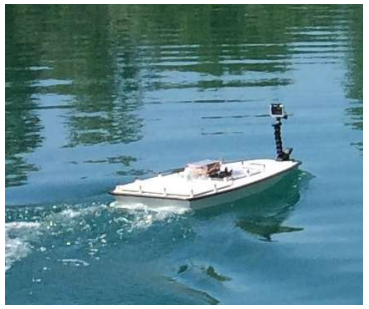
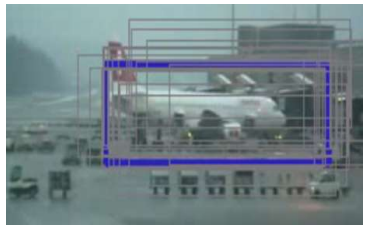
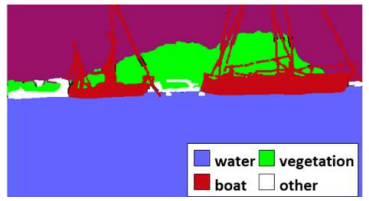
UNIVERSITÀ di VERONA | Dipartimento di INFORMATICA

Laurea magistrale in Ingegneria e scienze informatiche





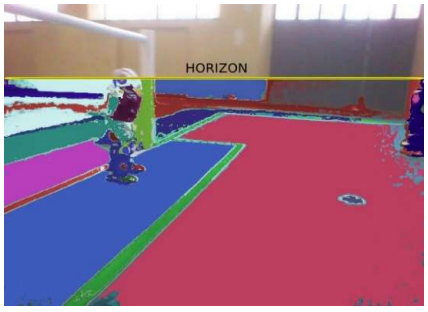




Introduzione a ROS

Corso di Robotica
Parte di Laboratorio

Docente:
Domenico Daniele Bloisi


Novembre 2017

ROS

ROS (Robot Operating System) is an open-source, flexible framework for writing robot software

Site: http://www.ros.org/

Blog: http://www.ros.org/news/

Documentation: http://wiki.ros.org/

Idea

- Use processes to isolate functionalities of the system
- Processes communicate through messages (less efficient than using shared memory, but safer)
- Benefits
  - If a process crashes, it can be restarted
  - A functionality can be exchanged by replacing a process that provides it
  - Decoupling of modules through inter-process communication

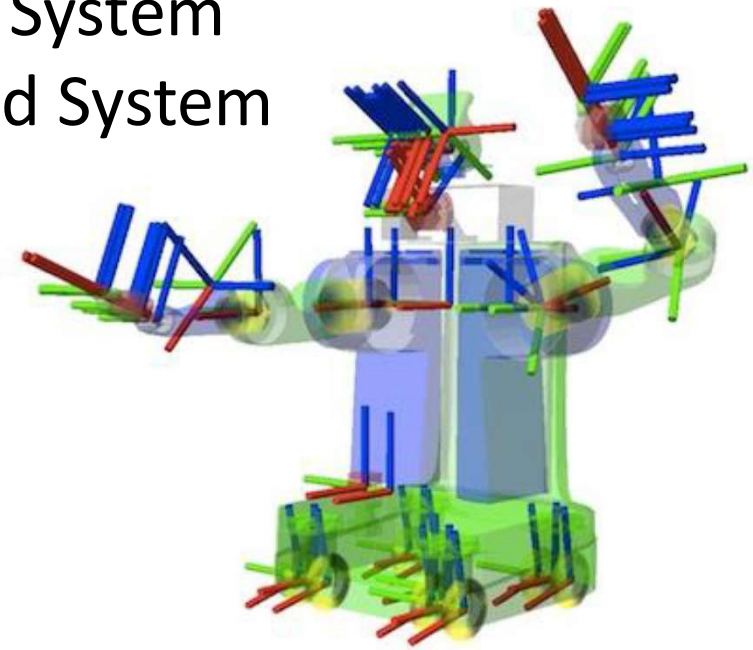
ROS features

- Code reuse (exec. nodes, grouped in packages)
- Distributed, modular design (scalable)
- Language independent (C++, Python, Java, …)
- ROS-agnostic libraries (code is ROS indep.)
- Easy testing (ready-to-use)
- Vibrant community & collaborative environment

