Taller 2: Introducción a ROS (Robot Operating System)

1st Erick Ramón
Universidad de Cuenca, Facultad de Ingeniería
Ingeniería en Telecomunicaciones
Cuenca, Ecuador
erick.ramon@ucuenca.edu.ec

***Abstract*—El presente informe describe el desarrollo e implementación del Taller 2 de Redes de Sensores, orientado a la introducción a *ROS 2 (Robot Operating System)* y su funcionamiento dentro de entornos virtualizados mediante contenedores *Docker*. Se aborda el uso del estándar *DDS (Data Distribution Service)* sobre el protocolo *RTPS (Real-Time Publish-Subscribe)* para la comunicación entre nodos, analizando la interacción entre publicadores, suscriptores y el sistema de transporte de datos en tiempo real. Además, se documenta la creación de nodos en Python, la organización de paquetes dentro de un *workspace*, y la captura del tráfico generado por ROS 2 mediante *Wireshark*, destacando la relevancia de la arquitectura distribuida para el diseño de sistemas de comunicación eficientes, modulares y escalables.**

***Index Terms*—ROS 2, Docker, DDS, RTPS, Publicador–Suscriptor, Wireshark, Comunicación distribuida, Sistemas en tiempo real.**

I. Introducción

El presente taller tiene como objetivo comprender el funcionamiento interno del sistema operativo para robots *ROS 2 (Robot Operating System)* y su interacción con las herramientas de virtualización modernas, en particular *Docker*. A través de un entorno controlado y reproducible, el estudiante adquiere experiencia en la creación de contenedores, la configuración de espacios de trabajo (*workspaces*), y la implementación de nodos funcionales en Python bajo el paradigma de comunicación *publish–subscribe*. Esta práctica permite entender cómo los datos se generan, distribuyen y procesan dentro de un sistema robótico modular, facilitando la integración entre múltiples dispositivos y servicios de software en tiempo real.

Asimismo, el taller enfatiza el estudio del estándar *DDS (Data Distribution Service)* y su implementación sobre el protocolo *RTPS (Real-Time Publish-Subscribe Protocol)*, base fundamental de la arquitectura de comunicación en ROS 2. Mediante el análisis del tráfico de red con *Wireshark*, se identifican los distintos tipos de mensajes intercambiados —como los de descubrimiento, sincronización y transferencia de datos— y se interpreta su comportamiento en las diferentes capas del modelo TCP/IP. Este proceso fortalece la comprensión de las comunicaciones distribuidas y proporciona al estudiante las competencias necesarias para diseñar sistemas escalables, confiables y deterministas en aplicaciones de robótica, sensores e Internet de las Cosas (IoT).

II. Marco Teórico

El presente taller se enmarca en el estudio de la arquitectura de comunicación distribuida empleada por *ROS 2 (Robot Operating System)*, así como en el análisis de su funcionamiento mediante contenedores *Docker* y el intercambio de datos sobre el protocolo *DDS (Data Distribution Service)*. Este marco teórico presenta los fundamentos conceptuales necesarios para comprender la estructura modular del sistema, la lógica de los nodos y paquetes, y el papel de las herramientas de virtualización y mensajería utilizadas.

A. ROS 2 y el modelo de comunicación distribuida

ROS 2 es una plataforma de código abierto diseñada para facilitar el desarrollo de sistemas robóticos modulares y distribuidos. Su arquitectura se basa en el paradigma de comunicación *publicador–suscriptor* (*publish–subscribe*), donde los nodos intercambian mensajes a través de *tópicos*. Cada nodo puede desempeñar el rol de publicador, suscriptor o ambos, permitiendo una comunicación flexible y desacoplada entre los componentes de un sistema robótico.

El núcleo de ROS 2 se fundamenta en el estándar *DDS (Data Distribution Service)*, el cual proporciona mecanismos de intercambio de datos en tiempo real y define parámetros de calidad de servicio (*QoS, Quality of Service*) como fiabilidad, persistencia y latencia. DDS utiliza el protocolo *RTPS (Real-Time Publish-Subscribe Protocol)* como medio de transporte sobre UDP/IP, garantizando la interoperabilidad entre distintas implementaciones y proveedores.

B. Uso de contenedores Docker en ROS 2

La virtualización mediante *Docker* permite encapsular aplicaciones y sus dependencias en imágenes ligeras, asegurando portabilidad, aislamiento y reproducibilidad. En este taller, se utiliza una imagen base de ROS 2 Jazzy (versión 2024) para desplegar el entorno de desarrollo dentro de un contenedor.

La Tabla I resume la comparación entre las principales imágenes oficiales de ROS 2 Jazzy, destacando las diferencias entre las variantes disponibles.

TABLE I
COMPARACIÓN DE LAS IMÁGENES DOCKER OFICIALES DE ROS 2 JAZZY.

Imagen	Descripción y uso recomendado
`ros:jazzy-ros-base`	Incluye el núcleo del sistema ROS 2 y las herramientas básicas de compilación. Recomendado para usuarios que deseen construir proyectos desde cero.
`ros:jazzy-desktop`	Contiene el entorno completo con interfaces gráficas, herramientas de depuración y utilidades de visualización (como `rviz` y `rqt`). Adecuado para desarrollo interactivo.
`ros:jazzy-perception`	Añade bibliotecas específicas de visión computacional y procesamiento de sensores, ideal para proyectos de percepción avanzada.
`ros:jazzy-slim`	Imagen optimizada de tamaño reducido para entornos de despliegue en dispositivos embebidos o recursos limitados.

