# Pre-requisitos

*Se han creado una serie de tutoriales para introducir al alumno en los primeros pasos para instalar Linux en un PC, instalar ROS, y configurar el entorno. Se recomienda ver y seguir estos vídeos antes de continuar con el resto del documento.*

***1º Instalar Ubuntu en una máquina virtual***

a) ***Usando VMWare:***
*https://www.dropbox.com/s/fybdng1wdto5dgy/Instalar%20Ubuntu%2014.04%20en%20VMware-.mov?dl=0*

b) ***Usando VirtualBox:***
*https://www.dropbox.com/s/wsb9vmvoeinkvt1/Instalar%20Ubuntu%2014.04%20en%20VirtualBox.mov?dl=0*

***2º Instalar ROS en Ubuntu:***
*https://www.dropbox.com/s/od8h91u8wv1e7po/instalando_ROS.mov?dl=0*

***3º Primeros pasos con ROS:***
*https://www.dropbox.com/s/4ihouro1ke45h6d/Primeros%20pasos%20con%20ROS.mov?dl=0*

# Instalar ROS Indigo en Ubuntu

*Añadir los repositorios de ROS a Ubuntu*

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_releas
e -sc) main" > /etc/apt/sources.list.d/ros-latest.list'


sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net --recv-
key 0xB01FA116


sudo apt-get update
```

*Instalar la versión completa*

```
sudo apt-get install ros-indigo-desktop-full
```

*Inicializar rosdep*

```
sudo rosdep init

rosdep update
```

*Instalar el instalador de paquetes ROS de Python*

```
sudo apt-get install python-rosinstall
```

*Es conveniente para trabajar con ROS usar un IDE que facilite la tarea. Es recomendable por su sencillez el QtCreator. Para ello:*

```
sudo apt-get install qtcreator
```

*Añadir variables de entorno en Ubunu, para ello, en el fichero* ***.bashr*** *que se encuentra en vuestra carpeta home (comando "cd" y pulsar enter), se deben añadir al final del fichero, los comando que se muestran a continuación (donde pone lualobus sustituir por vuestro nombre de usuario).*

*Este archivo de sistema oculto (.bashrc) se puede editar con cualquier editor como gedit, vim, nano, etc. Las líneas se escriben al final del fichero. Después de guardar y salir se puede recargar el fichero reseteando el terminal (comando reset) o abriendo un nuevo terminal. También se puede recargar el archivo .bashrc escribiendo: . .bashrc o bien: source .basrhc.*

```
source /opt/ros/indigo/setup.bash
source /home/lualobus/catkin_ws/devel/setup.bash
export ROS_WORKSPACE=/home/lualobus/catkin_ws
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$ROS_WORKSPACE
export ROSCONSOLE_FORMAT='[${severity}] [${time}]: ${message}'
```

*Crear workspace:*

```
mkdir -p ~/catkin_ws/src

cd ~/catkin_ws/src
```

*Iniciar el workspace*

```
rospack profile

roscd

catkin_init_workspace
```

*Primera compilación del workspace:*

```
cd ~/catkin_ws/

catkin_make

source devel/setup.bash
```

*Asegurarnos de que la variable ROS_PACKAGE_PATH apunta al workspace de catkin que acabamos de crear:*

```
echo $ROS_PACKAGE_PATH

/home/youruser/catkin_ws/src:/opt/ros/indigo/share:/opt/ros/indigo/st
acks
```

*Para crear nuestro propio paquete catkin:*

```
cd ~/catkin_ws/src

catkin_create_pkg my_package std_msgs rospy roscpp
```

*Para compilar nuestro paquete creado tenemos varias opciones:*

```
cd ~/catkin_ws

a) catkin_make  → compila todo el workspace
b) catkin_make --pkg my_package  → compila sólo el paquete y todas l
   as dependencias posibles
c) catkin_make --only-pkg-with-deps my_package → compila sólo el paq
   uete y las dependencias mínimas
```

*Para limpiar compilaciones anteriores:*

```
cd ~/catkin_ws

rm build devel install
```

# Usando un paquete de ejemplo

*Descargarmos los paquetes de ejemplo de ROS*

```
$ sudo apt-get install ros-indigo-ros-tutorials
```

*Lanzamos el roscore*

```
$ roscore
```

*Vamos a trabajar con el paquete de ejemplo de la Tortuga (turtlesim).*

