Técnicas de los Sistemas Inteligentes

Curso 2016-17

Práctica1: Robótica.
Sesion2. Introducción a la programación en ROS

Presentación basada en http://u.cs.biu.ac.il/~yehoshr1/89-685/

Sesion 2

- Workspaces de catkin
- Paquetes ROS (ROS Packages)
- IDEs para ROS
- Implementación de un nodo ROS publisher
- Implementación de un nodo ROS subscriber
- Simulador Gazebo
- Implementación de un nodo para publicar comandos de velocidad y mover un robot en Stage (MoveForward)
- Implementación para controlar un robot en Gazebo (Stopper):
  - Subscribiéndose a un sensor láser simulado en Gazebo
  - Publicando valores de velocidad a Gazebo

ROS packages

- El software en ROS se organiza en paquetes (packages).
- Un paquete puede contener todas o alguna de las siguientes cosas:
  - Nodos ROS, una librería independiente de ROS, un dataset, ficheros de configuración, software de terceros …
- Los paquetes se crean con herramientas como catkin.

Workspaces de catkin

- catkin
  - herramienta para la gestión de paquetes de software.
  - el primer paso a hacer siempre, antes de editar el fuente de un programa (nodo) ROS, es crear un paquete catkin.
  - antes de crear cualquier paquete de ROS con catkin hay que crear un workspace de catkin.

Workspaces de catkin

- catkin workspace
  - Un espacio de trabajo (directorio, subdirectorios, ficheros) para organizar el código fuente, donde pueden construirse uno o varios catkin packages.
- Un workspace básico:

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for package_n
      package.xml        -- Package manifest for package_n
```

CMakeList: un fichero de comandos que usa CMake para construir, probar y empaquetar software

Workspaces de catkin

- crear un catkin workspace.
  - http://wiki.ros.org/catkin/Tutorials/create_a_workspace

```
$ mkdir -p ~/sesion2/src
$ cd ~/sesion2/src
$ catkin_init_workspace
$ ls
```

  - El workspace contiene inicialmente solo el fichero CMakeLists.txt.
  - catkin_make
    - construye los ejecutables del workspace y todos los paquetes dentro de él.
    - puede hacerse sobre un espacio vacío.

```
cd ~/sesion2
catkin_make
```

Workspaces de catkin

- Los ejecutables se localizan en el directorio **devel.**

```
catkin_ws/                  -- WORKSPACE
  src/                      -- SOURCE SPACE
    ...
  build/                    -- BUILD SPACE
  devel/                    -- DEVEL SPACE
    setup.bash             \
    setup.sh               |-- Environment setup files
    setup.zsh              /
    etc/                    -- Generated configuration files
    include/                -- Generated header files
    lib/                    -- Generated libraries and other artifacts
      package_1/
        bin/
        etc/
        include/
        lib/
        share/
        ...
      package_n/
        bin/
        etc/
        include/
        lib/
        share/
    share/                  -- Generated architecture independent artifacts
    ...
```

ROS packages

- **ROS package**
  - un directorio dentro de un **catkin workspace** que tiene un fichero *package.xml* dentro.Un paquete se organiza
    - a partir de un directorio dentro de un espacio de trabajo
    - contiene los ficheros fuente de uno o varios nodos y ficheros de configuración.
- Los paquetes son la unidad más básica para construir ejecutables y para sus distintas versiones.
- **rospack (**http://wiki.ros.org/rospack): herramienta de línea de comandos para consultar información sobre paquetes.

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for package_n
      package.xml        -- Package manifest for package_n
```

ROS packages

- **Ficheros y Directorios habituales de un package.**

Directory	Explanation
include/	C++ include headers
src/	Source files
msg/	Folder containing Message (msg) types
srv/	Folder containing Service (srv) types
launch/	Folder containing launch files
package.xml	The package manifest
CMakeLists.txt	CMake build file

ROS packages

- **Ejemplo de fichero de manifiesto *package.xml***

```
<package>
  <name>foo_core</name>
  <version>1.2.4</version>
  <description>
    This package provides foo capability.
  </description>
  <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
  <license>BSD</license>

  <url>http://ros.org/wiki/foo_core</url>
  <author>Ivana Bildbotz</author>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>message_generation</build_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>

  <run_depend>message_runtime</run_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>

  <test_depend>python-mock</test_depend>
</package>
```

Paquetes necesarios para construcción

Paquetes necesarios para ejecución

ROS packages

- **Crear un paquete ROS**
  - http://wiki.ros.org/catkin/Tutorials/CreatingPackage
- Ir al directorio /**src** de un workspace previamente creado.

```
$cd ~/sesion2/src
```

- **catkin_create_pkg** crea el paquete

```
$ catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

  - Cuando se crea un paquete hay que indicar las librerías necesarias (dependencias) para compilar el código fuente.
  - Si no se conocen a priori, luego se pueden modificar en el CMakeLists.txt
- **Ejemplo:**
  - Estas son las dependencias básicas de cualquier paquete.

```
$ catkin_create_pkg test_package std_msgs rospy roscpp
```

IDEs para ROS

- Es posible usar IDEs (eclipse, codeblocks), pero nosotros trabajaremos desde la línea de comandos
- http://wiki.ros.org/IDEs
- Hay buena documentación sobre cómo usar Eclipse con ROS.
- Para codeblocks hay también, poca:
  - http://answers.ros.org/question/11145/how-to-convert-ros-packages-into-c-code-to-be-used-in-code-blocks/
  - http://ftp.isr.ist.utl.pt/pub/roswiki/IDEs.html#CodeBlocks

Implementación de un nodo ROS

1. Ciclo de vida desarrollo de un nodo ROS
2. roscpp: librería cliente para C++
3. Nodo publisher

Ciclo de vida desarrollo de un nodo ROS

1. Crear workspace (si no está creado)
2. Crear un ROS package (en un paquete pueden coexistir varios nodos)
3. Escribir el código en src/<package>/src
4. Actualizar CMakeLists.txt
5. Compilar el nodo, que genera el nodo executable en /<workspace>/devel
6. Ejecutar el nodo, usando **rosrun**

Implementación Nodo publisher

- Misión de un nodo publisher.
  - Publicar datos de interés para la aplicación (datos odométricos, datos de un sensor, datos de velocidad de motores, un mapa, …)
  - Mediante el uso de mensajes
  - Mensajes que están asociados a un topic.

Nodo Publisher: Esquema

Ejemplo: publicar indefinidamente el mensaje "`Hello World` " `i` , a una frecuencia de 10 Hz (10 publicaciones/iteraciones en cada segundo), incrementando i en cada publicación,

1. **Inicialización**
    1. Usar: ros::init(), ros::NodeHandle (descritos más adelante)
    2. Declaración de los tipos de mensaje que se publicarán y de los tópicos (en este caso los mensajes son de tipo String).
2. **Declaración de la frecuencia del bucle principal (10 Hz)**
3. **Bucle principal**
    1. Creación del mensaje
    2. Publicación del mensaje
    3. Administración del bucle.

Ejemplo: Nodo Publisher C++

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker"); // Initiate new ROS node named "talker"

    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        std_msgs::String msg;

        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());

        chatter_pub.publish(msg);

        ros::spinOnce(); // Need to call this function often to allow ROS to process incoming messages

        loop_rate.sleep(); // Sleep for the rest of the cycle, to enforce the loop rate
        count++;
    }
    return 0;
}
```

Ejemplo: Nodo Publisher C++

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker"); // Initiate new ROS node named "talke

    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter",
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        std_msgs::String msg;

