Taller 3: ROS (Robot Operating System) sensor de temperatura (Serial)

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

2nd Martín Vinces
Universidad de Cuenca, Facultad de Ingeniería
Ingeniería en Telecomunicaciones
Cuenca, Ecuador
carlosm.vinces@ucuenca.edu.ec

***Abstract*—El presente trabajo describe el desarrollo e integración de un entorno de monitoreo en tiempo real para sistemas basados en ROS 2, utilizando herramientas modernas como InfluxDB, Grafana y Docker. La práctica consistió en implementar una arquitectura contenedorizada que permite capturar, procesar, almacenar y visualizar datos sensoriales adquiridos desde un dispositivo físico (Arduino) a través de nodos de ROS 2 configurados con diferentes políticas de Calidad de Servicio (QoS). Los datos generados fueron almacenados en una base de datos de series temporales (InfluxDB) y posteriormente representados gráficamente mediante paneles interactivos en Grafana. Además, se realizaron capturas de tráfico RTPS con Wireshark para analizar el comportamiento del protocolo DDS bajo los modos *reliable* y *best_effort*. Este trabajo demuestra la eficacia del uso combinado de ROS 2 y Docker para el análisis experimental de QoS en redes de sensores distribuidas.**

***Index Terms*—ROS 2, QoS, InfluxDB, Grafana, Docker, RTPS, DDS, Monitoreo en tiempo real, Series temporales.**

I. Introducción

El crecimiento de los sistemas distribuidos y el Internet de las Cosas (IoT) ha impulsado la necesidad de infraestructuras capaces de adquirir, procesar y visualizar datos en tiempo real de manera eficiente y escalable. En este contexto, ROS 2 (Robot Operating System 2) se ha consolidado como una plataforma de referencia para el desarrollo de arquitecturas basadas en comunicación *publish-subscribe*, apoyándose en el protocolo DDS (Data Distribution Service) y su implementación RTPS (Real-Time Publish-Subscribe). La flexibilidad de ROS 2 permite definir políticas de Calidad de Servicio (QoS) adaptadas a las necesidades de transmisión, facilitando la experimentación con perfiles como *reliable* y *best_effort* para comparar desempeño y fiabilidad en entornos de red heterogéneos.

El presente trabajo se centra en la construcción de un entorno de monitoreo contenedorizado que integra ROS 2 con los servicios de almacenamiento y visualización de datos InfluxDB y Grafana, desplegados sobre Docker. Mediante esta infraestructura se implementa un flujo completo que abarca la adquisición de datos desde un sensor físico, su conversión a temperatura en grados Celsius y su almacenamiento en una base de datos de series temporales para su posterior análisis visual. Adicionalmente, se emplea Wireshark para la inspección del tráfico RTPS, permitiendo observar las diferencias en los patrones de comunicación entre los modos de QoS mencionados. Con ello se busca no solo validar el comportamiento del sistema, sino también fortalecer la comprensión del modelo DDS en aplicaciones de redes de sensores distribuidas.

II. Marco Teórico

El presente marco teórico aborda los fundamentos tecnológicos que sustentan el desarrollo de la práctica del Taller 3, la cual integra conceptos de monitoreo en tiempo real, almacenamiento de series temporales y comunicación distribuida mediante ROS 2. En particular, se analizan los componentes esenciales del entorno implementado: el sistema operativo en contenedores Docker, la base de datos InfluxDB, la herramienta de visualización Grafana y su interconexión con el ecosistema ROS 2 a través de nodos exportadores. Esta integración permite conformar una arquitectura moderna de *Internet de las Cosas* (IoT) para la adquisición, procesamiento y análisis de datos sensoriales en entornos distribuidos.

A. ROS 2 y el paradigma de comunicación distribuida

ROS 2 (Robot Operating System 2) es un marco de trabajo de código abierto orientado al desarrollo de sistemas robóticos y de automatización basados en comunicación distribuida. Su funcionamiento se sustenta en el estándar DDS (Data Distribution Service), que implementa el modelo de publicación-suscripción bajo el protocolo RTPS (Real-Time Publish-Subscribe). Este modelo permite que múltiples nodos intercambien datos de manera descentralizada mediante tópicos, sin requerir una jerarquía cliente-servidor. Cada tópico en ROS 2 se caracteriza por parámetros de *Quality of Service* (QoS), los cuales definen la política de entrega de mensajes. Entre las más relevantes se encuentran `RELIABLE` y `BEST_EFFORT`: la primera garantiza la recepción completa mediante confirmaciones y retransmisiones, mientras que la segunda prioriza la inmediatez y reduce la latencia a costa de posibles pérdidas de información. La correcta elección del perfil QoS

depende de los requisitos de la aplicación, balanceando confiabilidad y eficiencia.

B. Contenedores Docker y orquestación de servicios

Docker es una plataforma de virtualización a nivel de sistema operativo que permite empaquetar aplicaciones junto con sus dependencias en unidades portables denominadas contenedores. A diferencia de las máquinas virtuales tradicionales, los contenedores comparten el mismo kernel del sistema anfitrión, lo que reduce significativamente el consumo de recursos y mejora la portabilidad del entorno. En esta práctica, Docker se emplea para aislar y desplegar los distintos componentes del sistema: los servicios de almacenamiento y visualización (InfluxDB y Grafana) se ejecutan en contenedores dedicados, mientras que los nodos de ROS 2 operan en un contenedor independiente. Todos los servicios se interconectan a través de una red virtual común definida mediante *Docker Compose*, lo que permite la comunicación por nombre de servicio (por ejemplo, `influxdb`) sin necesidad de configurar direcciones IP estáticas. Este enfoque facilita la replicabilidad y el despliegue de la infraestructura en diferentes entornos, asegurando que el comportamiento del sistema sea idéntico sin importar el host físico o la distribución de Linux utilizada.

C. InfluxDB: base de datos para series temporales

InfluxDB es una base de datos optimizada para la gestión de series temporales, es decir, datos que varían en el tiempo y requieren registros asociados a una marca temporal. A diferencia de las bases de datos relacionales, InfluxDB se estructura en torno a mediciones (*measurements*), campos (*fields*) y etiquetas (*tags*). Cada registro incluye una marca de tiempo precisa que permite realizar consultas cronológicas, agregaciones y análisis estadísticos. En el contexto de esta práctica, InfluxDB actúa como repositorio de los valores de temperatura generados por los nodos ROS 2. El nodo *exporter_node* recibe los datos procesados y los envía al bucket `temperatures` de InfluxDB mediante la API HTTP o el cliente Python `influxdb-client`. La estructura resultante permite almacenar los valores de temperatura con su respectiva marca temporal, posibilitando posteriormente su análisis histórico o en tiempo real. El uso de contenedores facilita la inicialización automática de InfluxDB mediante variables de entorno, creando usuarios, organizaciones y buckets de manera reproducible. Esto permite un despliegue consistente sin necesidad de configuración manual, garantizando un entorno listo para registrar y consultar datos desde ROS 2.