Robot specific features

Provides tools for

- Message Definition
- Process Control
- File System
- Build System

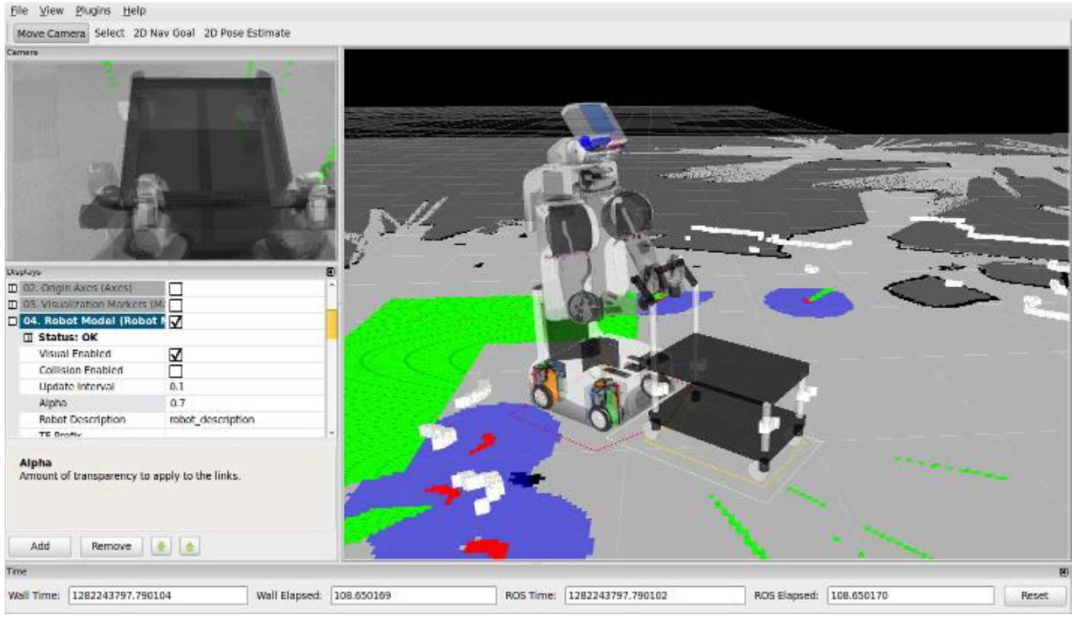


Provides basic functionalities like:

- Device Support
- Navigation
- Control of Manipulator
- Object Recognition

ROS tools

- Command-line tools
- Rviz
- rqt (e.g., rqt_plot, rqt_graph)



Integration with external libraries

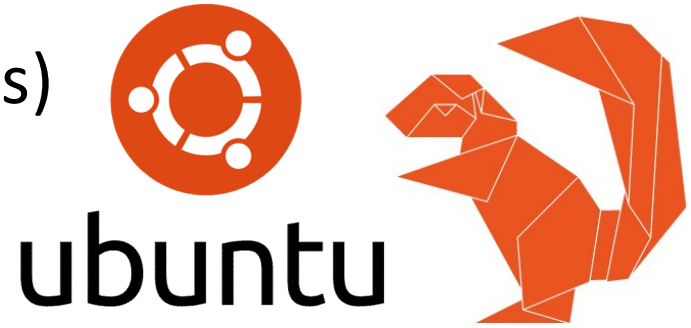
ROS provides seamless integration of external libraries and popular open-source projects

and many others

ROS installation

Suggested OS: Ubuntu 16.04.3 LTS (Xenial Xerus)

Suggested release: Kinetic Kame



- Install ROS from source (not recommended): http://wiki.ros.org/kinetic/Installation/Source
- Install ROS from Debian packages: http://wiki.ros.org/kinetic/Installation/Ubuntu

Post installation

Initialize rosdep in your system:

```
sudo rosdep init
rosdep update
```

rosdep is a tool for checking and installing package dependencies in an OS-independent way

Note: do not use sudo for rosdep_update

ROS filesystem

- **Package**
  unit for organizing software in ROS. Each package can contain libraries, executables, scripts, or other artifacts

- **Manifest** (package.xml)
  meta-information about a package (e.g., version, maintainer, license, etc.) and description of its dependencies (other ROS packages, messages, services, etc.)

http://wiki.ros.org/catkin/package.xml

package.xml

```
<?xml version="1.0"?>
<package>
<name>my_package</name>
<version>1.0</version>
<description>My package description</description>
<!-- One maintainer tag required, multiple allowed, one
person per tag -->
<maintainer email="my@mail.com">Your Name</maintainer>
<!-- One license tag required, multiple allowed, one
license per tag. Commonly used license strings: BSD,
MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1,
LGPLv3 -->
<license>LGPLv3</license>
```

