Butiá
Robótica Educativa

Integración de ROS a la Plataforma Butiá

Tutor: Gonzalo Tejera

Grupo 3
Ignacio Escudero
Andrés Tipoldi
Rodrigo Quinta

Contenido

1 INTRODUCCION

1.1 Robot Operating System

Robot Operating System (ROS) es una plataforma OpenSource de desarrollo de software de robótica. Esta plataforma ofrece servicios como comunicación entre procesos, instalación de dependencias y abstracción del hardware. Además, cuenta con bibliotecas de alto nivel que permiten realizar tareas complejas como manejo de sistemas de coordenadas, procesamiento de imágenes, integración de movimientos y visualización 3D.

1.2 Objetivo

El objetivo del proyecto es integrar el robot Butiá a ROS brindando la posibilidad de controlarlo usando esta plataforma.

Esta integración habilitaría la posibilidad de implementar comportamientos y arquitecturas de control de robots aprovechando las herramientas que brinda el sistema.

2 ANALISIS DE REQUERIMIENTOS

2.1 Plataforma ROS

ROS es un “Meta-Sistema Operativo” que se ofrece bajo una licencia de código abierto BSD.

Provee una forma de comunicación entre procesos por medio de mensajes en una red. Por medio de una abstracción de hardware, controladores de dispositivos, bibliotecas, visualizadores y herramientas ayuda a los desarrolladores de software a crear aplicaciones para robots.

El “runtime graph” de ROS es una red peer-to-peer de procesos poco acoplados, que usan la infraestructura de comunicación del sistema. ROS implementa diferentes formas de comunicación, incluyendo, comunicación sincrónica (RPC-style) sobre servicios, comunicación asincrónica basada en tópicos, y almacenamiento de información en un “Parameter Server”.

2.1.1 Nodos

Los nodos son procesos encargados de un comportamiento o una función específica. ROS está diseñado para ser modular; un sistema de control de un robot puede estar integrado por varios nodos.
Los nodos forman un grafo de comunicación, donde ésta se lleva a cabo utilizando los tópicos y servicios. Por lo general un nodo se encarga de una única función del robot (path planning, mover motores, etc). Esto proporciona modularidad al sistema y por lo tanto brinda beneficios tales como un mejor manejo de fallas, individualidad de los nodos, bajo acoplamiento (un nodo puede exponer una API). Para implementar un nodo el desarrollador dispone de una “Client Library”, que proporciona acceso a las funciones principales de ROS. Actualmente existen implementaciones para C++ y Python.

2.1.2 ROS Master

El ROS Master permite establecer conexiones entre los nodos ofreciendo funcionalidades de registro y búsqueda de los mismos, en un estilo similar a los servidores DNS. Como observación cabe aclarar que los nodos no se comunican mediante el Master, sino que la comunicación se da directamente entre nodos, el Master simplemente establece esta comunicación.
Este componente almacena la información para el registro y acceso a Tópicos y Servicios y provee actualizaciones a los nodos sobre el estado de los demás.
El Master provee el Parameter Server, el cual se describe a continuación.

2.1.3 Parameter Server

Es un servidor compartido que almacena parámetros. Éstos pueden ser registrados y consultados por los nodos en tiempo de ejecución. Su principal uso es el almacenamiento de parámetros de configuración.

2.1.4 Mensajes

Los nodos se comunican a través de mensajes. Éstos consisten en una estructura de datos con atributos tipados, los que contendrán datos que sean de interés para el receptor.
Aunque existen varios tipos estándar de mensajes ya definidos, también se pueden definir nuevos por medio de un archivo de texto plano. Los mensajes pueden tener una estructura anidada; un mensaje puede estar integrado por otros mensajes.

2.1.5 Tópicos

Proporcionan un vía de transporte con semántica publicación / suscripción. Cuando algún nodo envía un mensaje a un tópico, los nodos suscriptos a éste reciben una copia del mismo. Por lo general los suscriptores y los que efectúan la publicación no se conocen entre ellos. Los tópicos conforman un bus de mensajes y cualquiera se puede conectar para enviar o recibir. Permiten una comunicación asincrónica entre varios nodos (many-to-many). Están orientados a la comunicación unidireccional.

