Sistema de comunicaciones DDS entre robots en ROS 2

Victor Fusalba Raña

Julio 2023

Resumen–Este trabajo se enfoca en la comparación de las distintas implementaciones del estándar de comunicación Data Distribution Service (DDS) en robots que utilizan la tecnología Robot Operating System 2 (ROS2). El objetivo principal es analizar las diferencias y similitudes en términos de rendimiento, escalabilidad, confiabilidad y facilidad de integración, entre las diferentes implementaciones DDS. Los resultados de esta comparación proporcionarán información valiosa para la selección y optimización de la implementación DDS más adecuada para un entorno particular de robots que utilizan ROS2.

Paraules clau– ROS2, DDS, RTPS, arquitectura, middleware, interoperabilidad, robot.

Abstract– This work focuses on comparing the different implementations of the Data Distribution Service (DDS) communication standard among robots that use Robot Operating System 2 (ROS2). The main objective is to analyze the differences and similarities in terms of performance, scalability, reliability, and ease of integration among the different DDS implementations. The results of this comparison will provide valuable information for selecting and optimizing the most suitable DDS implementation for a particular environment of robots using ROS2.

Keywords– ROS2, DDS, RTPS, architecture, middleware, interoperability, robot.

1 Motivación del trabajo

1.1. Introducción

Como trabajo se quiere realizar el estudio de las diferentes implementaciones DDS que nos ofrece el nuevo framework ROS2. El motivo principal es valorar de entre todas las implementaciones cual es la más óptima y eficiente en cada uno de los diferentes estudios que analizaremos, viendo si hay alguna que destaque por encima de las demás. Incluso, analizaremos posibles alternativas a DDS, viendo sus ventajas e inconvenientes respecto a este middleware. Estudiaremos los diferentes conceptos como lo son ROS2, DDS, RTPS, entre otros. Para llevar a cabo este trabajo, Moovo Robotics proporcionará el soporte necesario para realizar los casos prácticos, colaborando en la realización de las pruebas precisas, a partir de las que interpretaremos los resultados para obtener una respuesta adecuada al análisis ejecutado.

• E-mail de contacte: victor.fusalba@autonoma.cat
• Menció realitzada: Tecnologia de la Informació
• Treball tutoritzat per: Bernat Gaston
• Curs 2022/23

1.2. Movvo Robotics

Movvo es una empresa que desarrolla la automatización de procesos intralogísticos por medio de soluciones tecnológicas. Podemos decir que está compuesta por tres divisiones:

- Movvo Contenedores.
- Movvo Móviles.
- Movvo Robotics, que nos dará soporte.

Esta última ofrece múltiples soluciones punteras en movilidad intralogística, incluyendo desde vehículos autónomos para el transporte de cargas hasta sistemas de automatización para vehículos estándar. Movvo Robotics proporcionará el material necesario para realizar las pruebas. Para ello, pondrá a nuestra disposición robots para realizar la parte práctica del estudio.

1.3. Objetivos

Con este proyecto se persiguen varios objetivos, siendo uno de los principales las conclusiones que se deriven del resultado del experimento práctico.

- Analizar en profundidad el sistema ROS2.

- Aprender sobre el concepto DDS.
- Estudiar las diferentes implementaciones DDS.
- Realizar un caso práctico que consistirá en la observación del comportamiento de las implementaciones DDS y posibles alternativas, sobre diferentes entornos (LAN, WIFI, VPN).
- Estudiar posibles alternativas de DDS.

1.4. Beneficios

Los beneficios que esperamos obtener con el desarrollo de este proyecto, sin tener en cuenta el aprendizaje de diferentes tecnologías como ROS, RTPS, DDS, entre otros conocimientos, están fundamentalmente orientados a las actividades de Movvo Robotics. Así, la empresa dispondrá de la posibilidad de aplicar los resultados obtenidos utilizando la implementación DDS más adecuada en el desarrollo de futuros proyectos prácticos de la empresa.

1.5. Metodologia

Para garantizar la adecuada ejecución del proyecto, iniciaremos examinando todas las tecnologías mencionadas. Una vez adquiramos un sólido entendimiento de ROS2, DDS, RTPS y posibles alternativas, nos enfocaremos en llevar a cabo el caso práctico. Durante este proceso, realizaremos una comparación exhaustiva basada en métricas predefinidas de las distintas implementaciones estudiadas con anterioridad. Una vez concluido el proyecto, procederemos a elaborar una conclusión detallada en la que analizaremos los resultados obtenidos, evaluaremos la efectividad de las tecnologías utilizadas y destacaremos los puntos clave del caso práctico.

2 ROS

2.1. Introducción

ROS o también conocido como Sistema Operativo Robótico es un framework y un conjunto de herramientas de código abierto desarrolladas originalmente en el 2007 por el Laboratorio de Inteligencia Artificial de Standford.

Esta tecnología nació por la problemática que comportaba la escasa reutilización de código entre los diferentes robots, ya que cada uno disponía de su propio sistema operativo y lenguaje de programación. Como consecuencia de la escasa reutilización de código, los investigadores centraban sus esfuerzos en desarrollar su prototipo de robot en lugar de realizar posibles innovaciones, de tal forma que el elevado coste (de tiempo) invertido en la construcción del robot obstaculizaba gravemente los avances en el ámbito teórico académico. El uso de un código exclusivo para cada robot no era el único inconveniente, sino que también había que aprender a utilizar distintas APIs, además de los desafíos y dificultades que se podían plantear en la fabricación del robot.

Detectado el problema anteriormente mencionado, dos estudiantes de la Universidad de Standford, Keenan Wyrobek y Eric Berger, conscientes de que los investigadores se pasaban mucho más tiempo reescribiendo códigos que innovando, propusieron una solución: ROS. El éxito de dicha propuesta hizo que rápidamente el interés sobre esta tecnología incrementara de forma considerable. El objetivo principal era que ROS fuera útil para muchos tipos de robots, por lo que se centraron en definir niveles de abstracción que permitieran reutilizar gran parte del software. [1]

En la actualidad, ROS brinda un amplio grupo de funcionalidades de un sistema operativo en un clúster de computadoras heterogéneas. Aunque su denominación puede hacernos pensar inicialmente que su uso es específico para robots, no es el caso como veremos a continuación. Su utilidad va mucho más allá, ya que la mayoría de las herramientas que nos proporciona están enfocadas en trabajar con hardware periférico [2]. Este framework nos proporciona una gran cantidad de funcionalidades; entre éstas encontramos la abstracción de hardware, control de dispositivos a bajo nivel, implementación de funcionalidades comunes, comunicación entre procesos en diferentes máquinas, herramientas para pruebas y visualización, manejo de paquetes, entre otras muchas más.