        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());

        chatter_pub.publish(msg);

        ros::spinOnce(); // Need to call this function often to allow ROS to process incoming messages

        loop_rate.sleep(); // Sleep for the rest of the cycle, to enforce the loop rate
        count++;
    }
    return 0;
}
```

- ros.h siempre necesario.

- En tiempo de diseño pensamos en los mensajes que vamos a usar, buscamos su definición y ponemos el .h adecuado.
  http://wiki.ros.org/std_msgs
  http://wiki.ros.org/common_msgs

- En este caso, vamos a publicar mensajes de tipo string, por tanto usamos `std_msgs/String.h`

roscpp: librería cliente para C++

- roscpp (http://wiki.ros.org/roscpp/Overview)
  - implementación de ROS en C++.
  - Documentación: http://docs.ros.org/api/roscpp/html/
  - Ficheros .h de ROS : /opt/ros/indigo/include
  - Ficheros binarios: /opt/ros/indigo/bin.
- `ros::init()` :método para inicialización del nodo
- `ros::NodeHandle` : tipo del manejador de nodo.
- `ros::Publisher` :tipo para declarar publicadores en topics
- `ros::Subscriber` :tipo para declarar suscriptores a topics
- `ros::Rate, ros::Spin` :Ayuda para ejecutar bucles
- `ros::ok()` :comprobar si va todo bien.

Ejemplo: Nodo Publisher C++

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker"); // Initiate new RO

    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user
    {
        std_msgs::String msg;

        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());

        chatter_pub.publish(msg);

        ros::spinOnce(); // Need to call this function often to allow ROS to process incoming messages

        loop_rate.sleep(); // Sleep for the rest of the cycle, to enforce the loop rate
        count++;
    }
    return 0;
}
```

Inicializamos. Asignamos al nodo el nombre “talker”. **Ojo!:** el **nombre del nodo** en ros::init puede ser diferente del **nombre del fichero ejecutable** y estos a su vez diferentes del **nombre del fichero fuente**. A veces esto es fuente de confusión.

- Se registra automáticamente la información en el Ros Master.
- Declaramos el manejador de nodo “n”, para uso interno de nuestro programa.

roscpp: ros::init()

- `ros::init()`
  - Inicializa un nodo y le asigna un nombre público en ROS
  - Debe llamarse antes de usar cualquier elemento de ROS.
  - Llamada típica en `main()`:

```
ros::init(argc, argv, “Node name”);
```

  - Recoge información de argumentos desde la línea de comandos.
  - Los nombres de nodo deben ser únicos, para evitar conflictos en la resolución de nombres.

roscpp: ros::NodeHandle

- Es un tipo de objeto que representa el punto de acceso principal para las comunicaciones con ROS, permite gestionar una referencia interna para el nodo.
  - Provee interfaces públicas para topics, services, parameters, etc.
- Para crear un manejador de nodo para el proceso actual

```
ros::NodeHandle n;
```

  - Declarar el objeto manejador interno de nodo después de ros::init()
  - Inicializa el nodo para permitir comunicación con otros nodos y con ROS Master
  - Nos permite interactuar con el nodo asociado con el proceso que estamos implementando.
  - Cuando destruimos el NodeHandle, destruimos el nodo.
    - Puede haber varios manejadores para un mismo nodo, pero poco usual para nosotros.

Ejemplo: Nodo Publisher C++

```
#include "ros/ros.h"
#include "std_msgs/Stri
#include <sstream>

int main(int argc, char
{
    ros::init(argc, argv

    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        std_msgs::String msg;

        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());

        chatter_pub.publish(msg);

        ros::spinOnce(); // Need to call this function often to allow ROS to process incoming messages

        loop_rate.sleep(); // Sleep for the rest of the cycle, to enforce the loop rate
        count++;
    }
    return 0;
}
```

Creamos el objeto **"chatter_pub"** como un publicador de mensajes de tipo **std_msgs::String** bajo el topic **"chatter"**, con un **tamaño de cola de 1000**.

El nombre del topic "chatter" lo decidimos nosotros y se registra automáticamente en ROS cuando se ejecuta el nodo. Por tanto hay que usarlo con coherencia en los publishers.

roscpp: ros::Publisher

- Hay que definir un objeto de este tipo si queremos hacer envío de mensajes bajo un topic específico.
- **`NodeHandle::advertise()`**
  - Método de NodeHandle usado para crear un Publisher
  - y para registrar un topic en el nodo master.

```
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
```

- Ejemplo de creación de un Publisher
  - Interpretación: “vamos a publicar mensajes de tipo string bajo el topic “chatter” con un tamaño de cola de 1000 mensajes”
  - El parámetro de la plantilla <T> indica el tipo de dato T que se va a publicar (en este caso String)
  - El primer parámetro del método es el nombre del topic.
  - El segundo parámetro es el tamaño de la cola de mensajes.
- La creación de un Publisher hace que automáticamente se registre el topic en el nodo master y que lo conozcan todos.
- Cuando todos los Publishers de un topic desaparecen, el topic queda “desanunciado” automáticamente.

Ejemplo: Nodo Publisher C++

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker"); // Initiate new ROS node named "talker"

    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        std_msgs::String msg;

        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());

        chatter_pub.publish(msg);

        ros::spinOnce(); // Need to call this function often to allow ROS to process incoming messages

        loop_rate.sleep(); // Sleep for the rest of the cycle, to enforce the loop rate
        count++;
    }
    return 0;
}
```

Configuramos la frecuencia de publicación de mensajes.

10 hz (ciclos/sg)

roscpp: ros::Rate

- Una clase que ayuda a ejecutar bucles a una frecuencia deseada.
- Especificar en el constructor la frecuencia deseada en Hz.

```
ros::Rate loop_rate(10);
```

- Se usa siempre junto con el Método **ros::Rate::sleep()**
  - Usado (*al final de un bucle*) para dormir el tiempo restante de ciclo.
  - Calculado desde la última vez que se llamó a sleep, reset o al constructor.

```
1 ros::Rate r(10); // 10 hz
2 while (should_continue)
3 {
4  ... do some work, publish some messages, etc.
   ...
5  r.sleep(); //duermo lo necesario para garantizar
              //la frecuencia del bucle (10 hz)
6  }
```

Ejemplo: Nodo Publisher C++

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker"); // Initiate new ROS node named "talker"

    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        std_msgs::String msg;

        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());

        chatter_pub.publish(msg);

        ros::spinOnce(); // Need to call this function often to allow ROS to process incoming messages

        loop_rate.sleep(); // Sleep for the rest of the cycle, to enforce the loop rate
        count++;
    }
    return 0;
}
```

Iterar mientras no se destruya el nodo.

roscpp: ros::ok()

- Método usado para comprobar en condiciones o bucles si el nodo debe continuar su ejecución.

```
ros::ok()
```

- Devuelve **false** si:
  - Se recibe SIGINT (Ctrl-C)
  - Nos han expulsado de la red porque hay otro nodo con el mismo nombre.
  - Se ha llamado a **ros::shutdown()** en otra parte de la aplicación.
  - Se han destruido todos los **ros::NodeHandles**

Ejemplo: Nodo Publisher C++

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker"); // Initiate

    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until
    {
        std_msgs::String msg;

        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());

        chatter_pub.publish(msg);

        ros::spinOnce(); // Need to call this fun

        loop_rate.sleep(); // Sleep for the rest
        count++;
    }
    return 0;
}
```

Definimos un objeto mensaje de tipo String.
Poblamos su estructura (con un string).

¿Cómo puedo saber la estructura de un tipo de mensaje concreto? Usando la herramienta de línea de comandos **rosmsg** http://wiki.ros.org/rosmsg

Por ejemplo: ejecutando en una terminal lo siguiente:

