EVALUACIÓN DE SISTEMAS DE DISTRIBUCIÓN PARA LA AUTOMATIZACIÓN DE APLICACIONES INDUSTRIALES INTELIGENTES

EVALUATION OF DISTRIBUTION SYSTEMS FOR THE AUTOMATION OF INTELLIGENT INDUSTRIAL APPLICATIONS

Trabajo de Fin de Máster
para acceder al

MÁSTER EN INGENIERÍA INFORMÁTICA

Autor: Mario Martín Pérez

Directora: Marta Elena Zorrilla Pantaleón

Julio - 2025

Agradecimientos

El presente Trabajo de Fin de Máster se ha llevado a cabo en el contexto del proyecto nacional **PRESECREL**[1], desarrollado por el grupo de investigación **ISTR** del **Departamento de Ingeniería Informática de la Universidad de Cantabria**.

Me gustaría expresar mi agradecimiento a dicho departamento por brindarme la oportunidad de participar en un entorno de investigación real y multidisciplinar, en el que seguir satisfaciendo mi curiosidad e interés por el entorno académico.

De forma especial, agradezco también a mi tutora, Marta Elena Zorrilla, su acompañamiento, dedicación y supervisión durante todo el desarrollo del presente trabajo, sin lo cual, hubiera sido mucho más difícil superar la curva de aprendizaje que implica un *middleware* como DDS.

[1] ISTR. Research activities. Disponible en: https://www.istr.unican.es/research.html

Resumen

La transición de la Industria 4.0 al IoT Continuum refleja un cambio fundamental en los modelos de procesamiento de datos, que pasan de estar centrados en la nube a distribuirse progresivamente hacia el *edge* y los dispositivos, con el objetivo de reducir la latencia, aumentar la resiliencia y habilitar capacidades de decisión en tiempo real a lo largo de toda la arquitectura. Esta continuidad operativa plantea importantes retos en términos de interoperabilidad, latencia y rendimiento.

En este contexto, tecnologías como DDS (Data Distribution Service) y Apache Kafka han demostrado ser eficaces, aunque en escenarios diferentes. DDS destaca por sus capacidades en entornos distribuidos con requisitos estrictos de comunicación en tiempo real, mientras que Kafka es ampliamente utilizado para procesar grandes volúmenes de datos en arquitecturas en la nube. Por lo tanto, la coexistencia de ambas tecnologías en un mismo sistema resulta un desafío.

Este trabajo propone una arquitectura, capaz de integrar DDS y Kafka, combinando sus fortalezas para facilitar la comunicación entre sistemas heterogéneos. A través de un caso de uso basado en datos sensóricos simulados de vehículos inteligentes, se implementa una solución completa que conecta dispositivos embebidos y ubicados en el *edge* con plataformas que pudieran encontrarse en la nube, con un despliegue automatizado. Además, se incorporan mecanismos de monitorización y se analizan métricas de rendimiento, lo que permite evaluar la viabilidad de la propuesta en escenarios industriales.

Palabras clave

IoT en el continuo, software de distribución de datos, Apache Kafka, Data Distribution Service (DDS), transmisión de eventos.

Abstract

The transition from Industry 4.0 to the IoT Continuum represents a fundamental shift in data processing models from cloud-centric to progressively distributed to the edge and devices, with the goal of reducing latency, increasing resiliency and enabling real-time decision capabilities across the entire architecture. This operational continuity poses significant challenges in terms of interoperability, latency and performance.

In this context, technologies such as DDS (Data Distribution Service) and Apache Kafka have proven to be effective, albeit in different scenarios. DDS stands out for its capabilities in distributed environments with strict real-time communication requirements, while Kafka is widely used to process large volumes of data in cloud architectures. Therefore, the coexistence of both technologies in the same system is a challenge.

This work proposes an architecture which integrates DDS and Kafka, combining their strengths to facilitate communication between heterogeneous systems. Through a use case based on simulated sensor data from smart vehicles, a complete solution is implemented that connects embedded and edge devices with platforms that could be found in the cloud, by means of an automated deployment. In addition, monitoring mechanisms are incorporated and performance metrics are analysed, allowing the feasibility of the proposal to be assessed in industrial scenarios.

Keywords

IoT Continuum, distribution middleware, Apache Kafka, Data Distribution Service (DDS), event streaming.

Índice

Índice de figuras

Índice de tablas

Glosario de acrónimos y siglas

CSV *Comma-Separated Values*. Formato de archivo donde los datos se representan en forma de tabla, separando las columnas mediante comas.

DDS *Data Distribution Service*. *Middleware* basado en publicación-suscripción, diseñado para comunicaciones distribuidas en tiempo real.

DOP *Dilution of Precision*. Métrica que evalúa la calidad de la señal GPS en función de la incertidumbre del posicionamiento. Subtipos:
HDOP: *Horizontal DOP*.
PDOP: *Position DOP*.
VDOP: *Vertical DOP*.

HTTP *Hypertext Transfer Protocol*. Protocolo de capa de aplicación para la transferencia de información en la red.

IDL *Interface Definition Language*. Lenguaje para la definición de tipos de datos en DDS.

IoT *Internet of Things*. Red de dispositivos interconectados entre sí, principalmente con equipación de sensórica, para el envío y recepción de datos.

JMX *Java Management Extensions*. Tecnología utilizada para la definición de servicios de administración o monitorización de aplicaciones Java.

JSON *JavaScript Object Notation*. Formato de texto ligero ideado para el intercambio de datos entre aplicaciones.

MTU *Maximum Transmission Unit*. Tamaño máximo permitido por paquete en un protocolo.

NAT *Network Address Translation*. Técnica de red utilizada en la simulación para la asignación de direcciones.

OMG *Object Management Group*. Organización responsable del desarrollo del estándar DDS, entre otros.

QoS *Quality of Service*. Conjunto de políticas configurables que definen fiabilidad, orden de entrega, latencia, etc.

RAM *Random Access Memory*. Memoria de acceso aleatorio para almacenamiento volátil.

RTI *Real-Time Innovations*. Compañía desarrolladora de RTI Connext DDS.

SSH *Secure Shell*. Protocolo para acceder remotamente a sistemas.

UDP *User Datagram Protocol*. Protocolo de capa de transporte usado por defecto en DDS.

V2I *Vehicle-to-Infrastructure*. Comunicación entre vehículo e infraestructura.

V2V *Vehicle-to-Vehicle*. Comunicación directa entre vehículos.

XML *eXtensible Markup Language*. Formato de configuración estructurada utilizado por DDS.

YAML Lenguaje de serialización comúnmente usado en ficheros de configuración.

1. Introducción

La creciente digitalización de los sistemas industriales, junto con la expansión del Internet de las Cosas (IoT), ha motivado la aparición de nuevos paradigmas que buscan acercar la ejecución de procesos lo más posible a la fuente de producción de los datos, a diferencia de la tendencia previa a derivar todo a la nube con el *big data*. En este contexto, la necesidad de integrar dispositivos IoT, nodos en el borde de la red (*edge*) y plataformas *cloud* ha dado lugar al concepto de IoT Continuum. Este paradigma, persigue una continuidad lógica y operativa entre los niveles de infraestructura, mejorando la eficiencia y latencia en la transmisión de datos al derivar procesamiento en el *edge*, pero manteniendo una respuesta flexible a las necesidades cambiantes de las aplicaciones, conservando el *cloud* para tareas de automatización, analítica o predicción, entre otras, cuando no existan restricciones de tiempo. Sin embargo, esta integración plantea numerosos retos. Entre ellos, destacan la heterogeneidad de dispositivos y protocolos, la necesidad de garantizar la calidad de servicio en las comunicaciones críticas y el tratamiento de grandes volúmenes de datos en tiempo real. Estos desafíos requieren soluciones robustas y escalables, que combinen la capacidad de respuesta en el *edge* con la potencia de procesamiento en el *cloud*.

Para hacer frente a los retos planteados, han surgido distintas tecnologías de distribución de datos, como las basadas en la comunicación de productores y consumidores, facilitando una mayor escalabilidad y resiliencia. Entre ellas, destacan DDS (Data Distribution Service), orientado a sistemas críticos con requisitos de comunicación en tiempo real o mayores restricciones de calidad de servicio, y Apache Kafka, ampliamente adoptado en contextos de análisis de grandes volúmenes de datos y arquitecturas centradas en el procesamiento en la nube. Ambas tecnologías han sido ampliamente utilizadas en la industria, si bien presentan enfoques muy diferentes en ámbitos como la persistencia, el descubrimiento de nodos en la red o la fiabilidad.

1.1. Objetivos

El objetivo principal de este trabajo es diseñar, implementar y evaluar una arquitectura de integración entre dos tecnologías clave en el ámbito de los sistemas distribuidos modernos: DDS y Apache Kafka. La solución planteada busca aprovechar las fortalezas de cada *middleware* para permitir la comunicación entre sistemas que operen en diferentes niveles del IoT Continuum, desde dispositivos o computadores embebidos en el borde de la red, hasta aquellos servicios encargados de tareas de análisis o predicciones en la nube. Como ejemplo al que aplicar esta solución, se ha empleado un caso de uso relacionado con vehículos inteligentes, un entorno en el cual la interoperabilidad entre sistemas de comunicación en tiempo real, la inferencia de IA, el mantenimiento de distancias con respecto a obstáculos, las plataformas de análisis en la nube y el entrenamiento de modelos IA resultan especialmente críticos.

Para cumplir con este objetivo general, se han completado las siguientes actividades:

- **Analizar y comparar las características técnicas de DDS y Kafka**. Con el fin de identificar los mejores escenarios para cada tecnología y las ventajas de una sobre la otra, se ha realizado un estudio detallado de ambos *middleware*, con principal énfasis en la comparación de sus prestaciones en términos de latencia, *throughput*, escalabilidad, fiabilidad y arquitectura.
- **Estudiar e implementar la interconexión DDS-Kafka**. Con el objeto de permitir la transmisión de datos entre ambas tecnologías, se ha estudiado el uso de RTI Routing Service y sus opciones de configuración, utilizando así una herramienta capaz de establecer flujos de datos entre dominios DDS y destinos externos como Apache Kafka.
- **Diseñar un caso de uso representativo**. Se ha procesado y adaptado un conjunto de datos real para simular la circulación de vehículos en un tramo de carretera. Este se ha utilizado para

validar las conclusiones obtenidas en los análisis teóricos de las tecnologías y verificar el correcto funcionamiento e interconexión de estas en una transmisión continua de eventos.

- **Instrumentar y caracterizar el sistema**. Mediante las capacidades de monitorización de las que disponen ambos *middleware* y el uso de herramientas complementarias para la recogida de métricas y la elaboración de *dashboards*, se ha evaluado el rendimiento del sistema, especialmente, en términos de latencia y *throughput*.
- **Desarrollar mecanismos de automatización del despliegue**. Con objeto de aplicar el despliegue del caso de uso en varios dispositivos, se automatizan los procesos de instalación de los componentes y su arranque y parada.
- **Analizar los resultados obtenidos**. Mediante el estudio de las métricas obtenidas, es posible validar las hipótesis realizadas en el análisis teórico, observando las ventajas de cada tecnología, identificando patrones de funcionamiento y planteando posibles líneas de mejora o trabajo a futuro.

Con el desarrollo de estas actividades, se pretende proporcionar una solución interoperable que sirva no solo como validación de las prestaciones de los *middleware* estudiados y sus ventajas, sino también como una guía práctica para abordar la integración de los mismos en el contexto del IoT Continuum.

1.2. Arquitectura de la solución

Debido a la naturaleza distribuida y heterogénea del sistema diseñado, ha sido necesario el uso de un conjunto variado de herramientas y tecnologías, desempeñando cada una de ellas un papel concreto dentro del flujo de datos. La figura 1 expone una representación gráfica de la solución software implementada.

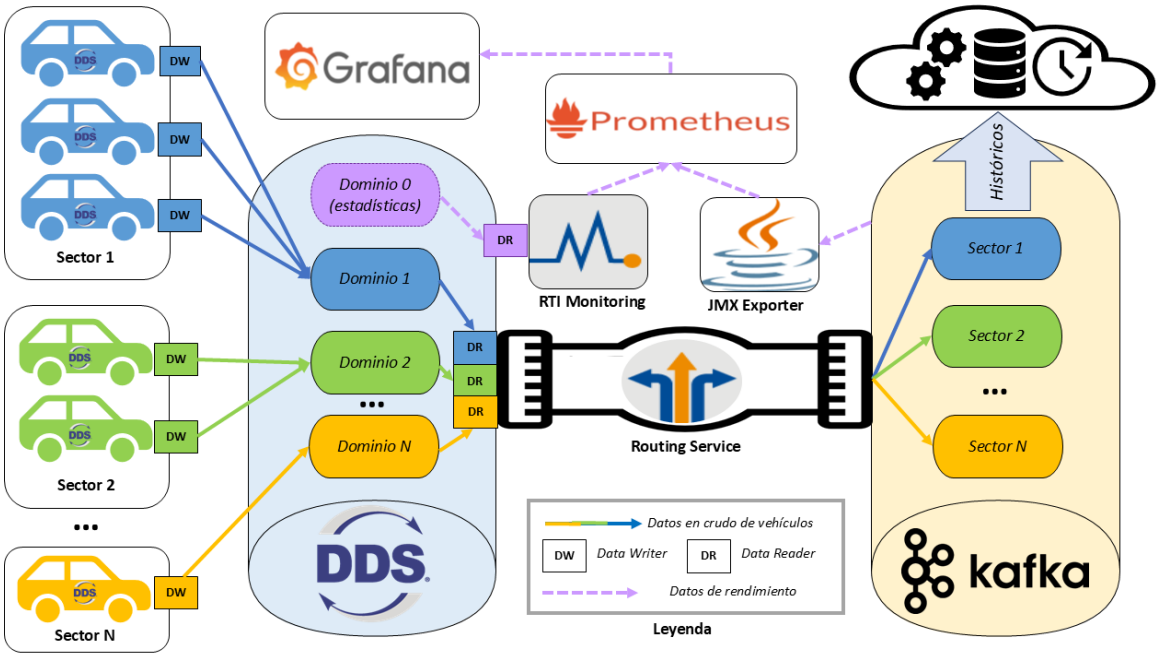


Figura 1: Arquitectura y solución software propuesta.

En ella se pueden observar distintas agrupaciones de vehículos distribuidos por sectores que publican datos en dominios DDS independientes. Estos datos son posteriormente redirigidos por un servicio de enrutamiento (Routing Service), que actúa como puente entre DDS y Kafka. Cada sector se asocia a un dominio concreto, y el flujo de datos se canaliza hacia los correspondientes tópicos en Kafka, donde posteriormente pueden ser persistidos, procesados o consumidos por aplicaciones externas, principalmente en la nube. Asimismo, el sistema incluye también una capa de monitorización del rendimiento de los distintos componentes. Por una parte, RTI Monitoring permite el acceso a las métricas internas del sistema DDS, mientras que JMX Exporter extrae métricas del *broker* Kafka, siendo ambas recopiladas por Prometheus para su posterior visualización en Grafana. Los principales componentes se describen a continuación:

- **RTI Connext DDS 7.5.0**. Es la implementación desarrollada por Real-Time Innovations (RTI) [16] del estándar DDS (Data Distribution Service), propuesto por el Object Management Group (OMG) [4], a la que se ha podido acceder mediante la solicitud de una licencia de investigación universitaria. Esta herramienta permite la comunicación distribuida mediante un modelo de publicación-suscripción desacoplado y altamente configurable en materia de políticas de *Quality of Service*. Además, dentro del *framework* completo, se han utilizado especialmente los siguientes componentes:
  - **Connext Gateway - Routing Service**. Un componente esencial que actúa como puente entre dominios DDS, así como entre DDS u otros sistemas externos según se configure. En este último caso, dicha configuración se ha centrado en la redirección de datos desde un *DataReader* de un dominio específico DDS hacia un adaptador de publicación en Kafka.
  - **RTI Monitoring Library**. Esta librería, utilizada junto con Python, permite acceder a tópicos internos de DDS para observar el estado y rendimiento de los recursos creados (dominios, servicios, tópicos, etc.).
- **Apache Kafka**. Plataforma de gestión de eventos distribuida, de alto rendimiento y basada en la persistencia temporal de eventos en logs particionados. Su aplicación en la arquitectura permite el almacenamiento temporal, la replicación y la distribución de los datos exportados desde DDS hacia sistemas externos, generalmente enfocados al procesamiento o la visualización de dicha información, especialmente en entornos *cloud*.
  - **Productores y consumidores de Kafka**. Implementados para la validación del correcto funcionamiento del puente con DDS, así como la elaboración de pruebas de rendimiento comparativas.
- **Python**. Lenguaje de programación empleado con la API de RTI Connext DDS y con la de Kafka, para la lectura de tópicos de monitorización, envío de datos y tests de rendimiento.
- **XML**. Lenguaje de marcado extensible necesario para la configuración de DDS y su puente con Kafka, incluyendo posibles transformaciones de datos, filtros y conexiones.
- **IDL**. Lenguaje de descripción de interfaces. Se ha utilizado para la definición de los tipos de datos a utilizar en DDS y su posterior generación automática al lenguaje Python.
- ***Scripts* de Bash**. Conjunto de instrucciones definidas en varios ficheros modulares para facilitar el despliegue de la arquitectura propuesta junto con el caso de uso, así como la instalación de dependencias y la ejecución coordinada de diferentes procesos implicados en los equipos utilizados.
- **JMX Exporter**. Programa configurado como un *Java Agent* lanzado junto con el *broker* de Kafka, permitiendo exponer así sus métricas de rendimiento mediante un servidor HTTP.

