



Universidad  
de Alcalá

## Guía Rápida

### Introducción a ROS

<b>1. INTRODUCCIÓN</b>	<b>4</b>
<b>2. INSTALACIÓN DE ROS</b>	<b>4</b>
2.1. Recomendaciones Máquina Virtual Ubuntu con ROS	4
<b>3. SIMULADOR STDR</b>	<b>6</b>
3.1. Instalación del simulador	6
3.2. Creación de un robot	7
3.3. Simulador: mapa	9
3.4. Directorios que manejar para usar el simulador STDR	9
<b>4. CONCEPTOS BÁSICOS DE ROS</b>	<b>10</b>
4.1. Roscore	10
4.2. Nodos	10
4.3. Topics	11
4.4. Servicios	13
4.5. Rosbag	13
4.6. TF (Transformadas)	13
4.7. Ejecución de un programa	14
<b>5. PROGRAMACIÓN EN ROS: CONCEPTOS BÁSICOS</b>	<b>16</b>
5.1. Creación de un paquete	16
5.2. Creación de un fichero .launch	16
5.3. Esqueleto genérico de un programa en C++ (nodo)	18
5.4. Ejemplo de programa en C++: mover el robot un metro hacia adelante	21
5.5. Compilación y ejecución del nodo en C++	24
<b>6. VISUALIZADOR RVIZ</b>	<b>25</b>
<b>7. ROS DISTRIBUIDO: SISTEMAS MULTIMÁQUINA</b>	<b>29</b>
<b>8. PROGRAMACIÓN MATLAB-ROS</b>	<b>30</b>
8.1. Configuración de la red	30
8.2. Programa cliente en Matlab	31
8.3. Ejecución	33
<b>9. CREACIÓN DE MODELOS SIMULINK-ROS</b>	<b>34</b>
9.1. Creación de un <i>Publisher</i>	35
9.2. Creación de un <i>Subscriber</i>	37
9.3. Ejemplo de diagrama Simulink: mover el robot un metro hacia adelante	38
9.4. Ejemplo de diagrama Simulink: lectura de sensores	40
<b>10. PRÁCTICA: SIMULACIÓN EN MATLAB</b>	<b>41</b>
<b>11. CONEXIÓN A UN ROBOT REAL</b>	<b>42</b>

<b>12.</b>	<b><i>PRÁCTICA: ROBOT REAL</i></b>	<b>43</b>
12.1.	PRÁCTICA: Robot Real en Matlab-ROS	43
12.2.	PRÁCTICA: Robot Real en Simulink-ROS	43
<b>13.</b>	<b><i>Generar código ROS C++ a partir de Simulink</i></b>	<b>44</b>
<b>14.</b>	<b><i>ANEXO - FAQs</i></b>	<b>45</b>

# 1. INTRODUCCIÓN

Robot Operating System o ROS, es una plataforma de desarrollo robótico que permite crear aplicaciones con múltiples sensores y actuadores de forma flexible. Es una colección de herramientas, bibliotecas y convenciones que tiene como objetivo simplificar la tarea de crear un comportamiento de robot complejo y robusto en una amplia variedad de plataformas robóticas.

ROS se creó desde cero para fomentar el desarrollo de software de robótica colaborativa. Por ejemplo, un laboratorio podría tener expertos en el mapeado de ambientes interiores, y podría contribuir con un sistema de clase mundial para producir mapas. Otro grupo podría tener expertos en el uso de mapas para navegar, y otro grupo podría haber descubierto un enfoque de visión por computadora que funciona bien para reconocer pequeños objetos en desorden. ROS fue diseñado específicamente para grupos de trabajo como estos para colaborar y construir sobre el trabajo de cada uno.

Esta guía rápida pretende ser un punto de partida en el desarrollo de las prácticas de laboratorio de diferentes asignaturas para que se puedan crear fácilmente aplicaciones tanto en modo nativo (C++, Python) como en Matlab / Simulink.

Se puede encontrar más información sobre ROS en su página oficial: [www.ros.org](http://www.ros.org)

## 2. INSTALACIÓN DE ROS

Se recomienda utilizar la máquina virtual proporcionada en la asignatura, en la que ya se encuentra realizada una preinstalación de ROS Kinetic y los paquetes más importantes a utilizar en las prácticas

- **Usuario:** alumno
- **Password:** alumno

### 2.1. Recomendaciones Máquina Virtual Ubuntu con ROS

- **Configuración de red:**
  - Para evitar conflictos de red, debemos cambiar MAC (con las flechas que generan un MAC aleatoria) varias veces, evitando así que coincidan varios maquinas virtuales con la misma MAC en la red.
  - Comprobar que está configurada como “Bridge”
  - Dar permisos a Matlab en el Firewall Windows en Redes públicas y privadas.
- **Para un mejor rendimiento de la máquina virtual, se recomienda:**
  - Aumentar RAM a 2-3Gb
  - Aumentar procesadores a 2 o 4
  - Aumentar memoria gráfica a 128-256Mb.
  - Bajar la resolución de la pantalla una vez arrancado Ubuntu: 1024x768 suele ser suficiente.
- Activar portapapeles bidireccional, para poder copiar y pegar entre la máquina virtual y el sistema operativo nativo. Para poder usar esta función es necesario instalar “Guest Additions” en la máquina virtual.

En el Anexo al final de este documento se encuentra más información sobre estas recomendaciones.

Si por el contrario se desea instalar ROS Kinetic sobre Ubuntu 16.04 en modo nativo (**no se recomienda para la asignatura**), deben seguirse las instrucciones dadas en la página <http://wiki.ros.org/Installation/Ubuntu> (Desktop-full install, instalación recomendada).

Una vez instalado ROS, ha de configurarse el entorno para poder generar código. Es necesario crear un espacio de trabajo<sup>1</sup> (*workspace*) donde se guardarán y compilarán los archivos de los diferentes paquetes creados.

Para ello, desde un terminal de Ubuntu, se realizarán los siguientes pasos:

### 1. Creación del espacio de trabajo:

```
mkdir -p ~/robotica_movil_ws/src
```

### 2. Inicialización del espacio de trabajo:

```
cd ~/robotica_movil_ws/src/  
catkin_init_workspace  
cd ~/robotica_movil_ws  
catkin_make
```

### 3. Añadir el espacio de trabajo al path por defecto:

```
sudo gedit ~/.bashrc
```

Añadir al final del fichero estas dos líneas:

```
source ~/robotica_movil_ws/devel/setup.bash  
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:~/robotica_movil_ws/
```

Cada vez que se modifica el fichero `.bashrc` se deben cerrar todas las ventanas de terminales abiertas y volver a abrirlas para que se ejecute de nuevo el fichero `.bashrc` actualizado con las modificaciones.

---

<sup>1</sup> Tutorial de creación de un workspace: [http://wiki.ros.org/catkin/Tutorials/create\\_a\\_workspace](http://wiki.ros.org/catkin/Tutorials/create_a_workspace)

### 3. SIMULADOR STDR

Se trata de un simulador específicamente creado para ROS (stdr\_simulator<sup>2</sup>), que permite modelar y simular diferentes robots y sistemas sensoriales dentro de un entorno de movimiento bidimensional.

#### 3.1. Instalación del simulador

El simulador puede instalarse desde los repositorios de ROS para Ubuntu con la siguiente instrucción.

```
sudo apt-get install ros-kinetic-stdr-simulator
```

O bien desde las fuentes, que se pueden descargar de la página: [https://github.com/stdr-simulator-ros-pkg/stdr\\_simulator](https://github.com/stdr-simulator-ros-pkg/stdr_simulator).

Sin embargo, esta versión tiene unos **bugs** por solucionar, **por lo que es aconsejable** instalar el simulador **a partir de las fuentes como se ha indica a continuación:**

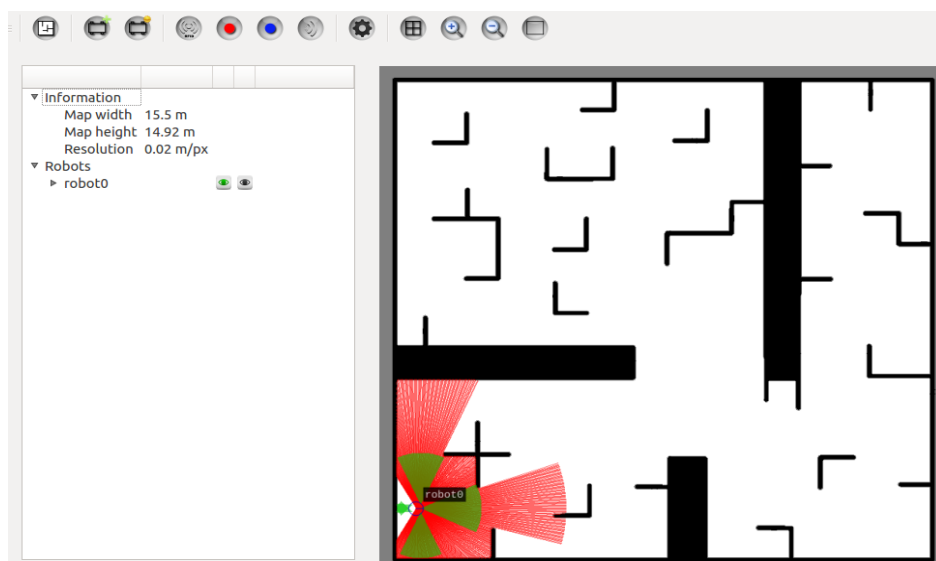
En caso de usar la máquina virtual, la descarga ya está realizada en la carpeta Downloads. Para instalarlo ha de descomprimirse dentro del directorio robotica\_movil\_ws/src y a continuación ejecutar:

```
cd ~/robotica_movil_ws  
catkin_make
```

Para probar que la instalación es correcta se puede ejecutar la siguiente instrucción:

```
roslaunch stdr_launchers server_with_map_and_gui_plus_robot.launch
```

Deberá aparecer la siguiente ventana mostrando el GUI del simulador, con un mapa y un robot cargados:



Para cerrar el simulador, además de cerrar la ventana de la aplicación, es necesario pulsar Ctrl+C en el terminal desde el cual se lanzó.

<sup>2</sup> Tutoriales de STDR: [http://wiki.ros.org/stdr\\_simulator](http://wiki.ros.org/stdr_simulator)

### 3.2. Creación de un robot

A continuación, se muestran los pasos para crear un robot AmigoBot con sus sensores asociados (iguales a los que utiliza el robot real), que se utilizará en prácticas posteriores:

- **Lanzar el simulador**, utilizando para ello cualquiera de los ficheros .launch que vienen con la instalación. Por ejemplo, el que carga sólo la GUI y un mapa:

```
roslaunch stdr_launchers server_with_map_and_gui.launch
```

- **Crear un nuevo robot**, clicando para ello en “Create robot” (tercer botón de la barra de herramientas superior). Por defecto los robots creados tienen forma circular, aunque se pueden definir formas poligonales utilizando la opción “Footprint” para definir los vértices. En este caso, por simplicidad de diseño se creará el robot circular, con las dimensiones del robot:
  - **AmigoBot**: medidas 33 x 28 cm (definiendo un radio de 15 cm como aproximación).
  - **P3DX**: medidas 44.5 x 39,3cm (definiendo un radio de 22.5 cm como aproximación).
- **Añadir un sensor láser**, clicando para ello en el botón + en la sección “Lasers”. Esto añadirá un láser llamado “laser\_1”. Seleccionando el botón de edición se pueden modificar sus parámetros. Los robots disponibles en laboratorio tienen instalado un sensor láser de uno de estos dos modelos, cuyos parámetros se indican (cada puesto debe configurar el láser del modelo que le corresponda):
  - Para simular un láser **RPLIDAR-A2** deben ajustarse los siguientes valores:  
Number of Rays: 400  
Max distance (m): 8.0  
Min distance (m): 0.15  
Angle span (degrees): 360  
Translation – x(m): 0.09
  - Para simular un láser **Hokuyo URG 04LX** deben ajustarse a los siguientes valores:  
Number of Rays: 672  
Max distance (m): 5.6  
Min distance (m): 0.06  
Angle span (degrees): 240  
Translation – x(m): 0.09

Finalmente clicar en “update” y salvar el laser\_1 en la carpeta “laser\_sensors” con el nombre “rplidar\_a2.xml” o “hokuyo\_urg04lx.xml” (*nota: comprobar que se dispongan de permisos de escritura en esa carpeta, puesto que dependiendo de la instalación, pueden no tener esos permisos por defecto*).

- **Añadir un sensor de ultrasonidos (sonar)**, clicando para ello en el botón + de la sección “Sonars”. Esto añadirá un sonar llamado “sonar\_1”. Seleccionando el botón de edición se pueden configurar los siguientes parámetros para que simule los sónares del robot AmigoBot:  
Max distance (m): 5.0  
Min distance (m): 0.1  
Con span (degrees): 15  
Frequency: 20  
Translation (de la tabla correspondiente)  
Translation (de la tabla correspondiente)  
Orientation: (de la tabla correspondiente)

Finalmente clicar en “update” y salvar el sonar\_1 en la carpeta “range\_sensors” con el nombre

“sonar\_amigobot.xml” o “sonar\_p3dx.xml”.

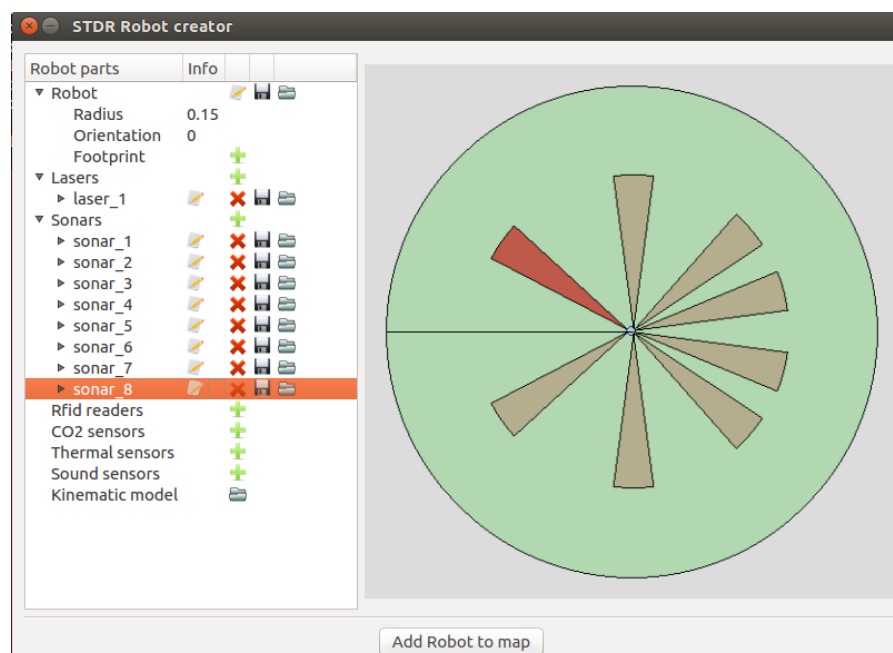
- **Crear el anillo de ultrasonidos**, añadiendo para ello otros 7 sonares más (de sonar\_2 a sonar\_8). Para ello, se puede cargar para cada sonar los datos de “sonar\_amigobot.xml” o “sonar\_p3dx.xml”. y a continuación modificar el parámetro de orientación y posición a los siguientes valores:
  - **Amigobot:**

Sónar	X	Y	Orientación (°)	Orientación (rad)
0	0.076	0.1	90	1.5708
1	0.125	0.075	44	0,767945
2	0.150	0.03	12	0,20944
3	0.150	-0.03	-12	- 0,20944
4	0.125	-0.075	-44	- 0,767945
5	0.076	-0.1	-90	-1.5708
6	-0.14	-0.058	-144	- 2,51327
7	-0.14	0.058	144	2,51327