```
$ rosmsg show String
[std_msgs/String]:
string data
```

Obtenemos información de los campos (y tipo) del mensaje de tipo std_msgs . En este caso es una estructura con un único campo llamado “data” que es de tipo “string”.

Otros ejemplos:
$rosmsg show Pose
$rosmsg show Odometry

roscpp: ros::Publisher

- Los mensajes por defecto que pueden manejarse en ROS está definidos en dos paquetes:
  - Mensajes estándar: **std_msgs** http://wiki.ros.org/std_msgs
    - std_msgs/Bool
    - std_msgs/String
    - std_msgs/Int32
    - std_msgs/Time
  - Mensajes comunes: **common_msgs** http://wiki.ros.org/common_msgs
    - Mensajes de geometría: **geometry_msgs**
      - geometry_msgs/Point
      - geometry_msgs/Pose
      - geometry_msgs/Twist
    - Mensajes de navegación**: nav_msgs**
      - nav_msgs/Odometry
    - Mensajes de sensores: **sensor_msgs**
      - sensor_msgs/LaserScan
    - Mensajes de acciones: **actionlib_msgs**

- Identificar aquí el tipo de mensaje
- Cada mensaje está definido en un fichero <paquete/mensaje.msg>
- En C++ hay un .h por cada .msg
  - std_msgs/Bool.h
  - geometry_msgs/Pose.h
- Usar

```
#include paquete/Tipo.h
#include std_msgs/String.h
```

  - para acceder a la clase que implemente el tipo de mensaje deseado.
- **También se puede usar rosmsg, lo veremos más adelante**

Ejemplo: Nodo Publisher C++

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker"); // Ini

    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertis
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop
    {
        std_msgs::String msg;

        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());

        chatter_pub.publish(msg);

        ros::spinOnce(); // Need to call this function often to allow ROS to process incoming messages

        loop_rate.sleep(); // Sleep for the rest of the cycle, to enforce the loop rate
        count++;
    }
    return 0;
}
```

Enviamos un mensaje de log a /rosout. El mensaje también se muestra en pantalla de consola. Hay más posibilidades (llamadas "verbosity levels"):

ROS_DEBUG: para mostrar información cuando estamos depurando, pero que no queremos que se muestre cuando el sistema ya funciona bien.
ROS_INFO: información útil para un usuario
ROS_WARN: por ejemplo, "No se pudo cargar el fichero configuración desde <path>",
ROS_ERROR: mostrar información cuando algo serio ha ido mal
ROS_FATAL: algo irrecuperable ha ocurrido.

Más información en:

http://wiki.ros.org/roscpp/Overview/Logging Y
http://wiki.ros.org/rosconsole

Ejemplo: Nodo Publisher C++

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker"); // Initiate new ROS node named "talker"

    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        std_msgs::String msg;

        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());

        chatter_pub.publish(msg);

        ros::spinOnce(); // Need to call this function often to allow ROS to process incoming messages

        loop_rate.sleep(); // Sleep for the rest of the cycle, to enforce the loop rate
        count++;
    }
    return 0;
}
```

Publicamos el mensaje

roscpp: ros::Publisher

- **`publisher.publish()`**
  - Los mensajes se publican bajo un tópico mediante una llamada al método *publish().*
- ***Ejemplo:***

```
std_msgs::String msg;
chatter_pub.publish(msg);
```

  - El tipo de mensaje es un objeto que tiene que emparejar con el tipo dado como parámetro de plantilla en la llamada a *advertise<type>(topic, queuesize)*
- ¿Cómo saber
  - qué tipo de mensaje usar y
  - qué clase implementa la estructura del mensaje?
  - Usando rosmsg o consultando la documentación en ROS como hemos visto antes.

Ejemplo: Nodo Publisher C++

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker"); // Initiate ne

    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_ms
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until u
    {
        std_msgs::String msg;

        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());

        chatter_pub.publish(msg);

        ros::spinOnce(); // Need to call this function often to allow ROS to process incoming messages

        loop_rate.sleep(); // Sleep for the rest of the cycle, to enforce the loop rate
        count++;
    }
    return 0;
}
```

Gestión de la ejecución en background de la hebra.

Hacemos que el proceso haga sus funciones de background (en este caso hacer efectiva la publicación del mensaje, en otros casos recibir mensajes,…). **Si no hacemos "spin" en algún momento, "es como si no hiciéramos nada".**

Ros answer: explicación sobre uso de spin, spinOnce.
http://answers.ros.org/question/11887/significance-of-rosspinonce/

El proceso se duerme por el resto de tiempo para garantizar la frecuencia.

roscpp: ros::spin()

- Método usado para gestionar la ejecución de la hebra del proceso.
- Si no se usa, no se produce procesamiento de background (llamada a servicios, subscripciones, llamadas de retorno,…)
- Otra opción es **ros::spinOnce() dentro de un bucle.**

```
ros::init(argc, argv, "my_node");
ros::NodeHandle nh;
ros::Subscriber sub =
nh.subscribe(...);
...
ros::spin();
```

```
1 ros::Rate r(10); // 10 hz
2 while (should_continue)
3 {
4  ... do some work,
   publish some messages, etc. ...
5  ros::spinOnce();
6  r.sleep();
7  }
```

Construir nodos

- Crear un package “beginner_tutorials” dentro del workspace que habéis creado antes.

```
$cd ~/sesion2/src
```

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

- Descargar fichero *codigofuente.zip* desde PRADO. Descomprimir en una carpeta auxiliar.
- Copiar el fichero “talker.cpp” en el directorio ~/sesion2/src/beginner_tutorials/src

Construir nodos

- Antes de compilar/construir el nodo (el ejecutable del fuente) hay que modificar el fichero ***CMakeLists.txt***
  - … que se generó previamente cuando se creó el paquete con **catkin_create_pkg**
  - la generación de CMakeLists.txt crea una guía interna en el fichero que sirve de ayuda para completarlo.
- Usar el fichero de la siguiente transparencia (en rojo están los cambios).

Construir nodos: CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)

## Find catkin macros and libraries
find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs genmsg)

## Declare ROS messages and services
# add_message_files(FILES Message1.msg Message2.msg)
# add_service_files(FILES Service1.srv Service2.srv)

## Generate added messages and services
# generate_messages(DEPENDENCIES std_msgs)

## Declare catkin package
catkin_package()

## Specify additional locations of header files
include_directories(${catkin_INCLUDE_DIRS})

## Declare a cpp executable
add_executable(talker src/talker.cpp)

## Specify libraries to link a library or executable target against
target_link_libraries(talker ${catkin_LIBRARIES})
```

Si al crear el paquete, no especificamos las dependencias, este es el sitio donde especificarlas.

Se especifica el nombre del ejecutable (talker) y la ruta del fuente (src/talker.cpp) relativa al directorio del paquete.

Construir nodos: CMakeLists.txt

- Si la compilación del nodo depende de otros ejecutables, hay que especificarlos en el CMakeLists.txt:

```
add_dependencies(talker beginner_tutorials_generate_message_cpp)
```

  - Esto asegura, por ejemplo, que los .h de mensajes se generan adecuadamente antes de ser usados
  - En general no es necesario, a no ser que queramos usar nuestros propios tipos de mensajes.
- **`catkin_make`**
  - Llamarlo después de cambiar *CMakeLists*
  - Llamarlo **desde el directorio del espacio de trabajo:**

```
$ cd ~/sesion2
$ catkin_make
```

Ejecutar nodos

- **Importante!!!!!!!!!!**
  - Asegurarse de ejecutar *setup.sh* en el workspace después de llamar a ***catkin_make***

