

Universidad de Guadalajara
Centro Universitario de Ciencias Exactas e
Ingenierías

Alumno: José de Jesús Fernández García

Código: 214758062

Carrera: INNI

Dependencia: CUCEI / Departamento de Ciencias

Computacionales

Programa: Apoyo Laboratorio de Sistemas Inteligentes

Receptor: Dr. Arturo Valdivia Gonzalez

Profesor: Dr. Carlos Alberto Lopez Franco



Reporte 2 26-05-2021 - 26-07-2021

INDICE DE CONTENIDO

INTRODUCCION.....	3
DESARROLLO	4
Introducción al espacio de trabajo ROS	4
Configuración de espacio de trabajo ROS usando Docker	4
Guardar cambios realizados en un contenedor (como un “commit” en git)	5
Extension “Remote - Containers” Visual Studio Code.....	5
Sistema de archivos ROS	6
Creación de paquetes ROS	7
Tareas ROS.....	9
Creación de un nodo publicador y suscriptor utilizando un mensaje personalizado.....	9
Creación de un servicio que realiza la conversión de metros a pies	13
Creación de una acción que realiza un conteo hasta un número en específico	18
Servicios y acciones en ROS	27
Estructura de archivos final.....	30
Practica.....	31
Descripción.....	31
Trabajo en ROS	33
Trabajo en Unity	53
Integración ROS-Unity	69
Repaso y material utilizado	76
Repaso.....	76
Material.....	79
CONCLUSION.....	80

INTRODUCCION

Algo del trabajo realizado es lo siguiente:

- Material de apoyo:

<https://www.edx.org/es/course/hello-real-world-with-ros-robot-operating-system>

Es un curso bastante bueno que me sirvió de apoyo para aprender los fundamentos de ROS más afondo.

- Se repasaron los conceptos fundamentales de ROS (nodos y temas).
- Se dio una introducción a las diferentes partes que conforman a una aplicación de ROS (ya una aplicación en la vida real).
- Se repasaron los diferentes archivos que conforman un espacio de trabajo ROS (a la hora de generar nuestro espacio de trabajo ROS se generan ciertas carpetas y archivos, los cuales son importantes comprender).
- Se crearon mensajes personalizados (también tenemos mensajes ya definidos que están contenidos en paquetes ROS) y se aprendió a utilizarlos en la práctica.
- Se habló sobre los servicios y acciones en ROS, y cuáles son sus diferencias. Así también a cómo implementarlos.
- Se habló sobre el paquete **actionlib** y como aprender a usarlo.
- Se habló más afondo sobre los archivos de lanzamiento y cómo es que podemos ejecutar una aplicación entera de ROS con un solo archivo de lanzamiento.
- Como práctica, todo lo visto anteriormente se puso en práctica para simular una comunicación entre ROS y Unity por medio de un escenario, simulando una cinta transportadora que transporta cubos de diferentes tamaños, en donde se ve involucrado un sensor que detecta la altura de los cubos y se toma una decisión sobre si se permite que continúe el cubo o no; si se le permite, entonces tiene que ser transportada de una posición a otra (simulando una planificación de trayectoria); si no se le permite, entonces se elimina dicho cubo.

Todo esto es llevado a cabo por medio de suscripciones, llamadas a servicio (servidor de servicio) y llamadas a acciones (servidor de acción), tratando de simular una fábrica y resolviendo el problema de la comunicación entre “robots” utilizando ROS.

DESARROLLO

Se trabajó en un sistema operativo Linux (en una distribución Ubuntu 20.0).

NOTA: Consultar el apartado “Material” para consultar el material utilizado.

Introducción al espacio de trabajo ROS

URL del curso de ROS que se tomó como apoyo:

<https://www.edx.org/es/course/hello-real-world-with-ros-robot-operating-system>

Configuración de espacio de trabajo ROS usando Docker

Lo primero que haremos será construir la imagen de ROS Docker mediante el archivo Dockerfile que se ha proporcionado ejecutando los siguientes comandos:

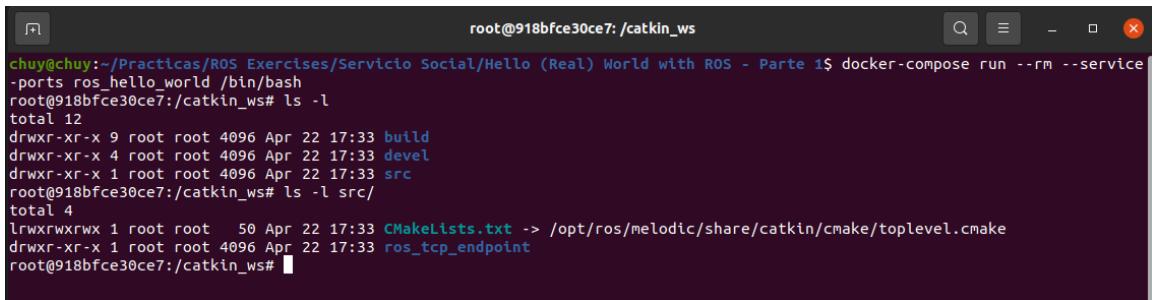
NOTA: Es necesario estar ubicados fuera del directorio.

- `docker build -t chuy7/ros_hello_world:practice1 -f ros_docker/Dockerfile`.
- El Dockerfile proporcionado utiliza la imagen base melodic de ROS (ros:melodic-ros-base) que se puede encontrar en **Docker Hub** (Docker Hub es un repositorio público en la nube para distribuir los contenedores).

Lo que hace este archivo Dockerfile es copiar unos archivos de configuración de nuestra máquina a la imagen ROS que estamos creando, así como de clonar desde el repositorio de GitHub el paquete ROS TCP Connector, el cual nos ayudará para la comunicación entre ROS y Unity.

Ahora iniciamos un contenedor Docker de la imagen recién construida utilizando **docker-compose**. Para esto necesitamos crear un archivo llamado “`docker-compose.yaml`”, similar al que se presenta en el material proporcionado. Despues ejecutamos el siguiente comando:

- `docker-compose run --rm --service-ports ros_hello_world /bin/bash`



```
root@918bfce30ce7:/catkin_ws
chuy@chuy:~/Practicas/ROS Exercises/Servicio Social/Hello (Real) World with ROS - Parte 1$ docker-compose run --rm --service
-ports ros_hello_world /bin/bash
root@918bfce30ce7:/catkin_ws# ls -l
total 12
drwxr-xr-x 9 root root 4096 Apr 22 17:33 build
drwxr-xr-x 4 root root 4096 Apr 22 17:33 devel
drwxr-xr-x 1 root root 4096 Apr 22 17:33 src
root@918bfce30ce7:/catkin_ws# ls -l src/
total 4
lrwxrwxrwx 1 root root    50 Apr 22 17:33 CMakeLists.txt -> /opt/ros/melodic/share/catkin/cmake/toplevel.cmake
drwxr-xr-x 1 root root 4096 Apr 22 17:33 ros_tcp_endpoint
root@918bfce30ce7:/catkin_ws#
```

Hasta aquí el espacio de trabajo ROS ya está listo para aceptar comandos.

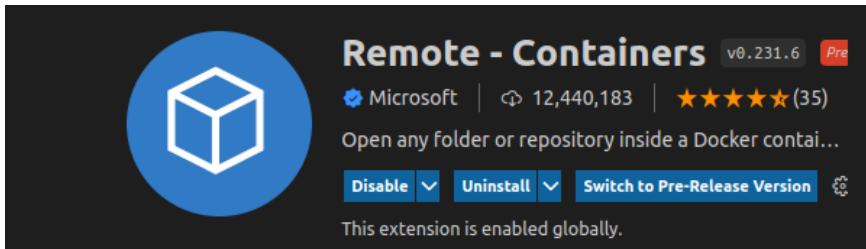
Guardar cambios realizados en un contenedor (como un “commit” en git)

Es posible guardar los cambios realizados dentro de un contenedor que se esta ejecutando mediante el siguiente comando:

```
docker commit <id-contenedor> <repositorio>:<etiqueta>
```

De esta manera si salimos del contenedor no perderemos nuestros cambios realizados, ya que tendremos en una “nueva imagen” los cambios que anteriormente se hicieron.

Extension “Remote - Containers” Visual Studio Code



Esta extensión nos permite acceder a un contenedor Docker en ejecución con el editor de código “Visual Studio Code”. Esto permite poder editar y crear de una manera más cómoda scripts, los cuales iremos generando.

Esto es simplemente como una ayuda, ya que crear y editar scripts desde consola es algo tedioso.

Lo primero que tenemos que hacer es ejecutar el siguiente comando para correr nuestro contenedor de ROS:

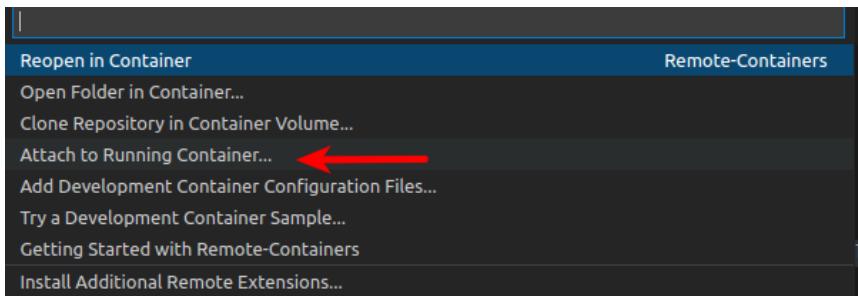
- `docker-compose run --rm --service-ports ros_hello_world /bin/bash`

Lo que se tiene que hacer es dar clic en la parte de color verde del editor de código “Visual Studio Code” (una vez ya instalada la extensión).

NOTA: Obviamente se tiene que tener instalado “Visual Studio Code” en nuestro ordenador.



Después seleccionar la opción “Attach to Running Container...” y seleccionar el contenedor al que se quiere acceder.



Ya con esto podremos editar y crear scripts dentro de un contenedor de una manera más cómoda, y además, como ya se mencionó, podremos guardar los cambios que hayamos realizados dentro del contenedor.

Sistema de archivos ROS

Para poder construir nuestra propia aplicación tendremos que escribir nuestros propios archivos de código.

Necesitamos entender dos términos muy usados en un sistema de archivos ROS:

- Espacio de trabajo ROS (en particular un espacio de trabajo catkin)

En un espacio de trabajo ROS tenemos:

- Un espacio de origen (“src”)

Todos nuestros archivos de código van en esta carpeta.

- Un espacio de desarrollo (“devel”)

- Un espacio de construcción (“build”)

catkin es una herramienta de construcción que compila los archivos de código fuente de un proyecto en binarios ejecutables (en realidad hace mucho mas).

En realidad un espacio de trabajo catkin es un espacio de trabajo ROS que usa **catkin** como herramienta de construcción.

```
root@918bfce30ce7:/catkin_ws# ls -la
total 24
drwxr-xr-x 1 root root 4096 Apr 22 17:33 .
drwxr-xr-x 1 root root 4096 Apr 22 17:39 ..
-rw-r--r-- 1 root root 98 Apr 22 17:33 .catkin_workspace
drwxr-xr-x 9 root root 4096 Apr 22 17:33 build
drwxr-xr-x 4 root root 4096 Apr 22 17:33 devel
drwxr-xr-x 1 root root 4096 Apr 22 17:33 src
root@918bfce30ce7:/catkin_ws#
```

- Paquetes ROS
Los paquetes ROS se pueden utilizar para organizar diferentes módulos funcionales en una aplicación de software ROS.

Creación de paquetes ROS

Crearemos dos paquetes ROS de la siguiente manera (en el orden en el que se encuentran):

NOTA: Nos ubicamos en \$ROS_WORKSPACE/src

- **hello_world_msgs**

Este paquete contendrá todo lo relacionado a mensajes (así como de servicios y de acciones)

Comando para la creación del paquete:

```
catkin_create_pkg hello_world_msgs rospy std_msgs
message_generation
```

- **hello_world**

Este paquete contendrá todo lo relacionado a scripts de Python, archivos de lanzamiento y archivos de configuración.

Comando para la creación del paquete:

```
catkin_create_pkg hello_world rospy std_msgs message_generation
ros_tcp_endpoint hello_world_msgs
```

Notar que el paquete "hello_world" depende del paquete "hello_world_msgs" que acabamos de crear (así como también del paquete "ros_tcp_endpoint" y otros).

El comando para la creación de paquetes tiene la siguiente sintaxis:

```
catkin_create_pkg <nombre-paquete> <dependencia1> <dependencia2> ...
```

Todas las dependencias de un paquete ROS recién creado se enumeran automáticamente en el archivo “package.xml” dentro de la carpeta del paquete ROS.

En el caso de que nos demos cuenta de que nuestra aplicación depende de más paquetes ROS, que no previmos en el momento de la creación, simplemente podemos agregarlo al archivo “package.xml”.

Si realizamos algunas modificaciones a estos archivos, para asegurarnos de que todo se actualice, ejecutamos el comando “catkin_make” en la raíz de nuestro espacio de trabajo ROS.

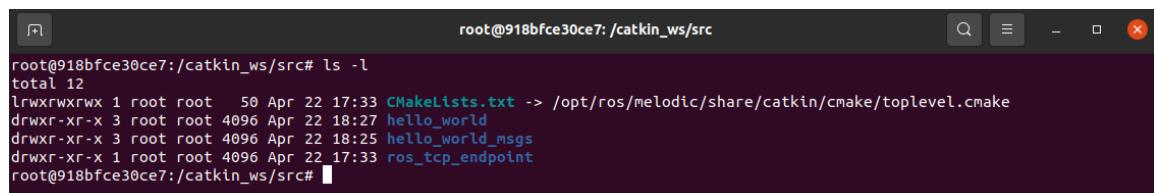
Nuestra estructura de archivos tenemos que tenerla de la siguiente manera:

NOTA: Algunas carpetas se tienen que crear.

- hello_world
 - config (carpeta vacía)
 - launch (carpeta vacía)
 - scripts (carpeta vacía)
 - src (carpeta vacía)
 - **CMakeLists.txt**
 - **package.xml**
- hello_world_msgs
 - action (carpeta vacía)
 - msg (carpeta vacía)
 - srv (carpeta vacía)
 - **CMakeLists.txt**
 - **package.xml**

Más adelante poblaremos estas carpetas con archivos que iremos creando.

NOTA: La presencia de los archivos “CMakeLists.txt” y “package.xml” es lo que convierte una carpeta normal en un paquete ROS.



```
root@918bfce30ce7:/catkin_ws/src# ls -l
total 12
lrwxrwxrwx 1 root root 50 Apr 22 17:33 CMakeLists.txt -> /opt/ros/melodic/share/catkin/cmake/toplevel.cmake
drwxr-xr-x 3 root root 4096 Apr 22 18:27 hello_world
drwxr-xr-x 3 root root 4096 Apr 22 18:25 hello_world_msgs
drwxr-xr-x 1 root root 4096 Apr 22 17:33 ros_tcp_endpoint
root@918bfce30ce7:/catkin_ws/src#
```

Hasta aquí hemos creado dos paquetes ROS desde cero.

Tareas ROS

Creación de un nodo publicador y suscriptor utilizando un mensaje personalizado

NOTA: Debemos asegurarnos de que todo archivo que creemos tenga permisos de ejecución, de lo contrario, debemos agregar permisos de ejecución. Para esto se debe ejecutar el siguiente comando:

```
chmod +x <archivo>
```

Crearemos un tipo de mensaje personalizado dentro del paquete “hello_world_msgs” en la carpeta “msg” con el nombre de “SensorInformation.msg” que contendrá la información de un sensor de distancia. Este mensaje contiene lo siguiente:

```
sensor_msgs/Range sensor_data
string maker_name
uint32 part_number
```

Donde:

- *sensor_data*: Datos leidos por el sensor.
- *maker_name*: Nombre del fabricante.
- *part_number*: Numero de parte.

Notar que estamos haciendo uso de otro paquete llamado “sensor_msgs” el cual contiene un tipo de mensaje llamado “Range” el cual utilizaremos. Para esto será necesario realizar algunas modificaciones a los archivos “CMakeLists.txt” y “package.xml” del paquete “hello_world_msgs”.

Hacemos las siguientes modificaciones en el paquete “hello_world_msgs” (modificaciones que serán necesarias más adelante):

- En el archivo “CMakeLists.txt”

```
find_package(catkin REQUIRED COMPONENTS
message_generation
rospy
```

```

    std_msgs
    sensor_msgs (se agrego)
    actionlib_msgs (se agrego)
)

add_message_files( (se descomento)
    DIRECTORY msg (se agrego)
)

add_service_files( (se descomento)
    DIRECTORY srv (se agrego)
)

add_action_files( (se descomento)
    DIRECTORY action (se agrego)
)

generate_messages( (se descomento)
    DEPENDENCIES
    std_msgs
    sensor_msgs (se agrego)
    actionlib_msgs (se agrego)
)

catkin_package( (se descomento)
    CATKIN_DEPENDS message_runtime (se agrego) rospy std_msgs
    sensor_msgs (se agrego) actionlib_msgs (se agrego)
)

```

- En el archivo “package.xml”
(se agregaron)

```

<exec_depend>message_runtime</exec_depend>

<build_depend>sensor_msgs</build_depend>
<build_export_depend>sensor_msgs</build_export_depend>
<exec_depend>sensor_msgs</exec_depend>

```

```

<build_depend>actionlib_msgs</build_depend>
<build_export_depend>actionlib_msgs</build_export_depend>
<exec_depend>actionlib_msgs</exec_depend>

```

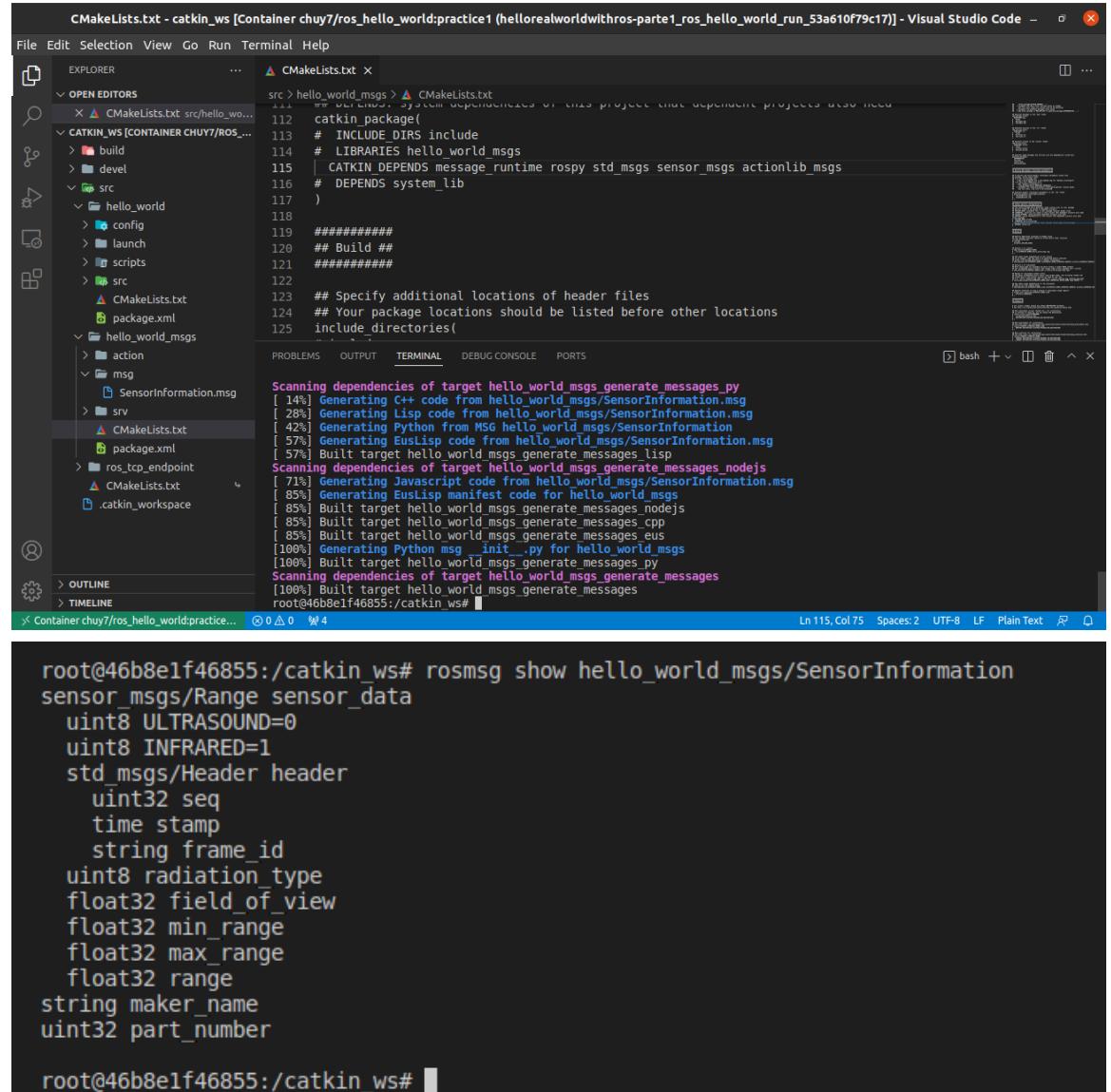
Por ultimo ejecutamos el comando “catkin_make” estando ubicados en el directorio de trabajo para que se vean reflejados los cambios que se hicieron:

```

cd $ROS_WORKSPACE
catkin_make

```

Resultado:



```

CMakeLists.txt - catkin_ws [Container chuy7/ros_hello_world:practice1 (hellorealworldwithros-parte1_ros_hello_world_run_53a610f79c17)] - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER CMakeLists.txt
CATKIN_WS [CONTAINER CHUY7/ROS... src/hello_wor...
src/hello_world_msgs > CMakeLists.txt
112 catkin_package()
113 # INCLUDE_DIRS include
114 # LIBRARIES hello_world_msgs
115 # CATKIN_DEPENDS message_runtime rospy std_msgs sensor_msgs actionlib_msgs
116 # DEPENDS system_lib
117 )
118 #####
119 ## Build ##
120 ## Your package locations should be listed before other locations
121 #####
122 ## Specify additional locations of header files
123 ## Your package locations should be listed before other locations
124 include_directories(
125
Scanning dependencies of target hello_world_msgs_generate_messages_py
[ 14%] Generating C++ code from hello_world_msgs/SensorInformation.msg
[ 28%] Generating Lisp code from hello_world_msgs/SensorInformation.msg
[ 42%] Generating Python code from MSG hello_world_msgs/SensorInformation
[ 57%] Generating EusLisp code from hello_world_msgs/SensorInformation.msg
[ 57%] Built target hello_world_msgs_generate_messages_lisp
Scanning dependencies of target hello_world_msgs_generate_messages_nodejs
[ 71%] Generating Javascript code from hello_world_msgs/SensorInformation.msg
[ 85%] Generating EusLisp manifest code for hello_world_msgs
[ 85%] Built target hello_world_msgs_generate_messages_nodejs
[ 85%] Built target hello_world_msgs_generate_messages_cpp
[ 85%] Built target hello_world_msgs_generate_messages_eus
[100%] Generating Python msg_init_.py for hello_world_msgs
[100%] Built target hello_world_msgs_generate_messages_py
Scanning dependencies of target hello_world_msgs_generate_messages
[100%] Built target hello_world_msgs_generate_messages
root@46b8e1f46855:/catkin_ws# rosmsg show hello_world_msgs/SensorInformation
sensor_msgs/Range sensor_data
uint8 ULTRASOUND=0
uint8 INFRARED=1
std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
uint8 radiation_type
float32 field_of_view
float32 min_range
float32 max_range
float32 range
string maker_name
uint32 part_number
root@46b8e1f46855:/catkin_ws#

```

Ahora creemos los scripts que ejecutaran los nodos publicador y suscriptor:

Los scripts estaran ubicados en el paquete “hello_world” en la carpeta “scripts”.