- **P3DX:**

Sónar	X	Y	Orientación (°)	Orientación (rad)
0	0.115	0.130	90	1,5708
1	0.155	0.115	50	0,872665
2	0.190	0.080	30	0,523599
3	0.210	0.025	10	0,174533
4	0.210	-0.025	-10	- 0,174533
5	0.190	-0.080	-30	- 0,523599
6	0.155	-0.115	-50	- 0,872665
7	0.115	-0.130	-90	-1,5708

Quedando el anillo de ultrasonidos configurado del siguiente modo:



*Ejemplo de amigobot con rplidar*



- **Salvar el robot**, con el nombre “amigobot.xml” o “p3dx.xml” en la carpeta “robots”.
- **Añadir el robot**, clicando en el segundo botón de la barra de herramientas (Load robot), seleccionando el robot que acabamos de crear, y a continuación clicando en la posición del mapa en la que queremos que se posicione el robot. (NOTA: si se obtiene un error al cargar el robot, editar el archivo amigobot.xml y eliminar la sección <kinematics> completa).
- **Modificar el robot**: Una vez generado el robot, el archivo .xml puede ser modificado de forma sencilla. Por coherencia con la numeración de sensores del amigobot se recomienda modificar el archivo para que los sonar vayan desde sonar\_0 a sonar\_7

### 3.3. Simulador: mapa

Los mapas en ROS vienen definidos por:

- Un **archivo de imagen** en escala de grises (habitualmente en formato **png**)
- Un **archivo de descripción .yaml**. Un ejemplo de archivo de descripción típico es el siguiente:

```
image: sparse_obstacles.png → indica el archivo de imagen a cargar.  
resolution: 0.02 → en metros por píxel.  
origin: [0.0, 0.0, 0.0] → Origen del mapa (x, y, orientación).  
occupied_thresh: 0.6 → Los píxeles con más de esta ocupación (escala de grises) se consideran ocupados.  
free_thresh: 0.3 → Los píxeles por debajo de esta ocupación (escala de grises) se consideran libres.  
negate: 0 → Si está a 0, los píxeles que tienden a blanco son libres y los negros ocupados. Si está a 1 se invierte esta consideración.
```

### 3.4. Directorios que manejar para usar el simulador STDR

#### 1. Directorio con los archivos *launch*

[~/robotica\\_movil\\_ws/src/stdr\\_simulator-indigo-devel/stdr\\_launchers/launch](#)

#### 2. Directorio donde se encuentran los mapas (png + yaml):

[~/robotica\\_movil\\_ws/src/stdr\\_simulator-indigo-devel/stdr\\_resources/maps](#)

#### 3. Directorio donde se encuentran las definiciones de los robots (.xml):

[~/robotica\\_movil\\_ws/src/stdr\\_simulator-indigo-devel/stdr\\_resources/resources/robots](#)

## 4. CONCEPTOS BÁSICOS DE ROS

En este apartado se explican algunos de los conceptos básicos para comenzar a trabajar con ROS.

### 4.1. Roscore

ROS necesita un nodo maestro en ejecución continuamente. Este nodo se encarga de coordinar y sincronizar todo el sistema de nodos/topics/servicios. Para ejecutarlo, debe teclearse en un terminal:

```
roscore
```

*Nota: si se ejecuta algún programa mediante `roslaunch` (se verá más adelante), éste se encargará de ejecutar `roscore` automáticamente.*

### 4.2. Nodos

Cada programa en ejecución dentro de ROS se considera un nodo. Los nodos se comunican entre sí mediante “topics” y “servicios”.

Para monitorizar y obtener información de los nodos que están en ejecución en un determinado momento, se dispone de la herramienta `rostopic`.

A modo de ejemplo, mientras ejecutamos el simulador con la prueba de instalación, podemos hacer uso de esta herramienta en otro terminal. Los comandos más utilizados son:

- **`rostopic list`** → devuelve una lista de los nodos que están activos.

En el ejemplo que nos ocupa:

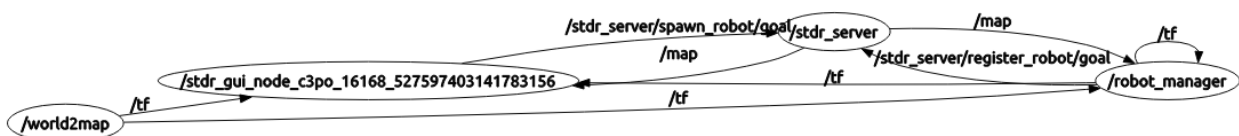
```
rostopic list
```

devuelve:

```
/robot_manager  
/rosout  
/stdrv_gui_node_c3po_12711_4502575229393065819  
/stdrv_server  
/world2map
```

- **`rostopic info “nombre_nodo”`** → devuelve información de ese nodo, como sus publicaciones, sus subscripciones, los servicios que ofrece y sus conexiones a los demás nodos.
- **`rostopic kill “nombre nodo”`** → elimina un nodo de la ejecución.

Una forma cómoda de visualizar todos los nodos que se encuentran en ejecución en un determinado momento, es ejecutando la herramienta **`rqt`** en un terminal y a continuación cargando el plugin “plugin/introspection/node\_graph” y la vista “nodes only”, obteniéndose un grafo como el siguiente:



### 4.3. Topics

Los topics son canales de comunicación que utilizan un tipo de mensaje determinado para enviar información de un nodo a otro. La conexión puede ser de tipo n:m (varios a varios), es decir que varios nodos pueden escribir en el mismo topic y varios nodos pueden leer el mismo topic. La única limitación que tienen es que cada topic solamente admite datos de un único tipo, por lo que si se quieren transmitir varios tipos de datos entre dos nodos habrá de crearse un topic para cada uno de ellos.

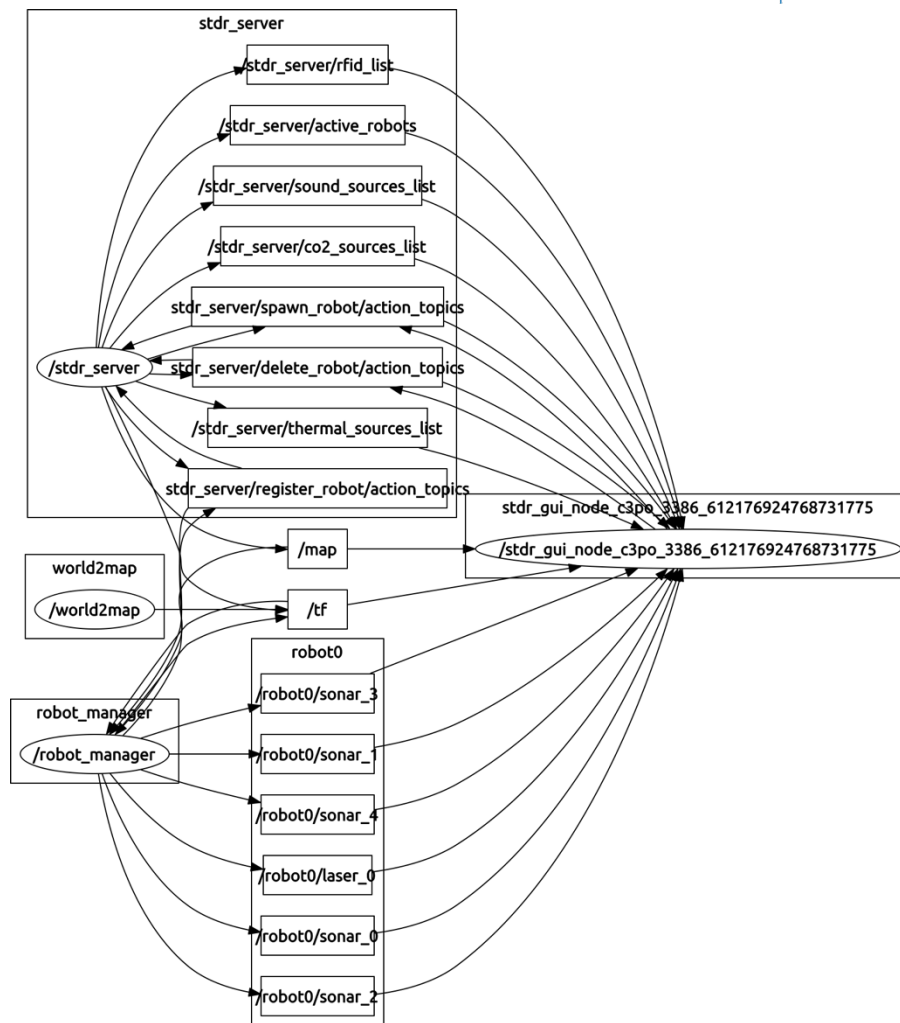
La herramienta `rostopic` nos permite inspeccionar estos canales de comunicación. Sus comandos más útiles son los siguientes:

- **rostopic list** → lista todos los topics activos en el sistema.
- **rostopic info “nombre\_topic”** → muestra la información del topic indicado: tipo de dato que transmite, nodos que están publicando en el topic y nodos que están leyendo de él.
- **rostopic type “nombre\_topic”** → muestra la información del tipo de dato que se transmite por el topic.
- **rostopic echo “nombre\_topic”** → crea un “subscriber” desde línea de comandos y vuelca a pantalla los datos que se están transmitiendo por ese topic.
- **rostopic pub “topic” “tipo de dato” “datos”** → crea un “Publisher” desde la línea de comandos. Permite publicar datos en un topic desde consola generando un “publisher”. Por ejemplo:

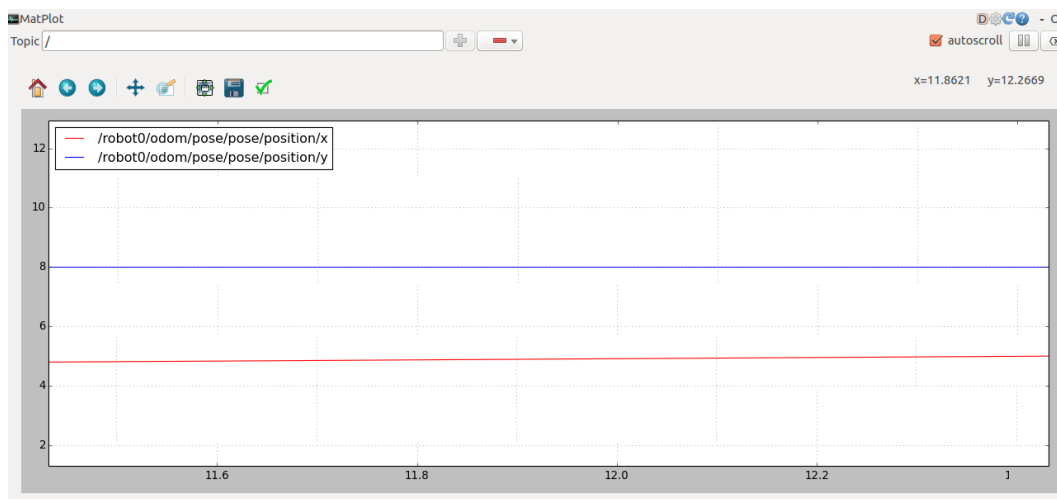
```
rostopic pub --once /robot0/cmd_vel geometry_msgs/Twist '{ linear : { x :  
0.2} }'
```

 publicará durante 3 segundos una velocidad lineal de 0.2 m/s al robot

Al igual que con los nodos, se puede realizar una inspección visual de los topics mediante la herramienta **rqt**, en este caso seleccionando el plugin “plugin/introspection/node\_graph” y la vista “node/topics (active)”, que muestra en elipses los nodos y en cuadrados los topics con los que se comunican.



La herramienta **rqt** también permite visualizar el valor de los campos numéricos de los topics. Para ello se ejecuta el plugin “plugin/Visualization/plot”, y se añade el campo que queramos mostrar gráficamente. Por ejemplo, podemos representar la posición en coordenadas cartesianas del robot añadiendo los campos “/robot0/odom/pose/pose/position/x” y “/robot0/odom/pose/pose/position/y” y se visualizará de manera similar a la siguiente imagen, donde el eje y es el valor del campo y el eje x el instante en el que se ha recibido (se visualiza de manera continua por simplicidad aunque los datos se reciben en tiempos discretos):



## 4.4. Servicios

Los servicios proporcionan otra manera de comunicación entre nodos. Esta comunicación en vez de ser continua es del tipo “llamada-respuesta” y por lo tanto suele utilizarse para eventos especiales como por ejemplo resetear una simulación, etc.

La herramienta que permite gestionar los servicios en modo comando es `rosservice` y sus opciones más utilizadas son las siguientes:

- **`rosservice list`** → lista los servicios que hay activos.
- **`rosservice info “nombre_servicio”`** → muestra la información de un servicio: su tipo y qué nodo lo publica.
- **`rosservice call “nombre_servicio” “argumentos”`** → sirve para llamar a un servicio desde consola de comandos. Por ejemplo:

```
rosservice call /robot_manager/list "{}" Este servicio proporciona como respuesta cuántos robots hay activos en el simulador.
```

## 4.5. Rosbag

Rosbag es un conjunto de herramientas que permiten guardar datos durante la ejecución de un programa para posteriormente poder reproducirlos de nuevo.

Las instrucciones más importantes de esta herramienta son las siguientes:

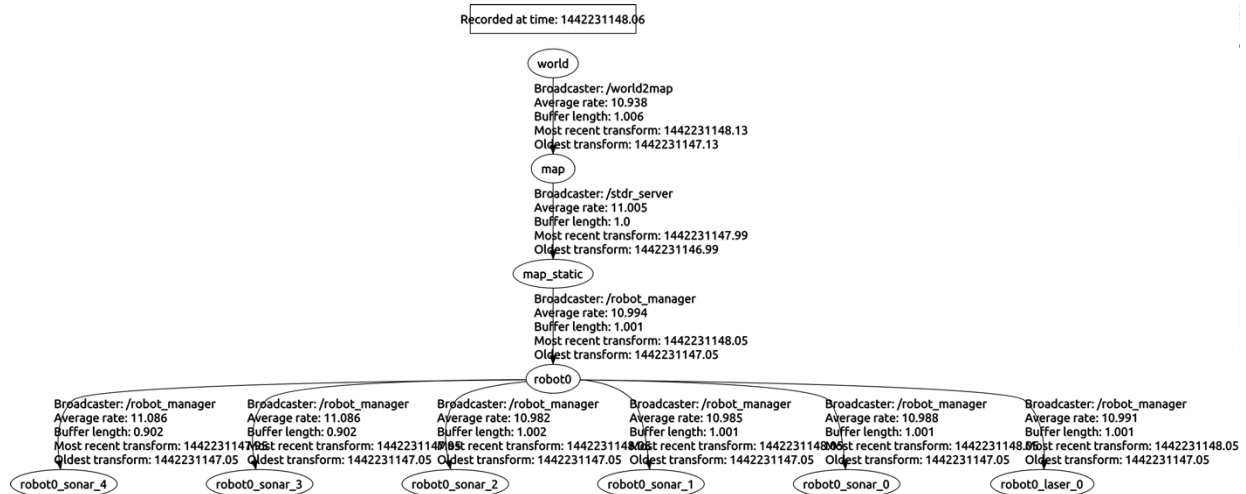
- **`rosv bag record -a`** → Salva a un archivo toda la información (topics) en ejecución en el programa
- **`rosv bag record “nombre de topics”`** → Salva solamente la información de los topics indicados. Es conveniente salvar siempre el topic “tf” para mantener la información sobre el estado del sistema.
- **`rosv bag play “archivo.bag”`** → Vuelve a ejecutar los datos guardados en un archivo. Útil para la depuración y análisis de resultados a posteriori.

