Instalación y configuración de ROS (26 dic)

Cada versión de ROS soporta un máximo de tres versiones de ubuntu, por lo que conviene tener claro qué versión vamos a instalar. Para Ubuntu 12.10 debemos instalar ROS groovy.

Ubuntu 12.10

Instalación de ROS Groovy

Añadir los repositorios de ros-groovy para ubuntu 12.10 (a 12 dic 2012)

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu quantal main" >
/etc/apt/sources.list.d/ros-latest.list'
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
sudo apt-get update
sudo apt-get upgrade
```

Instalar con apt-get (todas las dependencias se instalan automáticamente).

```
sudo apt-get install ros-groovy-desktop-full
```

Instalar también otras herramientas que facilitan el trabajo con ROS.

```
sudo apt-get install python-rosinstall python-rosdep
```

Configuración de ROS Groovy

Configuración inicial:

```
sudo rosdep init
rosdep update
source /opt/ros/groovy/setup.bash
```

Creamos nuestro espacio de trabajo (se recomienda para no ensuciar la instalación base)

```
cd /opt/ros
mkdir rosws
sudo chmod -R 775 /opt/ros/rosws
sudo chown user:user /opt/ros/rosws
rosws init /opt/ros/rosws /opt/ros/groovy
```

Añadimos el script de configuración al bashrc

```
echo "source /opt/ros/rosws/setup.bash" >> ~/.bashrc
. ~/.bashrc
```

Instalación de stacks desde repositorios

Para instalar cualquier stack desde repositorios:

```
roscd
roslocate info STACKNAME | rosws merge -
source setup.sh
rosws update STACKNAME
rosmake STACKNAME
```

Un ejemplo de instalación del stack ROSARIA. Este stack es un wrapper de ARIA para ROS que nos permite acceder al sonar y odometría de todos los robots de MobileRobots/ActivMedia.

```
roscd
roslocate info ROSARIA | rosws merge -
source setup.sh
rosws update ROSARIA
rosmake ROSARIA
```

Tutoriales de ROS

Tutorial de la herramienta rosws

Crear un paquete ROS

En este tutorial mostraremos cómo crear un paquete de software para ROS. A modo de ejemplo, crearemos un paquete ficticio con un único nodo ROS, que se encargará de las siguientes tareas:

1. Abrir un sensor ficticio.
2. Leer los datos que proporciona el sensor de forma periódica.
3. Publicar los datos a través del sistema de intercambio de mensajes ROS. El programa que se encargue de esto recibirá el nombre de *publisher*.

Nuestro sensor ficticio proporcionará la posición 3D de un objeto y la marca temporal asociada a cada dato:

```
timestamp x y z
```

Antes de comenzar el tutorial, debemos descargarnos los fuentes que utilizaremos en el paquete sensorfakesources.zip

- SensorFake.cpp y SensorFake.h: librería de interacción con el sensor.
- SensorFake_Publisher.cpp: *publisher*.

Diseño del paquete ROS

Al tratarse de un paquete sencillo, proponemos un diseño que no complique en exceso ni la implementación ni el mantenimiento del mismo. Nuestro paquete constará de los siguientes archivos:

1. Librería para el manejo del sensor. Proporcionará las funciones típicas: conexión, lectura, desconexión, etc. Debería ser totalmente independiente de ROS.
2. Nodo ROS encargado de comunicarse con el sensor (a través de su librería) y transmitir los datos a través del sistema de mensajes ROS.

Para paquetes más complejos, se recomienda encarecidamente utilizar un diseño más adecuado.

Creación del paquete

Nos situamos en la carpeta donde queramos crear nuestro paquete. Por ejemplo:

```
${HOME}/ros
```

Creamos el paquete. El nombre del paquete será *sensor_fake* y dependerá de *roscpp*.

```
roscreate-pkg sensor_fake roscpp
```

La salida del comando debería ser algo como:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!
WARNING: current working directory is not on ROS_PACKAGE_PATH!
Please update your ROS_PACKAGE_PATH environment variable.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!
Created package directory /home/citius_robot/ros_pruebas/sensor_fake
Created include directory
/home/citius_robot/ros_pruebas/sensor_fake/include/sensor_fake
Created cpp source directory /home/citius_robot/ros_pruebas/sensor_fake/src
Created package file /home/citius_robot/ros_pruebas/sensor_fake/Makefile
Created package file /home/citius_robot/ros_pruebas/sensor_fake/manifest.xml
Created package file
/home/citius_robot/ros_pruebas/sensor_fake/CMakeLists.txt
Created package file /home/citius_robot/ros_pruebas/sensor_fake/mainpage.dox

Please edit sensor_fake/manifest.xml and mainpage.dox to finish creating
your package
```

Esto nos indica: 1) que el paquete creado no está en la variable de entorno *ROS_PACKAGE_PATH* (solución en la siguiente sección), 2) que se ha creado correctamente la estructura de directorios y ficheros de configuración del paquete.

