ROS tutorial

Thomas Moulard

LAAS robotics courses, January 2012

So what is ROS?

- A component oriented robotics framework,
- A development suite,
- A (bad) package management system,
- An (active) community.

Why should I care?

- Because a PhD is short.
- Because we are all "standing on the shoulders of giants".
  If you are currently implementing one of the following things, you are *doing it wrong*: quaternion, rotation matrix, kinematics, graph computation, etc.
- Because some of you will work on the PR2 and have no choice ;)
- Because there is cool stuff on ROS...

But first...

We will play with ROS and ROS is huuuuuuge. If you do not have it yet on your laptop, install it now!
If you do not have a Ubuntu system, work with someone as two hours is probably not enough to download and compile ROS entirely!

Instructions are here:
`http://ros.org/wiki/ROS/Installation`

ROS as a framework (1)

The ROS framework is **component oriented**.
Each component is called a **node**. **Nodes** communicate using **topics** or **services**.

topics represents data-flow. For instance: camera images, robot configuration or position can be model as topics. Topics values are often published regularly to keep the whole system up-to-date.

services represents queries which are sent asynchroneously, and usually at a low frame-rate. Slow components such as motion planning nodes for instance usually provide services.

Topics (1)

Each **node** can **listen** or **publish** on a topic.
Messages types are defined using a dedicated syntax which is ROS specific:
MyMessage.msg

```
# this is a very useful comment!
Float64 myDouble
String myString
Float64[] myArrayOfDouble
```

Topics (2)

The authorized types are:

Built-in types string, float (32 or 64 bits), integer, booleans and Header.

Messages defined in other packages `MyOtherPackage/CustomMessage`

Arrays of authorized types `WhatEver[]` is an array of `WhatEver`.

It produces structures which can be serialized and unserialized automatically and transparently between different platforms and programming languages. ROS supports mainly C++ and Python for its API and Linux (Ubuntu) for its platforms.

This is changing slowly (more platforms, less/different programming languages).

Services

Each **node** can **call** or **provide** one or more services.
To declare a type of service, the syntax is pretty similar:
MyService.srv

```
Float64 x
Float64 y
---
Float64 result
```

The dashes separates the query type from the response type. The authorized types are the same.

Parameters

Each **node** can also **set** or **get** one or more parameters. Some are **public** and can be modified by other **nodes**. The other are **private** and are defined at startup only.

In the ROS documentation:

`publicParameter` is a public parameter,

`~privateParameter` is a private parameter.

Optionally, `dynamic_reconfigure` can be used to change the parameters in a GUI (or from command line) while the **node** is running. It can be used to change the camera grabbing framerate for instance.

ROS as a framework (2)

Now that we have `nodes` with input and output, how do we make them communicate together?

If **A** publishes on *foo* and **B** listens on *foo*, **B** will receive topics data from **A**.

The services and topics names are used to match clients and servers.

ROS introspection tools

Let's play around:

```
# Setup the environment variables needed by ROS.
source /opt/ros/electric/setup.bash
# Start the nameserver.
roscore
# Start the turtlesim_node which is in the turtlesim pkg.
rosrun turtlesim turtlesim_node
# List nodes
rosnode list
# Teleoperating the turtle.
rosrun turtlesim turtle_teleop_key
# Display the graph.
rxgraph
# Display the velocity
rostopic echo /turtle1/command_velocity
```

ROS introspection tools (2)

```
# Show message type.
rosmsg show turtlesim/Velocity
# Publish velocity instead of using the teleoperation node.
rostopic pub -1 /turtle1/command_velocity turtlesim/Velocity -- 2.0 1.8
rostopic pub /turtle1/command_velocity turtlesim/Velocity -r 1 -- 2.0 -1.8

# See how often the robot pose is refreshed.
rostopic hz /turtle1/pose

# Plot the pose.
rxplot /turtle1/pose/x,/turtle1/pose/y /turtle1/pose/theta
```

ROS introspection tools (3)

```
# List service.
rosservice list
# Show service description.
rosservice type spawn | rossrv show
# Create a new turtle by creating a service.
rosservice call spawn 2 2 0.2 ""
# ROS parameter list.
rosparam list
# Set and get background colors using rosparam
rosparam set background_r 150
rosparam get background_g
# Display parameters
rosparam get /
# Save parameters
rosparam dump params.yaml
# Load parameters into new namespace copy
rosparam load params.yaml copy
rosparam get copy/background_b
```

ROS introspection tools (4)

```
# Display debug information
rxloggerlevel
rxconsole
```

ROS launch files

A usual robotics behaviors is implemented by several **nodes** launched with specific parameters.

To do so, the easiest way is to use `roslaunch`.

