ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Máster

CARACTERIZACIÓN Y ANÁLISIS DE RENDIMIENTO DEL PROTOCOLO ZENOH PARA SU APLICACIÓN A ENTORNOS IIOT

(Characterization and performance analysis of Zenoh protocol for IIoT scenarios)

Para acceder al Título de

Máster Universitario en Ingeniería de Telecomunicación

Autor: Miguel Barón Herrá

Julio- 2024

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE MÁSTER

Realizado por: Miguel Barón Herrá

Director del TFM: Luis Francisco Diez Fernández, José Ramón Juárez Rodríguez

Título: "Caracterización y análisis de rendimiento del protocolo Zenoh para su aplicación a entornos IIoT"

Title: "Characterization and performance analysis of Zenoh protocol for IIoT scenarios"

Presentado a examen el día: 31 de julio de 2024

para acceder al Título de

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

Composición del Tribunal:

Presidente (Apellidos, Nombre): Muñoz Gutiérrez, Luis

Secretario (Apellidos, Nombre): Valle López, Luis

Vocal (Apellidos, Nombre): García Arranz, Marta

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo: El Presidente

Fdo: El Secretario

Fdo: El Vocal

Fdo: El director del TFM
(sólo si es distinto del Secretario)

VºBº del Subdirector

Trabajo Fin de Máster Nº
(a asignar por la Secretaría)

Agradecimientos

Quisiera aprovechar esta oportunidad para agradecer a mi director de Trabajo de Fin de Máster, Luis Francisco Díez Fernández, por su constante dedicación y apoyo. Me siento muy afortunado de poder contar con él, también, en la etapa que comenzaré el próximo curso.

Asimismo, a Ikerlan, S. Coop., la entidad en la que he tenido el privilegio de volver a desarrollar un proyecto tecnológico con una temática tan actual y con tanta proyección. De manera especial, agradezco a mi tutor, José Ramón Juárez, por la confianza depositada en mí y por, de nuevo, hacerme sentir allí como en casa.

Querría destacar la altísima calidad humana y profesional de los demás miembros de las distintas áreas de Ikerlan, así como del Grupo de Ingeniería Telemática del Departamento de Ingeniería de Comunicaciones de la Universidad de Cantabria. En particular, agradezco a Mihail Zverev, Fátima Fernández y al resto del equipo de Konektibitate Adimenduna (KOA); al Catedrático Ramón Agüero Calvo y a los distintos miembros del Laboratorio de Redes y Servicios, por su inestimable ayuda y por hacer mis días mucho más amenos, tanto en Mondragón como en Santander.

Por último, pero no menos importante, me gustaría expresar mi más sincero agradecimiento a mi familia y amigos, por su constante apoyo e infinita paciencia, con quienes me siento muy afortunado de poder contar. Sin las fuerzas y seguridad que me han dado, no habría podido llegar hasta aquí.

Resumen

La conectividad y la automatización se han convertido en aspectos esenciales dentro de la Industria 4.0. La profunda evolución del Internet Industrial de las Cosas (IIoT, Industrial Internet of Things) y de los protocolos Internet de las Cosas (IoT, Internet of Things) han facilitado el intercambio de datos en tiempo real y la aparición de nuevos paradigmas de comunicación como Vehicle-to-everything (V2X) y Robot-to-Anything (R2X) han revolucionado la interacción entre vehículos, infraestructura y robots, mejorando la seguridad y eficiencia. En este contexto surge el protocolo Zenoh, destacando por su flexibilidad y eficiencia, siendo adecuado para una amplia gama de aplicaciones en distintas topologías de red.

Este Trabajo profundiza en los fundamentos y en el funcionamiento de Zenoh a nivel de red. Se analiza su rendimiento en entornos IIoT mediante pruebas con dispositivos Raspberry Pi, comparándolo con Message Queuing Telemetry Transport (MQTT). El análisis se enfoca en la latencia, la fiabilidad de los datos y el manejo de la congestión, evaluando Zenoh con los protocolos de capa de transporte Transmission Control Protocol (TCP) y User Datagram Protocol (UDP).

Palabras clave: Control de congestión; IIoT ; Fiabilidad; Gossip Scouting; Middleware; MQTT; Multicast Scouting; UDP; Zenoh

Abstract

Connectivity and automation have become essential aspects of Industry 4.0. The profound evolution of IIoT and IoT protocols has facilitated real-time data exchange, and the emergence of new communication paradigms such as V2X and R2X has revolutionized the interaction between vehicles, infrastructure, and robots, enhancing safety and efficiency. In this context, the Zenoh protocol emerges, notable for its flexibility and efficiency, making it suitable for a wide range of applications across various network topologies.

This work delves into the fundamentals and operation of Zenoh at the network level. Its performance in IIoT environments is analyzed through tests with Raspberry Pi devices, comparing it with MQTT. The analysis focuses on latency, data reliability, and congestion management, evaluating Zenoh with both TCP and UDP transport layer protocols.

Keywords: Congestion control; IIoT; Gossip Scouting; Middleware; MQTT; Multicast Scouting; Reliability; UDP; Zenoh

Índice general

Índice de figuras

Índice de tablas

Índice de acrónimos

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks.
ACK	Acknowledgment.
API	Application Programming Interface.
CoAP	Constrained Application Protocol.
C-V2X	Cellular V2X.
DDS	Data Distribution Service.
eMBB	enhanced Mobile BroadBand.
FL	federated learning.
GEO	Geostationary Earth Orbit.
HLC	Hybrid Logical Clock.
HTTP	Hypertext Transfer Protocol.
IA	Inteligencia Artificial.
IIoT	Industrial Internet of Things.
IoT	Internet of Things.
IP	Internet Protocol.
ITU	International Telecommunication Union.
KE	Key Expression.
LEO	Low Earth Orbit.

M2M	Machine-to-Machine.
mMTC	massive Machine Type Communications.
MPI	Message Passing Interface.
MQTT	Message Queuing Telemetry Transport.
MTU	Maximum Transmission Unit.
NI	Network Intelligence.
NTN	Non-Terrestrial Networks.
NTP	Network Time Protocol.
P2P	peer-to-peer.
PtP	Point-to-Point.
Pub/Sub	Publish/Subscribe.
QoS	Quality of Service.
R2X	Robot-to-Anything.
REST	Representational State Transfer.
ROS	Robot Operating System.
RTT	Round-trip Time.
tc	Traffic Control.
TCP	Transmission Control Protocol.
TLS	Transport Layer Security.
UDP	User Datagram Protocol.
URI	Uniform Resource Identifier.
URL	Uniform Resource Locator.
URLLC	Ultra-Reliable and Low Latency Communications.
UUID	Universally Unique Identifier.
V2X	Vehicle-to-everything.

CAPÍTULO 1

Introducción

En la era de la Industria 4.0, los procesos de manufactura e industriales han experimentado profundas transformaciones, evidenciando la importancia de aspectos como la conectividad, el intercambio de datos y la automatización [1]. En este contexto, la Industria 4.0 incorpora sistemas de manufactura inteligentes, donde las máquinas están interconectadas a través de la denominada IIoT, facilitando el intercambio de datos en tiempo real y la toma de decisiones informadas. La llegada del 5G ha potenciado aún más estas capacidades con mejoras significativas en términos de fiabilidad, latencia y velocidad de transmisión de datos [2], permitiendo aplicaciones como comunicaciones ultra confiables y de baja latencia (URLLC, Ultra-Reliable and Low Latency Communications), banda ancha móvil mejorada (eMBB, enhanced Mobile BroadBand) y comunicaciones masivas de tipo máquina (mMTC, massive Machine Type Communications) [3], totalmente alineadas con las aplicaciones de la IIoT propias de la Industria 4.0 [4], [5]. Este cambio de paradigma se sustenta en marcos de comunicación robustos que garantizan una transferencia de datos confiable y eficiente.

Uno de los sectores que ha seguido una evolución más significativa es el sector automotriz, especialmente con el continuo desarrollo de la conducción asistida y autónoma. Aquí, la comunicación V2X desempeña un papel crucial, al permitir que los vehículos puedan comunicarse entre sí y con la infraestructura subyacente. En este marco, una tecnología a tener en cuenta es Cellular V2X (C-V2X), que utiliza la infraestructura de red celular desplegada (4G o 5G) para mejorar la seguridad y eficiencia en las carreteras [6]; así como el uso de las redes no terrestres (NTN, Non-Terrestrial Networks) con satélites de órbita geoestacionaria (GEO, Geostationary Earth Orbit) o de órbita baja (LEO, Low Earth Orbit), ofreciendo cobertura global y capacidades mejoradas de conectividad [7]. En la Figura 1.1 se muestra una abstracción de un posible escenario de V2X, con comunicaciones C-V2X y cobertura de redes NTN.

De manera similar, la robótica ha experimentado avances significativos impulsados por la convergencia de la Inteligencia Artificial (IA), la computación en la nube (*Cloud*), y en el borde de la red (*Edge*), y la conectividad avanzada. Estos desarrollos han permitido la automatización y la optimización de procesos en una amplia gama de aplicaciones industriales y de servicios [8]. Por ejemplo, en entornos de fabricación como el mostrado en la Figura 1.2, los robots colaborativos han revolucionado la línea de producción al trabajar de manera segura junto con los operadores humanos, aumentando notablemente la eficiencia [9]. En este contexto, el entorno de desarrollo Robot Operating System (ROS) [10] ha emergido como un marco estándar para el desarrollo de aplicaciones robóticas [11], requiriendo soluciones middleware confiables que faciliten la comunicación adecuada entre diversas plataformas robóticas.

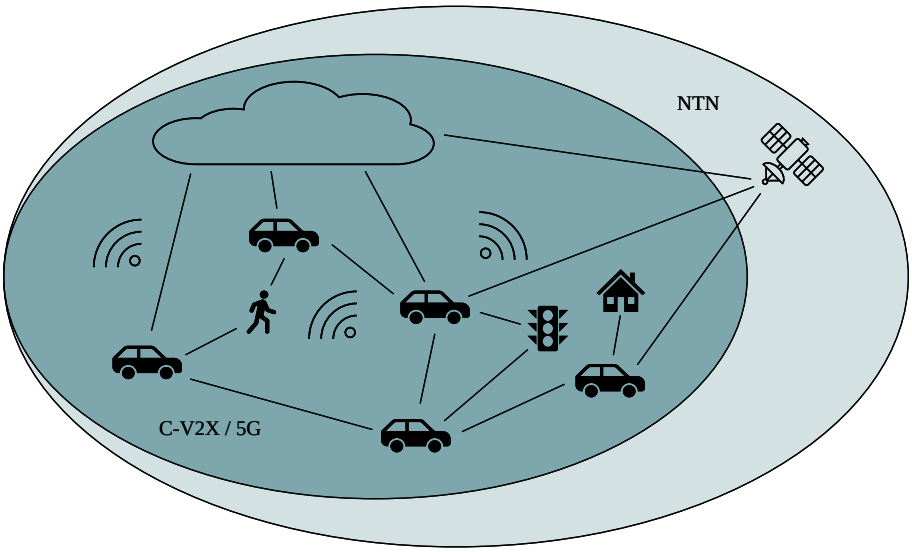


Figura 1.1: Comunicaciones V2X con cobertura NTN. Elaboración propia.

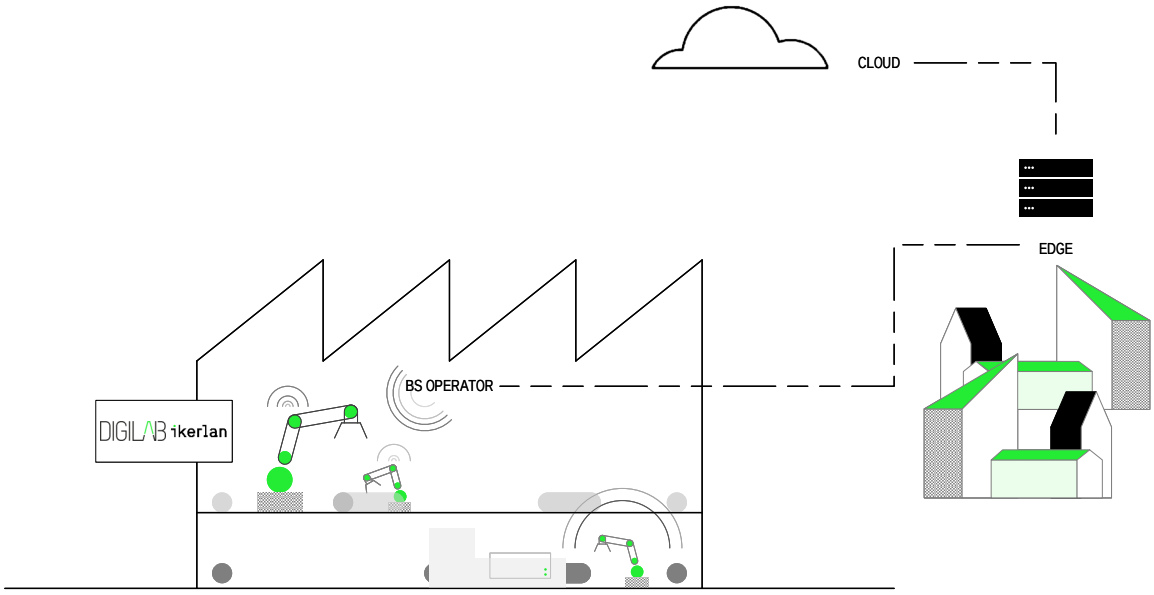


Figura 1.2: Ejemplo de un entorno de IIoT con comunicación R2X. Elaboración propia.

De un tiempo a esta parte, Zenoh [12] ha ganado reconocimiento como una arquitectura de protocolo muy prometedora, tanto en los sectores de robótica, como en el automotriz, gracias a su versatilidad. En robótica, Zenoh ha sido adoptado especialmente para soportar la comunicación R2X sobre redes inalámbricas [13]. Se ha convertido en el primer protocolo no-Data Distribution Service (DDS) [14] en ser soportado de manera nativa en ROS 2 [15], abordando el problema del *walled garden* [12], donde los protocolos propietarios o con menos compatibilidades limitan la interoperabilidad. Esto ha generado un aumento en las soluciones comerciales desplegadas basadas en Zenoh en el área de la robótica.

En la industria automotriz, Zenoh está cobrando gran relevancia debido a su idoneidad para las comunicaciones tanto dentro del vehículo como V2X [13], [16]. Su flexibilidad, que le permite operar en una amplia gama de plataformas, desde microcontroladores hasta centros de datos [16], lo convierte en una opción muy atractiva para aplicaciones V2X. Recientemente, Zenoh ha sido identificado por la Unión Internacional de Telecomunicaciones (ITU, International Telecommunication Union) como el protocolo más adecuado para estas aplicaciones, considerando su desempeño y escalabilidad [17].

1.1. Objetivos

Este trabajo propone realizar un análisis exhaustivo del rendimiento de Zenoh en diferentes topologías de red, utilizando un entorno de prueba real formado por dispositivos Raspberry Pi. Se comparará el rendimiento de Zenoh con el de MQTT [18], [19], con el objetivo de evaluar las posibles ventajas ofrecidas por Zenoh. Además, aunque el enfoque principal del análisis comparativo será el comportamiento de Zenoh sobre TCP [20] como capa de transporte, el estudio también explorará la capacidad de Zenoh para garantizar la fiabilidad de los datos y manejar escenarios potenciales de congestión de red mediante experimentos con el protocolo de transporte UDP [21]. El estudio profundizará en las capacidades de comunicación de Zenoh. Las principales contribuciones de este trabajo son:

- Un examen detallado de los principios subyacentes de Zenoh en lo referente a comunicaciones.
- Un análisis comparativo entre Zenoh y MQTT en términos de latencia, utilizando diversas topologías de red.
- Una evaluación de los mecanismos de fiabilidad y control de congestión de Zenoh a través de transmisiones, tanto por TCP, como por UDP.

1.2. Estructura

Seguidamente, se detalla la estructura del trabajo que se desarrollará a lo largo del documento, organizado en cinco capítulos. El Capítulo 1 está dedicado a la introducción y los objetivos del proyecto. En el Capítulo 2 se examina el estado del arte de los protocolos de comunicación, destacando las principales características de MQTT y Zenoh y revisando los estudios previos relacionados con Zenoh que preceden a este trabajo.

El Capítulo 3 profundiza en los principios fundamentales de Zenoh, ofreciendo una descripción detallada del protocolo. Posteriormente, en el Capítulo 4, se analiza el protocolo desde una perspectiva de comunicaciones, explicando su funcionamiento en detalle.

En el Capítulo 5 se lleva a cabo un análisis exhaustivo del rendimiento de Zenoh en comparación con MQTT en diferentes escenarios, además de examinar los mecanismos de fiabilidad y control de congestión que ofrece Zenoh sobre los protocolos de capa de transporte TCP y UDP. Finalmente, en el Capítulo 6 se presentan las conclusiones del estudio y se proponen futuras líneas de investigación derivadas del trabajo realizado.

CAPÍTULO 2

Estado del arte

En los últimos años, la investigación en el campo de los protocolos de comunicación para la IIoT ha experimentado un notable crecimiento. Protocolos como DDS, Constrained Application Protocol (CoAP) [22] o Apache Kafka [23] han sido objeto de análisis exhaustivos dada su gran relevancia en entornos industriales [24], [25]. Entre ellos, MQTT surge como una de las opciones principales. Reconocido por su diseño ligero y eficiente, su modelo flexible de Publish/Subscribe (Pub/Sub), su capacidad de conexión persistente y su soporte para calidad de servicio (QoS, Quality of Service), MQTT ha sido ampliamente adoptado en diversas industrias [26]. No obstante, recientemente, Zenoh [12] ha emergido como un competidor relevante, con características únicas y aplicaciones con gran potencial dentro de los entornos de la IIoT.

En este capítulo se describen los protocolos analizados a lo largo del trabajo, proporcionando una descripción detallada de los principios fundamentales de MQTT y presentando el estado del arte de Zenoh, dentro del ámbito investigador, destacando sus innovaciones y su relevancia en el campo de las telecomunicaciones.

2.1. MQTT

MQTT [18], [19] es un protocolo de comunicación ligero específicamente diseñado para aplicaciones de la IoT y, por tanto, ampliamente adoptado por sensores con capacidades reducidas y para aplicaciones que requieren tasas limitadas. Es conocido por su reducido tamaño de código y su facilidad de integración, lo que lo convierte en una opción muy conveniente para entornos dentro del paradigma de la IoT y el Machine-to-Machine (M2M).

Como se muestra en la Figura 2.1, MQTT utiliza un modelo de comunicación punto a punto (PtP, Point-to-Point), basado en el patrón de publicación/suscripción (Pub/Sub), donde los clientes publican mensajes en temas específicos (*topics*) a los que otros clientes pueden suscribirse para recibir dicha información. Un componente clave en la arquitectura de MQTT es el agente (*broker*), que actúa como una entidad intermediaria y es responsable de recibir, almacenar y reenviar los mensajes publicados a los clientes suscritos.

MQTT ofrece tres niveles de QoS para garantizar la entrega confiable de mensajes:

- QoS 0: Entrega como máximo una vez. El mensaje se envía sin acuse de recibo y puede llegar al

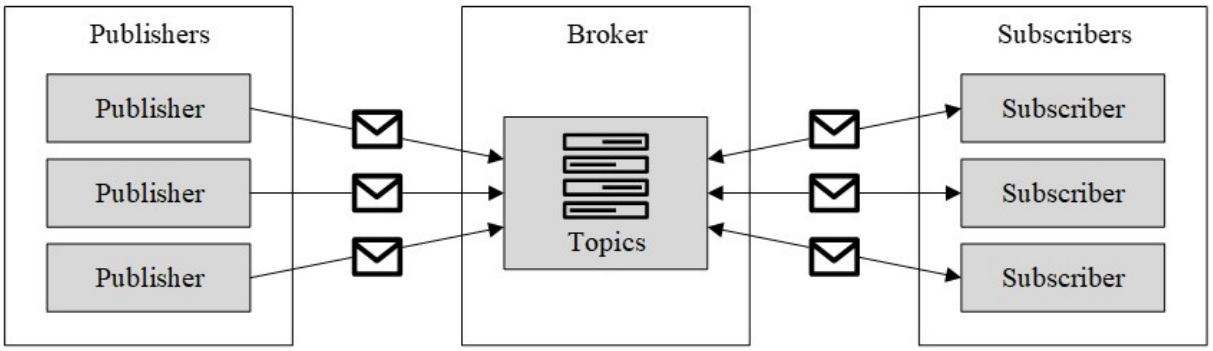


Figura 2.1: Modelo de comunicaciones de MQTT [27].

destino una vez o no llegar.

- QoS 1: Entrega al menos una vez. El mensaje se reenvía hasta que el receptor confirme su recepción exitosa mediante un paquete `PUBACK`.
- QoS 2: Entrega exactamente una vez. Se establece un proceso de intercambio de cuatro pasos (`PUBLISH`, `PUBREC`, `PUBREL`, `PUBCOMP`) entre el remitente y el receptor para garantizar la entrega única del mensaje, evitando duplicaciones.

En la Figura 2.2 se ilustra una transmisión típica en MQTT, destacando las conexiones con diferentes calidades de servicio en ambos extremos de la comunicación. Específicamente, se presentan conexiones con QoS 1 y QoS 2, mostrándose cómo varían las garantías de entrega de mensajes en función del nivel de calidad de servicio seleccionado.

La implementación típica de MQTT utiliza TCP en la capa de transporte, sobre Internet Protocol (IP) [28]. Además, la seguridad puede proporcionarse mediante el protocolo Transport Layer Security (TLS) [29]. El puerto estándar para MQTT es el 1883, aunque el puerto 8883 está reservado para conexiones MQTT con TLS [18], [19]. No obstante, recientemente se han propuesto implementaciones alternativas para aprovechar soluciones de transporte novedosas. Por ejemplo, en [30], MQTT se implementa sobre QUIC [31] para cumplir mejor con ciertos requisitos de rendimiento, como la latencia.

2.2. Zenoh

Zenoh [12] es un proyecto desarrollado por ZettaScale Technology[1] que surge como respuesta a la creciente necesidad de una infraestructura de comunicación unificada y eficiente en el ámbito de la IoT y los sistemas ciberfísicos. A medida que estos sistemas se vuelven más complejos e interconectados, enfrentan desafíos significativos en términos de interoperabilidad, latencia, escalabilidad y manejo de datos en tiempo real.

A diferencia de MQTT, Zenoh opera como un protocolo de Pub/Sub/Query, ofreciendo una solución más completa que incluye, no solo la transmisión de datos en tiempo real, sino también la gestión de datos en reposo mediante el uso de almacenamiento (*storages*), junto con la capacidad de ejecutar consultas (*queries*) directamente dentro del sistema. Esto proporciona una flexibilidad y escalabilidad significativas,

[1] https://www.zettascale.tech/ (accedido: 26 de junio de 2024).