```
$ roscd turtlesim
$ rosrun turtlesim turtlesim_node
```

- *Probamos todos los comandos vistos en las diapositivas vistas.*

*rosnode list, rostopic  list, rosnode ping…*

*Debemos haber descubierto que el paquete tiene:*

- *Un nodo en ejecución que se llama turtlesim*
- *Se usan tres topics llamados: color_sensor, command_velocity, pose*

*Ahora vamos a poner en ejecución otro nodo de ese paqute y usamos las flechas para mover la tortuga*

```
$ rosrun turtlesim turtle_teleop_key
```

*En otro terminal (Ctr+T) vamos a ver que se ha registrado el nuevo nodo que hemos lanzado (teleop_turtle)*

```
$ rosnode list
```

*Evidentemente el nodo teleop_turtle está emitiendo un topic con la información de la tecla pulsada, mientras que el nodo turtlesim_node está subscrito a dicho topic y cuando lo recibe cambia la posición en la pantalla de la tortuga.*

*Para ver gráficamente este proceso ejecutamos*

```
rosrun rqt_graph rqt_graph
```

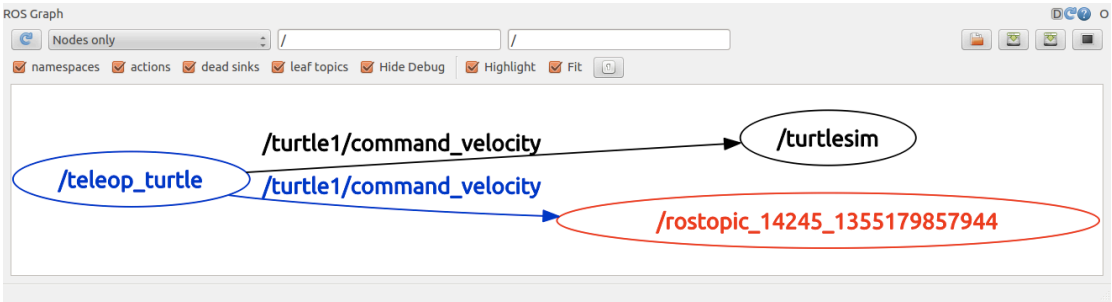

*Vemos que el topic mediante el que se están comunicando es command_velocity. Además si en un nuevo terminal ejecutamos el siguiente comando, nos mostrará un eco cada vez que se envíe este topic*

```
rostopic echo /turtle1/command_velocity
```

*Vamos a volver a mover la tortuga para comprobar que efectivamente el comando anterior muestra en pantalla información cada vez que se pulsa una flecha del teclado.*

*Si volvemos a lanzar el gráfico, veremos que un nuevo nodo está subscrito al topic anterior, este nodo corresponde al comando rostopic echo, que se ha subscrito a dicho topic.*

```
rosrun rqt_graph rqt_graph
```




*Si volvemos a ver los topics de este paquete, pero con mas detalle*

```
$ rostopic list -v
```

*Obtendremos algo como lo siguiente*

```
 Published topics:
```

```
* /turtle1/color_sensor [turtlesim/Color] 1 publisher
* /turtle1/command_velocity [turtlesim/Velocity] 1 publisher
* /rosout [roslib/Log] 2 publishers
* /rosout_agg [roslib/Log] 1 publisher
* /turtle1/pose [turtlesim/Pose] 1 publisher

Subscribed topics:
* /turtle1/command_velocity [turtlesim/Velocity] 1 subscriber
* /rosout [roslib/Log] 1 subscriber
```

*Esta información nos viene a decir fundamentalmente que hay un nodo (sabemos que se llama teleop_turtle) que publica el topic command_velocity (/turtle1 es el namespace), además este topic no es un tipo estándar (string, etc) sino que es del tipo turtlesim/Velocity, que es un mensaje definido para este proyecto. Además hay un nodo que está subscrito al topic command_velocity, que ya concemos que es turtlesim.*

*Ya hemos visto que el mensaje enviado bajo el topic command_velocity es de tipo turtlesim/Velocity pero lo podemos corroborar mediante la siguiente instrucción*

```
rostopic type /turtle1/cmd_velocity
```

*Ademas para ver la estructura intera de este tipo de mensaje se usa el siguiente comando*

