Universidad de Cantabria

Facultad de Ciencias

Diseño e implementación de un servicio de monitorización y aseguramiento de la calidad y seguridad de datos publicados en Kafka

Design and implementation of a service for monitoring and assuring the quality and security of data published in Kafka

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Mario Martín Pérez

Directora: Marta Elena Zorrilla Pantaleón

Julio - 2023

Agradecimientos

Por una parte, me gustaría agradecer a mi familia el apoyo brindado durante los cuatro años de duración del grado, tanto en lo personal como en lo académico. También a los diferentes compañeros y amigos conocidos en la facultad, gracias a los cuales mi experiencia universitaria se ha visto enriquecida no solo a nivel académico, sino también en lo personal.

Por último, también me gustaría dar las gracias al Departamento de Ingeniería Informática y Electrónica de la Universidad de Cantabria por la beca de colaboración del Ministerio de Educación y Formación Profesional concedida, gracias a la cual he podido participar en el proyecto de investigación que ha dado lugar también a la elaboración del presente Trabajo de Fin de Grado. En especial en esto último, le agradezco a Marta Elena Zorrilla el apoyo y supervisión dados tanto como directora del presente proyecto y como profesora.

Resumen

Hoy en día, con el establecimiento de las nuevas tecnologías en las actividades económicas en todos sus sectores, destaca la aplicación del IoT (*Internet of Things*), donde gran parte de los procesos recaen en el trabajo llevado a cabo por un amplio despliegue de dispositivos (computadores, sensores, etc) conectados a la red.

Este proyecto, implementa un servicio de análisis, procesado y monitorización de datos disponibles en una plataforma *big data* distribuida y escalable, bajo una arquitectura centrada en el dato e implementada mediante el gestor de datos Kafka y con aplicación a casos de uso del sector industrial.

En concreto, aborda la programación de reglas para la gestión de alarmas relativas a aspectos de calidad de los datos; genera un cuadro de mandos (*dashboard*) que permite la monitorización gráfica tanto del rendimiento de la plataforma como de los datos que se publican en ella, así como garantiza en todo este proceso la seguridad del entorno, con diversas políticas tanto de autorización como de autenticación. Además, durante el desarrollo, se barajaron alternativas para la elección del sistema de persistencia y se analizaron sus ventajas e inconvenientes con el foco en la monitorización.

Palabras clave

Monitorización de eventos, procesado en tiempo real, IoT industrial, series temporales, *big data*.

Abstract

Nowadays, with the establishment of new technologies in economic activities across all sectors, the application of Internet of Things (IoT) stands out, where a significant part of the processes relies on the work carried out by a wide variety of devices (computers, sensors, etc.) connected to the network.

This project aims to propose a service for analysis, processing, and monitoring of data available on a distributed and scalable big data platform, under a data-centered architecture implemented using Kafka data manager, specifically addressed to industrial sector.

In particular, the project develops the programming of rules for alarm management related to data quality issues. It also builds a dashboard that allows graphical monitoring of both platform performance and data published in it. Throughout this process, the security of the environment is ensured by implementing different authorization and authentication policies. Additionally, alternative persistence systems were considered, along with an analysis of their strengths and weaknesses for monitoring purposes.

Keywords

Event monitoring, real-time processing, industrial IoT, time series, big data.

Índice

Índice de figuras

Índice de cuadros

1. Introducción

El presente Trabajo de Fin de Grado se ha desarrollado en el contexto del proyecto de investigación: *Monitorización y análisis de datos en streaming en arquitecturas distribuidas y escalables*, del grupo ISTR a través de una beca de colaboración con el Departamento de Ingeniería Informática de la Universidad de Cantabria. El objetivo era desarrollar un servicio de monitorización y análisis de los datos disponibles en una plataforma *big data* bajo una arquitectura Kappa, esto es, distribuida, escalable, en tiempo real y acorde a la filosofía del metamodelo RAI4.0 [10] desarrollado por el grupo.

Este metamodelo, tiene por objeto proponer una solución a la implementación de aplicaciones intensas en el uso de datos en entornos distribuidos, donde los sistemas *Internet of Things* (IoT), la computación en la nube y el procesado distribuido están presentes para abordar el volumen, velocidad y variedad de datos que se generan.

El proyecto realizado pretende, por lo tanto, exponer de forma práctica un caso de uso proveniente del IoT industrial, trabajando sobre una plataforma *big data* distribuida y escalable, utilizando el gestor de eventos Kafka [5], así como otras tecnologías específicas para la visualización gráfica de los diversos datos procesados y la monitorización de los servicios que dan soporte a la solución.

1.1. Objetivos del desarrollo

El objetivo de este TFG, consiste en diseñar y desarrollar un servicio de monitorización y análisis de los datos en una plataforma *big data* diseñada bajo una arquitectura centrada en el dato, distribuida y escalable.

Para implementar esta solución, son varios los objetivos que han debido contemplarse:

- **Asegurar la calidad de los datos**: Las fuentes de datos en ocasiones pueden entregar mediciones o conjuntos de datos que dificultan los procesos de un negocio o resultan perjudiciales para la elaboración de determinados análisis, por ejemplo, si contienen valores imprecisos o no esperados, por lo que su tratamiento y limpieza resultan necesarios.
- **Permitir la persistencia de datos**: Si bien el gestor de eventos permite la publicación y obtención de datos, este no está diseñado como un sistema para almacenar información de forma indefinida. Por lo tanto, se han planteado alternativas entre diferentes tecnologías de bases de datos, e investigado las ventajas e inconvenientes que proporcionan, con el foco en la monitorización y visualización de la información publicada.
- **Garantizar la seguridad en el acceso al gestor de eventos**: Dado que el proyecto gira en torno a Kafka, se ha planteado también el uso de sistemas de autenticación y autorización, de forma que el modelo productor-consumidor cuente con restricciones con respecto a qué usuarios (*principals* en Kafka) pueden acceder a los tópicos, así como qué acciones son permitidas o no para los que tengan acceso. Para ello, se ha planteado un sistema basado en certificados y listas de control de acceso (ACL), donde los usuarios están ligados a los certificados.
- **Obtención de métricas de rendimiento**: Con el fin de permitir no solo la observación de los datos, sino también la monitorización de los sistemas bajo los que opera la arquitectura, ha sido necesario investigar y obtener una forma de extraer métricas de rendimiento de los servicios desplegados, las cuales serán también necesarias para la generación automatizada de gráficas.
- **Generación de cuadros de mando**: Con el objetivo de facilitar tanto la monitorización como el análisis de los datos y de los servicios desplegados, se ha llevado a cabo el desarrollo de *dashboards*,

cuya actualización se realiza en tiempo real, mostrando diversas agrupaciones de gráficas y datos de interés.

1.2. Herramientas empleadas e interconexión

Debido a la amplitud de los objetivos que se cubren y el propósito específico de cada uno, ha sido necesario el uso de diversas tecnologías interconectadas entre sí, las cuales han requerido de sus respectivos tiempos de estudio, comprensión, así como de pruebas individuales con casos de uso más sencillos para verificar su funcionamiento antes de plantear una interconexión completa entre todas ellas. A continuación, se realiza una breve explicación del propósito de cada herramienta utilizada:

- **Apache Kafka**: Plataforma de gestión de eventos distribuida y enfocada a un alto rendimiento. El modelo de intercambio de datos empleado se basa en el mecanismo productor-consumidor. Esta tecnología se emplea para transmitir los datos por parte de diversos productores, de modo que uno o más consumidores puedan hacer uso de estos. Incluye además sistemas de autenticación y autorización.

  En la Figura 1, se muestra una representación gráfica con el fin de explicar el mencionado modelo productor-consumidor, concretamente en Kafka [5], que ha sido el gestor de eventos empleado para el proyecto. Como puede observarse, entran en juego tres conceptos relevantes:

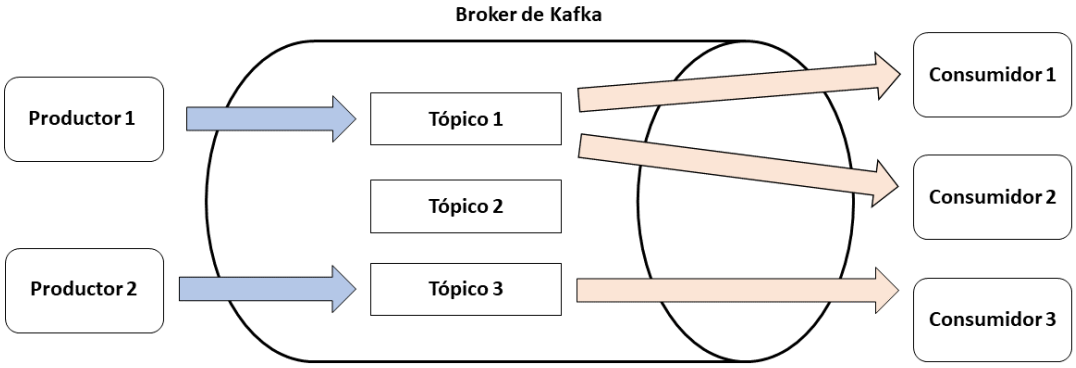

Figura 1: Modelo productor-consumidor simplificado.

  - **Tópico**: Cada una de las categorías que agrupan los diferentes datos/mensajes enviados y consumidos. Por ejemplo, en un sistema con varios equipos de mediciones, podría existir un tópico para cada equipo que agrupase sus respectivas muestras. Debido a esto, cada tópico debe ser único, pues su nombre servirá de identificador a la hora de establecer la conexión, ya sea por parte del productor o del consumidor.
  - **Productor**: Encargado de enviar datos a los tópicos. No existe un límite explícito de cuántos productores pueden enviar datos a un tópico, si bien cuanto mayor sea el número, se deberá tener en cuenta la sobrecarga de la red y los nodos operativos.
  - **Consumidor**: Como el propio nombre indica, consume u obtiene los eventos contenidos en los diversos tópicos. Del mismo modo, tampoco existe un límite concreto, si bien de nuevo es importante considerar factores como la red.

- **Apache Zookeeper**: Servicio empleado para la coordinación y el control de aplicaciones distribuidas. Se incluye por defecto tanto en la instalación de Kafka, como la de Druid, siendo necesario

para ambas tecnologías y encargándose principalmente del almacenamiento de los metadatos de estas y parte de su configuración.