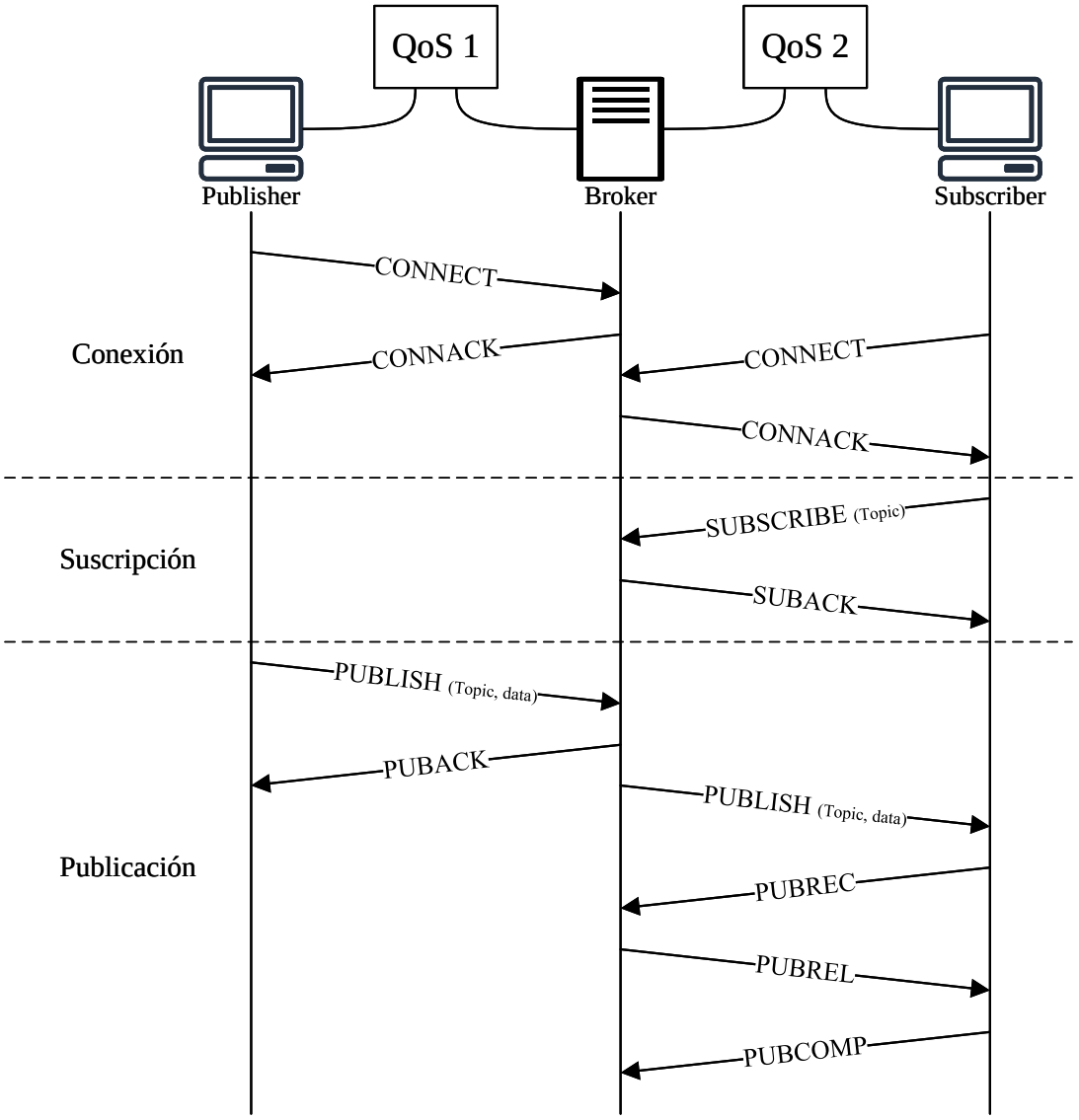


Figura 2.2: Sesión MQTT con calidades de servicio 1 y 2. Elaboración propia. Basado en [27].

simplificando el desarrollo de aplicaciones distribuidas y reduciendo la complejidad de la infraestructura. Además, Zenoh es conocido por su alta eficiencia, con una sobrecarga de solamente 5 B [13], lo que lo hace particularmente atractivo para integrarse sobre diversas capas del *stack* de protocolos.

Zenoh típicamente opera como un protocolo de aplicación sobre TCP. Sin embargo, tiene la capacidad única de poder operar sobre múltiples capas del *stack* de protocolos, incluyendo redes IP y no IP. Zenoh puede funcionar por encima de la capa de Enlace de Datos, de Red o de Transporte, gracias a su diseño que incorpora el Protocolo de Sesión Zenoh. Éste proporciona abstracciones para canales no confiables (*Best Effort*) y confiables (*Reliable*) con diversas prioridades y tiene una unidad de transmisión máxima (MTU, Maximum Transmission Unit) ilimitada. Por ejemplo, al extender el soporte a conexiones serie, Zenoh abre nuevas posibilidades para dispositivos que carecen de interfaces de red tradicionales como Wi-Fi, Bluetooth o Ethernet. Esta versatilidad es particularmente relevante en sectores como la robótica, el transporte y los ámbitos marítimos, agrícolas e industriales, donde las tecnologías de red convencionales a

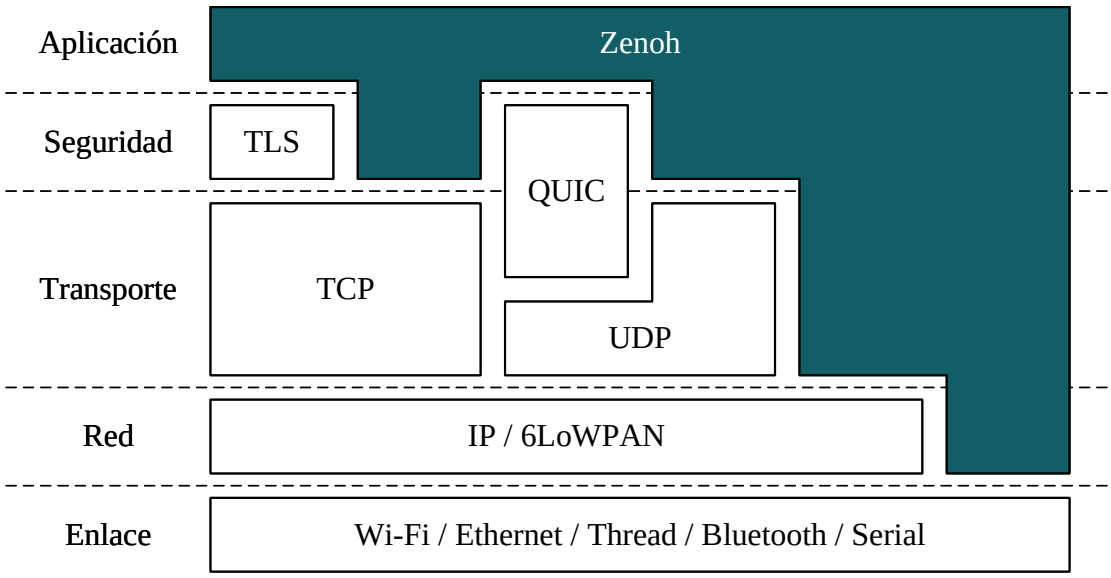


Figura 2.3: Posicionamiento de Zenoh en el *stack* de protocolos. Elaboración propia. Basado en [32].

menudo son insuficientes [32].

Como se ilustra en la Figura 2.3, Zenoh tiene la flexibilidad de operar sobre la capa de Transporte utilizando protocolos como TCP (con o sin TLS), UDP o QUIC; sobre la capa de Red, utilizando tanto IP como IPv6 over Low power Wireless Personal Area Networks (6LoWPAN); o directamente sobre la capa de Enlace de Datos: Wi-Fi, Ethernet, Thread, Bluetooth o enlace serie.

2.2.1. Antecedentes

Dada su reciente aparición y que se trata de una solución propietaria, los trabajos relacionados con Zenoh son bastante escasos hasta la fecha. Sin embargo, existen algunos estudios, principalmente realizados por los desarrolladores y mantenedores de Zenoh, como ZettaScale [13], [33], y ADLINK [34], que proporcionan la base de este trabajo.

El amplio abanico de topologías que ofrece Zenoh permite comparaciones en comunicaciones peer-to-peer (P2P) con protocolos distribuidos como Cyclone DDS o en comunicaciones que impliquen el uso de intermediarios, con protocolos con un enfoque centralizado, como MQTT o Kafka. En este contexto, Liang *et al.* proporcionan en [33] un análisis comparativo de estos protocolos, arrojando resultados favorables para Zenoh en términos de latencia y tasa de transmisión efectiva. En cuanto a la latencia, se estudiaron escenarios tanto con un solo *host* como con múltiples *hosts*. En el primer caso, se observó que con DDS se obtenían latencias más bajas que con Zenoh P2P. Los autores concluyeron que esto se debía a que DDS utiliza mecanismos de multidifusión (*multicast*) UDP. Dado que Zenoh aún no estaba implementado sobre UDP, se optó por probar también la implementación de Zenoh para dispositivos restringidos, Zenoh-Pico[2], que sí estaba implementada sobre UDP, proporcionando las mejores latencias. Además, Zenoh con intermediario (*brokered mode*) obtuvo mejores resultados en comparación con MQTT. En el escenario de múltiples *hosts*, que comprende tres *hosts* (publicador, suscriptor e intermediario) y utilizando un

[2]https://github.com/eclipse-zenoh/zenoh-pico (accedido: 7 de junio de 2024).

Ethernet de 100 Gbps, Zenoh superó consistentemente a MQTT y Cyclone DDS, demostrando valores significativamente más bajos en todos los escenarios analizados.

Corsaro *et al.* hacen referencia a estos resultados en [13], donde presentan los principios y características clave de Zenoh, como los recursos, las entidades o las primitivas, y muestran las topologías soportadas. El artículo también refleja la eficiencia de Zenoh en comparación con DDS y MQTT debido a su sobrecarga (*overhead*) mínima, significativamente menor que la de sus alternativas.

Shih *et al.* [34] centran su estudio en evaluar la capacidad de Zenoh en sistemas en tiempo real, escalables y que abarquen diferentes redes. Demuestran que Zenoh y Cyclone DDS exhiben comportamientos diferentes dependiendo del tamaño de la carga útil. Los resultados evidencian que Zenoh muestra un mejor rendimiento con cargas útiles más pequeñas, pero se ve afectado a medida que aumenta el tamaño de la carga útil. El análisis se realizó con los pares (*peers*) en una sola máquina, simulando las condiciones de red que dispositivos en el *Edge* podrían encontrar en Wi-Fi 802.11g [35], utilizando el núcleo de Linux para limitar la capacidad de la red.

Recientemente, se han llevado a cabo algunas otras investigaciones para explorar el potencial que Zenoh podría tener en diferentes casos de uso. Este es el caso de Zhang *et al.*, quienes en [36] estudian el rendimiento de Cyclone DDS, MQTT y Zenoh en sistemas distribuidos ROS 2. Se establecen tres entornos de red: dos redes locales, Ethernet y Wi-Fi, simulando comunicaciones *Edge-to-Edge*; y una configuración 4G con una red virtualizada ZeroTier [37] para emular la comunicación *Edge-to-Cloud*. En estos trabajos se logran buenos resultados para Zenoh cuando opera bajo condiciones 4G y Wi-Fi, en comparación con MQTT y DDS; sin embargo, sobre Ethernet, Cyclone DDS exhibe una latencia mínima y una tasa de transferencia máxima. Los autores, al igual que en [33] y [13], concluyen que esto es debido a la comparación injusta con DDS debido a la utilización de UDP como capa de transporte.

López Escobar *et al.* [38], por su parte, han observado un potencial significativo en la utilización de Zenoh para implementar servicios ligeros dentro de la infraestructura *Cloud-to-Edge*, lo que permite varios casos de uso como rescate de emergencia o aplicaciones de monitorización de salud a distancia. En este trabajo examinan el rendimiento de los enfoques de comunicación centralizados y descentralizados, haciendo uso de MQTT como una estrategia centralizada y de Zenoh P2P como una descentralizada. Se definen tres entornos de pruebas (*testbeds*) para representar varios escenarios de comunicación en el *Cloud-to-Edge Continuum* [3]: *Mist*, *Fog-Mist* y *Cloud-Fog-Mist*. Estos escenarios involucran computación distribuida a través de diferentes capas dependiendo de los requisitos de procesamiento y red. La evaluación se lleva a cabo sobre una infraestructura de acceso Wi-Fi (802.11ac) y 5G (núcleo Stand Alone (SA) con un gNodeB de banda n78 para interiores (3,3–3,8 GHz)). Tanto MQTT como Zenoh se comunican sobre TCP, con MQTT configurado en el nivel 0 de QoS y Zenoh en modo *Best Effort*.

Recientemente, Teixeira *et al.* han explorado en [39] el uso de Zenoh como una alternativa al Message Passing Interface (MPI) para tareas de aprendizaje distribuido en el contexto de aprendizaje federado (FL, federated learning) dentro de redes 6G. El artículo compara Zenoh con MPI en escenarios realistas de FL, utilizando entornos simulados. En el trabajo se demuestra que Zenoh exhibe un tiempo de ejecución mínimamente superior, pero destaca su reducida sobrecarga de comunicación y su mayor flexibilidad, lo que lo convierte en una solución adecuada para dispositivos con recursos limitados y conectividad dinámica en aplicaciones de FL.

[3] Paradigma donde los recursos y servicios se extienden desde la nube hasta los dispositivos en el borde de la red.

Tabla 2.1: Aspectos cubiertos por trabajos relacionados. Se utilizan los términos “BE” y “Rel”. para indicar si se analizan configuraciones de *Best Effort* y *Reliable*, respectivamente.

Trabajos	Rel., CC / QoS	Múltiples hosts	Conectividad inalámbrica	Topologías Zenoh	Zenoh sobre UDP
Este trabajo	BE/Rel., Block/Drop / QoS 0, 1, 2	✓	✓	P2P, Brokered, Routed	✓
[13]	BE, Block / QoS 0	✓	✗	P2P, Brokered	✗
[33]	BE, Block / QoS 0	✓	✗	P2P, Brokered	✗
[34]	BE, Drop / Volatile	✗	Simulado	P2P	✗
[36]	?	✓	✓	P2P, Brokered	✗
[38]	BE / QoS 0	✓	✓	P2P	✗
[39]	–	✗	Simulado	–	✗
[40], [41]	–	–	–	–	–

Cabe destacar el diferente enfoque presentado en [40], donde se introduce por primera vez *ZenohFlow*: una plataforma diseñada para simplificar el desarrollo de aplicaciones para robots autónomos, vehículos y otros casos de uso que requieren transferencia de datos entre la nube y los dispositivos, utilizando Zenoh. Este entorno proporciona herramientas y abstracciones para mejorar el rendimiento y la eficiencia en aplicaciones de control, al tiempo que soporta flujos de datos de nivel superior para inteligencia artificial y aprendizaje automático. *ZenohFlow* se desarrolla en base a los resultados positivos obtenidos en [41]. Además, otros investigadores exploran esta plataforma, como Gramaglia *et al.* [42], quienes la utilizan para implementar algoritmos heterogéneos de inteligencia de red (NI, Network Intelligence)[4].

En relación al estado del arte actual, este Trabajo complementa la investigación existente al explorar más a fondo el rendimiento comparativo de MQTT y Zenoh a través de topologías de red aún no analizadas, como la comunicación enrutada, donde dos routers pueden reenviar datos directamente entre ellos, eliminando la necesidad de que un cliente sirva como puente. Se propone investigar el rendimiento de Zenoh con conectividad inalámbrica, para proporcionar una evaluación más completa y precisa de su rendimiento y efectividad en situaciones más cercanas a las que podrían encontrarse en entornos industriales reales [44].

Además, se profundizará en los mecanismos de control de congestión y fiabilidad descritos en [13], centrándose en las conexiones utilizando Zenoh sobre los protocolos TCP y UDP, y analizando la pérdida de paquetes en una conexión completamente cableada. Además, se prestará especial atención al tráfico TCP y UDP generado durante las transmisiones, para analizar el comportamiento de Zenoh a nivel de comunicaciones.

La Tabla 2.1 muestra diversas características abordadas tanto por este trabajo como por los trabajos relacionados mencionados.

[4]Monitoreo que emplea datos, algoritmos y técnicas avanzadas para analizar en tiempo real la información de la red y los paquetes de datos IP mientras circulan por ella. Esto facilita la detección de patrones, la identificación de anomalías y la optimización del rendimiento y la seguridad de la red [43].

CAPÍTULO 3

Fundamentos de Zenoh

Este capítulo proporciona una visión integral de los fundamentos de Zenoh[1]. Se inicia con una descripción de la organización de la red, seguida de una explicación detallada de las principales abstracciones utilizadas en Zenoh. También se presentan ciertas extensiones que pueden ser añadidas y, finalmente, se analizan los mecanismos de fiabilidad y control de congestión disponibles en Zenoh.

El manual de referencia proporcionado en [12] y el documento desarrollado por Corsaro *et al.* [13] han servido como base bibliográfica para la documentación realizada en este capítulo.

3.1. Organización de la red

Una de las particularidades más significativas de Zenoh es la amplia gama de opciones que ofrece para la organización de la red. Zenoh permite definir el rol que debe desempeñar cada nodo de la red, lo que posibilita la implementación de diversas topologías, desde configuraciones centralizadas hasta arquitecturas completamente distribuidas.

Las aplicaciones de Zenoh pueden configurarse para operar como pares (*peer*) o clientes (*client*), mediante el despliegue de enrutadores Zenoh (*Zenoh router*, *zenohd*[2]), dependiendo de los requisitos específicos del sistema. La elección entre estos roles depende de diversos factores como la arquitectura de la red, las necesidades de comunicación y la carga de trabajo esperada.

Operar como pares es idóneo para situaciones donde se requiere comunicación descentralizada y directa entre dispositivos. Los pares establecen conexiones P2P, eliminando la necesidad de un intermediario y permitiendo una mayor flexibilidad y eficiencia en la transmisión de datos. Este tipo de comunicación directa reduce la latencia, ya que los datos no necesitan pasar por un enrutador central. Esto es crucial en aplicaciones sensibles al tiempo, como en sistemas de control industrial o vehículos autónomos. Además, en una red de pares, el fallo de un nodo no afecta significativamente a la comunicación global, ya que los demás pares pueden seguir comunicándose entre sí, mejorando así la resiliencia y robustez de la red.

Las comunicaciones P2P implican establecer sesiones con múltiples pares y mantener un estado para cada una de esas sesiones. Mantener dichos estados puede ser indeseable por razones de escalabilidad o porque la aplicación se ejecuta en un dispositivo con recursos limitados. En este caso, la aplicación

[1] Se hace referencia a Zenoh v0.10.0-rc. Características y mejoras de versiones posteriores no se tendrán en cuenta.

[2] Implementación específica del router Zenoh que actúa como un demonio (*daemon*) de Zenoh, ejecutándose como un servicio en segundo plano, que optimiza el despliegue y la gestión de la red Zenoh.

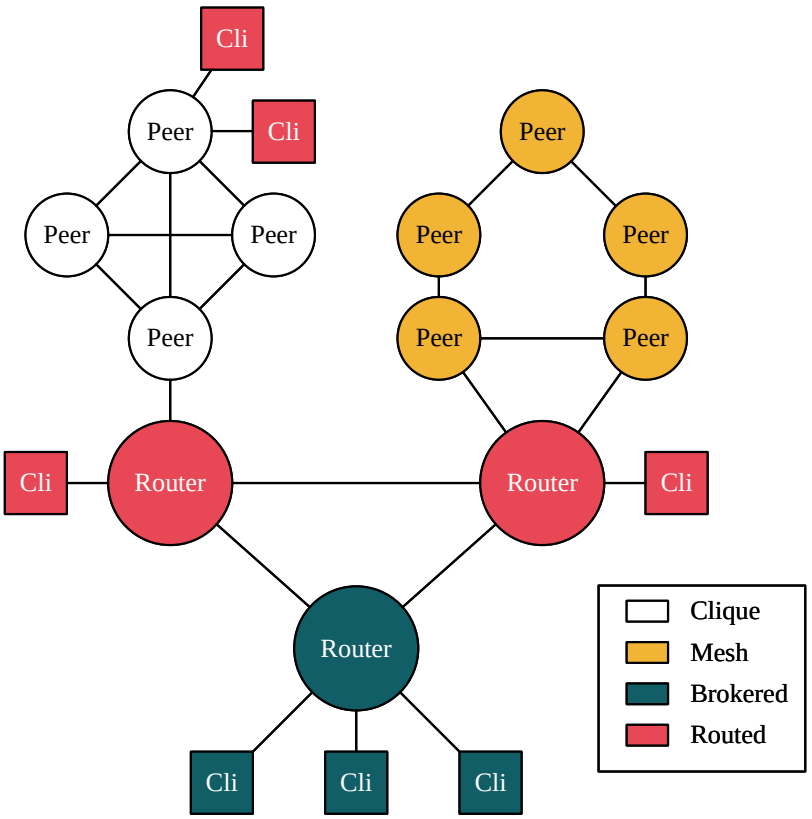


Figura 3.1: Posibles topologías con Zenoh. Se utiliza el término “Cli” para referirse a las aplicaciones Zenoh en modo cliente. Elaboración propia. Basado en [12].

Zenoh puede configurarse para operar en modo cliente, de manera que mantenga, en cualquier momento, una única sesión con otro proceso (generalmente un router Zenoh) que le otorgará conectividad con el resto del sistema. Por tanto, elegir operar como cliente es adecuado en escenarios donde se requiere una comunicación centralizada y controlada y para dispositivos con recursos limitados que pueden delegar la complejidad del enrutamiento y la gestión de la red al router Zenoh, optimizando así el uso de sus recursos.

El router Zenoh no solo facilita la comunicación entre clientes, sino que también soporta la carga de extensiones (*plugins*) que amplían sus capacidades. Estas incluyen una interfaz de programación de aplicaciones (API, Application Programming Interface) de transferencia de estado representacional (REST, Representational State Transfer), que permite el acceso y control de los datos a través de interfaces web, facilitando la integración con aplicaciones basadas en la nube y sistemas externos. Además, proporciona conectividad a diversas bases de datos, permitiendo almacenar y recuperar datos de manera eficiente. Asimismo, soporta la interoperabilidad con protocolos como DDS y MQTT, permitiendo a Zenoh actuar como un puente entre diferentes tecnologías de comunicación. Esto, además, simplifica el proceso de migración a Zenoh en escenarios donde ya haya otras aplicaciones IoT desplegadas.

Como ilustra la Figura 3.1, las aplicaciones Zenoh pueden configurarse para operar en diferentes topologías, adaptándose a los requisitos específicos de la red y la aplicación. Estas topologías incluyen las configuraciones P2P, *Brokered* y *Routed*, cada una con sus propias ventajas y aplicaciones.

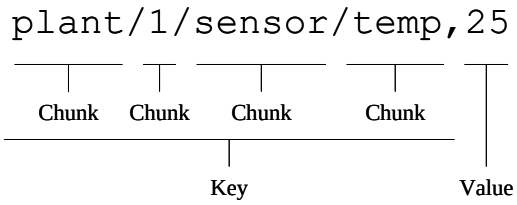


Figura 3.2: Abstracciones en Zenoh: recursos. Elaboración propia.