El motivo principal del uso de ROS es la forma cómo este sistema ejecuta el software y cómo se comunica. Gracias a ello, se facilita considerablemente el diseño de software complejo sin necesidad de saber cómo funciona exactamente el hardware. ROS proporciona una forma de conectar una red de procesos – nodos – con un eje central. Los nodos se pueden ejecutar en múltiples dispositivos y pueden conectarse a este eje de diversas maneras. Este framework está basado en una red de procesos peer-to-peer (red entre iguales) que se acoplan sencillamente por medio de las diferentes opciones de comunicación que nos proporciona esta tecnología. La primera versión de ROS se comunicaba a través de un sistema de mensajería basado en TCP/IP utilizando el formato de mensajes ROS. En este sistema, los diferentes nodos se comunican a través de mensajes intercambiados entre ellos. Cuando un nodo desea enviar un mensaje a otro nodo, primero debe registrar su tipo de mensaje en el ROS Master. Luego, el nodo que desea recibir el mensaje se suscribe al ROS Master indicando el tipo de mensaje que espera recibir. Una vez que el nodo que envía el mensaje publica el mensaje en el ROS Master, el ROS Master redirige el mensaje al nodo suscrito correspondiente. También hay q que tener en cuenta que ROS puede proporcionar distintos servicios de comunicación como por ejemplo comunicaciones del estilo RPC sincrónico, transmisiones de información asincrónica, guardado de datos en un servidor de parámetros... [3]

2.2. ROS2

2.2.1. Introducción

Como ya hemos adelantado en el epígrafe anterior, lo que se buscaba en la creación de ROS era construir un sistema que nos permitiera reutilizar el máximo código posible. Sin embargo, la creación de este sistema se hizo de una forma muy específica, lo que conllevó que se utilizara principalmente en proyectos académicos. Al ser un producto muy concreto, su introducción en la industria ha sido demasiado costosa. Esto se debe a que en el ámbito comercial había muchos requerimientos nuevos que en algunos casos ne-

cesitaba de diferentes parches para satisfacerlos. Como el proceso del uso de ROS en proyectos comerciales no era especialmente óptimo, se pensó en implementar una nueva versión que satisfaciera de forma más adecuada estos requerimientos, dando mayor soporte en su uso comercial e industrial. Esta nueva versión de ROS se empezó a implementar en el año 2014 y fue denominada ROS2.

Este nuevo framework no es un sistema totalmente nuevo, es decir, está desarrollado a partir de ROS, por lo tanto, muchos de los conceptos de ROS se encuentran en ROS2. Por este motivo, muchas aplicaciones que inicialmente fueron implementadas desde ROS han sido portadas a ROS2. A pesar de ello, no se considera como una nueva versión como tal, por lo que no mantiene compatibilidad con aplicaciones que usaban el primer framework. Esto significa que en los casos que se ha querido migrar a ROS2 se ha tenido que actualizar el código desarrollado en ROS.

2.2.2. Arquitectura

Middleware:

Tal como se ha comentado, ROS2 está implementado a partir de ROS. Esto significa que hay muchos conceptos del 'nuevo' framework que son iguales al del anterior. Aun así, ha recibido fuertes cambios, siendo de destacar por su importancia la selección del middleware end-to-end DDS para la capa de comunicación. La principal ventaja de usar este middleware es que ROS 2 puede aprovechar una implementación existente y bien desarrollada de este estándar. Igualmente, cabe destacar que un gran beneficio de utilizar este tipo de middleware end-to-end es que aparte de que hay mucho menos código que mantener, hay una gran documentación que contiene el comportamiento y especificaciones del mismo. [4]

Este middleware ha fortalecido de forma considerable la comunicación entre distintos componentes de los robots gracias a su arquitectura distribuida de publicador/suscriptor. Es decir, ROS2 dispone de una arquitectura del sistema en tiempo real donde encontramos distintos componentes de los robots – nodos – como son los sensores, controladores de movimiento, algoritmos, etc. El middleware del que dispone ROS2 permite el intercambio de datos de estos componentes para que se puedan comunicar entre sí en un entorno descentralizado.

En la actualidad, ROS2 ha sido adaptado a distintos productos DDS. Esto ha sido posible gracias a que ROS2 es de código abierto y, por lo tanto, distintas empresas han ofrecido bibliotecas DDS, también de código abierto, para el sector. Ésta ha sido seguramente una de las causas de los múltiples usos que tienen actualmente ROS2, los cuales van desde comunicación multirobot, comunicación en tiempo real e incluso vehículos autónomos. En las versiones actuales de ROS2, Fast DDS viene como la versión estándar de DDS. Las diferentes bibliotecas DDS que hay disponibles pueden ser escogidas por el propio usuario, pudiendo usar más de una a la vez en un mismo proyecto.

Interfaz de abstracción del middleware:

ROS2 no usa directamente el middleware DDS sino que proporciona su propia capa de abstracción por encima de DDS, llamada rmw. Rmw es una interfaz de primitivas de middleware que son utilizadas por las API ROS de nivel superior. Esta interfaz está disponible para diferentes productos DDS. [5]

El motivo por el que ROS2 utiliza esta interfaz es ofrecer una abstracción para las distintas especificaciones de las implementaciones DDS.

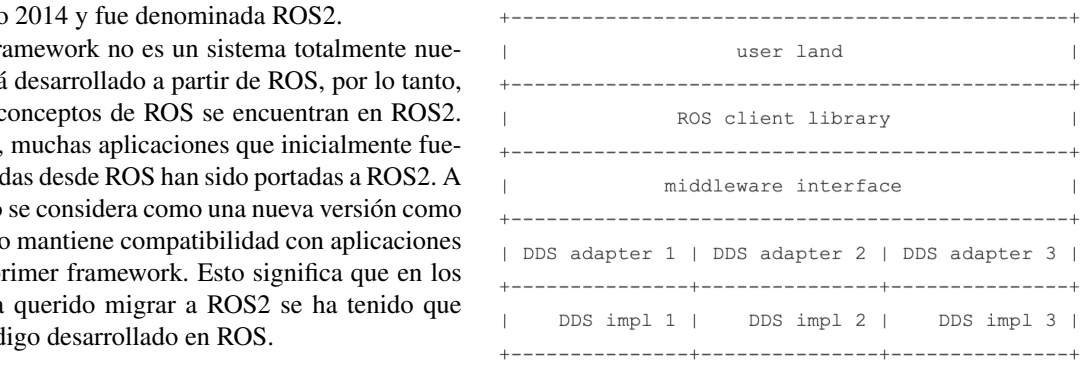


Fig 1. Esquema arquitectura ROS 2 [6]

Otro motivo por el que es necesario usar esta interfaz es evitar que la implementación DDS exponga algún detalle al usuario, es decir, esta interfaz se encargará de evitar que ningún código DDS llegue al usuario.

Tal y como se ha comentado, por encima de la interfaz se trabaja con estructuras de datos ROS. Por tanto, esta interfaz se encargará de traducir a la implementación DDS los datos ROS que tendrán un formato de datos personalizado y le serán facilitados por la biblioteca del cliente. Por otro lado, también deberá traducir los datos a la inversa, es decir, aquellos que le sean facilitados por la implementación DDS los traducirá a objetos de datos ROS. [6]

2.2.3. QoS

Aún disponiendo de un gran abanico de configuraciones DDS para permitir un buen control de la QoS, ROS sólo da soporte a un grupo de éstas. Los usuarios del framework pueden especificar distintos valores como el historial, la profundidad, la confiabilidad y la durabilidad a partir de una estructura de configuración de calidad del servicio.

A medida que han ido pasando los años aumentaba la necesidad de mejorar el soporte para nuevas configuraciones QoS de DDS debido al aumento de la complejidad de las aplicaciones. Por este motivo, se definieron distintas políticas, como por ejemplo, fecha límite, vivacidad, esperanza de vida, etc. Estas políticas están basadas en políticas QoS de DDS. [7]

Hoy en día, la selección de una QoS es uno de los motivos más comunes de problemas en cuanto a la comunicación en ROS2.

3 DDS

3.1. Introducción

Como hemos ido viendo durante el proyecto, DDS es un concepto muy importante para ROS2. A continuación, profundizaremos sobre este concepto para entender bien su funcionalidad dentro de nuestro framework.

DDS o también conocido como Servicio de distribución de datos para sistemas en tiempo real es la especificación

de la API de un middleware de comunicación de tipo publicación/suscripción para sistemas distribuidos [8]. Este estándar ha sido creado por la necesidad industrial de estandarizar la computación centrada de datos (data-centric systems). Las aplicaciones distribuidas que usan DDS, al usar este tipo de comunicación que hemos comentado, pueden publicar o suscribirse a diferentes temas de información y configurar distintos parámetros de calidad de servicio para modificar diferentes aspectos de la comunicación como pueden ser la fiabilidad, persistencia, redundancia, vida útil, recursos, etc [9]. Ha conseguido ser utilizado en cualquier tipo de situación distribuida, tanto en interproceso como en remoto.

