ROS architecture

Inteligencia Artificial en los Sistemas de Control Autónomo
Máster Universitario en Ingeniería Industrial

Departamento de Automática

Objectives

- Understand the ROS computational model
- Use the main ROS commands
- Handle the ROS file system

Bibliography

ROS tutorials (Link):

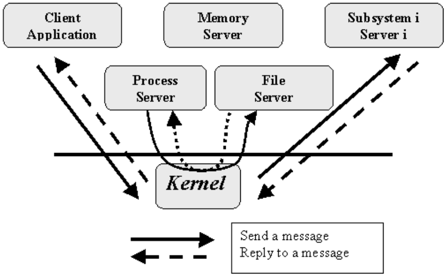
- Understanding ROS Nodes
- Understanding ROS Topics
- Understanding ROS Services and Parameters
- Navigating the ROS Filesystem

Table of Contents

Overview (I)

ROS follows the philosophy of a microkernel operating system

- Several independent processes
- The kernel handles messages (microkernel)
- Drivers are processes



Advantages

- Robustness
- Modularity
- Distributed

Disadvantages

- Complexity

Overview (II)

Key ROS concepts

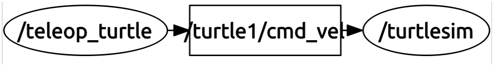
- Node: Like a process
- Topic: Like a message blackboard

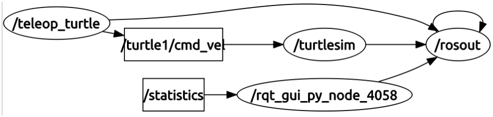
Key ROS commands

- `roscore`: Runs core nodes in ROS
- `rosrun`: Runs a node
- `roslaunch`: Runs several nodes

Practice

```
> roscore
> rosrun turtlesim turtlesim_node
> rosrun turtlesim turtle_teleop_key
> rosrun rqt_graph rqt_graph
```





Overview (III)

ROS defines a three-level architecture

- The community level
- The computation graph level
- The filesystem level

The community level

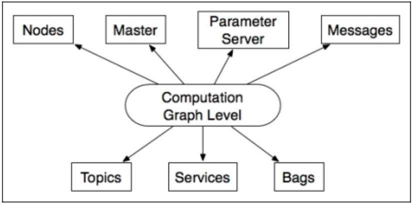
Resources to distribute software and share knowledge

- Distributions
- Repositories
- Wiki
- Forum

The computation graph level

ROS creates a network of processes (nodes) communicated by different means

- Nodes, Master, Parameter server, Messages, Topics, Services and Bags



The computation graph level

Nodes (I)

Node: Information processing unit in ROS

- One node, one specific function
  - One node controls a motor, another one the ultrasound sensor, etc
- Increased security and fault tolerance
- Unique name
- Nodes implemented in C++, Python or Matlab

The computation graph level

Nodes (II)

A handy command-line utility: `rosnode`

- `rosnode list`
- `rosnode info node`
- `rosnode kill node`

Other utilities

- `rosnode machine hostname`
- `rosnode ping node`
- `rosnode cleanup`

The computation graph level

Nodes (III)

Exercises

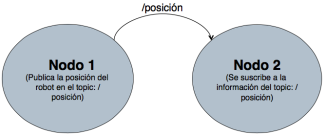
- Run the turtlebot simulation
  1. `> rosrun turtlesim turtlesim_node`
  2. `> rosrun turtlesim turtle_teleop_key`
- Identify the running nodes
- Get info about the node that runs the simulation
- Check connectivity with that node
- Kill the node

The computation graph level

Topics (I)

Topic: Communication buses used by nodes to transmit data

- Publisher/subscriber mechanism
- One-to-many communication
- Unique name
- Strongly typed



The computation graph level

Topics (II)

A handy command-line utility: `rostopic`

- `rostopic list`
- `rostopic echo topic`
- `rostopic info topic`
- `rostopic find message_type`

Two pretty useful commands

- `rostopic pub topic type args`
- `rostopic type topic`

The computation graph level

Topics (III)

Exercises

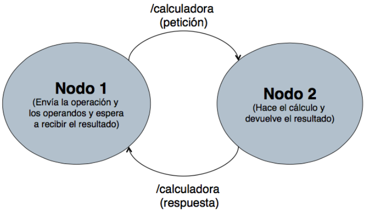
- Run the turtlebot simulation
  1. `> rosrun turtlesim turtlesim_node`
  2. `> rosrun turtlesim turtle_teleop_key`
- Identify the available topics
- Which topic publishes the turtle motion?
- Visualize that topic while you teloperate the turtle

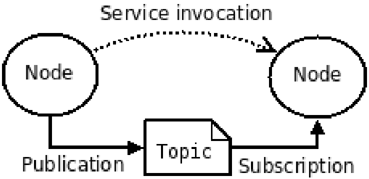
The computation graph level

Services (I)

