Técnicas de los Sistemas Inteligentes

Curso 2015-16

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

Presentación basada en http://u.cs.biu.ac.il/~yehoshr1/89-685/
(C)2013 Roi Yehoshua

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 Stage
- Implementación de un nodo para publicar comandos de velocidad y mover un robot en Stage (MoveForward)
- Implementación para controlar un robot en Stage (Stopper):
  - Subscribiéndose a un sensor láser simulado en Stage
  - Publicando valores de velocidad a Stage

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 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
```

Workspaces de catkin

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

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

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

```
cd ~/catkin_ws
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.
- Los paquetes son la unidad más básica para construir ejecutables y para sus distintas versiones.
- 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.

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>
```

ROS packages

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

```
$cd ~/catkin_ws/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
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**

roscpp: librería cliente para C++

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

roscpp: ros::init()

- `ros::init()`
  - 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
  - Provee interfaces públicas para topics, services, parameters, etc.
- Para crear un manejador de nodo para el proceso actual

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

  - 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.

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
  - 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.
  - Interpretación: "vamos a publicar mensajes de tipo string bajo el topic "chatter" con un tamaño de cola de 1000 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.

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?

roscpp: ros::Publisher

- Paquetes de ROS que definen tipos de mensajes:
  - 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.

roscpp: ros::ok()

- Método usado para comprobar si el nodo debería de continuar su ejecución.

- 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**

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  }
```

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  }
```

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

1. Inicialización
    1. ros::init(), ros::NodeHandle
    2. Declaración de los tipos de mensaje que se publicarán y de los tópicos.
2. Declaración de la frecuencia del bucle principal
    1. Ros answer sobre spinning
        1. http://answers.ros.org/question/11887/significance-of-rosspinonce/
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 "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;
}
```

- ros.h siempre necesario.
- En tiempo de diseño pensamos en los mensajes que vamos a usar, buscamos su definición y ponemos el .h adecuado.

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;
}
```

Inicializamos. Asignamos al nodo el nombre “talker”.

Se registra automáticamente la información en el Ros Master.

Declaramos el manejador de nodo “n”, para uso interno de nuestro programa.

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;
}
```

Creamos el objeto “chatter_pub” como un publicador de mensajes tipo string bajo el topic “chatter”, con un tamaño de cola de 1000.

El nombre del topic “chatter” lo decidimos nosotros. Luego hay que usarlo con coherencia en los publishers.

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 ciclos/sg.

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.

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;
}
```

Definimos un objeto mensaje de tipo String.

Rellenamos su estructura (con un string).

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
    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;
}
```

Enviamos un mensaje de log a /rosout. El mensaje también se muestra en pantalla de consola.

Más información en:

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

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>("chatt
    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;
}
```

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

Construir nodos

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

```
$cd ~/catkin_ws/src
```

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

- 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.

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 ~/catkin_ws
$ catkin_make
```

Ejecutar nodos

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

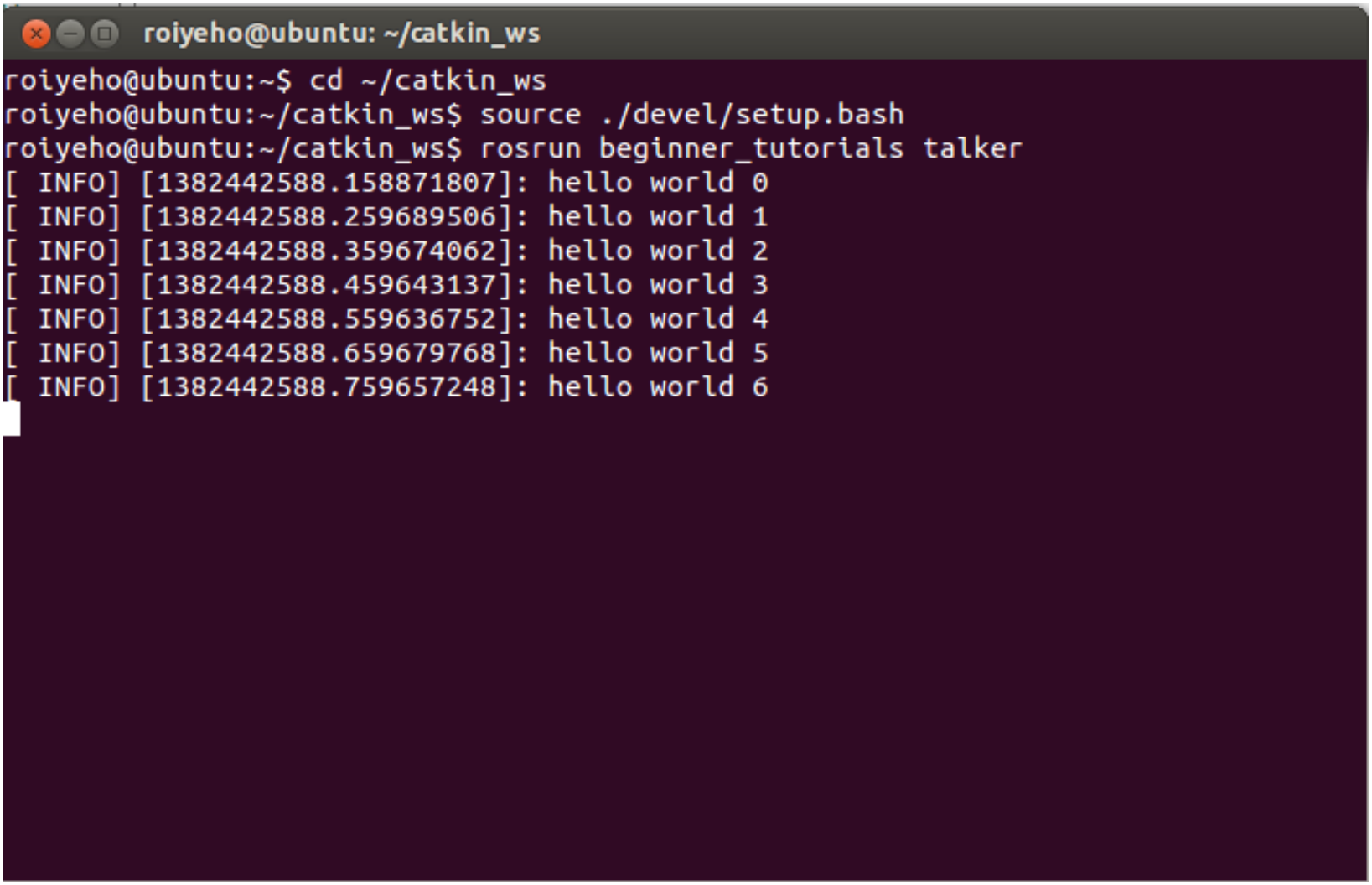
```
$ cd ~/catkin_ws
$ source ./devel/setup.bash
```

  - Actualiza variables de entorno para que la gestión de ROS encuentre el paquete.
- Puede añadirse esta línea a .bashrc, no olvidar entonces arrancar una nueva terminal para ejecutar.
- Usar **rosrun** para ejecutar el nodo