2.1.6 Servicios

Los nodos pueden publicar Servicios. Éstos proporcionan una forma de comunicación sincrónica, por medio de la cual un nodo “consumidor” invoca un servicio ofrecido por otro mediante el envío de un mensaje de solicitud y este último brinda una respuesta en forma de otro mensaje.

Esta es una forma de comunicación uno-a-uno en forma de RPC (llamada a procedimiento remoto).

Quien implementa el nodo que expone un servicio, define también los tipos de los mensajes de solicitud y respuesta

Una invocación a un servicio es bloqueante, es decir que el “consumidor” detiene su ejecución hasta recibir la respuesta correspondiente.

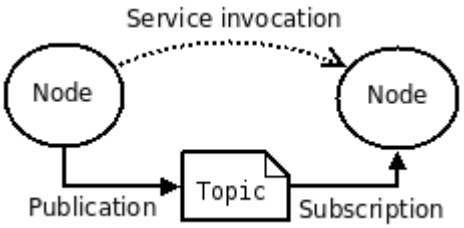


Figure 1 - Comunicación entre nodos

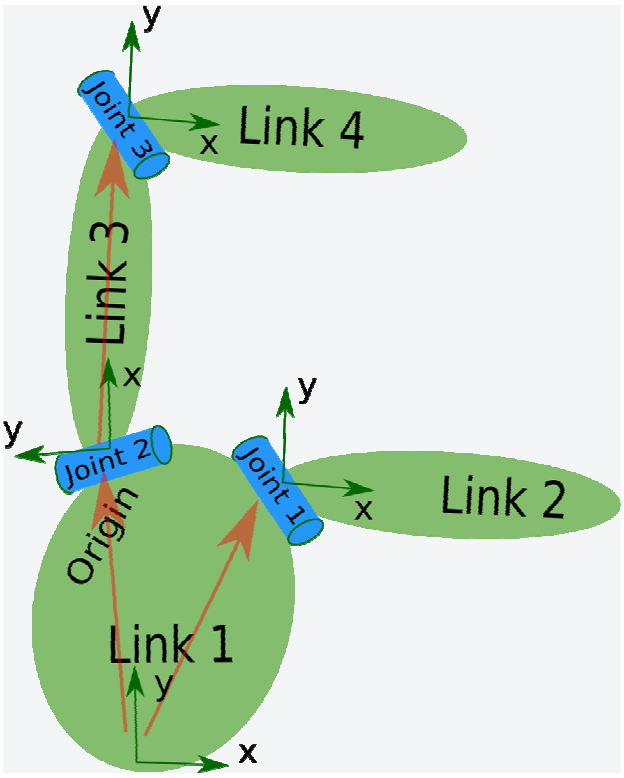
2.2 Modelo

2.2.1 URDF

El modelo URDF (Unified Robot Descriptor Format) ofrece una descripción del robot. Describe las partes que lo componen y la dinámica de éstas, además ofrece una representación visual y el modelado de colisiones de estas componentes.
Este modelo es una especificación en xml. Hoy en día solamente soporta tres componentes, el Robot, Links y Joints. La estructura general de un archivo urdf seria como la siguiente

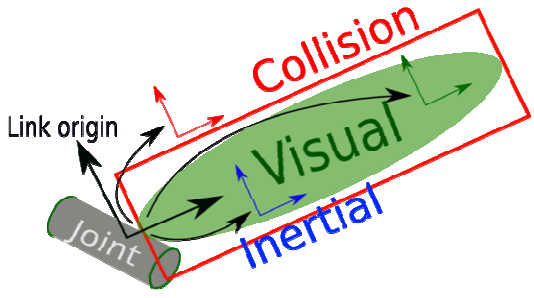
```
<robot name="pr2">
  <link> ... </link>
  <link> ... </link>
  <link> ... </link>

  <joint>  ....  </joint>
  <joint>  ....  </joint>
  <joint>  ....  </joint>
</robot>
```



El URDF representa el modelo completo del robot -de ahí su nombre- es un conjunto de links y joints y se identifica por un nombre.
Los links representan objetos rígidos con inercia, y describen las características de cada una de las partes del robot. Las componentes de un link, mediante las cuales se realiza esta descripción son:

- *Inertial:* guarda la información relevante para la inercia del objeto, origen y masa del mismo.
- *Visual:* representa características visuales del objeto, como la forma geométrica y el color. Son utilizadas por los simuladores para ofrecer una imagen virtual del robot.
- *Collision:* define las propiedades tales como los bounding boxes para poder resolver las colisiones del robot.