- **Prometheus**. Sistema de monitorización basado en series temporales y en el *scraping* periódico sobre una determinada ruta HTTP. Se utiliza para recolectar y almacenar las métricas de rendimiento tanto de Kafka como de RTI DDS.
- **Grafana**. Herramienta de visualización integrada con Prometheus y configurada para la construcción de *dashboards* interactivos, los cuales permiten observar el rendimiento en tiempo real del sistema y facilitan el análisis del comportamiento de la arquitectura propuesta.

1.3. Caso de estudio

Para validar y estudiar el sistema propuesto se ha utilizado el escenario 36 del conjunto de datos abierto DeepSense 6G [6]. Este proporciona datos y métricas capturados de sensórica real desplegada en contextos urbanos o interurbanos para su aplicación en sistemas inteligentes y comunicaciones V2I (Vehicle-to-Infrastructure) y V2V (Vehicle-to-Vehicle).

A continuación, se describe de forma detallada el conjunto utilizado, pudiéndose encontrar en el Anexo A una muestra de los datos originales y su análoga modificada para este caso de estudio, como se explicará en el apartado 3.2.

Vehículos participantes y sensórica

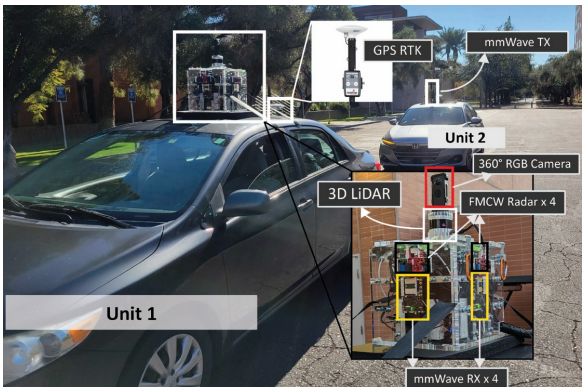
- *Vehículo 1 (principal)*. Dispone de:
  - Cuatro receptores de señal mmWave de 60 GHz ubicados en cada lateral del vehículo.
  - Una cámara RGB de 360º.
  - Cuatro radares FMWC, los cuales resultan útiles para:
    - Medición de distancias contra obstáculos. La diferencia de frecuencia entre la señal transmitida y la recibida puede relacionarse con el tiempo de viaje de la onda, y en consecuencia, con la distancia hacia el objeto.
    - Medición de velocidad relativa. La señal reflejada cuando un objeto está en movimiento, presenta un desplazamiento adicional en la frecuencia debido al efecto Doppler, lo que permite calcular la velocidad.
  - Un escáner LiDAR 3D, útil para mapear el entorno en tiempo real mediante pulsos láser.
  - Un sistema GPS RTK, capaz de proporcionar posiciones con una precisión aproximada de medio metro.
- *Vehículo 2 (comunicación)*. Su papel se encuentra reducido únicamente a la comunicación con la unidad 1, contando para ello con una antena omnidireccional y un sistema GPS RTK idéntico al de la primera unidad, que permite localizar también al segundo vehículo. Su información no fue utilizada en el caso de estudio.

En la figura 2, pueden observarse los instrumentos y sensores empleados para la generación del *dataset* por DeepSense.

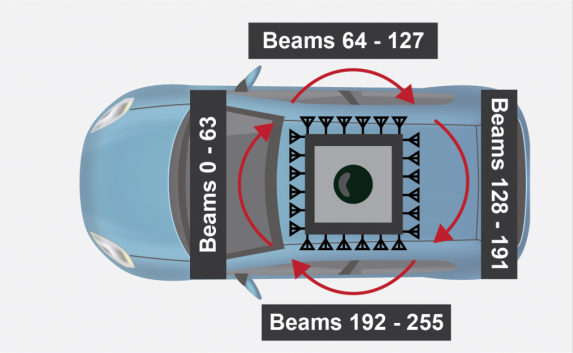
Campos seleccionados

Debido al enfoque del trabajo, centrado en la transmisión de eventos entre vehículos, se han seleccionado los siguientes campos como los más relevantes para el uso en el sistema DDS-Kafka:

- **ABS_Index**. Identificador único para cada muestra. Útil para trazabilidad y control de secuencias.



(a) Vehículos y sensórica.



(b) Distribución de haces del sistema LiDAR.

Figura 2: Instrumentación usada para la elaboración del *dataset* por DeepSense 6G [5].

- **timestamp**. Marca temporal de la medición en formato *hh-mm-ss.us*. Permite mantener el orden de las muestras y detectar posibles interrupciones o retrasos.
- **seq_index**. Índice con el que se agrupan las muestras en bloques de 20 segundos. También puede incrementarse su valor de secuencia si en algún momento se interrumpe la toma de datos a una frecuencia de 10 Hz.
- **unit1_gps1_lat, unit1_gps1_long**. Coordenadas geográficas de alta precisión (8 y 7 decimales, respectivamente).
- **unit1_gps1_altitude**. Altitud sobre el nivel del mar con 7 decimales.
- **unit1_gps1_hdop, pdop, vdop**. Son indicadores de calidad de la señal GPS. Su análisis puede ser útil para la detección de condiciones adversas o fallos en la localización.

Para llevar a cabo la simulación de la existencia de múltiples vehículos en un escenario distribuido, se han aplicado variaciones controladas sobre las posiciones y tiempos registrados por el Vehículo 1. Estas transformaciones incluyen:

- Desfase temporal entre muestras mediante la modificación del *timestamp*.
- Asignación de identificadores únicos por vehículo (`vehicle_id`).
- Variaciones en las coordenadas GPS para simular trayectorias con un ligero desplazamiento.
- Alteración de los valores DOP para la simulación de errores o degradación en la señal.

Justificación del uso

El conjunto de datos ofrece una base sólida adecuada para poder simular un entorno distribuido con tráfico de vehículos en diferentes nodos, los cuales publiquen información periódica en tiempo real. Especialmente, la información de la que se dispone permite:

- Realizar agrupaciones de vehículos por su ubicación geográfica.
- Estimar velocidades de los vehículos.
- Simular distintos niveles de calidad de señal o cobertura GPS.

Por otra parte, se ha optado por no utilizar algunos de los campos ofrecidos, como los relativos a datos capturados por sensores de alta carga, tales como pueden ser el LiDAR 3D o las cámaras RGB 360, debido a que:

- Su inclusión en el caso de uso podría enmascarar el rendimiento real del *middleware* al sobrecargar la transmisión de datos.
- El análisis de la información recabada por estos sensores requiere de procesos adicionales (compresión, segmentación, reconstrucción...), lo cual desviaría el foco del objetivo principal del TFM, centrado en evaluar el comportamiento y la interoperabilidad de las tecnologías utilizadas.
- Finalmente, gracias a la naturaleza periódica y estructurada de los datos GPS junto con sus indicadores DOP, estos resultan más adecuados para la simulación de comunicaciones y eventos V2V.

En resumen, esta decisión permite centrar el análisis en el flujo de eventos en tiempo real y en una evaluación de la interconexión entre DDS y Kafka, además de métricas globales como la latencia o el *throughput* de la solución desarrollada.

1.4. Metodología

El desarrollo del TFM se ha llevado a cabo utilizando una metodología iterativa y práctica, centrándose esta en la investigación, implementación y evaluación progresiva de la arquitectura planteada. Dado el carácter eminentemente técnico y experimental del trabajo, las diferentes tareas realizadas se han estructurado en varias fases interrelacionadas entre sí, evolucionando por lo tanto algunas de ellas de forma paralela.

Las principales etapas han sido las siguientes:

1. **Estudio preliminar y revisión del estado del arte**. Se inició un análisis introductorio, y más adelante detallado, de DDS, contrastándolo con Apache Kafka, ya usado previamente [13]. Esto incluye la revisión de documentación técnica y manuales de ambas tecnologías, así como posibles casos de uso existentes. Debido a las diversas implementaciones existentes de DDS, también fue necesario investigar las más utilizadas o soportadas, con el fin de seleccionar la más adecuada para el presente trabajo.
2. **Concepción y diseño de una arquitectura de integración, junto con la aplicación a un caso de uso**. Una vez identificadas las mayores diferencias entre ambos *middleware* se definió una topología capaz de explotar sus principales ventajas y, al mismo tiempo, compensar sus respectivas limitaciones. Este trabajo se realizó en paralelo al planteamiento de un caso de uso basado en la simulación de vehículos, con el fin de facilitar su aplicación en un entorno realista y evaluable.
3. **Selección del conjunto de datos y familiarización con RTI DDS**. Después de valorar varios *datasets* se eligió el conjunto elaborado por DeepSense por su calidad, realismo y adecuación al tipo de datos que pueden circular en un entorno V2X (especialmente V2V). Paralelamente, se inició el trabajo con RTI Connext DDS, partiendo de los ejemplos propuestos por la misma organización, generando tipos a partir de ficheros IDL, explorando los tópicos internos para su monitorización y realizando pruebas de publicación y suscripción para comprender su funcionamiento.
4. **Preparación del conjunto de datos y simulación de vehículos**. A partir de las muestras originales de la Unidad 1 del *dataset*, se diseñó una estrategia para la simulación de múltiples vehículos, aplicando variaciones en las trayectorias, marcas temporales y la agrupación por sectores en base a la ubicación.

5. **Implementación de los componentes DDS y monitorización**. Se desarrollaron los publicadores DDS que alimentan cada dominio, así como la configuración asociada al Routing Service para la redirección de datos a Kafka o entre dominios DDS. Al mismo tiempo, se habilitó la monitorización interna de DDS y Kafka, con la consecuente captura de información sobre el uso de recursos, latencias y *throughput*, incluyendo el Routing Service entre ambos *middleware*.

6. **Automatización del despliegue**. Se escribieron *scripts* capaces de instalar automáticamente todos los programas y dependencias necesarios, copiar los ficheros de configuración y datos, así como ejecutar los nodos de simulación mediante acceso remoto por SSH. Esta automatización permitió replicar de una forma más rápida y sencilla desde un solo equipo el entorno completo, optimizando así el tiempo y la consistencia de las pruebas. Además, para verificar su correcto funcionamiento, se probaron y mejoraron en máquinas virtuales.

7. **Despliegue y evaluación**. Finalmente, se realizó el despliegue en una red NAT virtualizada, analizando y contrastando las métricas obtenidas, junto con la correspondiente extracción de conclusiones sobre la viabilidad de su aplicación en entornos reales.

1.5. Estructura de la memoria

Esta memoria se encuentra organizada en seis capítulos, además de la presente introducción, la bibliografía y los anexos. A continuación, se describe brevemente el contenido de cada capítulo:

- **Capítulo 2 - Análisis de DDS, Kafka y su mecanismo de integración**. Presenta las tecnologías clave utilizadas en este trabajo, especialmente Apache Kafka, DDS y las implementaciones consideradas de este último. Asimismo, describe el papel desempeñado por RTI Routing Service como mecanismo de interconexión entre ambos *middleware*, detallando su funcionamiento en la arquitectura que se plantea. Además, se realizan tests de *throughput* y latencia con un número reducido de equipos, para verificar que la comparativa teórica es correcta y existe una razón de ser para la integración de Kafka y DDS.

- **Capítulo 3 - Arquitectura propuesta y desarrollo del caso de uso**. Detalla el diseño de la arquitectura de integración entre DDS y Kafka, aplicada de forma práctica a un caso de uso basado en la simulación de vehículos distribuidos por sectores. También se describen los flujos de datos existentes, así como la configuración pertinente de los dominios DDS, su conexión con Kafka y otros aspectos prácticos del desarrollo.

- **Capítulo 4 - Automatización del despliegue**. Explica el proceso de despliegue automatizado seguido sobre una red NAT virtualizada con varias máquinas, incluyendo la instalación de las dependencias necesarias, copia de ficheros y ejecución de procesos. Se describen también los *scripts* elaborados para ello y su funcionamiento.

- **Capítulo 5 - Visualización y validación del sistema mediante Grafana**. Presenta las métricas del comportamiento general del sistema, esto es, latencias, *throughput* y uso de recursos de DDS, Kafka y el resto de herramientas integradas en el trabajo.

- **Capítulo 6 - Conclusiones y líneas futuras de aplicación**. Expone las principales conclusiones obtenidas a la finalización de este TFM, destacando los logros alcanzados. Por último, se proponen las posibles líneas de trabajo futuras y mejoras que podrían implementarse en el desarrollo alcanzado.

Todo el código desarrollado para el proyecto se puede encontrar en el repositorio GitHub asociado al mismo [14].

2. Caracterización de DDS, Kafka y mecanismo de integración

Antes de describir la arquitectura desarrollada y la implementación llevada a cabo, resulta necesario presentar de forma más detallada las tecnologías que forman parte de la solución planteada. En particular, se analizan los fundamentos de DDS (Data Distribution Service) y Apache Kafka, dos soluciones ampliamente utilizadas en entornos distribuidos, aunque con enfoques sustancialmente diferentes.

El objetivo de este capítulo es, por lo tanto, examinar el funcionamiento interno, las fortalezas y limitaciones de ambas tecnologías, así como justificar la elección de herramientas concretas, como RTI Connext DDS frente a otro tipo de implementaciones disponibles en la industria. De igual forma, se plantea RTI Routing Service como el componente sobre el que recaerá la interconexión entre DDS y Kafka. Por último, se realiza una comparativa inicial de rendimiento entre DDS y Kafka, como complemento a la justificación teórica que motiva su integración.

2.1. DDS y configuración

El estándar DDS (Data Distribution Service) es una especificación publicada y gestionada por el Object Management Group. Su modelo, al igual que el de Apache Kafka, se basa en un esquema publicación-suscripción, pero en este caso completamente descentralizado, eliminando así la necesidad de un *broker* centralizado y favoreciendo el desacoplamiento entre productores y consumidores de datos. Además, se encuentra especialmente orientado a sistemas distribuidos que requieran comunicaciones en tiempo real con bajas latencias y una alta fiabilidad.

Entre las características que hacen atractivo el uso de DDS destacan:

- **Descubrimiento automático**. Los participantes de un mismo dominio DDS son capaces de detectar de forma autónoma la existencia de nuevos publicadores o suscriptores, lo cual facilita en gran medida la escalabilidad del sistema y, en cierto modo, su flexibilidad relativa a la creación de nuevas conexiones de nodos.
- **Modelos de datos fuertemente tipados**. Los datos que se publican en los tópicos DDS se definen utilizando interfaces en IDL, lo cual permite un tipado fuerte, de forma similar a las estructuras (`struct`) del lenguaje C.
- **Políticas configurables de calidad de servicio**. Los parámetros de calidad configurables permiten ajustar los niveles de fiabilidad, durabilidad, latencias máximas, orden de entrega e historiales de los datos, en función de las necesidades del sistema.
- **Operación distribuida con alta tolerancia a fallos**. Dado que no se depende de nodos o *brokers* centrales, la operación es más resiliente ante redes con mayor dinamismo, precisamente como pueden ser las presentes en entornos industriales y de movilidad.
- **Soporte para *multicast***. El uso de este tipo de mecanismos de envío, mejora las comunicaciones al permitir la reducción de carga presente en la red.

Este conjunto de propiedades, permiten identificar a DDS como una de las soluciones idóneas para la construcción de aplicaciones críticas en la automatización industrial, automoción u otros tipos de sistemas embebidos. En el contexto de este trabajo, se aprovecha especialmente la capacidad de DDS para operar con distintos dominios independientes y su mecanismo de descubrimiento, facilitando así la segmentación y agregación por sectores de los datos y sus dispositivos generadores.

2.1.1. Componentes principales del modelo DDS

A continuación, y con el fin de facilitar la comprensión del modelo DDS [17] al lector, se realiza la descripción de algunos de sus principales elementos, junto con una representación gráfica de los mismos que se muestra en la figura 3.