Url tags and Author tags

```
<!-- Url tags are optional, but mutiple are allowed, one per tag.
Optional attribute type can be: website, bugtracker, or repository -
->
<url type="website">http://wiki.ros.org/my_package</url>

<!-- Author tags are optional, mutiple are allowed, one per tag.
Authors do not have to be maintianers, but could be -->
<author email="my@mail.com">Your Name</author>

<!-- The *_depend tags are used to specify dependencies.
Dependencies can be catkin packages or system dependencies. Use
build_depend for packages you need at compile time. Use
buildtool_depend for build tool packages. Use run_depend for
packages you need at runtime. Use test_depend for packages you need
only for testing. -->
```

Dependencies

```
<buildtool_depend>catkin</buildtool_depend>

<build_depend>message_generation</build_depend>
<build_depend>roscpp</build_depend>
<build_depend>roslib</build_depend>

<run_depend>message_runtime</run_depend>
<run_depend>roscpp</run_depend>
<run_depend>roslib</run_depend>

<!-- The export tag contains other, unspecified, tags --> <export>
<!-- You can specify that this package is a metapackage here: -->
<!-- <metapackage/> -->
<!-- Other tools can request additional information be placed here -->

</export>
</package>
```

Catkin workspace configuration

```
$ source /opt/ros/kinetic/setup.bash
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
$ cd ~/catkin_ws/
$ catkin_make
```

Open ~/.bashrc and add the following lines:

```
#ROS
source ~/catkin_ws/devel/setup.bash
```

Catkin workspace

```
workspace_folder/          -- WORKSPACE
  src/                     -- SOURCE SPACE
    CMakeLists.txt         -- The 'toplevel' Cmake file
    package_1/
      CMakeLists.txt
      package.xml
      ...
    package_n/
      CMakeLists.txt
      package.xml
      ...
  devel/                   -- DEVELOPMENT SPACE
  build/                   -- BUILD SPACE
```

catkin_make

- `catkin_make` is a convenience tool for building code in a catkin workspace

- Execute `catkin_make` in the root of your catkin workspace

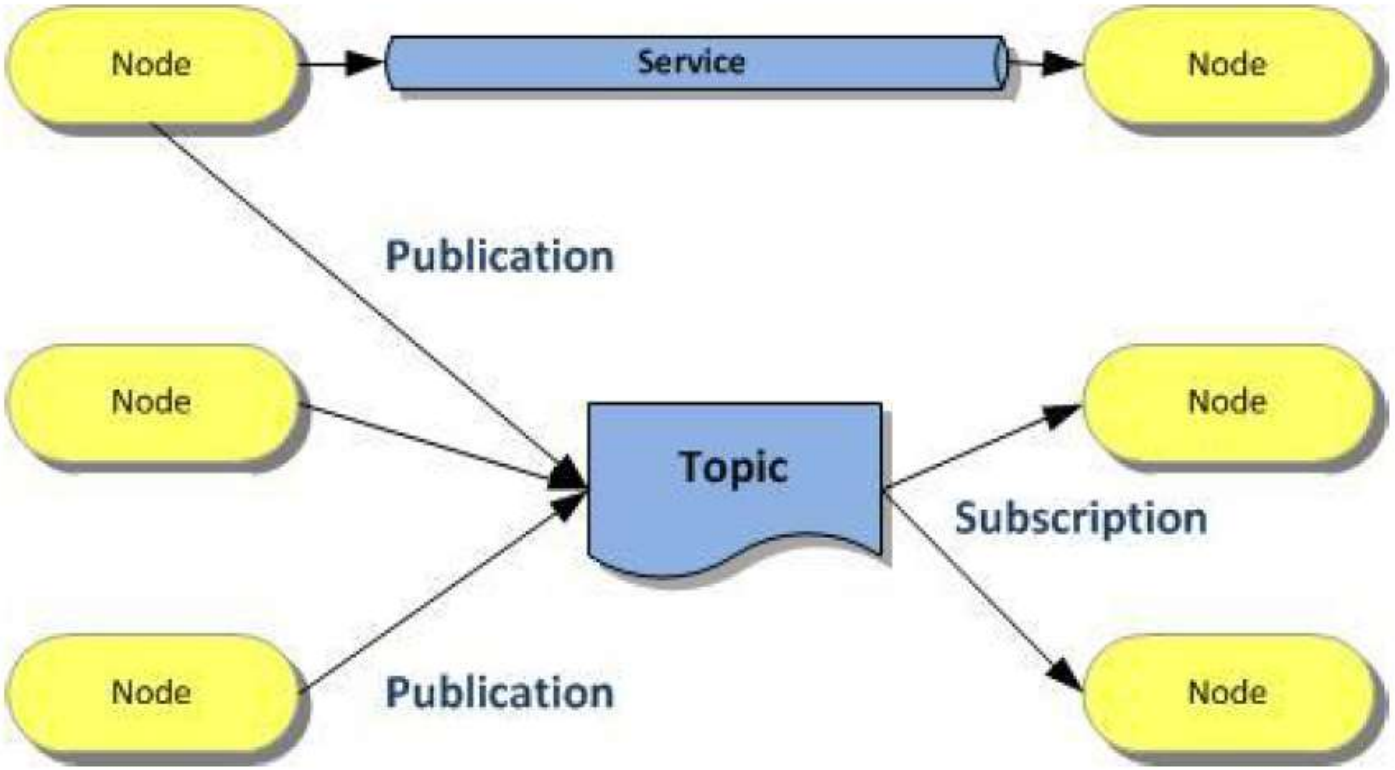
- Running the command is equivalent to:

```
$ mkdir build
$ cd build
$ cmake ../src -DCMAKE_INSTALL_PREFIX=../install
-DCATKIN_DEVEL_PREFIX=../devel
$ make
```

