

# Memoria Middleware

Roger Fernández

## Índice

<b>1. Instalar ROS1 Noetic en ubuntu 20.04 LTS</b>	<b>1</b>
<b>2. Instalar ROS2 humble en ubuntu 22.04 LTS</b>	<b>4</b>
<b>3. Emular ROS1 Noetic y ROS2 Humble en docker</b>	<b>5</b>
3.0.1. Dockerizando ROS 1 Noetic . . . . .	5
3.0.2. Dockerizando ROS 2 Humble . . . . .	8
<b>4. Dockerizando Yarp (Yet Another Robotic Platform!)</b>	<b>9</b>
4.1. Ejemplo de publish-subscribe básico en Yarp . . . . .	11
4.1.1. Prueba de escritura de puertos: . . . . .	12
<b>5. Coreographer para NAOqi</b>	<b>13</b>
<b>6. Creación de un servicio básico para Naoqi en python3 (Ubuntu 22.04)</b>	<b>13</b>
<b>7. Instalar drivers camara luxonis y correr demo</b>	<b>19</b>
<b>8. Instalar drivers RPLIDAR S2 en ROS2 humble</b>	<b>21</b>

## 1. Instalar ROS1 Noetic en ubuntu 20.04 LTS

Para la instalación acudimos a la página <http://wiki.ros.org/noetic/Installation/Ubuntu> y seguimos las instrucciones para la instalación de ROS Noetic en Ubuntu 20.04 LTS.

- 1. Configuración de los repositorios de Ubuntu:** Buscamos en nuestro ubuntu 20.04 el programa software y actualizaciones y permitimos que ubuntu use los repositorios “restricted”, “universe” y “multiverse” tal y como se muestra en la imagen:

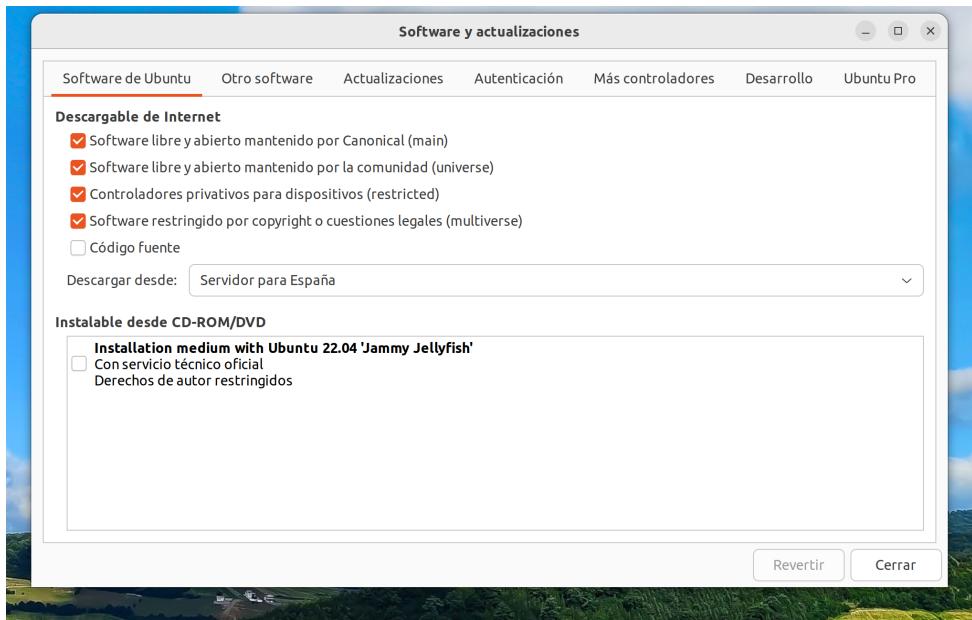


Figura 1: paquetes restricted, universe y multiverse

2. **Establecer la lista de fuentes para descargar software.** Abrimos una terminal de ubuntu y usamos el comando:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Ahora nuestro equipo ya puede descargar software de packages.ros.org.

3. **Establecimiento de las claves:** para ello introducimos en la terminal los comandos:

```
sudo apt install curl # solo es necesario si no tenemos curl instalado
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc
| sudo apt-key add -
```

4. **Instalación de ROS1 noetic:** En primer lugar nos aseguramos que el índice de paquetes de Debian esté actualizado mediante el comando:

```
sudo apt update
```

A continuación instalamos el paquete completo de ROS1 Noetic mediante la instrucción:

```
sudo apt install ros-noetic-desktop-full
```

**Nota:** Ésta se trata de la opción recomendada, sin embargo existen otras opciones más ligeras de instalación mediante los comandos de terminal:

```
sudo apt install ros-noetic-desktop
```

En este caso se instala la base de ROS y algunas herramientas como rqt y rviz

```
sudo apt install ros-noetic-ros-base
```

En este caso se instala la base de ROS1 Noetic: paquetes básicos de ROS, bibliotecas de construcción y comunicación de ROS. No se instalan herramientas de interfaz gráfica.

**Nota:** Existen muchos más paquetes disponibles para instalar en ROS1 Noetic. Estos paquetes pueden instalarse ejecutando en la terminal la instrucción:

```
sudo apt install ros-noetic-nombrePaquete
```

Para encontrar paquetes disponibles puede usarse el índice de ROS <https://index.ros.org/packages/page/1/time/#noetic> o usar la instrucción de terminal:

```
apt search ros-noetic
```

5. **Cargando el entorno de ROS:** cada vez que queramos usar ROS1 noetic en la terminal de ubuntu debemos cargar el entorno de ROS mediante la instrucción de terminal:

```
source /opt/ros/noetic/setup.bash
```

Suele ser conveniente que esta instrucción se ejecute al inicio de cada nueva sesión de terminal. Para ello debemos editar el archivo .bashrc e incluir la instrucción anterior:

```
source /opt/ros/noetic/setup.bash
```

Debe tenerse en cuenta que las instrucciones contenidas en .bashrc se ejecutan cada vez que se abre una nueva terminal (se están cargando todas las variables de entorno)

Otra opción es modificar directamente el archivo .bashrc mediante la instrucción:

```
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
```

Esto escribirá la instrucción anterior en el documento .bashrc. Finalmente se debe cargar el archivo .bashrc mediante

```
source ~/.bashrc
```

o iniciar una nueva sesión de terminal.

6. **Dependencias para la construcción de paquetes:** Hasta ahora sólo se ha instalado lo necesario para ejecutar los principales paquetes de ROS. Para crear y administrar tus propios espacios de trabajo de ROS, hay diversas herramientas y requerimientos distribuidos de modo separado. Por ejemplo, rosinstall es una herramienta en línea de comandos que te permite descargar el código fuente de los paquetes de ROS con un solo comando. Para instalar esta herramienta y otras dependencias para construir paquetes ROS, ejecuta en la terminal la instrucción:

```
sudo apt install python3-rosdep python3-rosinstall  
python3-rosinstall-generator python3-wstool build-essential
```

Otra herramienta útil es rosdep, la cual te permite instalar dependencias del sistema a partir del código fuente (el cual deberá ser compilado) y la cual permite ejecutar algunas de las componentes principales de ROS.

Para instalar rosdep se ejecuta en la terminal el comando:

```
sudo apt install python3-rosdep
```

Con el siguiente comando se inicia rosdep:

```
sudo rosdep init
```

y con este comando se actualiza rosdep:

```
rosdep update
```

## 2. Instalar ROS2 humble en ubuntu 22.04 LTS

De un modo análogo se puede instalar ROS 2 Humble en ubuntu 22.04. Siguiendo el tutorial de la página

<https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html>  
puede realizarse la instalación.

### 3. Emular ROS1 Noetic y ROS2 Humble en docker

Otro modo de usar Ros es mediante la virtualización usando docker. Lo más sencillo posiblemente sea descargar algunas imágenes de ubuntu que ya traigan ROS instalado.

