

# **Introducción a ROS (Robot Operating System)**

Técnicas de los Sistemas Inteligentes  
Curso 2017-2018

# El Problema

## Ausencia de estándares en Robótica



# ¿Qué es ROS?

- Un framework flexible para escribir software de robot.
- Una colección de herramientas, bibliotecas y convenciones que tienen como objetivo simplificar la tarea de crear un comportamiento robótico complejo y robusto.

<http://www.ros.org>

# Motivación y Objetivos de ROS

- Crear un software de robot verdaderamente robusto y polivalente es difícil. Desde la perspectiva del robot, los problemas que parecen triviales para los seres humanos a menudo varían enormemente. Tratar con estas variaciones es tan difícil que ningún individuo, laboratorio o institución puede esperar hacerlo por sí solo.
- **Objetivo** → fomentar la **reutilización de código** y el desarrollo colaborativo de software de robótica.
- Originalmente desarrollado en 2007 en el Stanford Artificial Intelligence Laboratory (<https://ai.stanford.edu/>). Su desarrollo continúa en "Willow Garage" (<http://www.willowgarage.com/>).

# Motivación y Objetivos de ROS

- Otros objetivos de ROS para hacer efectivo este objetivo principal:
  - Modularidad y P2P: los procesos se ejecutan (como hebras) de forma independiente conectados siguiendo una tipología peer-to-peer
  - Ligero: desarrollo sobre librerías «standalone» mínimamente dependientes de ROS
  - Independiente del lenguaje: está implementado en C++ (roscpp) y Python (rospy), entre otros.
  - Escalable
  - Free & open source: código fuente públicamente disponible y herramientas bajo distintas licencias (abiertas y cerradas)

# Características de ROS

- Proporciona los servicios esperables de un sistema operativo:
  - Abstracción del hardware y de comunicaciones por red
  - Control de dispositivos a bajo nivel
  - Paso de mensajes entre procesos
  - Funcionalidades comunes (de robots) pre-implementadas
  - Gestión de paquetes
- Además ...
  - Extendido entre una enorme comunidad internacional
  - Muy bien documentado
  - Actualmente solo funciona en sistemas basados en UNIX

<http://wiki.ros.org/ROS/Introduction>

# Robots que usan ROS

<http://wiki.ros.org/Robots>



[Fraunhofer IPA Care-O-bot](#)



[Videre Erratic](#)



[TurtleBot](#)



[Aldebaran Nao](#)



[Lego NXT](#)



[Shadow Hand](#)



[Willow Garage PR2](#)



[iRobot Roomba](#)



[Robotnik Guardian](#)



[Merlin miabotPro](#)



[AscTec Quadrotor](#)



[CoroWare Corobot](#)



[Clearpath Robotics Husky](#)



[Clearpath Robotics Kingfisher](#)



[Festo Didactic Robotino](#)

Introducción a ROS

# CONCEPTOS BÁSICOS DE ROS

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

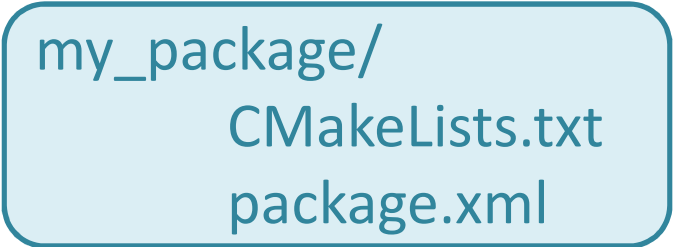


# ROS Filesystem

- **Packages:** unidad de organización del código de ROS. Cada paquete puede contener ejecutables, librerías, scripts...
- **Manifests** (package.xml): un *manifest* es la descripción de un paquete. Sirve para definir dependencias entre paquetes y meta-información sobre el paquete (versión, licencia, etc)