En la topología P2P, existen dos configuraciones principales: *Clique* y *Mesh*. En la configuración *Clique*, todos los pares establecen comunicación directa entre sí, creando una red plenamente interconectada. Esto es ideal para aplicaciones que requieren una comunicación rápida y directa entre todos los nodos. Por otro lado, la configuración *Mesh* permite una comunicación P2P más selectiva, ejecutando un protocolo de estado de enlace (*linkstate*), donde los nodos se conectan solo con ciertos pares basándose en criterios definidos, lo que puede mejorar la eficiencia y reducir la sobrecarga de la red.

La topología *Brokered* introduce un enrutador Zenoh que actúa como intermediario, similar a un *broker*, facilitando la comunicación entre clientes. Esta visión más centralizada simplifica el proceso de comunicación y aporta una mayor eficiencia [45], [46].

Finalmente, la topología *Routed* ofrece un enfoque más dinámico y escalable. En esta configuración, cada nodo de la red tiene la capacidad de comunicarse con otros nodos a través de enrutadores interconectados, combinando las ventajas de las topologías *Clique*, *Mesh* y *Brokered*. Este enfoque descentralizado, ideal para redes extensas y complejas en el contexto de la Industria 4.0, asegura una conectividad robusta y flexible [45].

3.2. Abstracciones

Para optimizar su funcionamiento, Zenoh se apoya en diversas abstracciones que facilitan la gestión y organización eficiente de los datos. Se exponen, seguidamente, los principios fundamentales de algunas de estas abstracciones, tales como los recursos, las expresiones de clave (KEs, Key Expressions), los selectores, las entidades, el almacenamiento, las marcas de tiempo y el espacio de administración.

3.2.1. Recursos

La comunicación en Zenoh se basa en la transmisión y recepción de valores (*values*) asociados con distintas claves (*keys*). Los recursos en Zenoh se definen como pares "(clave,valor)". La clave puede estar constituida por uno o varios segmentos individuales (*chunks*), separados jerárquicamente por "/", similar a una ruta en un sistema de archivos, tal y como indica la Figura 3.2.

3.2.2. Expresiones de clave

A la hora de obtener datos de la red, en lugar de utilizar una clave específica, se puede hacer uso de las denominadas KEs. Se trata de un patrón que permite especificar un conjunto de claves, mediante

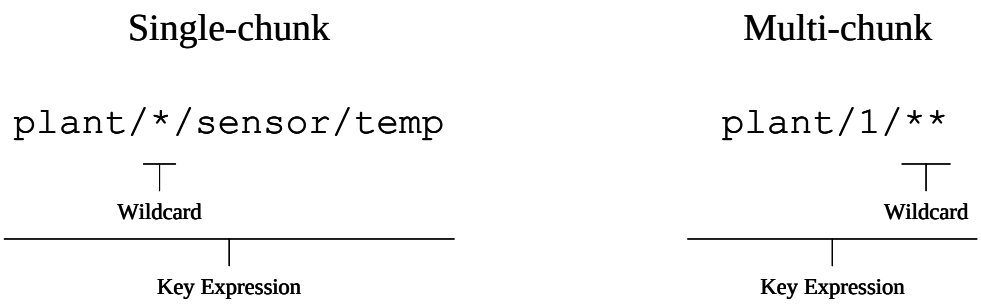


Figura 3.3: Abstracciones en Zenoh: expresiones de clave (KEs, Key Expressions). Elaboración propia.

comodines (*wildcards*) y otras técnicas para generalizar el acceso a los datos.

Como muestra la Figura 3.3, hay dos tipos principales de comodines: (1) comodines uninivel (*single-chunk*, "`*`"), que representan cualquier valor de un solo nivel en la jerarquía y deben ir precedidos y seguidos por "`/`", y (2) comodines multinivel (*multi-chunk*, "`**`"), que representan cualquier valor de múltiples niveles en la jerarquía, incluyendo la posibilidad de que no haya niveles adicionales, y que deben ir seguidos por "`/`".

Por ejemplo, "`plant/*/sensor/temp`" coincidirá con las claves "`plant/1/sensor/temp`" o "`plant/2/sensor/temp`", mientras que la KE "`plant/1/**`" será coincidente con las claves "`plant/1/sensor/temp`", "`plant/1/CO2`" o "`plant/1`".

Existe también otro comodín, "`$*`", con la misma utilidad que "`*`", pero que puede ir rodeado de otros caracteres: "`plant$*/sensor/temp`" coincidirá con las claves "`plant1/sensor/temp`" o "`plant2/sensor/temp`". No obstante, este comodín es más lento computacionalmente que "`*`", por lo que su uso debe ser evitado. Se considera, por tanto, buena práctica utilizar "`plant/1`" en lugar de "`plant1`".

3.2.3. Selectores

Un selector es otra de las herramientas con las que opera Zenoh. Es una combinación de una KE y un conjunto de parámetros que permite realizar búsquedas y filtrados complejos y específicos, proporcionando flexibilidad y potencia en la gestión de datos en sistemas distribuido. La KE será la parte del selector que los enrutadores tendrán en cuenta para el encaminamiento del mensaje Zenoh, mientras que los parámetros son pares clave-valor que pueden especificar argumentos para consultas remotas, aplicar filtros sobre los valores o filtrar por metadatos como marcas de tiempo.

Los selectores se representan, como se muestra en la Figura 3.4, en una forma similar a un identificador de recursos uniforme (URI, Uniform Resource Identifier), donde la KE aparece antes del primer "`?`" y, a continuación, se indican los parámetros, que se codifican como la sección de consulta de un localizador de recursos uniforme (URL, Uniform Resource Locator). De esta manera los pares clave-valor se separan con "`&`" y la clave y el valor se separan con "=". Zenoh estandariza ciertas claves, como "`_time`" para filtrar por rangos de tiempo, o para evitar conflictos y promover la coherencia en las operaciones de consulta.

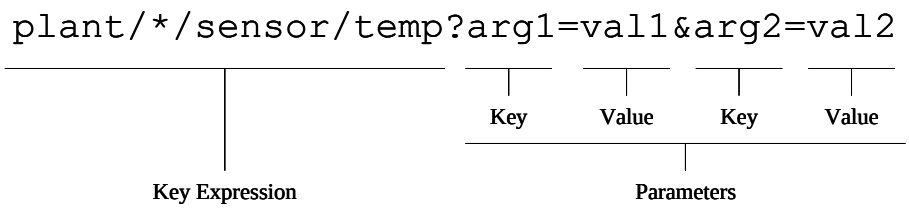


Figura 3.4: Abstracciones en Zenoh: selectores. Elaboración propia.

Por ejemplo, con "`plant/*/sensor/temp?_time=[<date1>..<date2>]`" se pueden obtener datos de temperatura dentro de un rango de tiempo específico, facilitando operaciones basadas en criterios temporales que conlleven una mayor precisión.

3.2.4. Entidades

Como se ha indicado en la Sección 2.2, Zenoh se basa en el modelo Pub/Sub/Query, operando no solo con el patrón Pub/Sub utilizado típicamente en escenarios IoT con protocolos de mensajería como MQTT [18, 19] o DDS [14], sino también ofreciendo la posibilidad de realizar peticiones (*queries*) de datos concretos, con un modelo petición-respuesta similar al de otros protocolos tales como CoAP[22] o protocolo de transferencia de hipertexto (HTTP, Hypertext Transfer Protocol) [47-49].

En este contexto, en una sesión Zenoh habrá, generalmente, diferentes entidades encargadas del intercambio de los mensajes. Se definen un total de tres entidades: (1) publicador (*publisher*), (2) suscriptor (*subscriber*) y (3) consultable (*queryable*).

El publicador[3] es el responsable de enviar datos al sistema (`put()`), poniéndolos a disposición de otros nodos de la red mediante una clave específica. El suscriptor, por su parte, se suscribe a una KE para recibir todos los datos asociados con claves coincidentes. Cabe destacar que existen dos modelos de suscriptores, suscriptores y suscriptores de modo *Pull*, con las siguientes funcionalidades:

- Suscriptor: recibe notificaciones en tiempo real de cada actualización (*put*) en las KEs especificadas, y procesa estas notificaciones inmediatamente mediante una función de llamada de retorno (*callback*) o una cola (*queue*).
- Suscriptor de modo *Pull*: recibe notificaciones de actualizaciones (*put*) en las KEs especificadas, pero, en lugar de recibir los datos automáticamente al ser notificado, los extrae bajo demanda cuando se realiza una solicitud explícita (`pull()`).

Por otro lado, se denomina consultable a aquella entidad que puede responder a consultas (*queries*) sobre datos. A diferencia de los publicadores, los consultables no envían datos de manera continua, sino que

[3]La declaración de una entidad como publicador en Zenoh es una técnica de optimización que mejora la eficiencia del sistema, especialmente en escenarios donde la transmisión de datos es frecuente. No obstante, Zenoh permite a cualquier aplicación realizar operaciones de escritura (`put()`) sin necesidad de ser declarado explícitamente como publicador. Esto proporciona flexibilidad y simplicidad en el manejo de datos dentro de un entorno distribuido, facilitando la integración y el desarrollo de aplicaciones que necesitan escribir datos de manera eficiente y directa.

responden (`reply()`) a solicitudes específicas de información bajo demanda, a partir de una función de llamada de retorno o una cola. Estas solicitudes puntuales las puede realizar cualquier aplicación Zenoh, sin necesidad de estar suscrito al contenido, realizando una petición (`get()`).

En el Apéndice A, se muestran tres ejemplos de aplicaciones Zenoh para cada una de las entidades abordadas. Se han realizado siguiendo el esquema aportado en los ejemplos para la API de Python de Zenoh v0.10.0-rc[4,5].

3.2.5. Almacenamiento

Otro aspecto diferencial de Zenoh es la capacidad que ofrece de sumar al modelo Pub/Sub/Query la posibilidad de tener un sistema de almacenamiento. En Zenoh, los nodos de almacenamiento (*storages*) son componentes clave que permiten la persistencia de datos, asegurando que estos puedan ser recuperados posteriormente, incluso si el publicador original ya no está disponible. Esta funcionalidad es crucial para aplicaciones que requieren acceso a históricos de datos, recuperación tras fallos, o persistencia a largo plazo.

Los *storages* en Zenoh tienen la capacidad de actuar tanto como suscriptores como con el rol consultables. Estos nodos se suscriben a KEs específicas y, al recibir publicaciones que coinciden con sus suscripciones, almacenan los valores correspondientes. Posteriormente, cuando se realiza una consulta utilizando un selector que coincide con sus suscripciones, los *storages* devuelven los valores más recientes para cada clave coincidente.

En la Sección 3.3, se abordará otro concepto clave, referente al almacenamiento en Zenoh: el *plugin Storage Manager*.

3.2.6. Marcas temporales

Cuando se introduce un valor en Zenoh, el primer router Zenoh que lo recibe asigna automáticamente una marca temporal (*timestamp)*. Esta marca está compuesta por dos elementos:

- Un tiempo generado por un Reloj Híbrido Lógico (HLC, Hybrid Logical Clock)[50] de 64 bits, similar en estructura a un *timestamp* Network Time Protocol (NTP) pero con un *epoch* diferente: los primeros 32 bits representan los segundos desde el 1 de enero de 1970 UTC y los últimos 32 bits, una fracción de segundo con un contador para asegurar que cada *timestamp* sea único.

- El identificador único universal (UUID, Universally Unique Identifier) del router Zenoh que generó esta marca temporal.

Esta combinación garantiza que cada valor en Zenoh tenga una marca temporal exclusiva, permitiendo que los valores se ordenen de manera coherente en todo el sistema sin requerir algoritmos de consenso externos.

[4] https://github.com/eclipse-zenoh/zenoh-python (accedido: 27 de junio de 2024), versión 0.10.0-rc.
[5] https://zenoh-python.readthedocs.io/en/0.10.0-rc/ (accedido: 27 de junio de 2024).

3.2.7. Espacio de administración

Zenoh ofrece la posibilidad de gestionar el router Zenoh en el denominado "espacio de administración" (*admin space*). Es accesible a través de las operaciones básicas de escritura y lectura de Zenoh (`get()`/`put()`), utilizando el prefijo `@/router/<router-id>`, donde `<router-id>` es el UUID del router Zenoh.

Por ejemplo, para la obtención de la información del estado del router podrá utilizarse la clave `@/router/<router-id>`, de solo lectura. Por otro lado, para editar la configuración del router en tiempo de ejecución, podrá usarse la clave `@/router/<router-id>/config/**`, de solo escritura.

3.3. Extensiones

Una característica que posee Zenoh, que le permite ampliar sus capacidades y le aporta una mayor flexibilidad, es la posibilidad que tienen los routers Zenoh de soportar la carga de extensiones (*plugins*), tanto al inicio como en tiempo de ejecución, siempre que se tenga permiso de escritura configurado en su espacio de administración.

3.3.1. REST

El *plugin* REST (*zplugin_rest*) proporciona el acceso a la API REST de Zenoh. Para ello, habilita un servidor HTTP en el nodo Zenoh en que se ha cargado la extensión.

El *plugin* puede iniciarse mediante el argumento de inicio `-rest-http-port=[PORT | IP:PORT | none]` de *zenohd*, que por defecto usa el puerto 8000. Alternativamente, se puede configurar en el archivo de configuración del router.

3.3.2. Storage Manager

Como se mencionaba en la Sección 3.2.5, el almacenamiento es una parte fundamental para la gestión de los datos en Zenoh. El *plugin Storage Manager* (*zplugin_storage_manager*) facilita el almacenamiento de valores asociados a diferentes claves a través de diversos *backends* que se cargan dinámicamente. Estos *backends* suelen emplear tecnologías como bases de datos, actuando también como interfaces entre la infraestructura de Zenoh y sistemas externos.

Además, es posible cargar múltiples instancias del mismo *backend* con configuraciones distintas, conocidas como *volúmenes*, que pueden ser utilizadas por varios *storages*, de un modo similar a cómo se gestionan múltiples archivos en un volumen de sistema de archivos. Los *backends* disponibles abarcan desde almacenamiento en bases de datos como *S3*[6], *InfluxDB*[7] y *RocksDB*[8], hasta el almacenamiento en el sistema de archivos local (*File System*[9]).

[6] https://github.com/eclipse-zenoh/zenoh-backend-s3 (accedido: 19 de julio de 2024).
[7] https://github.com/eclipse-zenoh/zenoh-backend-influxdb (accedido: 19 de julio de 2024).
[8] https://github.com/eclipse-zenoh/zenoh-backend-rocksdb (accedido: 19 de julio de 2024).
[9] https://github.com/eclipse-zenoh/zenoh-backend-filesystem (accedido: 19 de julio de 2024).

3.4. Mecanismos de fiabilidad y control de congestión

En los sistemas distribuidos, asegurar tanto la fiabilidad (*reliability*) como un control eficiente de la congestión (*congestion control*) es crucial para mantener comunicaciones robustas. Zenoh ofrece un marco integral para gestionar estos dos aspectos de manera efectiva.

Zenoh proporciona varias estrategias de fiabilidad para adaptarse a diferentes requisitos de aplicaciones y servicios, manteniendo al mismo tiempo la escalabilidad: (1) fiabilidad salto a salto, (2) extremo a extremo y (3) del primer al último router.

- Fiabilidad salto a salto (*hop to hop*). Esta estrategia, predeterminada en Zenoh, asegura la fiabilidad en cada salto de la red. Aunque es altamente escalable, puede resultar en pérdida de datos durante cambios en la topología.
- Fiabilidad extremo a extremo (*End to End*). Establece un canal de fiabilidad entre cada par productor-consumidor de datos, minimizando las pérdidas incluso durante cambios en la topología. Sin embargo, es menos escalable y puede consumir más recursos.
- Fiabilidad del primer al último router (*First Router to Last Router*). Al establecer un canal de fiabilidad entre el primer y el último router de cada ruta de datos, Zenoh reduce la carga sobre productores y consumidores, mejorando la escalabilidad.

Estas estrategias de fiabilidad en Zenoh se basan en dos capas de Zenoh: (1) el protocolo de sesión, que establece y gestiona conexiones bidireccionales entre los nodos de la red de Zenoh, maneja la configuración y el mantenimiento de canales de comunicación, incluyendo canales de esfuerzo mejorado y confiables, y optimiza el uso de la red mediante técnicas como el agrupamiento automático y la fragmentación; y (2) el protocolo de enrutamiento, que utiliza el protocolo de sesión para propagar y enrutar datos desde los productores hasta los consumidores en la red de Zenoh, estableciendo la ruta más eficiente para la transmisión y asegurando una entrega confiable. Este protocolo es crucial para dirigir de manera efectiva los flujos de datos a través de la red, independientemente de la estrategia de fiabilidad elegida.

3.4.1. Fiabilidad

En Zenoh, la fiabilidad es gestionada por los consumidores de datos, quienes especifican si requieren que le lleguen todas las publicaciones (`RELIABLE()`) o si tirar paquetes es aceptable (`BEST_EFFORT()`).

3.4.2. Control de congestión

Por otro lado, los productores son quienes definen el control de congestión que se aplicará durante la sesión en caso de congestión significativa en la red. Deciden, para cada envío, si permiten descartar el paquete (`DROP()`) o bloquear al publicador (`BLOCK()`), priorizando la entrega de mensajes sobre otras operaciones. Esta estrategia se propaga a todos los nodos involucrados y se implementa en toda la ruta.

Tanto los mecanismos de control de congestión como los de fiabilidad se detallan en la Tabla 3.1.

Tabla 3.1: Mecanismos de fiabilidad y control de congestión en Zenoh.

Mecanismo	Valor	Definición	Decisión
Control de Congestión	`BLOCK()`	Garantiza que los mensajes no se pierdan bajo ninguna circunstancia.	Productores
	`DROP()`	Permite descartar el mensaje si todos los buffers están llenos.	
Fiabilidad	`RELIABLE()`	Informa a la red que se requiere entregar todas las publicaciones de manera confiable.	Consumidores
	`BEST_EFFORT()`	Informa a la red que es aceptable descartar algunos mensajes.	

CAPÍTULO 4

Comunicaciones en Zenoh

Este capítulo ofrece una descripción detallada del funcionamiento del protocolo de comunicaciones de Zenoh, centrándose en los mecanismos y procedimientos implementados durante las sesiones. Inicialmente, se explica cómo se realiza el descubrimiento de nodos en la red, seguido de la apertura de puertos en Zenoh. Posteriormente, se describen los diferentes pasos de una sesión Zenoh, especificando los distintos paquetes de control disponibles. Finalmente, se presenta un intercambio de datos completo a modo de ejemplo para ilustrar el proceso.

Para la realización de este capítulo, se ha llevado a cabo un análisis exhaustivo del tráfico de Zenoh y de los protocolos de capas subyacentes que desempeñan un papel relevante en la comunicación, utilizando el disector de Zenoh para Wireshark 4.0.10[1].

4.1. Descubrimiento de nodos

Zenoh permite configurar directamente, desde el archivo de configuración, los puntos de conexión (*endpoints*) de los diferentes nodos de la red, pudiendo especificar tanto los *endpoints* a los que conectarse como aquellos en los que escuchar. Esta funcionalidad es apropiada en situaciones donde se requiere una configuración específica y controlada de la red, así como en aplicaciones estáticas, donde se desea evitar la sobrecarga de la red.

Sin embargo, Zenoh también ofrece la opción de descubrir automáticamente los nodos de la red (*scouting*), lo cual es muy útil en entornos dinámicos donde se necesita una rápida integración de nuevos dispositivos. Un ejemplo de caso de uso del descubrimiento automático son las comunicaciones V2X, donde los vehículos pueden conectarse con diferentes dispositivos a medida que se desplazan.

4.1.1. Multicast Scouting

En Zenoh, generalmente, el descubrimiento de nodos de la red se basa en mensajes *scout* enviados sobre UDP. Cualquier nodo que busque unirse a la red debe suscribirse al grupo *multicast* 224.0.0.224 en el puerto UDP 7446[2], con una trama en la que indique su rol en la red, tal y como se indica en la Tabla 4.1

[1] https://github.com/ZettaScaleLabs/zenoh-dissector (accedido: 13 de mayo de 2024).

[2] Tanto la dirección *multicast* como el puerto son configurables.

Tabla 4.1: *Payload* UDP de la trama de *scout* en que el nodo indica su rol al grupo *multicast*.

Rol	Bytes	Bits
Cliente	010801	0001
Par	010803	0011
Router Zenoh	010807	0111

Se detallan, seguidamente, los mecanismos de *scouting* que, dependiendo del rol del nodo, tienen algunas particularidades:

Pares y enrutadores Zenoh

Cuando un par se une a la red, continúa enviando mensajes de descubrimiento al grupo *multicast* en intervalos de tiempo que siguen una progresión geométrica: 1, 2, 4, 8 segundos; manteniéndose constante a partir del cuarto término. En este contexto, los enrutadores Zenoh se comportan de manera similar a los pares, anunciando su rol en el mensaje de *scout*, como se muestra en la Figura 4.1a.

Cada vez que uno de estos nodos no-cliente se une a la red, los pares y/o routers Zenoh ya existentes transmiten su dirección IP y puerto (en adelante, "Puerto Servidor") donde permanecen escuchando a comunicaciones entrantes (TCP, UDP, QUIC, etc.). En lo posterior se utilizará la notación "@IP:Port" para referirse a la dupla dirección IP y puerto. Como se muestra en la Figura 4.1b, esta transmisión sigue la progresión mencionada anteriormente. Posteriormente, el nodo no-cliente conectado envía su @IP:Port cada 8 segundos a cada nodo con el que se haya conectado previamente. De este modo, cada par y router Zenoh en la red conoce Puerto Servidor de todos los demás participantes.

Clientes

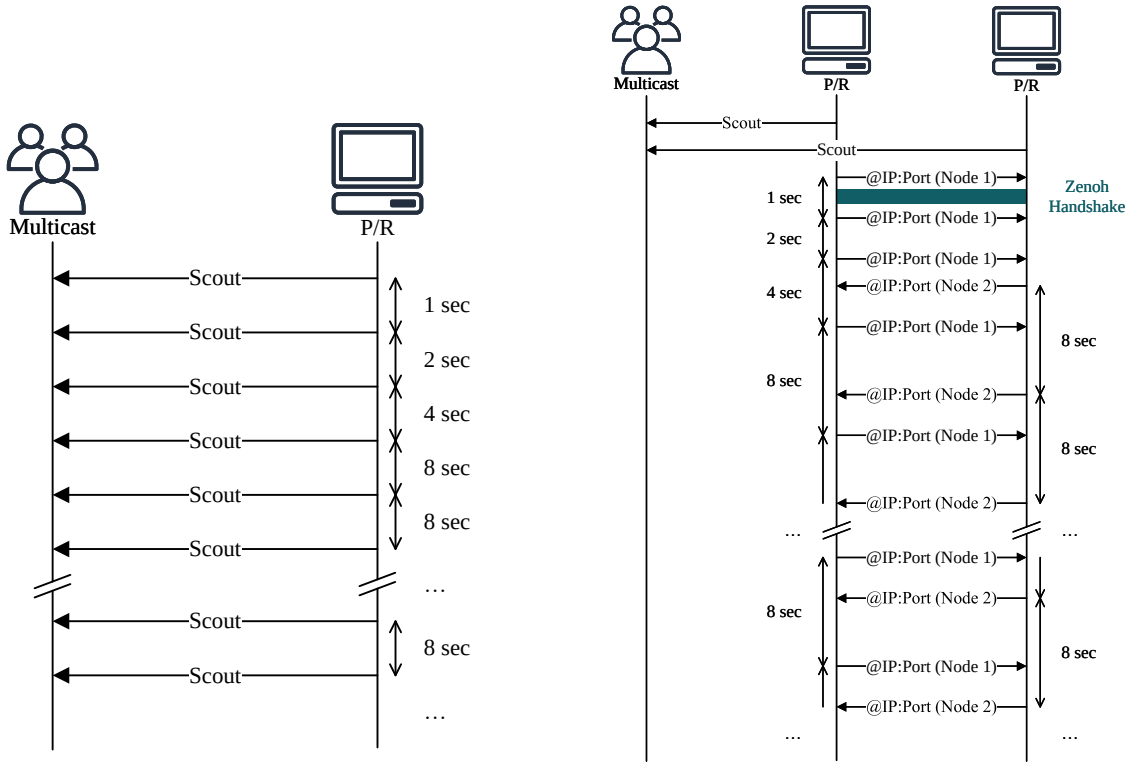
Los clientes, por su parte, envían un único mensaje de *scout* al grupo *multicast*. Un router Zenoh responderá enviando su @IP:Port (generalmente, 7447) para establecer la conexión. No se envían ni se reciben más tramas UDP por parte de los clientes. Esto está representado en la Figura 4.2.