#### 3.0.1. Dockerizando ROS 1 Noetic

Una forma conveniente es usar una imagen que venga con vnc para poder ver nuestro sistema virtualizado a través del navegador. En nuestro caso usaremos la imagen de docker que viene en el repositorio de github: <https://github.com/Tiryoh/docker-ros-desktop-vnc>

Desde la terminal hacemos el docker pull de esta imagen con el tag noetic:

```
docker pull tiryoh/ros-desktop-vnc:noetic
```

Para lanzar un contenedor de la imagen que acabamos de descargar ponemos en la terminal:

```
docker run -p 6080:80 --shm-size=512m --name = maquinaNoetic  
tiryoh/ros-desktop-vnc:noetic
```

Con esto se lanzará un contenedor de la imagen previamente descargada.

- Con la opción -p 6080:80 se le indica que conecte el puerto 6080 de nuestra máquina al puerto 80 del contenedor de la imagen . Lógicamente podemos elegir otro puerto que se encuentre sin ningún servicio.
- con la opción --shm-size=512m se le está indicando que la memoria RAM máxima que se le asigna al contenedor es 512 megabytes. Si queremos asignarle más podemos poner por ejemplo --shm-size = 4G (en este caso se le indican que el contenedor puede usar hasta 4 Gigabytes de memoria RAM)
- --name = maquinaNoetic indica el nombre del contenedor que se va a crear, en este caso el nombre es maquinaNoetic, pero lógicamente podemos darle el nombre que nosotros queramos.

Acabamos de lanzar una imagen de ubuntu con ros instalado.

Ahora debemos conectarnos de algún modo al contenedor. Si queremos conectarnos vía la terminal del anfitrión con una terminal del contenedor, bastará con ejecutar en terminal:

```
docker exec -it maquinaNoetic bash
```

Esta opción es posible, aunque en muchas ocasiones no es la más recomendable. Como la imagen de docker viene con VNC instalado y ya hemos hecho la configuración, podemos acceder al contenedor vía nuestro navegador a partir en este caso del puerto 6080 que fue configurado. Para ello entramos a la barra de búsquedas del navegador e ingresamos:

```
localhost:6080
```

Ahora ya tenemos acceso al escritorio del contenedor. Para comprobar que ros noetic está instalado basta con ingresar a la terminal y escribir:

```
rosversion -d
```

Nos devolverá noetic que es la versión instalada. Este comando devuelve la versión de ROS instalada en el contenedor (en este caso Noetic).

Para ver el estado de los contenedores (si están funcionando su estado es up y si están parados su estado será exited) docker podemos usar el comando:

```
docker ps -a
```

Si queremos parar el contenedor escribimos en la terminal:

```
docker stop maquinaNoetic
```

Por el contrario si queremos inicializar el contenedor no tenemos más que ingresar en la terminal el comando:

```
docker start maquinaNoetic
```

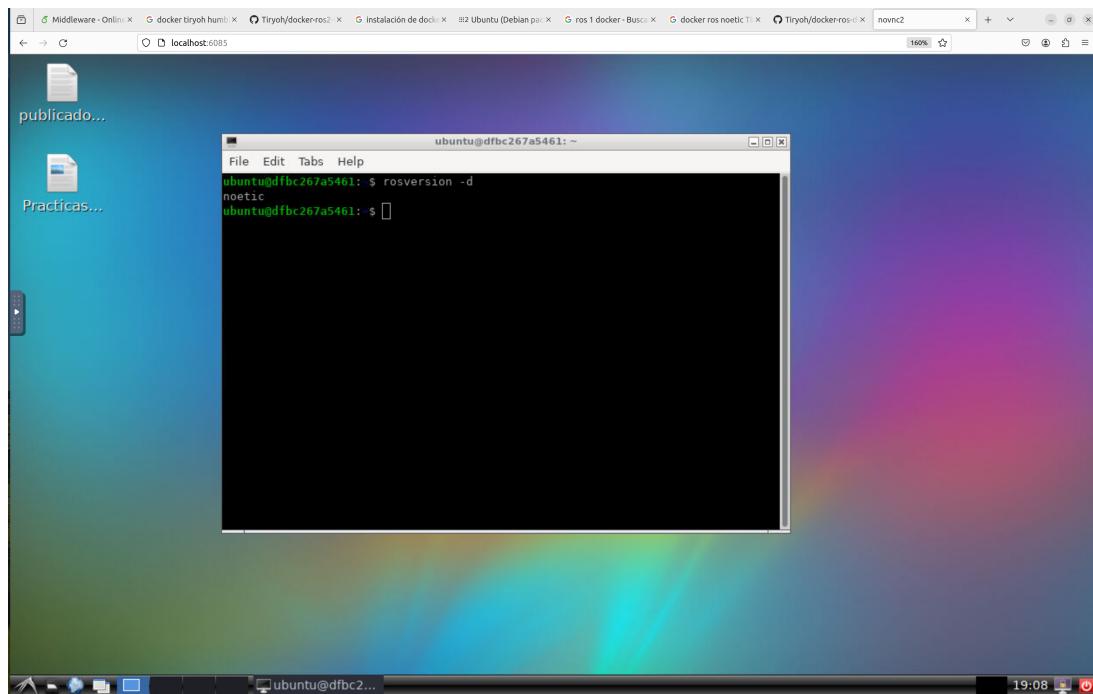


Figura 2: Dockerización para ros 1 Noetic

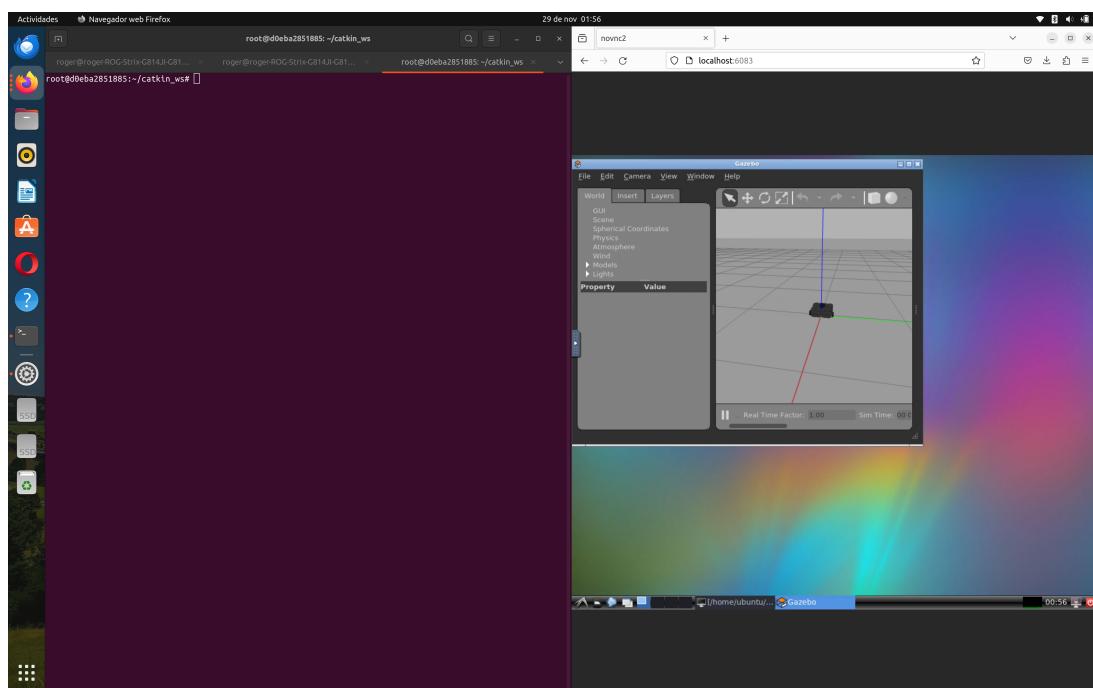


Figura 3: Dockerización para ros 1: Lanzando robot waffle pi en mundo vacío de Gazebo

