

Taller 2: Introducción a ROS (Robot Operating System)

Sebastián Stalin Guazhima Fernández
Facultad de Ingeniería en Telecomunicaciones
Universidad de Cuenca
Cuenca, Ecuador
sebastian.guazhima@ucuenca.edu.ec

Abstract—En este taller se desarrolló un entorno de ROS 2 dentro de Docker, para simular una red de sensores mediante varios nodos interconectados. Se implementaron tres nodos principales: uno sensor que genera valores de temperatura, un lector que recibe los datos y un nodo adicional que grafica las lecturas de manera continua.

Durante el proceso, se aplicaron comandos de ROS 2 para compilar y ejecutar el sistema, además de crear un archivo Dockerfile para automatizar la construcción del entorno. También se configuraron carpetas compartidas entre el host y el contenedor, y se realizó un análisis de tráfico con Wireshark para confirmar la comunicación basada en DDS/RTPS.

Los resultados mostraron la correcta interacción entre los nodos y el funcionamiento del sistema en tiempo real. En conjunto, este taller permitió comprender la estructura interna de ROS 2 y su importancia dentro de la robótica moderna, además de fortalecer el manejo de entornos virtualizados.

Index Terms—ROS 2, Docker, red de sensores, nodos, RTPS, Dockerfile, archivos compartidos

I. INTRODUCCIÓN

En este taller se trabajó con ROS 2 (Robot Operating System), una herramienta que permite la comunicación entre distintos nodos o programas dentro de un mismo sistema, simulando el funcionamiento de una red de sensores.

El objetivo principal fue desarrollar un entorno funcional en Docker donde se pueda crear, compilar y ejecutar varios nodos que interactúan entre sí: un nodo sensor que genera datos, un nodo lector que los recibe, y un nodo adicional que grafica los valores de temperatura obtenidos.

Para realizarlo, se construyó una imagen personalizada mediante un Dockerfile, se configuraron carpetas compartidas entre el host y el contenedor para la edición del código y la generación de resultados, y se implementó un paquete de ROS 2 con sus respectivos archivos y dependencias.

Además, se realizó una captura del tráfico de red con Wireshark para analizar la comunicación entre los nodos dentro de la red virtual de Docker, verificando la transmisión de mensajes en el protocolo DDS/RTPS.

Finalmente, como parte del proceso de documentación y respaldo, la imagen del contenedor fue subida al Docker Hub, y el código fuente del proyecto se publicó en GitHub, garantizando su disponibilidad y replicación en otros entornos de trabajo.

II. MARCO TEÓRICO

A. Concepto y Origen de ROS

El sistema operativo para robots, conocido como *Robot Operating System* (ROS), es un marco de trabajo de software de código abierto diseñado para facilitar el desarrollo de aplicaciones robóticas. Aunque su nombre puede llevar a confusión, ROS no es un sistema operativo en sí, sino una capa de comunicación y gestión de procesos que opera sobre plataformas como Linux, Windows o incluso dentro de contenedores Docker [1].

Su origen se remonta a la Universidad de Stanford (2006–2007) con Keenan WYROBEK y Eric BERGER, quienes buscaban evitar la constante reprogramación de controladores y protocolos de hardware en proyectos de robótica. Más tarde, con el apoyo de la empresa Willow Garage y posteriormente de la *Open Source Robotics Foundation (OSRF)*, ROS se consolidó como un estándar académico e industrial, permitiendo compartir código y herramientas en una comunidad global.

B. Importancia y Ventajas

ROS permite separar el desarrollo lógico del robot (software) del hardware físico, de modo que los algoritmos de percepción o control puedan funcionar en distintos robots sin modificaciones profundas. Esta filosofía ha acelerado la innovación en el ámbito de la robótica y ha permitido la integración de simuladores, entornos gráficos y herramientas de análisis.

Algunas de sus principales ventajas son:

- **Modularidad:** los programas se dividen en nodos, lo que facilita el mantenimiento y la reutilización del código.
- **Intercambio de datos estandarizado:** gracias a los mensajes (*msgs*) y tópicos, la comunicación entre módulos es simple y universal.
- **Amplia comunidad y soporte:** existen miles de paquetes de código abierto disponibles para tareas como SLAM, visión o navegación.
- **Compatibilidad con simuladores:** herramientas como Gazebo o Rviz permiten probar algoritmos sin hardware real.

C. Arquitectura y Comunicación

ROS utiliza una arquitectura distribuida basada en nodos. Cada nodo es un proceso independiente que cumple una

función específica, como leer un sensor o enviar comandos a un motor. Estos nodos se comunican entre sí mediante tres mecanismos principales:

- **Topics (tópicos):** permiten enviar mensajes de manera continua usando un esquema publicador–suscriptor.
- **Services (servicios):** realizan peticiones puntuales tipo cliente–servidor.
- **Actions (acciones):** se emplean para tareas que requieren retroalimentación durante su ejecución.

En ROS 2, esta comunicación está gestionada por el middleware **DDS (Data Distribution Service)**, que reemplaza la comunicación centralizada del ROS 1 por un sistema distribuido. DDS maneja la entrega de datos, la calidad del servicio (QoS) y la tolerancia a fallos entre nodos [2].

Esto hace que ROS 2 sea más seguro, confiable y adecuado para aplicaciones industriales, además de soportar sistemas en tiempo real y multiplataforma.

D. ROS 2: Evolución y Herramientas

La llegada de ROS 2 marcó una evolución significativa al incorporar características más robustas, soporte de tiempo real y compatibilidad con sistemas embebidos [3]. Además, integra seguridad nativa y opciones de comunicación determinista.

ROS 2 utiliza herramientas de compilación y ejecución más modernas. Algunas de las más importantes en su uso práctico son:

- `ros2 run <paquete> <nodo>` — ejecuta un nodo específico.
- `ros2 node list` — muestra los nodos activos en el sistema.
- `ros2 topic list / echo / pub` — permite listar, visualizar o publicar mensajes en los tópicos.
- `colcon build` — compila el proyecto completo o un paquete.
- `source install/setup.bash` — activa el entorno compilado.

