

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 1 26-03-2021 - 26-05-2021

INDICE DE CONTENIDO

INTRODUCCION.....	3
DESARROLLO	3
Tutorial Pick-and-place.....	3
Configuracion de espacio de trabajo ROS usando Docker	4
Escena Unity con URDF importado	6
Integracion de ROS y Unity para la comunicacion.....	10
Tarea pick-and-place mediante el uso de ROS	20
Tutoriales de iniciacion: Integracion ROS-Unity	27
Configuracion de espacio de trabajo ROS usando Docker	28
Publicacion en un tema desde una escena de Unity.....	35
Suscripcion a un tema desde una escena de Unity.....	50
Implementacion de un servicio dentro de una escena de Unity	62
Llamada a servicio externo desde una escena de Unity	67
Publicar en un tema desde ROS por linea de comandos (como complemento).....	71
Repaso.....	73
Material.....	74
CONCLUSION	75

INTRODUCCION

Algo del trabajo realizado es lo siguiente:

- Se instaló Unity para la simulación robótica y se configuro para tener instalados los paquetes de Unity Robotics y poder trabajar con ROS.
- Se visitó el repositorio central de herramientas, tutoriales, recursos y documentación para la simulación robótica en Unity para consultar el material y realizar los tutoriales.

(github: <https://github.com/Unity-Technologies/Unity-Robotics-Hub>)

DESARROLLO

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

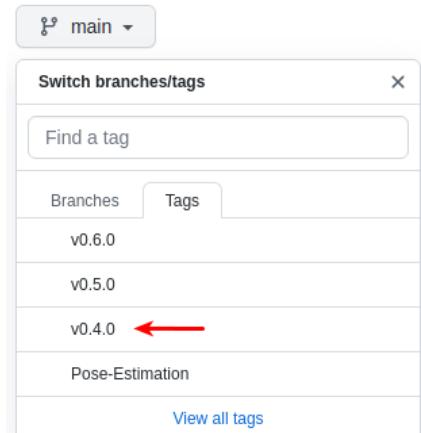
Tutorial Pick-and-place

URL del sitio: <https://github.com/Unity-Technologies/Unity-Robotics-Hub>

Para este tutorial, en específico se trabajó con el tag **v0.4.0** del repositorio (código más viejo y con algunos detalles).

El código comparado con el de la última versión que es **v0.6.0** es bastante diferente y mucho mejor, ya que se consideran algunas buenas prácticas y el código es más entendible.

De todas formas no hay tanto problema con haber trabajado con la versión **v0.4.0**, pero los tutoriales posteriores se estará usando la versión **v0.6.0** ya que nos puede dar menos problemas a la larga con algunas soluciones de errores y código mas legible.



El siguiente repositorio es proporcionado por el sitio web y contiene archivos de proyecto para el tutorial de pick-and-place (como archivos de Unity, archivos URDF y scripts ROS):

- `git clone --recurse-submodules https://github.com/Unity-Technologies/Unity-Robotics-Hub.git`

Configuración de espacio de trabajo ROS usando Docker

Se nos proporciona dos maneras de configurar el espacio de trabajo ROS:

- Usando Docker
- Instalando ROS en la propia maquina

Nosotros usaremos Docker, debido a que es muy práctico y no necesitamos instalar nada en nuestra computadora más que Docker. Es importante tener algunos conocimientos de Docker, en concreto, como trabajar con contenedores.

Otras de las ventajas que tenemos al usar Docker es que cualquiera puede recrear los pasos que se lleven a cabo simplemente teniendo Docker y bajando la imagen con el resultado final.

Primero que nada, debemos instalar Docker. Para ello, seguimos los pasos de instalación que se presentan en este sitio:

- <https://docs.docker.com/engine/install/>

Después, proseguimos a construir la imagen de ROS Docker proporcionada ejecutando los siguientes comandos:

NOTA: Primero debemos tener clonado el repositorio del proyecto.

- `cd /PATH/TO/Unity-Robotics-Hub/tutorials/pick_and_place`
- `git submodule update --init --recursive`
- `docker build -t unity-robotics:pick-and-place -f docker/Dockerfile .`

Banderas de línea de comando (command-line flags):

- `-t, --tag` : especifica un nombre y opcionalmente una etiqueta en el formato “nombre:etiqueta” a la imagen.
- `-f` : nos permite especificar la ruta al Dockerfile.

El Dockerfile proporcionado utiliza la imagen base melodica 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).

Veamos que contiene este archivo Dockerfile:

- Se especifica la imagen principal a partir de la cual se construirá la imagen (ros:melodic-ros-base).
- Se ejecutan algunos comandos (actualización

Un **Dockerfile** es un documento de texto que contiene todos los comandos que un usuario podría llamar en la linea de comandos para ensamblar una imagen.

e instalación de paquetes necesarios, los cuales son especificados por el propietario de la imagen y no es necesario agregar o quitar paquetes, normalmente suelen seguir buenas prácticas y especifican lo necesario).

- Se copian paquetes ROS (contenidos en el repositorio del proyecto que clonamos) dentro de la imagen.
Más adelante hablaremos sobre estos paquetes.
- Se copian dos archivos bash (contenidos en el repositorio del proyecto que clonamos) dentro de la imagen (son scripts de configuración).
 - Uno de ellos (el script de bash “set-up-workspace”) carga archivos de configuración ROS, establece el valor de una variable en un archivo y construye nuestro espacio de trabajo catkin en el directorio especificado por la variable de entorno “\$ROS_WORKSPACE”. Después se elimina el script inmediatamente.
 - El otro archivo (el script de bash “tutorial”) carga archivos de configuración ROS en nuestro espacio de trabajo catkin y establece el valor de una variable en un archivo.
- Se establece en la variable de entorno “\$ROS_WORKSPACE” el directorio que se utilizará como espacio de trabajo.
- Se agregan permisos de ejecución a ciertos scripts Python que se encuentran dentro de los paquetes ROS que anteriormente copiamos.
- Por último, se especifica el comando que se ejecutara cuando se ejecute un contenedor, el cual es el script de bash “tutorial”.

Creo es importante conocer como está conformado este archivo Dockerfile, ya que nos ayuda a entender un poco que se está haciendo.

La construcción de la imagen instalará los paquetes necesarios, copiará los paquetes y submodulos ROS proporcionados en el contenedor y creará el espacio de trabajo catkin. Ahora iniciamos un contenedor Docker de la imagen recién construida ejecutando el siguiente comando:

- `docker run -it --rm -p 10000:10000 -p 5005:5005 unity-robotics:pick-and-place /bin/bash`

Banderas de línea de comando (command-line flags):

- `-it` : para abrir un terminal interactivo.

- --rm : para eliminar el contenedor una vez que salgamos del contenedor.
- -p : para mapear los puertos de nuestro ordenador con los de dentro del contenedor.

En la consola debería abrirse un shell bash en la raíz del espacio de trabajo ROS.

```
root@30441fb9434:/catkin_ws
chuy@chuy:~$ docker run -it --rm -p 10000:10000 -p 5005:5005 unity-robotics:pick-and-place /bin/bash
root@30441fb9434:/catkin_ws# ls -l
total 16
drwxr-xr-x 13 root root 4096 Jun 17 14:25 build
drwxr-xr-x  5 root root 4096 Jun 17 14:25 devel
drwxr-xr-x  1 root root 4096 Jun 17 14:24 src
root@30441fb9434:/catkin_ws# pwd
/catkin_ws
root@30441fb9434:/catkin_ws#
```

```
root@2a5543b5a59e:/catkin_ws#
root@2a5543b5a59e:/catkin_ws# ls -l src/
total 24
lrwxrwxrwx 1 root root   50 Jun 17  2021 CMakeLists.txt -> /opt/ros/melodic/share/catkin/cmake/toplevel.cmake
drwxr-xr-x  1 root root 4096 Jun 17  2021 moveit_msgs
drwxr-xr-x  1 root root 4096 Jun 17  2021 niryo_moveit
drwxr-xr-x  1 root root 4096 Jun 17  2021 niryo_one_ros
drwxr-xr-x  1 root root 4096 Jun 17  2021 niryo_one_urdf
drwxr-xr-x  1 root root 4096 Jun 17  2021 ros_tcp_endpoint
root@2a5543b5a59e:/catkin_ws#
```

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

Escena Unity con URDF importado

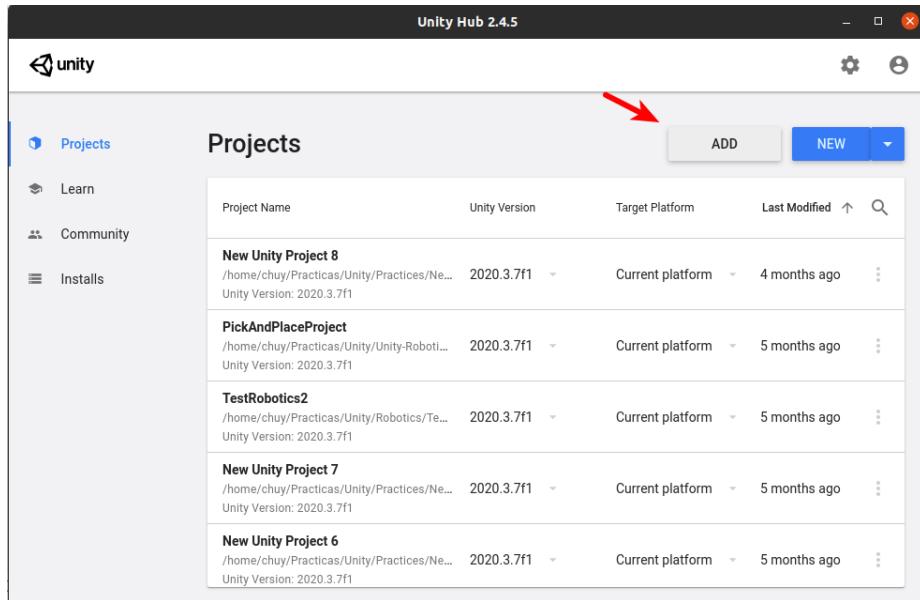
Primeramente, necesitamos descargar Unity Hub y después instalar una versión de Unity (en mi caso es la **2020.3.7f1**)

La descarga se realiza del siguiente sitio:

- <https://unity.com/es/download>

Una vez descargado Unity Hub e instalado una versión de Unity, agregamos un proyecto existente que se encuentra dentro del repositorio que ya hemos clonado, ubicado en:

- /PATH/TO/Unity-Robotics-Hub/tutorials/pick_and_place/ PickAndPlaceProject



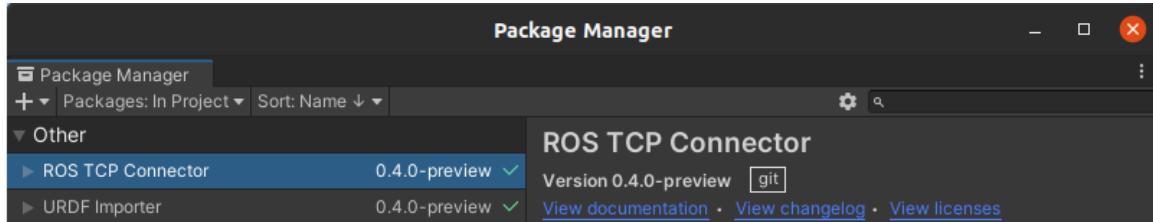
Abrimos el proyecto recién agregado.

IMPORTANTE: Una vez dentro de Unity, lo primero que haremos será seguir los pasos indicados en el siguiente enlace para instalar los paquetes de Unity Robotics:

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

NOTA: Al principio tuve problemas con estos paquetes porque no me permitían, en algunos pasos más adelante, comunicar Unity con ROS, y la solución se dio después de que actualizaran el repositorio con algunas mejoras en el código fuente.

Se trabajó con las siguientes versiones de estos paquetes:



Las versiones más recientes tienen muchas mejoras en el código fuente y bastantes cambios a comparación con esta versión que se utilizó. Solo para que se tenga en cuenta con qué versiones se trabaja.

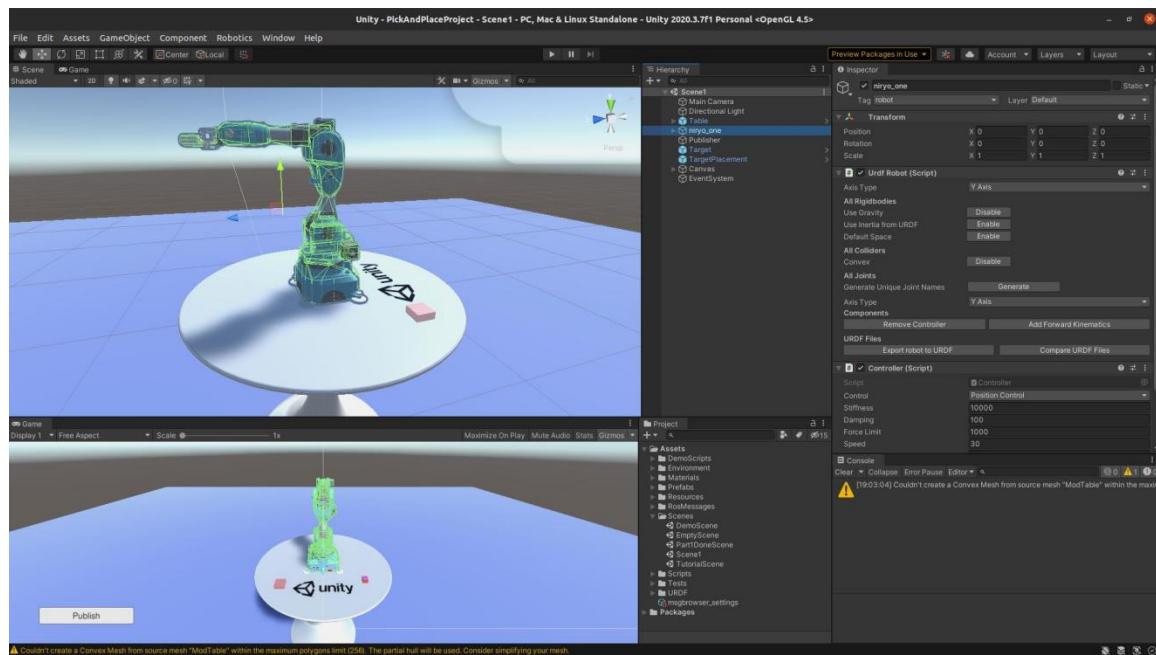
A continuación seguimos los pasos indicados en el siguiente enlace el cual nos muestra como configurar una escena básica de Unity y como importar un robot usando URDF

Importer:

- https://github.com/Unity-Technologies/Unity-Robotics-Hub/blob/main/tutorials/pick_and_place/1_urdf.md

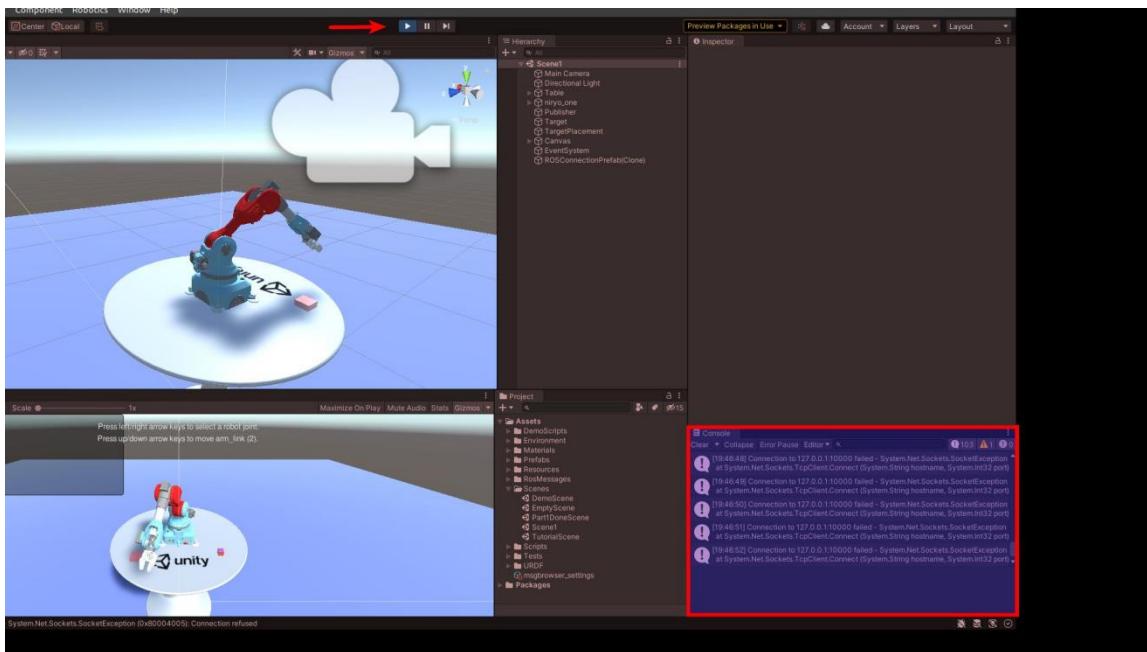
Como resultado tendremos lo siguiente:

URDF es un formato de lenguaje utilizado para describir robots en el marco de gramática XML. Es muy popular en el mundo de ROS.



Por último, presionamos el botón de reproducir de Unity para ingresar al modo de reproducción. Esto nos permitirá poder seleccionar las articulaciones del brazo robótico y moverlas usando las teclas de flecha de la siguiente manera:

- *Teclas de flecha izquierda/derecha:* para navegar a través de las articulaciones, donde la articulación seleccionada se resaltara en rojo.
- *Teclas de flecha arriba/abajo:* para controlar el movimiento de la articulación seleccionada.



Las advertencias indicadas en la imagen fueron un problema al principio, ya que este proyecto tiene instalados los paquetes de Unity Robotics no actualizados y, por tanto, la comunicación con ROS no funciona del todo bien. Pero de aquí en adelante, todo proyecto tendrá los paquetes de Unity Robotics actualizados.

Por el momento, funciona bien para interactuar con el brazo robótico.

NOTA: Por eso es importante verificar que estemos agregando los paquetes lo más actualizados posibles (ya que suelen tener correcciones de errores y, por supuesto, mejoras en el código fuente).

Para detener el modo de reproducción solo volvemos a dar clic en el botón de reproducción.

Aspectos a tomar en cuenta:

- Los lenguajes soportados nativamente por Unity son:
 - C#
 - UnityScript
- En C# cuando creamos un script y declaramos variables como públicas, podemos verlas y modificarlas en tiempo real en la ventana *Inspector* en Unity.
- El comportamiento de los “GameObjects” (objetos del juego) es controlado por los “componentes” que están adjuntos. Esto nos permite usar disparadores de eventos, modificar propiedades del componente y responder al input del

usuario de la forma que nosotros queramos.

Estos “GameObjects” aparecen en la ventana *Jerarquía* en Unity.

Integración de ROS y Unity para la comunicación

A continuación explicaremos como configurar nuestro espacio de trabajo ROS para este proyecto.

Los pasos que se cubrirán incluyen la creación de una conexión TCP entre Unity y ROS, la generación de scripts C# a partir de un mensaje ROS y la publicación y suscripción a un tema ROS.

Para habilitar la comunicación entre Unity y ROS, un TCP endpoint se ejecuta como un nodo ROS para manejar el paso de mensajes hacia y desde Unity y ROS.

En el lado de Unity, un componente “ROSConnection” (un script) proporciona las funciones necesarias para publicar, suscribirse o llamar a un servicio utilizando el nodo ROS del TCP endpoint.

En el protocolo TCP (protocolo de control de transmision) los datos pueden enviarse de forma bidireccional una vez establecida la conexión.

Algo de información sobre los nodos ROS:

- Los nodos se comunican entre sí publicando mensajes en temas. Estos mensajes se definen en archivos *.msg*.
 - Los mensajes son estructuras de datos simples, los cuales admiten los tipos primitivos estándar (entero, flotante, booleano, etc.) al igual que las matrices de tipos primitivos. Ejemplo:

fieldtype1fieldname1

fieldtype2fieldname2

El modelo de *publicacion/suscripcion* es un paradigma de comunicación asíncrono muy flexible.

- Los nodos también pueden intercambiar un mensaje de solicitud y respuesta como parte de una llamada de servicio ROS. Estos mensajes se definen en archivos *.srv*.

- La *solicitud/respuesta* se realiza a través de un servicio, que se define por un par de mensajes: uno para la solicitud y otro para la respuesta.

Un nodo ROS proveedor ofrece un servicio con un nombre (tipo string), y un cliente llama al servicio enviando el mensaje de solicitud y esperando la respuesta. Ejemplo:

fieldtypereq1 fieldnamereq1 (solicitud)

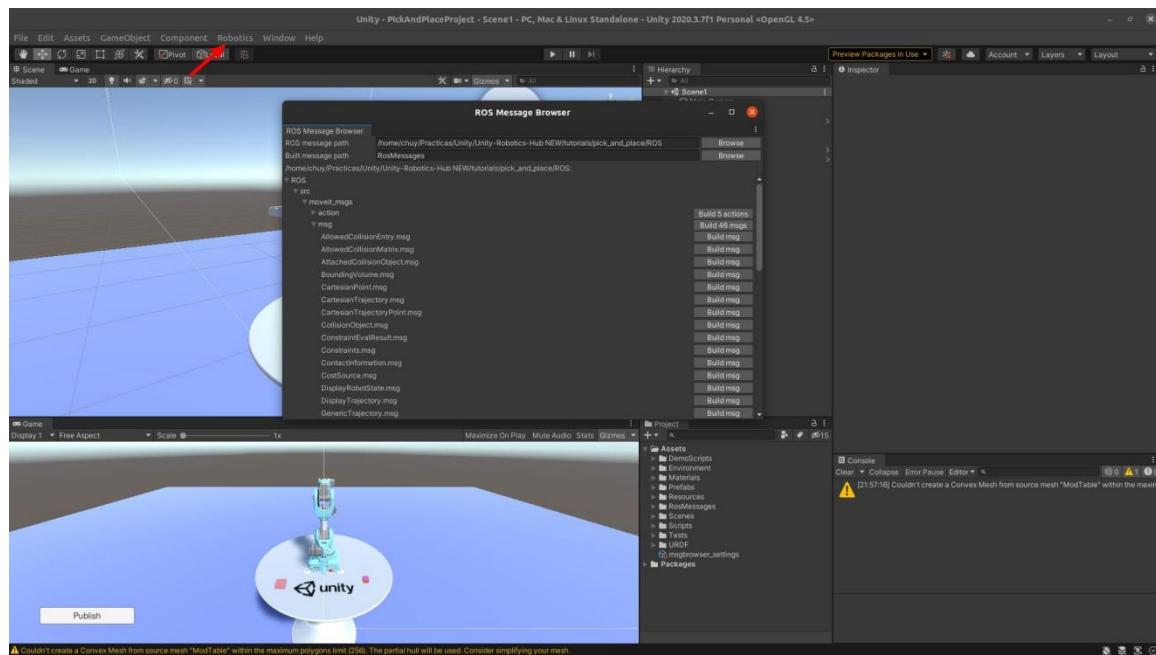
fieldtyperes2 fieldnameres2 (respuesta)

Lo primero que haremos será configurar la escena de Unity yendo a la barra de menú y seleccionando la opción “Robotics -> Generate ROS Messages...”.

En la ventana que se nos abre establecemos la ruta del mensaje ROS en:

- /PATH/TO/Unity-Robotics-Hub/tutorials/pick_and_place/ROS

Este directorio cuenta con archivos ya listos como base en este tutorial.



NOTA: Lo que haremos a continuación se logra gracias al complemento “MessageGeneration” que genera clases C#, incluidas las funciones de serializacion y deserializacion, a partir de mensajes ROS, de esta manera los mensajes ROS que se pasan entre Unity y ROS se serializan exactamente como ROS los serializa internamente.

Expandimos hasta la carpeta “ROS/src/moveit_msgs/msg” y nos desplazamos hasta encontrar el “RobotTrajectory.msg” y hacemos clic en su “Build msg” para generar un nuevo script C# con el mismo nombre que el del mensaje, con un prefijo “M” (para el mensaje).