El uso de estas imágenes permite abstraer la complejidad de las dependencias del sistema y mantener una configuración consistente entre distintos equipos o entornos de desarrollo. En este taller se selecciona la imagen `osrf/ros:jazzy-desktop`, la cual ofrece un equilibrio entre funcionalidad y facilidad de uso.

C. Estructura del Workspace y creación de paquetes

El desarrollo en ROS 2 se organiza dentro de un *workspace* (espacio de trabajo), que actúa como contenedor de uno o varios paquetes de software. Cada paquete agrupa nodos, bibliotecas, configuraciones y archivos de instalación. El comando *ros2 pkg create* permite generar automáticamente esta estructura, facilitando la organización modular del código.

La Tabla II describe la estructura generada por el comando `ros2 pkg create --build-type ament_python sensor_program`, que constituye la base del proyecto desarrollado en el taller.

TABLE II
ESTRUCTURA GENERADA POR `ROS2 PKG CREATE`.

Archivo o carpeta	Descripción
`package.xml`	Contiene los metadatos del paquete, como el nombre, autor, versión y dependencias.
`setup.py`	Define el proceso de instalación y los puntos de entrada para ejecutar los nodos.
`setup.cfg`	Indica las rutas de instalación de los archivos y módulos del paquete.
`resource/`	Permite que el sistema de construcción *ament* identifique el paquete durante la instalación.
`sensor_program/`	Carpeta principal que contiene el código fuente en Python, incluyendo los nodos del sistema.
`__init__.py`	Archivo que convierte el directorio en un módulo importable dentro de Python.
`test/`	Contiene scripts de prueba y verificación de calidad del código.

Esta estructura modular promueve la reutilización y escalabilidad del código, permitiendo agregar nuevos nodos o funcionalidades sin comprometer la integridad del sistema. Cada nodo puede ser ejecutado de manera independiente mediante `ros2 run`, lo que favorece la depuración y prueba de componentes específicos.

D. Comunicación en DDS y RTPS

El funcionamiento de la comunicación entre nodos se basa en DDS, el cual implementa una capa intermedia que gestiona el descubrimiento automático de participantes y la entrega confiable de mensajes. DDS se construye sobre RTPS, protocolo encargado del intercambio real de mensajes a través de UDP. Este modelo proporciona una comunicación determinista, eficiente y flexible, ideal para entornos de tiempo real, como robótica, sensores distribuidos o sistemas de control autónomo.

RTPS utiliza un esquema de mensajería que incluye distintos tipos de paquetes, entre ellos:

- **Mensajes de descubrimiento (handshake):** permiten la identificación de los participantes del dominio DDS y el establecimiento de canales de comunicación.
- **Mensajes de sincronización (INFO_TS):** sincronizan los tiempos de transmisión de los datos.
- **Mensajes de datos (DATA(p)):** contienen las muestras de información transmitidas entre publicadores y suscriptores.

El uso de los puertos UDP 7400 y 7500 en este contexto permite separar el tráfico de control (*meta-traffic*) y el tráfico de usuario (*user data*), facilitando el análisis y la depuración del sistema a través de herramientas como *Wireshark*.

E. Importancia del taller

Este taller permite al estudiante comprender la integración de los distintos componentes de ROS 2 dentro de un entorno aislado mediante contenedores Docker, reforzando conceptos de comunicación distribuida y arquitectura modular. Además, la captura y análisis de tráfico RTPS ofrece una visión práctica del funcionamiento interno de DDS, evidenciando la forma en que los datos son gestionados, sincronizados y distribuidos en una red robótica.

III. Desarrollo

A. Parte II

1. Preparando el contenedor:

- Primero, debemos ejecutar el contenedor de ROS 2, al que llamaremos ros_ws, con acceso a una terminal interactiva bash. Note que usaremos la imagen osrf/ros:jazzy-desktop.
- En segundo lugar, es necesario instalar el paquete python3-colcon-commonextensions, el cual amplía las funcionalidades de la herramienta de compilación colcon de ROS 2. Estas extensiones permiten compilar distintos tipos de paquetes (como CMake o Python) dentro de un workspace. Su instalación es fundamental, ya que posibilita la definición de nodos y la ejecución del programa utilizando el lenguaje Python.
- Finalmente, activamos la ubicación de las fuentes ROS 2 para compilar el proyecto.

Ejecutamos el contenedor con el comando:

docker run -it –name ros2_ws osrf/ros:jazzy-desktop bash

Instalamos las herramientas de compilación:

apt update apt upgrade -y apt install -y python3-colcon-common-extensions nano

La activación de las fuentes de ROS no se realizó en esta instancia debido a que originalmente al ejecutar el contenedor las instancias estan actualizadas.

2. Creando el Workspace: Dentro del contenedor, creamos la estructura básica para el proyecto ros2_ws donde se alojaran todos los archivos necesarios.

Creamos el Workspace en el root del contenedor, y nos dirigimos a el con:

mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src

3. Creando estructuras y paquetes: Dentro del contenedor, creamos el workspace de los archivos del proyecto con ros2 pkg create. Las opciones del comando incluyen:

- ros2 pkg create: Comando de ROS 2 para crear un nuevo paquete.
- –build-type ament_python: Indica que el paquete se construirá usando ament con soporte para Python. Esto significa que será un paquete ejecutable de Python, en lugar de un paquete de CMake/C++.
- sensor_program: Es el nombre del paquete que se está creando. Este será el nombre que usaremos para importar módulos o para ejecutar el nodo con ros2 run.

