Guía rápida para entender ROS (Robot Operating System)

Por: Adrián Ricárdez Ortigosa 10° Semestre Ing. Mecatrónica
Agradecimientos: M.I. Marco Antonio Negrete, FI – UNAM

Cualquier duda o comentario, mandar correo a: adrian_sigma17@hotmail.com
O contactarme a mi: 55 60 63 63 81
O a Marco Negrete en el laboratorio de bio robótica posgrado DIE: 55 29 63 86 87

Si no te llega a quedar clara alguna sección, de todas formas, en la carpeta de Dropbox "Bipedo_Adrián" de la tesis con el Dr. Edmundo Rocha, se encuentra el workspace completo para que lo puedas ver funcional. La única cosa que tendrías que editar conforme al tutorial es el bashrc (sección 5), ya que contiene alias y source (necesarios para compilar).

Bases para el tutorial

1. Instalar Ubuntu 16.04 como partición de la máquina
2. Descargar e instalar Arduino IDE de: https://www.arduino.cc/en/Main/Software dependiendo del tipo de máquina (32 o 64 bits) para Linux.
3. Instalar ROS kinetic de: http://wiki.ros.org/kinetic/Installation/Ubuntu
4. Instalar Sublime Text de: https://ubunlog.com/instalar-sublime-text-3-ubuntu/
5. De preferencia instalar Terminator:
sudo apt-get update
sudo apt-get install terminator
6. Conocimientos básicos de la IDE de Arduino
7. Familiarización con la terminal de Ubuntu
8. Conocimientos básicos de C++ y Python
9. Plataforma Arduino UNO + 1 potenciómetro de 50K o similar

1. Introducción

Según la página oficial, "Robot Operating System (ROS) es un conjunto de bibliotecas de software y herramientas que ayudan a crear aplicaciones referentes a robots. Desde los controladores, hasta los algoritmos de última generación y con potentes herramientas de desarrollo, ROS tiene lo que necesitas para tu próximo proyecto de robótica. Y es todo de código abierto."

En pocas palabras, ROS nos va a ayudar a establecer un entorno virtual sincronizado y controlado de los procesos que se realizan en cualquier proyecto de robótica.

¿Cuándo saber si usar ROS o no?
Cuando son proyectos colaborativos grandes, o hay mucho manejo de datos y aprovechamiento de señales en el proyecto.

2. Estructura de ROS

ROS: middleware, una combinación entre software y hardware

Sockets: una forma de comunicación entre computadoras, ROS ya tiene patrones de comunicación incluidos que se saltan esta parte.

Grafo de procesos: donde se realiza la ejecución.

ROS es un tipo de administrador de *paquetes*. Los *paquetes* son carpetas de archivos que contienen por defecto 2 elementales: *CMakelists.txt* y *package.xml.* Cualquier carpeta que contenga estos 2, es un *paquete*. Y contiene nodos y tópicos. Los tópicos son la vía por donde se van a transmitir los datos y su nombre es asignado en el código, y los nodos son los programas que realizamos dentro de los paquetes y utilizamos los datos, también, es donde declaramos el nombre del tópico por cómo lo va a reconocer ROS.

Hay dos tipos de patrones en ROS:
- publisher/subscriber: comunicación por tópico (análogo a variables globales), 1:n (de 1 a n nodos) no bloqueante, eso significa que no espera respuesta de los nodos donde publica.
- cliente/servidor: comunicación por servicios (análogo a funciones globales), 1:1 bloqueante, eso significa que espera respuesta para seguir su proceso.

ROS tiene un maestro y n nodos para comunicarse entre sí.

Sistema de archivos: donde se almacenan los paquetes (directorios) y archivos especiales. Éstos ya incluyen cabeceras (include/) y scripts (python/). Su estructura es la siguiente.

- workspace
    - devel
    - build
    - src
        - package
        - package
        - package
            - scripts (python)
            - src (c++)
            - CMakelists.txt: aquí se descomenta *add_executable, add dependencies y target link libraries*
            - package.xml (metadatos)

3. Configuración del espacio de trabajo

Abrimos una terminal.
Primero nos posicionamos en la ubicación de *HOME:*

cd

Luego, creamos una carpeta con el nombre que queramos, por ejemplo *Curso_ROS*:

mkdir Curso_ROS

Nos metemos dentro de esa carpeta:

cd Curso_ROS

Creamos el nombre de nuestro espacio de trabajo:

mkdir catkin_ws

Nos metemos dentro del espacio de trabajo:

cd catkin_ws

Creamos el src principal, donde va a estar casi todo lo que editemos en nuestro sistema de archivos:

mkdir src

Nos metemos en esa carpeta:

cd src

Le decimos a ROS que ésta carpeta será el workspace donde vamos a estar compilando:

catkin_init_workspace

Si en un futuro tenemos otro espacio de trabajo en *HOME*, es necesario indicarlo en el *src* de ese espacio de trabajo tal y como lo hicimos que ese es el nuevo workspace. Además, hay que cerrar y abrir terminales, aplicar el **source** (lo veremos más adelante), y probar si compila.

4. Creación de un paquete y compilación

La convención que seguiremos, es que el nombre del nodo (el .cpp, el que tiene el código), debe tener el mismo nombre del paquete, pero añadiéndole "_node.cpp".

Creamos nuestro primer paquete, estando en el src del workspace:

catkin_create_pkg *First_Package* std_msgs roscpp rospy

Lo que le agregamos después del nombre del paquete son las dependencias. Estas dependencias se pueden editar en el fichero manifest.xml. Este fichero contiene información del paquete. **catkin_create_pkg** creará un directorio con todos los ficheros necesarios de configuración.