```
$ rosmsg show geometry_msgs/Twist
```

*Nos aparece que este mensaje esta formado por otros dos de tipo básico (float32) que hacen referencia a la velocidad lineal y angular.*

*Como sabemos el topic que envía el nodo teleop_turlte y también conocemos el formato del mensaje, vamos a probar a enviar manualmente desde terminal uno de estos mensajes (sin usar el nodo teleop_turtle)*

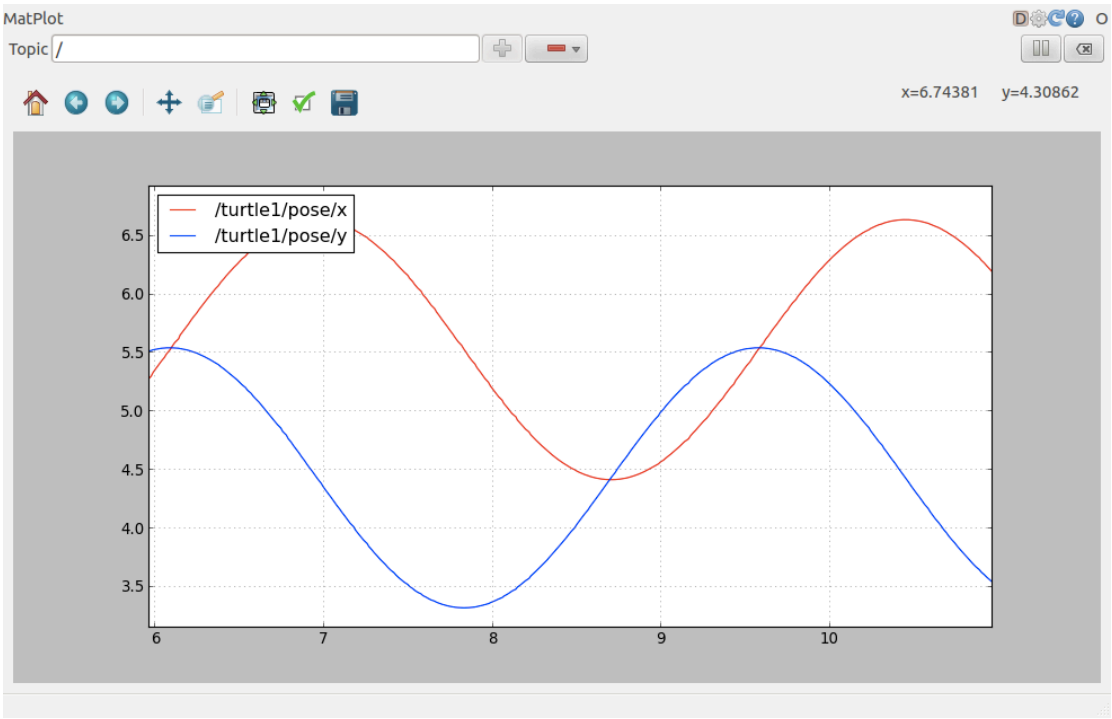
```
$ rostopic pub /turtle1/cmd_velocity geometry_msgs/Twist <pulsar tabu
lador dos veces para rellenar campos>  --once
```

*Este comando ha enviado un mensaje con el topic command_velocity que es de tipo turtlesim/Velocity con valores de velocidad lineal y angular respectivamente. Once se usa para indicar que sólo se envíe este mensaje una única vez (no de