Los joints son utilizados para describir las uniones de los diferentes links, es decir, representan las uniones de las partes del robot, describen su dinámica y su cinemática. Además especifican los límites de seguridad (Safety Limits), que son los límites que soporta esa unión sin romperse.

Existen 7 tipos de joints, basados en los movimientos que pueden realizar:

- Revolute
- Fixed
- Planar
- Continuous
- Floating
- Prismatic

Entre sus elementos además de los links que une (a los cuales identifica como parent y child) se encuentran elementos para definir el eje, la calibración, la dinámica y los límites.

2.2.2 Extensiones

Además de los elementos nombrados anteriormente, el URDF tiene otros que son utilizados como extensiones del mismo. Éstos son llamados Transmisión y Gazebo.
Transmission es un elemento recientemente agregado que es utilizado para describir la relación entre los joints y los actuadores del robot (por ejemplo radios de giro, etc). Este elemento tiene tipo, pero de momento solo existe el tipo “Simple Transmission” esto se debe a que como señalábamos anteriormente, el elemento fue incluido en recientes versiones de ROS.
La otra extensión disponible, Gazebo es utilizada por el simulador de igual nombre que nos permite visualizar el robot en 3D y simular sus movimientos.

2.2.3 Otros Componentes

Existen otros dos elementos que presentamos en esta sección debido a que aún no han sido utilizados en la práctica. Ellos son Sensor y Model State. El primero se encuentra en el dom de ROS pero el proyecto fue abandonado por el grupo de desarrollo, de todas formas se encuentra disponible para que alguien continúe con él. Este es el encargado de describir características de los sensores conectados al robot, como cámara, grises, etc.
Model State sin embargo es un proyecto que se encuentra en desarrollo y apunta a representar al modelo en el tiempo, aún no ha sido puesto en práctica. Esto hoy en día es realizado a través del SRDF (Semantic Robot Description File), el cual permite mostrar información de un conjunto de joints para un momento dado.

2.2.4 Xacro

ROS utiliza un macro lenguaje xml denominado Xacro para poder implementar archivos URDF más compactos y de fácil lectura. Xacro ofrece un conjunto de macros que hacen más fácil la especificación de los archivos xml, los cuales pueden llegar a ser demasiado largos, en especial en lo que refiere a geometría y cálculos matemáticos.
Los archivos Xacro son transformados en archivos URDF mediante un comando de ROS:

```
rosrun xacro xacro.py model.xacro > model.urdf
```

2.2.5 Herramientas

Para el manejo y visualización del modelo del robot se cuenta con 2 herramientas: Rviz y Graphiz. La primera de ellas permite ver el modelo del robot en 3D de manera estática. Como editor permite cambiar cierta información como los colores, tamaños o posiciones de ciertas componentes. Por otro lado Graphiz permite obtener una archivo pdf que contiene una representación en forma de árbol del modelo, donde los nodos son los links y las aristas son los joints.

Además se cuenta con Gazebo, un simulador 3D que permite ver al robot en movimiento. Este es una extensión para ROS.

2.2.6 Publicar URDF y Modelo del Mundo (tf Transform)

Luego de tener un modelo del robot éste debe ser publicado para que los nodos puedan consultar la información que ofrece. Esto se hace mediante el package robot_state_publisher, éste puede ser usado tanto como biblioteca o como un nodo de ROS. El robot_state_publisher recibe un archivo URDF y publica un modelo 3D del robot que puede ser consultado por todas las componentes que utilicen el package tf.
tf mantiene un conjunto de marcos que almacenan las posiciones de las componentes del robot, es quien mantiene el modelo del mundo. tf Transform cuenta con dos tareas principales que son escuchar y publicar transformaciones. La primera de éstas almacena todos los marcos de coordenadas que recibe y a su vez realiza consultas por cambios específicos entre los marcos. La segunda tarea, publicar, se encarga de publicar los marcos de coordenadas de las diferentes partes del robot.