## 4.6. TF (Transformadas)

`/tf` es un topic especial de ROS que está siempre presente y sirve para relacionar los distintos marcos de coordenadas que existen en el sistema. Es gestionado por un paquete llamado TF que incluye múltiples utilidades para trabajar con los marcos de coordenadas y gestionar sus transformaciones, y que se verá en detalle en prácticas posteriores. La relación entre los sistemas de coordenadas, incluso cuando es **constante** a lo largo del tiempo, ha de ser publicada de forma continua en el topic `/tf` para que el resto de nodos pueda utilizar esta información.

Usualmente, cada topic contiene información referida a un sistema de coordenadas. Este sistema de coordenadas se encuentra indicado explícitamente en la cabecera del tipo de datos (campo **`frame_id`**) con excepción de algunos tipos de datos que no están referidos a ningún marco de referencia y carecen de ese campo (por ejemplo los datos de tipo entero).

Para visualizar el árbol de transformadas se puede utilizar la herramienta **`rqt`**, plugin (plugins/visualization/TF Tree) el cual mostrará la relación entre los distintos marcos de coordenadas así como quién genera cada transformada y con qué frecuencia la actualiza. En el ejemplo que nos ocupa hasta ahora:



En este grafo se puede comprobar que existe un marco base de coordenadas llamado “world” relacionado con otro marco de coordenadas llamado “map”. Esta relación la realiza el nodo “world2map” el cual publica una transformada estática entre ambos. A su vez el marco “map” está relacionado por una transformada estática enviada por el simulador (stdr\_server) con el marco de coordenadas “map\_static”. Los marcos de coordenadas del mapa (map\_static) y el robot (robot0) están relacionados por la odometría del mismo, relación que cambia conforme se mueve el robot y está publicada por el simulador (nodo robot\_manager). También existen una serie de transformadas entre el robot y sus diferentes sensores, que son publicadas por el simulador.

Hay que tener en cuenta una restricción importante a la hora de crear un árbol de transformadas en ROS: un marco de referencia puede tener varios “hijos” pero solamente un único “padre”. De tal forma que siempre existirá un único elemento raíz (comúnmente “world” o “map” en sistemas de robots móviles) del que irán colgando los distintos elementos que existen en el entorno (distintos robots, sensores del entorno, etc...).

## 4.7. Ejecución de un programa

Para la ejecución de un programa en ROS existen dos maneras: el comando `roslaunch` y el comando `roslaunch`.

- **Rosrun**: este comando permite ejecutar un único nodo siempre que exista un nodo “roscore” activo. Su formato es el siguiente:

```
roslaunch "nombre_paquete" "nombre_nodo" "argumentos"
```

Los argumentos que se pueden pasar a un programa son de tres tipos:

- Redirecciones: se puede modificar el nombre de los topic en los que un nodo escucha o publica. El formato para ello es “nombre\_topic:=nuevo\_nombre\_topic”
- Parámetros: algunos nodos aceptan parámetros de configuración (publicados en el servidor de parámetros de ROS gestionable mediante las instrucciones “rosparam”). Para indicar estos parámetros ha de escribirse “\_nombre\_parámetro:=valor\_parámetro”.
- Argumentos clásicos: algunos nodos aceptan parámetros en el formato clásico de c/c++. Para incluirlos simplemente habrá que escribir estos parámetros (dependiendo del parser del nodo en cuestión) a continuación del nombre del nodo.

Como ejemplo podemos ejecutar un nodo que nos permite teleoperar el robot en el simulador:

```
roslaunch teleop_twist_keyboard teleop_twist_keyboard.py  
cmd_vel:=/robot0/cmd_vel _speed:=0.3 _turn:=0.5
```

Esta instrucción ejecuta el nodo “*teleop\_twist\_keyboard.py*” que se encuentra dentro del paquete “*teleop\_twist\_keyboard*”. Se realiza una redirección de topics cambiando el topic en el que escribe por defecto (*cmd\_vel*) al del robot (*/robot0/cmd\_vel*). También se modifican dos parámetros (*speed* y *turn*) limitando la velocidad máxima comandada a 0.3 m/s de velocidad lineal y 0.5 rad/s de velocidad angular.

- **Roslaunch:** dada la complejidad que puede conllevar la ejecución de ROS, se utilizan los comandos *roslaunch* a modo de “script” para lanzar distintos nodos en una única instrucción. Un detalle importante es que el archivo *.launch* ejecuta automáticamente un nodo “*roscore*” en caso de que el mismo no se encuentre activo.

El formato de ejecución es el siguiente:

```
roslaunch "nombre_paquete" "fichero.launch"
```

Para que un archivo pueda ser ejecutado ha de encontrarse dentro de la carpeta */launch* del paquete correspondiente. El fichero *roslaunch* puede ejecutar distintos nodos, incluir ficheros de configuración de parámetros e incluso incluir otros ficheros *.launch*. Posteriormente se detallará con un ejemplo el formato de este tipo de archivos.

## 5. PROGRAMACIÓN EN ROS: CONCEPTOS BÁSICOS

En los siguientes apartados se explica cómo **crear un nodo de ROS en C++**, y cómo lanzar aplicaciones formadas por uno o más nodos a través de ficheros `.launch`.

### 5.1. Creación de un paquete

Los programas en ROS se agrupan en paquetes. Un paquete puede contener múltiples nodos, ficheros de configuración, etc. destinados todos ellos a conseguir una aplicación final concreta.

Como ejemplo, se va a crear a continuación un paquete para realizar la práctica 1. Los paquetes deben crearse dentro del directorio `“/src”` del espacio de trabajo, del siguiente modo:

```
cd ~/robotica_movil_ws/src
catkin_create_pkg practica_1 std_msgs rospy roscpp tf
```

De esta manera se ha creado un paquete de nombre `“practica_1”` que depende de varios paquetes estándar de ROS (mensajes estándar, librerías de C++ y Python, y paquete de transformadas).

Un paquete por defecto contiene los siguientes directorios y archivos de importancia:

- `src/`: directorio donde se encontrarán los archivos fuente pertenecientes al paquete.
- `include/`: directorio donde se encontrarán las cabeceras de los archivos pertenecientes al paquete.
- `msg/`: directorio donde se encontrarán los tipos de mensaje definidos en el paquete.
- `srv/`: directorio donde se encontrarán los servicios definidos en el paquete.
- `launch/`: directorio donde se encontrarán los script de lanzamiento (archivos `.launch`) pertenecientes al paquete.
- `package.xml`: archivo de descripción del paquete requerido para poder compilarlo. Este archivo incluye una descripción del paquete así como sus dependencias.
- `CMakeLists.txt`: archivo para poder compilar haciendo uso de CMake.

### 5.2. Creación de un fichero `.launch`

Los ficheros `.launch` permiten lanzar simultáneamente un conjunto de nodos. A modo de ejemplo, se va a crear a continuación un archivo `“amigobot.launch”`, basado en el fichero `“server_with_map_and_gui.launch”` que permitirá lanzar el simulador con el robot amigobot que se ha creado en un apartado previo.

Para poder ejecutar el archivo correctamente ha de encontrarse dentro de la carpeta `/launch` de un paquete, por lo que creamos el directorio `“~/robotica_movil_ws/src/practica_1/launch/”` y dentro de él el archivo `amigobot.launch` de la siguiente manera:

```
<launch>

<include file="$(find stdr_robot)/launch/robot_manager.launch" />
    Inclusión de otro archivo .launch el cual lanza un nodo que proporciona la descripción
    del robot a ROS
<node type="stdr_server_node" pkg="stdr_server" name="stdr_server"
output="screen" args="$(find stdr_resources)/maps/robocup.yaml"/>
```



Nodo que lanza el mapa con el que trabajarán ROS y STDR. Se pasa la descripción del mapa (.yaml) como parámetro clásico (campo args). Se modifica el mapa respecto al original para representar un mapa de tipo laberinto.

```
<node pkg="tf" type="static_transform_publisher" name="world2map"
args="0 0 0 0 0 0 world map 100" />
```

Nodo que genera la relación entre los marcos de coordenadas map (mapa del simulador) y world (marco de coordenadas raíz). Se indica esta relación como argumentos del nodo y que siguen el formato "x y z yaw pitch roll frame\_origen frame\_destino frecuencia"

```
<include file="$(find stdr_gui)/launch/stdr_gui.launch"/>
```

Inclusión de archivo .launch que lanza la interfaz gráfica del simulador.

```
<node pkg="stdr_robot" type="robot_handler" name="$(anon robot_spawn)"
args="add $(find stdr_resources)/resources/robots/ amigobot.xml 4 8 0"
/>
```

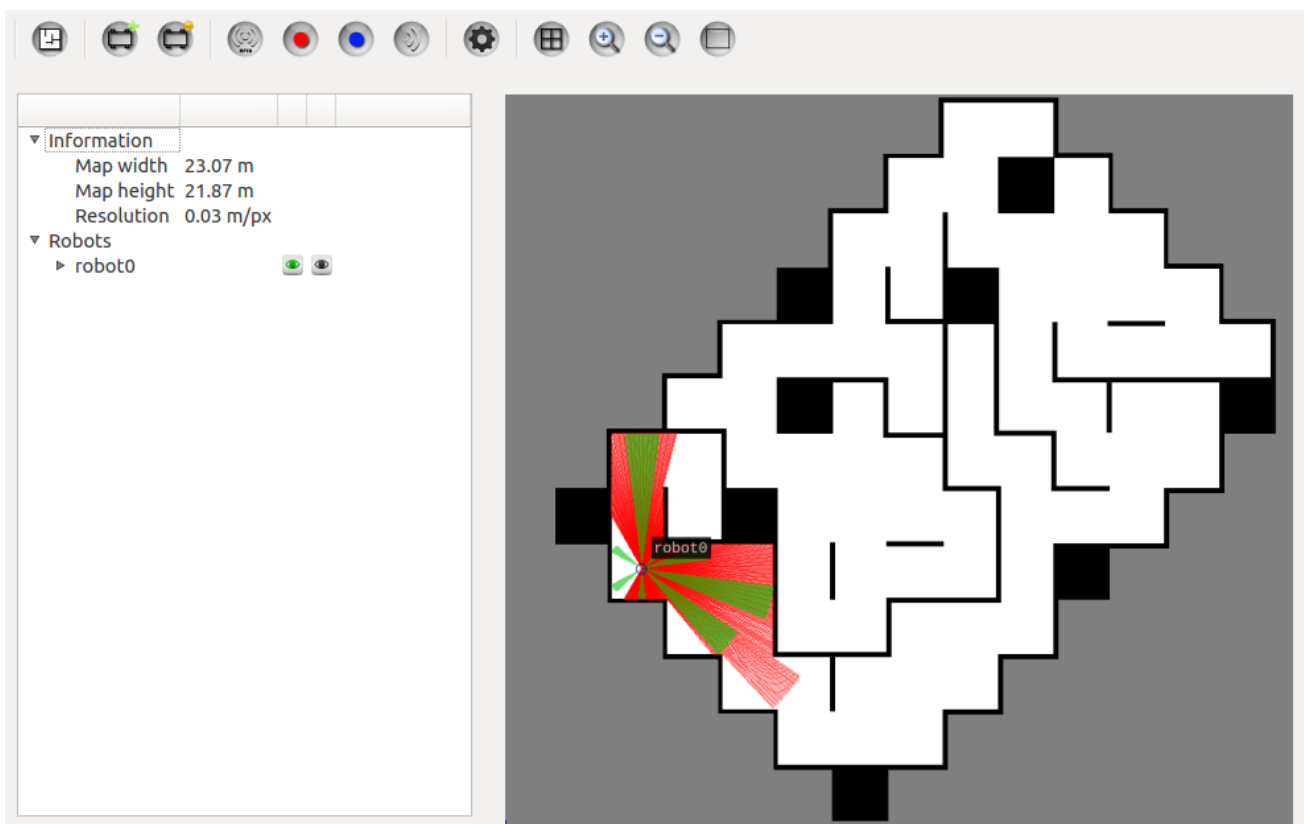
Nodo que carga el robot Amigobot creado en la posición (4,8,0)

```
</launch>
```

Para lanzar esta simulación, ejecutamos el fichero .launch anterior del siguiente modo:

```
roslaunch practica_1 amigobot.launch
```

y observaremos la siguiente pantalla del simulador:



### 5.3. Esqueleto genérico de un programa en C++ (nodo)

A continuación, se muestra el esqueleto básico de programación de un nodo en C++, con los comentarios necesarios para la comprensión del mismo. Este esqueleto puede descargarse de la página web de la asignatura ("esqueleto.cpp"). Básicamente, este programa crea una clase ("practica\_1\_node") que contiene, además de los "subscribers" y "publishers" para los diferentes topics, las llamadas (callbacks) a las funciones que se ejecutan cada vez que se lee un dato en alguno de estos topics. Además, se crea un reloj relacionado con una llamada ("spin"), el cual de forma periódica lleva a cabo la tarea principal de procesamiento del nodo.

```
// INCLUIR LA LIBRERIA DE ROS
#include "ros/ros.h"

// INCLUIR LOS TIPOS DE DATOS A USAR
#include "geometry_msgs/Twist.h"
#include "nav_msgs/Odometry.h"
#include "sensor_msgs/LaserScan.h"
#include "sensor_msgs/Range.h"
#include "tf/tf.h"
```

Para cada tipo de dato que se quiera utilizar, se necesita añadir el include de la librería que lo contiene. En este ejemplo no se utilizan todos los que aparecen anteriormente. El listado de los tipos de datos más comunes se encuentra en: estándar [http://wiki.ros.org/std\\_msgs](http://wiki.ros.org/std_msgs) ; sensores [http://wiki.ros.org/sensor\\_msgs](http://wiki.ros.org/sensor_msgs) ; navegación (mapas, planificadores) [http://wiki.ros.org/nav\\_msgs](http://wiki.ros.org/nav_msgs) . Por otro lado, "tf.h" es la librería de transformadas por si fuera necesaria.

```
// INCLUIR RESTO DE LIBERIAS A UTILIZAR
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
```

```
// DECLARAR LA CLASE
class practica_1_node{
public:
    // DECLARAR EL CONSTRUCTOR
    practica_1_node();
    // DECLARAR LAS FUNCIONES Y CALLBACKS A UTILIZAR
    void odomCallback(const boost::shared_ptr<nav_msgs::Odometry const>& msg);
```

Este callback se asociará (en el constructor de la clase) al subscriber "odom\_sub", de manera que se ejecutará cada vez que llegue un nuevo dato de odometría al topic suscrito. En la declaración del callback ha de indicarse explícitamente el tipo de dato que va a escuchar, en este caso de tipo odometría.

```
// DECLARAR EL CALLBACK CON TEMPORIZADOR
void spin(const ros::TimerEvent& e);
```

Este callback se asociará (en el constructor de la clase) al timer "timer\_", de manera que se ejecutará periódicamente con la frecuencia indicada.

```
private:
    // DECLARAR LOS SUBSCRIBER Y PUBLISHER
    ros::Subscriber odom_sub_;
    Declaración del "subscriber" que se suscribirá al topic de odometría.
    ros::Publisher cmd_vel_pub_;
    Declaración del "publisher" que publicará en el topic de velocidad.
    // DECLARAR EL TIMER
```

```
ros::Timer timer_;
```

Declaración de un “timer” de ROS.

```
};
```

```
// CALLBACK
```

```
void practica_1_node::odomCallback(const boost::shared_ptr<nav_msgs::Odometry  
const>& msg)
```

```
{
```

En este callback hay que escribir lo que se desea que se ejecute cada vez que llegue un dato nuevo al topic de odometría.

```
}
```

```
// CALLBACK DEL TIMER
```

```
void practica_1_node::spin(const ros::TimerEvent& e)
```

```
{
```

