Iniciación a ROS: Comandos, conceptos, prácticas, navegación, Behaviour Trees y otros conceptos y liberías de interés. Introducción a ROS 2 y diferencias con respecto a ROS.

Saúl Navajas Quijano*

19 de septiembre de 2023

*Grado en Ingeniería de Robótica Software, Universidad Rey Juan Carlos, `https://github.com/SaulN99`

Índice

1. Instalación de ROS Noetic en Ubuntu 20.04 **1**

2. Creación de un espacio de trabajo o workspace **2**

3. Comandos importantes: **3**

4. Conceptos sobre ROS. Conocimiento del funcionamiento de ROS y otros aspectos. **5**

4.1. Información importante sobre ROS: 5

4.2. Términos: 5

4.3. Otra información de ayuda: 6

5. Utilizando C++ para programar en ROS. Tipos de nodos y publicación de topics: **7**

5.1. Nodo simple: nodo_simple.cpp: 7

5.2. Nodo iterativo: nodo_iterativo.cpp: 8

5.3. Nodo publicador: nodo_pub.cpp: 9

5.3.1. ¿Cómo podemos publicar un topic? 11

5.4. Nodo subscriptor: nodo_sub.cpp: 11

6. Trabajar con robots reales. ROS en Kobuki: **13**

7. Como usar GitHub para trabajar en grupo. Repositorios remotos y pull-Requests: **15**

7.1. Gestionando los distintos repositorios: 15

7.2. Como trabajar en el repositorio grupal. GitHub Classroom: 16

8. Pautas y especificaciones para las prácticas: **17**

8.1. Creación de tests 17

9. Bump & Go básico **22**

10. Práctica 1: Bump & Go **29**

11. darknet_ros **30**

11.1. ¿Qué es darknet_ros? 30

11.2. Como usar y ejecutar darknet_ros en mi ordenador 30

12. Filtrado de color con OpenCV y espacio de color HSV **31**

13. TFs. Marcos de Referencia (frames) 32
13.1. Introducción . . . 32
13.2. Cambios de referencias . . . 32
13.3. Hacer un cambio de frames . . . 33

14. Behaviour Trees 35
14.1. ¿Qué es un Behaviour Tree? . . . 35
14.2. Ventajas principales frente a las máquinas de estados finitos: . . . 35
14.3. Conceptos básicos de los Behaviour Trees: . . . 35
14.4. Como funciona los ticks . . . 36
14.5. Tipos de Nodos . . . 36
14.6. Nodos Secuenciales ó Nodos Secuencia: . . . 36
14.7. Nodos Fallback . . . 37
14.8. Nodos Decoradores . . . 37

15. Práctica 2: Visual Behaviour 39

16. Mapeo y Navegación en ROS 40
16.1. localization.launch . . . 40
16.2. slam_mapping.launch . . . 41
16.3. navigation.launch . . . 42
16.4. costmap . . . 43
16.5. Funcionalidad de los nodos lanzamos en el mapeo y navegación . . . 44

17. Diálogo 46
17.1. DialogFlow: configuración y uso . . . 46
17.1.1. Intents . . . 46
17.1.2. Contexts . . . 46
17.1.3. Entities . . . 47
17.2. Como usar gb_dialog: . . . 47
17.3. Como registrar nuevos Intents: . . . 47
17.4. Acceder a los parámetros del Intento desde el código: . . . 48

18. Práctica Final: RoboCup Challenges 49

19. ROSBag 50
19.1. Comandos del ROSBag . . . 50

20. ROS WTF 51

21.PlotJuggler ROS **52**

22.rqt **53**

23.Últimos conceptos de ROS **54**
23.1. Configuración de variables 54

24.Introducción a ROS 2. De ROS a ROS 2 **55**
24.1. Introducción 55
24.2. Diseño de ROS 2 55
24.3. Empezar a trabajar con ROS 2 56
24.4. Comandos en ROS 2 y conectividad con otras máquinas del mismo dominio . 57
24.5. Primeros pasos en ROS 2 57
24.6. Interfaces, publicación y subscripción 57
24.7. Nodos en ROS 2 58
24.7.1. Nodo Simple 58
24.7.2. Nodo Iterativo Publicador 59
24.7.3. Nodo Iterativo Subscriptor 61
24.8. QoS en ROS 2 (Calidad de Servicios) 62
24.8.1. Compatibilidad entre distintos QoS 62
24.9. Launchers en ROS 2 63
24.10Parámetros en ROS 2 64
24.11Lifecycle Nodes 64

25.Preguntas de comprensión. Autoevaluación **65**

26.Referencias **67**

27.Agradecimientos **67**

1. Instalación de ROS Noetic en Ubuntu 20.04

El contenido de este libro está basado en la distribución de ROS Noetic.

Para instalar ROS Noetic en Ubuntu 20.04, se deben seguir los siguientes pasos especificados a continuación:

1. *Acceder a la Wiki de ROS*[1]*. Una vez estemos aquí, debemos seleccionar nuestra plataforma (en nuestro caso Ubuntu).*
2. *A continuación deberemos seguir los pasos 1.2, 1.3, y 1.4 (en este último, entre las 3 opciones de instalación disponibles, elegimos la versión DESKTOP-FULL y completamos los pasos de este apartado).*
3. *Una vez completados los pasos 1.2, 1.3 y 1.4, realizamos los pasos del apartado 1.5, pero solo aquellos que aparecen para "bash". Con esto modificaremos el fichero .bashrc y haremos que cada vez que abra una nueva terminal se ejecuta automáticamente el comando source ...*
4. *Por último, descargamos las depedencias especificadas en el apartado 1.6.*

Podemos comprobar mediante el comando ***roscore***[2] que la instalación se ha completado correctamente. También podemos comprobar la versión de ROS instalada mediante el comando ***rosversion -d***

[1] `http://wiki.ros.org/noetic/Installation`

[2] Se especifica el funcionamiento de este comando más abajo

2. Creación de un espacio de trabajo o workspace

Ahora que tenemos instalado ROS en nuestro ordenador, necesitamos crear un espacio de trabajo para poder trabajar. Los directorios de trabajo de ROS **por convenio deben acabar en catkin_ws**, donde ws viene de workspace. Para ello vamos a seguir los siguientes pasos:

1. *Abrimos una nueva terminal con **CTRL+ALT+T**. Creamos nuestra carpeta '**catkin_ws**' y dentro de ella una carpeta 'src'. A continuación se muestra un ejemplo de creación de estas carpetas dentro de una carpeta dedicada a ROS1 de una asignatura:*
   mkdir -p /arqSotwareRobots/ros1/catkin_ws/src

2. *Estando al nivel de la carpeta catkin ws en la terminal, escribimos **catkin_make**. Con este comando se inicializa el espacio de trabajo, y siempre se realiza desde el nivel de la carpeta catkin_ws.*

3. *Vamos a repetir el paso donde modificabamos el archivo .bashrc para añadir la ruta de ROS. Ahora tenemos que indicar ahí el directorio de trabajo que acabamos de crear. Para ello, abrimos el fichero .bashrc y escribimos source <rutaDirectorioTrabajo/devel/setup.bash>. En nuestro caso:*
   source /arqSoftwareRobots/ros1/catkin_ws/devel/setup.bash
   Guardamos los cambios y abrimos una nueva terminal.

4. *En la carpeta src clonamos el repositorio git de la asignatura con el que vamos a trabajar de forma **recursiva**. Esto es:*
   git clone –recursive https://github.com/<linkRepo>
   *Una vez clonado, **volvemos a catkin_ws** y volvemos a hacer **catkin_make** para inicializar el directorio de trabajo, ahora si, con todos los datos.*

NOTA: Durante el paso 4, al hacer catkin_make, **pueden ocurrir los siguientes problemas**:

a) Que se necesiten instalar paquetes de ros que no están instalados y son necesarios. Para ello, cuando nos ocurra un error de estos, instalamos el paquete con **sudo apt-get install ros-noetic-<paquete>** y volvemos a hacer **catkin_make**. Si el error surge con más de un paquete, seguir analogamente estos pasos. **Este problema no solo puede aparecer la primera vez que se instala todo lo de ROS, si no a lo largo de todo el curso cuando se empleen determinados paquetes o se ejecuten determinados launchers.**

b) Error **make -j8 -l8**. Para solucionarlo, instalar el siguiente paquete: **sudo apt-get install pyqt5-dev-tools** y volver a hacer **catkin_make**.

3. Comandos importantes:

A continuación se detallan la mayoría de los comandos básicos importantes. NOTA que esta lista no incluye todos los comandos recogidos en el libro ni en las prácticas propuestas. Es importante recordar que **todos estos comandos se lanzan desde el espacio de trabajo** que estamos empleando:

- **ls /opt/ros/** : Muestra todas las distribuciones de ROS que hay instaladas en el PC
- **rospack list** : Muestra el listado de paquetes en ROS.
- **apt-cache search ros-noetic** : Muestra el listado de paquetes disponibles para descargar que empiezen por ros-noetic.
- **catkin_make**: Compila el espacio de trabajo (catkin_ws).
- **catkin_make <package>**: Compila solo un paquete determinado del espacio de trabajo.
- **catkin_make -j1**: Compila el espacio de trabajo con un solo thread.
- **catkin_make roslint_nombrePaquete**: Pasa el roslint al paquete indicado.
- **roslaunch robots sim.launch** : Lanza la aplicación de Gazebo que muestra el robot (kobuki) y el escenario de trabajo.
- **roslaunch kobuki keyop keyop.launch** : Permite mover el robot kobuki del escenario en Gazebo utilizando el teclado.
- **roslaunch kobuki_node minimal.launch –screen** : Lanza todas las librerías del kobuki para poder utilizarlo.
- **rosnode list** : Ver todos los nodos que se están ejecutando
- **rostopic list** : Ver todos los topics del sistema
- **rosls <paquete>** : Enseña el contenido de un paquete.
- **rostopic info <topic>** : Ver información de un topic en concreto.
- **rosmsg list** : Ver lista de todos los mensajes.
- **roscd [directorioPaquete]**: Si solo pones roscd te lleva al devel de tu workspace, si indicas una dirección de un paquete, al de ese paquete.

- **rosmsg show <topicType>**: Ver contenido de un topic y sus campos.
- **rostopic echo scan [–noarr]** : Ver por pantalla el contenido de un mensaje (excepto arrays).
- **rosrun <paquete><ejecutable>**: Lanza el ejecutable de un paquete.
- **rosrun rviz rviz** : Lanza el visualizador rviz del paquete rviz. En la práctica, se lanza simplemente con el comando: rviz
- **rosrun rqtgraph rqtgraph** : Muestra a través del visualizador rqtgraph el grafo de computación.
- **rosrun rqt_console rqt_console**: Lanza la aplicación rqt_console del paquete rqt_console donde se puede observar todo lo que sale por ros_out con algo más de información.
- **rostopic pub -r <Hz><topic>+TAB +TAB** : publica un topic concreto a una frecuencia determinada en Hz.
- **roscore** : Inicializa el core/núcleo/master. Se apaga con CTRL+C

4. Conceptos sobre ROS. Conocimiento del funcionamiento de ROS y otros aspectos.

A continuación se detallan todos los conceptos aprendidos que hay que comprender para saber que estamos haciendo en cada momento, así como información básica pero muy importante sobre varios aspectos de ROS:

4.1. Información importante sobre ROS:

ROS emplea el *Sistema Internacional de medidas*, de modo que las medidas que se escriban en comandos o que nos aparezcan en la terminal como resultado de estos vendran en el S.I (esto es, en segundos, metros, radianes...). Además, en ROS decimos que somos **dextrogiros**.

En ROS hay 3 dimensiones: **workspace ó filesystem (formado por paquetes y metapaquetes, workspace...), comunidad (formada por distribuciones, repositorios, ROS Wiki, ROS Answers, ROS Discurse...) y grafo de computación (formado por nodos, el master, mensajes, topics, servicios, bags..)**

Para poder publicar un topic es necesario arrancar primero un core (con roscore). Lo que hace *roslaunch* es crear un core si no hay ninguno creado. El programa publica en el topicmsg.

4.2. Términos:

- **Nodo** : Es un proceso. Se distingue de los procesos de linux porque puede formar parte de un grafo de computación. Los nodos pueden comunicarse a través de 3 formas distintas: **publicaciones, servicios y acciones**. Cada vez que se inicia un nodo, se registra en el master/core, indicando que topic es y que topic va a publicar. Cada vez que alguien quiere subscribirse a un topic, le pregunta al core/master donde está ese topic (protocolo TCP/IP).

- **Grafo de computación** : Es un grafo creado por nodos y arcos. Cabe destacar que **es un modelo asíncrono (al utilizar una serie de publicadores y subscriptores)**. *Los servicios, por otro lado, si proporcionan un modelo síncrono al haber una conexión 1 a 1.* Por último, las acciones están orientadas para ordenes a largo plazo, de modo que se asigna una accion a un nodo y este va a actualizando el estado de la acción hasta que finaliza con éxito o con error. **El grafo de computación conforma la parte dinámica del robot** (pues son los procesos que están en ejecución).

- **Workspace** : Es la **parte estática del robot**. En él se indican que carpetas, ficheros y herramientas se encuentran instaladas, etc...
- **Topic** : Es un canal de comunicación para los mensajes entre nodos. Un topic NO puede tener varios mensajes de tipos distintos. Un nodo puede subscribirse o publicar en un topic.
- **Paquete** : Es la mínima unidad en la que se conserva el software de ROS, e incluye librerias, lanzadores, ejecutables, etc...
- **Distribución** : Es una colección de software basada en el núcleo de un determinado SS.OO.
- **Core** : Es el núcleo del grafo de computación con el que se comunican todos los nodos. El núcleo puede estár en remoto. Inicializará los principales procesos requeridos, entre ellos master, rosout y el servidor de parámetros. Master es el que provee los nombres de servicio para ROS.
- **rosout** : Es la salida por la que escriben todos los nodos (y MACROS[3]).

4.3. Otra información de ayuda:

Si en algún momento vemos que no tenemos un paquete instalado podemos hacer el comando **apt-cache search <paquete>** para que nos muestre los paquetes disponibles a descargar con ese nombre. Si lo querriamos instalar**, sudo apt-get install <paquete>**.

En caso de que al lanzar un nodo con rosrun se nos quede pillado y no podamos matarlo con CTRL+C, abrir una nueva terminal, buscar con **ps aux** el PID del proceso del nodo que se ha quedado pillado y hacer **kill -9 <PID>**.

[3]Veremos que es una MACRO más adelante.

5. Utilizando C++ para programar en ROS. Tipos de nodos y publicación de topics:

A continuación se explican varios nodos programados en lenguaje C++. Nota que en muchos de ellos se utilizarán referencias, que NO son lo mismo que los punteros.