Estos comandos son esenciales para desarrollar y verificar el funcionamiento de un sistema en ROS 2, especialmente cuando se trabaja en entornos con varios nodos y contenedores Docker, como en la práctica del taller.

E. Importancia del Middleware DDS

El middleware DDS (Data Distribution Service) constituye la base de comunicación de ROS 2. Actúa como un sistema de publicación–suscripción de datos en tiempo real, que permite que los nodos intercambien información sin conocerse entre sí. Su uso garantiza:

- Alta eficiencia en transmisión de datos distribuidos.
- Escalabilidad para múltiples nodos o dispositivos.
- Configuración de políticas QoS (fiabilidad, latencia, entrega garantizada).

Gracias a esta infraestructura, ROS 2 logra un desempeño más cercano al requerido por entornos industriales, donde la sincronización y la confiabilidad son fundamentales.

La utilización de comandos como `ros2 run`, `ros2 topic` o `colcon build` demuestra cómo este entorno permite integrar de forma sencilla sensores, actuadores y sistemas

de comunicación, haciendo posible el diseño de robots más inteligentes, seguros y adaptables.

III. DESARROLLO Y RESULTADOS

Siguiente sección se detallarán los pasos realizados y los resultados obtenidos siguiendo cada paso indicado por la guía del taller 2 de Redes de Sensores.

A. Preparación del contenedor

Para iniciar la práctica se utilizó la imagen oficial `osrf/ros:jazzy-desktop`, que ya contiene las librerías base de ROS 2 y las herramientas necesarias para trabajar con nodos en Python. El objetivo es preparar un entorno de trabajo dentro de un contenedor Docker, donde posteriormente se crearían los paquetes y nodos del sistema.

Primero, se creó el contenedor llamado **contenedor_ros2_t2_1**, con acceso a una terminal interactiva tipo bash. El comando ejecutado fue el siguiente:

```
1 docker run -it --name contenedor_ros2_t2_1
   ↪ osrf/ros:jazzy-desktop bash
```

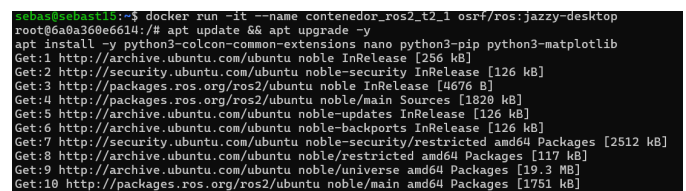
Una vez dentro del contenedor, se procedió a actualizar los paquetes del sistema e instalar las herramientas necesarias para compilar y ejecutar programas en Python dentro de ROS 2. Estas herramientas incluyen `python3-colcon-common-extensions`, que amplía las funciones de compilación de ROS 2, además de `nano`, `python3-pip` y `python3-matplotlib` para la edición y graficación de datos.

```
1 apt update && apt upgrade -y
2 apt install -y python3-colcon-common-extensions
   ↪ nano python3-pip python3-matplotlib
```

Después de la instalación, es necesario activar el entorno de ROS 2 para que los comandos del sistema sean reconocidos correctamente. Esto se hace cargando el archivo de entorno `setup.bash`, como se muestra a continuación:

```
1 source /opt/ros/jazzy/setup.bash
```

En la Figura 1, se observa la creación del contenedor y dentro de él los comandos para actualización y descarga de programas.



```
sebastian@sebasti:~$ docker run -it --name contenedor_ros2_t2_1 osrf/ros:jazzy-desktop
root@6a0a360e6614:/# apt update && apt upgrade -y
apt install -y python3-colcon-common-extensions nano python3-pip python3-matplotlib
Get:1 http://archive.ubuntu.com/ubuntu noble InRelease [256 kB]
Get:2 http://security.ubuntu.com/ubuntu noble-security InRelease [126 kB]
Get:3 http://packages.ros.org/ros2/ubuntu noble InRelease [4676 B]
Get:4 http://packages.ros.org/ros2/ubuntu noble/main Sources [1228 kB]
Get:5 http://archive.ubuntu.com/ubuntu noble-updates InRelease [126 kB]
Get:6 http://archive.ubuntu.com/ubuntu noble-backports InRelease [126 kB]
Get:7 http://security.ubuntu.com/ubuntu noble-security/restricted amd64 Packages [2512 kB]
Get:8 http://archive.ubuntu.com/ubuntu noble/restricted amd64 Packages [117 kB]
Get:9 http://archive.ubuntu.com/ubuntu noble/universe amd64 Packages [19.3 MB]
Get:10 http://packages.ros.org/ros2/ubuntu noble/main amd64 Packages [1751 kB]
```

Fig. 1: Creación del contenedor y descarga de paquetes

B. Creación del Workspace

Con el contenedor **contenedor_ros2_t2_1** ya operativo, se procedió a crear el espacio de trabajo o *workspace* donde se desarrollarán los nodos y paquetes de ROS 2.