```
$ cd ~/sesion2
$ source ./devel/setup.bash
```

  - Actualiza variables de entorno para que la gestión de ROS encuentre el paquete.
  - Recordar que se recomendable añadir esta línea a .bashrc, no olvidar entonces arrancar una nueva terminal para ejecutar.
- Usar **rosrun** para ejecutar el nodo
  - En un terminal

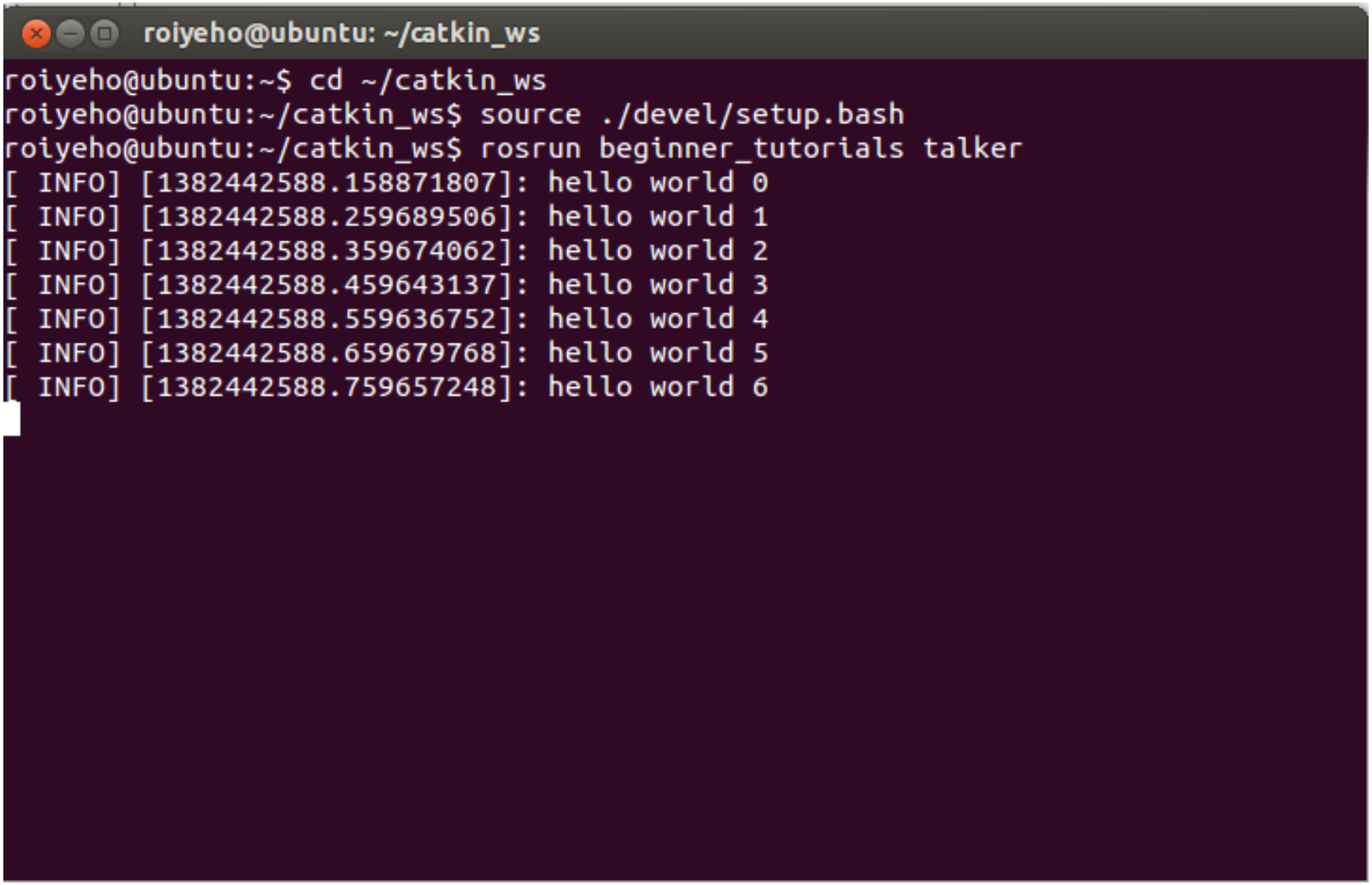
```
$ roscore
```

  - En otro terminal (asegurarse que las variables de entorno ROS están definidas)

```
$ rosrun beginner_tutorials talker
```

Parámetros de **rosrun**:
- Nombre del paquete
- Nombre de un nodo ejecutable del paquete

Ejecutar nodos



Implementación Nodo Subscriber

- Misión de un subscriber
- Esquema
- Ejemplo subscriber
- Ejemplo Subscriber como class Listener
- Construir un paquete con dos nodos (publisher, listener)
- Ejecutar los nodos
- Depuración desde línea de comandos

Misión de un subscriber

- Recibir mensajes de un *topic* previamente definido en un *Publisher.*
- Procesar la información recibida.
- La implementación de un subscriber está ***basada en eventos.***
  - Cada vez que se detecta un evento, se dispara una función de retorno (*callback function)* que gestiona el evento.
  - En nuestro caso el evento es una recepción de un mensaje.

Esquema

1. Definir la función de retorno (callback function), bien como función o como método de clase
2. Inicialización (como en un *Subscriber)*
3. Declaración de nodo
4. Subscribirse a un topic
5. Iterar lanzando callbacks cada vez que llega un msg

Ejemplo subscriber

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

// Topic messages callback
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
    // Initiate a new ROS node named "listener"
    ros::init(argc, argv, "listener");
    ros::NodeHandle node;

    // Subscribe to a given topic
    ros::Subscriber sub = node.subscribe("chatter", 1000, chatterCallback);

    // Enter a loop, pumping callbacks
    ros::spin();

    return 0;
}
```

- ros.h siempre necesario.
- Vamos a procesar los mismos tipos de mensajes que anuncia el Publisher

Ejemplo subscriber

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

// Topic messages callback
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
    // Initiate a new ROS node named "listener"
    ros::init(argc, argv, "listener");
    ros::NodeHandle node;

    // Subscribe to a given topic
    ros::Subscriber sub = node.subscribe("chatter", 1000, chatterCallback);

    // Enter a loop, pumping callbacks
    ros::spin();

    return 0;
}
```

- La función callback tiene como argumento un puntero a un mensaje del tipo definido en el publisher.
- Muestra en pantalla (y añade al log) la cadena recibida.

Ejemplo subscriber

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

// Topic messages callback
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
    // Initiate a new ROS node named "listener"
    ros::init(argc, argv, "listener");
    ros::NodeHandle node;

    // Subscribe to a given topic
    ros::Subscriber sub = node.subscribe("chatter", 1000, chatterCallback);

    // Enter a loop, pumping callbacks
    ros::spin();

    return 0;
}
```

•Inicializamos el nodo. Nuestro nodo se llama “listener”.