D. Grafana: visualización y monitoreo de datos

Grafana es una herramienta de código abierto especializada en la visualización de datos provenientes de fuentes heterogéneas, incluyendo bases de datos de series temporales como InfluxDB. Su principal funcionalidad consiste en crear paneles interactivos (*dashboards*) que representan gráficamente el comportamiento de las variables monitoreadas a través de gráficos, indicadores y alertas. En esta práctica, Grafana se conecta al servicio InfluxDB dentro del mismo entorno Docker mediante el nombre de host `influxdb`. La comunicación se realiza utilizando el lenguaje de consultas *Flux*, propio de InfluxDB 2.x, el cual permite filtrar, agrupar y agregar datos directamente desde la interfaz de Grafana. De esta forma, los valores de temperatura publicados por ROS 2 y almacenados en InfluxDB pueden visualizarse en tiempo real como una gráfica continua. El panel configurado para el reto permite comparar visualmente el comportamiento del sistema bajo distintos perfiles de QoS, identificando diferencias en la estabilidad, continuidad o frecuencia de actualización de los datos. Esta integración convierte a Grafana en una herramienta de diagnóstico y evaluación del desempeño de la red ROS 2.

E. Integración ROS 2 – InfluxDB – Grafana en entorno Docker

La integración completa de ROS 2 con InfluxDB y Grafana en un entorno contenedorizado representa una solución moderna para el monitoreo de sistemas distribuidos. El flujo de datos sigue una secuencia definida: (1) El nodo *sensor_node* adquiere los valores desde un dispositivo físico (Arduino) a través de un puerto serial y los publica en un tópico de ROS 2. (2) El nodo *processor_node* convierte los valores crudos en unidades físicas (temperatura en °C). (3) El nodo *exporter_node* escribe cada muestra en InfluxDB con su correspondiente marca temporal. (4) Grafana consulta periódicamente la base de datos y actualiza la visualización de los datos en tiempo real. Todos estos procesos se ejecutan en contenedores separados, interconectados mediante la red virtual de Docker y sincronizados bajo la orquestación del archivo *docker-compose.yml*. Esta arquitectura modular proporciona independencia entre las capas de adquisición, almacenamiento y visualización, a la vez que permite analizar el impacto de las configuraciones de QoS en la estabilidad y continuidad del flujo de información. En consecuencia, la práctica no sólo demuestra el funcionamiento técnico del entorno ROS 2, sino también el potencial de las tecnologías de contenedorización y monitoreo para el análisis de sistemas distribuidos en el ámbito de las redes de sensores.

III. Desarrollo del reto

En esta sección se documenta la ejecución del reto del Taller 3: integración de un flujo *Arduino* $\rightarrow$ *ROS 2* $\rightarrow$ *InfluxDB* $\rightarrow$ *Grafana*, habilitación de perfiles *QoS* alternables (*reliable* y *best_effort*) y captura/analítica del tráfico *DDS/RTPS* con *Wireshark* para contrastar ambos modos. La plataforma base (Arduino operando

y nodos ROS 2 funcionales) se asumió previamente validada.

A. Arquitectura y objetivos

- Un contenedor ROS 2 (`ros2_sniff`) ejecuta los nodos sensor, de adquisición, procesado, monitor y exportación de datos.
- InfluxDB 2.x almacena la serie temporal de temperatura (medición `temperature`, campo `celsius`).
- Grafana consume InfluxDB mediante consultas *Flux* para visualizar el comportamiento temporal.
- Un contenedor auxiliar (`netshoot`) captura tráfico RTPS (UDP 7400–7500) para su posterior análisis en Wireshark.
- Se realizan dos corridas **no simultáneas**: primero *reliable*, luego *best_effort*.

1) Nodo sensor:

El nodo *sensor_node* tiene como propósito principal leer muestras numéricas provenientes de un dispositivo conectado al puerto serial, como un Arduino, y publicarlas en el tópico `/sensor_data` en formato de mensajes de tipo `Int32`. Este nodo incorpora la posibilidad de alternar entre dos perfiles de Calidad de Servicio (QoS): `reliable` y `best_effort`, lo que permite estudiar la diferencia entre un esquema de transmisión confiable y uno de entrega no garantizada. La implementación permite configurar los parámetros básicos del nodo sin modificar el código fuente, garantizando flexibilidad y reproducibilidad en entornos diversos.

Al inicializarse el nodo, se declaran los parámetros `serial_port`, `baudrate`, `period` y `reliability`, con valores por defecto que pueden sobrescribirse desde la línea de comandos o mediante un archivo *launch*. El puerto serial, definido por defecto como `/dev/ttyUSB0`, puede variar según el sistema operativo o la forma en que se exponga el dispositivo USB al contenedor Docker, por lo que este parámetro se mantiene dinámico. El valor del parámetro `baudrate` determina la velocidad de comunicación, mientras que `period` establece el intervalo en segundos con el cual se ejecutará el temporizador que activa la lectura del sensor. Finalmente, el parámetro `reliability` controla la fiabilidad del enlace: el modo `reliable` garantiza la entrega de los mensajes mediante confirmaciones internas de la capa DDS, mientras que el modo `best_effort` prioriza la inmediatez de la transmisión con menor sobrecarga, pero sin mecanismos de reenvío ante pérdidas.

El perfil QoS se construye con una profundidad de cola (`depth`) de diez elementos y una política de historial `KEEP_LAST`, que mantiene solo los mensajes más recientes. Si el valor del parámetro `reliability` es igual a `'reliable'`, se asigna la política `RELIABLE`; en caso contrario, se aplica `BEST_EFFORT`. Esta configuración asegura que el

Algorithm 1 Nodo Sensor en ROS 2 (Serial → `/sensor_data` con QoS conmutable)