```
$ roscore
```

```
$ rosrun beginner_tutorials talker
```

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

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);
```

  - 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;
}
```

- 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. 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()!!!!

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

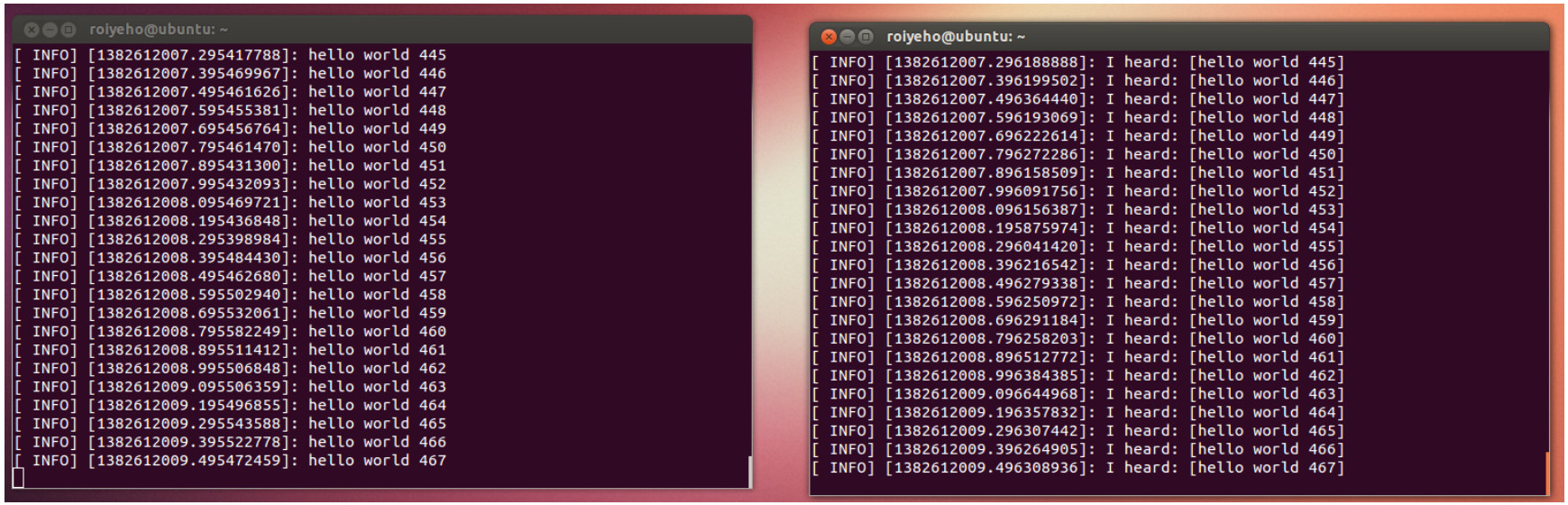
```
cd ~/catkin_ws
catkin_make
```

- Crea dos ejecutables , talker and listener, en ~/catkin_ws/devel/lib/<package>

Running the Nodes From Terminal

- Run the nodes in two different terminals:

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



Running the Nodes From Terminal

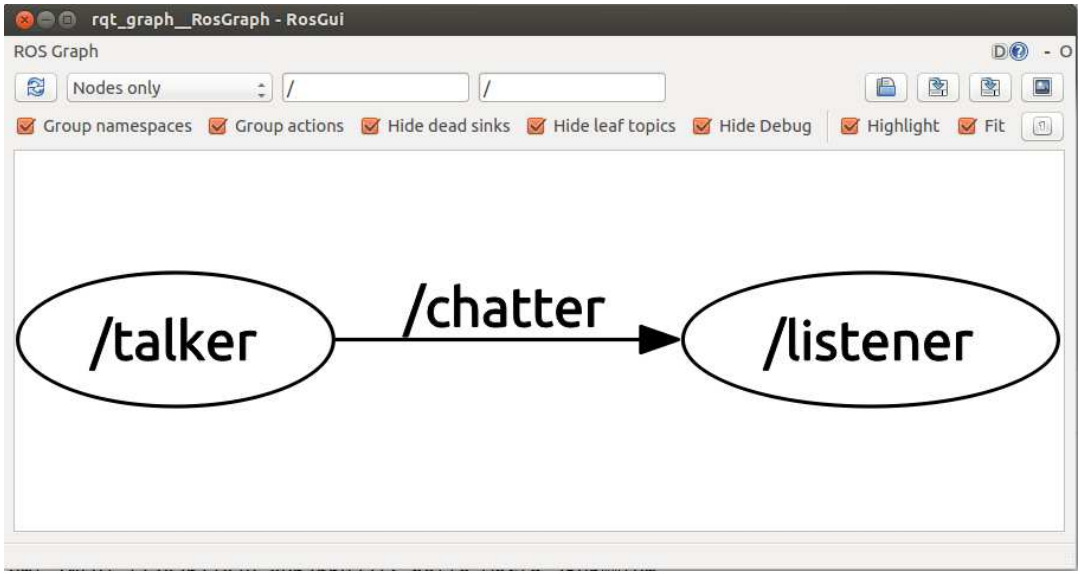
- Usar **rosnode** y **rostopic** para depurar y ver lo que los nodos hacen.
- ROS cheat-sheet
  - Una página resumen de los comandos/tareas más comunes en el sistema ROS.
- Examples:

```
$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
```



Simulador Stage

ROS Stage Simulator

- http://wiki.ros.org/stage_ros?distro=hydro
- Un simulador que provee un mundo virtual poblado por objetos, robots móviles y sensores. Los objetos pueden se detectados y manipulados por los robots.
- Stage provee modelos para varios tipos de sensores y actuadores:
  - sonar or infrared rangers
  - scanning laser rangefinder
  - color-blob tracking
  - bumpers
  - grippers
  - odometric localization
  - and more

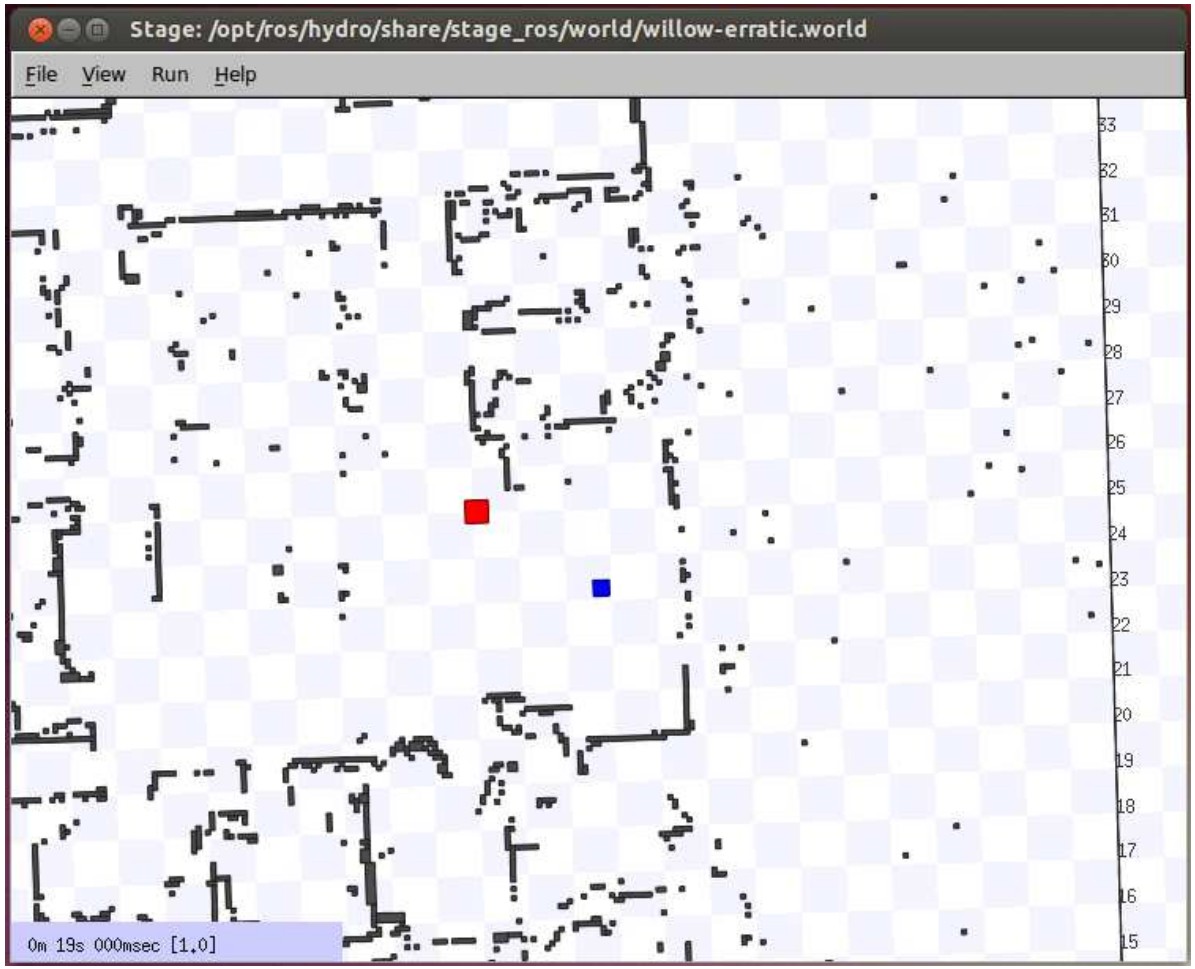
Run Stage with an existing world file

- Stage viene instaladdo con ROS Indigo
- Stage necesita como entrada ficheros de mundo (world files) con extensión .world.
  - Hay ejemplos: hacer **`roscd stage_ros/world`**
- Stage trae algunos ejemplos de *world* files incluyendo uno que pone un robot en un entorno similar al laboratorio de Willow Garage-like environment.
- Ejecutar:

```
rosrun stage_ros stageros `rospack find stage_ros`/world/willow-erratic.world
```

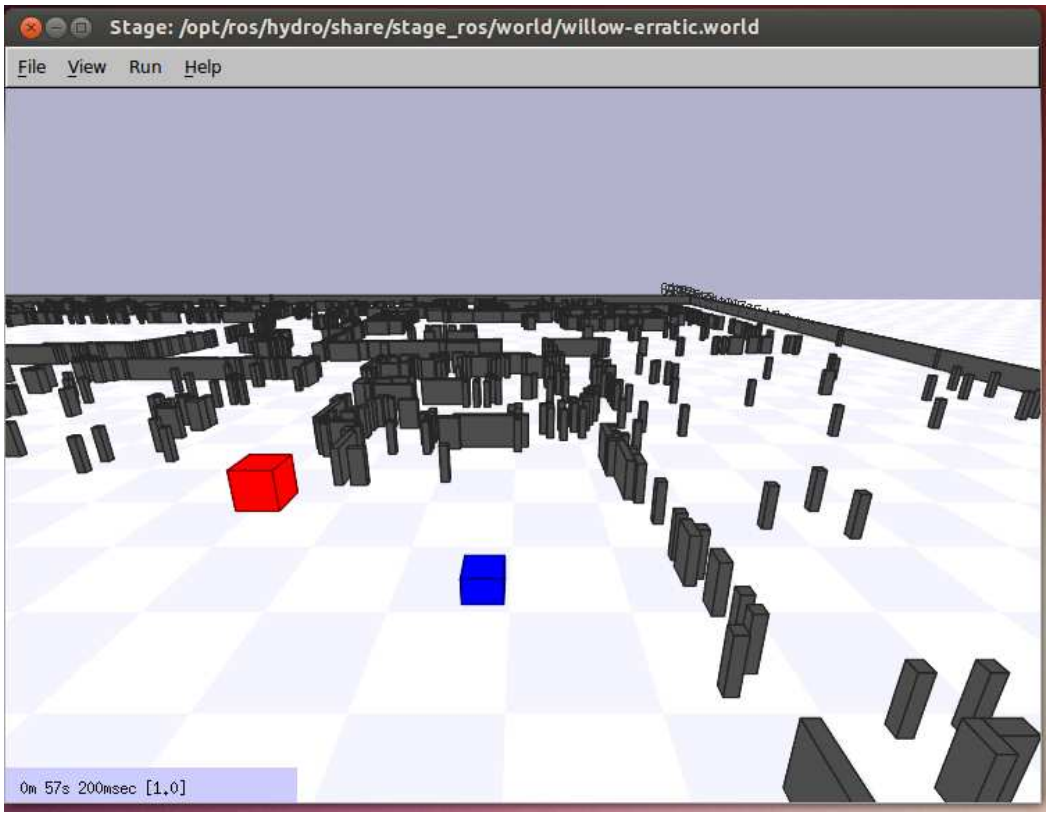
- Navegar por la ventana de Stage hasta ver dos cuadrados: el rojo es una caja, el azul un robot.
- En el material de prácticas hay un tutorial sobre cómo construir ficheros .world en Stage para dibujar objetos/robots más realistas.

Ejecutar Stage con un “world file” existente



Ejecutar Stage con un “world file” existente

- Click en la ventana de Stage y presionar “R” para una vista de perspectiva.



Moviendo el robot con teleoperación

- Para teleoperar un robot desde el teclado usaremos un ***paquete*** llamado *teleop_twist_keyboard* que es parte del ***stack*** *brown_remotelab*
  - http://wiki.ros.org/brown_remotelab
- **Installing**
  - `sudo apt-get install ros-indigo-teleop-twist-keyboard`
- **Running**
  - `rosrun teleop_twist_keyboard teleop_twist_keyboard.py`

Instalar ROS Packages Externos

- Create a directory for downloaded packages
  - For example, ~/ros/stacks
- Edit ROS_PACKAGE_PATH in your .bashrc to include your own ros stacks directory

```
export ROS_PACKAGE_PATH=~/ros/stacks:${ROS_PACKAGE_PATH}
```

- Check the package out from the SVN:

```
$cd ~/ros/stacks
$svn co https://brown-ros-pkg.googlecode.com/svn/trunk/distribution/brown_remotelab
```

- Compile the package