- **Apache Spark**: Framework de computación utilizado para el procesamiento de datos [6], en este caso, provenientes de Kafka. Su diseño se encuentra principalmente orientado a la velocidad en el procesado distribuido de datos y garantiza la gestión de *streams*, además de ser compatible con su uso en varios lenguajes; como Python, Scala o Java, siendo este último el empleado en el proyecto, por su madurez, rendimiento y amplio soporte.
- **Apache Cassandra y Apache Druid**: Bases de datos NoSQL estudiadas para la persistencia de la información. Más adelante, en la presente memoria, se explican ambas tecnologías con una mayor profundidad, así como sus ventajas e inconvenientes y la elección más apropiada para el propósito de este TFG.
- **Prometheus**: Sistema de monitorización y generación de alertas [21]. Mediante su uso, es posible recopilar métricas de rendimiento de los *brokers* de Kafka. Para obtener estas métricas y hacerlas visibles a Prometheus, es necesario hacer uso previamente de un *exporter* JMX, que obtiene métricas de programas Java (Kafka en este caso) y las hace visibles en un servidor HTTP.
- **Grafana**: Software libre utilizado para la consulta y visualización de métricas, así como la generación de alertas [12]. Su principal potencial en este proyecto radica en la posibilidad de generar diversos cuadros de mando sobre diferentes fuentes de datos. En este proyecto, dichas fuentes son Prometheus y Druid.
- **Keytool**: Herramienta que se incluye en el paquete de software Java Development Kit (JDK) y que ha sido utilizada para la generación de certificados.
- **Lenguajes**: Se ha utilizado Java para desarrollar el código de los productores y consumidores en el framework Apache Spark. Para la utilización de las diversas herramientas desde Linux (Ubuntu 22.04) se ha requerido del uso de Bash. Además, ha sido necesario también emplear los lenguajes de consulta correspondientes con las bases de datos:
  - **Cassandra Query Language (CQL)**: Utilizado para crear y consultar las estructuras de datos necesarias en Apache Cassandra.
  - **Druid SQL**: Su sintaxis es similar al lenguaje SQL tradicional, con la particularidad de que Druid posteriormente traduce las sentencias realizadas a su lenguaje nativo. Ambas opciones pueden usarse indistintamente.

  Finalmente, Prometheus también requiere del uso de PromQL para consultar desde Grafana las métricas almacenadas.
- **Formatos de texto**: Son varios los formatos que se han utilizado durante el desarrollo, destacando principalmente CSV y JSON para el trabajo con ficheros de datos (*datasets*), así como el uso de YAML para gestionar algunas configuraciones de las herramientas.

Si bien, a lo largo de la memoria se explicará en detalle el uso de estas herramientas durante el proyecto, a continuación se realiza una breve exposición de las conexiones principales, con el fin de contextualizar e introducir al lector a una visión general de la solución software completa, representada gráficamente en la Figura 2:

- **Productor**: En este proyecto, este elemento consistirá en un programa Java encargado de leer un conjunto de datos y enviarlo a un tópico en Kafka donde se publiquen los datos en crudo. Pueden ser varios los productores que manden datos al mismo tiempo.

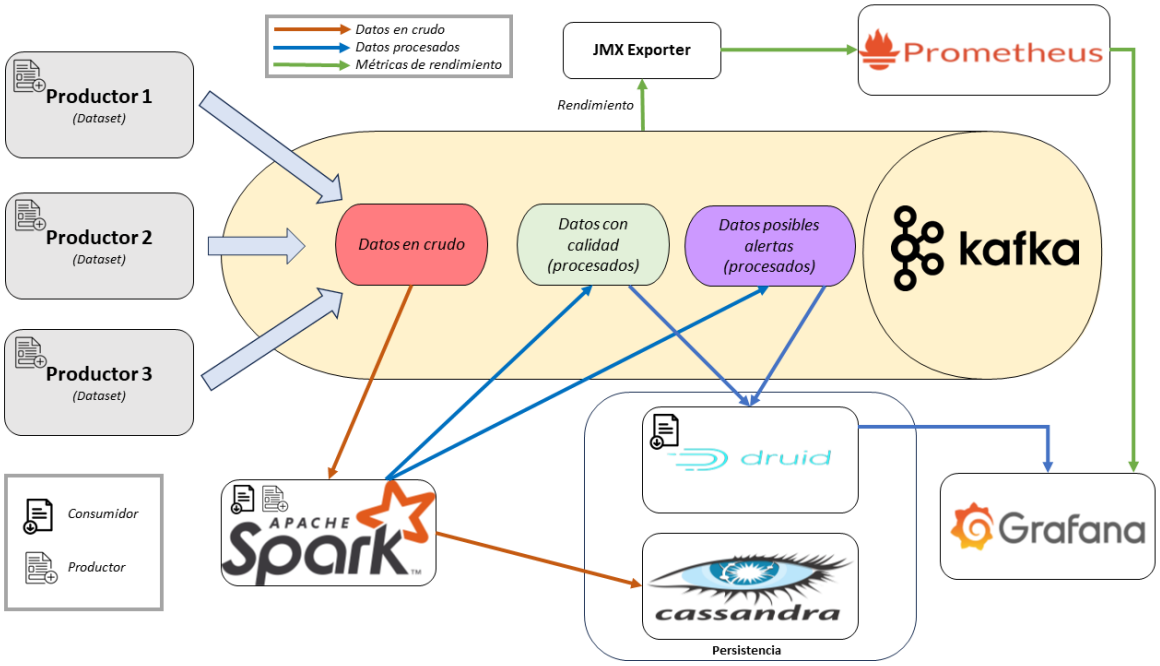

Figura 2: Arquitectura y solución software propuesta.

- **Consumidor-Productor Spark**: Este programa Java bajo Spark consumirá los datos en crudo de su respectivo tópico de Kafka. Posteriormente, los analizará y enviará a sus tópicos correspondientes una vez procesados, agrupando estos datos según sean considerados válidos o puedan dar lugar a posibles alarmas. Dado que Kafka no está diseñado para la persistencia a largo plazo, almacenará los datos en crudo en una base de datos.

  Debido a este carácter en el que el programa consume de un tópico y produce hacia otros, se le considera tanto consumidor como productor.

- **Consumidor Druid**: Apache Druid permite de forma nativa una conexión a los tópicos de Kafka, con sus certificados correspondientes en caso de requerir autenticación y autorización. En esta ocasión, realizará labores de ingesta de datos de los tópicos que poseen la información ya procesada o clasificada.

- **JMX Exporter**: Programa configurado para obtener métricas de rendimiento de Kafka y permitir a Prometheus leer para su almacenamiento o procesamiento posterior.

- **Prometheus**: Permite obtener métricas de rendimiento de diferentes sistemas software. En este caso, se configura una determinada ruta HTTP, donde el *exporter* hace visibles los datos recopilados.

- **Grafana**: Configurado para contar con Prometheus y Druid como fuentes de datos. Su finalidad es contener cuadros de mando con varias gráficas actualizadas en tiempo real, en base a las consultas definidas sobre las fuentes de datos mencionadas.

1.3. Conjuntos de datos empleados

A lo largo del desarrollo del trabajo, se han empleado varios conjuntos de datos con el fin de probar y estudiar las diferentes tecnologías, hasta finalmente plantear un caso de uso que permitiera probar el conjunto de todas ellas conectadas entre sí.

A continuación, se describen estos conjuntos de datos de forma resumida, así como finalmente de forma más detallada aquel que se ha empleado como caso de uso. Una muestra de los datos contenidos en estos ficheros se encuentra en el anexo A.

1.3.1. Datos ambientales

Este conjunto de datos se caracteriza por ser bastante reducido y sencillo de entender. Su acceso, además, se encuentra público a través del portal de datos abiertos de Santander [9] e incluye el siguiente conjunto de medidas tomadas a partir de sensores ambientales: identificador de la medición, fecha, tipo de medición, latitud, longitud, temperatura registrada, batería del sensor, niveles de luz y ruido y finalmente, una URI de la medida que puede ser accedida de forma individual.

Debido a su tamaño ligero y sencillez, ha resultado idóneo para estudiar el funcionamiento de Kafka y Spark, así como verificar, junto con la creación de consumidores y productores sencillos, los mecanismos de autenticación y autorización.

1.3.2. Datos de juego web

Este conjunto de datos obtenido del juego web *Koalas to the Max* [16], que consiste en pasar el ratón por varias filas de puntos en pantalla hasta descomponerlos al máximo, contiene un volumen de datos considerablemente mayor (> 465000 tuplas de diferencia), por lo que permite realizar comparaciones de rendimiento más interesantes, así como observar algunas particularidades, como pueden ser aquellos casos en los que existan datos compuestos, factor que resulta también de interés de cara a elegir una base de datos.

Además, ha permitido el estudio realizado en uno de los próximos capítulos, dedicado a la investigación de alternativas de persistencia. Por ello, se realiza a continuación una explicación ligeramente más detallada. Los campos más significativos son:

- **Timestamp**: Recoge la fecha y hora de la medición.
- **Session**: Un identificador de la sesión establecida en la web.
- **Event**: Recoge eventos de diversos tipos, requiriendo un campo compuesto. Por lo general, se recoge el tipo de evento y el dato relativo al mismo. Se adjunta, como ejemplo, el caso en el que la web detecta que la fila 6 de puntos de la imagen ha sido completamente mostrada:

```
1    "event":{
2        "type":"LayerClear",
3        "layer":6
4    }
```

- **Agent**: De nuevo recae en el uso de un campo compuesto, recogiendo datos relativos al dispositivo con el que se accede a la web.
- **Client_IP**: IP del cliente conectado.

- **Geo_IP**: Datos de geolocalización asociados a la IP del cliente. De nuevo, conforman un dato compuesto.
- **Languages**: Idiomas utilizados recogidos como una lista, lo cual afectará de nuevo también al método escogido para almacenar la información.

1.3.3. Datos del metro de Oporto (MetroPT)

Este conjunto de datos disponible públicamente en Zenodo [25] ha sido utilizado en un estudio para el mantenimiento predictivo de componentes mecánicos de metros de Oporto, en Portugal. Dicho estudio disponible también públicamente y realizado por Bruno Veloso, Rita P. Ribeiro, João Gama y Pedro Mota Pereira [26], ha sido además de gran utilidad para crear un caso de uso en el que poder establecer reglas de interés que den lugar a unas determinadas alarmas, pues en dicho estudio se realiza una explicación detallada de las diferentes métricas y su significado, como se irá desarrollando a lo largo de esta memoria.

Gracias a su gran volumen y variedad de campos, ha sido escogido para poner en práctica todas las herramientas estudiadas durante el proyecto. A continuación, se explican resumidamente algunos de los campos presentes más relevantes, de forma que futuras decisiones explicadas en esta memoria, puedan ser comprendidas con una mayor facilidad. Una descripción más técnica puede encontrarse en la investigación original en caso de interés [26].