Procedimiento principal
InicializarROS2()
nodo ← CrearNodo(*sensor_node*)
// Declaración de parámetros con valores por defecto
DeclararParametro(`serial_port`, `'/dev/ttyUSB0'`)
DeclararParametro(`baudrate`, `9600`)
DeclararParametro(`period`, `1.0`) {segundos}
DeclararParametro(`reliability`, `'reliable'`) {`reliable` — `best_effort`}
// Lectura de parámetros efectivos
port ← ObtenerParametro(`serial_port`)
baud ← ObtenerParametro(`baudrate`)
period ← ObtenerParametro(`period`)
rel_str ← ToLower(ObtenerParametro(`reliability`))
// Construcción del perfil QoS
qos ← CrearQoS(`depth`=10, `history`=`KEEP_LAST`)
if rel_str = `'reliable'` **then**
  qos.reliability ← `RELIABLE`
else
  qos.reliability ← `BEST_EFFORT`
end if
// Inicialización del puerto serie
serial ← AbrirSerial(puerto=port, baudios=baud, timeout=1)
// Creación del publicador y del temporizador
pub ← CrearPublicador(`Int32`, `/sensor_data`, qos)
CrearTemporizador(period, `PublicarDato`)
Función `PublicarDato`():
  try:
    linea ← LeerLinea(serial)
    linea ← DecodificarUTF8YTrim(linea) {ignora errores}
    if EsDigito(linea) **then**
      valor ← ConvertirAEntero(linea)
      msg ← Mensaje(`Int32`, data=valor)
      Publicar(pub, msg)
      LogInfo(`'crudo='` + valor)
    else if linea ≠ cadenaVacía **then**
      LogWarn(`'dato inválido: "'` + linea + `'"'`)
  catch e:
    LogError(`'Error serial: '` + e)
// Ciclo de vida del nodo
MantenerNodoActivo(nodo) {`rclpy.spin`}
DestruirNodo(nodo)
ApagarROS2() =0

nodo se adapte automáticamente al modo de transmisión deseado, sin requerir cambios en la estructura interna del código.

Posteriormente, el nodo inicializa el puerto serial con un tiempo de espera de un segundo, evitando bloqueos indefinidos por ausencia de datos. Una vez abierto el canal de comunicación, se crea un publicador asociado al tópico `/sensor_data` y se programa un temporizador que ejecuta la función `PublicarDato()` a intervalos regulares definidos por el parámetro `period`. Dicha función constituye el núcleo operativo del nodo. En cada ciclo, intenta leer una línea completa desde el puerto serial, decodificarla a formato UTF-8 y eliminar posibles espacios o saltos de línea. Si la lectura resulta ser una cadena de dígitos válida, se convierte a entero y se encapsula en un mensaje `Int32`, el cual se publica inmediatamente en el tópico configurado. En cambio, si el valor leído contiene caracteres no numéricos o está vacío, el sistema genera una advertencia mediante el registro interno de ROS, manteniendo el proceso en ejecución sin interrumpir el flujo del nodo. Cualquier excepción derivada de errores de lectura o desconexión del puerto es capturada por una estructura `try-except`, y se reporta a través del nivel de error correspondiente en el registro, garantizando la tolerancia ante fallos temporales del dispositivo.

Durante su ciclo de vida, el nodo permanece activo dentro del bucle de eventos de ROS 2, manteniendo operativo el temporizador y respondiendo a las señales del sistema hasta que se ordena su finalización. En la etapa de cierre, se destruye el objeto nodo y se realiza una llamada a `rclpy.shutdown()` para liberar los recursos del entorno. Este diseño modular y robusto permite una integración fluida con otros nodos de la arquitectura, como los encargados de procesar, visualizar o exportar los datos. Además, la flexibilidad del parámetro `reliability` facilita la comparación experimental entre el comportamiento de las configuraciones `RELIABLE` y `BEST_EFFORT` en términos de latencia, estabilidad de transmisión y volumen de tráfico de control, aspectos que pueden observarse posteriormente mediante las capturas analizadas en Wireshark.

2) Nodo procesador: El nodo *processor_node* constituye la segunda etapa del flujo de procesamiento dentro del sistema ROS 2, actuando como intermediario entre el nodo de adquisición y los módulos de monitoreo o almacenamiento. Su función principal consiste en recibir los valores enteros crudos publicados por el nodo *sensor_node* en el tópico `/sensor_data`, realizar una conversión lineal para expresar estos valores en grados Celsius y reenviar el resultado a un nuevo tópico denominado `/temperature_celsius`, utilizando mensajes de tipo `Float32`. Este diseño desacopla la lectura física del sensor del proceso de interpretación, permitiendo que los nodos subsiguientes trabajen directamente con magnitudes físicas significativas.

Algorithm 2 Nodo Procesador en ROS 2 (`/sensor_data` → `/temperature_celsius` con conversión lineal y QoS conmutable)

Procedimiento principal
InicializarROS2()
nodo ← CrearNodo(*processor_node*)
// Declaración del parámetro de fiabilidad
DeclararParametro(`reliability`, `'reliable'`) {Modos: `reliable` — `best_effort`}
// Lectura del parámetro y configuración de QoS
rel_str ← ToLower(ObtenerParametro(`reliability`))
qos_sub ← CrearQoS(`depth`=10, `history`=`KEEP_LAST`)
qos_pub ← CrearQoS(`depth`=10, `history`=`KEEP_LAST`)
if rel_str = `'reliable'` **then**
  qos_sub.reliability ← `RELIABLE`
  qos_pub.reliability ← `RELIABLE`
else
  qos_sub.reliability ← `BEST_EFFORT`
  qos_pub.reliability ← `BEST_EFFORT`
end if
// Creación de la suscripción y publicación
sub ← CrearSuscripcion(`Int32`, `/sensor_data`, callback=`ProcesarDato`, qos_sub)
pub ← CrearPublicador(`Float32`, `/temperature_celsius`, qos_pub)
Función `ProcesarDato`(msg):
  // Conversión lineal del valor ADC a temperatura
  temp ← (msg.data / 1023.0) * 100.0
  out ← Mensaje(`Float32`, data=Redondear(temp, 2))
  Publicar(pub, out)
  LogInfo(`'temp='` + out.data + `' °C'`)
MantenerNodoActivo(nodo) {`rclpy.spin`}
DestruirNodo(nodo)
ApagarROS2() =0

Durante la inicialización, el nodo declara un único parámetro denominado `reliability`, cuyo propósito es controlar la política de fiabilidad aplicada tanto al suscriptor como al publicador. Este parámetro puede adoptar los valores `'reliable'` o `'best_effort'`, determinando así el comportamiento de entrega dentro del middleware DDS. Posteriormente, el parámetro se lee y se convierte a minúsculas para garantizar una comparación consistente. Con base en su valor, se construyen dos perfiles de Calidad de Servicio (QoS): uno para la suscripción y otro para la publicación. Ambos perfiles comparten las mismas características: profundidad de cola igual a

diez mensajes y una política de historial `KEEP_LAST`. Si el valor del parámetro es `'reliable'`, se asigna la política `RELIABLE` a ambos perfiles; en caso contrario, se utiliza `BEST_EFFORT`. De este modo, la configuración del nodo mantiene la coherencia del canal de comunicación a lo largo de toda la cadena de procesamiento.

Una vez definidos los perfiles QoS, el nodo crea una suscripción al tópico `/sensor_data` y un publicador para el tópico `/temperature_celsius`. Cada vez que se recibe un mensaje `Int32`, se activa la función de *callback* `ProcesarDato()`. Esta función ejecuta una conversión lineal sencilla que traduce el valor crudo del conversor analógico-digital (ADC) a temperatura en grados Celsius. La expresión utilizada `(valor / 1023.0) * 100.0` supone un sensor cuya salida se encuentra normalizada en un rango de 0 a 100 °C sobre una resolución de 10 bits (1023 niveles). El resultado se redondea a dos decimales y se encapsula en un mensaje de tipo `Float32`. Posteriormente, el mensaje se publica en el tópico `/temperature_celsius`, donde podrá ser leído por los nodos de monitoreo, graficación o exportación de datos. Cada publicación se acompaña de un registro informativo en consola que muestra la temperatura actual calculada.

