

ROS (Robot Operating System)



Que es ROS?

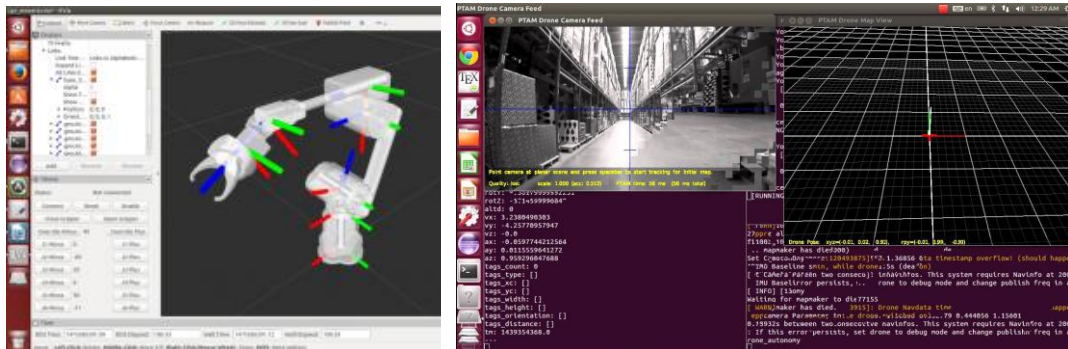
ROS (Robot Operating System) es un conjunto de herramientas software y librerías que agilizan la construcción de sistemas robóticos. ROS provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, paso de mensajes entre procesos, entre otros. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, entre otros. Se encuentra orientado para un sistema UNIX (Ubuntu (Linux)) aunque también se está adaptando a otros sistemas operativos. ROS es software libre bajo términos de licencia BSD. Esta licencia permite libertad para uso comercial e investigación.



Por qué usar ROS?

Aprender a utilizar un entorno de trabajo demanda tiempo y esfuerzo, por esto resulta necesario remarcar que ventajas nos trae el uso de ROS, y en qué casos conviene su utilización. ROS fue y es pensado para ser un conjunto de herramientas aplicables en entornos robóticos (no restringido a un tipo de robot en particular) con el objetivo de no reinventar la rueda. Conociendo esto existen algunos puntos de la implementación de ROS que lo hacen atractivo:

- **Sistema distribuido:** la estructura de ROS permite la comunicación entre múltiples procesos que pueden o no trabajar en la misma computadora. Por ejemplo, en robots complejos puede existir un proceso que maneje la información de un sensor en particular, otro que se ocupe de la odometría, otro del control de velocidad, etc. Esta segmentación del trabajo por un lado facilitará la depuración de la programación, y a su vez como todos estos procesos no necesariamente se encontrarán en la misma PC, nos habilitan a realizar tareas de colaboración o servicios a distancia.
- **Reutilización de código:** el crecimiento de aplicaciones robóticas conlleva a un crecimiento aun mayor de algoritmos aplicados en ellas. Por ejemplo, navegación, evitación de obstáculos, mapeo, etc. ROS define un estándar para la estructura de estos procesos y la comunicación entre ellos. Al realizar esto permite la exportación de un código de un sistema a otro idealmente sin modificaciones.
- **Rápida prueba de conceptos:** uno de los objetivos del sistema y su estandarización de código es el de poder desarrollar por un lado un software de alto nivel (del que se preocupa más ROS) y por otro el código de bajo nivel y hardware. Esta separación nos permitirá mediante herramientas de simulación probar nuestros códigos (reemplazando el bajo nivel y hardware) sin necesidad de utilizar con un robot real.



- **Almacenamiento de datos:** ROS provee servicios de captura de datos y visualización, los cuales nos permitirán saber que sucedió (bien o mal) en un ensayo experimental llevando estos datos a un entorno simulado, pudiendo así observar la evolución de cada variable de nuestro sistema.

Como comenzar?

Para comenzar debemos tener instalado nuestro sistema ROS, si vamos a la página oficial <http://www.ros.org/> encontraremos distintas versiones del sistema, fundamentalmente cambian una de otras en servicios, paquetes incorporados, y en las versiones más viejas estructura de mensajes. En este curso trabajaremos con la versión Kinetic, la razón de esto es porque por el momento es una de las que mantiene mayor número de compatibilidad con los paquetes disponibles en la web.



Para la instalación de ROS, contando con una versión de UBUNTU compatible funcionando, solo es cuestión de seguir los pasos listados en:

<http://wiki.ros.org/kinetic/Installation/Ubuntu>

(trabajaremos con la versión Desktop-Full, ya que contiene algunas utilidades graficas que nos serán útiles).

- **Primer ejemplo:**

Antes de comenzar con la estructura de ROS, realizaremos un mínimo ejemplo usando el simulador turtlesim, el objetivo aquí es comprobar la buena instalación de ROS, conocer el simulador y obtener una idea inicial de la forma de trabajo de ROS.

Abriendo tres consolas distintas ejecutamos una de las siguientes líneas de comando en cada una:

```
roscore  
roslaunch turtlesim turtlesim_node  
roslaunch turtlesim turtle_teleop_key
```

La primera línea iniciará el master, la segunda ejecutara el simulador y la tercera iniciará un programa (nodo) que se comunicará con el simulador a través del cual usando las flechas de teclado podremos mover la Tortuga del simulador:





Con este ejemplo podemos abordar los siguientes temas:

Paquetes (packages):

Todo el software de ROS está organizado en paquetes. Un paquete de ROS es una colección coherente de archivos, que en general incluyen ejecutables y archivos de soporte que sirven a un propósito específico. En el ejemplo anterior utilizamos dos ejecutables `turtlesim_node` y `turtle_teleop_key`, ambos pertenecientes al paquete `turtlesim`.

ROS incluye una serie de herramientas para trabajar con los paquetes:

rospack list	Nos da la lista de todos los paquetes ROS instalados
rospack find package-name	Nos servirá para encontrar el directorio de un paquete en particular
rosls package-name	Para investigar el contenido de un paquete
roscd package-name	Para ir al directorio del paquete

Cada paquete es definido por un manifiesto llamado *package.xml*. Este archivo define algunos detalles sobre el paquete, incluyendo su nombre, versión, quien lo mantiene, y sus dependencias. El directorio que contiene *package.xml* es llamado el directorio del paquete, allí encontraremos la mayor parte de los archivos del paquete. (En realidad cualquier directorio que ROS encuentre con un archivo llamado *package.xml* lo tomará como directorio de un paquete).