Desde su origen ha sufrido diferentes modificaciones haciendo que disponga de diferentes versiones. La primera fué lanzada en su creación el año 2003 (DDS 1.0) mientras que la más reciente la ubicamos en el año 2015 (DDS 1.4).

El uso de DDS tiene un cierto conjunto de ventajas, entre estas encontramos:

- Modelo publicación/suscripción: disminuye el acoplamiento entre entidades permitiendo un diseño más limpio de su sistema descentralizado.
- Protocolo de interoperabilidad: este protocolo nos ofrece una arquitectura flexible y adaptable gracias a la funcionalidad de descubrimiento automático.
- Eficiencia: provocada por el tipo de comunicación directa entre el publicador y el suscriptor.
- Escalabilidad: debido a la disminución de acoplamiento entre entidades provocado por el modelo publicación/suscriptor.
- Rendimiento: en el uso de la comunicación publicador/suscriptor, la latencia es más baja y el rendimiento es mayor que en otros tipos de comunicaciones.
- Parametrización de la QoS: disposición de gran abanico de parámetros que permiten ajustar diferentes aspectos de la comunicación.

3.2. Modelo publicador/suscriptor

El modelo usado por DDS, como hemos ido viendo durante el proyecto, es el modelo conocido de comunicación publicador/suscriptor. Este tipo de comunicación proporciona una comunicación asíncrona entre los participantes. Este modelo es de comprensión muy sencilla. Disponemos de distintos tipos de participantes que se pueden encargar de publicar (editar/añadir) y/o suscribirse (leer) a "temas". Estos "temas" se almacenan en un espacio llamado "Global Data Space".

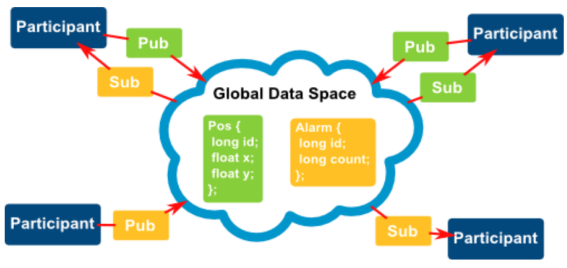


Fig 2. Esquema modelo publicador/suscriptor [10]

Como hemos visto al inicio del apartado, este tipo de modelo es asíncrono. En consecuencia, los publicadores podrán añadir los "temas" cuando ellos quieran y, los suscriptores podrán leerlo sin necesidad de estar conectados en ese mismo momento. Podrán hacerlo cuando ellos mismos deseen.

Hay una gran importancia sobre cómo se envían los "temas" a los participantes que están suscritos a éste, es decir, cómo podemos saber cuál es el destinatario. El descubrimiento de estos participantes remotos se hace de forma automática. [9]

Las características que destacan en este tipo de modelo se basan en la mayor escalabilidad que ofrece, el aseguramiento de la llegada del mensaje, la facilidad de flujos de trabajo asíncronos, una fácil integración de sistemas, un procesamiento de datos diferido, etc. [11]

3.3. Arquitectura

Según la arquitectura, DDS nos describe dos niveles de interfaces. Una primera, de nivel inferior, llamada DCPS y una segunda, de nivel superior, llamada DLRL.

- DLRL (Data Local Reconstruction Layer): esta capa se encarga de ofrecer una interfaz para las funcionalidades de la capa inferior (DCPS). Es una API estándar que permite crear vistas de objetos a partir de los "temas" y una integración simple de DDS en la capa de aplicaciones.
- DCPS (Data-Centric Publish-Subscribe): es una API estándar para la capa de publicación/suscripción en tiempo real. Se encarga de repartir de la forma más eficiente posible la información a los receptores apropiados. [8] [12]

En DDS, el mecanismo principal para que diferentes redes lógicas compartan una red física se conoce como ID de dominio. Los nodos ROS 2 en el mismo dominio pueden descubrir y enviarse mensajes libremente, mientras que los nodos ROS 2 en diferentes dominios no pueden. Todos los nodos ROS 2 utilizan el ID de dominio 0 de forma predeterminada. Para evitar interferencias entre diferentes grupos de computadoras que ejecutan ROS 2 en la misma red, se debe configurar una ID de dominio diferente para cada grupo. [13]

3.4. Seguridad

El objetivo principal de ROS fué permitirnos una mayor flexibilidad (permitiendo que los componentes de nuestros sistemas se puedan modificar con facilidad). Al inicio de ROS, al buscar el objetivo de la flexibilidad se dejó de lado otros campos de vital importancia, como son la seguridad. Por esto, durante el diseño de ROS2 se tuvo como objetivo intentar mejorar dicho aspecto. Y así, gracias a usar DDS, se consiguió mantener esta flexibilidad tan deseada mientras se logró igualmente la capacidad de tener mayor seguridad utilizando la especificación DDS-Security.

La especificación de DDS-Security nos ofrece muchas mejoras de seguridad mediante la definición de una arquitectura de interfaz de complemento de servicio (SPI), varias implementaciones integradas de estas SPI y el modelo

de seguridad impuesto por las propias SPI. Concretamente, hay cinco SPI definidas:

- Autenticación: proporciona medios para poder verificar la identidad de la aplicación y usuario que invoca operaciones en DDS. También nos facilita la autenticación mutua entre participantes y permite establecer un secreto compartido.
- Control de acceso: determina que operaciones relacionadas con DDS puede realizar un usuario autenticado.
- Criptografía: implementa distintas operaciones criptográficas como por ejemplo cifrado, descifrado, hash, firmas digitales, etc.
- Logging: proporciona la capacidad de auditar eventos relacionados con DDS-Security.
- Etiquetado de datos: proporciona la capacidad de añadir etiquetas a las muestras de datos.

Actualmente, ROS2 solo utiliza las tres primeras funciones de seguridad (Autenticación, Control de Acceso y Criptografía). El motivo principal es que las otras dos funciones indicadas no son requeridas para cumplir con las especificaciones de seguridad de DDS. Igualmente, como ya sabemos hay diferentes implementaciones DDS, por lo que hay distintas implementaciones que sí nos ofrecen estas funciones de seguridad. [14]

3.5. Alternativas

Alternativas Existen diferentes formas de comunicación entre robots similares a DDS. Como ya sabemos, DDS es un software conocido como middleware que sirve como capa de abstracción para los diferentes métodos de comunicación entre procesos del sistema operativo integrado. En términos de middlewares competidores podemos encontrar algunos como lo son LCM y ZeroMQ. Recientemente ha aparecido una alternativa a DDS, Zenoh, creado por especialistas en DDS con el objetivo de mejorar algunos aspectos de DDS problemáticos. A continuación, se examinan brevemente:

LCM:
Es un conjunto de librerías y herramientas que nos sirven para el paso de mensajes y la gestión de datos. Están dirigidas a sistemas en tiempo real donde el alto ancho de banda y la baja latencia son la prioridad máxima. Al igual que DDS también dispone de un modelo de paso de mensajes de publicación/suscripción. También dispone de variedad de lenguajes de programación (C, C + +, Lua, MATLAB, Python, Java, etc). [15]

Sus características principales son: comunicación de baja latencia, alto ancho de banda, UDP Multicast, comunicación directa (no está centralizado), sin demons y pocas dependencias.

ZeroMQ:
Es una librería, gratuita y de código abierto, de comunicaciones de alto rendimiento orientada al paso de mensajes, destinada principalmente a aplicaciones distribuidas. Está diseñado de tal forma que busca ser muy parecido al típico socket. Al igual que los demás frameworks vistos, también tiene una gran variedad de lenguajes de programación (C, Java, Erlang, Go, JavaScript, etc). [16]

Sus características principales son: es universal (conecta en cualquier idioma y plataforma), usa patrones inteligentes como pub-sub, push-pull y cliente-servidor, alta velocidad, múltiple transporte (UDP, TCP, TIPC, IPC, etc) y respaldado por una gran comunidad activa.