El ciclo de vida del nodo se mantiene activo mediante la función `rclpy.spin()`, que conserva operativas las colas de mensajes y callbacks hasta que se recibe una señal de terminación. Finalmente, al finalizar la ejecución, se destruye el nodo y se libera la instancia de ROS mediante `rclpy.shutdown()`. Esta estructura modular y parametrizable permite la conmutación del modo QoS sin modificar el código fuente, favoreciendo la reproducibilidad de pruebas comparativas entre *reliable* y *best_effort*. Además, la separación lógica entre lectura y procesamiento facilita la reutilización del nodo en otros proyectos que requieran conversión o calibración de valores provenientes de sensores analógicos.

3) Nodo monitor: El nodo *monitor_node* constituye la etapa final de visualización textual dentro del sistema de comunicación. Su función es recibir los mensajes de temperatura publicados en el tópico `/temperature_celsius` y mostrarlos en la consola del entorno ROS 2 en tiempo real. Este nodo no realiza procesamiento adicional, sino que actúa como observador o consumidor de datos, confirmando el correcto flujo de información entre los componentes previos del sistema.

En su inicialización, el nodo declara el parámetro `reliability`, el cual permite seleccionar entre las políticas de entrega `RELIABLE` y `BEST_EFFORT`. Con base en este parámetro, se genera un perfil de Calidad de Servicio (QoS) que determina la forma en que se reciben los mensajes. La configuración utiliza una profundidad de cola de diez elementos y un historial `KEEP_LAST`, suficiente para asegurar la recepción de las muestras más recientes sin saturar la memoria. Si el valor del parámetro es `'reliable'`, el nodo espera confirmaciones internas de recepción, garantizando la entrega de todos los mensajes; en caso contrario, se prioriza la inmediatez de la transmisión sin mecanismos de acuse.

Algorithm 3 Nodo Monitor en ROS 2 (`/temperature_celsius` → Consola de salida con QoS conmutable)

```
Procedimiento principal
InicializarROS2()
nodo ← CrearNodo(monitor_node)
// Declaración del parámetro de fiabilidad
DeclararParametro(reliability, 'reliable') {Modos: reliable — best_effort}
// Configuración del perfil QoS
rel_str ← ToLower( ObtenerParametro(reliability))
qos ← CrearQoS(depth=10, history=KEEP_LAST)
if rel_str = 'reliable' then
    qos.reliability ← RELIABLE
else
    qos.reliability ← BEST_EFFORT
end if
// Creación de la suscripción
sub ← CrearSuscripcion(Float32, /temperature_celsius, callback=MostrarDato, qos)
Función MostrarDato(msg):
    temperatura ← msg.data
    LogInfo('Temperatura actual: ' + temperatura + ' °C')
MantenerNodoActivo(nodo) {rclpy.spin}
DestruirNodo(nodo)
ApagarROS2() =0
```

Una vez definido el perfil QoS, el nodo crea una suscripción al tópico `/temperature_celsius` con tipo de mensaje `Float32`. Cada vez que llega un mensaje nuevo, se ejecuta la función `MostrarDato()`, que extrae el valor de temperatura del mensaje y lo imprime en la consola mediante una llamada de registro informativo. Esta visualización continua permite verificar la frecuencia de publicación, la estabilidad del flujo de datos y la coherencia de las conversiones realizadas por el nodo procesador.

El nodo se mantiene activo en el ciclo de eventos de ROS 2 a través de `rclpy.spin()`, procesando los mensajes que arriban hasta que se detiene la ejecución. En la fase de finalización, se destruye el nodo y se liberan los recursos del entorno. Su simplicidad lo convierte en una herramienta esencial para monitorear el comportamiento del sistema durante pruebas de campo o ajustes de configuración de QoS, ya que permite observar directamente los efectos de la fiabilidad

configurada sobre la tasa de recepción de datos.

4) Nodo exportador: El nodo *exporter_node* se encarga de la etapa de almacenamiento persistente dentro del flujo de datos. Su función consiste en recibir las temperaturas procesadas que publica el nodo *processor_node* en el tópico `/temperature_celsius` y escribirlas en una base de datos de series temporales InfluxDB, desde donde posteriormente pueden consultarse y visualizarse en Grafana. De esta forma, el nodo integra el ecosistema ROS 2 con un sistema de monitoreo orientado a datos históricos y análisis temporal.

Durante su inicialización, el nodo declara el parámetro `reliability` para permitir la conmutación del perfil QoS, y recupera las credenciales de conexión desde variables de entorno, las cuales deben haberse configurado previamente en el contenedor: `INFLUX_URL`, `INFLUX_TOKEN`, `INFLUX_ORG` e `INFLUX_BUCKET`. Esta estrategia evita incluir información sensible directamente en el código fuente, y permite la portabilidad del nodo entre entornos de ejecución diferentes. A partir de dichos valores se instancia el cliente de InfluxDB y se crea la interfaz de escritura (`writeAPI`), que será la encargada de enviar los registros a la base de datos.

Con la configuración del QoS, el nodo establece la fiabilidad de la suscripción al tópico de temperatura, siguiendo la misma política que los demás nodos del sistema. Si el modo `reliable` está activo, los mensajes se reciben con acuse de entrega, garantizando la persistencia completa de los datos. En modo `best_effort`, se prioriza la baja latencia sobre la garantía de recepción, permitiendo comparar el efecto de la pérdida de mensajes en el registro histórico.

Cada vez que llega un mensaje, la función de *callback* `ExportarDato()` extrae el valor de temperatura, crea un punto de medición con el nombre `temperature`, agrega el campo `celsius` con el valor correspondiente y envía el registro al bucket definido en InfluxDB. Si la operación se realiza correctamente, el nodo imprime un mensaje informativo en consola confirmando la exportación. Este mecanismo permite mantener una traza continua de la temperatura con resolución temporal de segundos, la cual puede ser posteriormente consultada mediante consultas *Flux* en InfluxDB o visualizada en tiempo real desde Grafana.

El nodo se mantiene en ejecución gracias al ciclo de eventos de ROS 2, permaneciendo operativo mientras el sistema completo se encuentre activo. En la etapa final, se destruye el nodo y se cierra la conexión con el cliente de base de datos, liberando los recursos utilizados. En conjunto, este componente representa el puente entre la infraestructura de publicación-suscripción de ROS 2 y la capa de persistencia de datos, completando el flujo de monitoreo requerido en el reto propuesto.