- A lo largo del apunte iremos nombrando herramientas de ROS en forma genérica, de forma que si se generan dudas sea más fácil de ser googleadas para aclarar su funcionamiento. En la tabla anterior por ejemplo cuando nos referimos a *package-name*, al ejecutar esta línea deberemos cambiar esto por el nombre del paquete de interés.

El maestro (master):

Uno de los objetivos principales de ROS es permitir a sus programadores diseñar una colección de programas pequeños, mayormente independientes, llamados **nodos** que se ejecutan al mismo tiempo. Para que esto funcione, cada nodo debe ser capaz de comunicarse con los otros. La parte de ROS que facilita esta comunicación es llamada ROSmaster. Iniciamos el master con el comando:

```
roscore
```




```
alumno@alumno: ~  
alumno@alumno: ~ 61x7  
alumno@alumno:~$ rostopic list  
/rosout  
/teleop_turtle  
/turtlesim  
alumno@alumno:~$
```

/rosout es un nodo iniciado automáticamente al iniciar el master, /turtlesim será el nodo del simulador y /teleop_turtle el nodo de comando. Resulta interesante ver que los nombres de los nodos no necesariamente coinciden con el nombre de su correspondiente ejecutable.



- Si quisiéramos darle un nombre en particular a un nodo podríamos utilizar:

roslaunch package-name executable-name __name:=node-name

En donde node-name será el nombre que le quisiéramos dar.

rostopic info node-name	Sirve para obtener información de un nodo. Este comando nos devolverá la lista de tópicos en los cuales el nodo está suscrito y en cual actúa como publicador, los servicios ofrecidos por el nodo, el identificador de proceso de Linux (PID) y un resumen de las conexiones que ha hecho con otros nodos.
rostopic kill node-name	Sirve para detener un nodo

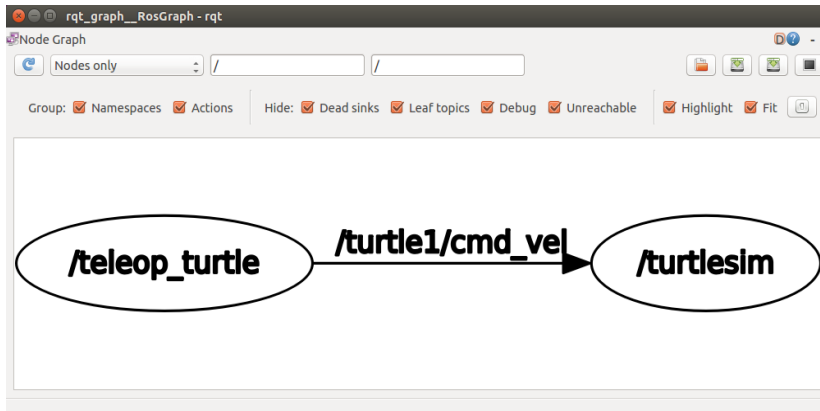
Tópicos y mensajes:

El principal mecanismo que posee ROS para comunicar nodos es el uso de mensajes. Los mensajes en ROS están organizados en tópicos (**topics**). La idea es que un nodo que desee compartir información publicará mensajes en un apropiado tópico o tópicos; un nodo que desee recibir información se suscribirá al tópico o tópicos en los que se encuentre la información buscada. Una vez realizada la suscripción de ambos, el mensaje al ser publicado es enviado directamente al suscriptor/es.

rqt_graph es una herramienta de ROS que permite ver fácil y rápidamente la relación entre nodos y tópicos, si ejecutamos este comando:

rqt_graph

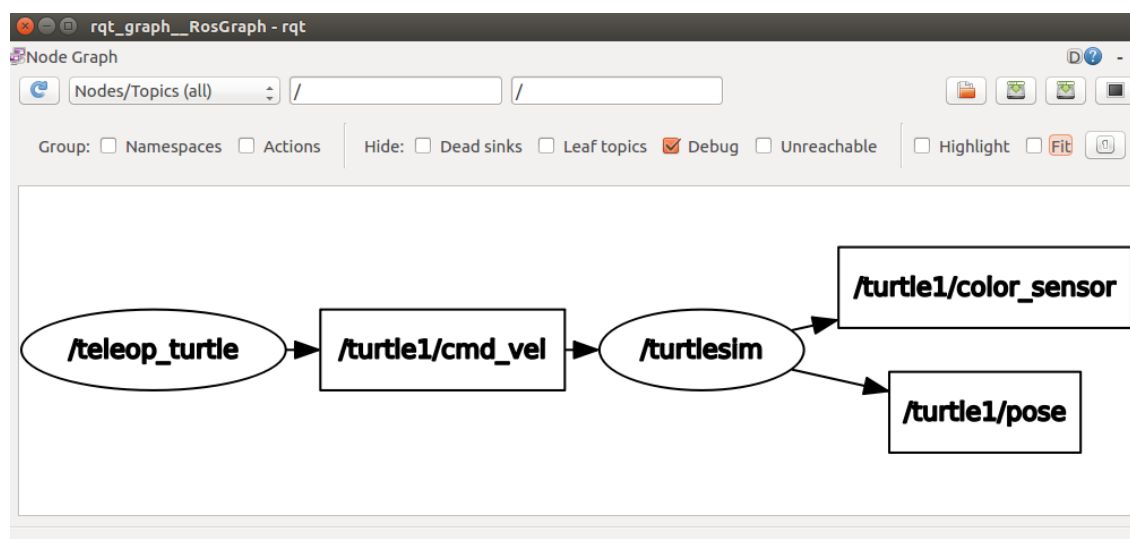
con el ejemplo del turtlesim nos devolverá:



En este gráfico, los óvalos representan nodos, y las flechas representan tópicos indicando la relación publicador-suscriptor. En el gráfico vemos que un nodo llamado /teleop_turtle publica mensajes en un tópico llamado /turtle1/cmd_vel, y que un nodo llamado /turtlesim está suscrito a estos mensajes.