En concreto, se van a explicar los siguientes tipos de nodos:

- **nodo_iterativo.cpp**
- **nodo_simple.cpp**
- **nodo_pub.cpp**
- **nodo_sub_obj.cpp**
- **nodo_sub_async.cpp**
- **nodo_sub.cpp**

Vamos a ver en detalle alguno de estos y su código:

5.1. Nodo simple: nodo_simple.cpp:

El fichero nodo_simple.cpp es el nodo más simple que vamos a ver. A continuación se muestra un snippet del código:

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

int main(int argc, char **argv)
{
    ros::init(argc, argv, "simple");
    ros::NodeHandle n;

    ros::spin();

    return 0;
}
```

Para empezar, **es MUY IMPORTANTE que pongamos siempre los espacios de nombres (es decir, escribir std::cout en vez de cout y evitarnos el using namespace) para evitar colisiones entre librerias si tienen el mismo método**. Es fundamental tener esto en cuenta.

La sentencia **ros::init()** parsea en busca de argumentos que me interesen, da un nombre al nodo y hace varias tareas más. (En efecto, se pueden pasar argumentos a los nodos, aunque todavía no lo hemos visto).

Un nodo es nodo (es decir, se incorpora al grafo de computación) cuando lo creamos con la sentencia **ros::NodeHandle()** (en este caso, creamos el nodo n). No obstante, también hay que tener en cuenta que *realmente solo existe un nodo internamente.*

Por último, la sentencia **ros::spin()** lo que hace es bloquear el programa (nodo) hasta que se hace CTRL+C. En verdad hace mas cosas: gestiona todos los mensajes entrantes que entran al nodo, permite a los subscriptores llamar a sus callbacks, etc...

5.2. Nodo iterativo: nodo_iterativo.cpp:

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

int main(int argc, char **argv)
{
    ros::init(argc, argv, "simple");
    ros::NodeHandle n;

    ros::Rate loop_rate(20);

    int count = 0;

    while (ros::ok())
    {
        ROS_INFO("iteration %d", count++);

        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

En este nodo aparecen varias sentencias que no conocemos, vamos a verlas:

ros::spinOnce(), a diferencia de spin(), se bloquea, mira a ver si hay que gestionar algún mensaje, y cuando acaba vuelve.

Con esta pequeña diferencia ya podemos entender una diferencia entre este nodo y el anterior. El nodo anterior *nodo_simple.cpp era un nodo pasivo* (pues se quedaba parado/-bloqueado), mientras que este *(nodo_iterativo.cpp) es un nodo activo*, y se puede decidir a que frecuencia se puede ir ejecutando.

ros::Rate loop_rate(20) hace que la frecuencia de iteración sea de 20 Hz (esto son, 50ms). Esta sentencia está vinculada con **loop_rate.sleep()**, pues esta sentencia intenta adaptarse a la frecuencia especificada. Si han pasado 10ms, espera 40ms. Si han pasado 40ms, espera 10ms. Si han pasado 2mins, no espera (y la ejecución de la siguiente instrucción se retrasa).

ROS_INFO(¨
iteration %d", count++) está en mayúscula porque es una **MACRO**. Una MACRO **es un tipo de constante**, se puede definir así:

*#**define HERZIOS(x) (20*x)***

Esta setencia que empieza con # es una *sentencia pre-procesador.* De modo que, si en el código se escribe HERZIOS(4), se sustituye por 20x4. Se comporta de manera similar al printf.

La MACRO imprime por ros_out (salida de los nodos que usan esta macro) su salida (sin necesidad de crear ningú publicador ni nada, porque está hecho así). En la salida aparece, como primer campo, **la criticidad del mensaje**, que pueden ser los siguientes (en orden de mayor a menor prioridad): [FATAL], [ERROR], [WARN], [INFO], [DEBUG]. Le sigue una marca de tiempo en segundos y el número de iteración.

Existe una herramienta llamada rqt_console del paquete rqt_console que lanza una aplicación donde poder ver todo lo que sale por ros_out con algo más de información. Se lanza, como es previsible, con *rosrun rqt_console rqt_console.*

5.3. Nodo publicador: nodo_pub.cpp:

El fichero nodo_pub.cpp puede intuirse por su nombre que está relacionado con el publisher[4]:

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

int main(int argc, char **argv)
{
    ros::init(argc, argv, "num_publisher");
    ros::NodeHandle n;
```

[4]Nota que hay una línea partida para que se vea la sentencia completa

```
ros::Publisher num_pub =
        n.advertise<std_msgs::Int64>("\message", 1);

ros::Rate loop_rate(10);

int count=0;

while (ros::ok())
{
    std_msgs::Int64 msg;
    msg.data = count++;

    num_pub.publish(msg);

    ros::spinOnce();
    loop_rate.sleep();
}

return 0
```

En la sentencia **ros::Publisher** se está especificando que se quiere crear un publicador, y que el topic se va a llamar /**message**. *Si NO estuviese la /, puede ocurrir que haya varios topics que se creen en un espacio de nombres*. Es decir, con la barra, si compilamos con catkin_make y vemos la lista de paquetes con rostopic list aparece como /**message**. No obstante, si quitamos la barra y seguimos los mismos pasos, podemos conseguir que el topic se llame, por ejemplo, **robot1/message.**

Esto se puede pensar de la siguiente forma: si tiene la /, se está escribiendo el topic desde la ruta absoluta, y siempre será /message. Si no se pone la barra, la ruta es relativa, y puede crearse ese topic en un espacio de nombres como hemos descrito en el ejemplo de arriba.

El 1 de la sentencia ros::Publisher *indica que el tamaño de la cola es 1*, y que solo puede publicar un solo mensaje entre spin y spin. La sentencia **while (ros::ok())** es equivalente a un while (true), siempre y cuando todo vaya bien (de ahí el ros::ok()).

5.3.1. ¿Cómo podemos publicar un topic?

Vamos a hacer un ejemplo publicando un topic de velocidades. Para ello, antes de nada lanzamos Gazebo con *roslaunch robots sim.launch*. En *rostopic list* vemos los distintos topics del robot. Entre ellos se encuentra **/mobile_base/commands/velocity**, correspondiente a la velocidad del robot. El tipo de este mensaje, como podemos observar con **rostopic info <topic>**, es *geometry_msgs/Twist*.

Con **rosmsg show <topicType>** podemos ver los campos del tipo de mensaje. En el caso de *geometry_msgs/Twist*, vemos que tiene dos vectores de dimensiones x y z, uno para la velocidad lineal y otro para la velocidad angular. Los vectores están compuestos por x y z que son de tipo float64.

Volvemos a nodo_pub.cpp y cambiamos el nombre del topic (que era /message) por /mobile_base/commands/velocity. Cambiamos el tipo (que era std_msgs::Int64) por geometry_msgs::Twist. Para que funcione es necesario hacer el include de la libreria con #include "geometry_msgs/Twist.h". Para hacer este paso es **IMPORTANTE** que vayamos al archivo **package.xml** y pongamos un *build depend y un exec depend* de geomtry_msgs, tal que así:

<build_depend>geometry_msgs</build_depend>
<exec_depend>geometry_msgs</exec_depend>

Ahora, vamos a **CMakeLists.txt** y donde ponía std_msgs (en find_package y en CATKIN_DEPENDS) ponemos geometry_msgs.

La diferencia que hemos de cambiar también en este nodo es que ya no vamos a publicar datos, si no que vamos a mover al robot. Cambiamos la sentencia que está dentro del while de msg.data = count++; por:

msg.linear.x = 1.0;

Compilamos de nuevo con *catkin_make*. Una vez compilado sin problemas, ya tenemos la posibilidad de publicar nuestro nodo publicador con

rosrun basics_ros nodo_pub

Vemos en Gazebo que nuestro robot kobuki se empieza a mover.

5.4. Nodo subscriptor: nodo_sub.cpp:

El fichero nodo_sub.cpp puede intuirse por su nombre que está relacionado con el subscriber[5]:

[5]Nota que hay dos líneas partidas para que se vean la sentencias completas

```
#include "ros/ros.h"
#include "std_msgs/Int64.h"

void messageCallback(consts
                     std_msgs::Int64::ConstPtr& msg)
{
    ROS_INFO("Data: [%ld]", msg->data)
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "num_subscriber");
    ros::NodeHandle n;

    ros::Subscriber sub = n.subscribe("\message", 1,
                                      messageCallback);

    ros::Rate loop_rate(10);

    while (ros::ok())
    {
        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0
```

Se parece mucho al nodo anterior (al publisher), a excepción de ciertas diferencias. Cuando se llama a spinOnce, se mira si hay algún mensaje. Si lo hay, llama al callback, de modo que se almacenan mensajes en la cola. En el método/función callback se usa como parámetro un puntero constante (const std_msg::Int::64ConstPtr & msg).

Por estilo, a los atributos miembros de una clasa se les pone al final de su nombre un subrayado _, para saber en todo momento que es cada cosa. Es muy recomendable hacerlo.

6. Trabajar con robots reales. ROS en Kobuki:

Para usar el kobuki, se recomienda primero enchufar el usb al portátil y después encender el kobuki (en ese orden). Recordamos que hay una serie de comandos y pasos para preparar el kobuki [6].

Una vez lanzado el kobuki, con *rostopic list* se puede observar los topics disponibles del robot. A continuación se muestran la mayoría (si no todos) los que nos aparecían disponibles:

- /diagnostics
- /diagnostics_agg
- /diagnostics_toplevel_state
- /joint_states
- /mobile_base/commands/digital_output
- /mobile_base/commands/external_power
- **/mobile_base/commands/led1** : Manejar led 1 del kobuki
- **/mobile_base/commands/led2**: Manejar led 2 del kobuki
- /mobile_base/commands/motor_power
- /mobile_base/commands/reset_odometry
- **/mobile_base/commands/sound**: Manejar sonido del kobuki
- **/mobile_base/commands/velocity**: Manejar velocidad del kobuki
- /mobile_base/debug/raw_data_command
- /mobile_base/debug/raw_data_stream
- /mobile_base/events/bumper
- **/mobile_base/events/button**: Manejar botón del kobuki
- /mobile_base/events/cliff
- /mobile_base/events/digital_input
- /mobile_base/events/power_system

[6] Ver algunos de estos en http://wiki.ros.org/kobuki/Tutorials/Examine Kobuki

- /mobile_base/events/robot_state
- /mobile_base/events/wheel_drop
- /mobile_base/sensors/bump_pc
- /mobile_base/sensors/core
- /mobile_base/sensors/dock_ir
- /mobile_base/sensors/imu_data
- /mobile_base/sensors/imu_data_raw
- /mobile_base/version_info
- /odom
- /rosout
- /rosout_agg
- /tf

En negrita aparecen algunos importantes y con los que se recomienda jugar un poco. Con *rostopic info <topic>*y *rosmsg show <topic>* se puede observar la información de los topics y sus **contenidos y campos**; para después, con ***rostopic pub -r <Hz><topic>+TAB +TAB***, publicar esos topics y manejar las distintas características del robot (velocidad, leds, sonido...). Recordamos que al publicar estos topics **se puede utilizar la opción *roslaunch kobuki keyop keyop.launch* para manejar el robot con las teclas** (por ejemplo, cuando publicamos el topic de velocidad).

También es importante repasar como funciona la cámara/sensor del kobuki en Gazebo y en rqt_rviz. En este último se puede ver gráficamente las coordenadas de sus ¨ items"(el de la base, el de la cámara que estaba algo más arriba en el eje Z, etc...).

7. Como usar GitHub para trabajar en grupo. Repositorios remotos y pullRequests:

A continuación se detalla como se debe usar GitHub para trabajar en un proyecto grupal, mediante el uso de remotes, forks y pullRequest.

7.1. Gestionando los distintos repositorios:

Imaginemos que existe un repositorio de GitHub de otra persona que queremos utilizar. Para empezar a trabajar en equipo, clonamos ese repositorio con **git clone <urlRepoGitHub>**. Si en este punto hacemos **git remote show**, vemos que tenemos una referencia/remote (llamada origin) al repositorio original.

Imaginemos que queremos colaborar con esa persona del repositorio que hemos clonado. Hacemos un **fork** de su repositorio, pulsando en el botón "fork"de la pagina de GitHub del repo original.

Desde nuestro repositorio local (clonado), **creamos una rama con *git checkout -b <nombreRama>*** y hacemos los cambios que veamos nosotros oportunos. Si procedemos ahora a hacer los pasos típicos para subir los cambios, es decir: **git add .**, luego **git commit -s**[7] **-m 'comentario'** y **git push**, vemos que a la hora de hacer el push nos pide que especifiquemos donde queremos subir los cambios. Esto es lógico, pues hemos clonado el repo a nuestro PC (estamos trabajando en local) pero también hemos hecho un fork (en GitHub) también del proyecto original.

Nosotros *NO podemos mandar los cambios a la rama origin, por que es la correspondiente del repositorio original que hemos clonado* (y que es de otra persona, no es nuestro). En este momento nos creamos un nuevo remote con **git remote add <nombre><urlForkedRepo>**. Nota que la URL debe ser la de tu repositorio forkeado, NO la del original. Especificamos que queremos subir los cambios a ese remote con **git push –set-upstream <nombreRemote><archivos>**. Al hacer esto estamos haciendo push a nuestro repo forkeado. Una vez los cambios los tengamos ahí, podemos hacer **pullRequest** para compartir estos cambios con el dueño del proyecto original. En el pullRequest podemos entablar conversación con la otra persona, y se verá una línea del tiempo donde se ven los mensajes, los commits hechos y si se acepta o se deniega esa petición.

Si esos cambios por ejemplo necesitán otros adicionales podemos seguir haciendo git push y se verán dichos commits y cambios en la línea de conversaciones del pullRequest. En el momento en el que la otra persona acepte nuestros cambios y le de a merge, *se pegarán*

[7] El -s sirve para firmar el cambio

los cambios al repositorio original del proyecto, y nos convertiremos automáticamente en contribuidores del repositorio de la otra persona. A partir de este momento ya podemos eliminar el branch con el que estabamos trabajando.

(En este punto se encuentra el repositorio de trabajo de la asignatura en GitHub Classroom).

7.2. Como trabajar en el repositorio grupal. GitHub Classroom:

Para trabajar con los integrantes del grupo en nuestro repo de la asignatura:

- 1º Creamos una rama para hacer nuestra tarea asignada: **git checkout -b <nombreRama>**
- 2º Hacemos los cambios y creamos los archivos y todo lo que tengamos que hacer de trabajo.
- 3º Llevamos los cambios al repo en GitHub: **git add .**, **git commit -s -m 'comentario'** y **git push** . Nos pondrá de nuevo el mensaje de antes, hacemos **git push –set-upstream origin <ficherosASubir>**.
- 4º En el proyecto nosotros somos contribuidores, luego podemos hacer un **pullRequest** como antes pero en este caso nosotros mismos podríamos aceptar nuestro pullRequest (Lo suyo es que lo acepten los otros integrantes del grupo para que vean los cambios).

8. Pautas y especificaciones para las prácticas:

Como es evidente, vamos a trabajar en C++ con dos tipos de ficheros: los .cpp que contienen el código y las cabeceras .h. Estas últimas **NO DEBEN ESTÁR UBICADAS EN LA CARPETA src**, si no en una llamada *includes*, *y dentro de esta en otra subcarpeta con el nombre de su paquete.*

Para indicar donde debe buscar el programa las cabeceras, abrimos el archivo **CMakeList.txt** y en el apartado include_directories.

8.1. Creación de tests

Tiene como objetivo comprobar de forma rápida y sencilla los componentes más basicos del código que se ha escrito, de modo que se pueda detectar fallos rápidamente.