`roslaunch` starts a set of nodes using an XML description:

```
<launch>
  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>
  <group ns="turtlesim2">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>
  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>
</launch>
```

tf: the Transform Broadcaster

tf is the ROS transformation directory. A classic architecture problems is the following: several independent modules computes transformation between bodies and publishes them between at a different frame-rate. Therefore, how to make sure that the computed transformations are valid?

```
roslaunch turtle_tf turtle_tf_demo.launch
rosrun tf view_frames
rosrun tf tf_echo turtle1 turtle2
rosrun rviz rviz -d \
  `rospack find turtle_tf`/rviz/turtle_rviz.vcg
```

tf: the Transform Broadcaster (2)

- **tf** avoids having to concatenate transformations correctly. It also motivates developers to set correctly the frame id so that everything works "out of the box", instead of relying on the documentation which is often obsolete.
- **tf** remembers the past up-to 10 seconds before (by default).
- **tf** can interpolate data to compute a transformation at a particular point of the past.

URDF: ROS robot model format

- **URDF** is the Unified Robot Description Format.
- It is used by ROS to describe the robot kinematics chain, dynamic and physical properties. It also stores an optional alternative representation for collision checking and a visual representation. These can be either built from basic geometrical shapes or through a COLLADA model.

URDF: ROS robot model format

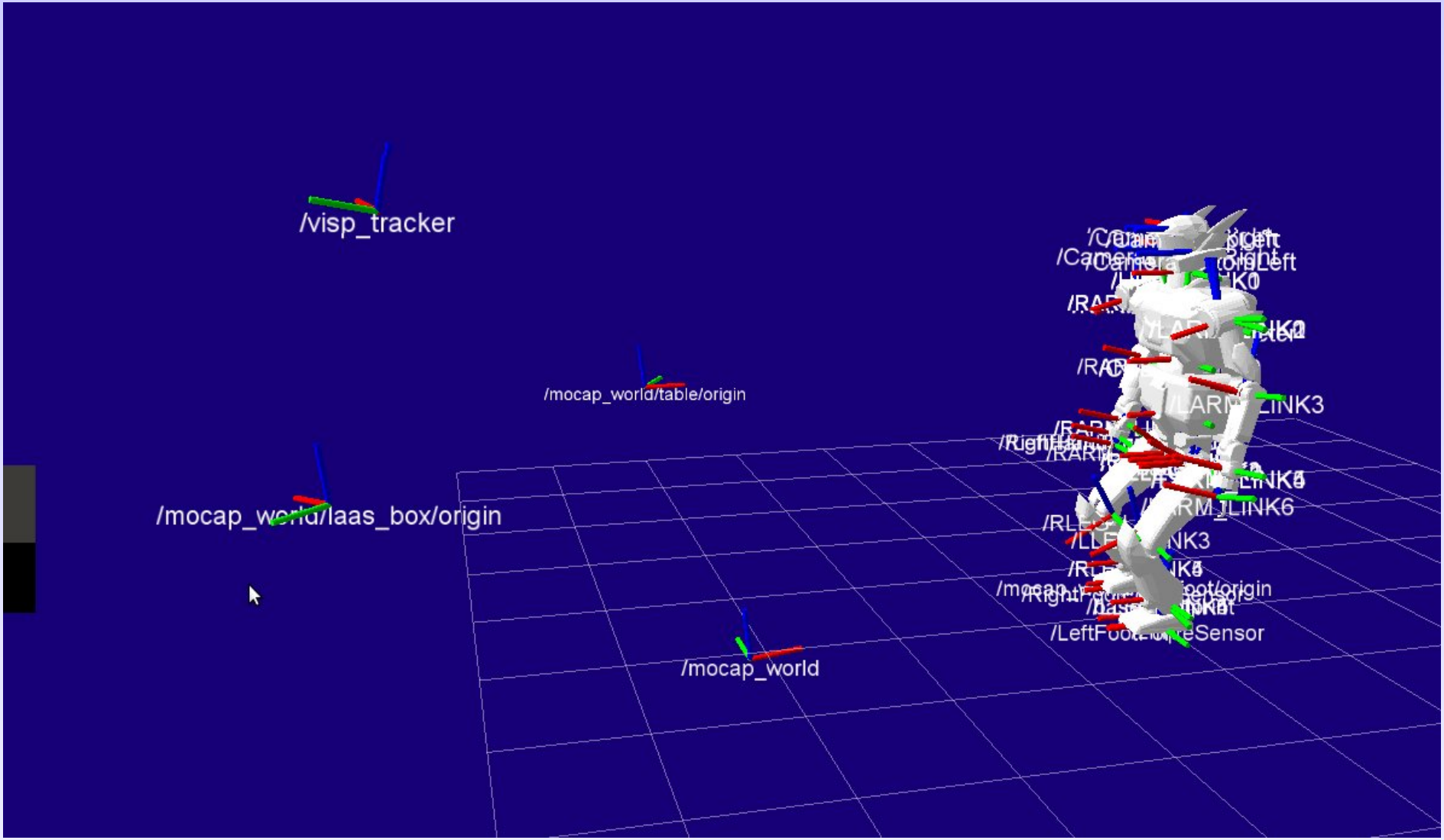
```
<?xml version="1.0"?>
<robot name="multipleshapes">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>
  <link name="right_leg">
    <visual>
      <geometry> <box size="0.6 .2 .1"/> </geometry>
    </visual>
  </link>
  <joint name="base_to_right_leg" type="fixed">
    <parent link="base_link"/>
    <child link="right_leg"/>
  </joint>
</robot>
```