Tal vez podría llamar la atención que el nodo rosout no aparece en este gráfico, esto es debido a que por defecto rqt_graph esconde los nodos que solo existen para depurar. Es posible desactivar esta opción desmarcando la casilla Debug.

Además, rqt_graph permite visualizar este grafico de distintas maneras las cuales podemos probar cambiando las casillas marcadas o los menús que aparecen en su ventana. Una opción conveniente es: **Nodes/Topics(all)** y desmarcar todas las casillas excepto **Debug**:



En esta configuración podemos ver que el nodo turtlesim además de estar suscrito a comandos de velocidad también publica su posición y datos sobre su sensor simulado de detección de color.

Notar que:

- Al simulador no le interesa quien publique los comandos de velocidad, por lo que cualquier nodo publicador que publique en el tópico puede controlar la tortuga.
- Por otro lado al nodo de tele operación no le interesa (ni sabe) quien está suscrito a sus mensajes, por lo que cualquier nodo suscrito a ese tópico podrá recibir sus mensajes.

ROS incluye una serie de herramientas para trabajar con tópicos y mensajes:

rostopic list	Nos devuelve una lista de los tópicos activos.
rostopic echo topic-name	En la terminal en que se ejecute nos mostrara los mensajes que se publiquen a partir de su ejecución.
rostopic hz topic-name	Nos dice la velocidad con que los mensajes son publicados en el tópico.
rostopic bw topic-name	Nos dice el ancho de banda consumido por los mensajes publicados en el tópico.
rostopic info topic-name	Sirve para obtener información del tópico.

Siguiendo con el ejemplo de turtlesim, si obtenemos información del tópico de color usando:

```
rostopic info /turtle1/color_sensor
```



```
alumno@alumno: ~  
alumno@alumno: ~ 77x17  
alumno@alumno:~$ rostopic info /turtle1/color_sensor  
Type: turtlesim/Color  
  
Publishers:  
* /turtlesim (http://alumno:37289/)  
  
Subscribers: None  
  
alumno@alumno:~$
```

De lo anterior centraremos nuestra atención en la primera línea, la cual nos muestra el tipo de mensaje en ese tópic. En este caso para **/turtle1/color_sensor**, el tipo de mensaje es **turtlesim/Color**. Aquí por tipo entendemos tipo de dato que compone el mensaje. El tipo de mensaje define como se organiza cada mensaje, y que tipo de variables o datos encontramos en él.

Como todo en ROS, cada mensaje pertenece a un paquete específico, el nombre de los mensajes siempre contiene una barra, la parte previa a esta indica el paquete del cual es parte. Esta configuración es útil para entender el contenido del mensaje y evitar colisiones con mensajes del mismo nombre, pero de paquetes distintos.

rosmmsg show message-type-name	Nos dice las características de un mensaje en particular
---------------------------------------	--

Por ejemplo, si deseamos ver el detalle del tipo turtlesim/Color

```
rosmmsg show turtlesim/Color
```

Obtenemos:

```
alumno@alumno: ~  
alumno@alumno: ~ 77x17  
alumno@alumno:~$ rosmmsg show turtlesim/Color  
uint8 r  
uint8 g  
uint8 b  
  
alumno@alumno:~$
```



Este resultado nos muestra la estructura del tipo de mensaje en particular. Aquí una serie de unsigned int con los nombres r g b. Estos representan la intensidad de los colores rojo verde y azul del pixel debajo de la tortuga del simulador.

Otro ejemplo de tipo de mensaje es **geometry_msgs/Twist**. Este es el tipo de mensaje del tópico /turtle1/cmd_vel:

```
alumno@alumno: ~  
alumno@alumno: ~ 77x17  
alumno@alumno:~$ rosmmsg show geometry_msgs/Twist  
geometry_msgs/Vector3 linear  
float64 x  
float64 y  
float64 z  
geometry_msgs/Vector3 angular  
float64 x  
float64 y  
float64 z  
alumno@alumno:~$
```

En este caso el mensaje está compuesto por los campos linear (velocidad lineal) y angular (velocidad angular). Vemos que ambos son datos de tipo **geometry_msgs/Vector3**. La indentación muestra que los campos llamados x,y,z son miembros de los dos campos de mayor nivel. Cada uno de estos campos por otra parte son floats de 64 bits.

Para publicar mensajes por la línea de comando podemos usar el siguiente comando:

```
rostopic pub -r rate-in-hz topic-name message-type message-content
```

Por ejemplo, si deseamos publicar una velocidad para que la Tortuga del simulador se mueva podremos usar:

```
rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist '[2, 0, 0]' '[0, 0, 0]'
```

Esto publica una velocidad lineal de 2m/s a una frecuencia de 1 Hz.

Es posible realizar una única publicación cambiando -r por -1 y omitiendo la frecuencia de repetición:



```
rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist '[2, 0, 0]''[0, 0, 0]'
```

Actividad



Mate los nodos que pudieran estar corriendo y al master si estaba activo. Luego reinicie el master y corra dos nodos `turtlesim_node` y dos nodos `turtle_teleop_key`. Para lo anterior primero piense las siguientes cuestiones:

- 1) ¿ROS acepta varios nodos con el mismo nombre?
- 2) ¿Qué tortuga esperamos se mueva con cada teleop?
- 3) ¿Cómo espera sea el grafico de nodos y tópicos?
- 4) ¿Cómo definiría la comunicación por tópicos?
- 5) ¿Cómo diría que es el acoplamiento entre nodos?



○ Hasta ahora vimos que al ejecutar un nodo necesariamente ocupamos un terminal, mas adelante veremos como ejecutar varios nodos desde un único terminal usando los archivos `launchfile`. Como atajo rápido ahora podemos usar cualquier emulador de terminales que nos permita un orden adecuado de las terminales abiertas, por ejemplo `terminator`. Para instalarlo use: `sudo apt-get install terminator`.



- **Primer programa: hola mundo!**

Hasta el momento hemos utilizado código pre-existente en ROS, lo que realizaremos a continuación es escribir un pequeño código que realizara el conocido *hello world!* con el que todos hemos comenzado a programar.

Crear un workspace (espacio de trabajo) y un paquete

Como hemos visto toda la programación de ROS se basa en paquetes, los paquetes que creamos se encontraran dentro de un espacio de trabajo. Por lo tanto, para comenzar a trabajar en ROS será necesario primero crear nuestro espacio de trabajo, para luego crear nuestros paquetes dentro de el mismo.