En este callback hay que escribir lo que se desea que se ejecute cada vez que llegue un nuevo tic del timer. Aquí suele ejecutarse la base del programa, si se desea que ésta sea periódica.

```
}
```

```
// CONSTRUCTOR DE LA CLASE
```

```
practica_1_node::practica_1_node() {
```

```
    // MANEJADOR DE NODOS DE ROS
```

```
    ros::NodeHandle n("~");
```

Instrucción de ROS necesaria para crear un manejador del nodo de ROS

```
    // SUBSCRIBERS
```

```
    odom_sub_ = n.subscribe<nav_msgs::Odometry> ("/robot0/odom", 1,
```

```
&practica_1_node::odomCallback, this);
```

Esta instrucción inicializa el subscriber “odom\_sub\_” para el nodo “n”. Este subscriber permite escuchar un topic (en este caso “/robot0/odom”) y le asocia un callback que se ejecutará cuando lleguen mensajes al mismo (en este caso “odomCallback”). Debe indicarse el tipo de mensaje del topic, que en este caso es “nav\_msgs::Odometry”. El tamaño de la cola de escucha en este caso es 1, lo cual significa que se queda sólo con el último dato. Si se aumenta este número se pueden almacenar más datos para el caso de que no dé tiempo a procesarlos.

```
    // PUBLISHERS
```

```
    cmd_vel_pub_ = n.advertise<geometry_msgs::Twist> ("/robot0/cmd_vel", 1, true);
```

Esta instrucción inicializa el publisher “cmd\_vel\_pub\_” para el nodo “n”. Este publisher permite publicar mensajes en un topic (en este caso “/robot0/cmd\_vel”) cuyo tipo se indica (“geometry\_msgs::Twist”). El tamaño de la cola de publicación en este caso es 1, lo cual significa que sólo se almacena el último dato publicado.

```
    // TIMER
```

```
    timer_ = n.createTimer(ros::Duration(0.1), &practica_1_node::spin, this);
```

Esta instrucción inicializa el timer “timer\_” para el nodo “n”. Este timer tendrá una frecuencia de 10 Hz (“ros::Duration(0.1)”) y ejecutará el callback “spin” cada vez que se cumpla un periodo del mismo.

```
    // INICIALIZAR VARIABLES
```

```
}
```

```
// PROGRAMA PRINCIPAL
```

```
int main(int argc, char **argv)
{
    // INICIALIZAR ROS
    ros::init(argc, argv, "practica_1_node");
    Creación del nodo de ROS, con nombre "practica_1_node"
    // CARGAR LA CLASE
    practica_1_node pr1_node;
    Se declara e inicializa la clase creada. En este momento se ejecuta el constructor de la
    clase, que crea los subscribers, publishers y timers, y los asocia a sus callbacks.
    // LANZAR EL NODO
    ros::spin();
    Al llegar a esta instrucción el programa queda ejecutándose de manera indefinida, leyendo
    los topics suscritos.
}
```

## 5.4. Ejemplo de programa en C++: mover el robot un metro hacia adelante

En este apartado, y basándonos en el esqueleto anterior, se va a crear un nodo de nombre “practica\_1\_node” que haga avanzar el robot un metro en línea recta. El código de la aplicación debe guardarse en un archivo, dentro de la carpeta “~/robotica\_movil\_ws/src/practica\_1/src”, con nombre “practica\_1\_node.cpp”.

```
// INCLUIR LA LIBRERIA DE ROS
#include "ros/ros.h"

// INCLUIR LOS TIPOS DE DATOS A USAR
#include "geometry_msgs/Twist.h"
#include "nav_msgs/Odometry.h"
#include "sensor_msgs/LaserScan.h"
#include "sensor_msgs/Range.h"
#include "tf/tf.h"

// INCLUIR RESTO DE LIBERIAS A UTILIZAR
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>

// DECLARAR LA CLASE
class practica_1_node{
public:
    // DECLARAR EL CONSTRUCTOR
    practica_1_node();
    // DECLARAR LAS FUNCIONES Y CALLBACKS A UTILIZAR
    void odomCallback(const boost::shared_ptr<nav_msgs::Odometry const>& msg);
    // DECLARAR EL CALLBACK CON TEMPORIZADOR
    void spin(const ros::TimerEvent& e);
private:
    // DECLARAR LOS SUBSCRIBER Y PUBLISHER
    ros::Subscriber odom_sub_;
    ros::Publisher cmd_vel_pub_;
    // DECLARAR EL TIMER
    ros::Timer timer_;
    // DECLARAR LAS VARIABLES A USAR
    nav_msgs::Odometry actualOdom, initOdom;
    geometry_msgs::Twist cmd_vel;
    int init;
    double dist;
    Declaramos las variables que vamos a utilizar para esta aplicación
};

// CALLBACK
void practica_1_node::odomCallback(const boost::shared_ptr<nav_msgs::Odometry
const>& msg)
{
    // SALVAR LA ULTIMA ODOMETRÍA LEIDA
    actualOdom=*msg;
```

```
// CUANDO LA PRIMERA ODOMETRÍA HAYA SIDO LEIDA, ACTIVAR FLAG
if (!init)
{
    init=1;
}
}
```

En el callback de la odometría, simplemente se almacena el mensaje de odometría recibido en la variable de odometría “actualOdom”, para que ésta esté actualizada y pueda ser usada en otros lugares del programa. Además la primera vez que se recibe un dato de odometría se pone la variable “init” a 1 para gestionar una máquina de estados que controla la ejecución del programa dentro del callback del timer “spin”.

```
// CALLBACK DEL TIMER
void practica_1_node::spin(const ros::TimerEvent& e)
{
    // SI SE HA LEIDO AL MENOS UNA ODOMETRÍA, TOMARLA COMO POSICIÓN DE INICIO
    if (init==1)
    {
        initOdom=actualOdom;
        init=2;
    }

    // SI YA TENEMOS LA POSICION DE INICIO, COMENZAMOS EL CONTROL
    if (init==2)
    {
        // CALCULAR LA DISTANCIA RECORRIDA
        dist=sqrt(pow(initOdom.pose.pose.position.x -
actualOdom.pose.pose.position.x,2) + pow (initOdom.pose.pose.position.y -
actualOdom.pose.pose.position.y,2));

        // MOSTRAR POR PANTALLA LA POSICIÓN ACTUAL Y LA DISTANCIA RECORRIDA
        ROS_INFO("ACTUAL POSITION X %f Y %f THETA %f DISTANCE TRAVELLED %f \n",
actualOdom.pose.pose.position.x, actualOdom.pose.pose.position.y,
tf::getYaw(actualOdom.pose.pose.orientation), dist);

        // SI SE HA RECORRIDO MENOS DE UN METRO, MOVER EL ROBOT. EN CASO CONTRARIO
DETENER EL ROBOT
        if (dist<1)
        {
            cmd_vel.linear.x=0.2;
            cmd_vel.linear.y=0.0;
            cmd_vel.angular.z=0.0;
            cmd_vel_pub_.publish(cmd_vel);
        }
        else
        {
            cmd_vel.linear.x=0.0;
            cmd_vel.linear.y=0.0;
            cmd_vel.angular.z=0.0;
            cmd_vel_pub_.publish(cmd_vel);
        }
    }
}
```

```
}
```

```
}
```

En el callback del timer se realiza el control real del programa, gestionado generalmente mediante una máquina de estados. En este caso, hasta que no se ha leído un mensaje de odometría (init=0), no se realiza ninguna acción. Una vez que se ha leído un mensaje de odometría (init=1), se almacena como posición inicial del robot, y se pasa al estado init=2, en el cual periódicamente (cada vez que sucede un evento en el timer) se calcula la distancia avanzada por el robot (distancia entre actualOdom e initOdom) y si esta distancia es menor que un metro, se publica un mensaje de velocidad lineal de 0.2 m/s en el topic "cmd\_vel". En caso contrario, se detiene el robot publicando una velocidad lineal de 0 m/s.

```
// CONSTRUCTOR
```

```
 practica_1_node::practica_1_node() {
```

```
    // MANEJADOR DE NODOS DE ROS
```

```
    ros::NodeHandle n("~");
```

```
    // SUBSCRIBERS
```

```
    odom_sub_ = n.subscribe<nav_msgs::Odometry> ("/robot0/odom", 1,  
&practica_1_node::odomCallback, this);
```

```
    // PUBLISHERS
```

```
    cmd_vel_pub_ = n.advertise<geometry_msgs::Twist> ("/robot0/cmd_vel", 1, true);
```

```
    // TIMER
```

```
    timer_ = n.createTimer(ros::Duration(0.1), &practica_1_node::spin, this);
```

```
    // INICIALIZAR VARIABLES
```

```
    init=0;    El estado inicial de la máquina de estados se pone a 0
```

```
    dist=0;    Inicialmente la distancia recorrida es 0
```

```
}
```

```
// PROGRAMA PRINCIPAL
```

```
int main(int argc, char **argv)
```

```
{
```

```
    // INICIALIZAR ROS
```

```
    ros::init(argc, argv, "practica_1_node");
```

```
    // CARGAR LA CLASE
```

```
    practica_1_node pr1_node;
```

```
    // LANZAR EL NODO
```

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

```
}
```

## 5.5. Compilación y ejecución del nodo en C++

Para poder compilar un paquete (con todos los nodos que contenga) se hace uso de la herramienta catkin. Esta herramienta de ROS ejecuta distintos comandos CMake en cada uno de los paquetes dentro de un workspace.

Para poder compilar nodos es necesario modificar el archivo “CmakeList.txt” dentro del directorio del paquete. Para compilar el primer programa de ejemplo habremos de descomentar las siguientes líneas:

```
add_executable(practica_1_node src/practica_1_node.cpp)
```

Se indica la creación del ejecutable practica\_1\_node a partir del archivo fuente src/practica\_1\_node.cpp

```
target_link_libraries(practica_1_node ${catkin_LIBRARIES})
```

Se indica que compile el ejecutable practica\_1\_node haciendo uso de las librerías catkin por defecto

Esto nos permitirá compilar el nodo “practica\_1\_node” haciendo uso de la instrucción “catkin\_make” desde el directorio raíz del espacio de trabajo. Para añadir nuevos nodos habrían de crearse más líneas en el archivo de manera similar.

Por lo tanto, para compilar el nodo ejecutaremos:

```
cd ~/robotica_movil_ws  
catkin_make
```

Y tras compilarlo, para ejecutarlo (estando ya el simulador abierto), escribiremos:

```
roslaunch practica_1 practica_1_node
```



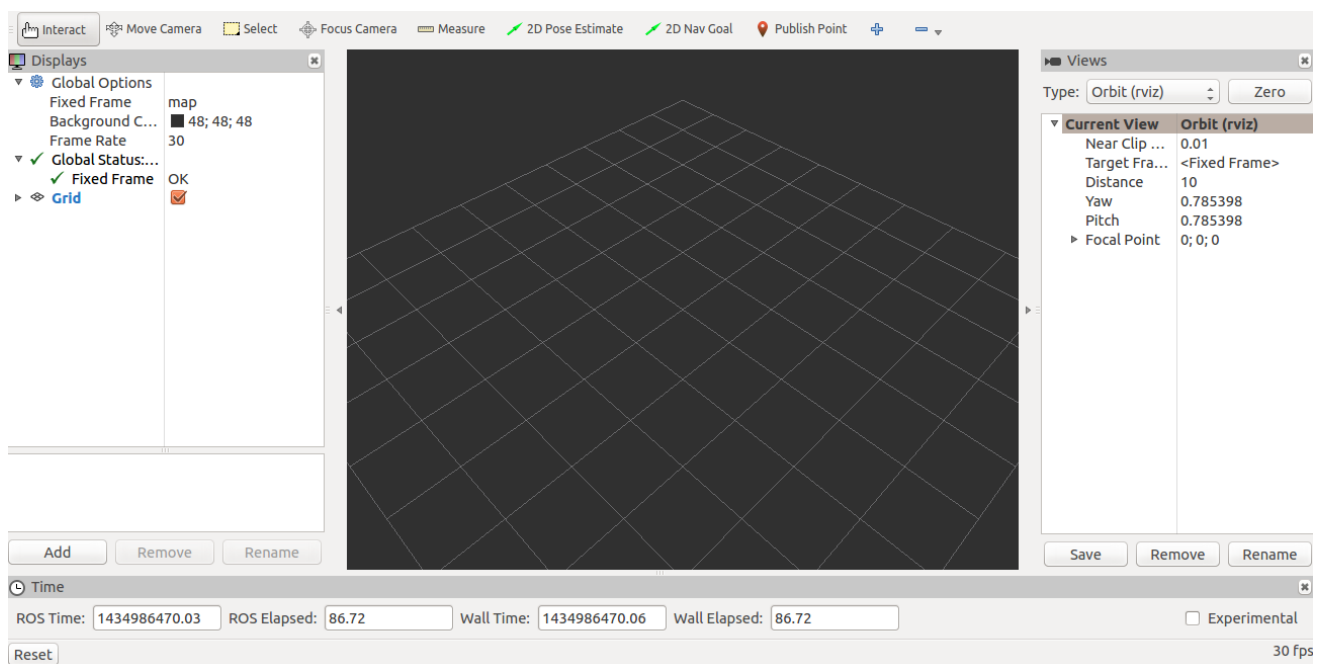
## 6. VISUALIZADOR RVIZ

Rviz es un programa que sirve para visualizar los distintos datos que existen dentro de una ejecución de ROS. Este programa está dotado de plugins que permiten representar los tipos de datos más comunes, tales como odometría, láser, sónar, mapas, etc...

Para ejecutar RViz, con una ejecución de ROS iniciada (nodo roscore activo), solamente ha de teclearse `roslaunch rviz rviz`. En caso de que la máquina sobre la que se esté ejecutando no soporte el software de aceleración OpenGL habría de añadirse antes el comando `export LIBGL_ALWAYS_SOFTWARE=1`

Una vez abierto el programa vemos una imagen como la mostrada en la siguiente figura. Podemos dividir la pantalla en cinco secciones:

- Barra superior: la cual nos permite cambiar entre herramientas para mover la visualización, obtener datos de la misma o incluso enviar algunos comandos de posición.
- Barra inferior: donde se muestra el tiempo de ROS.
- Barra izquierda: en esta barra es donde se añaden los distintos datos a representar.
- Barra derecha: en esta barra se puede cambiar el punto de vista de la visualización.
- Ventana central: en esta ventana se muestra la visualización como tal. Por defecto se muestra una rejilla vacía.

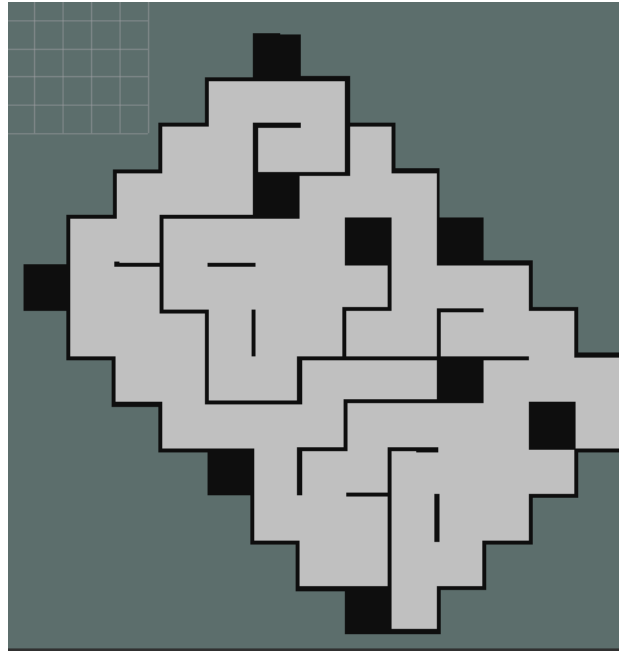


Si nos fijamos en la sección “Displays”, un parámetro importante que se puede modificar es Fixed Frame. Esto indica el marco de coordenadas que se va a utilizar en la visualización. En el caso de nuestra simulación, si queremos representar el mapa, podremos elegir “world”, “map” o “map\_static”.