URDF: ROS robot model format

```
roscd urdf_tutorial
roslaunch urdf_tutorial display.launch \
  model:=06-flexible.urdf gui:=True
  rostopic echo /joint_states
rosrun tf view_frames
```

Behind the scenes:

- Set the `robot_description` parameter.
- Start the `robot_state_publisher` (compute forward kinematics using KDL).
- Start an interactive `joint_state_publisher`.


/visp_tracker
/mocap_world/table/origin
/mocap_world/laas_box/origin
/mocap_world

Replaying data using rosbag

Topics can be recorded and replayed using `rosbag` and played step by step. The ROS time API ensures that the time data will remain consistent.

```
# Replay data.
roscore
rosbag play --clock ~/2012-01-12-09-53-29.bag
# Inspect bag file.
rxbag ~/2012-01-12-09-53-29.bag
```

ROS vision pipeline (1)

- ROS provides an image processing pipeline:
  - Image grabbing (1394, USB, etc.),
  - Color conversion,
  - Rectification using calibration data,
  - Disparity images for stereo pairs,
  - Streaming of compressed images to reduce network load.
- Easy to interface with OpenCV.
- Easy to use, automatic calibration method.

ROS vision pipeline (2)

```
rosrun image_view image_view \
  image:=/wide/left/image_raw
ROS_NAMESPACE=/wide/left rosrun image_proc image_proc
rosrun image_view image_view \
  image:=/wide/left/image_mono
rosrun image_view image_view \
  image:=/wide/left/image_rect_color compressed
```

Nodelets

- A limitation when doing vision using robotics components: images serialization/unserialization is extremely costly.
- Nodelets allow different nodes to be loaded into the same namespace to avoid copying data.
- Typically: grabbing, color conversion, rectification and stereo processing may be done in the same node. Unstable experimental algorithms may be run into a separate process to avoid impacting the whole architecture.

Next: Ecto `http://ecto.willowgarage.com/` is a data-flow framework dedicated to sensor/vision processing.