Creamos el Workplace con el siguiente comando:

ros2 pkg create –build-type ament_python sensor_program –license MIT

Este comando crea la estructura del proyecto que se muestra en la imagen 1, donde los nodos resder, reader2, sensor, sensor2, son los nodos que se va a crear.

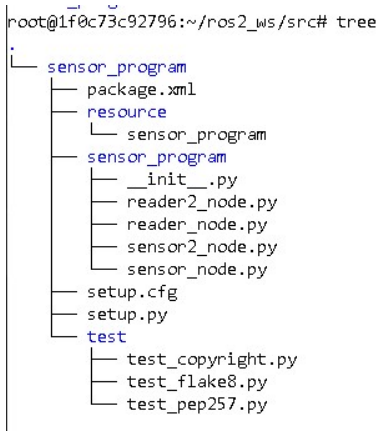

Fig. 1. Estructura del proyecto con comando tree

El archivo package.xml define los metadatos y dependencias del paquete, mientras que setup.py configura el proceso de instalación y el registro de los nodos ejecutables. El directorio resource Sirve para que ament (el sistema de construcción de ROS 2) reconozca este paquete al instalarlo y resolver recursos (no borrar). La carpeta sensor_program, que lleva el mismo nombre del paquete, contiene el código fuente en Python, incluyendo los nodos que implementan la funcionalidad del sistema y el archivo __init__.py permite que esta carpeta sea reconocida como un módulo de Python (paquete importable). El setup.cfg define dónde instalar los scripts y archivos.Por último la carpeta test contiene herramientas para verificar la calidad y el estilo del código.

4. Creando el Nodo-Sensor: Para desarrollar los nodos, utilizaremos el lenguaje Python. Vamos a simular un sensor de temperatura que emite valores aleatorios de temperatura cada segundo.

- Creamos el archivo Python sensor_node.py en el que definimos la operación del sensor.

nano ~/ros2_ws/src/sensor_program/sensor_program /sensor_node.py

- Creamos la lógica del Nodo-sensor: El algoritmo 1 refleja de manera simplificada la lógica del código Python para el nodo sensor, mostrando la secuencia de inicialización de un nodo ROS2, la publicación periódica de datos y cierre del nodo.

  En el algoritmo 1, (i) primero, se inicializa el sistema ROS 2 (rclpy.init(args=args)) y se crea un nodo llamado sensor_node junto con un publicador en el tópico sensor_data para enviar mensajes de tipo String. (ii) A continuación, mediante un bucle while true, el nodo genera cada segundo un valor aleatorio de temperatura entre 20 y 30 grados (rand_num[20,30]), construye un mensaje con dicho valor, lo publica en el tópico y registra la información en la consola. (iii) Finalmente, cuando se detiene el nodo, se destruye y se apaga ROS 2 correctamente, asegurando una terminación ordenada del programa.

Algorithm 1 Node-Sensor en ROS 2

```
1: InicializarROS2();
2: nodo = CrearNodo(sensor_node);
3: publisher = CrearPublicador(sensor_data, String,
   cola=10);
4: while true do
5:       esperar 1 segundo;
6:       temperatura = rand_num[20,30];
7:       mensaje = "Temperatura: " + temperatura +
   "grados C";
8:       publisher.publish(mensaje);
9:       RegistrarEnConsola("Publicando: " + men-
   saje);
10: MantenerNodoActivo(nodo);
11: DestruirNodo(nodo);
12: ApagarROS2(); =0
```

5. Creando el Nodo Lector (reader): Este nodo lee la información publicada del nodosensor y la muestra en pantalla. en futuros proyectos, se pueden realizar acciones con esta información, como alarmas o el encendido y apagado de dispositivos.

- Creamos el archivo Python reader_node.py en el que definimos la operación del sensor reader.

  nano ~/ros2_ws/src/sensor_program/sensor_program /reader_node.py

- Creamos la lógica del Nodo-reader: El algoritmo 2 refleja la lógica principal de la suscripción y manejo de mensajes en ROS 2.

  En el algoritmo 2, el reader_node está encargado de recibir y mostrar los datos publicados por otros nodos, como el nodo sensor. (i) Primero, se inicializa el sistema ROS 2 y se crea el nodo. (ii) A continuación, el nodo se suscribe al tópico sensor_data definiendo un callback (listener_callback) que se ejecuta cada vez que llega un mensaje; en este callback, el nodo registra en la consola el contenido del mensaje recibido almacenando hasta 10 msg en buffer. (iii) Finalmente, el nodo permanece activo mientras ROS 2 esté corriendo y, al finalizar, se destruye el nodo y se apaga el sistema de manera ordenada.

Algorithm 2 Nodo Lector en ROS 2

```
1: InicializarROS2()
2: nodo := CrearNodo(reader_node)
3: suscripcion := CrearSuscripcion(sensor_data,
   String, callback=listener_callback, cola=10)
4: Función listener_callback(mensaje):
5:       RegistrarEnConsola("Recibido: " + mensaje)
6: MantenerNodoActivo(nodo)
7: DestruirNodo(nodo)
8: ApagarROS2() =0
```

6. Agregar los nodos al setup.py: Modifique el archivo setup.py para configurar los nodos creados (nodo sensor y nodo reader). Dentro de dicho archivo ubique la funcion entry_points, para agregar los nuevos nodos, como se muestra en el siguiente código.

entry_points={ 'console_scripts': ['sensor_node = sensor_program.sensor_node:main', 'reader_node = sensor_program.reader_node:main',], },

7. Compilar el proyecto con Colcon: Vamos a utilizar la herramienta Colcon para compilar nuestro proyecto. Colcon es una herramienta de ROS2 que proporciona una forma estandarizada de construir y gestionar paquetes de ROS 2. Primero nos ubicamos en la raiz del workspace y construimos el proyecto.

cd ~/ros2_ws
colcon build

Cargamos el entorno en nuestro workspace.

source install /setup.bash

8. Ejecutar los nodos del proyecto: Primero, ejecutaremos el nodo que simula el sensor (sensor_node.py) y, después, el nodo reader (reader_node.py). Note que para ejecutar el segundo nodo, debemos abrir otra terminal de nuestro contenedor.