```
$rosmake brown_remotelab
```

- **rosmake** es una orden de ***rosbuild*** otro gestor de paquetes que convive con *catkin.* Pueden usarse ambos (siempre que no nos liemos y los usemos para paquetes distintos).

Move the robot around

- Now run teleop_twist_keyboard:

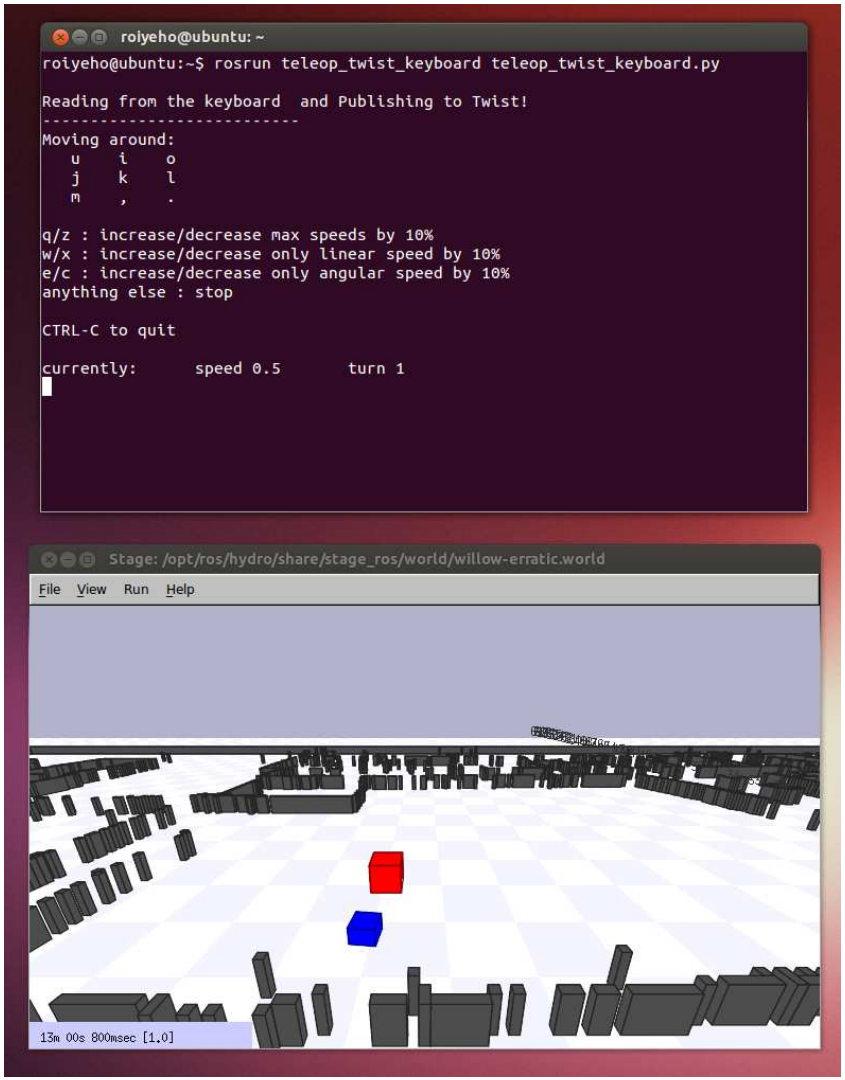
```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

- You should see console output that gives you the key-to-control mapping, something like this:

```
Reading from keyboard
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

anything else : stop
---------------------------
```

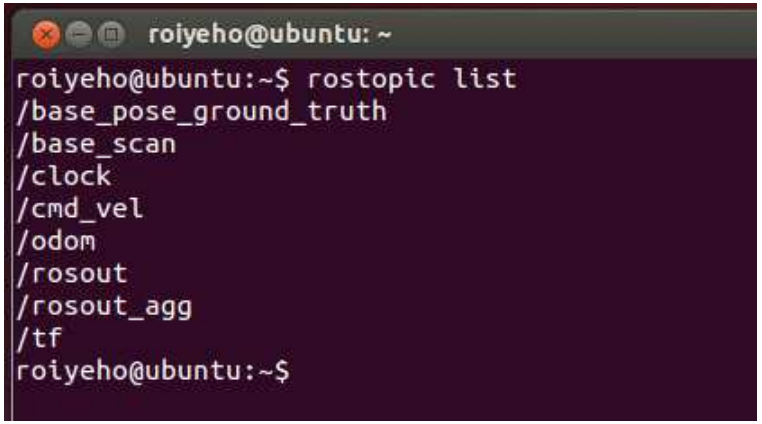
  - Hold down any of those keys to drive the robot. E.g., to drive forward, hold down the **i** key.
  - Mantener el foco en la terminal donde se ejecuta el nodo de teleoperación, si no no recibe las señales del teclado.

Move the robot around



Topics publicados por Stage

- Ver los *topics* disponibles con Stage con **rostopic list**



- Usar **rostopic echo -n 1** para ver una instancia de los datos en uno de los topics.

Mensajes de Odometría

roiyeho@ubuntu: ~

```
roiyeho@ubuntu:~$ rostopic echo /odom -n 1
header:
  seq: 11485
  stamp:
    secs: 1148
    nsecs: 600000000
  frame_id: odom
child_frame_id: ''
pose:
  pose:
    position:
      x: 1.16596142952
      y: -0.133586349782
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: -0.389418342309
      w: 0.921060994003
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
twist:
  twist:
```

Nodo "*Move Forward*"

- Vamos a implementar un nodo que guía al robot hasta que choca con un obstáculo.
- Para ello necesitamos publicar mensajes tipo ***Twist messages*** bajo el topic **cmd_vel** (al que está suscrito Stage)
  - Este *topic* es el responsable de enviar órdenes de velocidad al robot.