Other interesting packages

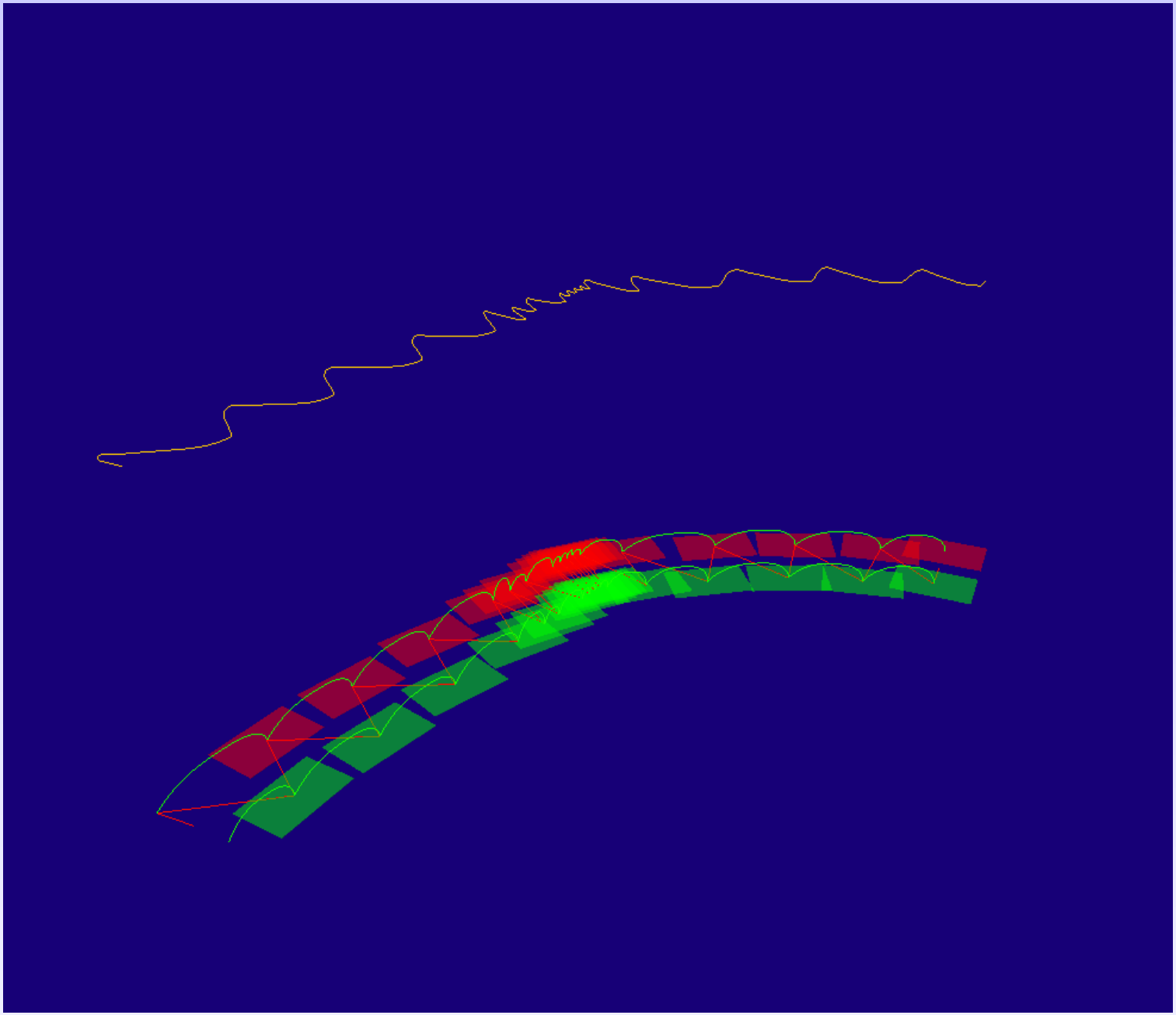
- **PCL** the Point Cloud Library
- **Navigation stack** `ekf_filter`, `robot_self_filter`
- **SLAM**: gmapping, vslam (visual SLAM with bundle adjustment), PTAM...
- **Tracking**: `ar_pose`
- **Motion planning**: OMPL
- **Simulation**: Gazebo
- **Drivers**: IMU, cameras, GPS, Kinect...
- **Finite state machine**: ActionLib...

LAAS packages

- **vision_visp** the ViSP model based tracker
- **motion_analysis_mocap** our MoCap system
- **humanoid_walk** generating walking trajectories for humanoids robots.

The complete list is available here:
`http://www.ros.org/wiki/laas-ros-pkg`

Consider contributing a package, especially if you intend to be hired by a private company after your PhD...



Tutorial: controlling the turtle

- The goal is sending a velocity to the turtle both in C++ and Python.

Initializing ROS and launching the required node

```
roscore
rosrun turtlesim turtlesim_node
```

Tutorial: controlling the turtle (2)

Setting up your workspace

```
mkdir ~/ros
cd ~/ros
mkdir workspace
emacs setup.sh
```

setup.sh

```
#!/bin/sh
export ROS_ROOT=/opt/ros/electric/ros
export PATH=$ROS_ROOT/bin:$PATH
export PYTHONPATH=$ROS_ROOT/core/roslib/src:$PYTHONPATH
if [ ! "$ROS_MASTER_URI" ]; then
 export ROS_MASTER_URI=http://localhost:11311
fi
export ROS_PACKAGE_PATH=$HOME/ros/workspace:/opt/ros/electric/stacks
export ROS_WORKSPACE=$HOME/ros
```

Tutorial: controlling the turtle (3)

Setting up your workspace (2)

```
source setup.sh
source ${ROS_ROOT}/tools/rosbash/rosbash
```

Creating your stack and your first package

```
mkdir -p ~/ros/workspace
cd ~/ros/workspace
roscreate-stack my_stack
roscreate-package my_turtle_controller
roscd my_turtle_controller
```

Tutorial: controlling the turtle (4)

- Edit the `manifest.xml` to add a dependency toward `turtlesim`.
- Edit the `CMakeLists.txt` and add a statement to compile your node (i.e. program). The node is only composed of `src/controller.cpp`
- Edit the `src/controller.cpp` and write your subscriber. You must subscribe to `/turtle1/command_velocity` and send regularly commands.
- Then, do the same thing in Python!