5) Lanzador: El archivo *t3_qos.launch.py* constituye el componente de orquestación del sistema de-

Algorithm 4 Nodo Exportador en ROS 2 (`/temperature_celsius` → InfluxDB con QoS conmutable)

Procedimiento principal
InicializarROS2()
nodo ← CrearNodo(*exporter_node*)
// Declaración del parámetro de fiabilidad
DeclararParametro(`reliability`, `'reliable'`) {Modos: `reliable` — `best_effort`}
// Lectura de credenciales desde variables de entorno
INFLUX_URL ← ObtenerVariableEntorno(`"INFLUX_URL"`)
INFLUX_TOKEN ← ObtenerVariableEntorno(`"INFLUX_TOKEN"`)
INFLUX_ORG ← ObtenerVariableEntorno(`"INFLUX_ORG"`)
INFLUX_BUCKET ← ObtenerVariableEntorno(`"INFLUX_BUCKET"`)
// Configuración del perfil QoS
rel_str ← ToLower(ObtenerParametro(`reliability`))
qos ← CrearQoS(`depth`=10, `history`=KEEP_LAST)
if rel_str = `'reliable'` **then**
  qos.reliability ← `RELIABLE`
else
  qos.reliability ← `BEST_EFFORT`
end if
// Inicialización del cliente InfluxDB
cliente ← CrearClienteInfluxDB(url=INFLUX_URL, token=INFLUX_TOKEN, org=INFLUX_ORG)
writeAPI ← cliente.CrearWriteAPI()
// Creación de la suscripción al tópico de temperatura
sub ← CrearSuscripcion(`Float32`, `/temperature_celsius`, callback=`ExportarDato`, qos)
Función `ExportarDato`(msg):
  temperatura ← msg.data
  punto ← CrearPuntoInflux(`"temperature"`)
  punto.AgregarCampo(`"celsius"`, temperatura)
  writeAPI.Escribir(bucket=INFLUX_BUCKET, org=INFLUX_ORG, record=punto)
  LogInfo(`'Exportado a InfluxDB: '` + temperatura + `' °C'`)
MantenerNodoActivo(nodo)
DestruirNodo(nodo)
ApagarROS2() =0

Algorithm 5 Archivo de lanzamiento `t3_qos.launch.py` (Orquestación de nodos con parámetro de fiabilidad global)

Procedimiento principal: Generar Descripción de Lanzamiento
Crear descripción de lanzamiento vacía ← *LaunchDescription()*
Declarar argumento `reliability` con valor por defecto `'reliable'`
Añadir el argumento a la descripción de lanzamiento
// Definición de la variable de configuración
rel ← *LaunchConfiguration*(`'reliability'`)
{Almacena el valor proporcionado por el usuario}
// Definición de nodos a ejecutar
sensor ← CrearNodo(*package*=`'sensor_serial'`, *executable*=`'sensor_node'`, *parameters*=[{`'reliability'`: rel}])
processor ← CrearNodo(*package*=`'sensor_serial'`, *executable*=`'processor_node'`, *parameters*=[{`'reliability'`: rel}])
monitor ← CrearNodo(*package*=`'sensor_serial'`, *executable*=`'monitor_node'`, *parameters*=[{`'reliability'`: rel}])
exporter ← CrearNodo(*package*=`'sensor_serial'`, *executable*=`'exporter_node'`, *parameters*=[{`'reliability'`: rel}])
// Agregar nodos a la descripción de lanzamiento
Añadir(sensor)
Añadir(processor)
Añadir(monitor)
Añadir(exporter)
Retornar descripción completa de lanzamiento =0

sarrollado para el reto. Su función principal es permitir la ejecución coordinada de los cuatro nodos que componen el flujo de comunicación (*sensor_node*, *processor_node*, *monitor_node* y *exporter_node*), propagando a todos ellos un mismo parámetro global de fiabilidad denominado `reliability`. Este archivo es fundamental para garantizar que cada experimento o corrida (ya sea con política `RELIABLE` o `BEST_EFFORT`) se ejecute bajo condiciones uniformes, evitando discrepancias entre configuraciones de nodos individuales.

La estructura del archivo sigue el formato estándar de lanzamiento de ROS 2 en Python, basado en la API *launch* y *launch_ros*. En la primera parte, se crea una descripción vacía mediante la clase *LaunchDescription()*, la cual servirá como contenedor de los nodos y argumentos que serán ejecutados. A continuación, se declara el argumento de línea de comandos `reliability`, con un valor por defecto igual a `'reliable'`. Este argumento puede ser sobrescrito al momento de ejecutar el comando de lanzamiento mediante la sintaxis:

```
ros2 launch sensor_serial
t3_qos.launch.py
reliability:=best_effort
```

De esta forma, el usuario puede alternar el comportamiento del sistema sin necesidad de modificar los archivos fuente de los nodos.

Posteriormente, se define una variable de configuración llamada `rel`, la cual utiliza la clase *LaunchConfiguration* para recuperar el valor del argumento en tiempo de ejecución. Este objeto actúa como enlace dinámico entre el valor pasado por el usuario y los parámetros de cada nodo.

La sección siguiente del archivo define los cuatro nodos que compondrán el sistema. Cada nodo se instancia a través de la clase *Node*, indicando el paquete al que pertenece (`sensor_serial`), el ejecutable correspondiente (`sensor_node`, `processor_node`, `monitor_node`, `exporter_node`), el nombre de instancia y la lista de parámetros que debe recibir. En este caso, todos los nodos reciben el mismo parámetro de fiabilidad `reliability`, cuyo valor proviene de la variable `rel`. Esta homogeneidad asegura que tanto publicadores como suscriptores empleen el mismo perfil de QoS, condición necesaria para la compatibilidad y correcta comunicación entre los nodos.

Finalmente, los nodos se agregan a la descripción de lanzamiento y la función `generate_launch_description()` retorna la estructura completa. Al ejecutar el archivo, ROS 2 procesa la descripción y lanza cada nodo de manera paralela dentro del mismo espacio de ejecución, permitiendo que los mensajes fluyan desde la adquisición hasta la exportación a la base de datos. Esta automatización simplifica el control del sistema, evita errores de ejecución manual y facilita la repetición de pruebas bajo condiciones experimentales controladas.

El diseño de este archivo de lanzamiento permite además extender fácilmente el sistema con nuevos nodos o parámetros adicionales. Por ejemplo, se podrían incluir argumentos para el puerto serial, la frecuencia de publicación o la dirección del servidor InfluxDB, manteniendo la centralización de configuraciones en un único punto. En conclusión, *t3_qos.launch.py* representa el núcleo de control del entorno ROS 2 implementado, ya que unifica la configuración, ejecución y experimentación del reto en una única interfaz reproducible y profesional.