- **publisher_node.py**

Publica un mensaje en el tema “message” una sola vez o a una cierta tasa.

Definimos una clase que hara lo siguiente:

- *Constructor(<nombre del tema>):*

Se inicializa un nodo ROS y se crea una instancia para publicar en el tema “message” (utilizando la libreria “rospy” de ROS).

- *Metodos:*

- *__publish(<mensaje>)*

Publica el mensaje.

- *publish(<mensaje>, <tasa de publicacion>)*

Publica una sola vez o a una cierta tasa el mensaje.

Funcion principal:

- *main():*

Se crea una instancia de la clase, se obtiene el mensaje y la tasa de publicacion (en segundos) de la lista de argumentos (que se pasan a la hora de ejecutar el script) y se realiza la tarea de publicar (hasta que el nodo se apague).

- **subscriber_node.py**

Se suscribe al tema “message”.

Definimos una clase que hara lo siguiente:

- *Constructor(<nombre del tema>):*

Se inicializa un nodo ROS (utilizando la libreria “rospy” de ROS).

- *Metodos:*

- *__callback(<datos del mensaje>)*

Funcion de devolucion de llamada (nos devuelve los datos del tema “message”).

- *subscribe(<nombre del tema>)*

Se suscribe al tema “message” y se especifica la funcion de devolucion de llamada (la funcion que capturara dichos datos del tema).

Funcion principal:

- *main():*

Se crea una instancia de la clase especificando el tema a suscribirnos y se

realiza la tarea de suscripción (hasta que el nodo se apague).

Resultados:

Comandos:

```
roscore &
rosrun hello_world publisher_node.py "Este es un mensaje" 2 &
rostopic echo /message
```

Imágenes:

```
root@ee0b2fb90469:/catkin_ws# rosrun hello_world publisher_node.py "Este es un mensaje" 2 &
[2] 15424
root@ee0b2fb90469:/catkin_ws# [INFO] [1650824107.215948]: Publishing message
root@ee0b2fb90469:/catkin_ws# rostopic echo /message
data: "Este es un mensaje"
---
data: "Este es un mensaje"
^Croot@ee0b2fb90469:/catkin_ws#
```

```
root@ee0b2fb90469:/catkin_ws# rostopic info /message
Type: std_msgs/String

Publishers:
* /publisher_node (http://ee0b2fb90469:45973/)

Subscribers: None

root@ee0b2fb90469:/catkin_ws#
```

Creación de un servicio que realiza la conversión de metros a pies

NOTA: Debemos asegurarnos de que todo archivo que creemos tenga permisos de ejecución, de lo contrario, debemos agregar permisos de ejecución. Para esto se debe ejecutar el siguiente comando:

```
chmod +x <archivo>
```

Recordemos que en un servicio tenemos:

- El servidor de servicio (el que sirve el servicio para que los clientes puedan utilizarlo).
- El cliente de servicio (el que hace uso del servicio).

Crearemos una declaracion de servicio dentro del paquete “hello_world_msgs” en la carpeta “srv” con el nombre de “ConvertMetresToFeet.srv” que contendra la declaracion del tipo de mensaje de solicitud y el tipo de mensaje de respuesta. Este mensaje contiene lo siguiente:

```
float64 measurement_metres      # request message
---
float64 measurement_feets       # response message
bool success                     # response message
```

Donde:

- Mensaje de solicitud
 - *measurement_metres*: Medida en metros.
- Mensaje de respuesta.
 - *measurement_feet*: Medida en pies.
 - *success*: Indica si se ha realizado con exito la conversion.

Por ultimo ejecutamos el comando “catkin_make” estando ubicados en el directorio de trabajo para que se vea reflejado la declaracion de servicio:

```
cd $ROS_WORKSPACE
catkin_make --force-cmake
```

NOTA: Es importante agregar la bandera *--force-cmake* al comando *catkin_make* porque sino no genera los mensajes para la declaracion de servicio.

Resultado:

```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE PORTS bash + ×
```

```
#### Running command: "make -j4 -l4" in "/catkin_ws/build"
#####
Scanning dependencies of target _hello_world_msgs_generate_messages_check_deps_ConvertMetresToFeet
[ 0%] Built target std_msgs_generate_messages_cpp
[ 0%] Built target sensor_msgs_generate_messages_cpp
[ 0%] Built target std_msgs_generate_messages_lisp
[ 0%] Built target actionlib_msgs_generate_messages_cpp
[ 0%] Built target hello_world_msgs_generate_messages_check_deps_ConvertMetresToFeet
[ 0%] Built target _hello_world_msgs_generate_messages_check_deps_SensorInformation
[ 0%] Built target sensor_msgs_generate_messages_eus
[ 0%] Built target actionlib_msgs_generate_messages_lisp
[ 0%] Built target std_msgs_generate_messages_eus
[ 0%] Built target sensor_msgs_generate_messages_lisp
[ 0%] Built target actionlib_msgs_generate_messages_eus
[ 0%] Built target actionlib_msgs_generate_messages_py
[ 0%] Built target std_msgs_generate_messages_py
[ 0%] Built target sensor_msgs_generate_messages_py
[ 0%] Built target actionlib_msgs_generate_messages_nodejs
[ 0%] Built target std_msgs_generate_messages_nodejs
[ 7%] Generating C++ code from hello_world_msgs/ConvertMetresToFeet.srv
[ 7%] Built target sensor_msgs_generate_messages_nodejs
[ 15%] Generating Lisp code from hello_world_msgs/ConvertMetresToFeet.srv
[ 23%] Generating EusLisp code from hello_world_msgs/ConvertMetresToFeet.srv
[ 30%] Generating Python code from SRV hello_world_msgs/ConvertMetresToFeet
[ 38%] Built target hello_world_msgs_generate_messages_lisp
[ 53%] Generating Javascript code from hello_world_msgs/ConvertMetresToFeet.srv
[ 61%] Built target hello_world_msgs_generate_messages_eus
[ 69%] Built target hello_world_msgs_generate_messages_nodejs
[ 76%] Generating Python srv __init__.py for hello_world_msgs
[ 84%] Generating Python msg __init__.py for hello_world_msgs
[ 92%] Built target hello_world_msgs_generate_messages_py
[100%] Built target hello_world_msgs_generate_messages_cpp
[100%] Built target hello_world_msgs_generate_messages
root@ee0b2fb90469:/catkin_ws#
```

```
root@ee0b2fb90469:/catkin_ws# rossrv show hello_world_msgs/ConvertMetresToFeet
float64 measurement_metres
---
float64 measurement_feets
bool success

root@ee0b2fb90469:/catkin_ws#
```

```
root@76c2ec24da9c:/catkin_ws# rossrv list
diagnostic_msgs/AddDiagnostics
diagnostic_msgs/SelfTest
dynamic_reconfigure/Reconfigure
hello_world_msgs/ConvertMetresToFeet ←
nav_msgs/GetMap
nav_msgs/GetPlan
nav_msgs/LoadMap
nav_msgs/SetMap
nodelet/NodeletList
nodelet/NodeletLoad
nodelet/NodeletUnload
roscpp/Empty
roscpp/GetLoggers
roscpp/SetLoggerLevel
sensor_msgs/SetCameraInfo
std_srvs/Empty
std_srvs/SetBool
std_srvs/Trigger
topic_tools/DemuxAdd
topic_tools/DemuxDelete
topic_tools/DemuxList
topic_tools/DemuxSelect
topic_tools/MuxAdd
topic_tools/MuxDelete
topic_tools/MuxList
topic_tools/MuxSelect
root@76c2ec24da9c:/catkin_ws#
```

Ahora creamos dos scripts para el servicio (un servidor y un cliente):

Los scripts estaran ubicados en el paquete “hello_world” en la carpeta “scripts”.

- **metres_to_feet_server.py**

Declara un servicio que convierte de metros a pies.

Definimos una clase que hara lo siguiente:

- *Constructor():*

Se inicializa un nodo ROS y se declara una variable que ayudara en la conversion de metros a pies (es un factor que multiplica la cantidad en metros y el resultado es la cantidad convertida en pies).

- *Métodos:*

- *createService(<nombre del servicio>)*

Crea un servicio y especifica la función que estara manejando las solicitudes y devolviendo una respuesta.

- *__handler(<solicitud>)*

Función que maneja las solicitudes y devuelve una respuesta (devuelve la conversión de metros a pies).

Función principal:

- ***main():***

Se crea una instancia de la clase y se crea el servicio para mantenerse ejecutando (hasta que el nodo se apague).

- ***metres_to_feet_client.py***

Hace uso del servicio de conversión de metros a pies (actúa como un cliente de servicio).

Definimos una clase que hará lo siguiente:

- ***Constructor():***

No se hace nada aquí.

- ***Métodos:***

- ***callService(<nombre del servicio>, <medida en metros>)***

Llama al servicio enviando una solicitud (la medida en metros) y capturando la respuesta del servicio (como respuesta es la medida en pies).

Funcion principal:

- ***main():***

Se crea una instancia de la clase, se obtiene la medida en metros de la lista de argumentos (que se pasa a la hora de ejecutar el script) y se realiza la tarea de llamar al servicio.

Resultados:

Comandos:

roscore &

rosrun hello_world publisher_node.py "Este es un mensaje" 2 &

rostopic echo /message

Imagenes:

```

root@ee0b2fb90469:/catkin_ws# rosrun hello_world metres_to_feet_server.py &
[2] 44860
root@ee0b2fb90469:/catkin_ws# [INFO] [1650826795.154075]: 
Service created

root@ee0b2fb90469:/catkin_ws# rosrun hello_world metres_to_feet_client.py 1
Request: measurement_metres: 1.0 mt
Response: 3.281 ft
root@ee0b2fb90469:/catkin_ws# rosrun hello_world metres_to_feet_client.py 12
Request: measurement_metres: 12.0 mt
Response: 39.372 ft
root@ee0b2fb90469:/catkin_ws# rosrun hello_world metres_to_feet_client.py 1.5
Request: measurement_metres: 1.5 mt
Response: 4.9215 ft
root@ee0b2fb90469:/catkin_ws# █

```

```

root@ee0b2fb90469:/catkin_ws# rosservice list
/metres_to_feet
/metres_to_feet_server/get_loggers
/metres_to_feet_server/set_logger_level
/rosout/get_loggers
/rosout/set_logger_level
root@ee0b2fb90469:/catkin_ws# rosservice info /metres_to_feet
Node: /metres_to_feet_server
URI: rosrpc://ee0b2fb90469:40523
Type: hello_world_msgs/ConvertMetresToFeet
Args: measurement_metres
root@ee0b2fb90469:/catkin_ws# rosnode list
/metres_to_feet_server
/rosout
root@ee0b2fb90469:/catkin_ws# rostopic list
/rosout
/rosout_agg
root@ee0b2fb90469:/catkin_ws# rosservice call /metres_to_feet "measurement_metres: 10"
Request: measurement_metres: 10.0 mt
measurement_feet: 32.81
success: True
root@ee0b2fb90469:/catkin_ws# █

```

En el ultimo comando se llama al servicio desde consola (sin necesidad de crear un cliente mediante un script).

Creación de una acción que realiza un conteo hasta un número en específico

NOTA: Debemos asegurarnos de que todo archivo que creemos tenga permisos de ejecución, de lo contrario, debemos agregar permisos de ejecución. Para esto se debe ejecutar el siguiente comando:

chmod +x <archivo>

En una acción en ROS tenemos:

- El servidor de acción (el que sirve la acción para que los clientes puedan utilizarlo).

- El cliente de acción (el que hace uso de la acción).

Crearemos una declaración de acción dentro del paquete “hello_world_msgs” en la carpeta “action” con el nombre de “CounterWithDelay.action” que contendrá la declaración del tipo de mensaje de objetivo, el tipo de mensaje de resultado y el tipo de mensaje de retroalimentación. Este mensaje contiene lo siguiente:

```
uint32 num_counts          # goal message
---
string result_message     # result message
---
uint32 counts_elapsed     # feedback message
```

Donde:

- Mensaje de objetivo
 - *num_counts*: Se indica hasta qué número se contará.
- Mensaje de resultado
 - *result_message*: Mensaje de que se ha terminado de ejecutar la acción.
- Mensaje de retroalimentación.
 - *counts_elapsed*: Se indica qué número va del conteo.

Por último ejecutamos el comando “catkin_make” estando ubicados en el directorio de trabajo para que se vea reflejado la declaración de acción:

```
cd $ROS_WORKSPACE
catkin_make --force-cmake
```

NOTA: Es importante agregar la bandera *--force-cmake* al comando *catkin_make* porque sino no genera los mensajes para la declaración de acción.

Resultado:

```

[ 35%] Generating Lisp code from hello_world_msgs/CounterWithDelayActionFeedback.msg
[ 37%] Generating EusLisp code from hello_world_msgs/CounterWithDelayAction.msg
[ 39%] Generating Javascript code from hello_world_msgs/CounterWithDelayActionResult.msg
[ 41%] Generating Lisp code from hello_world_msgs/CounterWithDelayActionResult.msg
[ 43%] Generating Javascript code from hello_world_msgs/CounterWithDelayResult.msg
[ 45%] Generating EusLisp code from hello_world_msgs/CounterWithDelayFeedback.msg
[ 47%] Generating Lisp code from hello_world_msgs/CounterWithDelayResult.msg
[ 50%] Generating Javascript code from hello_world_msgs/ConvertMetresToFeet.srv
[ 52%] Generating C++ code from hello_world_msgs/CounterWithDelayGoal.msg
[ 54%] Generating Lisp code from hello_world_msgs/ConvertMetresToFeet.srv
[ 56%] Generating EusLisp code from hello_world_msgs/CounterWithDelayActionFeedback.msg
[ 56%] Built target hello_world_msgs_generate_messages_nodejs
[ 58%] Generating Python from MSG hello_world_msgs/SensorInformation
[ 58%] Built target hello_world_msgs_generate_messages_lisp
[ 60%] Generating EusLisp code from hello_world_msgs/CounterWithDelayActionResult.msg
[ 62%] Generating C++ code from hello_world_msgs/CounterWithDelayAction.msg
[ 64%] Generating EusLisp code from hello_world_msgs/CounterWithDelayResult.msg
[ 66%] Generating EusLisp code from hello_world_msgs/ConvertMetresToFeet.srv
[ 68%] Generating C++ code from hello_world_msgs/CounterWithDelayFeedback.msg
[ 70%] Built target hello_world_msgs_generate_messages_eus
[ 72%] Generating Python from MSG hello_world_msgs/CounterWithDelayActionGoal
[ 75%] Generating C++ code from hello_world_msgs/CounterWithDelayActionFeedback.msg
[ 77%] Generating C++ code from hello_world_msgs/CounterWithDelayActionResult.msg
[ 79%] Generating C++ code from hello_world_msgs/CounterWithDelayResult.msg
[ 81%] Generating C++ code from hello_world_msgs/ConvertMetresToFeet.srv
[ 83%] Generating Python from MSG hello_world_msgs/CounterWithDelayGoal
[ 83%] Built target hello_world_msgs_generate_messages_cpp
[ 85%] Generating Python from MSG hello_world_msgs/CounterWithDelayAction
[ 87%] Generating Python from MSG hello_world_msgs/CounterWithDelayFeedback
[ 89%] Generating Python from MSG hello_world_msgs/CounterWithDelayActionFeedback
[ 91%] Generating Python from MSG hello_world_msgs/CounterWithDelayActionResult
[ 93%] Generating Python from MSG hello_world_msgs/CounterWithDelayResult
[ 95%] Generating Python code from SRV hello_world_msgs/ConvertMetresToFeet
[100%] Generating Python srv __init__.py for hello_world_msgs
[100%] Generating Python msg __init__.py for hello_world_msgs
[100%] Built target hello_world_msgs_generate_messages_py
[100%] Built target hello_world_msgs_generate_messages
root@76c2ec24da9c:/catkin_ws# 
```

```

root@76c2ec24da9c:/catkin_ws# rosmsg show hello_world_msgs/CounterWithDelay
Unable to load msg [hello_world_msgs/CounterWithDelay]: Cannot locate message [CounterWithDelay] in package [hello_world_msgs] with paths ['/catkin_ws/src/hello_world_msgs/msg', '/catkin_ws/devel/share/hello_world_msgs/msg']
root@76c2ec24da9c:/catkin_ws# rosmsg show hello_world_msgs/CounterWithDelayGoal
uint32 num_counts

root@76c2ec24da9c:/catkin_ws# rosmsg show hello_world_msgs/CounterWithDelayResult
string result_message

root@76c2ec24da9c:/catkin_ws# rosmsg show hello_world_msgs/CounterWithDelayFeedback
uint32 counts_elapsed

root@76c2ec24da9c:/catkin_ws# 
```

Los mensajes generados terminan en “Goal”, “Result” y “Feedback”, como se muestra en la busqueda.

Otros mensajes que tambien se generan terminan en “ActionResult”, “ActionResult” y “ActionFeedback”, como se muestra a continuacion:

```

root@76c2ec24da9c:/catkin_ws# rosmsg show hello_world_msgs/CounterWithDelayActionGoal
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
actionlib_msgs/GoalID goal_id
  time stamp
  string id
hello_world_msgs/CounterWithDelayGoal goal
  uint32 num_counts

root@76c2ec24da9c:/catkin_ws# rosmsg show hello_world_msgs/CounterWithDelayActionResult
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
actionlib_msgs/GoalStatus status
  uint8 PENDING=0
  uint8 ACTIVE=1
  uint8 PREEMPTED=2
  uint8 SUCCEEDED=3
  uint8 ABORTED=4
  uint8 REJECTED=5
  uint8 PREEMPTING=6
  uint8 RECALLING=7
  uint8 RECALLED=8
  uint8 LOST=9
actionlib_msgs/GoalID goal_id
  time stamp
  string id
  uint8 status
  string text

```

```

root@76c2ec24da9c:/catkin_ws# rosmsg show hello_world_msgs/CounterWithDelayActionFeedback
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
actionlib_msgs/GoalStatus status
  uint8 PENDING=0
  uint8 ACTIVE=1
  uint8 PREEMPTED=2
  uint8 SUCCEEDED=3
  uint8 ABORTED=4
  uint8 REJECTED=5
  uint8 PREEMPTING=6
  uint8 RECALLING=7
  uint8 RECALLED=8
  uint8 LOST=9
actionlib_msgs/GoalID goal_id
  time stamp
  string id
  uint8 status
  string text
hello_world_msgs/CounterWithDelayFeedback feedback
  uint32 counts_elapsed

```

Y por último, otro mensaje que se genera termina en “Action” (el mensaje más importante, ya que este es el que estaremos indicando a la hora de crear un servidor de acción), como se muestra a continuación:

```
root@76c2ec24da9c:/catkin_ws# rosmsg show hello_world_msgs/CounterWithDelayAction
hello_world_msgs/CounterWithDelayActionGoal action_goal
    std_msgs/Header header
        uint32 seq
        time stamp
        string frame_id
    actionlib_msgs/GoalID goal_id
        time stamp
        string id
    hello_world_msgs/CounterWithDelayGoal goal
        uint32 num_counts
hello_world_msgs/CounterWithDelayActionResult action_result
    std_msgs/Header header
        uint32 seq
        time stamp
        string frame_id
    actionlib_msgs/GoalStatus status
        uint8 PENDING=0
        uint8 ACTIVE=1
        uint8 PREEMPTED=2
        uint8 SUCCEEDED=3
        uint8 ABORTED=4
        uint8 REJECTED=5
        uint8 PREEMPTING=6
        uint8 RECALLING=7
        uint8 RECALLED=8
        uint8 LOST=9
    actionlib_msgs/GoalID goal_id
        time stamp
        string id
        uint8 status
        string text
    hello_world_msgs/CounterWithDelayResult result
        string result_message
hello_world_msgs/CounterWithDelayActionFeedback action_feedback
    std_msgs/Header header
        uint32 seq
        time stamp
        string frame_id
    actionlib_msgs/GoalStatus status
        uint8 PENDING=0
        uint8 ACTIVE=1
        uint8 PREEMPTED=2
        uint8 SUCCEEDED=3
        uint8 ABORTED=4
        uint8 REJECTED=5
        uint8 PREEMPTING=6
        uint8 RECALLING=7
        uint8 RECALLED=8
        uint8 LOST=9
    actionlib_msgs/GoalID goal_id
        time stamp
        string id
        uint8 status
        string text
    hello_world_msgs/CounterWithDelayFeedback feedback
        uint32 counts_elapsed
```

```
root@76c2ec24da9c:/catkin_ws#
```

Tener en cuenta que la declaración de un tipo de acción genera varios tipos de mensajes relacionados con el tipo de mensaje de objetivo, resultado y retroalimentación.

