

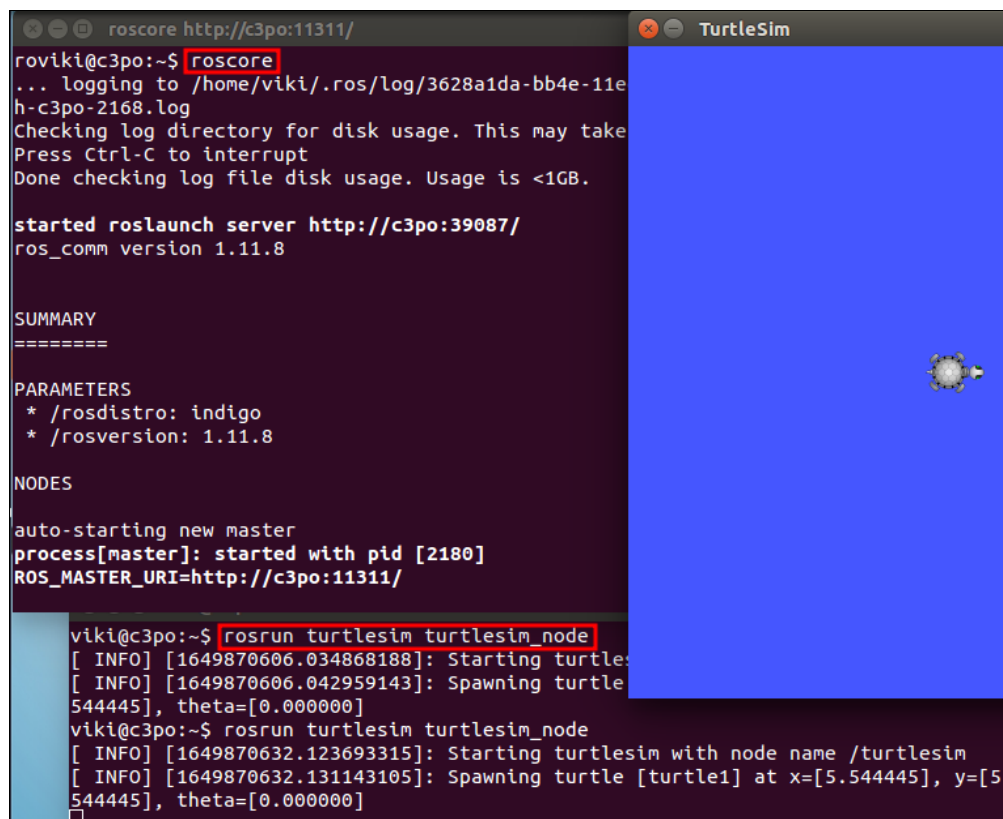
# Práctica 4. Robot Operating System

Hecho por Marcos Hidalgo Baños y Aquiles Fernández Gambero a día 01/04/2022

## Introducción y objetivos de la práctica.

Tal y como se muestra en su web oficial ([wiki.ros.org](http://wiki.ros.org)) 'ROS provee librerías y herramientas para ayudar a los desarrolladores de software a crear aplicaciones para robots'. Para esta práctica, utilizaremos la herramienta educativa **Turtlesim** que nos permitirá conocer los conceptos básicos sobre los nodos y paquetes en ROS.

Para poder visualizar a la tortuguita moverse necesitaremos realizar una serie de pasos en la terminal de nuestra máquina virtual. En primer lugar, hemos de abrir un nuevo terminal en el que escribiremos el comando **roscore** (tal y como se indica en la captura) para que los nodos de ROS se puedan comunicar entre sí. Esta instancia debe estar abierta durante todo el transcurso de la práctica, por lo que tendremos que abrir de nuevo otra terminal para poder seguir trabajando. Tras ejecutar **roslaunch turtlesim turtlesim\_node** veremos como finalmente aparece nuestra querida amiga (algo quieta para lo inquieta que es).



```
roscore http://c3po:11311/
roviki@c3po:~$ roscore
... logging to /home/viki/.ros/log/3628a1da-bb4e-11e
h-c3po-2168.log
Checking log directory for disk usage. This may take
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://c3po:39087/
ros_comm version 1.11.8

SUMMARY
=====
PARAMETERS
* /rostdistro: indigo
* /rosversion: 1.11.8

NODES

auto-starting new master
process[master]: started with pid [2180]
ROS_MASTER_URI=http://c3po:11311/

viki@c3po:~$ roslaunch turtlesim turtlesim_node
[ INFO] [1649870606.034868188]: Starting turtle:
[ INFO] [1649870606.042959143]: Spawning turtle
544445], theta=[0.000000]
viki@c3po:~$ roslaunch turtlesim turtlesim_node
[ INFO] [1649870632.123693315]: Starting turtlesim with node name /turtlesim
[ INFO] [1649870632.131143105]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
```