B. Puesta en marcha de InfluxDB y Grafana

El archivo *docker-compose.yml* define la infraestructura de servicios base utilizada en el desarrollo del reto,

Algorithm 6 Archivo `docker-compose.yml` (Despliegue de servicios InfluxDB y Grafana en red compartida)

Definir versión del formato Compose
version ← "3.9"
// Declaración de red compartida entre contenedores
CrearRed(`proyecto`, tipo=`bridge`)
// Declaración de volúmenes persistentes
CrearVolumen(`influx-data`)
CrearVolumen(`grafana-data`)
// Servicio InfluxDB
DefinirServicio(`influxdb`)
  image ← `influxdb:2.7`
  ports ← "`8086:8086`"
  environment ← {
   `DOCKER_INFLUXDB_INIT_MODE=setup`,
    `DOCKER_INFLUXDB_INIT_USERNAME=admin`,
    `DOCKER_INFLUXDB_INIT_PASSWORD=admin123`,
   `DOCKER_INFLUXDB_INIT_ORG=wsn`,
    `DOCKER_INFLUXDB_INIT_BUCKET=temperatures`,
    `DOCKER_INFLUXDB_INIT_RETENTION=30d` }
  volumes ← [`influx-data:/var/lib/influxdb2`]
  networks ← [`proyecto`]
// Servicio Grafana
DefinirServicio(`grafana`)
  image ← `grafana/grafana:11.1.0`
  ports ← "`3000:3000`"
  environment ← {
    `GF_SECURITY_ADMIN_USER=admin`,
    `GF_SECURITY_ADMIN_PASSWORD=admin` }
  volumes ← [`grafana-data:/var/lib/grafana`]
  networks ← [`proyecto`]
// Asociación de servicios y recursos
IncluirServicios(`influxdb`, `grafana`)
AsignarVolúmenes(`influx-data`, `grafana-data`)
AsociarRed(`proyecto`)
GuardarArchivo(`docker-compose.yml`) =0

específicamente los contenedores de **InfluxDB 2.x** y **Grafana 11**. Este archivo permite desplegar ambos servicios de forma simultánea y coherente, asegurando que compartan una red virtual interna, volúmenes persistentes para almacenamiento y configuraciones reproducibles, sin necesidad de ejecutar comandos manuales para cada contenedor.

En la primera línea del archivo se declara la versión del formato Compose, en este caso la 3.9, que proporciona compatibilidad con las funcionalidades modernas de Docker. Luego se define una red llamada `proyecto`, configurada bajo el modo `bridge`, que servirá como enlace de comunicación entre los servicios y cualquier contenedor adicional que se conecte, como el de ROS 2. De esta forma, los contenedores podrán comunicarse mediante nombres de servicio (por ejemplo, `influxdb`) sin necesidad de utilizar direcciones IP explícitas.

A continuación se definen dos volúmenes persistentes: `influx-data` y `grafana-data`. Estos volúmenes permiten conservar los datos generados por cada servicio incluso si los contenedores son eliminados o reiniciados. El primero almacena la base de datos y configuración de InfluxDB dentro de la ruta `/var/lib/influxdb2`, mientras que el segundo guarda la configuración y los paneles de Grafana en `/var/lib/grafana`. Este enfoque garantiza la durabilidad de la información y la posibilidad de reanudar el servicio sin pérdida de datos.

El servicio `influxdb` se define utilizando la imagen oficial `influxdb:2.7`, exponiendo el puerto 8086 tanto en el contenedor como en el host. En la sección de variables de entorno se establece un proceso de inicialización automática del entorno de trabajo, que incluye la creación de un usuario administrador (`admin`), su contraseña (`admin123`), la organización principal (`wsn`) y un bucket por defecto llamado `temperatures` con una política de retención de 30 días. Este procedimiento, facilitado por las variables `DOCKER_INFLUXDB_INIT_*`, evita la configuración manual posterior y permite comenzar a escribir datos inmediatamente después del despliegue.

El servicio `grafana` se define con la imagen `grafana/grafana:11.1.0`, y se expone el puerto 3000 para el acceso a la interfaz web de monitoreo. Se configuran las credenciales iniciales de administrador con el usuario `admin` y la contraseña `admin`. Este contenedor utiliza el volumen `grafana-data` para preservar los paneles y configuraciones creados por el usuario, y se conecta a la misma red `proyecto` para poder comunicarse con InfluxDB mediante su nombre de servicio interno. De esta manera, la URL `http://influxdb:8086` utilizada por Grafana se resuelve de forma directa dentro de la red virtual de Docker.

Finalmente, el archivo asocia ambos servicios, volúmenes y la red definida, estableciendo un entorno cohesivo y persistente. El despliegue completo se realiza con un único comando desde el directorio que contiene el archivo:

```
docker compose up -d
```

Este comando descarga las imágenes necesarias, crea los volúmenes y levanta los contenedores en modo desatendido. Una vez finalizado el proceso, InfluxDB queda disponible en la dirección `http://localhost:8086` y Grafana en `http://localhost:3000`, conformando el

subsistema de almacenamiento y visualización de datos requerido para el reto. Este archivo de orquestación constituye así la base de la infraestructura del proyecto, garantizando que los componentes de análisis y monitoreo funcionen de manera reproducible y escalable dentro del entorno Docker.

Tras el arranque se creó un token *All-Access* y el bucket `temperatures` en la organización `wsn`. En Grafana se configuró la fuente de datos con *Flux*:

- *URL*: `http://influxdb:8086`
- *Organization*: `wsn`
- *Token*: (token generado en InfluxDB)
- *Default Bucket*: `temperatures`

Nota: al estar Grafana e InfluxDB en la misma red de Docker, el *hostname* interno es `influxdb` y no `localhost`.

C. Alineación de redes y conectividad

El contenedor ROS 2 se conectó a la red de los servicios para resolver el nombre `influxdb` y evitar errores de DNS. El nombre real de la red en este caso fue `taller3_proyecto`.

```
# Host (PowerShell): conectar el
contenedor ROS a la red del compose
docker network connect
taller3_proyecto ros2_sniff

# Verificación dentro de ros2_sniff
docker exec -it ros2_sniff bash
-lc "getent hosts influxdb; \
curl -sSf http://influxdb:8086/health"
```

La respuesta `"status":"pass"` confirma salud del servicio y conectividad.

D. Variables de entorno para exportación a InfluxDB

Antes de lanzar los nodos, se exportaron las variables de entorno que consume el nodo exportador:

```
# Dentro de ros2_sniff
export INFLUX_URL=http://influxdb:8086
export INFLUX_TOKEN=<TOKEN>
export INFLUX_ORG=wsn
export INFLUX_BUCKET=temperatures
```

Estas variables parametrizan el cliente y evitan credenciales embebidas en código.