Ahora creemos dos scripts para la acción (un servidor y un cliente):

Los scripts estarán ubicados en el paquete “hello_world” en la carpeta “scripts”.

- **counter_with_delay_as.py**

Declara un servidor de acción que realiza un conteo desde 0 hasta un numero en específico menos 1 con un retardo de un segundo entre conteo.

- Como mensaje de objetivo se especifica hasta que numero se desea contar.
- Como mensaje de resultado se devuelve un mensaje de que ya se ha completado de ejecutar la acción.
- Como mensaje de retroalimentación se publica en un tema el valor en el que va el conteo con dicho tipo de mensaje (otros nodos pueden suscribirse en este tema).

Definimos una clase que hará lo siguiente:

- *Constructor():*

Se inicializa un nodo ROS y se inicializan dos variables para los tipos de mensaje de resultado y retroalimentación.

- *Métodos:*

- *createSimpleActionServer(<nombre del servidor de accion>)*

Crea un servidor de accion simple y especifica la funcion que estara manejando las solicitudes de objetivo, publicando la retroalimentacion y, al final, devolviendo un resultado (respuesta).

- *__execute_cb(<objetivo>)*

Funcion que maneja las solicitudes de objetivo, lleva a cabo dicha accion, publica la retroalimentacion mientras se lleva a cabo la accion y, al final, devuelve un resultado.

Funcion principal:

- *main():*

Se crea una instancia de la clase y se crea el servidor de acción simple (manteniéndose ejecutando hasta que el nodo se apague).

- **counter_with_delay_ac.py**

Hace uso del servidor de acción que realiza un conteo desde el 0 hasta un numero en específico menos 1 (actúa como un cliente de acción).

Definimos una clase que hará lo siguiente:

- *Constructor():*

Se inicializa un nodo ROS.

- *Métodos:*
 - *createSimpleActionClient(<nombre del servidor de accion>)*
Crea un cliente de accion simple.
 - *sendGoal(<numero hasta el cual contar>)*
Envia un objetivo al servidor de accion y espera la respuesta de la accion.

Funcion principal:

- *main():*

Se crea una instancia de la clase, se obtiene el número de conteos de la lista de argumentos (que se pasan a la hora de ejecutar el script) y se realiza la tarea de enviar el objetivo al servidor de acción.

Resultados:

Comandos:

```
roscore &  
rosrun hello_world counter_with_delay_as.py &  
rosrun hello_world counter_with_delay_ac.py 20 &  
rostopic echo /counter/feedback
```

Imágenes:

```
root@76c2ec24da9c:/catkin_ws# rosrun hello_world counter_with_delay_as.py &
[2] 38984
root@76c2ec24da9c:/catkin_ws# [INFO] [1650912131.950967]:
Simple action server created

root@76c2ec24da9c:/catkin_ws# rostopic list
/counter/cancel
/counter/feedback
/counter/goal
/counter/result
/counter/status
/rosout
/rosout_agg
root@76c2ec24da9c:/catkin_ws# rosrun hello_world counter_with_delay_ac.py 20 &
[3] 40256
root@76c2ec24da9c:/catkin_ws# [INFO] [1650912218.568779]: Goal has been sent to the action server

root@76c2ec24da9c:/catkin_ws# rostopic echo /counter/feedback
header:
  seq: 15
  stamp:
    secs: 1650912232
    nsecs: 609266042
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1650912218
      nsecs: 568506956
    id: "/counter_with_delay_ac-1-1650912218.569"
    status: 1
    text: "This goal has been accepted by the simple action server"
feedback:
  counts_elapsed: 14
---
header:
  seq: 16
  stamp:
    secs: 1650912233
    nsecs: 610764980
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1650912218
      nsecs: 568506956
    id: "/counter_with_delay_ac-1-1650912218.569"
    status: 1
    text: "This goal has been accepted by the simple action server"
feedback:
  counts_elapsed: 15
---
```

```

header:
  seq: 20
  stamp:
    secs: 1650912237
    nsecs: 619532108
  frame_id: ''
status:
goal_id:
  stamp:
    secs: 1650912218
    nsecs: 568506956
  id: "/counter_with_delay_ac-1-1650912218.569"
status: 1
text: "This goal has been accepted by the simple action server"
feedback:
  counts_elapsed: 19
---
Successfully completed counting
[WARN] [1650912238.661804]: Inbound TCP/IP connection failed: connection from sender terminated before handshake header received. 0 bytes were received. Please check sender for additional details.

^C[WARN] [1650912246.793414]: Inbound TCP/IP connection failed: connection from sender terminated before handshake header received. 0 bytes were received. Please check sender for additional details.
[3]+ Done                  rosrun hello_world counter_with_delay_ac.py 20
root@76c2ec24da9c:/catkin_ws# 
```

Como podemos observar nos hemos suscrito al tema de retroalimentación mientras se procesaba la acción en el servidor de acción.

Ahora repitamos el mismo procedimiento pero en vez de suscribirnos al tema de retroalimentación nos suscribimos al tema de resultado, de la siguiente manera:

```

root@76c2ec24da9c:/catkin_ws# rosrun hello_world counter_with_delay_ac.py 20 &
[3] 52774
root@76c2ec24da9c:/catkin_ws# [INFO] [1650913385.752575]: Goal has been sent to the action server

root@76c2ec24da9c:/catkin_ws# rostopic echo /counter/result
header:
  seq: 2
  stamp:
    secs: 1650913405
    nsecs: 810424089
  frame_id: ''
status:
goal_id:
  stamp:
    secs: 1650913385
    nsecs: 752193927
  id: "/counter_with_delay_ac-1-1650913385.752"
status: 3
text: ''
result:
  result_message: "Successfully completed counting"
---
Successfully completed counting

^C[3]+ Done                  rosrun hello_world counter_with_delay_ac.py 20
root@76c2ec24da9c:/catkin_ws# 
```

Hasta aquí ha sido un repaso de los elementos fundamentales en una aplicación ROS, como lo son las declaraciones de mensaje, de servicio, de acción, así como también lo

que hay detrás y cómo funcionan los publicadores, suscriptores, servidores de servicio y servidores de acción.

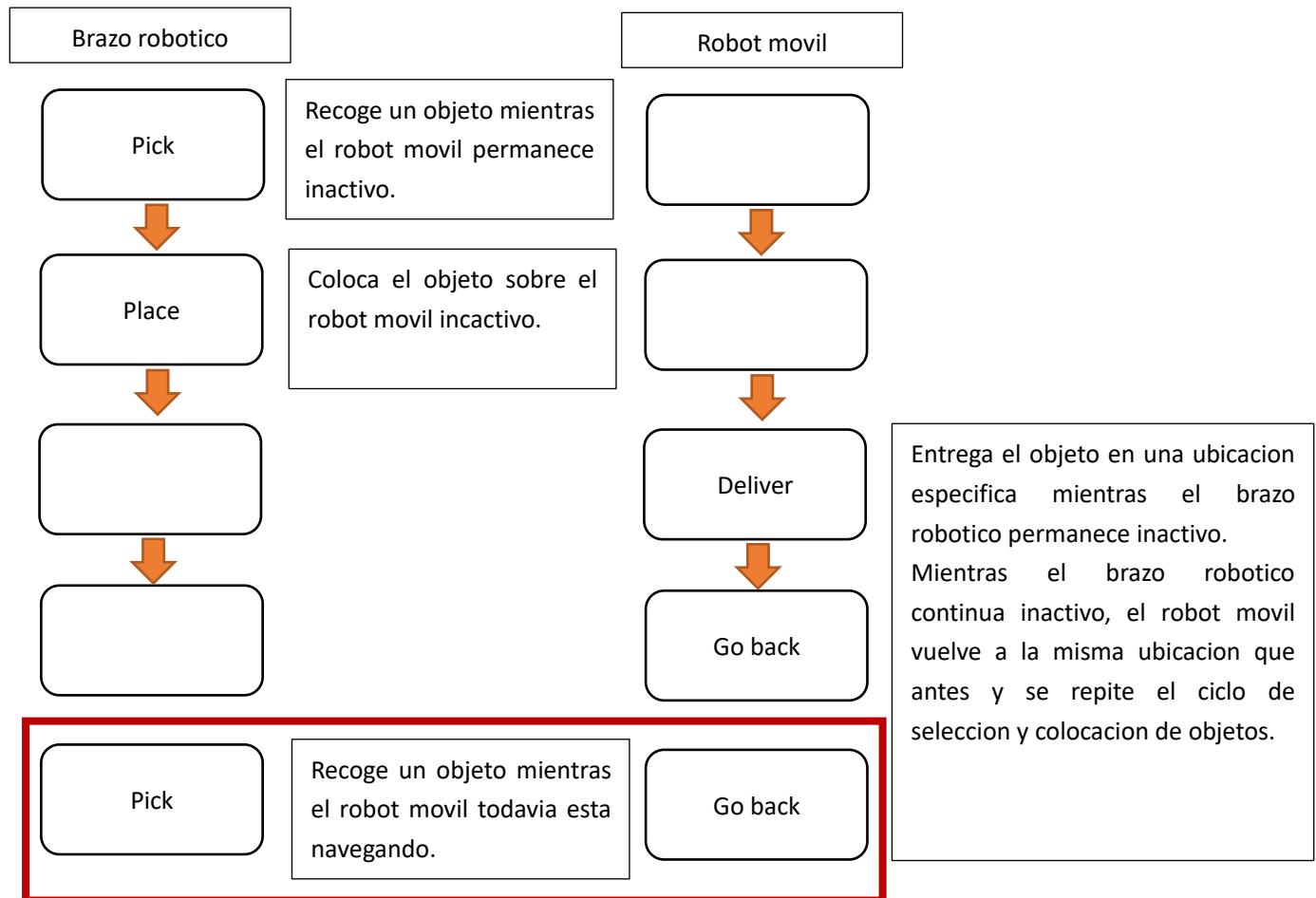
Más adelante se pondrá en práctica todo lo visto anteriormente utilizando ROS y Unity.

Servicios y acciones en ROS

Con los servicios ROS adquirimos y procesamos información solo cuando es necesario. Sin embargo, el bloqueo de las ejecuciones puede no ser siempre deseable.

Una acción puede no ser bloqueante pero un servicio si lo es.

Ejemplo de un caso de ejecución sin bloqueos:



Imaginemos que estamos en el proceso de analizar un proceso existente en una línea de producción de una fábrica donde tenemos un brazo robótico y un robot móvil trabajando juntos. Así como se muestra en el diagrama anterior.

Quizás, todavía sea aconsejable mantener el brazo del robot inactivo y esperar a que el robot móvil entregue con éxito el objeto a su destino. Pero no es eficiente mantener el robot inactivo mientras el robot está navegando de regreso para la próxima recogida.

Sin embargo, si las ejecuciones de movimiento de cada robot se implementan utilizando servicios ROS, esto no sería posible ya que siempre tendremos que esperar a que finalice la navegación antes de que podamos empezar a mover el brazo robótico.

Si todo componente (robots, maquinas, etc.) se está comunicando por medio del sistema ROS, es claro que necesitamos lo más posible un código no bloqueante entre la comunicación, ya que, en la vida real, hemos visto cómo trabajan empresas que utilizan robots para eficientar procesos donde de manera simultánea operan dos o más robots usando el sistema de ROS para la comunicación. Tareas en las que se ve involucrado un sistema de ROS:

- Planificación de trayectoria para robots móviles, brazos robóticos, etc.
- Recoger y colocar objetos.
- Lecturas de sensores publicadas en temas.
- En realidad en todo puede estar incluido un sistema de ROS.
 - Se pueden crear disparadores que estén suscritos a un tema y a la hora de leer algún dato poder tomar alguna acción.
 - Se pueden cancelar acciones (algún proceso).
 - Se puede programar que muchos nodos (cada uno realizando una tarea diferente pero complementaria) se lancen a la vez, y si alguno falla, que todos los demás nodos esperen hasta que todos estén levantados. Esto es fundamental cuando un grupo de nodos depende uno del otro para funcionar.
 - Es posible llevar un control de los nodos que se están ejecutando, los cuales pueden ser ejecutados en un espacio de nombres diferente para poder

Servicio:

Una vez llamado a un servicio, este no puede ser interrumpido, por lo que se debe esperar a que termine.

Acciones:

Una vez llamado a una acción, este puede ser interrumpido y cancelado en cualquier momento del procesamiento de la acción.

acceder a cada uno de ellos de acuerdo a la tarea que ejecutan.

- En general, hay muchas cosas que se pueden realizar en un sistema ROS (tiene bastante documentación y casi siempre se encuentra la manera de hacer algo).

¿Cuándo utilizar servicios?

Cuando se realiza alguna solicitud que tiene como respuesta una tarea rápida de hacer (es muy importante que los servicios estén bien programados, es decir, que sean lo más eficientes y rápidos posibles para que no sean tan bloqueantes).

¿Cuándo utilizar acciones?

Cuando se realiza alguna solicitud que tiene como respuesta una tarea que se lleva en el tiempo, es decir, no es inmediata, y a su vez se necesita que dicha tarea no sea bloqueante.

Es importante mencionar que las acciones ROS están más involucradas en una aplicación de software de robot que los servicios y temas.

Con respecto al paquete ***actionlib***, el cual es utilizado cuando trabajamos con acciones, tenemos lo siguiente:

En las acciones ROS (a diferencia de los servicios):

- El campo de solicitud se llama “objetivo” (“goal”).
 - El campo de respuesta se llama “resultado” (“result”).
 - Un nuevo campo se llama “retroalimentacion” (“feedback”).
- El cual proporciona una retroalimentación continua relacionada con el procesamiento de la solicitud de objetivo.

Una solicitud se puede adelantar y cancelar antes de alcanzar el objetivo (tareas prioritarias).

Tipos de mensajes que se generan en una declaración de acción:

- *<nombre-accion>ActionFeedback.msg*
- *<nombre-accion>ActionGoal.msg*

Nodo ROS:

Recordemos que un nodo ROS es un ***proceso*** que realiza cálculos, y que un nodo ROS se escribe utilizando una librería cliente ROS (como roscpp para C++, o rospy para Python).

actionlib:

La pila actionlib proporciona una interfaz estandarizada para interactuar con ***tareas prioritarias***.

El paquete actionlib proporciona herramientas para crear servidores que ejecutan objetivos a largo plazo, los cuales se pueden adelantar (priorizar).

- `<nombre-accion>ActionResult.msg`
- `<nombre-accion>Action.msg`

Este tipo de mensaje es el tipo de mensaje que se usa en el código como el tipo de mensaje de nuestro servidor de acción y cliente de acción.

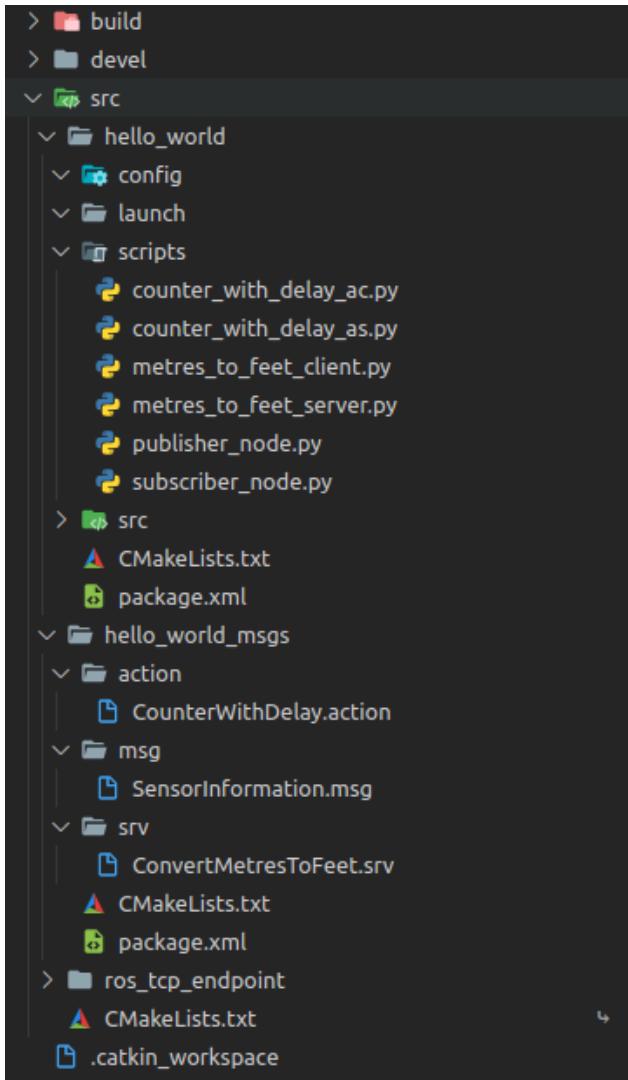
- `<nombre-accion>Feedback.msg`
- `<nombre-accion>Goal.msg`
- `<nombre-accion>Result.msg`

Los primeros cuatro tipos de mensaje son los que utiliza el paquete ***actionlib*** para el manejo interno de nuestros objetivos de acción.

Los últimos tres tipos de mensajes son los que nosotros utilizaremos en el código.

Hay mucha información muy detallada y avanzada sobre el tema de las acciones y sobre el paquete ***actionlib***. Este último paquete es muy importante comprender si deseamos crear aplicaciones de robot limpias e inteligentes. Más adelante veremos que no será el único paquete que se necesita comprender, sino que hay muchísimos paquetes que son importantes dominar para una aplicación de robot mucho más apegada a la realidad (como por ejemplo, el paquete ***navigation***, el cual es una pila de navegación que toma información de flujos de sensores y edometría y emite comandos de velocidad para enviar a una base móvil).

Estructura de archivos final



De esta manera es como queda todo lo que hemos creado hasta el momento, en donde hemos separado los scripts y todo lo que tenga que ver con código en el paquete "hello_world", y todo lo que tenga que ver con declaraciones de mensajes, servicios y acciones en el paquete "hello_world_msgs".

Práctica

Descripción

Como práctica se integrará una comunicación entre ROS y Unity por medio de un escenario, simulando una cinta transportadora que transporta cubos de diferentes tamaños, en donde se verá involucrado un sensor que detecta la altura de los cubos y se tomará una decisión sobre si se permite que continúe el cubo o no; si se le permite, entonces tiene que ser transportada de una posición a otra (simulando una planificación de trayectoria); si no se le permite, entonces se elimina dicho cubo.

Los pasos son los siguientes:

1. Se generaran 10 cubos de diferentes tamaños, cada uno se generara uno después del otro a una cierta tasa (que será aleatoria para cada cubo).
2. Cada cubo pasara por debajo de un sensor de distancia (simulado) que, por medio de unos cálculos, determina la altura del cubo.

A su vez el sensor:

- Llama a un servicio en ROS que recibe la altura del cubo en metros y devuelve una respuesta de si es o no admitido el cubo.

El mismo servicio llama a otro servicio que hace la conversión de metros a pies de una cantidad (solo a modo de ejemplo para involucrar más servicios).

Si la altura es mayor o igual a 6.56 ft (aprox. 2.0 mts) no pasa la prueba, de lo contrario, si pasa la prueba.

- El sensor captura la respuesta del servicio y se toma la decisión de si el cubo continua o se elimina.

3. Los cubos que pasaron la prueba de altura se colocan en una posición en específico para cada cubo.
4. Conforme vayan llegando, el cubo que llega primero, realiza una llamada a un servicio pasando la posición actual del cubo y la posición objetivo del cubo (a donde tiene que ser transportado), que a su vez dicho servicio llama a un servidor de acción con la misma información como objetivo para iniciar con el procesamiento del objetivo.
5. El cubo se suscribe al tema de “feedback” (de retroalimentación) en donde publica el servidor de acción mientras se procesa el objetivo, donde se publica una nueva posición cada segundo (cada vez más cerca de la posición objetivo), y con dicha información se mueve el cubo a tal posición.
6. Una vez que el cubo llega a su posición objetivo (el servidor de acción termina de procesar el objetivo), el cubo estará situado en otra posición.
7. El siguiente cubo repite los pasos 4), 5) y 6), hasta que ya no haya cubos que transportar.
8. Al final tendremos todos los cubos que pasaron la prueba de altura en una ubicación donde estarán todos ordenados.

Lo que se busco fue simular una planificación de trayectoria de una posición a otra, aunque falto mucho para simular algo de la realidad, ya que la idea era poder comunicar todo y poder tener una noción de como podría llevarse a cabo en un entorno más real.

Algo más apegado a la realidad seria como por ejemplo:

- En tiempo real, haciendo uso de sensores, cámaras, etc., planificar una trayectoria, ya que esto nos permite poder detectar obstáculos y actualizar la trayectoria del robot en tiempo real.
- Que la retroalimentación publicara comandos de velocidad para algunos motores (tal vez para los motores que hacen girar las ruedas de algún robot).

Esta práctica hace énfasis en el uso de acciones, ya que es un tema nuevo y bastante importante de conocer y de cómo implementarlo.

Trabajo en ROS

NOTA: Utilizaremos la misma imagen Docker con la que hemos estado trabajando. Los paquetes que ya hemos creado no los tocaremos mas, solo crearemos otros paquetes para la practica.