Este archivo se guarda en:

- “Assets/RosMessages/Moveit/msg”

Tiene por nombre:

- MRobotTrajectory.cs

Ahora expandimos hasta la carpeta “ROS/src/nyrio_moveit/msg” y hacemos clic en “Build 2 msgs” para generar nuevos scripts C# a partir de los archivos ROS .msg.

Estos archivos se guardaran en:

- “Assets/RosMessages/NyrioMoveit/msg”

Tienen por nombre:

- MNyrioMoveitJoints.cs

Describe un valor para cada articulacion en el brazo nyrio.

- MNyrioTrajectory.cs

Describe una lista de valores del tipo de mensaje “RobotTrajectory” que contendrá las trayectorias calculadas para la tarea de “recoger y colocar” (pick and place).

Finalmente, ahora que ya hemos generado los mensajes, crearemos el servicio para mover el robot. Para esto, expandimos hasta la carpeta “ROS/src/nyrio_moveit/srv” y hacemos clic en “Build 1 srv” para generar nuevos scripts C# a partir de los archivos ROS .srv.

Estos archivos se guardaran en:

- “Assets/RosMessages/NyrioMoveit/srv”

Tiene por nombre:

- MMoverServiceRequest.cs (solicitud)

- MMoverServiceResponse.cs (respuesta)

Es un tipo de mensaje personalizado para la solicitud y otro para la respuesta.

Antes de continuar me gustaría dar un resumen de los tipos de “msg” y “srv” disponibles en los distintos paquetes ROS de la pila “common_msgs”. Estos tipos se han estado usando en la declaración de mensajes y servicios:

- *actionlib_msgs*: define los mensajes comunes para interactuar con un servidor de acciones y un cliente de acciones.
- *geometry_msgs*: proporciona mensajes para primitivas geométricas comunes como puntos, vectores y poses.

- *nav_msgs*: define los mensajes comunes que se utilizan para interactuar con la pila de navegación.
- *sensor_msgs*: define mensajes para sensores de uso común, incluidas cámaras y telémetros laser de escaneo.
- Entre muchos otros.

Para más información en: http://wiki.ros.org/common_msgs

Continuando, en el repositorio que ya hemos bajado, navegamos hasta el siguiente directorio:

- /PATH/TO/Unity-Robotics-Hub/tutorials/pick_and_place

Seleccionamos y copiamos la carpeta “Scripts” en la carpeta “Assets” de nuestro proyecto en Unity. Ahora deberíamos de encontrar dos scripts C# en el directorio “Assets/Scripts”.

Veamos más a fondo que contienen estos scripts (es importante tratar de comprender que es lo que hacen estos scripts para poder tener una idea de que se está realizando):

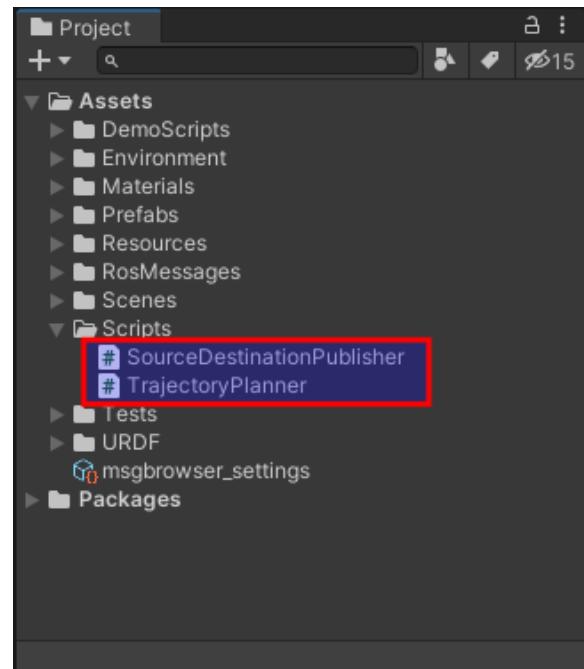
- SourceDestinationPublisher.cs

Este script se comunica con ROS, tomando las posiciones de los objetos, y enviándolo al tema ROS “SourceDestination_input”.

Se importan espacios de nombres (namespaces).

Se declaran variables.

En un método llamado “Start()” se obtiene una instancia de la clase “ROSConnection”.



El método que se usa para obtener dicha instancia de la clase “ROSConnection”, dentro el método, se le asigna el componente de tipo “ROSConnection” de un nuevo GameObject que se crea o de un Prefab (el Prefab es como un GameObject personalizado y listo para instanciar las veces que sea necesario).

Este Gameobject o Prefab solo se crea una vez cuando le damos play a la escena en Unity

En un método llamado “Publish()” se crea un objeto del tipo de clase para publicar.

Se toman todos los valores de las articulaciones y se guardan en campos respectivos del objeto.

Se toman las poses de los objetos (GameObject) y se guardan en campos respectivos del objeto.

Finalmente se envía el mensaje al “server_endpoint.py” que se ejecuta en ROS (más adelante hablaremos de este script “server_endpoint.py” que se encuentra en el paquete “ros_tcp_endpoint”).

- TrajectoryPlanner.cs

Al parecer este script se tratará más adelante, por tanto, lo comentaremos más adelante.

Regresando al editor de Unity, ahora agregaremos un objeto a la escena de Unity que ejecute el script “SourceDestinationPublisher.cs” (lo agregamos como componente al objeto).

Damos clic derecho en la ventana *Jerarquía* y seleccionamos la opción “Create Empty” para agregar un nuevo GameObject vacío el cual lo nombraremos como “Publisher”. Despues agregamos el componente que recién creamos llamado “SourceDestinationPublisher.cs” al objeto arrastrando el script desde la ventana *Proyecto* al objeto en la ventana *Jerarquía*.

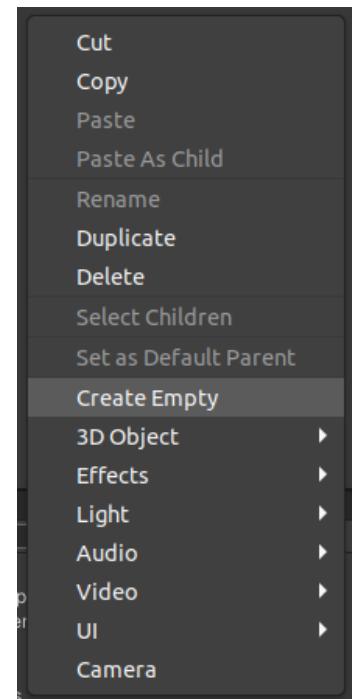
NOTA: Tener en cuenta que este componente muestra variables en la ventana *Inspector* que deben asignarse.

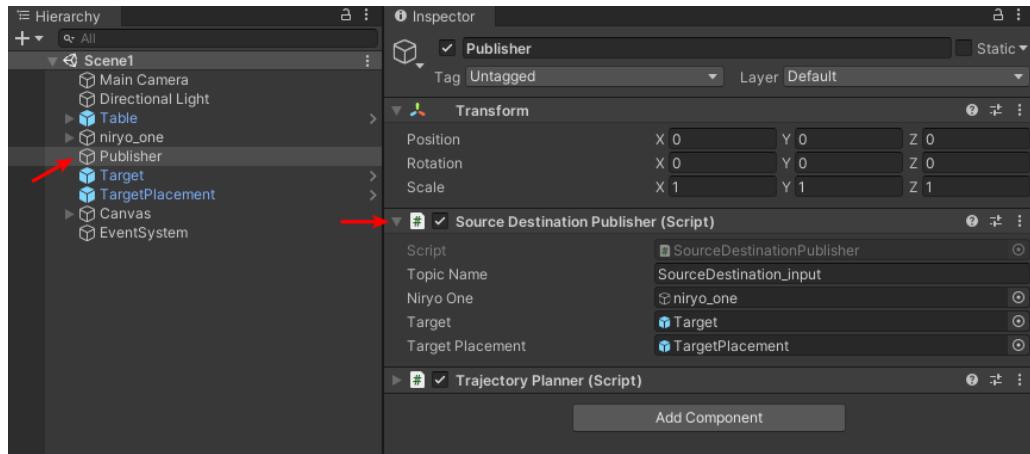
Para esto, asignamos cada objeto con su correspondiente variable (arrastramos de la ventana *Jerarquía* a la ventana *Inspector*).

Recordar que en la carpeta “Prefabs” tenemos lo siguiente:

- Table (mesa donde va situado el brazo robótico)
- Target (es un cubo)
- TargetPlacement (es la base donde se colocara el cubo)

Recordar que para pasar del espacio mundial Unity al espacio mundial ROS se requiere una conversion. Unity (x,y,z) es equivalente a la coordenada ROS (z,-x,y). Estas conversiones se proporcionan a través del componente “ROSGeometry” en el paquete “ROS-TCP-Connector”.





A continuación, se debe crear la conexión ROS TCP yendo a la barra del menú superior y seleccionar “Robotics -> ROS Settings”.

Esta configuración es utilizada por el script “ROSConnection.cs” del cual hablamos anteriormente (este es el script importante).

Veamos qué fue lo que configuramos:

- ROS IP Address

Colocamos esa dirección IP si estamos ejecutando ROS en un contenedor Docker.

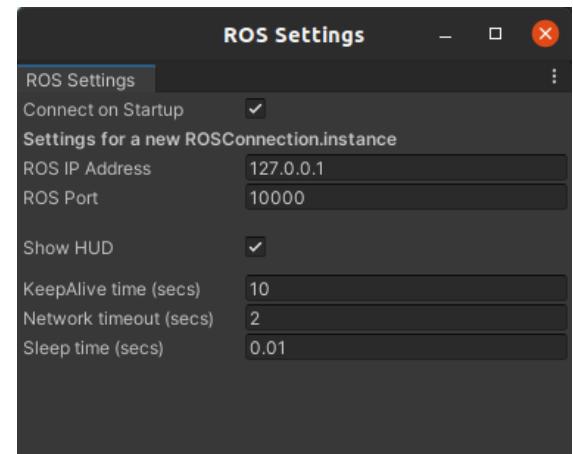
- ROS Port

Colocamos el puerto por el cual se hará la comunicación.

Hablemos al respecto del script “ROSConnection.cs”:

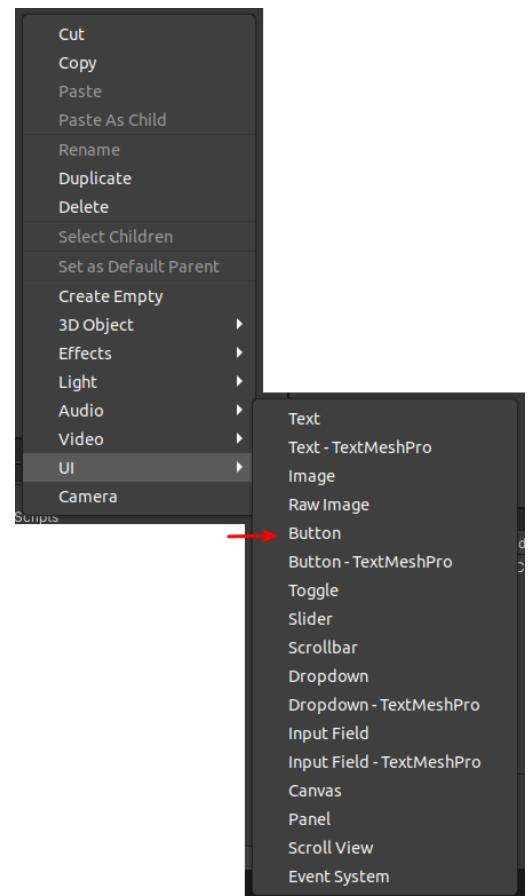
Es una clase bastante extensa de métodos, y entre esos métodos hay uno para la conexión, el cual utiliza los datos “ROS IP Address”, “ROS Port”, etc., para hacer la conexión (se utilizan algunas clases en C# sobre programación asíncrona, como por ejemplo: Task, Thread).

Más adelante hablaremos más con respecto a este script.



Ahora agregaremos un elemento de interfaz de usuario que permita al usuario activar la función “Publish()”, que recordemos se encuentra en el script “SourceDestinationPublisher.cs”.

Acomodamos el botón en la ventana Juego, después



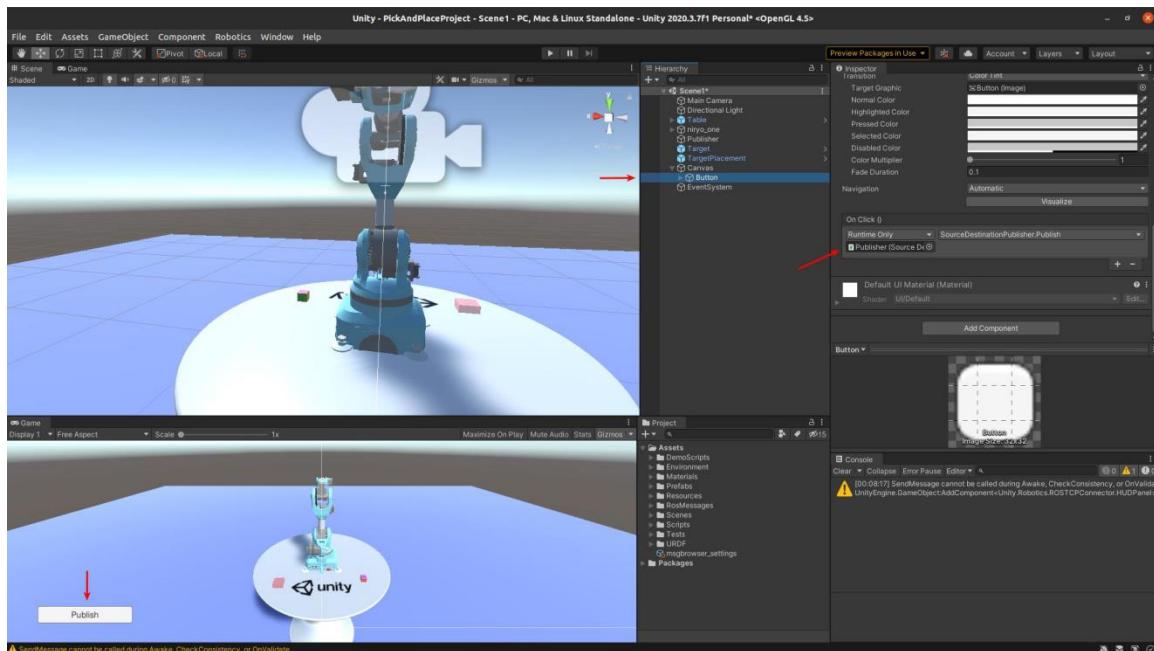
seleccionamos el botón recién creado y nos dirigimos a la ventana *Inspector* donde en el encabezado “onClick()” agregaremos un nuevo evento. Para esto seleccionamos el objeto “Publisher” en la ventana *Jerarquía* y lo arrastramos al nuevo evento que acabamos de agregar. Después hacemos clic en el menú desplegable donde dice “No Function” y seleccionamos:

“SourceDestinationPublisher -> Publish()”

Para cambiar el nombre del botón se modifica un objeto hijo de tipo “Text” seleccionándolo y en la ventana *Inspector*, en la propiedad “Text”, se cambia el nombre.

Este método “Publish()” del script “SourceDestinationPublisher.cs” contiene lo siguiente:

Se recuperan las posiciones de las articulaciones del brazo, del cubo y de la base, las cuales se almacenan en una variable de tipo de mensaje “MNiryoMoveitJoints” para después enviar el mensaje al “server_endpoint.py” corriendo en ROS.



Ya tenemos todo listo en Unity.

Nos dirigimos a nuestro espacio de trabajo ROS, aquí ejecutaremos un archivo “.launch” el cual iniciara los nodos ROS necesarios para la comunicación y cargara un archivo de parámetros al servidor de parámetros.

Ejecutamos el siguiente comando para crear un contenedor ROS Docker:

- `docker run -it --rm -p 10000:10000 -p 5005:5005 --name test unity-robotics:pick-and-place /bin/bash`

Una vez dentro, cargamos nuestros archivos de configuración y después ejecutamos el archivo de lanzamiento:

- `source devel/setup.bash`
- `roslaunch niryo_moveit part_2.launch`

Este archivo de lanzamiento contiene lo siguiente:

Se cargan parámetros al servidor de parámetros mediante un archivo “.yaml”.

Se declara un nodo con nombre “server_endpoint” que ejecuta el script “server_endpoint.py” del paquete “niryo_moveit”.

Recordar que la ejecución del comando “roslaunch” inicia automáticamente “roscore” si aún no se está ejecutando. “roscore” es una colección de nodos y programas que son requisitos previos de un sistema basado en ROS.

Contenido del script:

Se importa rospy, el cual es una biblioteca cliente de Python para ROS.

Se importan clases muy importantes como lo son “TcpServer”, “RosPublisher”, “RosSubscriber” y “RosService”, del módulo “ros_tcp_endpoint”.

Se importan clases de mensajes.

Se importan clases de servicios (de solicitud y respuesta, ya que, recordando, un mismo archivo .srv genera dos clases de mensajes).

Se inicializa un nodo ROS.

Se inicia el Server Endpoint con un diccionario de *objetos de comunicación ROS* para enrutar mensajes (estos objetos son “RosPublisher”, “RosSubscriber” y “RosService”). Este diccionario

El nombre del nodo especificado en la instrucción “`rospy.init_node(<name_node>)`” se sobrescribe con el que se especifica en el archivo de lanzamiento.

especifica los temas y los objetos de comunicación a los que

pertenecen. Por ejemplo:

Si Unity publicara en el tema “topic”:

clave: “topic”

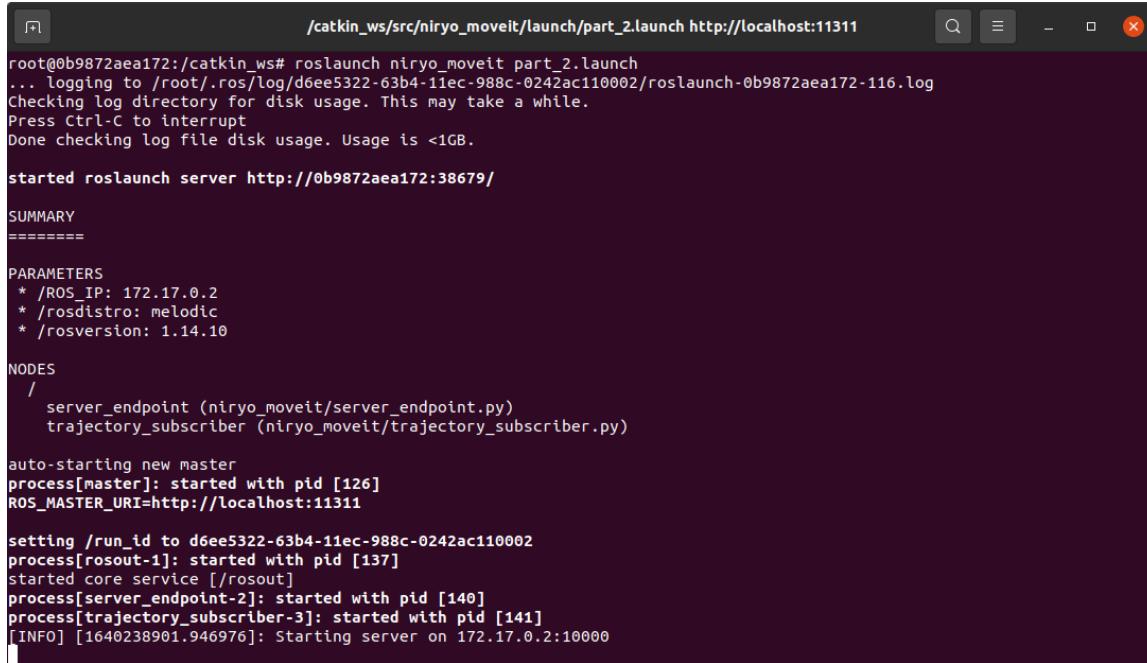
valor: RosPublisher(...) (tipo de objeto de comunicación)

Mediante una instrucción le decimos al nodo que entre en un bucle infinito hasta que reciba una señal de apagado, el cual, durante ese ciclo, procesara cualquier evento que ocurra.

Se declara un nodo con nombre “trajectory_subscriber” que ejecuta el script “trajectory_subscriber.py” del paquete “niryo_moveit”.

Este script se comentara más adelante (por el momento no lo hemos tocado).

Este comando imprimirá varios mensajes en la consola, incluidos los parámetros establecidos y los nodos lanzados.



```
/catkin_ws/src/niryo_moveit/launch/part_2.launch http://localhost:11311
root@0b9872aea172:/catkin_ws# roslaunch niryo_moveit part_2.launch
... logging to /root/.ros/log/d6ee5322-63b4-11ec-988c-0242ac110002/roslaunch-0b9872aea172-116.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://0b9872aea172:38679/

SUMMARY
=====

PARAMETERS
  * /ROS_IP: 172.17.0.2
  * /rostdistro: melodic
  * /rosversion: 1.14.10

NODES
  /
    server_endpoint (niryo_moveit/server_endpoint.py)
    trajectory_subscriber (niryo_moveit/trajectory_subscriber.py)

auto-starting new master
process[master]: started with pid [126]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to d6ee5322-63b4-11ec-988c-0242ac110002
process[rosout-1]: started with pid [137]
started core service [/rosout]
process[server_endpoint-2]: started with pid [140]
process[trajectory_subscriber-3]: started with pid [141]
[INFO] [1640238901.946976]: Starting server on 172.17.0.2:10000
```

Lo que hemos hecho es cargar los parámetros ROS especificados en el archivo de lanzamiento y hemos lanzado los nodos “server_endpoint.py” y “trajectory_subscriber”.

El archivo de parámetros que se carga al servidor de parámetros contiene lo

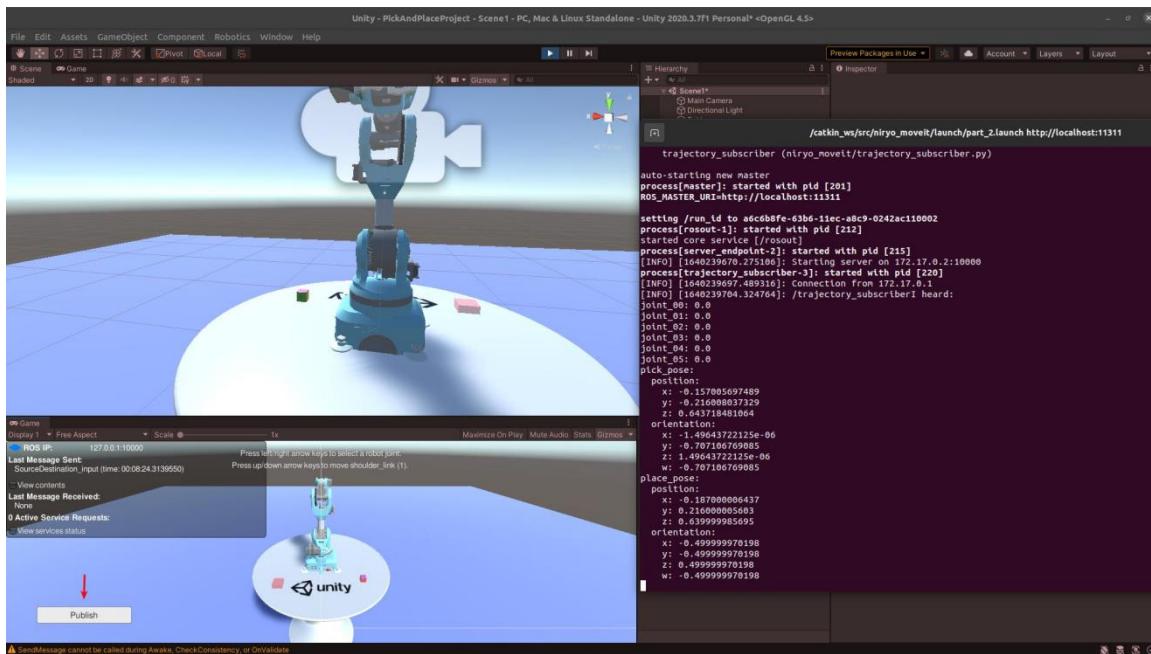
siguiente:

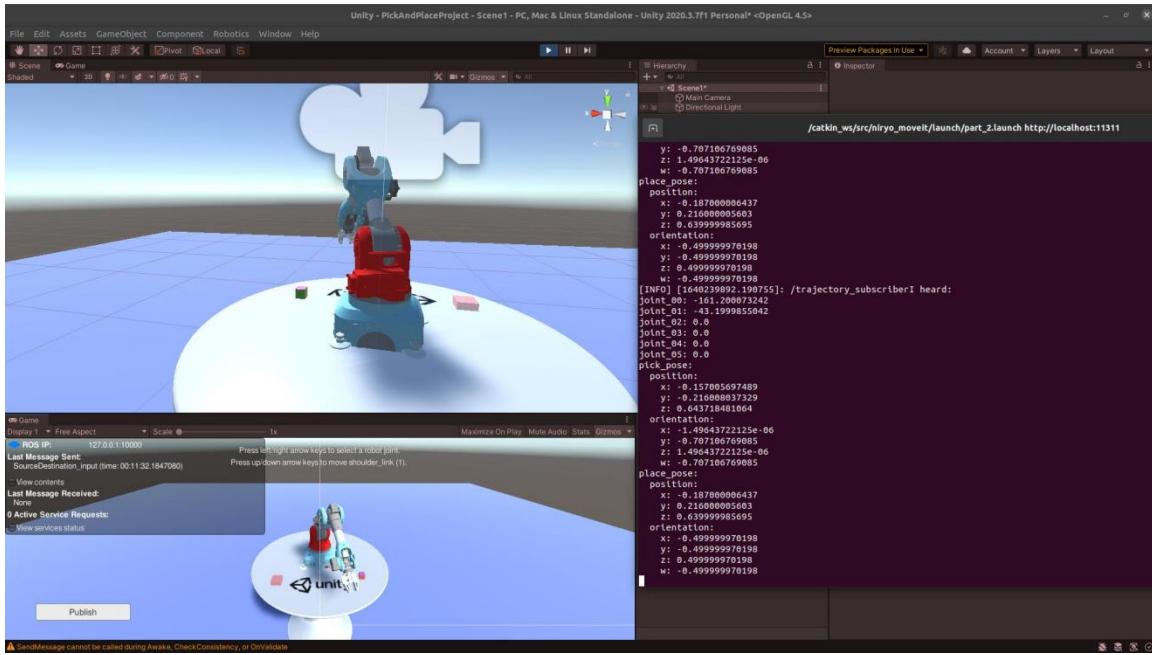
ROS_IP: 172.17.0.2

Esta es la dirección IP del contenedor (se obtiene usando el comando “hostname -i” en la terminal).