Para crear un test, abrimos la terminal, vamos a nuestro workspace (catkin_ws) y en src, creamos un paquete nuevo para los tests con el comando **catkin_create_pkg <nombre>dependencies <dependencia1>... <dependenciaN>**.

Pongamos por ejemplo que queremos crear un test para un laser que necesita las librerías roscpp y el std_msg. Hacemos *catkin_ create_ pkg tests_ clase dependencies roscpp std_ msgs*. Se creará un esqueleto del paquete, con su CMakeList.txt, su package.xml, un directorio src (para los .cpp) y un directorio include (para los .h).

Ahora creamos en el directorio include nuestra cabecera .h, por ejemplo, **FakeLaser.h**. En la cabecera escribimos lo correspondiente de las cabeceras .h y un namespace con el mismo nombre que el paquete que hemos creado. Dentro de ese namespace creamos nuestra clase FakeLaser:

```
#ifndef TESTS_CLASE_FAKE_LASER_H__
#degine TESTS_CLASE_FAKE_LASER_H__

namespace tests_clase
{

class FakeLaser
{
public:
    FakeLaser(); //Constructor
    void fill_vector(); //Rellena el vector
    std::vector<double> get_vector(); //Getter del vector
protected:
```

```
    std::vector<double> readings_;
}

} // namespace tests_clase

#endif //TESTS_CLASE_FAKE_LASER_H__
```

Recordamos que en C++ '**no' se usa arrays, se usan vectores** (de ahí que en el código tenga un atributo que es un vector de tipo double llamado readings_ (lecturas del laser).

Ahora definimos el archivo .cpp. Creamos el archivo **FakeLaser.cpp** dentro de src. Dentro del archivo debemos especificar los includes y lo que hace cada uno de los métodos de la clase creada:

```
//Incluimos libreria vector de C++
#include <vector>

// Incluimos cabecera FakeLaser.h del paquete tests_clase
#include "tests_clase/FakeLaser.h"

namespace tests_clase
{

FakeLaser::FakeLaser() //Constructor
{
}

void
FakeLaser::fill_vector(const std::vector<double> & values)
{
    readings_ = values;
}

std::vector<double>
FakeLaser:: get_vector()
{
    return readings_;
}
} //namespace tests_clase
```

Creamos ahora la función main.cpp del programa, que imaginemos que rellena el vector con ciertos datos:

```
#include "tests_clase/FakeLaser.h"

int main(int argc, char * argv[])
{
    tests_clase::FakeLaser fake_laser;

    fake_laser.fill_vector({0.0, 1.0, 2.0});

    return 0;
}
```

Ya tenemos todos los archivos listos para crear el Test. En **CMakeLists.txt** podemos añadir una línea para usar una versión específica de C++ (por ejemplo, la 17), y también indicamos que vamos a añadir la librería:

```
set(CMAKE_CXX_STANDARD 17)

...

include_directories(
    include
    $(catkin_INCLUDE_DIRS)}
    )

add_library(${PROJECT_NAME}
    src/${PROJECT_NAME}/FakeLaser.cpp
    )

add_dependencies(...)

add_executables(...)
add_dependencies(...)
target_link_libraries(...)

roslint_cpp()
```

Ahora compilamos con catkin_make. Una vez compilado, añadimos las dependencias de roscpp y std_msgs en **package.xml:**

```
<build_depend>roscpp</build_depend>
<build_depend>std_msgs</build_depend>

<build_export_depend>roscpp</build_export_depend>
<build_export_depend>std_msgs</build_export_depend>

<exec_depend>roscpp</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

Si ahora compilamos con **catkin_make roslint_tests_clase**, roslint nos indica si hemos tenido algún error de estilo. En efecto, **hay un formato de estilo global especificado por roslint que hay que acatar**.

Ahora, dentro del directorio tests_clase, creamos un nuevo directorio llamado tests y en él un archivo .cpp para nuestros tests. En este caso, se ha llamado **my_test.cpp**. Globalmente se utilizan los tests de Google, como se ve a continuación:

```
#include <vector>

#include <gtest.gtest.h>
#include "ros/ros.h"
#include "tests_clase/FakeLaser.h"

class TestFakeLaser : public tests_clase::FakeLaser
{
public:
    const std::vector<double> & get_vector()
    {
        return readings_;
    }
};

TEST(my_test; test_set_reading)
{
    TestFakeLaser fake_laser;
    fake_laser.fill_vector({0.0,1.0,2.0})
```

```
    const std::vector<double> & vector = fake_laser.get_vector();

    //TEST el tamano del vector es 3
    ASSERT_EQ(vector.size(), 3);
    //TEST el valor de la posicion 0 del vector es ~0
    ASSERT_NEAR(vector[0], 0.0, 0.000001);

}

int main(int argc, char**argv[])
{
    testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}
```

Ya solo tendríamos que añadir al CMakeList.txt un catkin_add_gest(my_tests tests/my_test.cpp)... para crear un test que se le indica donde está, los enlazados y dependencias.

A partir de este momento, ya solo tenemos que hacer **catkin_make** my_tests y nos dirá en que ruta se han creado. *Si copiamos esa ruta en la terminal y le damos al ENTER automáticamente se ejecutarán todos los tests creados*. Puede ser que los tests se pasen o se fallen.

9. Bump & Go básico

El comportamiento de un Bump & Go es similar a los robots aspiradora: el robot se mueve hacia adelante hasta que choca con la pared. Una vez que el robot choca con la pared, se mueve para atras y gira hacia la izquierda, para ir de nuevo hacia adelante hasta que choque de nuevo y repita el proceso.

Como vemos, el comportamiento del robot puede ser plasmado en el programa como una **máquina de estado finito.**. Para esta primera práctica se facilita en el GitHub classroom la carpeta del proyecto del bump and go, que hay que **clonar en el src de nuestro workspace con git clone <dir>**. Una vez clonada, ya se puede empezar a trabajar, creando una rama para nuestra tarea. Dentro de esta carpeta tenemos:

- **package.xml**
- **CMakeList.txt**
- **include/fsm_bump_go** : Carpeta include para las cabeceras .h. Dentro de esta se encuentra la cabecera BumpGo.h
- **src/** : Carpeta que contiene, por un lado, el fichero bumpgo_node.cpp, que es el fichero principal (main) del programa. Dentro de src se encuentra la subcarpeta **fsm_bump_go**, y dentro el fichero BumpGo.cpp.

En esta primera práctica se da todos los ficheros completos a excepción del BumpGo.cpp, que está incompleto. Este es el fichero que, por asi decirlo, contiene la máquina de estados finito (es decir, cuando el robot está girando, moviéndose hacia delante, etc...). Tenemos marcados lo que tenemos que completar, que son las siguientes cosas:

- **sub_bumber_ = n_.subscribe()**: Es decir, tenemos un nodo llamado n_ en el que tenemos que crear un subscriptor.
- **pub_vel_ = n_.advertise()**: Es decir, tenemos el nodo llamado n_ en el que tenemos que crear un publicador.
- **Método BumpGo::bumperCallback** : Este es el callback del nodo subscriptor, que está incompleto.
- **Velocidades en cada uno de los estados**: Tanto la velocidad lineal como la velocidad angular, que habra que poner un valor u otro dependiendo de si es velocidad angular o lineal y de en que estado nos encontramos.

- **pub_vel_.publish(...)**: Publicador del nodo, tenemos que indicar donde se va a publicar.

A continuación se muestra la solución[8]:

```
#include "fsm_bump_go/BumpGo.h"

#include "kobuki_msgs/BumperEvent.h"
#include "geometry_msgs/Twist.h"

#include "ros/ros.h"

namespace fsm_bump_go
{

BumpGo::BumpGo()
: state_(GOING_FORWARD),
  pressed_(false)
{
  /*
  Como queremos hacer que el robot se mueva, debemos
  crear un publicador y un subscriptor.

  Para crear el nodo subscriptor, tenemos que indicar,
  como vimos en las clases pasadas con
  el nodo_sub.cpp de basics_ros, el topic para
  el subscriptor, que es del bumper:
  */
  sub_bumber_ = n_.subscribe
  ("/mobile_base/events/bumper", 1,
    &BumpGo::bumperCallback, this);

  /*
  Como queremos hacer que el robot se mueva, debemos
  crear un publicador y un subscriptor.

  Para crear el nodo publicador, n_.advertise, tenemos que
```

[8]Hay muchas líneas cortadas para que se vea el código entero

```
  mirar con rostopic list cual era el
  topic que hacia mover el robot (en este caso es
  "/mobile/base/velocity") y ver su Type, que era
  geometry_msgs/Twist. Como ya tenemos el
  #include "geometry_msgs/Twist.h",
  simplemente creamos el nodo
  indicando el topic y el tipo:
  */
  pub_vel_ = n_.advertise<geometry_msgs::Twist>
             ("/mobile_base/commands/velocity", 1);

}

void
BumpGo::bumperCallback
        (const kobuki_msgs::BumperEvent::ConstPtr& msg)
{
  /*
  Tenemos que especificar en el callback que se debe hacer

  Vamos a especificar con la constante ROS_INFO que
  imprima por rosout el estado.
  Ademas, vamos a poner que en la variable pressed_
  este el puntero del mensaje al estado:
  */
  ROS_INFO("State: [%d]", msg->state);
  pressed_ = msg->state;
}

void
BumpGo::step()
{
  geometry_msgs::Twist cmd;

  /*
  Switch que emula una maquina de estados finito:
```

```
    dependiendo del estado del kobuki hace una cosa u otra.
    */
    switch (state_)
    {
      case GOING_FORWARD:
        /*
        Vamos a especificar a que velocidad debe ir lineal y
        angularmente cuando va hacia delante

        Tener en cuenta que una velocidad lineal de 1.0 es
        una velocidad MUY ALTA. Lo ideal es algo tipo 0.2.
        Cuando vaya para adelante no queremos que gire,
        luego ponemos una velocidad angular de 0.
        */
        cmd.linear.x = 0.2;
        cmd.angular.z = 0;

        if (pressed_)
        {
          press_ts_ = ros::Time::now();
          state_ = GOING_BACK;
          ROS_INFO("GOING_FORWARD -> GOING_BACK");
        }

        break;
      case GOING_BACK:
        /*
        Vamos a especificar a que velocidad debe ir lineal y
        angularmente cuando va hacia delante

        Ponemos -0.2 para que se mueva hacia atras con la
        misma velocidad que se movia para adelante.
        Cuando vaya para atras no queremos que gire,
        luego ponemos una velocidad angular de 0.
        */
        cmd.linear.x = -0.2;
        cmd.angular.z = 0;
```

```
      if ((ros::Time::now() - press_ts_).toSec() >
                                    BACKING_TIME )
      {
        turn_ts_ = ros::Time::now();
        state_ = TURNING;
        ROS_INFO("GOING_BACK -> TURNING");
      }

      break;
    case TURNING:
      /*
      Vamos a especificar a que velocidad debe ir lineal y
      angularmente cuando vaya a girar

      Cuando gire no queremos que se mueva a la vez que
      esta girando, luego la velocidad lineal es 0.
      Cuando gire vamos a especificar que lo haga con una
      velocidad del 0.4, que es una velocidad "normal".
      */
      cmd.linear.x = 0;
      cmd.angular.z = 0.4;

      if ((ros::Time::now()-turn_ts_).toSec() >
                                    TURNING_TIME )
      {
        state_ = GOING_FORWARD;
        ROS_INFO("TURNING -> GOING_FORWARD");
      }
      break;
    }

    /*
    Finalmente, ponemos que publique por la terminal:
    */
    pub_vel_.publish(cmd);
}
```

```
} // namespace fsm_bump_go
```

Recuerda que una vez modificado esto hay que modificar el CMakeList.txt para añadir los paquetes necesarios a buscar[9]:

```
cmake_minimum_required(VERSION 2.8.3)
project(fsm_bump_go)

set(CMAKE_CXX_STANDARD 17)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
  geometry_msgs
  kobuki_msgs
  roslint
)

catkin_package(
CATKIN_DEPENDS roscpp std_msgs kobuki_msgs sensor_msgs
                    geometry_msgs
)

include_directories(
  ${catkin_INCLUDE_DIRS}
  include
)

add_library(${PROJECT_NAME}
  src/fsm_bump_go/BumpGo.cpp
)

add_executable(bumpgo_node src/bumpgo_node.cpp)
target_link_libraries(bumpgo_node ${catkin_LIBRARIES}
                            ${PROJECT_NAME})
```

[9]Líneas cortadas para que se vea todo.