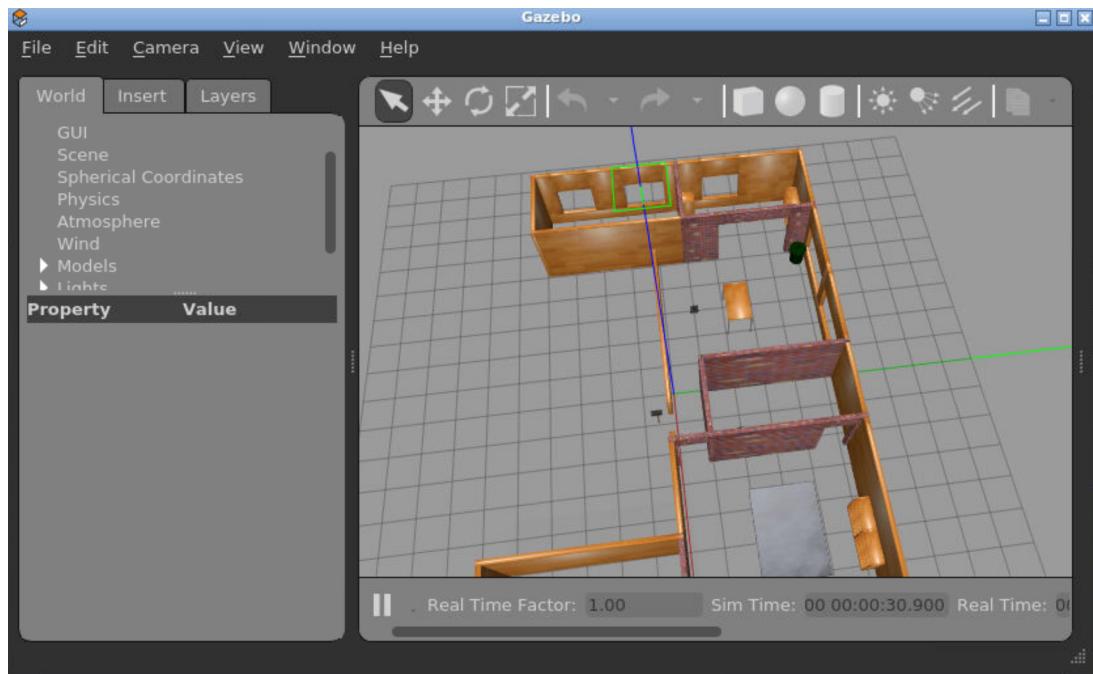


Figura 4: Lanzando waffle pi en el mundo del apartamento de Gazebo

### 3.0.2. Dockerizando ROS 2 Humble

EL proceso es análogo que para el caso de ROS 1 noetic. En nuestro caso hemos usado el repositorio de GitHub <https://github.com/Tiryoh/docker-ros2-desktop-vnc>. Para hacer el docker pull escribimos:

```
docker pull tiryoh/ros2-desktop-vnc:humble
```

Para lanzar un contenedor de la imagen escribimos:

```
docker run -p 6080:80 --security-opt seccomp=unconfined  
--shm-size=512m --name maquinaHumble tiryoh/ros2-desktop-vnc:humble
```

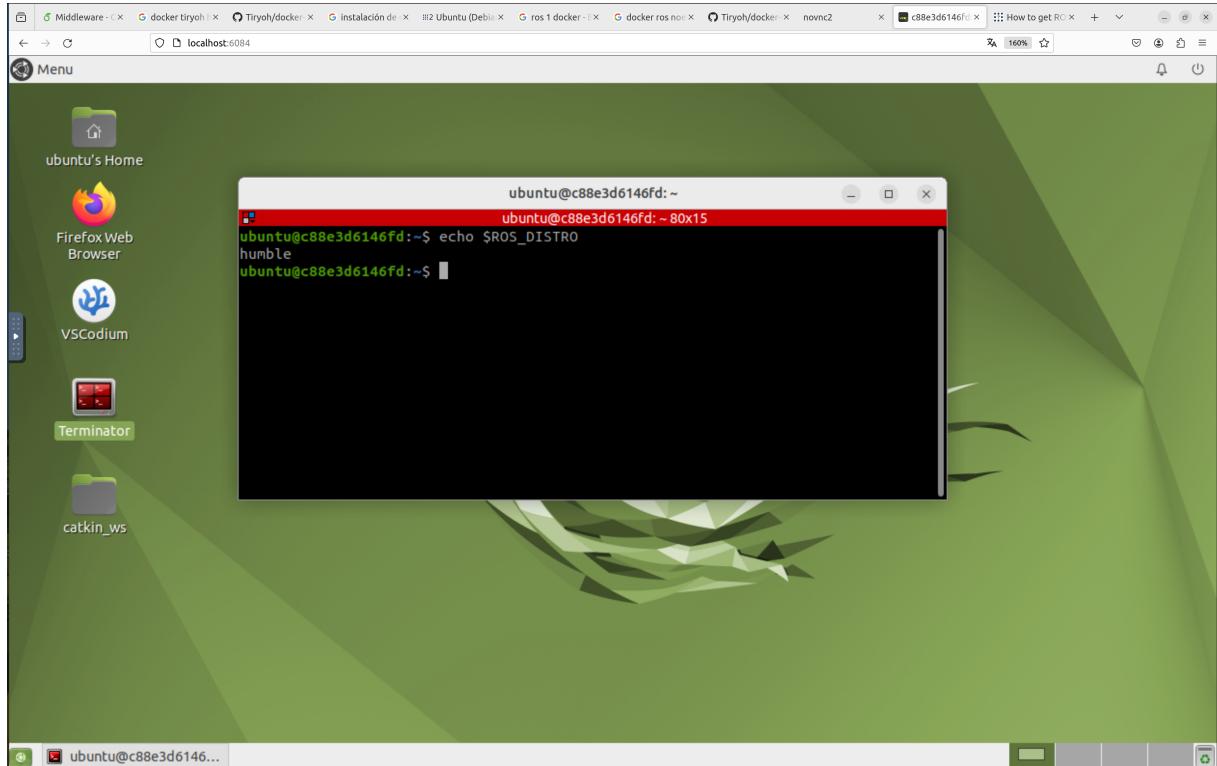


Figura 5: Dockerización para ros 2 Humble

## 4. Dockerizando Yarp (Yet Another Robotic Platform!)

Copiamos el repositorio de github de robontology yarp:

```
git clone https://github.com/robontology/yarp.git
```

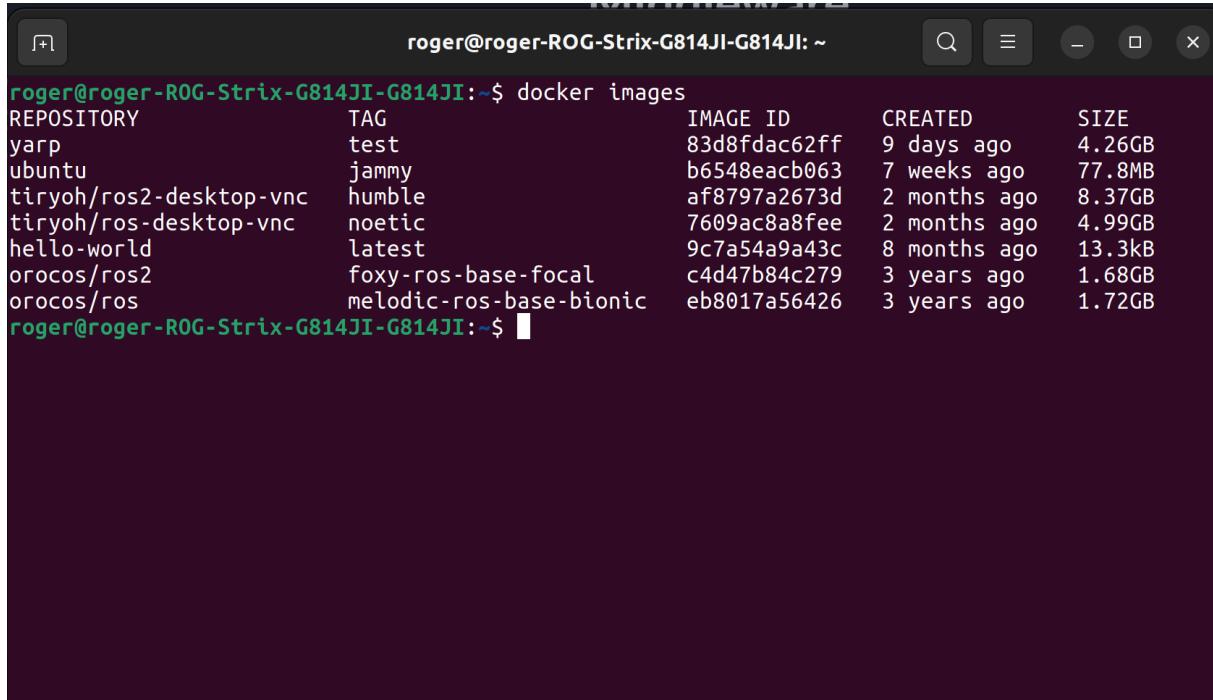
Aparece una carpeta con nombre yarp. Nos movemos a ese directorio:

```
cd yarp
```