Ejemplo subscriber

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

// Topic messages callback
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
    // Initiate a new ROS node named "listener"
    ros::init(argc, argv, "listener");
    ros::NodeHandle node;

    // Subscribe to a given topic
    ros::Subscriber sub = node.subscribe("chatter", 1000, chatterCallback);

    // Enter a loop, pumping callbacks
    ros::spin();

    return 0;
}
```

- Nos suscribimos al nodo.
- Aun no se hace procesamiento de mensajes, esto es una declaración de objeto .

- Los mensajes se procesan en la hebra que se lanza en background cuando llamamos a ros::spin()

- Siempre hacer ros::spin()!!!!, si no, no habrá gestión de eventos.

Subscripción a un *Topic*

- Asumimos que el *topic* lo anuncia un *Publisher* que ya conocemos.
- *Método* **subscribe()**
  - Hay que llamarlo para empezar a escuchar los mensajes de un *topic.*
  - Devuelve un objeto ***Subscriber*** que vamos a usar hasta que rechazemos la suscripción.
- Ejemplo:

```
ros::Subscriber sub = node.subscribe("chatter", 1000, messageCallback);
```

  - **Interpretación:** *quiero recibir mensajes del topic "chatter", con una cola de tamaño 1000. La función "messageCallback" gestionará la recepción del mensaje.*
  - Primer parámetro es el nombre del *topic,* el mismo nombre definido en el *Publisher.*
  - Segundo parámetro es el tamaño de la cola.
  - Tercer parámetro la **función** manejadora del mensaje.

Ejemplo subscriber

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

// Topic messages callback
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
    // Initiate a new ROS node named "listener"
    ros::init(argc, argv, "listener");
    ros::NodeHandle node;

    // Subscribe to a given topic
    ros::Subscriber sub = node.subscribe("chatter", 1000, chatterCallback);

    // Enter a loop, pumping callbacks
    ros::spin();

    return 0;
}
```

- Crea un bucle en el que el nodo empieza a leer mensajes del *Topic*,
  - cuando un mensaje llega se llama a la función *chatterCallback.*
- *ros::spin()* acaba cuando ros::ok() devuelve **false**
  - **Por ejemplo** cuando se presiona CTRL + C o cuando se llama desde programa a **ros::shutdown()**

ros::spin()

- Crea un bucle en el que el nodo empieza a leer mensajes del *Topic*,
  - cuando un mensaje llega se llama a la función *messageCallback.*
- *ros::spin()* acaba cuando ros::ok() devuelve **false**
  - **Por ejemplo** cuando se presiona CTRL + C o cuando se llama desde programa a **ros::shutdown()**

Callbacks como Clases de Métodos

- ¿Y si queremos definir ***Listener como una clase?***

```
class Listener
{
    public: void callback(const std_msgs::String::ConstPtr& msg);
};
```

- La inicialización de un subscriber, e.d., la llamada a ***NodeHandle::subscribe()*** tiene una sintaxis distinta:

```
Listener listener;
ros::Subscriber sub = node.subscribe("chatter", 1000, &Listener::callback, &listener);
```

Modificar CMakeLists.txt File

- Añadir el ejecutable al final de CMakeLists.txt
- Tendremos así un paquete con dos nodos.

```
cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)

…

## Declare a cpp executable
add_executable(talker src/talker.cpp)
add_executable(listener src/listener.cpp)

## Specify libraries to link a library or executable target against
target_link_libraries(talker ${catkin_LIBRARIES})
target_link_libraries(listener ${catkin_LIBRARIES})
```

Compilar los nodos

- Construir el paquete y compilar todos los nodos usando catkin

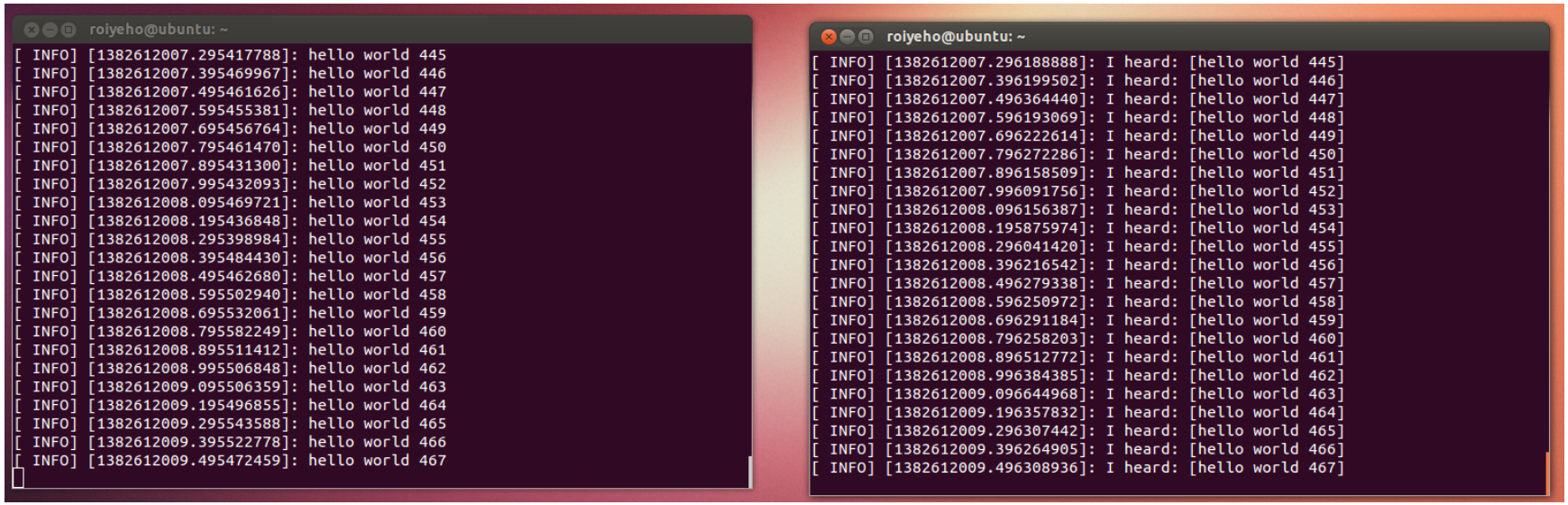
```
$ cd ~/sesion2
$ catkin_make
```

- Crea dos ejecutables , talker and listener, en ~/sesion2/devel/lib/beginner_tutorials

Ejecutar los nodos desde la terminal

- Ejecutar los nodos (y roscore) en terminales distintas.

```
$ roscore
$ rosrun beginner_tutorials talker
$ rosrun beginner_tutorials listener
```



Herramientas para depurar y consultar

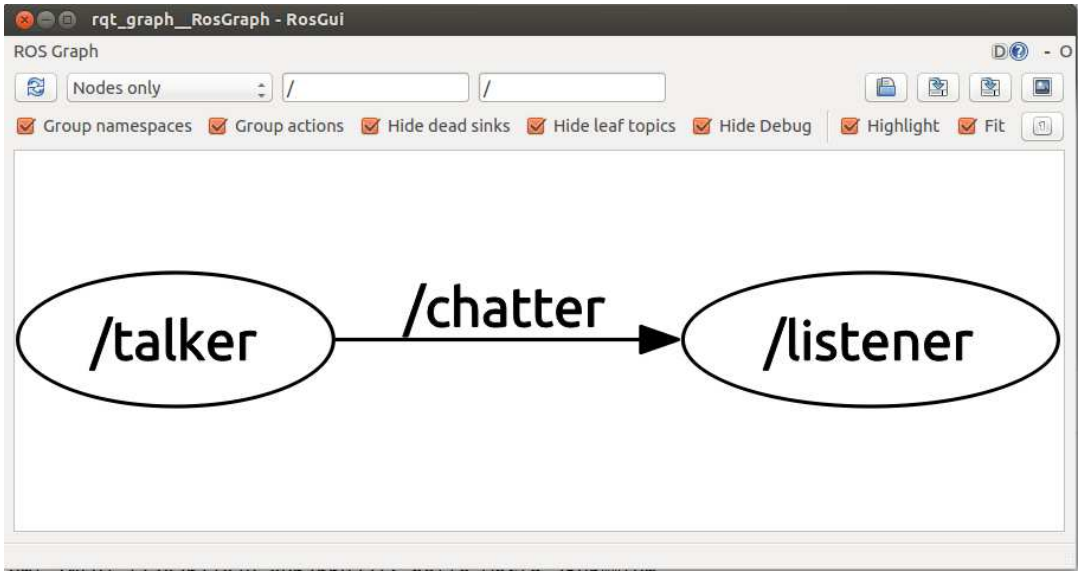
- Usar **rosnode**, **rostopic, rosmsg** para depurar y ver lo que los nodos hacen.
- Recordar: ROS cheat-sheet
  - Una página resumen de los comandos/tareas más comunes en el sistema ROS.
- Ejemplos:

```
$rosnode info /talker
$rosnode info /listener
$rostopic list
$rostopic info /chatter
$rostopic echo /chatter
```

rqt_graph

- rqt_graph creates a dynamic graph of what's going on in the system
- Use the following command to run it:

```
$ rosrun rqt_graph rqt_graph
```