```
roslint_cpp(
 src/bumpgo_node.cpp
 src/fsm_bump_go/BumpGo.cpp
 include/fsm_bump_go/BumpGo.h
)
```

En el package.xml ya están incluidas las dependencias de kobuki_msgs, luego no es necesario modificarlo. Una vez hecho todo esto, compilamos con *catkin_make* y una vez compilado, ya podemos probar nuestro código en gazebo lanzando primero **roslaunch robots sim.launch** y luego **rosrun fsm_bump_go bumpgo_node**. El robot empezara a moverse como hemos indicado que haga.[10]

[10]Nota que antes de todos estos pasos seguramente sea necesario lanzar el kobuki con **roslaunch kobuki_node minimal.launch –screen** para que aparezcan todos los nodos del koubki en rostopic list y funcione todo correctamente.

10. Práctica 1: Bump & Go

La primera práctica que se plantea consiste en la realización de un Bump & Go. Dicha práctica está dividida en 3 funcionalidades básicas:

- **Funcionalidad 1:** La descrita en el apartado anterior. Bump & Go básico que emplea una máquina de estados finito. El Bump & Go se mueve siempre hacia el mismo lado cuando choca.
- **Funcionalidad 2:** Partiendo de la funcionalidad 1, hacer que el kobuki se mueve a izquierda o derecha dependiendo del lado en el que choque el bumper. Cuando choque en el centro, girara a un lado arbitrario (da igual si a la derecha que si a la izquierda).
- **Funcionalidad 3:** Haciendo uso del sensor **lidar**, implementar una nueva funcionalidad que permita al kobuki no chocarse con los obstáculos, leyendo del lidar la distancia frontal que tiene con el obstaculo de delante. Cuando el kobuki se encuentre cerca del obstáculo se detendrá, girará y seguirá en otra dirección.

A continuación se facilita el link al repositorio público de la práctica realizada por el grupo **TayRos**. **Nota que dicha práctica está bajo licencia**:

- `https://github.com/Docencia-fmrico/bump-and-go-with-fsm-tayros`

11. darknet_ros

11.1. ¿Qué es darknet_ros?

El paquete de ros **darknet_ros**[11] es un paquete desarrollado para la detección de objetos en cámaras. Permite delimitar mediante **Bounding Boxes** determinados objetos que está percibiendo de la cámara.

Este paquete emplea el algoritmo **YOLO**, acrónimo de You-Only-Look-Once, es un sistema de código abierto del estado del arte para detección de objetos en tiempo real.

En el repositorio público del paquete se encuentra toda la información necesaria para poder usarlo. Lo más importante a tener en cuenta con darknet son los siguientes aspectos:

1. Puedes ajustar la precisión con la que se detectan los objetos en los ficheros de configuración .yaml que encontrarás dentro de la carpeta config del paquete. (En concreto, yolov2-tiny.yaml). "value" especifica el % de similitud que debe encontrar la cámara con un objeto para que lo muestre como tal.

2. darknet_ros emplea como dependencias OpenCV y boost (una librería de C++). Es imprescindible tenerlas instaladas para poder emplear darknet.

11.2. Como usar y ejecutar darknet_ros en mi ordenador

Una vez asegurado que tu espacio de trabajo compila sin errores (cartkin_make) y que la instalación del paquete de darknet_ros a sido correcta, los pasos a seguir para lanzar darknet son los siguientes:

1. Lanzar un core con **roscore**

2. Activar cámara del ordenador con **rosrun usb_cam usb_cam_node**

3. Abrir **rviz** y en add (by topic) elegir la cámara de tu ordenador. Copiar en el portapapeles el nombre del topic de la cámara

4. Lanzar darknet con **roslaunch darknet_ros darknet_ros.launch image:=TOPIC**. En mi caso:
   roslaunch darknet_ros darknet_ros.launch image:=/usb_cam/image_raw

[11] `https://github.com/leggedrobotics/darknet_ros`

12. Filtrado de color con OpenCV y espacio de color HSV

En el repositorio del curso, hay un paquete llamado **cameras_cpp**. En el se encuentran 3 herramientas muy importantes relacionados con las imagenes y el filtrado de color.

- **rosrun cameras_cpp nodo_camera**
- **rosrun cameras_cpp nodo_rgbd_filter**
- **rosrun cameras_cpp nodo_rgbd_tf**

El primero de estos simplemente crea un topic para la cámara filtrada (/hsv/image_filtered) que podemos ver en rviz. Al igual que con darknet, se le puede pasar como parámetro al ejecutar el rosrun el topic de la cámara.

El segundo sirve para filtrar la imagén mediante el espectro de color HSV (Hue-Saturation-Value). Al ejectuarlo, se abrirá una ventana que permite modificar los umbrales máximos y mínimos de cada componente del espectro de color.

El tercero de ellos se emplea más adelante junto con las TFs. Permite sacar la TF del objeto filtrado siempre y cuando este se encuentre en los límites de la cámara que se está empleando.

13. TFs. Marcos de Referencia (frames)

13.1. Introducción

Las TFs son una especie de vectores que unen marcos de referencia(frames), permitiendo al usuario co-relacionar las coordenadas de un robot u objeto desde distintas perspectivas.

Como ejemplo, si se utiliza un kobuki y las añadimos para poder visualizarlas en el rviz (con add, "TF"), veremos que hay muchos frames en el kobuki. Especial importancia y relevancia tienen los dos siguientes frames:

- **odom**: Es el frame de referencia que indica el origen de coordenadas del robot.
- **base_footprint**: Es el frame en donde se encuentra la base del kobuki (Ten en cuenta que la cámara se encuentra algo más elevada, la base a unos centímetros del suelo, etc...).

13.2. Cambios de referencias

Las TFs son una herramienta muy útil para, por ejemplo, hallar la posición relativa de un objeto con respecto a nuestro robot. Nota que hay un aspecto muy importante a tener en cuenta: no existe un único frame, si no que existen múltiples frames y cada uno de ellos se encuentra en terminos absolutos en lugares distintos.

Es por ello que cuando queramos, por ejemplo, sacar la posición relativa de un objeto con respecto a nuestro robot, habría que hacer un çambio de ejes de referencia"del frame donde se encuentra el objeto y pasarlo al mismo frame donde se encuentra el robot, para que en caso de querer seguir al objeto, el robot lo siga correctamente y no vaya a las coordenadas del robot que están en otra referencia distinta a la del propia robot.

En las TFs existe una fórmula para poder hacer cambios de frames fácilmente. Imagina que tenemos una TF (recuerda que es un vector que une dos frames) que va del frame A al frame B, llamada: fromA2B. Imagina que tenemos una segunda TF que va del frame B al C, llamada fromB2C. Podemos hallar la TF del frame A al C mediante la fórmula:

fromA2C = fromA2B * fromB2C

Es decir, sea el producto de dos TFs donde la primera TF tiene como origen X y como destino Y; y sea una segunda TF que tiene como origen Y y como destino Z. La TF que va de origen X a destino Z es igual a la multiplicación de las dos anteriores en el orden nombrado, esto es:

fromX2Z = fromX2Y * fromY2Z

Mira en la carpeta del curso, dentro del paquete **geometry_tf**, los nodos **nodo_tf_pub y nodo_tf_sub**. En ellos se encuentra el código relacionado con las TFs.

En el nodo publicador, la sentencia **tf2_ros::TransformBroadcaster br** crea un broadcaster de transformadas llamado br. Un broadcaster viene a ser el publicador de las TFs.

En el nodo subscriptor, las sentencias **tf2_ros::Buffer buffer y tf2_ros::TransformListener listener(buffer)** crean el buffer que emplea el Listener de transformadas. Esto viene a ser el subscriptor de las TFs, que va almacenando en su buffer las TFs que va escuchando del Broadcaster.

13.3. Hacer un cambio de frames

Imaginemos que, por ejemplo, queremos hacer un cambio del frame base_footprint al frame odom. En la siguiente imágen se muestra como se hace:

```
28  int main(int argc, char **argv)
29  {
30    ros::init(argc, argv, "tf_sub");
31    ros::NodeHandle n;
32
33    tf2_ros::Buffer buffer;
34    tf2_ros::TransformListener listener(buffer);
35
36    ros::Rate loop_rate(1);
37
38    while (ros::ok())
39    {
40      geometry_msgs::TransformStamped bf2odom_msg;
41      geometry_msgs::TransformStamped odom2bf_msg;
42      geometry_msgs::TransformStamped odom2obj_msg;
43      geometry_msgs::TransformStamped obj2odom_msg;
44
45      tf2::Stamped<tf2::Transform> bf2odom;
46      tf2::Stamped<tf2::Transform> odom2bf;
47      tf2::Stamped<tf2::Transform> odom2obj;
48      tf2::Stamped<tf2::Transform> obj2odom;
49
50      tf2::Transform bf2obj;
51      tf2::Transform obj2bf;
52
53      std::string error;
54
55      if (buffer.canTransform("base_footprint", "odom", ros::Time(0), ros::Duration(0.1), &error))
56      {
57        bf2odom_msg = buffer.lookupTransform("base_footprint", "odom", ros::Time(0));
58
59        tf2::fromMsg(bf2odom_msg, bf2odom);
60
61        double roll, pitch, yaw;
62        tf2::Matrix3x3(bf2odom.getRotation()).getRPY(roll, pitch, yaw);
63
64        ROS_INFO("base_footprint -> odom [%lf, %lf, %lf] %lf ago",
65          bf2odom.getOrigin().x(),
66          bf2odom.getOrigin().y(),
67          yaw,
68          (ros::Time::now() - bf2odom.stamp_).toSec());
69      }
70      else
71      {
72        ROS_ERROR("%s", error.c_str());
73      }
```

En la línea 55 hacemos **buffer.canTransform** para intentar cambiar de base_footprint al frame odom en ese preciso momento (ros::Time(0)), con una duración de 0.1 segundos y pasándole una referencia al string de error (para que en caso de que no podamos hacer la transformacion imprimamos un mensaje de error).

En la linea 57, se hace como tal la transformación (se "guarda.[en] b2odom_msg, que es del tipo **geometry_msgs::TransformStamped**.

Luego con **tf2::fromMsg** se pasa de ese mensaje a la tf como tal (variable bf2odom de tipo **tf2::Stamped<tf2::Transform**).

En las línea 61 y 62 obtenemos la orientación.

Por último, en las líneas 65 y 66 obtenemos las coordenadas X e Y. También podríamos haber obtenido:

- La **distancia** del frame basefootprint al frame odom con **bf2odom.getOrigin().length()**
- El **ángulo**, con **atan2(bf2odom.getOrigin().y(), bf2odom.getOrigin().x())**;

14. Behaviour Trees

14.1. ¿Qué es un Behaviour Tree?

En el apartado anterior se vio como realizar un sencillo funcionamiento de un Bump & Go empleando como estructura central una máquina de estados finitos.

No obstante, existe otra estructura muy importante en el mundo de la robótica: los **behaviour trees**. Los beaviour trees es una forma de estructurar los cambios entre diferentes tareas en un agente autónomo, como puede ser un robot.

14.2. Ventajas principales frente a las máquinas de estados finitos:

Las ventajas principales que tienen son:

- **Estructura jerárquica**: Como indica su propio nombre, la estructura es en forma de árbol con nodos y arcos, siguiendo una modalidad jerárquica, donde los nodos padres son los de más arriba y los nodos hijos son los de más abajo.
- **Representación gráfica con significado semántico**: Esto es, que mirando el gráfico es mucho más intuitivo y más fácil de entender que las FSM[12].
- **Son más expresivas** : Gracias a los distintos tipos de nodos que veremos a continuación permite crear flujos de trabajo más complejos que las FSM, creando comportamientos más avanzados.

NOTA IMPORTANTE: Cabe destacar que **estas estructuras deben especificarse en un archivo XML dentro del proyecto y añadirlo en el CMAKELIST con el fin de poder compilar y usarlos correctamente**. En la librería de *software_ course hay una subcarpeta llamada behaviour_ tree donde se puede ver como hacer esto*.

14.3. Conceptos básicos de los Behaviour Trees:

Cuando se quiere arrancar o activar un Behaviour Tree, **se manda una señal llamada "tick" a la raíz del árbol** (nodo con más importancia o nodo mayor) del Behaviour Tree, propagandose a continuación por el arbol hasta que llega a un nodo hoja (nodo menor).

Un nodo del arbol que recibe un "tick" ejecuta un callback. **Este callback puede retornar 3 posibles valores: SUCCESS, FAILURE o RUNNING**[13].

Además, si un nodo tiene 1 o más hijos, él es responsable de enviarles el tick, basándose en su estado.

Por último, decir que cada BT contiene una estructura de datos llamada **BlackBoard**, que mantiene variables y valores relevantes. *Podemos acceder a la BlackBoard a través de un nodo para leer o escribir datos durante la ejecución de un BT*. No obstante, las BlackBoards no pueden ser compartidas desde distintos BTs.

[12] A partir de ahora nos referiremos a las máquinas de estado finito como FSM

[13] En caso de que la acción sea asíncrona y necesite más tiempo para procesarse

14.4. Como funciona los ticks

La **secuencia** es el ejemplo más simple de **nodo de control**. La secuencia ejecuta a los nodos hijos 1 a 1, y va esperando a que los hijos le envien el callback. *Imaginemos una secuencia con dos hijos.* **El proceso es el siguiente**:

1. El primer tick establece el nodo Secuencia en **RUNNING**(Naranja).
2. La secuencia tickea al primer hijo, y este devuelve el callback, que imaginamos como **SUCCESS** (verde).
3. A continuación se tickea al siguiente hijo, que devuelve su callback. Imaginamos que también devuelve **SUCCESS** (verde),
4. Como los dos hijos han devuelto SUCCESS y ya no hay más hijos por tickear, la secuencia devuelve el callback **SUCCESS** (verde).

14.5. Tipos de Nodos

Tipo de Nodo	**Nº de hijos**	**Funcionamiento**
Nodo de Control	1,2,..,N	Tickea a un hijo basandose en el resultado de sus nodos hermanos o de su propio estado
Nodo Decorador	1	Puede alterar el resultado de su hijo o tickearlo múltiples veces
Nodo Condicional	0	No altera el sistema. No debe devolver RUNNING, solo SUCCESS o FAILURE
Nodo de Acción	0	Puede alterar el sistema

En este contexto de Nodos de acción, cabría distiguir entre nodos síncronos y asíncronos.

Los nodos síncronos se ejecutan automáticamente y bloquean el arbol hasta que devuelve SUCCESS o FAILURE.

Los nodos asíncronos pueden retornar RUNNING para comunicar que la acción está siendo aún ejecutada.

14.6. Nodos Secuenciales ó Nodos Secuencia:

Una secuencia tickea a todos los hijos siempre y cuando cada uno de ellos vaya devolviendo SUCCESS. Si cualquiera de los hijos devuelve FAILURE, la secuencia se aborta.

Existen 3 tipos de nodos secuenciales:

Tipo	**Hijo devuelve FAILURE**	**Hijo devuelve RUNNING**
Sequence	Reinicia	Tickea de nuevo
ReactiveSequence	Reinicia	Reinicia
Star Sequence	Tickea de nuevo	Tickea de nuevo

Con **reiniciar nos referimos a que la secuencia entera se empieza de nuevo por el primer hijo de la lista.**

Con **Tickear de nuevo nos referimos a que la siguiente vez que la secuencia se tickea, se empieza a tickear por el hijo por que el se iba antes. Es decir, los hijos que ya habían devuelto SUCCESS no se tickean de nuevo.**

14.7. Nodos Fallback

El proposito de estos nodos es **probar distintas opciones o estrategias, hasta que encontramos una que funcione**. Su funcionamiento es el siguiente:

Antes de tickear al primer hijo, el estado es *RUNNING*. Si un hijo devuelve FAILURE, *el fallback descarta ese hijo (opción) y tickea al siguiente hijo* con el objetivo de encontrar otra posible hijo(opción/estrategia) que funcione. *Si un hijo devuelve SUCCESS, se para y el fallback devuelve SUCCESS* y todos los hijos se apagan. *En caso de que el último de los hijos devuelve FAILURE y no haya ninguna opción que funcione, el fallback devuelve FAILURE* y todos los hijos se apagan.

Existen 2 tipos de nodos fallback:

Tipo	**Hijo devuelve RUNNING**
Fallback	Tickea de nuevo
ReactiveFallback	Reinicia

Con **reiniciar nos referimos a que la secuencia entera se empieza de nuevo por el primer hijo de la lista.**

Con **Tickear de nuevo nos referimos a que la siguiente vez que la secuencia se tickea, se empieza a tickear por el hijo por que el se iba antes. Es decir, los hijos que ya habían devuelto SUCCESS no se tickean de nuevo.**

14.8. Nodos Decoradores

Un Decorador es un nodo que solo puede tener un hijo.

Es decisión del propio Decorador decidir si, cuando y cuantas veces el hijo debe tickearse.

Existen varios tipos de nodos decoradores:

Inverter	Tickea al hijo una vez y retorna SUCCESS si el hijo falla y viceversa. Si el hijo devuelve RUNNING, el también devuelve RUNNING.
ForceSuccess	Siempre devuelve SUCCESS, excepto si el hijo devuelve RUNNING, que en ese caso también devuelve RUNNING.
ForceFailure	Siempre devuelve FAILURE, excepto si el hijo devuelve RUNNING, que en ese caso también devuelve RUNNING.
Repeat	Tickea al hijo hasta N veces, donde N se pasa como parámetro, siempre y cuando el hijo devuelva SUCCESS. El bucle se interrumpe si el hijo devuelve FAILURE(en ese caso devuelve también FAILURE). Si el hijo devuelve RUNNING, que en ese caso también devuelve RUNNING.
Retry	Tickea al hijo hasta N veces, donde N se pasa como parámetro, siempre y cuando el hijo devuelva FAILURE. El bucle se interrumpe si el hijo devuelve SUCCESS (en ese caso devuelve también SUCCESS). Si el hijo devuelve RUNNING, que en ese caso también devuelve RUNNING.

15. Práctica 2: Visual Behaviour

La segunda práctica planteada consiste en emplear los conceptos de DeepLearning, OpenCV, Behaviour Trees y Controlador Proporcional-Derivativo-Integrador (PID Controller) con el objetivo de seguir a un humano y a una pelota de un color predeterminado. La presente práctica está dividida en las siguientes funcionalidades:

- **Funcionalidad 1:** Seguir a una pelota de un color específico[14] empleando filtrado de color con OpenCV.
- **Funcionalidad 2:** Seguir a un humano empleado BoundingBoxes de la librería de **darket_ros**.
- **Funcionalidad 3:** Combinación de las dos funcionalidades anteriores. El kobuki seguirá a un humano hasta que este le lance una pelota del color filtrado, que es cuando el kobuki dejaŕa de seguir al humano y comenzaŕa a seguir a la pelota.

A continuación se facilita el link al repositorio público de la práctica realizada por el grupo **TayRos**. **Nota que dicha práctica está bajo licencia**:

- `https://github.com/Docencia-fmrico/visual-behavior-tayros`

[14]Se recomienda emplear el color rosa ya que es el color más fácil de filtrar.

16. Mapeo y Navegación en ROS

Dentro de la carpeta del curso, se encuentra el paquete **navigation** que contiene todo lo relativo a la navegación y mapeo en ROS 1.

Este paquete incluye varios launchers de interés, pero nos centraremos en 3 de ellos:

- **localization**
- **slam_mapping**
- **navigation**

16.1. localization.launch

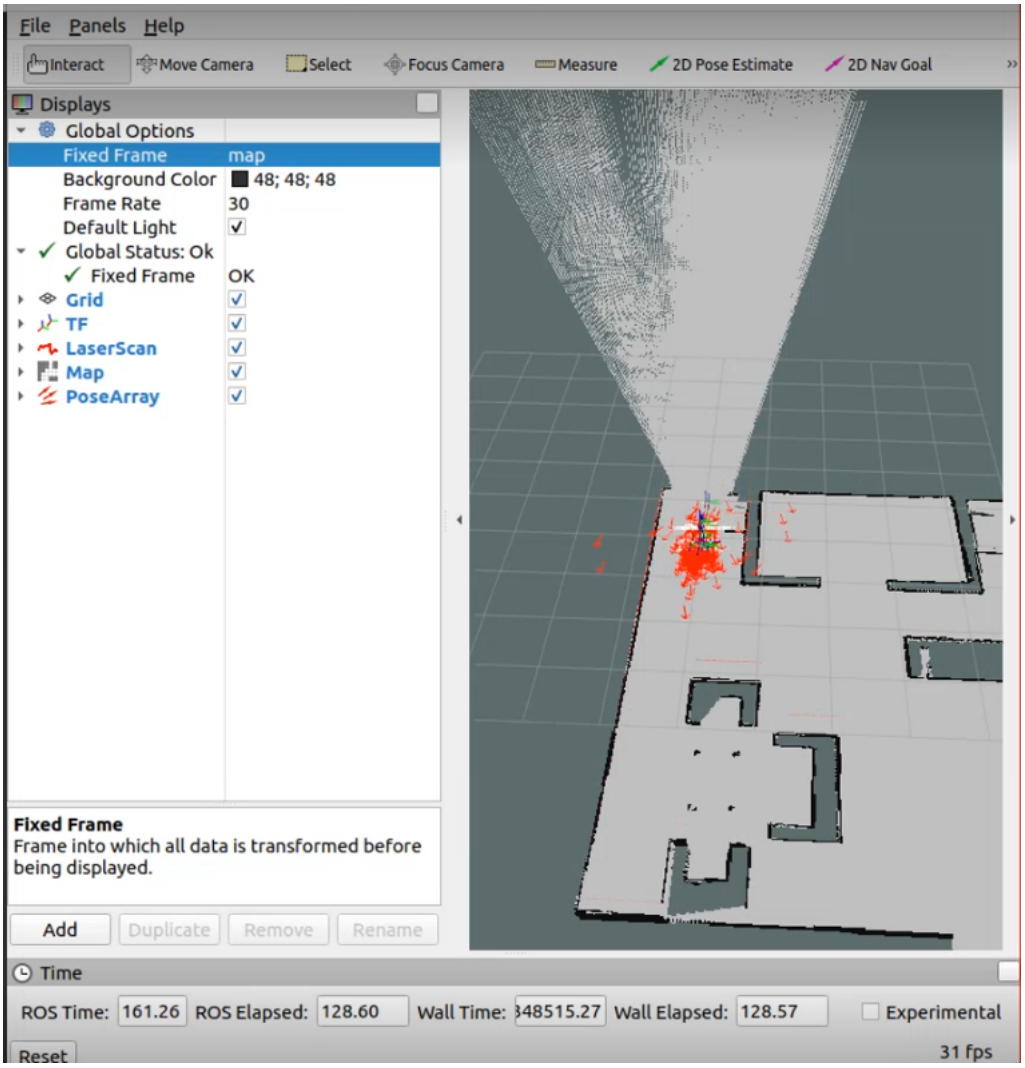
Este launcher implementa el **Monte Carlo Localization Approach o AMCL**, un sistema de localización probabilístico para el movimiento de robots en 2D.

La localización probabilística consiste en el proceso de **determinar la probabilidad de que el robot se encuentre en una determinada pose dada una historia de las lecturas tomadas por los sensores del robot y de una serie de acciones ejecutadas por él mismo**. *A cada posible pose del robot se le asocia una probabilidad* que refleja la verosimilitud de que sea la pose actual. *Las estimaciones se actualizan con la incorporación de nuevas observaciones y nuevos movimientos ejecutados por el robot* . **Esto permite al robot poder localizarse dentro del ambiente aun sin conocer su posición inicial**, *toma en cuenta los errores de odometría* y además permite la representación de situaciones ambiguas que se resuelven posteriormente, l**a eficiencia de este método depende del tamaño del mapa en el que se realiza la localización del agente autónomo.**

El launcher localization permite mostrar un mapa en rviz. Por defecto, en el launcher viene incluido que muestre el mapa del simulador, pero puede cambiarse por otro mapa.

Si activamos en el rviz la opcion de **PoseArray**, cambiamos el fixed frame a **map**[15] y activamos el keyop, veremos como a la vez que el robot se va moviendo se van mostrando unas flechas rojas (el pose array) que acompañan el robot en su trayectoria:

[15]Nota que el frame map indica el punto de origen del mapa y es completamente estático.



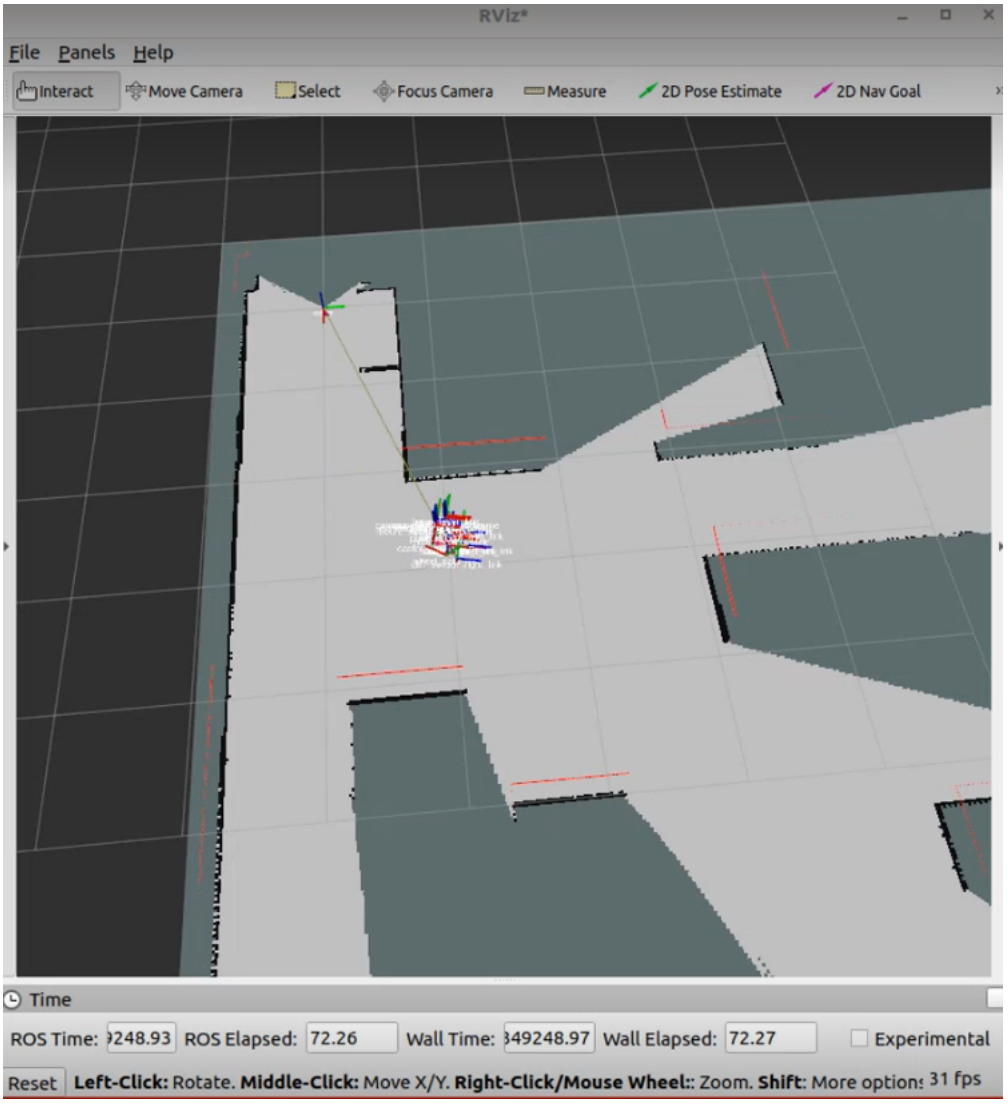
Con la flecha verde del rviz **2D Pose Estimate** podemos mover el robot a cualquier posición del mapa que clickeemos en el rviz.

16.2. slam_mapping.launch

Este launcher permite poner en marcha el **mapeo** del entorno en el que se encuentra el robot.

NOTA IMPORTANTE: En el launcher de slam_mapping se recomienda bajar los parametros de maxUrange y maxRange a 20 para que el mapeo sea más preciso y efectivo. Si estos parámetros se encuentran en valores más elevados pueden causar problemas a la hora de crear el mapa.

Una vez ponemos en marcha el launcher, podemos arrancar tambien el keyop e ir moviendo el robot por el entorno (ya sea real o simulado). Si tenemos en el rviz activado el mapa veremos como se va dando forma al mapa, marcando en blanco el espacio que tiene mapeado y por el que puede pasar, y en negro los bordes, paredes y otros obstáculos que se encuentra.



Ten en cuenta que si estás mapeando, por ejemplo, el laboratorio, si tu estás moviéndote detrás del robot no te detectará como obstáculo ni te pondrá en el mapa como tal.

Una vez se ha terminado de mapear todo, podemos guardar el mapa ejecutando el comando: **rosrun map_server map_saver**. El mapa se guardara en el directorio del espacio de trabajo en dos formatos: .pgm y .yaml[16]

16.3. navigation.launch

Una vez ya tenemos un mapa y queremos navegar por él, lanzamos **navigation.launch**, y, ayudándonos de la opción del rviz de **2D Nav Goal** (aparece como flecha rosa en la barra horizantal superior del rviz), podemos hacer que el robot se mueva por el mapa sin chocarse. También podemos añadir en el rviz, "by topic", la opción **path**, que permite ver el camino que va a seguir el robot cuando se le envía una posición a la que debe ir.

[16]Es posible editar el mapa en gimpz para arreglar imperfecciones.

```cpp
#include "actionlib/client/simple_action_client.h"
#include "move_base_msgs/MoveBaseAction.h"
#include "geometry_msgs/PoseStamped.h"
#include <string>

namespace navigation
{
class Navigator
{
  public:
    Navigator(ros::NodeHandle& nh) : nh_(nh), action_client_("/move_base", false), goal_sended_(false)
    {
      wp_sub_ = nh_.subscribe("/navigate_to", 1, &Navigator::navigateCallback, this);
    }

    void navigateCallback(geometry_msgs::PoseStamped goal_pose_)
    {
      ROS_INFO("[navigate_to_wp] Commanding to (%f %f)", goal_pose_.pose.position.x, goal_pose_.pose.position.y);
      move_base_msgs::MoveBaseGoal goal;
      goal.target_pose = goal_pose_;
      goal.target_pose.header.frame_id = "map";
      goal.target_pose.header.stamp = ros::Time::now();
      action_client_.sendGoal(goal);
      goal_sended_ = true;
    }

    void step()
    {
      if (goal_sended_)
      {
        bool finished_before_timeout = action_client_.waitForResult(ros::Duration(0.5));
        actionlib::SimpleClientGoalState state = action_client_.getState();
        if (finished_before_timeout)
        {
          actionlib::SimpleClientGoalState state = action_client_.getState();
          if (state == actionlib::SimpleClientGoalState::SUCCEEDED)
            ROS_INFO("[navigate_to_wp] Goal Reached!");
          else
            ROS_INFO("[navigate_to_wp] Something bad happened!");
          goal_sended_ = false;
        }
      }
    }

  private:
    ros::NodeHandle nh_;
    ros::Subscriber wp_sub_;
    bool goal_sended_;
    actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> action_client_;