Regresamos a Unity y damos clic en play. Después llamamos a la función “Publish()” del script “SourceDestinationPublisher.cs” dando clic en el botón que agregamos a la ventana “Juego”, el cual publicara las poses de los objetos al tema ROS “SourceDestination_input”. Esto lo podemos observar en la terminal donde ejecutamos el archivo de lanzamiento.

Ahora tenemos el control de publicar en el tema cuando nosotros queramos, solamente dando clic en el botón “Publish()”.





NOTA: Al final de cada parte del tutorial, en el sitio web, vienen recursos bastante interesantes para complementar algunas partes, como por ejemplo, como están compuestos algunos paquetes (TCP endpoint, ROS TCP Connector, etc.), documentación, entre otros.

Tarea pick-and-place mediante el uso de ROS

Esta parte incluye la preparación y configuración necesarias para ejecutar una tarea de “pick and place” con poses conocidas usando *Movelit*. Los pasos cubiertos incluyen crear e invocar un servicio de planificación de movimiento ROS, mover un cuerpo de articulación en Unity basado en una trayectoria calculada y controlar una herramienta para agarrar y soltar un objeto con éxito (el objeto es un cubo).

El script antes mencionado llamado “TrajectoryPlanner.cs” es donde viene toda la lógica para invocar un servicio de planificación de movimiento, así como la lógica para controlar la herramienta de pinza del robot.

Veamos que contiene este script:

Se declaran bastantes variables (entre ellas una variable de tipo “ROSConnection”).

Métodos que se implementan

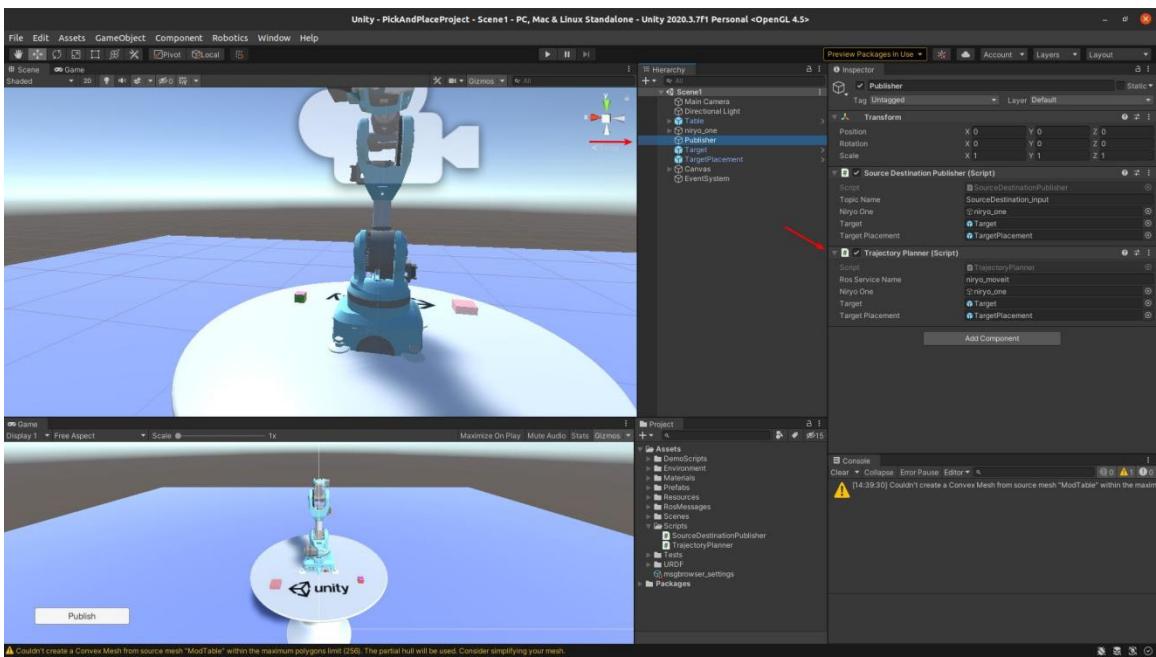
- CloseGripper

En C# una enumeracion o **enum**, es un tipo de dato que nos permite crear una variable, que define todos los posibles valores fijos que puede contener.

- **CloseGripper**
Se cierra la pinza estableciendo valores a la pinza izquierda y derecha para juntarse.
- **OpenGripper**
Se abre la pinza estableciendo valores a la pinza izquierda y derecha para separarse.
- **CurrentJointConfig**
Se devuelven los valores actuales de los ángulos de articulación del robot.
- **PublishJoints**
Se envía un mensaje a un servicio ROS con los valores actuales de los ángulos de articulación del robot y las posiciones y rotaciones actuales del cubo y de la base (donde se colocara el cubo), y se recibe una respuesta que es capturada por la función “TrajectoryResponse”, de la cual hablaremos a continuación.
Recordar que en un servicio se hace una solicitud y se recibe una respuesta.
- **TrajectoryResponse**
Las trayectorias (la respuesta del servicio) se pasan a la función “ExecuteTrajectories”, la cual se ejecuta como una corutina.
- **ExecuteTrajectories**
Aquí se ejecutan las trayectorias.
Se itera a través de las articulaciones para asignar un nuevo valor basado en la respuesta del servicio ROS, hasta que se hayan alcanzado las trayectorias objetivo. Según la asignación de pose, esta función puede llamar a los métodos “OpenGripper” o “CloseGripper” según sea necesario.
- **Start**
Se crea una instancia de ROSConnection (el cual es un método estático del propio script “ROSConnection.cs”).
Se buscan todas las articulaciones del robot y se guardan en variables.

En C# una **corutina** (coroutine) es como una función que tiene la capacidad de pausar la ejecución y devolver el control a Unity, pero luego continuar donde lo dejó en el siguiente cuadro (es como devolver un valor cada cuadro y permanecer dentro de la función hasta que ya no haya algo que devolver).

Volvemos a Unity y seleccionamos el GameObject llamado “Publisher” en la ventana *Jerarquía* y le agregamos el script “TrajectoryPlanner.cs” como un componente.

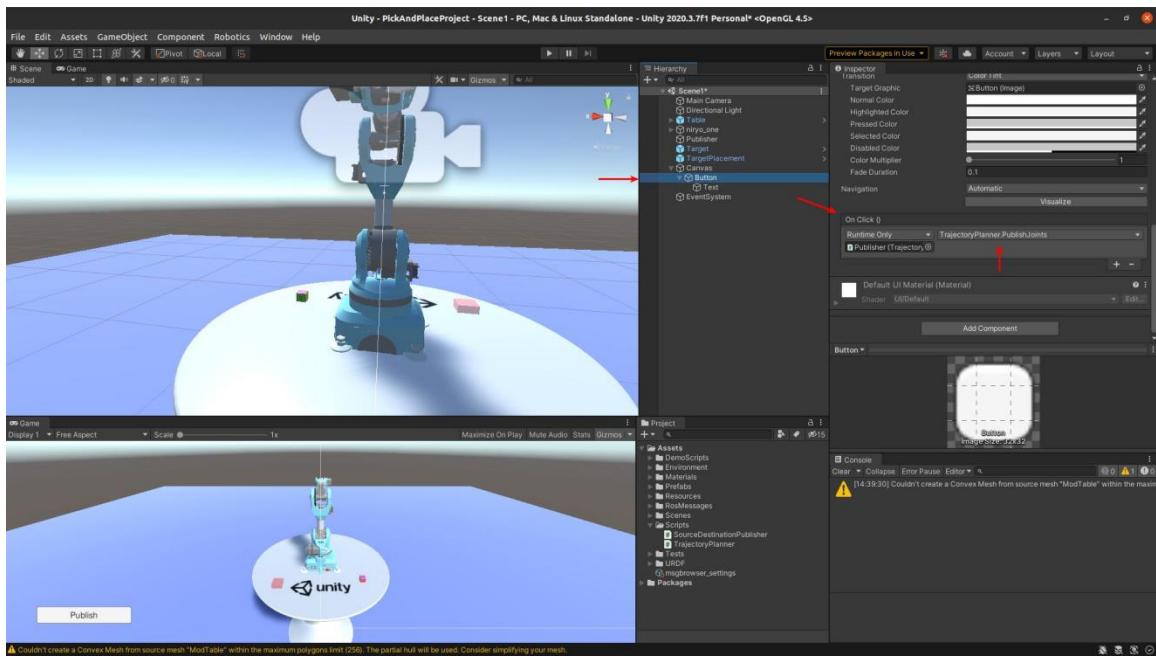


NOTA: Tener en cuenta que este componente muestra variables en la ventana *Inspector* que deben asignarse.

Ahora seleccionamos el objeto “Button” creado anteriormente en “Canvas/Button”.

Debajo del encabezado “OnClick()” reemplazamos ahora por:

“TrajectoryPlanner -> PublishJoints()”



Ya tenemos todo listo en Unity. Ahora nos dirigimos a nuestro espacio de trabajo ROS, aquí ejecutaremos un archivo de lanzamiento (“.launch”) llamado “part_3.launch” el cual iniciara los nodos ROS necesarios para la comunicación, entre otras cosas.

Este archivo de lanzamiento contiene lo siguiente:

Comando para editar: *rosed niryo_moveit part_3.launch*

Se carga un archivo de parámetros al servidor de parámetros.

Se declaran dos nodos:

- server_endpoint

Este nodo ejecuta el script “server_endpoint.py” (del que ya hablamos) del paquete “niryo_moveit”.

Sobre el script:

Se importan las clases de mensajes, servicios, entre otros.

Se inicia un nodo que sera el que ejecute el script.

Se crea una instancia de la clase “TcpServer” y se inicia el *server endpoint* con un diccionario de objetos de comunicacion ROS para enrutar mensajes.

El nodo se queda a la espera hasta que reciba una señal de apagado, donde estara procesando cualquier evento que ocurra.

- mover

Este nodo ejecuta el script “mover.py” del paquete “niryo_moveit”.

Sobre el script:

(Aquí se hace uso de MoveIt)

Se importan clases de mensajes, de servicios, entre otros.

Se definen algunas funciones:

- plan_trajectory

Dados los angulos de inicio del robot se planifica una trayectoria que termina en la pose de destino.

Aquí se llama al planificador para calcular el plan. Si existe un plan devuelve el plan, y si no, devuelve una excepción.

- plan_pick_and_place

(Este método es el que llama al anterior)

Se crea un plan de “recoger y colocar” (pick and place) utilizando los cuatro estados siguientes:

- Pre-Grasp (pre-agarre): se coloca la pinza directamente sobre el objeto (el cubo).

- Grasp (agarre): Se baja la pinza de modo que los dedos queden a ambos lados del objeto (el cubo).
- Pick-Up (recoger): Se levanta la pinza de nuevo a la posición “Pre-Grasp”.
- Place (colocar): Se mueve la pinza a la posición de colocación deseada.

NOTA: El comportamiento de la pinza (cerrar y abrir la pinza) se maneja fuera de esta planificación (recordemos que esta parte se controla desde el script C# llamado “TrajectoryPlanner.cs”).

Para cada estado se llama al metodo “plan_trajectory”.

Si la planificación de la trayectoria funciona para todos los estados/etapas de “recoger y colocar”, se agrega el plan a la respuesta.

■ moveit_server

Se inicializa un nodo.

Se crea un servicio creando una instancia “rospy.Service”.

El nodo se queda a la espera hasta que recibe una señal de apagado, donde estará procesando cualquier evento que ocurra.

Se incluye otro archivo de lanzamiento en el archivo actual (mediante la etiqueta “<include>”). Este archivo contiene lo siguiente:

Comando para editar: *rosed niryo_moveit demo.launch*

Se declara un argumento dentro del archivo (el cual solo es accesible desde este archivo).

Se incluyen otros archivos de lanzamiento en el archivo actual (lo interesante aquí es cómo podemos ir agregando más funcionalidad agregando archivos de lanzamiento).

Algunos de estos inicializan nodos, importan otros archivos de lanzamiento, se crean grupos en donde se declaran parámetros y se cargan parámetros, etc.

Se declaran parámetros para el servidor de parámetros (mediante la etiqueta “<param>”).

Como hemos visto, la ejecución de un simple archivo de lanzamiento ocasiona que se

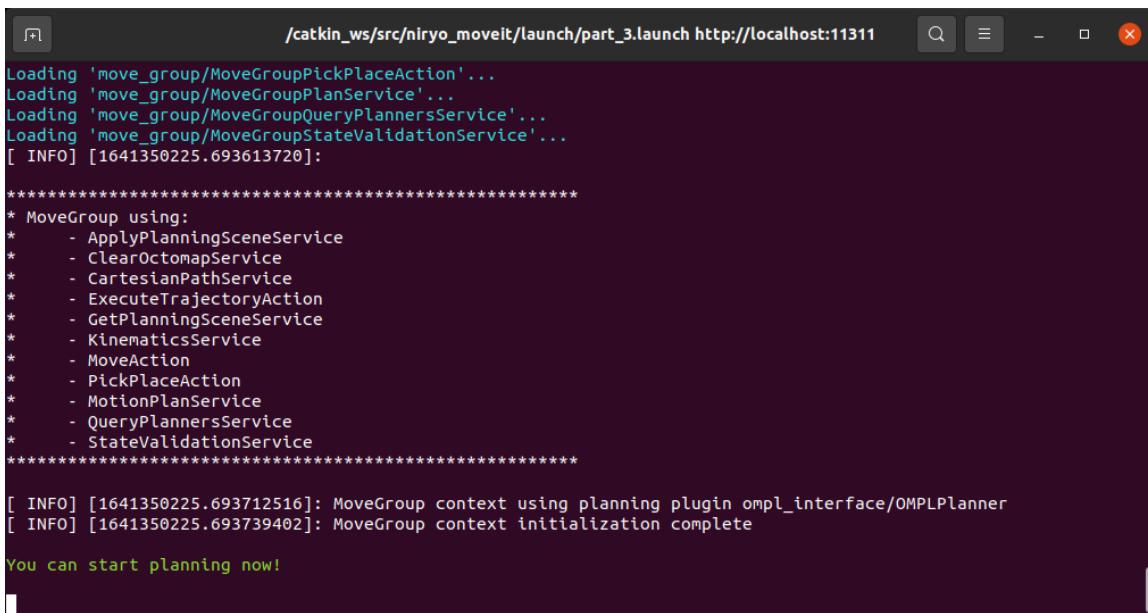
inician muchos nodos, se declaren y carguen parámetros, cada archivo puede llamar a otros archivos de lanzamiento, etc. Se vuelve algo muy grande y bastante interesante cuando deseamos entender cómo funciona todo esto.

Ahora sí, ejecutamos el siguiente comando dentro de nuestro espacio de trabajo ROS (dentro del contenedor Docker, recordando):

```
roslaunch niryo_moveit part_3.launch
```

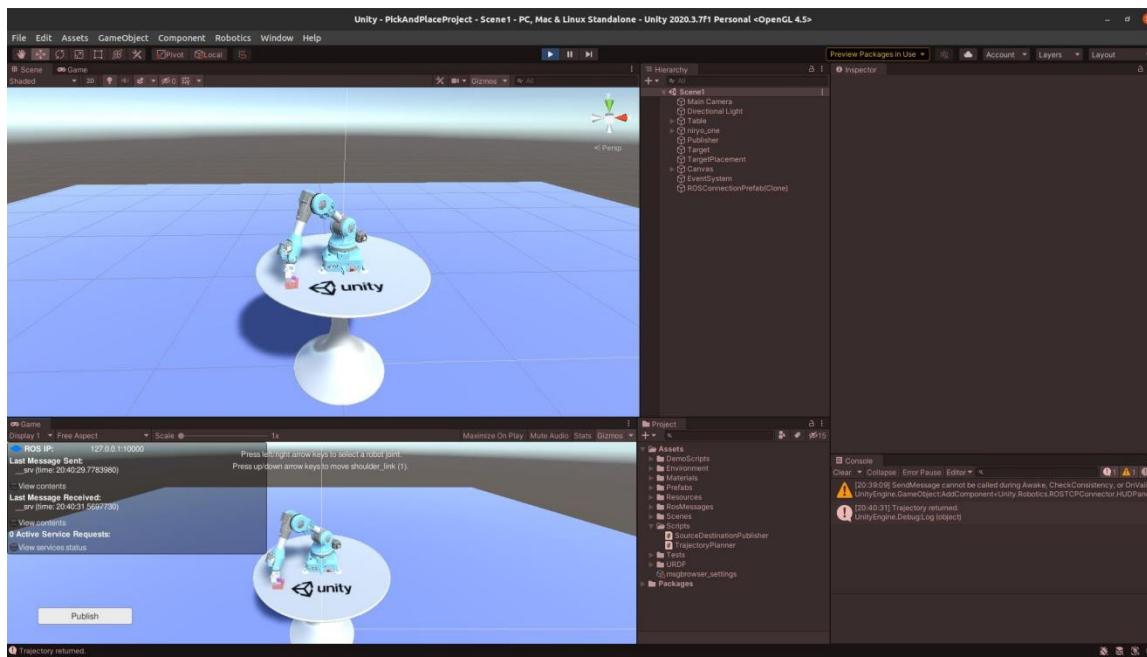
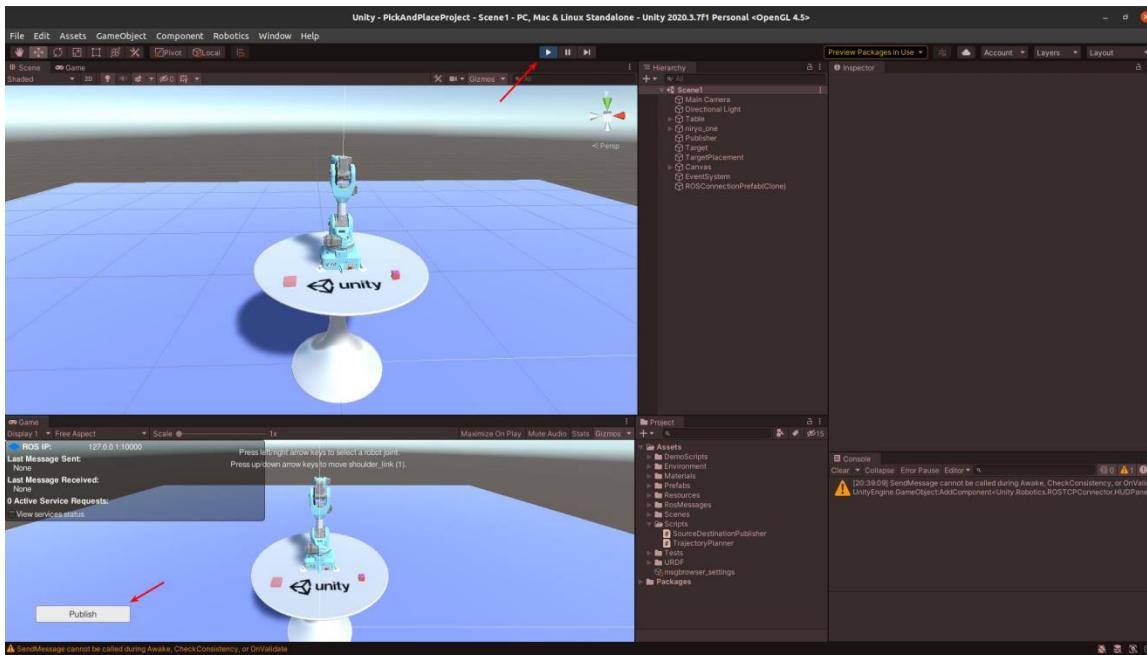
Este comando imprimirá varios mensajes en la consola, incluidos los parámetros establecidos y los nodos lanzados.

Recordemos que **MoveIt** incorpora planificación de movimiento, manipulación, percepción 3D, cinemática, control y navegación.



```
/catkin_ws/src/niryo_moveit/launch/part_3.launch http://localhost:11311
Loading 'move_group/MoveGroupPickPlaceAction'...
Loading 'move_group/MoveGroupPlanService'...
Loading 'move_group/MoveGroupQueryPlannersService'...
Loading 'move_group/MoveGroupStateValidationService'...
[ INFO] [1641350225.693613720]: [ INFO] [1641350225.693613720]: [ INFO] [1641350225.693712516]: MoveGroup context using planning plugin ompl_interface/OMPLPlanner
[ INFO] [1641350225.693739402]: MoveGroup context initialization complete
You can start planning now!
```

Regresamos a Unity y presionamos el botón de play seguido del botón “Publish”. Si todo va bien, veremos como el brazo robot levanta el cubo y lo coloca en la plataforma de destino.



```

You can start planning now!

[INFO] [1641350349.407675]: Connection from 172.17.0.1
[INFO] [1641350429.804987192]: Loading robot model 'niryo_one'...
[INFO] [1641350429.807201891]: No root/virtual joint specified in SRDF. Assuming fixed joint
[WARN] [1641350429.862502106]: Kinematics solver doesn't support #attempts anymore, but only a timeout.
Please remove the parameter '/robot_description_kinematics/arm/kInematics_solver_attempts' from your configuration.
[INFO] [1641350430.975943605]: Ready to take commands for planning group arm.
[INFO] [1641350431.193553427]: Planning request received for MoveGroup action. Forwarding to planning pipeline.
[INFO] [1641350431.222334625]: Planner configuration 'arm' will use planner 'geometric::RRTConnect'. Additional configuration parameters will be set when the planner is constructed.
[INFO] [1641350431.223421264]: RRTConnect: Starting planning with 1 states already in datastructure
[INFO] [1641350431.245295475]: RRTConnect: Created 5 states (2 start + 3 goal)
[INFO] [1641350431.245407985]: Solution found in 0.022136 seconds
[INFO] [1641350431.255025294]: SimpleSetup: Path simplification took 0.009478 seconds and changed from 4 to 2 states
[INFO] [1641350431.293598977]: Planning request received for MoveGroup action. Forwarding to planning pipeline.
[INFO] [1641350431.294274974]: Planner configuration 'arm' will use planner 'geometric::RRTConnect'. Additional configuration parameters will be set when the planner is constructed.
[INFO] [1641350431.30560958]: RRTConnect: Starting planning with 1 states already in datastructure
[INFO] [1641350431.305762216]: RRTConnect: Created 5 states (2 start + 2 goal)
[INFO] [1641350431.306142057]: Solution found in 0.01631 seconds
[INFO] [1641350431.311996544]: SimpleSetup: Path simplification took 0.005553 seconds and changed from 3 to 2 states
[INFO] [1641350431.353958474]: Planning request received for MoveGroup action. Forwarding to planning pipeline.
[INFO] [1641350431.354749708]: Planner configuration 'arm' will use planner 'geometric::RRTConnect'. Additional configuration parameters will be set when the planner is constructed.
[INFO] [1641350431.35520173]: RRTConnect: Starting planning with 1 states already in datastructure
[INFO] [1641350431.366694586]: RRTConnect: Created 5 states (2 start + 3 goal)
[INFO] [1641350431.367174115]: Solution found in 0.012194 seconds
[INFO] [1641350431.372690642]: SimpleSetup: Path simplification took 0.005150 seconds and changed from 4 to 2 states
[INFO] [1641350431.423686403]: Planning request received for MoveGroup action. Forwarding to planning pipeline.
[INFO] [1641350431.4242306503]: Planner configuration 'arm' will use planner 'geometric::RRTConnect'. Additional configuration parameters will be set when the planner is constructed.
[INFO] [1641350431.424694287]: RRTConnect: Starting planning with 1 states already in datastructure
[INFO] [1641350431.466077319]: RRTConnect: Created 5 states (2 start + 3 goal)
[INFO] [1641350431.466186260]: Solution found in 0.041587 seconds
[INFO] [1641350431.471952947]: SimpleSetup: Path simplification took 0.005694 seconds and changed from 4 to 2 states

```

NOTA: Ambos objetos se pueden cambiar de posición durante el tiempo de ejecución para diferentes cálculos de trayectoria (tener cuidado de no dejar muy retirados los objetos del brazo robótico porque de ser así no será posible una trayectoria planeada).