Las instucciones para levantar un contenedor y guardar los cambios realizados dentro de un contenedor ya se han explicado anteriormente, para que se tomen en cuenta.

Empezaremos creando dos paquetes ROS de la siguiente manera (en el orden en el que se encuentran):

NOTA: Nos ubicamos en \$ROS_WORKSPACE/src

- **practice_msgs**

Este paquete contendrá todo lo relacionado a mensajes (así como de servicios y de acciones)

Comando para la creación del paquete:

```
catkin_create_pkg practice_msgs rospy std_msgs geometry_msgs  
sensor_msgs actionlib_msgs message_generation
```

- **practice**

Este paquete contendrá todo lo relacionado a scripts de Python, archivos de lanzamiento y archivos de configuración.

Comando para la creación del paquete:

```
catkin_create_pkg practice rospy std_msgs message_generation  
ros_tcp_endpoint practice_msgs
```

Notar que el paquete “practice” depende del paquete “practice_msgs” que

acabamos de crear (asi como tambien del paquete “ros_tcp_endpoint” y otros).

Es importante mencionar que ahora si se estan incluyendo todas las dependencias que necesita cada paquete (algo que debemos tomar en cuenta si es que no deseamos modificar tanto los archivos “CMakeLists.txt” y “package.xml”).

Siempre que creemos paquetes tendremos que hacer algunas modificaciones en los archivos “CMakeLists.txt” y “package.xml”.

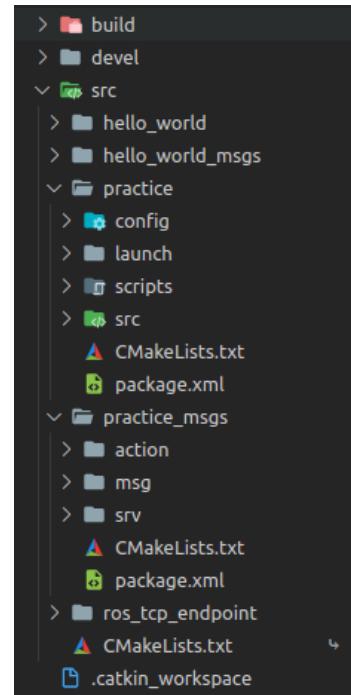
El comando para la creacion de paquetes tiene la siguiente sintaxis:

```
catkin_create_pkg <nombre-paquete> <dependencia1> <dependencia2> ...
```

Nuestra estructura de archivos tenemos que tenerla de la siguiente manera:

NOTA: Agunas carpetas se tienen que crear.

- practice
 - config (carpeta vacia)
 - launch (carpeta vacia)
 - scripts (carpeta vacia)
 - src (carpeta vacia)
 - **CMakeLists.txt**
 - **package.xml**
- practice_msgs
 - action (carpeta vacia)
 - msg (carpeta vacia)
 - srv (carpeta vacia)
 - **CMakeLists.txt**
 - **package.xml**



Mas adelante poblaremos estas carpetas con archivos que iremos creando.

NOTA: Debemos asegurarnos de que todo archivo que creemos tenga permisos de ejecucion, de lo contrario, debemos agregar permisos de ejecucion. Para esto se debe ejecutar el siguiente comando:

```
chmod +x <archivo>
```

Empezaremos creando las declaraciones de mensaje que necesitaremos para la practica. Según el tipo de declaración de mensaje, tenemos:

NOTA: Las declaraciones de mensaje se crearan dentro del paquete “practice_msgs” en sus respectivas carpetas con los nombres que se indican.

- action

- **MoveCube.action**

Declaracion de accion para la tarea de mover un cubo de una posicion a otra (simulando una planificacion de trayectoria).

Esta declaracion de accion contiene lo siguiente:

```
practice_msgs/PoseOriginTarget pose_origin_target
# goal message
---
string result_message
# result message
---
geometry_msgs/Pose current_pose
# feedback message
```

Donde:

- Mensaje de objetivo
 - *pose_origin_target*: Se indica la posicion origen y la posicion objetivo del cubo (el tipo de mensaje es un tipo de mensaje personalizado).
- Mensaje de resultado
 - *result_message*: Mensaje para indicar de que se ha terminado de ejecutar la accion.
- Mensaje de retroalimentacion.
 - *current_pose*: Se indica la posicion actual del cubo mientras se procesa la accion de mover al cubo de la posicion origen a la posicion objetivo (posicion del cubo en la que deberia de estar en cierto momento).
- msg
 - **PoseOriginTarget.msg**

Declaracion de mensaje para la declaracion de servicio “CallMoveCubeAS.srv” y para la declaracion de accion “MoveCube.action”.

Esta declaracion de mensaje contiene lo siguiente:

geometry_msgs/Pose pose_origin
geometry_msgs/Pose pose_target

Donde:

- *pose_origin*: Posicion origen del cubo.
- *pose_target*: Posicion objetivo del cubo.

- **srv**

- **CallMoveCubeAS.srv**

Declaracion de servicio para llamar al servidor de accion “MoveCubeAS” para que procese el objetivo (mover un cubo de una posicion origen a una posicion objetivo).

Esta declaracion de servicio contiene lo siguiente:

practice_msgs/PoseOriginTarget pose_origin_target

bool success
string message

Donde:

- Mensaje de solicitud
 - *pose_origin_target*: Se indica la posicion origen y la posicion objetivo del cubo (el tipo de mensaje es un tipo de mensaje personalizado).
- Mensaje de respuesta
 - *success*: Indica si se ha llamado con exito al servidor de accion.
 - *message*: Algun mensaje por parte del servicio.

- **ConvertMetresToFeet.srv**

Declaración de servicio para la conversión de metros a pies.

Esta declaracion de servicio contiene lo siguiente:

float64 measurement_metres

float64 measurement_feets
bool success

Donde:

- Mensaje de solicitud
 - *measurement_metres*: Medida en metros.

- Mensaje de respuesta
 - *measurement_feets*: Medida en pies.
 - *success*: Indica si la conversion ha sido exitosa.
- **HeightTest.srv**
 Declaración de servicio para la prueba de altura de los cubos (indica si un cubo con cierta altura pasa la prueba o no).
 Esta declaración de servicio contiene lo siguiente:

```
float32 height_cube_mts
---
bool result
```

Donde:

- Mensaje de solicitud
 - *height_cube_mts*: Altura del cubo en metros.
- Mensaje de respuesta
 - *result*: Indica si el cubo ha pasado la prueba o no.

Antes de ejecutar el comando “catkin_make” necesitamos hacer unas modificaciones a los archivos “CMakeLists.txt” y “package.xml” en el paquete “practice_msgs”:

```
● CMakeLists.txt
add_message_files(
  DIRECTORY msg
)

add_service_files(
  DIRECTORY srv
)

add_action_files(
  DIRECTORY action
)

generate_messages(
  DEPENDENCIES
  actionlib_msgs
```

```

geometry_msgs
sensor_msgs
std_msgs
)

catkin_package(
    CATKIN_DEPENDS message_runtime actionlib_msgs geometry_msgs
    rospy sensor_msgs std_msgs
)
● package.xml
Agregar:
<exec_depend>message_runtime</exec_depend>

```

Por ultimo ejecutamos el comando “`catkin_make`” estando ubicados en el directorio de trabajo para que se vean reflejadas las declaraciones de mensajes:

```

cd $ROS_WORKSPACE
catkin_make --force-cmake

```

NOTA: Es importante agregar la bandera `--force-cmake` al comando `catkin_make` porque sino no genera los mensajes para las declaraciones de mensajes.

Resultado:

```

[ 69%] Built target hello_world_msgs_generate_messages
[ 70%] Generating C++ code from practice_msgs/MoveCubeResult.msg
[ 71%] Generating EusLisp code from practice_msgs/MoveCubeActionFeedback.msg
[ 72%] Generating EusLisp code from practice_msgs/PoseOriginTarget.msg
[ 73%] Generating C++ code from practice_msgs/MoveCubeFeedback.msg
[ 74%] Generating EusLisp code from practice_msgs/MoveCubeResult.msg
[ 75%] Generating EusLisp code from practice_msgs/MoveCubeFeedback.msg
[ 75%] Generating EusLisp code from practice_msgs/MoveCubeActionResult.msg
[ 76%] Generating Python from MSG practice_msgs/MoveCubeActionFeedback
[ 78%] Generating EusLisp code from practice_msgs/MoveCubeAction.msg
[ 79%] Generating C++ code from practice_msgs/MoveCubeActionResult.msg
[ 80%] Generating EusLisp code from practice_msgs/MoveCubeActionGoal.msg
[ 81%] Generating EusLisp code from practice_msgs/ConvertMetresToFeet.srv
[ 82%] Generating C++ code from practice_msgs/MoveCubeAction.msg
[ 83%] Generating EusLisp code from practice_msgs/HeightTest.srv
[ 84%] Generating EusLisp code from practice_msgs/CallMoveCubeAS.srv
[ 85%] Generating EusLisp manifest code for practice_msgs
[ 86%] Generating C++ code from practice_msgs/MoveCubeActionGoal.msg
[ 87%] Generating Python from MSG practice_msgs/PoseOriginTarget
[ 89%] Generating Python from MSG practice_msgs/MoveCubeResult
[ 90%] Generating C++ code from practice_msgs/ConvertMetresToFeet.srv
[ 91%] Generating Python from MSG practice_msgs/MoveCubeFeedback
[ 92%] Generating Python from MSG practice_msgs/MoveCubeActionResult
[ 92%] Built target practice_msgs_generate_messages_eus
[ 92%] Generating C++ code from practice_msgs/HeightTest.srv
[ 92%] Generating Python from MSG practice_msgs/MoveCubeAction
[ 93%] Generating Python from MSG practice_msgs/MoveCubeActionGoal
[ 94%] Generating Python code from SRV practice_msgs/ConvertMetresToFeet
[ 95%] Generating C++ code from practice_msgs/CallMoveCubeAS.srv
[ 96%] Generating Python code from SRV practice_msgs/HeightTest
[ 97%] Generating Python code from SRV practice_msgs/CallMoveCubeAS
[ 98%] Generating Python msg __init__.py for practice_msgs
[100%] Generating Python srv __init__.py for practice_msgs
[100%] Built target practice_msgs_generate_messages_cpp
[100%] Built target practice_msgs_generate_messages_py
Scanning dependencies of target practice_msgs_generate_messages
[100%] Built target practice_msgs_generate_messages
root@b58d0d819126:/catkin_ws# 
```

```

root@b58d0d819126:/catkin_ws# rosmsg list | grep practice_msgs
practice_msgs/MoveCubeAction
practice_msgs/MoveCubeActionFeedback
practice_msgs/MoveCubeActionGoal
practice_msgs/MoveCubeActionResult
practice_msgs/MoveCubeFeedback
practice_msgs/MoveCubeGoal
practice_msgs/MoveCubeResult
practice_msgs/PoseOriginTarget
root@b58d0d819126:/catkin_ws# 
```

```

root@b58d0d819126:/catkin_ws# rossrv list | grep practice_msgs
practice_msgs/CallMoveCubeAS
practice_msgs/ConvertMetresToFeet
practice_msgs/HeightTest
root@b58d0d819126:/catkin_ws# 
```

Ahora creamos los siguientes scripts que necesitaremos para la practica:

Los scripts estaran ubicados en el paquete “practice” en la carpeta “scripts”.

- **convert_metres_to_feet_server.py**

Declara un servidor de servicio para la conversion de metros a pies de una cantidad.

- Como mensaje de solicitud se especifica la cantidad en metros.
- Como mensaje de respuesta se devuelve la cantidad convertida a pies.

Definimos una clase que hara lo siguiente:

- *Constructor():*

Se inicializa un nodo ROS y se declara una variable para el factor de conversion de metros a pies.

- *Métodos:*

- *createService(<nombre del servidor de servicio>)*

Crea un servidor de servicio que convierte una cantidad en metros a una cantidad en pies.

- *__handler(<solicitud>)*

Funcion que maneja las solicitudes de los clientes y devuelve como respuesta la conversion de metros a pies de la cantidad recibida del cliente.

Funcion principal:

- *main():*

Se crea una instancia de la clase y se crea el servidor de servicio (manteniendose ejecutando hasta que el nodo se apague).

- **convert_metres_to_feet_client_test.py (script de prueba para probar el script “convert_metres_to_feet_server.py”)**

Actua como cliente del servidor de servicio que realiza la conversion de metros a pies de una cantidad.

Definimos una clase que hara lo siguiente:

- *Constructor():*

No se hace nada aqui.

- *Métodos:*

- *callService(<nombre del servidor de servicio>,<cantidad en metros>)*

Llama al servidor de servicio y devuelve la respuesta por parte del servicio.

Funcion principal:

- *main():*

Se crea una instancia de la clase, se obtiene la cantidad en metros de la lista de argumentos (que se pasan a la hora de ejecutar el script) y se realiza la tarea de llamar al servidor de servicio.

- **height_test_server.py**

Declara un servidor de servicio que realiza una prueba de altura de cubos (devuelve verdadero o falso sobre si un cubo pasa la prueba de altura o no).

- Como mensaje de solicitud se especifica la altura del cubo en metros.
- Como mensaje de respuesta se devuelve la respuesta de la prueba de altura (si paso o no la prueba).

Definimos una clase que hara lo siguiente:

- *Constructor():*

Se inicializa un nodo ROS y se declara una variable con el nombre del servidor de servicio para la conversion de metros a pies de una cantidad.

- *Metodos:*

- *createService(<nombre del servidor de servicio>)*

Crea un servidor de servicio que realiza una prueba de altura de cubos.

- *__handler(<solicitud>)*

Funcion que maneja las solicitudes de los clientes y devuelve como respuesta el resultado de la prueba de altura del cubo.

- *__callServiceConvertMetresToFeet(<numero hasta el cual contar>)*

Llama al servidor de servicio para la conversion de metros a pies de la altura del cubo.

Funcion principal:

- *main():*

Se crea una instancia de la clase y se crea el servidor de servicio (manteniendose ejecutando hasta que el nodo se apague).

- **height_test_client_test.py (script de prueba para probar el script "height_test_server.py")**

Actua como cliente del servidor de servicio que realiza la prueba de altura de cubos.

Definimos una clase que hara lo siguiente:

- *Constructor():*

No se hace nada aqui.

- *Metodos:*

- *callService(<nombre del servidor de servicio>,<altura del cubo en metros>)*

Llama al servidor de servicio y devuelve la respuesta por parte del servicio.

Funcion principal:

- *main():*

Se crea una instancia de la clase, se obtiene la altura del cubo en metros de la lista de argumentos (que se pasan a la hora de ejecutar el script) y se realiza la tarea de llamar al servidor de servicio.

- **move_cube_as.py**

Declara un servidor de acción que realiza una planificación de trayectoria de una posición a otra.

- Como mensaje de objetivo se especifica la posición actual y la posición objetivo (por medio de un tipo de mensaje personalizado).
- Como mensaje de resultado se devuelve un mensaje de que ya se ha completado de ejecutar la acción.
- Como mensaje de retroalimentación se publica en un tema la posición actual en el recorrido de la posición de origen a la posición de destino (otros nodos pueden suscribirse en este tema).

Definimos una clase que hará lo siguiente:

- *Constructor():*

Se inicializa un nodo ROS y se inicializan dos variables para los tipos de mensaje de resultado y retroalimentación.

- *Métodos:*

- *createSimpleActionServer(<nombre del servidor de accion>)*

Crea un servidor de acción simple y especifica la función que estará manejando las solicitudes de objetivo, publicando la retroalimentación y, al final, devolviendo un resultado (respuesta).

- *__execute_cb(<objetivo>)*

Funcion que maneja las solicitudes de objetivo, lleva a cabo dicha accion, publica la retroalimentacion mientras se lleva a cabo la accion y, al final, devuelve un resultado.

Funcion principal:

- *main():*

Se crea una instancia de la clase y se crea el servidor de acción simple (manteniéndose ejecutando hasta que el nodo se apague).

- **move_cube_ac_test.py (script de prueba para probar el script "move_cube_as.py")**

Hace uso del servidor de acción que realiza una planificación de trayectoria de una posición a otra (actúa como un cliente de acción).

Como ejemplo se establece una posición origen en [0,0,0] y una posición destino en [10,0,0] (donde: [x,y,z]).

Definimos una clase que hará lo siguiente:

■ *Constructor():*

Se inicializa un nodo ROS y se declara una variable que especifica el nombre del tema para la retroalimentación.

■ *Métodos:*

- *createSimpleActionClient(<nombre del servidor de acción>)*

Crea un cliente de acción simple.

- *sendGoal(<Objetivo>)*

Envía un objetivo (una posición origen y una posición destino) al servidor de acción, se suscribe al tema de retroalimentación y espera la respuesta de la acción.

- *__subscribe_feedback()*

Se suscribe al tema de la retroalimentación y se especifica la función de devolución de llamada donde se estarán recibiendo los datos de dicho tema.

- *__callback_feedback()*

Función de devolución de llamada donde se reciben los datos de dicho tema.

Función principal:

■ *main():*

Se crea una instancia de la clase, se crea el cliente de acción simple, se realiza la tarea de enviar el objetivo al servidor de acción y se espera a la respuesta.

● **call_move_cube_as_server.py**

Declara un servidor de servicio que hace una llamada al servidor de acción que realiza una planificación de trayectoria de una posición a otra.

- Como mensaje de solicitud se especifica la posición actual y la posición objetivo (por medio de un tipo de mensaje personalizado).
- Como mensaje de respuesta se devuelve una respuesta de si ha sido exitosa la llamada al servicio y también devuelve un mensaje de respuesta por parte del servicio.

Definimos una clase que hará lo siguiente:

- *Constructor():*
Se inicializa un nodo ROS.
- *Métodos:*
 - *createSimpleActionClient(<nombre del servidor de acción>)*
Crea un cliente de acción simple (actuara como cliente del servidor de acción simple que realiza una planificación de trayectoria de una posición a otra).
 - *createService(<nombre del servidor de servicio>)*
Crea un servidor de servicio que realiza una llamada a un servidor de acción simple.
 - *_sendGoal(<Objetivo>)*
Envía un objetivo (una posición origen y una posición destino) al servidor de acción.
 - *_handler(<solicitud>)*
Función que maneja las solicitudes de los clientes y devuelve como respuesta si se ha llamado correctamente al servidor de acción y también devuelve un mensaje por parte del servicio.

Función principal:

- *main():*
Se crea una instancia de la clase, se crea un cliente de acción simple y se crea el servidor de servicio (manteniéndose ejecutando hasta que el nodo se apague).
- ***call_move_cube_as_client_test.py (script de prueba para probar el script “call_move_cube_as_server.py”)***
Actúa como cliente del servidor de servicio que realiza una llamada a un servidor de acción.

Definimos una clase que hará lo siguiente:

- *Constructor():*
No se hace nada aquí.
- *Métodos:*
 - *callService(<nombre del servidor de servicio>)*
Llama al servidor de servicio y devuelve la respuesta por parte del servicio.

Función principal:

- *main():*
Se crea una instancia de la clase y se realiza la tarea de llamar al servidor de

servicio.

NOTA: Para cada script que ejecuta un servidor (ya sea de servicio o de acción) se crea un script que ejecuta un cliente para poder probar que todo funcione correctamente (de esta manera nos aseguramos de que no tendremos inconvenientes a la hora de comunicar Unity con ROS).

Resultados:

Ejecutamos solo una vez el siguiente comando:

```
roscore &
```

- Conversion de metros a pies

Comando (se especifica la cantidad en metros):

```
rosrun practice convert_metres_to_feet_server.py &
rosrun practice convert_metres_to_feet_client_test.py 20
rosrun practice convert_metres_to_feet_client_test.py 10
rosrun practice convert_metres_to_feet_client_test.py 1
```

Imagenes:

```
root@f0eca2d7ac38:/catkin_ws# rosrun practice convert_metres_to_feet_server.py &
[2] 51492
root@f0eca2d7ac38:/catkin_ws# [INFO] [165133302.259222]: 
Service 'convert_metres_to_feet' created

root@f0eca2d7ac38:/catkin_ws# rosrun practice convert_metres_to_feet_client_test.py 20
Response: 65.62 ft
root@f0eca2d7ac38:/catkin_ws# rosrun practice convert_metres_to_feet_client_test.py 10
Response: 32.81 ft
root@f0eca2d7ac38:/catkin_ws# rosrun practice convert_metres_to_feet_client_test.py 1
Response: 3.281 ft
root@f0eca2d7ac38:/catkin_ws# █
```

- Prueba de altura

Requisitos:

- Que un nodo este ejecutando el script
“convert_metres_to_feet_server.py”

Si la altura es mayor o igual a 6.56 ft (aprox 2.0 mts) no pasa la prueba, de lo contrario, si pasa la prueba.

Comandos (se especifica la altura en metros):

```
rosrun practice height_test_server.py &
rosrun practice height_test_client_test.py 2
rosrun practice height_test_client_test.py 3
```

```
rosrun practice height_test_client_test.py 1  
rosrun practice height_test_client_test.py 1.5
```

Imagenes:

```
root@f0eca2d7ac38:/catkin_ws# rosrun practice height_test_server.py &  
[3] 53919  
root@f0eca2d7ac38:/catkin_ws# [INFO] [1651333481.494756]:  
Service 'height_test' created  
  
root@f0eca2d7ac38:/catkin_ws# rosrun practice height_test_client_test.py 2  
[INFO] [1651333802.088294]:  
Call made  
Test not passed  
root@f0eca2d7ac38:/catkin_ws# rosrun practice height_test_client_test.py 3  
Test not passed  
root@f0eca2d7ac38:/catkin_ws# rosrun practice height_test_client_test.py 1  
Test passed  
root@f0eca2d7ac38:/catkin_ws# rosrun practice height_test_client_test.py 1.5  
Test passed  
root@f0eca2d7ac38:/catkin_ws# █
```

- Accion para la planificacion de una trayectoria de una posicion a otra
Como ejemplo se establece una posicion origen en [0,0,0] y una posicion destino en [10,0,0] (donde: [x,y,z]).
En Unity **x** y **z** especifican adelante/tras izquierda/derecha y **y** especifica arriba/abajo.