Ejecutamos el nodo sensor,

ros2 run sensor_program sensor_node

luego abrimos otra terminal y ejecutamos el comando exec con el comando bash, esto nos permite

abrir una terminal sin interumpir el proceso realizado en la terminal que esta corriendo el nodo sensor,

docker exec -it ros2_ws bash
cargamos el entorno de ROS2,

source /opt/ros/jazzy/setup.bash
cargamos el entorno de nuestro proyecto

source ~/ros2_ws/install/setup.bash
y finalmente ejecutamos el nodo reader.

ros2 run sensor_program reader_node

La figura 2 muestra el nodo sensor corriendo, mientras la figura 3 muestra el nodo lector corriendo. La figura 4 muestra los dos nodos corriendo y se puede observar como los valores son los mismos.

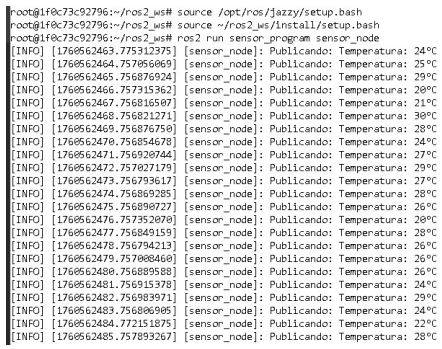

Fig. 2. Caption

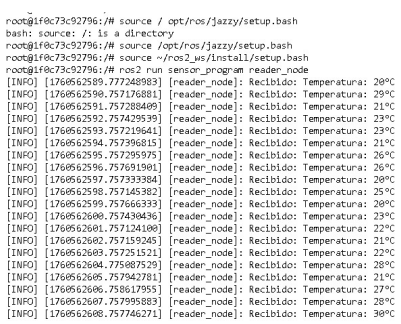

Fig. 3. Caption

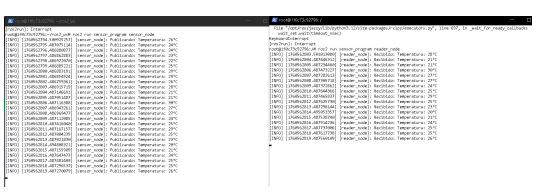

Fig. 4. Caption

9. Modifique y compile nuevamente el proyecto ros2_ws: Para este último punto, agregue un nuevo nodo reader 2 que se subscriba al tópico sensor_data. Ejecute ambos nodos subscriptores y verifique su funcionamiento. Para agregar un nuevo nodo y se suscriba al topico de la data del sensor simplemente se tiene que realizar una copia del código del nodo lector 1, pero para esta instancia se ha decidido aplicar dos nodos, uno con los datos del sensor de nivel, y otro nodo que se suscribe a este, entonces se procedió a modificar los algoritmos 3 y 4, obteniendo los siguientes algoritmos.

Algorithm 3 Node-Sensor2 en ROS 2

```
    InicializarROS2();
    nodo = CrearNodo(sensor2_node);
 3: publisher = CrearPublicador(sensor2_data, String,
    cola=10);
    while true do
        esperar 1 segundo;
 6:     nivel = rand_num[1,100];
        mensaje = "Nivel: " + nivel + "%";
        publisher.publish(mensaje);
 9:     RegistrarEnConsola("Publicando: " + mensaje);
    MantenerNodoActivo(nodo);
    DestruirNodo(nodo);
12: ApagarROS2(); =0
```

Algorithm 4 Nodo Lector2 en ROS 2

```
    InicializarROS2()
    nodo := CrearNodo(reader2_node)
 3: suscripcion := CrearSuscripcion(sensor2_data,
    String, callback=listener_callback, cola=10)
    Función listener_callback(mensaje):
        RegistrarEnConsola("Recibido: " + mensaje)
 6: MantenerNodoActivo(nodo)
    DestruirNodo(nodo)
    ApagarROS2() =0
```

Seguidamente se modifica el archivo setup.py de la siguiente manera:

entry_points= 'console_scripts': ['sensor_node = sensor_program.sensor_node:main', 'reader_node = sensor_program.reader_node:main', 'sensor2_node = sensor_program.sensor2_node:main', 'reader2_node = sensor_program.reader2_node:main',], ,

Luego cargamos el entorno en nuestro workspace.

source install /setup.bash

Ejecutamos el nodo sensor,

ros2 run sensor_program sensor2_node
luego abrimos otra terminal y ejecutamos el comando exec con el comando bash, esto nos permite abrir una terminal sin interumpir el proceso realizado en la terminal que esta corriendo el nodo sensor,

docker exec -it ros2_ws bash
cargamos el entorno de ROS2,

source /opt/ros/jazzy/setup.bash
cargamos el entorno de nuestro proyecto

source ˜/ros2_ws/install/setup.bash
y finalmente ejecutamos el nodo reader.

ros2 run sensor_program reader2_node

La figura 5 muestra estos dos nodos corriendo.