Ahora, nos metemos a nuestro paquete:

cd *First_Package*

Y, con ayuda de Sublime Text 3, un editor de texto enriquecido, vamos a editar el CMakeLists de

subl CMakeLists.txt

Descomentamos las líneas de **add_executable**, **add_dependencies** (hay 2, descomentar el segundo), y el **tarjet_link_libraries** con todo y llaves, los cuales se encuentran entre las líneas 100 y 140 del código. Guardamos (**Ctrl + S**) y salimos.

Nos regresamos a la carpeta raíz de nuestro espacio de trabajo, *catkin_ws:*

cd ..
cd ..

Y compilamos con:

catkin_make

Debe salirnos un mensaje con letras azules y un tipo “m – j4” o algo por el estilo, eso quiere decir que ya reconoció el paquete, pero ahora falta crear el nodo (código).

Ejecutamos el ROS MASTER con el comando en una terminal diferente (**Ctrl + Alt + T**) o, si se tiene terminator, dar click derecho en la terminal actual y darle Split, en esa nueva, ingresar:

roscore

Vemos los nodos actuales que se encuentran corriendo, debe arrojar *rosout/*, que es el nodo que se encarga de puras salidas:

rosnode list

Ahora, hay que decirle a ROS que el espacio creado es el espacio de trabajo que vamos a usar para compilar, esto se hace cada vez que se abra una terminal:

source devel/setup.bash

En ROS indigo (para Ubuntu 14.04) hay que agregar las dependencias *run depend* y *build depend* al archivo *package.xml* de los paquetes, pero estamos en kinetic, no hay que hacer eso. También, asegurarse que las dependencias que vamos a utilizar son *std_msgs, roscpp* y *rospy,* y esas van especificadas en las primeras líneas del *Cmakelist.txt.* Si se crea correctamente el paquete, no tenemos por qué agregarlas, pero asegurarse de que estén especificadas, editandolas con Sublime.

En este punto, ya hemos hecho nuestro primer paquete y entendido la compilación de ROS.

Siempre que haya que correr un nodo, que es lo que vamos a hacer a continuación, hay que levantar el nodo MASTER primero (con **roscore**), y después lo demás (con **rosrun**, etc).

Creamos dentro de la carpeta src de nuestro paquete *First Package* un código:

```
#include "ros/ros.h"

int main(int argc, char **argv)       // El argc permite pasarle parámetros al ejecutable (contador de argumentos), y el **argv arreglo de cadenas del roscpp, rospy, std_msgs, pkg_name y el nombre del paquete (5)
{
```

```
    std::cout << "Inicializando paquete: First_Package" << std::endl;       // Imprime en la terminal
    ros::init (argc,argv,"First_Package_node");  // Necesario para que se comunique con el ROS_MASTER, no puede empezar con numero, este es el nombre con el que el MASTER identificará al nodo, el nombre del ejecutable está en CMakeList.txt
    ros::NodeHandle n;          // Se encarga de manejar la comunicación entre el MASTER y los sockets de los nodos, sin este, te va a salir un error en la terminal
    ros::Rate loop(10);         // Frecuencia en Hz, es como un tipo de delay en ROS pero se acopla a lo que se tarde el spinOnce()

    while(ros::ok())       // Esta funcion nos da "true" si todo esta bien, si esta false quiere decir que hubo error
    {
      ros::spinOnce();       // Ejecuta todo lo necesario para ejecutar y checar si hay nuevos datos recibidos por sockets, siempre hay que estarla ejecutando, el procesador se atasca si lo dejamos sin ningún sleep
      loop.sleep();          // Es un tipo de sleep, mide el tiempo a partir de la ultima llamada del spinOnce y se espera el tiempo restante, mejor usar este
    }
    return 0;      // Todo está bien
  }
```

Y lo guardamos como *First_Package_node*.cpp.
Nos vamos a la carpeta raíz (*catkin_ws*), sourceamos ROS y compilamos:

cd
cd Curso_ROS
cd catkin_ws
source devel/setup.bash
catkin_make

En este punto, debe aparecer 100% de compilación, junto con algunas letras de varios colores (generalmente verde, azul y amarillo, rojo brillante es error). Si hubo un error hasta aquí, revisar los pasos anteriores o el código hasta que vuelva a salir. Si se desea eliminar algun directorio por completo y empezar de cero:

rm -rf *Nombre_del_directorio_o_archivo*

5. Ejecución de un nodo

Ahora, hay que hacer lo mismo que hicimos unos pasos arriba, ejecutamos el MASTER, ejecutamos el nodo con **rosrun** y debe de mostrar en la terminal “Inicializando paquete: First Package”, como convención en cada nodo agregamos ese despliegue de información para que sepamos que se ejecutó correctamente:

rosrun *First_Package First_Package_node*

Nótese que no pusimos la extensión. Además, siempre hay que intentar autoacompletar con tabulador, eso ayuda a saber al usuario si existe o no la instrucción o archivo.
Este nodo ejemplo será la plantilla para todos los demás nodos que creemos
Ahora, para no estar poniendo el **source** cada vez que abramos una terminal, editamos el *bashrc*, que es un script que se ejecuta cada vez que se abre una de ellas:

subl ~/.bashrc

Y, hasta abajo del código, agregamos las siguientes líneas:

source ~/Curso_ROS/catkin_ws/devel/setup.bash
alias cm="catkin_make -C ~/Curso_ROS/catkin_ws"