A la hora de crear paquetes ROS nos ofrece una herramienta llamada **catkin**, esta nos permitirá la rápida síntesis de paquetes y en una forma ordenada, un espacio de trabajo en general tendrá la forma:

```
workspace_folder/  -- directorio espacio de trabajo
  src/             -- carpeta source donde se encontraran nuestro paquetes
    CMakeLists.txt -- 'Toplevel' CMake file, generado por catkin
  package_1/
    CMakeLists.txt -- CMakeLists.txt file del package_1
    package.xml    -- Manifiesto del package_1
    src/           -- carpeta source donde guardaremos nuestros codigos
  ...
  package_n/
    CMakeLists.txt -- CMakeLists.txt file del package_n
    package.xml    -- Manifiesto del package_n
    src/           -- carpeta source de el paquete_n
```

Nuestro espacio de trabajo puede estar localizado en cualquier parte del disco, aquí por ejemplo llamaremos a la carpeta de trabajo `catkin_ws` y se encontrara en el home. Para crear nuestro espacio de trabajo realizamos:



```
mkdir -p ~/catkin_ws/src  
cd ~/catkin_ws/  
catkin_make
```

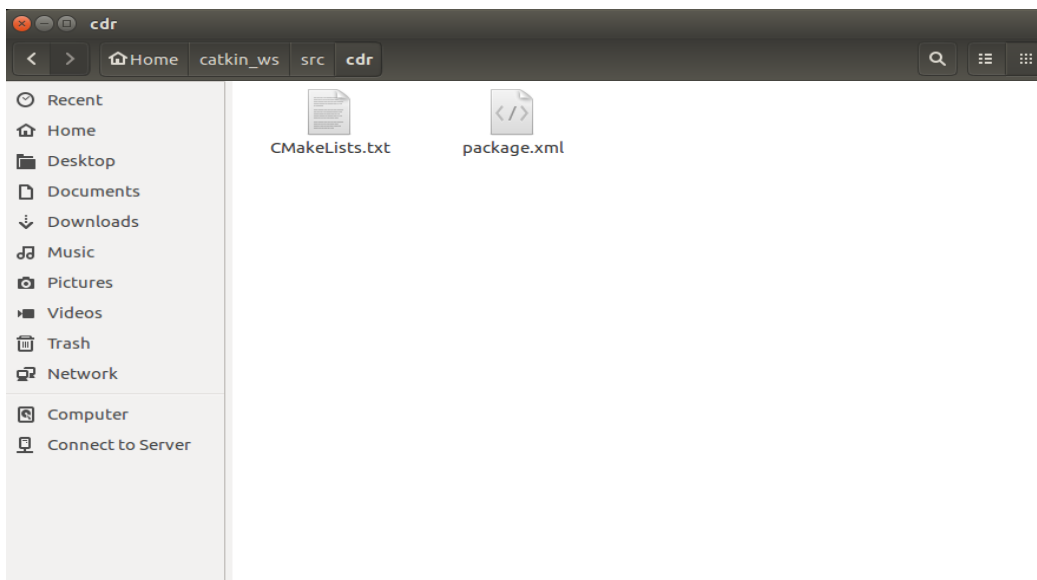
Lo anterior creará la carpeta **catkin_ws** (nuestro espacio de trabajo), y dentro de ella la carpeta **src**, luego desde la carpeta **catkin_ws** lanzamos **catkin_make**. Este comando creará un archivo **CMakeLists.txt** en la subcarpeta **src** y las carpetas **devel** y **build** en el directorio principal.

Ahora desde dentro de la carpeta **src** utilizamos el comando:

```
catkin_create_pkg cdr
```

Luego de este comando veremos que en el directorio se crean dos archivos de configuración:

- **package.xml** conocido como manifiesto tendrá información útil de nuestro paquete (en realidad este archivo solo es informativo acerca de las dependencias y utilidades de nuestro paquete).
- **CMakeList.txt** este archivo contiene una lista de instrucciones de compilación, que ejecutables deben ser creados, que código fuente utilizar para compilar cada uno y donde encontrar archivos y librerías necesarios para estos ejecutables.





catkin_create_pkg package-name	Crea un paquete, debe ser llamado desde el directorio src de nuestro espacio de trabajo.
---------------------------------------	--

Podemos editar el manifiesto con cualquier compilador de texto, aunque este archivo no aporta nada a la etapa de compilación ni de ejecución de nuestros programas, será útil si en algún momento compartimos nuestro paquete con el mundo. De forma que quien lo valla a utilizar sepa que dependencias tiene. Al abrir el manifiesto que genera por defecto catkin_create_package vemos que se encuentra lo suficiente comentado como para ir completando con la información deseada.

Para nuestro ejemplo el código de este podría ser algo similar a:

```
package.xml (~/.catkin_ws/src/cdr) - gedit
Open [icon] Save
<?xml version="1.0"?>
<package>
  <name>cdr</name>
  <version>0.0.1</version>
  <description>The cdr package</description>
  <maintainer email="alumno@todo.todo">alumno</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>geometry_msgs</build_depend>
  <run_depend>geometry_msgs</run_depend>
  <build_depend>turtlesim</build_depend>
  <run_depend>turtlesim</run_depend>
  <build_depend>roscpp</build_depend>
  <run_depend>roscpp</run_depend>

</package>
XML Tab Width: 8 Ln 17, Col 11 INS
```