manera periódica). Si queremos enviar varios topics de manera periódica entonces sustituimos once por –r frecuencia_mensaje.*

*Ros incorpora una interesante herramienta para visualizar la evolución de los mensajes enviados. Vamos a ver un ejemplo. Ejecutamos*

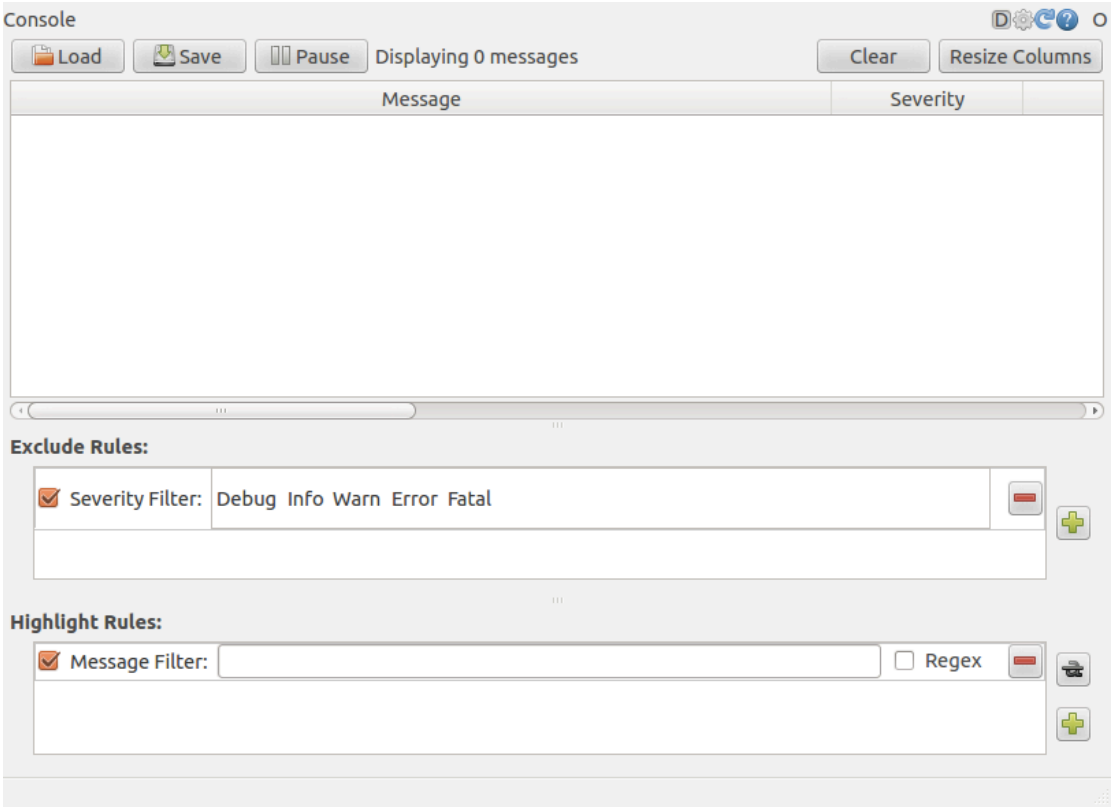
```
$ rosrun rqt_plot rqt_plot
```

*En la ventana que aparece añadirmos el siguiente parámetro* `turtle1/pose/x` *y pulsamos el botón con el '+'. Añadimos otro más* `/turtle1/pose/y`*. Podemos ver como evolucionan en el tiempo estos dos parámetros enviados mientras se mueve o movemos la tortuga.*



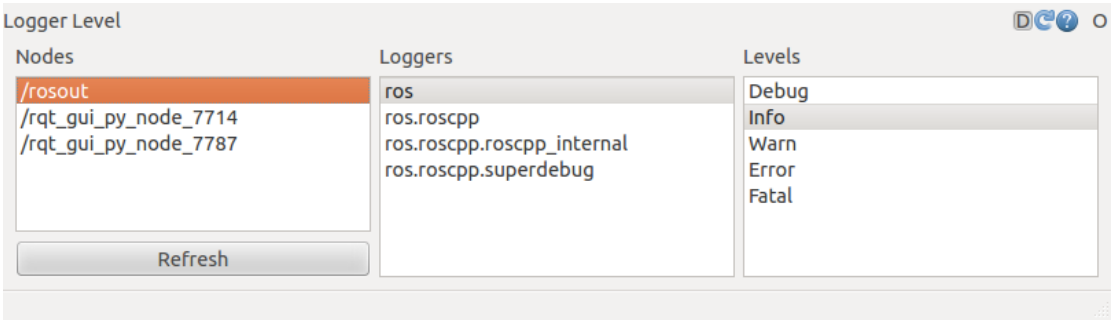

*En ROS todos los mensajes que se muestran quedan almacenados en ficheros de log y que se pueden ver y filtar de manera gráfica. Ejecutemos:*

