What is ROS?

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

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


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

Suggested OS: Ubuntu 14.04

Suggested release: Indigo
ROS Features

«Why ROS instead of OROCOS, Player, Robotics Studio, (...)?»

• 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

- Standard Message Definitions for Robots
- Robot Geometry Library
- Robot Description Language
- Preemptable Remote Procedure Calls
- Diagnostics
- Pose Estimation
- Localization
- Mapping
- Navigation

30/09/2014
ROS Tools

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

ROS provides seamless integration of famous libraries and popular open-source projects.
Installation

3 possibilities for installing ROS:

• Install ROS from source (not recommended):
  – http://wiki.ros.org/indigo/Installation/Source

• Install ROS from Debian packages:

• Install virtual machine (Ubuntu 14.04 + ROS):
  – https://drive.google.com/file/d/0B6Nvp-r2hOVvWE1BSIBPbGl3XzA/edit?usp=sharing
  – Virtualbox instructions: http://www.virtualbox.org/manual/ch01.html#ovf
  – Login: user indigo password indigo
Post Installation

You must initialize the rosdep system 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

- **Packages**: 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 (1)

<?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">Roberto Capobianco</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>BSD</license>

<!-- Url tags are optional, but multiple 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, multiple are allowed, one per tag. Authors do not have to be maintainers, but could be -->

<author email="my@mail.com">Roberto Capobianco</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. -->
package.xml (3)

<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: -->
</export>

<!-- Other tools can request additional information be placed here -->
</export>
</package>
Catkin vs Rosbuild

ROS build systems: catkin, rosbuild (old, do not use rosbuild if not needed).

«So, why are you talking about rosbuild?»
Some packages are still developed for rosbuild.

Main differences between catkin and rosbuild:
http://wiki.ros.org/catkin_or_rosbuild
Catkin Workspace Configuration

$ source /opt/ros/indigo/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 (1)

workspace_folder/

src/

   CMakeLists.txt

package_1/

   CMakeLists.txt

   package.xml

   ...

package_n/

   CMakeLists.txt

   package.xml

   ...

-- WORKSPACE

-- SOURCE SPACE

-- The 'toplevel' Cmake file
Catkin Workspace (2)

**build/**  
CATKIN_IGNORE  
-- BUILD SPACE  

**devel/**  
setup.bash \  
setup.zsh |-- Environment setup files  
setup.sh /  
env.bash  

**etc/**  
-- Generated configuration files  

**include/**  
-- Generated header files  

30/09/2014
Catkin Workspace (3)

lib/  -- Generated libraries and other artifacts (bin included)

share/  -- Generated architecture independent artifacts

...  -- INSTALL SPACE (set by CMAKE_INSTALL_PREFIX)

install/
bin/
etc/
include/lib/
share/  
env.bash

Introduction to ROS  30/09/2014
Catkin Workspace (4)

```
setup.bash
setup.sh
...
```
Rosbuild Workspace Configuration

Catkin and rosbuilt workspaces can coexist (if needed).
Add to ~/.bashrc also:

```bash
export
ROS_PACKAGE_PATH=~/path/to/your/rosbuilt/workspace/:$ROS_PACKAGE_PATH
```

Additional info:

- [http://wiki.ros.org/rosbuild](http://wiki.ros.org/rosbuild)
- [http://wiki.ros.org/rosmake](http://wiki.ros.org/rosmake)
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
  ```
Building Specific Packages

• If you want to build specific packages in the workspace, invoke (always in the root of the workspace):

$ catkin_make -DCATKIN_WHITELIST_PACKAGES="package1;package2"

For reverting back:
$ catkin_make -DCATKIN_WHITELIST_PACKAGES=""

• If you want to build a single package, invoke:

$ catkin_make --pkg my_package
Installing Packages

• You can pass to `catkin_make` arguments that you would normally pass to `make` and `cmake`. For example, you can invoke the install target:

```
$ catkin_make install
```

Which is equivalent to:
```
$ cd ~/catkin_ws/build
# If cmake hasn't already been called
$ cmake ..:/src -DCMAKE_INSTALL_PREFIX=../install -D_CATKIN_DEVEL_PREFIX=../devel
$ make
$ make install
```
Listing and Locating Packages

rospack allows to get information about packages:

• Listing all ROS packages:
  rospack list

• Find the directory of a single package:
  rospack find package-name

• When you don’t know (remember) the complete name of the package, you can simply use tab completion for package names.

Hands on: find the roscpp package
Inspecting Packages

- To view the files in a package directory:
  `rosls package-name`
- To go to a package directory:
  `roscd package-name` (also without package name)
- These tools will *only* find ROS packages that are within the directories listed in your `ROS_PACKAGE_PATH`. To see what is in this variable, type: `echo $ROS_PACKAGE_PATH`

**Hands on:** list all the images in the *turtlesim* package; try `roscd` without a package name
Creating Packages

• A package must contain:
  – A catkin compliant package.xml
  – A CMakeLists.txt which uses catkin

• No nested packages are allowed (one per folder)

• You can create metapackages

```bash
# You should have created this!
$ cd ~/catkin_ws/src
# catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
$ catkin_create_pkg my_first_pkg std_msgs rospy roscpp
```
Checking Dependencies

• Packages can have direct or indirect dependencies

• Direct dependencies can be checked with:
  
  rospack depends1 package-name

• A full list of dependencies is available with:
  
  rospack depends package-name

• System dependencies for a package package-name can be solved:
  
  rosdep install package-name

Hands on: list all the dependencies in my_first_pkg
CMakeLists.txt (1)

- CMake version 2.8.3 or higher
- Your CMakeLists.txt file **MUST** follow this format otherwise your packages will not build correctly:
  - Required CMake Version (`cmake_minimum_required`)
  - Package Name (`project()`)
  - Find other CMake/Catkin packages needed for build (`find_package()`)
  - Message/Service/Action Generators (`add_message_files()`, `add_service_files()`, `add_action_files()`)
  - Invoke message/service/action generation (`generate_messages()`)

30/09/2014 Introduction to ROS
CMakeLists.txt (2)

- Specify package build info export (catkin_package())
- Libraries/Executables to build (add_library() / add_executable() / target_link_libraries())
- Tests to build (catkin_add_gtest())
- Install rules (install())

• If you have self-defined messages / services / actions, remember:
  - You must follow the order presented here (in particular before the catkin_package() macro) in order for generate stuff correctly
  - Your catkin_package() macro must have a CATKIN_DEPENDS dependency on message_runtime
  - You must use find_package() for the package message_generation, either alone or as a component of catkin
CMakeLists.txt (3)

- Your package.xml file must contain a build dependency on `message_generation` and a runtime dependency on `message_runtime`. This is not necessary if the dependencies are pulled in transitively from other packages.

- If you have a package which builds messages and/or services as well as executables that use them, you need to create an explicit dependency on the automatically-generated message target so that they are built in the correct order, e.g.:

```
add_dependencies(some_target
  ${PROJECT_NAME}_generate_messages_cpp)
```
Metapackages (1)

• Useful for grouping multiple packages in a single logical package
• Conceptually similar to rosbuild stacks, but no strict hierarchy in directory structure
• Normal package with the following tag in the package.xml:

```xml
<export>
  <metapackage />
</export>
```
Metapackages (2)

- Required buildtool\_depends dependency on catkin
- Can only have run dependencies on packages of which they group
- Required CMakeLists.txt:

  ```cmake
  cmake_minimum_required(VERSION 2.8.3)
  project(<PACKAGE_NAME>)
  find_package(catkin REQUIRED)
  catkin_metapackage()
  ```
- Other packages should not depend on metapackages
The master (1)

(One of) The goal(s) 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. By providing naming and registration services, the ROS master, enables the nodes to locate each other and, therefore, to communicate.

- To execute it, launch this command: roscore
- The master MUST be always running while using ROS.
The master (2)

What the master **does:**

- Naming
- Registration
- Publisher and Subscriber tracking (both for services and messages)
- Parameter server

What the master **does not:**

- Nodes do not communicate through the master
Nodes (1)

- Running instance of a ROS program
- Starting a node:
  
  
  \texttt{rosrun package-name executable-name}

Hands on: run an instance of \texttt{turtlesim_node} and \texttt{turtle_teleop_key} (hint: you need 3 terminals); focus on the terminal with \texttt{turtle_teleop_key} and press the Up, Down, Left, Right keys to move the turtle.
Nodes (2)

- Listing running nodes:
  
  ```
  rosnodes list
  ```
  
  - `/rosout` is a node started by roscore (similar to std output)
  - `/` indicates the global namespace
  - Node names are not necessarily the same as the names of their executables. You can explicitly set the name of a node using `rosrun`:
    
    ```
    rosrn `package-name` `executable-name` __name__:=`node-name`
    ```

Hands on: list nodes in the previous exercise
**Nodes (3)**

- Inspecting a node (list of topics published and subscribed, services, PID and summary of connections with other nodes):
  
  ```
  rosnod info node-name
  ```

- Kill a node (also CTRL+C, but unregistration may not happen)
  
  ```
  rosnod kill node-name
  ```

- Remove dead nodes:
  
  ```
  rosnod cleanup
  ```
Topics and Messages

- Communication in ROS through *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
Viewing the Graph

- Graphically intuitive, easy to visualize the publish-subscribe relationships between nodes:

  ```
  rqt_graph
  ```

- `rqt_graph` itself appears as a node

- All nodes publish on the topic `/rosout` (not the node!) subscribed by the node `/rosout`

- Topics without a subscriber (or a publisher) are possible (not both)

Hands on: analyze the graph of the previous exercise
rqt_graph
Messages and Topics

• Listing active topics:
  rostopic list

• You can see messages published on a topic:
  rostopic echo topic-name

• Checking publishing rate and bandwidth consumed:
  rostopic hz topic-name
  rostopic bw topic-name

• Inspecting a topic (also message type)
  rostopic info topic-name
Messages and Message Type

- Inspecting a message type (structure of the message):
  
  `rosmsg show message-type-name`

- Data types of composite fields are message types in their own (useful for preventing code duplication)

- Message types can also contain arrays with fixed or variable length (show with square brackets)

**Hands on: check the structure of all the messages in the topics of the previous exercise**
rqt_plot

• Data published on topics can be time plotted
Publishing Messages from Terminal

• Useful for debugging
• Publish message from terminal:
  rostopic pub -r rate-in-hz topic-name message-type message-content
• The message content can be tabbed once the message type is chosen

Hands on: publish a velocity command at 1Hz rate to the /turtle1/cmd_vel topic and plot the position and velocity of the turtle
Services (1)

- Realize request/reply communications
- Defined as a structure composed by a pair of messages (one for the request and one for the reply):
  
  rostopic pub -r rate-in-hz topic-name message-type message-content

- A providing node or provider offers a service
- A client interested in a service sends a request and waits for a reply
Services (2)

- Display all services of a specific type: `rosservice find service-type`
- List of services: `rosservice list`
- Print information about a specific service: `rosservice info service-name`
- Display the node that provides a particular service: `rosservice node service-name`
- Display the type of a service: `rosservice type service-name`
- Call a service from the command line: `rosservice call service-name service-args`
- `rossrv` is similar to `rosmsg`
Creating Messages and Services (1)

- Messages (Services) in ROS are `.msg` (.srv) files stored in the corresponding package folder, within the `msg` (srv) dir.
- Supported field types for both 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
- srv files have two different message definitions, separated by ---
Creating Messages and Services (2)

Example of msg:

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

Example of srv:

int64 A
int64 B
---
int64 Sum

Hands on: create a message Num.msg with field num of type int64; create a service AddTwoInts.srv and build the package.
Parameters

- Hierarchy matching the namespaces
- `rosparam` for setting and reading parameters
  
  ```
  rosparam set param-name
  rosparam get param-name
  ```

- Parameters can also be listed or deleted
  
  ```
  rosparam list
  rosparam delete param-name
  ```

Hands on: explore and use services of the turtlesim node
roslaunch

- Launch file usually bring up a set of nodes (roscore is automatically launched by roslaunch)
- Uses XML files that describe the nodes that should be run, parameters that should be set, and other attributes

Hands on: create a launch file launching two turtlesim nodes
Bags and rosbag

- 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`

Hands on: record a bag while you are teleoperating the turtlesim, then kill every node; start again the turtlesim node and play the bag
Checking for Problems

- Useful when ROS is not behaving the way you expect:
  roswtf
- Broad variety of sanity checks (e.g., examination of environment variables, installed files, running nodes)
- Details at: [http://wiki.ros.org/roswtf](http://wiki.ros.org/roswtf)
Homework (1)

• 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
  – Publish the message *Num* (created earlier) on topic *oddNums*:
    • the message *Num* should be sent if the variable count is odd
    • *Num* should contain the value of count
  – Additionally subscribe to topic *oddNums*
  – Create a callback function *oddNumsCallback* to print the content of the received message
Homework (2)

• 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