E. Ejecución sin QoS ni Grafana

F. Ejecución: corrida 1 (QoS reliable)

Con el entorno cargado de ROS 2 y el paquete instalado, se lanzó el *launch* que orquesta los nodos y propaga el parámetro de QoS. El *sniffer* se ejecutó en paralelo para registrar el tráfico RTPS.

```
# Terminal A (host): iniciar captura
RTPS y guardar .pcap
docker run --rm --network
taller3_proyecto ^
  -v "$PWD\pcap:/pcap" ^
  --cap-add=NET_ADMIN --cap-add=
  NET_RAW ^
  nicolaka/netshoot ^
  tcpdump -i eth0 -w /pcap/reliable_capture.pcap

# Terminal B (ros2_sniff): lanzar
los nodos con QoS reliable
source /opt/ros/jazzy/setup.bash
source /root/ros2_ws/install/setup.bash
ros2 launch sensor_serial
t3_qos.launch.py reliability:=reliable
```

Explicación: `tcpdump` filtra los puertos 7400–7500 (*RTPS* sobre *DDS*). El *launch* inicia los cuatro nodos (adquisición, procesado, monitor y exportación) con fiabilidad garantizada. Se mantuvo la ejecución 60–90 s para acumular datos y tráfico de control (*heartbeat/acknack*).

G. Ejecución: corrida 2 (QoS best_effort)

Se repitió el procedimiento cambiando el perfil de fiabilidad y el nombre de la captura:

```
# Terminal A (host): nueva captura
docker run --rm --network taller3_proyecto ^
  -v "$PWD\pcap:/pcap" ^
  --cap-add=NET_ADMIN --cap-add=NET_RAW ^
  nicolaka/netshoot ^
  tcpdump -i eth0 -w /pcap/best_effort.pcap

# Terminal B (ros2_sniff): lanzar con
QoS best_effort
source /opt/ros/jazzy/setup.bash
source /root/ros2_ws/install/setup.bash
ros2 launch sensor_serial
t3_qos.launch.py reliability:=best_effort
```

Explicación: en *best_effort* los mensajes de control disminuyen; la entrega puede no estar garantizada, lo que simplifica el tráfico y puede reducir sobrecarga a costa de posibles pérdidas.

H. Visualización en Grafana

Se creó un panel *Time series* con consulta *Flux* para el bucket `temperatures` (rango de 10–60 min, unidad en Celsius). La consulta utilizada fue:

```
from(bucket: "temperatures")
  |> range(start: -30m)
  |> filter(fn: (r) => r._measurement
  == "temperature" and r._field == "celsius")
  |> aggregateWindow(every: 1s, fn:
  mean, createEmpty: false)
  |> yield(name: "mean")
```

La figura 1 muestra la grafica en grafana de los valores de temperatura.

Fig. 1. Grafica en grafana

I. Análisis en Wireshark

Los archivos `.pcap` se abrieron en Wireshark aplicando el filtro:

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

En *reliable* se observaron submensajes *DATA*, *HEARTBEAT* y *ACKNACK*, evidenciando control de fiabilidad. En *best_effort* predominan *DATA* y *discovery* (*SPDP/SEDP*) con menor intercambio de confirmaciones.

La imagen 2 muestra la ejecución de nicolaka para la captura de tráfico en la red.

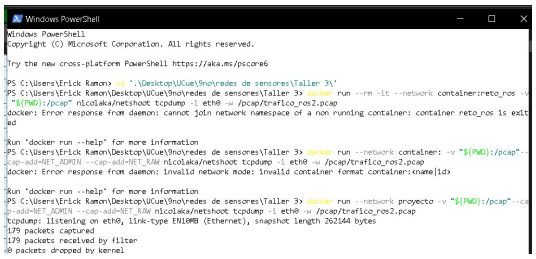

Fig. 2. Ejecución de nicolaka

Las Figuras 3 y 4 muestran extractos de las capturas realizadas con Wireshark al ejecutar el sistema ROS 2 bajo dos políticas de *QoS*: *best_effort* y *reliable*, respectivamente. En ambos casos el filtro empleado fue `udp.port >= 7400 && udp.port <= 7500`, que cubre el rango típico de puertos RTPS/DDS. En las dos trazas se aprecia tráfico con destino multicast `239.255.0.1:7400`, identificado por Wireshark como *Real-Time Publish-Subscribe Wire Protocol* (`RTPS`) con *Protocol version 2.3* y *vendorId 0x015* (eProsima Fast-DDS/Fast-RTPS). En el panel de detalle se observan, para cada datagrama, submensajes `INFO_TS (0x09)` seguidos de `DATA (0x15)`. Esto corresponde al patrón normal de envío de datos de usuario: `INFO_TS` fija la marca temporal y `DATA` transporta el payload del tópico.

En la Figura 3 (*QoS best_effort*) sólo aparecen submensajes `INFO_TS`+`DATA`, lo cual es consistente con la semántica de entrega "mejor esfuerzo": el escritor no emite latidos (`HEARTBEAT`) y el lector no envía confirmaciones (`ACKNACK`); por tanto, no existe tráfico de control asociado a fiabilidad. En consecuencia, la traza muestra un flujo limpio de datos sin mensajes de acuse.

En la Figura 4 (*QoS reliable*) también se aprecia, en el intervalo capturado, únicamente `INFO_TS`+`DATA`. Esto *no* implica que la política *reliable* no esté activa; más bien refleja que, bajo condiciones de canal estable y sin pérdidas, Fast-DDS reduce al mínimo el intercambio de control. Existen tres razones técnicas que explican la ausencia de `HEARTBEAT (0x07)` y `ACKNACK (0x06)` en la traza:

a) (1) Ventana de captura corta frente al periodo de latidos.: Los latidos en Fast-DDS se emiten de forma periódica con un periodo por defecto del orden de varios segundos. Si la captura abarca sólo unos pocos segundos, o si el recorte mostrado en pantalla es breve, es perfectamente posible que ningún `HEARTBEAT` caiga dentro de la ventana temporal registrada. Al prolongar la captura ($\geq$ 60 s) y aplicar el filtro `rtps.submessageid == 0x07 || rtps.submessageid == 0x06` es habitual comenzar a observar dichos submensajes.

b) (2) Ausencia de pérdidas y sesión ya estabilizada.: En una topología local (un escritor y un lector en la misma red/bridge de Docker), sin congestión ni pérdidas, el mecanismo fiable puede operar sin necesidad de retransmisiones ni solicitudes de reparación. En estas condiciones, el lector no genera `ACKNACK` adicionales y el escritor no fuerza `HEARTBEAT` fuera del ritmo nominal. El resultado es una traza dominada por `DATA`, aun cuando la fiabilidad esté garantizada.

c) (3) Direccionamiento y puertos de metatraffic.*:* Parte del *metatraffic* (control) puede circular por unicast entre escritor y lector, o por puertos adyacentes dentro del rango RTPS. Si el explorador se centra visualmente en el multicast `239.255.0.1`, los `ACKNACK` unicast del lector al escritor pueden pasar desapercibidos. Para detectarlos conviene combinar el rango de puertos con filtros de direcciones unicast de los participantes (p. ej., `ip.addr == 172.19.0.4`) y el filtro por submensaje: `rtps.submessageid == 0x06 || rtps.submessageid == 0x07`.