Zenoh:
Es un protocolo publicador/suscriptor/query. Esto significa que aparte de disponer de las funcionalidades del típico publicador/suscriptor, añade la funcionalidad de almacenar los datos enviados y poder hacer consultas a estos. Haciendo que disponga adicionalmente de la funcionalidad de disponer de una especie de historial de datos enviados.

Dicho protocolo alternativo a DDS tiene un conjunto de características que lo hacen un rival muy competidor:

- Elocuencia: su abstracción para la publicación/suscripción, el almacenamiento geodistribuido y las consultas simplifican fuertemente el desarrollo de aplicaciones distribuidas a cualquier escala.
- Escalabilidad: tanto el propio protocolo como sus implementaciones están distribuidas y, por lo tanto, dando una muy buena escalabilidad.
- Rapideza: muy fácil de aprender y extremadamente eficiente. Ofreciendo un alto rendimiento y una baja latencia.
- Transparencia: Zenoh aporta el gran beneficio que nos ofrece la transparencia de la ubicación tanto de los datos en movimiento como de los datos que se encuentran estáticos.
- Eficiencia energética: actualmente no somos conscientes de que la comunicación es increíblemente costosa en cuanto a energía. Mucha de esta energía la estamos gastando sin darnos cuenta al conectarnos con centros de datos. Zenoh fue diseñado con el objetivo de destacar en cuanto a la eficiencia energética. [17]

Para ver una cierta comparativa con DDS, Zenoh resuelve algunos de los problemas que se encuentran comúnmente en este protocolo, como la complejidad de la configuración, falta de eficiencia en entornos distribuidos y la falta de interoperabilidad.

Para configuraciones complejas, Zenoh proporciona una forma simplificada de administrar los datos distribuidos, lo que reduce la necesidad de una configuración manual y simplifica la administración de la red.

Cuando se trata de interoperabilidad, Zenoh adopta un enfoque agnóstico de los lenguajes de programación, los sistemas operativos y los protocolos de red, lo que permite que los dispositivos y los sistemas de información se comuniquen entre sí de manera más eficiente.

Finalmente, en términos de eficiencia en entornos distribuidos, Zenoh proporciona una arquitectura de red más escalable y eficiente que DDS, que puede manejar mejor grandes cantidades de datos y sistemas con recursos limitados.

En resumen, Zenoh nos facilita una alternativa más simple, escalable e interoperable a DDS que puede ayudar a

resolver algunos de los problemas comunes de la administración de datos distribuidos en entornos complejos y heterogéneos. Esto no significa que Zenoh sea mejor que DDS, simplemente que nos ofrece unas ventajas que el otro protocolo no. Por otro lado, DDS también tiene ciertos puntos en los que se encuentra por encima de Zenoh así que la elección de un protocolo u otro dependerá de las características del proyecto. [18]

4 RTPS

Como ya podemos saber, existen el conjunto de protocolos de transporte TCP/UDP/IP. Pero este tipo de protocolos son de muy bajo nivel para ser utilizados directamente por una aplicación que no sea muy simple, por lo tanto, han ido surgiendo distintos protocolos que han ido satisfaciendo las necesidades de las aplicaciones. Entre estos protocolos podemos ver por ejemplo HTTP, DHCP, FTP, RTPS, entre otros. Cada uno de estos protocolos está adaptado a unas ciertas funcionalidades que cumplen los requerimientos de las aplicaciones que los utilizan. [19]

RTPS o también conocido como protocolo de suscripción/publicación en tiempo real es uno de estos protocolos, que se encuentra por encima de los de transporte, para comunicaciones de participantes publicadores y suscriptores confiables. Este tipo de comunicación funciona mejor sobre transportes no confiables como UDP. Puede ser usado como protocolo de interoperabilidad para las implementaciones DDS, la cual será una capa intermedia de la capa DDS y la capa de transporte.

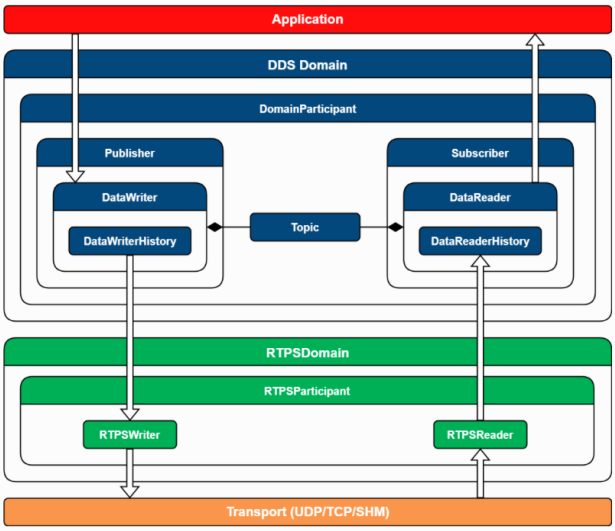


Fig 3. Arquitectura comunicación DDS-RTPS [20]

5 IMPLEMENTACIONES

5.1. Fast DDS

5.1.1. Introducción

Fast DDS es un marco de suscripción/publicación de alto rendimiento, desarrollado por la empresa eProsima, con el objetivo de conseguir la compartición de datos en sistemas distribuidos mediante un modelo desacoplado basado en editores, suscriptores y temas de datos. Esta implementación middleware implementa muchas especificaciones DDS [21]. Fast DDS proporciona los estándares de protocolo de cable interoperables OMG DDS 1.4 y OMG RTPS 2.2 consiguiendo una implementación sorprendentemente rápida.

Dicha implementación tiene una serie de ventajas muy importantes que deberemos de tener en cuenta en el momento de seleccionar una implementación DDS. Entre estas encontramos un alto rendimiento, baja latencia, fácil integración multiplataforma, cumplimiento de estándares DDS 1.4 y RTPS 2.2, servicios personalizados, etc.

5.1.2. Arquitectura

La arquitectura de Fast DDS está formada de un seguido de capas que se encuentran en los entornos que hemos ido viendo a lo largo del proyecto: capa de aplicación, capa Fast DDS, capa RTPS y capa de transporte.

Capa DDS:
Hay distintos elementos clave de comunicación definidos en la capa DDS en la implementación Fast DDS. En dicha implementación, el usuario creará estos elementos en la capa de aplicación dando la responsabilidad a Fast DDS de definirlos como entidades. Una de estas entidades DDS es cualquier objeto que admite una configuración QoS y que dispone de un oyente. Las entidades DDS existentes son:

- Domain: valor entero que se encarga de identificar el dominio DDS.
- Domain Participant: entidad que engloba distintos tipos de entidades (editores, suscriptores, temas y multitemas).
- Publisher: publica los datos en un tema mediante un Data Writer. Se encarga de crear y configurar las entidades Data Writer que contiene. También tiene la responsabilidad de administrar el almacenamiento en búfer de los mensajes salientes.
- Data Writer: encargado de publicar los mensajes. El usuario proporcionará un tema al crear esta entidad. Los datos publicados se harán bajo este tema. La publicación será posible gracias a la entidad Data Writer History.
- Data Writer History: lista de cambios en los objetos de datos. Cuando el DataWriter procede a publicar datos bajo un tema específico, en realidad crea un cambio en estos datos. Es este cambio el que se registra en el "History".
- Subscriber: se suscribe a un tema utilizando la entidad Data Reader. Esta entidad se encarga de crear y configurar sus entidades Data Readers. También tienen la responsabilidad de administrar el almacenamiento en búfer de los mensajes entrantes.
- Data Reader: se encarga de suscribir los temas para la recepción de publicaciones. El usuario debe proporcionar un tema de suscripción al crear la entidad. Un Data Reader recibe mensajes como cambios en su "Data Reader History"
- Data Reader History: contiene los cambios en los objetos de datos que recibe su Data Reader.