Service: RPC-like communication

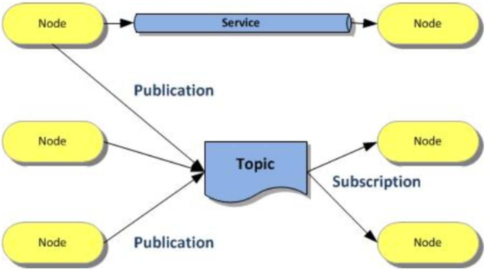
- Strongly typed
- One-to-one communication





The computation graph level

Services (II)



A handy command-line utility: `rosservice`

- `rosservice list`
- `rosservice info service`
- `rosservice call service`
- `rosservice type service`

The computation graph level

Services (III)

Exercises

- Run the turtlebot simulation
  1. `> rosrun turtlesim turtlesim_node`
  2. `> rosrun turtlesim turtle_teleop_key`
- Identify the available services
- Get information about the `spawn` service
- Find out the parameter type used by the service `reset`
- Call `spawn` with correct arguments
- Idenfity again the available services

The computation graph level

Messages (I)

Message: Data structure used by nodes to communicate

- `rosmsg list`
- `rosmsg show`
- `rosmsg package package`
- `rosmsg packages`

Twist

```
Vector3  linear
Vector3  angular
```

Vector3

```
float64 x
float64 y
float64 z
```

The computation graph level

Messages (II)

	Type	Keyword
Data types	Integer	int8, int16, int32, int64 (plus uint*)
	Float	float32, float64
	String	string
	Time	time, duration
	Struct	other msg files
	Array	variable-length array[] and fixed-length array[C]

The computation graph level

Others (I)

- **Bags**: File contaning messages, topics, services and others. Usefull for debugging
- **Master**: Naming and registration services. Run by `roscore`. Provides the parameter server
- **Parameter server**: Dictionary that stores shared parameters, implemented with XML-RPC
  - `rosparam list`
  - `rosparam get parameter`
  - `rosparam set parameter parameter`

The computation graph level

Others (II)

Exercises

- Run the turtlebot simulation
  1. `> rosrun turtlesim turtlesim_node`
  2. `> rosrun turtlesim turtle_teleop_key`
- Identify the available messages
- Extract the message format used to move the turtle
- Move the turtle using `rostopic`
- Identify the available parameters
- Get ROS version and distro name by using parameters
- Change the background color of the turtle simulation

The computation graph level

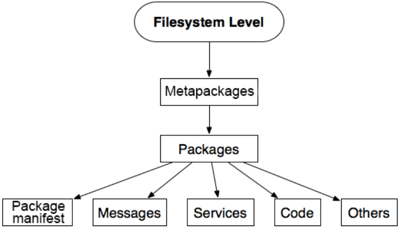
Practice

- Run the Hector UAV simulation
  1. `> roslaunch hector_quadrotor_demo outdoor_flight_gazebo.launch`
  2. `> roslaunch hector_quadrotor_teleop xbox_controller.launch`
- Identify the node that publishes the UAV motion
- Move the UAV using `rostopic`
- Visualize in real-time the laser scan messages
- Explore and understand the laser scan message format

The filesystem level

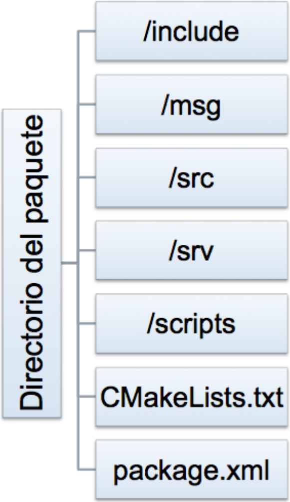
ROS resources stored on disk

- **Packages**: Main unit of organization
- **Metapackages**:Collection of packages
- **Package Manifests**: Metadata about a package (`package.xml`)
- **Repositories**: Collection of packages sharing a common VCS system
- **Message types**: Message descriptions
- **Service types**: Service descriptions



The filesystem level

Packages (I)



A package contains

- One or more nodes
- Messages description
- Services description
- ROS libraries
- Config files

Exercises

1. Go to `/opt/ros/indigo/share/`
2. List the ROS packages installed
3. Browse the package `turtlesim`

The filesystem level

Packages (II)

Several tools to manage ROS package

- `rospack list`
- `rospack find package`

To move

- `roscd`
- `roscd package`
- Try `roscd log`

To list

- `rosls`
- `rosls package`

Package path *must* be contained in `$ROS_PACKAGE_PATH`

Hint: Tab completion

The filesystem level

Important ROS packages

Package	Description
`roscpp`	C++ client library
`rospy`	Python client library
`std_msgs`	Standard messages
`geometry_msgs`	Geometric messages
`TF`	Coordinate systems mapping
`gmapping`	SLAM based on laser sensors
`amcl`	2D Monte Carlo localization
`stdr_simulator`	STDR simulator
`stage_ros`	Stage simulator
`gazebo_ros_pkgs`	Interface for Gazebo