- Hacer
  - `rosrun rqt_graph rqt_graph`
  - `Observar nodes y topics.`

- http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html
- This message has a **linear** component for the (x,y,z) velocities, and an **angular** component for the angular rate about the (x,y,z) axes.

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

Crear un ROS package nuevo

- For the demo, we will create a new ROS package called my_stage

```
$cd ~/catkin_ws/src
$ catkin_create_pkg my_stage std_msgs rospy roscpp
```

- El resto de argumentos son los paquetes de los que depende *my_stage*

move_forward.cpp

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

int main(int argc, char **argv)
{
    const double FORWARD_SPEED_MPS = 0.2;

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

    // A publisher for the movement data
    ros::Publisher pub = node.advertise<geometry_msgs::Twist>("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();
    }
}
```


```
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(move_forward src/move_forward.cpp)

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

Move Forward Demo

- Compilar y ejecutar el nodo

```
cd ~/catkin_ws
catkin_make
```

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

```
rosrun stage_ros stageros `rospack find stage_ros`/world/willow-erratic.world
```

```
$ rosrun my_stage move_forward
```

- El robot se mueve constantemente en el simulador hasta que choca con un obstáculo.

Nodo Stopper

- Haremos que el robot se detenga antes de chocar con un obstáculo.
- Necesitamos leer datos de sensores (subscribirnos al topic que represente información de sensor láser)
- Necesitamos enviar datos de velocidad (publicar el *topic* cmd_vel como el nodo *MoveForward.*
- Crearemos un nodo llamado *stopper.*

Datos de sensores

- La simulación produce datos de sensores (sin ruido)
  - Publicados por **Stage** bajo el *topic* **/base_scan**
- El tipo de mensaje usado por Stage para publicar datos de láser es sensor_msgs/LaserScan
- Puede verse directamente la estructura del mensaje:

```
$rosmsg show sensor_msgs/LaserScan
```

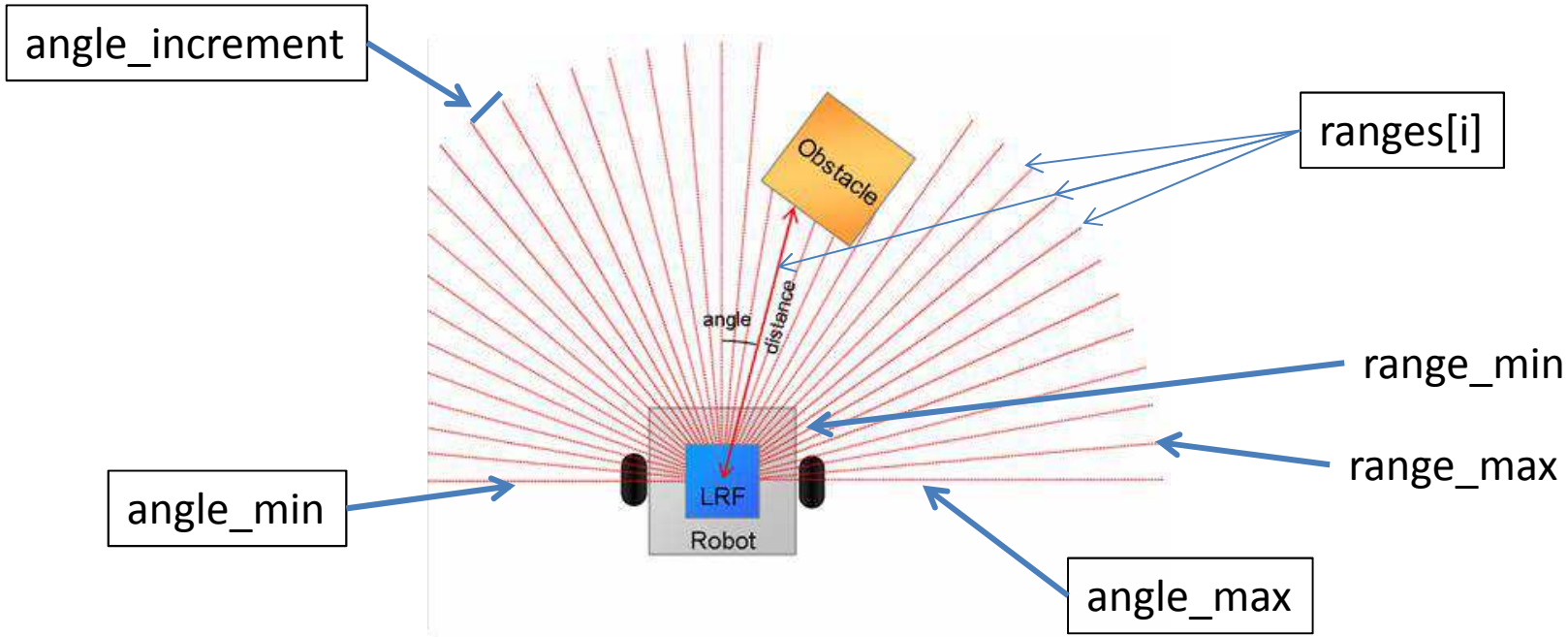
- Stage produce lasers scans *perfectos*
  - Los robots reales y los lásers reales incorporan ruido real que Stage no está simulando.