- Topic: vincula los Data Writers de los publicadores con los Data Readers de los suscriptores.

Capa RTPS:
En el protocolo RTPS, Fast DDS permite la abstracción de entidades de aplicación DDS de la capa de transporte. En esta capa encontramos cuatro entidades principales:

- Domain RTPS: extensión del dominio DDS al protocolo RTPS.
- Participant RPTS: al igual que en la capa superior, contiene otras entidades RTPS también se encarga de su configuración y creación de estas.
- Writer RTPS: lee los cambios escritos en el Data Writer History y los transmite a todos los RTPS Readers con los que está emparejado.
- Writer RTPS: lee los cambios escritos en el Data Writer History y los transmite a todos los RTPS Readers con los que está emparejado. [22]

5.2. Cyclone DDS

5.2.1. Introducción

Cyclone DDS es una tecnología, desenvolupada por la empresa ZettaScale, de intercambio de datos basada en el estándar OMG-DDS de alto rendimiento que permite a los diseñadores de sistemas crear gemelos digitales de las entidades de sus sistemas para compartir sus estados, eventos, flujos de datos y mensajes en la red de manera en tiempo real y tolerante a fallos. El estándar OMG DDS es reconocido como un estándar altamente aplicable para el intercambio de datos confiable y robusto en entornos empresariales y de misión crítica. Por este motivo, hoy en día, Cyclone DDS está ganando impulso en los mercados de robótica, vehículos autónomos y automoción, así como en otros sistemas de IoT. En cuanto a lenguaje tiene soporte a C, C + + y Python. [23]

Las características de esta implementación se basan en su rapidez y fiabilidad, su alta consistencia y escalabilidad, una buena seguridad y la capacidad de interactuar con otras implementaciones. [24]

5.2.2. Arquitectura

En Cyclone DDS los componentes son independientes, se implementan como microservicios que se comunican entre sí a partir de una api. Al igual que en Fast DDS, esta implementación dispone de unas entidades que en este caso tienen una serie de diferencias en cuanto a sus características. Estas diferencias en las entidades son:

- Domain Participant: la diferencia que hay entre esta entidad y la de la implementación de Fast DDS se basa en que en Cyclone aparte de ser responsable de crear las entidades, también se encarga de enrutar mensajes para la comunicación entre nodos.
- Publisher: publica los datos en un tema mediante un Data Writer. En este caso no tiene la responsabilidad de administrar el almacenamiento en búfer como en Fast DDS.
- Subscriber: se suscribe a un tema utilizando la entidad Data Reader. En este caso no tiene la responsabilidad de administrar el almacenamiento en búfer como en Fast DDS.
- Data Writer: encargado exclusivamente de publicar los mensajes. En Fast DDS se puede dar el caso que también se encargue de administrar el almacenamiento en búfer.
- Data Reader: encargado exclusivamente de publicar los mensajes. En Fast DDS se puede dar el caso que también se encargue de administrar el almacenamiento en búfer. [23]

6 Caso práctico

Como hemos avanzado al inicio del proyecto, el objetivo que tenemos es descubrir si hay algún benefició en cuanto a rendimiento al usar una de las implementaciones DDS vistas. Para ello, se realizarán unos casos prácticos donde a partir de unas métricas compararemos las implementaciones en las distintas pruebas que haremos. Cabe comentar que durante la realización de dicho caso práctico hemos encontrado distintos problemas con DDS. Estos se basan en que este protocolo no es eficiente cuando el mensaje que se publica es de un volumen elevado de datos. Para solucionar esto hemos planteado añadir al estudio una de las alternativas de DDS, Zenoh. Por lo que hemos visto brevemente, este protocolo es capaz de manejar una gran cantidad de datos, por lo que veremos si es capaz de solucionar este problema de DDS.

6.1. Métricas de estudio

Uno de los apartados más importantes del estudio es determinar cómo compararemos las implementaciones, para ello, hemos definido el uso de unas métricas que nos permitirán hacer ampliamente la comparación.

Dichas métricas son las siguiente:

- Latencia: se refiere a la cantidad total de tiempo que lleva enviar un mensaje completo del nodo origen hasta que llega al nodo destinatario (sin contar en el tiempo de procesado del destinatario).
- Número de paquetes enviados: veremos la cantidad de tráfico en la red que hay entre el nodo publicador y suscriptor.
- Número de bytes enviados: la cantidad de bytes que se envían en la conversación de los nodos publicador y suscriptor.

6.2. Caso de uso

Para comenzar realizaremos el caso más básico posible. Este consiste en la publicación de un mensaje a partir de un nodo y la suscripción a este mismo por parte de otro nodo. Para la realización de este caso se tendrán en cuenta los siguientes escenarios: LAN, Wifi (802.11ac) y VPN (zerotier) usando las implementaciones de DDS Cyclone DDS y Fast DDS, y para tener datos sobre el otro protocolo, también lo haremos con Zenoh (comentado en alternativas de DDS).

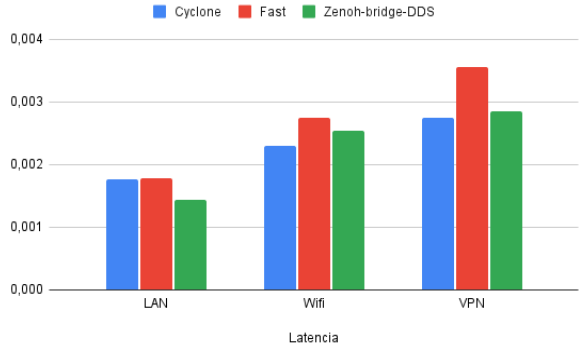
Para la implementación de este protocolo alternativo crearemos una especie de puente entre los nodos ROS2/DDS con Zenoh (explicado en el apéndice). Cabe comentar que para determinar la latencia haremos la media de cien mensajes mientras que para determinar el tráfico (número de paquetes y cantidad de bytes) lo haremos usando la herramienta de Wireshark, analizando durante 5 minutos cada conversación entre el nodo publicador y suscriptor.

Para el siguiente caso de uso intentaremos ver si Zenoh es una posible mejora en entornos donde el volumen de datos enviados es elevado. Sabemos desde un inicio del estudio que DDS puede dar problemas en estos casos. Para ello hemos introducido anteriormente la alternativa Zenoh. En teoría, dicha alternativa tiene que ser capaz de enviar esta cantidad de datos sin ninguna dificultad. Para hacer este estudio utilizaremos una cámara donde el publicador irá enviando imágenes y el nodo suscriptor irá recibiéndolas.