El workspace fue creado dentro del contenedor en la ruta del usuario raíz. La carpeta `src` contendrá los paquetes del proyecto. El proceso se realizó con los siguientes comandos:

```

1 # Crear la carpeta principal del workspace
2 mkdir -p ~/ros2_ws/src
3
4 # Acceder a la carpeta src
5 cd ~/ros2_ws/src

```

La estructura generada hasta este punto quedó de la siguiente forma:

```

1 ros2_ws/
2   src/

```

C. Paso 3: Creación de estructuras y paquete del proyecto

Dentro del workspace recién creado, se utilizó el comando `ros2 pkg create` para generar la estructura de un paquete ROS2 de tipo Python. El paquete se denominó **sensor_program** y es el encargado de contener todos los nodos y configuraciones del taller.

```

1 # Crear el paquete principal del proyecto
2 ros2 pkg create --build-type ament_python
  ↳ sensor_program --license MIT

```

Realizado esto, se observa en la Figura 2 el directorio completo y dentro de `ros2_ws` y `src` el árbol completo del paquete del proyecto.

```

root@6a0a360e6614:~/ros2_ws/src# tree
.
├── sensor_program
│   ├── LICENSE
│   ├── package.xml
│   ├── resource
│   │   └── sensor_program
│   ├── sensor_program
│   │   ├── __init__.py
│   │   ├── setup.cfg
│   │   ├── setup.py
│   │   └── test
│   │       ├── test_copyright.py
│   │       ├── test_flake8.py
│   │       └── test_pep257.py

```

Fig. 2: Árbol de contenido de la carpeta para el funcionamiento del proyecto

D. Creación del nodo sensor

En este paso se implementó el primer nodo del sistema, llamado **sensor_node**, que se encarga de simular un sensor de temperatura. Este nodo genera valores aleatorios en un rango determinado y los publica periódicamente en el tópico `sensor_data`, de manera que otros nodos del sistema puedan suscribirse a estos datos.

Para crear el archivo del nodo dentro del paquete `sensor_program`, se ejecutó el siguiente comando desde el contenedor:

```

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

```

El contenido del nodo fue desarrollado en Python utilizando la librería `rclpy`, que permite crear y administrar nodos en ROS 2. La estructura del código se resume en el siguiente pseudocódigo:

Algorithm 1 Funcionamiento del Nodo Sensor

- 1: Inicializar ROS 2
- 2: Crear nodo `sensor_node`
- 3: Crear publicador en el tópico `sensor_data`
- 4: **while** ROS 2 esté activo **do**
- 5: Generar un valor de temperatura aleatorio (20–30 °C)
- 6: Crear mensaje con el valor generado
- 7: Publicar mensaje en el tópico
- 8: Esperar 1 segundo
- 9: **end while**
- 10: Finalizar nodo y cerrar ROS 2

Este algoritmo describe el ciclo de funcionamiento: el nodo se mantiene activo publicando datos en intervalos regulares, sirviendo como fuente principal de información dentro de la red de sensores. De esta manera, el sistema reproduce el flujo continuo de mediciones propio de un sensor físico.

E. Creación del nodo lector

Después de tener el nodo que publica la información del sensor, se desarrolló el segundo nodo del sistema, denominado **reader_node**. Este nodo cumple la función de receptor o suscriptor, y su tarea principal es escuchar el tópico `sensor_data` donde el nodo sensor envía sus mensajes, para luego mostrar en pantalla los valores de temperatura recibidos.

El nodo fue implementado también en Python, utilizando el mismo paquete `sensor_program`. Para su creación se ejecutó dentro del contenedor el siguiente comando:

```

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

```

El funcionamiento general del nodo se describe en el siguiente algoritmo:

Algorithm 2 Funcionamiento del Nodo Lector (reader_node)

- 1: Inicializar el entorno de ROS 2.
- 2: Crear el nodo llamado `reader_node`.
- 3: Definir una suscripción al tópico `sensor_data`.
- 4: Registrar una función de escucha (`listener_callback`) que se ejecuta cada vez que llega un mensaje.
- 5: **while** ROS 2 esté activo **do**
- 6: Esperar mensajes publicados en el tópico.
- 7: Cuando llega un mensaje, mostrar en consola: "Recibido: <dato del sensor>".
- 8: **end while**
- 9: Finalizar el nodo y cerrar ROS 2 correctamente.

El algoritmo indica que el nodo lector, su tarea es escuchar continuamente el flujo de datos publicado por el nodo sensor

y mostrarlos por consola, verificando que la comunicación *publisher-subscriber* funcione correctamente. Este nodo no procesa ni modifica los datos; simplemente confirma la recepción y comunicación dentro del entorno ROS 2.

F. Integración de los nodos en el archivo `setup.py`

En este punto fue necesario registrar los nodos creados dentro del archivo `setup.py` del paquete `sensor_program`. Este archivo define los parámetros de instalación del paquete y los scripts que pueden ejecutarse como nodos en ROS 2.

Para editar el archivo se utilizó el siguiente comando:

```
1 nano ~/ros2_ws/src/sensor_program/setup.py
```

Dentro del archivo, se verificó la existencia del bloque de configuración y se agregaron los *entry points* para los nodos `sensor_node` y `reader_node`.

El bloque quedó de la siguiente forma:

```
1 entry_points={
2     'console_scripts': [
3         'sensor_node =
4             ↪ sensor_program.sensor_node:main',
5         'reader_node =
6             ↪ sensor_program.reader_node:main',
7     ],
8 }
```

Con este bloque, cada nodo se registra como un script ejecutable dentro del paquete.

G. Compilación del proyecto con Colcon

Una vez configurado el paquete y registrados los nodos en el archivo `setup.py`, el siguiente paso fue compilar el proyecto completo para generar los ejecutables de ROS 2. Para ello se utilizó la herramienta **Colcon**, que permite construir y gestionar paquetes dentro de un workspace.

Desde la raíz del workspace `ros2_ws`, se ejecutaron los siguientes comandos:

```
1 cd ~/ros2_ws
2 # Compilar el proyecto completo
3 colcon build
```

Listing 1: Compilación del proyecto con Colcon

Una vez finalizada la compilación (Figura 3), se activó el entorno de trabajo local del proyecto para cargar las variables de entorno generadas:

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

```
root@6a0a360e6614:~/ros2_ws# colcon build
Starting >>> sensor_program
Finished <<< sensor_program [3.02s]

Summary: 1 package finished [3.49s]
root@6a0a360e6614:~/ros2_ws#
root@6a0a360e6614:~/ros2_ws#
root@6a0a360e6614:~/ros2_ws#
root@6a0a360e6614:~/ros2_ws# source install/setup.bash
```