- **Valores analógicos**: Se corresponden con mediciones de diversas válvulas y sensores.
  - TP2: Presión en bares del compresor.
  - TP3: Presión en bares generada en el panel neumático.
  - H1: Presión en bares de una válvula que se activa únicamente con detecciones superiores a 10.2 bares.
  - DV_Pressure: Presión en bares que se ejerce por una caída de presión generada cuando se descarga agua.
  - Reservoirs: Presión en bares de los tanques de aire.
  - Oil_Temperature: Temperatura del aceite en el compresor.
  - Flowmeter: Aire en m^3/h medido en el panel neumático.
  - Motor_current: Corriente presente en el motor.
- **Valores digitales**: Señales que se corresponden con el funcionamiento de diversas válvulas y sensores.
  - LPS: Se activa en el momento que se detecta una caída de presión por debajo de los 7 bares.
  - Towers: Valor binario que indica qué torre del tren se encuentra bajo carga.
  - Oil_level: Se activa si el nivel de aceite cae por debajo de los valores esperados.
- **Información GPS**: Mediante una antena auxiliar, el metro es capaz de obtener información mediante GPS relativa a su posición:
  - GpsLong: Longitud de la posición en grados.
  - GpsLat: Latitud de la posición en grados.
  - Speed: Velocidad en kilómetros por hora.
  - GpsQuality: Valor binario que indica si se ha podido obtener la información del GPS en la medida. Si la señal no es válida, todos los valores del GPS aparecerán como 0.

El caso de uso definido para el propósito de este TFG, consiste en elaborar un productor Java capaz de enviar todo el conjunto de datos a un tópico de Kafka. Los datos serán leídos de dicho tópico y procesados para garantizar su calidad. Para ello, se tendrá en cuenta si existen riesgos por caída de presión y se separarán también si se pierde la señal GPS. De esta forma, finalmente en un cuadro de mando podrá visualizarse un mapa con las posiciones donde se producen alertas de presión, comprobar durante qué periodos de tiempo se activa la señal LPS y el valor de la presión en TP3, la temperatura del aceite, o mostrar el estado de la cobertura del GPS.

A lo largo de las próximas secciones, se mostrarán los pasos seguidos para hacer esto posible.

1.4. Organización del trabajo

Esta memoria se encuentra organizada en otras seis secciones, donde en cada una de ellas se abordarán aspectos concretos del trabajo elaborado para la construcción del servicio de monitorización y aseguramiento de la calidad y seguridad de datos:

- **Sección 2 - Garantía de calidad en los datos**: Explica la configuración básica de Kafka para publicar y distribuir los tópicos definidos en el entorno. Además, se especifican los pasos seguidos para la elaboración del programa Java con Spark encargado de revisar los datos entrantes, procesarlos y filtrarlos a los tópicos adecuados para generar las alertas. De forma resumida, también se detallará en qué consiste el productor utilizado.
- **Sección 3 - Sistema de almacenamiento de datos**: En esta sección, se realiza un análisis de Cassandra y Druid, como bases de datos alternativas para garantizar la persistencia de los datos operados con Kafka a largo plazo. Se realiza un breve resumen de ambas tecnologías, al mismo tiempo que se detallan ventajas e inconvenientes relativos a estas para el objetivo que se persigue, finalizando con un estudio que analiza su rendimiento con el conjunto de datos de *Koalas to the Max*.
- **Sección 4 - Seguridad en el acceso al dato**: Está compuesta de dos subsecciones principales. En la primera de ellas, se tratará el mecanismo de autenticación escogido para el proyecto, así como su forma de ser aplicado a las diferentes tecnologías conectadas en el sistema. Finalmente, la segunda subsección recogerá las diferentes reglas elaboradas para crear listas de control de acceso y su funcionamiento, dando lugar así a una solución para garantizar la autorización.
- **Sección 5 - Monitorización del rendimiento**: Detalla el proceso seguido para obtener métricas de rendimiento de Kafka a través de un *exporter* JMX y hacerlas visibles en un servidor HTTP. Seguidamente, se especifican los pasos necesarios para conectar Prometheus a dicho servidor, de forma que pueda realizarse un almacenamiento de las métricas leídas.
- **Sección 6 - Visualización de datos y métricas de rendimiento**: Expone los pasos seguidos para conectar Grafana con las diversas fuentes datos, así como finalmente crear los cuadros de mando con diferentes gráficas que reflejen tanto la monitorización del rendimiento, como los datos gestionados y posibles alarmas originadas por estos.
- **Sección 7 - Conclusiones y líneas futuras de aplicación**: Por último, en esta sección se detallan las conclusiones sobre el trabajo realizado, así como ampliaciones que pueden realizarse tomando el proyecto como raíz.

2. Garantía de calidad en los datos

La posibilidad ofrecida por el gestor de eventos Apache Kafka de poder agrupar diversos mensajes por categorías, a las que denominamos tópicos, resulta de gran utilidad para organizar las diferentes métricas en grupos de datos que hayan sido recién tomados, aquellos cuya calidad se encuentre garantizada y otros que puedan dar lugar a unas alertas determinadas.

Dentro de esta sección, se desarrollan y tienen en cuenta los componentes y conexiones presentados en la Figura 3.

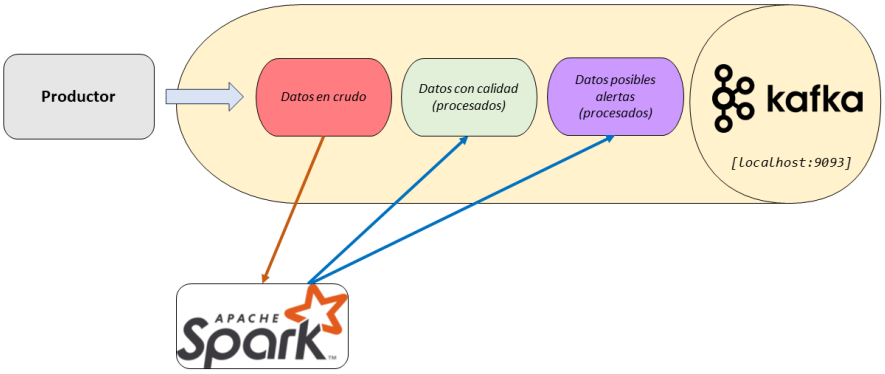

Figura 3: Componentes implicados en la gestión de la calidad de los datos.

Con el fin de facilitar la explicación de la configuración y despliegue de cada componente, se seguirá el siguiente orden:

- **Apache Zookeeper y Apache Kafka**: Arranque de ambos sistemas y creación de las estructuras necesarias para las agrupaciones de datos.
- **Productor**: Programa Java encargado de leer el fichero que contiene el conjunto de datos y enviar los datos a Kafka. Concretamente, dentro del caso de uso de trenes utilizado como ejemplo, se trata de un fichero CSV, con las tuplas separadas por filas y los elementos de cada tupla por comas.
- **Apache Spark**: Dentro del programa Java desarrollado, se tratará el consumo de datos en crudo, así como la producción hacia los tópicos con datos procesados y las reglas utilizadas para ello.

2.1. Configuración y despliegue de Apache Zookeeper y Apache Kafka

Apache Zookeeper es un servicio que facilita la coordinación entre aplicaciones distribuidas, como es en esta ocasión Kafka y próximamente Druid. Para ello, utiliza un espacio de nombres compartido, el cual queda organizado jerárquicamente en forma de árbol, como se muestra en la Figura 4 y cuyas ramificaciones se asemejan a las creadas por un sistema de archivos tradicional, conociéndose cada uno de sus nodos como *Znodes*.

Este será el sistema del que se sirva Kafka para poder almacenar la información de su clúster, empleando diversos nodos en función de la información a almacenar, los cuales serán descendientes de

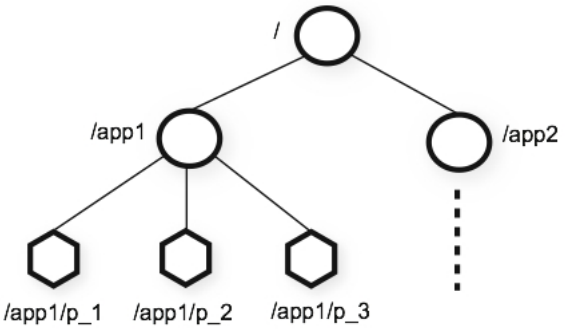

Figura 4: Espacio de nombres de Zookeeper [8].

un nodo dedicado a Kafka. Un nodo del espacio de nombres, servirá como punto de entrada y representará al clúster desplegado y a partir de él, nacerán nodos *znodes* que contengan información sobre los diferentes *brokers*, tópicos, particiones... del despliegue, siendo todo esto gestionado automáticamente entre Kafka y Zookeeper, si bien también puede recibir ajustes por parte del administrador. Finalmente, es una tecnología caracterizada por una fuerte consistencia entre los diversos servidores replicados, planteados con la finalidad de que cada aplicación establezca una conexión TCP con uno de ellos y, en caso de esta verse cortada en algún momento, ser capaz de cambiar de servidor, garantizando además la disponibilidad.

Por otra parte, Apache Kafka sirve como plataforma para la distribución de datos, siguiendo el modelo productor-consumidor. Debido a su diseño preparado para gestionar grandes volúmenes de información, es necesario su planteamiento como aplicación distribuida, para lo cual usamos el servicio Zookeeper previamente descrito. Gracias a esto, conseguimos una solución para el manejo de los datos en tiempo real, que además goza de bastante reconocimiento y alternativas capaces de conectarse a ella como consumidores, desde bases de datos como Apache Druid, a frameworks de computación como Apache Spark.

2.1.1. Inicialización

Antes de comenzar con los pasos correspondientes con la inicialización de Zookeeper y Kafka, cabe señalar que las versiones descargadas desde la web oficial de Apache Kafka [4] incluyen ya de por sí una versión de Zookeeper, facilitando así la obtención del software necesario, si bien este servicio también puede ser incorporado de forma independiente. A fecha de comienzo de la elaboración del proyecto, la versión más reciente se consideraba la 3.3.1, la cual ha sido escogida por su resolución de *bugs*, así como para garantizar una mayor seguridad, incorporando la autenticación mediante SSL/TLS, la cual puede no encontrarse en las versiones más primitivas de Kafka y de la cual se hará uso posteriormente.

Además, cabe destacar que Zookeeper y Kafka pueden configurarse como servicios dentro del sistema Linux. No obstante, con el fin de garantizar una mayor flexibilidad a la hora de estudiar y experimentar con las plataformas, su uso se realiza con la ejecución manual de sus *scripts*.

- **Arranque de Zookeeper**: Para realizar este paso, se presentan dos alternativas disponibles, debido a la inclusión de forma nativa también del servicio Zookeeper en Apache Druid:
  - Inicializar Druid al comienzo, de forma que el servicio de Zookeeper desplegado automáticamente sea aprovechado tanto por Kafka como Druid. La configuración ofrecida por defecto, es suficiente para las características del proyecto y facilita el despliegue de las aplicaciones. Dado que cada aplicación usa su respectivo conjunto de nodos, no existen conflictos. Debido

a su sencillez y eficacia, se ha considerado como la alternativa a escoger. Para ello, en la siguiente sección dedicada al estudio de las alternativas de persistencia, se indicará cómo arrancar Druid.