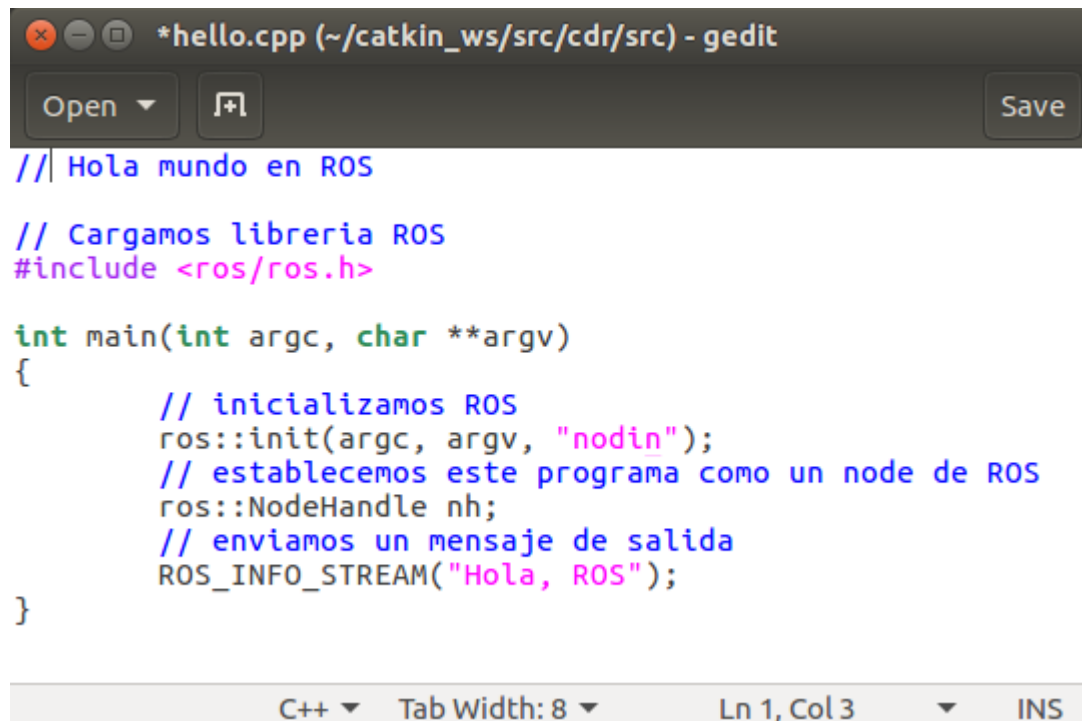
IMPORTANTE: puede que catkin_make nos genere por defecto un manifiesto que indique package format ="2", en este caso las líneas que indican "run_depend" deben ser omitidas.

Lo último antes de comenzar a escribir nuestro programa, es crear una nueva carpeta src dentro de la carpeta de nuestro paquete, si bien esto no es estrictamente necesario al momento de aumentar en complejidad nuestro proyecto será notoria la prolijidad que esto nos brinda.



hello.cpp

El siguiente código es el hola mundo en ROS, este archivo deberá estar dentro de la carpeta src de nuestro paquete.



```
// Hola mundo en ROS

// Cargamos libreria ROS
#include <ros/ros.h>

int main(int argc, char **argv)
{
    // inicializamos ROS
    ros::init(argc, argv, "nodin");
    // establecemos este programa como un node de ROS
    ros::NodeHandle nh;
    // enviamos un mensaje de salida
    ROS_INFO_STREAM("Hola, ROS");
}
```

Pasamos a explicar brevemente que hace cada línea:

#include <ros/ros.h> cargara la librería con las clases de ROS. Esta línea estará presente en todos los programas ROS.

ros::init esta línea inicializa la librería de ROS cliente, solo la llamaremos una vez al inicio de nuestros programas, su ultimo parámetro es un string que será el nombre por defecto de nuestro nodo.

ros::NodeHandle nos servirá para que nuestro programa se comunice con el sistema ROS. Con la creación de este objeto este programa se registra como nodo del ROSmaster.

ROS_INFO_STREAM genera un mensaje, este mensaje es enviado a diferentes puntos, uno de ellos la consola donde ejecutamos el nodo.



Una vez que tenemos el código de nuestro programa, debemos compilarlo para esto debemos realizar cuatro tareas:

1. Declarar dependencias: Necesitamos indicarle a la herramienta **catkin** de que otros paquetes y librerías dependerá nuestro paquete. Para esto deberemos editar el CMakeList.txt de nuestro paquete.

Mediante:

```
find_package(catkin REQUIRED COMPONENTS package-names)
```

Para nuestro caso, solo dependeremos del paquete roscpp el cual brinda la librería cliente de C++ ROS.

También es recomendable agregar esta dependencia en el manifiesto, la forma de agregarlas sería:

```
<build_depend>package-name</build_depend>  
<run_depend> package-name</run_depend>
```

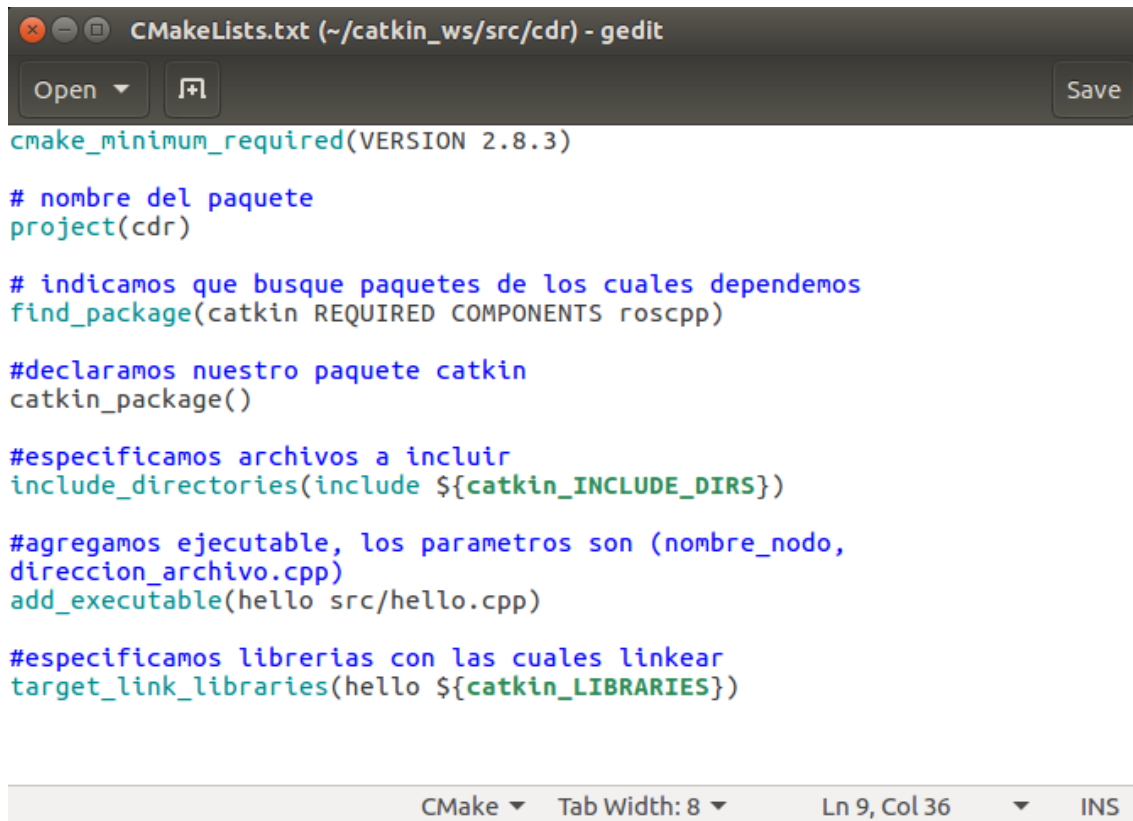
Nuevamente esto no es obligatorio, pero al momento de compartir nuestro trabajo le ahorrara dolores de cabeza al próximo programador.