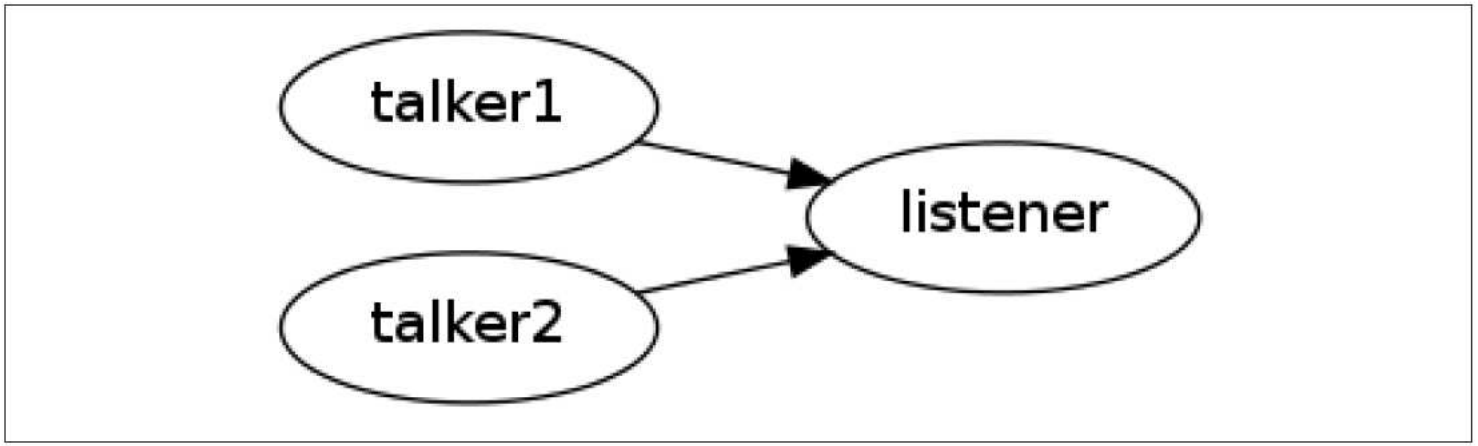
Nombres en ROS

- Los nombres en ROS deben ser únicos
- Si el mismo nodo se lanza dos veces ***roscore*** lo finaliza (redirecciona a exit).
- Cambiar el nodo en la línea de comandos sintaxis especial: **__name** remapea el nombre del nodo
- Los siguientes comandos lanzan dos instancias de talker: talker1 y talker2

```
$ rosrun beginner_tutorials talker __name:=talker1
$ rosrun beginner_tutorials talker  __name:=talker2
```

Nombres en ROS

Instanciando dos nodos talker y enrutándolos a un mismo receptor.



- Una herramienta para lanzar fácilmente múltiples nodos ROS así como asignar parámetros en el Parameter Server.
- **roslaunch** opera en ficheros “launch” que son ficheros XML que especifican una colección de nodos a lanzar con sus parámetros
  - Estos ficheros tienen la extensión “.launch”
  - Sintaxis:

```
$ roslaunch PACKAGE LAUNCH_FILE
```

- roslaunch ejecuta automáticamente roscore

Ejemplo fichero Launch

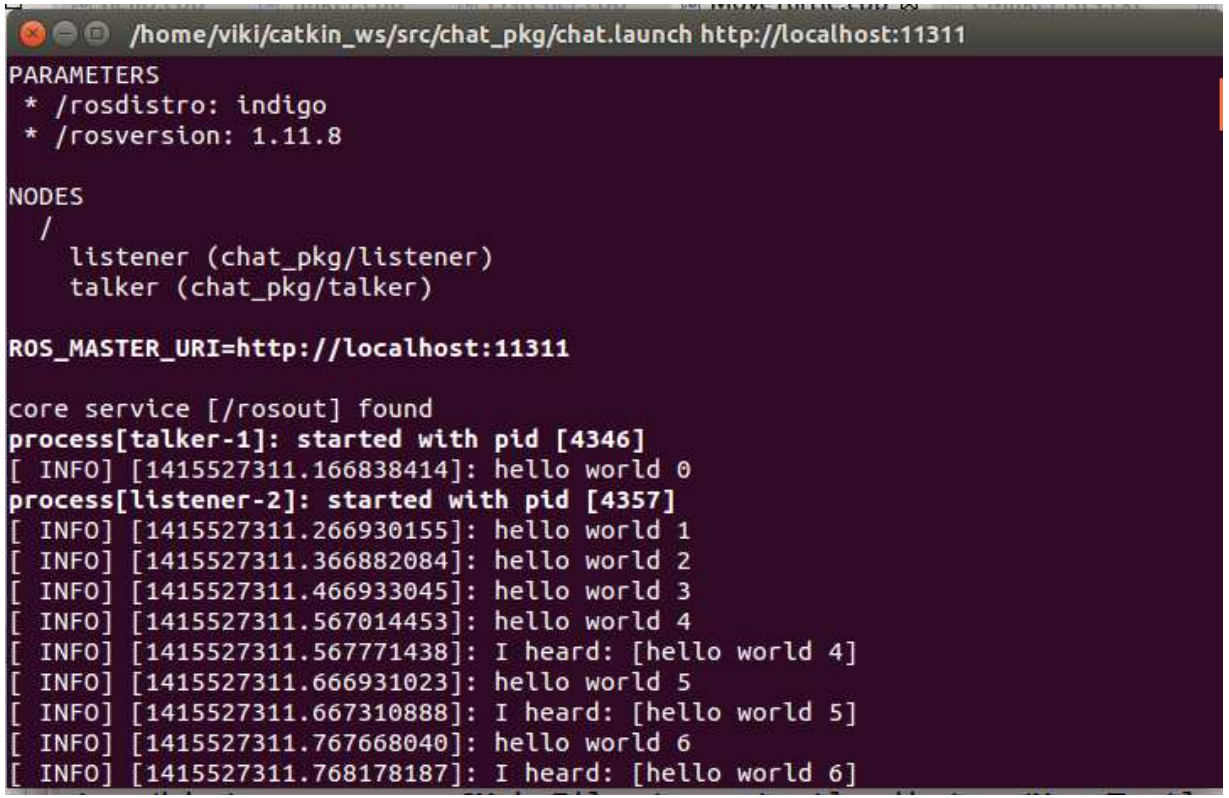
- Fichero launch para lanzar los nodos talker y listener.

```
<launch>
  <node name="talker" pkg="beginner_tutorials" type="talker" output="screen"/>
  <node name="listener" pkg="beginner_tutorials" type="listener" output="screen"/>
</launch>
```

- Cada tag <node> incluye atributos declarando el nombre del nodo en el grafo ROS, el paquete en el que se encuentra, y el tipo de nodo, que es el fichero del ejecutable
- output="screen" hace que los mensajes de log de ROS aparezcan en la terminal
- Crear un archivo "chat.launch" con el contenido de arriba en el directorio launch del paquete "beginner_tutorials".