- Utilizar el servicio Zookeeper incluido con Kafka: Ofrece el mismo funcionamiento que la alternativa previa, aunque es necesario tener en cuenta el conflicto de puertos con respecto a Zookeeper. Para no originar problemas entre el arranque de Druid y Kafka, es recomendable cambiar el puerto asignado por defecto a la hora de realizar el despliegue, alternativa disponible en el fichero de configuración *zookeeper.properties*. Ya que esta aproximación implica mayores dificultades que la previa, se ha descartado. No obstante, se adjunta a continuación el comando de arranque, ya que esta alternativa ha sido de utilidad en las fases de estudio que no han involucrado el uso de Druid:

```
./bin/zookeeper-server-start.sh config/zookeeper.properties
```

- **Arranque de Kafka**: De forma similar, cuenta también con su propio *script* de arranque, así como un fichero de configuración denominado *server.properties*. Diversas modificaciones se irán mencionando tanto en el *script* de arranque como en el fichero de configuración, según sean abordados los aspectos de seguridad y monitorización.

```
./bin/kafka-server-start.sh config/server.properties
```

Este proceso debería realizarse en los diferentes equipos involucrados en el clúster, si bien el trabajo ha sido realizado en un solo equipo de forma local.

2.1.2. Creación de tópicos

Atendiendo al estudio del *dataset* MetroPT [26], se establecen un conjunto de reglas para asegurar la calidad de los datos. Debido al carácter técnico del estudio, se contemplan algunas sentencias que no requieran aplicar modelos predictivos sobre valores analógicos, ya que esto quedaría fuera del ámbito del trabajo a desarrollar. Las reglas definidas son las mostradas en el Cuadro 1:

N.º	Regla de calidad
1	Una tupla se considerará de calidad cuando sus valores no se vean afectados por fallos en los sistemas de presión de los trenes y pueda determinarse su posición, es decir, cuando la señal LPS tenga un valor igual a 0 y la señal de calidad de GPS sea igual a 1.
2	Una tupla dará lugar a una alarma por posibles averías relacionadas con la presión, siempre que se active el indicador LPS (señal LPS igual a 1).
3	Una tupla que no disponga de señal GPS (señal de calidad de GPS igual a 0), deberá ser analizada por separado, con el fin de que el personal a cargo identifique si se corresponde con un paso por túnel o un estacionamiento o si, por el contrario, requiere un tipo de atención especial.

Cuadro 1: Reglas de calidad.

Por lo tanto, es necesario realizar cuatro agrupaciones de datos en total, o lo que es lo mismo, crear cuatro tópicos de la siguiente forma:

- **Lecturas**: Contiene las tuplas enviadas en crudo y sin ningún tipo de procesamiento.

```
./bin/kafka-topics.sh --bootstrap-server mario:9092 --create --topic lecturas
```

- **Lecturas filtradas**: Contiene las tuplas que, después de haber sido procesadas, se considera que tienen la calidad necesaria, cumpliendo con la regla número uno.

```
./bin/kafka-topics.sh --bootstrap-server mario:9092 --create --topic lecturas_filtradas
```

- **Lecturas con anomalías**: Contiene aquellas tuplas con algún dato fuera de lo normal, en este caso aquellas que pueden generar una alerta relativa a valores de presión en el sistema, cumpliendo con la regla número dos.

```
./bin/kafka-topics.sh --bootstrap-server mario:9092 --create --topic
    ↪lecturas_anomalas_alert
```

- **Lecturas sin GPS**: Contiene tuplas cuyos valores de GPS no han podido ser registrados, cumpliendo con la regla número tres.

```
./bin/kafka-topics.sh --bootstrap-server mario:9092 --create --topic lecturas_gps_alert
```

La creación de los tópicos, de esta forma, lleva de manera implícita a indicar que el factor de replicación (copias de los datos) y el número de particiones serán igual a 1, debido a que se está realizando el despliegue en un solo equipo de forma local. En un clúster real, deberían especificarse los parámetros *replication-factor* y *partitions*. Por último, utilizando *bootstrap-server*, indicamos el *broker* con el que debe realizarse la conexión, que en este caso es el equivalente a aquel con dirección IP igual a 127.0.0.1 en el puerto 9092, es decir, el propio equipo.

En conexiones futuras, como se verá posteriormente, una vez se aplique comunicación con protocolo SSL/TLS, el puerto pasará a ser 9093 y requerir certificados, ya que el puerto 9092 está reservado para el uso de conexiones que emplean texto plano.

2.2. Programa "productor"

Consiste en un programa Java que envía tuplas de un fichero CSV, simulando el comportamiento que realizarían varios productores enviando datos en tiempo real. El caso de uso se encuentra limitado a las medidas sobre un único tren, como se aprecia en el conjunto de datos. No obstante, ampliarlo resultaría un proceso sencillo donde podrían existir tópicos para cada ferrocarril o al menos como alternativa podría agregarse un campo en cada tupla con un identificador en función del vehículo.

El programa diseñado cuenta con una serie de parámetros a introducir por consola:

- Servidor *bootstrap*: *Broker* de Kafka al que se realiza la conexión, comprobando la validez del formato mediante una expresión regular que permite incluir dirección y puerto. Se permite, por lo tanto, incluir una dirección tanto mediante número como un nombre asociado y, opcionalmente, el puerto:

```
^[a-zA-Z0-9._-]+(:[0-9]+)?$
```

  El *broker* será *localhost:9093*, debido a que el estudio se realiza sobre un único equipo.

- Tópico: Nombre del tópico al que deben ser enviados los datos leídos del fichero CSV. Al igual que antes, se verifica que el nombre cumple un formato básico. En esta ocasión, debe estar compuesto por letras minúsculas o mayúsculas, permitiendo dígitos, guiones y puntos:

```
^[a-zA-Z0-9._-]+$
```

- Archivo CSV: Nombre o ruta del fichero CSV del que extraer la información.

Una vez verificados los parámetros, se realiza la configuración del productor de Kafka, estableciendo el servidor que se indica como parámetro y fijando los serializadores a usar tanto para las claves como los valores de los mensajes:

```
Properties propiedades = new Properties();
propiedades.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, servidoresBootstrap);
propiedades.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.
    ↪getName());
propiedades.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class
    ↪.getName());

// Productor Kafka
KafkaProducer<String, String> productor = new KafkaProducer<>(propiedades);
```

A continuación, el proceso de envío se realiza con una lectura del fichero línea por línea, debido al amplio tamaño del conjunto de datos (1.6 GB). Cabe destacar que antes de realizar el envío, cada cadena de texto leída debe de ser encapsulada dentro de un registro concreto ofrecido por la librería de Kafka para Java, de forma que para cada línea se realiza lo siguiente:

```
ProducerRecord<String, String> record = new ProducerRecord<>(topico, string);
productor.send(record);
productor.flush();
```

Finalmente, es recomendable siempre realizar una llamada al productor para cerrarlo de manera ordenada y a través del propio programa Java encargado de su apertura, la cual se ha incluido al final del programa, que termina una vez se han leído todas las filas del fichero CSV y han sido estas enviadas al tópico correspondiente de Kafka.

2.3. Programa de preprocesado de datos en Apache Spark

Apache Spark es un motor de computación orientado al alto rendimiento y la escalabilidad, permitiendo el procesado de grandes volúmenes de datos de forma eficiente. De cara a esto último, resultará de gran interés en particular para este trabajo el procesado de datos en *streaming* en tiempo real. Además, gracias a disponer de una arquitectura distribuida, su despliegue en un entorno con varios equipos favorecería aún más el rendimiento, debido a su capacidad de procesamiento en paralelo con alta tolerancia a errores.

A la hora de aplicar esta tecnología en el proyecto, se utiliza una extensión del núcleo de Spark, denominada Spark Streaming, enfocada a la ingesta de *streamings* de datos, para su procesado y finalmente su persistencia en ficheros o bases de datos [7]. Por lo tanto, el programa realizado, debe cumplir con los siguientes objetivos:

- Establecer un *stream* como consumidor de Kafka: El tópico de interés será el que contiene los datos en crudo.

```
JavaInputDStream<ConsumerRecord<String, String>> kafkaStream = KafkaUtils.
    ↪createDirectStream(jsc, LocationStrategies.PreferConsistent(),
    ↪ConsumerStrategies.<String, String>Subscribe(topics, kafkaConf));

// Convierte los registros de Kafka en registros del tipo TrainRecord
JavaDStream<TrainRecord> registros = kafkaStream.map(registro -> new TrainRecord(
    ↪registro.value()));
```

- Procesar los datos: Identificar los tópicos a los que redirigir cada tupla. Las posibilidades ofrecidas por Spark permiten incluso elaborar métodos personalizados para descartar conjuntos de datos. Como ejemplo, se adjunta una sección simplificada de código utilizada para determinar si una tupla se considera de calidad suficiente. Para ello, se obtiene una nueva tupla y se analiza el estado de las señales GPS y LPS.

```
registros.foreachRDD(rdd -> {
  rdd.foreachPartition(partitionOfRecords -> {
    KafkaProducer<String, String> productor = new KafkaProducer<>(kafkaConf);
    TrainRecord aux;
    ProducerRecord<String, String> record;

    while (partitionOfRecords.hasNext()) {
      aux = partitionOfRecords.next();
      String medidas = buildRow(aux);

      if (aux.getLps() != LPS_ACTIVO && aux.getGpsQuality() != GPS_PERDIDO) {
        record = new ProducerRecord<>(TOPIC_FILTRADO, medidas);
        productor.send(record);
      }
      ...
```

- Enviar datos a sus respectivos tópicos de destino: Para ello, se configura un productor de la misma forma que en la subsección previa.

- Persistir datos en crudo para su uso futuro: Será necesario para ello elaborar un *pool* de conexiones con Cassandra y utilizarlo para enviar los datos correspondientes.

```
CassandraConnectionPool pool = CassandraConnectionPool.getInstance();
Session session = pool.getConnection();
...
query = buildCassandraQuery(CASSANDRA_TABLA, medidas);
session.execute(query);
```

Con el fin de aportar cierta flexibilidad al programa, se permite indicar por parámetro el *broker* de Kafka, el tópico del que deben leerse los datos en crudo y finalmente, la dirección de conexión con Cassandra. De igual forma que con el código presentado previamente para el productor de la sección anterior, se realiza una comprobación con expresiones regulares para controlar la validez de los argumentos. Por último, el consumidor de Spark también requiere de la elaboración de una configuración:

```
Map<String, Object> kafkaConf = new HashMap<String, Object>();
kafkaConf.put("bootstrap.servers", servidoresBootstrap); // Indicado como parametro
kafkaConf.put("key.serializer", StringSerializer.class);
kafkaConf.put("key.deserializer", StringDeserializer.class);
kafkaConf.put("value.deserializer", StringDeserializer.class);
kafkaConf.put("value.serializer", StringSerializer.class);
kafkaConf.put("group.id", "1");
kafkaConf.put("auto.offset.reset", "latest");
kafkaConf.put("enable.auto.commit", true);
```

Concretamente, con las dos últimas líneas, se especifica que el consumidor obtenga la información más reciente disponible de Kafka (aquella que comience a llegar tras su ejecución) y que notifique a

Kafka del proceso de la lectura realizada de forma automática respectivamente.