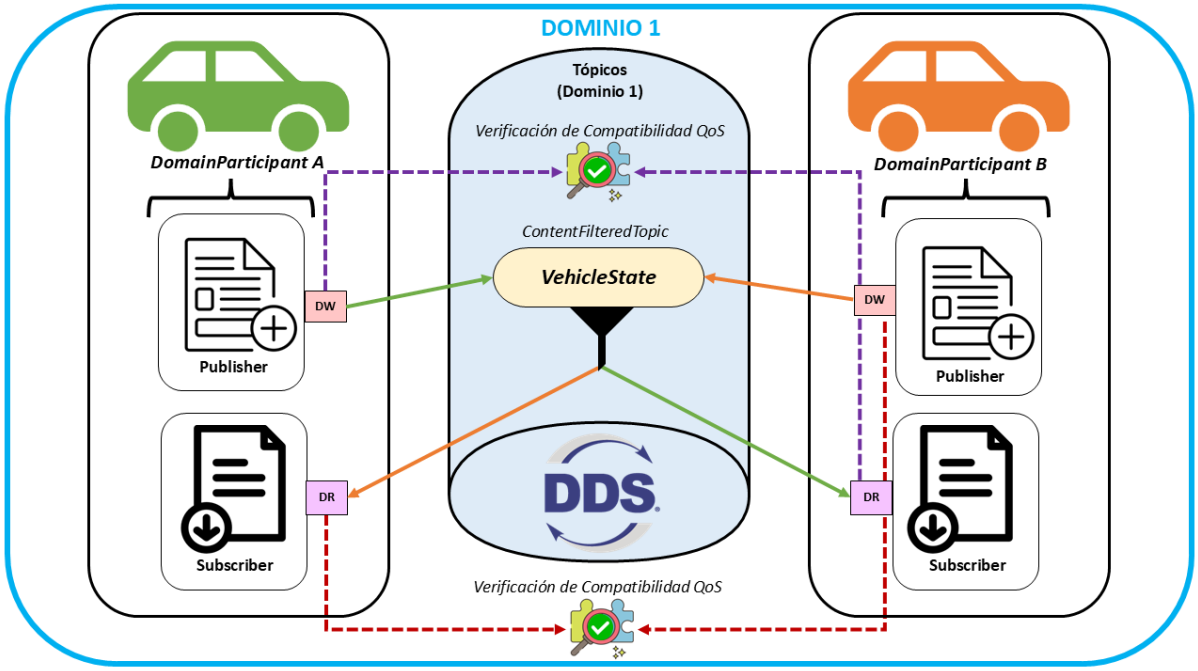


Figura 3: Representación de las entidades de un modelo DDS.

- **DomainParticipant**. Es la entidad raíz de las aplicaciones DDS, y por lo general, solo es necesaria una por cada aplicación. Representa un nodo dentro del dominio de comunicación y se encarga de la gestión de los recursos asociados. En el caso de uso empleado, sería equivalente a cada uno de los vehículos simulados, los cuales pertenecerán a un determinado dominio, que se asigna en función del sector de la vía en el que se encuentran.
- **Topic**. Se trata de un canal lógico de comunicación. Este viene definido por su nombre, el tipo de dato asignado y el conjunto de propiedades de QoS. Los tópicos serán los canales de comunicación entre publicadores y suscriptores. Por ejemplo, *VehicleState*, con su correspondiente estructura de datos definida mediante IDL. Además, se permite también el uso de una variante especial (*ContentFilteredTopics*), que permite a los *DataReaders* establecer ciertos filtros en los datos leídos.
- **DataWriter**. Entidad encargada de la publicación de datos en un tópico. Por simplicidad, el caso de uso cuenta con uno por vehículo, aunque en la realidad podrían existir varios, uno vinculado a cada sensor.
- **DataReader**. Su funcionalidad es exactamente la opuesta al *DataWriter*, encargándose de la suscripción y la lectura de datos de un tópico. Su configuración de QoS debe ser compatible (igual o menos exigente) con respecto a la del publicador y viceversa.

- **Publisher/Suscriber**. Son agrupaciones lógicas de las dos entidades previamente definidas, resultando de utilidad en el caso de compartir configuraciones o mejorar la eficiencia en el acceso a los datos. En el caso de uso, dado que por lo general solo se definen un *DataWriter* y un *DataReader* por *DomainParticipant*, solo existirá un *publisher* y un *suscriber* respectivamente para cada uno.
- **QoS (Quality of Service)**. DDS permite el manejo de amplios conjuntos de políticas para adaptar el comportamiento de la entrega de datos, en función de los requisitos de las aplicaciones.
- **Mecanismo de descubrimiento automático**. DDS incluye su propio protocolo interno que permite que los diferentes *DomainParticipant* se detecten automáticamente entre sí en el mismo dominio, si su QoS es compatible, sin necesidad de configuraciones o asignaciones manuales o centralizadas. La definición de los *DomainParticipant* no es solo de gran importancia para no duplicar el uso de recursos, sino para garantizar también que el descubrimiento sea más eficiente y no cause saturación en la red.

2.1.2. Implementación DDS

Son múltiples las implementaciones existentes de DDS, tanto de código abierto como de carácter comercial o propietario. Entre las más destacadas y que se han evaluado en favor del estudio realizado por Bode et al. [2], se encuentran:

- **RTI Connext DDS**. Solución propietaria desarrollada por Real-Time Innovations. Destaca por su soporte completo del estándar, comunidad amplia y herramientas avanzadas para su configuración, monitorización y enrutamiento, además de una amplia documentación.
- **Fast DDS**. Destaca especialmente en robótica, aunque su configuración puede ser más manual y complicada en escenarios de mayor complejidad.
- **Cyclone DDS**. Implementación de código abierto que mantiene la Eclipse Foundation. Destaca por su modularidad y también se emplea en entornos relacionados con la robótica, aunque su rendimiento puede ser algo inferior.
- **OpenDDS**. Es una de las implementaciones de código abierto con más recorrido, aunque presenta limitaciones en cuanto a su integración con herramientas modernas, como puede ser Kafka.

De entre las alternativas consideradas, la elección de RTI Connext DDS se debe a los siguientes motivos:

- **Rendimiento**. Como se comprueba en el estudio mencionado previamente que compara el rendimiento de diversas implementaciones DDS [2], RTI demuestra un comportamiento que es ligeramente superior a Fast DDS y Cyclone DDS, especialmente en términos de latencia y ancho de banda.
- **Herramientas avanzadas**. La disponibilidad de componentes como Routing Service, Admin Console (una consola gráfica de administración, útil para debuggear) o Monitoring Library, ha resultado útil y de especial interés para el desarrollo de este TFM, especialmente en lo relativo a la observación de métricas de rendimiento internas de las entidades DDS y la conexión realizada con Kafka.
- **Facilidad de integración**. Permite trabajar con distintos lenguajes de programación (C, C++, Java y Python) y además dispone de un generador de tipos desde IDL para cada uno de estos lenguajes, entre otros formatos.

- **Disponibilidad de licencias académicas y de investigación**. La posibilidad de disponer de una licencia universitaria (previa solicitud), facilita el acceso al ecosistema completo y las diferentes versiones disponibles de RTI Connext DDS.

Cabe mencionar que también se valoró el uso de Zenoh, el cual es un proyecto relativamente reciente de la fundación Eclipse, que busca integrar el paradigma de publicación y suscripción junto con el almacenamiento de datos y su consulta. Aunque Zenoh presenta una arquitectura ligera que también promete alto rendimiento, como así revela el estudio realizado por Liang, Yuan, & Lin [11], debido a que aún carece del mismo nivel de documentación, adopción industrial e integración con Kafka, se ha descartado su uso en favor de DDS.

2.2. Apache Kafka

Apache Kafka es una plataforma distribuida que permite la publicación, suscripción y el procesamiento de flujos de datos, mantenida actualmente por la Apache Software Foundation. Su modelo se encuentra basado en el registro de logs distribuidos, donde los mensajes son escritos secuencialmente en una o diversas particiones (si se habilita su replicación) y pueden ser consumidos paralelamente por diferentes procesos, utilizando como canal para ello el mismo concepto de "tópico" empleado por DDS y que es común a varios modelos de publicación-suscripción.

Además, cabe destacar que Kafka se encuentra diseñado para la operación en entornos que requieren alto rendimiento en la transmisión y procesamiento de grandes volúmenes de datos. En base a su documentación [1] y el TFG realizado previamente por el mismo autor de este TFM [13], se pueden señalar las siguientes características como unas de las más relevantes de esta plataforma:

- **Persistencia de mensajes**. El almacenamiento de los datos en logs persistentes, permite que los consumidores accedan a ellos incluso si se incorporan al sistema después de su publicación. Además, pueden establecerse plazos para la limpieza de estos mensajes. En comparación con DDS, esta plataforma se encuentra más orientada a los históricos de eventos por defecto, mientras que en DDS es necesaria una configuración concreta de QoS para este fin.
- **Alta disponibilidad y tolerancia a fallos**. Si bien es un sistema más centralizado que el propuesto por el estándar DDS, los *brokers* permiten replicar las particiones de los tópicos en diferentes nodos, lo cual garantiza que no haya una caída del sistema en cuanto uno o un número reducido de nodos fallen o no se encuentren accesibles.
- **Escalabilidad horizontal**. Gracias a que su arquitectura contempla el uso de particiones, la capacidad de procesamiento de Kafka puede verse incrementada con la inclusión de un mayor número de *brokers*, pudiendo distribuir mejor la carga de trabajo entre ellos.
- **Consumo asíncrono**. Gracias al uso de logs, los consumidores pueden leer los mensajes de los tópicos al ritmo que les resulte más conveniente. Esto permite un mayor desacoplamiento con respecto a la producción, al no depender de plazos para la caducidad de los datos (a no ser que sea de interés especificarlo de forma explícita).
- **Alto *throughput***. El particionamiento de los datos en diferentes nodos junto con el uso de configuraciones y una arquitectura que maximiza el rendimiento de las lecturas y escrituras secuenciales en disco, han permitido a Kafka desde sus inicios contar con un *throughput* destacable, como se puede observar en estudios realizados por LinkedIn [10], entidad que inició su desarrollo antes de ser mantenido por Apache.
- **Integración y compatibilidad con sistemas enfocados al *cloud* y *Big Data***. Al ser compatible con herramientas como Kubernetes, Flink y Spark, entre otras, Kafka se elige frecuentemente para entornos que requieren analizar *pipelines* complejos o en tiempo diferido, en lugar de tiempo real.

2.2.1. Componentes principales de un ecosistema Kafka

Al igual que en la subsección anterior con DDS, y con el propósito de familiarizar al lector con la terminología de Kafka, se describen a continuación sus elementos principales, apoyándose en la representación gráfica que se muestra en la figura 4.

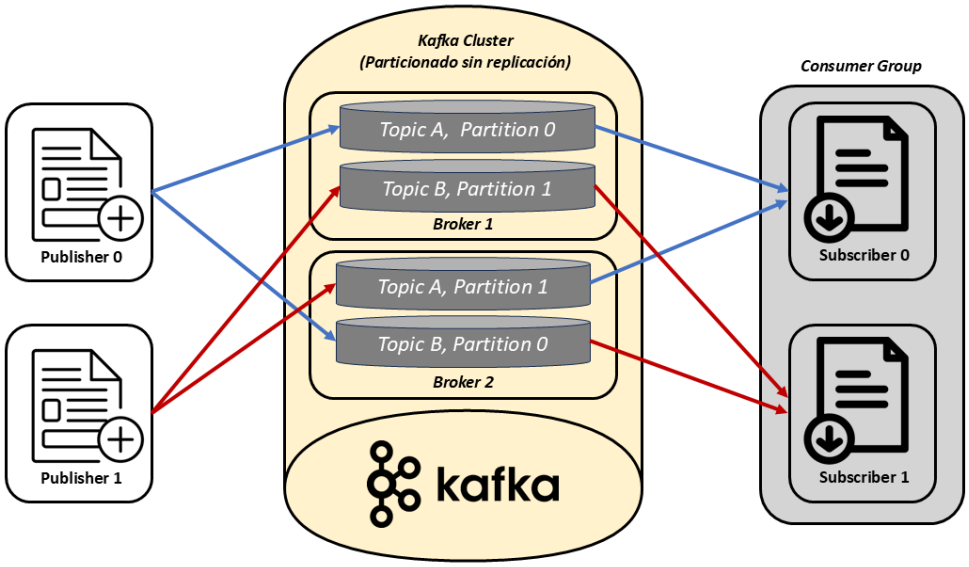


Figura 4: Representación de las entidades de una arquitectura de Kafka.

- **Producer**. Entidad encargada de la publicación de mensajes en uno o varios tópicos, enviando para ello los datos al *broker* que corresponda.
- **Consumer**. De forma análoga al productor, se suscribe a uno o varios tópicos para consumir los mensajes almacenados en los mismos, o que se estén generando en tiempo real.
- **Topic**. Como ya se definió previamente, se trata de un canal lógico que permite la comunicación entre publicadores y consumidores, agrupando mensajes por categorías (establecidas por los nombres de los tópicos). Cada uno de estos tópicos puede dividirse en varias particiones y así escalar de forma horizontal.
- **Brokers**. Son los servidores encargados de almacenar las particiones donde se encuentran los tópicos. Por lo general, los clústeres suelen contar con varios *brokers*, con el fin de soportar una mayor tolerancia a fallos o un mayor rendimiento.
- **Partition**. Es la unidad de almacenamiento empleada por los tópicos. Un tópico puede tener varias particiones para permitir así su lectura en paralelo y garantizar la redundancia.
- **Consumer Groups**. Son agrupaciones lógicas de consumidores para la lectura paralela de tópicos, lo que garantiza la coordinación entre ellos y evita duplicaciones en dicha lectura.
- **ZooKeeper**. Se trata de un servicio de coordinación, también propiedad de Apache, que se utiliza como sistema auxiliar para la coordinación de los *brokers* o nodos del clúster. No obstante, en las últimas versiones ya se ofrece una arquitectura completamente independiente, KRaft, que no requiere el uso de este servicio.

En este TFM, Kafka es utilizado como el destino final de los datos agregados desde los diferentes dominios DDS existentes, lo cual permitiría posteriormente al análisis de los mismos o su procesamiento en la nube para otros fines, como predicciones. Además, debido a su capacidad de almacenar mensajes en el tiempo y permitir su procesamiento de forma asíncrona, se convierte en una solución idónea para este fin.

Para un análisis más detallado del funcionamiento interno de Kafka o sus configuraciones, se recomienda la lectura del Trabajo de Fin de Grado realizado por el mismo autor [13], donde se profundiza en algunos de los aspectos mencionados y su integración con herramientas de análisis y monitorización.

2.3. Componente de interconexión

Si bien, como se ha podido leer, DDS y Kafka son tecnologías de mensajería ampliamente utilizadas y aplicables en los sistemas distribuidos, estas presentan enfoques arquitectónicos diferentes, lo cual dificulta su interconexión cuando se desea utilizar ambos *middleware* en un solo sistema. Mientras DDS se centra en un modelo de publicación-suscripción más descentralizado, basado en el descubrimiento automático y configuraciones estrictas de calidad de servicio, Kafka se enfoca en una mayor centralización con la intermediación de *brokers*, cierto grado de persistencia y un relativo acoplamiento entre algunos grupos de consumidores.

Con el objetivo de permitir la interoperabilidad entre los dos entornos, ha sido necesario encontrar un mecanismo o intermediario capaz de ello. En este caso, dicho intermediario es un componente ofrecido por RTI, denominado Routing Service. Este permite el intercambio de información entre diferentes dominios DDS, o debidamente configurado, entre DDS y otras tecnologías externas. En este apartado, se realiza una introducción al funcionamiento de Routing Service, su arquitectura y cómo se puede emplear para una conexión entre DDS y Kafka, empleando un adaptador oficial proporcionado también por RTI.

2.3.1. RTI Routing Service

RTI Routing Service es uno de los componentes oficiales ofrecidos por el ecosistema de Connext, con el fin de facilitar la conexión entre dominios y otros sistemas externos, entre los que pueden estar bases de datos y plataformas de procesamiento de eventos, entre otros. De esta forma, se convierte en una solución intermedia, con mayor desacople entre tecnologías y con un amplio espectro de configuración, sin necesidad de modificar el código original de los publicadores o suscriptores que ya se encuentran funcionando en el sistema.

Además, en lugar de requerir generar nuevo código en las aplicaciones, puede configurarse únicamente mediante el uso de ficheros XML, en los que declarar los datos a recibir, las transformaciones que se deben realizar o los reenvíos y rutas a seguir hacia otros destinos.

2.3.2. Arquitectura interna: Inputs, Outputs y Routes.

A lo largo del manual y especificación del servicio de enrutamiento [15] se pueden encontrar una gran variedad de etiquetas y elementos para llevar a cabo la configuración de comunicación, aunque con la finalidad de introducir una idea general al lector para el entendimiento del sistema, se describen a continuación los elementos más importantes de su arquitectura, aparte de otros ya mencionados del propio modelo DDS:

- **Inputs**. Son las fuentes de datos del servicio, las cuales por lo general son *DataReaders* suscritos a algún tópico de un dominio DDS.

- **Outputs**. Vienen marcados por los destinos de los datos que se redirigen con el servicio. En el caso de redirigir información entre dominios, suelen ser *DataWriters*, mientras que si el destino es un sistema externo, como el caso de Kafka, el destino resultará ser un adaptador hacia dicho sistema.
- **Routes**. Definen el flujo que deben seguir los datos desde el *input* hasta el *output* dentro del servicio de enrutamiento. En una misma configuración se pueden definir diferentes rutas, lo que aporta mayor flexibilidad en la redistribución.
- **Transformations**. Permiten modificar los formatos de los datos, antes de su envío. Esto resulta útil, ya que Kafka y DDS no emplean el mismo formato en la información (JSON vs estructura definida por IDL).

Además, cabe destacar que estos elementos pueden organizarse bajo "sesiones" durante su configuración, funcionando como unidades lógicas de procesamiento. Esto permite modularizar el sistema en caso de contar con diferentes flujos o tipos de datos involucrados, facilitando así también su monitorización y gestión. Toda esta configuración se realiza mediante ficheros XML donde se definen los dominios, tipos de datos o políticas de QoS utilizados.