La primera es para “sourcear” el espacio de trabajo y ahorrarnos ese paso que comentamos anteriormente, y el alias que agregamos es para compilar tecleando “cm” en cualquier ubicación de la terminal, para no estar poniendo “catkin_make” en la raíz del espacio de trabajo siempre.
Si hay más de 1 espacio de trabajo, hay que comentar el alias y el source de otro espacio del que no se está utilizando, o modificarlo para que pertenezca al nuevo espacio, por ejemplo:

source ~/Curso_ROS/catkin_ws/devel/setup.bash
alias cm="catkin_make -C ~/Curso_ROS/catkin_ws"
#source ~/ROS_CROFI/catkin_ws/devel/setup.bash
#alias cm="catkin_make -C ~/ROS_CROFI/catkin_ws"

Cerramos terminales, abrimos una con el MASTER, y compilamos en otra:

roscore
cm (ya no es **catkin_make** gracias al alias que usamos en el .bashrc)

Corremos nuestro nodo *First_Package*:

rosrun *My_First_Package My_First_Package_node* (intentar autoacompletar siempre)

Para ver si el nodo está corriendo correctamente, abrimos una terminal aparte (se recomienda descargar Terminator para el uso simultáneo de terminales), y escribir lo siguiente, devuelve el tiempo de ejecución:

rosnode ping /First_Package_node

Deben aparecer unas líneas en la terminal de la descripción de ejecución a través del tiempo.

6. Publisher/Suscriber y uso de rViz

rViz es un visualizador virtual que nos ayuda a crear objetos gráficos vinculados a algún tipo de dato que querramos ver a través del tiempo, por ejemplo, dibujar un robot que mueva la mano cada vez que reciba una señal del exterior, del mundo real. Vamos a hacer uso de él en esta sección.

A partir de aquí nos saltaremos algunos comandos, como **cd** para ir a *HOME*, sólo se dejará indicado.

Creamos otra carpeta dentro del src de nuestro espacio de trabajo llamada *hardware*, en donde se va a controlar los actuadores y se va a recibir las señales, interfaz de bajo nivel con el hardware como tal.

mkdir hardware

Ahora, creamos un paquete dentro de la carpeta *hardware*:

catkin_create_pkg robot_joints roscpp rospy std_msgs geometry_msgs tf

Recordar configurar el CMakelists.txt de este nuevo paquete descomentando **add dependencies**, **add executable** y **target link libraries** explicado en la sección 4 del tutorial.