Tutoriales de iniciación: Integración ROS-Unity

La propia documentación ya ofrece una serie de tutoriales, los cuales se encuentran en el siguiente sitio:

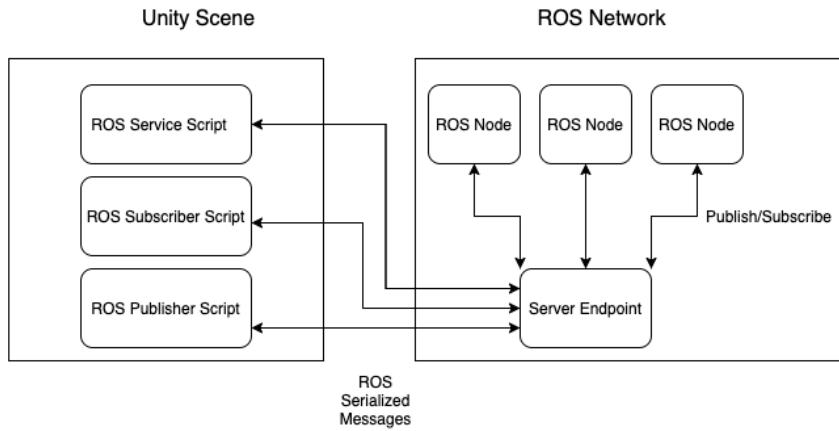
- https://github.com/Unity-Technologies/Unity-Robotics-Hub/blob/main/tutorials/ros_unity_integration/README.md

NOTA: De aquí en adelante ya estaremos usando versiones más recientes de los paquetes que se encuentran en estos enlaces:

- Paquetes proyecto (**v0.6.0**):
<https://github.com/Unity-Technologies/Unity-Robotics-Hub>
- Paquetes “ROS TCP Connector” (**v0.6.0**) y “URDF-Importer”:
https://github.com/Unity-Technologies/Unity-Robotics-Hub/blob/main/tutorials/quick_setup.md
- Paquete “Visualizations”:
<https://github.com/Unity-Technologies/ROS-TCP-Connector/blob/main/com.unity.robotics.visualizations/Documentation~/README.md>

Por el momento basta con tener los paquetes del proyecto y el paquete “ROS TCP Connector” (los otros son opcionales y por el momento no estaremos trabajando con ellos, pero si es importante que se mencionen).

Configuración de espacio de trabajo ROS usando Docker



En Unity, el complemento **“ROSConnection.cs”** (del paquete **ROS TCP CONNECTOR**), del cual ya hemos hablado en repetidas ocasiones y que es bastante importante, proporciona los scripts necesarios para publicar, suscribirse o llamar a un servicio.

Configuraremos ROS para la integración con Unity.

Sera algo similar la construcción de una nueva imagen Docker a como se realizó en el tutorial de “pick and place”.

En el directorio donde me encuentro tengo las siguientes carpetas:

Estos archivos se encuentran en mi repositorio en **GitHub**:

<https://github.com/ChuyFernandez/ROS-Tutoriales-de-iniciacion-Integracion-ROS-Unity.git>

(el material presente en mi repositorio ya esta modificado, sobre todo en la parte de la programación, esto con el motivo de adaptarlo a mis necesidades e intentar hacerlo diferente a como se nos presenta)

Este material es proporcionado por el propio sitio web, el cual tiene archivos de ayuda para llevar a cabo los tutoriales.

Lo que estaremos haciendo en el transcurso será estar modificando el código y adaptándolo a nuestras necesidades. Lo más importante es que estaremos comentando el código y explicando la

GitHub es un servicio basado en la nube que aloja un sistema de control de versiones (VCS) llamado **Git**. Este repositorio online gratuito nos permite gestionar proyectos y controlar versiones de código.

funcionalidad del mismo.

- ros_docker

- Dockerfile
- set-up-workspace

Estos archivos son similares a los que se utilizaron en el tutorial de “pick and place” pero con algunas modificaciones para adaptarlos a este tutorial.

- ros_packages

NOTA: Es buena práctica separar un paquete en dos partes, separando la declaración de los mensajes del código.

- unity_robotics_demo
 - CmakeLists.txt (archivo)
 - package.xml (archivo)
 - launch (carpeta)
 - scripts (carpeta)

Otras carpetas que tambien se agregan son: “actions”, “src” (que por el momento no las incluimos).

- unity_robotics_demo_msgs
 - CmakeLists.txt (archivo)
 - package.xml (archivo)
 - msg (carpeta)
 - srv (carpeta)

- ros_tcp_endpoint (En realidad tiene por nombre “ROS-TCP-Endpoint” pero le cambiamos el nombre)
(v0.6.0)

Este paquete no viene incluido pero lo podemos obtener del siguiente enlace:

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

Ya sea que lo clonemos en la carpeta “ros_packages” o lo descarguemos como un archivo comprimido y lo descomprimamos en la carpeta “ros_packages”,

Bibliotecas cliente para ROS:

- rospy (Python)
(otra es “rclpy”)
- roscpp (C++)

El codigo de las bibliotecas cliente para ROS se puede encontrar en el siguiente enlace (por si se desea saber que hay detras de estas bibliotecas):

https://github.com/ros/ros_comm/tree/melodic-devel/clients

Aqui vienen todos los metodos que estaremos usando en el coigo que implementemos a lo largo de estos tutoriales.

Con estos paquetes estaremos trabajando, incluyendo además el paquete ROS “ros_tcp_endpoint” (paquete ROS utilizado para crear un server endpoint para

aceptar mensajes ROS enviados desde una escena de Unity usando los scripts de ROS TCP Connector).

Para más información sobre ROS TCP Connector en el siguiente sitio:

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

Lo que haremos sera construir una nueva imagen con el Dockerfile que se encuentra en “./ros_docker/Dockerfile” ejecutando el siguiente comando:

`docker build -t chuy7/ros:tutorial-integracion-ros-unity -f ros_docker/Dockerfile .`

El *tag* que le agregamos a la imagen hace ilusión al repositorio que tengo en *Docker Hub*, esto para facilitar el trabajo a la hora de subir la imagen terminada a mi repositorio y que pueda ser accedido por el publico en general.

`-t <mi-repositorio>:<algun-nombre>`

Tendremos una salida como la siguiente en la terminal si todo sale bien:

```
[100%] Built target unity_robots_demo_msgs_generate_messages_py
[100%] Built target unity_robots_demo_msgs_generate_messages_eus
Scanning dependencies of target unity_robots_demo_msgs_generate_messages
[100%] Built target unity_robots_demo_msgs_generate_messages
Base path: /catkin_ws
Source space: /catkin_ws/src
Build space: /catkin_ws/build
Devel space: /catkin_ws/devel
Install space: /catkin_ws/install
Creating symlink "/catkin_ws/src/CMakeLists.txt" pointing to "/opt/ros/melodic/share/catkin/cmake/toplevel.cmake"
#####
##### Running command: "cmake /catkin_ws/src -DCATKIN_DEVEL_PREFIX=/catkin_ws/devel -DCMAKE_INSTALL_PREFIX=/catkin_ws/install -G Unix Makefiles" in "/catkin_ws/build"
#####
#####
##### Running command: "make -j4 -l4" in "/catkin_ws/build"
#####
Removing intermediate container 99fbcf89c33c
--> 9db6d5d40cd5
Step 7/9 : WORKDIR $ROS_WORKSPACE
--> Running in edecaa77a157
Removing intermediate container edecaa77a157
--> 855d291081ba
Step 8/9 : RUN echo ". devel/setup.bash" >> ~/.bashrc
--> Running in 83380c68879f
Removing intermediate container 83380c68879f
--> c69003f24e62
Step 9/9 : RUN chmod +x src/roslaunch/src/roslaunch/*.py
--> Running in 57a642c9781b
Removing intermediate container 57a642c9781b
--> 5ce5a9317574
Successfully built 5ce5a9317574
Successfully tagged chuy7/ros:tutorial-integracion-ros-unity
```

Ya construida la imagen, corremos un nuevo contenedor en base a la imagen que acabamos de construir ejecutando el siguiente comando:

```
docker run -it --rm -p 10000:10000 -p 5005:5005
chuy7/ros:tutorial-integracion-ros-unity /bin/bash
```

```

chuy@chuy:~/Practicas/ROS Exercises/Servicio Social/Tutoriales de iniciacion - Integracion ROS-Unity$ docker run -it --rm -p 10000:10000 -p 5005:5005 chuy7/rostutorial-integracion-ros-unity /bin/bash
root@3d418a77c048:/catkin_ws# ls -l
total 12
drwxr-xr-x 11 root root 4096 Jan 20 06:05 build
drwxr-xr-x  5 root root 4096 Jan 20 06:05 devel
drwxr-xr-x  1 root root 4096 Jan 20 06:05 src
root@3d418a77c048:/catkin_ws# ls -l src/
total 12
lrwxrwxrwx 1 root root   50 Jan 20 06:05 CMakeLists.txt -> /opt/ros/melodic/share/catkin/cmake/toplevel.cmake
drwxr-xr-x  1 root root 4096 Jan 20 06:01 ros_tcp_endpoint
drwxrwxr-x  4 root root 4096 Dec 12 01:17 unity_robots_demo
drwxrwxr-x  4 root root 4096 Dec 12 01:17 unity_robots_demo_msgs
root@3d418a77c048:/catkin_ws# 

```

Para cargar volúmenes (esto nos ayuda a hacer persistentes nuestros datos, es decir, lo que agreguemos, modifiquemos o eliminemos dentro del contenedor también se verá reflejado en los archivos/directorios en nuestro ordenador, es como “mapear” nuestros archivos/directorios de nuestro ordenador con archivos/directorios dentro del contenedor) ejecutaríamos el siguiente comando:

NOTA: Nos aseguramos de estar en el directorio donde creamos la imagen.

```

docker run -it --rm -p 10000:10000 -p 5005:5005 -v
"$(pwd)/ros_packages/ros_tcp_endpoint:/catkin_ws/src/ros_tcp_endpoint" -v
"$(pwd)/ros_packages/unity_robots_demo:/catkin_ws/src/unity_robots_demo"
" -v
"$(pwd)/ros_packages/unity_robots_demo_msgs:/catkin_ws/src/unity_robots_demo_msgs" chuy7/rostutorial-integracion-ros-unity /bin/bash

```

De esta manera hemos logrado mapear nuestros paquetes que se encuentran en nuestro ordenador con los paquetes que se encuentran dentro del contenedor. Así podremos realizar cambios desde nuestro ordenador dentro de la carpeta y se verán reflejados los cambios en el contenedor.

Podemos crear un archivo ***docker-compose*** para evitar escribir el comando anterior en la terminal cada vez que queramos iniciar un contenedor, para que tan solo necesitemos ejecutar este archivo. El archivo quedaría de la siguiente manera:

NOTA: Es necesario instalar ***docker-compose***.

(Recordar estar ubicados en el directorio antes mencionado, ya que allí será donde se cree el archivo ***docker-compose.yaml***)

docker-compose es una herramienta para definir y ejecutar aplicaciones Docker multicontenedor que permite simplificar el uso de Docker a partir de archivos YAML.

```

version: "3"
services:
  ros:
    image: chuy7/ros:tutorial-integracion-ros-unity
    ports:
      - 10000:10000          (puerto ROS)
      - 5005:5005           (puerto Unity)
    volumes:
      - ./ros_packages/unity_robots_demo:/catkin_ws/src/unity_robots_de
        mo
      - ./ros_packages/unity_robots_demo_msgs:/catkin_ws/src/unity_robots_
        demo_msgs
      - ./ros_packages/ros_tcp_endpoint:/catkin_ws/src/ros_tcp_endpoint

```

NOTA: De forma predeterminada, el nodo “server_endpoint” escuchara en el puerto 10000.

Comando a ejecutar:

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

NOTA: Es importante resaltar que si usamos “run” en lugar de “up” debemos agregar la bandera “--service-ports” para que se expongan los puertos, ya que de lo contrario no se expondran.

Utilizar un archivo “docker-compose” podría ser necesario si necesitáramos lanzar uno o varios contenedores como servicios. En nuestro caso no es así, pero tenemos la libertad de elegir una de estas opciones.

TIP: Para obtener un shell en un contenedor en ejecución:

```
docker exec -it <nombre-contenedor> /bin/bash
```

Ahora, ingresamos a nuestro espacio de trabajo ROS ejecutando el siguiente comando (es importante estar ubicados donde tenemos el archivo *docker-compose.yaml*):

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

Dentro de nuestro espacio de trabajo ROS tendremos tres carpetas:

```

root@ce6c0e350cec:/catkin_ws# ls -l
total 12
drwxr-xr-x 11 root root 4096 Jan 20 06:05 build
drwxr-xr-x  5 root root 4096 Jan 20 06:05 devel
drwxr-xr-x  1 root root 4096 Jan 20 06:05 src
root@ce6c0e350cec:/catkin_ws# ls -l src/
total 12
lrwxrwxrwx 1 root root  50 Jan 20 06:05 CMakeLists.txt -> /opt/ros/melodic/share/catkin/cmake/toplevel.cmake
drwxr-xr-x  1 root root 4096 Jan 20 06:01 ros_tcp_endpoint
drwxrwxr-x  4 1000 1000 4096 Dec 12 01:17 unity_robots_demo
drwxrwxr-x  4 1000 1000 4096 Dec 12 01:17 unity_robots_demo_msgs
root@ce6c0e350cec:/catkin_ws#

```

Las carpetas “build” y “devel” son generadas despues de ejecutar el comando “catkin_make”, el cual es llamado a la hora de construir la imagen (especificamente, se llama en el script “*set-up-workspace*” ya mencionado). Ademas el comando anterior construira cualquier paquete ubicado en la carpeta “src”.

La carpeta “src” es la carpeta que creamos al momento de crear la imagen (dicha carpeta tiene paquetes ROS, los cuales ya se mencionaron).

La estructura básica de un paquete en ROS es similar a la siguiente:

NOTA: Recordar que es buena práctica separar un paquete en dos partes, separando la declaración de los mensajes del código.

(revisaremos el paquete “unity_robots_demo” y el paquete “unity_robots_demo_msgs”)

```

root@ce6c0e350cec:/catkin_ws# ls -l src/unity_robots_demo
total 16
-rw-rw-r-- 1 1000 1000 750 Dec 12 01:17 CMakeLists.txt
drwxrwxr-x 2 1000 1000 4096 Dec 12 01:17 launch
-rw-rw-r-- 1 1000 1000 1198 Dec 12 01:17 package.xml
drwxrwxr-x 2 1000 1000 4096 Dec 12 01:17 scripts
root@ce6c0e350cec:/catkin_ws# ls -l src/unity_robots_demo_msgs/
total 16
-rw-rw-r-- 1 1000 1000 771 Dec 12 01:17 CMakeLists.txt
drwxrwxr-x 2 1000 1000 4096 Dec 12 01:17 msg
-rw-rw-r-- 1 1000 1000 889 Dec 12 01:17 package.xml
drwxrwxr-x 2 1000 1000 4096 Dec 12 01:17 srv
root@ce6c0e350cec:/catkin_ws#

```

Veamos mas a fondo esta estructura de carpetas y archivos:

(aunque aun faltan algunas otras carpetas como “src”, “actions”, “config”)

- CMakeLists.txt (archivo)

Este archivo es la entrada al sistema de compilación “CMake” para compilar paquetes de software.

Para mas informacion consultar:

<http://wiki.ros.org/catkin/CMakeLists.txt>

- package.xml (archivo)

Este archivo XML debe incluirse en la carpeta raíz de cualquier paquete compatible con catkin (por defecto se genera solo después de la instrucción “*catkin_make*”, de la cual ya se habló).

En este archivo se definen propiedades sobre el paquete, como el nombre del paquete, los números de versión, los autores, las dependencias del paquete, entre otros.

Para mas información consultar:

<http://wiki.ros.org/catkin/package.xml>

- launch (carpeta)

Aqui se encontraran todos los archivos de lanzamiento que vayamos creando.

Los archivos “.launch” son muy comunes en ROS. Proporcionan una forma conveniente de iniciar varios nodos y un maestro, asi como tambien la configuración de parámetros, entre otros.

El comando “*roslaunch*” se utiliza para abrir archivos de lanzamiento.

- msg (carpeta)

Aqui se encontraran todas las declaraciones de mensajes que vayamos creando.

Los archivos “.msg” son un lenguaje de descripción de mensajes simplificado para describir los valores de los datos que publican los nodos ROS.

Ejemplo:

```
float32 pos_x
float32 pos_y
float32 pos_z
```

- srv (carpeta)

Aqui se encontraran todas las declaraciones de servicios que vayamos creando.

Los archivos “.srv” son un lenguaje de descripción de servicios simplificado para describir los tipos de servicios ROS.

Ejemplo:

Un archivo de descripción de servicio consta de un tipo de mensaje de solicitud y un tipo de mensaje de respuesta, separados por “---” (tres guiones medios).

string str	(tipo de mensaje de solicitud)
<hr/>	
string str	(tipo de mensaje de respuesta)

- scripts (carpeta)

Aqui se encontraran todos los scripts de Python que vayamos creando (en una carpeta llamada “src” se encontrarían todos los scripts de C++ que se vayan creando).

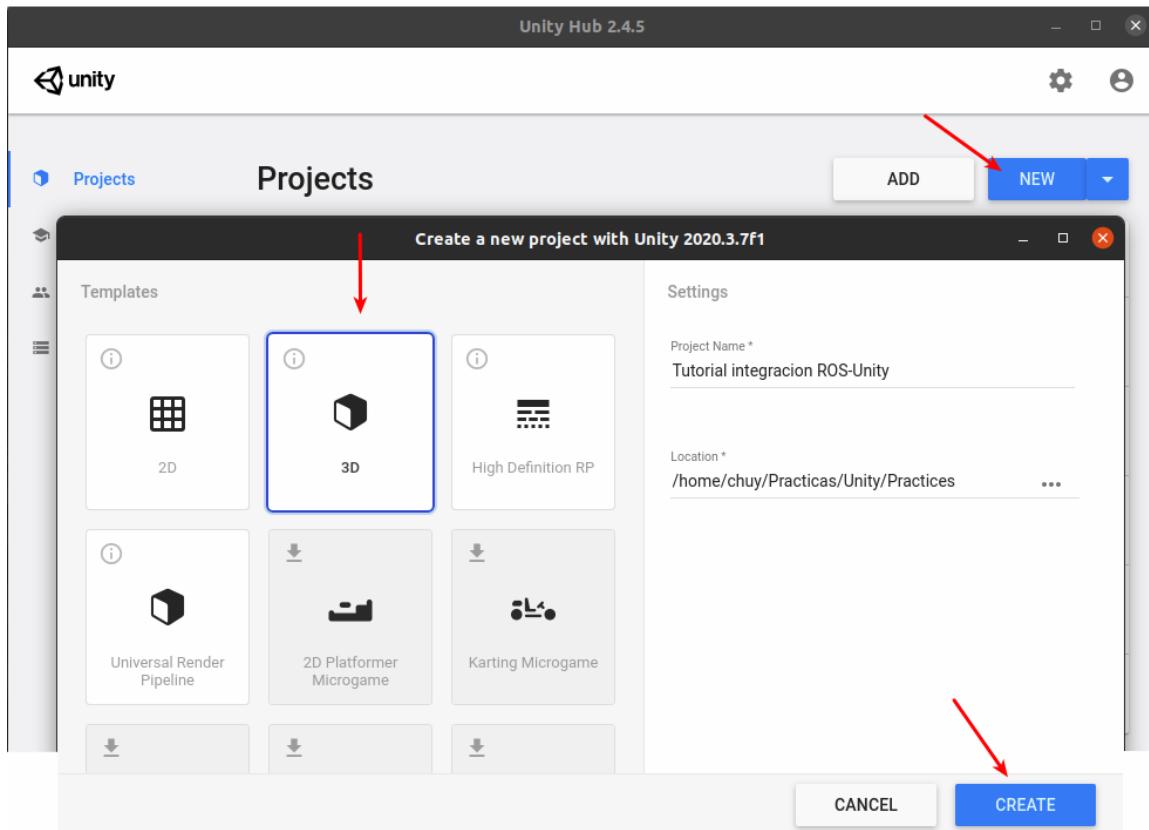
Hasta aquí ya tenemos nuestro espacio de trabajo ROS en Docker listo para trabajar (recordar que esta nueva imagen es muy aparte de la que se generó para el tutorial de “pick and place”).

Publicación en un tema desde una escena de Unity

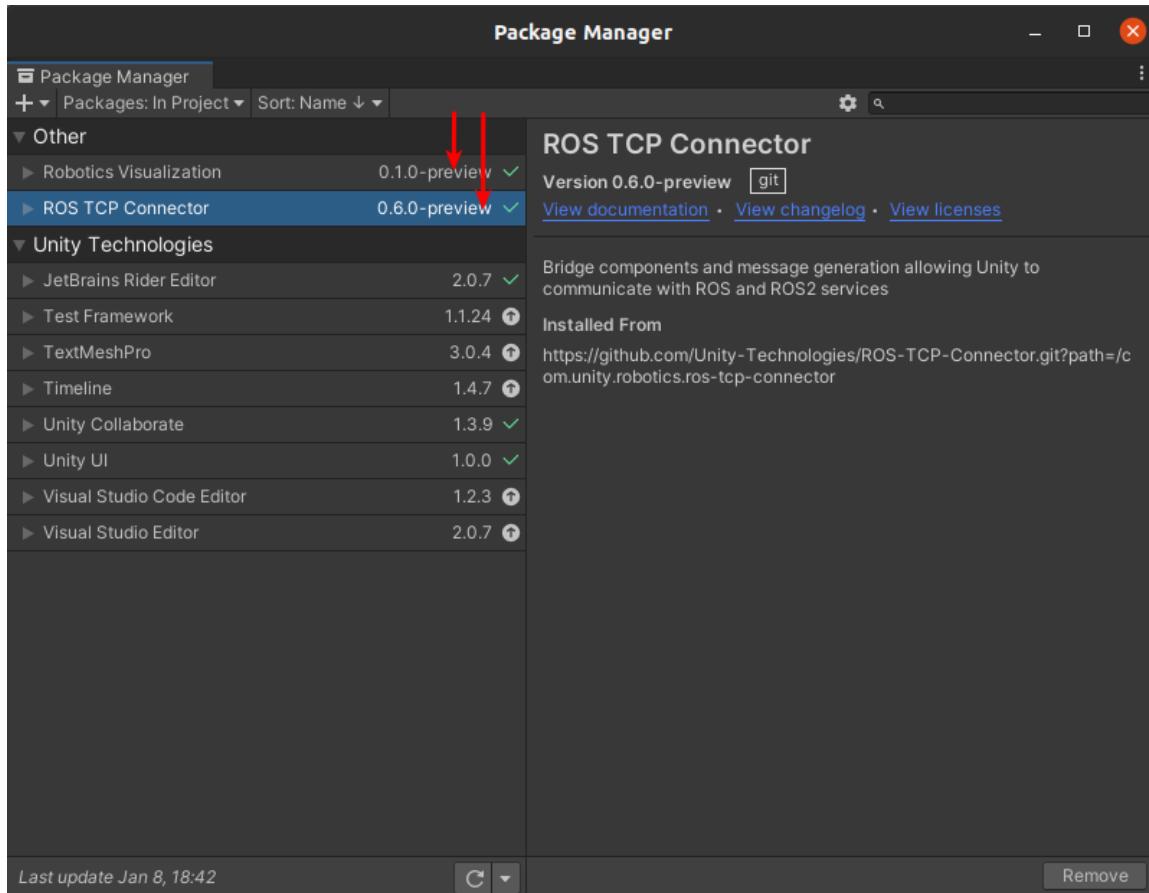
Crearemos una escena de Unity simple que publique la posición y rotación de un “GameObject” en un tema ROS.