```
rosrun rqt_console rqt_console
```




*Para cambiar el nivel de "verbosity" ejecutamos:*

```
$ rosrun rqt_logger_level rqt_logger_level
```




*Podemos cambiar entre el nivel Debug de máxima verbosidad hasta el nivel de Fatal de mínima verbosidad. Si en el ejemplo anterior cambiamos a Debug vemos como nos empiezan a aparecer de manera continuada los mensajes de posición de la tortuga. Se pueden establecer multitud de filtros a estos mensajes, bien sea por nodo, por topic, etc.*

*Finalmente vamos a usar una de las funciones mas interesantes de ROS: rosbag. Rosbag permite almacenar todos los eventos que suceden en nuestra arquitectura. Para ello, eliminamos todos los nodos lanzados de una vez:*

```
pkill –f ros
```

*Y volvemos a lanzar los dos nodos y el roscore, en terminales separados:*

```
$ roscore    → en un terminal
$ roscd turtlesim → en un terminal
$ rosrun turtlesim turtlesim_node  → en un terminal
```

*Creamos una carpeta donde almacenar los ficheros de bag:*

```
mkdir ~/bagfiles
cd ~/bagfiles
rosbag record -a
```

*Nos situamos en el terminal de teleoperación y volvemos a mover la tortuga mediante las flechas. Todos los mensajes que se están produciendo mientras se mueve la tortuga se empiezan a almacenar en la carpeta bagfiles. Cuando nos cansemos de mover la tortuga detenemos rosbag pulsando Ctr+c.*

*Podemos ver el contenido del fichero haciendo un ls. Para ver fácilmente su contenido escribimos el siguiente comando. Nos dirá el nombre de los topics intercambiados, además de su tipo y la cantidad de mensajes enviados por cada topic.*

```
rosbag info <your bagfile>
```

*Ahora vamos a ver la potencia de rosbag, por ello vamos a reproducir la secuencia grabada.*

```
rosbag play <your bagfile>
```

*Debemos ver como la tortuga vuelve a repetir los mismos movimientos que había realizado previamente al ser teleoperada por nosotros. Esto es así puesto que rosbag play ha vuelto a publicar los mismos eventos que previamente se habían publicado.*

# Creando nuestro propio paquete beginner_tutorials

*Creando nuestro propio paquete con algunas dependencias*

```
$ roscd
$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

*Actualizando la lista de paquetes conocidos*

```
rospack profile
```

*Comprobar que se localiza el paqute creado*

```
$ rospack find beginner_tutorials
```

*Comprobar que el paquete creado depende de std_msgs rospy y roscpp*

```
rospack depends1 beginner_tutorials
```

*Curiosear con el fichero package.xml para ver su estructura y como se especifican las dependencias*

```
cat package.xml
```

- *roscpp permite que cualquier nodo escrito en C++ pueda comunicarse con el "roscore" para posibilitar la comunicación con otros nodos.*
- *rospy permite que cualquier nodo escrito en python pueda comunicarse con el "roscore" para posibilitar la comunicación con otros nodos.*

*Las dependencias que hemos establecido (std_msgs, rospy, roscpp) tienen a su vez otras dependencias. Vara poder ver todas ellas (recursivamente)*

```
rospack depends beginner_tutorials
```

*Vamos a compilar nuestro primer paquete*

```
catkin_make
```

- *Podemos ver una estructura básica de carpetas dentro del paquete que se han generado hasta ahora: bin, build, include, src, lib. Hasta ahora la mayoría están vacias, pués no hemos escrito código.*
- *catkin_make lo ejecutamos la primera vez que compilamos un paquete o cuando un paquete del que dependemos ha cambiado su código fuente.*

*Vamos a volver a compilar, pero sin usar catkin_make.*

```
cd build
make
```

- *La compilación ha sido mucho más rápida, puesto que no se han compilado los paquetes de que se depende.*

*Ahora vamos a crear un tipo de mensaje que usaremos en nuestro ejemplo para comunicar entre varios nodos. Para ello primero creamos la carpeta donde se almacenan los mensajes:*

```
$ mkdir msg
$ cd msg
$ touch mensajeTest.msg
$ qtcreator
```

*Desde Qtcreator abrimos el Cmakelist.txt situado en la raíz del paquete y elegimos como carpeta para compilar la carpeta build del paquete. Ya podemos editar y incluso compilar (Ctr+B) desde Qtcreator. Editamos el archivo mensajeTest.msg con la siguiente información:*

```
uint32 numero
```

*Grabamos el archivo (Ctr+S), y editamos el archivo CMakeList.txt en la raíz de nuestro paquete, descomentamos (quitar la #) la línea siguiente para que compile el mensaje que acabamos de crear:*

```
cmake_minimum_required(VERSION 2.8.3)
project(beginer_tutorial)
```

```
## Find catkin macros and libraries
```

```
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xy
z)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)



## Generate messages in the 'msg' folder
add_message_files(
    FILES
    mensajeTest.msg
)


## Generate added messages and services with any dependencies listed
here
generate_messages(
 DEPENDENCIES
  std_msgs
 )
```

*Guardamos los cambios (Ctr+S) y compilamos (Ctr+B). También podemos compilar desde el terminal escribiendo catkin_make desde la raíz del paquete. Si hemos compilado desde qtcreator la barra de progreso acabará en color verde y sin mostrar mensajes de error. Además se deberá haber creado una carpeta msg_gen en la raíz del paquete y creado en la carpeta include el .h para usar el mensaje creado.*

*Par asegurarnos que el mensaje es reconocido por ROS escribimos:*

```
$ rosmsg show beginner_tutorials/mensajeTest
```

*Ya hemos definido nuestro primer mensaje. Vamos ahora a programar nuestros primeros nodos que intercambien mensajes. Primero creamos uno de ellos con el siguiente código C++ (nodo_emisor.cpp dentro de la carpeta src) :*