Ejecutando ls vemos que dentro existe una carpeta llamada docker. Nos movemos a ese directorio cd docker (ejecutamos un ls y vemos que aquí dentro hay un dockerfile que hay que construir con un docker build). Construimos la imagen de docker:

```
docker build . -f Dockerfile -t yarp:test
```

Si hacemos docker images, nos aparecerá una imagen de docker con nombre yarp y tag test.



The screenshot shows a terminal window with the following text:

```
roger@roger-ROG-Strix-G814JI-G814JI:~$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
yarp                test     83d8fdac62ff  9 days ago   4.26GB
ubuntu              jammy   b6548eacb063  7 weeks ago  77.8MB
tiryoh/ros2-desktop-vnc  humble  af8797a2673d  2 months ago  8.37GB
tiryoh/ros-desktop-vnc  noetic  7609ac8a8fee  2 months ago  4.99GB
hello-world         latest   9c7a54a9a43c  8 months ago  13.3kB
orocos/ros2         foxy-ros-base-focal  c4d47b84c279  3 years ago  1.68GB
orocos/ros          melodic-ros-base-bionic  eb8017a56426  3 years ago  1.72GB
roger@roger-ROG-Strix-G814JI-G814JI:~$
```

Figura 6: Imagen de Yarp para docker

Finalmente lanzamos un contenedor de docker para la imagen de yarp:

```
docker run -it --name maquinaYarp yarp:test
```

Ejecutamos la instrucción:

```
lsb_release -a
```

Esto devuelve la información de que el sistema operativo es una máquina de ubuntu 22.04.3 LTS (jammy).

Ejecutamos la instrucción:

```
yarp server
```

Esto es algo análogo a ejecutar el roscore en ROS1.

```

user1 ~/robotology
roger@roger-ROG-Strix-G814JI-G814JI:~ 
REPOSITORY          TAG        IMAGE ID      CREATED     SIZE
yarp                test       83d8fdac62ff  9 days ago   4.26GB
ubuntu              jammy     b6548eacb063  7 weeks ago  77.8MB
tiryooh/ros2-desktop-vnc  humble   af8797a2673d  2 months ago  8.37GB
tiryooh/ros-desktop-vnc  noetic    7609ac8a8fee  2 months ago  4.99GB
hello-world         latest    9c7a54a9a43c  8 months ago  13.3kB
orocos/ros2          foxy      c4d47b84c279  3 years ago   1.68GB
orocos/ros            melodic   eb8017a56426  3 years ago   1.72GB
roger@roger-ROG-Strix-G814JI-G814JI:~$ docker run -it --name maquinaYarp yarp:test
user1 ~/robotology $ lsb release -a
bash: lsb: command not found
user1 ~/robotology $ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.3 LTS
Release:        22.04
Codename:      jammy
user1 ~/robotology $ yarp server
=====
| \V/ /--| | --\ | / |
| } // / | | / \ } | / |
| / / /-| | / / | \ / |
| / / /-| | / / | \ / |
=====
Call with --help for information on available options
Using port database: :memory:
Using subscription database: :memory:
IP address: default
Port number: 10000
[INFO] |yarp.os.Port|/root| Port /root active at tcp://172.17.0.3:10000/
Registering name server with itself
* register "/root" tcp "172.17.0.3" 10000
+ set "/root" offers http name_ser local tcp fast_tcp mcast udp text text_ack bayer mjpeg portmonitor priority shmem unix_stream
+ set "/root" accepts http name_ser local tcp fast_tcp mcast udp text text_ack bayer mjpeg portmonitor priority shmem unix_stream
+ set "/root" ips "127.0.0.1" "172.17.0.3"
+ set "/root" process 14
* register fallback mcast "224.2.1.1" 10000
+ set fallback offers http name_ser local tcp fast_tcp mcast udp text text_ack bayer mjpeg portmonitor priority shmem unix_stream
+ set fallback accepts http name_ser local tcp fast_tcp mcast udp text text_ack bayer mjpeg portmonitor priority shmem unix_stream
+ set fallback ips "127.0.0.1" "172.17.0.3"
+ set fallback process 14
* set "/root" nameserver "true"
Name server can be browsed at http://172.17.0.3:10000/
Ok. Ready!
Name server running happily

```

Figura 7: Imagen del Yarp Server

### 4.1. Ejemplo de publish-subscribe básico en Yarp

Conectamos otras dos terminales al contenedor de yarp que se está ejecutando mediante la instrucción:

```
docker exec -it maquinaYarp bash
```

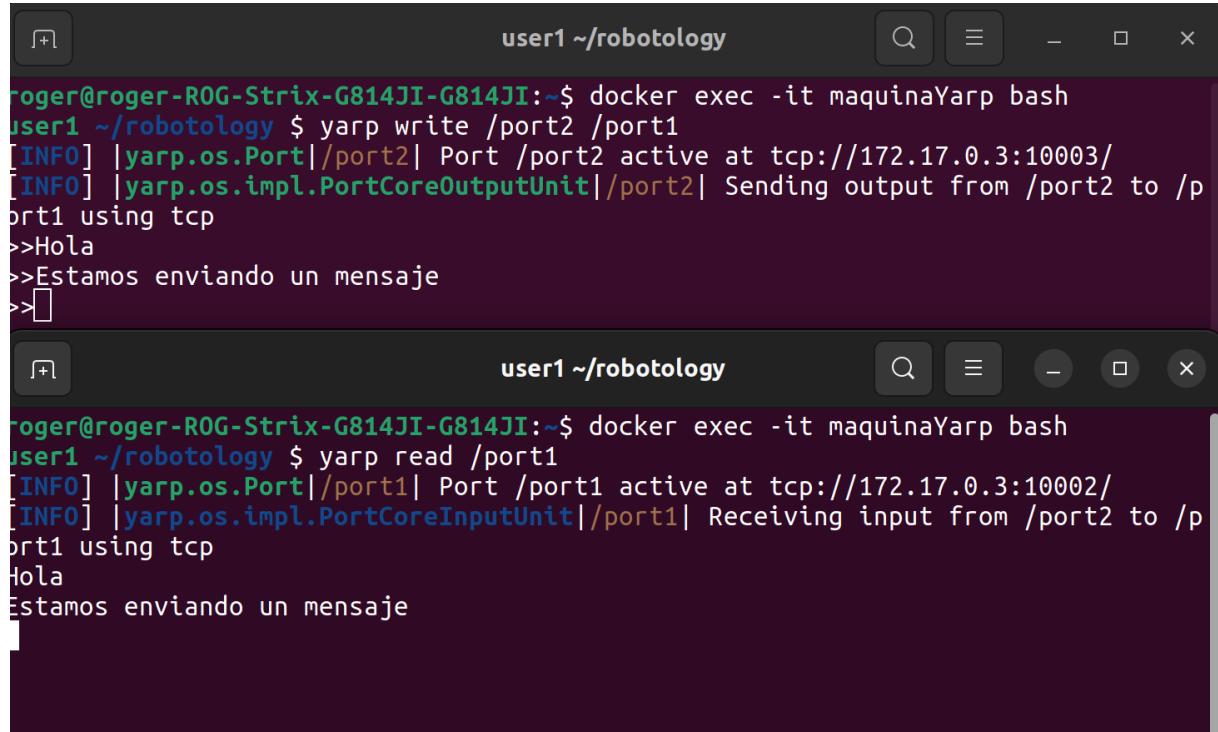
En una de las terminales ejecutamos:

```
yarp read /port1
```

