Introduction to ROS

Slides adapted from: http://www.rsl.ethz.ch/education-students/lectures/ros.html

Credit to Markus Grotz, Joshua Smith and others on the EE 545 staff
ROS Terminology

> ROS versions are identified by name
  – first letter of name increments with each new version...

> In this class, we are using ROS Melodic (latest version is Noetic)

> Previous versions were Lunar and Kinetic
ROS 1 vs. ROS 2

> Currently ROS vs ROS2
> Major difference
  – Single base library for C++ and Python
  – No roscore
  – Services are now asynchronous
  – Quality of Service (QoS)
What is ROS?

ROS = Robot Operating System

- **Plumbing**
  - Process management
  - Inter-process communication
  - Device drivers

- **Tools**
  - Simulation
  - Visualization
  - Graphical user interface
  - Data logging

- **Capabilities**
  - Control
  - Planning
  - Perception
  - Mapping
  - Manipulation

- **Ecosystem**
  - Package organization
  - Software distribution
  - Documentation
  - Tutorials

[ros.org]
History of ROS

> Originally developed in 2007 at the Stanford Artificial Intelligence Laboratory
> Since 2013 managed by OSRF
> Today used by many robots, universities and companies
> De facto standard for robot programming
ROS Philosophy

> Peer to peer
  — Individual programs communicate over defined API (ROS messages, services, etc.).

> Distributed
  — Programs can be run on multiple computers and communicate over the network.

> Multi-lingual
  — ROS modules can be written in any language for which a client library exists (C++, Python, MATLAB, Java, etc.).

> Light-weight
  — Stand-alone libraries are wrapped around with a thin ROS layer.

> Free and open-source
  — Most ROS software is open-source and free to use.
ROS Nodes

- Single-purpose, executable program
- Individually compiled, executed, and managed
- Organized in packages

Run a node with
- `rosrun package_name node_name`

See active node list
- `rosnodes list`

Retrieve information about a node with
- `rosnodes info node_name`
ROS Master

> Manages the communication between nodes (processes)
> Every node registers at startup with the master
> No longer required in ROS2
**ROS Topics**

- Nodes communicate over topics
  - Nodes can publish or subscribe to a topic
  - Typically, 1 publisher and n subscribers
- Topic is a name for a stream of messages

List active topics with
- `rostopic list`

Subscribe and print the contents of a topic with
- `rostopic echo /topic`

Show information about a topic with
- `rostopic info /topic`
rostopic: info + echo

```bash
robotics @ mushr-vm → ~ rostopic info /car/car_pose
Type: geometry_msgs/PoseStamped

Publishers:
* /mushr_sim (http://mushr-vm:38347/)

Subscribers: None
```

```bash
robotics @ mushr-vm → ~ rostopic echo -n 1 /car/car_pose
header:
  seq: 1
  stamp:
    secs: 1696316266
    nsecs: 936288118
    frame_id: "map"
pose:
  position:
    x: -0.0001300085021457966
    y: 0.00010512683808957599
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.00010826673162798999
    w: 0.9999999941391574
```

---

```
robotics @ mushr-vm → ~
```
Note hierarchical naming of topics

Here we went further down the tree to display just orientation, not all the other parts of the car_pose topic.

Many things in ROS use hierarchical naming (Naming is similar to paths in a filesystem)
ROS Messages

- Data structure defining the type of a topic
- Comprised of a nested structure of integers, floats, booleans, strings etc. and arrays of objects
- Defined in *.msg files

See the type of a topic
➔ rostopic type /topic

Publish a message to a topic
➔ rostopic pub /topic type data
car_pose

> Get the message type of a topic

```
robotics @ mushr-vm ➤ rostopic type /car/car_pose
geometry_msgs/PoseStamped
robotics @ mushr-vm ➤
```
# ROS Messages

## Pose Stamped Example

**geometry_msgs/Point.msg**
- float64 x
- float64 y
- float64 z

**sensor_msgs/Image.msg**
- std_msgs/Header header
  - uint32 seq
  - time stamp
  - string frame_id
- uint32 height
- uint32 width
- string encoding
- uint8 is_bigendian
- uint32 step
- uint8[] data

**geometry_msgs/PoseStamped.msg**
- std_msgs/Header header
  - uint32 seq
  - time stamp
  - string frame_id
- geometry_msgs/Pose pose
  - geometry_msgs/Point position
    - float64 x
    - float64 y
    - float64 z
  - geometry_msgs/Quaternion orientation
    - float64 x
    - float64 y
    - float64 z
    - float64 w
Some ROS Messages we will use

- geometry_msgs/PoseStamped
- sensor_msgs/LaserScan
- ackermann_msgs/AckermannDriveStamped
- geometry_msgs/Quaternion
ROS Workspace Environment

> Defines context for the current workspace
> Default workspace loaded with
  ➔ `source /opt/ros/noetic/setup.bash`