4.1.2. Gossip Scouting

Diseñado específicamente para escenarios P2P donde las comunicaciones *multicast* no estén disponibles, Zenoh introduce el descubrimiento *gossip*, permitiendo a los pares descubrirse a nivel de aplicación propagando los @IP:Port de todos los pares descubiertos a lo largo de la red. Sin embargo, para que un par comience el proceso de *gossip scouting*, debe haber establecido previamente una conexión con otro nodo en la red. Por lo tanto, es esencial que los nodos estén conectados a un punto de entrada inicial, típicamente un router Zenoh, para descubrir al resto de la red.

Al tratarse de un mecanismo que forma parte del protocolo de sesión de Zenoh, se hará alusión a tipos de paquetes propios de Zenoh, como `OAM`, que serán explicados, posteriormente, en la Sección 4.3 y referidos en la Tabla 4.2.

A diferencia del descubrimiento *multicast*, la propagación de los pares @IP:Port no ocurre periódicamente. Cada vez que un nuevo nodo entra en la red, el nodo al que se conecta envía su tabla de direcciones IP y asignaciones de Puertos Servidor de los nodos conocidos dentro de un paquete `Frame` (`OAM`) al nodo recién descubierto. Luego, propaga el par @IP:Port de este último al siguiente salto, encapsulado también



(a) Mensaje de descubrimiento inicial.

(b) Transmisión de @IP:Port.

Figura 4.1: Proceso de descubrimiento y comunicación de los pares o enrutadores con *multicast scouting* en Zenoh. Se utilizan los términos “P” y “R” para referirse a pares y routers Zenoh, respectivamente. Elaboración propia.

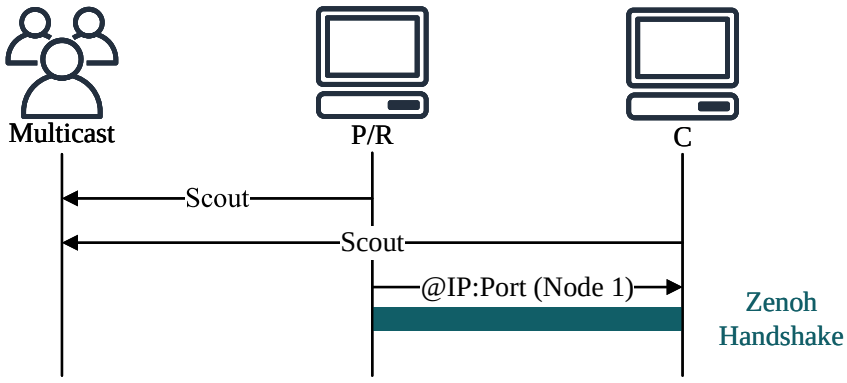


Figura 4.2: Proceso de conexión de descubrimiento y comunicación de los clientes con *multicast scouting* en Zenoh. Se utilizan los términos “P”, “R” y “C” para referirse a pares, routers Zenoh y clientes, respectivamente. Elaboración propia.

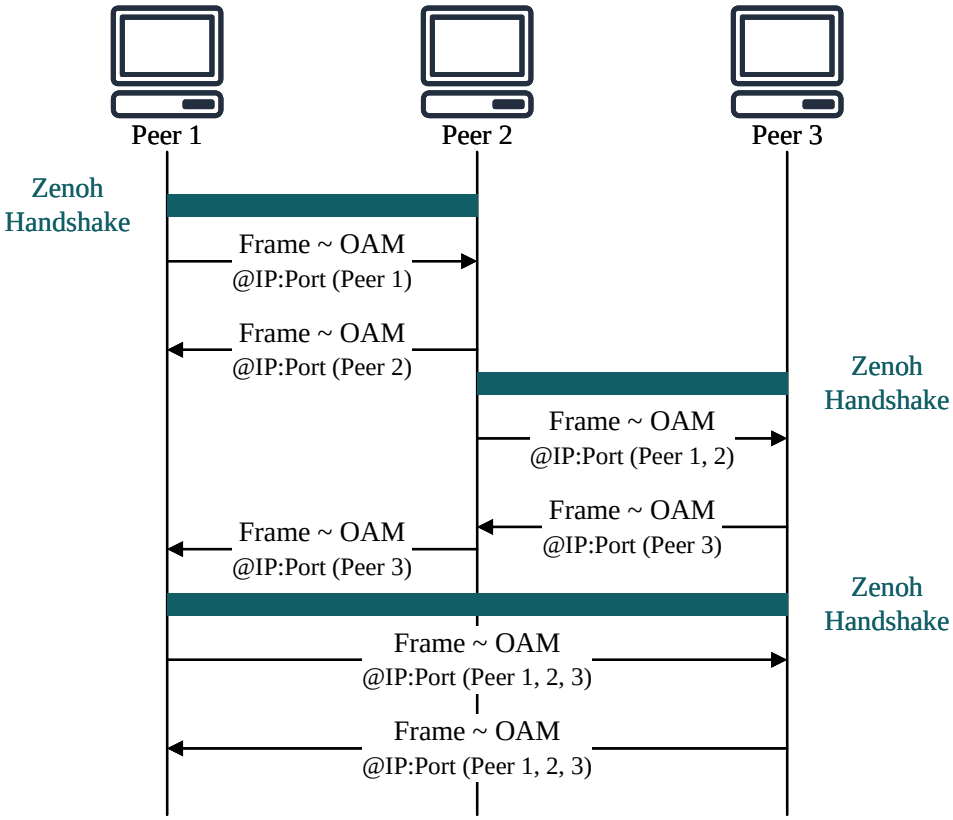


Figura 4.3: Proceso de conexión de descubrimiento y comunicación de los pares con *gossip scouting* en Zenoh. Elaboración propia.

dentro de un paquete `Frame` (`OAM`). Posteriormente, el nuevo nodo establece conexiones con los otros nodos e intercambia sus tablas entre ellos. Este procedimiento se muestra en la Figura 4.3.

Cuando los pares operan en el modo de enrutamiento *linkstate*, que permite la comunicación en una red de malla donde no todos los nodos tienen conectividad directa entre sí, es conveniente activar el *multihop gossip scouting*. Esto implica un aumento en el tráfico de *scouting* y una reducción en la escalabilidad, pero permite que la información del descubrimiento *gossip* se propague a través de múltiples saltos a todos los nodos en la red local, asegurando un conocimiento completo entre ellos.

4.2. Apertura de puertos

En Zenoh, la apertura de puertos para los nodos en una red depende del rol que desempeñan. Por esta razón, se ha decidido abordar este procedimiento a partir de las distintas topologías disponibles.

4.2.1. Comunicaciones punto a punto

Para que un par inicie la comunicación Zenoh con otro par, simplemente necesita abrir un puerto (en adelante, “Puerto Cliente”) para establecer una conexión con el Puerto Servidor de cada uno de los otros pares en la red. En el caso de las comunicaciones punto a punto el primer nodo únicamente abriría su

Puerto Servidor. Al incorporarse el segundo nodo, éste abriría un Puerto Cliente para comunicarse con el nodo existente y su Puerto Servidor. El tercer nodo abriría dos Puertos Clientes (uno para cada nodo existente) y un Puerto Servidor. El proceso continuaría por cada nodo que se incorporase a la red Zenoh.

Si se define P_{S_i} y P_{C_i} al número de Puertos Servidor y Cliente, respectivamente, en el par que se incorpora en la posición i-ésima a las comunicaciones punto a punto, para todos los nodos se tienes que

$$P_{S_i} = 1, P_{C_i} = i - 1$$

.

4.2.2. Comunicaciones Brokered

Analizando las comunicaciones en modo *Brokered*, se ha observado que los clientes no abren ningún Puerto Servidor y únicamente un Puerto Cliente. Este se abre para comunicarse a través del router Zenoh, que tiene un Puerto Servidor y no necesita abrir ningún Puerto Cliente para comunicarse con los clientes.

Usando la misma notación que en las comunicaciones P2P y si se identifica al router con el nodo 0 de la red, el número de puertos Servidor y Cliente que abre el router y el resto de nodos se puede expresar como:

$$P_{S_0} = 1, P_{C_0} = 0; \quad P_{S_i} = 0, P_{C_i} = 1 \quad \forall i \neq 0$$

4.2.3. Comunicaciones Routed

En un escenario de comunicación *Routed* se ha observado una configuración de las comunicaciones que incluye las dos comentadas anteriormente. El router Zenoh puede abrir Puertos Cliente para comunicarse a través de los Puertos Servidor de otros pares o routers Zenoh. Por su parte, los nodos cliente únicamente abren un Puerto Cliente ya sea con el router o con un par.

Si tenemos un conjunto de nodos clientes $\mathcal{N}_C$ y de nodos no cliente $\mathcal{N}_N$ (ya sean router o pares), se puede definir el número de puertos de cada tipo abiertos como:

$$P_{S_i} = 1, P_{C_i} = i - 1 \quad \forall i \in \mathcal{N}_C; P_{S_j} = 0, P_{C_j} = 1 \quad \forall j \in \mathcal{N}_N$$

4.2.4. Ejemplo de apertura de puertos

Tomando como referencia la Figura 4.4, se aporta un ejemplo que aporte una mayor claridad de los conceptos relativos al descubrimiento de nodos *multicast* y a la apertura de puertos en Zenoh utilizando TCP como protocolo de Capa de Transporte.

Dentro de una fábrica, se tienen dos plantas industriales. En la primera hay dos sensores ambientales (temperatura y CO_2) y una pantalla en que se indican los valores de estos dos parámetros en la planta. En la segunda se encuentra un sensor de temperatura y una pantalla que refleja los valores de temperatura de ambas plantas. Enlazando las dos plantas, tenemos un router Zenoh.

Se ha decidido determinar, por tanto, tres escenarios diferentes: *Clique* P2P (1), *Brokered* (2) y *Routed* (3).

En (1) se tienen tres pares interconectados. La aplicación Zenoh en el sensor de temperatura (*Pub1*) es la que primero se conecta. Se subscribe al grupo *multicast*, enviando un mensaje de *scout* indicando su rol (par, `010803`) y abre su Puerto Servidor (`1234`). Posteriormente, la del sensor de CO_2 (*Pub2*) se

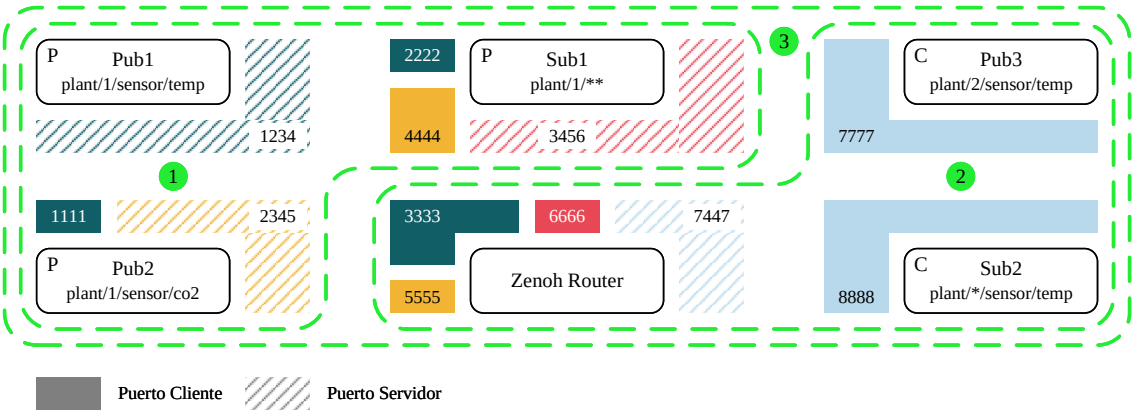


Figura 4.4: Apertura de puertos en Zenoh. Visualización de seis aplicaciones diferentes de Zenoh agrupadas en tres escenarios: (1) escenario 1: tres pares (dos publicadores (*Pubs*) y un suscriptor (*Sub*)) formando una topología *Clique*; (2) escenario 2: dos clientes (un *Pub* y un *Sub*) y un router Zenoh, formando una topología *Brokered*; (3) escenario 3: los seis nodos combinados, formando una topología *Routed*. La figura muestra los puertos abiertos por cada nodo, distinguiendo entre *Puertos Servidor* y *Puertos Cliente*; ilustrando los caminos de comunicación directa entre nodos. El color de los puertos indica la comunicación directa: nodos con puertos del mismo color se comunican directamente entre sí. Se indican también las expresiones de clave en que publican o las que están suscritos los nodos. Elaboración propia.

subscribe al grupo *multicast* indicando que también es un par en el mensaje de *scout* (`010803`), abre su Puerto Servidor (`2345`) y descubre a *Pub1*. Se intercambian sus @IP:Port y *Pub2* abre un Puerto Cliente (`1111`) para comunicarse con el Puerto Servidor de *Pub1*.

A continuación, la aplicación de la pantalla de la primera planta (*Sub1*) se subscribe al grupo *multicast* con un mensaje `010803`, descubre a ambos pares e intercambia con ellos sus respectivos @IP:Port. *Sub1* abre, entonces, dos Puertos Cliente (`2222` y `4444`), para establecer la comunicación con los Puertos Servidor de *Pub1* y *Pub2*, respectivamente.

En (2), se conectan dos clientes a través de un router Zenoh que actúa como *broker*. En primer lugar, el router Zenoh se subscribe al grupo *multicast* (`010807`) y abre su Puerto Servidor (`7447`[3]). El cliente en el sensor de temperatura (*Pub3*) se subscribe al grupo *multicast* indicando su rol (`010801`) y descubre al router, que le manda su @IP:Port. *Pub3* abre su Puerto Cliente (`7777`) y establece la comunicación con el Puerto Servidor del router.

El cliente de la pantalla (*Sub2*) se suscribe al grupo *multicast* (`010801`) y descubre al router Zenoh. A continuación abre su Puerto Cliente (`8888` y se conecta con el Puerto Servidor del router.

Aunando los escenarios (1) y (2), obtenemos el escenario (3), permitiendo a los clientes *Pub3* y *Sub2* comunicarse con los pares de (1) a través del router Zenoh. Para ello, el router, al haberse unido a la red posteriormente, abre un Puerto Cliente por cada par de (1) (`3333`, `5555` y `6666`), estableciendo una conexión con los Puertos Servidor `1234`, `2345` y `3456`.

[3]Por defecto, el puerto `7447` sería el Puerto Servidor en TCP de todas las aplicaciones Zenoh. No obstante, se ha utilizado solo en el router, para generalizar, abriendo la posibilidad de que todas las aplicaciones propuestas estuvieran en un mismo *host*.

Tabla 4.2: Paquetes de control en Zenoh.

Tipo de paquete			Descripción	Valor
InitSyn			Solicitud de inicialización de la conexión.	81
InitAck			Reconocimiento de inicialización de la conexión.	a1
OpenSyn			Solicitud de apertura de sesión.	42
OpenAck			Reconocimiento de apertura de sesión.	62
Close			Cierre de la conexión.	03
KeepAlive			Mantenimiento de la conexión.	04
Frame	Declare	DeclareKeyExpr	Declaración de la KE utilizada.	25
		DeclareSubscriber	Declaración de un suscriptor.	
		DeclareQueryable	Declaración de un consultable.	
		UndeclareSubscriber	Anulación de la declaración de un suscriptor.	
		UndeclareQueryable	Anulación de la declaración de un consultable.	
	OAM		Transmisión de información de la red de nodos.	
	Push	Put	Envío de un valor.	
		Del	Eliminación de un valor.	
	Request	Query	Envió de una consulta.	
	Response	Reply	Respuesta a una consulta.	

4.3. Sesión Zenoh

Una vez se han conocido dos aplicaciones Zenoh y han abierto un canal de comunicaciones entre ellas, pueden iniciar una sesión. Para gestionar todos los intercambios que se pueden realizar a lo largo de una sesión, se definen varios tipos de paquetes de control para las cabeceras de las tramas Zenoh, detallados en la Tabla 4.2[4].

A continuación, se detallan los diferentes procesos, que incluyen el saludo inicial, la propia transmisión de la información, el mantenimiento de la conexión y el cierre de la misma.

4.3.1. Saludo

Al descubrir un nuevo nodo, se inicia un intercambio de tramas a nivel de aplicación para establecer la comunicación y abrir el canal correspondiente: (1) `InitSyn`, (2) `InitAck`, (3) `OpenSyn` y (4) `OpenAck`, tal y como se muestra en la Figura 4.5.

4.3.2. Conversación

Durante una sesión Zenoh, los valores pueden ser transmitidos o eliminados utilizando tramas `Push` (`Put`/`Del`). Como se mencionó en la Sección 3.2.4, las entidades que envíen valores frecuentemente a la misma KE pueden declararse como publicadores. Además, una entidad puede expresar interés en recibir todas las actualizaciones relacionadas con una o más claves mediante la declaración como suscriptor o como suscriptor en modo *pull* de una KE específica. Alternativamente, puede solicitar ciertos datos de interés a través de consultas. Estas solicitudes pueden ser respondidas por entidades que se han declarado como consultables para tales KEs. Las consultas se envían dentro de una trama `Request` (`Query`), y las respuestas se entregan mediante una trama `Response` (`Reply`).

[4]Los datos para la realización de esta Tabla han sido, en su mayoría, obtenidos a partir de capturas de Wireshark.

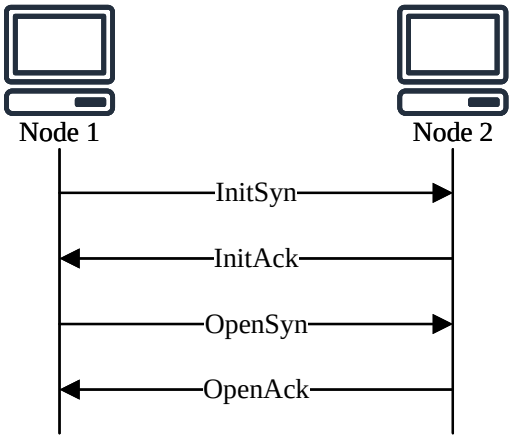


Figura 4.5: Saludo entre nodos en Zenoh. Elaboración propia.

Una vez establecida la sesión Zenoh, los nodos pueden comenzar a intercambiar tramas `Declare`, indicando interés de suscripción o disponibilidad para consultar claves específicas. Los nodos conservan la capacidad de anular la declaración según sea necesario.

4.3.3. Mantenimiento y cierre de la conexión

Para el mantenimiento de la conexión, los nodos transmiten periódicamente tramas `KeepAlive` después de 2,5 segundos de inactividad[5]. El cierre de la sesión, por su parte, se realiza mediante una trama `Close`. Ambos intercambios se muestran en la Figura 4.6.

4.3.4. Ejemplo de intercambio de datos

Para ilustrar el comportamiento de Zenoh a lo largo de una sesión, se propone retomar el ejemplo visto en la Figura 4.4.

Con la topología (3) se ha conseguido que todos los nodos tengan un conocimiento completo de la red y, de esta forma, que los datos producidos en los sensores de las plantas acaben llegando a las aplicaciones de las pantallas correspondientes.

Los pares realizan el saludo entre ellos y con el router Zenoh, que hace lo propio con los clientes. *Sub1* se declara suscriptor de la KE `plant/1/**`, enviando una trama `Declare` a todos los nodos. *Sub2*, por su parte, envía también una trama `Declare`, a través del router a toda la red, declarándose suscriptor de la KE `plant/*/sensor/temp`.

De este modo, *Pub1* (publica en la clave `plant/1/sensor/temp`) sabrá que debe enviar todas las actualizaciones de la clave a *Sub1* y, a través del router, a *Sub2*. Por su parte, *Pub2*, que publica en la clave `plant/1/sensor/co2`, notificará a *Sub1* y *Pub3*, que publica en la clave `plant/2/sensor/temp`, a *Sub2*.

[5]El periodo de tramas *KeepAlive* es configurable.

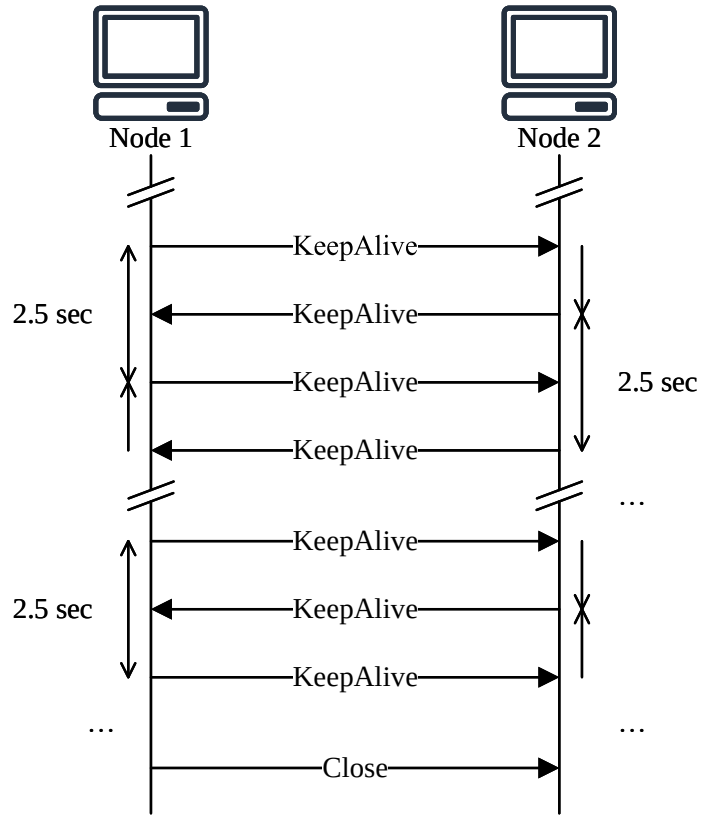


Figura 4.6: Mantenimiento de la conexión y cierre de sesión entre nodos en Zenoh. Elaboración propia.