Realizamos un nuevo código (nodo) llamado *robot_joints_node* (recordar la convención):

```
#include "ros/ros.h"
#include "std_msgs/UInt16.h"    // Va a estar publicando un entero sin signo de 16 bits
#include "geometry_msgs/PointStamped.h"                         // geometry_msgs contiene mensajes que nos sirven para representar abstracciones geométricas: puntos, vectores, vectores en dimensiones, velocidades en el espacio, cuaterniones, etc. Revisa la página de ROS.

int main(int argc, char **argv)       // El argc permite pasarle parámetros al ejecutable (contador de argumentos), y el **argv arreglo de cadenas del roscpp, rospy, std_msgs, pkg_name y el nombre del paquete (5)
{
   std::cout << "Inicializando paquete: robot_joints" << std::endl;      // Imprime en la terminal
   // Nombre del nodo con el que lo reconoce ROS, pero no es el nombre del ejeutable:
   ros::init (argc,argv,"robot_joints");  // Necesario para que se comunique con el ROS_MASTER, no puede empezar con numero, este es el nombre con el que el MASTER identificara al nodo, el nombre del ejecutable está en CMakeList.txt
   ros::NodeHandle n;          // Se encarga de manejar la comunicación entre el MASTER y los sockets de los nodos, sin este, te va a salir un error en la terminal
   ros::Rate loop(10);         // Frecuencia en Hz, es como un tipo de delay en ROS pero se acopla a lo que se tarde el spinOnce()
   //ros::Publisher pubInt = n.advertise<std_msgs::TIPO_DE_DATO>(NOMBRE_TOPICO,TAMAÑO_COLA)

   // Primer punto "planeta"
   //ros::Publisher pubInt_Planeta = n.advertise<std_msgs::UInt16>("POSITION_Planeta",10);    // Crea un publicador, el nombre con el que ponemos el advertise es el que va a desplegar la terminal con rostopic list, el 1 es el tamaño de la cola
   ros::Publisher pubPlaneta = n.advertise<geometry_msgs::PointStamped>("/Planeta",1);                // Las colas son independientes ya que depende de cuantas veces mandes llamar el publish de este tipo de dato
   //std:std_msgs::UInt16 msg;                         // Este es nuestro mensaje, en c++ recuerda que uint16_t = unsigned short
   geometry_msgs::PointStamped msgPlaneta;      // Creamos el mensaje

   //msg.data= 0;                                                  // Este no puede tener "-1" porque es unsigned, "-2" es 65534, y no 0 ni NAN, ya que es complemento a 2
   msgPlaneta.header.frame_id = "point_frame";                   // Le podemos poner el que sea, pero para que lo podamos usar debe existir la transformación desde el que ponemos hasta la referencia, en este caso es el sistema de referencia del punto
   msgPlaneta.point.x = 0.0;                         // Todo está en metros
   msgPlaneta.point.y = 0.0;                         // Todo está en metros
   msgPlaneta.point.z = 0.0;                         // Todo está en metros, recordar no usar en ninguna de estas 3 operaciones comoi: 1/2, porque entero entre entero me va a dar entero por el conjunto, y va a dar 0, a menos que uses 1.0/2, ya que flotante entre entero da flotante, 0.5

   // Segundo punto "satélite"
   //ros::Publisher pubInt_Satelite = n.advertise<std_msgs::UInt16>("POSITION_Satelite",10);    // Crea un publicador, el nombre con el que ponemos el advertise es el que va a desplegar la terminal con rostopic list, el 1 es el tamaño de la cola
   ros::Publisher pubSatelite = n.advertise<geometry_msgs::PointStamped>("/Satelite",1);                // Las colas son independientes ya que depende de cuantas veces mandes llamar el publish de este tipo de dato
   //std:std_msgs::UInt16 msg;                         // Este es nuestro mensaje, en c++ recuerda que uint16_t = unsigned short
   geometry_msgs::PointStamped msgSatelite;      // Creamos el mensaje

   //msg.data = 0;                                                 // Este no puede tener "-1" porque es unsigned, "-2" es 65534, y no 0 ni NAN, ya que es complemento a 2
   msgSatelite.header.frame_id = "point_frame";                  // Le podemos poner el que sea, pero para que lo podamos usar debe existir la transformación desde el que ponemos hasta la referencia, en este caso es el sistema de referencia del punto
   msgSatelite.point.x = 0.0;                        // Todo está en metros
   msgSatelite.point.y = 0.0;                        // Todo está en metros
   msgSatelite.point.z = 0.0;                        // Todo está en metros, recordar no usar en ninguna de estas 3 operaciones comoi: 1/2, porque entero entre entero me va a dar entero por el conjunto, y va a dar 0, a menos que uses 1.0/2, ya que flotante entre entero da flotante, 0.5

   float time = 0;

   while(ros::ok())       // Esta funcion nos da "true" si todo esta bien, si esta false quiere decirue hubo error. PONER () porque si no se le pone, nunca se detendra, y hay que usr KILL
   {
            msgPlaneta.header.stamp = ros::Time::now();                  // Primero agregamos el header, Time es la clase, now es el método estático que regresa el tiempo actual
            msgSatelite.header.stamp = ros::Time::now();                 // Primero agregamos el header, Time es la clase, now es el método estático que regresa el tiempo actual

            /*for(int i = 0; i < 10; i++)
            {
                     msg.data ++;                        // Incrementamos nuestro dato
                     pubInt_Planeta.publish(msg);        // Y publicamos todo el mensaje, no hay que ponerle .data
                     pubInt_Satelite.publish(msg);       // Y publicamos todo el mensaje, no hay que ponerle .data
            }*/

            msgSatelite.point.x = 1.5*cos(2*time);                       // Todo está en metros
            msgSatelite.point.y = 1.5*sin(2*time);                       // Todo está en metros
            msgSatelite.point.z = 0.0;                   // Todo está en metros, recordar no usar en ninguna de estas 3 operaciones comoi: 1/2, porque entero entre entero me va a dar entero por el conjunto, y va a dar 0, a menos que uses 1.0/2, ya que flotante entre entero da flotante, 0.5

            pubPlaneta.publish(msgPlaneta);              // Publicamos el mensaje
            pubSatelite.publish(msgSatelite);            // Publicamos el mensaje

            time += 0.1;

      ros::spinOnce();      // Ejecuta todo lo necesario para ejecutar y checar si hay nuevos datos recibidos por sockets, siempre hay que estarla ejecutando, el procesador se atasca si lo dejamos sin ningún sleep
      loop.sleep();         // Es un tipo de sleep, mide el tiempo a partir de la ultima llamada del spinOnce y se espera el tiempo restante, mejor usar este
   }
   return 0;      // Todo está bien
```

}

Ahora, cerramos todo, abrimos 3 terminales:

- En la primera llamamos al MASTER
  roscore
- En la segunda compilamos y ejecutamos robot_joints_node
  cm
  rosrun robot_joints robot_joints_node
- Y, finalmente en la tercera, vemos la lista de nodos corriendo
  rosnode list

En la tercera terminal deben de aparecer:
/rosout
/robot_joints_node

Para poder ver cuántas veces está ejecutando el nodo mediante el tópico:

rostopic echo /robot_joints (que es el nombre del tópico)

Y lo va a imprimir 10 veces por segundo, ya que en el programa lo configuramos como:
ros::Rate loop(10);

¿Cómo modificar este parámetro? Hagamos un ejemplo:

Si el tamaño de la cola es 1:

```
for(int i = 0; i < 10; i++)
        {
                msg.data ++;                    // Incrementamos nuestro dato
                pubInt.publish(msg);            // Y publicamos todo el mensaje, no hay que ponerle .data
        }
```

Estamos encolando 10 mensajes antes de mandar el spinOnce(), que revisa la cola, y después envía la cola. Encola 1 y se pierden todos los demás, se pierden 9 datos. Si le cambiamos el tamaño de la cola a 10 ya no se pierde ningún dato y se imprime el msg++ cada 1 unidad de msg. Todo viene comentado en el código para más información.

Pero en fin, regresemos al loop a 1 HZ, y ahora publica 1 mensaje por cada segundo

Explicando un poco los headers del programa “robot_joints_node”:
Los tipos de datos con *Stamped* contienen un archivo de header, y contiene: entero de 32 bits sin signo para conteo de mensajes, los *time stamps*, que son marcas de tiempo, es una marca estandarizada de marcar en las computadoras (en número de segundos que han transcurrido desde 1ro de enero 1970) sirve para hacer correcciones de retraso de tiempo para indicar en qué tiempos se tomó un dato, es muy exacto. Y el último dato que tiene header, es un frame ID con el nombre del sistema de referencia con respecto al cual están expresados los datos. Para sincronizar computadoras se necesita CRONY, ya que cada computadora tiene diferente reloj y retraso.