ROS definitions

- **Node**: process
- **Message**: Type of a data structure used to communicate between processes
- **Topic**: stream of message instance of the same type used to communicate the evolution of a quantity
  e.g. a CameraNode will publish a stream of images. Each image is of type ImageMessage (a matrix of pixels)
- **Publishing**: the action taken by a node when it wants to broadcast a message
- **Subscribing**: requesting messages of a certain topic

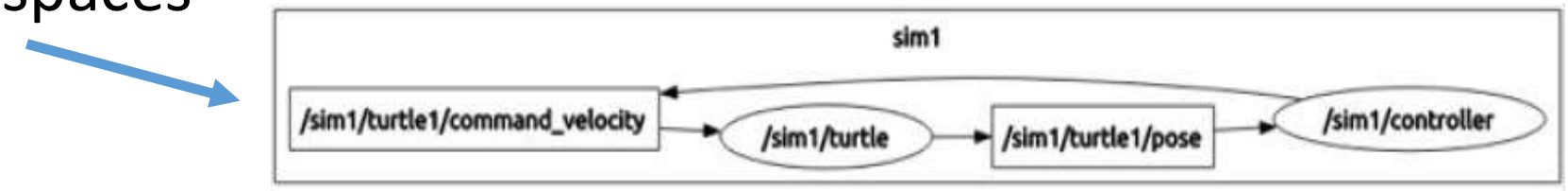
ROS definitions



http://wiki.ros.org/ROS/Concepts

Topics and Messages

- Communication in ROS exploits *messages*
- Messages are organized in *topics*
- A node that wants to share information will *publish* messages on a topic(s)
- A node that wants to receive information will *subscribe* to the topic(s)
- ROS master takes care of ensuring that publishers and subscribers can find each other
- Use of namespaces



Inspecting topics

- Listing active topics:
  `rostopic list`
- Seeing all messages published on topic:
  `rostopic echo topic-name`
- Checking publishing rate:
  `rostopic hz topic-name`
- Inspecting a topic (message type, subscribers, etc…):
  `rostopic info topic-name`
- Publishing messages trough terminal line:
  `rostopic pub -r rate-in-hz topic-name message-type message-content`

http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics/

ROS master

- One of the goals of ROS is to enable the use of small and mostly independent programs (nodes), all running at the same time
- For doing this, communication is needed
- The ROS master provides naming and registration services to enable the nodes to locate each other and, therefore, to communicate

roscore

- Start the ROS master on a terminal with
  `roscore`
- It provides bookkeeping of which nodes are active, which topics are requested by whom, and other facilities
- Nodes need to communicate with the master only at the beginning to know their peers, and which topics are offered
- After that the communication among nodes is peer-to-peer

Nodes

- Running instance of a ROS program
- Starting a node:
  `rosrun package-name executable-name`
- Listing running nodes:
  `rosnode list`
  - `/rosout` is a node started by roscore (similar to std output)
  - `/` indicates the global namespace

Nodes

- Inspecting a node (list of topics published and subscribed, services, PID and summary of connections with other nodes):
  `rosnode info` *`node-name`*
- Kill a node (also CTRL+C, but unregistration may not happen)
  `rosnode kill` *`node-name`*
- Remove dead nodes:
  `rosnode cleanup`