2.3.3. Integración con Apache Kafka

Esta integración es posible mediante el uso del Kafka Adapter, que se incluye para su compilación en el paquete de Connext Gateway (instalable desde un repositorio GitHub [20], externo a la distribución por defecto de RTI). La función de este y otros adaptadores de Connext, consiste en actuar como un *Output Plugin* dentro del servicio de enrutamiento, permitiendo así la salida de datos desde un tópico DDS a un sistema externo, como puede ser en esta caso, un tópico de Kafka, sin desarrollar módulos adicionales de código. De forma simplificada e iterativa, el proceso se realiza en las siguientes fases:

1. Un *DataReader* en el Routing Service se suscribe a un tópico de DDS definido en un dominio concreto.
2. Cada muestra recibida dentro del tópico es también capturada por el *DataReader* para su posterior procesamiento conforme se haya definido en la configuración XML.
3. El adaptador de Kafka transforma la muestra para poder ser compatible con un mensaje propio de Kafka (normalmente JSON).
4. Una vez hecha la transformación, el mensaje se publica en el *broker* de Kafka correspondiente al tópico especificado en el XML.

Gracias a esta integración, ya es posible explotar las fortalezas de DDS junto con las de Kafka, sin la necesidad de ningún diseño o programación de lógica adicionales que permita la interoperabilidad entre ambos *middleware*.

En la figura 5 se proporciona una referencia visual de las entidades mencionadas que actúan dentro del Routing Service, junto con el flujo de la integración, con el fin de facilitar su entendimiento. Cabe destacar que el flujo descrito también puede configurarse de forma inversa desde Kafka hacia DDS.

2.3.4. Ventajas de utilizar Routing Service

El uso de Routing Service, ya sea con Kafka u otro tipo de tecnologías externas, aporta beneficios que son múltiples en los entornos distribuidos y heterogéneos:

- **Desacoplamiento total**. Las aplicaciones DDS y las de la tecnología de destino (o viceversa, en función de la dirección del flujo), pueden operar de forma independiente sin que sea necesario conocer detalles sobre la implementación o la arquitectura de los demás sistemas.

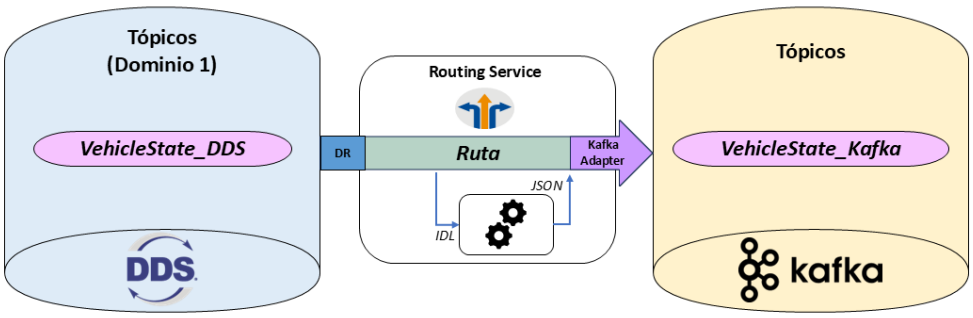


Figura 5: Representación de un flujo de mensajes a través del tópico DDS al tópico Kafka.

- **Flexibilidad de configuración**. Si bien la curva de aprendizaje no es trivial y la cantidad de opciones es amplia, el uso de esquemas XML facilita la extensión del sistema ante la posible integración de nuevos tipos de datos, tópicos, o dominios, sin requerir recompilaciones de otros sistemas en funcionamiento.
- **Monitorización**. Dado que es un componente distribuido y soportado por RTI, Routing Service también expone métricas de rendimiento, que pueden ser monitorizadas a través de los tópicos internos de DDS (en este caso, `rti/service/monitoring/periodic`), facilitando el análisis y control del puente de interconexión.
- **Mayor modularización**. Esto es especialmente importante en entornos industriales o distribuidos reales, donde las aplicaciones DDS ya pueden estar desplegadas o en funcionamiento.

2.4. Comparativa DDS vs Kafka

Una vez se han analizado por separado tanto RTI Connext DDS como Apache Kafka, resulta conveniente e interesante sintetizar toda esta información en una comparativa técnica entre ambas tecnologías, a la vez que se verifican algunas de las afirmaciones realizadas teóricamente. Si bien se ha podido comprobar que los dos *middleware* comparten un paradigma basado en la "publicación-suscripción", su funcionamiento y diseño aplican cada uno para casos de uso diferentes. El objetivo final de esta comparación es poder ofrecer una visión más clara y directa de los puntos fuertes, así como de las limitaciones de ambas plataformas, lo cual refuerza así la decisión de integrarlas en una arquitectura común. A continuación, se recoge en la tabla 1 un resumen comparativo de las principales características técnicas de DDS y Kafka. Cabe reseñar que esta comparativa no solo se apoya en la documentación oficial o literatura ya citadas, sino también en algunas pruebas prácticas realizadas, cuya implementación se encuentra en GitHub bajo el directorio *Benchmark* [14]. Principalmente, se ha evaluado tanto la latencia como el *throughput*, tomando como variable el tamaño del mensaje (*payload)*. Los resultados obtenidos se muestran a continuación, agrupando en cada caso los datos obtenidos por ambos *middleware*.

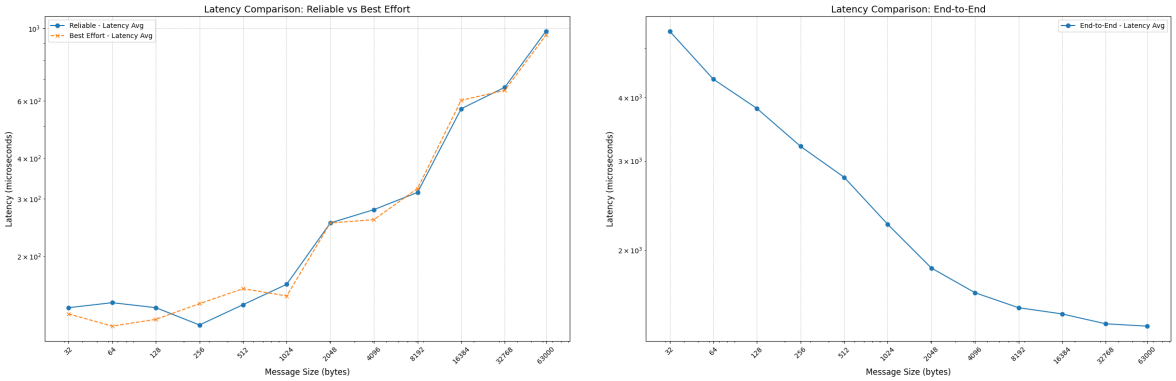
- **Latencia**. En la figura 6 se muestra la evolución de la latencia media en función del tamaño del mensaje. Puede observarse cómo DDS mantiene valores significativamente bajos, especialmente en tamaños más reducidos, en el orden de los microsegundos, lo cual confirma su adecuación a entornos con necesidades de comunicación en tiempo real.

  Por otra parte, se puede apreciar que en el caso de Kafka, estos valores son más elevados, lo cual concuerda con su arquitectura centrada en la persistencia y una cierta penalización del disco. No

	DDS	**Kafka**
Arquitectura	Descentralizada, sin *broker* (*peer to peer*)	Centralizada, basada en *brokers*
Latencia	Muy baja ($< 1ms$ con RTI Connext)	Baja-media ($> 1ms$), o mayor con persistencia (+ latencia disco)
Persistencia de datos	Opcional (En función de QoS)	Integrada
Escalabilidad	Limitada por el entorno *multicast* y red	Altamente escalable horizontalmente (*brokers*, particiones)
QoS	Extenso	Escaso (soluciones externas para configuración avanzada)
Descubrimiento	Automático, entre participantes del dominio DDS	No (configuración manual)

Tabla 1: Comparativa entre DDS y Apache Kafka.

obstante, se puede observar su estabilidad e incluso decrecimiento aunque aumente el tamaño del mensaje, demostrando que Kafka es igualmente robusto para los flujos de datos más pesados para los que está pensado.



(a) Latencia en DDS.

(b) Latencia en Kafka.

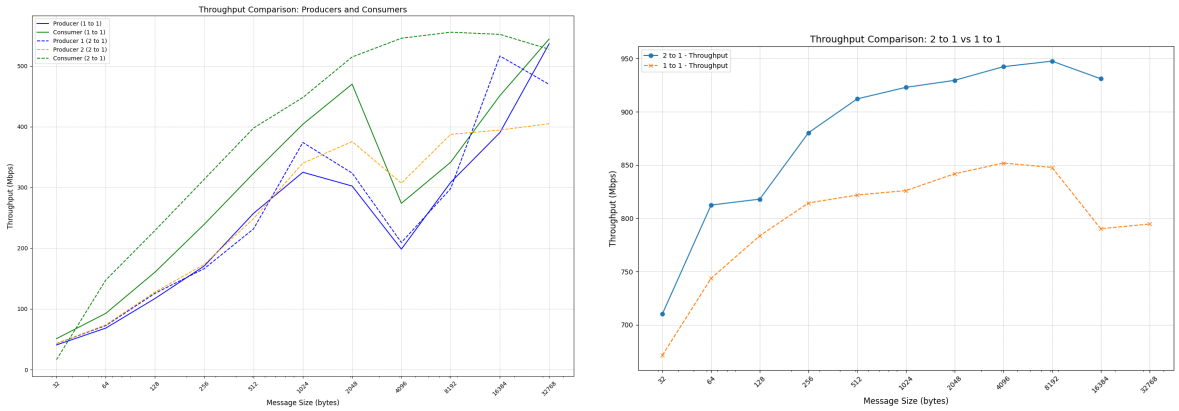
Figura 6: Comparativa de latencia media en función del tamaño del mensaje.

- ***Throughput***. La figura 7 muestra una transmisión efectiva alta para ambas alternativas, tanto en una comunicación *publisher:subscriber* de 1:1 como 2:1. Se puede comprobar que DDS, además de con una baja latencia, es capaz de aprovechar el máximo de velocidad de la interfaz de red, perfecto para entornos industriales o un contexto como el V2V.

  No obstante, sí que para un *payload* de entre 2048 y 4096 bytes se puede observar un descenso en en el número de Mbps transferidos, aunque esto puede deberse al mecanismo de fragmentación de datos detallado por RTI en su manual de librerías [18, 19]. La fragmentación es aplicada para paquetes a partir de 1400-1500 bytes aproximadamente, ya que aunque DDS use UDP como protocolo de transporte, los datagramas no pueden superar la MTU típica de Ethernet de 1500 bytes. Al propio nivel de DDS también se aplica por defecto fragmentación apenas antes de los 10000 bytes, en caso de querer garantizar la integridad de los datos [19]. Todo esto implica mayor

carga en la red y en el ensamblado y confirmación de mensajes, lo que refleja los límites también en el tamaño de los flujos de datos para DDS.

Kafka, por su parte, puede parecer menos adecuado en términos de *throughput*. Sin embargo, es un rendimiento que se corresponde con las argumentaciones teóricas al tratarse de un caso reducido con pocos equipos y un solo *broker*. Esto último, junto con una mayor optimización de Kafka de cara a un procesamiento de mayores cargas y un uso intensivo en lotes, explica los resultados obtenidos, donde por ejemplo, para una carga de 8192 bytes, no se aprecia la caída anterior. Sin embargo, sí se reduce ligeramente al pasar de los 16384 bytes de *payload*, tamaño por defecto asignado para el procesamiento por lotes [3].



(a) *Throughput* en DDS.

(b) *Throughput* en Kafka.

Figura 7: Comparativa de *throughput* en función del tamaño del mensaje.

3. Arquitectura propuesta y desarrollo del caso de uso

Después del análisis y comparativa de los *middleware* seleccionados, se procede en este capítulo a presentar la arquitectura práctica diseñada para la validación de la interconexión planteada. De esta forma, se demuestra la viabilidad técnica de una solución basada en la combinación de DDS y Kafka en un entorno distribuido.

Para ello, se ha desarrollado un caso de uso basado en la simulación de varios vehículos, cuya circulación se divide en varios sectores, publicando datos geolocalizados de forma periódica. Estos datos son posteriormente filtrados, agregados y enviados a Kafka, donde quedan disponibles para su procesamiento o visualización a cargo de otras herramientas externas. Toda esta implementación se ha realizado modularmente, permitiendo así el despliegue y monitorización del sistema en múltiples nodos, ya sean físicos o virtuales.

3.1. Diseño general de la arquitectura

La arquitectura propuesta, cuya visión global se mostró en la figura 1, se organiza en tres capas funcionales, las cuales pueden concebirse como los tres niveles del IoT-Continuum, como se puede observar en la figura 8:

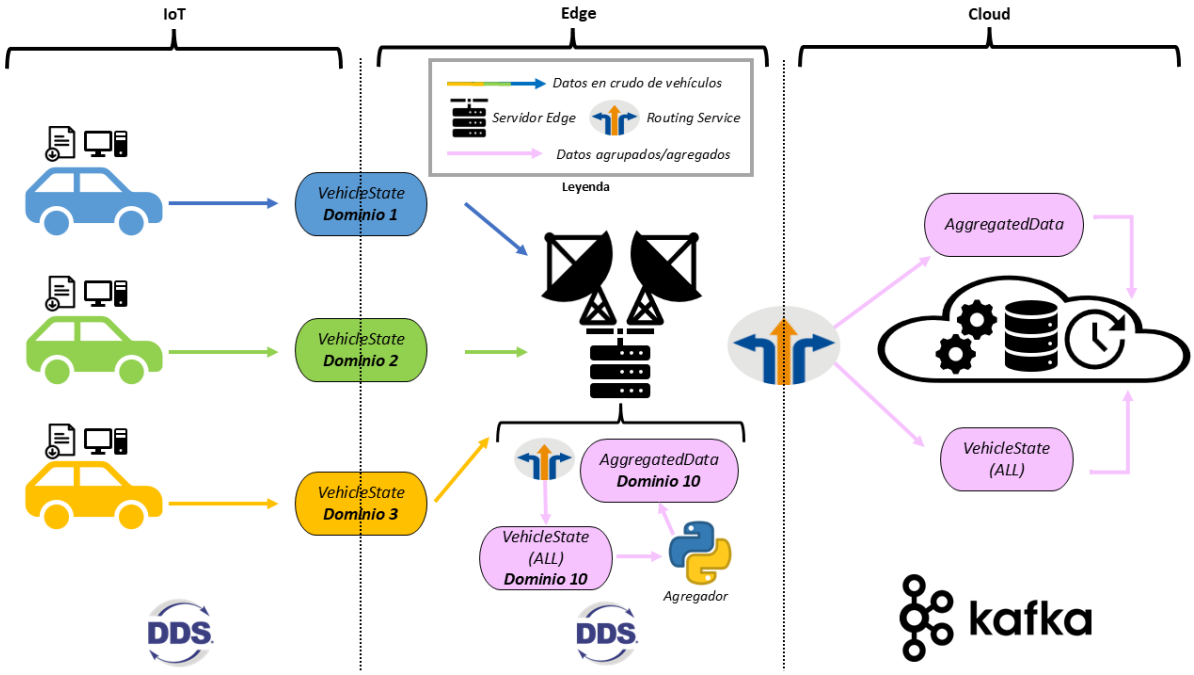


Figura 8: Arquitectura en el IoT Continuum.

- **Capa IoT**. Contiene los nodos de simulación, cada uno con la ejecución de un proceso independiente que simula el comportamiento de un vehículo inteligente. Para la simulación, se sustituye la sensórica por datos de entrada desde un fichero CSV, siendo estos publicados mediante DDS en el tópico `VehicleState` en un dominio concreto, en función del sector de la carretera en el que se encuentra cada vehículo. En estas publicaciones se encuentran datos relativos a la posición, calidad de la señal GPS, velocidad y el identificador único del vehículo.

- **Capa Edge**. En esta capa se encuentra un nodo agregador que tiene en cuenta la calidad de los datos, filtrando por la calidad de la señal GPS (HDOP, PDOP, VDOP). Se pone en práctica el primer Routing Service, reconduciendo las muestras de todos los dominios (sectores del circuito) hacia el Dominio 10 y que el agregador procesa para mostrar estadísticas de cada sector (velocidad media, número de vehículos y congestión), publicando los resultados en el tópico `AggregatedVehicleData`. Otra alternativa posible, sería emplear un agregador para cada nodo, aunque al ser este un estudio reducido, se considera suficiente un solo nodo, poniendo además en práctica la capacidad de enrutamiento entre dominios de RTI DDS.

  Un segundo Routing Service se encarga también del reenvío de los datos originales (`VehicleState`) y los agregados (`AggregateVehicleData`) hacia Apache Kafka, actuando como puente. Este componente transforma los datos de DDS a formato JSON antes de su publicación en los mismos dos tópicos correspondientes en Kafka.

- **Capa Cloud**. Si bien esta capa queda fuera del alcance de este TFM, todo el proceso realizado en las capas inferiores, deja disponibles los datos para su posterior almacenamiento o análisis por otros consumidores, como pueden ser tareas de predicción. En este último caso, los mismos resultados podrían ser reconducidos con otro enrutamiento desde Kafka a DDS, utilizando el mismo servicio que ofrece RTI, aunque variando su configuración, con una nueva transformación de datos y la alternación de *inputs* y *outputs*.