En la otra ejecutamos:

```
yarp write /port2 /port1
```

Ahora si escribimos algo en la segunda terminal y pulsamos intro, el mensaje se envía a la terminal 1 (Se envía output de /port1 a /port2)



The image shows two terminal windows side-by-side. Both windows have a dark background and a title bar 'user1 ~/robotology'. The left terminal window contains the following text:

```
roger@roger-ROG-Strix-G814JI-G814JI:~$ docker exec -it maquinaYarp bash
user1 ~/robotology $ yarp write /port2 /port1
[INFO] |yarp.os.Port|/port2| Port /port2 active at tcp://172.17.0.3:10003/
[INFO] |yarp.os.impl.PortCoreOutputUnit|/port2| Sending output from /port2 to /port1 using tcp
>>Hola
>>Estamos enviando un mensaje
>>[]
```

The right terminal window contains the following text:

```
roger@roger-ROG-Strix-G814JI-G814JI:~$ docker exec -it maquinaYarp bash
user1 ~/robotology $ yarp read /port1
[INFO] |yarp.os.Port|/port1| Port /port1 active at tcp://172.17.0.3:10002/
[INFO] |yarp.os.impl.PortCoreInputUnit|/port1| Receiving input from /port2 to /port1 using tcp
Hola
Estamos enviando un mensaje
```

Figura 8: Enviando mensaje en Yarp

#### 4.1.1. Prueba de escritura de puertos:

Ejecutamos:

yarp read /port1:

En este caso el puerto 1 está a la escucha.

En otra terminal conectada al contenedor de Yarp se ejecuta:

yarp write /port2

Ya se pueden escribir mensajes, pero nadie los recibe porque nadie se ha conectado al puerto 2.

Finalmente conectamos el puerto de escritura 2 al puerto de lectura 1:

yarp connect /port2 /port1

(cuidado: se debe conectar el puerto que está escribiendo al que está leyendo pero no al revés ya que no funciona.)

Todo lo que escribamos en la máquina conectada al puerto 2 de escritura sera recibido en la máquina del puerto 1 de lectura.

```

user1 ~/robotology $ yarp read /port1
[INFO] [yarp.os.Port] [/port1] Port /port1 active at tcp://172.17.0.3:10002/
[INFO] [yarp.os.impl.PortCoreInputUnit] [/port1] Receiving input from /port2 to /port1 using tcp
>>Este mensaje lo recibira el puerto 1
>>
user1 ~/robotology $ yarp write /port2
[INFO] [yarp.os.Port] [/port2] Port /port2 active at tcp://172.17.0.3:10003/
>>Este mensaje nadie lo recibira
>>[INFO] [yarp.os.impl.PortCoreOutputUnit] [/port2] Sending output from /port2 to /port1 using tcp
>>Este mensaje lo recibira el puerto 1
>>
user1 ~/robotology $ yarp connect /port2 /port1
[INFO] [yarp.os.Network] Success: Added connection from /port2 to tcp://port1
user1 ~/robotology $ 

```

Figura 9: Escritura de puertos en Yarp

## 5. Coreographer para NAOqi

Decargamos el software coreographer para naoqi de la página aldebaran:

<https://www.aldebaran.com/en/support/pepper-naoqi-2-9/downloads-software>

El Choreographer es una herramienta de programación visual que permite a los usuarios diseñar secuencias de movimientos, interacciones y comportamientos para el robot Pepper de manera intuitiva, sin necesidad de escribir código. Con esta herramienta, puedes crear fácilmente coreografías complejas, programar comportamientos específicos y controlar las acciones del robot. Simplemente nos conectamos al robot pepper por ssh:

ssh nao@iprobot

Introducimos la contraseña abrimos el Coreographer y ya podemos interactuar fácilmente con el robot mediante la interfaz.

## 6. Creación de un servicio básico para Naoqi en python3 (Ubuntu 22.04)

- Para la creación de un servicio básico en Naoqi usaremos el SDK de C++ para linux que encontramos en la página: de aldebaran: <https://www.aldebaran.com/en/support/nao-6/downloads-software>.

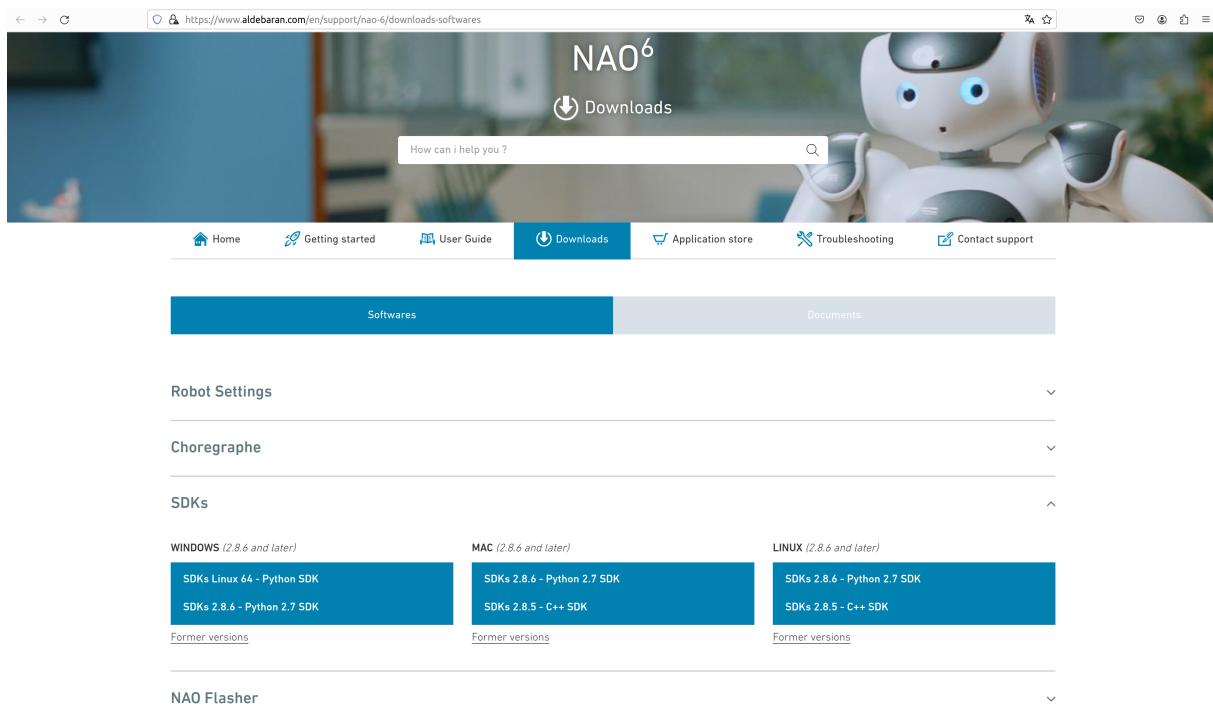


Figura 10: Página oficial de descarga del SDK para naoqi

Una vez descargado el SDK de nombre naoqi-sdk-2.8.5.10-linux64.tar.gz, lo descomprimimos. Desde la terminal accedemos a la carpeta del SDK y ejecutamos el binario de nombre naoqi presente en la misma mediante la instrucción:

```
./naoqi
```

Esto lanza un análogo al roscore para ROS. Sin esta instrucción ejecutada no nos van a funcionar el servicio que creemos.

2. A continuación instalamos la biblioteca qi para python3 mediante la instrucción:

```
pip install qi
```

Necesitamos hacer la instalación de la misma ya que no viene incluida por defecto en python3.