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

```
#include "beginner_tutorials/mensajeTest.h"


/**

  * Este nodo llamado nodo_emisor emite mensajes "mensajeTest_topic"
del tipo beginner_tutorials::mensajeTest

*/

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


    ros::init(argc, argv, "nodo_emisor"); //registra el nombre del no
do

    ros::NodeHandle nodo;   //Creamos un objeto nodo

    ROS_INFO("nodo_emisor creado y registrado"); //to screen and file


    //es necesario "advertir" el tipo de mensaje a enviar y como lo h
emos llamado (el topic). En este caso es de tipo userInfo y el topic
se llama user_info_topic

    ros::Publisher publicadorMensajes = nodo.advertise<beginner_tutor
ials::mensajeTest>("mensajeTest_topic",0);


    //tiempo a dormir en cada iteracción

    ros::Duration seconds_sleep(1);


    //ejecuta constantemente hasta recibir un Ctr+C

    int contador = 0;

    while (ros::ok()){


        //instancimaos un mensaje que queremos enviar

        beginner_tutorials::mensajeTest mensajeAEnviar;

        //en el mensaje enviamos el número de veces que se ha iterado
 en este bucle

        mensajeAEnviar.numero = contador;


        //se publica el mensaje

        publicadorMensajes.publish(mensajeAEnviar);


        //en este programa no es necesario spinOnce, pero si tuviera
una funcion de callback es imprescindible para que se ejecute

        ros::spinOnce();
```

```
        ROS_DEBUG ("Se duerme el nodo emisor durante un segundo");

        //dormimos el nodo durante un tiempo
        seconds_sleep.sleep();

        //incrementamos el contador
        contador++;
    }

}
```

*Creamos ahora un nodo que reciba los mensajes que emite el otro nodo. En este caso lo llamamos nodo_receptor.cpp dentro de src:*

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

/**
  * Se implementa un nodo que espera recibir mensajes cuyo topic es "mensajeTest_topic" del tipo beginner_tutorials::mensajeTest.
  * Muestra en pantalla este mensaje recibido
  */

/**
 * Esta función muestra por pantalla el mensaje recibido que es de tipo mensajeTest
 */
void funcionCallback(const beginner_tutorials::mensajeTest::ConstPtr& msg){
  ROS_INFO("He recibido un mensaje de test con el numero: %d", msg->numero);
}

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

```
  //registra el nombre del nodo: nodo_receptor

  ros::init(argc, argv, "nodo_receptor");

  ros::NodeHandle nodoReceptor;

  ROS_INFO("nodo_receptor creado y registrado"); //muestra en pantalla



  //si recibimos el mensaje cuyo topic es: "mensajeTest_topic" llamamos a la
función manejadora: funcionCallback

  ros::Subscriber subscriptor = nodoReceptor.subscribe("mensajeTest_topic", 0
, funcionCallback);



  /** Loop infinito para que no finalice la ejecución del nodo y siempre se p
ueda llamar al callback */

  ros::spin();



  return 0;

}
```

*Para compilar los nodos que acabamos de crear debemos añadir (descomentando al final del fichero) al CMakeList.txt del directorio raíz del paquete. Esto creará dos ejecutables dentro de la carpeta bin llamados nodo_emisor y nodo_receptor:*

```
## Declare a C++ executable

add_executable(nodo_emisor src/nodo_emisor.cpp)

add_executable(nodo_receptor src/nodo_receptor.cpp)

add_executable(nodo_servidor src/nodo_servidor.cpp)

add_executable(nodo_cliente src/nodo_cliente.cpp)


target_link_libraries(nodo_emisor

  ${catkin_LIBRARIES}

)

target_link_libraries(nodo_receptor

  ${catkin_LIBRARIES}

)

target_link_libraries(nodo_servidor
```

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

*Finalmente compilamos nuevamente desde la raíz del workspace:*

```
catkin_make
```

*Ahora vamos a ejecutar cada uno de estos nodos para ver su funcionamiento. Abrimos tres terminales (Ctr+T) y en cada uno de ellos escribimos los siquintes comandos:*

```
roscore
```

```
rosrun beginner_tutorials nodo_emisor
```

```
rosrun beginner_tutorials nodo_receptor
```

*Deberemos obtener una salida por pantalla en el nodo_receptor como la siguiente:*

```
 INFO /nodo_receptor:nodo_receptor creado y registrado
 INFO /nodo_receptor:He recibido un mensaje de test con el numero: 1
 INFO /nodo_receptor:He recibido un mensaje de test con el numero: 2
 INFO /nodo_receptor:He recibido un mensaje de test con el numero: 3
 INFO /nodo_receptor:He recibido un mensaje de test con el numero: 4
 INFO /nodo_receptor:He recibido un mensaje de test con el numero: 5
```