};
}  // namespace bica_dialog
```

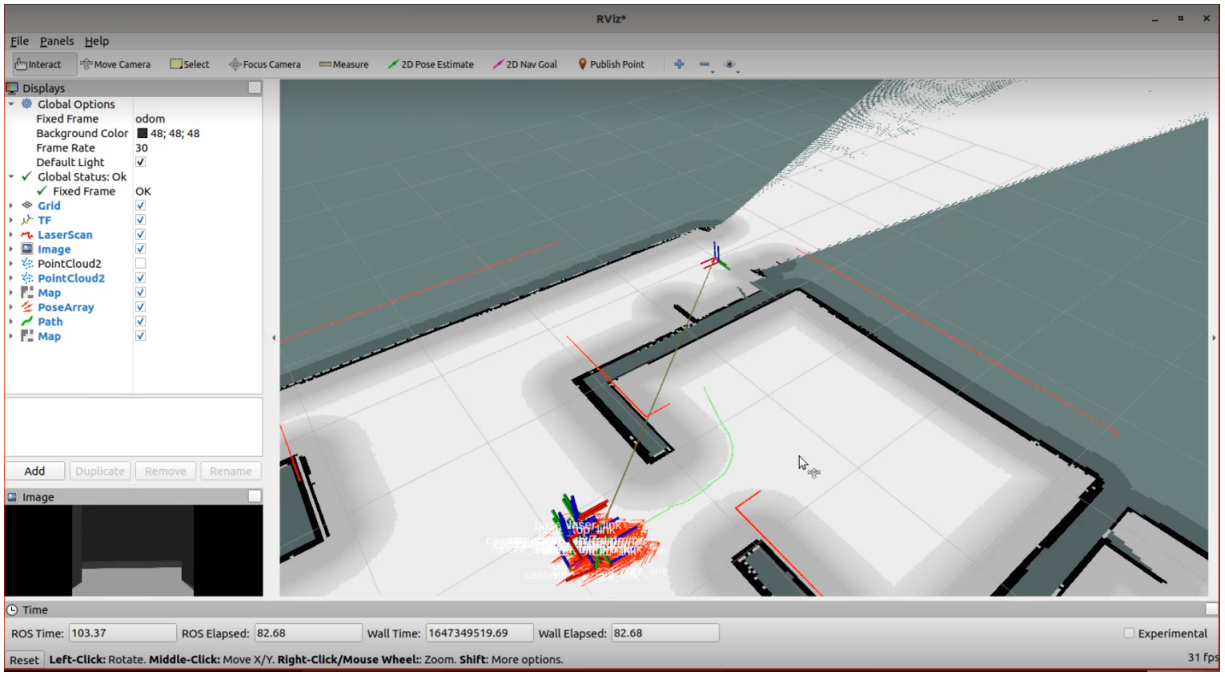
16.4. costmap

Otra opción a añadir por topic en el rviz es el /costmap de /global_costmap. Esto, por así decirlo, representa el coste (cost) o dificultad del robot para atravesar diferentes zonas del mapa.

Si se activa está opción en el rviz veremos que hay un sombreado a los bordes del mapa, que representan una zona por la que el robot no puede moverse. Por tanto, si hay una zona estrecha en el mapa, el costmap .^{es}trecha.^{a}un más esa zona, y puede significar que el robot no pueda atravesar dicha zona.

La zona de inflado de los bordes puede cambiarse modificando el valor del parámetro **robot_radius** en el **costmap_common_params.yaml**. En este mismo fichero pueden modificarse otros parámetros importantes.

En la siguiente imagen se muestra en el rviz, como el robot no es capaz de calcular su path hasta el punto que se le ha indicado debido al costmap (tiene un camino muy estrecho para entrar en esa habitación):

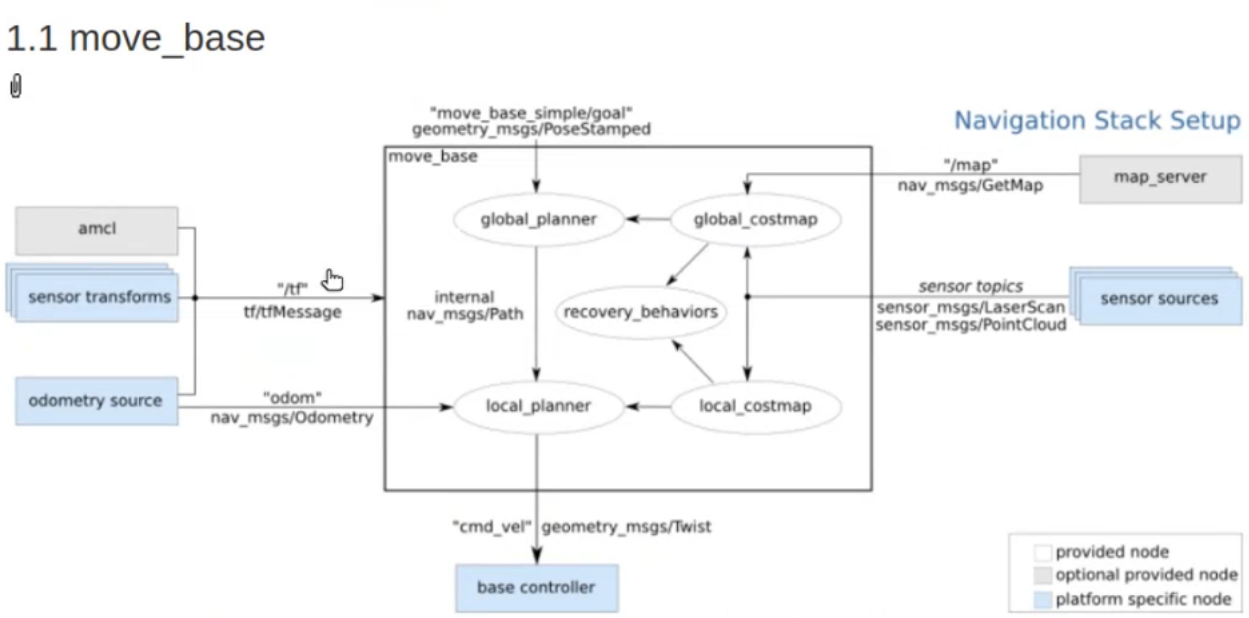


16.5. Funcionalidad de los nodos lanzamos en el mapeo y navegación

A continuación se detallan la funcionalidad de los distintos nodos vistos que se emplean en la navegación y mapeo:

- nodo **map_server**: se encarga de coger el mapa y publicarlo en el topic /map.
- **amcl**: crea las distintas flechitas rojas (posiciones), calculando su probabilidad y la posición. **El resultado es una TF de odom a map**
- **global_planner**: Coge el mapa, la localización del robot, y la posición indicada a la que tiene que ir el robot, y crea la ruta desde la posición del robot al destino, publicandola en un topic (ver imagen de abajo). Por encima le llega la orden del goal, a través de una acción.
- **global_costmap**: Le llega el mapa pero ensanchando las paredes. Este mapa se envía al global_planner. NO tiene en cuenta los objetos dinámicos.
- **local_planner**: Esto ya es un proceso local. Intenta seguir la ruta marcada. Genera las velocidad necesarias para que el robot siga lo mejor posible esa ruta. El resultado es una velocidad que se pasa a base controller.
- **local_costmap**: Es un pequeño mapa local al robot, que SI tiene en cuenta los objetos dinámicos. Este mapa, visto desde el rviz, es un pequeño cuadrado que avanza junto al robot.

Esto, gráficamente, puede representarse de la siguiente forma:



Adapted from ROS Wiki

17. Diálogo

Existen muchos paquetes en ROS para gestionar el text to speech y el dialogo de un robot. Uno de los más usados es festival, que está hecho en python. Puede leer un string que se le pasa como parámetro.

Nosotros nos centraremos en **DialogFlow**, que permite mantener conversaciones con un robot.

Usaremos el repositorio público de GitHub de **gb_dialog**. En el propio repo de GitHub se detallan todos los pasos para su configuración e instalación.

17.1. DialogFlow: configuración y uso

En la página de **DialogFlow** podemos crear distintos Agentes para poder empezar a trabajar con el diálogo.

17.1.1. Intents

Una Intención consiste en intentar entender que es lo que quiere la persona con la que está interactuando el robot. Una intención está compuesta por frases de detección (frases que puede decir la persona y que se corresponden con esa intención) y con respuestas (que puede decir el robot. Si hay más de una respuesta, se elige una de forma aleatoria).

DialogFlow permite detectar frases de entrada parecidas a las especificadas (no tienen por que ser exáctamente la misma frase).

Por defecto, DialogFlow incorpora una intención de bienvenida (con varias frases de bienvenida/saludo) y una intención de **Default Fallback Intent**. Esta intención se corresponde con lo siguiente: cuando la persona le diga una frase al robot que no se reconoce con ninguna intención creada, DialogFlow entrará en esa intención y responderá con alguna de las frases de respuesta configuradas por defecto (que vienen a decir que no ha entendido lo que le ha dicho la persona).

17.1.2. Contexts

En las intenciones hay una opción llamada **Contexts**. Esto permite que, cuando el robot entre a una Intención concreta después de que un usuario le haya dicho una frase, podemos configurar para que el robot entre en un contexto de habla.

Por ejemplo, imagina que la persona le dice al robot que le cuente un chiste. Podemos configurar un intento que devuelva un chiste, y que a su vez al entrar en esta intención el robot entre en un contexto de: estamos de risas. Así, podríamos configurar otros intentos que solo se activen cuando estemos en ese contexto.

17.1.3. Entities

DialogFlow contiene lo que se conoce como **entidades**. Esto son varias listas referentes a algo concreto. DialogFlow contiene por defecto, algunas de las siguientes listas[17]:

- **@sys.geo-country**: Lista de países del mundo
- **@sys.geo-capital**: Lista de capitales del mundo
- **@sys.color**: Lista de colores
- **@sys.unit-speed**: Lista de velocidades con su medida
- ...

Las listas permiten crear frases de entrada de los intentos que se correspondan con colores, capitales, países, nombres de personas, etc..., permitiendo la creación de conversaciones más dinámicas. También se permite que una cierta entity tenga que ser detectada obligatoriamente en una frase de entrada (En caso contrario el robot la pedirá despues con el mensaje configurado para pedirla, en el apartado de Action and parameters).

Para ello, en las frases de entrada, o bien un nombre, país, etc es detectado automáticamente por DialogFlow como lista registrada y lo resalta (pudiendo acceder a ese dato como si una variable fuese), o bien podemos registrar una o varias palabras como una entidad concreta, seleccionándolas y poniendo @la-entity-que-queremos.

Podemos crear entidades custom que no estén registradas en el apartado de Entities de DialogFlow

17.2. Como usar gb_dialog:

Para usar gb_dialog debemos hacer lo siguiente[18].

Por un lado, lanzamos desde una terminal el launcher de gb_dialog: **roslaunch gb_dialog gb_dialog_services_soundplay.launch**

Por otro lado, lanzamos desde otra terminal: **rosrun gb_dialog example_df_node**.

Una vez hecho esto estará escuchando, esperando a que se le diga alguna frase correspondiente a alguno de los Intentos registrados en el programa.

17.3. Como registrar nuevos Intents:

Para registrar un nuevo Intent a nuestro programa hay que hacerlo con la sentencia this->registerCallBack(), tal y como se muestra en la siguiente imagen:

[17]Lista con todas las entities del sistema: https://cloud.google.com/dialogflow/es/docs/reference/system-entitieslimits

[18]Nota que hay que tener todo configurado antes para que funcione

```
39  // MODIFIED BY 2022 TayRos. Rey Juan Carlos University, Software Robotics Eng.
40  #include "taymsgs/person_data.h"
41  #include <gb_dialog/DialogInterface.h>
42  #include <dialog_cbs/dialog_cbs.h>
43  #include <std_msgs/String.h>
44  #include <string>
45  #include "std_msgs/Int32.h"
46
47  namespace ph = std::placeholders;
48
49  namespace gb_dialog
50  {
51  DialogManager::DialogManager()
52  : nh_()
53  {
54    this->registerCallback(std::bind(&DialogManager::noIntentCB, this, ph::_1));
55    this->registerCallback(
56        std::bind(&DialogManager::carReachedCB, this, ph::_1),
57        "carReached");
58    this->registerCallback(
59        std::bind(&DialogManager::askForNameCB, this, ph::_1),
60        "askForName");
61    this->registerCallback(
62        std::bind(&DialogManager::pointBagDialogCB, this, ph::_1),
63        "pointBagDialog");
64    this->registerCallback(
65        std::bind(&DialogManager::startNavCB, this, ph::_1),
66        "readyToMove");
```

Como primer parámetro debe ir el nombre del callback que va a tener ese intent, y como último parametro el nombre del callback de DialogFlow.

Una vez registrado el intent, podemos configurar el callback de la manera que queramos. Si el callback recibe como parámetro **dialogflow_ros_msgs::DialogflowResult result**, que es el tipo de mensaje del resultado del intento, podemos hacer que diga dicho resultado de la frase configurada como respuesta de DialogFlow con la sentencia: **speak(result.fulfillment_text);**

17.4. Acceder a los parámetros del Intento desde el código:

Hay varias formas de acceder a los parámetros/variables del Intento.

Una sencilla forma es configurar en DialogFlow que la respuesta sea, simplemente, la variable de la entity que queremos. Así, en **result.fulfillment_text** tendremos el parámetro deseado en forma de string.

Otra forma es recorrer el tipo de mensaje de result y acceder al parámetro, como se muestra en la siguiente imagen:

```
73
74      for (const auto & param : result.parameters) {
75        std::cerr << "Param: " << param << std::endl;
76        for (const auto & value : param.value) {
77          std::cerr << "\t" << value << std::endl;
78        }
79      }
```

18. Práctica Final: RoboCup Challenges

La práctica final consiste en realizar al menos 2 de las 3 pruebas propuestas en la RoboCup Spain 2022. Estas eran: **Carry My Luggage**, **Find My Mates** y **Recepcionist**.[19]. Estas pruebas contenián todo el temario dado en este libro:

- Filtros de Color
- Detección de Personas y objetos
- TFs
- Mapeo
- Navegación por zonas mapeadas y zonas sin mapear
- Diálogo
- Behaviour Trees
- Máquinas de Estados

A continuación se facilita el link al repositorio público de la práctica realizada por el grupo **TayRos**. **Nota que dicha práctica está bajo licencia**:

- `https://github.com/Docencia-fmrico/robocup-home-education-tayros`

[19]Encuentra toda la información relevante a que consisten en el siguiente enlance RoboCup Spain

19. ROSBag

ROSBag es una serie de herramientas que permiten guardar la información de los topics de un robot para poder reproducirlas en otro momento.

Entre sus usos más habituales, se encuentran guardar el movimiento y las acciones del robot para poder simularlos de manera idéntica en momentos posteriores.

19.1. Comandos del ROSBag

Los comandos más importantes relacionados con el ROSBag son los siguientes:

- **rosbag record**: permite guardar topics, los cuales se les pasan en orden. Con la opcion **-a** se guardan todos los topics. Para ejar de grabarlos, hacer **CTRL+C**.
- **rosbag play**: permite reproducir un rosbag previamente guardado con **record**. A la hora de reproducirlo, se puede avanzar paso por paso con la tecla **S** y pausarlo con la tecla **espacio**.
- **rosbag compress**: comprime los archivos utilizados.
- **rosbag decompress**: descomprime los archivos previamente comprimidos con compress.

Ejemplo: Lanzamos el simulador y hacemos:

rosbag record /mobile_base/commands/velocity

Podemos hacer **ls -lh** en nuestro workspace para ver por cuanto tamaño va el rosbag que estamos guardando. Movemos el kobuki con el keyop de modo que haga una trayectoria que queremos reproducir en otro momento. Guardamos el record (CTRL+C). Si ponemos el kobuki en la posición inicial de antes y hacemos rosbag play <nombreBag>, se reproducirá el bag y el robot volverá a hacer la trayectoria de antes.

20. ROS WTF

roswtf es una herramienta que empieza a analizar tu configuración, grafo de computación y otros diagnósticos para intentar averiguar que está mal:

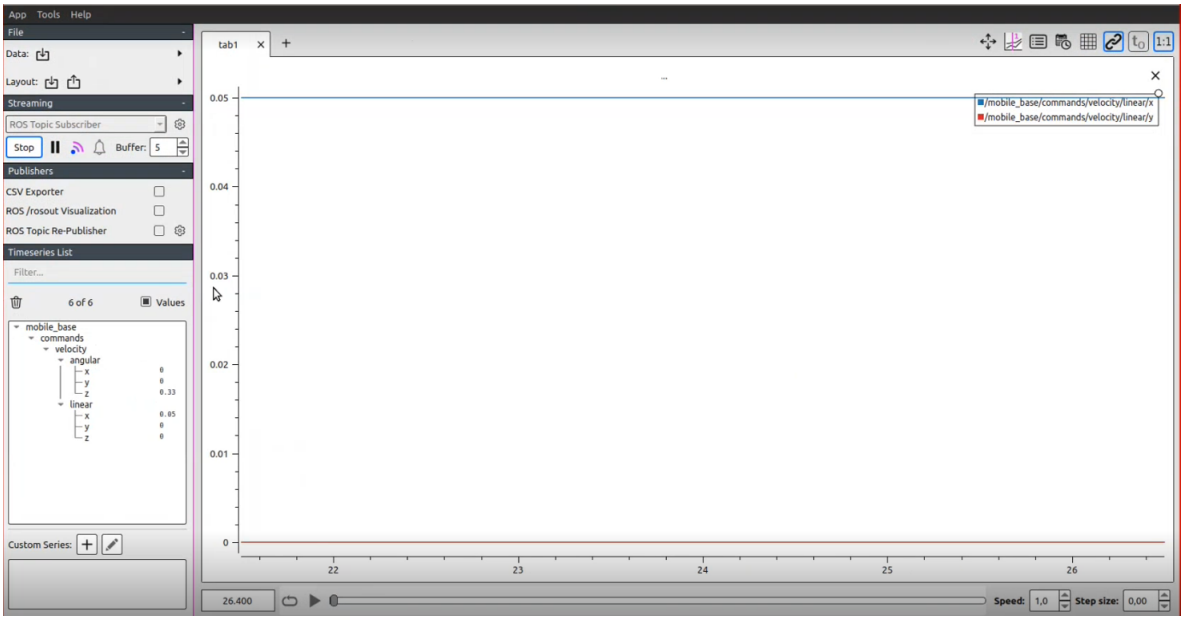
```
sauln@sauln:~/arqSoftwareRobots/ros1/catkin_ws$ roswtf
Loaded plugin tf.tfwtf
Loaded plugin openni2_launch.wtf_plugin
No package or stack in the current directory
================================================================================
Static checks summary:

No errors or warnings
================================================================================
Beginning tests of your ROS graph. These may take a while...
analyzing graph...
... done analyzing graph
running graph rules...
... done running graph rules
running tf checks, this will take a second...
... tf checks complete
Traceback (most recent call last):
  File "/opt/ros/noetic/lib/python3/dist-packages/roswtf/__init__.py", line 224,
 in _roswtf_main
    p(ctx)
  File "/opt/ros/noetic/lib/python3/dist-packages/openni2_launch/wtf_plugin.py",
 line 114, in roswtf_plugin_online
    error_rule(r, r[0](ctx), ctx)
  File "/opt/ros/noetic/lib/python3/dist-packages/openni2_launch/wtf_plugin.py",
 line 85, in sensor_notfound
    devices = _device_notfound_subproc(
  File "/opt/ros/noetic/lib/python3/dist-packages/openni2_launch/wtf_plugin.py",
 line 58, in _device_notfound_subproc
    for i in df.split('\n'):
TypeError: a bytes-like object is required, not 'str'
a bytes-like object is required, not 'str'

Aborting checks, partial results summary:

Found 2 warning(s).
Warnings are things that may be just fine, but are sometimes at fault

WARNING The following node subscriptions are unconnected:
 * /rosout:
   * /rosout

WARNING No tf messages

sauln@sauln:~/arqSoftwareRobots/ros1/catkin_ws$
```

21. PlotJuggler ROS

plotjuggler es una herramienta gráfica de ROS que permite subscribirse a ciertos topics y visualizar de forma gráfica los valores y parámetros de los tipos de mensaje de esos topics. Si tenemos instalado en nuestro ordenador ros-noetic-plotjuggler-ros, podemos ejecutar el comando:

rosrun plotjuggler plotjuggler

En la parte izquierda, en la sección de Streaming ponemos **ROS Topic Subscriber** y seleccionamos un topic al que queremos subscribirnos con plotjuggler. Uno muy visual sería poner el /mobile_base/commands/velocity, de manera que podemos visualizar gráficamente las velocidades lineares y angulares de x, y, z.



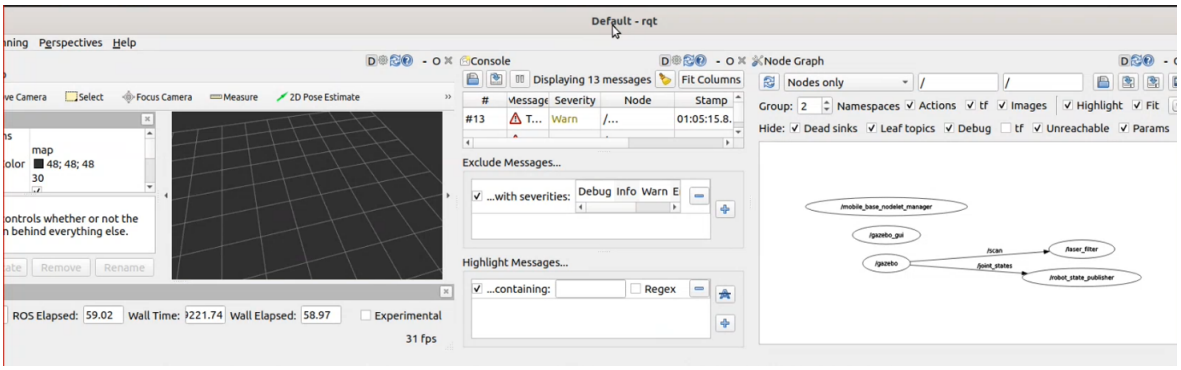
También puede ser interesante analizar los valores medidos por los sensores de los robots (ruidos de las ondas), etc...

22. rqt

rqt es una serie de herramientas gráficas de ROS. Entre ellas se encuentran **rqt_console**, **rqt_graph**, etc...[20]

Uno muy importante y útil es **rqt_gui**, que se lanza con el comando **rosrun rqt_gui rqt_gui [–clear-config]** o simplemente **rqt**. Al lanzarlo sale una ventana en blanco y una ventana que permite añadir distintos plugins por separado, ya sea un rqt_console, un image_view de un topic concreto, el rviz, etc.

Incluso podemos hacer una configuración customizada muy compleja y guardarla en la ventana de perspectivas, exportar, y guardar el archivo. Así, al cargarlo de nuevo al abrir rqt_gui podemos tener la configuración que hayamos creado.



Se pueden crear con herramientas como **qtcreator**. Estos rqt están compuestos por un archivo .ui que está escrito en xml y especifica todo lo que tiene la interfaz.

[20] hacer rosrun rqt_+TAB para ver todos.

23. Últimos conceptos de ROS

Hasta ahora hemos trabajado con los robots de la siguiente manera: Tenemos nuestro ordenador con los programas y la CPU y lo conectamos al robot por medio de USBs.

Lo normal es que el Robot tenga su propia CPU y tu te conectes a él, usando, entre otras cosas, variables de entorno.

23.1. Configuración de variables

Si nosotros ponemos en nuestro **.bashrc** las variables de entorno:

export ROS_IP=<IP1> export ROS_MASTER_URI=http://<IP2>:11311

y lanzamos un roscore, lo que estamos haciendo es lanzar el roscore a través de un puerto determinado, que se conecta con la IP de otro ordenador y podemos reconocerlo por su nombre (mirar el archivo de configuración /**etc**/**host**).

24. Introducción a ROS 2. De ROS a ROS 2

24.1. Introducción

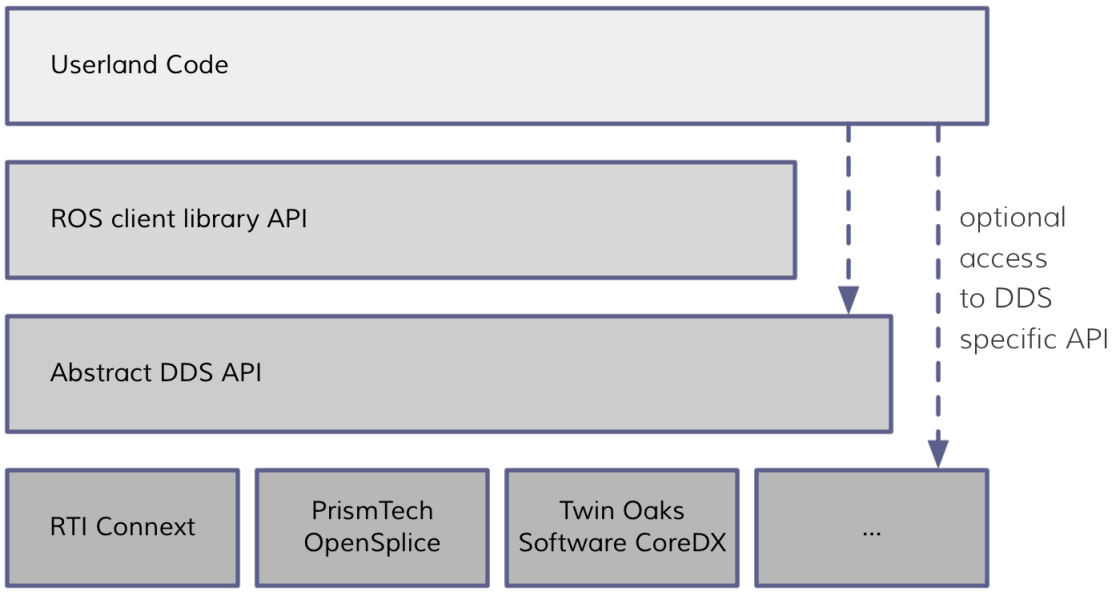
Hasta ahora, hemos estado trabajando con ROS(1), que tiene las siguientes **limitaciones**:

- Está **centralizado en un Master**: si se cae el master, se cae todo.
- Tiene su **propia implementación de red**: **sin multicast real ni calidad de servicios**(esto es, que queremos hacer cuando perdemos un paquete, decidir si hay reenvío o no, etc...)
- **No hay soporte para multi-robot**
- **No está pensado para sistemas de tiempo real**
- **No es multiplataforma**
- **Muchos componentes importantes son parches**(por ejemplo, las acciones de ROS no estaban en el diseño original)
- **C++ y Python son implementaciones independientes**
- **No existe seguridad**

24.2. Diseño de ROS 2

Lo primero a destacar en ROS 2 es que **no existe master**.

Por otro lado, ROS 2 tiene una arquitectura en capas:



Adapted from ROS 2 Design

En ROS 2 la capa de red se basa en DDS (Data Distribution System), que tiene comunicaciones por publicación/subscripción para sistemas en tiempo real. Hay distintos DDS: rti, Cyclone, etc...

Para usar en tu máquina uno u otro DDS, simplemente hay que cambiar una variable de entorno en el .bashrc:

- **OpenSlice:** export RMW_IMPLEMENTATION=rmw_openslice_cpp
- **RTI Connext:** export RMW_IMPLEMENTATION=rmw_connext_cpp
- **Eclipse Cyclone:** export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp

Por otro lado está la RCL (ROS Client Layer). **rcl** es una implementación core donde se encuentra toda la funcionalidad básica de ROS y está escrita en C. rcl está escrito en C por que casi todos los lenguajes de programación pueden usar librerías en C.

Encima de rcl se construyen las librerías cliente de ROS (implementaciones para distintos lenguajes de programación):

- **rclcpp**: implementación para C++
- **rclpy**: implementación para Python
- **rclc**: implementación para C
- ...

24.3. Empezar a trabajar con ROS 2

Para empezar a trabajar con ROS 2:

1. Cambiar en .bashrc "noetic" por "foxy". Una vez hecho, abrir una nueva terminal.
2. Crear un directorio que va a ser nuestro workspace para ROS 2. Llamarlo **colcon_ws**.
3. Compilar el workspace con **colcon build**
4. Activar el workspace con **source install/setup.bash**
5. Clonamos en nuestro src los proyectos que queramos usar...
6. Compilamos desde el workspace con **colcon build –symlink-install**[21]

Una vez seguidos todos estos pasos, si hacemos **ls** en el workspace veremos que tenemos los siguientes directorios:

- src

[21]SIEMPRE QUE COMPILEMOS NUESTRO WORKSPACE VA A SER CON ESTA FLAG: cuando compilamos, los resultados de la compilación, además de copiarlos al directorio install, crea enlaces simbólicos a su lugar de origen.

- build
- install
- log

A diferencia de ROS(1), ya no hay directorio devel.

En el directorio **log** se guardarán los logs: cuando se compila o cuando se pasa un test toda la información relativa a ellos se guarda en este directorio.

En el directorio **build** estan los ficheros intermedios de compilación.

Por último, en **install** está realmente todos los ficheros que existen. Si algo no está en install no existe como tal.

24.4. Comandos en ROS 2 y conectividad con otras máquinas del mismo dominio

En ROS 2 todos los comandos empiezan por **ros2 <verbo>**, donde <verbo>es: topic, run, etc...

Si nosotros hacemos un ros2 topic pub -r 10 <topic><tipoMsg><data>, DDS manda los topics y nodos disponibles al resto de la red que esten **dentro del mismo dominio**.

Si en nuestor .bashrc ponemos la variable de entorno:

export ROS_DOMAIN_ID=<DOMINIO>

lo que haremos será cambiar nuestro dominio y evitar que podamos ver información de otros ordenadores que antes si estaban en el mismo dominio.

24.5. Primeros pasos en ROS 2

Ahora en ROS 2, los **Nodos ya NO son procesos, son objectos**. Además, ya no necesitamos usar **NodeHandle**. Además, en el código de C++, ya no haremos ros::<algo>, si no que ahora será **rclcpp::<algo>**.

24.6. Interfaces, publicación y subscripción

Ahora en ROS 2 ya no va a ver mensajes, si no que van a ser interfaces:

- **ros2 interface list [-m]**: Muestra todos los interfaces que hay. Se muestran tanto mensajes, como servicios y acciones. Con la flag -m solo se muestran los mensajes.
- **ros2 interface show <interface>**: Muestra la información de una interface.

24.7. Nodos en ROS 2

24.7.1. Nodo Simple

A continuación se muestra un ejemplo de un nodo simple en ROS2:

```
14
15  #include "rclcpp/rclcpp.hpp"
16
17  int main(int argc, char * argv[])
18  {
19    rclcpp::init(argc, argv);
20
21    auto node = rclcpp::Node::make_shared("simple_node");
22
23    rclcpp::spin(node);
24
25    rclcpp::shutdown();
26
27    return 0;
28  }
```

Como ahora los nodos son objetos, **existe la posibilidad de que en un mismo proceso tengamos varios nodos**.

Fíjate en la línea 21: estamos creando con un nodo (con auto para que pille el tipo automáticamente) que es un rclcpp::Node::make_shared(<NombreNodo>).

El make_shared es realmente como se declara un tipo de puntero inteligente (Shared). La forma de declarar realmente un puntero inteligente es:

std::shared_ptr<rclcpp::Node>

Los punteros inteligentes, además de tener un puntero a la dirección de memoria tiene un contador de cuantos objetos están apuntando a la misma zona de memoria.

Lo que hace pues make_shared es realmente reservarte espacio de memoria para el tipo de datos que se le pasa. Aquí están las 4 formas de declarar un nodo, ordenados de sentencia más compleja a más compacta:

```
21  std::shared_ptr<rclcpp::Node> node = std::shared_ptr<rclcpp::Node>(new rclcpp::Node("simple_node"));
22  std::shared_ptr<rclcpp::Node> node = std::make_shared<rclcpp::Node>("simple_node");
23  auto node = std::make_shared<rclcpp::Node>("simple_node");
24  auto node = rclcpp::Node::make_shared("simple_node");
```

Ahora, en el **rclcpp::spin(<NombreNodo>)** se le debe pasar el nombre del nodo al que queremos hacerle spin. Además veremos más adelante que el **spinOnce** ahora es **spin_some**.

24.7.2. Nodo Iterativo Publicador

```
45  int main(int argc, char * argv[])
46  {
47    rclcpp::init(argc, argv);
48
49    auto node = std::make_shared<MyNodePublisher>();
50
51    rclcpp::Rate loop_rate(500ms);
52    while (rclcpp::ok()) {
53      node->doWork();
54
55      rclcpp::spin_some(node);
56      loop_rate.sleep();
57    }
58
59    rclcpp::shutdown();
60
61    return 0;
62  }
```

Lo que hacemos en este nodo es declararnos un nodo de MyNodePublisher y llamamos todo el rato a node->doWork();

Realmente, lo único que está haciendo es publicar todo el rato un mensaje en un topic. Nota además que ahora los includes están en un .hpp y los tipos de mensaje están todos en minúscula. También hay una diferencia en el ROS_INFO, que se llama ahora RCLCPP_INFO al que se le mete un get_logger().