3 DESCRIPCION DE LA SOLUCION

3.1 Componentes

En ROS cada tipo de robot es modelado mediante un “Robot Stack”. Para integrar el robot Butiá a esta plataforma es necesario implementar un Stack específico.
Éstas estructuras están integradas por cuatro componentes principales:

- Driver:
  Consiste en el controlador para el robot. Es el componente que ejecuta los comandos específicos de control, quien “sabe hablar” con el mismo.
  Para el caso del robot Butiá se consideró la utilización de un PyBot Client cumpliendo la función de driver, en comunicación con un PyBot Server.

- Nodo:
  Es un nodo de ROS que funciona como “envoltura” (wrapper) para el Driver. Implementa una API de ROS exponiendo las formas de comunicación con el robot (servicios, tópicos).
  Éste puede comunicarse con otros nodos.

- Modelo:
  Consiste en el URDF del robot, que provee una descripción física del mismo; ubicación de sensores, actuadores.
  En el caso del robot Butiá, dado que su estructura no es estática (los sensores son móviles pueden colocarse en ubicaciones diferentes), una primera aproximación podría consistir en un modelo fijo con una configuración típica.

- Inicialización:
  Contiene los scripts de startup necesarios. Aquí se invocan funciones de inicialización.

3.2 Implementación

Se implementaron dos soluciones para el manejo del robot Butiá desde la Plataforma ROS.

Como se mencionó anteriormente ROS soporta principalmente dos formas de comunicación entre nodos. Usando Tópicos, o por medio de servicios. Mientras que los primeros permiten una comunicación asíncrona, los segundos aseguran una comunicación uno a uno.

Al analizar las necesidades del robot Butiá, y las formas de comunicación con el ROS, se decidió implementar ambas formas de comunicación. Esto permite tener una integración más versátil, permitiendo a los desarrolladores de aplicaciones para ROS/Butiá, utilizar cualquiera de las dos soluciones según les sea mas conveniente.

3.2.1 Implementación de Servicios

Para la implementación de servicios se creó un script python que expone los servicios existentes definidos en el pybot. El butia_ros_server.py, provee estos servicios a cualquier nodo que se desee utilizar el robot Butiá.

Cada vez que un consumidor invoque un servicio provisto por el butia_ros_server.py, éste invocara el servicio deseado por medio de una instancia de pybot_client.

A continuación se muestra el código del butia_ros_server. Encargado de exponer los servicios del pybot.

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

import roslib; roslib.load_manifest('Butia')

from Butia.srv import *

import rospy

from pybot import pybot_client


def handle_get_gray(req):

    robot = pybot_client.robot()

    value = robot.getGray(req.a)

    print "Returning [robot.getGray(%s):%s]"%(req.a, value)

    return ButiaGetGrayResponse(value)


def handle_get_distance(req):

    robot = pybot_client.robot()
```

```
        value = robot.getDistance(req.a)
        print "Returning [robot.getDistance(%s):%s]"%(req.a, value)
        return ButiaGetDistanceResponse(value)

    def handle_get_button(req):
        robot = pybot_client.robot()
        value = robot.getButton(req.a)
        print "Returning [robot.getButton(%s):%s]"%(req.a, value)
        return ButiaGetButtonResponse(value)

    def handle_set_2_motor_speed(req):
        robot = pybot_client.robot()
        robot.set2MotorSpeed(req.a, req.b, req.c, req.d)
        print "Setting Motor Speed to: %s, %s, %s, %s "%(req.a, req.b, req.c, req.d)
        return ButiaSet2MotorSpeedResponse()

    def butia_ros_server():
        rospy.init_node('butia_ros_server')
        s = rospy.Service('butia_get_gray', ButiaGetGray, handle_get_gray)
        g = rospy.Service('butia_get_distance', ButiaGetDistance, handle_get_distance)
        y = rospy.Service('butia_get_button', ButiaGetButton, handle_get_button)

        m = rospy.Service('butia_set_2_motor_speed', ButiaSet2MotorSpeed,
    handle_set_2_motor_speed)

        print "Butia ROS Server Ready."
        rospy.spin()

    if __name__ == "__main__":
        butia_ros_server()
```

Para demostrar cómo se consumirían los servicios, también se implementó un cliente que llama a los servicios expuestos por el server. El cliente acepta como parámetro los distintos sensores disponibles. Para ejecutarlo se debe correr la siguiente línea de comando.

```
$ rosrun Butia butia_ros_client.py get_button 2
```

Una vez invocado, el programa llama al servicio expuesto por el butia_ros_server.py especificado (get_button en el caso de arriba) y muestra en consola el resultado de la invocación al servicio.

El siguiente código hace el llamado al servicio del server. En el caso de abajo se está llamando al servicio que expone el sensor de escala de grises.