El que usaremos es *Poin Stamped* y contiene un header y el tipo de mensaje.

Podemos usar *geometry_msgs* porque pusimos como dependencia a la hora de crear el paquete en la lista de dependencias en el CMakeLists.txt y package.xml del paquete.

Abrimos otra terminal, y corremos rViz:

rosrun rviz rviz

Nótese que el nombre del paquete es rviz, y el nombre del nodo es rviz.

Solo es un visualizador, no un simulador, sólo dibuja geometrías, trayectorias, etc.

Estructura de Rviz:
En el panel de la izquierda muestra todo lo que está dibujando. En la de la derecha es para opciones generales de visuaulización: donde está la cámara, orbital, etc.

Para visualizar un tópico, hacer click en *ADD* (abajo a la izquierda) → *By Topic* → Y poner el tópico que se supone está publicando.

El zero por default está enmedio del mapa, pero con OFFSET en la izquierda podemos cambiarle el centro, en Grid.

Recordar que *robot_joints_node* es en el que estamos trabajando como ejemplo para poder ir publicando varias cosas.

7. Serial con Arduino y Python en ROS

Es muy sencillo leer serial con python. Vamos a leer el serial del Arduino y mover una pequeña unión de eslabones en Rviz. Nos vamos a la carpeta *hardware*.

Creamos un paquete llamado *pot_reader* (porque ibamos a leer los valores de un potenciómetro), al lado del paquete *robot_joints*, que está en la carpeta de *hardware* en el src de la raíz.

catkin_create_pkg pot_reader roscpp rospy std_msgs tf sensor_msgs

Creamos una carpeta dentro de ese paquete que se llama *scripts*, donde van a estar todos los códigos en python. Por convención, debe llamarse así, los .cpp van en la de *src* del paquete.

Vamos a crear un nodo en python que se llame *pot_reader_node.py*. La primer línea siempre que tiene que ir en todos los códigos de python para que funcione:

#!/bin/env/user python

Recordar que las sangrías son elementales para que funcione. Aguas, no poner acentos a menos que pongas el comando para que acepte caracteres raros.

Código:

```
#!/usr/bin/env python

import rospy                                # Esto es para podernos comunicar con ROS mediante python
import serial
from sensor_msgs.msg import JointState   # Vamos a publicar ese valor de giro del robot, necesitamos publicar un mensaje de tipo JointState
Arduino = serial.Serial("/dev/ttyACM0",115200)       # Crea el puerto serial a cierta velocidad baudrate

rospy.init_node("pot_reader_node");                  # Inicializa el nodo, es con el que ROS va a identificar el nodo, puede ser distinto el nombre al del nodo, o igual. El punto y
coma no es necesario.
loop = rospy.Rate(20);

joints = JointState();
# Arreglo de los nombres de los joints
joints.name = ["link0_link1"];       # Este valor lo sacamos del urdf

# Publicador
pub = rospy.Publisher("joint_states",JointState,queue_size=1);  # Este nombre viene por default, viene en ROS en el joint_state_publisher, la cola es de 1, porque es un byte

while not rospy.is_shutdown():                                  # Este es el que nos va a devolver si todo esta bien, es como el while ros ok en C++
   #print ord(Arduino.read(1));     # Lo convierte a entero lo que lea del serial, el 1 es para que lea un solo byte
   val = (ord(Arduino.read(1))*2*3.1416)/255+180;      # Guardamos en una variable, regla de 3 para que de una vuelta completa
   joints.position=[val];
   joints.header.stamp=rospy.Time.now();
   pub.publish(joints);        # Publicamos

   loop.sleep();
```

Ya que hicimos el programa, en una terminal:

roscore

No es necesario compilar porque es python.
Y ejecutamos en otra terminal:

cd
rosrun pot_reader pot_reader_node.py

Aquí sí lleva la extensión, porque como tal ESE es el ejecutable, no como en C++.

Pero salió un error, de que no es ejecutable, así que hay que marcarle los permisos para hacerlo ejecutable:

Al mismo nivel del .pot_reader_node.py en la carpeta, poner en una terminal:

chmod a+x pot_reader_node.py

Conectamos un Arduino, en este caso usamos el UNO para el tutorial.
Para evitar problemas de permisos, busquemos el puerto serial al que se conectó:

ls /dev

Generalmente es ttyACM0 o ttyUSB0, o 1, o 2. Hay que buscar conectando y desconectando el Arduino, cuál es. Checar que en el programa sea ACM0 o correspondiente, en la línea de “Arduino = serial.Serial("/dev/ttyACM0",115200)   # Crea el puerto serial a cierta velocidad baudrate”. Y ahora le damos permisos:

chmod 777 /dev/ttyACM0

Creamos un programa en arduino llamado pot_reader_withROS y se lo cargamos al Arduino:

```
void setup() {
  Serial.begin(115200);
}

void loop() {
  Serial.write(analogRead(A0)>>2);    // Manda el valor en crudo
  delay(100);
}
```

Nota: RECUERDA SIEMPRE USAR EL MISMO IDENTADO, no aceptará 4 espacios, y en otro lado un tabulador.

Ejecutamos el programa y debe de aparecer los bytes en la terminal cómo los va recibiendo moviendo el potenciómetro.

cd
rosrun pot_reader pot_reader_node.py

Agregaremos a nuestro launch el nodo de pot_reader, y faltantes.

Con esto, es un protocolo para cachar el serial en arduino, pero podemos poner un nodo en el Arduino con la biblioteca ros.h.