Anatomy of a ROS Node

```
ros::Publisher pub;

// function called whenever a message is received
void my_callback(MsgType* m) {
    OtherMessageType m2;
    … // do something with m and valorize m2
    pub.publish(m2);
}

int main(int argc, char** argv){
    // initializes the ros ecosystem
    ros::init(argc, argv);

    // object to access the namespace facilities
    ros::NodeHandle n;

    // tell the world that you will provide a topic named "published_topic"
    pub.advertise<OtherMessageType>("published_topic");

    // tell the world that you will provide a topic named "published_topic"
    Subscriber s =n.subscribe<MessageType*>("my_topic",my_callback);
    ros::spin();
}
```

Parameters

- Setting values to nodes
- Actively queried by the nodes, they are most suitable for configuration information that will not change (much) over time

```
double max_tv;
private_nh.param("max_tv", max_tv, 2.0);
double max_rv;
private_nh.param("max_rv", max_rv, 2.0);
planner->setMaxVelocity(max_tv, max_rv);
```

http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams

roslaunch

The ROS master and the nodes can be activated all at once, using a launch file

See details at:
http://wiki.ros.org/roslaunch/XML

```
<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>
```

```
roslaunch package-name launch-file-name
```

rosbag

- A bag is a serialized message data in a file

- rosbag for recording or playing data
  `rosbag record -a` Record all the topics
  `rosbag info` *`bag-name`* Info on the recorded bag
  `rospag play --pause` *`bag-name`* Play the recorded bag, starting paused
  `rospag play -r` *`#number bag-name`* Play the recorded bag at rate #number

Creating messages

- Messages in ROS are .msg files stored in the corresponding package folder, within the msg dir
- Supported field types are:
  - int8, int16, int32, int64 (plus uint*)
  - float32, float64
  - string
  - time, duration
  - other msg files
  - variable length array [] and fixed length array [C]
  - Header: timestamp and coordinate frame information

Example: creating messages

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

Exercise

Create a message Num.msg with a field `num` of type `int64`

Exercise

- Follow the ROS beginner tutorials:
  – Build and run the “Simple Publisher and Subscriber”
  – Build and run the “Simple Service and Client”

- Modify the talker node and the listener node
  1. Publish the message Num (created earlier) on the topic oddNums:
     - the message Num should be sent if the variable count is odd
     - Num should contain the value of count
  2. Additionally subscribe to topic oddNums
  3. Create a callback function oddNumsCallback to print the content of the received message

Exercise

Create a package with a client and a server:

- The server should take in input a service with an integer and an array of strings and return an array of strings, that are substrings of the corresponding input strings
- The client should input a sequence of strings and request a service

References and Credits

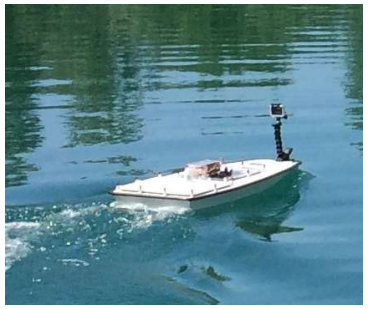
- Introduction to ROS
  Roberto Capobianco, Daniele Nardi

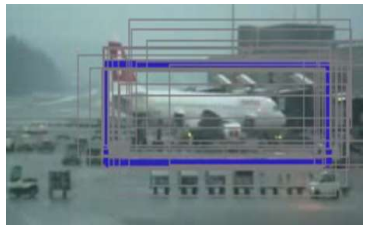
- Robot Programming - Robotic Middlewares
  Giorgio Grisetti, Cristiano Gennari

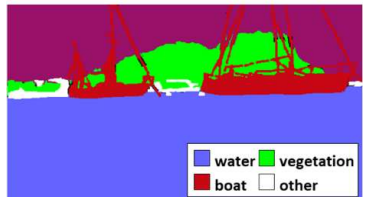
UNIVERSITÀ di VERONA | Dipartimento di INFORMATICA

Laurea magistrale in Ingegneria e scienze informatiche





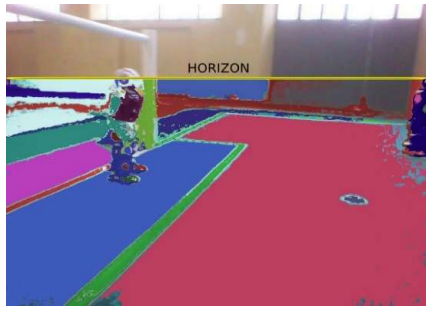




Introduzione a ROS

Corso di Robotica
Parte di Laboratorio

Docente:
Domenico Daniele Bloisi



Novembre 2017