Además, cabe recordar que toda la arquitectura se encuentra acompañada de un sistema de monitorización, como se mostró previamente en la figura 1. Desde DDS, se recopilan métricas a través de su tópico interno `rti/service/monitoring/periodic`, mientras que en Kafka es necesario emplear un agente JMX Exporter. Esto permite también el análisis en tiempo real del comportamiento de todo el sistema, incluyendo el uso de recursos, latencias o la actividad en los flujos de datos configurados.

3.2. Preparación del conjunto de datos

El punto de partida para la construcción del caso de uso es el escenario 36 de DeepSense 6G, descrito en la sección 1.3. Aunque cuente originalmente con solo dos vehículos, el conjunto de datos ofrece muestras realistas con información relativa a su posicionamiento y calidad de señal GPS, lo que lo hace especialmente adecuado para una simulación de comunicaciones entre vehículos (V2V).

Sin embargo, para poder representar un entorno hipotético distribuido con múltiples vehículos, ha sido necesario realizar una transformación del *dataset* original, generando nuevas trayectorias a partir de los datos de uno de los vehículos. Estas transformaciones se aplican mediante el *script* `process_dataset.py` [14] y se describen resumidamente a continuación.

1. **En primer lugar, se simulan cinco vehículos a partir de uno**. Se introduce un desfase temporal entre las trayectorias, aplicando además pequeñas variaciones o ruido en las coordenadas GPS, lo cual permite simular movimientos similares a los de un *platooning* (tren de camiones que circulan de forma coordinada) [12, 8], aunque sin ser completamente idénticos entre sí.

```
df["gps_latitude"] += i * LAT_OFFSET + np.random.uniform(-LAT_NOISE, LAT_NOISE)
df["gps_longitude"] += i * LON_OFFSET + np.random.uniform(-LON_NOISE, LON_NOISE)
df["timestamp"] = df["timestamp"].apply(lambda ts: shift_time(ts, i))
def shift_time(timestamp, i):
    h, m, s_us = timestamp.split('-') # Original (12-34-56.123456)
    s, us = map(float, s_us.split('.'))
    total = int(h) * 3600 + int(m) * 60 + s + us / 1e6
    return total + i * TIME_OFFSET
```

2. **A continuación, se asignan identificadores unívocos para cada vehículo**. Útil para su uso como clave a la hora de definir los tipos de datos en DDS, diferenciando así los flujos que se publican sobre el estado de los vehículos.

3. **Posteriormente, se agrega un campo de sectorización**. Se determina en función de la posición del vehículo, concretamente su latitud. Esto permite asignar un dominio cuyo ID es el mismo que el del sector, en el cual se publicarán los datos de estado de los vehículos que se encuentren en dicha región geográfica.

```
# Calcular los sectores a simular
min_lat = df_0["gps_latitude"].min()
max_lat = df_0["gps_latitude"].max()
sector_bounds = np.linspace(min_lat, max_lat, NUM_SECTORS + 1) # Limites totales
...
df["sector_id"] = df["gps_latitude"].apply(lambda lat: calculate_sector(lat,
    ↪sector_bounds))
...
def calculate_sector(lat, sector_bounds):
    for i in range(NUM_SECTORS):
        if sector_bounds[i] <= lat < sector_bounds[i + 1]:
           return i + 1
    return NUM_SECTORS
```

4. **Después, se introduce variabilidad en los valores HDOP, PDOP y VDOP**. Para ello, se simulan diferentes condiciones de calidad en el muestreado de la posición GPS a lo largo del tiempo, lo cual puede ser útil a la hora de realizar la agregación de los datos garantizando la calidad de los mismos, por ejemplo, descartando aquellos con alguno de estos valores por encima de cinco, lo cual es posible que ocurra en un entorno real.

```
df["gps_hdop"] += np.random.uniform(0.0, 0.2, size=len(df))
df["gps_pdop"] += np.random.uniform(0.0, 0.2, size=len(df))
df["gps_vdop"] += np.random.uniform(0.0, 0.2, size=len(df))
```

5. **Finalmente, se genera un fichero CSV para cada vehículo**. Así, se facilita el despliegue distribuido y el simulador del vehículo correspondiente solo tiene que leer los datos de él mismo. El *script* toma como entrada el CSV original de DeepSense y genera el conjunto de ficheros para cada vehículo bajo la carpeta `Dataset/vehicles_csv/`, disponible también en el repositorio GitHub [14]. Una muestra reducida puede encontrarse en el Anexo A.

Además, en el procesado también se incluyen otras funciones como el cálculo de velocidad en base a la librería *geodesic* y las coordenadas de cada vehículo, la invalidación de algunas muestras asignando altos valores de HDOP, PDOP o VDOP, así como también la garantía de que cada muestra se genere a una frecuencia de 10 Hz, ya que en el *dataset* original existen algunos cortes.

En paralelo, también se desarrolló un *script* adicional (`validate.py`), con el fin de verificar:

- Que los vehículos circulan por varios sectores y cambian, por lo tanto, de dominio DDS.
- Que la frecuencia de las muestras es constante a 10 Hz y, en caso contrario, se han rellenado los espacios con muestras vacías (a excepción del *timestamp*), cuyo tratamiento de calidad pueda delegarse posteriormente en la simulación.
- Que existen comunicaciones de proximidad entre los vehículos, para lo cual se crean "ventanas de proximidad" en base a sus coordenadas con posiciones promedio.

3.3. Simulación distribuida de vehículos

Para la simulación de los múltiples vehículos por sectores se ha desarrollado un *script* principal denominado `simulator.py`, el cual se ejecuta en cada nodo asignado a un vehículo. Su función consiste en simular el recorrido de un vehículo utilizando su correspondiente fichero CSV preprocesado con anterioridad y publicando las muestras con una frecuencia de 10 Hz.

A continuación, se describen las operaciones del programa:

1. **Carga de datos**. Se realiza mediante la lectura del fichero CSV.
2. **Creación de las entidades DDS**. Cada vehículo consta de un *DomainParticipant* creado para el dominio asignado, que se actualiza cuando el vehículo cambia de sector y, a partir de este, se generan un *DataWriter* para la publicación de muestras y un *DataReader* con tópico filtrado para la detección de vehículos cercanos.

```
def create_entities(domain, filter_parameters):
    participant = dds.DomainParticipant(domain)
    topic = dds.Topic(participant, STATE_TOPIC, VehicleSimulation.VehicleState)
    writer = dds.DataWriter(participant.implicit_publisher, topic)
    filter_expression = "gps_latitude > %0 AND gps_latitude < %1 AND gps_longitude >
        ↪%2\
           AND gps_longitude < %3"
    ftr = dds.Filter(filter_expression, filter_parameters)
    filtered_topic = dds.ContentFilteredTopic(topic, "Filtered_Vehicle_State", ftr)
    reader = dds.DataReader(participant.implicit_subscriber, filtered_topic)
    return participant, writer, reader, filtered_topic
```

3. **Creación de una ventana de proximidad geográfica**. Permite establecer un área de proximidad en torno a los vehículos. Está centrada en la posición actual de cada uno y tiene un margen configurable mediante la constante `DELTA_FILTER`. De esta forma, se definen los límites a evaluar para leer datos del `ContentFilteredTopic`, que permite recibir muestras únicamente dentro de una determinada proximidad, pudiendo encontrarse la expresión utilizada en el anterior bloque de código, resaltada en color azul.

```
def create_window(lat, lon):
    return [
           str(lat - DELTA_FILTER),
           str(lat + DELTA_FILTER),
           str(lon - DELTA_FILTER),
           str(lon + DELTA_FILTER)
    ]
```

Durante la simulación, cada fila del CSV es publicada siguiendo el desfase temporal asignado en el *dataset* modificado. Además, cuando el vehículo cambia de sector en su trayectoria, se cierra el *DomainParticipant* en uso, creando uno nuevo en el dominio DDS que corresponda. También se actualiza la ventana de filtrado para tener en cuenta las nuevas posiciones.

```
if sector != new_sector: # Cambio de sector
    participant.close()
    participant, writer, reader, filtered_topic = create_entities(new_sector,
        ↪filter_parameters)
    sector = new_sector
```

```
    if old_parameters != filter_parameters:
            filtered_topic.filter_parameters = filter_parameters
```

Asimismo se leen en cada instante, si procede, los datos de otros vehículos próximos, utilizando el *DataReader* con el filtro geográfico asignado. Si existe un vehículo distinto dentro del rango, se emite por consola un aviso de proximidad, diferenciando además si este aviso se genera con métricas fiables en su señal GPS.

```
for data in reader.take():
    if data.info.valid and data.data.vehicle_id != vehicle_id:
        neighbour = data.data
        reliability_1 = sample.gps_pdop > 5.0 or sample.gps_hdop > 5.0 or sample.gps_vdop >
            ↪5.0
        reliability_2 = neighbour.gps_pdop > 5.0 or neighbour.gps_hdop > 5.0 or neighbour.
            ↪gps_vdop > 5.0
        if reliability_1 or reliability_2:
            print(f"POSIBLE [{vehicle_id}] CERCANO A : {data.data.vehicle_id} EN ({data.data
                ↪.gps_latitude}, {data.data.gps_longitude})")
        else:
            print(f"[{vehicle_id}] CERCANO A : {data.data.vehicle_id} EN ({data.data.
                ↪gps_latitude}, {data.data.gps_longitude})")
```

De esta forma, con la lógica de publicación y suscripción simultánea presentada, es posible simular comunicaciones V2V basadas en proximidad, lo cual puede ser un componente esencial en muchos sistemas distribuidos y, además, con cooperación entre nodos.

3.4. Agregación de datos y enrutamiento entre dominios

Para conocer el estado general de cada sector, es útil realizar una agregación de los datos de los vehículos que se encuentran en el mismo. Para ello, se centraliza la recepción de datos en un nodo dedicado a dicha agregación en el *edge*. Dado que en este caso, el número de dominios es reducido, se agrupa toda la información por agregar en un solo dominio del que consumirá el agregador. Para facilitar esto, se divide la lógica en dos componentes diferenciados: un Routing Service para redirigir las publicaciones de los dominios de cada sector a un dominio común, y un proceso programado con Python, encargado de la agregación de los datos, garantizando su calidad y resumiendo periódicamente la información que se recibe.

3.4.1. Routing Service: enrutamiento interno hacia un dominio de agregación

Este servicio actúa como reenviador de las muestras entre dominios DDS, sin que sea necesario modificar el código de las entidades que ya se encuentran publicando o consumiendo datos. Como ya se mencionó anteriormente en el apartado 2.3, es necesario definir su configuración en un fichero XML, el cual está disponible por completo en el repositorio GitHub [14].

A continuación, se resumen las principales características de dicha configuración:

- **Definición de las rutas entre dominios**. Para cada sector existente, se define una ruta que empareje cada uno de ellos con el dominio de agregación. Esto incluye:
  - La definición de sesiones. Facilitan la granularidad de la monitorización y actúan como unidad de concurrencia que agrupa los *StreamReaders* y *StreamWriters*, entidades del Routing Service para la lectura y escritura de datos en las rutas del servicio.

- Especificación de los *DomainParticipants* involucrados, asignando un nombre que sirve como identificador y el dominio al que se suscriben. Se define uno por cada dominio involucrado en la ruta, de forma que pueda contener los *readers* y *writers* correspondientes para los tópicos de dicho dominio.
- Configuración del enrutamiento del tópico *VehicleState*. Se asigna también un nombre identificador que permita la posterior monitorización de la entidad. Incluye la definición de un *input* y un *output*, los cuales se conforman por: un *DomainParticipant*, un tópico (en este caso el mismo), las cualidades de QoS y el tipo de dato utilizado.

Con el fin de facilitar la comprensión de esta estructura, que puede no ser trivial en un inicio, se incluye una sección reducida de la configuración en el Anexo B.

- **Habilitación de la monitorización**. Para habilitar la publicación de datos de rendimiento en los tópicos internos utilizados por RTI DDS, es necesario realizar una configuración adicional, especificando el dominio en el que se publica la información, los periodos de muestreo y los de publicación.

Gracias a esto, el proceso encargado de la agregación no necesita una suscripción manual a múltiples dominios ni mantener múltiples participantes activos.

3.4.2. Agregador

La lógica correspondiente a la agregación de datos se implementa en el *script* `aggregator.py`. La funcionalidad completa puede dividirse en los siguientes puntos:

- **Dominio único de operación**. El agregador se conecta únicamente al dominio DDS encargado de la agregación, concretamente, el número 10 en este caso. Ahí se escuchan las publicaciones del tópico `VehicleState`, que han sido reenviadas desde el resto de sectores previamente.
- **Filtrado por calidad de la señal GPS**. Las muestras recibidas se validan mediante un umbral ajustable por la constante `DOP_LIMIT_FILTER`, el cual en esta ocasión ha sido fijado en 5, tomando como inspiración un estudio externo sobre la integridad de métricas GPS en base a los valores HDOP, PDOP y VDOP [9]. Aquellas métricas con alguno de estos valores por encima de este umbral, son descartadas de cara al cálculo de las métricas de agregación.
- **Agrupación por sectores en base a intervalos**. Cada muestra válida es almacenada temporalmente en una lista dentro de un diccionario, cuya clave está determinada por el ID del sector, que también figura entre los datos enviados en cada muestra. Esto permite acumular datos de cada sector durante un cierto periodo, que en este caso se establece en 5 segundos (`AGGREGATION_PERIOD`).
- **Cálculo de métricas agregadas**. Transcurrido el periodo definido de agregación, se calcula para cada sector existente:
  - La velocidad media de los vehículos y, en consecuencia, del sector.
  - El número total de vehículos en circulación.
  - Un índice de congestión, el cual simula el nivel de concentración de vehículos en el sector. Dado que el establecimiento o estudio de fórmulas para este fin queda fuera del alcance del TFM, se simula el índice realizando un cálculo sencillo en el que el número de vehículos es dividido por diez.
- **Publicación de métricas agregadas**. Los resultados se publican dentro del tópico *AggregatedVehicleData*, usando un nuevo tipo de dato específico para ello, también en el mismo dominio. Esto permite que los datos en bruto queden separados de aquellos a los que se les ha aplicado un filtro de calidad y han sido agregados.

- **Reinicio de almacenamiento temporal**. Después de cada escritura de agregaciones, se vacía el diccionario con las muestras temporales para el sector, de forma que pueda volver a comenzar todo el ciclo de nuevo.

Este nuevo tópico permite integrar la información que pueda considerarse de interés para análisis posteriores o su reenvío a otros sistemas de distribución hacia la nube, como Kafka.

3.5. Integración entre DDS y Kafka

El siguiente paso consiste en enviar los datos de ambos tópicos presentes en el dominio 10 a Kafka, de forma que posteriormente puedan utilizarse para su persistencia, análisis, visualización u otro tipo de procesamiento. Para ello, se utiliza de nuevo un Routing Service, aunque enfocado a un enrutamiento externo, en lugar de interno entre tópicos, aprovechando la existencia de un adaptador para Kafka.

Esta integración cuenta con diversos desafíos, debido a las diferencias en los modelos de datos, técnicas de persistencia y acoplamiento presentes entre tecnologías. Todo esto puede resolverse de forma transparente, aplicando el Kafka Adapter de RTI en la configuración XML del Routing Service. Las partes clave de esta configuración son las siguientes:

- **Suscripción al dominio de agrupación**. El servicio actúa como suscriptor dentro del dominio 10, en el que se encuentran los tópicos `VehicleState` y `AggregatedVehicleData`, con los datos brutos y agregados respectivamente.
- **Traducción de tipos a formato JSON**. Mediante el adaptador de Kafka, se traducen los mensajes DDS automáticamente a JSON antes de su publicación en Kafka, sin requerir la programación de código adicional, más allá de la establecida en la configuración XML del Routing Service.
- **Configuración de rutas**. Se sigue una estructura similar a la ya expuesta en el anterior servicio, declarando los participantes involucrados, el dominio del que se obtienen los datos, una sesión para los recursos y los tópicos implicados. Finalmente, se enruta cada tópico DDS (*input*) a otro Kafka (*output*):
  - *vehicle_state*. Como indica el nombre, recibe las muestras originales correspondientes al tipo y tópico *VehicleState*.
  - *aggregated_data*. De igual forma, recibe los datos agrupados en el tópico DDS *AggregatedVehicleData*, generados previamente por el *script* de agregación.

  Por último, también es necesario especificar en la configuración la publicación de métricas de monitorización. Esta puede ser consultada en el repositorio del TFM [14].

3.6. Sistema de monitorización de los *middleware*

Por último, con la finalidad de poder supervisar el comportamiento del sistema en tiempo real y facilitar la evaluación de su rendimiento, se ha incorporado un sistema de monitorización con las tecnologías Prometheus y Grafana. Para la primera fase de la recolección de métricas se han definido dos procesos de exportación:

- **Exportación desde DDS**: utilizando Monitoring Library 2.0, combinada con un *script* personalizado de Python que exponga las métricas a un *endpoint scrapeable* por Prometheus.
- **Exportación desde Kafka**: utilizando JMX Exporter, herramienta ampliamente empleada en entornos que utilizan Java como Kafka.