```
if cmd == "get_gray":

        rospy.wait_for_service('butia_get_gray')

        service = rospy.ServiceProxy('butia_get_gray', ButiaGetGray)
```

Además de los servicios que brindan las lecturas de los sensores (get_button, get_gray, get_distance) se implementó otro para setear la velocidad de los motores. Éste puede ser invocado utilizando el cliente mediante el siguiente comando:

```
$ rosrun Butia butia_ros_client.py set_2_motor_speed 1 400 0 400
```

3.2.2 Implementación de Tópicos

Al ser los servicios bloqueantes y uno a uno, surgió la necesidad de implementar también un modulo que permitiera generar tópicos asíncronos que expusieran los valores de los distintos sensores del robot Butiá.

El butia_ros_server_topics.py permite generar tópicos específicos por sensores. El mismo recibe como parámetro cuatro elementos:

Nombre del tópico: Nombre que se le asignará al tópico

Velocidad (hertz): Velocidad con la que el valor del sensor será expuesto en tópico.

Comando: Sensor que nos interesa exponer en el tópico.

Numero de Sensor: Numero de sensor a ser consultado.

El programa toma los parámetros anteriores, y genera un tópico. Luego según la velocidad establecida en hertz, el script consulta el sensor especifico del robot (mediante el pybot_client) y lo publica en el tópico.

De esta forma, cualquier nodo, que quiera estar al tanto del valor de los sensores del robot, puede simplemente consumir los mensajes de los tópicos.

El siguiente código es el encargado de generar los tópicos según los sensores que se quieran consultar:

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('Butia')
from Butia.srv import *
import rospy
from pybot import pybot_client
from std_msgs.msg import String

def butia_ros_server_topics(topic,vel,cmd,a):
    pub = rospy.Publisher(topic, String)
    rospy.init_node('butia_ros_server_topics', anonymous=True)
    r = rospy.Rate(vel) # 10 = 10hz

    while not rospy.is_shutdown():
        robot = pybot_client.robot()

        if cmd == "get_gray":
            value = robot.getGray(a)
        elif cmd == "get_distance":
            value = robot.getDistance(a)
        elif cmd == "get_button":
            value = robot.getButton(a)
        else:
            print "Invalid Command: %s"%cmd
            sys.exit(1)

        print "Publishing [%s(%s):%s]"%(cmd, a, value)
        pub.publish("%s"%value)
        r.sleep()
```

```
def usage():

    return "usasge:\nbutia_ros_server_topics topic_name rate cmd a\ncmd: get_gray
| get_distance | get_button"


if __name__ == "__main__":

    if len(sys.argv) == 5:

        topic = sys.argv[1]

        vel = int(sys.argv[2])

        cmd = sys.argv[3]

        a = int(sys.argv[4])

        print " %s(%s)"%(cmd,a)

        butia_ros_server_topics(topic,vel,cmd,a)

    else:

        print usage()

        sys.exit(1)
```

3.2.3 Comparación entre el uso de Tópicos y Servicios

Las siguientes son algunas consideraciones a tener en cuenta al momento de usar tópico o servicios.

- Siendo que los Tópicos funcionan de forma asincrónica, el nodo consumidor, no se bloquea para esperar una respuesta. El valor del sensor siempre estará disponible para el cliente. El cliente puede escuchar el tópico y modificar una variable local, permitiendo al programa tener acceso a la información necesaria de forma inmediata.

- Dependiendo de la velocidad previamente estipulada de consulta al tópico, la información disponible puede no estar actualizada. Quiere decir que si un proceso precisa tener la información exacta, tendrá que utilizar un servicio, o definir el tópico con un intervalo de consulta muy pequeño, para minimizar la diferencia.

- Como el valor es expuesto en un tópico, el estado del sensor se encontrara disponible para todos los subscriptores al mismo tiempo. De otra forma, con el uso de servicios, todos los clientes interesados en saber el estado de un sensor tendrían que hacer llamadas independientes al servicio.

- Con el uso de servicios, uno se puede asegurar de que no se están haciendo llamadas innecesarias al servidor. Esto permite minimizar la posible sobrecarga de la conexión debido a las llamadas iterativas del topic_server al pybot.