Empezamos creando un nuevo proyecto en Unity, al cual le instalaremos los paquetes de ROS TCP Connector (los que se indican en este sitio web: <https://github.com/Unity-Technologies/ROS-TCP-Connector>).

NOTA: Recordar que esto ya lo hicimos para el tutorial de “pick and place”, asi que es solo cuestion de volverlo a hacer, ya que es un nuevo proyecto de Unity.



A continuación agregamos los paquetes necesarios que anteriormente se mencionaron
NOTA: El paquete “Robotics Visualizaion” es opcional.



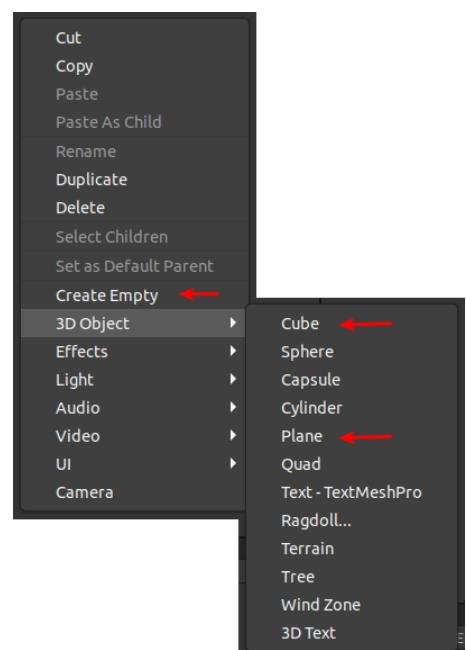
Ahora, agregamos los siguientes objetos (“GameObject”) a la escena:

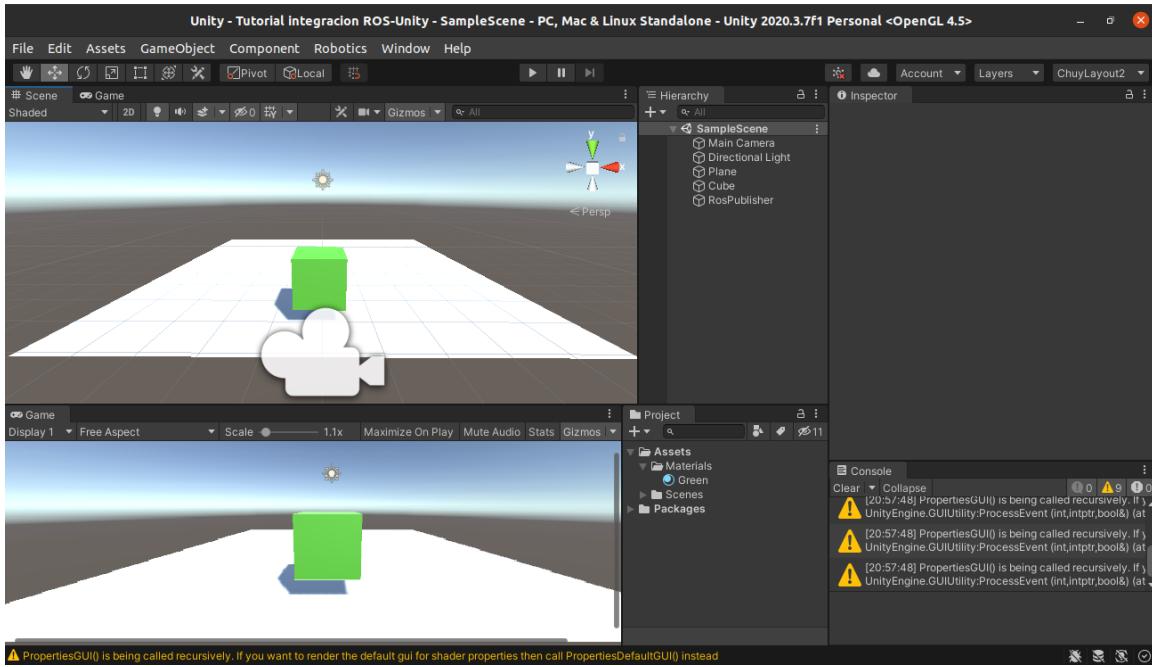
Para ello, damos clic derecho en la ventana *Jerarquía* y seleccionamos los objetos.

- Plane
- Cube
- Empty GameObject (lo llamaremos “RosPublisher”)

Los acomodamos en la escena de la siguiente manera:

NOTA: No es nada complicado mover los objetos de una posición a otro y cambiar su rotación. El color que se le da al cubo es simplemente por gusto, ya que no es necesario.





Configuramos la escena de Unity yendo a la barra de menu y seleccionamos “Robotics -> Generate ROS Messages...”.

NOTA: Aqui estaremos haciendo uso de los archivos del paquete “unity_robots_demo_msgs”.

En la ventana que se nos abre establecemos la ruta del paquete “unity_robots_demo_msgs” y hacemos lo siguiente:

- Expandimos la subcarpeta “unity_robots_demo_msgs” y hacemos clic en “Build 2 msgs” para generar nuevos scripts C# a partir de los archivos ROS “.msg”.

Estos archivos se guardaran en el directorio predeterminado:

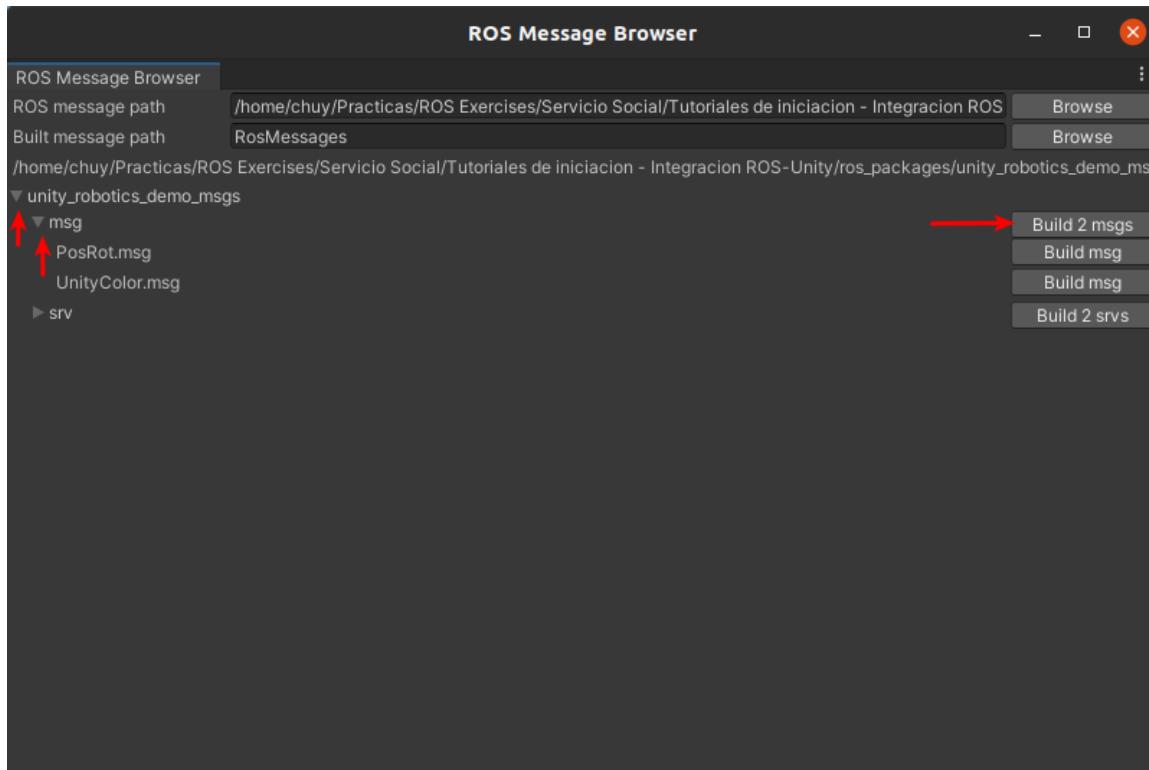
“Assets/RosMessages/UnityRoboticsDemo/msg”

Estos dos mensajes son:

- PosRot.msg (para esta parte lo necesitaremos)
Se declaran campos para especificar la posicion y rotacion en tres dimensiones de un objeto.
- UnityColor.msg (mas adelante lo necesitaremos)
Se declaran campos para especificar el color en el formato “rgba”.

Lo que hace Unity con esos archivos “.msg” es construir nuevos scripts C# a partir de estos archivos, de los que crea una clase con sus respectivos

atributos, constructores, métodos para serializar y deserealizar, y un método “ToString()”.

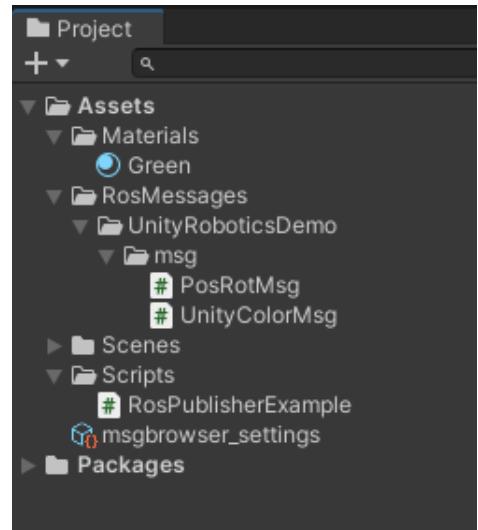


Continuando, ahora creamos un nuevo directorio en “Assets” llamado “Scripts”, en el cual crearemos un nuevo script llamado “RosPublisherExample.cs” (en este directorio estaremos incluyendo todos los scripts C# que creamos). Este script hará lo siguiente:

Se especifican las clases que usaremos (las cuales están ubicadas en otros archivos, como las clases de conexión, etc.).

Se declaran varias variables:

- ROSConnection ros
Variable para la conexión.
- public string topicName="pos_rot"
Se especifica el nombre del tema al que se publicara.
- public GameObject cube
Importante hacer pública esta variable para poder relacionarla con nuestro



cubo en la escena de Unity.

- **public float publishMessageFrequency=2.0f**
Se establece la frecuencia de publicacion (recordar que estaremos publicando la posicion y orientacion del cubo en un tema ROS).
- **private float timeElapsed**
Esta variable nos ayudara para determinar cuánto tiempo ha transcurrido desde el último mensaje (es una variable auxiliar).

Se declaran dos métodos (por defecto el “start()” y el “update()):

- **public void start()**
Aquí se crea una instancia de conexión con ROS y se registra el tema al que se publica.
- **private void update()**
Aqui de publica cada cierto periodo de tiempo la posicion y orientacion del cubo en el tema ROS previamente establecido usando la instancia de conexion ROS (la cual es de tipo “ROSConnection”).

IMPORTANTE: Es importante recordar que en nuestro espacio de trabajo ROS **SIEMPRE** se tiene que estar ejecutando un nodo que ejecute el script de Python “default_server_endpoint.py” que se encuentra en el paquete “ros_tcp_endpoint” cuando realizamos nuestras prácticas. Así que es importante comprender a fondo este script:

(Mostraremos el script completo para explicar a detalle cada parte del código)

```
#!/usr/bin/env python

# Todo esto es como una ayuda para Unity, para que tenga todas las
# funcionalidades como las tenemos en nuestro espacio de trabajo ROS
# (poder publicar en un tema, etc.).

# El nodo "server_endpoint" que se ejecuta en ROS (en el contenedor Docker)
# es el que escucha a traves de los puertos a Unity y ejecuta lo que se
# ordena desde Unity.

# Biblioteca cliente ROS (Python)
import rospy
# Ubicacion de las clases:
$ROS_WORKSPACE/src/ros_tcp_endpoint/src/ros_tcp_endpoint
```

```

# En el script "__init__.py" se especifican los modulos y lo que se importa
from ros_tcp_endpoint import TcpServer

def main():
    # Se obtienen parametros del servidor de parametros
    # Este nombre puede ser reemplazado por el que se declara en un archivo
    ".launch"
    ros_node_name = rospy.get_param("/TCP_NODE_NAME", "TCP_Server")
    # Instanciamos una clase del servidor TCP con sus respectivos
    # parametros
    tcp_server = TcpServer(ros_node_name, anonymous=True)
    # Inicializamos un nodo ROS
    rospy.init_node(ros_node_name, anonymous=True)
    # Iniciamos el servidor TCP
    tcp_server.start()

    # Con esto le indicamos al nodo que quede en espera hasta que sea apagado
    rospy.spin()

if __name__ == "__main__":
    main()

```

Con respecto a las clases que se utilizan, es importante conocer mas a donde sobre estas:

- La conexión TCP (*TcpServer*)

Constructor: (*node_name, buffer_size=1024, connections=10, tcp_ip=""*, ...)

Aquí se inicializa un nodo ROS y el servidor TCP.

Se hace uso de parámetros como: "/ROS_IP", "/ROS_TCP_PORT", ...

Aquí sucede la magia entre la conexión ROS y Unity.

- Clase para publicar (*RosPublisher*)

Constructor: (*topic, message_class, queue_size=10*)

En ROS: *rospy.Publisher(topic, message_class, queue_size=10)*

Devuelve un objeto.

- Clase para suscribirse (*RosSubscriber*)

Constructor: (*topic, message_class, tcp_server, queue_size=10*)

En ROS: *rospy.Subscriber(topic, message_class, callback)*

Devuelve un objeto.

function callback(data) (data es de tipo "message_class")

Aquí se reciben los datos del tema.

- Clase para llamar a un servicio en ROS (**RosService**)

Constructor: *(name_service, service_class)*

En ROS: *rospy.ServiceProxy(name_service, service_class)*

Devuelve un objeto.

Uso:

```
srv = rospy.ServiceProxy(name_service, service_class)
```

rospy.ServiceProxy se usa para llamar a un servicio

```
response_message = srv(request_message)
```

(los parametros dependeran de como se implemente)

- Clase para registrar un servicio implementado en Unity (**UnityService**)

Constructor: *(name_service, service_class, tcp_server, queue_size=10)*

En ROS: *rospy.Service(name_service, service_class, callback)*

Devuelve un objeto.

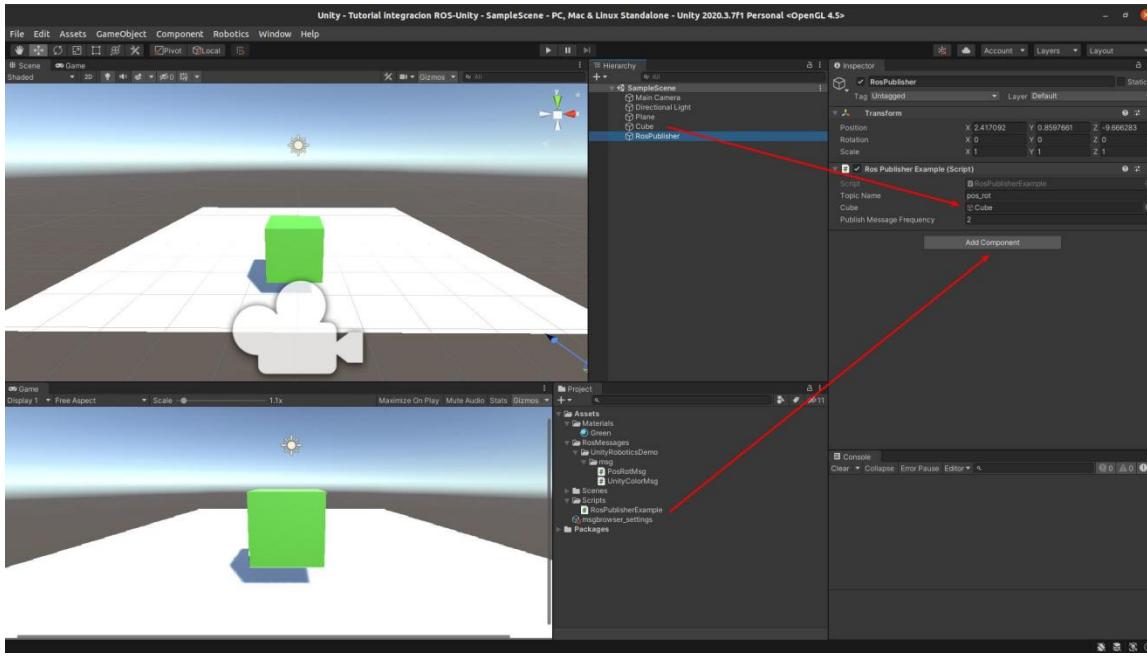
function callback(data) (data es de tipo “<msg_solicitud>”)

Aquí se reciben las solicitudes y se devuelve una respuesta.

A grandes rasgos a si se comportan estas clases, que realmente no estaremos tocando mucho este código porque ya se nos proporciona y solo hay que utilizar los métodos más importantes que se nos presentan.

NOTA: Es importante conocer el código y como funciona para que cuando se nos presente un problema saber como depurarlo y solucionarlo.

Ahora, en Unity toca asociar el script que acabamos de crear con el objeto llamado “RosPublisher” que se encuentra en la ventana *Jerarquía* (le agregamos el script como componente). Por último, le asociamos el cubo de nuestra escena al atributo “cube” del script (esto es lo divertido, ya que podemos controlar cualquier objeto en escena desde el código).



Ahora nos dirigimos a nuestro espacio de trabajo ROS, y para esto ejecutamos los siguientes comandos:

■ *docker-compose run --rm --service-ports ros /bin/bash*

Es importante estar ubicados donde se encuentra el archivo "docker-compose.yaml".

■ *roscd unity_robots_demo*

Nos dirigimos al directorio de este paquete.

■ *chmod +x -R scripts/*

Agregamos permisos de ejecucion a todos los archivos dentro de la carpeta "scripts".

■ *cd -*

Volvemos a nuestro espacio de trabajo ROS.

Antes de ejecutar el siguiente comando haremos algunos cambios:

Modificamos el archivo "params.yaml" ubicado en:

ros_tcp_endpoint/config/params.yaml

Agregamos la siguiente linea:

ROS_IP: 127.0.0.1

Comentamos esta linea

ROS_IP: 0.0.0.0

Modificamos el archivo "robo_demo.launch" ubicado en:

unity_robots_demo/launch/robo_demo.launch

Agregamos la siguiente linea:

```

<!--<param name="ROS_IP" value="127.0.0.1" />-->
Comentamos esta linea
<rosparam command="load" file="$(find
ros_tcp_endpoint)/config/params.yaml" />

```

Ahora si ejecutamos el siguiente comando:

- `roslaunch unity_robotics_demo robo_demo.launch &`
`roslaunch <package> <launch-file>`

El archivo de lanzamiento “robo_demo.launch” hace lo siguiente:

Carga parametros especificados en el archivo “params.yaml” al servidor de parametros.

Al terminar el comando con “&” indicamos que el comando se ejecute en segundo plano, de esta manera no muere al presionar *Ctrl+c*.

Inicia dos nodos:

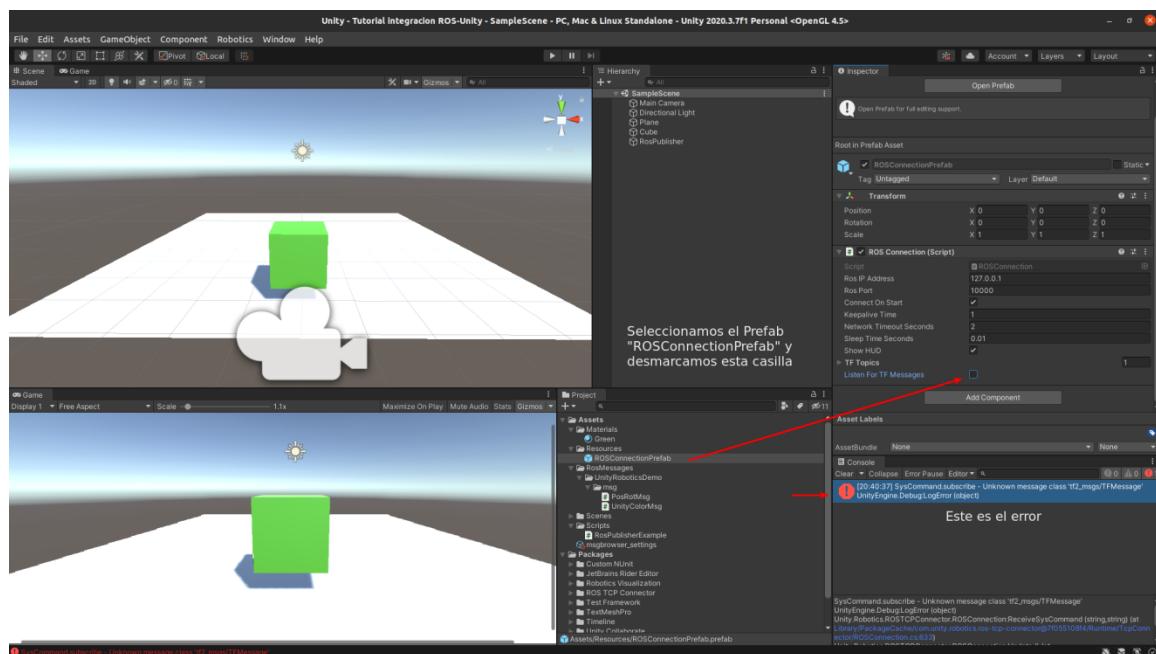
- El nodo “position_service” ejecuta el script “position_service.py” del paquete “unity_robotics_demo”.
Este nodo hay que quitarlo de este archivo de lanzamiento, ya que por el momento no lo utilizaremos y para que este archivo de lanzamiento solo ejecute el nodo “server_endpoint” (del cual se hablará a continuación).
- El nodo “server_endpoint” ejecuta el script “default_server_endpoint.py” del paquete “ros_tcp_endpoint”.
NOTA: Este nodo es el que siempre se tiene que estar ejecutando cuando deseamos que se comuniquen Unity y ROS.

Algunos parámetros de la etiqueta “node” en un archivo de lanzamiento (Es importante conocerlos para cuando se desee cargar nodos desde un archivo de lanzamiento):

- `pkg="my_package"`
Paquete del nodo.
- `type="node_type"`
Tipo de nodo (debe haber un ejecutable con el mismo nombre en el paquete).
- `name="node_name"`
Nombre del nodo (el nombre se puede contener en un espacio de nombres si se desea).
- `(opcional) args="arg1 arg2 arg3"`
Pasar argumentos al nodo.