3. Creación del servicio en python3: Para la creación de un servicio básico de prueba usamos la información de la página: <http://doc.aldebaran.com/2-5/dev/libqi/>

guide/py-service.html. Creamos un documento de texto plano con nombre servicio.py en otra carpeta (no hace falta hacerlo dentro del SDK) e incluimos el código python necesario, el cual se muestra a continuación:

```

1 import qi
2 import sys
3
4 class MyFooService:
5     def __init__(self, *args, **kwargs):
6         #se define la señal 'onBang'
7         self.onBang = qi.Signal()
8
9     #se define el método bang que activará la señal onBang
10    def bang(self):
11        #activamos la señal con el valor 42
12        self.onBang(42)
13        print("Servicio - funcionando")
14
15 #Creamos una aplicación
16 app = qi.Application()
17 app.start()
18
19 #Creamos una instancia de la clase MyFooService
20 myfoo = MyFooService()
21
22 s = app.session
23 #Registramos nuestro servicio con el nombre "foo"
24 id = s.registerService("foo", myfoo)
25
26 #Ejecutamos la aplicación
27 app.run()
```

#### 4. Creación del cliente en python3:

Para la creación del cliente también seguimos las instrucciones de la página <http://doc.aldebaran.com/2-5/dev/libqi/guide/py-service.html>. Creamos un documento de texto plano con nombre cliente.py en la misma carpeta en la que se encuentra el servicio e incluimos el código python necesario, el cual mostramos a continuación:

```

1 import qi
2 import sys
```

```
3
4 def onBangCb(i):
5     print ("bang: " , + i)
6
7 app = qi.Application()
8 app.start()
9 s = app.session
10 foo = s.service("foo")
11
12 #reistramos un callback en 'onBang'
13 foo.onBang.connect(onBangCb)
14 #call bang
15 foo.bang()
```

**Nota:** El código puede verse también desde mi repositorio de GitHub

<https://github.com/RogerFerEnr/servicioNaoqi>

5. Ejecución con Python3 del servidor y el cliente:

Necesitamos tres terminales abiertas para la ejecución del servidor y el cliente. En primer lugar tenemos que ejecutar el binario del SDK de naoqi (**sólo si no lo hicimos en el paso 1**). Para ello dentro de la carpeta del SDK ejecutamos la instrucción:

```
./naoqi
```

En otra terminal vamos a la carpeta en la que se encuentra el servidor y ejecutamos la instrucción:

```
python3 servidor.py
```

Finalmente en otra terminal vamos a la carpeta en la que se encuentra el cliente y ejecutamos la instrucción:

```
python3 cliente.py
```

La imagen que se muestra a continuación muestra el resultado de las operaciones realizadas:

## 6 Creación de un servicio básico para Naoqi en python3 (Ubuntu 22.04) 17

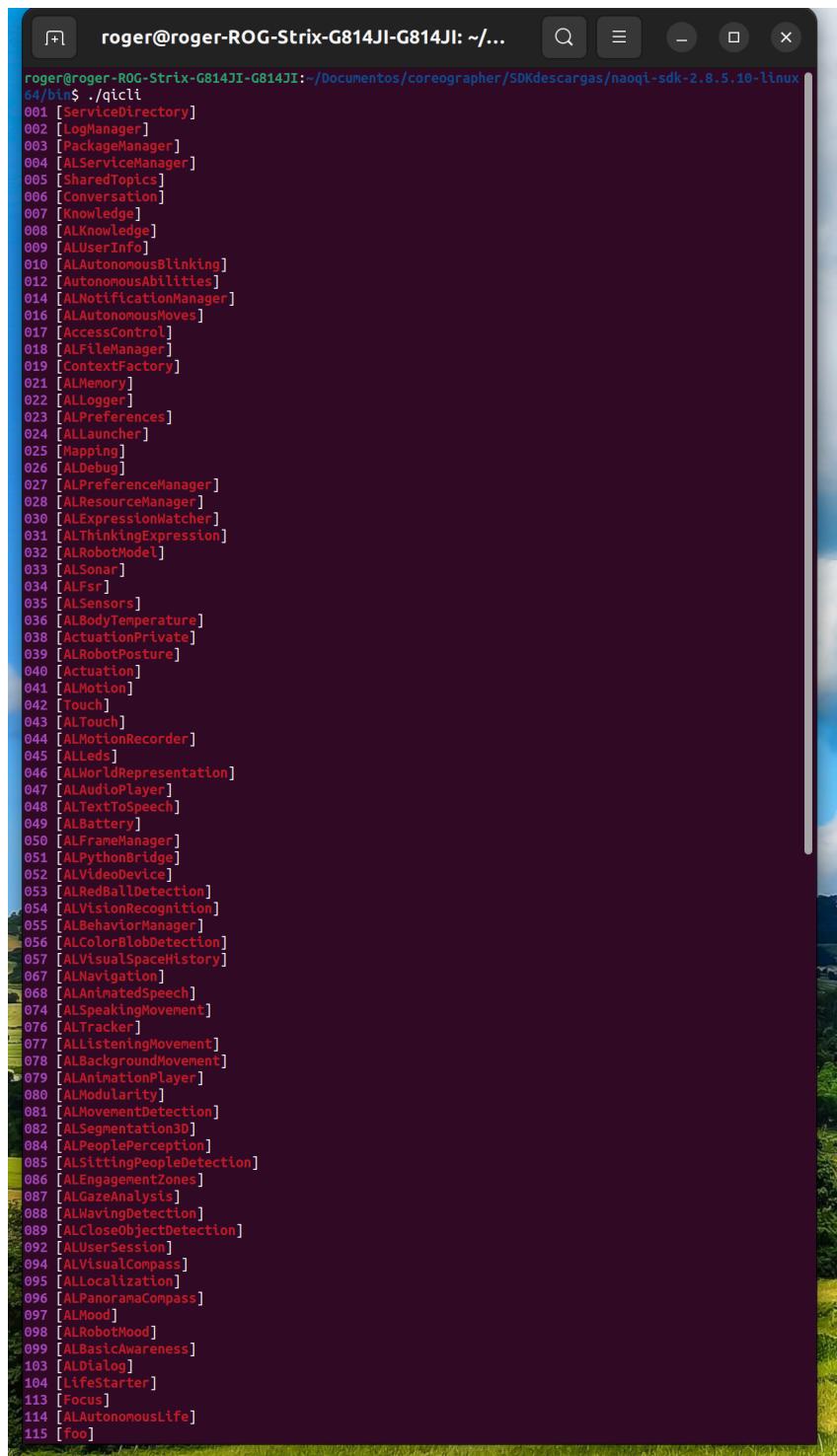
The figure consists of three separate terminal windows, each showing a different step in the process of creating a basic service for Naoqi in Python3.

- Terminal 1:** Shows the command `python3 servicio.py` being run. The output indicates that the application was created successfully, and it shows the service names "servicio" and "LogManager".
- Terminal 2:** Shows the command `python3 cliente.py` being run. The output indicates that the application was created successfully, and it shows the service names "PackageManager" and "expressivity".
- Terminal 3:** Shows the command `python3 cliente.py` being run again. The output indicates that the application was created successfully, and it shows the service names "dialog" and "Conversation".