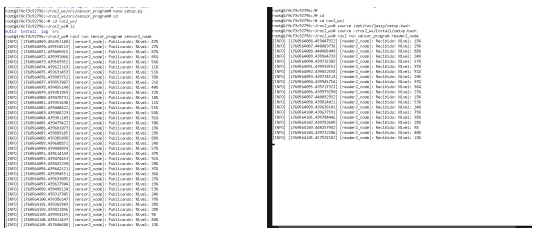

Fig. 5. Nodos lector y sensor 2 corriendo

B. Analizando trafico con WIRESHARK

1. Crear una red personalizada de Docker. Crearemos una red llamada ”proyecto”.

docker network create proyecto

2. Agregar los contenedores de nuestro taller a la red creada en el punto anterior.

docker network connect proyecto ros2_ws

3. Ejecutar el contenedor sniff dentro de la misma red de docker. Ejecutamos el contenedor con la imagen de nicolaka/netshoot que contiene el sniffer listo para capturar los paquetes. Usaremos un volumen compartido para guardar el .pcap para luego ser abierto con wireshark de Windows en modo grafico. Se debe ejecutar el nodo con permiso de administrador de red NET-ADMIN para que pueda tener acceso total al trafico de nuestra red, y la opción NET-RAW que permite escuchar todas las interfaces. Además, la interfaz eth0 no es la interfaz física de la máquina, sino la interfaz de red de todos los contenedores. Por último, el comando me permitira guardar el archivo en el volumen compartido bajo el nombre de trafico-ros2.pcap.

docker run –network proyecto -v ”C:ÙsersĖrick RamonĎesktopÙCueĢnoṙedes de sensoresṪaller 2:/pcap” –cap-add=NET_ADMIN –cap-add=NET_RAW nicolaka/netshoot tcpdump -i eth0 -w /pcap/trafico_ros2.pcap

En este escenario eth0 en el sniffer sí ve todo lo que pasa entre sensor-node y readernode, no necesitas tocar loopback ni interfaces extra ya que Docker enruta todo por el bridge virtual. Como resultado podremos visualizar en el contenedor del sniff el siguente mensaje de la figura 6.

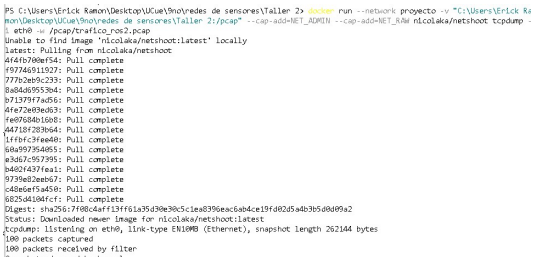

Fig. 6. Captura de tráfico con nikola

Pero para obtener este resultado, necesitabamos de saber si efectivamente eth0 es la interface de red que usa nuestro contenedor donde se encuentran los nodos sensor y reader, para ello ejecutamos *ifconfig* en el contenedor. La figura 7 muestra que la interface de red que utiliza nuestro contenedor es eth0, por lo que se prosiguió a ejecutar el comando con nikolaka.

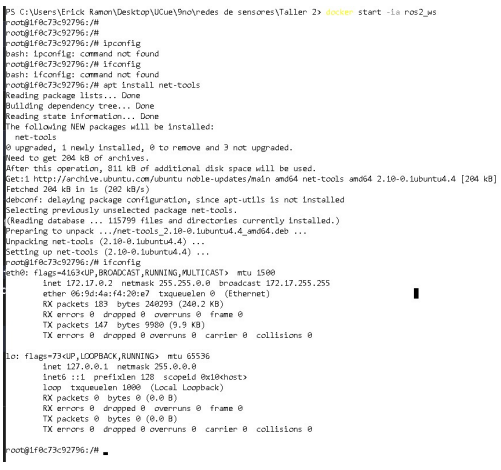

Fig. 7. Verificación de la interface de red

Ahora podremos ver en nuestro volumen compartido C: el archivo pcap que podremos abrir ya con Wireshark. Recordar que ROS usa el protocolo DDS sobre RTPS para comunicarse entre nodos, por lo que los puertos que deberemos monitorear son del 7400 al 7500 por lo tanto, dentro de wireshark podemos ejecutar:

udp.port¿=7400 udp.port¡=7500

obteniendo como resultado lo que muestra la figura 8.

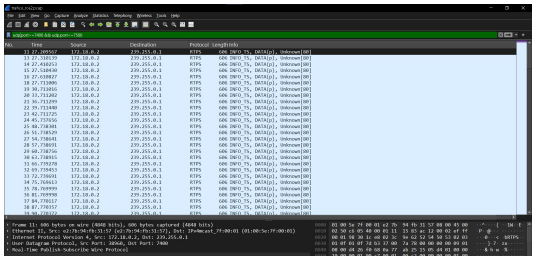

Fig. 8. Captura de tráfico con wireshark

El *Data Distribution Service* (DDS) es un estándar de comunicación orientado a datos utilizado en sistemas distribuidos en tiempo real. Su objetivo principal es permitir el intercambio eficiente y confiable de información entre múltiples nodos dentro de una red, siguiendo un modelo de comunicación *publish-subscribe*. En este modelo, los nodos productores de información se denominan *publicadores* (*publishers*) y los consumidores se denominan *suscriptores* (*subscribers*). DDS abstrae la complejidad de las conexiones punto a punto mediante el uso de un *middleware* que administra automáticamente la entrega, fiabilidad y calidad del servicio (*Quality of Service* - QoS).