# ROS Filesystem

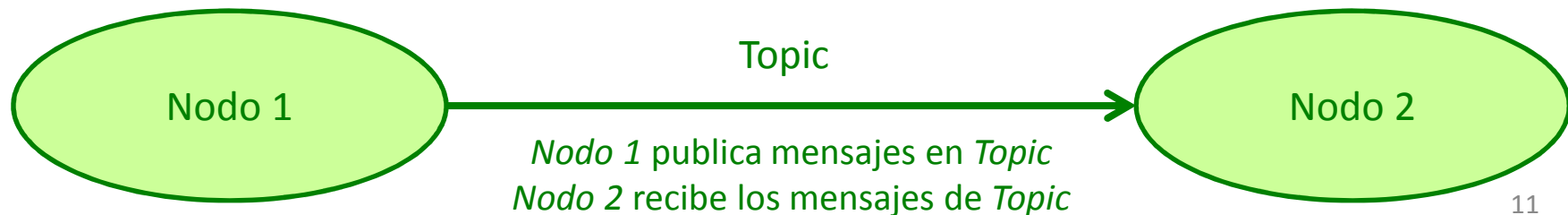
- La herramienta de manejo de paquetes se llama **catkin**
- Un paquete catkin debe:
  - Tener un archivo package.xml con metainformación sobre el paquete
  - Incluir un archivo CMakeLists.txt
  - Cada paquete debe tener su propia carpeta/directorio
- Un paquete puede contener: uno o varios nodos, librerías independientes de ROS, datasets, ficheros de configuración, software de terceros ...
- Estructura del paquete más sencillo posible:

A light blue rounded rectangle containing the text 'my\_package/' followed by 'CMakeLists.txt' and 'package.xml' on separate lines, representing the file structure of a ROS package.

my\_package/  
CMakeLists.txt  
package.xml

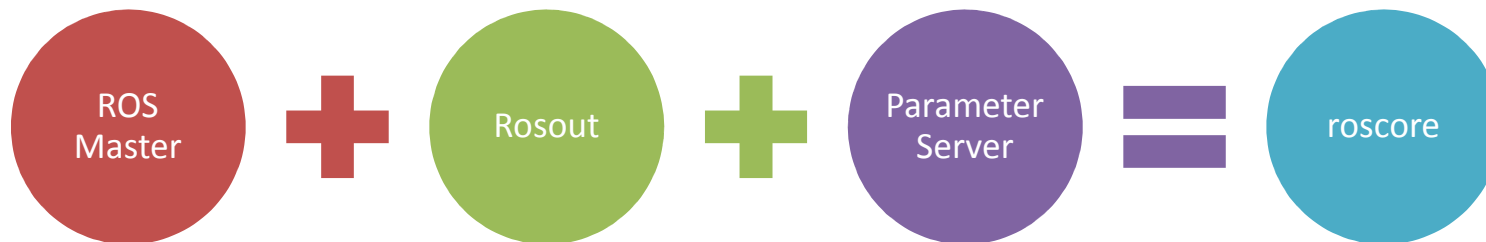
# Estructura de ROS

- **Nodes:** un nodo es un ejecutable que usa ROS para comunicarse con otros nodos.
- **Topics:** los nodos pueden publicar mensajes en un topic o subscribirse a un topic para recibir los mensajes publicados en éste.
- **Messages:** tipo de dato de ROS que se utiliza cuando se subscribe a o publica en un topic.



# Estructura de ROS

- **Master:** es el servicio de nombres de ROS. Ayuda a los nodos a localizarse entre ellos.
- **rosout:** equivalente de stdout/stderr en ROS.
- **roscore:** Master + rosout + parameter server (lo veremos más adelante)



# Estructura de ROS: Nodes

Un **nodo** no es mucho más que un archivo ejecutable dentro de un paquete de ROS.

- **roscore** es lo primero que hay que ejecutar cuando usemos ROS:

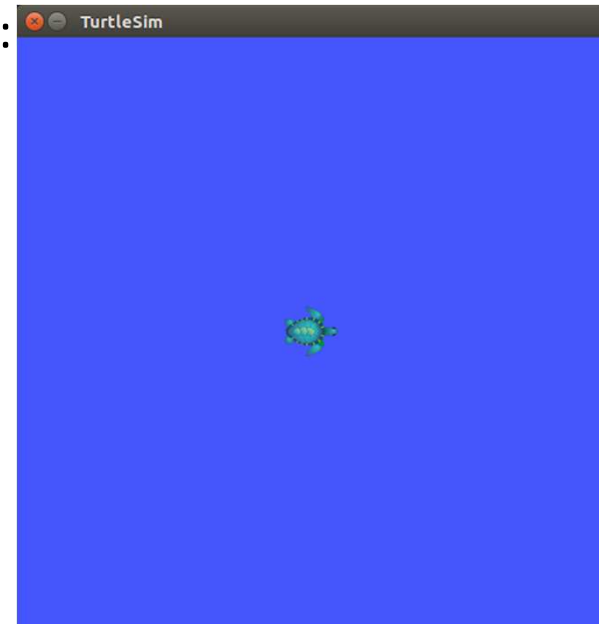
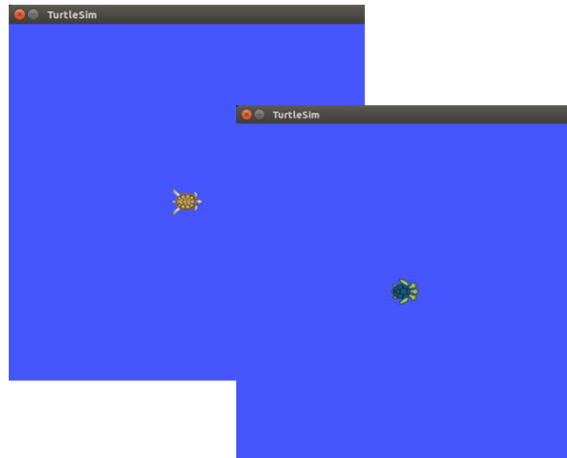
```
$ roscore
```

- Para ejecutar un nodo dentro de un paquete se usa **roslaunch**:

```
roslaunch [package_name] [node_name]
```

- Abrimos una segunda terminal y ejecutamos:

```
$ roslaunch turtlesim turtlesim_node
```



# Estructura de ROS: Topics

- Necesitamos dirigir a la tortuga para que se mueva. Para comunicarnos con ella, abrimos una nueva terminal y ejecutamos:

```
$ rosrun turtlesim turtle_teleop_key
```

Ahora podemos usar las teclas del teclado para que la tortuga se mueva.



- Los nodos *turtlesim\_node* y *turtle\_teleop\_key* se están comunicando a través de un **Topic**.

# Estructura de ROS: Topics

- *turtle\_teleop\_key* publica las pulsaciones de teclado en un **Topic** mientras *turtlesim\_node* está suscrito a dicho topic para recibir las pulsaciones de teclado
- Podemos usar el comando **rqt\_graph** para ver un gráfico dinámico de lo que está ocurriendo en el sistema. En una nueva terminal:

```
$ sudo apt-get install ros-<distro>-rqt  
$ sudo apt-get install ros-<distro>-rqt-common-plugins  
$ rosrun rqt_graph rqt_graph
```



# Estructura de ROS: Messages

- La comunicación a través de los **Topics** se produce a través del envío de **Mensajes** entre **Nodos**.
- El nodo Publisher y el Subscriber deben enviar y recibir el mismo tipo de mensajes.
- El **tipo de un Topic** se define según el **tipo de Mensaje** publicado en él.
- Para saber el tipo de mensajes que se publica en un determinado Topic utilizamos **rostopic type**:

```
rostopic type [topic]
```

- En una nueva terminal:

```
$ rostopic type /turtle1/cmd_vel
```

- Para comprobar los detalles de un tipo de Mensaje usamos rosmmsg:

```
$ rosmmsg show geometry_msgs/Twist
```



# Estructura de ROS: Messages

- Para publicar mensajes en un determinado Topic podemos usar **rostopic pub**:

```
rostopic pub [topic] [msg_type] [args]
```

- Podemos usar **rostopic pub** para hacer que la tortuga se mueva:

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]'
```

```
'[0.0, 0.0, 1.8]'
```

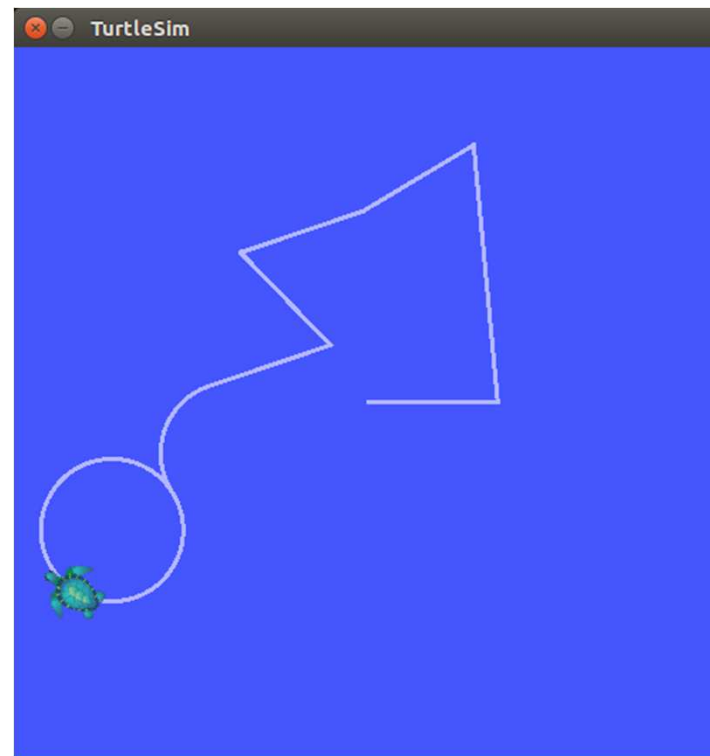


# Estructura de ROS: Messages

- Si queremos que la tortuga no pare de moverse, en vez de enviar un único mensaje enviamos un flujo de mensajes con una frecuencia de 1 Hz:

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]'
```

```
'[0.0, 0.0, -1.8]'
```



# Estructura de ROS: Messages

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]'  
'[0.0, 0.0, 1.8]'
```

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]'  
'[0.0, 0.0, -1.8]'
```

# Estructura de ROS: Messages

Topic en el que  
publicar los mensajes

Tipo de mensajes

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
```

# Estructura de ROS: Messages

Publicar un solo mensaje

Topic en el que  
publicar los mensajes

Tipo de mensajes

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]'
```

```
'[0.0, 0.0, 1.8]'
```

Publicar mensajes con  
una frecuencia de 1 Hz

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]'
```

```
'[0.0, 0.0, -1.8]'
```

# Estructura de ROS: Messages

Publicar un solo mensaje

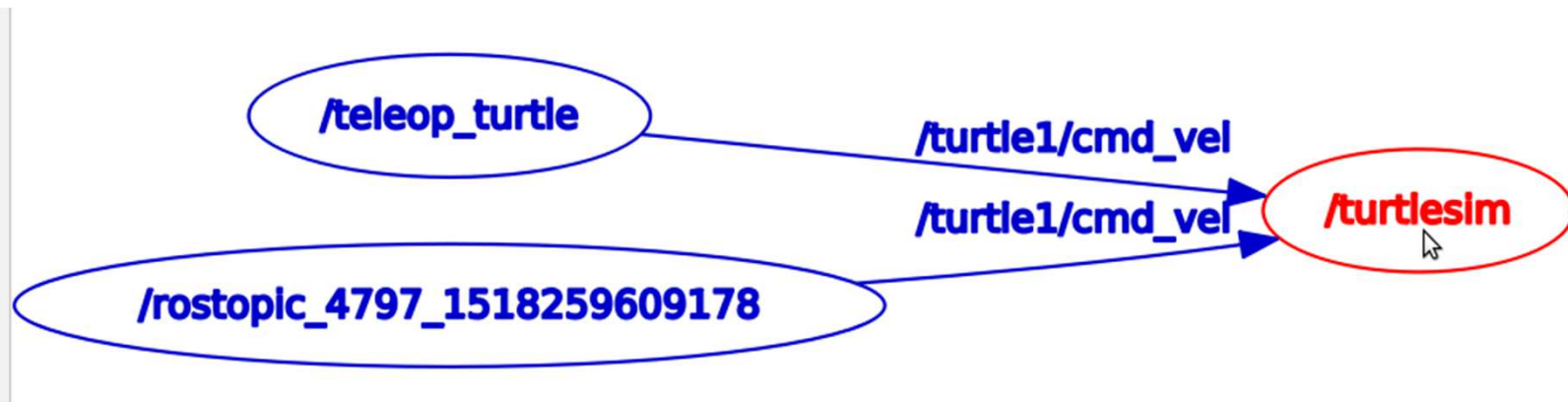
Topic en el que publicar los mensajes

Tipo de mensajes

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

Publicar mensajes con una frecuencia de 1 Hz

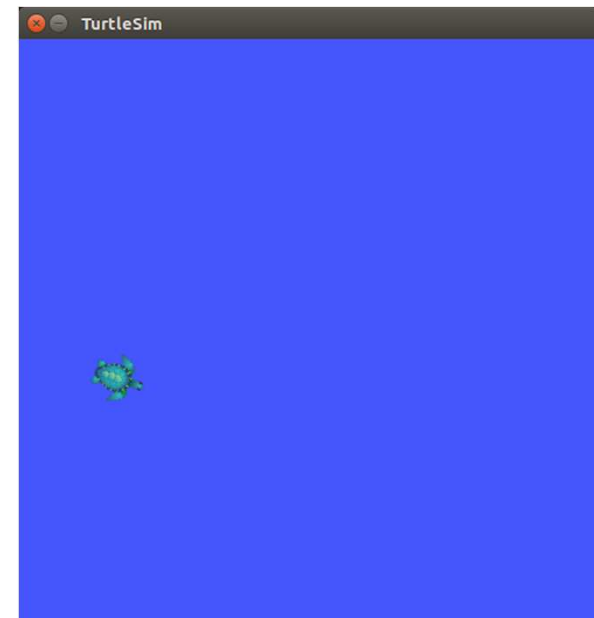
```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
```



# Estructura de ROS: Services & Parameters

- **Services:** son otra forma de comunicación entre nodos. Permiten a los nodos enviar una petición y recibir una respuesta.
- Usan comunicación **síncrona**. Los **Topics** son comunicación **asíncrona**.
- Algunos servicios requieren parámetros y otros no.
- Aquellos servicios que no tienen parámetros son de tipo **empty**
- Podemos consultar el tipo de un servicio con  
**rosservice type:** `rosservice type [service]`
- Podemos llamar a un servicio con el comando  
**rosservice call:** `rosservice call [service] [args]`
- Para limpiar el fondo de turtlesim\_node llamamos al servicio *clear*:

```
$ rosservice call /clear
```



# Estructura de ROS: Services & Parameters

- **Parameter Server:** es un diccionario compartido. Los nodos lo usan para almacenar y recuperar parámetros en tiempo de ejecución. Como no está diseñado para un alto rendimiento, se utiliza mejor para datos estáticos no binarios, como los parámetros de configuración.
- **rosparam** permite almacenar y manipular datos del Parameter Server.

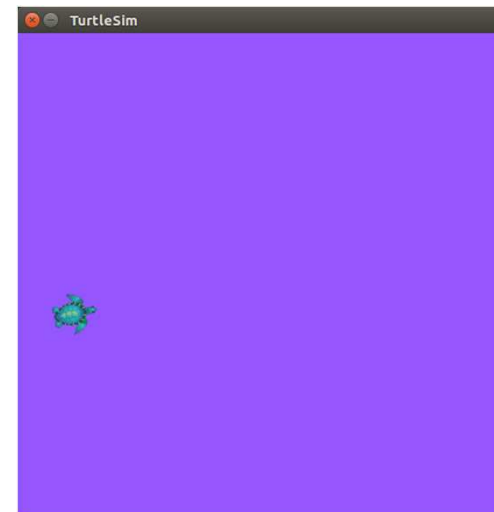
```
$ rosparam list
```

- Así podemos ver que el nodo turtlesim tiene tres parámetros para el color de fondo: /background\_r, /background\_g y /background\_b

```
$ rosparam set /background_r 150
```

- Para hacer efectivo el cambio:

```
$ rosservice call /clear
```





# *roslaunch & launch files*

- **roslaunch** ejecuta nodos según se define en un *archivo launch*

```
roslaunch [package] [filename.launch]
```

- Pasos para crear un archivo *launch* en un paquete:

- Movernos al paquete: `$ roscd beginner_tutorials`

- Crear un directorio *launch* (el directorio podría tener otro nombre):

```
$ mkdir launch  
$ cd launch
```

- Crear un archivo con extensión *.launch* dentro del directorio

- El archivo tiene una estructura XML y comienza y finaliza con la etiqueta `<launch>`:

```
<launch>  
    .....  
</launch>
```

# *roslaunch & launch files*

- Ejemplo de *launch file*:

```
<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 & launch files*

- Ejemplo de *launch file*:

Abre dos grupos con distinto namespace. Esto nos permite iniciar dos simuladores sin conflictos de nombres

`<launch>`

El archivo comienza abriendo la etiqueta *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>`

El archivo termina cerrando la etiqueta *launch*

Inicia nodo *mimic*:  
*turtlesim2* va a imitar a *turtlesim1*

# Trabajo en casa para esta semana

Realizar hasta el décimo tutorial (incluido) de 'Beginner Level' <http://wiki.ros.org/ROS/Tutorials>