Fig. 3: Compilación correcta con colcon

H. Ejecución de los nodos del proyecto

Después de compilar el paquete, se procedió a ejecutar los nodos para verificar la comunicación entre ellos. Primero se ejecutó el nodo del sensor, que publica los valores de temperatura, y luego el nodo lector, que se suscribe al mismo tópico y muestra los datos recibidos.

```
1 ros2 run sensor_program sensor_node
```

Listing 2: Ejecución del nodo sensor

En una segunda terminal, se abrió otra sesión dentro del mismo contenedor usando el siguiente comando:

```
1 docker exec -it contenedor_ros2_t2_1 bash
```

Listing 3: Abrir una nueva terminal dentro del contenedor

Dentro de esta nueva terminal se cargó nuevamente el entorno de ROS 2 y el entorno local del workspace:

```
1 source /opt/ros/jazzy/setup.bash
2 source ~/ros2_ws/install/setup.bash
```

Finalmente, se ejecutó el nodo lector:

```
1 ros2 run sensor_program reader_node
```

Listing 4: Ejecución del nodo lector

En las dos terminales se observaron mediante la Figura 4 los mensajes publicados y recibidos, confirmando que la comunicación *publisher-subscriber* entre ambos nodos se estableció correctamente.

```
1 [INFO] [sensor_node]: Publicando: Temperatura:
2     ↪ 26 C
3 [INFO] [reader_node]: Recibido: Temperatura: 26
4     ↪ C
```

```
[INFO] [1768664195.962375736] [sensor_node]: Publicando: Temperatura: 27 °C
[INFO] [1768664195.977488297] [sensor_node]: Publicando: Temperatura: 28 °C
[INFO] [1768664197.981557163] [sensor_node]: Publicando: Temperatura: 28 °C
[INFO] [1768664199.986134613] [sensor_node]: Publicando: Temperatura: 28 °C
[INFO] [1768664200.987722292] [sensor_node]: Publicando: Temperatura: 28 °C
[INFO] [1768664201.99225966] [sensor_node]: Publicando: Temperatura: 28 °C
[INFO] [1768664202.997122894] [sensor_node]: Publicando: Temperatura: 28 °C
[INFO] [1768664204.993542357] [sensor_node]: Publicando: Temperatura: 30 °C
[INFO] [1768664195.962375736] [reader_node]: Recibido: Temperatura: 27 °C
[INFO] [1768664196.967220283] [reader_node]: Recibido: Temperatura: 28 °C
[INFO] [1768664197.981557163] [reader_node]: Recibido: Temperatura: 28 °C
[INFO] [1768664199.986134613] [reader_node]: Recibido: Temperatura: 28 °C
[INFO] [1768664200.987722292] [reader_node]: Recibido: Temperatura: 28 °C
[INFO] [1768664201.99225966] [reader_node]: Recibido: Temperatura: 28 °C
[INFO] [1768664202.997122894] [reader_node]: Recibido: Temperatura: 28 °C
[INFO] [1768664204.993542357] [reader_node]: Recibido: Temperatura: 30 °C
```

Fig. 4: Funcionamiento de los nodos sensor y lector (*publisher* y *subscriber*) en el entorno ROS2.

Con este paso se verificó que el flujo de datos en ROS 2 funciona adecuadamente, y se completó la comunicación entre los nodos básicos del sistema.

IV. RETO

A. Punto 1: Creación de un Dockerfile

Antes de automatizar el entorno de ROS 2, fue necesario crear un archivo `Dockerfile` que contenga las instrucciones necesarias para construir la imagen del proyecto. Este archivo define la instalación de dependencias, la creación del workspace y la preparación del entorno donde se ejecutarán los nodos del sistema.

Ubicación del archivo Dockerfile: Desde el sistema host (Windows), se creó una carpeta para el taller en la ruta:

```
1 C:\Users\sebas\Documents\9no\REDES_SENSORES\
2 taller2\
```

Dentro de esta carpeta se guardó el archivo Dockerfile. Al trabajar desde WSL, se puede acceder a este directorio usando la ruta equivalente en Linux:

```
1 /mnt/c/Users/sebas/Documents/9no/REDES_SENSORES
2 /taller2/
```

Para editar el archivo directamente desde el entorno Ubuntu en WSL se ejecutó:

```
1 cd /mnt/c/Users/sebas/Documents/9no
2 /REDES_SENSORES/taller2/
3
4 nano Dockerfile
```

subsubsection*Contenido y explicación del Dockerfile

El archivo Dockerfile fue diseñado paso a paso de la siguiente forma:

- 1) **Imagen base:** se utiliza `osrf/ros:jazzy-desktop`, una imagen oficial de ROS 2 que incluye las herramientas gráficas, librerías y soporte de Python.
- 2) **Configuración del entorno:** se define el uso de `bash` como shell principal y se desactiva la interacción manual con `ENV DEBIAN_FRONTEND=noninteractive` para evitar pausas durante la instalación.
- 3) **Instalación de dependencias:** mediante el comando `apt`, se actualiza el sistema e instalan los paquetes necesarios para compilar y ejecutar programas en Python, entre ellos: `colcon-common-extensions`, `matplotlib`, `nano` y `pip`.
- 4) **Creación del workspace:** se crea el directorio de trabajo `/root/ros2_ws/src` y dentro de él se genera automáticamente el paquete base `sensor_program`, utilizando el comando:

```
1 ros2 pkg create --build-type ament_python
2   ↳ sensor_program --license MIT
```

- 5) **Copia de nodos:** se copian los archivos Python (`sensor_node.py`, `reader_node.py` y `plotter_node.py`) desde la carpeta local del host `./nodes` hacia la ubicación correspondiente del paquete en el contenedor.
- 6) **copia de setup.py:** Adicionalmente, se carga anteriormente en una carpeta (en el host) el archivo `setup` con los `entrypoints` correspondientes.
- 7) **Compilación automática:** una vez copiados los nodos, el proyecto se compila automáticamente mediante `colcon build`, generando los binarios ejecutables del paquete `sensor_program`.
- 8) **Configuración persistente:** se agregan los comandos de entorno `source` al archivo `.bashrc`, de manera que ROS 2 y el workspace se carguen automáticamente cada vez que se inicia el contenedor.
- 9) **Entrypoint y comando final:** se copia el script `ros_entrypoint.sh` desde la imagen base para

mantener la inicialización estándar de ROS 2, y se define el comando por defecto `CMD ["bash"]`, que abre la terminal interactiva al iniciar el contenedor.