DDS se implementa sobre el protocolo *RTPS* (*Real-Time Publish-Subscribe Protocol*), que define cómo se transmiten los mensajes a través de la red utilizando el protocolo UDP/IP. RTPS es responsable de la comunicación en tiempo real y asegura que los datos sean entregados de manera determinista, priorizando la latencia baja y la eficiencia. En esencia, DDS define las políticas y abstracciones de comunicación, mientras que RTPS implementa el transporte físico de los mensajes a nivel de red.

Durante la comunicación, DDS sobre RTPS utiliza un conjunto de puertos específicos para separar diferentes tipos de tráfico. Los puertos 7400 y 7500 son particularmente relevantes:

- El puerto **7400** se utiliza principalmente para el *metatraffic unicast*, encargado de la comunicación entre los participantes del dominio DDS. En este tráfico se intercambian mensajes de control, descubrimiento y mantenimiento de sesiones.
- El puerto **7500** suele utilizarse para el *user traffic*, es decir, los datos de usuario correspondientes a los tópicos publicados por los nodos.

El filtrado de estos puertos en la figura 8 permite aislar el tráfico relacionado exclusivamente con la comunicación DDS/RTPS, descartando otros flujos de red irrelevantes. De esta forma, se pueden analizar los mensajes internos del protocolo, tales como los de `INFO_TS` (timestamp) y `DATA(p)`, que corresponden a tramas de publicación y sincronización de datos.

Dentro del flujo RTPS, pueden observarse diferentes tipos de mensajes:

1) **Mensajes de Handshake:** se utilizan en las fases iniciales del descubrimiento para establecer las identidades de los participantes (*participants*) y asociar publicadores con suscriptores que pertenezcan al mismo dominio DDS.
2) **Mensajes INFO_TS:** proporcionan marcas de tiempo (*timestamps*) para los paquetes de datos, garantizando la correcta sincronización temporal de las muestras transmitidas.
3) **Mensajes DATA(p):** contienen la información del usuario publicada por los nodos productores, representando las muestras de datos transmitidas a los suscriptores.

En la captura de Wireshark se observan tramas RTPS con mensajes de tipo `INFO_TS` y `DATA(p)` transmitidos desde la dirección IP `172.18.0.2` hacia la dirección multicast `239.255.0.1`. Esto indica que el nodo correspondiente está publicando información hacia todos los suscriptores del dominio DDS configurado, utilizando comunicación UDP multicast.

Desde la perspectiva del modelo TCP/IP, los protocolos observados en la captura corresponden a las siguientes capas:

- **Capa de Aplicación:** DDS, implementado sobre RTPS, maneja la lógica de publicación, suscripción y control de calidad de servicio.
- **Capa de Transporte:** se utiliza el protocolo *UDP* (*User Datagram Protocol*), seleccionado por su baja latencia y simplicidad, adecuada para transmisión en tiempo real.
- **Capa de Red:** el protocolo *IP versión 4 (IPv4)* gestiona el direccionamiento y encaminamiento de los datagramas.
- **Capa de Enlace:** se emplea *Ethernet II*, responsable de encapsular los paquetes IP en tramas físicas que se transmiten sobre el medio.

C. Reto

1. Dockerfile: Desarrolle un archivo Dockerfile que automatice la mayor parte de los pasos [1-8]. Verifique la funcionalidad del script ros_entrypoint.sh que viene en el contenedor oficial. Ha de ejecutar este script antes de cualquier cambio. Para la correcta creación de este archivo docker se siguen los siguientes pasos. Tome en cuenta que estos pasos son los pasos 1-8 anteriores.

Paso 1: Definición de la Imagen Base:

- **FROM osrf/ros:jazzy-desktop**: Esta instrucción inicializa el proceso de construcción utilizando una imagen base oficial de ROS 2. La variante `jazzy-desktop` incluye un sistema Ubuntu con la distribución ROS 2 Jazzy completa y herramientas de desarrollo de escritorio, sirviendo como un fundamento robusto para el entorno.

Paso 2: Instalación de Dependencias del Sistema:

- **RUN apt-get update && apt-get install -y ...**: Este comando ejecuta la instalación de software adicional dentro de la imagen.
  - `apt-get update`: Actualiza la lista de paquetes disponibles.
  - `apt-get install -y`: Instala las dependencias especificadas:

    * `python3-colcon-common-extensions`: Herramientas necesarias para compilar paquetes de Python con `colcon`.
    * `net-tools` & `iputils-ping`: Utilidades de red para diagnóstico.
    * `tree`: Herramienta para visualizar la estructura de directorios.
    * `python3-matplotlib`: Librería de Python esencial para que el nodo graficador genere las imágenes.
  - `rm -rf /var/lib/apt/lists/*`: Limpia la caché de APT después de la instalación para reducir el tamaño final de la imagen.

Paso 3: Configuración del Espacio de Trabajo:

- **WORKDIR /ros2_ws**: Establece el directorio de trabajo por defecto a `/ros2_ws`. Las instrucciones subsecuentes se ejecutarán desde esta ruta dentro del contenedor.
- **COPY src/ /ros2_ws/src/**: Copia el código fuente del proyecto, ubicado en la carpeta local `src/`, al directorio `/ros2_ws/src/` dentro de la imagen.

Paso 4: Compilación del Proyecto ROS 2:

- **SHELL ["/bin/bash", "-c"]**: Modifica el shell por defecto para garantizar la correcta ejecución del comando `source`.
- **RUN source /opt/ros/jazzy/setup.bash && colcon build**: Este es el paso de compilación. Primero, activa el entorno de ROS 2 con `source` para que los comandos de ROS estén disponibles. Luego, ejecuta `colcon build`, la herramienta que compila los paquetes encontrados en el directorio `src/`[cite: 344, 348].