6.3. Comparación de resultados

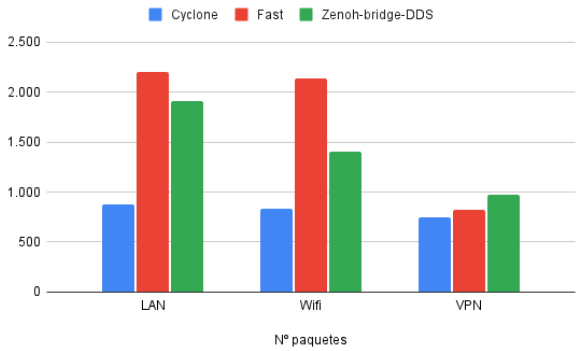
6.3.1. Pub/Sub simple

En el primer estudio hemos obtenido los siguientes resultados. Como podemos observar en los diferentes gráficos vemos que en cuanto a latencia, la diferencia que hay es mínima. Habiendo una diferencia tan baja no se puede decir que haya alguna implementación que destaque por encima de las demás. Lo que si podemos comentar es el caso de Zenoh. Todo y ser una tecnología que aún está en desarrollo consigue competir quedando en muy buen lugar con las implementaciones DDS. Los resultados obtenidos en cuanto a la latencia son los siguientes:



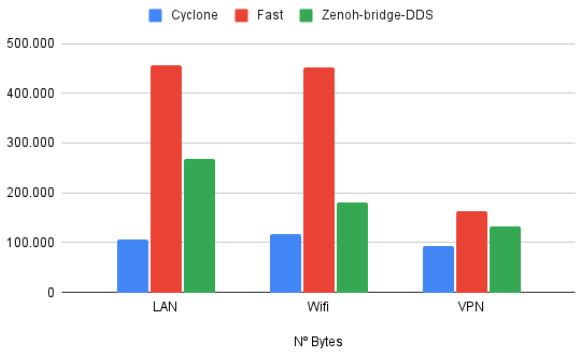
En los siguientes gráficos podemos ver que hay una fuerte diferencia en cuanto a las implementaciones DDS, podemos ver que Cyclone queda bastante por encima respecto a Fast DDS. Esto no significa que dicha implementación sea mejor, pero que en un ejemplo tan sencillo como el que hemos propuesto haya una diferencia tan grande ya nos dice mucho. Con este estudio podemos llegar a interpretar que Fast puede llegar a tener problemas en ejemplos más complejos donde haya mayor tráfico de red, como por ejemplo, en un entorno donde se envíen más datos con un mayor número de nodos que publican y se suscriben. En cuanto a Zenoh queremos destacar que no se queda para nada atrás de las implementaciones DDS, consiguiendo mejores resultados que Fast y compitiendo en algunos de los escenarios contra Cyclone.

Para la comparación de la cantidad de paquetes que se han enviado durante el estudio por cada escenario podemos ver lo siguiente:



Como hemos comentado, vemos que Cyclone se mantiene siempre en un muy buen punto, mientras que Fast se queda bastante por detrás. Cabe destacar la constancia de Cyclone DDS, donde en los distintos estudios que hemos hecho, en todos los escenarios ha conseguido siempre unos resultados muy similares. Para acabar con este caso de uso veremos el número de bytes enviados. Como vemos tiene cierto sentido con el número de paquetes que se han enviado. En este caso se puede ver aún más la diferencia entre Cyclone y Fast, dejando a la primera comentada por encima en todo el estudio hecho. Una posible explicación para la diferencia en la cantidad de paquetes y bytes enviados entre las implementaciones DDS podría estar relacionada con la forma en que cada una está configurada o implementada. Específicamente, creemos que Fast podría experimentar una mayor pérdida de paquetes, lo que resultaría en la necesidad de reenviarlos. Vemos también que Zenoh sigue quedándose en buen lugar respecto a Cyclone llegando en algunos casos a tener resultados muy parecidos.

A continuación, podremos ver el número de bytes enviados durante la conversación entre el nodo publicador y suscriptor:



Como conclusión de este estudio cabe destacar los buenos resultados de Cyclone, tanto en rendimiento como en constancia. Lo que nos sorprende es el gran número de paquetes/bytes que se envían por parte de la implementación Fast DDS, que siendo un ejemplo tan simple destaca la gran diferencia que hay. Y para finalizar, el buen rendimiento de Zenoh teniendo buenos resultados comparándolo con las implementaciones DDS.

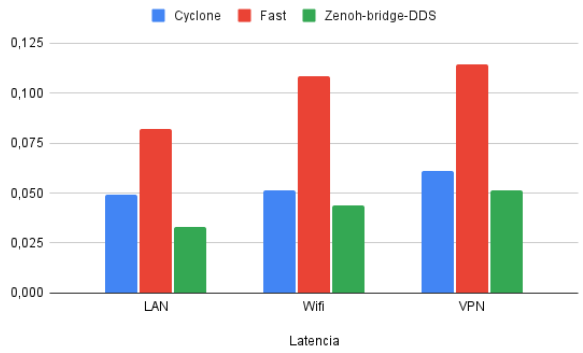
6.3.2. Pub/Sub cámara

En este pequeño estudio, a diferencia del anterior, se quiere ver como se comportan las implementaciones cuan-

do hay un fuerte aumento del tráfico de red, es decir, intentaremos llevar más al límite todas las implementaciones para ver como se comportan. Queremos destacar que en las implementaciones DDS hemos tenido algún problema (sobretodo con Fast DDS), mientras que usando Zenoh no hemos tenido ningún inconveniente.

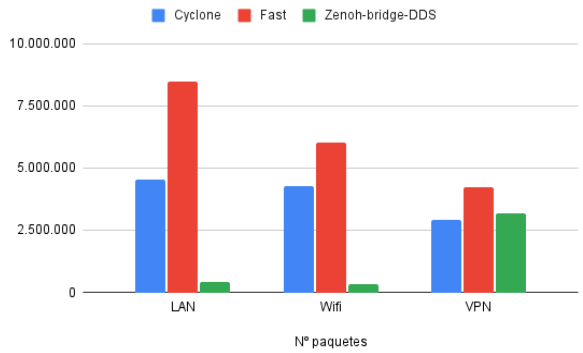
Como vemos con la latencia, resaltamos los malos resultados de Fast DDS, quedándose muy por detrás de Cyclone DDS. En cuanto a esta, resaltamos la constancia que sigue teniendo en todos los entornos. Como ganador, bastante parejo con Cyclone esta Zenoh, confirmándonos la teoría que teníamos sobre este protocolo.

En cuanto a la latencia hemos obtenido los siguientes resultados:



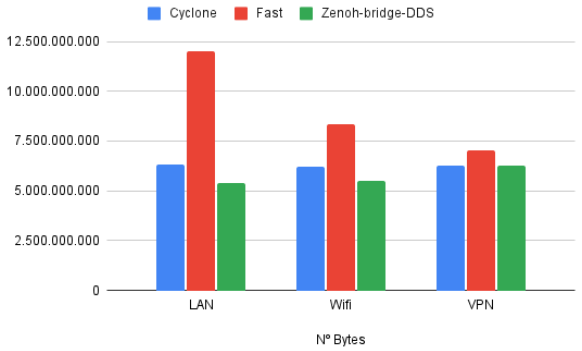
En cuanto al número de paquetes destacaremos tres puntos que podemos ver. Empezaremos comentando lo que podíamos intuir del estudio anterior, Fast DDS tiene problemas en cuanto al número de paquetes enviados. Vemos que la diferencia que hay es bastante grande, pudiendo ser uno de los motivos por los cuales hemos tenido problemas en la realización del estudio para esta implementación. En cuanto a Cyclone DDS, comentamos también lo que hemos ido viendo durante todos los casos de uso, su buena constancia. Por parte de Zenoh podemos ver que destaca fuertemente el número de paquetes en los entornos LAN y Wifi. Durante el experimento, hemos podido ver que este protocolo enviaba paquetes de un tamaño muy superior a las implementaciones DDS, haciendo que el número de paquetes disminuya considerablemente. En cuanto a la VPN, vemos que esto no es así, vemos que los resultados son muy parecidos para todas las implementaciones. La respuesta que damos a esto es que la propia VPN tiene una serie de restricciones, como por ejemplo el tamaño del paquete, esto puede impedir que Zenoh envié estos paquetes tan grandes, haciendo que se iguale con las otras implementaciones.

El número de paquetes enviados son:



Para acabar, la cantidad de bytes enviados, vemos que tienen sentido con lo que hemos ido comentando. Como Zenoh, en los entornos LAN y Wifi, envía paquetes mucho más grandes, enviando muchos menos consigue enviar los mismo bytes que por ejemplo Cyclone DDS. En los otros casos destacamos lo mismo que hemos ido viendo, los resultados de Fast DDS siendo inferiores que las otras implementaciones.