CAPÍTULO 5

Análisis de rendimiento

En este capítulo, se presenta un análisis detallado del rendimiento de Zenoh en comparación con MQTT, centrándose en la latencia (tiempo de ida y vuelta (RTT, Round-trip Time)) en distintos escenarios. Además, se evalúa el comportamiento de los mecanismos de fiabilidad (*reliability*) y control de congestión de Zenoh.

El capítulo comienza con una sección dedicada a la instalación y despliegue, proporcionando una guía detallada para la instalación y configuración de MQTT y Zenoh, así como la captura de tráfico y configuración de red. Se describen también los parámetros configurables en las aplicaciones de Zenoh. Además, se especifica el entorno de pruebas, que incluye un *switch*, un *router* Wi-Fi y dispositivos Raspberry Pi.

A continuación, se presentan las pruebas de latencia, explicando en qué consisten estas pruebas y cómo se han configurado y describiendo los diversos escenarios en los que se han realizado. Los resultados de estas pruebas se analizan para ofrecer una visión comparativa entre Zenoh y MQTT.

Finalmente, se aborda el análisis de los mecanismos de fiabilidad y control de congestión de Zenoh. Se detalla en qué consisten las pruebas realizadas, la configuración utilizada, la presentación del escenario de pruebas y los resultados obtenidos.

5.1. Instalación y despliegue

Para la realización de las pruebas de Zenoh, se ha utilizado la API de Python de Eclipse Zenoh[1,2], que está basada en la implementación principal de Zenoh en Rust[3]. Se hace uso también del router Zenoh, *zenohd*[4]. En relación con los clientes y *brokers* de MQTT, se ha empleado la librería de cliente MQTT de Eclipse Paho[5] y Eclipse Mosquitto[6], respectivamente. Para el análisis del tráfico de Zenoh, se ha utilizado la herramienta de Linux `tcpdump` y el disector de Zenoh para Wireshark 4.0.10. Para ciertas medidas `tc`, la herramienta de control de tráfico de Linux, también ha sido utiliza.

[1] https://github.com/eclipse-zenoh/zenoh-python (accedido: 13 de mayo de 2024), versión 0.10.0-rc.
[2] https://zenoh-python.readthedocs.io/en/0.10.0-rc/ (accedido: 13 de mayo de 2024).
[3] https://github.com/eclipse-zenoh/zenoh (accedido: 13 de mayo de 2024).
[4] https://github.com/eclipse-zenoh/zenoh/tree/main/zenohd (accedido: 13 de mayo de 2024), versión v0.10.1-rc-1-g15b36a0f.
[5] https://github.com/eclipse/paho.mqtt.python (accedido: 13 de mayo de 2024), versión 1.6.1.
[6] https://github.com/eclipse/mosquitto (accedido: 13 de mayo de 2024), versión 2.0.11.

A continuación se indican los pasos para desplegar y configurar el entorno de medida, incluyendo tanto Zenoh como MQTT. Asimismo, se describe el entorno de pruebas que se ha implementado para realizar el análisis correspondiente.

Respecto a la instalación, en el Apéndice B se proporciona una guía detallada con una serie de comandos que pueden ser utilizados para instalar, configurar y ejecutar aplicaciones con MQTT (B.1) y con Zenoh (B.2). Estos incluyen:

- MQTT: *Broker* Mosquitto, Clientes Paho de Python.
- Zenoh: Librería Zenoh de Python, *Router* Zenoh (*zenohd*).

Otros comandos de uso más general, para la captura de tráfico (analizador de protocolos Wireshark, disector de Zenoh para Wireshark, herramienta de análisis de red de Linux `tcpdump`) y la configuración de la red (herramienta de control de tráfico de Linux `tc`, ajuste de los *buffers* de transmisión y recepción de los *sockets* UDP), se muestran en el Apéndice C.

Todos los comandos incluidos en estos apéndices son válidos para sistemas operativos basados en distribuciones de Linux, con excepción de los referentes a Wireshark, que están dirigidos a sistemas Windows.

A la hora de desplegar una aplicación en Zenoh, es fundamental comprender y ajustar los distintos parámetros de configuración para asegurar un rendimiento óptimo y una funcionalidad acorde a las necesidades del entorno de implementación.

En el Apéndice D se aporta una tabla completa con estos parámetros (Tabla D.1), realizada a partir del archivo de configuración por defecto (*DEFAULT_CONFIG.json5*[7]) disponible en el repositorio de Zenoh. La tabla muestra, para cada uno de los parámetros, la siguiente información:

- Nombre del parámetro: Identificador del parámetro dentro del archivo de configuración.
- Descripción: Explicación del propósito y el uso del parámetro.
- Posibles valores: Opciones que se pueden asignar al parámetro, incluyendo valores por defecto.

Con esta tabla se busca proporcionar una referencia rápida y práctica, que permita a los usuarios ajustar los parámetros de manera eficiente y efectiva. De este modo, se facilita la personalización y optimización del despliegue de aplicaciones utilizando Zenoh, asegurando que todos los aspectos configurables estén claramente documentados y accesibles.

Las pruebas se han llevado a cabo sobre una plataforma de pruebas física que comprende dispositivos *Raspberry Pi 3 Model B V1.2*[8] y *B+*[9], usando ambos el sistema operativo *Linux raspberrypi 6.1.21-v7+* y *Raspbian GNU/Linux 11 (bullseye)*. Las conexiones inalámbricas se han establecido a través de un *router Asus RT-N18U*[10] operando en la banda de 2,4 GHz, capaz de ofrecer velocidades de hasta

[7] https://github.com/eclipse-zenoh/zenoh/blob/master/DEFAULT_CONFIG.json5 (accedido: 13 de mayo de 2024).
[8] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ (accedido: 14 de julio de 2024).
[9] https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/ (accedido: 14 de julio de 2024).
[10] https://www.asus.com/es/supportonly/rt-n18u/helpdesk_manual/ (accedido: 14 de julio de 2024).

Tabla 5.1: Propiedades de escenarios para pruebas de latencia.

P: Par; R: Router Zenoh; C: Cliente, B: Broker MQTT.

Escenario	Nº de hosts		Nº de aplicaciones		Roles		Nº de redes		Conectividad		Retardo (*ms*)	
	Zenoh	MQTT	Zenoh	MQTT	Zenoh	MQTT	Zenoh	MQTT	Zenoh	MQTT	Zenoh	MQTT
α	3	3	3	3	P-R-C	C-B-C	1	1	Wi-Fi	Wi-Fi	–	–
β	3	3	3	3	P-R-C	C-B-C	2	2	Wi-Fi/Eth	Wi-Fi/Eth	–	–
β + Retardo	3	3	3	3	P-R-C	C-B-C	2	2	Wi-Fi/Eth	Wi-Fi/Eth	200	200
γ	4	4	4	5	P-R-R-C	C-B-C-B-C	3	3	Wi-Fi/Eth/Eth	Wi-Fi/Eth/Eth	–	–
δ	2	–	2	–	P-P	–	1	–	Wi-Fi	–	–	–

600 Mbps (estándar *802.11n* en modo infraestructura). Además, se ha utilizado un *switch Gigabit Ethernet NETGEAR GS108*[11] para las conexiones cableadas.

5.2. Pruebas de latencia

En primer lugar, se ha realizado un análisis comparativo de retardos entre Zenoh y MQTT, considerando escenarios típicos en entornos industriales, desplegando diversas aplicaciones que emulan sensores o actuadores y enrutando la comunicación a través de *routers* o *brokers*. En todas las pruebas, se ha establecido una conexión inalámbrica entre los dos primeros nodos de la red, dada la creciente presencia de estas tecnologías inalámbricas en las comunicaciones M2M y dentro de la IIoT [51]. Además, la primera aplicación de Zenoh ha operado en modo *peer* en todos los escenarios. Es importante destacar que se ha utilizado TCP como protocolo subyacente en la capa de transporte en todos los experimentos realizados en esta primera fase.

Como se muestra en la Figura 5.1, se han propuesto tres escenarios distintos (α, β, γ), que incluyen tres (α, β) o cuatro *hosts* (γ):

- Escenario α: dos aplicaciones *brokered* dentro de la misma red, con conectividad totalmente inalámbrica.
- Escenario β: dos aplicaciones *brokered* en dos redes diferentes, una de las cuales sobre Ethernet.
- Escenario γ: dos aplicaciones *routed* a través de dos *routers* o *brokers*, en tres redes diferentes, estando las dos últimas conectadas mediante Ethernet.

Para Zenoh, se ha considerado un escenario adicional, δ, que permite analizar su rendimiento en topologías P2P, con conectividad completamente inalámbrica, a fin de evaluar las ventajas que ofrece este enfoque en comparación con otras configuraciones de red. A modo de resumen, en la Tabla 5.1, se indican las características principales de todos los escenarios.

La latencia para las diferentes conexiones se ha obtenido mediante la medición del RTT, basado en la comunicación entre publicadores y suscriptores mediante un mecanismo de solicitud/respuesta de eco, con acuse de recibo de los mensajes entregados. El RTT, tal y como muestra la Ecuación 5.1, se ha calculado restando el tiempo inicial t_0 del tiempo final t_1. El valor de t_0 se registra cuando el primer nodo en la red envía datos a una clave a la que el último nodo está suscrito. Este valor se entrega al último nodo, a través de nodos intermedios, si existieran. Cuando el último nodo recibe el mensaje, publica un mensaje

[11] https://www.netgear.com/es/business/wired/switches/unmanaged/gs108/ (accedido: 14 de julio de 2024).

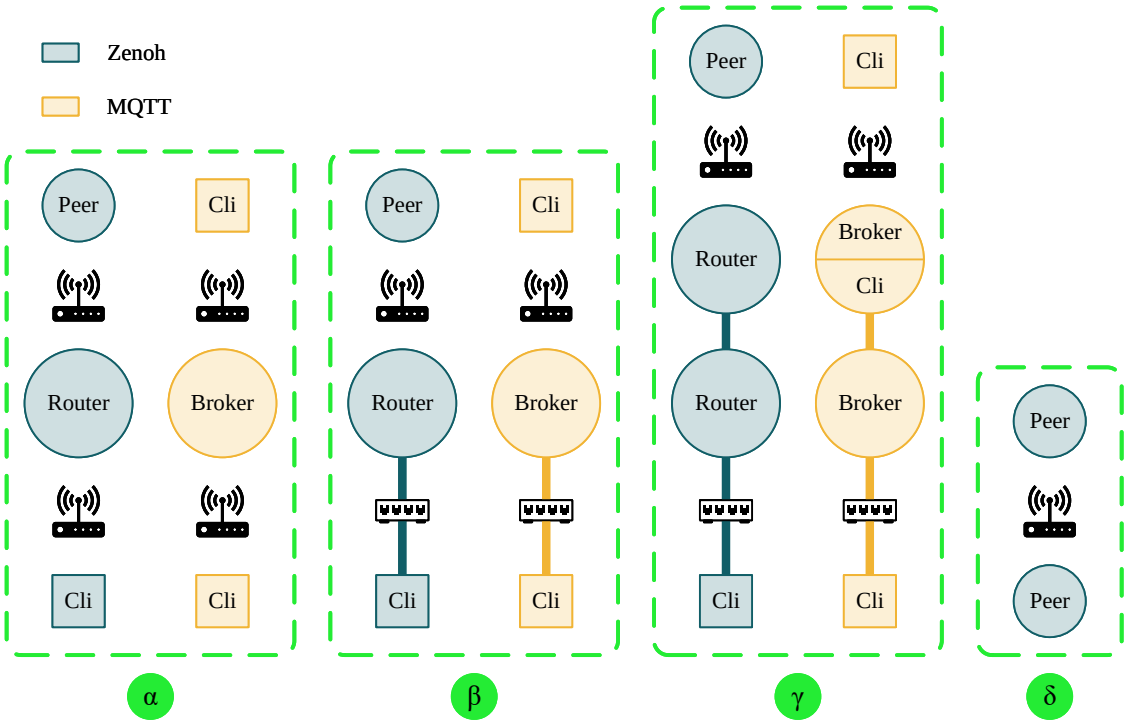


Figura 5.1: Configuración de los cuatro escenarios propuestos para evaluar el rendimiento de las aplicaciones en diferentes topologías de red: (1) escenario α: dos aplicaciones intermediadas dentro de la misma red, utilizando conectividad inalámbrica; (2) escenario β: dos aplicaciones intermediadas en dos redes diferentes, con la segunda red conectada mediante Ethernet; (3) escenario γ: dos aplicaciones enrutadas a través de dos routers en tres redes diferentes, con las dos últimas redes conectadas mediante Ethernet; (4) escenario δ: dos aplicaciones conectadas punto a punto en una red con conectividad inalámbrica. Para los escenarios α, β y γ, Zenoh está representado a la izquierda y MQTT a la derecha. La tecnología utilizada en cada red (Wi-Fi o Ethernet) está debidamente indicada. Se utiliza el término "Cli" para referirse a las aplicaciones Zenoh en modo cliente. Elaboración propia.

de confirmación (ACK, Acknowledgment) con una carga útil de 2 B a otra clave a la que el primer nodo en la red está suscrito. Por otro lado, t_1 se registra cuando el primer nodo recibe dicho ACK. Este proceso, que está ilustrado en la Figura 5.2, se ha repetido mil veces sin agregar tiempo de espera entre la recepción y la transmisión, con el fin de obtener una latencia promedio más precisa.

$$RTT = t_1 - t_0 \tag{5.1}$$

Para ambos protocolos, se ha establecido una sesión en los nodos inicial y final, con un cliente que tanto publica como se suscribe a dos claves/temas[12] distintos. En el primer nodo, el cliente publica en la clave/tema `test/data` y se suscribe a `test/ack`. Por su parte, el cliente en el último nodo se suscribe a `test/data` y publica en `test/ack`.

En lo referente a la fiabilidad y al control de congestión, se han considerado dos escenarios para Zenoh: (1) `RELIABLE` y `BLOCK`, y (2) `BEST_EFFORT` y `DROP`. Para MQTT, se realizaron análisis para sus tres niveles de QoS. El nivel 0 de QoS de MQTT se asemejaría más al escenario (2) [33].

[12] Clave (*key*) en el caso de las pruebas con Zenoh, tema (*topic*) con MQTT.

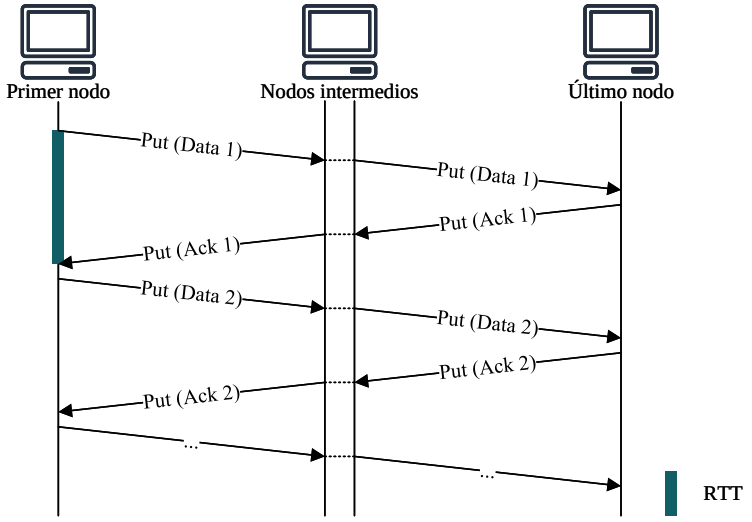


Figura 5.2: Método utilizado para la medición del RTT de la conexión. Elaboración propia.

Las aplicaciones Zenoh fueron configuradas siguiendo la estructura descrita en el esquema[13] proporcionado por ZettaScale Technology, tal y como se indica en el Apéndice D. Los parámetros de configuración incluyen el tipo de nodo (par o cliente) y la dirección del *endpoint* para el establecimiento de la conexión. Se habilitó el *scouting multicast*, con los ajustes predeterminados, dado su potencial uso en entornos IIoT. Además, se empleó `Peer_to_peer` como la estrategia de enrutamiento para los pares, mientras que los parámetros de transporte se mantuvieron en sus valores por defecto. Para el router Zenoh, se utilizó la configuración predeterminada de *zenohd*. Los códigos utilizados para realizar las pruebas de latencia se incluyen en el Apéndice E.

5.2.1. Escenario α

El primer escenario propuesto comprende una red que opera exclusivamente con conectividad Wi-Fi, con tres nodos: un publicador, un router de Zenoh o un *broker* de MQTT y un suscriptor.

En la Figura 5.3 se muestra el diagrama de caja de la latencia medida. Para cada configuración los límites de la caja indican el primer y tercer cuartil, percentiles 25 y 75 respectivamente. La línea intermedia de cada caja indica la mediana (percentil 50) y el círculo la media. Finalmente los bigotes indican el rango de valores que contiene aproximadamente el 99 % de las muestras.

Los resultados mostrados en la Figura 5.3 evidencian que Zenoh, utilizando tanto mecanismos de *Best Effort* como *Reliable*, muestra un rendimiento similar al observado para MQTT con niveles de QoS 0 y 1, obteniendo latencias ligeramente menores en ambos escenarios. Zenoh supera significativamente a MQTT con QoS 2, lo cual puede no ser una solución adecuada cuando se emplea TCP como protocolo de transporte subyacente. Esto se debe a la implementación de TCP de mecanismos de control de flujo, control de congestión y retransmisión de paquetes, que tienen como objetivo asegurar una mínima pérdida de paquetes. No obstante, la adopción de MQTT con QoS 2 podría justificarse en aplicaciones que requieran un nivel adicional de fiabilidad y garantía en la entrega de mensajes.

[13] https://github.com/eclipse-zenoh/zenoh/blob/master/commons/zenoh-config/src/lib.rs (accedido el 13 de mayo de 2024)

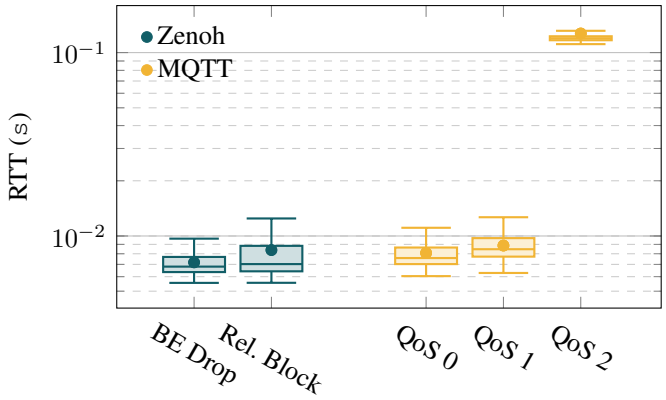
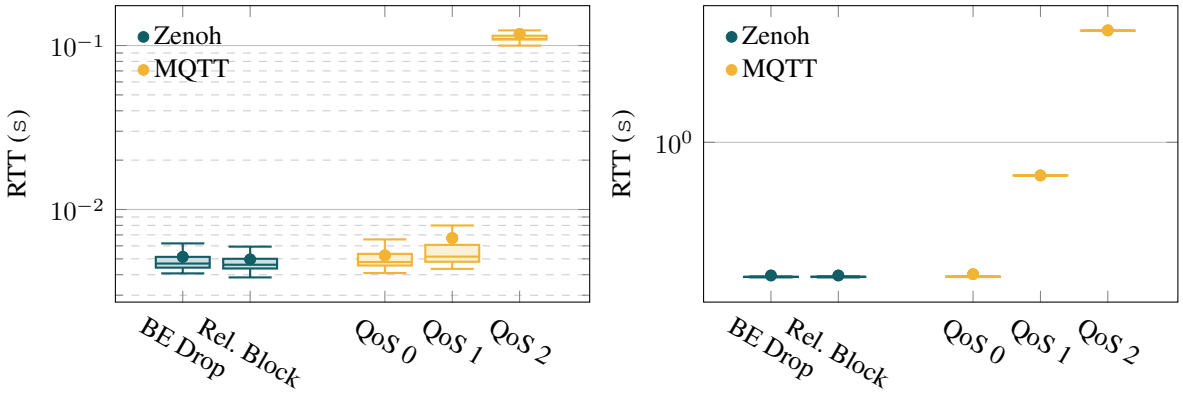


Figura 5.3: RTT obtenido en el escenario α.



(a) Sin retardo.

(b) Con retardo de 200 *ms*.

Figura 5.4: RTT obtenido en el escenario β con y sin retardo adicional.

5.2.2. Escenario β

En este segundo escenario, una de las conexiones inalámbricas (router Zenoh/broker de MQTT al cliente) se reemplaza con una conexión Ethernet. Como era de esperar, este cambio resulta en menores latencias en comparación con el escenario α, como se muestra en la Figura 5.4a. Los resultados siguen evidenciando mejoras ligeras al usar Zenoh, en comparación con MQTT.

Utilizando el escenario β, se han obtenido resultados adicionales, modificando el tráfico de red mediante la herramienta Traffic Control (tc) de Linux. Con esto, se simulan mayores latencias, introduciendo retardos de 200 ms sobre la red Ethernet (tanto en subida como en bajada), imitando un escenario basado en la nube, donde los retardos pueden ser significativamente mayores.

Los resultados obtenidos, que se muestran en la Figura 5.6, reflejan una diferencia mucho mayor entre Zenoh y MQTT que la observada previamente, alcanzando mejoras de más de $0{,}5$ ms en el RTT en el mejor de los casos.

5.2.3. Escenario γ

En el escenario γ con MQTT, se ha configurado un único *host* para actuar como *broker* y cliente, sirviendo de puente entre los dos *brokers*. Esta configuración asegura que los resultados obtenidos sean adecuados

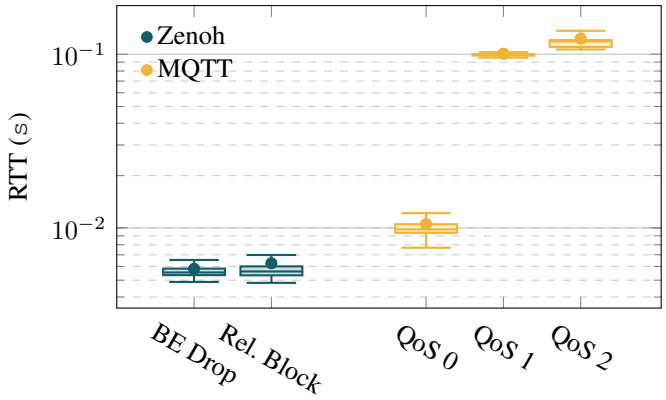


Figura 5.5: RTT obtenido en el escenario γ.