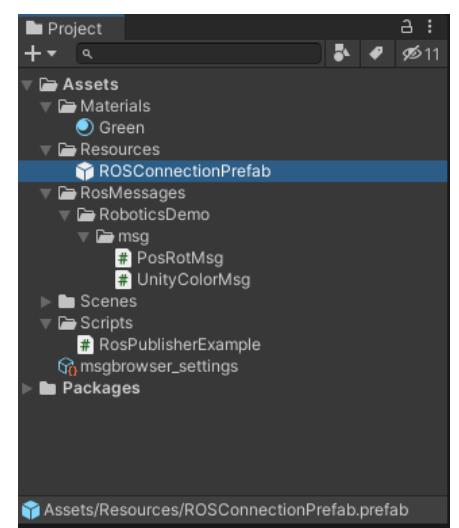
- (opcional) `respawn="true"` (por defecto es “`false`”)
Reinicia el nodo automaticamente si se cierra.
- (opcional) `ns="node_namespace"`
Inicia el nodo en un espacio de nombres.
- (opcional) `output="log|screen"`
Para dirigir la salida “`stdout/stderr`” del nodo a la pantalla o a un archivo de registro (log).
- (opcional) `if="true|false"`
Si es “`true`”, el nodo se iniciara como de costumbre. Si es “`false`”, el nodo no se iniciara (condicional).

Antes de continuar con el siguiente paso, debemos desmarcar la siguiente casilla en Unity, ya que de lo contrario tendremos un error (es relacionado a un tipo de mensaje que no tenemos):

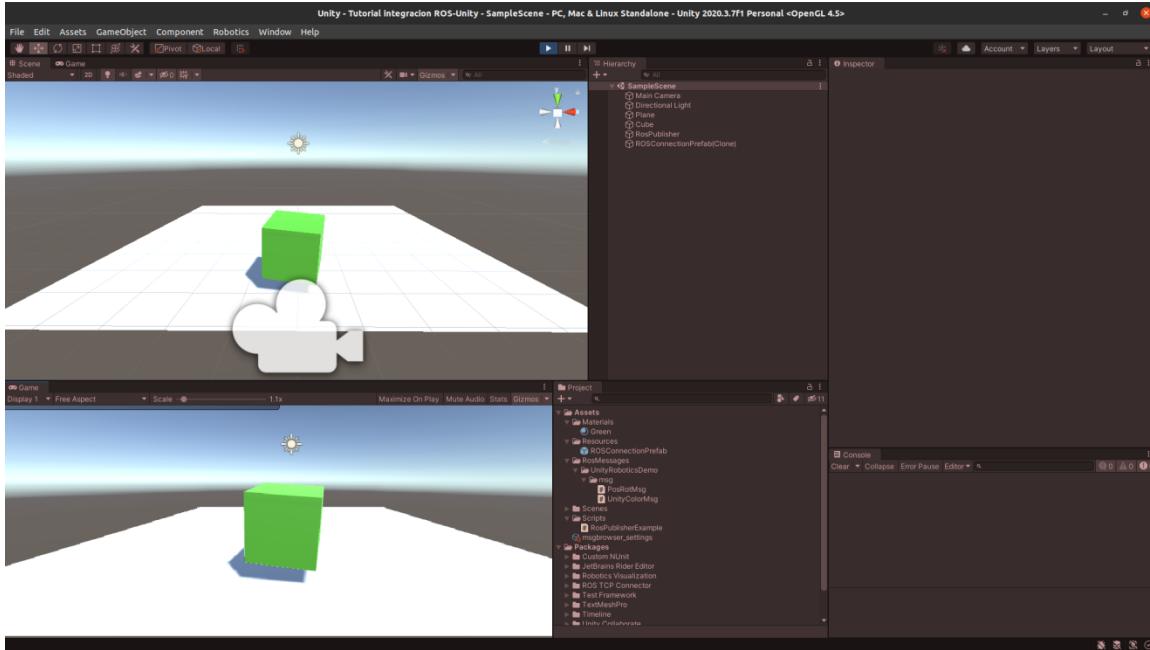


Por último, nos dirigimos a Unity y le damos play a la escena.

NOTA: Es importante mencionar que la primera vez que le demos play a la escena se nos creara una carpeta en “Assets” llamada “Resources” y dentro un prefab (GameObject) llamado ROSConnectionPrefab, el cual se estara instanciando a la escena en Unity cada vez que se llame al



metodo “GetOrGetInstance()” de la clase “ROSConnection”.



Como podemos observar no tenemos errores.

Una cosa importante a destacar es que en nuestro espacio de trabajo ROS tenemos muchos comandos para diferentes tareas, como por ejemplo:

- Listar nodos
- Listar temas
- Listar servicios
- etc.

Algo que podemos hacer es consultar la información del tema “pos_rot” (tema donde se publica la posición y rotación del cubo desde Unity):

```
root@795115cbcd0:/catkin_ws# rostopic info /pos_rot
Type: unity_robotsim_demo_msgs/PosRot

Publishers:
* /server_endpoint (http://795115cbcd0:43439/)

Subscribers: None

root@795115cbcd0:/catkin_ws#
```

E

El nodo “server_endpoint” es el que publica en dicho tema.

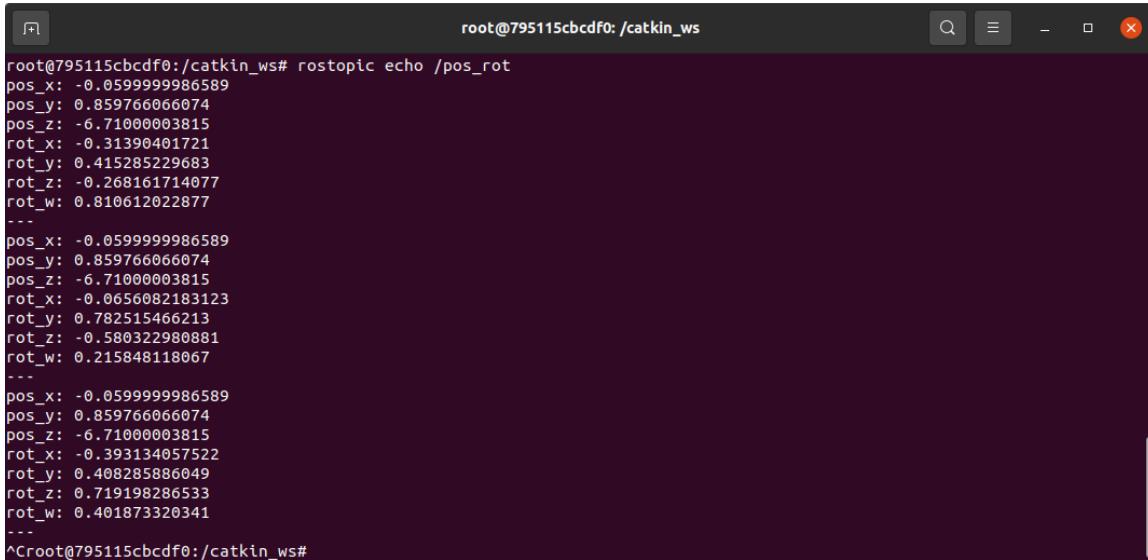
Recordar que este nodo llamado “server_endpoint” ejecuta el script “default_server_endpoint.py” del cual ya hemos hablado y sabemos que es muy

importante que se esté ejecutando.

Podemos ver lo que se publica en dicho tema desde ROS ejecutando el siguiente comando:

NOTA: Esto mientras este en play la escena en Unity.

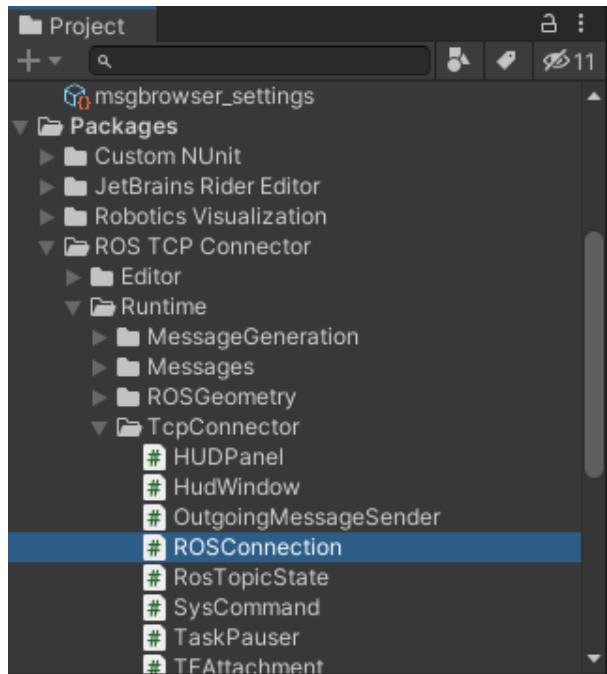
`rostopic echo /pos_rot`



A terminal window titled "root@795115cbcdf0:/catkin_ws" showing the output of the command "rostopic echo /pos_rot". The output displays a series of messages with position and rotation values being published at regular intervals. The window has a dark theme with light-colored text and standard Linux-style window controls.

```
root@795115cbcdf0:/catkin_ws# rostopic echo /pos_rot
pos_x: -0.0599999986589
pos_y: 0.859766066074
pos_z: -6.71000003815
rot_x: -0.31390401721
rot_y: 0.415285229683
rot_z: -0.268161714077
rot_w: 0.810612022877
...
pos_x: -0.0599999986589
pos_y: 0.859766066074
pos_z: -6.71000003815
rot_x: -0.0656082183123
rot_y: 0.782515466213
rot_z: -0.580322980881
rot_w: 0.215848118067
...
pos_x: -0.0599999986589
pos_y: 0.859766066074
pos_z: -6.71000003815
rot_x: -0.393134057522
rot_y: 0.408285886049
rot_z: 0.719198286533
rot_w: 0.401873320341
...
^Croot@795115cbcdf0:/catkin_ws#
```

Cada 2 segundos se publica desde Unity y en consola se ve reflejado.



Este script llamado “ROSConnection” es el script mas importante para la comunicacion entre ROS y Unity, y es el que mas se estara utilizando en los scripts C# que vayamos creando.

Esta clase cuenta con bastantes atributos de configuracion y metodos, como por ejemplo:

- Publish
- RegisterPublisher
- Subscribe
- Unsubscribe
- SubscribeByMessageName
- AddSubscriberInternal
- ImplementService
- SendServiceMessage
- RegisterRosService
- GetOrGetInstance

Devuelve una instancia de tipo “ROSConnection”. A esta instancia se le asigna el componente de tipo “ROSConnection” de un nuevo GameObject que se crea o de un Prefab (el Prefab es como un GameObject personalizado y listo para instanciar las veces que sea necesario).

Este metodo se usara mucho en nuestros scripts C# que creemos.

- instance (es un getter)
Llama al metodo “GetOrGetInstance”.
- Connect
- Disconnect
- ConnectionThread
- Entre muchos mas...

NOTA: Algunos metodos se volvieron obsoletos a comparacion de la primera vez que los utilice (al parecer han estado mejorando la implementacion del codigo, por lo que hay que tener cuidado con las versiones de codigo con las que trabajemos).

Antes de continuar quisiera dejar claros algunos conceptos:

- Cuando usamos el comando “roslaunch” automaticamente iniciara “roscore” si detecta que aun no es está ejecutando.
“roscore” es una colección de nodos y programas que son requisitos previos de un sistema basado en ROS.

Para iniciar “roscore” basta con ejecutar el siguiente comando:

roscore &

“roscore” se iniciara:

- Un ROS Master: realiza un seguimiento de los editores y suscriptores de temas y servicios. Su papel principal es permitir que los nodos ROS se puedan ubicar entre si para que puedan comunicarse.
- Un servidor de parámetros: los nodos utilizan este servidor para almacenar y recuperar parámetros en tiempo de ejecución.
- Un nodo de registro llamado “rosout”.

- Instrucciones ROS para todo tipo de tareas:

- roscore
- roslaunch
- rosmsg
- rossrv
- rosnode
- rosrun
- rosparam
- rosservice
- rostopic
- ...

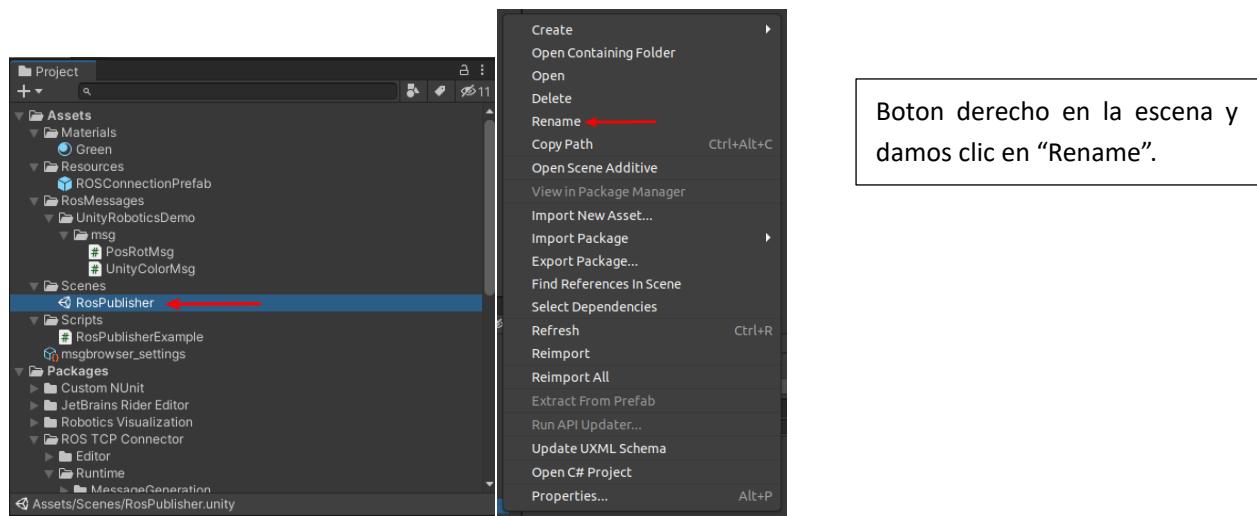
- Los comandos que se terminan con “&”, es decir que se mandan a ejecutar a segundo plano, se pueden terminar de la siguiente manera:

En la terminal tenemos estos comandos para ejecutar:

- jobs: enumera los procesos que se están ejecutando en segundo plano y en primer plano.
- fg %<numero_proceso>: nos trae a la terminal el proceso que indiquemos (con *Ctrl+c* los terminamos).

Por último, para tener un orden de nuestras escenas en Unity renombraremos la escena

actual y le pondremos por nombre “RosPublisher” (ya que mas adelante estaremos creando nuevas escenas dentro del mismo proyecto).



Suscripción a un tema desde una escena de Unity

Crearemos una escena de Unity que se suscriba a un tema ROS para cambiar el color de un “GameObject” (en particular un cubo).

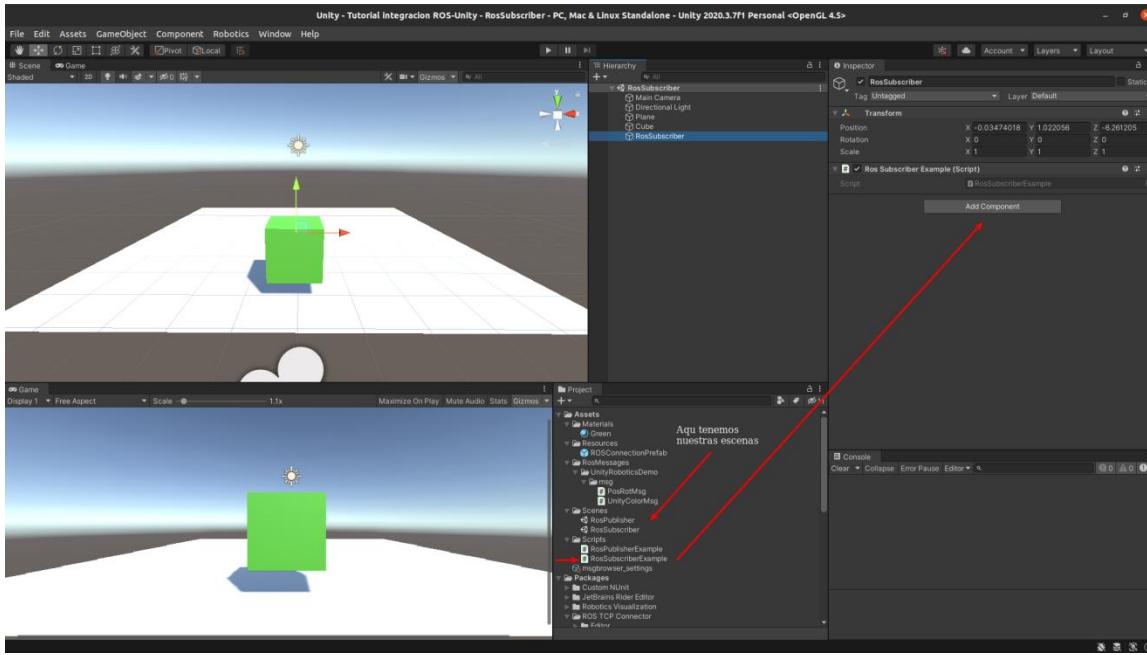
La escena nueva de Unity tendrá por nombre “RosSubscriber” y tendrá en escena los siguientes objetos:

- Un cubo (tendrá por nombre “Cube”)
- Un plano (tendrá por nombre “Plane”)
- Un objeto vacío (tendrá por nombre “RosSubscriber”)

Este objeto tendrá por componente un script llamado “RosSubscriberExample.cs” el cual tendremos que crear primero.

Tanto el cubo como el plano se pueden copiar de la escena anterior y pegar en esta escena nueva (para ahorrar trabajo).

NOTA: Recordar que además tenemos un Prefab llamado “ROSConnectionPrefab” (este, recordando, se creó automáticamente dentro de una carpeta llamada “Resources” en el tutorial pasado) el cual tiene asociado como componente el script “ROSConnection.cs” el cual permite establecer la conexión entre Unity y ROS. Este Prefab se instancia a la escena cada vez que damos play a la escena.



El script llamado “RosSubscriberExample.cs” hará lo siguiente:

Se especifican las clases que usaremos (las cuales están ubicadas en otros archivos, como las clases de conexión, etc.).

Se declaran varias variables:

- ROSConnection ros
- public string topicName="color"
Se especifica el nombre del tema al que se suscribirá.
- public GameObject cube
Importante hacer pública esta variable para poder relacionarla con nuestro cubo en la escena de Unity.

Se declaran dos métodos (por defecto el “start()” y el “update()”):

- public void start()
Aquí se crea una instancia de conexión con ROS y se suscribe al tema especificando el nombre del tema y una función de devolución de llamada para capturar los datos provenientes de ese tema.
- private void update()
Aquí no se hace nada

Se declara la función de devolución de llamada:

- public ColorChange(<tipo_mensaje> <objeto>)
Esta es la función de devolución de llamada donde se reciben los datos provenientes del tema al que se suscribió. Una vez tengamos los datos se

procede a cambiar el color del cubo que se encuentra en nuestra escena (recibimos un color aleatorio).

IMPORTANTE: Siempre hay que recordar que si en los scripts C# declaramos variables de tipo GameObject para asociar objetos de la escena de Unity, entonces estamos obligados a asociar esos objetos desde la escena a los campos correspondientes de dichas variables.

Una vez configurado todo en Unity, proseguimos a hacerlo ahora con ROS. Para esto crearemos un script de Python en la carpeta “scripts” del paquete “unity_robots_demo”, el cual tendra por nombre “color_publisher.py” y hara lo siguiente:

Se define una clase, la cual inicializara un nodo y se especificara el nombre del tema al cual se publicara a la hora de su construcción (en su constructor), además tendrá un método para publicar (ya sea una vez o a una cierta tasa).

Se define un método que creara una instancia de la clase antes mencionada y llamara al método para publicar.

Para mas detalles sobre la libreria cliente ROS para Python consultar el siguiente enlace:

<http://wiki.ros.org/rospy/Overview>

Es muy importante comprender esta libreria porque es con ella con la que trabajaremos del lado de ROS.

Ahora nos dirigimos a nuestro espacio de trabajo ROS, y para esto ejecutamos los siguientes comandos:

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

Iniciamos nuestro contenedor Docker, lo cual es importante estar ubicados donde se encuentra el archivo “docker-compose.yaml”.

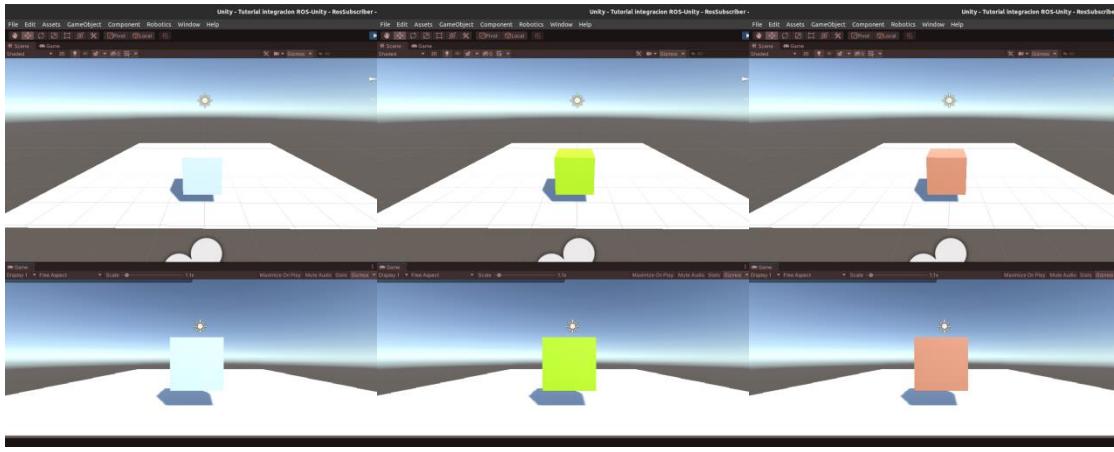
- `roslaunch unity_robots_demo robo_demo.launch &`

Iniciamos nuestro servidor TCP (recordar que en este archivo de lanzamiento solo se ejecuta un nodo).

- `rosrun unity_robots_demo color_publisher.py 1 &`

La escena de Unity debe estar en play para que el cubo pueda estar cambiando de color.

El parámetro “1” que se le agrega al final indica que se estará publicando en el tema cada segundo (se codifico de esta manera).



```

root@4874a5acc05e:/catkin_ws# rostopic echo /color
r: 135
g: 226
b: 81
a: 219
...
r: 175
g: 48
b: 198
a: 242
...
r: 104
g: 168
b: 199
a: 70
...
^Croot@4874a5acc05e:/catkin_ws# 

root@4874a5acc05e:/catkin_ws# rostopic info /color
Type: unity_robots_demo_msgs/UnityColor
Publishers:
 * /color_publisher (http://4874a5acc05e:42655/)
Subscribers:
 * /server_endpoint (http://4874a5acc05e:37257/)

root@4874a5acc05e:/catkin_ws#

```

Como podemos observar, efectivamente un nodo está publicando en el tema “/color” y otro esta suscrito a dicho tema.

Ya que estamos un poco más familiarizados con el uso de la librería cliente ROS para Python, es bueno conocer un poco más sobre las funcionalidades que presenta con el motivo de familiarizarnos aun mas y que no nos sea tan complicado seguir avanzando con los tutoriales.

NOTA: Cabe mencionar que también existe otra librería cliente ROS para Python que anteriormente no había escuchado hablar de ella y se llama “rclpy”. Por el momento seguiremos utilizando “rospy”, pero si por alguna razon “rclpy” ofrece alguna otra caracteristica nueva que podamos necesitar, entonces podemos usar “rclpy” sin problema (no creo que haya problema si las usamos las dos).

Así que echemos un vistazo a la librería “**rospy**”:

NOTA: Los parámetros que estén marcados con asterisco “*” significa que son opcionales.

- **Inicialización y apagado de nodos**
 - Inicializar nodo

```
rospy.init_node(node_name, anonymous*=True)
```

NOTA: rospy registra los controladores de señales para que se pueda salir (apagar la ejecucion de un nodo) con “Ctrl+C”.

- Patrones mas comunes para probar el apagado de un nodo

```
while not rospy.is_shutdown():
```

```
# Hacer algo
```

```
# ... Configurar devoluciones de llamada
```

```
rospy.spin()
```

Con “spin()” hacemos que se mantenga activo el subproceso principal de Python hasta que el indicador “is_shutdown()” sea verdadero.

Hay varias formas en que un nodo puede recibir una solicitud de apagado, por lo que es importante utilizar uno de estos metodos anteriores para que nuestro programa finalice correctamente.