Para visualizar los distintos datos debemos de pulsar el botón Add. Si conocemos el tipo de dato a representar podemos añadirlo “By Type”, pero es más sencillo añadirlo con la opción “By Topic” puesto que solamente aparecerán los topic activos en el sistema en ese momento.

Algunos de los tipos de datos que se pueden visualizar son los siguientes:

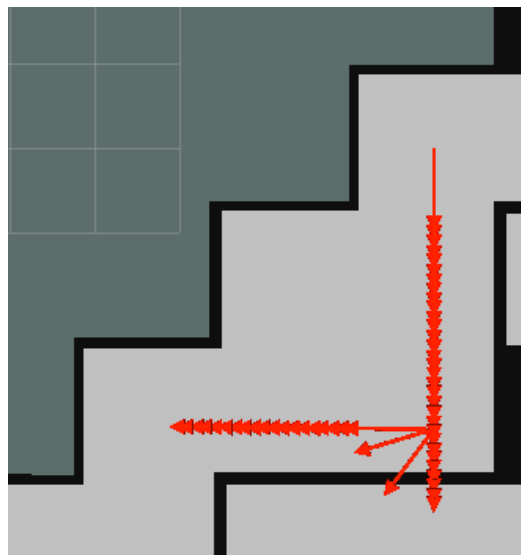
**Map:**



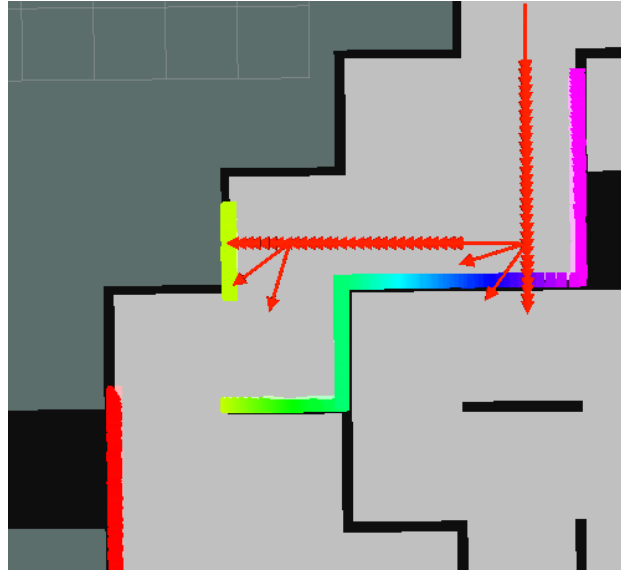
Representación de un mapa de tipo rejilla de ocupación. Permite representar los datos en dos formatos:

- Map: En negro se representan las celdas ocupadas, en gris claro las celdas libres y en gris oscuro las celdas desconocidas.
- Costmap: se visualiza la ocupación de cada celda en distintos colores.

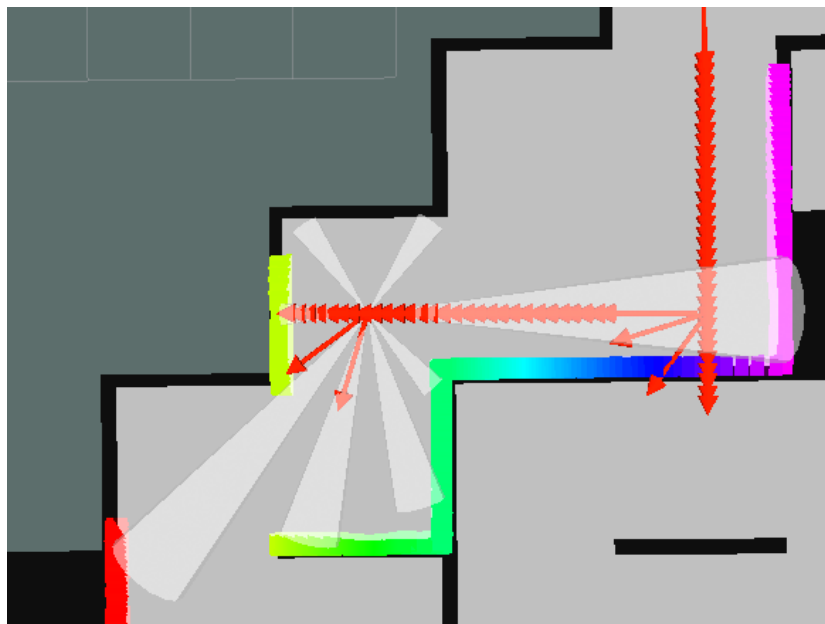
**Odometría (Odometry):** Este tipo de dato permite representar la posición del robot dentro del mapa como una flecha. Para obtener el recorrido del robot se puede variar el número de posiciones acumuladas a representar (variable *Keep*). También se pueden variar el tamaño y color de las flechas para representar distintos robots en el mismo mapa.



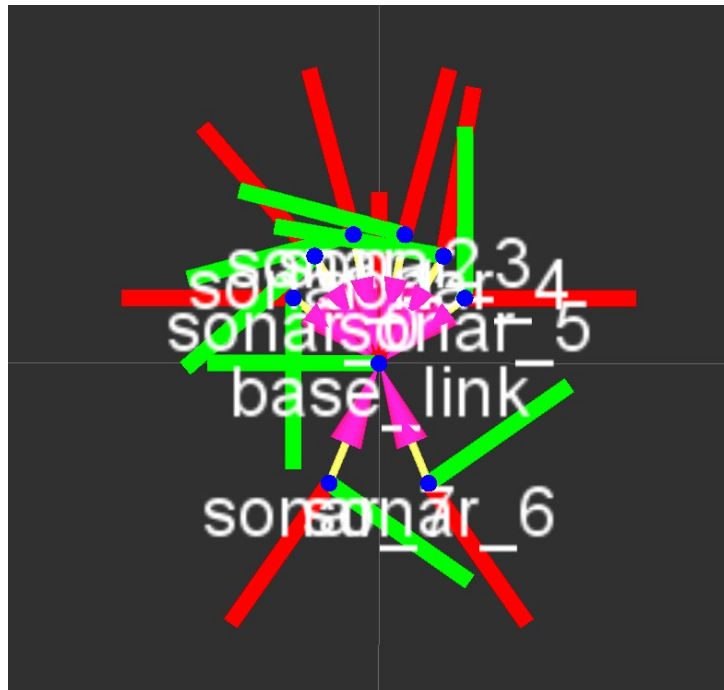
**Láser (LaserScan):** Permite representar las medidas de un sensor láser. Haciendo uso de las transformadas entre marcos de coordenada pinta sobre el mapa los impactos del láser (no representa las medidas máximas del mismo). Mediante sus parámetros podemos modificar el tamaño de los impactos a representar así como colorearlos de distintas maneras (en colores planos, dependiendo del valor en alguno de sus ejes, por intensidad, etc...).



**Sensores de distancia (Range):** Se pueden añadir sensores de distancia sónar o ultrasonidos a la visualización. Éstos se representan como un cono dependiendo de su apertura y con una distancia proporcional a su medida.



**Árbol de transformadas (TF):** Representa el eje de coordenadas de cada uno de los marcos de coordenadas en el sistema así como las relaciones entre ellos. En la imagen vemos la relación entre los sonar y el centro del robot (es algo confuso porque son 8 ejes de coordenadas muy cercanos unos a otros).



Una vez añadidos los Topic a mostrar y modificado el punto de vista al deseado se puede guardar esta configuración mediante el comando “file/save config as” y cargarla de nuevo cuando abramos el visualizador con:

**roslaunch rviz rviz -D “nombre guardado”**

También RViz posee por defecto algunas herramientas con las que se puede interactuar como “2D Pose Estimate” o “2D Nav Goal” que interactúan con los paquetes de navegación que se verán en prácticas posteriores.

Por otra parte, RViz tiene sus limitaciones y no permite visualizar todos los tipos de datos por defecto. Un dato muy habitual es el tipo “posewithcovariance stamped” el cual no puede ser visualizado directamente y por lo tanto se suelen desarrollar nodos intermedios para la visualización o plugins que convierten estos datos a unos que pueda manejar RViz.

## 7. ROS DISTRIBUIDO: SISTEMAS MULTIMÁQUINA

Tal y como se ha mostrado en los apartados anteriores, ROS presenta una arquitectura de programación distribuida, en la que diferentes nodos se comunican entre sí a través de mensajes enviados a los topics. Yendo un poco más allá, ROS permite que estos nodos se distribuyan en diferentes máquinas comunicadas por una red, para ajustarse a los recursos disponibles. Configurar ROS en un sistema multimáquina es sencillo, teniendo en cuenta los siguientes aspectos:

- Sólo es necesario **un único Master de ROS (roscore)**, que se ejecutará en una de las máquinas.
- Todas las máquinas deben conocer dónde se ejecuta el Master, lo cual se indica configurando en todas ellas (incluida la propia máquina en la que se ejecuta) la variable de entorno `ROS_MASTER_URI`, del siguiente modo:

```
export ROS_MASTER_URI=http://IP_ROSMaster_MACHINE:11311
```

donde `IP_ROSMaster_MACHINE` es la dirección IP de la máquina que ejecuta el Máster.

- Si existe conectividad completa entre las máquinas a través de la red, los topics creados por cada una de ellas serán accesibles a todas las demás. Para ello, cada máquina debe comunicar su propia dirección de red dentro del sistema ROS a través de la variable de entorno `ROS_IP`, del siguiente modo:

```
export ROS_IP=IP_LOCAL_MACHINE
```

donde `IP_LOCAL_MACHINE` es la dirección IP de la propia máquina.

Se recomienda configurar las variables de entorno anteriores en el fichero `.bashrc`, editándolo y añadiéndolas al final.

```
sudo gedit ~/.bashrc
```

Añadir al final del fichero estas dos líneas:

```
export ROS_MASTER_URI=http://IP_ROSMaster_MACHINE:11311
export ROS_IP=IP_LOCAL_MACHINE
```

Cada vez que se modifica el fichero `.bashrc` se deben cerrar todas las ventanas de terminales abiertas y volver a abrirlas para que se ejecute de nuevo el fichero `.bashrc` actualizado con las modificaciones.

Para comprobar que la conectividad es correcta, puede ejecutarse un 'rostopic list' en todas las máquinas y comprobar que se tiene acceso a todos los topics creados por todos los nodos en cualquiera de las máquinas.

## 8. PROGRAMACIÓN MATLAB-ROS

MATLAB dispone de una toolbox que permite crear nodos en Matlab/Simulink con conectividad al sistema ROS. Esto permite realizar aplicaciones sencillas de manera más rápida que programando en C++ y compilando en ROS nativo, por lo que se va a utilizar para desarrollar buena parte de las prácticas.

- En versiones anteriores a Matlab R2019b se llama “**Robotics System Toolbox**”
- A partir de la versión Matlab R2019b se llama “**ROS Toolbox**”

Para comprender la conexión entre Matlab y ROS y sus diferencias con la programación en ROS nativo vamos a realizar el mismo programa que mueve el robot un metro en el entorno desde Matlab.

### 8.1. Configuración de la red

Si estamos ejecutando Matlab (con la Toolbox instalada) y ROS en máquinas distintas (en el caso del laboratorio Matlab se ejecutará en el sistema operativo anfitrión Windows y ROS en una máquina virtual bajo Ubuntu), ha de configurarse la red de tal manera que exista conexión entre ellas. Para ello, se recomienda comprobar que todos los ordenadores que se están empleando en el sistema (incluida la máquina virtual) se encuentran en la misma red y todos son visibles a todos (emplear herramienta ping para comprobar la visibilidad). Si se están empleando ordenadores que no son los propios del laboratorio, se recomienda conectarse a la red WiFi identificada como:

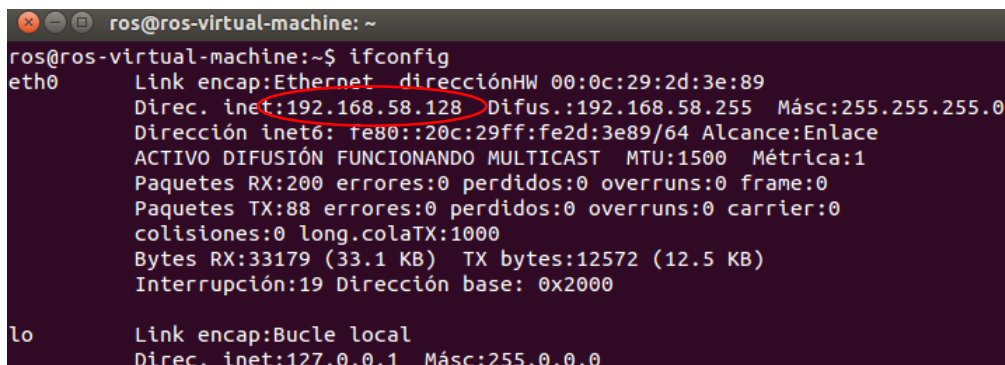
```
Essid: AmigobotWiFi  
Password: Robotical718!
```

El Máster de ROS (roscore) se ejecutará únicamente en una de las máquinas (en nuestro caso en la máquina virtual con el simulador o en el robot real), y el resto de nodos deben conocer la dirección IP de la máquina en la que se ejecuta el máster.

Para conocer la dirección IP actual de la máquina virtual debe abrirse un terminal y ejecutarse el comando:

```
ifconfig
```

Se mostrará una tabla de información, en la que la dirección IP se encuentra en **ethX /enoX -> Direc. Inet**



```
ros@ros-virtual-machine: ~  
ros@ros-virtual-machine:~$ ifconfig  
eth0      Link encap:Ethernet  direcciónHW 00:0c:29:2d:3e:89  
          Direc. inet:192.168.58.128 Difus.:192.168.58.255 Másc:255.255.255.0  
          Dirección inet6: fe80::20c:29ff:fe2d:3e89/64 Alcance:Enlace  
          ACTIVO DIFUSIÓN FUNCIONANDO MULTICAST MTU:1500 Métrica:1  
          Paquetes RX:200 errores:0 perdidos:0 overruns:0 frame:0  
          Paquetes TX:88 errores:0 perdidos:0 overruns:0 carrier:0  
          colisiones:0 long.colatX:1000  
          Bytes RX:33179 (33.1 KB) TX bytes:12572 (12.5 KB)  
          Interrupción:19 Dirección base: 0x2000  
  
lo        Link encap:Bucle local  
          Direc. inet:127.0.0.1 Másc:255.0.0.0
```

Para configurar correctamente estas variables de entorno con la dirección IP actual de la máquina virtual debe editarse el fichero **.bashrc**:

```
sudo gedit ~/.bashrc
```

Al final del fichero hay que añadir las siguientes líneas con la IP obtenida anteriormente:

```
export ROS_MASTER_URI=http://192.168.58.128:11311
export ROS_IP=192.168.58.128
```

Cada vez que se modifica el fichero `.bashrc` se deben cerrar todas las ventanas de terminales abiertas y volver a abrirlas para que se ejecute de nuevo el fichero `.bashrc` actualizado con las modificaciones.