## Elaboración del Script con las instrucciones.

Como hemos podido comprobar, nuestra tortuguita está a la espera de que algún nodo ROS le dé instrucciones sobre cómo moverse por el plano de la pantalla, así que es función del equipo programador implementar un **programa C++** (puede ser también Python) que a partir de los **datos de entrada** albergados en un documento de texto mande a través de un *Subscriber* los datos de velocidad lineal y angular de la tortuga, por lo que las nuevas posiciones y orientación se deberán mostrar por pantalla mediante un *Publisher*.

```
tortuguita.cpp x
#include "turtlesim/Pose.h"
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"

#include <iostream>
#include <sstream>
#include <fstream>
#include <array>
#include <string>
using namespace std;

/** Variables globales **/
ros::Publisher vel_pub;
ros::Subscriber pose_sub;
int const NUMERO_DATOS = 6;
typedef array<double, NUMERO_DATOS> arrayDatos;

/** Funciones **/
void poseCallback(const turtlesim::Pose::ConstPtr& msg);
void move(double speed, double distance, bool isForward);
void rotate(double angular_vel, double relative_angle, bool clockwise);
arrayDatos procesarEntrada(string fila);
```

Dicho archivo tiene como nombre **tortuguita.cpp** y se encuentra en el paquete con el mismo nombre.

Como se puede observar en la captura de pantalla, este archivo hará las veces de Publisher mediante la variable **vel\_pub** y de Subscriber mediante **pose\_sub**.

Además, emplearemos un array para almacenar cada línea de datos que nuestro programa ejecute.

El programa realizará la labor de un nodo de ROS, es decir, publicar al topic que sea oportuno para actualizar la velocidad del robot, que en nuestro caso es **cmd\_vel**, y suscribirse al nodo turtlesim (ver función poseCallback).

A continuación deberá obtener los valores de velocidad del fichero datos.txt por lo que nos serviremos de la librería **ifstream** de C++ para acceder a su contenido.

Una vez estén en el array de datos mandaremos la orden mediante las funciones descritas a continuación.

```
*tortuguita.cpp x
int main(int argc, char **argv)
{
    ros::init(argc, argv, "pr4");
    ros::NodeHandle n;

    /** Utilizamos el mismo fichero para publicar y suscribir **/
    vel_pub = n.advertise<geometry_msgs::Twist>("/turtle1/cmd_vel", 1000);
    pose_sub = n.subscribe("turtlesim", 1000, poseCallback);

    /** Lectura del fichero **/
    /** Ej de linea de fichero: 2 2 0 0 1.8; **/
    ifstream indata("/home/viki/catkin_ws/src/tortuguita/datos.txt");
    string fila;
    arrayDatos valores;

    /** Abrimos el fichero **/
    if(indata.is_open())
    {
        /** Mientras haya lineas por leer... **/
        while(getline(indata, fila))
        {
            /** ... obtenemos los datos **/
            valores = procesarEntrada(fila);
            if(valores[0] != 0.0)
                move(valores[0], 1, true);
            if(valores[5] != 0.0)
                rotate(valores[5], 1, true);
            ros::spinOnce();
        }
    }
    else cout << "Error al abrir el fichero." << endl;
    ros::shutdown();
    return 0;
}
```