- Controlador de cierre de nodo

```
def controller():
```

```
    print "shutdown time"
```

```
rospy.on_shutdown(controller)
```

Se especifica el controlador que se llamara cuando el proceso comience a cerrarse (la funcion no debe contener parametros).

● **Mensajes**

- Generacion de mensajes

Como todas las bibliotecas cliente de ROS, “rospy” toma archivos “msg” y genera el codigo fuente de Python para ellos. Veamos el patron de estos:

- **msg**

```
<nombre_paquete>/msg/<nombre_mensaje>.msg ->
<nombre_paquete>.msg.<nombre_mensaje>
```

- **srv**

```
<nombre_paquete>/srv/<nombre_mensaje>.srv ->
<nombre_paquete>.srv.<nombre_mensaje>
```

Ejemplo:

```
import std_msgs.msg
msg=std_msgs.msg.String()
```

Se utiliza el mensaje “std_msgs/String” del paquete “std_msgs”.

```
root@d124978b3cc6:/catkin_ws# rosmsg show std_msgs/String  
string data
```

```
root@d124978b3cc6:/catkin_ws# █
```

- Inicialización de mensajes

Sin argumentos

```
msg=std_msgs.msg.String()  
msg.data="hello world"
```

*Argumentos en orden (*args)*

```
msg=std_msgs.msg.String("hello world")
```

*Argumentos de palabras clave (**args)*

```
msg=std_msgs.msg.String(data="hello world")
```

- **Publicadores y suscriptores**

- Publicar en un tema

```
rospy.Publisher(topic_name, msg_class, queue_size)
```

Publisher.publish(msg_class)

Se tienen dos estilos:

- *Estilo explicito (como el ejemplo anterior)*
- *Estilo implicito con argumentos en orden*

Este metodo es sincrono de forma predeterminada, lo que significa que la invocacion se bloquea hasta que haya terminado la ejecucion de la funcion.

Es importante conocer ademas los parametros opcionales de la clase "Publisher" (son un tema mas avanzado).

Ejemplo:

```
import rospy  
from std_msgs.msg import String  
  
rospy.init_node('node_name')  
pub=rospy.Publisher('topic_name', String, queue_size=10)  
rate=rospy.Rate(10) # 10hz  
while not rospy.is_shutdown():  
    pub.publish(String("hello world"))  
    rate.sleep()
```

- Suscribirse a un tema

```
rospy.Subscriber(topic_name, msg_class, callback)
```

Ejemplo:

```
import rospy
from std_msgs.msg import String

# data es de tipo std_msgs/String
def callback(data):
    rospy.loginfo("data: %s", (data.data))

def listener():
    rospy.init_node('node_name')
    rospy.Subscriber('topic_name', String, callback)
    # Mantenemos ejecutando el nodo hasta que se le de una
    # señal de apagado
    rospy.spin()
```

- **Servicios**

- Definiciones de servicio, mensajes de solicitud y mensajes de respuesta

Los archivos “srv” contienen un mensaje de solicitud y un mensaje de respuesta.

“rospy” convierte estos archivos “srv” en código fuente de Python y crea tres clases con las que se debe familiarizar:

- *Definiciones de servicio*

*<nombre_paquete>/srv/<nombre_mensaje>.srv ->
<nombre_paquete>.srv.<nombre_mensaje>*

- *Mensajes de solicitud*

*<nombre_paquete>/srv/<nombre_mensaje>.srv ->
<nombre_paquete>.srv.<nombre_mensaje>Request*

- *Mensajes de respuesta*

*<nombre_paquete>/srv/<nombre_mensaje>.srv ->
<nombre_paquete>.srv.<nombre_mensaje>Response*

Definición de servicio:

```
srv=rospy.ServiceProxy(srv_name, srv_class)
```

- Service proxies

Se llama a un servicio creando una instancia de “rospy.ServiceProxy” con el nombre del servicio al que se desea llamar.

Ejemplo:

```

# Esperamos hasta que el servicio este disponible
rospy.wait_for_service('add_two_ints')
#Creamos una instancia del servicio (estas instancias #son invocables,
lo que significa que podemos #invocarlas como si fueran metodos)
add_two_ints=rospy.ServiceProxy('add_two_ints', AddTwoInts)
try:
    #Realizamos la solicitud al servicio pasando como #parametros
    los datos de solicitud y obtenemos #una respuesta
    response=add_two_ints(x, y)
#Se genera esta excepcion si un servicio devuelve un #error
except rospy.ServiceException as exc:
    pass

```

Hay tres formas diferentes de pasar argumentos a una instancia de servicio:

- *Estilo explicito (proporcionar una instancia de mensaje).*
- *Dos estilos implicitos (con argumentos en orden y con argumentos de palabras clave) que crean el mensaje de solicitud por nosotros (como en el ejemplo anterior).*

Hay tres tipos de excepciones que pueden ocurrir:

- *TypeError*
La solicitud no es del tipo valida.
- *ServiceException*
La comunicacion con el servicio remoto fallo.
- *ROSSerializationException*
Esto generalmente indica un error de tipo con uno de los campos

También es posible tener conexiones persistentes a servicios (tema más avanzado).

■ Prestación de servicios

En “rospy” se puede proporcionar un servicio creando una instancia de “rospy.Service” con una devolución de llamada para invocar cuando se reciben nuevas solicitudes.

NOTA: Cada solicitud entrante se maneja en su propio subproceso (los servicios deben ser seguros para subprocesos).

```
rospy.Service(name, srv_class, handler, buff_size=65536)
```

Ejemplo:

```

# "req" es de tipo de mensaje de solicitud
def add_two_ints(req):
    # Devolvemos una respuesta
    return pkg_name.srv.AddTwoIntsResponse(req.a+req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    # Creamos un servicio
    srv=rospy.Service('add_two_ints', pkg_name.srv.AddTwoInts,
                      add_two_ints)
    # Mantenemos ejecutando el nodo hasta que se le de una
    señal de apagado
    rospy.spin()

```

También podemos responder con un objeto de mensaje de respuesta (en el ejemplo se crea el mensaje de respuesta por nosotros).

Tenemos dos patrones de uso común para cerrar un servicio:

- *Terminarlo explicitamente*

Ejemplo:

```

srv=rospy.Service(...)
...
srv.shutdown("shutdown reason")

```

- *Usando su metodo propio de servicio "spin()"*

Ejemplo:

```

srv=rospy.Service(...)
srv.spin()

```

NOTA: El metodo "rospy.spin()" es mas general, ya que solo espera hasta que el nodo se apague (pero son similares).

- Encabezados de conexión de servicio

Los encabezados de conexión son una característica de ROS Topics y ROS Services que permiten enviar metadatos adicionales cuando se realiza la conexión inicial entre dos nodos (es un tema más avanzado que por el momento no tocaremos).

- **Servidor de parámetros**

El servidor de parámetros puede almacenar cadenas, enteros, flotantes, booleanos, listas, diccionarios (deben tener claves de cadenas), fechas y codificación base64.

NOTA: Los métodos del servidor de parámetros no son seguros para subprocessos.

- Obtener parámetros

```
rospy.get_param(param_name)  
rospy.get_param(param_name, default_value)
```

Si se usa “get_param()” para obtener un espacio de nombres, se devuelve un diccionario con las claves iguales a los valores de los parámetros en ese espacio de nombres.

Los nombres se resuelven en relación con el espacio de nombres del nodo.

Se genera una excepción “KeyError” si el parámetro no está configurado.

- Configurar parametros

```
rospy.set_param(param_name, value)
```

- Existencia de parametros

```
rospy.has_param(param_name)
```

Es util si no deseamos utilizar sentencias “try/except”.

- Eliminar parametros

```
rospy.delete_param(param_name)
```

Se genera una excepcion “KeyError” si el parametro no esta configurado.

- Lista de nombres de parametros

```
rospy.get_param_names()
```

Para obtener una lista de nombres de parametros existentes como una lista de cadenas.

- Busqueda por clave de parametros

```
rospy.search_param(param_name)
```

Se

Encuentra el nombre del parámetro más cercano, comenzando en el espacio de nombres privado y buscando hacia arriba hasta el espacio de nombres global.

- **Logging**

ROS tiene su propio mecanismo basado en temas llamado “rosout” para registrar mensajes de registro de nodos. Estos mensajes transmiten el estado de un nodo.

“rospy” tiene varios metodos para escribir mensajes de registro, todos comenzando con “log”:

```
PARAMETERS  
* /rosdistro: melodic  
* /rosversion: 1.14.12  
  
NODES  
  
auto-starting new master  
process[master]: started with pid [77]  
ROS_MASTER_URI=http://df9b9a10b3aa:11311/  
  
setting /run_id to a0edcd4c-8566-11ec-9244-0242ac130002  
process[rosout-1]: started with pid [88]  
started core service [/rosout]  
^C  
root@df9b9a10b3aa:/catkin_ws# rosnode list  
/rosout  
root@df9b9a10b3aa:/catkin_ws#
```

- `rospy.logdebug(msg, *args, **kwargs)`
- `rospy.info(msg, *args, **kwargs)`
- `rospy.logwarn(msg, *args, **kwargs)`
- `rospy.logerr(msg, *args, **kwargs)`
- `rospy.logfatal(msg, *args, **kwargs)`

Si “msg” es una cadena de formato, se puede pasar los argumentos de la cadena por separado, por ejemplo:

```
rospy.logerr("Error: %s", some_error)
```

■ Lectura de mensajes de registro

Lugares en los que puede terminar un mensaje de registro según el nivel de detalle:

- `stdout: loginfo`
- `stderr: logerr`

■ Registro periodico

“rospy” admite la escritura de mensajes de registro periódicamente.

Ejemplo:

```
while True:
```

```
    rospy.loginfo_throttle(60, "This message will print every
    60 seconds")
```

■ Registro solo una vez

“rospy” admite la escritura de mensajes de registro solo una vez después de generarse.

Ejemplo:

```
while True:
```

```
    rospy.loginfo_once(60, "This message will print only
    once")
```

● **Nombres e información de nodos**

■ Acceso a la información del nodo

Veamos algunos métodos:

- `rospy.get_name()`

Para obtener el nombre completo del nodo.

- `rospy.get_namespace()`

Para obtener el espacio de nombres del nodo.

- `rospy.get_node_uri()`

Para obtener el URI XMLRPC del nodo.

■ Manipulación de nombres

Es un tema más avanzado.

- **Tiempo**

- Tiempo y duración

"rospy" proporciona dos clases:

- **rospy.Time(arg*)**

Una hora es un momento específico (por ejemplo, "hoy a las 17:00").

- **rospy.Duration(arg*)**

Una duración es un periodo de tiempo (por ejemplo, "2 horas").

ROS tiene la capacidad de configurar un reloj simulado para nodos (nos evitamos utilizar el modulo "time.time" de Python).

Veamos algunos métodos:

- **rospy.Time.now(), rospy.get_rostime()**

Para obtener la hora actual.

- **rospy.get_time()**

Para obtener la hora actual en segundos.

- Dormir y tasas (rates)

Veamos algunos métodos:

- **rospy.sleep(duration)**

ROS dormira durante el periodo especificado (en segundos).

Este metodo generara una excepcion "rospy.ROSInterruptException" si ocurre un cierre de nodo.

- **rospy.Rate(hz)**

Hace un mejor esfuerzo para mantener una tasa particular para un ciclo.

Esta clase tiene un metodo "sleep()".

Ejemplo:

```
rate=rospy.Rate(10) # 10hz
while not rospy.is_shutdown():
    pub.publish("hello")
    rate.sleep()
```

- Timer

- rospy.Timer(period, callback, oneshot=False)**

Llama periodicamente a una devolucion de llamada (callback).

El argumento "oneshot" especifica si el temporizador es o no un temporizador de accion unica, si es asi, solo se disparara un vez.

Ejemplo:

```
def my_callback(event):
    print 'Timer called at ' + str(event.current_real)

rospy.Timer(rospy.Duration(2), my_callback)
```

- **Excepciones**

Excepciones:

- *ROSException*
Clase base de excepcion para clientes ROS.
- *ROSSerializationException*
Excepcion para errores de serializacion de mensajes.
- *ROSInitException*
Excepcion por errores al inicializar el estado de ROS.
- *ROSInterruptException*
Excepcion para operaciones de interrupciones.
- *ROSInternalException*
Excepcion para errores internos de ROS.
- *ServiceException*
Excepcion para errores relacionados con la comunicacion con ROS Services.

Por ultimo, otros conceptos importantes no relacionados a “rospy” que son importantes mencionarlos son los siguientes:

- **Guía de estilo de python** (<http://wiki.ros.org/PyStyleGuide>)
Aqui se define una guia de estilo a seguir a la hora de escribir codigo Python para ROS (tambien hay para C++ y Javascript).
- **TF (transform)** (<http://wiki.ros.org/tf/Overview>)
Realmente este es un tema bastante interesante, extenso y avanzado, pero por el momento no se tocara el tema. Mas adelante, cuando se trabajen con modelos de robot, se requerira conocer mucho este tema.

Implementación de un servicio dentro de una escena de Unity

Crearemos una escena de Unity que cree un servicio en Unity que tome una solicitud con el nombre de un “GameObject” y responda con la pose del “GameObject” (posicion y orientacion) en el sistema de coordenadas ROS.

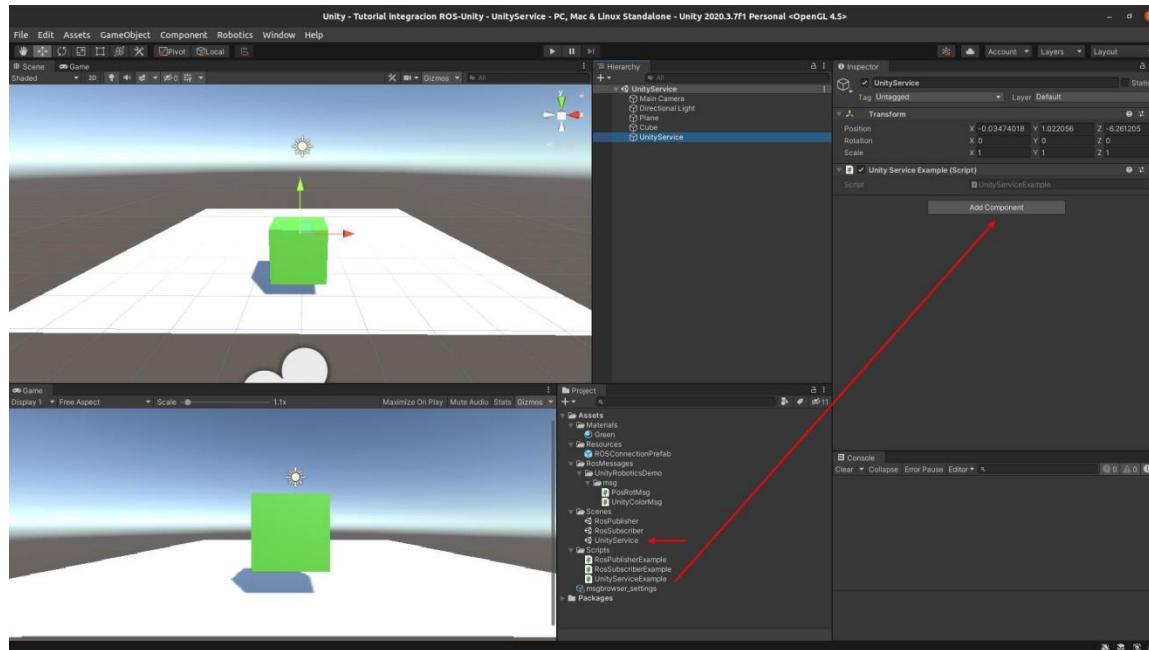
Recordemos que en el tema de servicios tenemos:

- Un servicio (aquí se implementa el servicio, el cual recibe una solicitud y devuelve una respuesta)
- Un cliente (el cual hace una solicitud al servicio y obtiene una respuesta)

La escena nueva de Unity tendrá por nombre “UnityService” y tendrá en escena los siguientes objetos:

- Un cubo (tendrá por nombre “Cube”)
- Un plano (tendrá por nombre “Plane”)
- Un objeto vacío (tendrá por nombre “UnityService”)

Este objeto tendrá por componente un script llamado “UnityServiceExample.cs” el cual tendremos que crear primero.



Antes de continuar necesitamos construir los mensajes de servicios, para esto iremos a la barra de menú y seleccionamos “Robotics -> Generate ROS Messages...”.

NOTA: Aquí estaremos haciendo uso de los archivos del paquete “unity_robots_demo_msgs”.

En la ventana que se nos abre establecemos la ruta del paquete “unity_robots_demo_msgs” y hacemos lo siguiente:

- Expandimos la subcarpeta “unity_robots_demo_msgs” y hacemos clic en

“Build 2 srvs” para generar nuevos scripts C# a partir de los archivos ROS “.srv”.

Estos archivos se guardaran en el directorio predeterminado:

“Assets/RosMessages/UnityRoboticsDemo/srv”

Estos dos mensajes son:

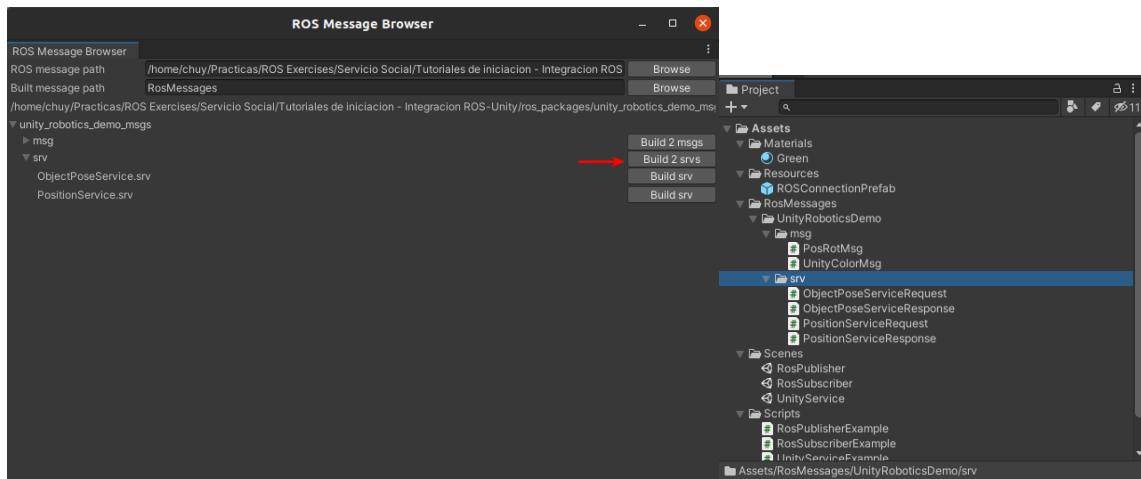
- ObjectPoseService.srv (para esta parte lo necesitaremos)

Se declara el tipo de mensaje de solicitud (“string” para el nombre del “GameObject”) y el tipo de mensaje de respuesta (“geometry_msgs/Pose” para la pose del “GameObject”) separados por “---” .

- PositionService.srv (más adelante lo necesitaremos)

Se declara el tipo de mensaje de solicitud y el tipo de mensaje de respuesta separados por “---”.

Lo que hace Unity con esos archivos “.srv” es construir nuevos scripts C# a partir de estos archivos, de los que crea una clase con sus respectivos atributos, constructores, metodos para serializar y deserealizar, y un metodo “ToString()”.



El script llamado “UnityServiceExample.cs” hara lo siguiente:

Se especifican las clases que usaremos (las cuales están ubicadas en otros archivos, como las clases de conexión, etc.).

Se declaran varias variables:

- ROSConnection ros
- public string serviceName="obj_pose_srv"

Se especifica el nombre del servicio.

Se declaran dos metodos (por defecto el “start()” y el “update()):

- `public void start()`

Aqui se crea una instancia de conexion con ROS y se llama al metodo para implementar un servicio en Unity especificando el nombre del servicio y una funcion de devolucion de llamada que estara capturando las solicitudes y devolviendo una respuesta.

- `private void update()`

Aqui no se hace nada

Se declara la funcion de devolucion de llamada:

- `private <tipo_msg_respuesta> GetObjectPose (<tipo_msg_solicitud> <objeto>)`

Se capturan las solicitudes y se devuelve la pose del “GameObject” que coincide con el nombre del “GameObject” que se indica en el mensaje de solicitud.

Una vez configurado todo en Unity, proseguimos a hacerlo ahora con ROS. Para esto crearemos un script de Python en la carpeta “scripts” del paquete “unity_robots_demo”, el cual tendrá por nombre “object_pose_client.py” y hará lo siguiente:

Se define una clase, la cual tendrá un método en el que se especificara el nombre del servicio al cual se llamará y el nombre del objeto al cual se quiere conocer su posición y orientación en la escena de Unity.

Se define un método que creará una instancia de la clase antes mencionada y llamará al servidor para obtener una respuesta dada una solicitud.

NOTA: No es necesario crear un nodo ROS para realizar llamadas de servicio.

Ahora nos dirigimos a nuestro espacio de trabajo ROS, y para esto ejecutamos los siguientes comandos:

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

Iniciamos nuestro contenedor Docker, lo cual es importante estar ubicados donde se encuentra el archivo “docker-compose.yaml”.

- `roslaunch unity_robots_demo robo_demo.launch &`

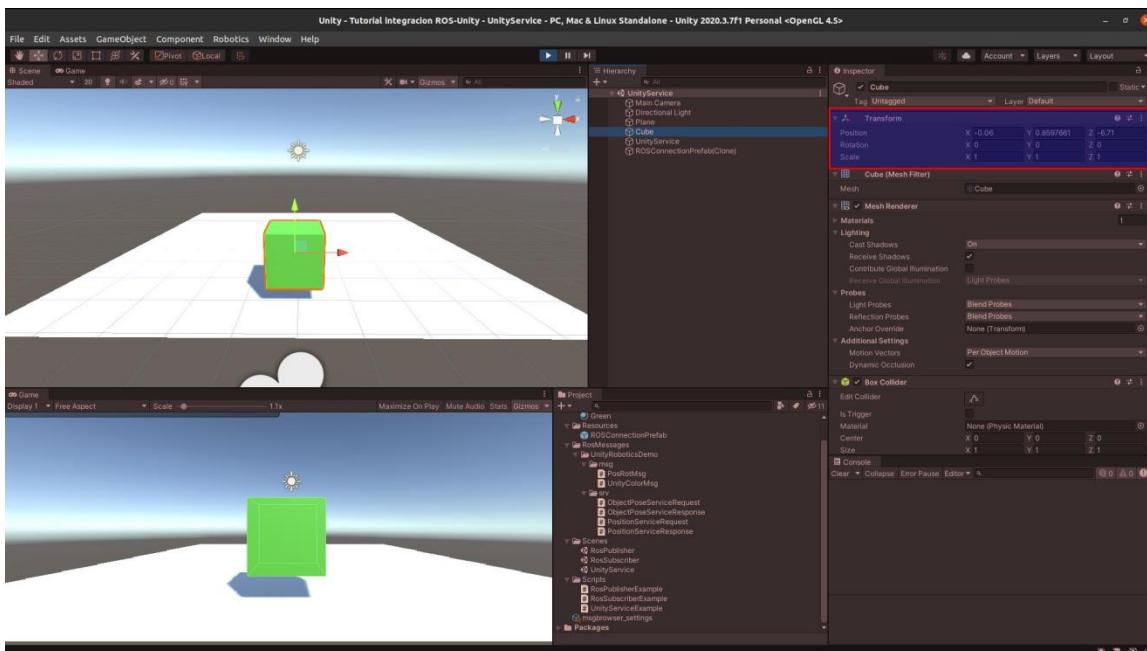
Iniciamos nuestro servidor TCP (recordar que en este archivo de lanzamiento solo se ejecuta un nodo).