Finalmente, los bytes enviados durante la conversación son:



Como conclusión del caso de uso, podemos decir que el protocolo Zenoh es una muy buena alternativa a DDS, dándonos diferentes opciones que como hemos visto nos puede dar incluso mejores resultados. En cuanto a las implementaciones DDS, hemos podido confirmar la constancia de Cyclone DDS, nos ha sorprendido que haya conseguido en algunos casos acercarse mucho a los resultados de Zenoh. Y para finalizar, comentar los malos resultados de Fast DDS, dando en casi todo el caso de uso resultados muy inferiores a sus competidores.

7 Conclusión

Como conclusión final del trabajo podemos ver los resultados obtenidos. En estos nos choca un poco con que para la versión de ROS2 que hemos utilizado (Humble), se tiene definido de serie el uso de Fast DDS. Por lo que hemos podido ver por parte de la comunidad tiene cierto sentido. El motivo es que según esta, Fast DDS es superior a Cyclone DDS una vez se haya configurado correctamente. Aquí es donde podemos ver el problema con Fast DDS. Por lo que hemos visto y por lo que también dice la propia comunidad, la configuración de Fast DDS es bastante compleja. Sin esta configuración el rendimiento de esta implementación disminuye considerablemente. Por este motivo en muchos proyectos se usa Cyclone DDS, ya que por lo visto destaca su facilidad, donde haciendo una configuración muy sencilla hemos obtenido muy buenos resultados, siendo muy constante en todos los entornos, por lo que podemos decir que tiene muy buen rendimiento.

Por otro lado, queremos destacar la alternativa Zenoh, que siendo un protocolo muy poco usado debido a su corta 'edad' hemos visto que es una muy buena opción para el intercambio de datos. Vemos que al igual que Cyclone DDS tiene una configuración muy sencilla y nos da unos resultados muy estables en todo momento. Pero a diferencia de la implementación DDS, esta nos da mejores resultados en entornos donde el tráfico de paquetes es muy superior.

En este punto, nos surge la pregunta sobre cuál opción

podría ser la mejor. Podemos considerar utilizar Cyclone DDS, una implementación DDS sencilla que se sabe que tiene un buen rendimiento. Otra posibilidad sería intentar configurar Fast DDS, que aunque sea más complejo según la comunidad, una vez configurada adecuadamente ofrece un rendimiento superior a Cyclone. Y finalmente, también podríamos explorar la nueva alternativa Zenoh, que parece ser muy fácil de configurar y ha demostrado brindar excelentes resultados en todos los casos que hemos estudiado.

REFERENCIAS

[1] R. Suárez, J. Rosell, M. Vinagre, F. Cortes, A. Ansuategui, I. Maurtua, D. Martin, A. Guash, J. Azpiazu, D. Serrano, N. García. Robot Operating System (ROS). https://www.aer-automation.com/wpcontent/uploads/2022/03/ROS_articuloAER.pdf (accedido el 26 de febrero de 2023)

[2] A. Ademovic. An Introduction to Robot OS Toptal. Toptal Engineering Blog. https://www.toptal.com/robotics/introduction-to-robot-operating-system (accedido el 26 de febrero de 2023).

[3] es/ROS/Introduccion - ROS Wiki. Documentation - ROS Wiki. http://wiki.ros.org/es/ROS/Introduccion (accedido el 28 de febrero de 2023).

[4] DDS as a middleware for ROS2. Design. https://design.ros2.org/articles/ros_on_dds.html (accedido 2 de marzo de 2023)

[5] RMW: ROS Middleware Abstraction Interface. https://docs.ros2.org/foxy/api/rmw/index.html (accedido el 2 de marzo de 2023).

[6] ROS 2 middleware interface". Design. https://design.ros2.org/articles/ros_middleware_interface.html (accedido el 5 de marzo de 2023).

[7] ROS2 QoS - Deadline, Liveliness and Lifespan. https://design.ros2.org/articles/qos_deadline_liveliness_lifespan.html (accedido el 5 de marzo de 2023).

[8] Data Distribution Service - Wikipedia, la enciclopedia libre. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Data_Distribution_Service#cite_note-1 (accedido el 6 de marzo de 2023).

[9] DDS middleware. eProsima: RTPS/DDS Experts. https://www.eprosima.com/index.php/resources-all/whitepapers/dds (accedido el 6 de junio de 2023).

[10] DDS middleware. eProsima: RTPS/DDS Experts. https://www.eprosima.com/images/diagrams/GlobalDataSpace.png (accedido el 6 de marzo de 2023).

[11] Patron de publicador y suscriptor - Azure Architecture Center. Microsoft Learn: Build skills that open doors in your career. https://learn.microsoft.com/es-es/azure/architecture/patterns/publisher-subscriber (accedido el 7 de marzo de 2023).

[12] DDS Protocol Architecture basics. DDS Protocol in IoT. RF Wireless World. https://www.rfwireless-world.com/Terminology/DDS-protocol-architecture.html (accedido el 7 de marzo de 2023).

[13] About-Domain-ID — ROS 2 Documentation: Humble documentation. ROS Documentation. https://docs.ros.org/en/humble/Concepts/About-Domain-ID.html (accedido el 9 de marzo de 2023).

[14] ROS 2 DDS-Security integration. Design. https://design.ros2.org/articles/ros2_dds_security.html (accedido el 9 de marzo de 2023).

[15] LCM Project. LCM: Lightweight Communications and Marshalling. Github. https://lcm-proj.github.io/ (accedido el 10 de marzo de 2023).

[16] ZeroMQ, "The Intelligent Transport Layer for IoT". https://zeromq.org/ (accedido el 10 de marzo de 2023).

[17] Zenoh - The Zero Overhead, Pub/Sub, Store, Query, and Compute Protocol. https://zenoh.io/ (accedido el 10 de marzo de 2023).

[18] Discourse Zenoh Performance for ROS2. Discourse ROS ORG. https://discourse.ros.org/t/zenoh-performance/30494/4. (accedido el 12 de marzo de 2023)

[19] RTPS. Wiki. https://wiki.wireshark.org/Protocols/rtps (accedido el 12 de marzo de 2023).

[20] DDS and RPTS architecture eschema. Implementation Fast DDS 2.11.0 documentation. https://fast-dds.docs.eprosima.com/en/latest/_images/library_overview.svg (accedido el 19 de marzo de 2023).

[21] Fast RTPS is now Fast DDS! eProsima: RTPS/DDS Experts. https://eprosima.com/index.php/company-all/news/146-fast-rtps-is-now-fast-dds (accedido el 19 de marzo de 2023).

[22] The most complete open source DDS middleware — Fast DDS. eProsima: RTPS/DDS. https://eprosima.com/index.php/products-all/eprosima-fast-dds (accedido el 19 de marzo de 2023).

[23] Cyclone DDS, OMG DDS. ADLINK. ADLINK Technology. https://www.adlinktech.com/en/CycloneDDS (accedido el 27 de marzo de 2023).

[24] Eclipse Cyclone DDS. Eclipse Cyclone DDS - Home. https://cyclonedds.io/ (accedido el 27 de marzo de 2023).

[25] Driving Autoware with Zenoh - Autoware. Autoware. https://autoware.org/driving-autoware-with-zenoh/ (accedido el 28 de mayo de 2023).

APÉNDICE

A.1. Hardware y Software

A continuación, presentaremos una lista del hardware y software que hemos utilizado en este proyecto. Cabe destacar que los recursos utilizados durante el estudio han sido dados por la empresa Movvo. Los componentes utilizados que han contribuido al desarrollo y éxito de este proyecto son por parte del hardware:

- Intel(R) Core(TM) i5-8259U CPU @ 2.30GHz (with SSE4.2)
- Cámara de seguimiento Intel® RealSense™ T265

Mientras que por parte del software hemos utilizado:

- Linux 5.19.0-41-generic"
- ROS2 Humble

Cabe mencionar que en esta lista solo se ha incluido los componentes que han tenido un impacto directo con los resultados obtenidos, es decir, no hemos mencionado otras herramientas tecnológicas que no han desempeñado un papel crucial en la implementación y logros alcanzados.