## 8.2. Programa cliente en Matlab

Una vez que hemos configurado la red se ha de escribir el programa cliente en Matlab. Para ello se abre un editor (comando `edit`) y se codificará el archivo `ejemplo_1.m` de la siguiente manera:

```
%% INICIALIZACIÓN DE ROS

% Se definen las variables de entorno ROS_MASTER_URI (ip del Master) y ROS_IP (IP
de la máquina donde se ejecuta Matlab). Si se está conectado a la misma red, la
variable ROS_IP no es necesario definirla.

setenv('ROS_MASTER_URI','http://192.168.11.103:11311')
setenv('ROS_IP','192.168.11.100')

rosinit % Inicialización de ROS

%% DECLARACIÓN DE SUBSCRIBERS
odom=rossubscriber('/robot0/odom'); % Subscripción a la odometría

%% DECLARACIÓN DE PUBLISHERS
pub = rospublisher('/robot0/cmd_vel', 'geometry_msgs/Twist'); %

%% GENERACIÓN DE MENSAJE
msg=rosmesssage(pub) %% Creamos un mensaje del tipo declarado en "pub"
(geometry_msgs/Twist)

% Rellenamos los campos del mensaje para que el robot avance a 0.2 m/s

% Velocidades lineales en x,y y z (velocidades en y o z no se usan en robots
diferenciales y entornos 2D)
msg.Linear.X=0.2;
msg.Linear.Y=0;
msg.Linear.Z=0;

% Velocidades angulares (en robots diferenciales y entornos 2D solo se utilizará
el valor Z)
msg.Angular.X=0;
msg.Angular.Y=0;
msg.Angular.Z=0;
```

```
%% Definimos la periodicidad del bucle (10 hz)
r = robotics.Rate(10);

%% Nos aseguramos recibir un mensaje relacionado con el robot "robot0"
pause(1); % Esperamos 1 segundo para asegurarnos que ha llegado algún mensaje odom,
porque sino la función strcmp da error al tener uno de los campos vacíos.

while (strcmp(odom.LatestMessage.ChildFrameId,'robot0')~=1)
    odom.LatestMessage
end

%% Inicializamos la primera posición (coordenadas x,y,z)
initpos=odom.LatestMessage.Pose.Pose.Position;

%% Bucle de control infinito
while (1)
    %% Obtenemos la posición actual
    pos=odom.LatestMessage.Pose.Pose.Position;
    %% Calculamos la distancia euclídea que se ha desplazado
    dist=sqrt((initpos.X-pos.X)^2+(initpos.Y-pos.Y)^2)

    %% Si el robot se ha desplazado más de un metro detenemos el robot (velocidad
lineal 0) y salimos del bucle
    if (dist>1)
        msg.Linear.X=0;
        % Comando de velocidad
        send(pub,msg);
        % Salimos del bucle de control
        break;
    else
        % Comando de velocidad
        send(pub,msg);
    end

    % Temporización del bucle según el parámetro establecido en r
    waitfor(r)
end

%% DESCONEXIÓN DE ROS
roshutdown;
```

Algunas de las diferencias que encontramos con la programación en ROS nativo en C++ son las siguientes:

- El programa es secuencial sin necesidad de declaración de clases.
- El comando `ros::init` para conectar con el sistema ROS es sustituido por el comando `rosinit()` al inicio del programa.
- Podemos subscribirnos a un topic sin necesidad de generar un callback (internamente Matlab genera el hilo correspondiente a la lectura de ese dato).



- Podemos generar un mensaje del tipo declarado en el topic mediante la instrucción `rosmesssage` añadiendo el subscriber/publisher de dicho topic.
- Los mensajes tienen los mismos campos que en ROS pero el nombre de los mismos inicia en mayúscula.
- Podemos acceder al último dato leído en un Subscriber mediante la instrucción `subscriber.LatestMessage`
- Para ejecutar un bucle de forma periódica añadiremos las instrucciones `r=robotics.Rate(ratio)` y `waitfor(r)` dentro del bucle.
- La instrucción para enviar un mensaje en un publisher es `send(publisher,mensaje)`.
- Al acabar el programa necesitamos desconectarnos del sistema ROS mediante la instrucción `roshutdown`. En caso de que nuestra ejecución haya quedado interrumpida o no pueda terminar necesitaremos ejecutar ese comando en Matlab antes de iniciar una nueva conexión.
- Las herramientas “`roscpp`”, “`rostopic`”, “`rosservice`”, etc, pueden ejecutarse en la ventana de comandos de Matlab de la misma forma que en el terminal de Ubuntu de la Máquina Virtual.

### 8.3. Ejecución

Para ejecutar este programa cliente necesitaremos lanzar el simulador en Ubuntu y una vez lanzado, ejecutar el programa en Matlab. Matlab no necesita que compilemos previamente el programa ni que generemos un paquete agilizando la creación de programas sencillos para ROS.

## 9. CREACIÓN DE MODELOS SIMULINK-ROS

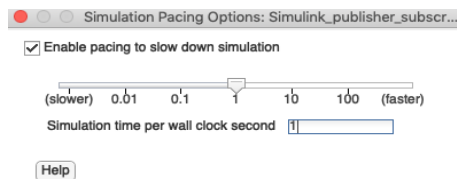
Simulink dispone de una toolbox que permite crear diagramas en Simulink con conectividad al sistema ROS.

- En versiones anteriores a Matlab R2019b se llama “**Robotics System Toolbox**”
- A partir de la versión Matlab R2019b se llama “**ROS Toolbox**”

Se debe realizar la misma **configuración de red** que en el caso de Matlab-ROS

Para comprender la conexión entre Simulink y ROS vamos a realizar el mismo programa que mueve el robot un metro en el entorno desde Simulink.

- Debemos habilitar la opción “**Simulation Pacing**” en la flecha debajo de la tecla Run.

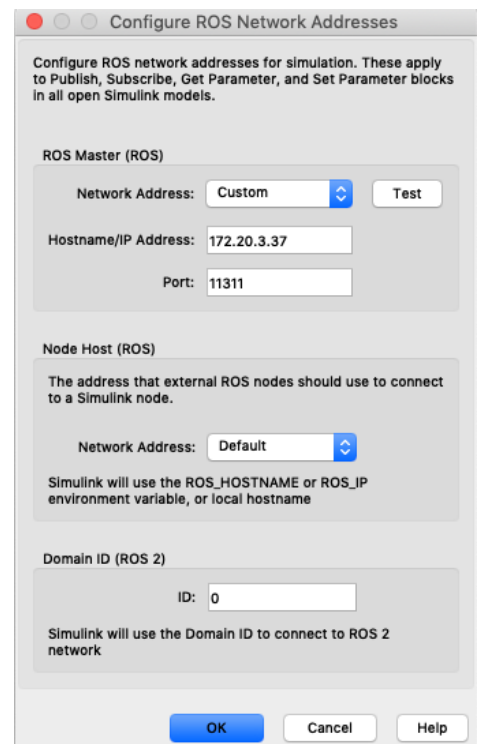
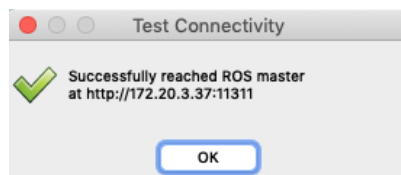


- Configurar el **Time Step**. En este caso 0.1s (10Hz como en los ejemplos anteriores)



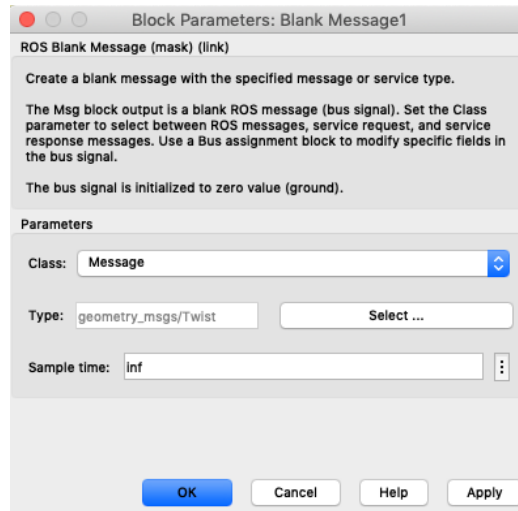
- Configurar la conexión con el simulador / Robot donde se ejecuta el Roscore:
  - SIMULATION / PREPARE / ROS Network:
    - **Network Address:** Custom
    - **IP Address:** Dirección IP del simulador / Robot.
    - **Port:** 11311 (puerto por defecto)

Una vez configurado, podemos comprobar que tenemos conexión con Roscore pulsando el botón “Test”:

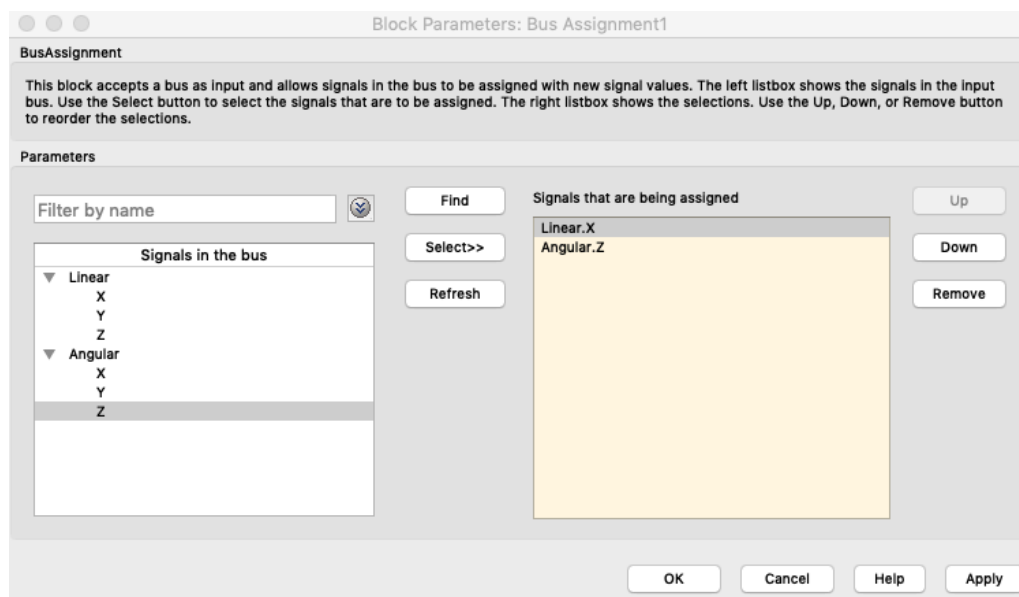


## 9.1. Creación de un *Publisher*

1. Añadimos un bloque “**Blank Message**”
  - Seleccionamos que tipo de mensaje nos queremos publicar, en este caso ‘/geometry\_msgs/Twist’

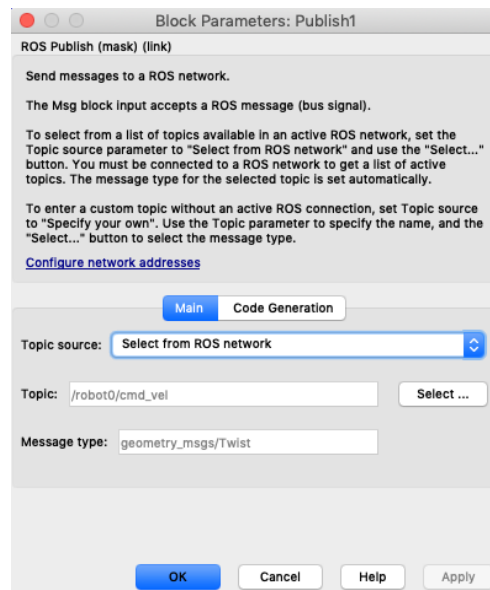


2. Añadimos un bloque “**Bus Assignment**”
  - Seleccionamos que señales utilizaremos, en este caso Linear.X y Angular.Z

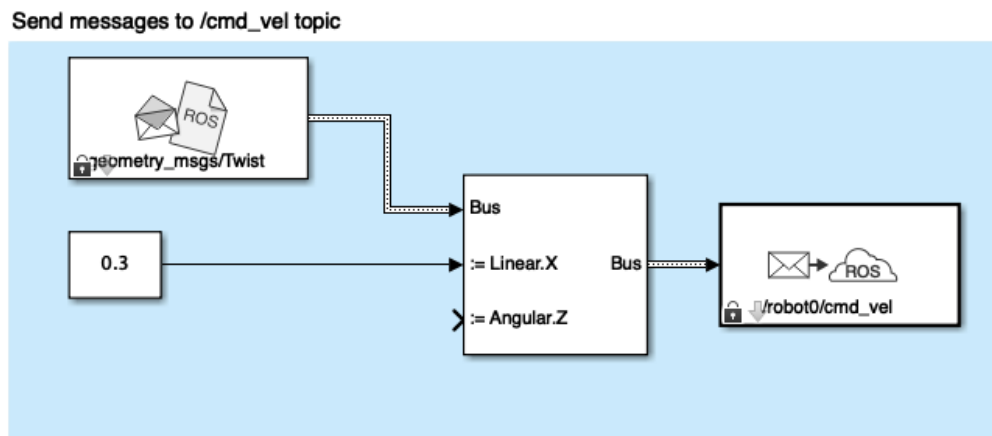


### 3. Añadimos un bloque **"Publish"**

- Seleccionamos en que topic de los disponibles deseamos publicar. En este caso `/robot0/cmd_vel`

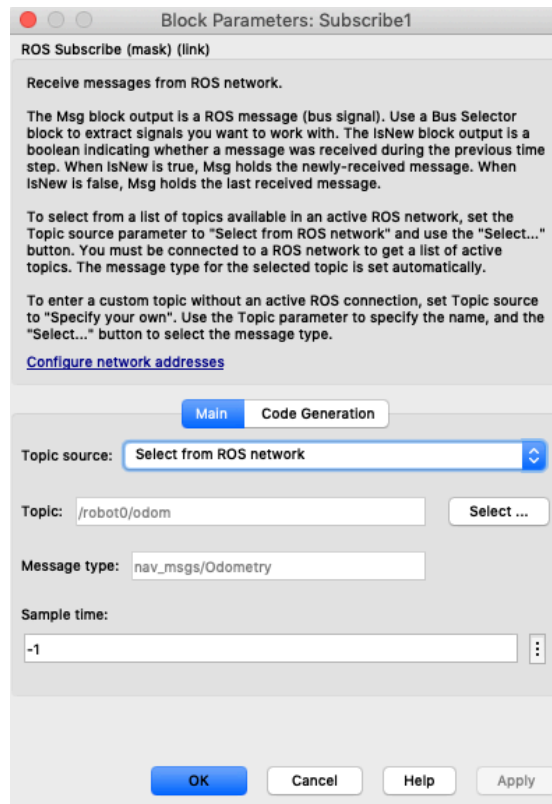


Quedando el diagrama de bloques del Publisher como se muestra a continuación:



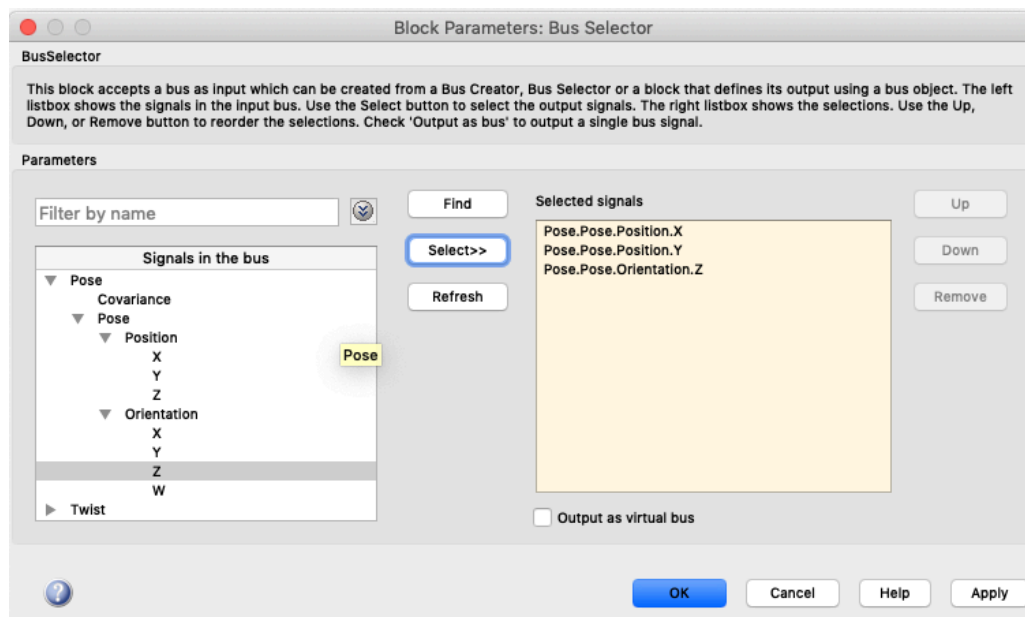
## 9.2. Creación de un *Subscriber*

### 1. Añadimos un bloque “*Subscribe*”



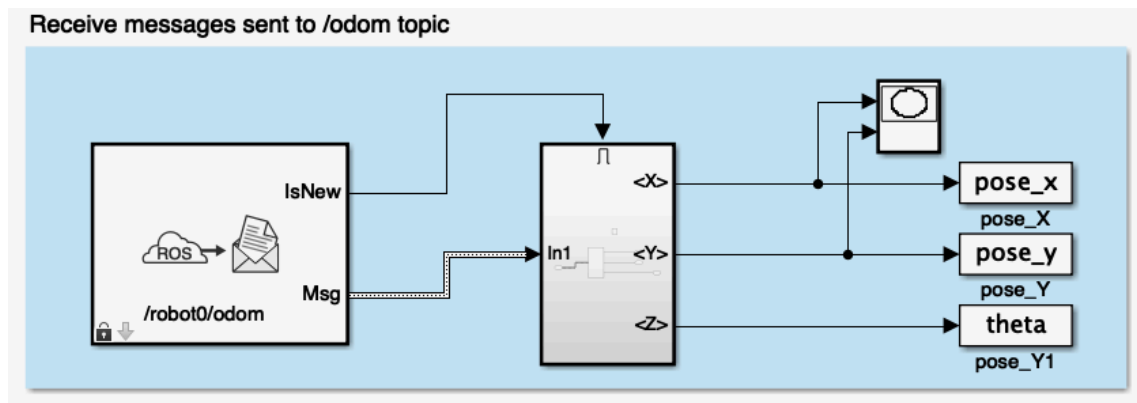
### 2. Añadimos un bloque “*Bus Selector*”

- Seleccionamos que señales utilizaremos, en este caso:
  - Pose.Pose.Position.X
  - Pose.Pose.Position.Y
  - Pose.Pose.Orientation.Z



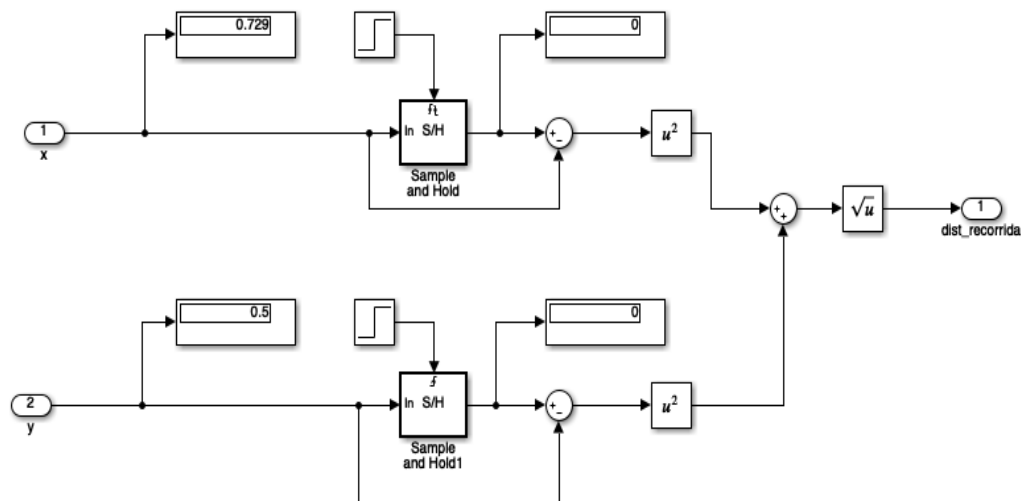
3. Creamos un Subsistema con el bloque “Bus Selector”
4. Añadimos un bloque “Enable” a este subsistema
5. Conectamos la salida “IsNew” del bloque “Subscribe” a la entrada de enable que se crea en el subsistema.
6. Si queremos representar las salidas añadimos un bloque “XY Graph”
7. Si queremos trabajar con estas variables en Matlab, añadiremos 3 bloques “To Workspace” para llevar estas señales X, Y y theta a Matlab.

Quedando el diagrama de bloques del Subscriber como se muestra a continuación:

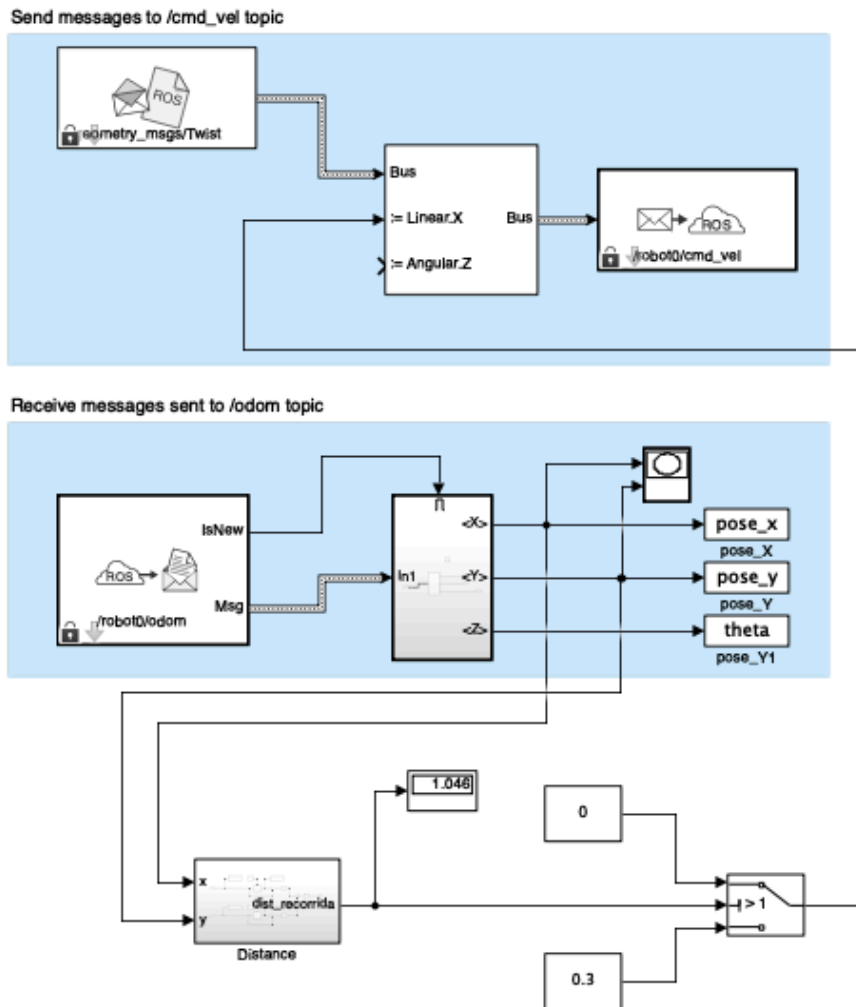


### 9.3. Ejemplo de diagrama Simulink: mover el robot un metro hacia adelante

Al ejemplo del Publisher y Subscriber hay que añadir el siguiente diagrama de bloques que calcula la distancia recorrida por el robot.



Generamos un subsistema con el diagrama anterior llamado “Distance”, quedando el diagrama de bloques de Simulink completo que mueve el robot 1 metro de la siguiente manera:

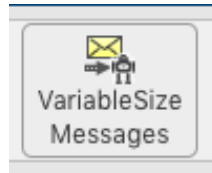


**NOTA:** Si la velocidad angular ( $w$ ) que enviamos desde Simulink al simulador STDR es muy pequeña, el robot se para. Por tanto, debemos filtrarla para que cuando sea muy pequeña, mandar  $w = 0$ .

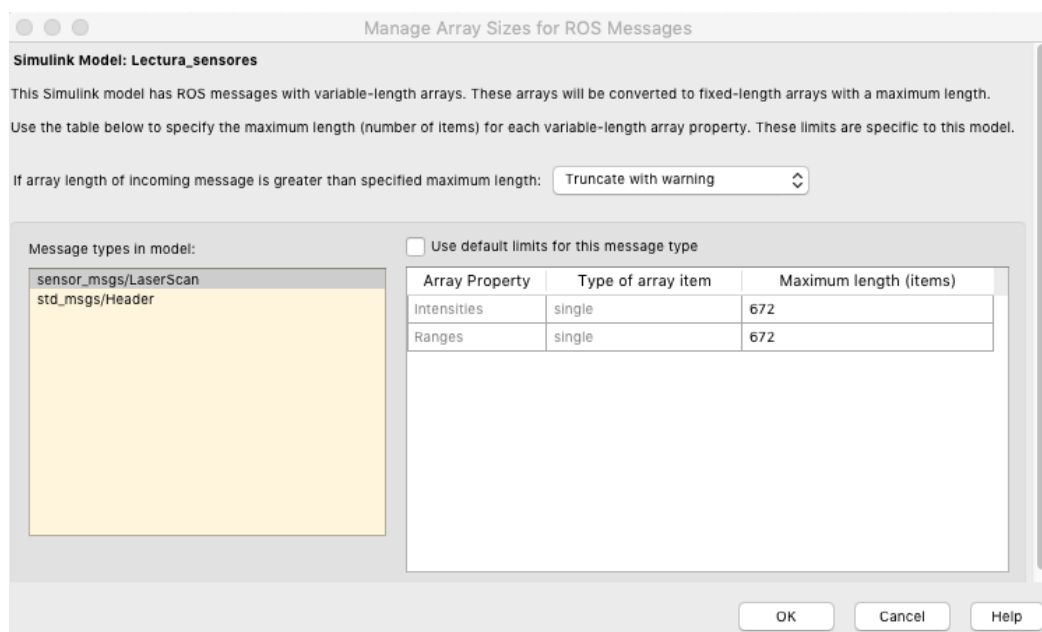
## 9.4. Ejemplo de diagrama Simulink: lectura de sensores

Para leer los sensores en Simulink, basta con leer el topic correspondiente creando un *Subscriber* y un “*Bus Selector*” como en el apartado 9.2 de esta guía.

En Simulink, los mensajes se guardan en arrays que tienen un tamaño máximo por defecto. Por ejemplo, *sensor\_msgs/LaserScan* tiene un tamaño máximo de 128. Por tanto, para poder leer todos los datos del laser Hokuyo (por ejemplo) que tiene 672 medidas, debemos cambiarlo desde el menú “VariableSize Messages”



Y modificar el tamaño máximo de cada mensaje:





## 10. PRÁCTICA: SIMULACIÓN EN MATLAB

Se desea desarrollar un **conjunto de scripts en Matlab** que permitan teleoperar de forma sencilla el robot así como leer y representar gráficamente las lecturas de los sensores proporcionadas por el simulador.

Para ello, se pide:

1. Realizar dos scripts, de nombre *conectar.m* y *desconectar.m* para iniciar y terminar respectivamente la conexión con el sistema ROS cuyo master se ejecuta en la VM.
2. Realizar un script de inicialización de variables, llamado *ini\_simulador.m* que realice las siguientes inicializaciones:
  - Declaración de subscribers (a odometría, láser y sonares)
  - Declaración de publishers (para envío de velocidades)
  - Declaración de los mensajes a utilizar
  - Definir la periodicidad del timer
  - Asegurarse de que recibe algún mensaje del simulador
3. Realizar un script de lectura de sensores, llamado *lee\_sensores.m* para realizar la lectura del láser y de los sonares:
  - En ambos casos, realizar la lectura utilizando la función 'receive'
  - Mostrar gráficamente los datos del láser utilizando la función 'plot'
  - Mostrar por pantalla las lecturas de los sonares
3. Realizar un script llamado *avanzar.m* que haga avanzar el robot la distancia indicada en la variable 'd'. Para ello:
  - Es necesario crear un bucle que lea la odometría y compruebe la distancia avanzada hasta el momento.
  - Opcionalmente puede llamarse al script *lee\_sensores* dentro del bucle para comprobar las lecturas realizadas.
4. Crear un script llamado *girar.m* que haga girar el robot el ángulo indicado en la variable 'th'.
  - Necesario crear un bucle que lea la odometría y compruebe el ángulo girado hasta el momento.
  - Utilizar la función 'quat2eul' para obtener la orientación a partir del cuaternio que proporciona el topic.
  - Opcionalmente puede llamarse al script *lee\_sensores* dentro del bucle para comprobar las lecturas realizadas.

**NOTA:** hay que tener en cuenta que el topic odom devuelve la orientación en formato de cuaternión, por lo que se recomienda hacer uso de la función de Matlab **quat2eul** y obtener la orientación del robot (respecto del eje Z) del primer campo devuelto por la función tal y como se muestra en el siguiente ejemplo.

```
quaternion=[odom.LatestMessage.Pose.Pose.Orientation.W  
odom.LatestMessage.Pose.Pose.Orientation.X  
odom.LatestMessage.Pose.Pose.Orientation.Y  
odom.LatestMessage.Pose.Pose.Orientation.Z];  
  
euler=quat2eul(quaternion,'ZYX');  
Theta=euler(1);
```

(Cuidado al copiar y pegar, ya que los retornos de carro pueden dar problemas)

## 11. CONEXIÓN A UN ROBOT REAL

Dada la naturaleza de ROS hay que tener en cuenta que el código creado puede ser ejecutado en cualquier plataforma. Por lo tanto, el código creado para el simulador puede ser ejecutado en un robot real siempre que se disponga del mismo tipo de sensores.

Para realizar las prácticas de laboratorio se dispone de un robot Amigobot de la familia Pioneer. Este robot, con sistema de tracción diferencial, incorpora sensores de 500 p/rev y 8 sensores de ultrasonidos con un rango de detección de 5m. Además, incorporan un microcontrolador con sistema operativo propio (ARCOS) que se encarga básicamente de procesar la información de estos sensores y gestionar la comunicación con un procesador externo a través de un puerto serie.

A cada robot se le ha incorporado como procesador externo una tarjeta Raspberry PI con Linux y ROS, a la cual se ha conectado un sensor láser RPLIDAR-A2. La Raspberry PI de cada robot tiene configurada una dirección IP fija para poder conectarse en red con cualquier otro ordenador externo (puestos del laboratorio, ordenadores portátiles, etc.)

En el arranque del robot, el sistema operativo ejecuta un script que configura y lanza de forma automática un Master de ROS (roscore), así como los “drivers” de ROS necesarios para controlar el robot y acceder a los datos de los sensores. Estos drivers son, principalmente:

- “**p2os**”: driver que gestiona la comunicación con el microcontrolador del Amigobot, para enviar comandos de velocidad, y para leer la odometría y los sonares. Algunos de los topics más importantes creados por este driver son:

/pose	Odometría
/cmd_vel	Comandos de velocidad
/cmd_motor_state	Habilitación de los motores
/sonar_i	Datos del sonar i (i desde 0 hasta 7)

- “**rplidar**”: driver que accede a los datos del láser, dejándolos en el siguiente topic:

/scan	Laser
-------	-------

## 12. PRÁCTICA: ROBOT REAL

### 12.1. PRÁCTICA: Robot Real en Matlab-ROS

Finalmente, se van a probar los scripts de teleoperación y lectura de sensores que se programaron en el apartado PRÁCTICA: SIMULACIÓN EN MATLAB sobre el robot real.