*Ahora vamos a crear un launcher que permite poner en ejecución los dos nodos de manera conjunta. Para ello creamos una carpeta dentro de nuestro propio paquete:*

```
$ mkdir launch

$ cd launch
```

*Creamos el archivo "beginner_tutorials.launch" que es como se va llamar el launcher que vamos a crear de este proyecto y lo rellenamos con las siguientes líneas:*

```
<launch>

    <node pkg="beginner_tutorials" name="nodo_emisor" type="nodo_emis
or" respawn="true" output="screen" launch-prefix="xterm –e"/>

    <node pkg="beginner_tutorials" name="nodo_receptor" type="nodo_re
ceptor" respawn="true" output="screen" launch-prefix="xterm –e"/>

</launch>
```

*El launcher como vemos está escrito en formato xml y en el se especifica todos los nodos que queremos activar. Los argumenos usados son pkg para indicar el nombre del paquete, name para indicar el nombre con el que se registra el nodo, type indica el nombre del fichero ejecutable (el que está en bin), respawn se usa para si el nodo muere inesperádamente que automáticamente se vuelva a lanzar, y finalmente output para indicar si la salida es por pantalla o a ficheros de logs.*

*Finalmente lanzamos el launcher que activa los nodos con la siguiente instrucción (el resultado debe ser similar a lanzar cada nodo en terminales separados):*

```
roslaunch beginner_tutorials beginner_tutorials.launch
```

*Vamos a repetir los mismos experimentos que hicimos con el paquete de la tortuga. Primero vemos los nodos activos:*

```
rosnode list
```

*Vemos tambien los topics activos:*

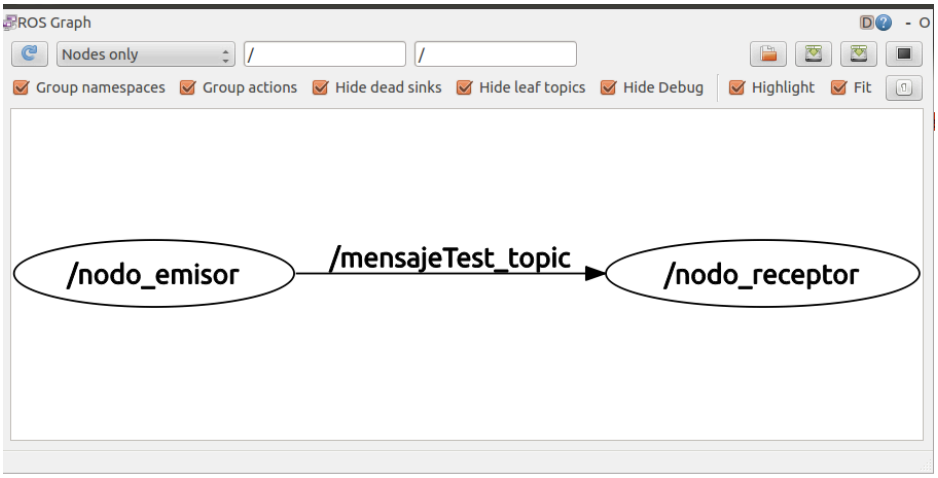
```
rostopic list
```

*Además hacemos un eco para ver a que ritmo se está enviando el mensaje (muy útil cuando las salidas las tenemos desviadas a log y no a pantalla):*

```
rostopic echo mensajeTest_topic
```

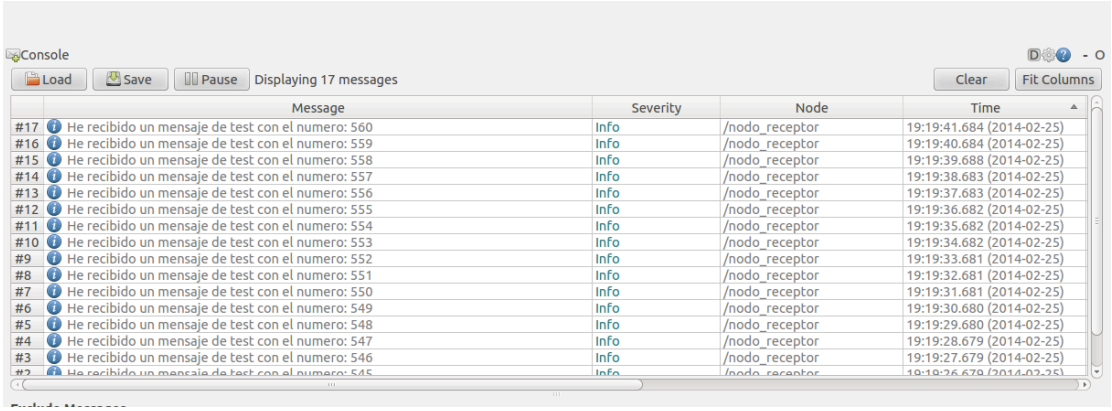
*Vemos también el esquema que hemos seguido:*

```
rqt_graph
```