En la Figura 5, se muestra el esquema que recoge todos los procesos realizados dentro de este programa:

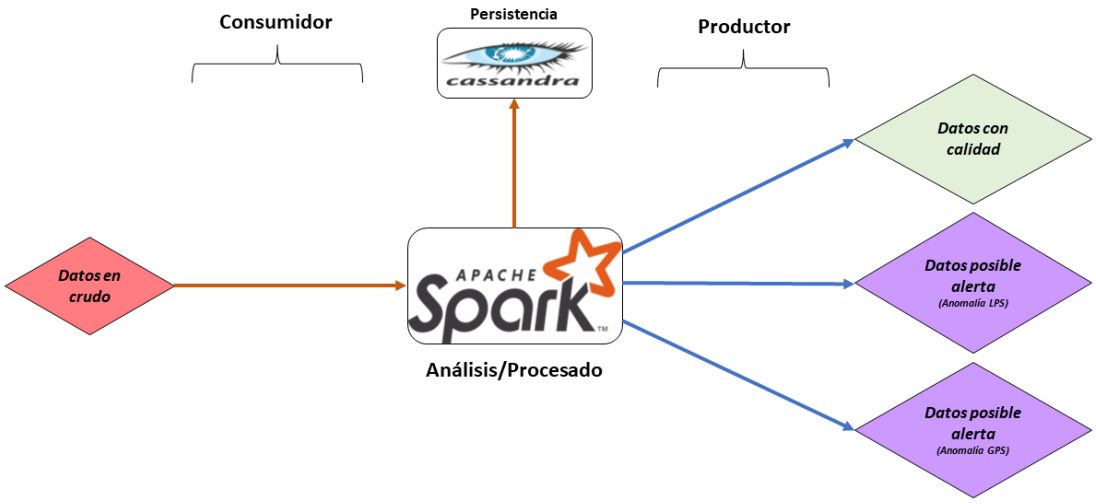

Figura 5: Esquema de procesado de datos con Spark.

La finalización del mismo, puede ser realizada de forma manual deteniéndolo a través del terminal de Linux. De lo contrario, mientras no existan excepciones durante su ejecución, este quedará a la espera de recibir nuevos datos que deban ser analizados y persistidos. Cabe destacar, además, que el usuario podrá observar en todo momento si ocurre algún tipo de imprevisto, ya que Spark emplea la librería Log4j con el fin de mostrar y gestionar mensajes de registro.

3. Sistema de almacenamiento de datos

Si bien Kafka es el sistema distribuido utilizado como tubería de datos entre los distintos dispositivos o aplicativos implicados, no está diseñado para la persistencia de dichos datos o consultas sobre estos. Para ello, es necesario utilizar bases de datos, que nos permitan realizar tanto el almacenamiento como la consulta de la información, proporcionando además al usuario diversas utilidades en la consulta, como puede ser la elaboración de agregaciones.

El uso de una base de datos, no solo resulta relevante de cara al almacenamiento de la información, sino que también permitirá hacer uso de otros servicios que puedan conectarse a ella, como se realizará próximamente con Grafana, con el fin de mostrar gráficamente los datos almacenados y facilitar su monitorización.

Debido a que el sistema planteado debe de ser capaz de realizar la ingesta de datos con un ritmo relativamente alto, para poder procesar los datos en tiempo real, se considera que la tecnología a utilizar debe de encontrarse dentro de las clasificadas como NoSQL. Es decir, una que emplee un modelo más flexible que el relacional tradicional, facilitando la ingesta y manejo de grandes conjuntos de datos con baja latencia y escalabilidad horizontal.

Para ello se ha planteado el uso de dos alternativas relevantes dentro del mundo de las bases de datos NoSQL, siendo Cassandra una de las mayores representantes, seguida de Druid, como alternativa más novedosa y reciente, pero capaz de ofrecer una serie de ventajas en el ámbito de la ingesta de datos en tiempo real y en la conexión a realizar con Kafka.

A continuación, se analizan las ventajas de cada una de estas alternativas, de forma que pueda decidirse qué tecnología usar basándose en sus virtudes y su adecuación a los requerimientos. Para ello se han empleado las correspondientes documentaciones de Apache Cassandra [1] y Apache Druid [2], así como un artículo elaborado por Rachel Pedreschi [20], ingeniera que compara ambas tecnologías después de haber trabajado en DataStax, compañía que distribuye y soporta Cassandra, e Imply, creadores de Druid.

3.1. Apache Cassandra

Cassandra es una base de datos distribuida NoSQL, altamente escalable y que pertenece al paradigma de familias de columnas, siendo reconocida en gran parte por su uso por compañías de renombre, como Apple o Facebook, entre otras.

Entre sus principales virtudes, cabe destacar la tolerancia a caídas de nodos, debido a su capacidad de replicación en diferentes centros de datos, permitiendo el reemplazo de nodos y ofreciendo así un servicio ininterrumpido ante este tipo de fallos. Dicha replicación, provoca también menores latencias.

Por otra parte, entre los diversos nodos que además contienen copias de los mismos datos, se aplica un protocolo P2P, donde la arquitectura pretende que sus diferentes nodos compartan información de forma continua. Una vez el cliente se conecta a uno de ellos, siendo este nodo el coordinador, se encargará dicho nodo de decidir quién será el encargado de responder las respectivas consultas, como se muestra en la Figura 6.

Cabe destacar que su almacenamiento es eventualmente consistente, es decir, las diversas copias del dato pueden tomar su tiempo en encontrarse actualizadas por igual.

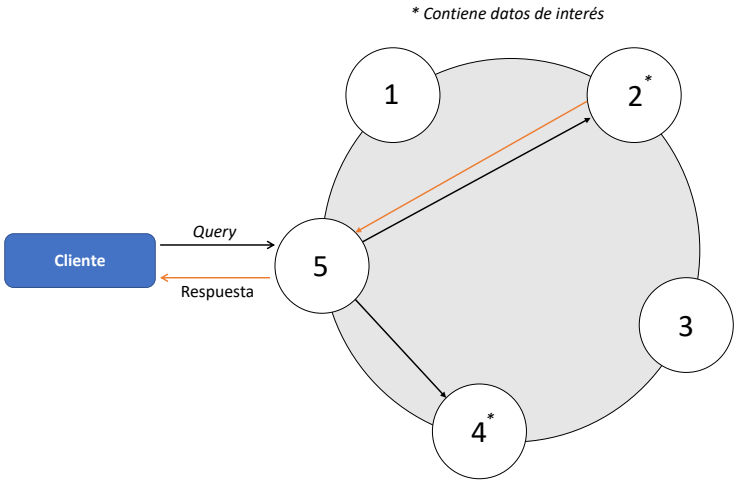

Figura 6: Esquema de comunicación entre nodos de Cassandra.

Una vez realizada esta breve introducción, se exponen a continuación las ventajas que ofrece Cassandra en su empleo con Kafka:

- **Velocidad de escritura**: Su diseño permite una de las escrituras más veloces y con menor latencia dentro de las bases de datos NoSQL, aunque encontrándose más enfocada a casos de uso OLTP (*OnLine Transaction Processing*), ya que en estos suele ser más relevante la escritura de grandes volúmenes de datos, mientras las consultas son más ligeras y restringidas. No obstante, hay que tener en cuenta que la ingesta de datos puede verse limitada al encontrarse Spark como intermediario entre Kafka y Cassandra.
- **Consultas por ID**: Si la *query* o consulta está perfectamente definida con anterioridad y se conocen los campos de interés para las cláusulas *WHERE* (siendo estas de una columna a ser posible), Cassandra ofrece una mejor optimización que Druid. No obstante, las consultas se ven más limitadas para conservar su eficiencia según se amplíen los criterios de búsqueda, para lo cual Druid da más facilidades.
- **Replicación integrada y tolerancia a fallos**: Siendo este uno de los pilares principales del diseño de la arquitectura de Cassandra, mientras que Druid depende de segundas tecnologías para permitir la replicación.
- **Permite variedad de aplicaciones**: Mientras que los objetivos principales de Druid se enfocan hacia la consulta analítica, Cassandra resulta más flexible gracias al modelo de familia de columnas, en contraposición con el columnar. No obstante, como se ha mencionado previamente, deben conocerse con exactitud las consultas a realizar y diseñar las tablas en consecuencia. Realizar esto con respecto a grandes volúmenes de datos donde se busca variedad de consultas, resulta complejo.

3.2. Apache Druid

Druid es una base de datos distribuida NoSQL, centrada en el análisis de datos en tiempo real o procesamiento analítico en línea (OLAP). Además, su diseño se centra en permitir una rápida ingesta de datos, así como en el almacenamiento de eventos, sobre la base de los cuales se pueden elaborar las

diversas particiones.

Entre otros de sus beneficios, se encuentra el almacenamiento de los datos en formato columnar en lugar del enfocado a filas por parte de Cassandra, lo cual permite agilizar las consultas y agregaciones, tomando únicamente en cuenta las columnas deseadas. Cabe destacar que la ingesta de datos, aparte de poder realizarse en tiempo real, también puede organizarse por lotes, permitiendo en ambos casos la inmediata consulta de la información almacenada.

Al igual que Cassandra, Druid también cuenta con replicación de datos entre los diversos servidores implicados, además de autobalanceos, con el fin de garantizar particiones de tamaños similares. Por último, gracias a su surgimiento en un momento donde la tecnología en la nube comenzó a cobrar importancia, cuenta con la ventaja, respecto a Cassandra y otras tecnologías, de permitir separar procesos para que estos se ejecuten sobre el hardware más apropiado.

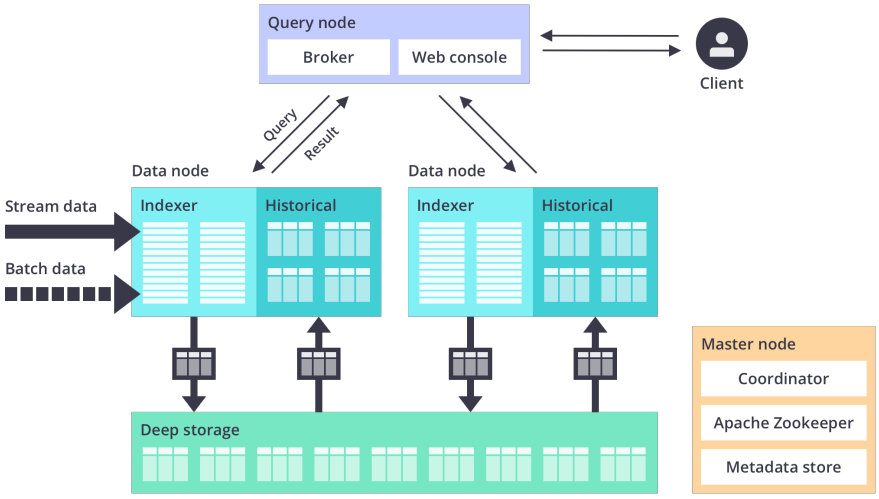

Figura 7: Arquitectura de Druid [3].