The filesystem level

Messages types (I)

Messages types define data structures

- Stored in a `.msg` file
- Folder `/msg`
- Automatic code generation
- Well documented in the Web

geometry_msgs/Twist.msg

```
# This expresses velocity in free space broken
# into its linear and angular parts.
Vector3  linear
Vector3  angular
```

The filesystem level

Messages types (II)

geometry_msgs/Vector3.msg

```
float64 x
float64 y
float64 z
```

Exercise

1. Move to the `turtlesim` package folder
2. Identify the messages defined by the package `turtlesim`
3. Visualize the message file `Color.msg`

The filesystem level

Service types (I)

Service types define the request and response structure

- Stored in a `.srv` file
- Folder `/srv`
- Automatic code generation
- Well documented in the Web

turtlesim/srv/Spawn.srv

```
float32 x
float32 y
float32 theta
string name # Optional.  A unique name will be
            # created and returned if this is empty
---
```

The filesystem level

Service types (II)

Exercise

1. Move to the `turtlesim` package folder
2. Identify the services defined in the package
3. Visualize their format

The filesystem level

Others

Package Manifests: Metadata about a package

- Package name, version, dependences, etc
- Stored in the `package.xml` file
- XML format

Metapackages: Package that contains other packages

- It only installs one file: `package.xml`
- Uses to contain specialized features

Repositories: Collection of packages sharing a VCS system

Exercises

1. Go to the `turtlesim` package root folder
2. Read its package manifiest file
3. Which dependences does `turtlesim` have?

Node execution

Single node (rosrun)

rosnode: Executes a single node

```
rosrun <package> <node> [parameters]
```

Example: `rosrun my_package my_node _my_param:=value`

Warning: The node must be in `$ROS_PACKAGE_PATH`! (ROS init scripts)

Node execution

Several nodes (roslaunch) (I)

Usually, any ROS application is composed of several nodes

- Executing each node is unefficient
- Automate node execution

Features

- Run one or several nodes, group nodes, set up parameters, define environment variables, remap topics, respawn nodes

(More info)

roslaunch: Node execution control

```
roslaunch <package_name> <file.launch>
```

Node execution

Several nodes (roslaunch) (II)

roslaunch uses an XML file

- Usually stored in the launch folder

launch/minimal.launch (rospy_tutorials package)

```
<launch>
   <node name="nodeName" pkg="package" type="executable"/>
</launch>
```

Launch files might be quite complex

launch/server_no_map.launch (stdr_launchers package)

```
<launch>
  <include file="$(find stdr_robot)/launch/robot_manager.launch" />
  <node type="stdr_server_node" pkg="stdr_server" name="stdr_server"
       output="screen"/>
  <node pkg="tf" type="static_transform_publisher" name="world2map" args=
       "0 0 0 0 0 0  world map 100" />
</launch>
```

Node execution

Several nodes (roslaunch) (III)

Minimal launch file

```
<launch>
  <!-- local machine already has a definition by default.
       This tag overrides the default definition with
       specific ROS_ROOT and ROS_PACKAGE_PATH values -->
  <machine name="local_alt" address="localhost" default="true" ros-root="/u/user/ros/ros/" ros-package
        -path="/u/user/ros/ros-pkg" />
  <!-- a basic listener node -->
  <node name="listener-1" pkg="rospy_tutorials" type="listener" />
  <!-- pass args to the listener node -->
  <node name="listener-2" pkg="rospy_tutorials" type="listener" args="-foo arg2" />
  <!-- a respawn-able listener node -->
  <node name="listener-3" pkg="rospy_tutorials" type="listener" respawn="true" />
  <!-- start listener node in the 'wg1' namespace -->
  <node ns="wg1" name="listener-wg1" pkg="rospy_tutorials" type="listener" respawn="true" />
  <!-- start a group of nodes in the 'wg2' namespace -->
  <group ns="wg2">
    <!-- remap applies to all future statements in this scope. -->
    <remap from="chatter" to="hello"/>
    <node pkg="rospy_tutorials" type="listener" name="listener" args="--test" respawn="true" />
    <node pkg="rospy_tutorials" type="talker" name="talker">
      <!-- set a private parameter for the node -->
      <param name="talker_1_param" value="a value" />
      <!-- nodes can have their own remap args -->
      <remap from="chatter" to="hello-1"/>
      <!-- you can set environment variables for a node -->
      <env name="ENV_EXAMPLE" value="some value" />
    </node>
  </group>
</launch>
```

Node execution

Several nodes (roslaunch) (IV)

Exercise

1. Move to the `stdr_launchers` package folder
2. Follow this tutorial: `http://wiki.ros.org/stdr_simulator/Tutorials/Running%20STDR%20Simulator`
3. For each execution of `roslaunch`, open and read the launch file