8. Launch files

Ahora, no sólo veremos los datos en la terminal, sino que también moveremos al Robot en Rviz. Pero primero, visualizaremos el código de robot_joints (un sistema planetario) en Rviz.

Creamos dentro del paquete de *robot_joints* una carpeta que se llama *launch*.
Creamos un codigo fuente “robot.launch” dentro de la carpeta launch. Estos archivos siempre tienen una estructura tipo .xml o .html:

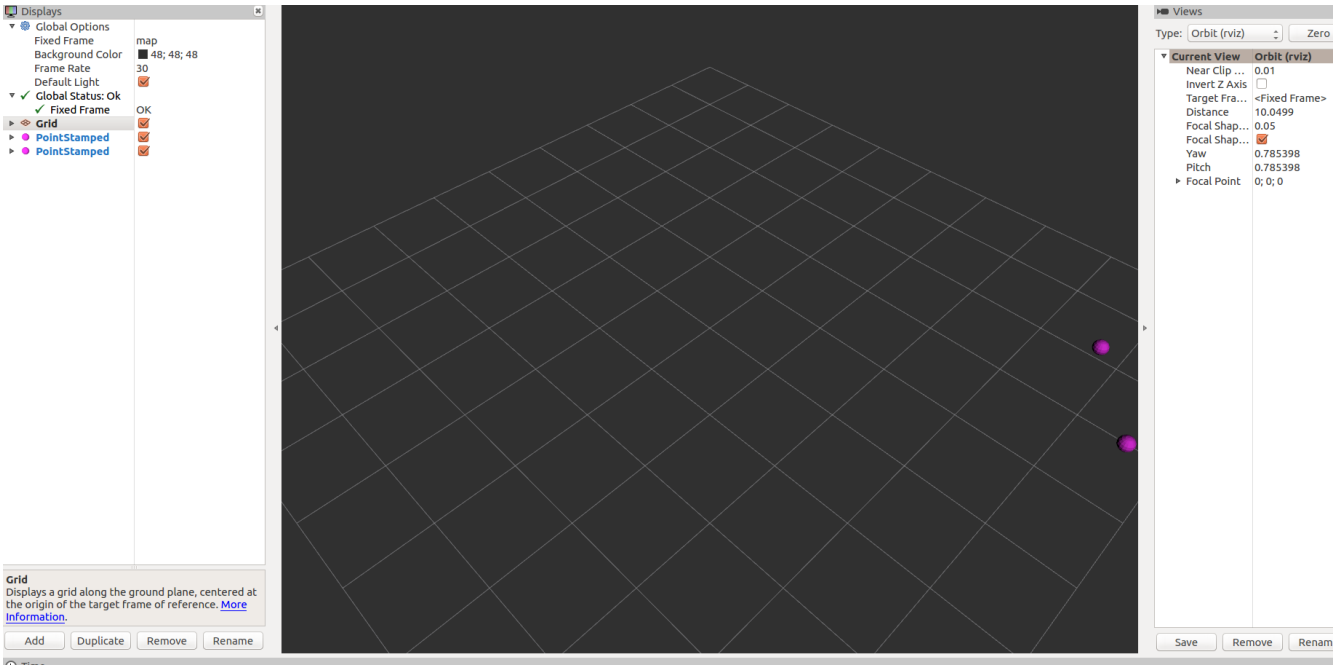
```
<launch>
<!-- FORMATO PARA COMENTAR -->
<!-- ESTA DOCUMENTADO EN LA CUARTA SESIÓN -->

	<group ns="hardware">
		<node name="robot_joints_node" pkg="robot_joints"
			type="robot_joints_node" output = "screen" />
		<node name = "transform_publisher_node" pkg="robot_joints"
			type = "transform_publisher_node" output = "screen"/>
		<node name = "rviz" pkg="rviz"
			type = "rviz" />
	</group>

</launch>
```

Si quieres hacer que sí imprima algo tu nodo, ya que el roslaunch sólo imprime en una sóla terminal, usa “output = “screen” “ con los que quieras que impriman algo en la terminal a la hora de ejecutar el launch.
Al ejecutar un launch por defecto se corre roscore. Al agregar ns es namespace para poder agrupar.
El archivo launch un tipo de script pero manual, que conjunta todos los nodos que quieras ejecutar de golpe.

Para correr un launch:

roslaunch robot_joints robot.launch

Y vamos a ver un sistema planetario girando en Rviz, debe verse parecido a esto:



Recordar añadir en Add, By Topic, */Satelite* y */Planeta*. Jugar con la referencia de map, point_frame.

En el launch se pueden establecer parámetros. Lo puedes fijar en el launch, como el tamaño del robot (por ejemplo). También, el launch sirve para remapear los datos de un tópico, si es que a otro tópico al que te quieres suscribir se llama de otra forma.

Si rViz no encuentra la transformación, simplemente arroja un error. Debe existir esa transformación en el árbol de transformaciones.