En síntesis, la ausencia de `HEARTBEAT`/`ACKNACK` en las Figuras 3 y 4 se explica por el corto periodo de captura y por un enlace limpio sin pérdidas, no por una configuración incorrecta de *QoS*. La correcta activación de *reliable* puede corroborarse en ROS 2 con `ros2 topic info -v` (mostrando *Reliability: RELIABLE*) y repitiendo la captura durante un intervalo más largo o forzando condiciones que induzcan control (por ejemplo, reiniciando el lector durante la transmisión). En contraste, en *best_effort* no deberían aparecer `HEARTBEAT`/`ACKNACK` en ningún caso, pues dicha política carece de acuses y retransmisiones.

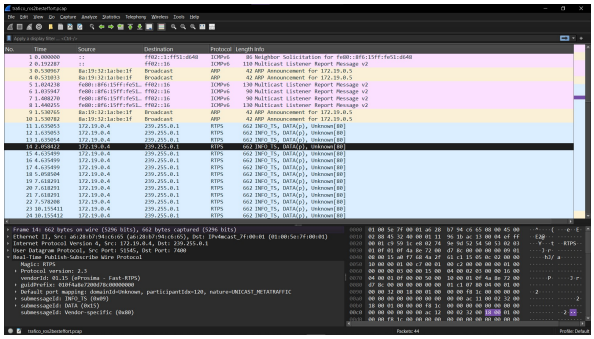

Fig. 3. Tráfico RTPS con *QoS best_effort*: se observan submensajes `INFO_TS` y `DATA` sin presencia de control fiable.

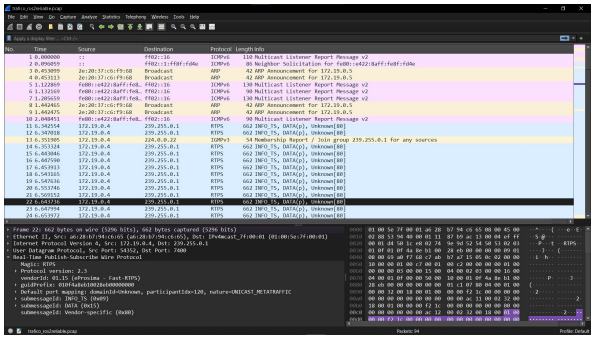

Fig. 4. Tráfico RTPS con *QoS reliable*: en el intervalo capturado se ven `INFO_TS`+`DATA`; la falta de `HEARTBEAT`/`ACKNACK` se atribuye al periodo de latidos y a la ausencia de pérdidas.

Para futuras reproducciones se recomienda: (i) extender la captura a al menos un minuto y aplicar el filtro `rtps.submessageid == 0x06 || rtps.submessageid == 0x07`; (ii) añadir un filtro por direcciones unicast de los participantes; y (iii) introducir un pequeño evento de desincronización (reinicio del lector) para provocar la emisión de `HEARTBEAT`/`ACKNACK`, evidenciando de forma inequívoca el comportamiento fiable del modo *reliable*.

J. Repositorio y Dockerhub

La imagen de este taller se encuentra en `https://hub.docker.com/repository/docker/heliosxs/ros2_sniff/general`.

El repositorio con todos los códigos se encuentra en: `https://github.com/HeliosSm/ROS-en-docker-y-conexi-n-serial-con-arduino`

IV. Conclusiones

En primer lugar, la implementación del entorno contenedorizado con ROS 2, InfluxDB y Grafana demostró ser una solución eficiente y modular para el monitoreo de sistemas distribuidos en tiempo real. El uso de Docker permitió garantizar la portabilidad del sistema, facilitando la replicación del entorno sin dependencia del sistema operativo. La integración entre los nodos de ROS 2 y los servicios de almacenamiento y visualización proporcionó una arquitectura coherente y flexible, capaz de registrar y graficar datos sensoriales de manera continua con mínima intervención del usuario.

En segundo lugar, la comparación entre las políticas de *QoS reliable* y *best_effort* evidenció las diferencias fundamentales entre ambos modos. Mientras que *reliable* prioriza la integridad y entrega garantizada de los mensajes mediante mecanismos de control, el modo *best_effort* ofrece menor latencia y menor tráfico de control a costa de posibles pérdidas de datos. Los resultados obtenidos mediante Wireshark confirmaron que, en entornos locales estables, la ausencia de mensajes `HEARTBEAT` y `ACKNACK` no implica un fallo, sino un comportamiento optimizado del middleware DDS.

En tercer lugar, se comprobó la efectividad de InfluxDB como base de datos de series temporales en aplicaciones de monitoreo de sensores. Su integración con ROS 2 a través del nodo exportador permitió registrar mediciones con precisión temporal y acceder a ellas en tiempo real mediante consultas *Flux*. A su vez, Grafana brindó una interfaz de visualización altamente configurable, que permitió observar las variaciones de temperatura y comparar la estabilidad de la comunicación bajo diferentes configuraciones de QoS. Esta sinergia entre ambas herramientas consolidó un flujo de análisis robusto y eficiente.

Finalmente, la práctica reafirmó el valor de las tecnologías de virtualización y monitoreo en la investigación y docencia en ingeniería de redes de sensores. La estructura modular implementada puede escalarse fácilmente para incorporar nuevos tipos de sensores, mayor cantidad de nodos o algoritmos de procesamiento más complejos. Además, la capacidad de capturar y analizar el tráfico RTPS a nivel de protocolo ofrece un recurso didáctico poderoso para comprender los fundamentos del DDS y evaluar empíricamente el impacto de las políticas de QoS sobre la comunicación distribuida.

References

[1] Open Robotics, “ROS 2 Documentation,” 2024. [En línea]. Disponible en: https://docs.ros.org/en/ros2$_{d}ocumentation/index.html$

[2] Docker Inc., “Docker Documentation,” 2024. [En línea]. Disponible en: https://docs.docker.com/

[3] InfluxData, “InfluxDB Documentation,” 2024. [En línea]. Disponible en: https://docs.influxdata.com/influxdb/v2.7/

[4] Grafana Labs, “Grafana Documentation,” 2024. [En línea]. Disponible en: https://grafana.com/docs/grafana/latest/

[5] Object Management Group (OMG), “Data Distribution Service (DDS) Specification,” Version 1.4, 2020. [En línea]. Disponible en: https://www.omg.org/spec/DDS/1.4/

[6] eProsima, “Fast DDS Documentation,” 2024. [En línea]. Disponible en: https://fast-dds.docs.eprosima.com/

[7] Wireshark Foundation, “Wireshark User’s Guide,” 2024. [En línea]. Disponible en: https://www.wireshark.org/docs/