Este Dockerfile permitió construir un entorno funcional, que incluye todas las dependencias necesarias y el proyecto preconfigurado, dejando el sistema listo para ejecutar los nodos de ROS 2 sin pasos manuales adicionales.

Este archivo junto con los demás serán cargados en un GitHub correspondiente.

B. Punto 2: Configuración de carpetas compartidas (Shared Folder)

Una parte importante del reto consistió en crear carpetas compartidas entre el host (Windows) y el contenedor Docker, con el objetivo de facilitar la edición de los nodos y la transferencia de resultados. Gracias a esta configuración, los archivos `.py` pueden editarse directamente desde un editor en el host, y las imágenes generadas (`.png`) se almacenan automáticamente en el mismo directorio sin necesidad de copiar archivos manualmente.

Estructura de carpetas en el host: Las carpetas se organizaron dentro del proyecto en la siguiente ubicación:

```
1 \Users\sebas\Documents\9no\REDES_SENSORES\taller2\
2 |
3 --Dockerfile
4 --nodes\      # Archivos .py
5 --data\       # imagenes generadas
```

Montaje de las carpetas en el contenedor: Al crear o ejecutar el contenedor, se usó la opción `-v` (de `volume`) para enlazar las carpetas del host con rutas dentro del contenedor. El comando utilizado fue el siguiente:

```
1 docker run -it --name contenedor_ros2_t2_final \
2
3 -v
4   ↳ /mnt/c/Users/sebas/Documents/9no/REDES_SENSORES/
5   taller2/nodes:/root/ros2_ws/nodes \
6
7 -v
8   ↳ /mnt/c/Users/sebas/Documents/9no/REDES_SENSORES/
9   taller2/data:/root/ros2_ws/data \
10  imag_taller2_wsn
```

Listing 5: Ejecución del contenedor con carpetas compartidas

Explicación del comando:

- `-name contenedor_ros2_t2_1`: define el nombre del nuevo contenedor.
- `-v host:contenedor`: crea un volumen compartido entre ambas rutas.
- `/root/ros2_ws/nodes`: ruta dentro del contenedor donde se leen los nodos.
- `/root/ros2_ws/data`: ruta donde los nodos guardan las imágenes generadas.
- `imag_taller2_wsn`: nombre de la imagen generada a partir del Dockerfile anterior.

Verificación dentro del contenedor: Una vez iniciado el contenedor, se verificó la correcta sincronización de carpetas con el siguiente comando:

```
1 cd /root/ros2_ws
2 ls
```

La salida mostraba las carpetas `src`, `nodes` y `datatal` como se muestra en la Figura 5, confirmando que los volúmenes estaban montados correctamente. Cualquier archivo creado o modificado en el host aparecía de inmediato en el contenedor y viceversa.

```
root@efdadb46006a:~/ros2_ws# ls
build data install log nodes src
```

Fig. 5: Verificación de carpetas compartidas.

C. Punto 3: Verificación del funcionamiento de los nodos

En este punto se ejecutan los nodos usando los comandos de `colcon build` y entorno ROS2 / entorno proyecto junto con la ejecución de cada nodo en diferentes terminales. De esta forma en las Figuras 6 y 7, se observa que cada nodo está ejecutándose correctamente, ya que el nodo sensor está publicando diferentes valores de temperatura y el nodo lector está suscribiéndose al tópico correspondiente para poder leer la temperatura.

```
root@efdadb46006a:~/ros2_ws# ros2 run sensor_program sensor_node
[INFO] [1760649302.267877435] [sensor_node]: Nodo Sensor iniciado.
[INFO] [1760649303.257840505] [sensor_node]: Publicando: Temperatura: 27.07 °C
[INFO] [1760649304.251747753] [sensor_node]: Publicando: Temperatura: 21.30 °C
[INFO] [1760649305.251086778] [sensor_node]: Publicando: Temperatura: 29.85 °C
[INFO] [1760649306.245797405] [sensor_node]: Publicando: Temperatura: 22.30 °C
[INFO] [1760649307.253695205] [sensor_node]: Publicando: Temperatura: 29.22 °C
[INFO] [1760649308.296040487] [sensor_node]: Publicando: Temperatura: 25.93 °C
[INFO] [1760649309.273494104] [sensor_node]: Publicando: Temperatura: 29.52 °C
```

Fig. 6: Verificación de funcionamiento del nodo sensor.

```
root@efdadb46006a:~/ros2_ws# ros2 run sensor_program reader_node
[INFO] [1760649317.582499512] [reader_node]: Nodo Reader iniciado y escuchando sensor_data.
[INFO] [1760649303.261680545] [reader_node]: Recibido: Temperatura: 27.07 °C
[INFO] [1760649304.252337140] [reader_node]: Recibido: Temperatura: 21.30 °C
[INFO] [1760649305.251360999] [reader_node]: Recibido: Temperatura: 29.85 °C
[INFO] [1760649306.246418068] [reader_node]: Recibido: Temperatura: 22.30 °C
[INFO] [1760649307.254132785] [reader_node]: Recibido: Temperatura: 29.22 °C
[INFO] [1760649308.295340430] [reader_node]: Recibido: Temperatura: 25.93 °C
[INFO] [1760649309.274012991] [reader_node]: Recibido: Temperatura: 29.52 °C
[INFO] [1760649310.288003740] [reader_node]: Recibido: Temperatura: 25.43 °C
[INFO] [1760649311.271753666] [reader_node]: Recibido: Temperatura: 26.17 °C
[INFO] [1760649312.252303277] [reader_node]: Recibido: Temperatura: 27.56 °C
```

Fig. 7: Verificación funcionamiento del nodo lector.