LaserScan Message

- http://docs.ros.org/api/sensor_msgs/html/msg/LaserScan.html

```
# Single scan from a planar laser range-finder

Header header
# stamp: The acquisition time of the first ray in the scan.
# frame_id: The laser is assumed to spin around the positive Z axis
# (counterclockwise, if Z is up) with the zero angle forward along the x axis

float32 angle_min # start angle of the scan [rad]
float32 angle_max # end angle of the scan [rad]
float32 angle_increment # angular distance between measurements [rad]

float32 time_increment # time between measurements [seconds] - if your scanner
# is moving, this will be used in interpolating position of 3d points
float32 scan_time # time between scans [seconds]

float32 range_min # minimum range value [m]
float32 range_max # maximum range value [m]

float32[] ranges # range data [m] (Note: values < range_min or > range_max should be
discarded)
float32[] intensities # intensity data [device-specific units]. If your
# device does not provide intensities, please leave the array empty.
```

LaserScan Message



LaserScan Message

- Ejemplo de un scan laser del simulador Stage:

```
roiyeho@ubuntu: ~
---
header:
  seq: 1594
  stamp:
    secs: 159
    nsecs: 500000000
  frame_id: base_laser_link
angle_min: -2.35837626457
angle_max: 2.35837626457
angle_increment: 0.00436736317351
time_increment: 0.0
scan_time: 0.0
range_min: 0.0
range_max: 30.0
ranges: [2.427844524383545, 2.42826247215271, 2.4287266731262207, 2.4292376041412354, 2.429795026779175, 2.430398941
040039, 2.4310495853424072, 2.4317471981048584, 2.4324913024902344, 2.4332826137542725, 2.4341206550598145, 2.435005
6648254395, 2.4359381198883057, 2.436917543411255, 2.437944173812866, 2.439018487930298, 2.4401402473449707, 2.44130
94520568848, 2.4425265789031982, 2.443791389465332, 2.4451043605804443, 2.446465253829956, 2.4478745460510254, 2.449
3319988250732, 2.450838088989258, 2.452392816543579, 2.453996419906616, 2.455648899078369, 2.457350492477417, 2.4591
01438522339, 2.460901975631714, 2.462752103805542, 2.4646518230438232, 2.466601848602295, 2.468601942062378, 2.47065
23418426514, 2.4727535247802734, 2.474905490875244, 2.4771084785461426, 2.479362726211548, 2.481668472290039, 2.4840
259552001953, 2.4864354133605957, 2.4888970851898193, 2.4914112091064453, 2.4939777851104736, 2.4965975284576416, 2.
4992706775665283, 2.5019969940185547, 2.504777193069458, 2.5076115131378174, 2.510500192642212, 2.5134434700012207,
2.516441822052002, 2.5194954872131348, 2.5226047039031982, 2.5257697105407715, 2.5289909839630127, 2.53226900100708,
 2.5356037616729736, 2.5389959812164307, 2.542445659637451, 2.5459535121917725, 2.5495197772979736, 2.55314469337463
4, 2.5568289756774902, 2.560572624206543, 2.56437611579895, 2.568240165710449, 2.572165012359619, 2.576151132583618,
 2.5801987648010254, 2.584308624267578, 2.5884809494018555, 2.5927164554595947, 2.597015380859375, 2.601378202438354
5, 2.6058056354522705, 2.610297918319702, 2.6148557662963867, 2.6194796562194824, 2.6241698265075684, 2.628927230834
961, 2.6337523460388184, 2.63478422164917, 2.6436073780059814, 2.6486384868621826, 2.6537396907806396, 3.44798207283
02, 3.4547808170318604, 3.461672306060791, 3.4686577320098877, 3.4757378101348877, 3.4829134941101074, 3.49018549919
1284, 3.4975550174713135, 3.5050225257873535, 3.5125889778137207, 3.5202558040618896, 3.5280232429504395, 3.53589296
3409424, 3.543865442276001, 3.5519418716430664, 3.5601232051849365, 3.568410634994507, 3.5768051147460938, 3.5853075
```

Esquema Stopper

- Clase Stopper
  - Público:
    - Parámetros configurables de la lectura láser
    - Constructor
      - Crea un publisher del topic *cmd_vel* con mensajes tipo *geometry_msgs/Twist*
      - Crea un subscirber del topic *base_scan* con mensajes del tipo *sensor_msgs/LaserScan*
    - Método *startMoving*
      - Implementa un bucle cerrado: mientras puede avanzar (dependiendo de la lectura del sensor) llama a la función *MoveForward.*
  - Privado:
    - Manejador del nodo
    - Publisher y Subscriber
    - Método *MoveForward*
    - Método *callback* para manejar lectura de sensor suscrito.