- `rosrun unity_robots_demo object_pose_client.py Cube`

La escena de Unity debe estar en play para que el servicio este activo.

```
root@5f6b4eb44d2d:/catkin_ws# rostopic list
/rosvout
/rosvout_agg
root@5f6b4eb44d2d:/catkin_ws# rosnode list
/rosvout
/server_endpoint
root@5f6b4eb44d2d:/catkin_ws# rosservice list
/obj_pose_srv
/rosvout/get_loggers
/rosvout/set_logger_level
/server_endpoint/get_loggers
/server_endpoint/set_logger_level
root@5f6b4eb44d2d:/catkin_ws# rosservice info /obj_pose_srv
Node: /server_endpoint
URI: rosrpc://5f6b4eb44d2d:46205
Type: unity_robots_demo_msgs/ObjectPoseService
Args: object_name
root@5f6b4eb44d2d:/catkin_ws#
```

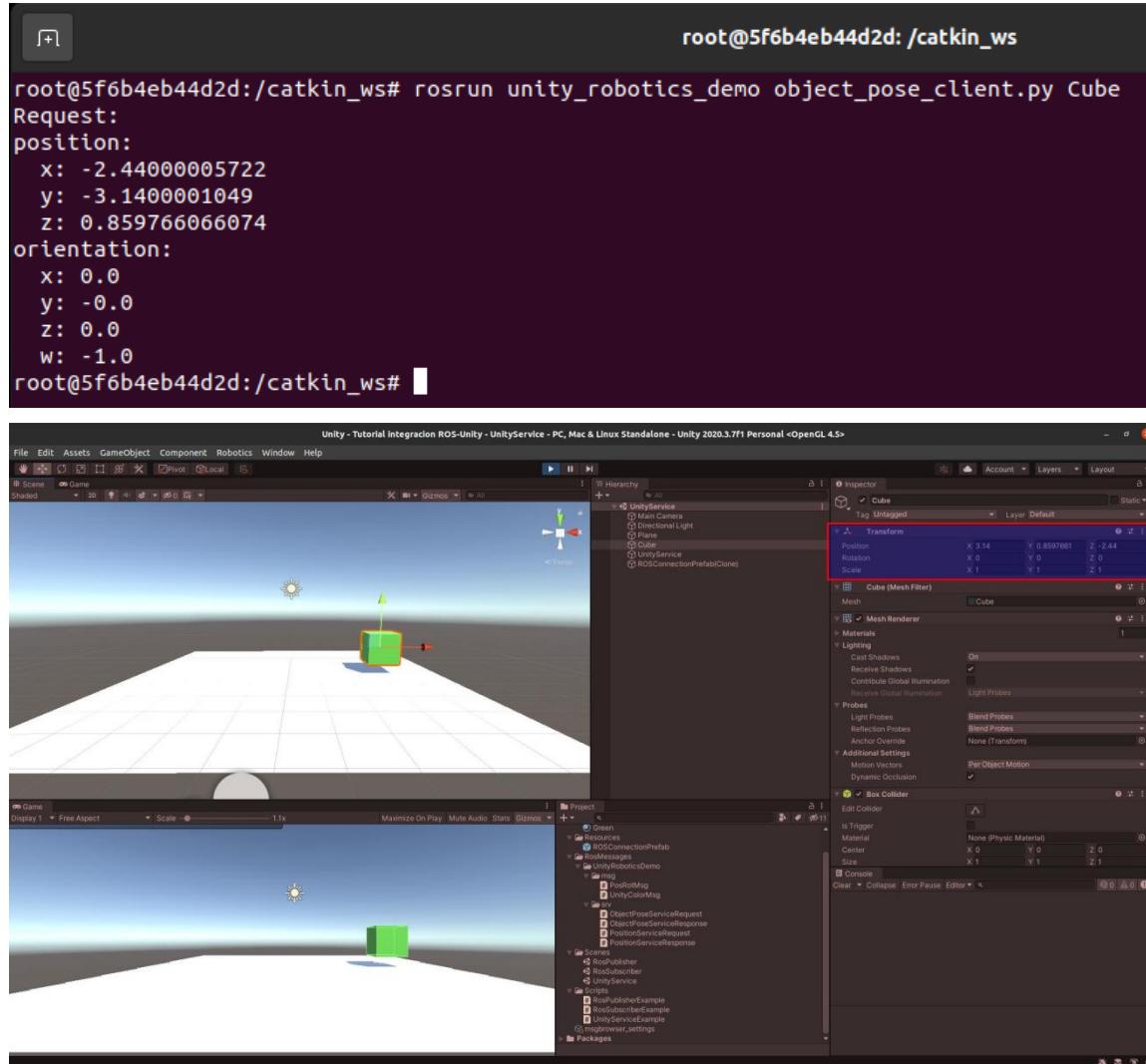
```
root@5f6b4eb44d2d:/catkin_ws# rosrun unity_robots_demo object_pose_client.py
/catkin_ws/src/unity_robots_demo/scripts/object_pose_client.py [object_name]
root@5f6b4eb44d2d:/catkin_ws# rosrun unity_robots_demo object_pose_client.py Cube
Request:
position:
  x: -6.71000003815
  y: 0.0599999986589
  z: 0.859766066074
orientation:
  x: 0.0
  y: -0.0
  z: 0.0
  w: -1.0
root@5f6b4eb44d2d:/catkin_ws#
```



NOTA: Como podemos observar la posición y orientación del cubo no coincide, pero

recordemos que esto es debido a que hemos hecho una conversión del sistema de coordenadas Unity al sistema de coordenadas ROS. Por lo que no hay de qué preocuparnos, los resultados son correctos.

Si movemos al cubo de posición tendremos otros resultados:



Llamada a servicio externo desde una escena de Unity

Crearemos una escena de Unity que llame a un servicio implementado en ROS con la posición y rotación de un GameObject (como solicitud) para recibir una nueva posición y rotación (como respuesta) hacia la que mover el GameObject.

Ahora el servicio estará implementado en ROS y no en Unity como en el tutorial pasado.

Sobre el mensaje de solicitud y respuesta:

El tipo de mensaje de solicitud y respuesta es un tipo de mensaje personalizado llamado “PosRot.msg” el cual tiene campos para especificar la posicion y rotacion (ya se habia mencionado este tipo de mensaje anteriormente pero hasta ahora haremos uso de el).

La escena nueva de Unity tendrá por nombre “RosService” y tendrá en escena los siguientes objetos:

- Un cubo (tendrá por nombre “Cube”)
- Un plano (tendrá por nombre “Plane”)
- Un objeto vacío (tendrá por nombre “RosService”)

Este objeto tendrá por componente un script llamado “RosServiceExample.cs” el cual tendremos que crear primero.

NOTA: Se ha colocado el cubo y el plano en el origen del sistema de coordenadas de Unity.

El script llamado “RosServiceExample.cs” hara lo siguiente:

Se especifican las clases que usaremos (las cuales están ubicadas en otros archivos, como las clases de conexión, etc.).

Se declaran varias variables:

- ROSConnection ros
- public string serviceName="pos_srv"
Se especifica el nombre del servicio.
- public GameObject cube
Variable para asociar el cubo de nuestra escena.
- public Vector3 destination
Variable para guardar la posicion de destino (la posicion que devuelve el servicio hacia donde tiene que ir el cubo).
- public float speed
Variable para especificar la velocidad de traslado del cubo a la posicion de destino.
- public float minDistance
Variable para especificar la distancia minima entre el cubo y la posicion de destino (nos puede servir para evitar que colisionen si es que el destino es un objeto).

- `public bool serviceAnsweredUs, isDone`

Variables auxiliares.

Se declaran dos metodos (por defecto el “start()” y el “update()):

- `public void start()`

Aqui se crea una instancia de conexion con ROS y se llama al metodo para registrar un servicio en ROS especificando el nombre del servicio.

- `private void update()`

Aqui se verifica cuando el cubo ha llegado a su destino, y si es asi, se llama al servicio enviando un mensaje de solicitud (la posicion y rotacion del cubo en nuestra escena) para recibir una respuesta que sera una nueva posicion. El proceso es iterativo, se estaran asignando posiciones aleatorias donde el cubo tendra que ir.

Se declara la funcion de devolucion de llamada:

- `private void CallbackDestination(<tipo_msg_respuesta> <objeto>)`

Se capturan las respuestas por parte del servicio y se asigna a la variable de destino para especificar un nuevo destino al que tiene que dirigirse el cubo.

Una vez configurado todo en Unity, proseguimos a hacerlo ahora con ROS. Para esto crearemos un script de Python en la carpeta “scripts” del paquete “unity_robots_demo”, el cual tendrá por nombre “position_service.py” y hará lo siguiente:

Se define una clase, la cual tendrá un método en el que se define el nombre que tendrá el servicio y a la vez creara el servicio, y además, que no podría faltar, un método para controlar los mensajes de solicitud y respuesta.

Ahora nos dirigimos a nuestro espacio de trabajo ROS, y para esto ejecutamos los siguientes comandos:

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

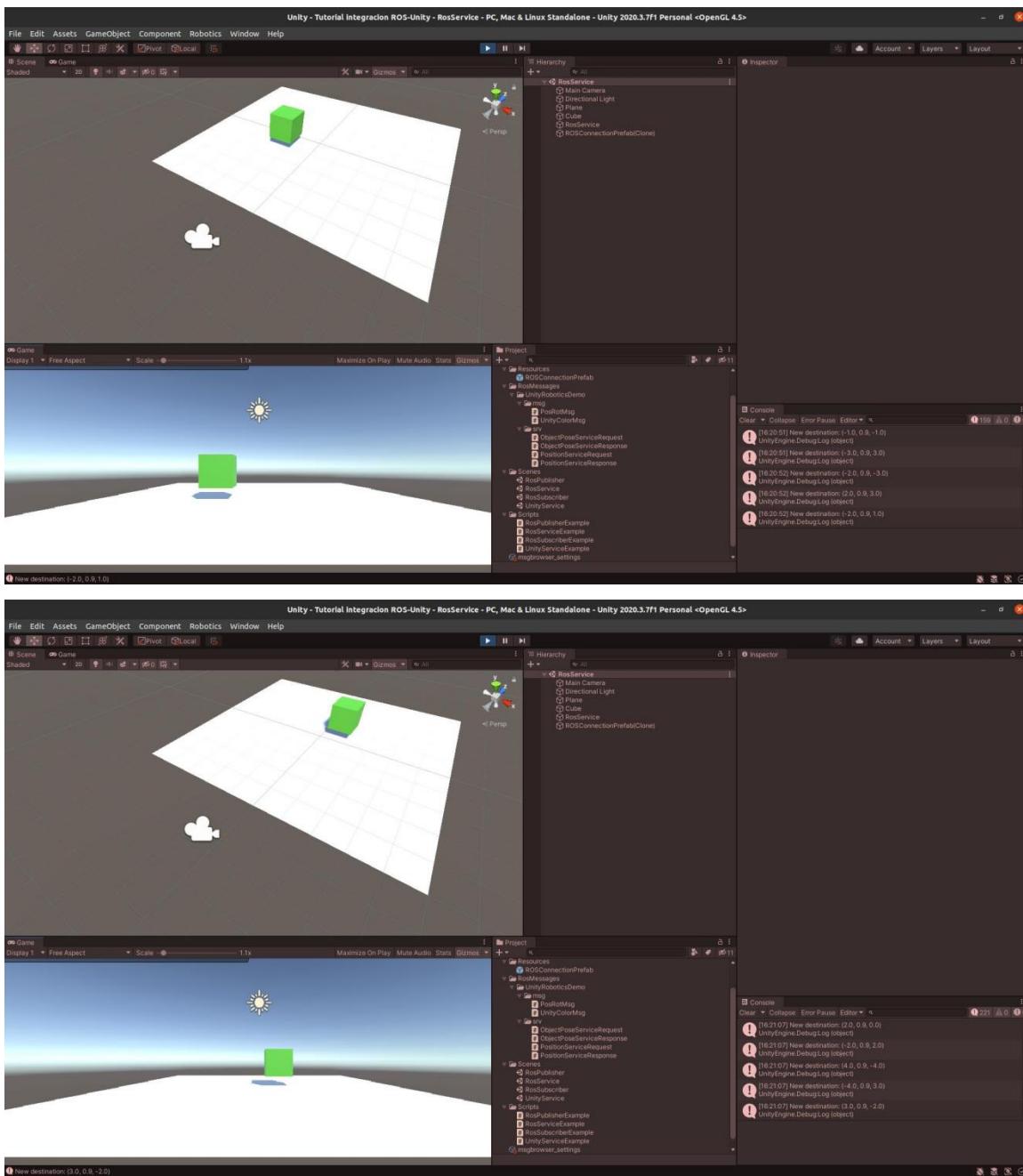
Iniciamos nuestro contenedor Docker, lo cual es importante estar ubicados donde se encuentra el archivo “docker-compose.yaml”.

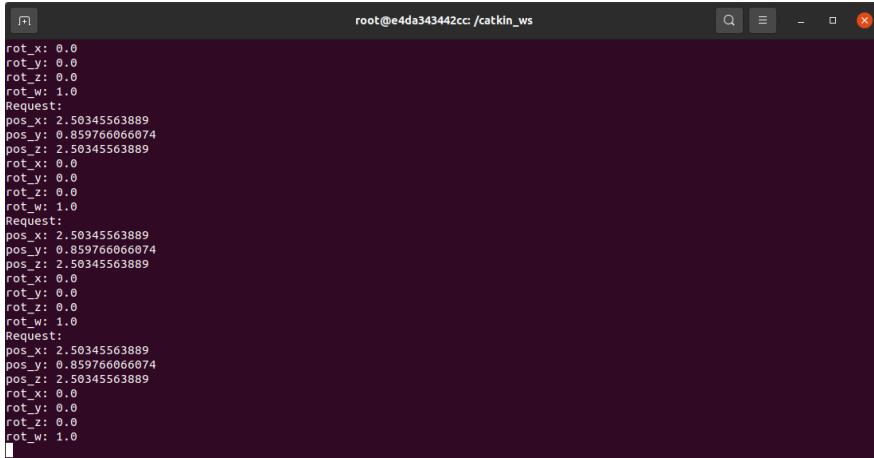
- `roslaunch unity_robots_demo robo_demo.launch &`

Iniciamos nuestro servidor TCP (recordar que en este archivo de lanzamiento solo se ejecuta un nodo).

- `rosrun unity_robots_demo position_service.py &`

La escena de Unity debe estar en play para que pueda moverse el cubo (dado a la respuesta del servicio que se ejecuta en ROS).



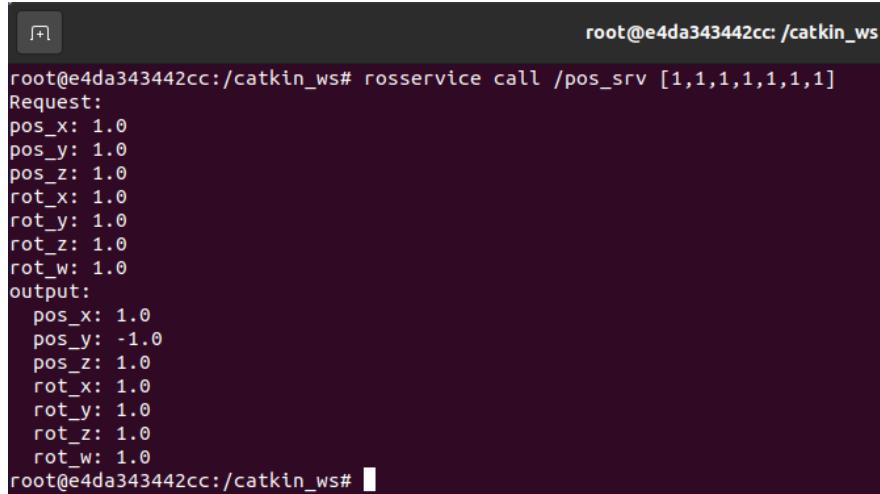


```
root@e4da343442cc: /catkin_ws
rot_x: 0.0
rot_y: 0.0
rot_z: 0.0
rot_w: 1.0
Request:
pos_x: 2.50345563889
pos_y: 0.859766666074
pos_z: 2.50345563889
rot_x: 0.0
rot_y: 0.0
rot_z: 0.0
rot_w: 1.0
Request:
pos_x: 2.50345563889
pos_y: 0.859766666074
pos_z: 2.50345563889
rot_x: 0.0
rot_y: 0.0
rot_z: 0.0
rot_w: 1.0
Request:
pos_x: 2.50345563889
pos_y: 0.859766666074
pos_z: 2.50345563889
rot_x: 0.0
rot_y: 0.0
rot_z: 0.0
rot_w: 1.0
```

NOTA: Las posiciones que devuelve el servicio en ROS son en base al sistema de coordenadas de Unity (por el momento no hacemos conversiones entre sistemas de coordenadas).

Podemos llamar a un servicio desde ROS para probar el funcionamiento del servicio. Para esto ejecutamos el siguiente comando:

```
rosservice call /pos_srv [1,1,1,1,1,1]
rosservice call [service] [args]
```



```
root@e4da343442cc: /catkin_ws
root@e4da343442cc: /catkin_ws# rosservice call /pos_srv [1,1,1,1,1,1]
Request:
pos_x: 1.0
pos_y: 1.0
pos_z: 1.0
rot_x: 1.0
rot_y: 1.0
rot_z: 1.0
rot_w: 1.0
output:
    pos_x: 1.0
    pos_y: -1.0
    pos_z: 1.0
    rot_x: 1.0
    rot_y: 1.0
    rot_z: 1.0
    rot_w: 1.0
root@e4da343442cc: /catkin_ws#
```

Publicar en un tema desde ROS por línea de comandos (como complemento)

Ejecutamos los siguientes comandos:

- `roscore &`

Es un prerequisito tener ejecutando “roscore” antes de empezar a trabajar con

nodos y programas de un sistema basado en ROS.

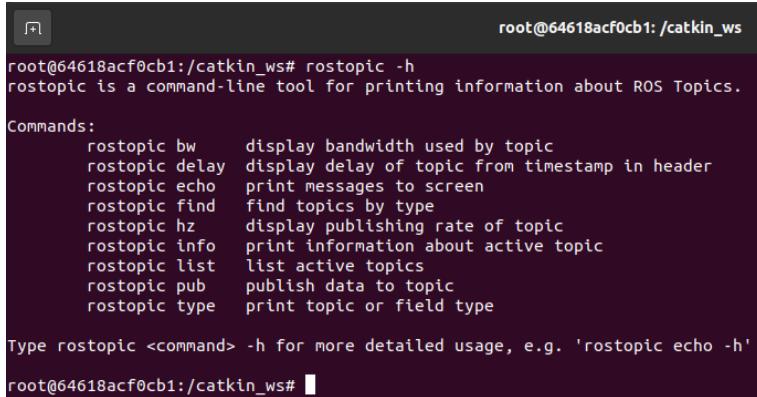
- `rostopic echo /test`

Nos suscribimos a un tema que por el momento aun no se publica nada en el.

Abrimos otra terminal e ingresamos al contenedor ejecutando el siguiente comando:

```
docker exec -it <ID_contenedor> /bin/bash
```

Continuamos con la ejecucion de comandos:



```
root@64618acf0cb1:/catkin_ws# rostopic -h
rostopic is a command-line tool for printing information about ROS Topics.

Commands:
  rostopic bw      display bandwidth used by topic
  rostopic delay   display delay of topic from timestamp in header
  rostopic echo    print messages to screen
  rostopic find    find topics by type
  rostopic hz      display publishing rate of topic
  rostopic info    print information about active topic
  rostopic list    list active topics
  rostopic pub     publish data to topic
  rostopic type   print topic or field type

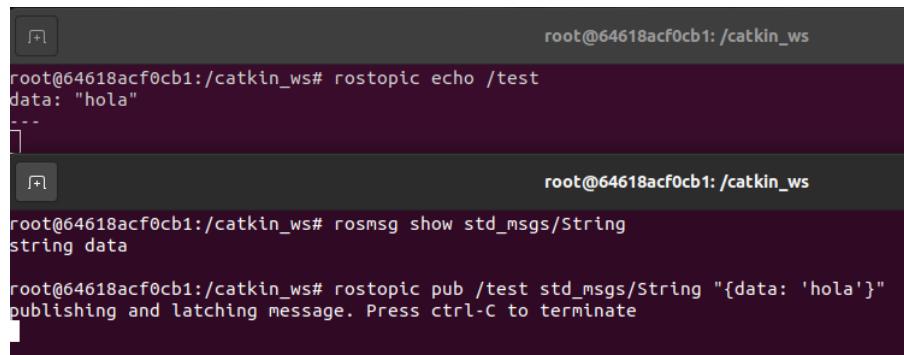
Type rostopic <command> -h for more detailed usage, e.g. 'rostopic echo -h'
root@64618acf0cb1:/catkin_ws#
```

- `rosmsg show std_msgs/String`

Mostramos el tipo de mensaje a utilizar (para ver los campos y los tipos de datos).

- `rostopic pub /test std_msgs/String "{data: 'hola'}"`

Publicamos en el tema “/test” (solo se publica una vez).



```
root@64618acf0cb1:/catkin_ws# rostopic echo /test
data: "hola"
---

root@64618acf0cb1:/catkin_ws# rosmsg show std_msgs/String
string data

root@64618acf0cb1:/catkin_ws# rostopic pub /test std_msgs/String "{data: 'hola'}"
publishing and latching message. Press ctrl-C to terminate
[
```

- `rostopic pub --rate 10 /test std_msgs/String "{data: 'hola'}"`

Publicamos en el tema “/test” a una cierta tasa (especificada en Hz).

Repaso

Hasta el momento hemos realizado lo siguiente:

- Tutorial pick and place

Este tutorial presenta muchos conceptos avanzados pero introduce de buena manera lo que es posible hacer combinando varias tecnologías como lo es ROS, Unity, lenguajes de programación, etc.

Recordando, además, que es posible crear modelos de robot desde cero, pero con su debida complejidad, que podemos simular y manipular como el de este tutorial.

- Tutoriales de iniciación ROS-Unity

- #### ■ Configuración del espacio de trabajo ROS con Docker

Hemos utilizado una tecnología como Docker para poder crear un entorno de trabajo ROS en un contenedor Docker, facilitando la implementación de aplicaciones.

- Publicación en un tema desde una escena de Unity
 - Suscripción a un tema desde una escena de Unity
 - Implementación de un servicio dentro de una escena de Unity
 - Llamada a servicio externo desde una escena de Unity
 - Publicación en un tema desde ROS por línea de comandos

Más adelante nos adentraremos más a fondo a lo que es ROS, hablaremos de cómo está conformado, como crear paquetes ROS desde cero y haremos una práctica simulando

algún proceso real que involucre la comunicación entre máquina y 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>

Es un curso gratuito y bastante bueno.

Material

- 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>

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:tutorial-integracion-ros-unity-terminado`

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:tutorial-integracion-ros-unity-terminado /bin/bash`

- Archivos ROS

- Servidor TCP (para la comunicacion entre ROS y Unity) para el proyecto “Tutorial Integracion ROS-Unity” (**v0.6.0**):

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

- Archivos ROS del proyecto “Tutorial Integracion ROS-Unity” :

<https://github.com/ChuyFernandez/ROS-Tutoriales-de-iniciacion-Integracion-ROS-Unity.git>

Viene incluido:

- Paquetes ROS
 - unity_robotics_demo
 - unity_robotics_demo_msgs
- Archivos para la construccion de la imagen ROS Docker

Estos archivos no contienen el mismo codigo de programacion que viene por defecto en el tutorial que se nos presenta en GitHub, ya

que, recordando, se estuvieron modificando.

- Archivos Unity
 - Paquetes “ROS TCP Connector” (**v0.6.0**):
https://github.com/Unity-Technologies/Unity-Robotics-Hub/blob/main/tutorials/quick_setup.md
 - Paquetes proyecto “Pick and Place” (**v0.6.0**):
<https://github.com/Unity-Technologies/Unity-Robotics-Hub>
 - Proyecto “Tutorial Integracion ROS-Unity” :
<https://github.com/ChuyFernandez/UNITY-Tutoriales-de-iniciacion-Integracion-ROS-Unity.git>

Este proyecto fue creado desde cero para llevar a cabo la serie de tutoriales presentados anteriormente.
Solo incluye la carpeta “Assets” (es la carpeta en donde se crearon las escenas, los scripts, etc.).

CONCLUSION

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

Todo lo que se ha hecho se ha ido documentando en un cuaderno, tanto como los problemas que me han surgido, algo de código, comentarios, conceptos importantes, etc.

Cabe destacar que es importante verificar las versiones de código de los repositorios que se encuentran en el sitio <https://github.com/Unity-Technologies/Unity-Robotics-Hub>, ya que por el mismo motivo estuve aclarando que versiones utilice para llevar a cabo la práctica, esto debido a que el código cambia bastante de una versión a otra.