Overlay your catkin workspace with
  ➔ `cd ~/catkin_ws`
  ➔ `source devel/setup.bash`

Check your workspace with
  ➔ `echo $ROS_PACKAGE_PATH`

See setup with
  ➔ `cat ~/.zshrc`
The catkin build system

- catkin is the ROS build system to generate executables, libraries, and interfaces
- We suggest to use the Catkin Command Line Tools

Navigate to your catkin workspace with
  ➔ \texttt{cd ~/catkin\_ws}

Build a package with
  ➔ \texttt{catkin\_make}

Whenever you build a new package, update your environment
  ➔ \texttt{source\_devel/setup.bash}
The catkin build system

The catkin workspace contains the following spaces

**Work here**
- src
  - The source space contains the source code. This is where you can clone, create, and edit source code for the packages you want to build.

**Don’t touch**
- build
  - The build space is where CMake is invoked to build the packages in the source space. Cache information and other intermediate files are kept here.

**Don’t touch**
- devel
  - The development (devel) space is where built targets are placed (prior to being installed).

If necessary, clean the entire build and devel space with

→ catkin clean
ROS Launch

- launch is a tool for launching multiple nodes (as well as setting parameters)
- Are written in XML as *.launch files
- If not yet running, launch automatically starts a roscore

Browse to the folder and start a launch file with:

➔ roslaunch file_name.launch

Start a launch file from a package with:

➔ roslaunch package_name file_name.launch
ROS Launch

File Structure

```
<launch>
  <node name="listener" pkg="roscpp_tutorials" type="listener" output="screen"/>
  <node name="talker" pkg="roscpp_tutorials" type="talker" output="screen"/>
</launch>
```

- **launch**: Root element of the launch file
- **node**: Each `<node>` tag specifies a node to be launched
- **name**: Name of the node (free to choose)
- **pkg**: Package containing the node
- **type**: Type of the node, there must be a corresponding executable with the same name
- **output**: Specifies where to output log messages (`screen`: console, `log`: log file)

---

Attention when copy & pasting code from the internet

Notice the syntax difference for self-closing tags: `<tag></tag>` and `<tag/>`
ROS Launch

Arguments

- Create re-usable launch files with `<arg>` tag, which works like a parameter (default optional):
  ```xml
  <arg name="arg_name" default="default_value"/>
  ```

- Use arguments in launch file with
  ```bash
  $(arg arg_name)
  ```

- When launching, arguments can be set with
  ```bash
  > roslaunch launch_file.launch arg_name=value
  ```

```xml
<?xml version="1.0"?>
<launch>
  <arg name="use_sim_time" default="true"/>
  <arg name="world" default="gazebo_ros_range"/>
  <arg name="debug" default="false"/>
  <arg name="physics" default="ode"/>

  <group if="$(arg use_sim_time)">
    <param name="/use_sim_time" value="true"/>
  </group>

  <include file="$(find gazebo_ros)
    /launch/empty_world.launch">
    <arg name="world_name" value="$(find gazebo_plugins)/
      test/test_worlds/$(arg world).world"/>
    <arg name="debug" value="$(arg debug)"/>
    <arg name="physics" value="$(arg physics)"/>
  </include>
</launch>
```
Including Other Launch Files

- Include other launch files with `<include>` tag to organize large projects
  `<include file="package_name"/>

- Find the system path to other packages with
  `$(find package_name)`

- Pass arguments to the included file
  `<arg name="arg_name" value="value"/>`

---

`range_world.launch` (simplified)

```xml
<?xml version="1.0"?>
<launch>
  <arg name="use_sim_time" default="true"/>
  <arg name="world" default="gazebo_ros_range"/>
  <arg name="debug" default="false"/>
  <arg name="physics" default="ode"/>

  <group if="$(arg use_sim_time)">
    <param name="/use_sim_time" value="true" />
  </group>

  <include file="$(find gazebo_ros)
    /launch/empty_world.launch">
    <arg name="world_name" value="$(find gazebo_plugins)/
      test/test_worlds/$(arg world).world"/>
    <arg name="debug" value="$(arg debug)"/>
    <arg name="physics" value="$(arg physics)"/>
  </include>
</launch>
```
RVIZ Simulator
Further References

**ROS Wiki**
> http://wiki.ros.org/

**Installation**
> http://wiki.ros.org/ROS/Installation

**Tutorials**
> http://wiki.ros.org/ROS/Tutorials

**Available packages**
> https://index.ros.org/packages/
Further References II

ROS Cheat Sheet
> https://www.clearpathrobotics.com/ros-robot-operating-system-cheat-sheet/
> https://kapeli.com/cheat_sheets/ROS.docset/Contents/Resources/Documents/index

ROS Best Practices
> https://github.com/leggedrobotics/ros_best_practices/wiki

ROS Package Template
> https://github.com/leggedrobotics/ros_best_practices/tree/master/ros_package_template