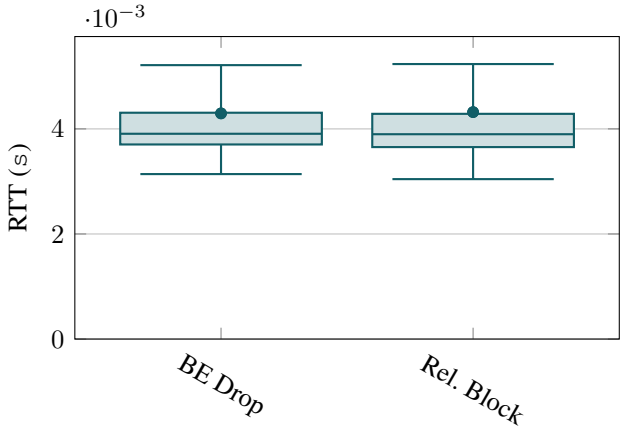


Figura 5.6: RTT obtenido en el escenario δ.

para compararse con una topología enrutada de Zenoh.

La Figura 5.5 muestra que el rendimiento (latencias) logrado con Zenoh es significativamente mejor (menor) en comparación con MQTT. Cabe indicar que, a pesar de que Zenoh mostró resultados ligeramente peores en el escenario β, debido a un salto adicional en la comunicación, aún se puede apreciar una mejora respecto a los resultados logrados con MQTT en el escenario α. Por el contrario, MQTT ahora (γ) muestra latencias incluso más altas que las observadas en α. Esto resalta el potencialmente bajo rendimiento de MQTT en sistemas distribuidos con múltiples saltos entre productores y consumidores. En contraste, Zenoh no solo asegura un rendimiento adecuado, sino que también ofrece una implementación más sencilla en dichos escenarios.

5.2.4. Escenario δ

Finalmente, se propone analizar un escenario δ, con el objetivo de caracterizar la latencia que se obtendría con Zenoh en su escenario, a priori, más favorable. Esto es, una comunicación directa entre dos pares dentro de la misma red, en este caso, con cobertura Wi-Fi. Como se muestra en la Figura 5.6, como cabría esperar, los retardos obtenidos en los dos casos son mucho menores que los observados en el resto de los escenarios analizados.

A modo de resumen, la Tabla 5.2 incluye los valores de las medianas de los RTT obtenidos para cada escenario analizado, para facilitar la comparación de los comportamientos observados.

Tabla 5.2: Mediana del RTT obtenido en todos los escenarios analizados.

Escenario	Zenoh		MQTT		
	Best Effort	Reliable	QoS 0	QoS 1	QoS 2
α	6.807	7.027	7.582	8.469	119.73
β	4.680	4.604	4.782	5.159	109.66
β + Retardo	408.997	409.082	409.659	802.57	2099
γ	5.539	5.613	9.804	99.002	118.39
δ	3.907	3.899	–	–	–

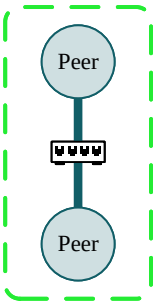


Figura 5.7: Configuración del escenario propuesto para evaluar los mecanismos de fiabilidad y control de congestión de Zenoh: dos aplicaciones en modo `peer` en dos *hosts* separados conectados vía Ethernet. Elaboración propia.

Tabla 5.3: Propiedades de escenario para pruebas de análisis de mecanismos de fiabilidad y control de congestión.

P: Par.

Escenario	Nº de hosts Zenoh	Nº de aplicaciones Zenoh	Roles Zenoh	Nº de redes Zenoh	Conectividad Zenoh	Retardo (*ms*) Zenoh
1	2	2	P-P	1	Eth	–

5.3. Pruebas de fiabilidad y control de congestión

Por otra parte, se han realizado varias pruebas para analizar los mecanismos de fiabilidad y control de congestión de Zenoh sobre TCP y UDP.

En este caso, el escenario se ha simplificado para analizar exclusivamente la interacción de Zenoh con estos mecanismos. Como se muestra en la Figura 5.7, dos aplicaciones, operando en modo `peer` en hosts separados, están conectadas vía Ethernet para evitar pérdidas inducidas por el canal. La configuración de este escenario está nuevamente indicado en la Tabla 5.3.

Las pruebas han consistido en la publicación, por parte del primer par de la red, de un archivo de 5 MB segmentado en: (1) 5000 paquetes de 1000 B, (2) 10000 paquetes de 500 B y (3) 50000 paquetes de 100 B. No se ha aplicado deliberadamente tiempo de espera entre las transmisiones para inducir congestión. El segundo par, cuando recibe la actualización, publica un mensaje de reconocimiento. Este mensaje es de tan solo 2 B para evitar que afecte a la congestión del *buffer*. Este proceso está ilustrado en la Figura 5.8.

Se han ajustado los archivos de configuración (Apéndice D) para que los pares se conecten y escuchen en *endpoints* específicos, alternando entre los protocolos TCP y UDP. También se han deshabilitado

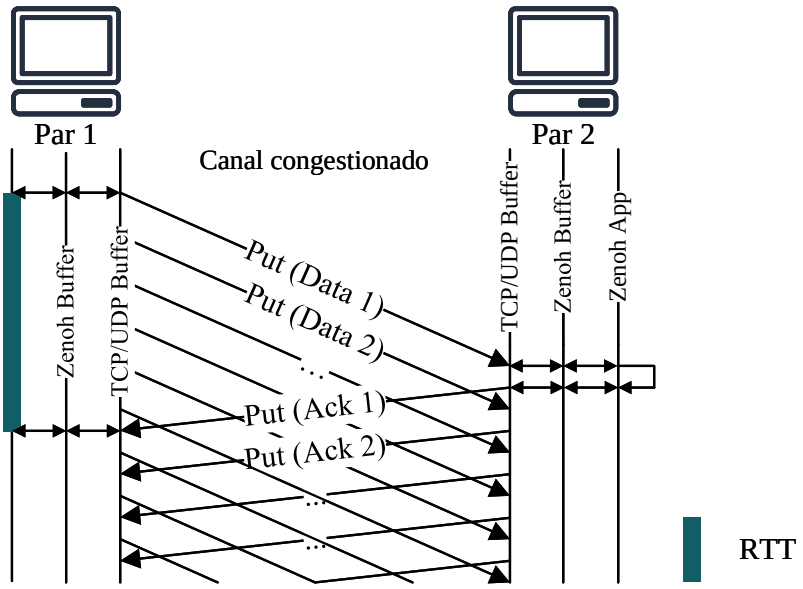


Figura 5.8: Método utilizado para la congestión de la red y el análisis del comportamiento de los mecanismos de fiabilidad y control de congestión de Zenoh. Elaboración propia.

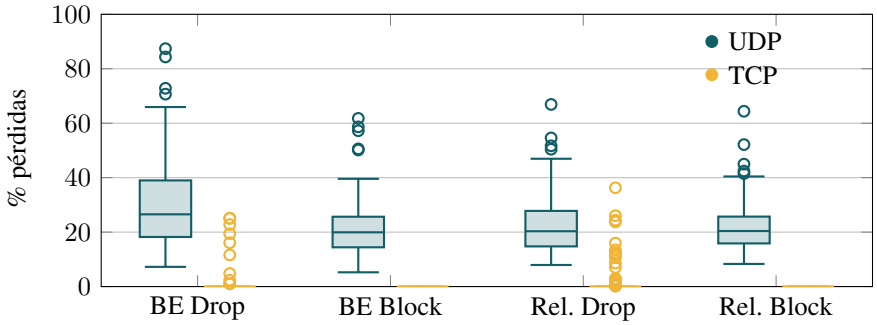
las funciones de *multicast* y *gossip scouting*, y el descubrimiento P2P se ha gestionado directamente, minimizando el tráfico adicional de *scouting*. Se ha deshabilitado el *timestamping* para prevenir la pérdida de mensajes no intencionada. Los pares han operado en modo `peer-to-peer`, simplificando el enrutamiento y reduciendo la sobrecarga de la red. Los ajustes de transmisión se han determinado para optimizar el rendimiento, con un tamaño máximo de lote de 65535 B. Los ajustes de recepción han incluido un *buffer* de 65535 B y se admiten tamaños de mensaje de hasta 1 GiB.

En las pruebas se han utilizado las configuraciones de *buffer* predeterminadas en dispositivos Raspberry Pi. Esto es, con longitudes máximas y por defecto, en buffers de recepción y transmisión, de 180224 B para los *sockets* de red.

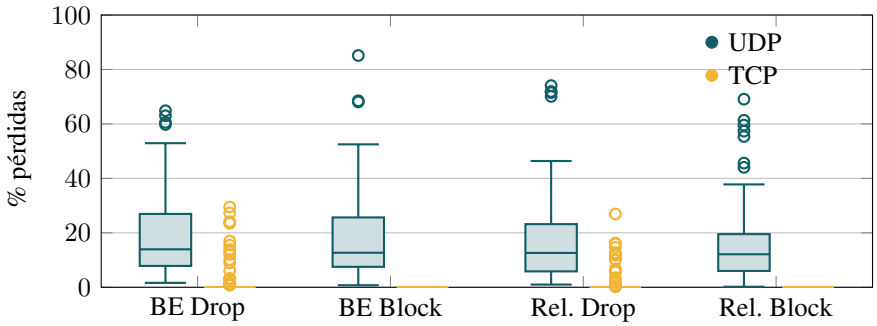
Se ha tomado un total de 100 muestras para cada una de las cuatro configuraciones analizadas: (1) `BEST_EFFORT` con `DROP`, (2) `BEST_EFFORT` con `BLOCK`, (3) `RELIABLE` con `DROP`, y (4) `RELIABLE` con `BLOCK`. Se ha estudiado el impacto de cada mecanismo en la pérdida de paquetes y en el RTT, debido a la congestión de la red.

En base a los resultados mostrados en la Figura 5.9, se puede concluir que el uso de UDP como capa de transporte conlleva un número significativo de pérdidas de paquetes para todas las configuraciones, con diferencias menores entre ellas. A medida que aumenta el número de paquetes enviados, causando congestión en la red, la tasa de pérdida se incrementa significativamente, especialmente con la configuración (1).

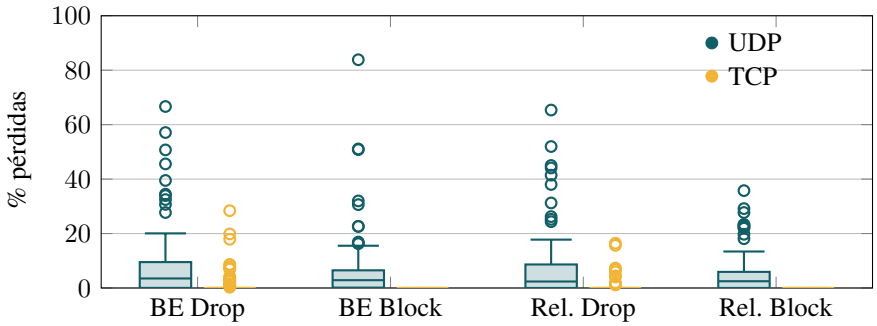
Aunque los mecanismos de control de congestión de Zenoh han demostrado tener una efectividad limitada con UDP, los resultados obtenidos con TCP evidencian que no ha habido pérdidas de paquetes al emplear la opción `BLOCK()` para detener las transmisiones. Cabe destacar, como se discute en [52], que el requisito de fiabilidad en el receptor puede no cumplirse consistentemente si el productor utiliza `DROP()` para el control de congestión. Esto se observa claramente con las pérdidas de paquetes que ocurren incluso



(a) Paquetes de 100 B.



(b) Paquetes de 500 B.



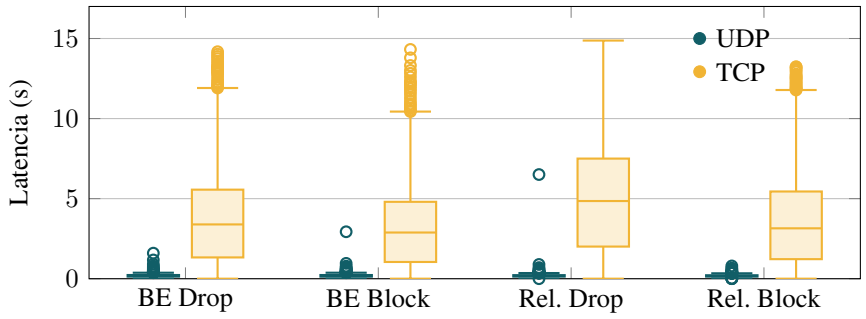
(c) Paquetes de 1000 B.

Figura 5.9: Distribución del porcentaje de pérdidas para UDP y TCP utilizando diferentes configuraciones y para varios tamaños de paquete.

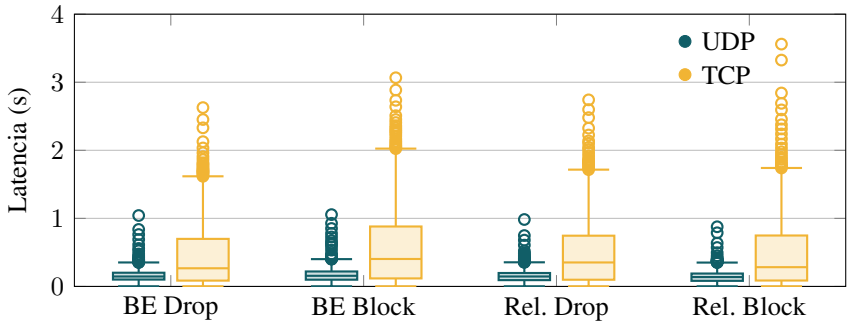
con TCP —diseñado con control de flujo y congestión—, ya que Zenoh descarta paquetes antes de alcanzar la capa de transporte, lo que no activa ningún mecanismo de control de pérdida, ni retransmisiones de paquetes.

La Figura 5.10 ilustra la distribución del RTT para cada paquete. Como era de esperar, las transmisiones por UDP muestran una latencia considerablemente menor en comparación con TCP, reflejando una compensación entre la pérdida de paquetes y el retardo. Mientras que los valores de RTT son similares cuando la conexión implica un menor número de paquetes, se pueden observar diferencias significativas cuando este número aumenta. En casos extremos, Zenoh sobre TCP conduce a tiempos de RTT cercanos a 15 segundos para paquetes individuales.

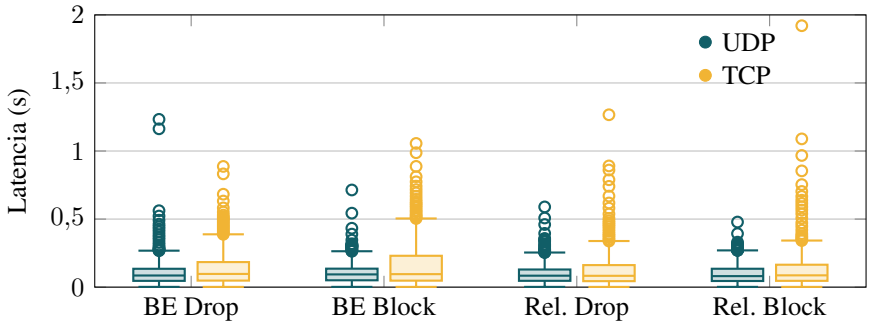
Los códigos utilizados para analizar los mecanismos de fiabilidad y control de congestión de Zenoh se incluyen en el Apéndice F.



(a) Paquetes de 100 B.



(b) Paquetes de 500 B.



(c) Paquetes de 1000 B.

Figura 5.10: Distribución de la latencia por paquete para UDP y TCP utilizando diferentes configuraciones y varios tamaños de paquete.

CAPÍTULO 6

Conclusiones y líneas futuras

En este último capítulo, se detallan las principales conclusiones obtenidas a lo largo del desarrollo del Trabajo. Además, se presentan las diversas líneas de trabajo que se derivan del estudio realizado.

6.1. Conclusiones

La Cuarta Revolución Industrial, marcada por la integración de tecnologías avanzadas y la automatización inteligente, está transformando radicalmente los entornos industriales, dando lugar a lo que conocemos como la Industria 4.0. Esta nueva era se caracteriza por un continuo desarrollo y perfeccionamiento en el ámbito tecnológico, con un enfoque primordial en la innovación. En este contexto, los protocolos de comunicación juegan un papel crucial, facilitando la interoperabilidad y eficiencia de los sistemas IoT. Protocolos establecidos, como MQTT, han sido fundamentales en esta transición, proporcionando mecanismos robustos para la transmisión de datos en entornos industriales.

En busca de un mayor rendimiento, surge el protocolo Zenoh, que se presenta como una solución prometedora para mejorar la eficiencia en la comunicación y el intercambio de datos en entornos IIoT. Especialmente pensado para comunicaciones V2X y R2X, Zenoh se ha desarrollado con el objetivo de ofrecer una mejor gestión de datos y una latencia mínima, aspectos críticos en aplicaciones industriales que requieren alta precisión y fiabilidad.

Dado que Zenoh es un protocolo relativamente nuevo y propietario, la escasez de información disponible ha planteado un desafío considerable. Por esta razón, en este Trabajo, se ha analizado su funcionamiento a nivel de red para comprender mejor sus capacidades y cómo gestiona las comunicaciones. La investigación se ha centrado en evaluar el rendimiento de Zenoh mediante la comparación de latencias frente a MQTT en diversos escenarios. Este análisis ha revelado que Zenoh ofrece mejoras en términos de latencia, lo cual es esencial para aplicaciones en tiempo real donde la rapidez de respuesta es crucial.

Además, se ha profundizado en los mecanismos de fiabilidad y control de congestión de Zenoh. Estos mecanismos son fundamentales para asegurar la integridad y estabilidad de las comunicaciones, especialmente en entornos industriales donde los sistemas deben operar de manera continua y sin interrupciones.

6.2. Líneas futuras

Vistos los resultados obtenidos a lo largo del trabajo, se considera oportuno seguir estudiando el protocolo Zenoh para aprovechar al máximo sus capacidades y potencial en diversas aplicaciones. Las siguientes líneas de investigación se proponen para continuar y expandir los resultados presentados en este Trabajo:

- Análisis de los cambios y mejoras aportadas con la versión más actual de Zenoh.
- Evaluación del rendimiento de Zenoh sobre QUIC en comparación con Zenoh sobre TCP con TLS.
- Análisis más amplio de las extensiones (*Plugins*) disponibles en Zenoh (Storage, REST API, MQTT, etc.).
- Prueba de Zenoh con comunicación serie y Zenoh-Pico sobre dispositivos de recursos limitados.
- Exploración del marco de trabajo nativo de Programación de Flujo de Datos de Zenoh, *Zenoh-Flow*.

Apéndice A

Entidades con Python API

A.1. Publicador

```python
import zenoh
# Iniciar una sesión Zenoh
session = zenoh.open(zenoh.Config())

# Declarar un publicador para una clave determinada
pub = session.declare_publisher(key)

# Publicar un valor
pub.put(value)

...

# Cancelar declaración
pub.undeclare()

# Cerrar sesión Zenoh
session.close()
```

A.2. Suscriptor

```python
import zenoh
# Iniciar una sesión Zenoh
session = zenoh.open(zenoh.Config())

# Definir una función de retorno como manejador
def listener(sample):
    print(f">> [Subscriber] Received {sample.kind} ('{sample.key_expr
        }': '{sample.payload.decode('utf-8')}')")
```

```

# Declarar un suscriptor a una clave determinada
sub = session.declare_subscriber(key, listener)

...

# Cancelar declaración
sub.undeclare()

# Cerrar sesión Zenoh
session.close()
```

A.3. Consultable

```
import zenoh
# Iniciar una sesión Zenoh
session = zenoh.open(zenoh.Config())

# Definir una función de retorno como manejador
def queryable_callback(query):
    query.reply(Sample(key, value))

# Declarar un consultable para una clave determinada
qabl = session.declare_queryable(key, queryable_callback)

...

# Cancelar declaración
queryable.undeclare()

# Cerrar sesión Zenoh
session.close()
```

APÉNDICE B

Guía de Instalación y Ejecución

B.1. MQTT

B.1.1. Instalación del Broker

- Para instalar el *broker* MQTT Mosquitto:

Listing B.1: Instalación de Mosquitto en Linux

```
sudo apt update
sudo apt-get install mosquitto mosquitto-clients
sudo systemctl enable mosquitto
sudo systemctl start mosquitto
```

B.1.2. Configuración del *broker* Mosquitto

- Editar el archivo de configuración de Mosquitto para permitir clientes remotos y especificar el puerto y el protocolo:

Listing B.2: Editción del archivo de configuración de Mosquitto

```
sudo nano etc/mosquito/conf.d/conf_file.conf
```

- Añadir las siguientes líneas al archivo de configuración:

Listing B.3: Configuración de Mosquitto

```
allow_anonymous true
listener 1883
protocol mqtt
```

B.1.3. Instalación de la librería de clientes Paho de Python

- Para instalar la librería de clientes Paho MQTT para Python usando `pip`:

Listing B.4: Instalación de Paho MQTT

```
pip3 install paho-mqtt
```

B.1.4. Ejecución del *broker* y clientes MQTT

- Para ejecutar el *broker* Mosquitto:

Listing B.5: Ejecución del *broker* Mosquitto

```
sudo mosquitto -c /etc/mosquitto/conf.d/conf_file.conf
```

- Para ejecutar un cliente Paho MQTT en Python:

Listing B.6: Ejecución de un cliente MQTT en Python

```
python3 MQTT_pub.py
```

B.2. Zenoh

B.2.1. Instalación de la librería Zenoh de Python

- Para instalar la librería de Zenoh para Python v0.10.0-rc utilizando `pip`:

Listing B.7: Instalación de Zenoh para Python

```
pip install eclipse-zenoh==0.10.0-rc
```

B.2.2. Instalación del *router* Zenoh (*zenohd*)

- La instalación del enrutador Zenoh se realiza añadiendo el repositorio de Eclipse y luego instalando el paquete:

Listing B.8: Instalación del *router* Zenoh (*zenohd*)

```
echo "deb [trusted=yes] https://download.eclipse.org/zenoh/debian-
    repo/ /" | sudo tee -a /etc/apt/sources.list > /dev/null
sudo apt update
sudo apt install zenoh
```

B.2.3. Ejecución de Zenoh y *zenohd*

- Para ejecutar una aplicación Zenoh con un archivo de configuración personalizado:

Listing B.9: Ejecución de Zenoh

```
python3 z_app.py --config z_app_config.json5
```

- Para ejecutar el enrutador Zenoh (*zenohd*) con un archivo de configuración personalizado:

Listing B.10: Ejecución del *router* Zenoh (*zenohd*)

```
zenohd --config zenohd_config.json5
```

APÉNDICE C

Configuración de Red

C.1. Wireshark y el disector de Zenoh en Windows

C.1.1. Instalación de Wireshark

- Instalar Chocolatey (gestor de paquetes para Windows):

Listing C.1: Instalación de Chocolatey usando PowerShell

```
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.
   ServicePointManager]::SecurityProtocol = [System.Net.
   ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object
    System.Net.WebClient).DownloadString('https://community.
   chocolatey.org/install.ps1'))
```

- Instalar Wireshark y los componentes necesarios:

Listing C.2: Instalación de Wireshark y dependencias

```
choco install -y --force --no-progress asciidoctorj xsltproc docbook-
   bundle nsis winflexbison3 cmake wireshark
```

- Descargar y compilar el disector de Zenoh:

Listing C.3: Compilación del disector de Zenoh

```
unzip zenoh-dissector-master.zip
cd zenoh-dissector-master
cargo build --release
```

C.2. Captura de Paquetes con `tcpdump`

- Instalar `tcpdump` utilizando `apt`:

Listing C.4: Instalación de tcpdump

```
sudo apt-get update
sudo apt-get install tcpdump
```

- Realizar una captura de paquetes en la interfaz deseada y guardar los resultados en un archivo:

Listing C.5: Captura de paquetes con tcpdump

```
sudo tcpdump -i NOMBRE_INTERFAZ -w NOMBRE_ARCHIVO.pcap
```

C.3. Modificación de Parámetros de Red con `tc`

- Añadir un retraso en la red utilizando `tc`:

Listing C.6: Añadir retraso con tc

```
sudo tc qdisc add dev eth0 root netem delay 200ms
```

- Elimina el retraso añadido con `tc`:

Listing C.7: Eliminar retraso con tc

```
sudo tc qdisc del dev eth0 root
```

C.4. Ajuste del *Buffer* UDP

- Ajustar los valores del *buffer* de recepción por defecto para los *sockets* UDP:

Listing C.8: Ajuste del buffer de recepción por defecto

```
sudo sysctl -w net.core.rmem_default=XXXXX
```

- "net.core.rmem_default" define el tamaño por defecto del *buffer* de recepción para los *sockets*. Sustituir "XXXXX" con el valor deseado en bytes.
- Ajustar los valores del *buffer* de envío por defecto para los *sockets* UDP:

Listing C.9: Ajuste del *buffer* de envío por defecto

```
sudo sysctl -w net.core.wmem_default=XXXXX
```

- “net.core.wmem_default” define el tamaño por defecto del buffer de envío para los *sockets*. Sustituir “XXXXX” con el valor deseado en bytes.
- Ajustar el tamaño máximo del *buffer* de recepción para los *sockets* UDP:

Listing C.10: Ajuste del tamaño máximo del *buffer* de recepción

```
sudo sysctl -w net.core.rmem_max=XXXXX
```

- “net.core.rmem_max” define el tamaño máximo del *buffer* de recepción para los *sockets*. Sustituir “XXXXX” con el valor deseado en bytes.
- Ajustar el tamaño máximo del *buffer* de envío para los *sockets* UDP:

Listing C.11: Ajuste del tamaño máximo del *buffer* de envío

```
sudo sysctl -w net.core.wmem_max=XXXXX
```

- “net.core.wmem_max” define el tamaño máximo del *buffer* de envío para los *sockets*. Sustituir “XXXXX” con el valor deseado en bytes.