Para ello, los únicos cambios que hay que realizar son los siguientes:

1. Configurar correctamente el sistema distribuido (variables de entorno y función 'rosinit' de Matlab).
2. Sustituir el fichero 'ini\_simulador.m' por otro llamado 'ini\_robot.m' que se suscriba a los topics creados por el robot. Además, en el robot real es necesario habilitar los motores antes de empezar a desplazarse por el entorno, para lo cual habrá que crear un Publisher y un mensaje de enable/disable de los motores como se muestra a continuación.

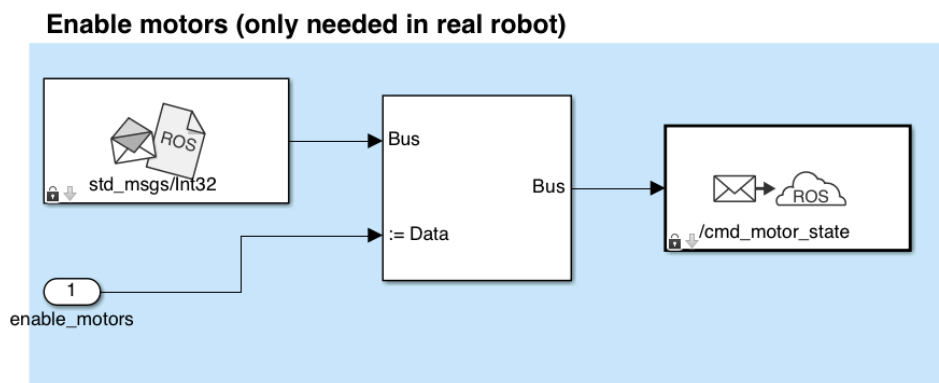
```
%publisher
pub_enable=rospublisher('/cmd_motor_state','std_msgs/Int32');
%declaración mensaje
msg_enable_motor=rosmessage(pub_enable);
%activar motores enviando enable_motor = 1
msg_enable_motor.Data=1;
send(pub_enable,msg_enable_motor);
```

3. El resto de los scripts no necesitan ser modificados.

### 12.2. PRÁCTICA: Robot Real en Simulink-ROS

Para probar el ejemplo de Simulink en el robot real, debemos realizar son los siguientes cambios:

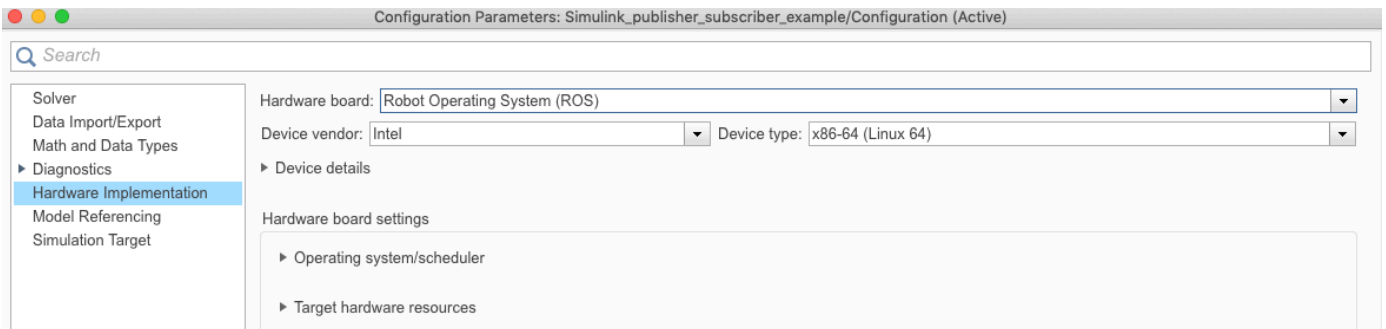
1. Configurar correctamente el sistema distribuido (variables de entorno y la configuración de red de Simulink-ROS).
2. Añadir la habilitación de los motores como se muestra el siguiente diagrama:



## 13. Generar código ROS C++ a partir de Simulink

Si queremos que Simulink convierta el diagrama en código C++, lo transfiera y ejecute en la máquina destino (máquina virtual o robot real), debemos configurarlo de la siguiente manera:

1. Instalar la toolbox “Simulink Coder”
2. Configurar la plataforma de destino:
  - Menú ROBOT / Hardware Settings
  - Seleccionar Hardware Board: Robot Operating System (ROS)



3. Conectarnos a la plataforma de destino:

Menú ROBOT / Connect to ROS device:

- **IP:** IP de la máquina virtual o del Robot real
- **User:** alumno (si nos conectamos a la máquina virtual)
- **Password:** alumno (si nos conectamos a la máquina virtual)

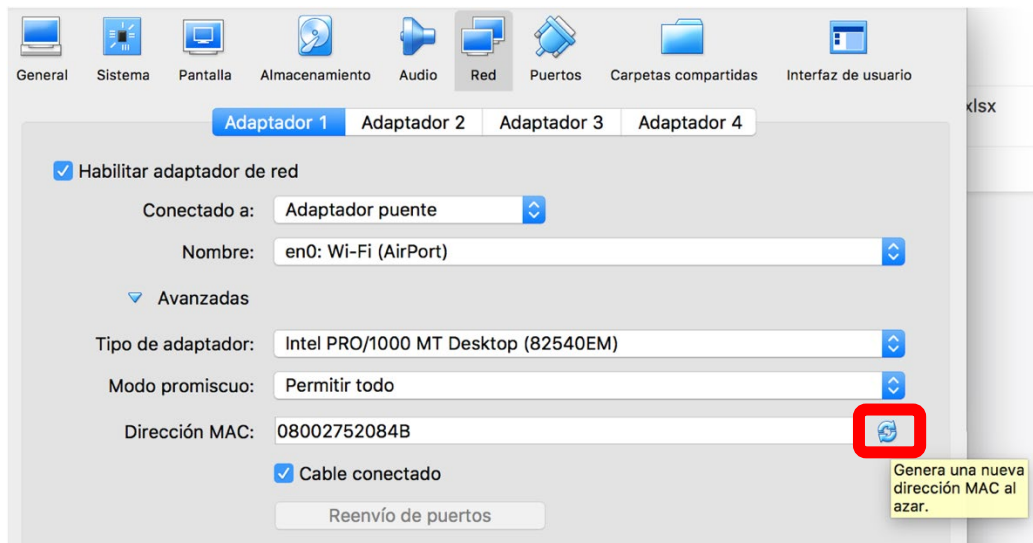


4. Una vez configurado, cada vez que ejecutemos el diagrama de Simulink, convierte dicho diagrama en código C++, lo transfiere y ejecuta en la máquina destino (máquina virtual o robot real).

## 14. ANEXO - FAQs

- **Cambiar dirección MAC de la Máquina Virtual.**

Debido a que VirtualBox genera la misma dirección MAC para todas las instancias de la máquina virtual, varios ordenadores terminen teniendo la misma dirección IP, provocando problemas importantes de diversa índole en la red. Para resolver este problema, es necesario entrar en la configuración de red de VirtualBox, en la sección Avanzadas y generar una nueva dirección MAC al azar, tal y como indica la siguiente figura:



- **Aumentar memoria de video por encima de 128Mb**

Para mejorar el rendimiento de la máquina virtual es recomendable aumentar la memoria de video. Para aumentar la memoria de video por encima de 128Mb debemos abrir el archivo "**Ubuntu16\_04\_ROS.vbox**" (que se genera una vez importada la máquina virtual) con un editor de texto y cambiar el valor de la etiqueta:

```
<Display VRAMSize="128"/>
```

Por ejemplo a 256:

```
<Display VRAMSize="256"/>
```

Este archivo se encuentra en el siguiente directorio según el sistema operativo anfitrión:

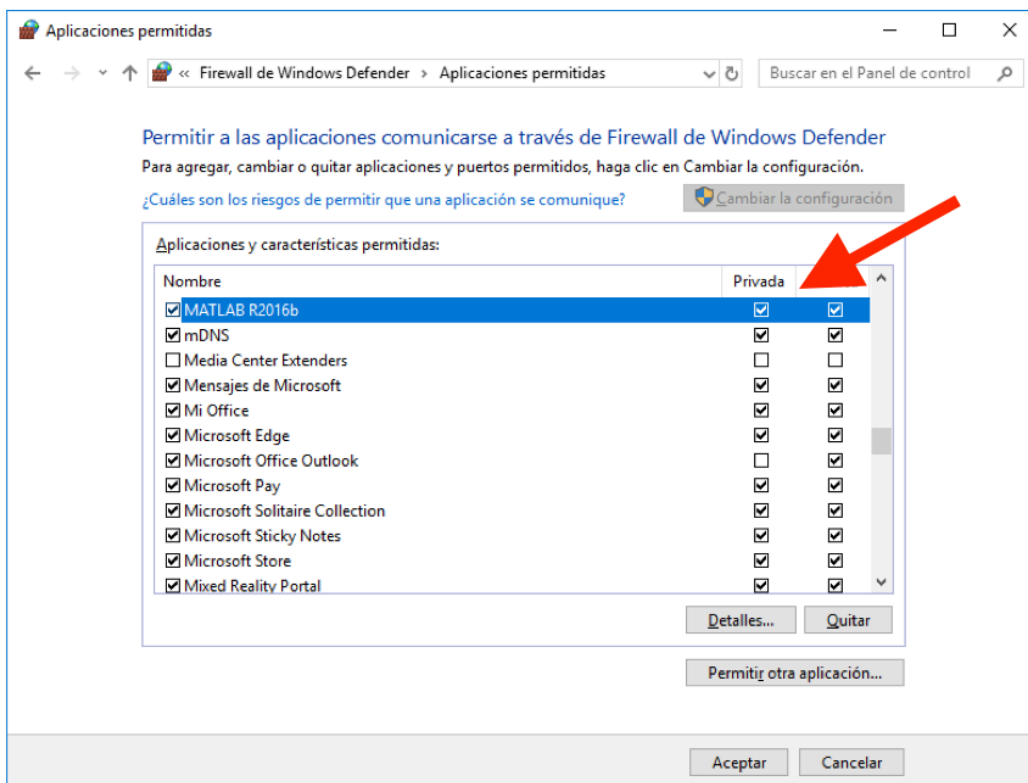
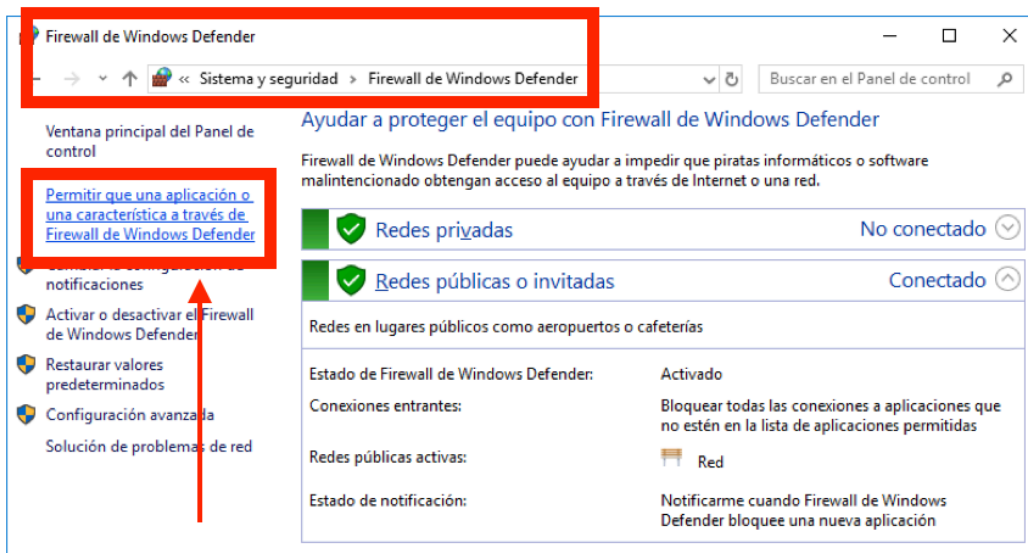
- **Linux and Oracle Solaris:** \$HOME/.config/VirtualBox.
- **Windows:** \$HOME/.VirtualBox.
- **Mac OS X:** \$HOME/Library/VirtualBox.

- **Instalar “Guest Additions”**

- Dentro de la máquina virtual, seleccionar en el menú superior **Dispositivos / Insertar imagen de Cd de las “Guest Additions”**
- Seleccionar “Run”

- **Dar permisos a Matlab en el Firewall Windows en Redes públicas y privadas.**

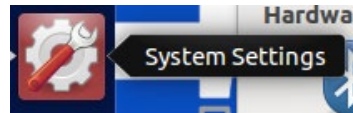
Algunos usuarios han experimentado problemas al publicar topics como la velocidad (cmd\_vel). En estos casos, es posible leer desde Matlab los topics publicados por el robot o el simulador, pero no publicar topics (como la velocidad del robot o el simulador, por ejemplo). Esta situación se ha detectado en ordenadores con Windows 10 y el problema lo está generando el Firewall de Windows Defender. Para resolverlo, es necesario dar permiso a Matlab para comunicar a través del Firewall tanto en redes privadas como públicas, tal y como muestra la siguiente imagen:



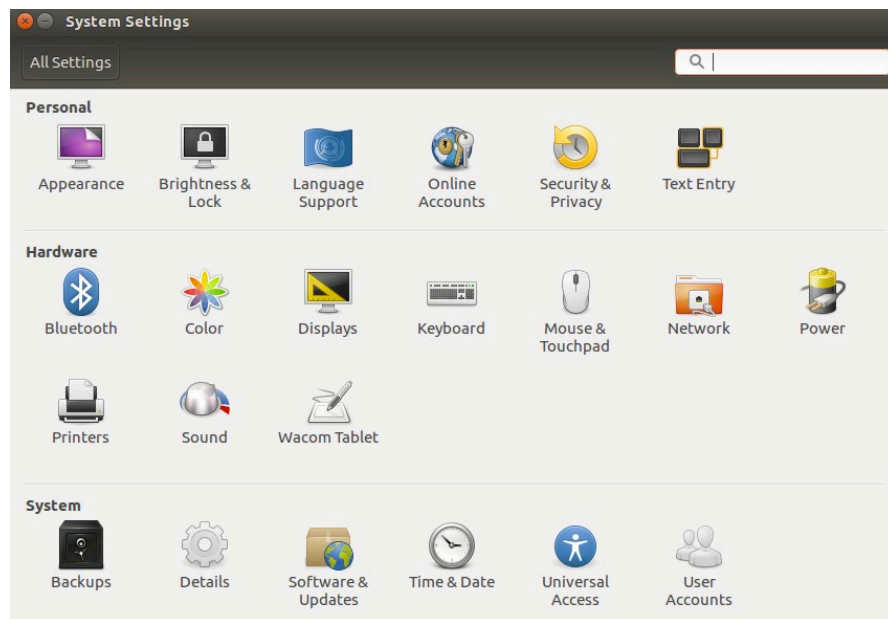
- **Cambiar la resolución de la pantalla dentro de la máquina virtual.**

Para un mejor funcionamiento de la máquina virtual es recomendable disminuir la resolución de la pantalla. Una resolución de 1024x768 puede ser suficiente.

1. Seleccionamos "System Settings":



2. Seleccionamos "Displays":



3. Cambiamos la resolución en el desplegable "Resolution":