D. Reto – Punto 4: Creación del nodo `plotter_node`

Como parte final del reto, se desarrolló un nodo adicional llamado **plotter_node**, encargado de recibir los datos del nodo lector y generar de manera periódica un gráfico con las temperaturas recibidas. Este nodo complementa la red de sensores simulada, ya que permite visualizar el comportamiento de las lecturas a lo largo del tiempo.

Algoritmo de funcionamiento: El siguiente algoritmo resume las tareas principales que realiza el nodo `plotter_node`:

Algorithm 3 Funcionamiento del nodo `plotter_node`

- 1: Inicializar el entorno de ROS 2.
- 2: Crear el nodo llamado `plotter_node`.
- 3: Suscribirse al tópico `sensor_data`.
- 4: Inicializar listas vacías para almacenar los datos y marcas de tiempo.
- 5: **while** ROS 2 esté activo **do**
- 6: Esperar la llegada de un mensaje desde el nodo lector.
- 7: **if** mensaje recibido **then**
- 8: Extraer el valor numérico de temperatura.
- 9: Guardar el valor en la lista de datos.
- 10: **end if**
- 11: **if** han pasado 5 segundos **then**
- 12: Generar una figura con `matplotlib`.
- 13: Dibujar la gráfica con los valores acumulados.
- 14: Guardar la imagen en la carpeta compartida
 `/root/ros2_ws/data/`.
- 15: **end if**
- 16: **end while**
- 17: Finalizar el nodo y cerrar ROS 2 correctamente.

Descripción del funcionamiento: El nodo `plotter_node` se implementó en Python usando la librería `rcipy` para la comunicación ROS 2 y `matplotlib` para la generación de gráficos. Su funcionamiento se basa en un esquema de suscripción: escucha el tópico `sensor_data` al que el nodo lector publica los datos de temperatura.

Cada vez que llega un mensaje, el nodo extrae el valor numérico (en °C) y lo almacena en una lista. Mediante un temporizador interno (`timer`), cada 5 segundos se genera una gráfica de los valores acumulados, mostrando cómo ha variado la temperatura a lo largo del tiempo.

La imagen se guarda automáticamente en la carpeta compartida:

```
1 /root/ros2_ws/data/sensor_plot.png
```

La carpeta `/root/ros2_ws/data` está vinculada con el directorio del host:

```
1 C:\Users\sebas\Documents\9no\REDES_SENSORES
2 \taller2\data
```

Por lo tanto, el archivo generado puede visualizarse directamente desde el sistema Windows. El gráfico se actualiza de manera automática cada vez que el nodo ejecuta la función de guardado, mostrando las muestras recibidas hasta ese momento.

Verificación de Funcionamiento: De igual forma que para los otros nodos, para el nodo de gráfica se ejecuta el contenedor en otra terminal y ejecutando los comandos correspondientes para cargar el entorno y el nodo. De esta forma se observa que en la Figura 8, el nodo se está ejecutando, la gráfica se actualiza cada 5 segundos y además se observa de forma directa en la carpeta vinculada al host.

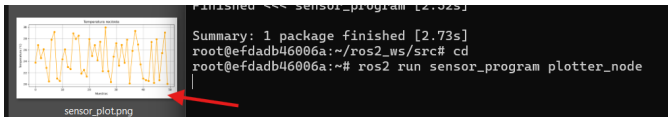


Fig. 8: Verificación funcionamiento del nodo plotter.

Finalmente, se observa en la Figura 9 el gráfico de el número de muestra y su temperatura correspondiente, corroborando los datos obtenidos mediante el nodo lector que fue proporcionado por el nodo sensor.

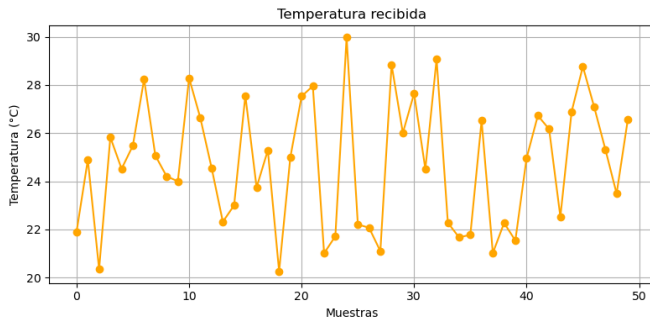


Fig. 9: Gráfica obtenida de los datos de temperatura del sensor.

V. ANÁLISIS DE TRÁFICO CON WIRESHARK

Como parte adicional al taller 2, se pidió realizar un análisis de tráfico usando Wireshark mediante la creación de una red de contenedores y usando un Docker nicolaka / netshoot como sniffer.

A. Configuración inicial y red Docker

Para permitir la captura del tráfico, se creó una red personalizada de Docker llamada **proyecto**, donde se conectaron tanto el contenedor principal con los nodos de ROS 2 como el contenedor que actuaría como sniffer. De esta manera, ambos comparten el mismo entorno de red virtual.

```
1 docker network create taller2_network
```

Listing 6: Creación de la red Docker personalizada

Una vez creada, se añadió el contenedor principal (donde se ejecutan los nodos) a esta red:

```
1 docker network connect proyecto
  ↳ container_ros2_taller2
```

Listing 7: Conexión del contenedor de ROS2 a la red

B. Captura de tráfico con Netshoot

Para capturar los paquetes, se utilizó la imagen **nicolaka/netshoot**, que contiene las herramientas de diagnóstico de red como `tcpdump`. Se montó una carpeta compartida para guardar el archivo `.pcap` directamente en el sistema host, dentro del proyecto.

```
1 docker run -it --rm --network taller2_network \
2
3 -v /mnt/c/Users/sebas/Documents/9no
4 /REDES_SENSORES/taller2/pcap:/pcap \
5
6 --cap-add=NET_ADMIN --cap-add=NET_RAW \
7
8 nicolaka/netshoot tcpdump -i eth0 -w
  ↳ /pcap/trafico_ros2.pcap
```

Listing 8: Ejecución del contenedor sniffer

Descripción del comando:

- `-network proyecto`: conecta el sniffer a la misma red Docker de los nodos ROS 2.
- `-v /mnt/...:/pcap`: crea un volumen compartido para guardar el archivo de captura.
- `-cap-add=NET_ADMIN -cap-add=NET_RAW`: otorgan permisos de administración y captura de red.
- `tcpdump -i eth0 -w /pcap/trafico_ros2.pcap`: inicia la captura en la interfaz virtual `eth0` y guarda el archivo.