A.2. Instalación ROS2/DDS

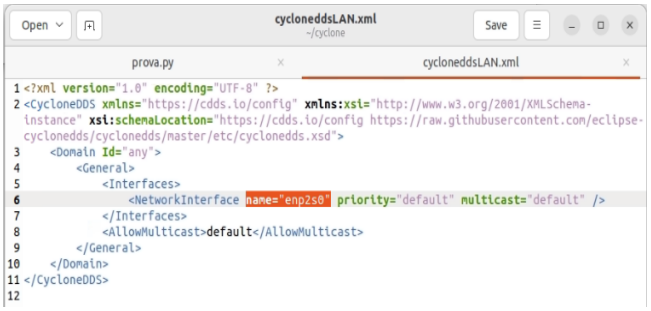
Para poder realizar los casos de uso, partimos de la instalación de ROS2 Humble. Este software lo podemos instalar directamente de la documentación de ROS Humble ejecutando el comando de la instalación de dicho paquete. Esta versión viene con la implementación Fast DDS, por lo que para realizar todos los estudios hemos tenido que realizar la instalación de Cyclone DDS. Para hacerlo solamente tendremos que ejecutar el comando de instalación del paquete correspondiente a Cyclone DDS.

Es importante comentar, que para poder alternar de implementación de DDS, deberemos modificar una variable 'RMW_IMPLEMENTATION' en el archivo .bashrc. Poniendo como valor 'rmw_fastrtps_cpp' para activar la implementación de Fast DDS y 'rmw_cyclonedds_cpp' para Cyclone DDS. En este mismo archivo, también deberemos seleccionar el dominio de nuestros nodos. Obviamente, para que se puedan comunicar entre sí pondremos el mismo valor en ambos nodos.

Una vez toda la instalación de ROS2 y DDS este lista deberemos crear los nodos. Para ello hemos realizado la implementación de un nodo publicador en un ordenador y un nodo suscriptor en el otro. Para la programación de estos hemos usado Python debido a la sencillez del lenguaje.

A.3. Configuración Cyclone DDS

Para que la implementación Cyclone DDS funcione correctamente es necesario un archivo de configuración xml, con el cual se deberá definir unos ciertos campos. Uno de los que nos será más útil es el que nos determinará que tipo de conexión utilizaremos, es decir, si será LAN, Wifi o VPN. Esto nos será necesario para poder realizar el estudio en todos los ámbitos comentados.



Configuración Cyclone DDS

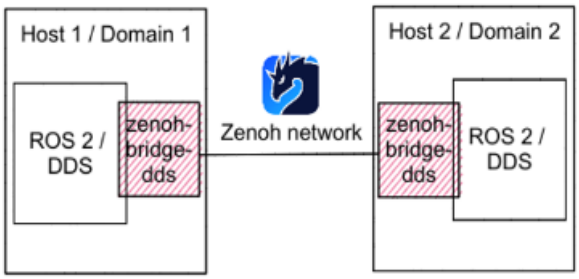
Como podemos ver, hemos definido la interfaz por la cual se comunicaran nuestros nodos. Por lo que ambos nodos deberán tener la misma configuración para que se realize la comunicación correctamente. También comentar que en este archivo podemos añadir una gran cantidad de variables que modificaran la configuración de Cyclone DDS al gusto de cada usuario.

En el caso de Fast DDS no será necesario esta configuración que hemos hecho debido a que se elige automáticamente la interfaz de conexión más eficiente. Aún así, es dicho por la comunidad que dicha configuración es necesaria para poder optimizar mucho esta implementación. En contra hay que decir que esta configuración es mucho más compleja de realizar que la de Cyclone DDS.

A.4. Instalación Zenoh-bridge-dds

Antes de comentar la instalación de Zenoh explicaremos brevemente en que consiste esta implementación. Consiste en mantener los nodos ROS2/DDS pero creando una especie de puente que se encarga de convertir el middleware DDS en Zenoh.

Para tener una idea de zenoh-bridge-dds más clara podemos ver el siguiente esquema:



Esquema modelo zenoh-bridge-dds [25]

En cuanto a la instalación de esta implementación hemos tenido que realizar la instalación del paquete 'zenoh-bridge-dds'. Una vez instalado en los dos nodos, hemos tenido que realizar la configuración que veremos en el siguiente apartado del apéndice.

A continuación hemos modificado el dominio de cada nodo para que el intercambio de información se haga usando solamente este puente y no mediante DDS. Para finalizar hemos ejecutado los puentes y los mismos nodos utilizados para las implementaciones DDS, obteniendo en el nodo suscriptor lo publicado por el nodo publicador a través de Zenoh.

A.5. Configuración Zenoh-bridge-dds

Para realizar la configuración de estos 'puentes' lo haremos a partir de modificar un archivo de configuración json5 que se nos descargará una vez instalemos el paquete de 'zenoh-bridge-dds'. En este caso, a diferencia de las implementaciones DDS, disponemos de un archivo que contiene todas las posibles variables que podemos modificar. En nuestro caso, como queremos tener una comparación justa entre las implementaciones, solamente hemos modificado la variable para poder alternar sobre nuestras interfaces de red:

```
listen: {
  endpoints: [
       "tcp/192.168.194.120:7447"      // VPN
       "tcp/172.21.2.217:7447"         // Wifi
       "tcp/192.168.0.1:7447"          // LAN
    // "<proto>/<address>"
  ],
},
```

```
connect: {
  endpoints: [
       "tcp/192.168.194.130:7447"      // VPN
       "tcp/172.21.2.213:7447"         // Wifi
       "tcp/192.168.0.2:7447"          // LAN
    // "<proto>/<address>"
  ],
},
```

Configuración zenoh-bridge-dds

Como podemos ver hemos modificado las variables 'listen' y 'connect'. La primera variable comentada nos permitirá poner que direcciones IP queremos que puedan escuchar, por lo que deberemos poner las direcciones de la misma máquina. En la segunda, deberemos poner las direcciones IP de donde nos queremos conectar, es decir, de la otra máquina. En nuestro proyecto, nos ha sido muy útil para poder realizar el estudio en las diferentes interfaces de red, permitiéndonos seleccionar directamente que interfaz de red queremos utilizar.

A.6. Instalación Zenoh

Otra forma para incluir Zenoh seria mediante un nodo Zenoh y un router Zenoh. Para ello deberemos hacer tanto la instalación del paquete Zenoh como la del router Zenoh. Una vez hecha, deberemos crear los nodos. En este caso, a diferencia de 'zenoh-bridge-dds', no podremos usar los mismos nodos ROS2/DDS. Una vez programados dichos nodos usando la librería Zenoh, deberemos realizar la configuración del router Zenoh. En este caso para el intercambio de datos, solo será necesario un router.

Esta implementación también ha sido llevada a cabo durante el estudio, pero no se ha reflejado en este debido a no usar ROS2. Igualmente, queremos destacar que los resultados de esta implementación Zenoh han sido sorprendentemente buenos, igualando a otras implementaciones como Cyclone DDS o 'zenoh-bridge-dds'.

A.7. Configuración Zenoh router

Para poder realizar una implementación a partir de nodos Zenoh será necesario un router Zenoh que se encargará de conectar estos nodos. Este solo será necesario que se instale en uno de los nodos.

De la misma manera que la otra implementación de Zenoh vista, también obtendremos un archivo de configuración json5 al descargar el paquete del router Zenoh. Esta configuración tiene una semejanza muy grande con la vista en la implementación 'zenoh-bridge-dds'. Por lo que de la misma, para poder realizar correctamente esta implementación hemos modificado los mismos campos.