4 EXPERIMENTOS Y PRUEBAS

4.1 Ejecución de la solución.

Para ejecutar una prueba de la solución presentada se deben ejecutar los siguientes pasos.

Se deben levantar 5 consolas y ejecutar los comandos a continuación:

1. **Consola 1** - Levantar el ROS core.

   $ roscore

2. **Consola 2** - Levantar el pybot_server

   $ python pybot_server.py

3. **Consola 3** - Levantar el server que expone los servicios

   $ rosrun Butia butia_ros_server.py

4. **Consola 4** - Levantar el client que consume los servicios (en este caso get_button). Verificar que el resultado se muestre en consola.

   $ rosrun Butia butia_ros_client.py get_button 2

5. **Consola 5** - Levantar el server para generar un topico. Verificar que los valores se muestren en consola con el intervalo de tiempo especificado

   $ rosrun Butia butia_ros_server_topics.py Button 10 get_button 2

4.2 Prueba Local

En caso de no contar con un robot Butiá, que pueda atender los pedidos del pybot server, se puede ejecutar este mismo en modo "chotox".

```
$ python pybot_server.py chotox
```

Al ejecutarlo en este modo, el pybot server simula una conexión con un robot butiá, y devuelve valores a las llamadas efectuadas desde el client. Esto permite ejecutar pruebas sobre ROS, sin tener que disponer de un robot Butiá.

5 CONCLUSIONES Y TRABAJO A FUTURO

Mediante los componentes desarrollados es posible controlar el robot Butiá utilizando la plataforma ROS, accediendo a las lecturas de sus sensores de Grises, Distancia y Botón, ya sea en forma sincrónica mediante la invocación de servicios o asincrónica mediante la creación y suscripción a tópicos, así como controlar sus motores seteando su velocidad utilizando el servicio implementado con tal fin.
Con este resultado se alcanza el objetivo planteado haciendo posible la implementación de comportamientos y arquitecturas de control de robots aprovechando las herramientas que brinda el sistema.

Considerando el trabajo realizado como una primera etapa de investigación, con productos generados que permiten controlar el robot, mencionamos a continuación posibles líneas de trabajo a futuro:

- Implementación de servicios y tópicos para las lecturas otros sensores. Ésta puede ser realizada de manera relativamente sencilla, tomando como base lo ya implementado.

- Modelo URDF. En esta etapa se experimentó con la creación del modelo utilizando la herramienta xacro e investigando modelos existentes. Sin embargo la creación de un modelo preciso para el robot Butiá queda planteada como trabajo a futuro.

- Investigación en torno a otros sensores, como por ejemplo una cámara, cuyas entradas no se obtienen directamente de PyBot Server. En este punto es posible aprovechar los controladores genéricos brindados por ROS.

- Butia Stack. Luego de contar con el modelo, el próximo paso consistiría en generar el Componente de Inicialización (bringup), en el cual se podría iniciar el PyBot Server por ejemplo y realizar las tareas de arranque necesarias. A continuación los cuatro componentes (driver, nodo, modelo, inicialización) se “empaquetarían” para formar el Robot Stack del Butiá.

6 REFERENCIAS

Wiki Proyecto Butiá
http://www.fing.edu.uy/inco/proyectos/butia/mediawiki/index.php/P%C3%A1gina_principal

Página oficial de ROS
http://www.ros.org/

Wiki oficial de ROS - Documentación / Tutoriales
http://wiki.ros.org/

Guía de instalación ROS en Ubuntu
http://wiki.ros.org/hydro/Installation/Ubuntu

Ejemplo Robot Lego NXT
http://wiki.ros.org/Robots/NXT

Presentación Introductoria a la Plataforma ROS (Benjamin Cohen)
https://alliance.seas.upenn.edu/~meam620/wiki/index.php?n=Roslab.ROSTutorials?action=download&upname=meam620-IntroductionToROS.pdf

ROS Cheat Sheet
http://download.ros.org/downloads/ROScheatsheet.pdf

Modelo URDF
http://wiki.ros.org/urdf

Modelado para ROS tutoriales
http://www.instructables.com/id/Getting-Started-with-ROS-Robotic-Operating-Syste/?ALLSTEPS

PyBot
http://www.fing.edu.uy/inco/proyectos/butia/mediawiki/index.php/PyBot