Launch File Example

```
$ roslaunch beginner_tutorials chat.launch
```



Velocity Commands

- Vamos a ver un ejemplo de cómo mover un robot con el paquete turtle_sim.
- Ejecutamos
  - **roscore** en una terminal y
  - **rosrun turtlesim turtlesim_node** en otra
- Ejecutamos rostopic list para ver qué topics se publican con el grafo ROS actual.
- ¿Qué tipo de mensaje recibe el topic turtle1/cmd_vel?
  - Ejecutamos: rostopic type /turtle1/cmd_vel.
  - Resultado: geometry_msgs/Twist
- ¿Qué estructura tiene este tipo de mensaje?
  - Ejecutamos: rosmsg show geometry_msgs/Twist
  - Resultado

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

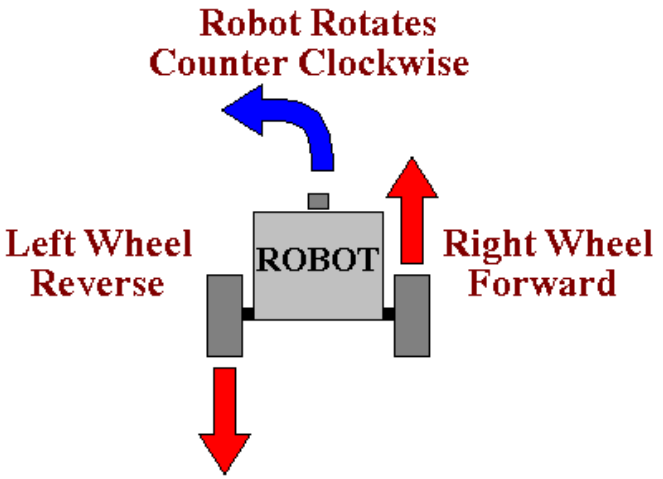
Velocity Commands

- Para hacer que un robot se mueva en ROS necesitamos enviar mensajes tipo **Twist** a un topic que normalmente se llama **cmd_vel**
- Este mensaje tiene un componente lineal para las velocidades en los ejes (x,y,z), y un componente angular para los ejes (x,y,z)

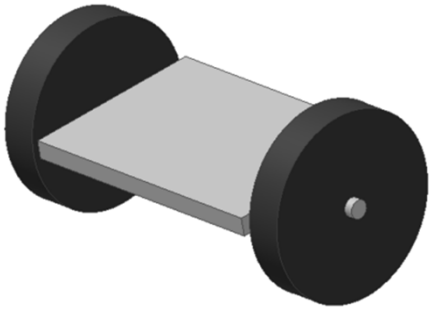
```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Robots de conducción diferencial

- Un robot con control diferencial tiene dos ruedas con actuación independiente.



- El robot se mueve hacia adelante (atrás) cuando ambas ruedas giran hacia adelante (atrás) y gira sobre su propio eje (z) cuando las ruedas tienen distinta dirección de giro.



Robots de conducción diferencial

- Estos robots solo pueden moverse hacia adelante/atrás sobre su eje longitudinal y solo giran sobre el eje vertical (z)
  - No puede moverse hacia los lados.
- Por tanto, necesitamos solo poner un valor de velocidad lineal en el componente **x** y un valor de velocidad angular en el componente **z en el mensaje tipo Twist**

A Move Turtle Node

- Para la demostración vamos a crear un paquete ROS llamado my_turtle

```
$ cd ~/sesion2/src
$ catkin_create_pkg my_turtle std_msgs rospy roscpp
```

- Descargar my_turtle.zip desde Prado.
- Copiar el siguiente fichero (move_turtle.cpp) en el directorio fuente del paquete **my_turtle**.

MoveTurtle.cpp

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

int main(int argc, char **argv)
{
    const double FORWARD_SPEED_MPS = 0.5;

    // Initialize the node
    ros::init(argc, argv, "move_turtle");
    ros::NodeHandle node;

   // A publisher for the movement data
    ros::Publisher pub = node.advertise<geometry_msgs::Twist>("turtle1/cmd_vel", 10);

    // Drive forward at a given speed.  The robot points up the x-axis.
    // The default constructor will set all commands to 0
    geometry_msgs::Twist msg;
    msg.linear.x = FORWARD_SPEED_MPS;

    // Loop at 10Hz, publishing movement commands until we shut down
    ros::Rate rate(10);
    ROS_INFO("Starting to move forward");
    while (ros::ok()) {
        pub.publish(msg);
        rate.sleep();
    }
}
```

- Incluimos el .h necesario para publicar los mensajes tipo Twist
- Declaramos el publisher de tipo Twist en el topic "turtle1/cmd_vel.
- Rellenamos el mensaje tipo Twist.
- Publicamos mensajes de velocidad a una frecuencia de 10Hz

Launch File

- Compilar, modificando adecuadamente el Cmakelists. Al hacer catkin_make no olvidar hacer el source …..!!!!
- Añadir move_turtle.launch al paquete my_turtle:

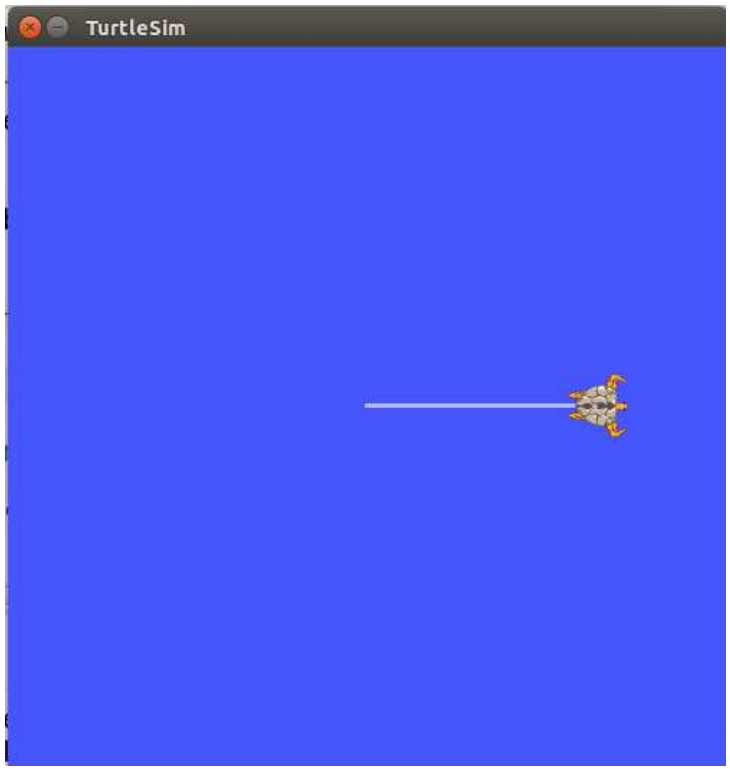
```
<launch>
  <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node" />
  <node name="move_turtle" pkg="my_turtle" type="move_turtle"
output="screen" />
</launch>
```

- Copiarlo/crearlo en el directorio launch
- Ejecutar el fichero launch:

```
$ roslaunch my_turtle move_turtle.launch
```

Demo Move Turtle

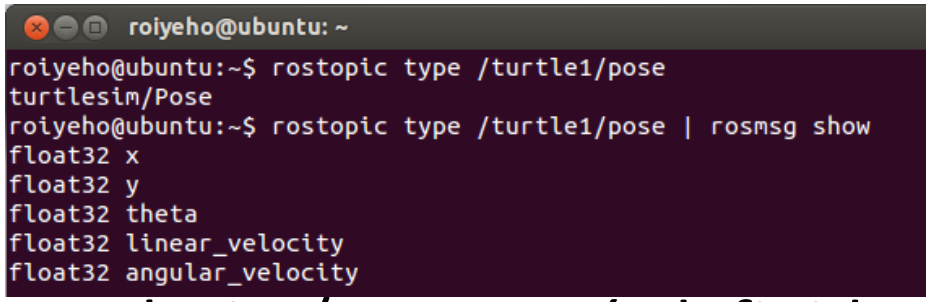
- Debería verse la tortuga el su simulador moviéndos hasta que chaca con el muro.