Comandos:

```
rosrun practice move_cube_as.py &  
rosrun practice move_cube_ac_test.py
```

Imagenes:

```
root@f0eca2d7ac38:/catkin_ws# rosrun practice move_cube_as.py &
[4] 60358
root@f0eca2d7ac38:/catkin_ws# [INFO] [1651334039.901984]: Simple action server 'move_cube' created

root@f0eca2d7ac38:/catkin_ws# rosrun practice move_cube_ac_test.py
[INFO] [1651334346.389798]: Goal has been sent to the action server
[INFO] [1651334346.391505]:
Subscribing to feedback message
After subscriber
header:
  seq: 1
  stamp:
    secs: 1651334346
    nsecs: 406833887
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1651334346
      nsecs: 389448881
    id: "/move_cube_ac_node-1-1651334346.389"
  status: 1
  text: "This goal has been accepted by the simple action server"
feedback:
  current_pose:
    position:
      x: 2.0
      y: 0.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 0.0
```

```
header:
  seq: 2
  stamp:
    secs: 1651334347
    nsecs: 407737970
  frame_id: ''
status:
  goal_id:
  stamp:
    secs: 1651334346
    nsecs: 389448881
  id: "/move_cube_ac_node-1-1651334346.389"
  status: 1
  text: "This goal has been accepted by the simple action server"
feedback:
  current_pose:
    position:
      x: 4.0
      y: 0.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 0.0
header:
  seq: 3
  stamp:
    secs: 1651334348
    nsecs: 408364057
  frame_id: ''
status:
  goal_id:
  stamp:
    secs: 1651334346
    nsecs: 389448881
  id: "/move_cube_ac_node-1-1651334346.389"
  status: 1
  text: "This goal has been accepted by the simple action server"
feedback:
  current_pose:
    position:
      x: 6.0
      y: 0.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 0.0
```

```

header:
  seq: 4
  stamp:
    secs: 1651334349
    nsecs: 409679889
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1651334346
      nsecs: 389448881
      id: "/move_cube_ac_node-1-1651334346.389"
    status: 1
    text: "This goal has been accepted by the simple action server"
feedback:
  current_pose:
    position:
      x: 8.0
      y: 0.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 0.0
header:
  seq: 5
  stamp:
    secs: 1651334350
    nsecs: 411619901
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1651334346
      nsecs: 389448881
      id: "/move_cube_ac_node-1-1651334346.389"
    status: 1
    text: "This goal has been accepted by the simple action server"
feedback:
  current_pose:
    position:
      x: 10.0
      y: 0.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 0.0
Result message: Succesfully completed move cube
root@f0eca2d7ac38:/catkin_ws# █

```

Como podemos observar la posición en x esta cambiando conforme se

procesa la acción en el servidor de acción, la cual cada vez se acerca más a la posición objetivo.

Este es un ejemplo muy basico, pero abarca muy bien el tema del cliente de accion, la llamada al servidor de accion, el procesamiento de la accion (el cual uno mismo lo programa) y los temas involucrados en un servidor de accion.

- Llamar al servidor de accion que planifica una trayectoria de una posicion a otra
- Requisitos:

- Que un nodo este ejecutando el script “*move_cube_as.py*”

Realiza una llamada al servidor de accion para que procese un objetivo.

En el objetivo se establece una posicion origen en [0,0,0] y una posicion destino en [10,0,0] (donde: [x,y,z]).

Otros nodos pueden suscribirse al tema de retroalimentacion y resultado (que es como se tiene pensado).

Comandos:

```
rosrun practice call_move_cube_as_server.py &
rosrun practice call_move_cube_as_client_test.py
```

Para escuchar lo que se publica en los temas de retroalimentacion y resultado:

```
rostopic echo /move_cube/feedback
rostopic echo /move_cube/result
```

Imagenes:

```
root@5f03eb1ec8be:/catkin_ws# rosrun practice call_move_cube_as_server.py &
[3] 175428
root@5f03eb1ec8be:/catkin_ws# [INFO] [1651431850.512610]:
Service 'call_move_cube as' created
root@5f03eb1ec8be:/catkin_ws# rosrun practice call_move_cube_as_client_test.py
[INFO] [1651431865.606029]: Goal has been sent to the action server
Action server called
root@5f03eb1ec8be:/catkin_ws# rostopic list
/move_cube/cancel
/move_cube/feedback
/move_cube/goal
/move_cube/result
/move_cube/status
/rosout
/rosout_agg
root@5f03eb1ec8be:/catkin_ws#
```

```

root@5f03eb1ec8be:/catkin_ws# rosrun practice call_move_cube_as_client_test.py
[INFO] [1651432314.740251]: Goal has been sent to the action server
Action server called
root@5f03eb1ec8be:/catkin_ws# rostopic echo /move_cube/result
header:
  seq: 5
  stamp:
    secs: 1651432319
    nsecs: 750349044
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1651432314
      nsecs: 739809036
    id: "/call_move_cube_as_service_node-5-1651432314.740"
  status: 3
  text: ''
result:
  result_message: "Successfully completed move cube"
...

```

Podemos lanzar todos los nodos (solo los servidores de servicio y de accion, no los clientes) por medio de un archivo de lanzamiento, de la siguiente manera:

Primero creamos un archivo llamado “params.yaml” en la carpeta “config” del paquete “ros_tcp_endpoint”, el cual contendra lo siguiente:

ROS_IP: 0.0.0.0

Despues creamos un archivo de lanzamiento llamado “practice_nodes.launch” en la carpeta “launch” del paquete “practice”, el cual contendra lo siguiente:

```

<launch>
  <rosparam command="load" file="$(find
    ros_tcp_endpoint)/config/params.yaml" />
  <node name="server_endpoint" pkg="ros_tcp_endpoint"
    type="default_server_endpoint.py" args="--wait"
    output="screen" respawn="true" />
  <node name="convert_metres_to_feet_service_node"
    pkg="practice" type="convert_metres_to_feet_server.py"
    output="screen" respawn="true"/>
  <node name="height_test_service_node" pkg="practice"
    type="height_test_server.py" output="screen"
    respawn="true"/>
  <node name="move_cube_as_node" pkg="practice"
    type="move_cube_as.py" output="screen"

```

```
    respawn="true"/>
<node name="call_move_cube_as_service_node" pkg="practice"
      type="call_move_cube_as_server.py" output="screen"
      respawn="true"/>
</launch>
```

Lo que hacemos es:

- Cargar un archivo de parametros ubicado en la carpeta “config” del paquete “ros_tcp_endpoint”.
- Iniciar el nodo que ejecuta el servidor TCP (muy importante, porque de lo contrario no se podran comunicar Unity y ROS).
- Iniciar los nodos para la practica (respetando los mismos nombres de nodo que se les dio en los scripts).

Por ultimo ejecutamos el siguiente comando para lanzar el archivo de lanzamiento:

```
roslaunch practice practice_nodes.launch &
```

El comando “roslaunch” inicia “roscore” si es que no se ha iniciado.

Resultados:

```

root@5f03eb1ec8be:/catkin_ws# roslaunch practice practice_nodes.launch &
[1] 185698
root@5f03eb1ec8be:/catkin_ws# ... logging to /root/.ros/log/5820685c-c983-11ec-847f-0242ac140002/roslaunch-5f03eb1ec8be-185698.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://5f03eb1ec8be:42881/
SUMMARY
========
PARAMETERS
  * /ROS_IP: 0.0.0.0
  * /rosdistro: melodic
  * /rosversion: 1.14.12

NODES
  /
    call_move_cube_as_service_node (practice/call_move_cube_as_server.py)
    convert_metres_to_feet_service_node (practice/convert_metres_to_feet_server.py)
    height_test_service_node (practice/height_test_server.py)
    move_cube_as_node (practice/move_cube_as.py)
    server_endpoint (ros_tcp_endpoint/default_server_endpoint.py)

auto-starting new master
process[master]: started with pid [185729]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 5820685c-c983-11ec-847f-0242ac140002
process[rosout-1]: started with pid [185766]
started core service [/rosout]
process[server_endpoint-2]: started with pid [185779]
process[convert_metres_to_feet_service_node-3]: started with pid [185793]
[INFO] [1651432654.131940]: Starting server on 0.0.0.0:10000
[INFO] [1651432654.557071]:
Service 'convert_metres_to_feet' created
process[height_test_service_node-4]: started with pid [185815]
[INFO] [1651432655.430537]:
Service 'height_test' created
process[move_cube_as_node-5]: started with pid [185835]
[INFO] [1651432656.285624]:
Simple action server 'move_cube' created
process[call_move_cube_as_service_node-6]: started with pid [185857]
[INFO] [1651432657.118497]:
Service 'call_move_cube_as' created
root@5f03eb1ec8be:/catkin_ws# 

```

```

root@5f03eb1ec8be:/catkin_ws# rosnode list
/call_move_cube_as_service_node
/convert_metres_to_feet_service_node
/height_test_service_node
/move_cube_as_node
/rosout
/server_endpoint
root@5f03eb1ec8be:/catkin_ws# 

```

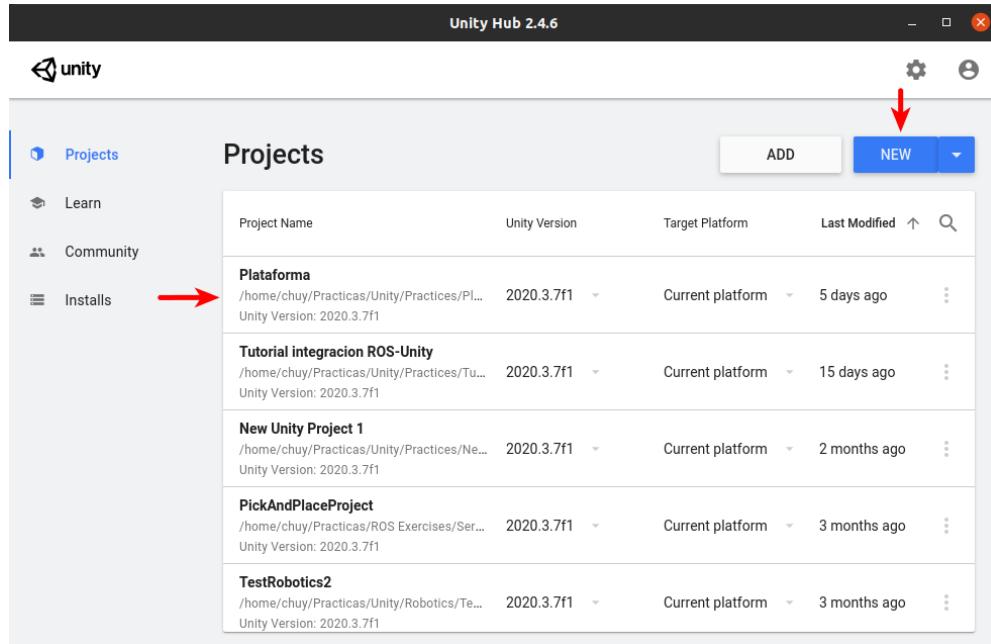
Hasta aquí ya tenemos configurado todo en ROS para la práctica. A continuación continuaremos a configurar Unity para la práctica.

NOTA: Cuando se trabaje en la parte de la integración Unity-ROS, el archivo de lanzamiento deberá lanzarse antes de cualquier simulación en Unity.

Trabajo en Unity

Primeramente, necesitamos tener descargado Unity Hub (en mi caso es la **2.4.6**) y despues instalar una version de Unity (en mi caso es la **2020.3.7f1**).

Una vez descargado Unity Hub e instalado una versión de Unity, creamos un nuevo proyecto con el nombre “Plataforma” y lo abrimos.

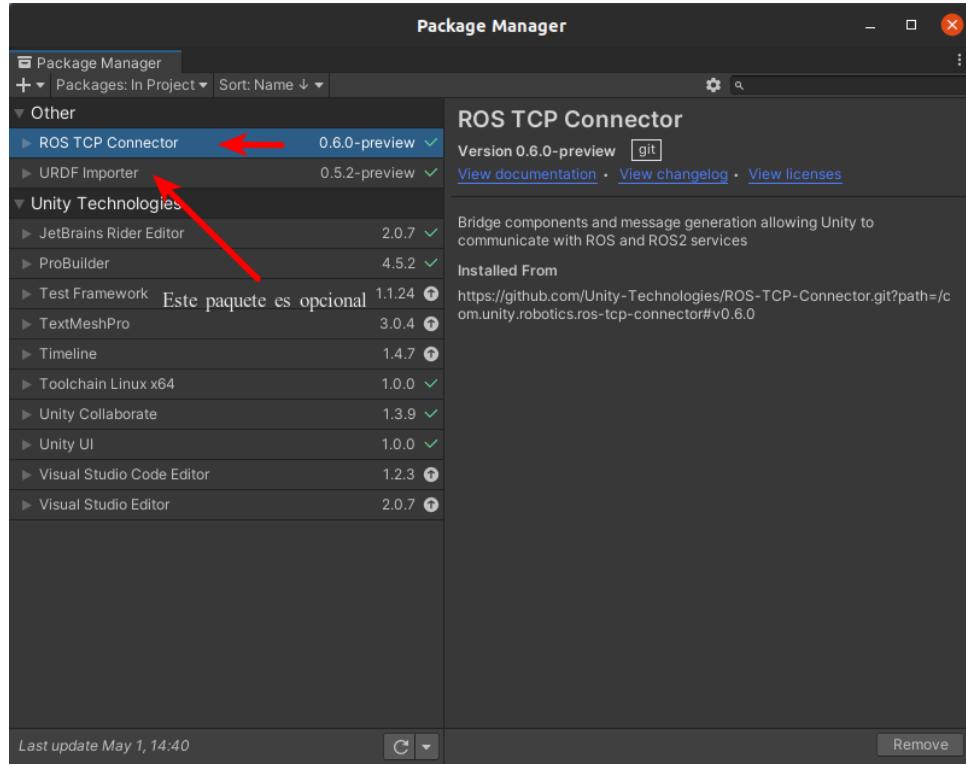


Una vez dentro de Unity, lo primero que haremos sera instalar los paquetes de Unity Robotics.

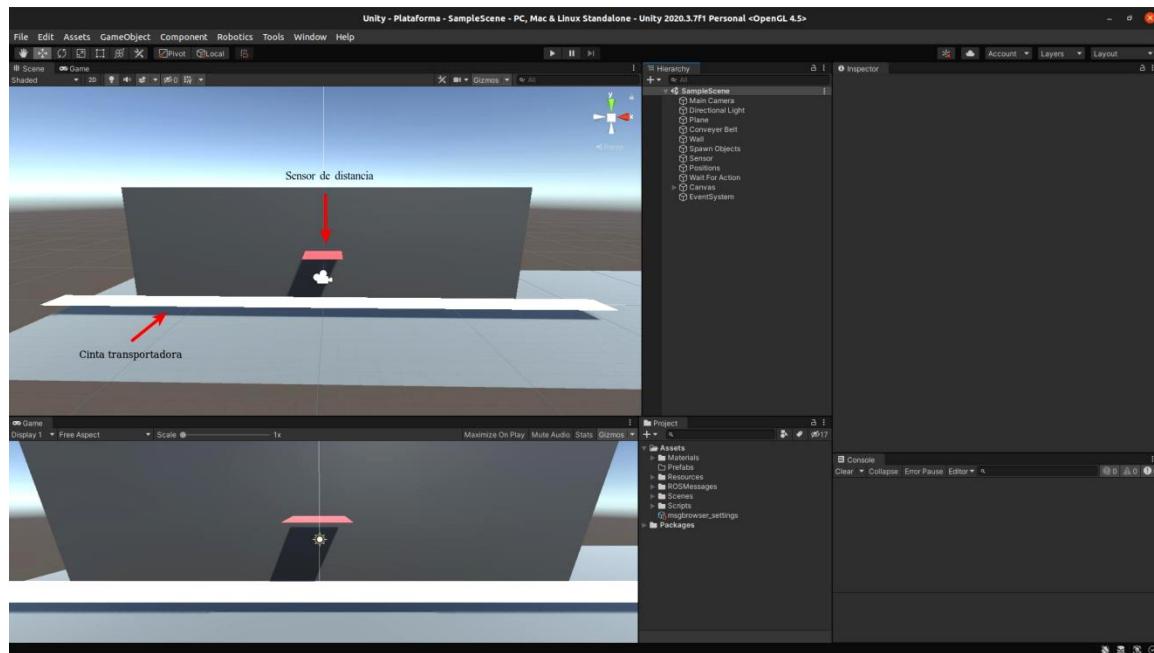
URL:

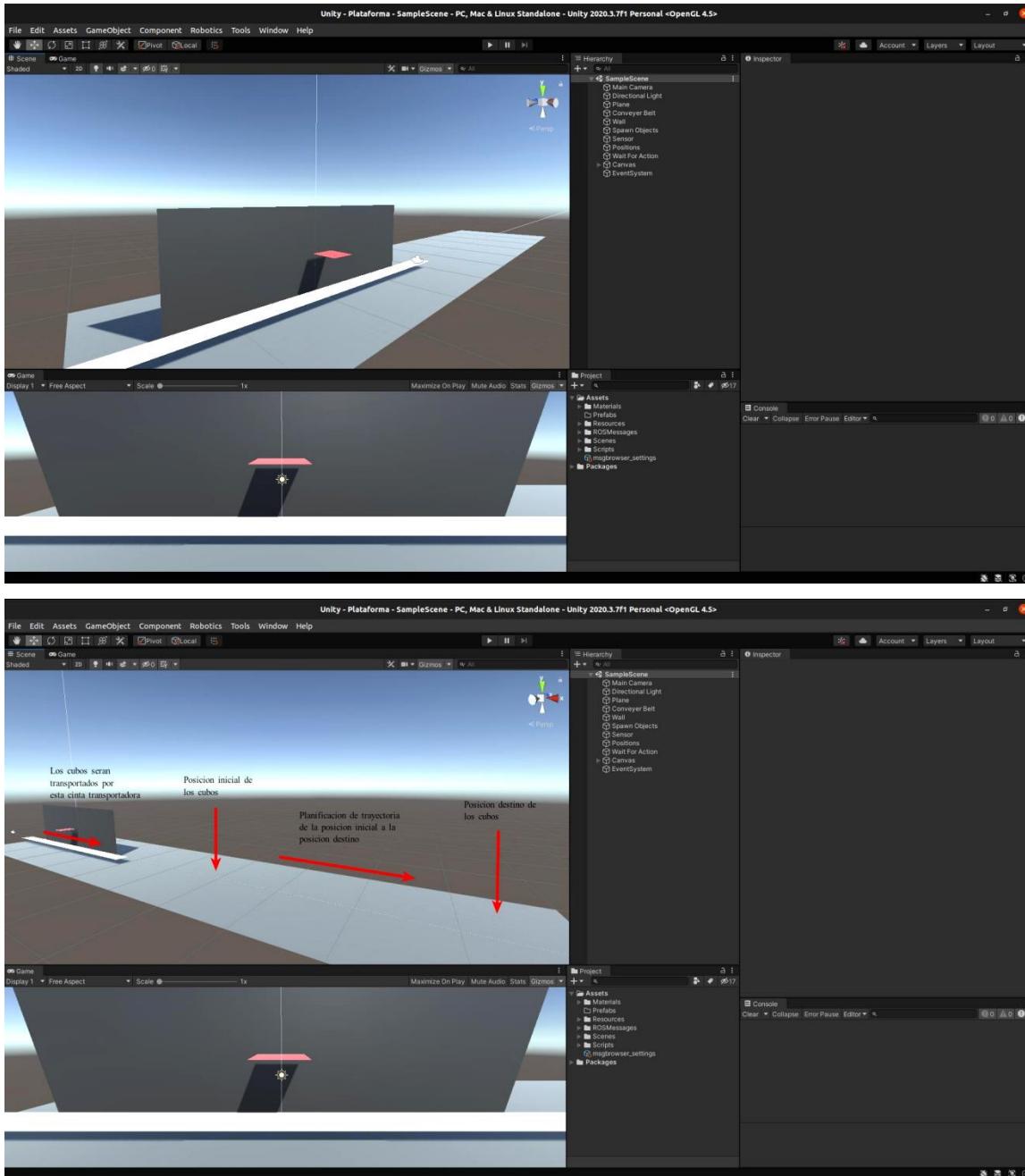
https://github.com/Unity-Technologies/Unity-Robotics-Hub/blob/main/tutorials/quick_setup.md

El paquete “ROS TCP Connector” es el que realmente nos importa.