2. Declarar un ejecutable: también en el archivo CMakeList.txt deberemos agregar las siguientes líneas por cada nodo que tenga nuestro paquete:

```
add_executable(executable-name source-files)  
target_link_libraries(executable-name ${catkin_LIBRARIES})
```

La primera declara el nombre del ejecutable que deseamos y una lista de archivos fuentes que deben ser combinados para generarlo. Si tenemos más de un archivo fuente deben ir separados por espacios.

La segunda línea dice al compilador **CMake** con que librerías debe linkear el ejecutable. Si nuestro paquete tuviese más de un ejecutable (que podría ser un nodo) tendremos que agregar este par de líneas por cada ejecutable. En la siguiente figura podemos ver cómo quedaría el CMakeList.txt para nuestro ejemplo:



```
CMakeLists.txt (~/.catkin_ws/src/cdr) - gedit

cmake_minimum_required(VERSION 2.8.3)

# nombre del paquete
project(cdr)

# indicamos que busque paquetes de los cuales dependemos
find_package(catkin REQUIRED COMPONENTS roscpp)

#declaramos nuestro paquete catkin
catkin_package()

#especificamos archivos a incluir
include_directories(include ${catkin_INCLUDE_DIRS})

#agregamos ejecutable, los parametros son (nombre_nodo,
direccion_archivo.cpp)
add_executable(hello src/hello.cpp)

#especificamos librerias con las cuales linkear
target_link_libraries(hello ${catkin_LIBRARIES})
```

CMake ▾ Tab Width: 8 ▾ Ln 9, Col 36 ▾ INS



- Si abrimos el CMakeList.txt que genera automáticamente el creador de paquetes, vemos que tiene muchas mas opciones que las que hemos utilizado. Estas opciones permitirán cargar librerías, mensajes etc. Un dato importante es el orden de las sentencias en este archivo ya que un orden inadecuado conducirá a errores de compilación.
- CMake es una herramienta multiplataforma de generación o automatización de código. El nombre es una abreviatura para "cross platform make" (make multiplataforma); más allá del uso de "make" en el nombre, CMake es una suite separada y de más alto nivel que el sistema make común de Unix. CMake es una familia de herramientas diseñada para construir, probar y empaquetar software. CMake se utiliza para controlar el proceso de compilación del software usando ficheros de configuración sencillos e independientes de la plataforma.
<https://es.wikipedia.org/wiki/CMake>.



3. Compilando el espacio de trabajo: Siempre que terminemos de editar nuestros códigos fuentes deberemos compilar el espacio de trabajo, para esto yendo a la carpeta del espacio de trabajo ejecutamos:

```
catkin_make
```

Esto compilara los paquetes creando dos carpetas devel y build. Estas contendrán archivos auxiliares de la compilación.

4. Sourcing: una vez que la compilación ha sido exitosa debemos indicarle a ros donde buscar los archivos para la ejecución, esto lo debemos hacer una vez en cada nueva terminal que utilicemos, desde el directorio de trabajo mediante la siguiente línea:

```
source devel/setup.bash
```

Terminada la compilación llego la hora de probar nuestro programa, para esto según lo que ya vimos en el primer ejemplo lanzamos el **roscore** y en otra terminal llamamos a nuestro nodo:

```
roslaunch cdr hello
```

La salida que obtendremos será algo como:

```
alumno@alumno: ~/catkin_ws
alumno@alumno: ~/catkin_ws 68x16
alumno@alumno:~/catkin_ws$ roslaunch cdr hello
[ INFO] [1539536302.706974388]: Hola, ROS
alumno@alumno:~/catkin_ws$
```



• Curiosamente, el numero que acompaña la salida representa el tiempo medido en segundos desde el primero de Enero de 1970, hasta el momento en que nosotros ejecutamos ROS_INFO_STREAM.



- **Programas de base**

En los siguiente describiremos mediante ejemplos los códigos mínimos para los dos tipos de programas básicos de ROS: publicador y suscriptor.

Programa publicador

```
// Este programa publica aleatoriamente mensajes de velocidad
// para el simulador turtlesim
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>      //incluimos tipo de mensaje
#include <stdlib.h>                    // para poder utilizar funcion rand()
int main(int argc, char **argv)
{
    //inicializamos sistema ROS
    ros::init(argc, argv, "publish_velocity");
    ros::NodeHandle nh;
    //creamos un objeto publicador
    ros::Publisher pub =nh.advertise<geometry_msgs::Twist>("turtle1/cmd_vel", 1000);
    //iniciamos el generador de numeros aleatorio
    srand(time(0));
    // iniciamos un bucle que se repitira a 2Hz hasta que el nodo sea apagado
    ros::Rate rate(2);
    while(ros::ok())
    {
        //creamos nuestro mensaje
        geometry_msgs::Twist msg;      //definomos variable del tipo mensaje
        msg.linear.x=double(rand())/double(RAND_MAX);
        msg.angular.z=2*double(rand())/double(RAND_MAX)-1;
        //publicamos el mensaje
        pub.publish(msg);
        //opcionalmente, sacamos nuestro mensaje por consola
        ROS_INFO_STREAM("Comando de velocidad aleatorio:"
        <<" lineal=" << msg.linear.x
        <<" angular=" << msg.angular.z);
        //espero hasta que sea tiempo de otra iteracion
        rate.sleep();
    }
}
```



Este código de ejemplo realizará publicaciones aleatorias en el tópico de velocidad de nuestro simulador turtlesim, como resultado tendremos que la tortuga se moverá aleatoriamente en su entorno y conoceremos cuales son las instrucciones para este tipo de programa.