Si nos desplazamos a la carpeta correspondiente (*${HOME}/ros_pruebas/sensor_fake*), observamos la estructura del paquete:

- include: directorio para ficheros ".h".
- * sensor_fake: directorio para ficheros ".h" de la librería que se encarga de la interacción con el sensor.
- src: directorio para ficheros ".cpp"
- mainpage.dox: fichero para generacion de documentación con Doxygen.
- CMakeLists.txt: configuración de la compilación con CMake (enlace).
- manifest.xml: información sobre el paquete (enlace).

Añadiendo el paquete al workspace

Desde ${HOME}/ros, añadimos el paquete al workspace de ROS.

```
rosws set sensor_fake
```

Alternativamente, podemos ejecutar:

```
rosstack profile && rospack profile
```

Cerramos el terminal y lo abrimos de nuevo, para que se recarguen las variables de entorno correspondientes. A continuación, ejecutamos:

```
rospack profile
```

En este punto, ROS debería ser capaz de encontrar nuestro paquete. Lo comprobamos mediante:

```
rospack find sensor_fake
```

Comprobamos que todo funciona correctamente ejecutando una compilación de prueba:

```
rosmake sensor_fake
```

Cuya salida debería ser algo como:

```
[ rosmake ] rosmake starting...
[ rosmake ] Packages requested are: ['sensor_fake']
[ rosmake ] Logging to directory
/home/david/.ros/rosmake/rosmake_output-20121115-180018
[ rosmake ] Expanded args ['sensor_fake'] to:
['sensor_fake']
[rosmake-0] Starting >>> roslang [ make ]
[rosmake-0] Finished <<< roslang  No Makefile in package roslang
[rosmake-0] Starting >>> roscpp [ make ]
[rosmake-0] Finished <<< roscpp  No Makefile in package roscpp
[rosmake-0] Starting >>> sensor_fake [ make ]
[rosmake-0] Finished <<< sensor_fake [PASS] [ 5.56 seconds ]
[ rosmake ] Results:
[ rosmake ] Built 3 packages with 0 failures.
[ rosmake ] Summary output to directory
[ rosmake ] /home/david/.ros/rosmake/rosmake_output-20121115-180018
```

Importación a Eclipse

Si queremos importar este paquete a un proyecto Eclipse, podemos seguir las instrucciones que se encuentran en: http://www.ros.org/wiki/IDEs#Eclipse

Compilación de la librería para interacción con sensor_fake

En primer lugar, copiamos el archivo *SensorFake.cpp* en *${HOME}/ros/sensor_fake/src* y el archivo *SensorFake.h* en *${HOME}/ros/sensor_fake/include/sensor_fake* (fuentes: sensorfakesources.zip). A continuación, modificamos el *CMakeLists.txt* para que los compile como una librería dinámica.

```
#common commands for building c++ executables and libraries  -> Añadimos el comando debajo de este comentario
rosbuild_add_library(${PROJECT_NAME} src/SensorFake.cpp)
#Esto indica que queremos compilar SensorFake.cpp como una librería dinámica. Esta librería tendrá el mismo nombre que el proyecto
# (sensor_fake)
```

Compilamos:

```
rosmake sensor_fake
```

Nota importante: SensorFake.cpp asume que el SensorFake.h está en *include/sensor_fake*, al hacer:

```
#include "sensor_fake/SensorFake.h"
```

De no se así (por ejemplo, si le hemos dado otro nombre al paquete), debemos modificar este *include*.

Si la compilación es satisfactoria, *rosmake* nos habrá creado una nueva carpeta *${HOME}/ros/sensor_fake/lib*, que contendrá la librería compilada *libsensor_fake.so*.

Definición de los mensajes ROS

Ahora debemos definir los mensajes que nuestro paquete va a utilizar para comunicarse con ROS. Para ello, creamos la carpeta *${HOME}/ros/sensor_fake/msg* y copiamos en ella el archivo de definición del mensaje (*XYZ.msg*). A continuación, modificamos el archivo *CMakeLists.txt* para que compile la definición del mensaje.

```
#uncomment if you have defined messages
rosbuild_genmsg()      #---> Descomentamos esta línea
```

Compilamos:

```
rosmake sensor_fake
```

Si la compilación es correcta, *rosmake* habrá creado dos nuevas carpetas: 1) *${HOME}/ros/sensor_fake/msg_gen* y 2) *${HOME}/ros/sensor_fake/src/sensor_fake/*. Estas carpetas

contienen las clases necesarias para el manejo del mensaje generado.