Universal Robot description Format (URFD), en un xml definimos las partes móviles del robot.
El URDF nos ayuda a crear el árbol de transformaciones por ejemplo en un brazo robótico, como las traslaciones y rotaciones que hiciste en robótica. Para más información: http://wiki.ros.org/urdf

Dentro del URDF se tienen dos etiquetas: link (los eslabones) y joint (cómo se une cada eslabón).
http://wiki.ros.org/urdf/Tutorials/Building%20a%20Visual%20Robot%20Model%20with%20URDF%20from%20Scratch

Creamos un paquete en src principal (la raiz) que se llama **config_files** con sus atributos roscpp y rospy:

catkin_create_pkg config_files roscpp rospy

Esta vez no es necesario modificar el CMakelists.txt, ya que no usamos ningún nodo.
En la raiz de esta carpeta creamos el código llamado *manipulator.urdf***:**

```
<!--COMENTARIOS ASI-->

<robot name="manipulator">
        <link name="link0">
                <visual>
```

```
                        <geometry>
                                <cylinder length="0.3" radius="0.05"/>
                        </geometry>
                        <origin xyz="0.15 0 0" rpy="0 1.5708 0"/>
                </visual>
        </link>

        <link name="link1">
                <visual>
                        <geometry>
                                <cylinder length="0.2" radius="0.02"/>
                        </geometry>
                        <origin xyz="0.1 0 0" rpy="0 1.5708 0"/>
                </visual>
        </link>

        <joint name="link0_link1" type="revolute">
                <axis xyz="0 0 1"/>
                <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
                <parent link="link0"/>
                <child link="link1"/>
                <origin xyz="0.3 0 0"/>
        </joint>

</robot>
```

Se publica de la siguiente manera, explicando:
El robot_state_publisher, el manipulator.urdf y demás, se ponen en el launch en parámetros.
Para tener ordenado, creamos una paquete llamado *launch_files* en el src de la raiz del espacio de trabajo, donde vamos a meter todos los launch, y creamos una carpeta llamada *launch* dentro de ésta. Ahí dentro creamos un archivo que se llama *manipulator.launch*.

Ejecutamos el launch, para eso debemos estar en el src de la raiz o alguna ubicación padre del archivo , y ya debe, con el potenciómetro conectado al A0, mover el robot:

roslaunch *launch_files manipulator.launch*

Nota para Computadora: con **ctrl + “+” zoom, y con “-”** alejas o acercas en terminal y en sublime.

Código de *manipulator.launch*:

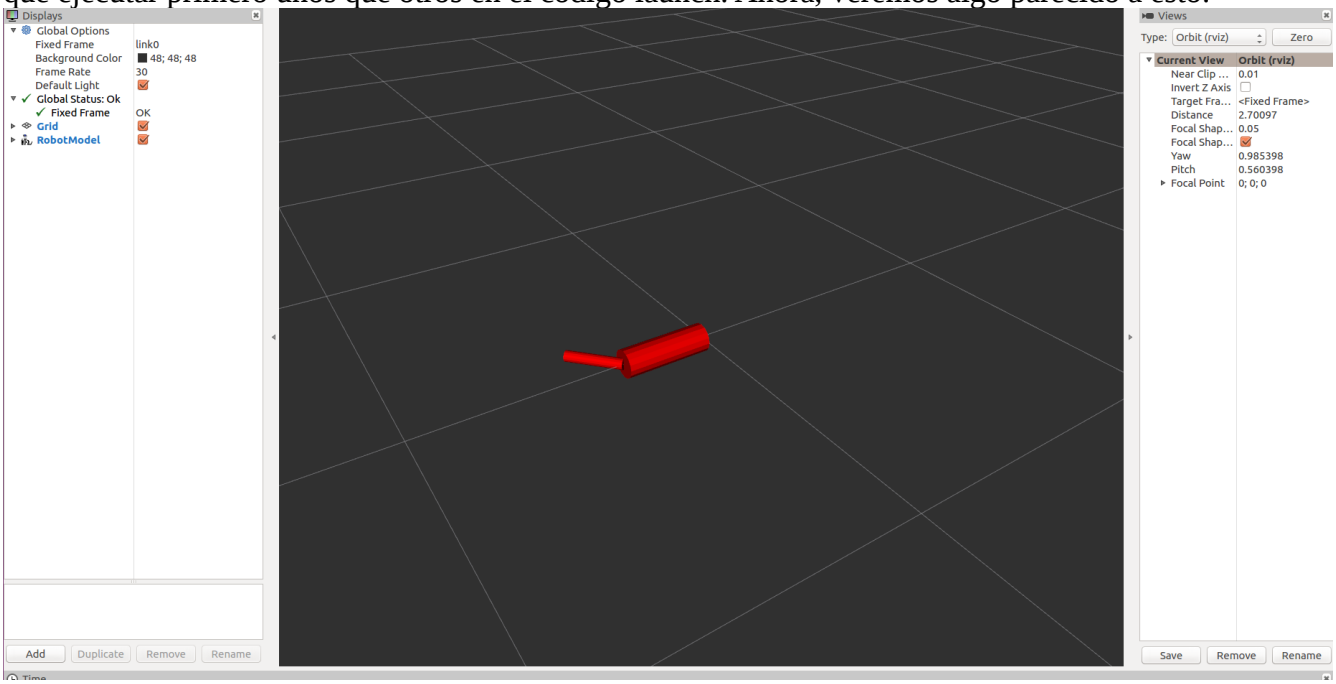
```
        <!--ESTO ES UN COMENTARIO-->

<launch>
        <!--Para definir parametros:-->
        <param name="robot_description" textfile="$(find config_files)/manipulator.urdf"/>                <!--Ruta, no conviene poner rutas absolutas, por eso conviene crear paquetes, porque abre el archivo relativo con referencia a un paquete-->
        <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" output="screen"/>                <!--robot_state_publisher se dejo por convencion--> <!--Nombre paquete ejecutable output(para ver si salio algo mal)-->
        <node name="rviz" pkg="rviz" type="rviz"/>
        <node name="pot_reader_node" pkg="pot_reader" type="pot_reader_node.py"/>        <!-- el type es el nombre del ejecutable-->
        <!--node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher"/--> <!-- Quitamos este ya que sino habrian dos nodos publicando y entraria en conflico con el pot_reader_node-->
        <param name="use_gui" value="true"/>
</launch>
```

Si uso sublime, puedo visualizar la sintaxis deXLM en VIEW.