Apéndice D

Archivo de configuración

Tabla D.1: Configuraciones posibles en Zenoh

Parámetro				Significado	Posibles valores	Valor por defecto	Notas
mode				El modo del nodo.	“router”, “peer”, “client”	“peer”	
metadata	name			El nombre del nodo.	“<name>”	—	
	location			La localización del nodo.	“<location>”	—	
connect	endpoints			La dirección del nodo al que conectarse para establecer una sesión Zenoh.	“<proto>/<address>”	—	Par o router.
listen	endpoints			La dirección en la que el nodo escucha que otros nodos pueden utilizar para establecer una sesión Zenoh con él.	“<proto>/<address>”	—	
scouting	timeout			El tiempo dedicado al descubrimiento de routers antes de fallar (*ms*).	<time>	3000	Solo en modo *client*.
	delay			El tiempo dedicado al descubrimiento de pares remotos antes de intentar otras operaciones (*ms*).	<time>	200	
	multicast	enabled		Habilita el descubrimiento *multicast*.	“true”, “false”	“true”	
		address		El *socket* que se utilizará para el descubrimiento *multicast*.	“<address>”	“224.0.0.224:7446”	
		interface		La interfaz de red que se utilizará para el descubrimiento *multicast*.	“auto”, “<ip_address>”, “<interface>”	“auto”	
		autoconnect	router	El tipo de instancia Zenoh con el que un router puede establecer sesiones en el descubrimiento *multicast*.	“router”, “peer”, “client”	“”	Un único valor o una combinación.
			peer	El tipo de instancia Zenoh con el que un par puede establecer sesiones en el descubrimiento *multicast*.	“router”, “peer”, “client”	“router|peer”	Un único valor o una combinación.
		listen		Habilita la escucha y respuesta de mensajes *multicast*.	“true”, “false”	“true”	
	gossip	enabled		Habilita el descubrimiento *gossip*.	“true”, “false”	“true”	
		multihop		Habilita la propagación con múltiples saltos de la información del descubrimiento *gossip*.	“true”, “false”	“false”	Implica un mayor tráfico y menor escalabilidad. Preferible usando enrutamiento *linkstate*.
		autoconnect	router	El tipo de instancia Zenoh con el que un router puede establecer sesiones en el descubrimiento *gossip*.	“router”, “peer”, “client”	“”	Un único valor o una combinación.
			peer	El tipo de instancia Zenoh con el que un par puede establecer sesiones en el descubrimiento *gossip*.	“router”, “peer”, “client”	“router|peer”	Un único valor o una combinación.
timestamping	enabled	router		Establece que los routers deben poner una marca temporal a los mensajes de datos.	“true”, “false”	“true”	
		peer		Establece que los pares deben poner una marca temporal a los mensajes de datos.	“true”, “false”	“false”	
		client		Establece que los clientes deben poner una marca temporal a los mensajes de datos.	“true”, “false”	“false”	
	drop_future_timestamp			Establece que deben tirarse los mensajes con una marca temporal futura.	“true”, “false”	“false”	
queries_default_timeout				El tiempo de espera para las consultas (*ms*).	<time>	10000	
	router	peers_failover_brokering		Establece que el router debe reenviar datos entre dos pares conectados a él al detectar que no están conectados entre sí.	“true”, “false”	“true”	Solo cuando el descubrimiento *gossip* está activado.

Tabla D.1 Continuación de página previa

Parámetro						Significado	Posibles valores	Valor por defecto	Notas
routing	peer	mode				La estrategia de enrutamiento para los pares.	“peer_to_peer”, “linkstate”	“peer_to_peer”	
aggregation	subscribers					Una lista de expresiones de clave a las que se unirán todos los suscriptores.	“<key_expression>”		
	publishers					Una lista de expresiones de clave a las que se unirán todos los publicadores.	“<key_expression>”	—	
	unicast	accept_timeout				El tiempo de espera al abrir un enlace (*ms*).	<time>	10000	
		accept_pending				El número máximo de sesiones pendientes de aceptar.	<sessions>	100	
		max_sessions				El número máximo de sesiones simultáneas.	<sessions>	1000	
		max_links				El número máximo de enlaces permitidos en una sesión.	<links>	1	
		lowlatency				Habilita el transporte de baja latencia.	“true”, “false”	“false”	
	qos	enabled				Habilita el QoS.	“true”, “false”	“true”	
		protocols				La lista de protocolos que se pueden utilizar para aceptar y abrir sesiones en Zenoh.	“tcp”, “udp”, “tls”, “quic”, “ws”, “unixsock-stream”	“tcp”, “udp”, “tls”, “quic”, “ws”, “unixsock-stream”	Un único valor o una combinación.
		tx	sequence_number_resolution			La resolución que se usará en los números de secuencia de los mensajes (*bit*).	“8bit”, “16bit”, “32bit”, “64bit”	“32bit”	Al establecer una sesión con otra instancia, se utilizará el valor más bajo.
			lease			El tiempo que un nodo anuncia su presencia a otros nodos para mantener la conexión activa (*ms*).	<time>	10000	
			keep_alive			El número de mensajes de *KeepAlive* enviados durante el tiempo de *lease*.	<messages>	4	Es recomendable ajustar el tiempo de espera de *KeepAlive* a una cuarta parte del tiempo de *lease*.
			batch_size			El tamaño del lote (*bytes*).	<size>	65535	Tamaño máximo: 65535.
			queue	size	control	El tamaño de la cola de prioridad de paquetes de control.	<size>	1	
					real_time	El tamaño de la cola de prioridad de paquetes de tiempo real.	<size>	1	
					interactive_high	El tamaño de la cola de prioridad de paquetes interactivos de alta prioridad.	<size>	1	
					interactive_low	El tamaño de la cola de prioridad de paquetes interactivos de baja prioridad.	<size>	1	
					data_high	El tamaño de la cola de prioridad de paquetes de datos de alta prioridad.	<size>	2	
					data	El tamaño de la cola de prioridad de paquetes de datos.	<size>	4	
					data_low	El tamaño de la cola de prioridad de paquetes de datos de baja prioridad.	<size>	4	
					background	El tamaño de la cola de prioridad de paquetes de segundo plano.	<size>	4	
				backoff		El tiempo de *backoff* exponencial inicial (*ns*).	<time>	100	Valores elevados conducen a un *batching* más agresivo, pero introduce latencia adicional.

Tabla D.1 Continuación de página previa

Parámetro				Significado	Posibles valores	Valor por defecto	Notas
transport	link	rx	buffer_size	El tamaño del *buffer* de recepción (*bytes*).	<size>	65535	Puede ser incrementado para escenarios con altas tasas de datos.
			max_message_size	El tamaño máximo del *buffer* de desfragmentación en el receptor (*bytes*).	<size>	1073741824	Los mensajes que supereen el tamaño serán descartados.
		tls	root_ca_certificate	La ruta al certificado de la autoridad certificadora.	“<path/to/ca-certificate.pem>”	—	
			server_private_key	La ruta a la clave privada del servidor TLS.	“<path/to/server-private-key.pem>”	—	
			server_certificate	La ruta al certificado público del servidor TLS.	“<path/to/server-certificate.pem>”	—	
			client_auth	Habilita la autenticación del cliente (mTLS).	“true”, “false”	“false”	
			client_private_key	La ruta a la clave privada del cliente TLS.	“<path/to/client-private-key.pem>”	—	
			client_certificate	La ruta al certificado público del cliente TLS.	“<path/to/client-certificate.pem>”	—	
			server_name_verification	Habilita la verificación de nombre de servidor.	“true”, “false”	“true”	
		compression	enabled	Habilita la compresión de datos en cada nodo intermedio.	“true”, “false”	“false”	Deben estar habilitadas *transport_compression* y *unstable*.
	shared_memory	enabled		Habilita la memoria compartida.	“true”, “false”	“false”	
	auth	usrpwd	user	El nombre de usuario para la autenticación.	“<user>”	—	
			password	La contraseña para la autenticación.	“<password>”	—	
			dictionary_file	La ruta al diccionario de contraseñas para la autenticación.	“<path/to/dictionary-file.dict>”	—	
		pubkey	public_key_pem	La clave pública en formato PEM.	“<public_key>”	—	
			private_key_pem	La clave privada en formato PEM.	“<private_key>”	—	
			public_key_file	La ruta a la clave pública.	“<path/to/public-key.pem>”	—	
			private_key_file	La ruta a la clave privada.	“<path/to/private-key.pem>”	—	
			key_size	El tamaño de la clave (*bits*).	<size>	—	
			known_keys_file	La ruta a las claves conocidas.	“<path/to/known-keys.txt>”	—	
adminspace	permissions	read		Habilita la lectura en el espacio de administración.	“true”, “false”	“true”	
		write		Habilita la escritura en el espacio de administración.	“true”, “false”	“false”	

Apéndice E

Códigos para pruebas de latencia

E.1. Zenoh

E.1.1. Nodo inicial

```
import sys
import time
import os
import argparse
import json
import zenoh
import numpy as np
from zenoh import config, Reliability, Sample, CongestionControl

# --- Command line argument parsing --- --- --- --- --- ---
parser = argparse.ArgumentParser(
    prog='z_pub_rtt',
    description='zenoh pub rtt')
parser.add_argument('--mode', '-m', dest='mode',
                    choices=['peer', 'client'],
                    type=str,
                    help='The zenoh session mode.')
parser.add_argument('--connect', '-e', dest='connect',
                    metavar='ENDPOINT',
                    action='append',
                    type=str,
                    help='Endpoints to connect to.')
parser.add_argument('--listen', '-l', dest='listen',
                    metavar='ENDPOINT',
                    action='append',
                    type=str,
                    help='Endpoints to listen on.')
parser.add_argument('--key', '-k', dest='key',
                    default='tfm/rtt',
                    type=str,
                    help='The key expression to publish onto.')
```

```
parser.add_argument('--key_ack', dest='key_ack',
                    default='tfm/ack',
                    type=str,
                    help='The key expression to to obtain the ack.')
parser.add_argument("--iter", dest="iter",
                    type=int,
                    default=1000,
                    help="How many puts to perform")
parser.add_argument('--config', '-c', dest='config',
                    metavar='FILE',
                    type=str,
                    help='A configuration file.')
parser.add_argument('--long_buf', '-b', dest='long_buf',
                    default=10,
                    type=int,
                    help='The longitude of the sent buffer.')
parser.add_argument('--csv', dest='csv_file_path',
                    default="Zenoh_rtt",
                    type=str,
                    help='A log file.')

args = parser.parse_args()
conf = zenoh.Config.from_file(args.config) if args.config is not None else
    zenoh.Config()
if args.mode is not None:
    conf.insert_json5(zenoh.config.MODE_KEY, json.dumps(args.mode))
if args.connect is not None:
    conf.insert_json5(zenoh.config.CONNECT_KEY, json.dumps(args.connect))
if args.listen is not None:
    conf.insert_json5(zenoh.config.LISTEN_KEY, json.dumps(args.listen))
key = args.key
key_ack = args.key_ack
iter = args.iter
long_buf = args.long_buf
buffer = bytearray(long_buf)
csv_file_path = args.csv_file_path

idx = 0

txs = []
txsPl = []
rxs = []
rxsPl = []

def listener(sample: Sample):
    global idx
    idx += 1
    timestamp = int(time.time_ns())
```

```
79      rxs.append(timestamp)
80      rxsPl.append(int.from_bytes(sample.payload[0:2], 'big'))
81      print(f">> [Subscriber] Received {sample.kind} {sample.key_expr} {int.
            from_bytes(sample.payload[0:2], 'big')}")
82
83      if idx < iter:
84          buffer[:2] = idx.to_bytes(2, 'big')
85          print(f"Putting Data ('{key}': '{idx}')...")
86          timestamp = int(time.time_ns())
87          pub.put(bytes(buffer))
88          txs.append(timestamp)
89          txsPl.append(idx)
90          print(timestamp)
91
92  os.system('sudo service ntp restart')
93
94  zenoh.init_logger()
95
96  try:
97      print("Opening session...")
98      session = zenoh.open(conf)
99
100     print(f"Declaring Publisher on '{key}'...")
101     pub = session.declare_publisher(key, congestion_control=
            CongestionControl.BLOCK())
102     # pub = session.declare_publisher(key, congestion_control=
            CongestionControl.DROP())
103     print(f"Declaring Subscriber on '{key_ack}'...")
104     sub = session.declare_subscriber(key_ack, listener, reliability=
            Reliability.RELIABLE())
105     # sub = session.declare_subscriber(key_ack, listener, reliability=
            Reliability.BEST_EFFORT())
106
107     if iter is not None:
108         buffer[:2] = idx.to_bytes(2, 'big')
109         print(f"Putting Data ('{key}': '{idx}')...")
110         timestamp = int(time.time_ns())
111         pub.put(bytes(buffer))
112         txs.append(timestamp)
113         txsPl.append(idx)
114         while idx < iter:
115             time.sleep(0.1)
116 finally:
117     pub.undeclare()
118     sub.undeclare()
119     session.close()
120
121     if len(txs) > len(rxs):
```

```
        diff = len(txs) - len(rxs)
        txs = np.delete(txs, -diff)
        txsPl = np.delete(txsPl, -diff)

    with open(f'{csv_file_path}.csv', 'w') as file:
        file.write(''.join(f'{txsPl[i]} \t {txs[i]} \t {rxsPl[i]} \t  {rxs[
            i]}\n' for i in range(len(txs))))

    aux = np.subtract(rxs, txs)/1e9
    aux.tofile(f'{csv_file_path}_rtt.csv', sep = '\n')
```

E.1.2. Nodo final

```
import sys
import time
import os
import argparse
import json
import zenoh
from zenoh import config, Reliability, Sample, CongestionControl

# --- Command line argument parsing --- --- --- --- --- ---
parser = argparse.ArgumentParser(
    prog='z_sub_rtt',
    description='zenoh sub example')
parser.add_argument('--mode', '-m', dest='mode',
                    choices=['peer', 'client'],
                    type=str,
                    help='The zenoh session mode.')
parser.add_argument('--connect', '-e', dest='connect',
                    metavar='ENDPOINT',
                    action='append',
                    type=str,
                    help='Endpoints to connect to.')
parser.add_argument('--listen', '-l', dest='listen',
                    metavar='ENDPOINT',
                    action='append',
                    type=str,
                    help='Endpoints to listen on.')
parser.add_argument('--key', '-k', dest='key',
                    default='tfm/rtt',
                    type=str,
                    help='The key expression to subscribe to.')
parser.add_argument('--key_ack', dest='key_ack',
                    default='tfm/ack',
                    type=str,
                    help='The key expression to publish onto.')
parser.add_argument('--config', '-c', dest='config',
```

```
                    metavar='FILE',
                    type=str,
                    help='A configuration file.')

args = parser.parse_args()
conf = zenoh.Config.from_file(
    args.config) if args.config is not None else zenoh.Config()
if args.mode is not None:
    conf.insert_json5(zenoh.config.MODE_KEY, json.dumps(args.mode))
if args.connect is not None:
    conf.insert_json5(zenoh.config.CONNECT_KEY, json.dumps(args.connect))
if args.listen is not None:
    conf.insert_json5(zenoh.config.LISTEN_KEY, json.dumps(args.listen))
key = args.key
key_ack = args.key_ack

def listener(sample: Sample):
    pub.put(sample.payload[0:2])
    print(f">> [Subscriber] Received {sample.kind} {sample.key_expr} {int.
        from_bytes(sample.payload[0:2], 'big')}")
    print(f">> [Publisher] Published {sample.kind} {pub.key_expr} {int.
        from_bytes(sample.payload[0:2], 'big')}")

os.system('sudo service ntp restart')

zenoh.init_logger()

print("Opening session...")
session = zenoh.open(conf)

print(f"Declaring Subscriber on '{key}'...")
sub = session.declare_subscriber(key, listener, reliability=Reliability.
    RELIABLE())
# sub = session.declare_subscriber(key, listener, reliability=Reliability.
    BEST_EFFORT())
print(f"Declaring Publisher on '{key_ack}'...")
pub = session.declare_publisher(key_ack, congestion_control=
    CongestionControl.BLOCK())
# pub = session.declare_publisher(key_ack, congestion_control=
    CongestionControl.DROP())

print("Enter 'q' to quit...")
c = '\0'
while c != 'q':
    c = sys.stdin.read(1)
    if c == '':
        time.sleep(1)

```

```
sub.undeclare()
pub.undeclare()
session.close()
```

E.2. MQTT

E.2.1. Cliente inicial

```
import time
import os
import argparse
import itertools
import paho.mqtt.client as mqtt
import logging
import numpy as np

# --- Command line argument parsing --- --- --- --- --- ---
parser = argparse.ArgumentParser(
    prog='MQTT_init',
    description='MQTT initial client')
parser.add_argument('--host', '-b', dest='host',
                    action='append',
                    type=str,
                    help='Hostname or IP address of the remote broker.')
parser.add_argument('--port', '-p', dest='port',
                    default=1883,
                    type=int,
                    help='Network port of the server host to connect to.')
parser.add_argument('--topic', '-t', dest='topic',
                    default='tfm/rtt',
                    type=str,
                    help='Topic that the message should be published on.')
parser.add_argument('--ack_topic', '-r', dest='ack_topic',
                    default='tfm/ack',
                    type=str,
                    help='Topic to subscribe to in order to obtain the ack.
                        ')
parser.add_argument('--qos', '-q', dest='qos',
                    default=0, # 0, 1, 2
                    type=int,
                    help='The quality of service level to use.')
parser.add_argument("--iter", dest="iter",
                    type=int,
                    default=1000,
                    help="How many pubs to perform")
parser.add_argument('--long_buf', '-l', dest='long_buf',
                    default=10,
                    type=int,
```

```
                    help='The longitude of the sent buffer.')
parser.add_argument('--csv', dest='csv_file_path',
                    default="MQTT_rtt",
                    type=str,
                    help='A log file.')

args = parser.parse_args()
host = args.host
port = args.port
topic = args.topic
ack_topic = args.ack_topic
qos = args.qos
long_buf = args.long_buf
csv_file_path = args.csv_file_path
iter = args.iter
buffer = bytearray(long_buf)
idx = 0

txs = []
txsPl = []
rxs = []
rxsPl = []

os.system('sudo service ntp restart')

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def on_connect(client, userdata, flags, rc):
    if rc == 0:
        logger.info("Connected successfully")
    else:
        logger.error(f"Connection failed with code {rc}")

def on_disconnect(client, userdata, rc):
    if rc != 0:
        logger.warning(f"Unexpected disconnection with code {rc}.")
    else:
        logger.info("Disconnected successfully")

def on_subscribe(client, userdata, mid, granted_qos):
    logger.info(f"Subscription completed with QoS = {granted_qos}" )

def on_message(client, userdata, message):
    timestamp = int(time.time_ns())
    rxs.append(timestamp)
    rxsPl.append(int.from_bytes(message.payload[0:2], 'big'))
    logger.info(f"Received {message.topic} {int.from_bytes(message.payload
```

```
        [0:2], 'big')}")
88
89      global idx
90      idx += 1
91      if idx < iter:
92          buffer[:2] = idx.to_bytes(2, 'big')
93          logger.info(f"Putting Data ('{topic}': '{idx}')...")
94          timestamp = int(time.time_ns())
95          client.publish(topic, bytes(buffer), qos, False)
96          txs.append(timestamp)
97          txsPl.append(idx)
98
99  def on_publish(client, userdata, mid):
100     logger.info(f"Message published successfully.")
101
102 if __name__ == "__main__":
103     client = mqtt.Client()
104     client.on_connect = on_connect
105     client.on_publish = on_publish
106     client.on_disconnect = on_disconnect
107     client.on_subscribe = on_subscribe
108     client.on_message = on_message
109
110     try:
111         logger.info("Opening session...")
112         client.connect(host, port, 60)
113         client.subscribe(ack_topic, qos)
114         client.loop_start()
115
116         while not client.is_connected():
117             time.sleep(0.1)
118         if iter is not None:
119             buffer[:2] = idx.to_bytes(2, 'big')
120             logger.info(f"Putting Data ('{topic}': '{idx}')...")
121             timestamp = int(time.time_ns())
122             client.publish(topic, bytes(buffer), qos, False)
123             txs.append(timestamp)
124             txsPl.append(idx)
125             while idx < iter:
126                 time.sleep(0.1)
127
128     except Exception as e:
129         logger.error(f"An error occurred: {e}")
130     finally:
131         logger.info("Closing session...")
132         client.disconnect()
133
134         if len(txs) > len(rxs):
```

```
            diff = len(txs) - len(rxs)
            txs = np.delete(txs, -diff)
            txsPl = np.delete(txsPl, -diff)

        with open(f'{csv_file_path}.csv', 'w') as file:
            file.write(''.join(f'{txsPl[i]} \t {txs[i]} \t {rxs[i]}\n' for
                i in range(len(txs))))

        aux = np.subtract(rxs, txs)/1e9
        aux.tofile(f'{csv_file_path}_rtt.csv', sep = '\n')
```

E.2.2. Cliente final

```
import time
import os
import argparse
import paho.mqtt.client as mqtt
import logging

# --- Command line argument parsing --- --- --- --- --- ---
parser = argparse.ArgumentParser(
    prog='MQTT_final',
    description='MQTT final client')
parser.add_argument('--host', '-b', dest='host',
                    metavar='HOST',
                    action='append',
                    type=str,
                    help='Hostname or IP address of the remote broker.')
parser.add_argument('--port', '-p', dest='port',
                    metavar='PORT',
                    default=1883,
                    type=int,
                    help='Network port of the server host to connect to.')
parser.add_argument('--topic', '-t', dest='topic',
                    default='tfm/rtt',
                    type=str,
                    help='Topic to subscribe to.')
parser.add_argument('--ack_topic', '-r', dest='ack_topic',
                    default='tfm/ack',
                    type=str,
                    help='topic that the message should be published on in
                        order to send the ack.')
parser.add_argument('--qos', '-q', dest='qos',
                    metavar='QOS',
                    default=0, # 0, 1, 2
                    type=int,
                    help='The quality of service level to use.')