Compilación del //publisher//

Ahora estamos en disposición de escribir un programa que lea los datos de nuestro *sensor_fake* y los publique en ROS (utilizando el tipo de mensaje generado en el apartado anterior). En primer lugar, copiamos el archivo *SensorFake_Publisher.cpp* en *${HOME}/ros/sensor_fake/src* (fuentes: sensorfakesources.zip). A continuación, modificamos el *CMakeLists.txt* para enlazar la librería *{HOME}/ros/sensor_fake/lib/libsensor_fake.so* y compilar el *publisher* como ejecutable.

```
#common commands for building c++ executables and libraries  ## -> Ya
estaba!
rosbuild_add_library(${PROJECT_NAME} src/SensorFake.cpp)   ## -> Ya estaba!
rosbuild_add_executable(SensorFake_Publisher src/SensorFake_Publisher.cpp)
target_link_libraries(SensorFake_Publisher ${PROJECT_NAME})
```

Si la compilación es satisfactoria, *rosmake* habrá creado el ejecutable *${HOME}/ros/sensor_fake/bin/SensorFake_Publisher*.

Ejecución del nodo ROS

Finalmente, probamos nuestro paquete. Para ello, abrimos un terminal y ejecutamos el servidor ROS:

```
roscore
```

En otro terminal, ejecutamos nuestro *publisher* (nodo ROS):

```
rosrun sensor_fake SensorFake_Publisher
```

La salida debería ser algo como:

```
Hello, I am the SensorFake_Publisher
Time 1353001026.799379: Sending x [0] y [5] z [10]
Time 1353001026.899426: Sending x [1] y [6] z [11]
Time 1353001026.999419: Sending x [2] y [7] z [12]
```

En otro terminal, ejecutamos un *listener* de ROS, que recibirá los mensajes de nuestro *publisher* y los mostrará por pantalla.

```
rostopic echo /SensorFake_Topic
```

La salida debería ser algo como:

```
header:
  seq: 31
  stamp:
    secs: 1353001134
    nsecs: 532552707
```

```
  frame_id: ''
timestamp:
  secs: 1353001134
  nsecs: 532516000
x: 1
y: 6
z: 11
---
header:
  seq: 32
  stamp:
    secs: 1353001134
    nsecs: 632531208
  frame_id: ''
timestamp:
  secs: 1353001134
  nsecs: 632489000
x: 2
y: 7
z: 12
---
```

Compartir el paquete

Si queremos compartir el paquete (por ejemplo, enviarlo por email), debemos borrar la carpeta *${HOME}/ros/sensor_fake/build*. En el ordenador destino, copiamos el paquete (por ejemplo, en *${HOME}/ros*) y ejecutamos los pasos de la Sección “Añadiendo el paquete al workspace”.

Stage + AMCL + RVIZ

En este tutorial veremos:

1. Cómo simular un robot con Stage.
2. Cómo ejecutar el algoritmo de localización AMCL. Este algoritmo estimará la posición del robot sobre un mapa en base a la información del simulador.
3. Cómo visualizar todos los datos con el *rviz*.

Para ello, necesitamos instalar:

1. Stack Stage: *sudo apt-get install ros-groovy-stage*
2. Nodos AMCL y map_server(stack Navigation): *sudo apt-get install ros-groovy-navigation*
3. Nodo rviz: *sudo apt-get install ros-groovy-rviz*

Archivos de configuración

En primer lugar, debemos descargar el archivo world.zip. Podemos descomprimir este archivo en

cualquier lugar a nuestro gusto (en adelante, llamaremos *${WORLD_PATH}* a esta carpeta). En particular, nos van a interesar los siguientes archivos:

1. *${WORLD_PATH}/citius/plantabajacitius.yaml*: archivo descriptor del mapa que utilizaremos en este tutorial.
2. *${WORLD_PATH}/citius/plantabajacitius.pgm*: imagen que representa dicho mapa.
3. *${WORLD_PATH}/citius/plantabajacitius.world*: archivo "world" del Stage (utiliza el archivo plantabajacitius.pgm).

A continuación, descargamos el archivo amcl_diff_stage.launch.zip y lo extraemos en la carpeta "examples" del nodo AMCL. Para localizar la carpeta del nodo AMCL, hacemos:

```
roscd amcl
```

Este archivo contiene los parámetros que se le pasarán al AMCL. Podemos modificar los parámetros editando el archivo.

Finalmente, descargamos el archivo stage_amcl.rviz.zip y lo extraemos en una carpeta a nuestro gusto (en adelante, *${RVIZ_CONFIG}*). Este archivo contiene parámetros de configuración para adaptar el *rviz* (visualizador) a las necesidades de este ejemplo.