Figura 11: Servicio y cliente en acción

- Podemos comprobar la existencia del servicio creado ejecutando el binario qicli. Para ello dentro de la carpeta bin del SDK de C++ ejecutamos la instrucción:

```
./qicli
```



```
roger@roger-ROG-Strix-G814JI-G814JI:~/Documentos/coreographer/SDKdescargas/naoqi-sdk-2.8.5.10-linux
64/bin$ ./qicli
001 [ServiceDirectory]
002 [LogManager]
003 [PackageManager]
004 [ALServiceManager]
005 [SharedTopics]
006 [Conversation]
007 [Knowledge]
008 [ALKnowledge]
009 [ALUserInfo]
010 [ALAutonomousBlinking]
012 [AutonomousAbilities]
014 [ALNotificationManager]
016 [ALAutonomousMoves]
017 [AccessControl]
018 [ALFileManager]
019 [ContextFactory]
021 [ALMemory]
022 [ALLogger]
023 [ALPreferences]
024 [ALLauncher]
025 [Mapping]
026 [ALDebug]
027 [ALPreferenceManager]
028 [ALResourceManager]
030 [ALExpressionWatcher]
031 [ALThinkingExpression]
032 [ALRobotModel]
033 [ALSonar]
034 [ALFsr]
035 [ALSensors]
036 [ALBodyTemperature]
038 [ActuationPrivate]
039 [ALRobotPosture]
040 [Actuation]
041 [ALMotion]
042 [Touch]
043 [ALTtouch]
044 [ALMotionRecorder]
045 [ALLEds]
046 [ALWorldRepresentation]
047 [ALAudioPlayer]
048 [ALTextToSpeech]
049 [ALBattery]
050 [ALFrameManager]
051 [ALPythonBridge]
052 [ALVideoDevice]
053 [ALRedBallDetection]
054 [ALVisionRecognition]
055 [ALBehaviorManager]
056 [ALColorBlobDetection]
057 [ALVisualSpaceHistory]
067 [ALNavigation]
068 [ALAnimatedSpeech]
074 [ALSpeakingMovement]
076 [ALTracker]
077 [ALListenningMovement]
078 [ALBackgroundMovement]
079 [ALAnimationPlayer]
080 [ALModularity]
081 [ALMovementDetection]
082 [ALSegmentation3D]
084 [ALPeoplePerception]
085 [ALSittingPeopleDetection]
086 [ALEngagementZones]
087 [ALGazeAnalysis]
088 [ALWavingDetection]
089 [ALCloseToObjectDetection]
092 [ALUserSession]
094 [ALVisualCompass]
095 [ALLocalization]
096 [ALPanoramaCompass]
097 [ALMood]
098 [ALRobotMood]
099 [ALBasicAwareness]
103 [ALDialog]
104 [LifeStarter]
113 [Focus]
114 [ALAutonomousLife]
115 [foo]
```

Figura 12: Comprobación de servicios de naoqi existentes

Al hacer esto se nos muestran todos los servicios activos, entre los que encontramos el que hemos creado con nombre foo.

## 7. Instalar drivers camara luxonis y correr demo

Veamos el proceso para instalar los drivers de la cámara luxony. Vamos a la página web de la cámara OAK D-PRO y descargamos un archivo python para ejecutar los drivers de la cámara.

Primero actualizamos el sistema:

```
sudo apt-get update  
sudo apt-get upgrade
```

Como no tengo pip instalado para python3 me da error al ejecutar el script:

Instalamos pip con el comando:

```
sudo apt install python3-pip
```

Ahora ejecutamos el comando python3 de nuevo:

```
python3 install_requirements.py
```

Finalmente lanzamos la demo mediante:

```
python3 depthai_demo.py
```

Esto produce los errores:

Run the following commands to set USB rules:

```
$ echo 'SUBSYSTEM=="usb", ATTRS{idVendor}=="03e7", MODE=="0666"' | sudo tee /etc/udev/rules.d/99-depthai.rules  
$ sudo udevadm control --reload-rules && sudo udevadm trigger
```

Para solucionarlos ejecutamos los comandos:

```
echo 'SUBSYSTEM=="usb", ATTRS{idVendor}=="03e7", MODE=="0666"' | sudo tee /etc/udev/rules.d/99-depthai.rules  
sudo udevadm control --reload-rules && sudo udevadm trigger
```

Después de eso nos da el aviso:

After executing these commands, disconnect and reconnect USB cable to your OAK device

Desconectamos el usb de la cámara y lo volvemos a conectar.

Lanzamos la demo de la cámara mediante el comando:

```
python3 depthai_demo.py
```

y la cámara ya debería funcionar sin problemas.

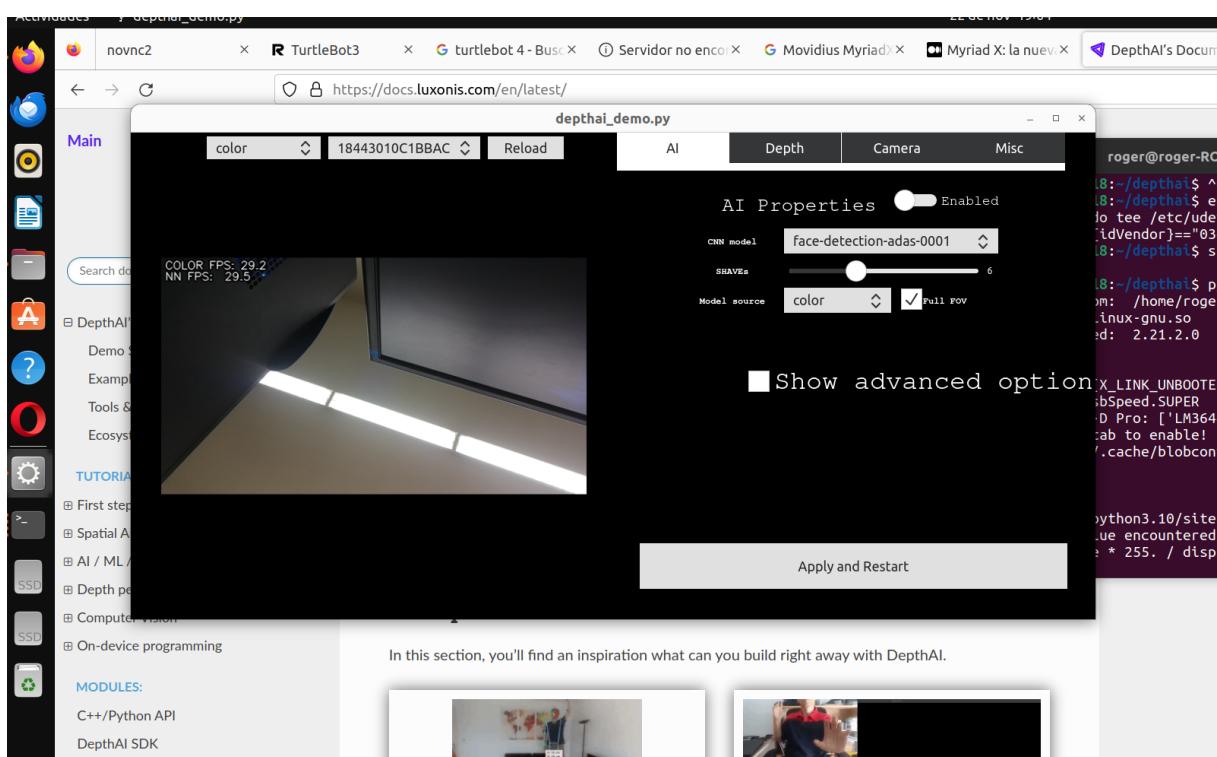


Figura 13: Lanzando la demo de la cámara Luxonis OAK D-PRO

Los comandos que usamos son de la página <https://docs.luxonis.com/en/latest/> para instalar los drivers de la cámara en el ordenador. En concreto los siguientes comandos:

```
git clone https://github.com/luxonis/depthai.git
```

```
cd depthai
```

```
python3 install_requirements.py
```

```
python3 depthai_demo.py
```

Para instalar los repositorios de la cámara para ROS2 humble usamos la página <https://github.com/luxonis/depthai-ros>. La instalación no es difícil. Estos son los resultados:

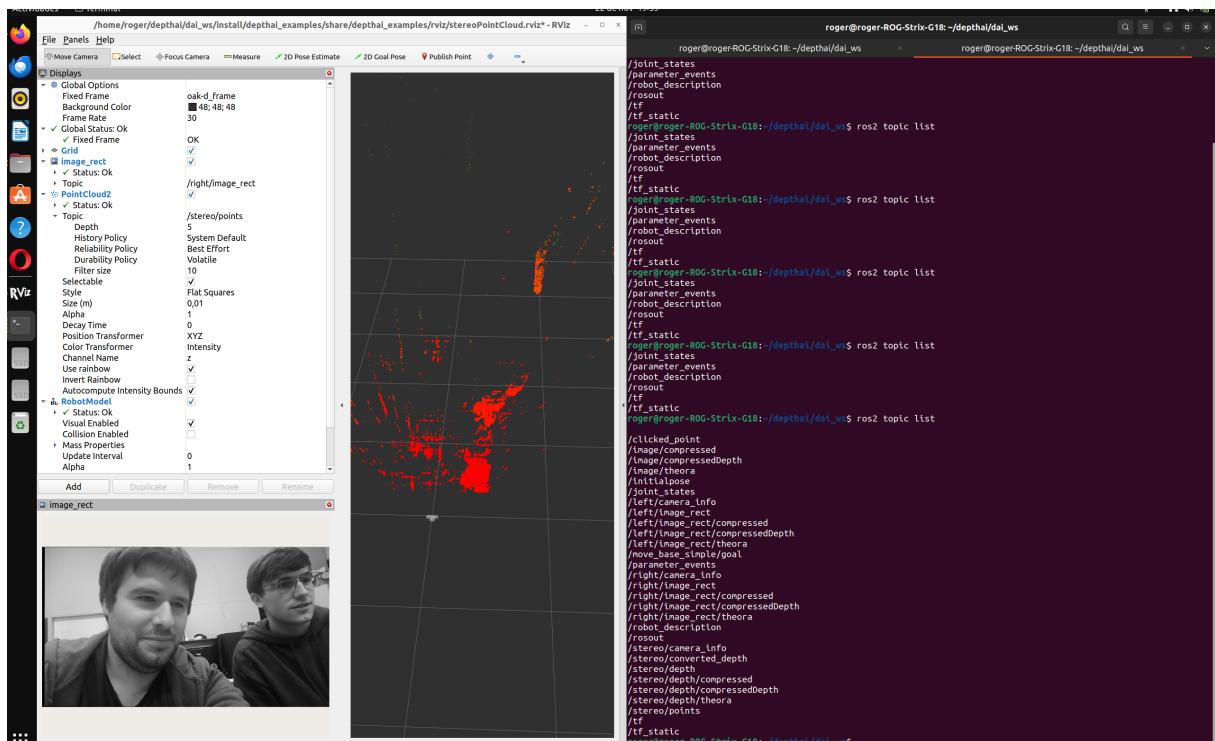


Figura 14: Usando cámara Luxonis en ROS 2 Humble

## 8. Instalar drivers RPLIDAR S2 en ROS2 humble

Un Lidar es un dispositivo que permite determinar la distancia desde un emisor láser a un objeto o superficie utilizando un haz láser pulsado. A continuación vamos a explicar como configurar el RPLIDAR S2 de Slamtec para hacerlo funcionar dentro del entorno ROS2 Humble en ubuntu 22.04. Para ello seguiremos el tutorial de la empresa Slamtec que se encuentra en el repositorio de GitHub [https://github.com/Slamtec/rplidar\\_ros/tree/ros2](https://github.com/Slamtec/rplidar_ros/tree/ros2).

1. Creamos nuestro workspace de ROS2 Humble (sólo hacemos este paso si no tenemos ya un entorno de trabajo creado). Para ello ejecutamos en la terminal la instrucción:

```
mkdir -p ~/ros2_ws/src
```

2. Nos movemos a la carpeta src de nuestro workspace de nombre ros2\_ws mediante la instrucción:

```
cd ~/ros2_ws/src
```

3. Clonamos el repositorio de GitHub de Slamtec mediante la instrucción:

```
git clone -b ros2 https://github.com/Slamtec/rplidar_ros.git
```

4. Nos movemos al workspace mediante:

```
cd ~/ros2_ws/
```

A continuación cargamos las variables de entorno de ROS2 Humble con la instrucción:

```
source /opt/ros/humble/setup.bash
```

Finalmente compilamos el código con la herramienta colcon mediante la instrucción:

```
colcon build --symlink-install
```

5. Dentro de la carpeta ros2\_ws cargamos las variables de entorno del workspace de ROS2 mediante la instrucción:

```
source ./install/setup.bash
```

Si no lo hemos hecho ya previamente podemos añadir una instrucción al .bashrc para que se cargen las variables de entorno de nuestro ROS2 workspace cada vez que abrimos una nueva terminal:

```
echo "source <your_own_ros2_ws>/install/setup.bash" >> ~/.bashrc
```

Cargamos el nuevo .bashrc en la terminal actual mediante el comando:

```
source ~/.bashrc
```

6. Para ejecutar el rplidar se necesitan permisos de lectura y escritura de nuestro dispositivo usb. Este parámetro puede modificarse mediante el comando:

```
sudo chmod 777 /dev/ttyUSB0
```

Una opción mejor que esta es cargar las reglas que vienen incluidas dentro de la carpeta sllidar\_ros2. Para ello bastará con ejecutar:

```
cd src/slldiar_ros2/
```

y

```
source scripts/create_udev_rules.sh
```

7. Finalmente lanzando el nodo view\_sllidar\_s2.launch.py del paquete sllidar\_ros:

```
ros2 launch sllidar_ros2 view_sllidar_s2.launch.py
```

se nos abre la aplicación Rviz en la que tendremos el lidar Rplidar S2 funcionando.

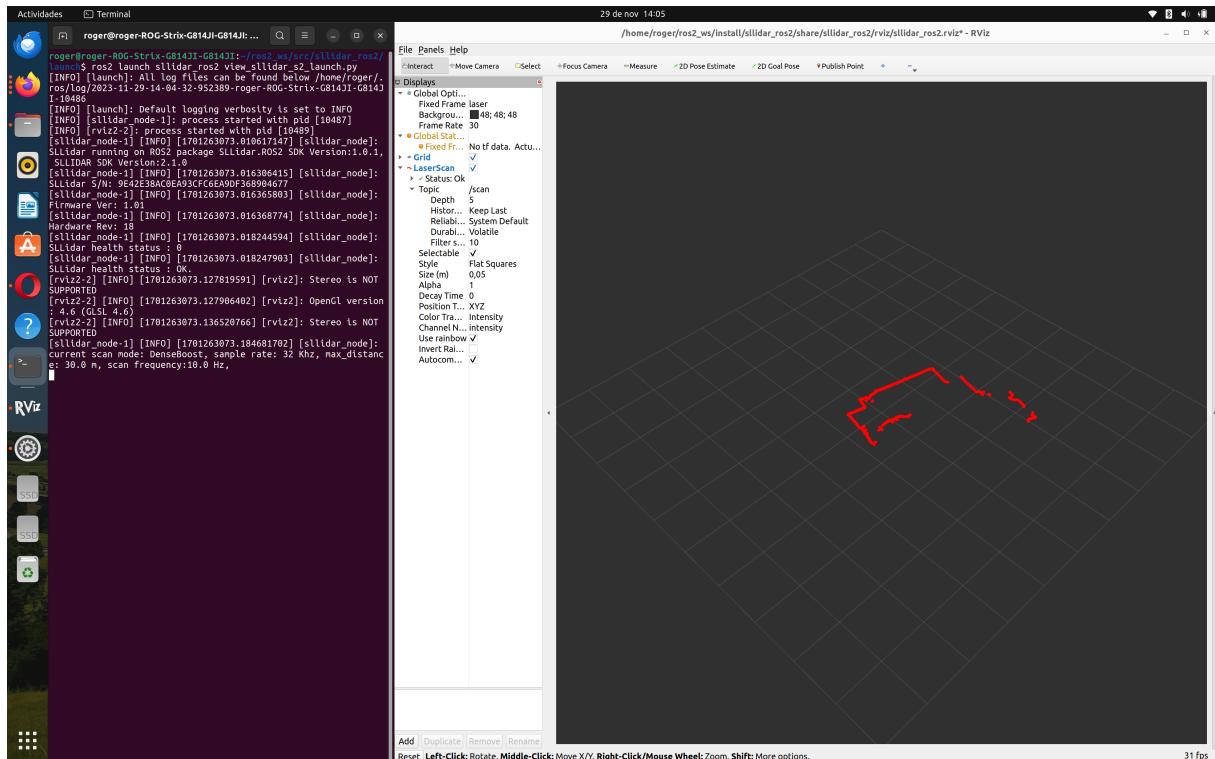


Figura 15: Usando Rplidar S2 en ROS2 Humble