Imprimir en pantalla la pose de la tortuga

- Para imprimir en pantalla la pose de la tortuga necesitamos subscribirnos al topic /turtle1/pose
- Podemos encontrar **el tipo de mensaje del topic** y su **estructura** ejecutando el comando

```
$ rostopic type /turtle1/pose| rosmsg show
```



- El mensaje turtlesim/Pose está definido en el paquete turtlesim, por tanto necesitamos incluir el fichero header “turtlesim/Pose.h” en nuestro código

MoveTurtle.cpp (1)

- Hacer las modificaciones marcadas.

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "turtlesim/Pose.h"

// Topic messages callback
void poseCallback(const turtlesim::PoseConstPtr& msg)
{
    ROS_INFO("x: %.2f, y: %.2f", msg->x, msg->y);
}

int main(int argc, char **argv)
{
    const double FORWARD_SPEED_MPS = 0.5;

    // Initialize the node
    ros::init(argc, argv, "move_turtle");
    ros::NodeHandle node;

    // A publisher for the movement data
    ros::Publisher pub = node.advertise<geometry_msgs::Twist>("turtle1/cmd_vel", 10);

    // A listener for pose
    ros::Subscriber sub = node.subscribe("turtle1/pose", 10, poseCallback);
```

- Callback para capturar la información de la posición de la tortuga.

- Declaración del listener de la “pose”.

MoveTurtle.cpp (2)

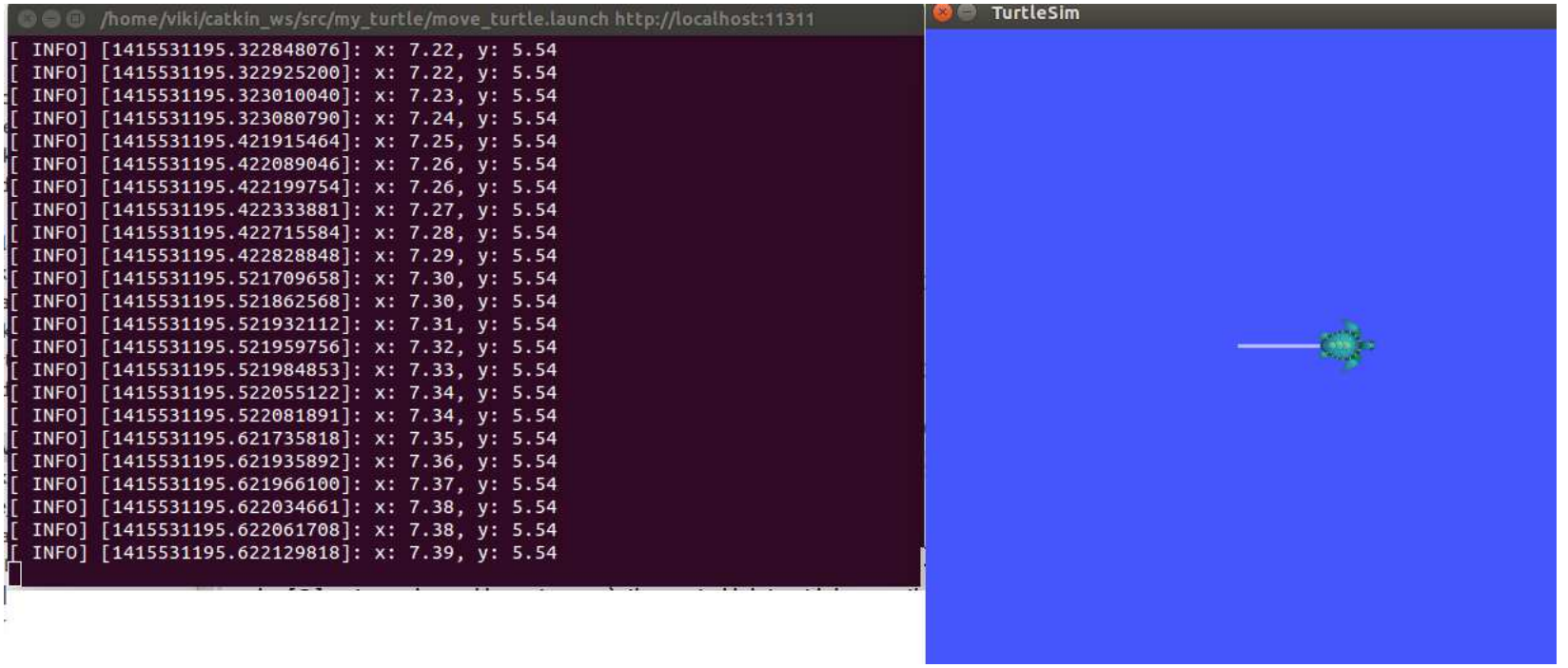
```
    // Drive forward at a given speed.  The robot points up the x-axis.
    // The default constructor will set all commands to 0
    geometry_msgs::Twist msg;
    msg.linear.x = FORWARD_SPEED_MPS;

    // Loop at 10Hz, publishing movement commands until we shut down
    ros::Rate rate(10);
    ROS_INFO("Starting to move forward");
    while (ros::ok()) {
        pub.publish(msg);
        ros::spinOnce(); // Allow processing of incoming messages
        rate.sleep();
    }
}
```

- Llamamos a spinOnce() para procesar en background los mensajes que lleguen y perm que se dispare la callback cad vez que llegue el mensaje.

Imprimir en pantalla la pose de la tortuga

- roslaunch my_turtle move_turtle.launch



Pasando argumentos a nodos

- Se puede usar en el fichero launch el atributo **args** para pasar argumentos de línea de comando al nodo.
- En este caso, pasamos el nombre de la tourtuga como un argumento al nodol move_turtle

```
<launch>
  <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node" />
  <node name="move_turtle" pkg="my_turtle" type="move_turtle"
args="turtle1" output="screen"/>
</launch>
```

MoveTurtle.cpp

```
int main(int argc, char **argv)
{
    const double FORWARD_SPEED_MPS = 0.5;
    string robot_name = string(argv[1]);

    // Initialize the node
    ros::init(argc, argv, "move_turtle");
    ros::NodeHandle node;

    // A publisher for the movement data
    ros::Publisher pub = node.advertise<geometry_msgs::Twist>(robot_name + "/cmd_vel", 10);

    // A listener for pose
    ros::Subscriber sub = node.subscribe(robot_name + "/pose", 10, poseCallback);

    geometry_msgs::Twist msg;
    msg.linear.x = FORWARD_SPEED_MPS;

    ros::Rate rate(10);
    ROS_INFO("Starting to move forward");
    while (ros::ok()) {
        pub.publish(msg);
        ros::spinOnce(); // Allow processing of incoming messages
        rate.sleep();
    }
}
```

- El nombre del topic depende del valor de la variable robot_name pasado por argumento.

Ejercicio propuesto (no para entregar)

- Escribir un programa que mueva la tortuga 1m hacia adelante desde su posicióna actual, gire entonces 45 grados y se pare.
- Mostrar en pantalla las posiciones iniciales y finales de la tortuga.


DECSAI
XXV