Analizaremos paso a paso las líneas de código que difieren de nuestro ejemplo anterior. En el encabezado vemos que ahora aparecen nuevas librerías, en particular centraremos nuestra atención en la línea:

#include <geometry_msgs/Twist.h> Esta carga el encabezado relacionado al tipo de mensaje que usaremos, por cada nuevo tipo de mensaje que utilicemos deberemos cargar su encabezado la forma genérica para esto será:

```
#include <package_name/type_name.h>
```

Continuando en el código encontramos la creación de un objeto publicador, quien será el responsable de realizar las publicaciones en el tópico indicado. La forma genérica para un publicador es la siguiente:

```
ros::Publisher pub = node_handle.advertise<message_type>(topic_name,  
queue_size);
```

node_handle es el objeto creado mediante la clase `ros::NodeHandle`, para nuestro caso `nh`.

message_type es el tipo de dato que deseamos publicar.

topic_name es un string que contendrá el nombre del tópico en el que deseamos publicar

queue_size es un entero que representa el tamaño de la cola de mensajes del publicador.

En el caso en que nuestro programa genere mayor número de mensajes que lo que el publicador puede enviarlos, los mensajes más viejos en la cola son descartados.

Si nos interesara publicar en otro tópico con el mismo nodo solo debemos agregar otro objeto publicador.

Continuando en el código encontramos la línea:

```
ros::Rate rate(2);
```

Esta línea crea un nuevo objeto `ros::Rate` el cual controlara la frecuencia con que se ejecutara el bucle principal (expresado en Hz).



Vemos que esta línea es seguida de un bucle while donde aparece como condición la instrucción:

ros::ok()

Esta devolverá verdadero siempre que el nodo al que pertenece este activo, para desactivar un nodo existen varias opciones:

- Usando **rostopic kill**
- **Ctrl+C** en la consola del nodo
- Usando: **ros::shutdown();** en alguna parte del programa del propio nodo.

Finalmente, dentro del bucle tenemos tres partes a destacar, en la primera la construcción del mensaje, luego la línea:

pub.publish(msg);

La cual publica el contenido de la variable en la que definimos nuestro mensaje. Para concluir encontramos la instrucción:

rate.sleep();

Esta llama a un método del objeto `ros::Rate`, básicamente esta línea frenará la ejecución del nodo hasta la próxima iteración (habilitada de acuerdo a la frecuencia que definimos antes).

Antes de correr nuestro nuevo programa, debemos compilarlo, para esto procederemos en forma similar a lo explicado para el caso de `hello.cpp`. Debemos tener en cuenta que aquí hemos agregado una dependencia al tipo de mensaje **geometry_msgs** por lo que deberemos modificar las correspondientes líneas en el `CMakeList.txt`:

```
find_package(catkin REQUIRED COMPONENTS roscpp geometry_msgs)
```



y en el archivo package.xml:

```
<build_depend>geometry_msgs</build_depend>  
<run_depend>geometry_msgs</run_depend>
```

Si quisiéramos ejecutar nuestro programa necesitaríamos:

- Arrancar el master:

```
roscore
```

- Iniciar el simulador:

```
roslaunch turtlesim turtlesim_node
```

- Iniciar nuestro programa:

```
roslaunch cdr publicador
```

(recuerde que si esto lo hace en una nueva terminal debe decirle a ROS mediante **source devel/setup.bash** donde se encuentran sus archivos).



Programa suscriptor

A continuación, siguiendo el ejemplo del uso del simulador turtlesim, realizaremos un programa que se suscriba al tópico de posición del simulador y además muestre en su consola los datos recibidos. El código para este lo encontramos a continuación:

```
// este programa se suscribe al topico turtle1/pose y muestra en pantalla sus mensajes
#include <ros/ros.h>
#include <turtlesim/Pose.h>
#include <iomanip>      // contiene std::setprecision y std::fixed

// funcion del topico, esta funcion es llamada cada vez que un nuevo mensaje es recibido
void poseMensajeRecibido(const turtlesim::Pose& msg)
{
    ROS_INFO_STREAM(std::setprecision(2) <<std::fixed
    << "posicion=(" << msg.x << " " << msg.y << " "
    << " direccion=" << msg.theta);
}

int main(int argc, char **argv)
{
    //inicio comunicacion con sistema ROS
    ros::init(argc, argv, "suscriptor_pose");
    ros::NodeHandle nh;
    //creamos un objeto suscriptor
    ros::Subscriber sub= nh.subscribe("turtle1/pose", 1000, &poseMensajeRecibido);
    //cedemos control a ROS
    ros::spin();
}
```

Aquí vemos una estructura similar al caso anterior solo algunas cosas a remarcar:



Vemos que aparece un objeto suscriptor, cuya forma genérica es:

```
ros::Subscriber sub= node_handle.subscribe(topic_name, queue_size,pointer_to_callback_function);
```

Aquí los argumentos son:

node_handle es el objeto creado mediante la clase `ros::NodeHandle`, `nh`.

topic_name es el nombre del tópico al cual nos suscribiremos.

queue_size largo de la cola de recepción de mensajes.

pointer_to_callback_function es un puntero a la función que ha de ejecutarse cuando un mensaje es recibido.

Respecto de la “**callback function**”, esta será una función genérica que tendrá la siguiente forma:

```
void function_name(const package_name::type_name &msg)
{
    ...
}
```

El nombre del paquete y el tipo de mensaje deben ser los mismo que utilice el publicador. El cuerpo de esta función tiene acceso a todos los campos del mensaje recibido y puede almacenar, usar o descartar la información contenida en él.