Ahora, lo que haremos sera crear un escenario como el siguiente:

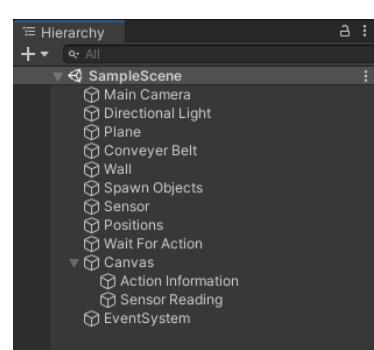


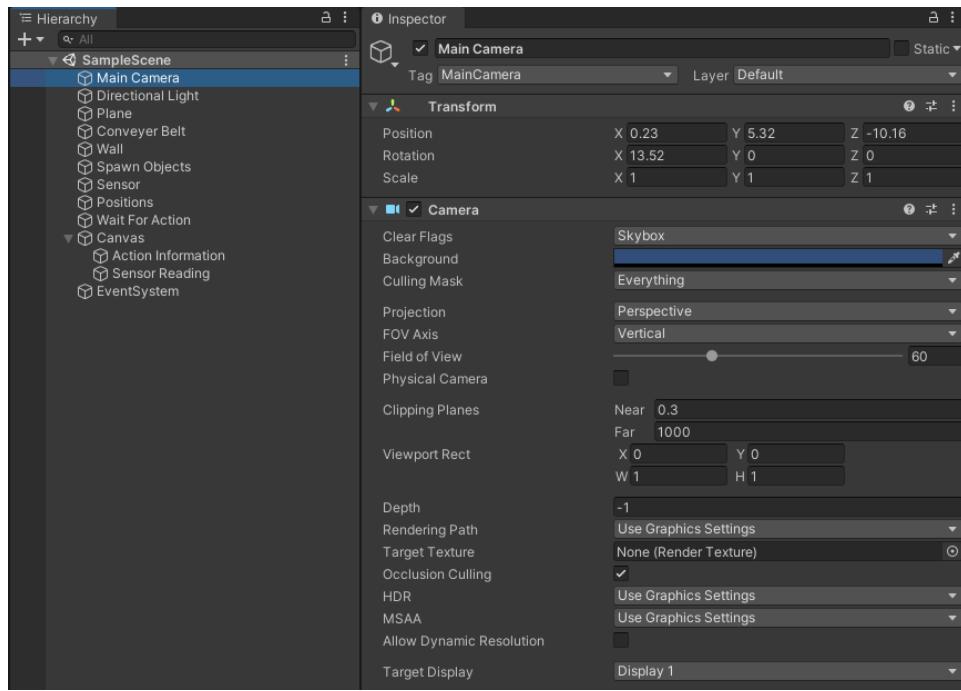


Los GameObjects creados son los siguientes (todos se encuentran en la misma escena que es creada por defecto):

- **Main Camera**

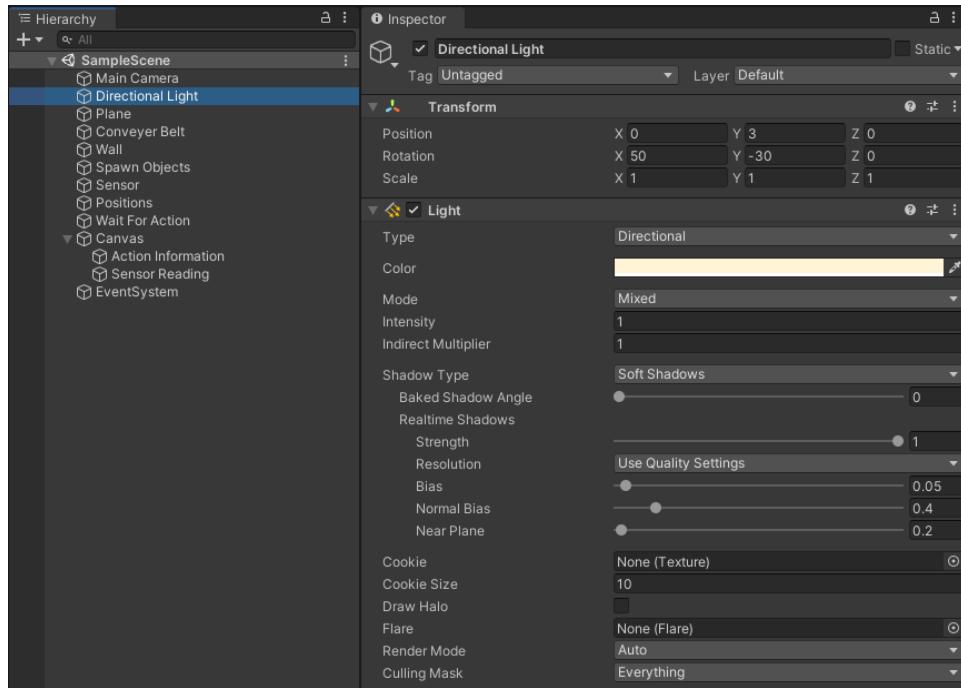
Camara principal de la escena (se crea automaticamente).





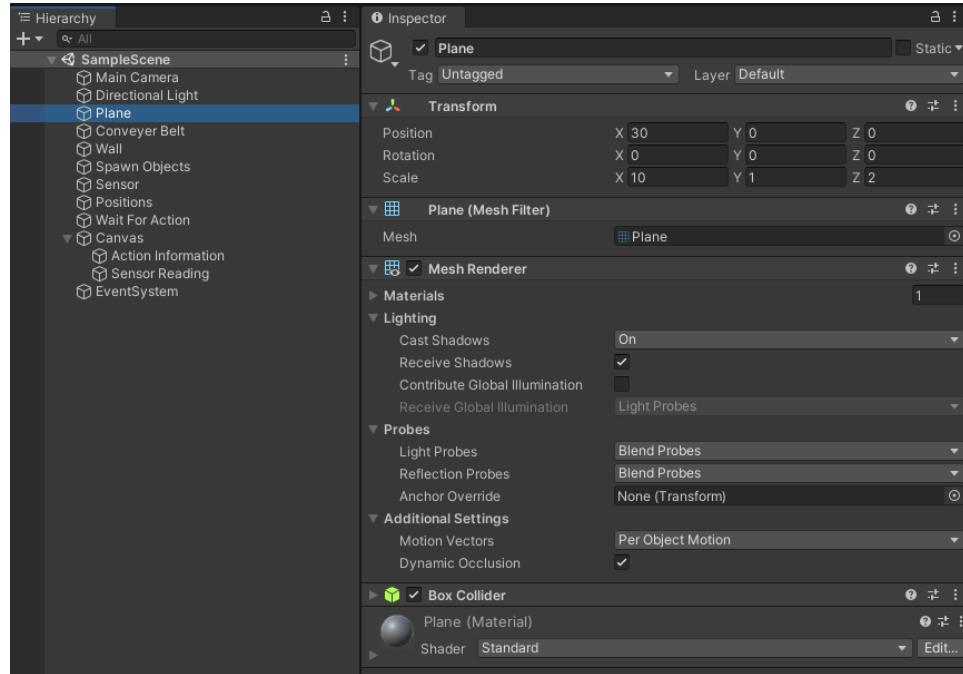
● ***Directional Light***

Luz de la escena (se crea automáticamente).



● ***Plane***

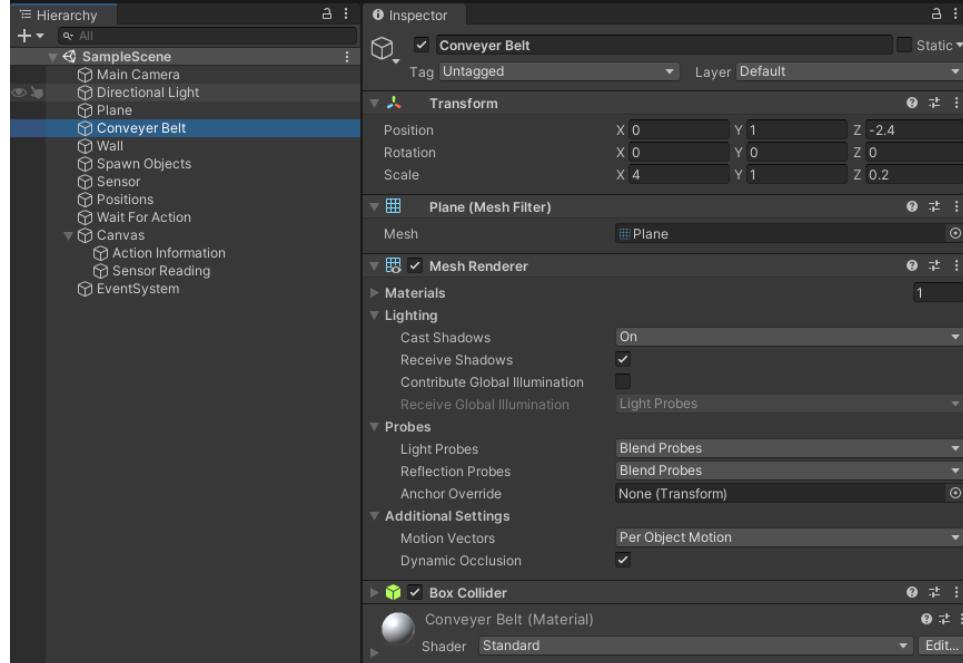
Piso de la escena.



Componentes que se agregaron:

- Box Collider (para detectar colisiones)
- Material (para darle un color)
- **Conveyer Belt**

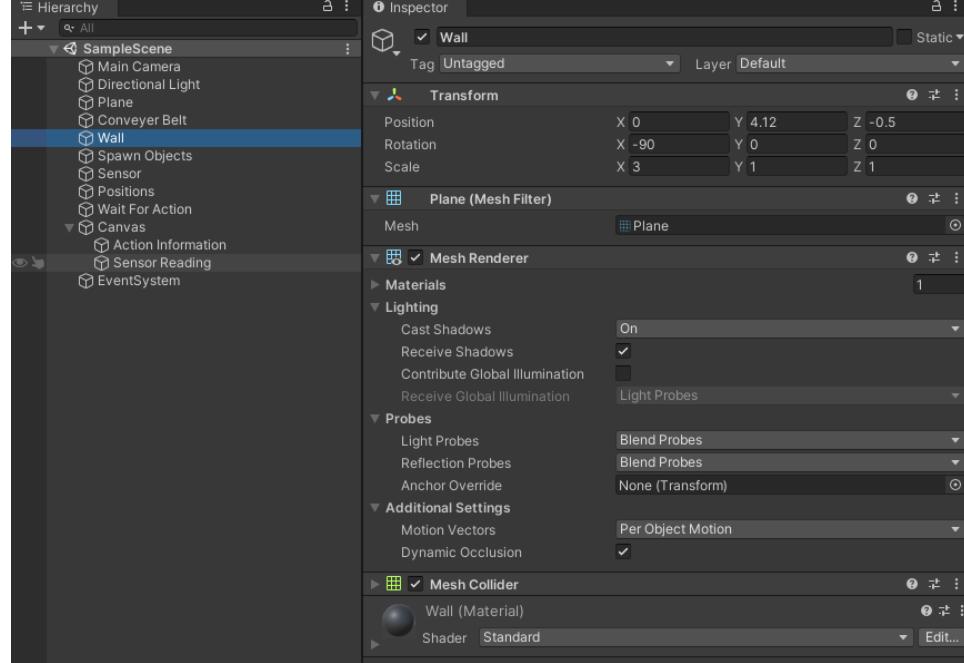
Cinta transportadora.



Componentes que se agregaron:

- Box Collider (para detectar colisiones)
- Material (para darle un color)
- **Wall**

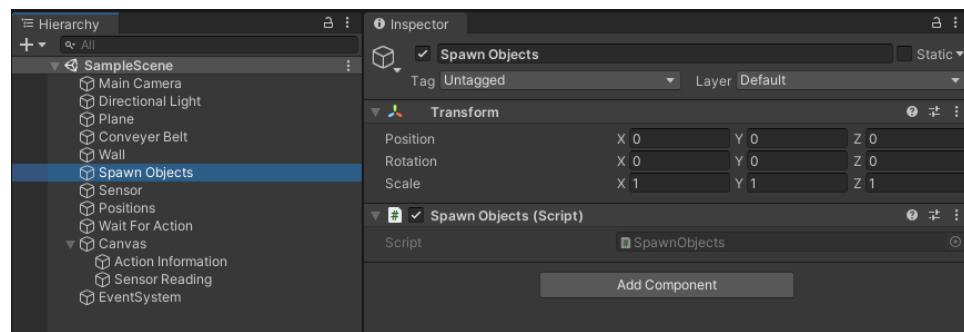
Pared.



Componentes que se agregaron:

- Material (para darle un color)
- **Spawn Objects**

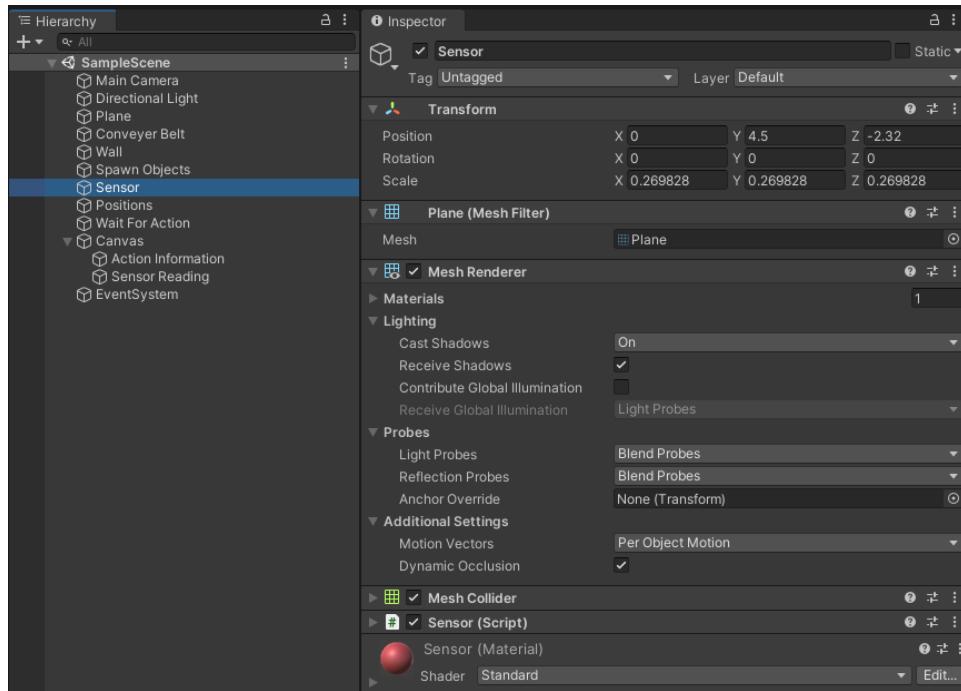
Utilizado para generar objetos (cubos).



Componentes que se agregaron:

- Script C# "SpawnObjects"
- **Sensor**

Utilizado para simular un sensor de distancia.

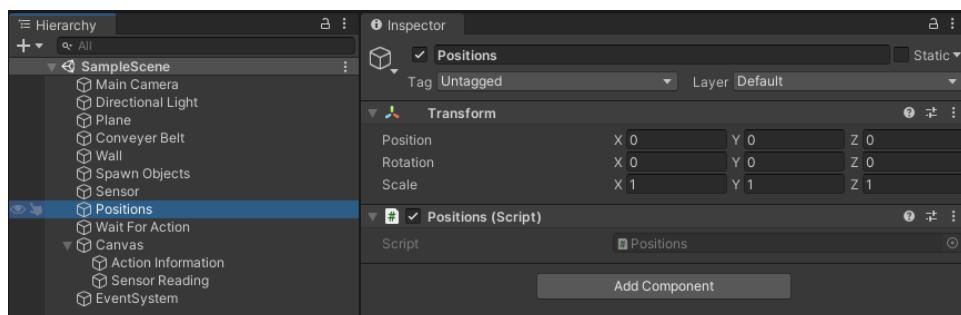


Componentes que se agregaron:

- Script C# “Sensor”
- Material (para darle un color)

● **Positions**

Utilizado para generar las posiciones de origen y de destino de cada uno de los cubos cuando estos son transportados de una posición a otra.



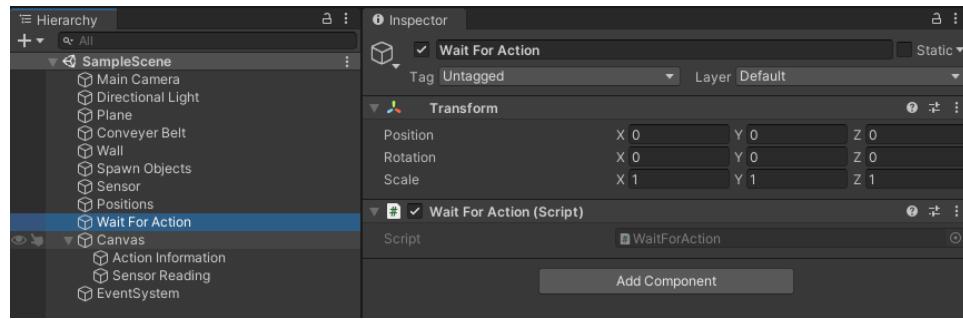
Componentes que se agregaron:

- Script C# “Positions”

● **Wait For Action**

Utilizado para llevar un control en las llamadas al servidor de acción para la planificación de trayectoria, ya que solo se procesara un objetivo a la vez, los demás objetivos a procesar tendrán que esperar a que el servidor de acción termine de ejecutar la acción actual. El método implementado para las llamadas al servidor de acción es de tipo FIFO (First In - First Out, Primero en entrar -

Primero en salir).

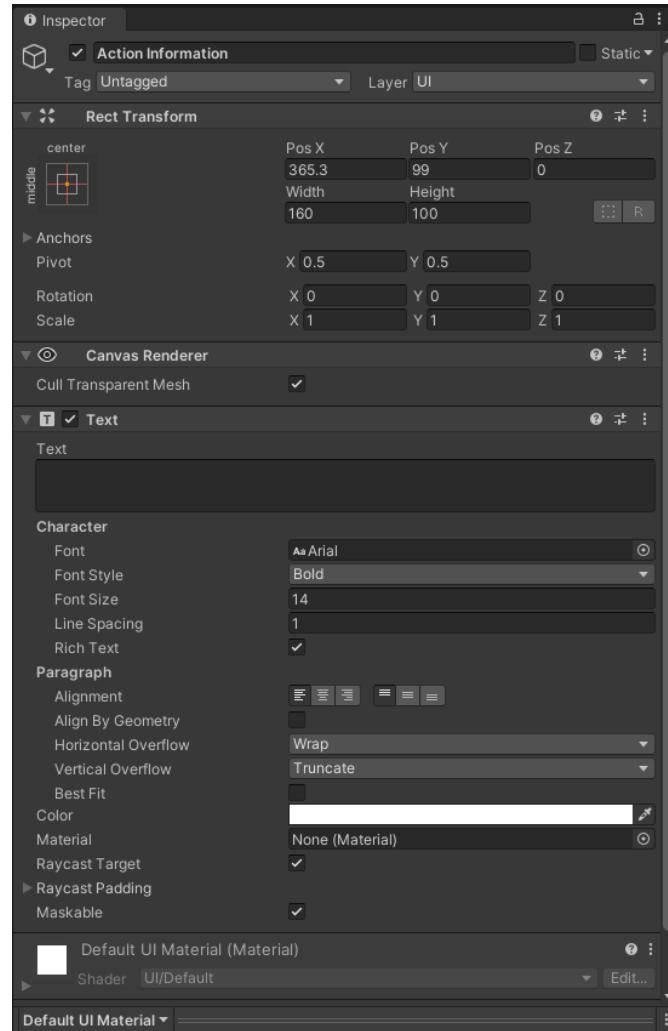


Componentes que se agregaron:

- Script C# "WaitForAction"
- **Canvas (es donde todos los elementos UI deben estar)**

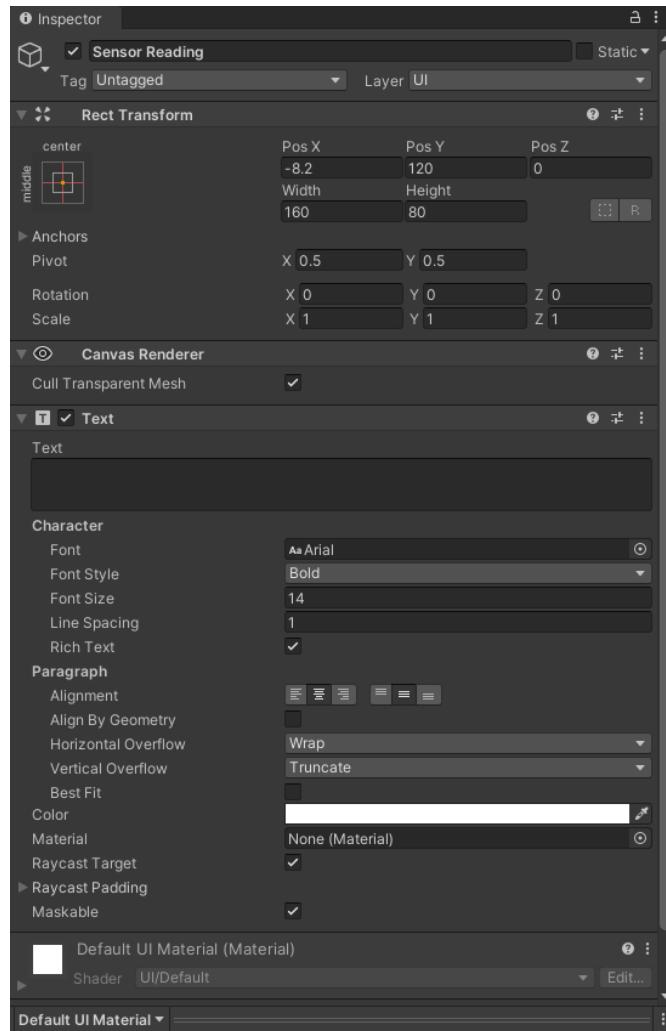
- Action Information (de tipo Text)

Muestra información sobre la retroalimentación del objetivo que está siendo procesada por el servidor de acción.



■ Sensor Reading (de tipo *Text*)

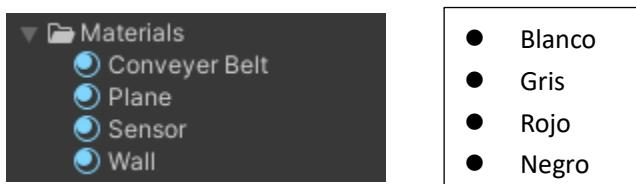
Muestra informacion sobre las lecturas del sensor (altura del cubo) y si han pasado o no la prueba de altura los cubos que pasan por el sensor.



● *Event System*

Es creado automaticamente al crear un Canvas.