Cabe destacar que se hace uso de Zookeeper, al igual que con Kafka, para la coordinación de los diversos nodos. Por otra parte, se ofrece un *deep storage* que radica en un almacén que contiene copias de los diversos datos ingeridos, como puede observarse en la Figura 7, ante la posibilidad de pérdidas de datos por la caída de nodos si estos no se encuentran replicados en más de un servidor.

Una vez comentadas las principales características, se ofrecen a continuación las ventajas que se obtendrían en su empleo con Kafka:

- **Velocidad de consulta**: Ofrece un rendimiento rápido, especialmente dentro del ámbito de las consultas analíticas, facilitando la realización de las mismas en entornos donde priman las operaciones en tiempo real.
- **Compresión de datos**: El almacenamiento de datos de forma columnar resulta más eficiente, más allá de las diversas opciones ofrecidas por Druid en términos de compresión y generación de particiones.
- **Mecanismo de ingesta nativo**: Druid cuenta por sí mismo con un mecanismo que únicamente requiere de la dirección de un *broker* de Kafka y el tópico de interés, para comenzar a realizar

la ingesta de los datos almacenados en dicho tópico, sin necesitar contar con un programa que conecte ambas tecnologías o se encuentre encargado de construir *queries* para insertar datos, como sí se ha necesitado hacer para Cassandra. En cambio, Druid solo necesita definir el patrón a seguir para la ingesta, en lo que se conoce como *spec*. Pueden observarse dos ejemplos en los anexos B y C.

- **Diseño enfocado a IoT, métricas e integración con herramientas**: El diseño de Druid está principalmente orientado al uso de métricas temporales (uso de *timestamps*), de las cuales justo se hace uso en el caso planteado. Además, también se encuentra pensado para su conexión con otras herramientas, como las especializadas en visualización de datos como Grafana, que es la empleada en este TFG.

De esta forma, puede concluirse que Druid es especialmente útil para almacenar los datos incorporados en los tópicos correspondientes con datos que ya han sido procesados, debido a la importancia de realizar consultas en tiempo real sobre estos. Por otro lado, Cassandra puede ser de utilidad para mantener una copia persistente de los datos en crudo que la aplicación Spark recibe de forma continuada, poniendo así ambas alternativas en práctica en sus mejores campos de utilidad.

3.3. Comparativa de rendimiento

Con el fin de verificar las conclusiones previas realizadas sobre la base de razonamientos teóricos, se elaboró un breve estudio de rendimiento de la ingesta de datos por parte de Druid y por parte de Cassandra, requiriendo esta última alternativa el uso de un programa con Spark, capaz de consumir de Kafka y enviar datos a Cassandra. Para ello, se ha utilizado el segundo conjunto de datos de un juego web. Al consistir en un fichero JSON donde algunos atributos son compuestos, se requiere el uso de dimensiones específicas en Druid, como se puede observar en el *spec* del anexo B o el uso de tipos de datos personalizados en Cassandra (anexo D), presentando el diagrama UML de la Figura 8:

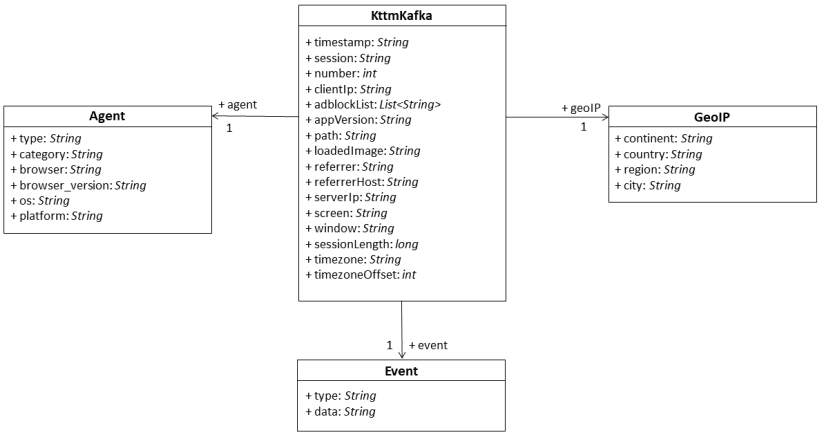

Figura 8: Modelo de datos del *dataset* de juego web.

El patrón seguido para la elaboración del programa Spark, es similar al expuesto en la sección de calidad, aunque en este caso se añade el uso de la librería Jackson, con el fin de poder mapear sencillamente cada fila del fichero JSON a objetos Java a través del uso de anotaciones. Esto no es necesario para Druid, pues se encuentra preparado de forma nativa para ingerir datos en diversos formatos, entre los que se encuentran CSV y JSON. De esta forma, con este conjunto de datos alternativo y más ligero,

con un peso aproximado de 360 MB, se puede elaborar un test de rendimiento más completo que pone en juego particularidades también de los diferentes gestores de bases de datos, como es en esta ocasión el uso de tipos de datos compuestos.

Para visualizar también la posible diferencia en tiempos según aumenta el volumen de datos, se han realizado pruebas en 4 tramos: 25, 50, 75 y 100 % del contenido del *dataset*. Para dividir el conjunto de datos, se ha utilizado el comando *split* de Linux en función del número de líneas totales (una por cada fila de datos), si bien son varias las alternativas posibles. En el caso de los tiempos, estos se han obtenido observando los ficheros de *log* de Druid y en el caso de Cassandra, midiendo dentro del propio programa en Java.

Porcentaje / Tecnología	25 %	50 %	75 %	100 %
Druid	4 s	8 s	11 s	13 s
Cassandra	41 s	69 s	98 s	133 s

Cuadro 2: Tiempo de ingesta en segundos del *dataset* de juego web.

Puede observarse en el Cuadro 2 cómo el uso necesario de Spark como intermediario, también, afecta negativamente en parte a los tiempos de ingesta de Cassandra. Para poder considerarlo una alternativa viable, sería necesario paralelizar el tratamiento del conjunto de datos y, a ser posible, desplegar varios nodos de Spark operando dicho conjunto.

Se presenta además la gráfica de la Figura 9 que recoge los valores indicados anteriormente, con el fin de poder observar cómo se incrementa la diferencia de rendimiento según aumenta el volumen de datos procesados, además de mostrar una evolución prácticamente lineal para ambos casos, si bien la pendiente para Cassandra revela una penalización mayor.

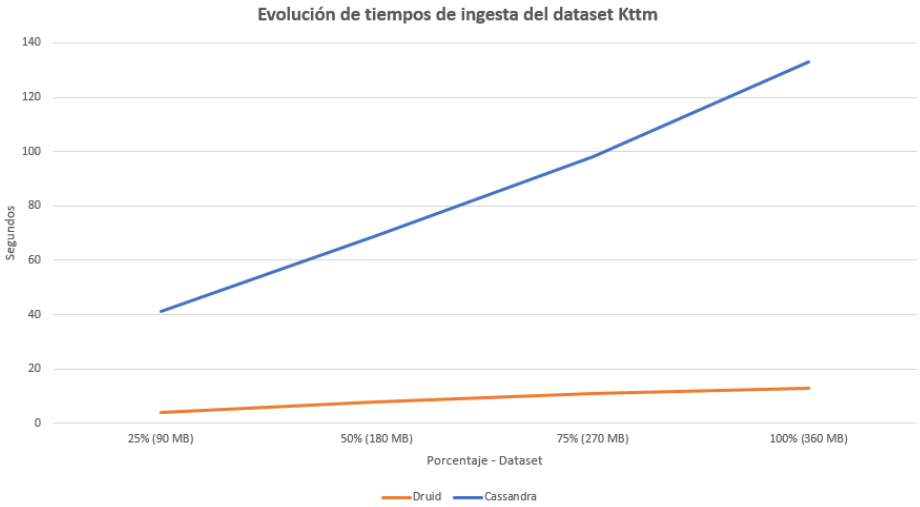

Figura 9: Gráfica de tiempos de ingesta del *dataset* de juego web.

Basándose en estos resultados obtenidos y las argumentaciones teóricas elaboradas con anterioridad, se define, por lo tanto, como una alternativa más viable utilizar Druid sobre Cassandra, al menos de cara a los datos que vayan a ser consultados próximamente para la creación de cuadros de mando.

Mientras tanto, Cassandra puede resultar útil para almacenar la copia de los datos en crudo, que a la larga serán mucho menos consultados, salvo posibles imprevistos. Pueden consultarse los esquemas de datos empleados para los casos de uso puestos en práctica, en los anexos D y E.

4. Seguridad en el acceso al dato

Una vez establecida Kafka como la plataforma para la gestión de eventos y transmisión de datos, resulta necesario garantizar también la seguridad del acceso sobre estos. Para ello, es imprescindible controlar qué sujetos (*principals* para Kafka) acceden a la plataforma, así como qué conjuntos de datos pueden ser capaces de obtener o proveer. Es decir, deben aplicarse los procesos necesarios para llevar a cabo la autenticación de usuarios, así como la autorización de los mismos para su acceso a los tópicos de interés, delimitando asimismo las operaciones que pueden realizar, siguiendo para ello el esquema mostrado en la Figura 10.

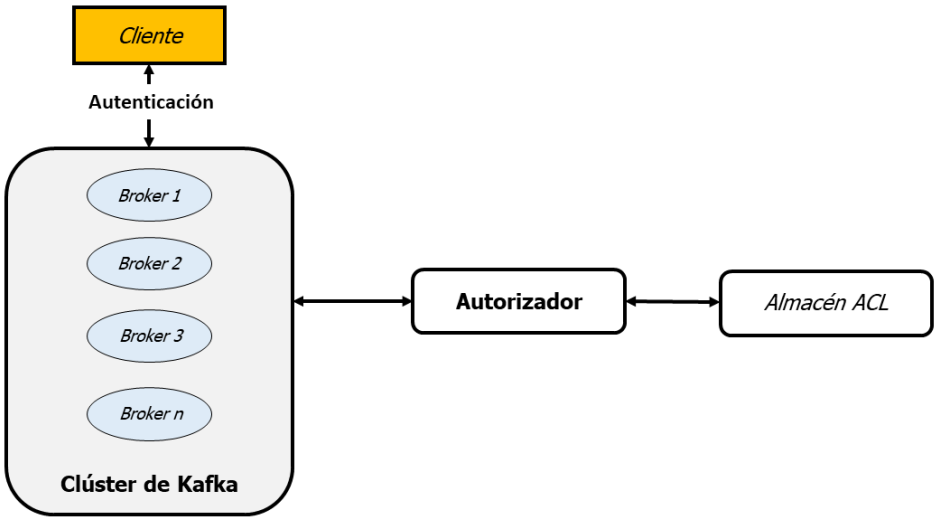

Figura 10: Esquema global del uso de autenticación y autorización cliente-servidor.