```cpp
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;

class MyNodePublisher : public rclcpp::Node
{
public:
  MyNodePublisher()
  : Node("componsable_node_pub"),
    counter(0)
  {
    pub_ = create_publisher<std_msgs::msg::String>("chatter", 10);
  }

  void doWork()
  {
    std_msgs::msg::String message;
    message.data = "Hello, world! " + std::to_string(counter++);

    RCLCPP_INFO(get_logger(), "Publishing [%s]", message.data.c_str());

    pub_->publish(message);
  }

private:
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr pub_;
  int counter;
};

```

24.7.3. Nodo Iterativo Subscriptor

```
20  class MyNodeSubscriber : public rclcpp::Node
21  {
22  public:
23    MyNodeSubscriber(const std::string & name)
24    : Node(name)
25    {
26      sub_ = create_subscription<std_msgs::msg::String>(
27        "chatter", 10, std::bind(&MyNodeSubscriber::callback, this, _1));
28    }
29
30  private:
31    void callback(std_msgs::msg::String::UniquePtr msg)
32    {
33      RCLCPP_INFO(this->get_logger(), "I heard: [%s] in %s",
34        msg->data.c_str(), get_name());
35
36      temp_ = std::move(msg);
37    }
38
39  private:
40    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr sub_;
41
42    std_msgs::msg::String::UniquePtr temp_;
43  };
44
45  int main(int argc, char * argv[])
46  {
47    rclcpp::init(argc, argv);
48
49    auto node_A = std::make_shared<MyNodeSubscriber>("node_A");
50    auto node_B = std::make_shared<MyNodeSubscriber>("node_B");
51
52    rclcpp::executors::SingleThreadedExecutor executor;
52    rclcpp::executors::SingleThreadedExecutor executor;
53    executor.add_node(node_A);
54    executor.add_node(node_B);
55
56    executor.spin();
```

En las líneas 52 a 56 lo que estamos haciendo es crear un rclcpp::executors::SingleThreadedExecutor executor que se le permite añadir nodos con la sentencia **executor.add_node(<NombreNodo>)**, de modo que luego podemos hacer **executor.spin()** para hacer spin a todos los nodos que hallamos añadido al executor a la vez (pero en 1 Thread).

24.8. QoS en ROS 2 (Calidad de Servicios)

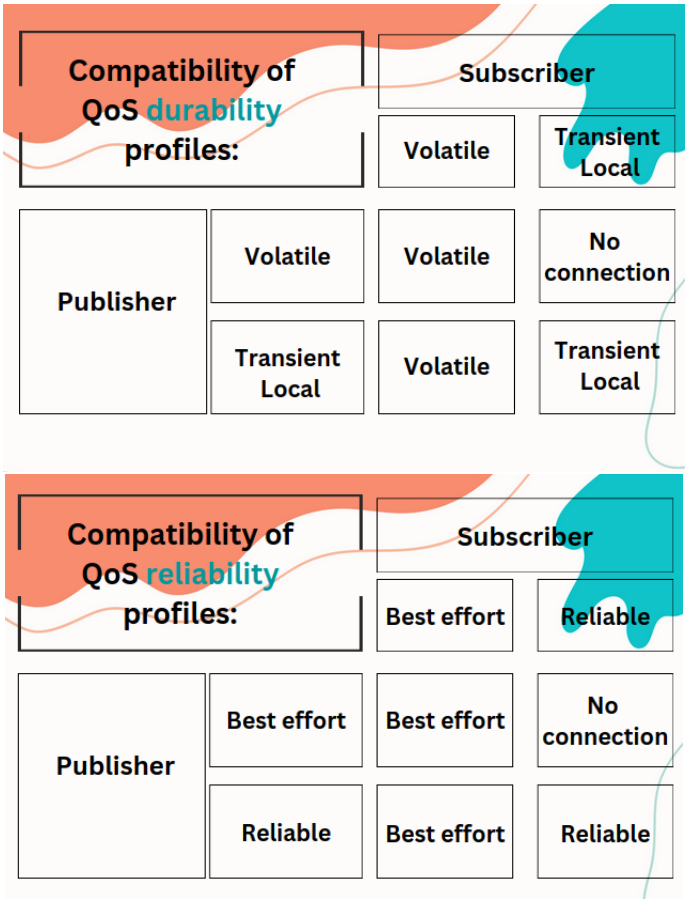
La calidad de servicio de un nodo viene a ser las propiedades que va a tener un nodo a la hora de comunicarse:

- **Fiabilidad**:
  - **Best effort (no fiable)**: cuando un topic tiene que ir rápido y no importa si se pierde un mensaje.
  - **Reliable (fiable)**: cuando es importante que un mensaje queremos que llegue sí o sí.
- **Profundidad**: Tamaño de cola de mensajes.
- **Durabilidad**:
  - **Transient local**: te subscribes a un topic y recibes todos los mensajes que se han publicado hasta ese momento (incluidos aquellos que se han enviado por el topic antes de subscribirse), respetando el tamaño de la cola.
  - **Volatile**: te subscribes a un topic y recibes todos los mensajes a partir de ese momento.
- **Deadline**: Se indica un límite de tiempo: si un mensaje se recibe pasado el deadline, se descarta.
- **Lifespan**: máxima cantidad de tiempo entre que se publica y se recibe un mensaje sin considerarse expirado o perdido.
- **Liveliness**
- **History**
- **Lease duration**

24.8.1. Compatibilidad entre distintos QoS

Puede existir el caso de que un publicador y un subscriptor tengan disinta durabilidad o fiabilidad de la calidad de sus servicios, dando lugar a incompatibilidades.

A continuación se detalla la tabla de correspondencias en estos casos:



Nota que los casos más importantes en esta tabla son 2:

Por un lado, si un publicador es Volatile y un subscriptor es Transient Local **No hay conexión**.

Por otro lado, si un publicador es Best effort y un subscriptor es Reliable **No hay conexión**.

Una forma de acordarse por que pasa esto es pensar que **los publicadores tienen que tener una calidad(durabilidad/fiabilidad) mayor o igual a la de los subscriptores**, con (Transient Local >Volatile y Reliable >Best effort).

24.9. Launchers en ROS 2

En ROS(1) los launchers los hacíamos en xml. Ahora en ROS2 existe la posibilidad de hacer los launchers de 3 formas distintas:

- **XML**: recomendado para launchers sencillos
- **Python**: recomendado para launchers algo más complejos
- **YAML**

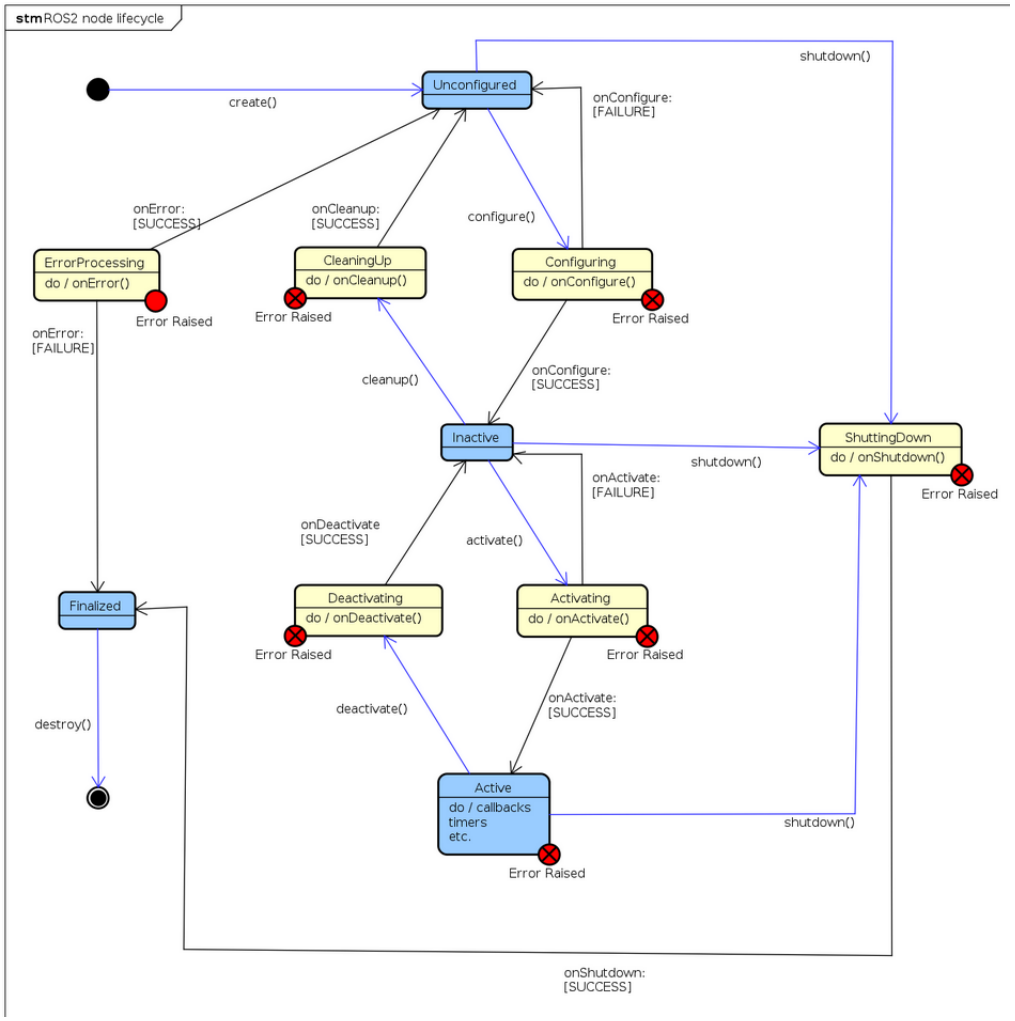
24.10. Parámetros en ROS 2

Ahora **los parámetros pertenecen a un nodo**, de modo que ahora cuando hacemos un .yaml tenemos que especificar a que nodo pertenecen esos parámetros y en el nodo tenemos que declarar los parámetros con **node->declare_parameter(<nombreParametro>)**.

24.11. Lifecycle Nodes

En ROS(1), los nodos se ejecutaban desde el principio. Pero puede dar el caso de que no queramos que esto sea así, si no que un nodo queremos que se ejecute después de otro (por ejemplo si un nodo abre dispositivos).

Los lifecycle nodes implementan una máquina de estados, de modo que los nodos están en todo momento en un estado concreto:



Adapted from ROS 2 Design

Además, otros nodos pueden conocer "desde fuera" en que estado se encuentra otro nodo, de modo que puedan esperar a que un nodo concreto esté en un estado determinado para que se activen.

25. Preguntas de comprensión. Autoevaluación

NOTA: Las preguntas que se detallan a continuación tienen el propósito de autoevaluar el conocimiento adquirido de la lectura de todos los capítulos, así como de de los trabajos y ejercicios realizados que se han mencionado en este libro.

1. ¿Qué es un ROS Core?
2. ¿Qué hace la sentencia ros::init?
3. ¿Qué hace la sentencia ros::spin?
4. ¿Puede usarse la misma blackboard para dos nodos distintos?
5. ¿Con qué comando podemos obtener los nodos subscritos a un determinado topic?
6. ¿Que espectro de color es el más robusto para cambios bruscos de luz?
7. ¿Con qué sentencias podemos manejar la velocidad lineal y angular de un robot omnidireccional?
8. ¿Qué topics usan las transformadas (TFs)?[22]
9. ¿Cuál es el tipo de mensaje principal en OpenCV?
10. ¿Qué significa que el topic de PointCloud2 sea depth_registered?
11. En ROS, decimos que somos:
    - Dextrógiros
    - Levógiros
12. ¿Puede un topic contener dos mensajes de distintos tipos?
13. ¿Que es el número que va dentro de la sentencia ros_rate(N)?
    - Segundos
    - Milisegundos
    - Nanosegundos
    - Frecuencia (Hz)
14. ¿Puede un Nodo Decorador devolver RUNNING?
15. ¿En qué topic se publica el goal en la navegación?
16. ¿Si se crea un publicador llamado pub_something y se le especifica que se publique el mensaje en /**miTopic** , cual es el nombre exacto del topic en donde se publica?

[22]Nota que es una pregunta trampa, no usan topics, si no que usan el Listener, buffer y Broadcaster. Ver apartado de TFs del libro para + info

17. Sabiendo que frameX2Y.invert() es equivalente a frameY2X, como podríamos obtener una transformada del frame X al frame Z (frameX2Z)

18. En ROS, ¿qué variables debemos utilizar en nuestro ordenador para poder conectarnos con una CPU de un robot en remoto?

19. Si tenemos un subscriptor con un tamaño de cola 1 y un sensor publica 2 mensajes, ¿qué ocurre con el subscriptor?:

    - Se desborda la cola
    - Pilla el mensaje más reciente y desprecia el más antiguo
    - Pilla el mensaje más antiguo y desprecia el más reciente
    - Ninguna es correcta
    - Todas son correctas

20. Si tenemos un sensor que publica un determinado tipo de mensaje a una frecuencia de 20Hz, lo ideal es...

    - Crear un publicador que itere a 20Hz
    - Crear un publicador que itere a menos de 20Hz
    - Crear un publicador que itere a más de 20Hz
    - Ninguna es correcta
    - Todas son correctas

21. ¿Como podemos localizar el master en ROS 2?

22. ¿Es correcta la comunicación entre publicador y subscriptor en ROS 2 si especificamos al subscriptor que sea Volatile y Transient Local?

26. Referencias

Sección 1 y 2: *ROS Wiki*
Sección 10: *GitHub Proyect*
Sección 11: *GitHub Proyect*
Sección 14: *BT WebPage*
Sección 15: *GitHub Proyect*
Sección 18: *GitHub Proyect*

27. Agradecimientos

Adrián Madinabeitia Portanova
Guillermo Alcocer Quesada
Iván Porras Estébanez

Por su trabajo y contribución en los proyectos realizados en los repositorios mencionados en las secciones 10, 15 y 18. Miembros del equipo **TayROS**