Materiales que se crearon (para dar color a los GameObjects):



Declaraciones de mensajes/servicios ROS utilizados para la generacion de scripts:

Las declaraciones de mensajes y servicios se encuentran en la carpeta "ROSMessages" (la cual ya se ha proporcionado) como se muestra a continuacion:

```

chuy@chuy:~/Practicas/ROS Exercises/Servicio Social>Hello (Real) World with ROS - Parte 1$ tree ROSMessages/
ROSMessages/
└── msg
    ├── MoveCubeActionFeedback.msg
    ├── MoveCubeActionResult.msg
    ├── MoveCubeFeedback.nsg
    ├── MoveCubeResult.msg
    └── PoseOriginTarget.msg
└── srv
    ├── CallMoveCubeAS.srv
    └── HeightTest.srv
2 directories, 7 files

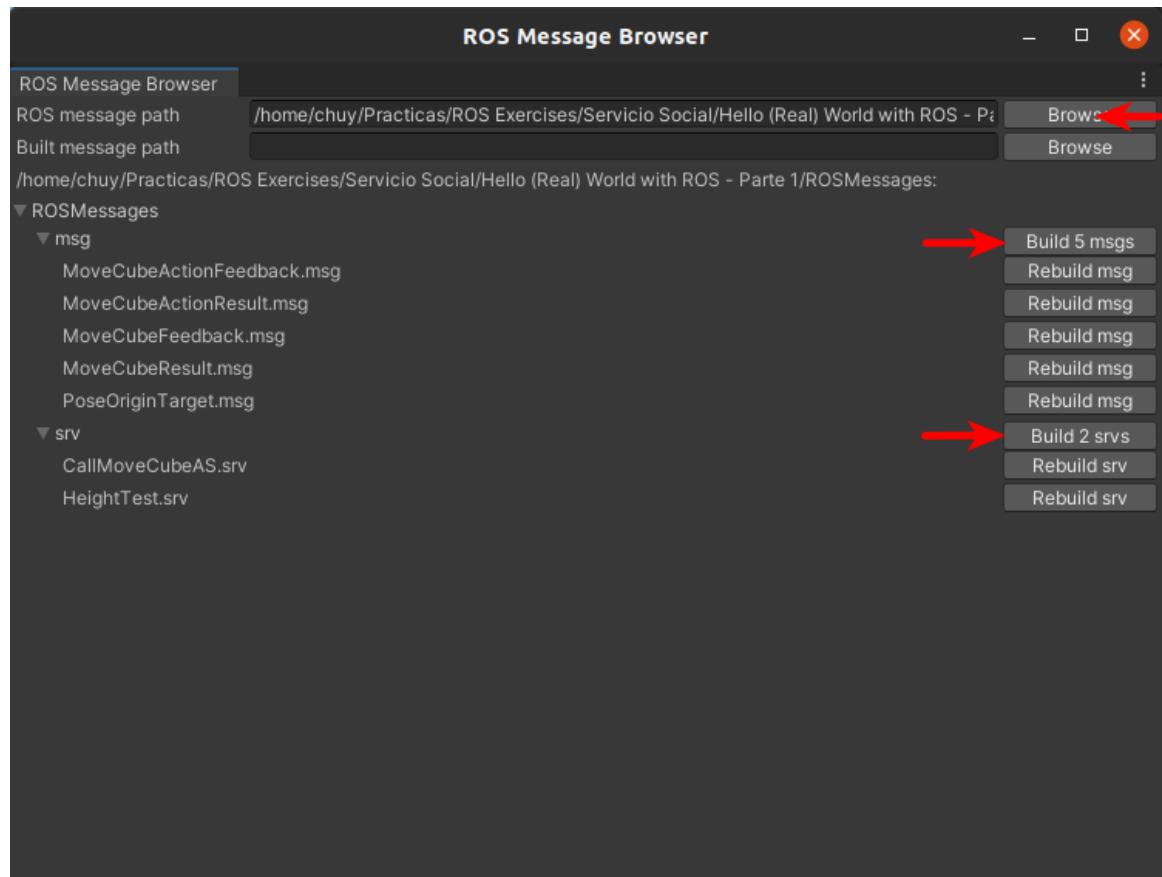
```

Se utilizaran para generar los scripts C# correspondientes.

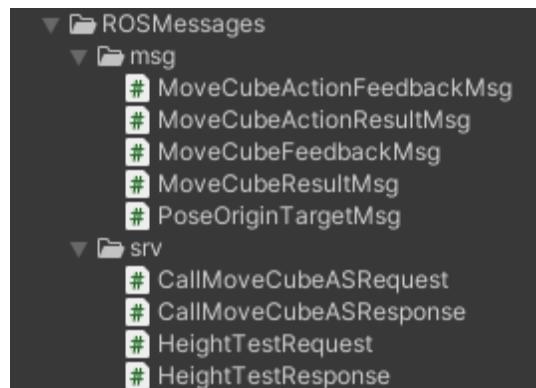
- msg (ubicacion de los scripts generados: Assets/ROSMessages/msg)
 - ***MoveCubeActionFeedback.msg***
Tipo de mensaje donde se publica la retroalimentación del servidor de acción que procesa el objetivo.
 - ***MoveCubeActionResult.msg***
Tipo de mensaje donde se publica el resultado del servidor de acción.
 - ***MoveCubeFeedback.msg***
Tipo de mensaje que es parte del mensaje “MoveCubeActionFeedback.msg”.
 - ***MoveCubeResult.msg***
Tipo de mensaje que es parte del mensaje “MoveCubeActionResult.msg”.
 - ***PoseOriginTarget.msg***
Tipo de mensaje personalizado para especificar la posición origen y la posición destino.
- srv (ubicacion de los scripts generados: Assets/ROSMessages/srv)
 - ***CallMoveCubeAS.srv***
Declaracion de servicio para llamar al servidor de accion que planifica una trayectoria de una posicion origen a una posicion destino.
 - ***HeightTest.srv***
Declaracion de servicio para la prueba de altura de los cubos.

En Unity:

Nos dirigimos a “Robotics -> Generate ROS Messages...”.



Resultado:



Correcciones que se tuvieron que realizar a los scripts generados:

En todos los scripts generados se tuvo que modificar la variable `k_RosMessageName` especificando el nombre del mensaje ROS como esta definido en nuestro entorno de trabajo ROS.

Ya que todos iniciaban con “`ROSMessage/<mensaje>`” y no es correcto (a

menos de que realmente iniciara con el nombre del paquete en donde se tiene dicho mensaje).

- msg (ubicacion de los scripts generados: Assets/ROSMessages/msg)

- **MoveCubeActionFeedbackMsg**

```
public const string k_RosMessageName =  
    "practice_msgs/MoveCubeActionFeedback";
```

- **MoveCubeActionResultMsg**

```
public const string k_RosMessageName =  
    "practice_msgs/MoveCubeActionResult";
```

- **MoveCubeFeedbackMsg**

```
public const string k_RosMessageName =  
    "practice_msgs/MoveCubeFeedback";
```

- **MoveCubeResultMsg**

```
public const string k_RosMessageName =  
    "practice_msgs/MoveCubeResult";
```

- **PoseOriginTargetMsg**

```
public const string k_RosMessageName =  
    "practice_msgs/PoseOriginTarget";
```

- srv (ubicacion de los scripts generados: Assets/ROSMessages/srv)

- **CallMoveCubeASRequest**

```
public const string k_RosMessageName =  
    "practice_msgs/CallMoveCubeAS";
```

- **CallMoveCubeASResponse**

```
public const string k_RosMessageName =  
    "practice_msgs/CallMoveCubeAS";
```

- **HeightTestRequest**

```
public const string k_RosMessageName = "practice_msgs/HeightTest";
```

- **HeightTestResponse**

```
public const string k_RosMessageName = "practice_msgs/HeightTest";
```

Scripts creados (componentes de GameObjects):

El orden en como estan es mas o menos el flujo de trabajo.

- **Positions.cs**

GameObjects que hacen uso de este script:

- Positions (es un GameObject vacio)

Genera en dos matrices las posiciones de inicio y las posiciones de destino de los cubos, es decir:

- Matriz de posiciones de inicio (x,y)

Posiciones donde van a parar los cubos una vez terminen de ser transportados en la cinta transportadora.

- Matriz de posiciones de destino (x,y)

Posiciones donde van a parar los cubos despues de que sean transportados de la posicion de inicio a la posicion destino (la posicion de destino, junto con la posicion de inicio, son las que se le pasan al servidor de accion para que procese el objetivo, el cual es planificar una trayectoria de una posicion a otra).

- ***SpawnObjects.cs***

GameObjects que hacen uso de este script:

- Spawn Objects (es un GameObject vacio)

Genera cubos aleatorias de diferentes alturas cada 2 a 3 segundos de la siguiente manera:

- Se le da un nombre unico (“Cube <numero>”)
- Se le agrega un componente de tipo “RigidBody”
- Se le agrega un componente de tipo “Speed” (script C#)
- Se le da una escala vertical (de altura) aleatoria
- Se le da un color aleatorio al cubo (para que se vean todos diferentes)
- Se posiciona encima de la cinta transportadora
- Cada cubo se almacena en una lista de GameObjects (para llevar un control de lo que se genera)

- ***Speed.cs***

GameObjects que hacen uso de este script:

- Los cubos (solo mientras esten en la cinta transportadora)

Se le da una cierta velocidad al cubo en una direccion (simulando que la cinta transportadora se esta moviendo).

- ***Sensor.cs***

GameObjects que hacen uso de este script:

- Sensor (es el de color rojo, el que esta situado por encima de la cinta transportadora)

Para cada cubo generado, se empieza una corutina que verifica cuando el cubo esta pasando por debajo del sensor para poder tomar la altura del cubo. Una vez se toma la altura del cubo llama al servidor de servicio que ejecuta la prueba de altura, donde:

- Si pasa la prueba, se le agrega un componente de tipo "MoveToPosition" (script C#)
 - Si no pasa la prueba, se le agrega un componente de tipo "DeleteOutOfRange" (script C#)
- ***DeleteOutOfRange.cs***
GameObjects que hacen uso de este script:
 - Los cubos (cuando no pasan la prueba de altura)

Elimina al cubo una vez superado un limite en el eje x.

- ***MoveToPosition.cs***
GameObjects que hacen uso de este script:

- Los cubos (cuando pasan la prueba de altura)

Espera a que el cubo toque el suelo, una vez terminado de ser transportado en la cinta transportadora, y se coloca en una posición inicial (unica para cada cubo, como ya se menciono), y finalmente se inicia una corutina de espera para esperar su turno para ser transportado de la posición inicial a la posición destino utilizando el servidor de acción de planificación de trayectoria.

Cuando se ha completado el tiempo de espera (cuando ya es turno de un cubo), se le agrega un componente de tipo "CallMoveCubeAS" (script C#) y se indica que el servidor de acción esta siendo ocupado.

Es importante mencionar que cuando el cubo toca el suelo se elimina el componente de tipo "Speed" (script C#) para que ya no se siga moviendo el cubo.

- ***WaitForAction.cs***
GameObjects que hacen uso de este script:

- Wait For Action (es un GameObject vacio)

Lleva un control de los turnos para hacer uso del servidor de acción de planificación de trayectoria, es decir:

- Indica si el servidor de acción esta siendo ocupado por algun cubo
- Indica el turno del siguiente cubo una vez el servidor de acción termine

de procesar el objetivo

- ***CallMoveCubeAS.cs***

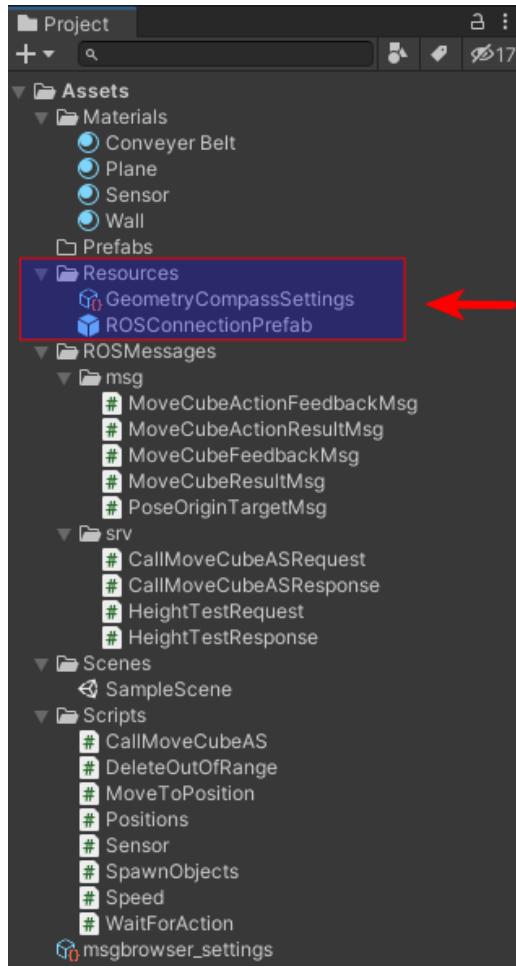
GameObjects que hacen uso de este script:

- Los cubos (cuando han esperado su turno y les toca ser transportados de una posición a otra, haciendo uso del servidor de acción de planificación de trayectorias)

Llama al servidor de servicio que llama al servidor de acción para que procese el objetivo (que es la posición inicial y la posición destino del cubo). Se hace lo siguiente:

- Se llama al servidor de servicio que llama al servidor de acción y se espera una respuesta por parte del servicio
- Si todo ha salido correctamente, entonces:
 - Se suscribe al tema de retroalimentación en donde publica el servidor de acción mientras procesa el objetivo
 - Se suscribe al tema de resultado en donde publica el servidor de acción una vez finalizado el objetivo

Como resultado, nuestro proyecto quedara de la siguiente manera (estructura de carpetas y archivos):



Esta carpeta y archivos se crean en automatico cuando utilizamos por primera vez la clase “ROSConnection”.

Integración ROS-Unity

Ahora integraremos ROS y Unity para realizar la tarea antes mencionada.

La comunicación entre ROS y Unity será la siguiente:

- Se generaran cubos aleatorios (en concreto solo 10) que estarán pasando por la cinta transportadora.
- Un sensor de distancia calculará la altura de los cubos que pasen por debajo de él y dicha información se mandará como solicitud al servidor de servicio que ejecuta la prueba de altura. Donde:
 - Si pasa la prueba, continua y después será transportado de una posición a otra haciendo uso del servidor de acción de planificación de trayectorias.
 - Si no pasa la prueba, continua y después será eliminado.
- Los cubos que pasen la prueba de altura serán posicionados, después de que

toquen el suelo, y mediante turnos (los primeros que llegan son los primeros que harán uso del servidor de acción) harán las llamadas al servidor de acción para la planificación de trayectoria.

- Cada cubo tendrá una posición inicial única y una posición destino única.
- Cada cubo estará suscribiéndose al tema de retroalimentación (donde estará publicando el servidor de acción mientras procesa el objetivo) para modificar su posición y poco a poco ir llegando a la posición destino.
- Mientras un cubo este haciendo uso del servidor de acción los otros permanecerán esperando.
- Todo termina cuando los cubos que pasaron la prueba de altura han sido transportados desde su posición inicial a su posición de destino.

Para comenzar, primero nos dirigimos a nuestro espacio de trabajo ROS y ejecutamos el siguiente comando:

```
roslaunch practice practice_nodes.launch &
```

El comando “roslaunch” inicia “roscore” si es que no se ha iniciado.

Este comando lanza un archivo de lanzamiento, el cual levanta el servidor TCP, los nodos que ejecutan servidores (que ya hemos creado), y carga algunos parametros al servidor de parametros (todo en un simple comando).

Resultados:

```

root@5be7ed3ef0a8:/catkin_ws# roslaunch practice practice_nodes.launch &
[1] 89
root@5be7ed3ef0a8:/catkin_ws# ... logging to /root/.ros/log/d2479c70-cbbf-11ec-a2fd-0242ac150002/roslaunch-5be7ed3ef0a8-89.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://5be7ed3ef0a8:45881/
SUMMARY
========
PARAMETERS
  * /ROS_IP: 0.0.0.0
  * /rosdistro: melodic
  * /rosversion: 1.14.12

NODES
  /
    call_move_cube_as_service_node (practice/call_move_cube_as_server.py)
    convert_metres_to_feet_service_node (practice/convert_metres_to_feet_server.py)
    height_test_service_node (practice/height_test_server.py)
    move_cube_as_node (practice/move_cube_as.py)
    server_endpoint (ros_tcp_endpoint/default_server_endpoint.py)

auto-starting new master
process[master]: started with pid [99]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to d2479c70-cbbf-11ec-a2fd-0242ac150002
process[rosout-1]: started with pid [110]
started core service [/rosout]
process[server_endpoint-2]: started with pid [113]
[INFO] [1651678532.129257]: Starting server on 0.0.0.0:10000
process[convert_metres_to_feet_service_node-3]: started with pid [118]
process[height_test_service_node-4]: started with pid [126]
[INFO] [1651678532.768750]:
Service 'convert_metres_to_feet' created
[INFO] [1651678533.277424]:
Service 'height_test' created
process[move_cube_as_node-5]: started with pid [133]
[INFO] [1651678533.797991]:
Simple action server 'move_cube' created
process[call_move_cube_as_service_node-6]: started with pid [144]
[INFO] [1651678534.208575]:
Service 'call_move_cube_as' created
root@5be7ed3ef0a8:/catkin_ws# 

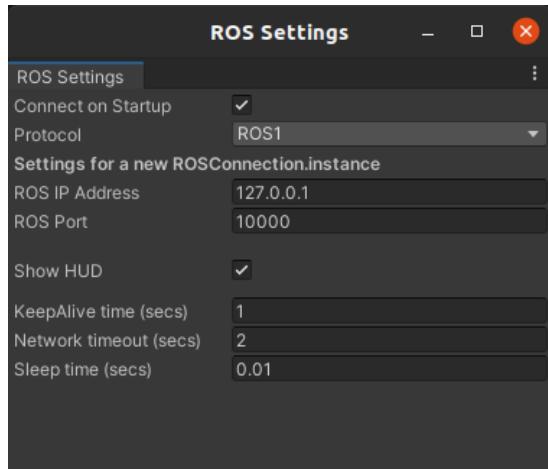
```

```

root@5be7ed3ef0a8:/catkin_ws# rosnome list
/call_move_cube_as_service_node
/convert_metres_to_feet_service_node
/height_test_service_node
/move_cube_as_node
/rosout
/server_endpoint
root@5be7ed3ef0a8:/catkin_ws# rostopic list
/move_cube/cancel
/move_cube/feedback
/move_cube/goal
/move_cube/result
/move_cube/status
/rosout
/rosout_agg

```

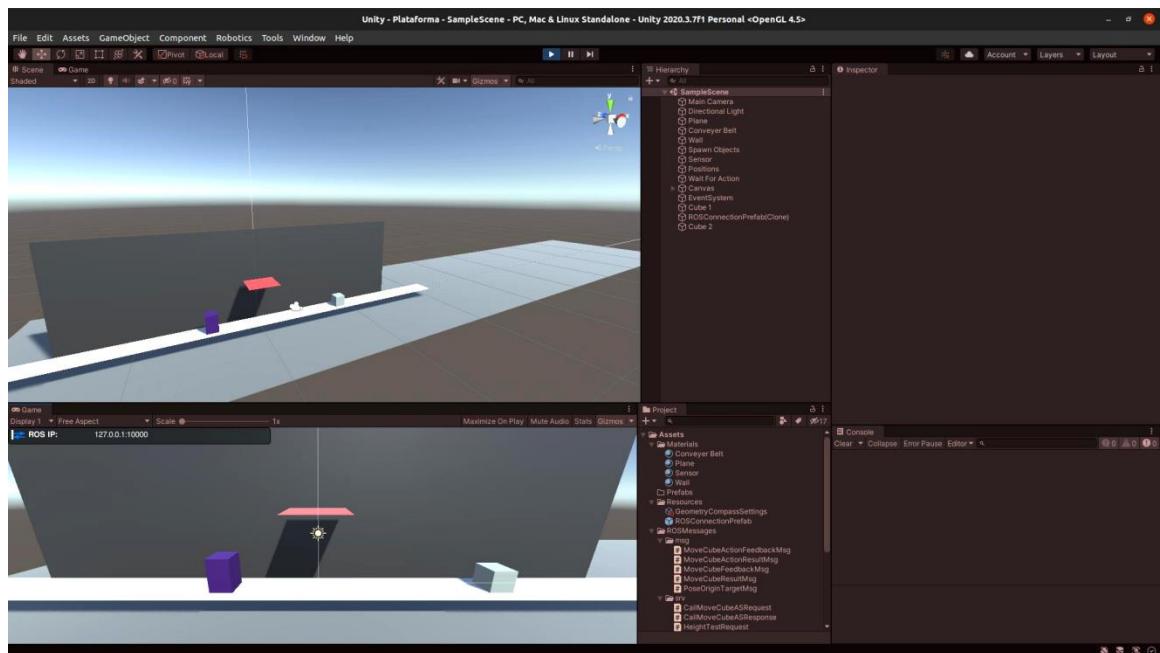
Despues, en Unity establecemos la siguiente configuracion en el apartado “Robotics -> ROS Settings”:

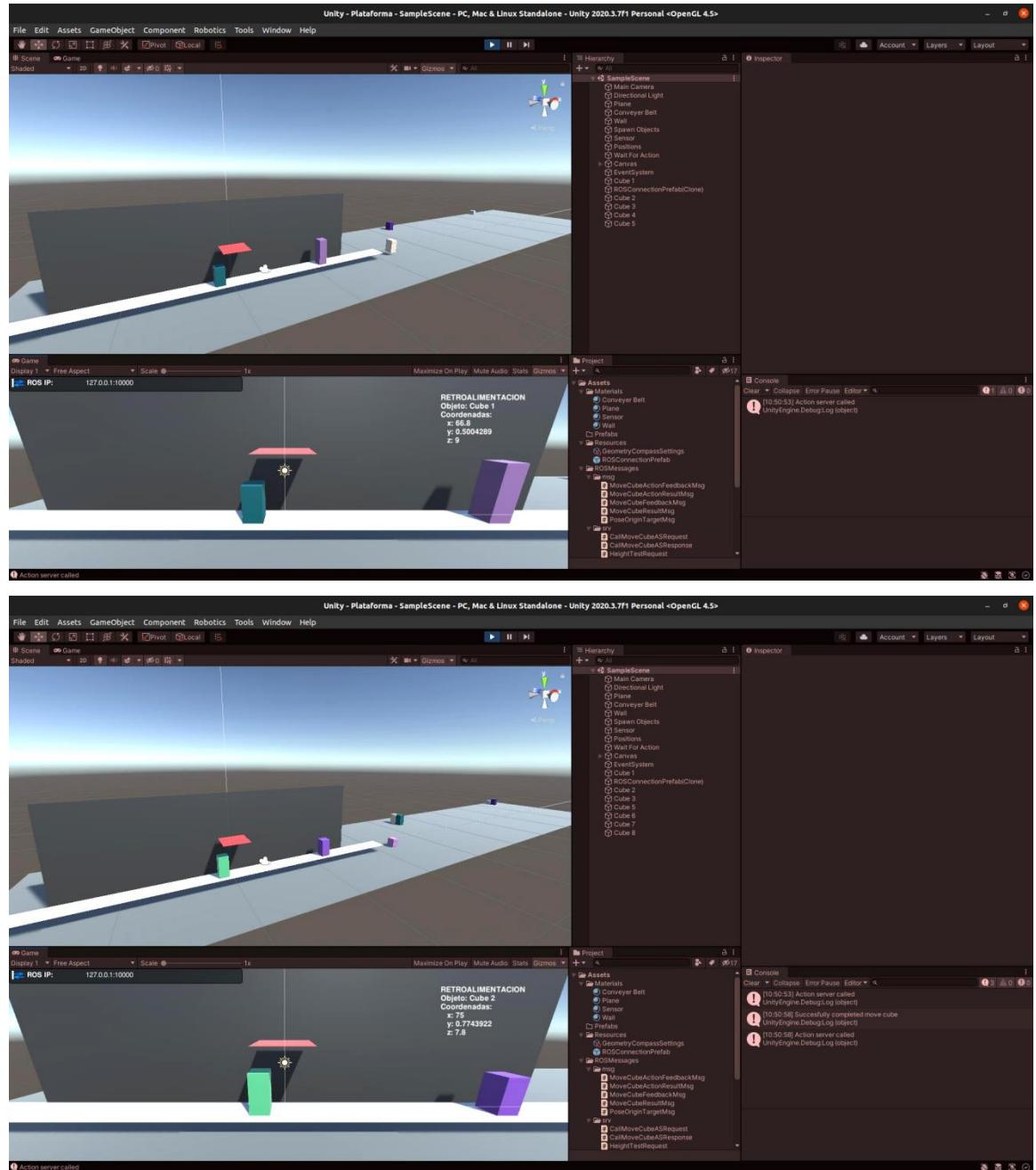


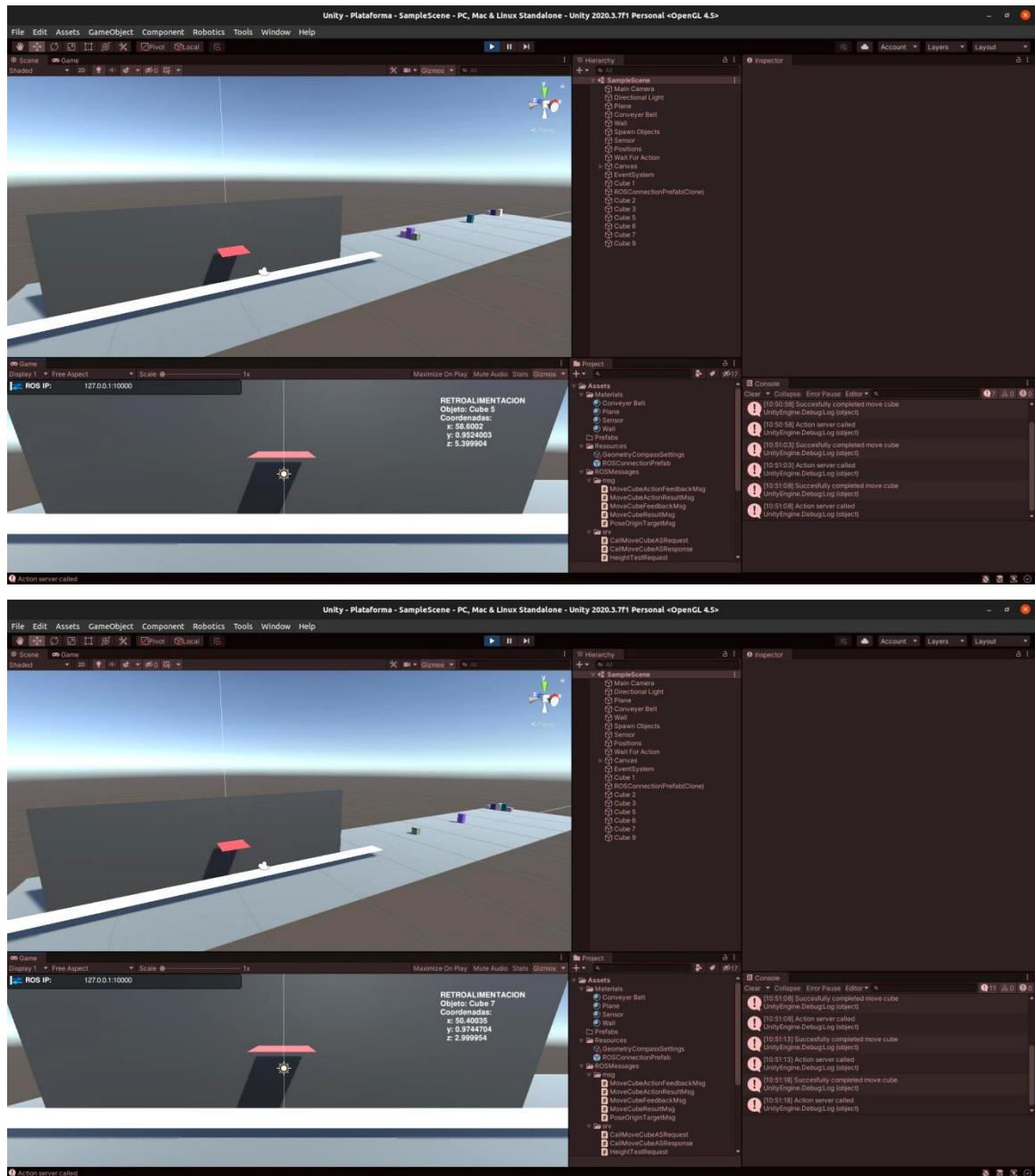
Y por ultimo, damos play en la escena para que se ejecuta la simulacion.

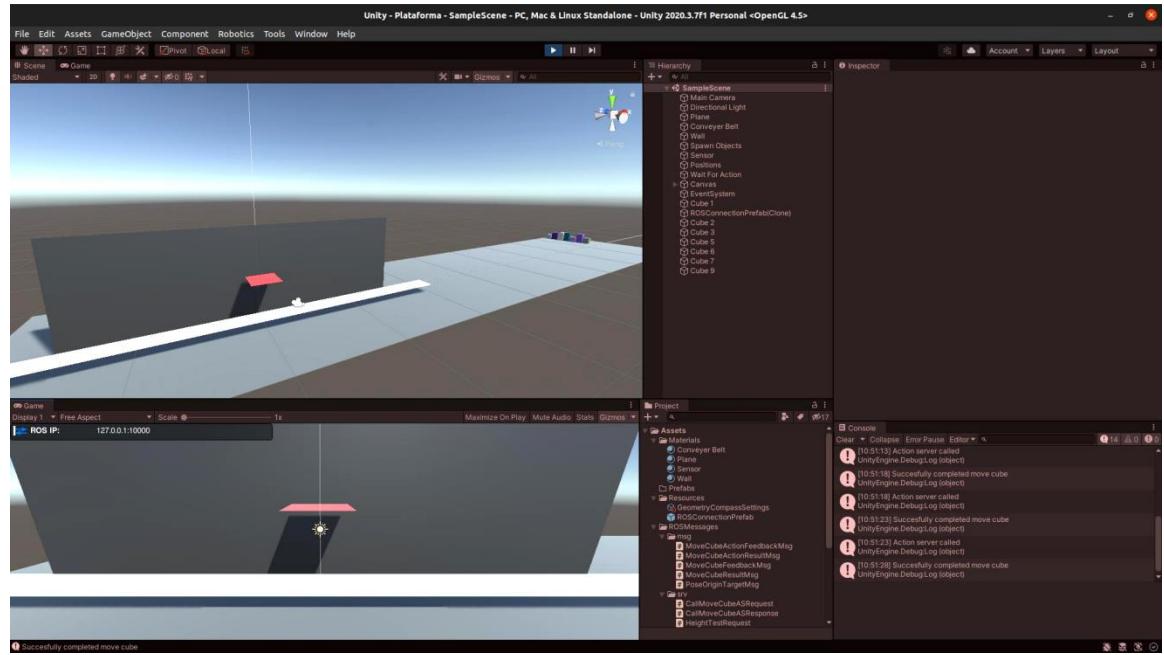
Resultados:

● Unity









NOTA: Un video de la ejecución completa de la simulación se encuentra en el material proporcionado.

- ROS

```
root@5be7ed3ef0a8:/catkin_ws# [INFO] [1651679447.498550]: Connection from 172.21.0.1
[INFO] [1651679447.502511]: RegisterRosService(/height_test, <class 'practice_msgs.srv. HeightTest.HeightTest'>) OK
[INFO] [1651679447.505047]: RegisterRosService(/height_test, <class 'practice_msgs.srv. HeightTest.HeightTest'>) OK
[INFO] [1651679453.387818]: RegisterRosService(/call_move_cube_as, <class 'practice_msgs.srv._CallMoveCubeAS.CallMoveCubeAS'>) OK
[INFO] [1651679453.397081]: Goal has been sent to the action server
[INFO] [1651679453.433393]: RegisterSubscriber(/move_cube/feedback, <class 'practice_msgs.msg. MoveCubeActionFeedback.MoveCubeActionFeedback'>) OK
[INFO] [1651679453.440355]: RegisterSubscriber(/move_cube/result, <class 'practice_msgs.msg. MoveCubeActionResult.MoveCubeActionResult'>) OK
[INFO] [1651679458.486022]: RegisterRosService(/call_move_cube_as, <class 'practice_msgs.srv._CallMoveCubeAS.CallMoveCubeAS'>) OK
[INFO] [1651679463.494143]: Goal has been sent to the action server
[INFO] [1651679463.541352]: RegisterRosService(/call_move_cube_as, <class 'practice_msgs.srv._CallMoveCubeAS.CallMoveCubeAS'>) OK
[INFO] [1651679468.594439]: Goal has been sent to the action server
[INFO] [1651679468.600454]: RegisterRosService(/call_move_cube_as, <class 'practice_msgs.srv._CallMoveCubeAS.CallMoveCubeAS'>) OK
[INFO] [1651679473.738405]: RegisterRosService(/call_move_cube_as, <class 'practice_msgs.srv._CallMoveCubeAS.CallMoveCubeAS'>) OK
[INFO] [1651679473.743052]: Goal has been sent to the action server
[INFO] [1651679478.779911]: RegisterRosService(/call_move_cube_as, <class 'practice_msgs.srv._CallMoveCubeAS.CallMoveCubeAS'>) OK
[INFO] [1651679478.789744]: Goal has been sent to the action server
[INFO] [1651679483.914135]: RegisterRosService(/call_move_cube_as, <class 'practice_msgs.srv._CallMoveCubeAS.CallMoveCubeAS'>) OK
[INFO] [1651679483.919994]: Goal has been sent to the action server
[ERROR] [1651679526.936642]: Exception: No more data available
[INFO] [1651679526.939987]: Disconnected from 172.21.0.1

root@5be7ed3ef0a8:/catkin_ws#
```

Son los resultados en nuestro espacio de trabajo ROS a la hora de ejecutar la simulación en Unity.

Con esto concluimos la práctica.

Repaso y material utilizado

Repaso

Hasta el momento hemos realizado lo siguiente:

- Introducción al espacio de trabajo ROS
Se presentaron todos los componentes involucrados en un espacio de trabajo ROS, así como los archivos de configuración que se generan.
También se introdujo sobre la creación de paquetes en ROS, los cuales son una forma de darle modularidad a una aplicación ROS, y que estos paquetes son llamados así por el hecho de que existen los archivos “CMakeLists.txt” y “package.xml”.
- Tareas ROS
 - Se implementaron suscriptores y publicadores.
 - Se implementaron servidores de servicio y servidores de acción.
 - Se crearon mensajes personalizados y se utilizaron en ciertas tareas.
 - Se utilizó el paquete **actionlib** para tareas que requerían que no fueran bloqueantes.
 - Con bases más sólidas se logró modificar implementaciones de tutoriales anteriores para adaptarlos a cualquier necesidad.
- Práctica
Se realizó una integración de Unity (para la simulación) y ROS (para la comunicación entre componentes) para realizar una tarea que, a grandes rasgos, puede simular una comunicación entre “robots” utilizando ROS para diversas tareas que pueden surgir en un entorno real, en donde se ven involucrados sensores, robots móviles, etc. (existen datos que pueden ser procesados, robots que pueden ser manipulados por comandos, algoritmos para la planificación de trayectorias, etc., todo esto fácilmente puede estar dentro de un sistema ROS).

Todo esto de la mano de un curso llamado “Hello (Real) World with ROS – Robot Operating System” que lo podemos encontrar en el siguiente sitio:

<https://www.edx.org/es/course/hello-real-world-with-ros-robot-operating-system>

Más adelante nos centraremos ahora en los siguientes temas:

- Aprenderemos a como modelar un mundo de tal manera que ROS pueda interactuar en él, ya que modelar un robot y su entorno es una parte importante de la creación de una aplicación ROS exitosa.
- Hablaremos sobre el formato de archivo URDF (Descripción Unificada de Robot) y aprenderemos a usarlo. Se trataran conceptos como enlaces, uniones y geometría (links, joints, geometry).
- Crearemos un modelo de robot visual con URDF desde cero (en general, es el robot R2-D2, personaje de Star Wars).
- Hablaremos sobre RViz, el cual es una herramienta de visualización 3D para ROS.
- Hablaremos sobre Gazebo, el cual es un simulador de robótica 3D de código abierto (Unity también es un simulador).
- Hablaremos sobre los tipos de articulaciones, ya que estaremos usándolas a la hora de crear nuestro modelo de robot con URDF (estas articulaciones pueden ser *fixed*, *continuous*, *revolute*, *prismatic*, *planar* y *floating*).

Recordemos que en un robot se ven involucradas muchos tipos de articulaciones (de ruedas, de cabeza, de pinza, de brazo, etc.).

- Hablaremos sobre cómo agregar propiedades físicas básicas a nuestros modelos URDF (así como de inercia, coeficientes de contacto, dinámica conjunta, etc.), ya que para que un modelo se simule correctamente se debe definir varias propiedades físicas del robot.
- Una vez introducido lo que es URDF, se hablará de Xacro, el cual nos sirve para limpiar un archivo URDF, es decir, para reducir la cantidad de código en un archivo URDF (es el siguiente paso al modelado de robot, ya que nos facilita aún más la tarea de modelar robots).
- Aprenderemos a simular un modelo de robot creado en URDF en Gazebo.
Recordemos que en el tutorial de “pick and place” se utilizó un modelo de robot en URDF que se simuló en Unity, por lo que podemos simular modelos de robot tanto en Unity como en Gazebo. Pero ahora se aprenderá a simular un modelo de robot en Gazebo, ya que es un simulador más fácil de usar que Unity y es más fácil de utilizar en un entorno de trabajo ROS.
- Hablaremos sobre la relación que existe entre URDF y SDF (Formato de Descripción de Simulación, el cual es un formato XML que describe objetos y entornos para simuladores, visualización y control de robots).
- Hablaremos sobre los complementos Gazebo, ya que son parte fundamental

para que el robot sea interactivo con nosotros y ROS.

Los complementos de Gazebo brindan a nuestros modelos URDF una mayor funcionalidad, los cuales pueden vincular mensajes ROS y llamadas de servicio para, por ejemplo, la salida de un sensor y la entrada de un motor (los complementos que se pueden utilizar son *World*, *Model*, *Sensor*, *System*, *Visual* y *GUI*, pero no en todos se puede hacer referencia a través de un archivo URDF).

- Hablaremos sobre cómo usar RViz para monitorear el estado de nuestro robot simulado mediante la publicación directamente desde Gazebo.
- Hablaremos sobre controladores (los cuales sirven para activar las articulaciones de nuestro robot), trasnmisiones (son los elementos encargados de transmitir el movimiento desde los actuadores hasta las articulaciones), interfaces (cada controlador tiene su respectiva interfaz), entre otros.

Cuando se trata de robots, debemos asegurarnos de que su movimiento este controlado, es decir, que las acciones no sean ni demasiado lentas ni demasiado rápidas, y que sigan la trayectoria especificada. En este tema se hablara un poco sobre los controladores PID.

- Aprenderemos a usar un robot móvil llamado “TurtleBot” (en específico el “TurtleBot3”, ya que el “TurtleBot2” no me funciono muy bien).
- Abarcaremos el tema de la navegación utilizando el robot móvil “TurtleBot” (así como lo que es el mapeo, la localización, planificación de trayectorias, entre otros).

Comentarios:

- Cuando se utiliza la clase “ROSConnection” en los scripts C#, solo se instancia un objeto de dicha clase (ya que la instancia es un de tipo estática) para toda la simulación, por lo que si hacemos uso de dicha clase en diferentes scripts C#, es importante tener en cuenta que lo que se haga en dicha instancia tendrá efecto para los demás scripts C# que están haciendo uso de él.
- La clase “ROSConnection” no tiene métodos para implementar acciones, solo para suscribirse a un tema, publicar en un tema, crear un servicio en Unity y registrar un servicio desde ROS, por lo que se utilizó un servicio intermedio que pudiera llamar a un servidor de acción y que pudiera procesar el objetivo, haciéndolo posible desde Unity (ya que se pueden realizar llamadas a servicio).
- Es importante mencionar que lo que se ve en videos, páginas web, documentación, etc., sobre ROS, no todo se puede recrear a la primera, ya que normalmente siempre se tiene el problema de la versión que se está utilizando,

ya que toda documentación y archivos están versionados, por lo que siempre es importante trabajar con alguna versión que nos funcione bien, que exista buena documentación al respecto y que no tengamos problemas al querer incluir más funcionalidad o al querer seguir trabajando con esa versión (lo cual incluye opiniones de personas en foros sobre si han tenido o no problemas).

Material

- Video de la simulación de la practica

El video se encuentra en la carpeta “Material” con el nombre de “Simulación practica.mp4”.

- Imagen Docker

La imagen ROS Docker como resultado de todo lo hecho, la podemos encontrar en el siguiente repositorio en Docker Hub:

https://hub.docker.com/r/chuy7/ros_hello_world/tags

(**TAG: practice1**)

Para descargar la imagen a nuestro repositorio local tendremos que ejecutar el siguiente comando en la terminal (recordando que esta imagen es el resultado de lo que se estuvo haciendo):

`docker pull chuy7/ros_hello_world:practice1`

Para crear un contenedor de la imagen anterior ejecutamos el siguiente comando en la terminal:

```
docker run --rm -it -p 10000:10000 -p 5005:5005  
chuy7/ros_hello_world:practice1 /bin/bash
```

- Archivos ROS

- Servidor TCP (para la comunicación entre ROS y Unity) para la practica (**v0.6.0**):

<https://github.com/Unity-Technologies/ROS-TCP-Endpoint>

- Archivos ROS de la practica :

<https://github.com/ChuyFernandez/ROS-practice1>

Viene incluido:

- Paquetes ROS (son los paquetes ROS resultantes de la practica).
Carpeta “**ros_packages**”.
- Declaraciones de mensajes ROS y servicios ROS (para la generacion de Scripts C# en Unity).

- Carpeta “***ROSMessages***”.
- Archivos para la construcción de la imagen ROS Docker.
- Carpeta “***ros_docker***”.
- Archivo “***docker-compose.yaml***”.
- Archivos Unity
 - Paquetes “ROS TCP Connector” (**v0.6.0**):
https://github.com/Unity-Technologies/Unity-Robotics-Hub/blob/main/tutorials/quick_setup.md
 - Proyecto :
<https://github.com/ChuyFernandez/UNITY-practice1>
 Solo incluye la carpeta “Assets” (es la carpeta en donde se crearon los scripts, etc.).

CONCLUSION

A todos los scripts creados, tanto en C# como en Python, se les ha colocado comentarios para una mayor comprensión para el lector.

Ya conocidos los aspectos más importantes de una aplicación ROS (la comunicación entre nodos, el tema de suscriptor/publicador, servidores de servicio y de acción, tareas sincrónicas y asincrónicas, tipos de mensajes en la comunicación, simulaciones utilizando Unity y ROS, etc.), lo que sigue es el modelado del robot (ya que en la realidad uno mismo es el que modela algún robot para una tarea en específico), manejo de todas las partes que conforman a un robot (articulaciones, transmisiones, etc.) y como integrar ROS para la comunicación conjunta de todas las partes que conforman a un robot. Aquí las simulaciones entran en juego mucho porque son la herramienta para llevar a cabo una tarea de manera simulada y controlada antes de llevarlo a la práctica en la vida real (depuramos nuestra aplicación ROS en un entorno de simulación y prueba).