3.6.1. Monitorización de DDS - Routing Service

Gracias a la configuración realizada previamente para los servicios de enrutamiento, es posible utilizar el mecanismo de publicación periódico de métricas ofrecido de forma nativa por RTI. Por una parte, el tópico interno `rti/service/monitoring/periodic`, contiene métricas de rendimiento asociadas a los diferentes recursos o entidades del servicio en cuestión (en este caso, Routing Service). Estos recursos se identifican mediante un GUID, lo que puede dificultar su identificación directa. Para ello, se utiliza también el tópico `rti/service/monitoring/config`, donde aparecen tanto el GUID como el nombre del recurso, pudiendo mapear ambos para su exportación.

Sin embargo, en DDS no existe una conexión directa con Prometheus, al contrario que con el JMX Exporter existente para Kafka. A pesar de que RTI está trabajando para ofrecer esta funcionalidad, aún no está disponible de forma nativa, como Monitoring Library 2.0, por lo que debe ser programado por separado. Aunque existen algunas alternativas basadas en contenedores, su uso aún no está suficientemente probado y RTI sigue explorando un despliegue independiente y hacia más *backends*.

Su funcionamiento puede resumirse en los siguientes pasos:

- **Suscripción a los tópicos de monitorización de DDS**. La suscripción a los dos tópicos previamente mencionados se realiza en dos dominios diferentes. Esto viene determinado por la configuración de la monitorización hecha previamente en los ficheros XML del Routing Service. Por una parte, las métricas relativas al enrutamiento interno entre dominios, son publicadas en el dominio 11, mientras que aquellas relativas al enrutamiento externo a Kafka, se publican en el dominio 12. Esto se hace así con el objeto de facilitar el *debugging* y aislar las métricas de cada servicio, aunque su publicación puede realizarse también en cualquiera de los dominios existentes, ya que todos pueden albergar los tópicos internos de monitorización de RTI.

  Cabe recalcar que esta fase es especialmente delicada, ya que al igual que el resto de datos intercambiados en DDS, las métricas de monitorización requieren del uso de tipos de datos concretos. Al ser un sistema nativo de RTI, se proporcionan los ficheros IDL para la generación de los tipos correspondientes en Python, si bien esto ha supuesto algunas dificultades por incompatibilidades detectadas en versiones previas (necesitando actualizar de la versión 7.3.0 a la 7.5.0). También algunos de los manuales disponibles se encuentran con los nombres de algunas de las métricas desactualizados (por ejemplo, estos últimos identifican las métricas salientes como "output_*", mientras que en la implementación real solo aparecen como "out_*"). Esto ha supuesto la necesidad de invertir también tiempo de desarrollo en el conocimiento en profundidad de estos tipos de datos y otras librerías de RTI, con ajustes manuales para asegurar la correcta suscripción y deserialización de las muestras desde Python.

- **Lectura de métricas por entidades**. Las muestras recibidas en el tópico *periodic* son estructuras que agrupan varios datos (identificadas como *union*), y que discriminan por tipo de recurso. En este caso, se han seleccionado tres de los más relevantes:

  - `ROUTING_SERVICE`. Incluye información relativa al estado general del proceso (CPU, memoria, uptime, etc.).
  - `ROUTING_DOMAIN_ROUTE`. Métricas relativas a los flujos de reenvío a nivel de dominio.
  - `ROUTING_ROUTE`. Métricas específicas de cada ruta de tópicos DDS en una sesión. Cobra más sentido cuando se usan varios tópicos por dominio, lo que permite un análisis con mayor granularidad que `ROUTING_DOMAIN_ROUTE`.

- **Exposición a través de Prometheus**. Las métricas extraídas se registran en objetos estándar de Prometheus denominados *Gauge* y distinguiendo por etiquetas los recursos del mismo tipo (`instance`, `domain_route` y `route`).

```
# DOMAIN ROUTE
domain_route_in_samples = Gauge("rti_domain_route_input_samples",
                                "Input samples/s", ["domain_route"])
domain_route_in_bytes = Gauge("rti_domain_route_input_bytes",
                              "Input bytes/s", ["domain_route"])
domain_route_out_samples = Gauge("rti_domain_route_output_samples",
                                 "Ouput samples/s", ["domain_route"])
domain_route_out_bytes = Gauge("rti_domain_route_output_bytes",
                               "Output bytes/s", ["domain_route"])
domain_route_latency = Gauge("rti_domain_route_latency",
                             "Avg. Latency (ms)", ["domain_route"])
```

Por último, se establece un puerto HTTP indicado como parámetro al *script*. De lo contrario, se establece 8000 por defecto. Este expone la información de forma continua hacia otros servicios, como Grafana. A continuación, se muestra un fragmento del código, donde se seleccionan las medias de las métricas de uno de los recursos mencionados:

```
# Cargar o actualizar metricas
for sample in reader_periodic.take():
    if not sample.info.valid:
        continue
    data = sample.data
    guid = guid_to_str(data.object_guid.value)
    name = guid_name.get(guid, guid)
    union = data.value
    kind = union.discriminator
    try:
        ...
        elif kind == Kind.ROUTING_DOMAIN_ROUTE:
            metrics = union.routing_domain_route
            domain_route_in_samples.labels(domain_route=name).set(metrics.
                ↪in_samples_per_sec.publication_period_metrics.mean)
            domain_route_in_bytes.labels(domain_route=name).set(metrics.
                ↪in_bytes_per_sec.publication_period_metrics.mean)
            domain_route_out_samples.labels(domain_route=name).set(metrics.
                ↪out_samples_per_sec.publication_period_metrics.mean)
            domain_route_out_bytes.labels(domain_route=name).set(metrics.
                ↪out_bytes_per_sec.publication_period_metrics.mean)
            domain_route_latency.labels(domain_route=name).set(metrics.latency_millisec
                ↪.publication_period_metrics.mean)
        except Exception as e:
            print(f"Error con la muestra de {kind}:{name}: {e}")

    time.sleep(3) # Actualizar cada tres segundos
```

Si bien este es un enfoque complejo, facilita la personalización en función de las métricas de interés y estableciendo los periodos de actualización que se consideren adecuados. En el repositorio GitHub [14], se encuentra una muestra completa de toda la información expuesta en el *endpoint* 8000 (enrutamiento interno) y 8001 (enrutamiento externo).

3.6.2. Monitorización de Kafka

La monitorización de Kafka emplea JMX Exporter, una herramienta que ya es ampliamente utilizada para la exposición de métricas internas de procesos JVM. De esta forma, sin necesidad de modificación del código fuente, es posible obtener información detallada del rendimiento de los *brokers*.

El proceso requiere los siguientes elementos:

- **Agente JMX**. Se añade como variable de entorno para su ejecución junto con Kafka y se indica el puerto de exposición HTTP (8002, continuando con la serie de puertos usados en DDS).

```
export KAFKA_OPTS="-javaagent:${dir_kafka}/jmx_prometheus_javaagent-1.3.0.jar=${
    ↪jmx_port}:${dir_kafka}/jmx_exporter.yml"
```

- **Listado de métricas de interés en configuración YAML**. En el fichero `jmx_exporter.yml` se definen las métricas a recolectar, optando en este trabajo por métricas clave por tópico, tales como el número de bytes o mensajes entrantes y salientes, las cuales pueden considerarse como métricas de *throughput* por tópico.

  Además, esta definición se hace para los diferentes tópicos existentes, lo que permite conocer el tamaño de los flujos para cada uno de ellos. Una sección del fichero puede encontrarse en el Anexo C.

Esta monitorización permite conocer los volúmenes de datos procesados en ambos *middleware*, entre otro tipo de métricas, pudiendo analizar la carga de los flujos existentes. Además, son métodos poco invasivos que pueden acoplarse fácilmente con posterioridad a un despliegue ya existente sin introducir una penalización significativa en latencia o rendimiento general.

4. Automatización del despliegue

Una vez detallada la arquitectura propuesta del sistema y desarrollada la implementación de los componentes involucrados en su funcionamiento, se describe a continuación el mecanismo utilizado para realizar el despliegue del entorno de una forma rápida, reproducible y coherente entre los nodos implicados en el caso de uso. El hecho de que un sistema distribuido, como el que se plantea en este trabajo, esté compuesto por diversos nodos y con funciones diferenciadas (agregación, monitorización, captura de métricas...), da lugar a una complejidad inherente que hace que el proceso de despliegue manual resulte costoso en tiempo. La existencia de ficheros de configuración variados, tipos de datos o programas dependientes de otras librerías externas, agrava aún más esta problemática.

Con este propósito, se ha diseñado un sistema modular compuesto por diferentes *scripts* en Bash, que permiten la instalación remota de servicios, la ejecución coordinada de los procesos que simulan vehículos y la preparación de entornos, incluidos los correspondientes a la agregación y monitorización. Todo esto permite, además:

- **Replicación**. Se acelera considerablemente la replicación del entorno en los dispositivos que lo componen, ya sean físicos o virtuales.
- **Garantía de coherencia**. Se mantiene una misma configuración de los nodos sin variar sus roles ante redespliegues o correcciones de errores.
- **Instalación de dependencias**. Después de su depuración manual y garantizar la correcta instalación e interconexión de los paquetes utilizados, la automatización mediante *scripts* ayuda a evitar incompatibilidades de versiones o errores de configuración.
- **Facilitar los ciclos de mejora**. El redespliegue es más rápido y sencillo. Además, permite la ejecución y detención ordenada de los nodos que simulan la actividad en el sistema.

4.1. Estructura del sistema de automatización

Los *scripts* desarrollados están diseñados y organizados de forma que se puedan ejecutar desde un nodo central, por ejemplo, el nodo *Admin* con capacidad de conexión remota por SSH al resto de máquinas. En la figura 9, se plantea de forma gráfica la asignación de roles realizada a los nodos que componen el caso de uso. En este caso, se plantea un ejemplo con máquinas virtuales e IPs fijas asignadas dentro de una red NAT compartida entre todas ellas. Cada una de estas IPs queda, por lo tanto, ligada a un rol en particular, facilitando la identificación de los nodos y sus funciones.

En cuanto a los *scripts* disponibles en GitHub [14] bajo el directorio `Deploy_Scripts`, estos pueden agruparse funcionalmente en cuatro bloques:

- **Instalación de los *middleware* y otras dependencias**. Comprenden la instalación automatizada de RTI Connext DDS, Apache Kafka, Prometheus y Grafana en sus correspondientes nodos.
- **Preparación del entorno**. Se realiza la copia y configuración de *scripts* de Python, los ficheros CSV con los conjuntos de datos modificados, así como otros recursos que puedan ser necesarios para cada nodo, en función del rol que le haya sido asignado (monitorización, agregación (*edge*), vehículo...).
- **Ejecución distribuida**. Lanza de forma ordenada los procesos asignados a los simuladores de vehículos, con el fin de que no exista un excesivo desfase, como sería en el caso de una ejecución manual. Estos procesos se mantienen en segundo plano, mientras su supervisión o la trazabilidad de errores es posible gracias a la escritura de las salidas en ficheros locales de log.

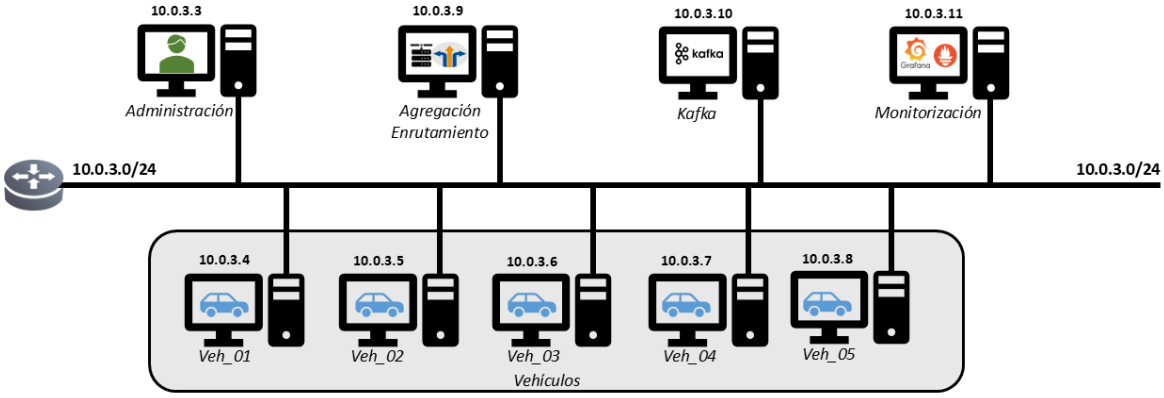


Figura 9: Red NAT de dispositivos utilizados en la simulación del caso de estudio.

- **Parada**. Finaliza todas las simulaciones remotas de vehículos para facilitar la preparación del sistema para una nueva prueba. Dado que los nodos asignados a monitorización y agregación son únicos, no se ha considerado tan relevante el desarrollo de un *script* de arranque y parada para ellos.

Además, para facilitar que los ejecutables puedan identificar los nodos asignados a determinados roles o tecnologías a instalar, se recogen sus direcciones IP en ficheros de texto, tales como: `nodes_dds.txt` y `nodes_vehicles.txt`. También se utiliza un fichero con variables comunes a varios ejecutables (`common.sh`). De esta forma, es posible automatizar la repetición de acciones para diferentes nodos, con bucles como el mostrado en el siguiente fragmento:

```
1 while IFS= read -r node <&3; do
2     install_dds "$node"
3 done < nodes_dds.txt
```

En la figura 10 se esquematiza la agrupación de los *scripts* implementados, organizándolos según los bloques funcionales descritos anteriormente. Todo este diseño modular no solo facilita la mantenibilidad y actualización, sino también la posibilidad de ampliar el sistema con nuevos componentes sin afectar al trabajo realizado para los ya presentes en el sistema.

4.2. Instalación automatizada de paquetes por nodo

El primer grupo de *scripts* se encarga de la instalación de los paquetes necesarios en cada nodo, en función del rol correspondiente. Para ello, se han programado tres ficheros:

- ***install_dds.sh***. Se encarga de la descarga e instalación de RTI Connext DDS 7.5.0 en los nodos que actúan como simuladores de vehículos o hipotéticos agregadores o *bridges* en el *edge*, siendo recogidas sus direcciones IP en el fichero `nodes_dds.txt`. Principalmente se utiliza la distribución ofrecida por RTI a través de *apt* tras importar las claves necesarias y actualizar el repositorio Debian:

```
1 echo $PASS | sshpass -p $PASS ssh -tt -p $PORT $USER@$node "sudo curl -sSL -o /usr/
    ↪share/keyrings/rti-official-archive.gpg https://packages.rti.com/deb/official/
    ↪repo.key"
```

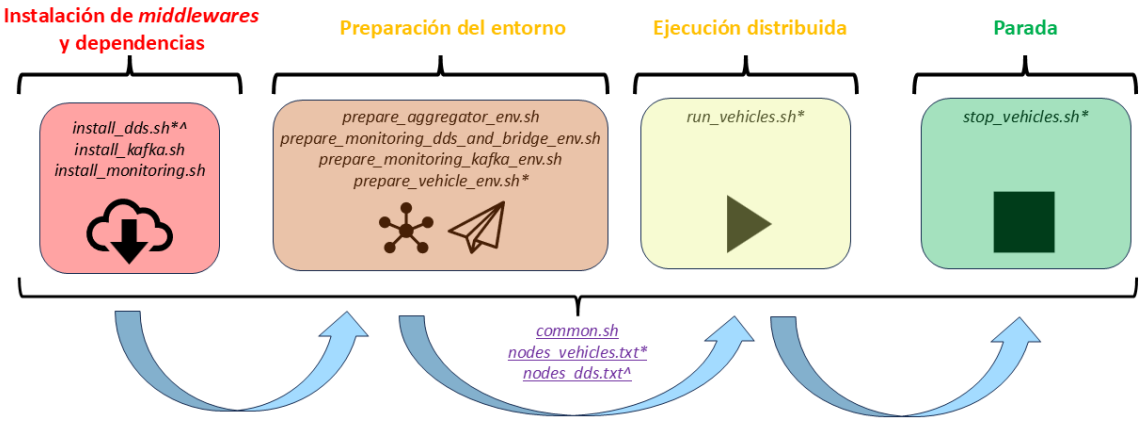


Figura 10: Agrupación de ficheros de despliegue por fases.

```
echo $PASS | sshpass -p $PASS ssh -tt -p $PORT $USER@$node "echo 'deb [arch=$(dpkg --print-architecture), signed-by=/usr/share/keyrings/rti-official-archive.gpg] https://packages.rti.com/deb/official jammy main' | sudo tee /etc/apt/sources.list.d/rti.list > /dev/null"
echo $PASS | sshpass -p $PASS ssh -tt -p $PORT $USER@$node "sudo apt-get update"
echo $PASS | sshpass -p $PASS ssh -tt -p $PORT $USER@$node "echo 'rti-connext-dds-7.5.0-common rti-connext-dds-7.5.0/license/accepted select true' | sudo debconf-set-selections"
echo $PASS | sshpass -p $PASS ssh -tt -p $PORT $USER@$node "sudo apt-get install -y rti-connext-dds-7.5.0 curl gnupg git cmake build-essential default-jdk"
```

En el caso del *Gateway* con el adaptador de Kafka, es necesario recurrir a un repositorio GitHub externo [20], también de RTI, incluyendo la compilación además de la descarga:

```
echo $PASS | sshpass -p $PASS ssh -tt -p $PORT $USER@$node "sudo git clone --recurse-submodule https://github.com/rticommunity/rticonnextdds-gateway.git /opt/rti.com/rti_connext_dds-7.5.0/rticonnextdds-gateway"
echo $PASS | sshpass -p $PASS ssh -tt -p $PORT $USER@$node "cd /opt/rti.com/rti_connext_dds-7.5.0/rticonnextdds-gateway && sudo mkdir -p build && cd build && sudo cmake .. -DCONNEXTDDS_DIR=/opt/rti.com/rti_connext_dds-7.5.0  -DCMAKE_INSTALL_PREFIX=../install && sudo cmake --build . --target install"
```

- ***install_kafka.sh***. Realiza la descarga y desempaquetado de Kafka desde la propia fuente oficial de Apache, ya que no se encuentra disponible como paquete Debian a través de *apt* o como fichero *.deb*:

```
sshpass -p $PASS ssh -tt -p $PORT $USER@$node "wget https://dlcdn.apache.org/kafka/3.9.0/kafka_2.13-3.9.0.tgz -O /tmp/kafka.tgz"
echo $PASS | sshpass -p $PASS ssh -tt -p $PORT $USER@$node "sudo tar -xzf /tmp/kafka.tgz -C /opt/kafka --strip-components=1"
```

- ***install_monitoring.sh***. Se ejecuta sobre el nodo encargado de la monitorización que alberga a Prometheus y Grafana. En el caso de Grafana, es posible instalarlo a través de *apt*, previo registro de claves y actualización del repositorio. Por otra parte, Prometheus requiere una descarga desde su repositorio oficial de GitHub, siguiendo por lo tanto ambas opciones procesos similares a los ya expuestos de ejemplo previamente.