Este método nos permitirá trackear la posición relativa de la tortuguita en todo momento.

```
tortuguita.cpp x
/** Muestra la posicion de la tortuga **/
void poseCallback(const turtlesim::Pose::ConstPtr& msg)
{
    ROS_INFO("Position of turtle: x:[%f] y:[%f] theta:[%f] linVel:[%f] angVel:[%f]",
    msg->x, msg->y, msg->theta, msg->linear_velocity, msg->angular_velocity);
}
```

```
tortuguita.cpp x
/** Metodo para mover el robot en linea recta **/
void move(double speed, double distance, bool isForward)
{
    geometry_msgs::Twist vel_msg;

    // linear velocity
    if (isForward)
        vel_msg.linear.x = abs(speed);
    else
        vel_msg.linear.x = -abs(speed);
    vel_msg.linear.y = 0;
    vel_msg.linear.z = 0;
    // angular velocity
    vel_msg.angular.x = 0;
    vel_msg.angular.y = 0;
    vel_msg.angular.z = 0;

    double t0 = ros::Time::now().toSec();
    double distancia_actual = 0.0;
    ros::Rate loop_rate(10);

    do{
        vel_pub.publish(vel_msg);
        double t1 = ros::Time::now().toSec();
        /** Def fisica --> dist = vel * tiempo **/
        distancia_actual = speed * (t1-t0);
        ros::spinOnce();
        loop_rate.sleep();
    }while(distancia_actual < distance);

    vel_msg.linear.x = 0;
    vel_pub.publish(vel_msg);
}
```

Si la entrada indica un desplazamiento sobre el eje OX del robot un método deberá ser el encargado de elaborar y mandar el mensaje a turtlesim con dicho propósito.

El resto de dimensiones y las velocidades angulares serán iguales a cero, por lo que necesitaremos algún otro método que nos permita handlear las rotaciones.

Esto es debido a que la tortuga se mueve sobre el plano, por lo que la dimensión OZ no tiene sentido. Dicho desplazamiento se realizará en un tiempo que tendremos que medir para poder calcular cuánto se ha desplazado la tortuguita (mediante la fórmula física indicada en el código).

De igual manera que move, rotate permite al robot realizar un giro en función al valor proporcionado por angular\_vel (que a su vez se corresponde con el quinto elemento del array de datos).

La principal diferencia con su predecesor es que debemos contemplar la posibilidad de que el giro se realice en sentido horario o antihorario, pero esto también se solventa de forma similar.

En el resto de aspectos, el razonamiento que impulsa a construir el código así es el mismo.

```
tortuguita.cpp x
/** Metodo para rotar el robot **/
void rotate(double angular_vel, double relative_angle, bool clockwise)
{
    geometry_msgs::Twist vel_msg;

    vel_msg.linear.x = 0;
    vel_msg.linear.y = 0;
    vel_msg.linear.z = 0;

    vel_msg.angular.x = 0;
    vel_msg.angular.y = 0;

    if(clockwise)
        vel_msg.angular.z = -abs(angular_vel);
    else
        vel_msg.angular.z = abs(angular_vel);

    double angulo_actual = 0.0;
    double t0 = ros::Time::now().toSec();
    ros::Rate loop_rate(10);

    do{
        vel_pub.publish(vel_msg);
        double t1 = ros::Time::now().toSec();
        angulo_actual = angular_vel * (t1-t0);
        ros::spinOnce();
        loop_rate.sleep();
    }while(angulo_actual < relative_angle);

    vel_msg.angular.z = 0;
    vel_pub.publish(vel_msg);
}
```

```

arrayDatos procesarEntrada(string fila)
{
    size_t pos = 0;
    arrayDatos datos;
    string valor;
    string delimitador = " ";
    int cont = 0;

    while((pos = fila.find(delimitador)) != string::npos)
    {
        valor = fila.substr(0,pos);
        datos[cont] = stod(valor);
        fila.erase(0, pos+1);
        cont++;
    }
    datos[cont] = stod(fila.substr(0,fila.length()-1));
    return datos;
}

```

También deberemos procesar los datos de entrada desde el fichero. El siguiente método procesarEntrada se encargará de trocear según el formato proporcionado, que consiste en valores separados por espacios, agrupados en líneas terminadas en el carácter punto y coma.

Por último, podemos ver cómo para la entrada **'2.0 0.0 0.0 0.0 0.0 1.8'** la tortuguita dibujará un círculo que recorrerá en función del número de veces que escribamos dicha sentencia en el archivo datos.txt descrito anteriormente. Teniendo en cuenta lo que realiza el código con el resto de valores que no son el primero (velocidad lineal en el eje OX) y el último (velocidad angular en el eje OZ), cualquier otro ejemplo con otros valores en dichas casillas también servirá y el resto de casillas (actualmente con ceros) serán ignoradas.