```

```
args = parser.parse_args()
host = args.host
port = args.port
topic = args.topic
ack_topic = args.ack_topic
qos = args.qos

os.system('sudo service ntp restart')

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def on_connect(client, userdata, flags, rc):
    if rc == 0:
        logger.info("Connected successfully")
    else:
        logger.error(f"Connection failed with code {rc}")

def on_disconnect(client, userdata, rc):
    if rc != 0:
        logger.warning(f"Unexpected disconnection with code {rc}.")
    else:
        logger.info("Disconnected successfully")

def on_subscribe(client, userdata, mid, granted_qos):
    logger.info(f"Subscription completed with QoS = {granted_qos}")

def on_message(client, userdata, message):
    client.publish(ack_topic, message.payload[0:2], qos, False)
    logger.info(f"Message published {message.topic} {int.from_bytes(message
        .payload[0:2], 'big')}")

def on_publish(client, userdata, mid):
    logger.info(f"Message published successfully.")

if __name__ == "__main__":
    client = mqtt.Client()
    client.on_connect = on_connect
    client.on_subscribe = on_subscribe
    client.on_message = on_message
    client.on_disconnect = on_disconnect
    client.on_publish = on_publish

    try:
        logger.info("Opening session...")
        client.connect(host, port, 60)
        client.subscribe(topic, qos)
        client.loop_forever()
```

```
82
83      except Exception as e:
84          logger.error(f"An error occurred: {e}")
85      finally:
86          logger.info("Closing session...")
87          client.disconnect()
```

E.2.3. Cliente puente[1]

```
1  import time
2  import os
3  import argparse
4  import itertools
5  import paho.mqtt.client as mqtt
6  import logging
7  import numpy as np
8
9  # --- Command line argument parsing --- --- --- --- --- ---
10 parser = argparse.ArgumentParser(
11     prog='MQTT_bridge',
12     description='MQTT bridge')
13 parser.add_argument('--host_loc', dest='host_local',
14                     action='append',
15                     type=str,
16                     help='Hostname or IP address of the local broker.')
17 parser.add_argument('--host_rem', dest='host_rem',
18                     action='append',
19                     type=str,
20                     help='Hostname or IP address of the remote broker.')
21 parser.add_argument('--port', '-p', dest='port',
22                     default=1883,
23                     type=int,
24                     help='Network port of the server host to connect to.')
25 parser.add_argument('--topic', '-t', dest='data_topic',
26                     default='tfm/rtt',
27                     type=str,
28                     help='Topic of the message.')
29 parser.add_argument('--ack_topic', '-r', dest='ack_topic',
30                     default='tfm/ack',
31                     type=str,
32                     help='Topic of the ack.')
33 parser.add_argument('--qos', '-q', dest='qos',
34                     default=0, # 0, 1, 2
35                     type=int,
36                     help='The quality of service level to use.')
37
38 args = parser.parse_args()
```

[1] Solo para escenario γ.

```
host_loc = args.host_local
host_rem = args.host_rem
port = args.port
data_topic = args.data_topic
ack_topic = args.ack_topic
qos = args.qos

os.system('sudo service ntp restart')

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def on_connect(client, userdata, flags, rc):
    if rc == 0:
        logger.info("Connected successfully")
    else:
        logger.error(f"Connection failed with code {rc}")

def on_disconnect(client, userdata, rc):
    if rc != 0:
        logger.warning(f"Unexpected disconnection with code {rc}.")
    else:
        logger.info("Disconnected successfully")

def on_subscribe(client, userdata, mid, granted_qos):
    logger.info(f"Subscription completed with QoS = {granted_qos}" )

def on_message_loc(client, userdata, message):
    logger.info(f"Received {message.topic} {int.from_bytes(message.payload
        [0:2], 'big')}")
    logger.info(f"Putting Data ('{data_topic}': '{int.from_bytes(message.
        payload[0:2], 'big')}')...")
    client_rem.publish(data_topic, bytes(message.payload), qos, False)

def on_message_rem(client, userdata, message):
    logger.info(f"Received {message.topic} {int.from_bytes(message.payload
        [0:2], 'big')}")
    logger.info(f"Putting Data ('{ack_topic}': '{int.from_bytes(message.
        payload[0:2], 'big')}')...")
    client_loc.publish(ack_topic, bytes(message.payload), qos, False)

def on_publish(client, userdata, mid):
    logger.info(f"Message published successfully.")

if __name__ == "__main__":
    client_loc = mqtt.Client()
    client_rem = mqtt.Client()

```

```
    client_loc.on_connect = on_connect
    client_loc.on_publish = on_publish
    client_loc.on_disconnect = on_disconnect
    client_loc.on_subscribe = on_subscribe
    client_loc.on_message = on_message_loc

    client_rem.on_connect = on_connect
    client_rem.on_publish = on_publish
    client_rem.on_disconnect = on_disconnect
    client_rem.on_subscribe = on_subscribe
    client_rem.on_message = on_message_rem

    try:
        logger.info("Opening session...")

        client_loc.connect(host_loc, port, 60)
        client_rem.connect(host_rem, port, 60)

        client_loc.subscribe(data_topic, qos)
        client_rem.subscribe(ack_topic, qos)

        client_loc.loop_start()
        client_rem.loop_start()

        while not client_loc.is_connected() or not client_rem.is_connected
            ():
            time.sleep(0.1)

    except Exception as e:
        logger.error(f"An error occurred: {e}")
    finally:
        logger.info("Closing session...")
        client_loc.disconnect()
        client_rem.disconnect()
```

Apéndice F

Códigos para pruebas de fiabilidad y control de congestión

F.1. Zenoh

F.1.1. Nodo inicial

```
import sys
import time
import os
import argparse
import json
import zenoh
import numpy as np
from zenoh import config, Reliability, Sample, CongestionControl

# --- Command line argument parsing --- --- --- --- --- ---
parser = argparse.ArgumentParser(
    prog='z_pub_rel',
    description='zenoh pub rel')
parser.add_argument('--mode', '-m', dest='mode',
                    choices=['peer', 'client'],
                    type=str,
                    help='The zenoh session mode.')
parser.add_argument('--connect', '-e', dest='connect',
                    metavar='ENDPOINT',
                    action='append',
                    type=str,
                    help='Endpoints to connect to.')
parser.add_argument('--listen', '-l', dest='listen',
                    metavar='ENDPOINT',
                    action='append',
                    type=str,
                    help='Endpoints to listen on.')
parser.add_argument('--key', '-k', dest='key',
```

```
                    default='tfm/rel',
                    type=str,
                    help='The key expression to publish onto.')
parser.add_argument('--key_ack', dest='key_ack',
                    default='tfm/ack',
                    type=str,
                    help='The key expression to to obtain the ack.')
parser.add_argument("--iter", dest="iter",
                    type=int,
                    default=5000, # 50000, 10000, 5000 packets
                    help="How many puts to perform")
parser.add_argument('--config', '-c', dest='config',
                    metavar='FILE',
                    type=str,
                    help='A configuration file.')
parser.add_argument('--long_buf', '-b', dest='long_buf',
                    default=1000, # 100, 500, 1000 Bytes
                    type=int,
                    help='The longitude of the sent buffer.')
parser.add_argument('--csv', dest='csv_file_path',
                    default="rel_zenoh",
                    type=str,
                    help='A log file.')

args = parser.parse_args()
conf = zenoh.Config.from_file(args.config) if args.config is not None else
    zenoh.Config()
if args.mode is not None:
    conf.insert_json5(zenoh.config.MODE_KEY, json.dumps(args.mode))
if args.connect is not None:
    conf.insert_json5(zenoh.config.CONNECT_KEY, json.dumps(args.connect))
if args.listen is not None:
    conf.insert_json5(zenoh.config.LISTEN_KEY, json.dumps(args.listen))
key = args.key
key_ack = args.key_ack
iter = args.iter
long_buf = args.long_buf
buffer = bytearray(long_buf)
csv_file_path = args.csv_file_path

idx = 0

txs = []
txsPl = []
rxs = []
rxsPl = []

def listener(sample: Sample):
```

```
    timestamp = int(time.time_ns())
    rxs.append(timestamp)
    rxsPl.append(int.from_bytes(sample.payload[0:2], 'big'))
    print(f">> [Subscriber] Received {sample.kind} {sample.key_expr} {int.
        from_bytes(sample.payload[0:2], 'big')}")

os.system('sudo service ntp restart')

zenoh.init_logger()

try:
    print("Opening session...")
    session = zenoh.open(conf)

    print(f"Declaring Publisher on '{key}'...")
    pub = session.declare_publisher(key, congestion_control=
        CongestionControl.BLOCK())
    # pub = session.declare_publisher(key, congestion_control=
        CongestionControl.DROP())
    print(f"Declaring Subscriber on '{key_ack}'...")
    sub = session.declare_subscriber(key_ack, listener, reliability=
        Reliability.RELIABLE())
    # sub = session.declare_subscriber(key_ack, listener, reliability=
        Reliability.BEST_EFFORT())
    time.sleep(2)

    while idx < iter:
        buffer[:2] = idx.to_bytes(2, 'big')
        print(f"Putting Data ('{key}': '{idx}')...")
        timestamp = int(time.time_ns())
        pub.put(bytes(buffer))
        txs.append(timestamp)
        txsPl.append(idx)
        idx+=1
    time.sleep(1)

finally:
    pub.undeclare()
    sub.undeclare()
    session.close()

    with open(f'{csv_file_path}_pub.csv', 'w') as file:
        file.write(''.join(f'{txsPl[i]}\n' for i in range(len(txs))))

    with open(f'{csv_file_path}_sub.csv', 'w') as file:
        file.write(''.join(f'{rxsPl[i]}\n' for i in range(len(rxs))))

    with open(f'{csv_file_path}_time_pub.csv', 'w') as file:
```

```
119         file.write(''.join(f'{txs[i]}\n' for i in range(len(txs))))
120 
121     with open(f'{csv_file_path}_time_sub.csv', 'w') as file:
122         file.write(''.join(f'{rxs[i]}\n' for i in range(len(rxs))))
```

F.1.2. Nodo final

* Igual que E.1.2, salvo "`key`" (`tfm/rel`).

Bibliografía

[1] M. Soori, F. Karimi Ghaleh Jough, R. Dastres y B. Arezoo. *Connectivity, automation, and data exchange in advanced manufacturing of Industry 4.0*. 2024. DOI: 10.13140/RG.2.2.18189.55522.

[2] B. S. Khan, S. Jangsher, A. Ahmed y A. Al-Dweik. «URLLC and eMBB in 5G Industrial IoT: A Survey». En: *IEEE Open Journal of the Communications Society* 3 (2022), págs. 1134-1163. DOI: 10.1109/OJCOMS.2022.3189013.

[3] ETSI. *3GPP; Study on scenarios and requirements for next generation access technologies (3GPP TR 38.913 version 14.3.0 Release 14)*. Technical Report ETSI TR 138 913 V14.3.0. Accessed: 2024-06-07. European Telecommunications Standards Institute (ETSI), 2017. URL: https://www.etsi.org/deliver/etsi_tr/138900_138999/138913/14.03.00_60/tr_138913v140300p.pdf.

[4] W.-P. Nwadiugwu, C. I. Nwakanma, J.-M. Lee y D.-S. Kim. «Achieving Reliable URLLC-based Network in Industrial and Military Embedded Systems». En: *2019 International Conference on Information and Communication Technology Convergence (ICTC)*. 2019, págs. 642-645. DOI: 10.1109/ICTC46691.2019.8939838.

[5] Larrañaga, Ana and Lucas-Estañ, M. Carmen and Martinez, Imanol and Gozalvez, Javier. «5G Configured Grant Scheduling for 5G-TSN Integration for the Support of Industry 4.0». En: *2023 18th Wireless On-Demand Network Systems and Services Conference (WONS)*. 2023, págs. 72-79. DOI: 10.23919/WONS57325.2023.10062219.

[6] L. Miao, S.-F. Chen, Y.-L. Hsu y K.-L. Hua. «How Does C-V2X Help Autonomous Driving to Avoid Accidents?» En: *Sensors* 22.2 (2022). DOI: 10.3390/s22020686. URL: https://www.mdpi.com/1424-8220/22/2/686.

[7] L. Martino, J. Deutschmann, K.-S. Hielscher y R. German. *Towards a 5G Satellite Communication Framework for V2X*. workingpaper. 2023. DOI: 10.25972/OPUS-32214.

[8] A. Sayeed, C. Verma, N. Kumar, N. Koul y Z. Illés. «Approaches and Challenges in Internet of Robotic Things». En: *Future Internet* 14.9 (2022). DOI: 10.3390/fi14090265. URL: https://www.mdpi.com/1999-5903/14/9/265.

[9] L. Altomonte. *An Introduction to Collaborative Robotics*. SafetyCulture. Accessed: 2024-06-07. 2023. URL: https://safetyculture.com/topics/collaborative-robotics/.

[10] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng et al. «ROS: an open-source Robot Operating System». En: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, pág. 5.

[11] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. R. Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke y E. F. Perdomo. «ros_control: A generic and simple control framework for ROS». En: *Journal of Open Source Software* 2.20 (2017), pág. 456. DOI: 10.21105/joss.00456. URL: https://doi.org/10.21105/joss.00456.

[12] Eclipse Foundation. *Zenoh*. https://zenoh.io/. 2024.

[13] A. Corsaro, L. Cominardi, O. Hecart, G. Baldoni, J. Avital, J. Loudet, C. Guimares, M. Ilyin y D. Bannov. «Zenoh: Unifying Communication, Storage and Computation from the Cloud to the Microcontroller». En: *2023 26th Euromicro Conference on Digital System Design (DSD)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023, págs. 422-428. DOI: 10.1109/DSD60849.2023.00065. URL: https://doi.ieeecomputersociety.org/10.1109/DSD60849.2023.00065.

[14] Object Management Group (OMG). *Data Distribution Service (DDS)*. https://www.omg.org/spec/DDS/. Version 1.4. 2015.

[15] ROS 2 Core Team. *ROS 2 Middleware Interface (RMW) alternate*. Study Item. Open Source Robotics Alliance (OSRA), 2023.

[16] ZettaScale. *Redefining Automotive Connectivity with Zenoh*. Blog. 2024. URL: https://www.zettascale.tech/news/redefining-automotive-connectivity-with-zenoh/.

[17] Focus Group on AI for autonomous and assisted driving (FG-AI4AD). *ITU-T Automated driving safety data protocol – Specification*. Inf. téc. TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU, 2022.

[18] A. Banks y R. Gupta. *MQTT Version 3.1.1*. Inf. téc. Latest version: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html. OASIS Standard, 2014. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[19] A. Banks, E. Briggs, K. Borgendale y R. Gupta. *MQTT Version 5.0*. Inf. téc. Latest version: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. OASIS Standard, 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

[20] W. Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. 2022. DOI: 10.17487/RFC9293. URL: https://www.rfc-editor.org/info/rfc9293.

[21] *User Datagram Protocol*. RFC 768. 1980. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/info/rfc768.

[22] Z. Shelby, K. Hartke y C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. 2014. DOI: 10.17487/RFC7252. URL: https://www.rfc-editor.org/info/rfc7252.

[23] Apache Software Foundation. *Apache Kafka*. https://kafka.apache.org/. 2024.

[24] T. Anitha, S. Manimurugan, S. Sridhar, S. Mathupriya y G. C. P. Latha. «A Review on Communication Protocols of Industrial Internet of Things». En: *2022 2nd International Conference on Computing and Information Technology (ICCIT)*. 2022, págs. 418-423. DOI: 10.1109/ICCIT52419.2022.9711544.

[25] C. Bayılmış, M. A. Ebleme, Ü. Çavuşoğlu, K. Küçük y A. Sevin. «A survey on communication protocols and performance evaluations for Internet of Things». En: *Digital Communications and Networks* 8.6 (2022), págs. 1094-1104. DOI: https://doi.org/10.1016/j.dcan.2022.03.013. URL: https://www.sciencedirect.com/science/article/pii/S2352864822000347.

[26] MQTT.org. *MQTT: The Standard for IoT Messaging*. https://mqtt.org/. Accessed: April 24, 2024.

[27] M. Barón Herrá. *Evaluación de las prestaciones de un entorno 5G para la Internet táctil*. 2022. URL: https://hdl.handle.net/10902/25905.

[28] *Internet Protocol*. RFC 791. 1981. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/info/rfc791.

[29] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[30] F. Fernández, M. Zverev, P. Garrido, J. R. Juárez, J. Bilbao y R. Agüero. «And QUIC meets IoT: performance assessment of MQTT over QUIC». En: *2020 16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. 2020, págs. 1-6. DOI: 10.1109/WiMob50308.2020.9253384.

[31] J. Iyengar y M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. 2021. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000.

[32] C. Guimarães y G. Baldoni. *There is Land Besides IP: How to Cross It with Zenoh*. https://zenoh.io/blog/2022-08-12-zenoh-serial/. 2022.

[33] W.-Y. Liang, Y. Yuan y H.-J. Lin. *A Performance Study on the Throughput and Latency of Zenoh, MQTT, Kafka, and DDS*. 2023.

[34] C.-S. Shih, H.-J. Lin, Y. Yuan, Y.-H. Kuo y W.-Y. Liang. «Scalable and Bounded-time Decisions on Edge Device Network using Eclipse Zenoh». En: *2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2022, págs. 170-179. DOI: 10.1109/RTCSA55878.2022.00024.

[35] J. Jun, P. Peddabachagari y M. Sichitiu. «Theoretical Maximum Throughput of IEEE 802.11 and its Applications». En: *Proceedings of the Second IEEE International Symposium on Network Computing and Applications*. NCA '03. USA: IEEE Computer Society, 2003, pág. 249.

[36] J. Zhang, X. Yu, S. Ha, J. P. Queralta y T. Westerlund. *Comparison of DDS, MQTT, and Zenoh in Edge-to-Edge and Edge-to-Cloud Communication for Distributed ROS 2 Systems*. 2023.

[37] ZeroTier. *ZeroTier*. https://www.zerotier.com/. Accessed: 2024-06-07.

[38] J. J. López Escobar, R. P. Díaz-Redondo y F. Gil-Castiñeira. «Unleashing the power of decentralized serverless IoT dataflow architecture for the Cloud-to-Edge Continuum: a performance comparison». En: *Annals of Telecommunications* 79.3 (2024), págs. 135-148. DOI: 10.1007/s12243-023-01009-x. URL: https://doi.org/10.1007/s12243-023-01009-x.

[39] R. Teixeira, G. Baldoni, M. Antunes, D. Gomes y R. L. Aguiar. «Leveraging Decentralized Communication for Privacy-Preserving Federated Learning in 6G Networks». En: *SSRN Electronic Journal* (2024). DOI: 10.2139/ssrn.4817067. URL: https://ssrn.com/abstract=4817067.

[40] G. Baldoni, J. Loudet, L. Cominardi, A. Corsaro e Y. He. «Zenoh-based Dataflow Framework for Autonomous Vehicles». En: *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 2021, págs. 555-560. DOI: 10.1109/QRS-C55045.2021.00085.

[41] G. Baldoni, J. Loudet, L. Cominardi, A. Corsaro e Y. He. «Facilitating distributed data-flow programming with Eclipse Zenoh: the ERDOS case». En: *Proceedings of the 1st Workshop on Serverless Mobile Networking for 6G Communications*. MobileServerless'21. Virtual, WI, USA: Association for Computing Machinery, 2021, págs. 13-18. DOI: 10.1145/3469263.3469858. URL: https://doi.org/10.1145/3469263.3469858.

[42] M. Gramaglia, M. Camelo, L. Fuentes, J. Ballesteros, G. Baldoni, L. Cominardi, A. Garcia-Saavedra y M. Fiore. «Network Intelligence for Virtualized RAN Orchestration: The DAEMON Approach». En: *2022 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*. 2022, págs. 482-487. DOI: 10.1109/EuCNC/6GSummit54941.2022.9815816.

[43] DNSstuff. *Network Intelligence*. https://www.dnsstuff.com/network-intelligence. Accessed: 2024-06-07. 2019.

[44] T. Zhang, C. Xue, J. Wang, Z. Yun, N. Lin y S. Han. *A Survey on Industrial Internet of Things (IIoT) Testbeds for Connectivity Research*. 2024.

[45] O. Salman, I. Elhajj, A. Kayssi y A. Chehab. «An architecture for the Internet of Things with decentralized data and centralized control». En: *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*. 2015, págs. 1-8. DOI: 10.1109/AICCSA.2015.7507265.

[46] Raymond Andrè Hagen. *The Future of IT: A Balanced Perspective on Centralized and Decentralized Systems*. Accessed: 2024-05-27. 2023. URL: https://www.linkedin.com/pulse/future-balanced-perspective-centralized-decentralized-hagen/.

[47] H. Nielsen, R. T. Fielding y T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. 1996. DOI: 10.17487/RFC1945. URL: https://www.rfc-editor.org/info/rfc1945.

[48] R. T. Fielding, M. Nottingham y J. Reschke. *HTTP/1.1*. RFC 9112. 2022. DOI: 10.17487/RFC9112. URL: https://www.rfc-editor.org/info/rfc9112.

[49] M. Thomson y C. Benfield. *HTTP/2*. RFC 9113. 2022. DOI: 10.17487/RFC9113. URL: https://www.rfc-editor.org/info/rfc9113.

[50] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva y M. Leone. «Logical Physical Clocks». En: *Principles of Distributed Systems*. Ed. por M. K. Aguilera, L. Querzoni y M. Shapiro. Cham: Springer International Publishing, 2014, págs. 17-32.

[51] Y. Liu, M. Kashef, K. Lee, L. Benmohamed y R. Candell. «Wireless Network Design for Emerging IIoT Applications: Reference Framework and Use Cases». En: *Proceedings of the IEEE* 107 (2019). DOI: 10.1109/JPROC.2019.2905423. URL: https://doi.org/10.1109/JPROC.2019.2905423.

[52] ZettaScale Zenoh team. *Zenoh API Reference*. https://zenoh-python.readthedocs.io/en/0.10.0-rc/. Ver. 0.10.0-rc. Accessed: 2024-06-20. 2023.