Para concluir el análisis de este código vemos en la última línea:

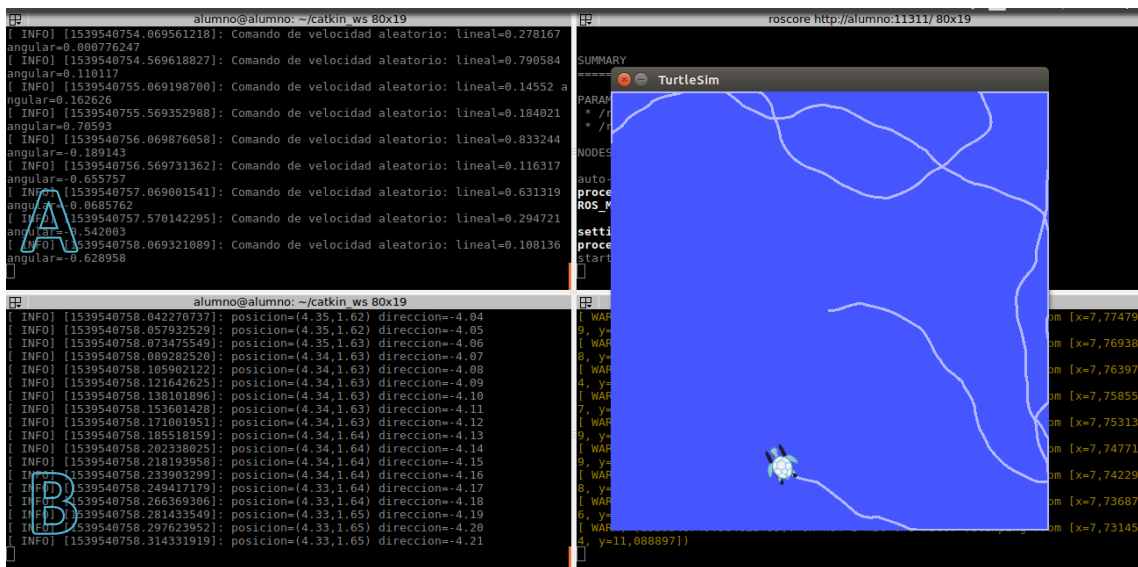
```
ros::spin();
```

Esta línea, habilita al programa a atender la “interrupción ” producida por la llegada de un mensaje por lo tanto la ejecución de la **callback function**.

En el caso que nuestro suscriptor además realice algún código en forma repetida, deberíamos utilizar la instrucción **ros::spinOnce()**; dentro de un bucle, de esta forma se estaría atendiendo los mensajes llegados cada un cierto tiempo sin bloquear la otra tarea del suscriptor:

```
while(ros::ok())  
{  
  ros::spinOnce();  
}
```

La compilación y ejecución de este programa es similar a lo que ya hemos visto, al ejecutar ambos programas tendremos como salida algo similar a la siguiente figura:



Donde en la consola A vemos los mensajes enviados por el publicador, en B la salida del suscriptor, y en el simulador el movimiento aleatorio de la tortuga.



Actividad : Algo fácil!

Modifique el código del publicador de forma que el comando enviado haga que la tortuga quede realizando círculos.



Actividad : Algo para pensar!

En esta actividad/ejemplo usaremos un simulador 2D, y aplicaremos algunos de los comandos vistos hasta ahora para escribir un nodo que realice una acción en particular.

1) Corriendo el simulador: mediante el siguiente comando podremos ejecutar un simulador 2D incluido en ROS llamado Stage, con el comando cargamos un mapa aleatorio el cual contiene una serie de obstáculos, y un cuadrado azul que simula un robot terrestre. (recuerde ejecutar el master antes)

```
roslaunch stage_ros stageros $(rospack find stage_ros)/world/willow-erratic.world
```

Vemos que si presionamos la tecla `r` podemos cambia la forma de ver el mapa.

2) Deseamos controlar nuestro robot, para esto debemos utilizar un nodo de teleoperación (similar al que utilizamos con turtlesim), podemos googlear por un paquete ROS que sea compatible y ver que encontramos.

Uno compatible puede instalarse usando:

```
sudo apt-get install ros-kinetic-teleop-twist-keyboard
```

y luego podemos ejecutarlo mediante

```
roslaunch teleop_twist_keyboard teleop_twist_keyboard.py
```

Felicidades ha instalado y ejecutado un paquete ROS de la infinidad que son disponibles en la red.

Algo más llamativo de este paquete es la extensión del archivo ejecutable, aquí `.py` esto nos indica que el código en el que ha sido realizado es Python. Algo que hemos omitido hasta aquí es que en ROS los códigos pueden ser escritos en varios lenguajes (principalmente C++ y Python), ya que gracias a la estandarización de mensajes y procesos estos son compatibles.

3) Ejecute el comando:

```
roslaunch rviz rviz -d `rospack find stage_ros`/rviz/stage.rviz
```

Este comando abrirá una aplicación de ROS para la muestra de los datos recuperados del robot, por ahora es solo a modo informativo, pero si deseamos hacer mapeo esta aplicación sería muy útil. Luego de explorar un poco las opciones de esta aplicación (por ejemplo, agregando una odometria: `add->odometry`, y en el campo topic indicamos el generado por el simulador) la cerraremos ya que no la seguiremos usando.



- 4) Inicie `rqt_graph` que observa ?
- 5) Mediante las herramientas asociadas a `rostopic` obtenga información del tópico `/base_scan`, cuál es su tipo de mensaje ? que estructura tiene este?
- 6) Busque en la web el tipo de mensaje anterior para obtener más información, como podría obtener la medida de distancia del objeto directamente al frente del robot?
- 7) Con lo encontrado en el punto anterior y el código de ejemplo del suscriptor, programe un nodo que muestre por consola solo la distancia a un obstáculo al frente del robot detectada en cada medida del sensor.
- 8) Con lo desarrollado en el punto anterior, más el código ejemplo del publicador podría generar una estrategia simple que haga avanzar al robot en forma autónoma esquivando obstáculos que encuentre directamente al frente en el mapa (no es necesario que sea 100% efectiva).

(Ayudas: el tópico en el que publica el nodo `teleop` son comandos de velocidad, para evitar obstáculos directamente en frente del robot una medida es suficiente, pero si agregamos más medidas tendremos un mejor performance)



Bibliografía

- A gentle Introduction to ROS. Jason M. O,Kane
- ROS Tutorials. <http://wiki.ros.org/ROS/Tutorials>
- ROS Robotics By Example. Carol Fairchild