Mientras el sniffer estaba activo, se ejecutaron los nodos `sensor_node`, `reader_node` y `plotter_node` dentro del contenedor principal. De esta forma, el sniffer capturó todas las tramas de comunicación entre ellos.

Al detener la captura se generó el archivo `trafico_ros2.pcap` que se guardó automáticamente en el host, en la siguiente ruta:

```
1 C:\Users\sebas\Documents\9no\REDES_SENSORES
2 \taller2\pcap\trafico_ros2.pcap
```

C. Visualización en Wireshark

El archivo `.pcap` se abrió en **Wireshark** desde Windows para analizar los protocolos y puertos utilizados por ROS 2. Como ROS 2 utiliza el protocolo **DDS (Data Distribution Service)** sobre **RTPS (Real-Time Publish-Subscribe)**, la comunicación ocurre principalmente sobre el protocolo **UDP**.

Para filtrar los paquetes correspondientes al tráfico DDS/RTPS, se aplicó el siguiente filtro en la barra de Wireshark:

```
1 udp.port >= 7400 && udp.port <= 7500
```

De esta forma en la Figura 10 se puede observar: una secuencia de paquetes con origen en la dirección interna 172.21.0.2 y destino 239.255.0.1, que corresponde a una dirección de difusión (multicast) dentro de la red Docker.

Todos los paquetes capturados están etiquetados como RTPS DATA(p) o INFO_TS, lo que significa que contienen tanto los datos publicados como la información de tiempo que permite mantener la sincronización entre los nodos.

En la parte inferior del análisis se muestran detalles como el vendor eProsima Fast-RTPS, lo que confirma que ROS 2 está utilizando el middleware FastDDS para manejar la comunicación. También se visualizan los campos "reader" y "writer", que identifican a los nodos que envían y reciben los mensajes, indicando que hay un intercambio activo entre los participantes del sistema.

udp.port == 7400 && udp.port == 7500

No.	Time	Source	Destination	Protocol	Length	Info
11	69.405497	172.21.0.2	239.255.0.1	RTPS	606	INFO_TS, DATA(p), Unknown[80]
12	69.506265	172.21.0.2	239.255.0.1	RTPS	606	INFO_TS, DATA(p), Unknown[80]
13	69.606688	172.21.0.2	239.255.0.1	RTPS	606	INFO_TS, DATA(p), Unknown[80]
14	69.708546	172.21.0.2	239.255.0.1	RTPS	606	INFO_TS, DATA(p), Unknown[80]
15	69.811085	172.21.0.2	239.255.0.1	RTPS	606	INFO_TS, DATA(p), Unknown[80]
16	69.915018	172.21.0.2	239.255.0.1	RTPS	606	INFO_TS, DATA(p), Unknown[80]
17	72.916921	172.21.0.2	239.255.0.1	RTPS	606	INFO_TS, DATA(p), Unknown[80]
18	74.294768	172.21.0.2	239.255.0.1	RTPS	606	INFO_TS, DATA(p), Unknown[80]

Frame 15: 606 bytes on wire (4848 bits), 606 bytes captured (4848 bits) on interface eth0
 Ethernet II, Src: 2e:40:41:93:68:4b (2e:40:41:93:68:4b), Dst: IPv4multicast_7f:00:01 (01:00:5e:7f:00:01)
 Internet Protocol Version 4, Src: 172.21.0.2, Dst: 239.255.0.1
 User Datagram Protocol, Src Port: 42889, Dst Port: 7400
 Real-Time Publish-Subscribe Wire Protocol
 Magic: RTPS
 Protocol version: 2.3
 vendorId: 0115 (eProsima - Fast-RTPS)
 guidPrefix: 010ff30d50146e5000000000
 Default port mapping: domainId=Unknown, participantId=120, nature=UNICAST_METATRAFFIC
 [domainId: 4294967295]
 [participantId: 120]
 [traffic_nature: UNICAST_METATRAFFIC (0)]
 submessageId: INFO_TS (0x009)
 Flags: 0x01, Endianness octetsToNextHeader: 8
 Timestamp: Oct 16, 2025 21:48:41.817900500 UTC
 submessageId: DATA (0x05)
 Flags: 0x05, Data present, Endianness octetsToNextHeader: 468
 0000 0000 0000 0000 = Extra flags: 0x0000
 Octets to inline QoS: 16
 readerEntityId: ENTITYID_BUILTIN_PARTICIPANT_READER (0x000100c7)
 readerEntityKey: 0x000100
 readerEntityKind: Built-in reader (with key) (0xc7)
 writerEntityId: ENTITYID_BUILTIN_PARTICIPANT_WRITER (0x000100c2)
 writerEntityKey: 0x000100
 writerEntityKind: Built-in writer (with key) (0xc2)
 writerSeqNumber: 1
 serializedData
 submessageId: Vendor-specific (0x00)

Fig. 10: Captura de tráfico RTPS usando el filtro indicado.

En la Figura 11 se muestra la vista general del tráfico dentro de la red virtual de Docker. Se identifican varios tipos de mensajes de red, entre ellos:

- **RTPS (606 bytes):** tramas generadas por los nodos de ROS 2 que contienen los datos publicados por el sensor y los mensajes de control del middleware DDS.