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

class Stopper
{
public:
    // Tunable parameters
    const static double FORWARD_SPEED_MPS = 0.2;
    const static double MIN_SCAN_ANGLE_RAD = -30.0/180*M_PI;
    const static double MAX_SCAN_ANGLE_RAD = +30.0/180*M_PI;
    const static float MIN_PROXIMITY_RANGE_M = 0.5; // Should be smaller than
sensor_msgs::LaserScan::range_max

    Stopper();
    void startMoving();

private:
    ros::NodeHandle node;
    ros::Publisher commandPub; // Publisher to the simulated robot's velocity command topic
    ros::Subscriber laserSub; // Subscriber to the simulated robot's laser scan topic
    bool keepMoving; // Indicates whether the robot should continue moving

    void moveForward();
    void scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan);
};
```

Stopper.cpp (1)

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

Stopper::Stopper()
{
    keepMoving = true;

    // Advertise a new publisher for the simulated robot's velocity command topic
    commandPub = node.advertise<geometry_msgs::Twist>("cmd_vel", 10);

    // Subscribe to the simulated robot's laser scan topic
    laserSub = node.subscribe("base_scan", 1, &Stopper::scanCallback, this);
}

// Send a velocity command
void Stopper::moveForward() {
    geometry_msgs::Twist msg; // The default constructor will set all commands to 0
    msg.linear.x = FORWARD_SPEED_MPS;
    commandPub.publish(msg);
};
```

Stopper.cpp (2)

```
// Process the incoming laser scan message
void Stopper::scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan)
{
    // Find the closest range between the defined minimum and maximum angles
    int minIndex = ceil((MIN_SCAN_ANGLE_RAD - scan->angle_min) / scan->angle_increment);
    int maxIndex = floor((MAX_SCAN_ANGLE_RAD - scan->angle_min) / scan->angle_increment);

    float closestRange = scan->ranges[minIndex];
    for (int currIndex = minIndex + 1; currIndex <= maxIndex; currIndex++) {
        if (scan->ranges[currIndex] < closestRange) {
            closestRange = scan->ranges[currIndex];
        }
    }

    ROS_INFO_STREAM("Closest range: " << closestRange);

    if (closestRange < MIN_PROXIMITY_RANGE_M) {
        ROS_INFO("Stop!");
        keepMoving = false;
    }
}
```

Stopper.cpp (3)

```
void Stopper::startMoving()
{
    ros::Rate rate(10);
    ROS_INFO("Start moving");

    // Keep spinning loop until user presses Ctrl+C or the robot got too close to an
obstacle
    while (ros::ok() && keepMoving) {
        moveForward();
        ros::spinOnce(); // Need to call this function often to allow ROS to process
incoming messages
        rate.sleep();
    }
}
```


```cpp
#include "Stopper.h"

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

    // Create new stopper object
    Stopper stopper;

    // Start the movement
    stopper.startMoving();

    return 0;
};
```

Stopper Output

Problems Tasks Console Properties Call Graph

<terminated> stopper Configuration [C/C++ Application] /home/roiyeho/catkin_ws/devel/lib/my_stage/stopper (10/25/13, 3:35 AM)

```
[0m[ INFO] [1382661340.576015273, 18109.700000000]: Closest range: 0.760001[0m
[0m[ INFO] [1382661340.667596025, 18109.800000000]: Closest range: 0.740001[0m
[0m[ INFO] [1382661340.768342773, 18109.900000000]: Closest range: 0.720001[0m
[0m[ INFO] [1382661340.867396332, 18110.000000000]: Closest range: 0.680001[0m
[0m[ INFO] [1382661340.966313085, 18110.100000000]: Closest range: 0.680001[0m
[0m[ INFO] [1382661341.067020022, 18110.200000000]: Closest range: 0.660001[0m
[0m[ INFO] [1382661341.172239501, 18110.300000000]: Closest range: 0.640001[0m
[0m[ INFO] [1382661341.269401730, 18110.400000000]: Closest range: 0.620001[0m
[0m[ INFO] [1382661341.371178013, 18110.500000000]: Closest range: 0.600001[0m
[0m[ INFO] [1382661341.465327863, 18110.600000000]: Closest range: 0.580001[0m
[0m[ INFO] [1382661341.565325407, 18110.700000000]: Closest range: 0.540001[0m
[0m[ INFO] [1382661341.668388784, 18110.800000000]: Closest range: 0.520001[0m
[0m[ INFO] [1382661341.771353545, 18110.900000000]: Closest range: 0.500001[0m
[0m[ INFO] [1382661341.868324993, 18111.000000000]: Closest range: 0.500001[0m
[0m[ INFO] [1382661341.967389002, 18111.100000000]: Closest range: 0.480001[0m
[0m[ INFO] [1382661341.967489121, 18111.100000000]: Stop![0m
```

roslaunch

- herramienta para lanzar fácilmente múltiples nodos ROS
  - local o via SSH
  - asignar valores a parámetros del *Parameter Server*
- Toma como entrada uno o más ficheros de configuración XML (con extensión **.launch)**, especificando:
  - parámetros a asignar
  - nodos a lanzar
- Si se usa ***roslaunch*** no hay que ejecutar ***roscore***

Ejemplo Launch File

- Fichero launch para lanzar el simulado Stage y el nodo *stopper*:

```
<launch>
  <node name="stage" pkg="stage_ros" type="stageros" args="$(find
stage_ros)/world/willow-erratic.world"/>
  <node name="stopper" pkg="my_stage" type="stopper"/>
</launch>
```

- Para ejecutarlo usar:

```
$roslaunch package_name file.launch
```

Entrega 1

- Implementar un algoritmo de navegación aleatoria. Ayudará basarse en *Stopper*.
- Ver la descripción en el material de la práctica.
- Entrega
  - GrupoLunes: 7 de Marzo a las 14:00.
  - GrupoMartes: 8 de Marzo a las 14:00.