## Creación del paquete ROS y ejecución del Script.

Antes de poder ejecutar nuestro programa en la tortuguita debemos crear una jerarquía de directorios y archivos específicos de ROS. Esto lo conseguiremos mediante el comando **catkin\_create\_pkg** dentro de la carpeta src de catkin. Como vemos a continuación de ejecutar el comando **echo \$ROS\_PACKAGE\_PATH** la ruta para que ROS acceda a este directorio no está aún habilitada (esto lo vemos porque no aparece la ruta hasta el directorio catkin\_ws/src). Para que esto se vea reflejado, deberemos ejecutar en cada nueva terminal **source devel/setup.bash** y comprobar si queremos que se haya efectuado.

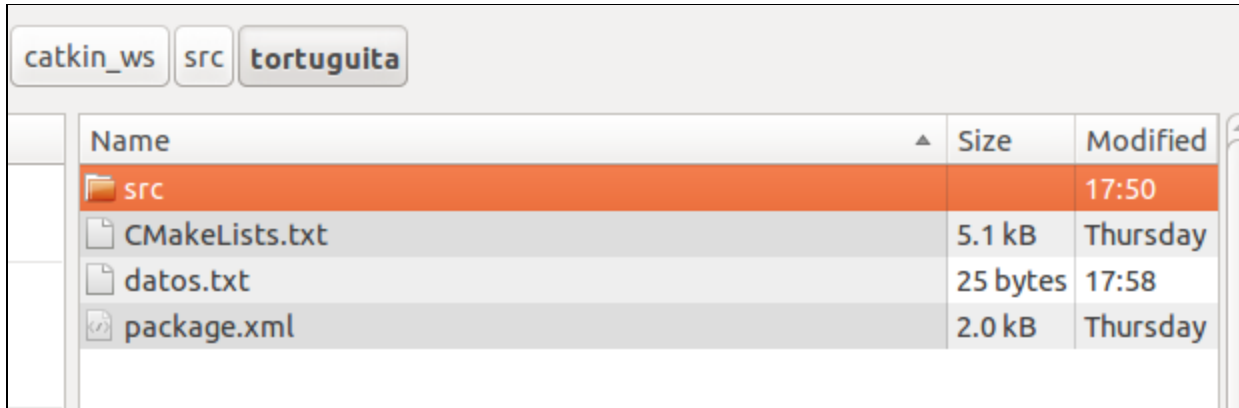
```

viki@c3po:~/catkin_ws$ cd src
viki@c3po:~/catkin_ws/src$ catkin_create_pkg tortuguita
Created file tortuguita/package.xml
Created file tortuguita/CMakeLists.txt
Successfully created files in /home/viki/catkin_ws/src/tortuguita. Please adjust
the values in package.xml.
viki@c3po:~/catkin_ws/src$ echo $ROS_PACKAGE_PATH
/opt/ros/indigo/share:/opt/ros/indigo/stacks
viki@c3po:~/catkin_ws/src$ cd ..
viki@c3po:~/catkin_ws$ source devel/setup.bash
viki@c3po:~/catkin_ws$ echo $ROS_PACKAGE_PATH
/home/viki/catkin_ws/src:/opt/ros/indigo/share:/opt/ros/indigo/stacks

```

Una vez realizada la secuencia de pasos, veremos cómo en la misma terminal en la que estamos trabajando aparece el paquete 'tortuguita' tras realizar un **rospack list**. Esto nos indica que efectivamente hemos podido establecer un directorio de carpetas accesibles para ROS que podrán ejecutarse a la vez que roscore y rosrunturtlesim turtlesim\_node.

La situación final en la que deberíamos encontrar queda descrita en la siguiente captura, donde el contenido de la carpeta src es únicamente el archivo del programa tortuguita.cpp



catkin_ws	src	tortuguita
Name	Size	Modified
src		17:50
CMakeLists.txt	5.1 kB	Thursday
datos.txt	25 bytes	17:58
package.xml	2.0 kB	Thursday

Por último, solamente nos queda realizar los mismos pasos descritos al principio del informe y añadir una nueva terminal que efectúe **roslaunch tortuguita tortuguita.cpp**

#### Referencias y enlaces de utilidad.

- Videos para principiantes de turtlesim.  
<http://wiki.ros.org/turtlesim>
- Blog sobre cómo crear un paquete ROS desde cero.  
<https://aniekan.blog/2022/01/24/creating-your-first-package-in-ros/>
- Documentación oficial sobre la creación de paquetes ROS.  
<http://wiki.ros.org/catkin/Tutorials/CreatingPackage>