4.1. Autenticación

Con el fin de controlar el proceso de autenticación, se ha recurrido a la generación de certificados SSL/TLS a través de Keytool, siendo el uso de certificados uno de los métodos más empleados en la actualidad y con un soporte sólido por parte de Kafka y el resto de tecnologías que requieren establecer una conexión con la plataforma. Además, su uso permite establecer una comunicación encriptada [19].

Durante las siguientes subsecciones se resume el proceso seguido para la generación de los certificados, explicando varios de sus parámetros y argumentando el porqué de sus valores, así como también las configuraciones necesarias para aplicar el uso de los certificados en las tecnologías implicadas.

4.1.1. Generación de certificados

- Crear pareja de claves para la Autoridad de Certificación (CA):

```
keytool -genkeypair -keyalg RSA -keysize 2048 -keystore brokers.ca.p12 -storetype
    ↪PKCS12 -storepass brokers-ca-password -keypass brokers-ca-password -alias
    ↪BrokersCA -dname "CN=BrokersCA" -ext bc=ca:true -validity 365
```

Parámetros utilizados:

- **genkeypair**: Generación de un par de claves (pública y privada) para emplear encriptación asimétrica.
- **keyalg**: Determina el algoritmo utilizado para el cifrado en la generación de las claves. En este caso, se emplea RSA (Rivest, Shamir and Adleman) propuesto en 1978 y uno de los más populares a día de hoy, tanto por su uso extendido como la dificultad que supone romper su cifrado asimétrico [11]. Así mismo, otras alternativas como DSA y DES también pueden ser escogidas [17].
- **keysize**: Tamaño en bits de la clave. Se indica de forma explícita, si bien 2048 es el valor empleado por defecto a la hora de generar un par de claves con encriptación asimétrica mediante algoritmo RSA [17].
- **keystore**: Localización del *keystore*, en este caso, un fichero *.p12*, ampliamente extendido hoy en día para el almacenamiento de la información criptográfica generada.
- **storetype**: El tipo de *keystore* que se emplea, utilizando en este caso uno acorde con el fichero mencionado previamente. Existen una gran variedad de formas en la que la información criptográfica generada puede ser almacenada, tal y como se especifica por Oracle [18].
- **storepass y keypass**: Determinan la contraseña del *keystore* y la clave privada respectivamente, pudiendo existir, por lo tanto, para un *keystore* una *storepass* y tantas *keypass* como claves privadas de certificados a proteger en dicho *keystore*.
- **alias**: Nombre con el que se identificará al certificado en el *keystore*.
- **dname**: Nombre que debe de actuar como identificador único.
- **ext.**: Permite añadir extensiones X.509, estándar propio de los certificados basados en claves públicas. *bc*, indica concretamente que será una extensión básica [14]. En esta ocasión, se indica que el certificado pertenece a una autoridad de certificación, pues los certificados para este proyecto están firmados por nosotros mismos.
- **validity**: Marca la durabilidad del certificado antes de su expiración. En este caso, se ha elegido una duración de un año con el fin de no ser necesario renovar con frecuencia los certificados a lo largo de la elaboración del proyecto.

- Exportar el certificado de la Autoridad de Certificación con la clave pública para su uso posterior:

```
keytool -export -file brokers.ca.crt -keystore brokers.ca.p12 -storetype PKCS12 -
    ↪storepass brokers-ca-password -alias BrokersCA -rfc
```

El certificado queda exportado en el fichero *servidor.ca.crt* y en formato PEM, siendo esto último gracias al parámetro *rfc*. En caso de no indicarse, por defecto se exportaría como un binario de Java, lo cual limita su interpretación en caso de querer compartirse. Sin embargo, PEM utiliza codificación *base64*, lo que permite compartirlo como fichero de texto.

- Generar claves privadas para los *brokers* de Kafka a autenticar (si se emplean nombres basados en comodines, como '*', es posible emplear un solo *keystore* para todos los *brokers*, es decir, a nivel de clúster):

```
keytool -genkey -keyalg RSA -keysize 2048 -keystore mario.ks.p12 -storetype PKCS12 -
    ↪storepass mario-ks-password -keypass mario-ks-password -alias Mario -dname "CN=
    ↪mario, OU=Facultad de Ciencias, O=Universidad de Cantabria, C=ES" -validity 365
```

En este caso, se ha decidido incluir algo más de información que permita identificar a los diversos *brokers* con una mayor precisión. Se explica a continuación el significado de los campos empleados, siguiendo las indicaciones del manual de *Keytool* [17].

- **CN**: Nombre del dominio para el que se emite el certificado. Generalmente utilizado para identificación.
- **OU**: Unidad de la organización.
- **O**: Organización.
- **C**: País. Debe emplearse un código de dos caracteres, identificando a España en esta ocasión, de acuerdo con el estándar internacional de normalización ISO 3166-2.

- Generar las correspondientes peticiones de firma de certificados para cada *broker*:

```
keytool -certreq -file mario.csr -keystore mario.ks.p12 -storetype PKCS12 -storepass
    ↪mario-ks-password -keypass mario-ks-password -alias Mario
```

Es necesario para ello indicar dónde se han almacenado las claves privadas (*keystore*)

- Firmar los certificados de cada *broker* gracias al certificado para la Autoridad de Certificación creado con anterioridad:

```
keytool -gencert -infile mario.csr -outfile mario.crt -keystore ../BrokersCA/brokers.ca
    ↪.p12 -storetype PKCS12 -storepass brokers-ca-password -alias BrokersCA -validity
    ↪ 365 -rfc
```

Para esta firma, empleamos el *keystore* utilizado previamente por el CA, que contiene su certificado. Dentro de este almacén, se utilizará concretamente la clave privada perteneciente al alias señalado, por lo que estaremos realizando la firma con la clave privada de la Autoridad de Certificación.

Se emplea de nuevo el parámetro *rfc* para en el próximo paso contar con una cierta uniformidad en el formato de los certificados.

- Incorporar al *keystore* de cada *broker* la correspondiente cadena de certificados, de forma que pueda asegurarse que el certificado empleado por el *broker* es válido. Para ello, puede incluirse un certificado en el que se contiene toda la información, tal y como se especifica a continuación:

```
cat mario.crt brokers.ca.crt > mariochain.crt
```

De esta forma, queda volcada toda la información en un mismo fichero. En un caso real, donde se depende de Autoridades de Certificación existentes, la cadena de certificados suele contar con otros certificados intermedios.

Finalmente, se realiza la importación previamente mencionada:

```
keytool -importcert -file mariochain.crt -keystore mario.ks.p12 -storetype PKCS12 -
    ↪storepass mario-ks-password -keypass mario-ks-password -alias Mario -noprompt
```

Si el proceso se ha realizado correctamente, deberá mostrarse el siguiente mensaje:

"*Se ha instalado la respuesta del certificado en el almacén de claves.*"

Para comprobar los certificados que se encuentran almacenados en un *keystore*, se puede emplear también el siguiente comando:

```
keytool -list -keystore mario.ks.p12 -storepass mario-ks-password
```

En este caso, se mostraría únicamente el recién añadido al *keystore* en el paso previo:

```
mario, 26 abr. 2023, PrivateKeyEntry,
Huella de certificado (SHA-256): F2:CC:3C:50:BA:68:7A:EA:2A:30:6B:3E:47:7B:0A:58:3D:2B
    ↪:71:61:05:70:81:92:77:B2:5E:2C:C9:01:4A:13
```

Es importante tener en cuenta que el empleo de SSL conlleva un *overhead* que no puede pasarse por alto, más aún en sistemas de tiempo real, donde procesar los datos con rapidez es de considerable importancia. De hecho, el *overhead* generado de cara al procesamiento por la CPU, puede rondar en valores de entre un 20 y un 30 % [24].

Por ello, si se considera la red interna (entre *brokers*) como segura, su uso puede evitarse, lo cual será la aproximación tomada en la elaboración de este proyecto, donde la autenticación estará limitada a la comunicación de clientes externos con los *brokers* de Kafka.

En el hipotético caso de tener interés en asegurar la comunicación entre *brokers* dentro de Kafka, estos deberían contar con los correspondientes almacenes de certificados que indiquen en qué *brokers* pueden confiar (*truststores*), permitiendo así autenticarse entre sí.

Una vez finalizada la preparación de certificados y almacenes de los mismos para los *brokers* de Kafka, se definen a continuación los pasos a seguir para permitir la autenticación de los clientes externos a través de SSL/TLS. Varios de los pasos implican la ejecución de comandos similares a los empleados para los *brokers*, por lo que se omiten explicaciones más extendidas, especialmente en cuanto a los parámetros empleados se refiere.

- Crear un *truststore* para los clientes, de forma que puedan verificar la autenticidad de los certificados de los *brokers*:

```
keytool -import -file brokers.ca.crt -keystore clientes.ts.p12 -storetype PKCS12 -
    ↪storepass clientes-ts-password -alias BrokersCA -noprompt
```

  Como puede observarse, se utiliza para ello el certificado utilizado previamente para firmar los correspondientes certificados de los *brokers*, el cual sería la raíz de la cadena de certificados, garantizando su validez.

  Empleando el comando mencionado previamente para listar los certificados almacenados, obtenemos el siguiente resultado si todo ha sido llevado a cabo correctamente:

```
brokersca, 18 abr. 2023, trustedCertEntry,
Huella de certificado (SHA-256): 3A:1B:85:18:D8:5C:7B:E6:21:01:1D:6E:FF:94:B4:CE:4A:B1:
    ↪E0:24:66:94:3B:89:DF:41:9B:50:54:69:F1:41
```

- Generar una nueva Autoridad de Certificación, con un certificado firmado por nosotros mismos.
  - Crear la pareja de claves pública-privada.

```
keytool -genkeypair -keyalg RSA -keysize 2048 -keystore clientes.ca.p12 -storetype
    ↪ PKCS12 -storepass clientes-ca-password -keypass clientes-ca-password -alias
    ↪ ClientesCA -dname CN=ClientesCA -ext bc=ca:true -validity 365
```

- Exportar el certificado del nuevo CA.

```
keytool -export -file clientes.ca.crt -keystore clientes.ca.p12 -storetype PKCS12
    ↪-storepass clientes-ca-password -alias ClientesCA -rfc
```

- Crear nuevos almacenes de claves (*keystores*) para los clientes, donde se repetirán los mismos pasos seguidos previamente para los *brokers*. El *truststore* creado previamente, puede ser compartido por los clientes.