Va a abrir todo de golpe después del roslaunch anterior, obviamente, hay que cerrar las terminales del sistema planetario, ahora con esto, dibujará una pequeña unión de eslabones que moveremos con el Arduino. Hay que tener el Arduino conectado y un potenciómetro al A0.

En Rviz, podemos cambiar en link0 o en link1, que es el centro, 0;0;0 según el Offset que se le configure en Rviz. AGREGAR en ADD RobotModel para visualizar el robot. Checar si existe algún error o advertencia y corregirla.

Recordar que todo eso del launch lo hicimos para ahorrarnos la molestia de poner todos los procesos en terminales diferentes, además de que se ejecutan sincronizadamente. Recordar que hay veces que hay que ejecutar primero unos que otros en el código launch. Ahora, veremos algo parecido a esto:



El eslabon pequeño debe girar mientras se mueva el potenciómetro.

9. ROS_MASTER

El ROS_MASTER es donde se comunica todo. En una actividad con Marco, realizamos la comunicación entre la computadora de Luis y la mía.

En el maestro:

ifconfig es para checar tu dirección ip, y es la que está wlo1 o parecido.

Vamos a editar un archivo llamado */etc/hosts* con sudo y creamos una variable de entorno llamada "ROS_MASTER_URI". Esto creo que es para cuando quieres que se conecte automáticamente.

echo $ROS_MASTER_URI

y debe salir la dirección: http://localhost:11311

y ponemos el ROS_MASTER_URI en la ip que sale en **ifconfig** computadora.
export ROS_MASTER_URI=http://192.168.20.131:11311 que es la dirección ip "192.168.20.131".
export ROS_IP=192.168.20.131
roscore

En el esclavo:
export ROS_MASTER_URI=http://192.168.20.131:11311 que es la dirección ip del maestro.
export ROS_IP= la ip del esclavo, que sale en ifconfig

Hicimos un publicador y suscriptor. Él era el publicador y yo el suscriptor.

La compu de Luis se llama Markov, la mía es Robotoshiba, se deben poner adecuadamente los nombres.

Él corrió un pequeño publicador en su computadora llamado /test:

rostopic pub -r 2 /test(es el nombre del tópico) **std_msgs/String “data: 'Hello World'”**

rostopic list

Y salieron /rosout_agg y /test, el /test ya es el publicador, y yo me suscribí:

rostopic echo /test

Y salió “Hello World”.

Ahora yo hice un publicador:

rostopic pub -r 2 /tester(es el nombre del tópico) **std_msgs/String “data: 'Hello Luis'”**

Él hizo un **rostopic echo /tester** y le salió “Hello Luis”.

Intenta siempre autoacompletar. Si no autoacompleta, intenta quitar -r 2 y luego que autoacompleto todo lo demás ponlos. El “/tester” tu lo pones, es el nombre propuesto del tópico.

Y yo ya publicaba, y Luis podía ver lo que publicaba.
Ahora hicimos la inversa, pusimos el ROS_MASTER_URI en la de Luis.

Para buscar comandos hacia atrás en la terminal: ctrl+r (recordatorio)

Luego **rostopic list** y debe salir el nombre del tópico de Luis, en este caso se llamó /test2
rostopic echo /test2

Y salió “Hello World2”

vamos a hacer uso de un sensor laser.

sudo apt-get install ros-indigo-hokuyo-node para la paqueteria del laser

ls /dev para ver los dispositivos usb

Hice un **rostopic list** y salió /scan. Me suscribí **rostopic echo /scan** y ya salía la matriz del láser que conectó Luis, me veía yo en el salón.

Abrimos un rViz. Agregamos by topic un laser.

Sale un error, que no encuentra la transformación, pero ponemos como Fixed Frame a [laser]. Y listo! Ya se dibujaba en rViz el láser.

Consejo

¿No Autoacompleta en la Terminal?

Soluciones: A veces, aunque no autoacomplete, no quiere decir que hay algo mal, hay que forzarlo poniendo los comandos manualmente para que ejecute, por ejemplo, un nodo con **rosrun**, a veces no encuentra los paquetes, aún haya compilado y todo esté en orden. Si no funciona eso, hay que abrir y cerrar la terminal

Algunos comandos de Ubuntu y ROS

cd *nombre del directorio* → ingresa al directorio especificado

cd .. → sube un nivel de la ubicación actual

mkdir [nombre de la carpeta] → crea una carpeta en la ubicación actual

rm *nombre del archivo* → elimina un archivo

rm -rf *nombre del directorio* → elimina un directorio y su contenido

catkin_create_pkg *nombre del paquete* **std_msgs roscpp rospy**

***Ctrl+Z*:** Manda un proceso a segundo plano, lo sigue ejecutando, para poderlo regresar a primer plano usar **%**
***Ctrl+C*:** Detiene el proceso totalmente

rosnode ping /NOMBRE_DEL_NODO para ver que sí esté corriendo y devuelve el tiempo de ejecución

Usamos **rostopic echo /NOMBRE_TOPICO** y lo imprime 10 veces por segundo ya que es lo que programamos en ros::Rate loop(10);

roscd te manda directo a la ubicación del paquete o carpeta del paquete, ejemplo:
roscd pot_reader/src, y te manda directo hasta allá