Como se puede observar en los fragmentos de código incluidos, junto con SSH también se emplea *sshpass*, que permite automatizar en mayor medida la autenticación sin requerir la intervención del usuario o administrador. Por lo general, es más recomendable el uso de claves SSH cuando se utiliza un entorno de producción o con mayor necesidad de seguridad. Sin embargo, al desarrollarse el trabajo en un entorno cerrado se ha priorizado la simplicidad de la conexión para acelerar las pruebas.

Por último, cabe destacar que los ficheros incluyen también otras tareas menores, como la definición de variables de entorno o la creación de directorios, pudiendo ser estos consultados por completo en el repositorio GitHub correspondiente a este TFM [14]. Además, todos estos ejecutables son independientes entre sí, por lo que se pueden ejecutar para reinstalar únicamente los componentes necesarios en los subconjuntos de nodos que se estimen oportunos.

4.3. Preparación de entornos

Una vez se realiza la instalación de los servicios necesarios, el siguiente paso consiste en la preparación del entorno de ejecución de cada nodo. Esta fase, implica el copiado de archivos necesarios desde el nodo de control hacia cada nodo del sistema en función de su rol, así como la creación de entornos virtuales de Python y sus dependencias de ejecución.

Con el fin de facilitar esta tarea, se han elaborado los siguientes *scripts*:

- **`prepare_vehicle_env.sh`**. Recorre los nodos cuya IP se ha recogido en el fichero `nodes_vehicles.txt`, enviando a cada uno una copia del *script* encargado de la simulación del vehículo (`simulator.py`), el fichero generado a partir de IDL con los tipos DDS utilizados, así como un fichero CSV personalizado para cada nodo, con el muestreo simulado a publicar de forma periódica por cada vehículo. La asignación de identificadores se realiza de forma secuencial, en función de la orden de aparición de su dirección IP en el listado.

  De esta forma, cada CSV queda nombrado como `veh_XX.csv`, siendo `XX` un identificador de dos dígitos. A continuación, se adjunta una sección de código que ejemplifica la copia de ficheros, la creación del entorno virtual de Python y la instalación de los paquetes necesarios para la ejecución en este último (como la API de RTI DDS para Python):

```
# Copiar ficheros necesarios
sshpass -p $PASS scp -P $PORT ../Dataset/vehicles_csv/$csv $USER@$node:$dest/ # CSV
sshpass -p $PASS scp -P $PORT ../Simulation/VehicleSimulation/vehicle.py $USER@$node:
    ↪$dest/ # Script Simulacion
sshpass -p $PASS scp -P $PORT ../Simulation/DataTypes/python/vehicle_data.py
    ↪$USER@$node:$dest/ # Tipo de datos
# Crear entorno vistual
sshpass -p $PASS ssh -p $PORT $USER@$node "python3 -m venv $dest/venv"
# Instalar dependencias
sshpass -p $PASS ssh -p $PORT $USER@$node "source $dest/venv/bin/activate && \
pip install --upgrade pip && pip install rti.connextdds"
```

- **`prepare_aggregator_env.sh`**. Despliega en el nodo agregador de datos el `script aggregator.py`, junto con los tipos de datos DDS y el fichero XML asociado a la configuración del Routing Service que interconecta los dominios DDS.
- **`prepare_monitoring_dds_and_bridge_env.sh`**. Prepara el entorno encargado de la monitorización de DDS y el enrutamiento con Kafka. Para ello, copia también los ficheros con los tipos de datos necesarios y el XML que configura el Routing Service.
- **`prepare_monitoring_kafka_env.sh`**. Copia los ficheros de configuración del JMX Exporter en el nodo del *broker* Kafka, así como el JAR ejecutado para su puesta en funcionamiento.

Al igual que en la fase de instalación, también se declaran variables de entorno, encontrándose de igual forma todos estos ficheros disponibles en el repositorio GitHub. Debido a la finalidad de este grupo, se hace un uso intensivo del comando *scp* para la transferencia de archivos. Por último, cabe reseñar que esta modularización diseñada permite ajustar configuraciones o copiar nuevos ficheros sin afectar a las funciones de instalación elaboradas con anterioridad.

4.4. Arranque y parada de simulación de vehículos

Una vez preparados los entornos de los nodos de la simulación, el sistema se encuentra listo para iniciar los procesos de publicación correspondientes a cada vehículo, previo arranque de los sistemas de monitorización y Kafka.

El *script* `run_vehicles.sh` lanza de forma automatizada los simuladores cuya IP se encuentra en el archivo `nodes_vehicles.txt` . Para ello, se itera sobre cada nodo registrado, asociando un identificador al vehículo (`veh_XX.csv`) en función del origen. Posteriormente, se establece una conexión SSH con el nodo, ejecutando el *script* `simulator.py`, pasando el fichero CSV con los datos que simulan los vehículos como argumento. Además, el proceso se lanza en segundo plano y se redirige la salida a un fichero de log, para facilitar el *debugg* y la comprobación del correcto funcionamiento del sistema.

De esta forma se facilita la ejecución simultánea desde el nodo de control, lo que facilita la sincronización entre los vehículos durante las pruebas. De lo contrario, podría dificultarse la existencia de oportunidades donde algunos vehículos estén próximos entre sí.

De forma análoga, `stop_vehicles.sh` detiene las ejecuciones buscando aquellos procesos que contienen el patrón del nombre del *script* de simulación y ejecutando un comando `kill` sobre su identificador de proceso (PID). Así, se garantiza que los procesos no dejen residuos en memoria y se permite un reinicio de la ejecución en caso de ser necesario. Finalmente, no se incluye la limpieza de los logs en esta ejecución, ya que pueden ser útiles para diagnóstico y trazabilidad de errores durante el desarrollo.

5. Visualización y validación del sistema mediante Grafana

Por último, en este capítulo se aborda la comprobación del comportamiento del sistema en tiempo real y su capacidad para ser monitorizado de forma efectiva. Para esto, se emplea la herramienta Grafana, que tomando como fuente el servidor Prometheus, ya conectado a los *endpoints* mencionados en los apartados 3.6.1 y 3.6.2, recoge métricas tanto de DDS (Routing Service) y Kafka.

El objetivo principal de esta sección no es solo la validación de la entrega de datos o la existencia de flujos, sino también demostrar que los mecanismos de instrumentación y monitorización pueden ser correctamente integrados, permitiendo la supervisión de los componentes en entornos replicables o reales.

A continuación, se presentan los *dashboards* configurados, con una breve descripción de cada uno.

5.1. Monitorización de recursos en DDS

La figura 11 muestra la monitorización de los recursos del sistema encargado de la ejecución de las instancias del Routing Service. Las instancias se corresponden con *Internal Routing* (enrutamiento entre dominios DDS) y *External Routing* (enrutamiento de DDS a Kafka).

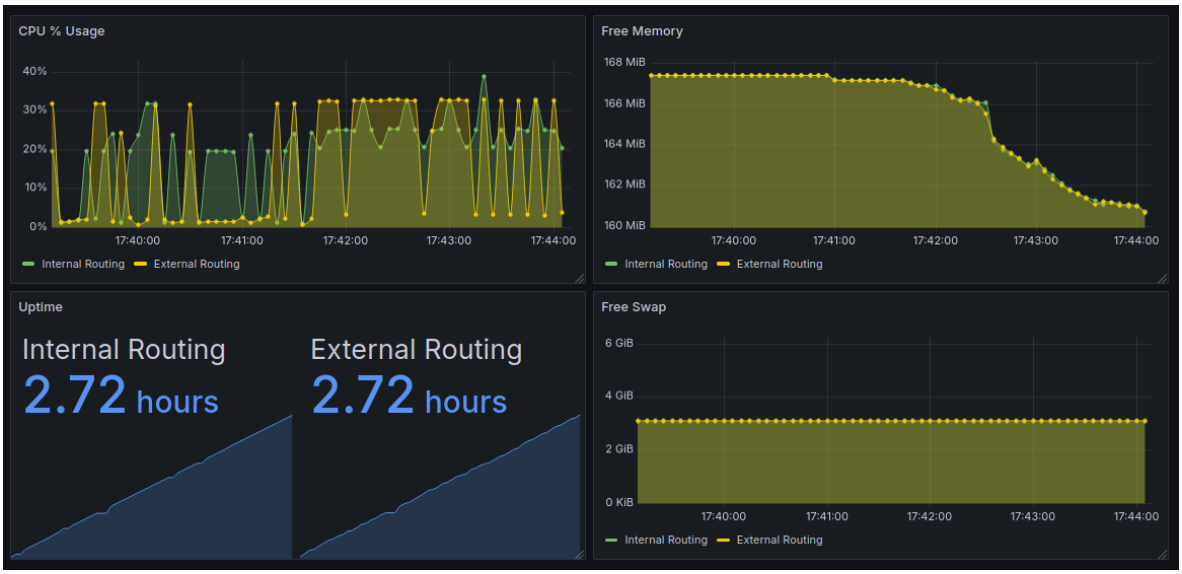


Figura 11: Uso de CPU, memoria y estado de ejecución de las instancias de Routing Service.

El panel superior izquierdo muestra el uso de CPU en porcentaje, mientras que los dos paneles derechos reflejan la memoria libre disponible (incluyendo la SWAP). Se puede observar que la RAM se reduce ante el aumento de tráfico en DDS, que coincide también con el mayor consumo en CPU. Por otra parte, en la sección inferior izquierda se muestra el *uptime*, donde las instancias se han mantenido en ejecución sin interrupciones ni reinicios. En general, estas métricas permiten validar la estabilidad de los procesos de enrutamiento del sistema y la existencia de posibles saturaciones.

5.2. Monitorización del *throughput* y latencia entre dominios DDS

La figura 12 muestra el *dashboard* encargado de reflejar las métricas centradas en la actividad de las rutas entre dominios DDS, separándola en función de las tres definidas, como se observa en la leyenda: *Domain 1 to 10*, *Domain 2 to 10* y *Domain 3 to 10*. Son apreciables los traspasos de un

dominio a otro progresivamente, con la consecuente transferencia de mensajes o bytes por segundo, cuando los vehículos cambian de sector. Además, puede observarse la coherencia de las mediciones, ya que utilizando cinco vehículos con una frecuencia de muestreo de 10 Hz, esto debe dar un total aproximado de 50 por segundo, tal y como se aprecia en la figura.

Figura 12: Tasa de mensajes y bytes por segundo entre dominios DDS.

Además, de forma complementaria a la información previa, también se muestra en la figura 13 una pareja de paneles encargados de monitorizar la latencia media para cada Domain Route. Se puede observar que los valores son muy reducidos ($<$ 1ms), si bien es necesario tener en cuenta que estas capturas se han tomado en máquinas virtuales, por lo que la red NAT es simulada realmente. No obstante, como ya se mostró previamente en el *benchmark* del capítulo 2.4, las latencias reales no suelen tener valores significativamente más altos.

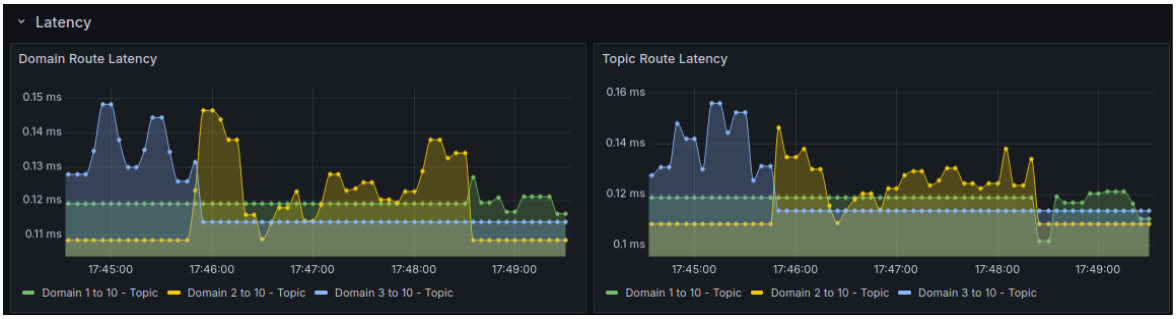


Figura 13: Latencia media en las rutas entre dominios DDS.

Gracias a la configuración que se realizó en la captura de métricas de DDS y, como se puede observar ya con las latencias, es posible obtener las mismas métricas de *throughput* a nivel de cada ruta que conecta los dominios, en caso de que existieran varias rutas (por ejemplo, para distintos tópicos) entre dos dominios. En este caso, como se observa en la figura 14, las gráficas son similares, ya que para cada pareja de dominios solo existe una ruta, por lo que aunque la granularidad sea mayor, sigue siendo el mismo caudal de datos.

Figura 14: Tasa de mensajes y bytes por segundo entre dominios DDS a nivel de ruta.

5.3. Monitorización del *throughput* y latencia hacia Kafka

Además de validar que la conexión entre los dominios DDS funciona adecuadamente, es relevante también verificar que el puente que conecta DDS con Kafka funciona adecuadamente y el volumen de datos transferido es coherente con respecto al flujo original o el resultante de la agregación. Para ello, se crean nuevos *dashboards* con varios paneles similares a los propuestos anteriormente, observando tanto la actividad propia del Routing Service como del *broker* de Kafka.

En las figuras 15 y 16, pueden observarse de forma análoga a las figuras 12 y 14 los datos de *throughput* del Routing Service a nivel de dominio y a nivel de ruta respectivamente. En este caso solo

Figura 15: Tasa de mensajes y bytes por segundo entre DDS y Kafka a nivel de dominio.

existe información del dominio 10, ya que es el que se toma como fuente de datos para conectar con Kafka. Puede observarse que mantiene el rendimiento de 50 mensajes por segundo, por lo que se están enviando todos los datos disponibles. En algunas ocasiones la media es incluso ligeramente superior, lo cuál se explica también por el enrutamiento de los mensajes de agregación, que en comparación, son mucho menores (del orden de 1 mensaje por dominio cada 5 segundos aproximadamente). Esto se aprecia mejor a nivel de ruta en la figura 16 donde se puede observar que la media correspondiente al tópico de agregación es mucho más baja con respecto al estado del vehículo, lo cual concuerda con la simulación descrita a lo largo de la memoria.

Figura 16: Tasa de mensajes y bytes por segundo entre DDS y Kafka a nivel de ruta.

Finalmente, también es posible comprobar que todas las métricas que proceden del Routing Service son recibidas por el *broker* de destino de Kafka, gracias a la habilitación de métricas realizada previamente, tal y como se puede consultar en la figura 17. Si bien en un primer momento puede parecer que la tasa de mensajes es menor, esto se debe al procesamiento en lotes que hace Kafka de los datos, lo cual se refleja en el alcance de mayores tasas promedio de bytes por segundo.

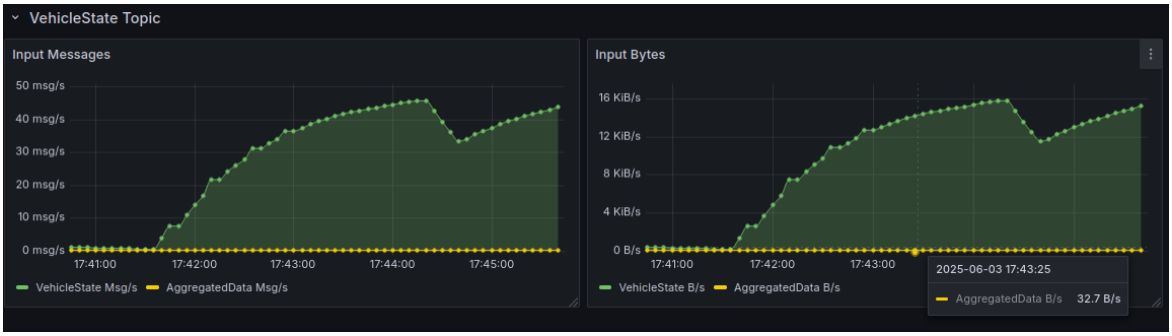


Figura 17: Tasa de mensajes y bytes por segundo recibidos en Kafka.

