severin.lemaignan@plymouth.ac.uk**

Slides:
github.com/severin-lemaignan/module-mobile-and-humanoid-robots