Paso 5: Definición del Comando de Ejecución por Defecto:

- **CMD ["/ros_entrypoint.sh", "bash"]**: Define el comando que se ejecutará al iniciar un contenedor a partir de la imagen. El script `/ros_entrypoint.sh` configura el entorno de ROS, y luego se inicia un terminal `bash` interactivo, dejando al usuario en un entorno listo para usar.

2. Shared folder: Cree una carpeta compartida entre su computadora y el contenedor. En esta carpeta, coloque los archivos sensor_node.py y reader_node.py. Al iniciar el contenedor, podrá copiar estos archivos a su ubicación definitiva; aunque también podrá editarlos directamente desde su IDE de programación favorito, lo que facilita su edición y desarrollo. Explique el proceso en detalle.

La funcionalidad se implementa mediante el flag `-v` (o `--volume`) dentro del comando `docker run`. A continuación, se desglosa la sintaxis utilizada:

```
docker run ... -v "${PWD}\shared_data:
/ros2_ws/data" ...
```

Componentes de la Instrucción de Volumen: La sintaxis del flag `-v` se estructura como `"ruta_en_anfitrión:ruta_en_contenedor"`.

- **Ruta en el Anfitrión (`"${PWD}\shared_data"`)**:
  - `${PWD}`: Es una variable de entorno utilizada en la terminal de PowerShell que se expande automáticamente a la ruta completa del directorio de trabajo actual. Esto permite que el comando sea portable y no dependa de una ruta absoluta codificada.
  - `\shared_data`: Es el nombre del directorio específico en la computadora anfitriona que servirá como punto de intercambio de archivos.
- **Separador (`:`)**: Este carácter delimita la ruta del anfitrión de la ruta del contenedor.
- **Ruta en el Contenedor (`/ros2_ws/data`)**:
  - Esta es la ruta dentro del sistema de archivos aislado del contenedor donde se montará el volumen. El código del nodo `plotter_node` fue diseñado para guardar la imagen del gráfico precisamente en este directorio.

Mecanismo de Funcionamiento: Al iniciar el contenedor con este flag, Docker establece una sincronización bidireccional entre los dos directorios:

1) **Escritura desde el Contenedor al Anfitrión**: Cuando el nodo `plotter_node` se ejecuta y llama a la función `plt.savefig('/ros2_ws/data /sensor_plot.png')`, el archivo se escribe en el sistema de archivos del contenedor. Gracias al volumen montado, Docker intercepta esta operación y replica el archivo `sensor_plot.png` de forma inmediata en la carpeta `shared_data` de la computadora anfitriona. Esto permite visualizar los resultados generados por el contenedor directamente en el sistema local.
2) **Modificación desde el Anfitrión al Contenedor**: Aunque no fue el caso de uso principal en este reto, si se modificara o añadiera un archivo en la carpeta `shared_data` del anfitrión, dicho cambio se reflejaría instantáneamente dentro del directorio `/ros2_ws/data` del contenedor. Esta capacidad es crucial para el desarrollo de software, ya que permite editar el código fuente en el anfitrión con un IDE como Visual Studio Code y que el contenedor utilice la versión actualizada sin necesidad de reconstruir la imagen.

3. Verificar que todos los nodos estén funcionando: Todos los nodos deben estar ejecutándose y mostrando la información de temperatura.

La figura 9 muestra que los nodos están ejecutándose perfectamente.

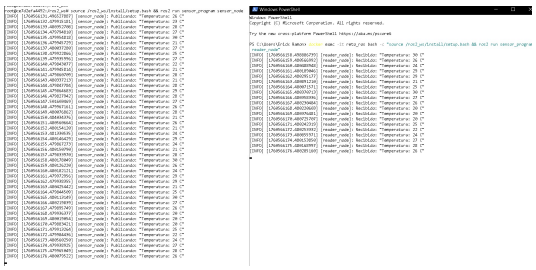

Fig. 9. Nodos lector y sensor del reto en ejecución

4. Cree un nodo adicional, llamado plotter_node, que se encargue de generar un gráfico de los datos recibidos por el nodo reader. Este nodo realiza el gráfico cada 5 segundos y guarda la imagen resultante en el volumen compartido, por ejemplo: plt.savefig('/ros2_ws/data/sensor_plot.png').

El script comienza importando las librerías necesarias para su funcionamiento:

- `rclpy` y `rclpy.node.Node`: Fundamentales para crear y gestionar nodos en ROS 2 utilizando Python.
- `std_msgs.msg.String`: Define el tipo de mensaje estándar (`String`) que el nodo espera recibir del tópico.
- `matplotlib.pyplot as plt`: La librería principal para la generación de gráficos en Python. Se utiliza para crear la figura, trazar los datos y guardar la imagen.
- `re`: La librería de expresiones regulares de Python, utilizada para extraer de forma robusta el valor numérico de la temperatura a partir del mensaje de texto recibido.
- `os`: Proporciona funciones para interactuar con el sistema operativo, en este caso, para asegurar que el directorio de destino del gráfico exista.

1) Inicialización del Nodo (Constructor `__init__`): Dentro del constructor de la clase se inicializan los componentes principales del nodo:

1) **Suscripción**: Se crea un suscriptor al tópico `'sensor_data'`. Este suscriptor espera mensajes de tipo `String` y, cada vez que recibe uno, invoca al método `listener_callback` para procesarlo.
2) **Almacenamiento de Datos**: Se inicializa una lista vacía llamada `self.temperatures`, que servirá para almacenar los valores numéricos de temperatura a medida que se reciben.
3) **Temporizador (Timer)**: Se crea un temporizador que se activa cada 5 segundos. Cada vez que el temporizador se dispara, ejecuta el método `plot_data`. Este es el mecanismo que controla la frecuencia de actualización del gráfico.
4) **Configuración del Directorio**: Se define la ruta del directorio compartido (`/ros2_ws/data`) y se utiliza `os.makedirs` para crearlo si no existe, asegurando que el programa no falle si la carpeta no ha sido creada previamente.