También se mantienen unas latencias reducidas (ver figura 18), que demuestran que el adaptador no introduce retardos significativos, si bien conviene recordar de nuevo que la simulación mostrada es

sobre una red NAT virtual.

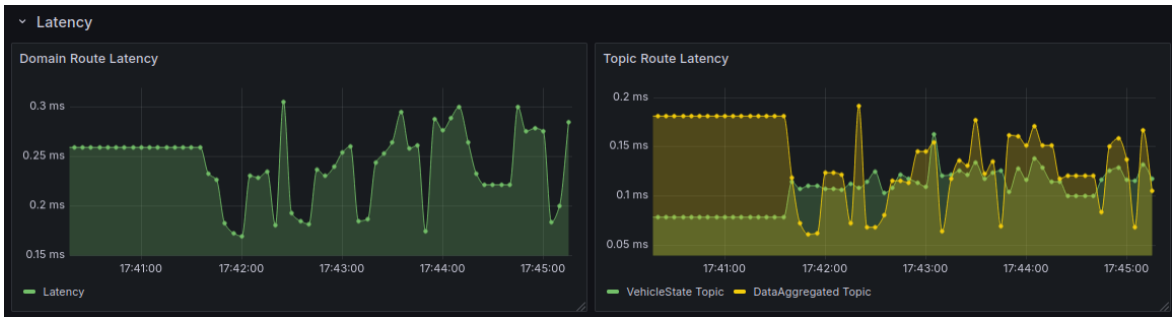


Figura 18: Latencia media en las rutas hacia Kafka.

6. Conclusiones y líneas futuras de aplicación

A lo largo de este TFM se ha diseñado, desarrollado y puesto en práctica una posible arquitectura distribuida y monitorizada con la combinación de dos *middleware*, RTI Connext DDS y Apache Kafka, la cual permite explotar las ventajas de ambos en sistemas que requieran ajustarse a criterios de tiempo real y que, al mismo tiempo, necesiten enviar u obtener información de la nube. Mediante el planteamiento de un caso de uso, se ha demostrado la capacidad para manejar múltiples fuentes distribuidas de datos (vehículos), agregarlos en el *edge* y redirigirlos de manera eficiente y estructurada a otras plataformas en la nube donde se realicen otras tareas de procesamiento.

Concretamente, entre los principales logros del proyecto, se encuentran:

- El uso de RTI Routing Service para la interconexión de dominios DDS, simplificando conexiones entre sectores y permitiendo topologías con mayor escalabilidad en los mismos.
- La agregación de datos en el *edge*, al mismo tiempo que se garantiza su calidad de forma periódica.
- La integración de dos *middleware* con enfoques diferentes, pero cooperando entre sí, mediante Kafka Adapter y RTI Routing Service. DDS dirigido a facilitar la integración de fuentes en tiempo real y Kafka a labores de persistencia o análisis de datos en la nube.
- Una implementación completa del sistema diseñado, validando la arquitectura propuesta a través del desarrollo de un método de monitorización específico.
- Desarrollo y automatización del despliegue de una simulación utilizando el conjunto de datos DeepSense 6G.

Estos resultados permiten asentar las bases de una interconexión entre dos de los *middleware* de distribución más utilizados a día de hoy, con aplicaciones prácticas, monitorización y opciones de extensión.

Como líneas de trabajo futuro se identifican las siguientes:

- **Ampliación en el alcance de la simulación**. Incorporar otros campos del *dataset* seleccionado, tales como imágenes de las cámaras o información del LiDAR, con el fin de enriquecer la simulación y evaluar la interoperabilidad con datos de mayor heterogeneidad en un entorno más real.
- **Incremento del número de vehículos y despliegue de la simulación en nodos físicos**. De esta forma, se pretende escalar el sistema y evaluar su comportamiento ante condiciones de mayores exigencias de latencia, *throughput* o descubrimiento de nodos.
- **Integración de sistemas de predicción e inferencia en el *edge***. A partir de modelos entrenados en la nube, cuyas notificaciones pueden retroalimentar a dispositivos IoT. Además, puede incorporarse aprendizaje federado [7] para enriquecer los modelos respetando la privacidad de los datos, y técnicas de *real-time machine learning* que permitan adaptaciones dinámicas en entornos con requisitos de inmediatez.
- **Avanzar hacia una publicación de los resultados obtenidos**. Una vez incorporadas nuevas pruebas de validación y finalizadas algunas de las líneas de trabajo futuro que se plantean, se pretende compartir los resultados y el sistema elaborado con el resto de la comunidad investigadora.

Por último, en el plano personal, este trabajo de fin de máster me ha permitido conocer en mayor profundidad los problemas surgidos de la falta de interoperabilidad entre sistemas distribuidos y muchas de las soluciones software existentes, lo cual ha reforzado mi interés en la investigación y consolidado mi idea de orientar mi futuro hacia la elaboración de una tesis doctoral en el campo de la interoperabilidad.

Bibliografía

[1] Apache Kafka. *Documentation.* [Último acceso: 26 mayo 2025]. Disponible en: https://kafka.apache.org/documentation/

[2] Bode, V., Buettner, D., Preclik, T., Trinitis, C. & Schulz, M. 2023. Systematic analysis of DDS implementations. Proceedings of the 24th International Middleware Conference (Middleware '23) (pp. 234–246). Disponible en: https://doi.org/10.1145/3590140.3629118

[3] Conduktor. *Kafkademy, Kafka Producer Batching.* [Último acceso: 4 junio 2025]. Disponible en: https://learn.conduktor.io/kafka/kafka-producer-batching/

[4] DDS Foundation. *DDS Portal - Data Distribution Services.* [Último acceso: 23 mayo 2025]. Disponible en: https://www.dds-foundation.org/

[5] DeepSense. *Data Collection, DeepSense Testbeds.* [Último acceso: 23 mayo 2025]. Disponible en: https://www.deepsense6g.net/data-collection/

[6] DeepSense. *Scenarios 36-39.* [Último acceso: 24 mayo 2025]. Disponible en: https://www.deepsense6g.net/scenarios36-39/

[7] H.B. McMahan, E.Moore, D.Ramage, S.Hampson & B.A. y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. Conference on Artificial Intelligence and Statistics. Disponible en: https://doi.org/10.48550/arXiv.1602.05629

[8] Hou, J., Chen, G., Huang, J., Qiao, Y., Xiong, L., Wen, F., Knoll, A. & Jiang, C. 2023. Large-scale vehicle platooning: Advances and challenges in scheduling and planning techniques. Engineering, 28, 26–48. Disponible en: https://doi.org/10.1016/j.eng.2023.01.012

[9] Isik, O. K., Hong, J., Petrunin, I. & Tsourdos, A. 2020. Integrity analysis for GPS-based navigation of UAVs in urban environment. Robotics, 9(3), 66. Disponible en: https://doi.org/10.3390/robotics9030066

[10] Kreps, J. 2014, 27 abril. Benchmarking *Apache Kafka: 2 Million writes per second (on three cheap machines).* LinkedIn Engineering. Disponible en: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

[11] Liang, W.-Y., Yuan, Y. & Lin, H.-J. 2023. A performance study on the throughput and latency of Zenoh, MQTT, Kafka, and DDS. Zenoh Taiwan Team, Zettascale Technology; Department of Computer Science and Information Technology, National Taiwan University. Disponible en: https://doi.org/10.48550/arXiv.2303.09419

[12] Lu, D., Li, Z. & Huang, D. 2017. Platooning as a service of autonomous vehicles. IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM). Disponible en: https://doi.org/10.1109/WoWMoM.2017.7974353

[13] Martín, M. 2023. *Diseño e implementación de un servicio de monitorización y aseguramiento de la calidad y seguridad de datos publicados en Kafka.* Universidad de Cantabria. Disponible en: https://hdl.handle.net/10902/30033

[14] Martín, M. *TFM_Middleware.* Repositorio GitHub. Disponible en: https://github.com/mmp819/TFM_Middleware

[15] Real-Time Innovations. *Connext DDS Docs, Manuals, Services, RTI Routing Service.* [Último acceso: 30 mayo 2025]. Disponible en: https://community.rti.com/static/documentation/connext-dds/current/doc/manuals/connext_dds_professional/services/routing_service/routing_data.html

[16] Real-Time Innovations. *Products, DDS Standard.* [Último acceso: 23 mayo 2025]. Disponible en: https://www.rti.com/products/dds-standard

[17] Real-Time Innovations. *RTI Connext Getting Started.* [Último acceso: 27 mayo 2025]. Disponible en: https://community.rti.com/static/documentation/connext-dds/current/doc/manuals/connext_dds_professional/getting_started_guide/index.html

[18] Real-Time Innovations. *RTI Core Libraries, Part 5: Sending Data, Chapter 38 Data Fragmentation.* [Último acceso: 9 junio 2025]. Disponible en: https://community.rti.com/static/documentation/connext-dds/current/doc/manuals/connext_dds_professional/users_manual/users_manual/LargeData_Fragmentation.htm

[19] Real-Time Innovations. *RTI Core Libraries, Part 11: Working with Transports in Connext, Chapter 64 UDPv4, UDPv6, and Shared Memory Transport Plugins.* [Último acceso: 9 junio 2025]. Disponible en: https://community.rti.com/static/documentation/connext-dds/current/doc/manuals/connext_dds_professional/users_manual/users_manual/Setting_Builtin_Transport_Properties_wit.htm

[20] RTI Community. *RTI Connext Gateway.* Repositorio GitHub. [Último acceso: 23 mayo 2025]. Disponible en: https://github.com/rticommunity/rticonnextdds-gateway

[21] RTI Community. *RTI Connext Gateway, Plugins, Adapters, Kafka.* Repositorio GitHub. [Último acceso: 23 mayo 2025]. Disponible en: https://github.com/rticommunity/rticonnextdds-gateway/tree/master/plugins/adapters/kafka

A. Muestras de los conjuntos de datos utilizados

- **Dataset original**:

```
abs_index,timestamp,seq_index,unit1_gps1,unit1_gps1_lat,unit1_gps1_lon,unit1_gps1_altitude,unit1_gps1_hdop,unit1_gps1_pdop,unit1_gps1_vdop,unit1_rgb1,unit1_rgb2,unit1_rgb3,unit1_rgb4,unit1_rgb5,unit1_rgb6,unit1_pwr1,unit1_pwr1_best-beam,unit1_pwr1_max-pwr,unit1_pwr1_min-pwr,unit1_pwr2,unit1_pwr2_best-beam,unit1_pwr2_max-pwr,unit1_pwr2_min-pwr,unit1_pwr3,unit1_pwr3_best-beam,unit1_pwr3_max-pwr,unit1_pwr3_min-pwr,unit1_pwr4,unit1_pwr4_best-beam,unit1_pwr4_max-pwr,unit1_pwr4_min-pwr,unit1_radar1,unit1_radar2,unit1_radar3,unit1_radar4,unit1_lidar1,unit2_gps1,unit2_gps1_lat,unit2_gps1_lon,unit2_gps1_altitude,unit2_gps1_hdop,unit2_gps1_pdop,unit2_gps1_vdop,satellite_img,unit1_pwr1_best-beam_v2,unit1_pwr1_max-pwr_v2,unit1_pwr1_min-pwr_v2,unit1_pwr2_best-beam_v2,unit1_pwr2_max-pwr_v2,unit1_pwr2_min-pwr_v2,unit1_pwr3_best-beam_v2,unit1_pwr3_max-pwr_v2,unit1_pwr3_min-pwr_v2,unit1_pwr4_best-beam_v2,unit1_pwr4_max-pwr_v2,unit1_pwr4_min-pwr_v2,unit1_overall-beam
2674,11-46-31.214536,1,unit1/gps1/gps_8869_11-46-31.233334.txt,33.42170563,-111.9301714,354.8673333,0.5,1.07,0.94,unit1/rgb1/frame_11-46-31.205818.jpg,unit1/rgb2/frame_11-46-31.205818.jpg,unit1/rgb3/frame_11-46-31.205818.jpg,unit1/rgb4/frame_11-46-31.205818.jpg,unit1/rgb5/frame_11-46-31.205818.jpg,unit1/rgb6/frame_11-46-31.205818.jpg,unit1/pwr1/pwr_11-46-30.559536.txt,14,0.00035276,0.000127547,unit1/pwr2/pwr_11-46-30.559536.txt,43,0.000467878,0.000138018,unit1/pwr3/pwr_11-46-30.559536.txt,34,0.404067338,0.00044006,unit1/pwr4/pwr_11-46-30.559536.txt,61,0.000184658,0.000119566,unit1/radar1/data_9354_11-46-32.532000.mat,unit1/radar2/data_9273_11-46-32.488000.mat,unit1/radar3/data_9187_11-46-32.499000.mat,unit1/radar4/data_9102_11-46-32.512000.mat,unit1/lidar1/lidar_frame_11-46-32.474528.csv,unit2/gps1/gps_33301_11-46-31.250000.txt,33.42163314,-111.9301749,356.394,0.47,0.97,0.85,"resources/satellite_images/[33.42166934,-111.93017314]_20.png",14,0.00035276,0.000127547,43,0.000467878,0.000138018,34,
0.404067338,0.00044006,61,0.000184658,0.000119566,162
```

- **Dataset modificado de uno de los vehículos**:

```
vehicle_id,timestamp,gps_latitude,gps_longitude,gps_altitude,gps_hdop,gps_pdop,gps_vdop,speed,sector_id
veh_01,42636.4,33.40992188502363,-111.92638195544525,354.801,0.5840195658885602,
1.0601252269895198,0.978463983440186,16.459307239034178,3.0
veh_01,42636.5,33.40990702502363,-111.92638205544526,354.813,0.5354191982453957,
1.0955179482532709,0.9457487882825779,16.48175204713426,3.0
veh_01,42636.6,33.40989220502363,-111.92638205544526,354.797,0.5785069067545549,
1.176362021183725,0.9644321782305646,16.437124812597823,2.0
```

B. Ejemplo reducido de configuración de Routing Service

```
<dds>
    ...
    <routing_service name="AggregationRouting">
        <annotation>
            <documentation>
            ...
            </documentation>
        </annotation>
        <monitoring>
            ...
        </monitoring>
        <domain_route name="AggregationDomainRoute1to10">
            <participant name="1">
                <domain_id>1</domain_id>
            </participant>
            <participant name="1to10">
                <domain_id>10</domain_id>
            </participant>
            <session name="Session_1">
                <topic_route name="Route1to10" enabled="true">
                    <publish_with_original_info>true</publish_with_original_info>
                    <input participant="1">
                        <topic_name>VehicleState</topic_name>
                        <registered_type_name>VehicleSimulation::VehicleState</
                            ↪registered_type_name>
                        <datareader_qos>
                            <reliability>
                                <kind>RELIABLE_RELIABILITY_QOS</kind>
                            </reliability>
                        </datareader_qos>
                    </input>
                    <output participant="1to10">
                        <topic_name>VehicleState</topic_name>
                        <registered_type_name>VehicleSimulation::VehicleState</
                            ↪registered_type_name>
                        <datawriter_qos>
                            <reliability>
                                <kind>RELIABLE_RELIABILITY_QOS</kind>
                            </reliability>
                        </datawriter_qos>
                    </output>
                </topic_route>
            </session>
        </domain_route>
        ...
    </routing_service>
</dds>
```

C. Sección de configuración YAML para métricas de Kafka

```yaml
rules:
  # Bytes entrantes por segundo en ambos topicos
  - pattern : kafka.server<type=BrokerTopicMetrics, name=BytesInPerSec, topic=Vehicle_State
      ↪><>OneMinuteRate
    name: kafka_vehicle_states_bytes_in_per_sec
    help: Bytes entrantes por segundo del topico vehicle_states.
    type: GAUGE
    labels:
      topic: "Vehicle_State"

  - pattern : kafka.server<type=BrokerTopicMetrics, name=BytesInPerSec, topic=
      ↪Aggregated_Data><>OneMinuteRate
    name: kafka_aggregated_data_bytes_in_per_sec
    help: Bytes entrantes por segundo del topico aggregated_data.
    type: GAUGE
    labels:
      topic: "Aggregated_Data"

  # Mensajes entrantes por segundo en ambos topicos
  - pattern : kafka.server<type=BrokerTopicMetrics, name=MessagesInPerSec, topic=
      ↪Vehicle_State><>OneMinuteRate
    name: kafka_vehicle_states_messages_in_per_sec
    help: Mensajes entrantes por segundo del topico vehicle_states.
    type: GAUGE
    labels:
      topic: "Vehicle_State"

  - pattern : kafka.server<type=BrokerTopicMetrics, name=MessagesInPerSec, topic=
      ↪Aggregated_Data><>OneMinuteRate
    name: kafka_aggregated_data_messages_in_per_sec
    help: Mensajes entrantes por segundo del topico aggregated_data.
    type: GAUGE
    labels:
      topic: "Aggregated_Data"
```