En esta etapa se confirma que el tráfico RTPS (en color azul en Wireshark) fluye dentro de la red 172.21.0.0/16, la cual es la red interna asignada por Docker a la red personalizada proyecto. Esto indica que todos los nodos ROS 2 están comunicándose correctamente a través de esa red virtual.

1	0.000000	::	ff02::16	ICMPv6	110	Multicast Listener Report Message v2
2	0.000013	::	ff02::1:ff02:2084	ICMPv6	85	Neighbor Solicitation for fe80:b461:19ff:fe08:2084
3	0.263985	dc1e:2feb:16:2d	Broadcast	ARP	42	ARP Announcement for 172.21.0.3
4	0.264021	dc1e:2feb:16:2d	Broadcast	ARP	42	ARP Announcement for 172.21.0.3
5	1.403956	fe80:b461:19ff:fe08:2	ff02::16	ICMPv6	120	Multicast Listener Report Message v2
6	1.069945	fe80:b461:19ff:fe08:2	ff02::16	ICMPv6	90	Multicast Listener Report Message v2
7	1.109944	fe80:b461:19ff:fe08:2	ff02::16	ICMPv6	130	Multicast Listener Report Message v2
8	1.263950	dc1e:2feb:16:2d	Broadcast	ARP	42	ARP Announcement for 172.21.0.3
9	1.263981	dc1e:2feb:16:2d	Broadcast	ARP	42	ARP Announcement for 172.21.0.3
10	1.040911	fe80:b461:19ff:fe08:2	ff02::16	ICMPv6	90	Multicast Listener Report Message v2

Frame 10: 90 bytes on wire (720 bits), 90 bytes captured (720 bits) on interface eth0
 Ethernet II, Src: b6:61:19:08:28:84 (b6:61:19:08:28:84), Dst: IPv4multicast_16 (33:33:00:00:00:16)
 Internet Protocol Version 4, Src: fe80:b461:19ff:fe08:2084, Dst: ff02::16
 Internet Control Message Protocol v6
 type: Multicast Listener Report Message v2 (143)
 Code: 0
 Checksum: 0x7abb [correct]
 [Checksum Status: Good]
 Reserved: 0000
 Number of Multicast Address Records: 1
 Multicast Address Record Changed to exclude: ff02::1:ff00:0

Fig. 11: Captura de tráfico general.

CONCLUSIONES

En conclusión, este taller permitió comprender de forma práctica la estructura y funcionamiento de ROS 2, desarrollando un entorno funcional dentro de Docker. Se alcanzó el objetivo de implementar una red de nodos que simulan sensores, lectores y un proceso de graficación, demostrando la comunicación efectiva bajo el modelo publicador-suscriptor del middleware DDS.

Durante el desarrollo, se puso en práctica el uso de comandos esenciales como `colcon build`, `ros2 run` y `source setup.bash`, los cuales fueron importantes para compilar, ejecutar y enlazar correctamente los nodos.

El reto presentó una dificultad mayor, ya que se tuvo automatizar todo el entorno mediante un `Dockerfile`, configurar carpetas compartidas entre el host y el contenedor, y finalmente

generar una imagen lista para ejecutarse en cualquier sistema. Este proceso mejoró uso de contenedores como herramientas de despliegue reproducible confirmando sus ventajas respecto a maquinas virtuales.

Además, el análisis con Wireshark permitió visualizar el tráfico RTPS y confirmar la transmisión de mensajes entre nodos, validando la comunicación distribuida de ROS 2. El desarrollo del nodo `plotter_node` añadió un componente visual que permitió analizar la evolución de los datos, reforzando la integración entre los módulos.

Finalmente, este taller resultó importante para mejorar los conocimientos sobre redes de sensores y comunicación entre nodos usando ROS2. Además, se resalta la importancia de ROS 2 como herramienta esencial en redes de sensores, ya que presenta una buena modularidad al momento de implementar sensores (y posteriormente diferentes elementos).

REFERENCES

- [1] Open Source Robotics Foundation, *Robot operating system (ros) – ros wiki*, <https://wiki.ros.org/>, Accedido: 14 octubre de 2025, 2024.
- [2] Real-Time Innovations (RTI), *Ros 2 and dds: Understanding the middleware layer*, <https://community.rti.com/page/ros-2-what-dds>, Accedido: 14 octubre de 2025, 2024.
- [3] Robotnik Automation, *Ros 2: Visión general y puntos clave del software de robótica*, <https://robotnik.eu/es/ros-2-robot-operating-system-vision-general-y-puntos-clave-del-software-de-robotica/>, Accedido: 14 octubre de 2025, 2024.
- [4] U. de Cuenca, “Taller 2: Introducción a ros (robot operating system),” Facultad de Ingeniería, Carrera de Telecomunicaciones, Tech. Rep., 2025, Guía práctica académica.

ANEXOS

Anexo A – Repositorio del proyecto en GitHub

El código fuente completo del taller, incluyendo los nodos `sensor_node`, `reader_node` y `plotter_node`, así como el archivo `Dockerfile`, se encuentra disponible en el siguiente repositorio público de GitHub.

- **Repositorio GitHub:** https://github.com/sebast15g/Taller2_ROS2_Redes_Sensores

Anexo B – Imagen del contenedor en Docker Hub

Como parte final, la imagen Docker generada a partir del archivo `Dockerfile` fue publicada en Docker Hub para facilitar la reutilización del entorno. Desde este repositorio se puede descargar directamente la imagen preconfigurada de ROS 2 que contiene todos los nodos y el paquete `sensor_program` ya compilado.

- **Imagen del taller (reto, creación de la imagen a partir del Dockerfile):** https://hub.docker.com/repository/docker/sebast15g/ros2_taller2_reto/general
- **Imagen del desarrollo manual (creación de imagen a partir del contenedor ejecutado paso a paso):** https://hub.docker.com/repository/docker/sebast15g/ros2_taller2_manual/general