*Podemos ver también el paso de mensajes de manera gráfica:*

```
rqt_console
```




*Y cambiar el nivel de "verbosity" para ver los mensajes del nivel de Debug del nodo_emisor, ya que por defecto no se muestran (el nivel de verbosity de por defecto es INFO):*

```
$ rosrun rqt_logger_level rqt_logger_level
```

## *Usando Servicios en nuestro paquete beginner_tutorials*

*Volvemos a nuestro primer paquete para practicar con los servicios, hasta ahora hemos usado sólo paso de mensajes.*

```
roscd beginner_tutorials
```

*Creamos dos nuevos ficheros:*

```
cd src
touch nodo_servidor.cpp
touch nodo_cliente.cpp
```

*Es necesario además crear la carpeta srv en el raíz de nuestro paquete:*

```
mkdir srv
```

*Como vamos a usar servicios debemos descomentar en el CmakeList del directorio raíz del paquete la siguiente línea:*

```
## Generate services in the 'srv' folder
add_service_files(
     FILES
     tipo_servicio.srv
#   Service2.srv
)
```

*Además, siguiendo en el CmakeList, hay que añadir los nuevos dos ficheros para que se compilen, por ello añadimos:*

```
rosbuild_add_executable(nodo_servidor src/nodo_servidor.cpp)
rosbuild_add_executable(nodo_cliente src/nodo_cliente.cpp)
```

*Dentro de la carpeta  srv vamos a crear el fichero  tipo_servicio.srv:*

```
int64 argumento1
---
int64 resultado1
```

*Y dentro de src el fichero nodo_servidor.cpp:*

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


/** Funcion ofertada: servicio */
```

```
bool servicio(beginner_tutorials::tipo_servicio::Request  &req,  beginner_tutorials::tipo_servicio::Response &res){


    res.resultado1 = req.argumento1 + 1;

    ROS_INFO("Petición: x = %d", (int)req.argumento1);

    ROS_INFO("Respuesta: %d", (int)res.resultado1);

    return true;

}



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


  //registra el nombre del nodo

  ros::init(argc, argv, "nodo_servidor");

  ros::NodeHandle n;


  //registra el servicio

  ros::ServiceServer service = n.advertiseService("nombre_servicio", servicio);

  ROS_INFO("Servicio registrado.");


  //nos quedamos a la espera de llamadas al servicio

  ros::spin();


  return 0;

}
```

*Y también dentro de src el fichero nodo_cliente.cpp:*

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

#include "beginner_tutorials/tipo_servicio.h"

#include <cstdlib>


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

```
  //registra el nombre del nodo
  ros::init(argc, argv, "nodo_cliente");
  ros::NodeHandle n;

  //vamos a invocar el servicio llamado Servico
  ros::ServiceClient client = n.serviceClient<beginner_tutorials::tipo_servicio>("nombre_servicio");
  beginner_tutorials::tipo_servicio srv;
  srv.request.argumento1 = 2;  //le damos un valor de prueba

  if (client.call(srv)){
    ROS_INFO("Respuesta del servicio: %d", (int)srv.response.resultado1);
  }else{
    ROS_ERROR("Fallo al llamar al servicio: nombre_servicio");
    return 1;
  }

  //para que no finalice el proceso
  ros::waitForShutdown();

  return 0;
}
```

*Ya podemos compilar el paquete:*

```
$ catkin_make
```

*Y finalmente ejecutar cada uno de los nuevos nodos en terminales diferentes:*

```
$ roscore
$ rosrun beginner_tutorials nodo_servidor
$ rosrun beginner_tutorials nodo_cliente
```

*Finalmente vamos a añadir estos dos nuevos nodos al launcher general, para ello añadimos:*

```
<node pkg="beginner_tutorials" name="nodo_servidor" type="nodo_servidor" respawn="true" output="screen"/>

<node pkg="beginner_tutorials" name="nodo_cliente" type="nodo_cliente" respawn="true" output="screen"/>
```

*Y lo ejecutamos:*

```
roslaunch beginner_tutorials beginner_tutorials.launch
```