2) Recepción y Procesamiento de Datos (`listener_callback`): Este método es el corazón del procesamiento de datos. Se ejecuta cada vez que llega un nuevo mensaje al tópico `'sensor_data'`.

- **Extracción del Valor Numérico**: En lugar de asumir un formato fijo, se utiliza una expresión regular (`re.search(r'(\d+)', msg.data)`) para buscar y extraer el primer grupo de dígitos del mensaje. Esto hace que el código sea más robusto ante pequeños cambios en el texto del mensaje (ej. "Temp: 25C" o "Temperatura: 25 C").
- **Almacenamiento en la Lista**: Si se encuentra un número, se convierte a entero y se añade al final de la lista `self.temperatures`.
- **Gestión del Tamaño de la Lista**: Para evitar que el gráfico se sature con demasiados puntos de datos y que la lista consuma memoria indefinidamente, se implementa una lógica para mantener solo los últimos 50 puntos. Si la longitud de la lista supera 50, se elimina el elemento más antiguo (el primero de la lista) con `self.temperatures.pop(0)`.

3) Generación del Gráfico (`plot_data`): Este método se ejecuta cada 5 segundos gracias al temporizador.

1) **Verificación de Datos**: Primero, comprueba si la lista `self.temperatures` contiene datos. Si está vacía, imprime un aviso y no intenta generar un gráfico.
2) **Creación de la Figura**: Utiliza `matplotlib` para crear una nueva figura y un gráfico de líneas (`plt.plot`) con los datos almacenados.
3) **Estilización del Gráfico**: Se añaden elementos para que el gráfico sea más informativo y legible, como un título, etiquetas para los ejes X e Y, una cuadrícula y un rango fijo para el eje Y (`plt.ylim(18, 32)`), lo que estabiliza la visualización y facilita la comparación de valores.
4) **Guardado del Archivo**: El paso final es guardar la figura generada en un archivo de imagen. Se utiliza `plt.savefig()` para guardar el gráfico en la ruta del directorio compartido con el nombre `sensor_plot.png`. El archivo se sobrescribe en cada ejecución, mostrando siempre la versión más reciente de los datos.
5) **Liberación de Memoria**: Se llama a `plt.close()` para cerrar la figura después de guardarla. Este es un paso importante para evitar que se consuma memoria RAM con figuras que ya no se necesitan.

4) Función Principal (`main`): Esta es la sección estándar que permite ejecutar el script como un nodo de ROS 2.

- Inicializa la librería de cliente de ROS (`rclpy.init`).
- Crea una instancia de la clase `PlotterNode`.
- Inicia el bucle de eventos de ROS con `rclpy.spin(node)`, que mantiene el nodo activo para que pueda recibir mensajes y ejecutar los callbacks.
- Se asegura de que, al detener el programa (ej. con Ctrl+C), el nodo se destruya correctamente (`node.destroy_node()`) y el sistema se apague de forma ordenada (`rclpy.shutdown()`).

El código se resume en el algoritmo 5 a continuación.

Algorithm 5 Nodo Graficador en ROS 2

```
InicializarROS2()
nodo := CrearNodo(plotter_node)
suscripcion := CrearSuscripcion(sensor_data, String, callback=listener_callback, cola=10)
Inicializar lista temperatures vacía
Crear temporizador de 5 segundos para ejecutar plot_data()
Definir directorio de guardado: /ros2_ws/data
Si el directorio no existe, crearlo
RegistrarEnConsola("Nodo graficador iniciado")
Función listener_callback(mensaje):
    extraer valor numérico de mensaje
    si se encuentra coincidencia:
        temperatura := valor entero extraído
        añadir temperatura a la lista temperatures
        si longitud(temperatures) ¿ 50 entonces eliminar el primer elemento
Función plot_data():
    si la lista temperatures está vacía:
        RegistrarAdvertencia("No hay datos para graficar aún")
        retornar
    Crear nueva figura (10x5)
    Graficar lista temperatures con línea azul y marcadores
    Título := "Historial de Temperatura en Tiempo Real"
    Etiquetas de ejes: X = "Muestras Recientes", Y = "Temperatura (°C)"
    Activar cuadrícula y limitar eje Y entre 18 y 32
    Guardar gráfico en /ros2_ws/data/sensor_plot.png
    Cerrar figura
    RegistrarEnConsola("Gráfico actualizado y guardado en /ros2_ws/data")
MantenerNodoActivo(nodo)
DestruirNodo(nodo)
ApagarROS2() =0
```

La figura 10 muestra la imagen resultante del nodo plotter.

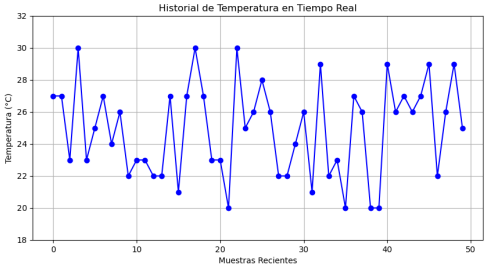

Fig. 10. Figura resultante de la captura de datos de temperatura del noto plotter_node

IV. Conclusiones

References