Ejecución de los nodos (uno por terminal)

Arrancamos el *roscore*:

```
roscore
```

Ejecutamos el simulador Stage con un archivo "world". Por ejemplo:

```
rosrun stage stageros ${WORLD_PATH}/citius/plantabajacitius.world
```

Ejecutamos el map_server:

```
rosrun map_server map_server ${WORLD_PATH}/citius/plantabajacitius.yaml
```

Ejecutamos el AMCL con el archivo de configuración descargado:

```
roslaunch amcl amcl_diff_stage.launch
```

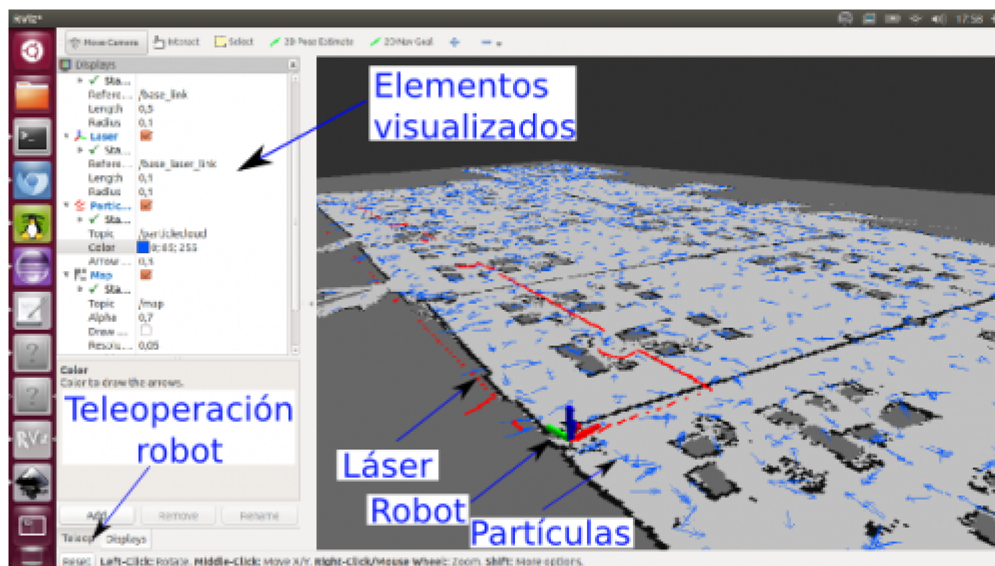
AMCL expone un servicio (global_localization) mediante el cual se puede activar el proceso de localización global del robot. En otro terminal, hacemos:

```
rosservice call /global_localization
```

Comprobamos con la herramienta *rxgraph* que todos los nodos están: 1) levantados, 2) comunicándose correctamente. A continuación, podemos visualizar el resultado de los diferentes nodos utilizando la herramienta *rviz*:

```
rosrun rviz rviz ${RVIZ_CONFIG}/stage_amcl.rviz
```

Deberíamos ver algo como:



:

Ahora podemos mover el robot (pestaña teleop) por el entorno para que se localice.

Rosbag

Rosbag es una herramienta para grabar, reproducir y modificar logs con los mensajes que recibe el nodo roscore (pose, medidas del láser, transformaciones geométricas, etc.). En terminología ROS, a estos archivos de log se les conoce como bags.

Rosbag record

Rosbag play

Rosbag filter

Rosbag nos permite generar nuevos archivos bag a partir de un bag ya existente. Presentamos aquí algunos ejemplos:

Generar un bag que sólo contenga mensajes "/RosAria/pose":

```
rosbag filter original.bag filtrado.bag  'topic == "/RosAria/pose"'
```

Generar un bag que sólo contenga mensajes "/RosAria/pose" y "/tf":

```
rosbag filter original.bag filtrado.bag  'topic == "/RosAria/pose"  or topic
=="/tf"'
```

Generar un bag que contenga todos los mensajes excepto el "/RosAria/pose":

```
rosbag filter original.bag filtrado.bag  'topic != "/RosAria/pose"'
```

Creación de un controlador básico

Descargar el siguiente paquete: basicsample.zip. Compilar y ejecutar.

Para probar el código con el Stage necesitamos hacer un re-mapping de los topics como si fuera el robot real:

```
<launch>
  <node pkg="stage" type="stageros" name="stage" output="screen"
args="$(find stage)/world/citius/plantabajacitius.world">//RUTA AL ARCHIVO
WORLD
    <remap from="/base_scan" to="/scan"/>
    <remap from="/odom" to="/RosAria/pose"/>
    <remap from="/base_laser_link" to="/laser"/>
  </node>
  <arg name="map_file" default="$(find
stage)/world/citius/plantabajacitius.yaml" /> //RUTA AL MAPA
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg
map_file)" />
</launch>
```

From:
https://wiki.citius.usc.es/ - **Wiki do CiTIUS**

Permanent link:
https://wiki.citius.usc.es/inv:por-clasificar:ros

Last update: **2013/03/27 10:55**