  - Generación de claves privadas para los clientes a autenticar. En este proyecto se utiliza, entre otros, un programa con Spark, encargado de consumir de Kafka y producir hacia otros tópicos, por lo que un ejemplo sería el siguiente:

```
keytool -genkey -keyalg RSA -keysize 2048 -keystore sparkcliente.ks.p12 -storetype
    ↪ PKCS12 -storepass sparkcliente-ks-password -keypass sparkcliente-ks-
    ↪password -alias SparkCliente -dname "CN=Spark, OU=Facultad de Ciencias, O=
    ↪Universidad de Cantabria, C=ES" -validity 365
```

  - Crear las solicitudes pertinentes de firma de certificados.
  - Firmar las solicitudes con el certificado del nuevo CA.
  - Incorporar al *keystore* la correspondiente cadena de certificados para asegurar su validez.

4.1.2. Configuración de los *brokers*

Por defecto, Kafka hace uso de un protocolo *PLAINTEXT*, es decir, se lleva a cabo una comunicación sin autenticación y sin cifrado con y entre los *brokers*. Dado que, como se ha mencionado previamente, la comunicación entre *brokers* se considera dentro de una red segura, este protocolo seguirá siendo mantenido para dicha intercomunicación. No obstante, será necesario modificar la configuración de Kafka, para añadir el uso del protocolo SSL que emplee los certificados para clientes externos.

Para ello, deben llevarse a cabo un conjunto de modificaciones en el fichero *server.properties*, que contiene las propiedades de los *brokers*, siguiendo concretamente un formato sencillo *propiedad* = *valor*. Dado que el desarrollo se está realizando en un equipo de forma local, se mostrarán ejemplos para un *broker*.

En el caso de desear conservar la configuración original de Kafka, también puede realizarse una copia del fichero y modificarla, ya que a la hora de iniciar los *brokers*, se indicará como parámetro concretamente el fichero que contiene sus propiedades.

```
######### Authentication #########
## SERVER - BROKER1
listeners=PLAINTEXT://:9092, SSL://9093
ssl.keystore.location=/home/mario/Escritorio/TFG/CertificadosKafka/mario/mario.ks.p12
ssl.keystore.password=mario-ks-password
ssl.key.password=mario-ks-password
ssl.keystore.type=PKCS12
ssl.truststore.location=/home/mario/Escritorio/TFG/CertificadosKafka/mario/mario.ts.p12
ssl.truststore.password=mario-ts-password
ssl.truststore.type=PKCS12
ssl.client.auth=required
```

Agregando las líneas de configuración expuestas, se logra lo siguiente:

- Habilitar un nuevo puerto (9093) que permita la comunicación mediante protocolo SSL. El puerto 9092 es el empleado por Kafka por defecto mediante protocolo *PLAINTEXT*. En un despliegue real donde requiramos autenticación entre todos los elementos, lo más óptimo sería utilizar únicamente el puerto 9093.
- Señalar el almacén de claves donde se encuentra el certificado previamente creado para el *broker*, así como las correspondientes contraseñas para poder acceder tanto al almacén como la propia clave.
- Indicar el tipo de almacén utilizado. Como se ha indicado con anterioridad, se ha trabajado con un formato PKCS12.
- Dado que los *brokers* deben de verificar la identidad de los clientes conectados, se especifica también el almacén donde encontrar aquellos clientes en los que confiar, así como de nuevo la respectiva contraseña de acceso a dicho almacén. Cabe destacar que estos ficheros deberían encontrarse en localizaciones protegidas.
- Establecer como necesaria la autenticación de los clientes para poder realizar una conexión.

4.1.3. Configuración de los clientes de consola

De igual forma que para los *brokers*, es necesario contar con un fichero de configuración que contenga las propiedades necesarias para habilitar el protocolo SSL para los clientes. Estas serán prácticamente las mismas que las utilizadas para los *brokers*, especificando en esta ocasión el almacén de claves propio del cliente, así como dónde se encuentran los certificados en los que puede confiar para establecer una conexión.

Puede encontrarse una configuración completa en el anexo F, pudiéndose utilizar por ejemplo esta configuración para ejecutar comandos por terminal hacia el puerto 9093.

4.1.4. Seguridad en conexión con Prometheus

Como ya se ha mencionado en capítulos previos, Prometheus será el software de monitorización utilizado para exportar información del rendimiento de Kafka.

Para proveer los datos a este servicio, se utiliza un *exporter* que obtiene la información de interés, a la cual Prometheus puede acceder y que servirá de fuente para generar las gráficas en Grafana. Dado que el *exporter* se habilita en el momento del arranque de los *brokers*, como se explicará en la próxima sección, no es necesario llevar a cabo acciones adicionales para autenticar Prometheus, ya que depende exclusivamente de los datos remitidos en un servidor por el *exporter*, sin requerir una conexión directa a Kafka. No obstante, en otros casos donde se requiera dicha conexión, Prometheus permite también una configuración TLS [22].

Sin embargo, hacer uso de otras tecnologías como Spark o Druid, sí requiere de una conexión directa a Kafka como consumidor, exponiéndose en las siguientes subsecciones la configuración necesaria para poder realizar la autenticación.

4.1.5. Autenticación en conexión con cliente Java

Ya sea a la hora de programar un productor o un consumidor, es necesario indicar una serie de propiedades añadidas, que especifiquen el protocolo a utilizar, así como el *keystore* y *truststore* correspondientes. Por ejemplo, en el caso de utilizar un mapa en Java, podría realizarse el procedimiento de la siguiente forma mediante pares clave-valor:

```
kafkaConf.put("security.protocol", "SSL");
kafkaConf.put("ssl.keystore.location", "/home/mario/Escritorio/TFG/CertificadosKafka/" + "
    ↪Clientes/Spark-Cassandra/sparkcliente.ks.p12");
kafkaConf.put("ssl.keystore.password", "sparkcliente-ks-password");
kafkaConf.put("ssl.key.password", "sparkcliente-ks-password");
kafkaConf.put("ssl.truststore.location", "/home/mario/Escritorio/TFG/CertificadosKafka/" + "
    ↪Clientes/clientes.ts.p12");
kafkaConf.put("ssl.truststore.password", "clientes-ts-password");
```

En los anexos G y H se puede seguir la creación de cada uno de los certificados.

4.1.6. Autenticación en conexión con Druid

Se siguen las mismas propiedades que las vistas en el caso anterior, pues al fin y al cabo Druid actúa como un consumidor, aunque en este caso se modifican las propiedades del mismo en el fichero *spec* definido para la ingesta de datos. Para ello, se deben añadir las siguientes líneas:

```
"consumerProperties": {
    "bootstrap.servers": "mario:9093",
    "ssl.keystore.location": "/home/mario/Escritorio/TFG/CertificadosKafka/Clientes/Druid/
        ↪druidcliente.ks.p12",
    "ssl.keystore.password": "druidcliente-ks-password",
    "ssl.key.password": "druidcliente-ks-password",
    "security.protocol": "SSL",
    "ssl.truststore.location": "/home/mario/Escritorio/TFG/CertificadosKafka/Clientes/
        ↪clientes.ts.p12",
    "ssl.truststore.password": "clientes-ts-password"
}
```

Puede consultarse la creación del certificado en el anexo I.

4.2. Autorización

Podemos entender la autorización como el conjunto de pasos o procesos necesarios para comprobar si un usuario o elemento de otro tipo dispone de los privilegios o permisos suficientes para poder acceder a un recurso y/o realizar acciones sobre él.

Para esto, son múltiples las opciones usadas en el mundo de la informática, como gestiones a través de roles, listas de control de acceso, comprobación de tokens de acceso... Una de las alternativas más consolidadas en Kafka es precisamente el uso de listas de control de acceso (ACL), las cuales han sido empleadas en esta ocasión para garantizar el proceso de autorización, creando diversas reglas vinculadas a los certificados establecidos para la autenticación.

Por lista de control de acceso, se entiende un mecanismo consistente en una lista de entradas, las cuales definen una serie de reglas que pueden ser aplicadas a usuarios en concreto o roles. Por norma general, a la hora de elaborar una entrada de la lista, se deben tener en cuenta los siguientes elementos:

- **Sujetos**: Entidades a las que afecta la regla, desde usuarios individuales a determinados roles.
- **Permisos**: Acciones que se permiten a unos sujetos sobre unos recursos determinados. Algunas de las más comunes son, por ejemplo: lectura, escritura, creación y eliminación de tópicos.
- **Recursos**: Elementos que se pretenden restringir o proteger a través de las reglas de acceso. En un caso a nivel general, podría tratarse de directorios o diversos ficheros. En este caso concreto, se trabaja a nivel de tópico.
- **Localización**: El recurso puede encontrarse de forma local o requerir el acceso a servidores o equipos externos.

En el caso de Kafka y siguiendo el manual elaborado por Confluent [24], el formato para definir las reglas es similar a este planteamiento general realizado. Se incluye a continuación el Cuadro 3 con los elementos que se utilizan:

Recurso	Patrón	Nombre del recurso	Operación	Permiso	Sujeto	Host
Tópico	Literal	lecturas lecturas_anomalas_alert lecturas_gps_alert lecturas_filtradas	Read Write Create* Delete*	Allow Deny	Spark Druid Productor	localhost

Cuadro 3: Campos de una regla de ACL en Kafka.

Concretamente, *Patrón* hace alusión a la posibilidad de poder indicar el nombre explícito del tópico (*Literal*) o utilizar patrones basados en prefijos (*Prefixed*). En un entorno donde el número de tópicos es amplio, esta última opción resulta de mayor interés, al permitir agrupar varias reglas, garantizando mayor eficiencia y ahorrando memoria en Zookeeper, donde se almacenan, como se muestra en la Figura 11.

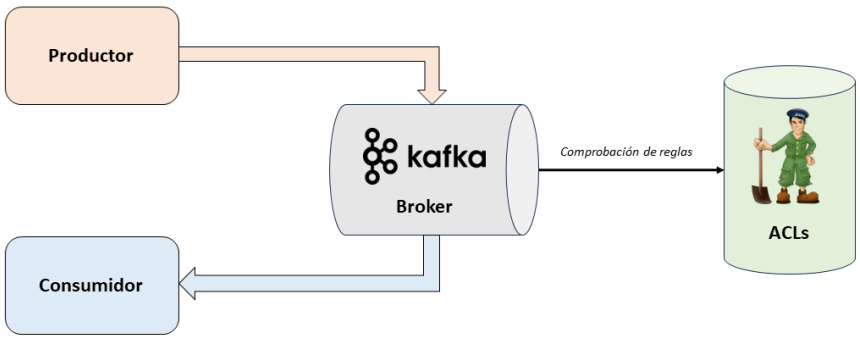

Figura 11: Esquema de autorización mediante ACLs.

Dado que la protección se realiza para los clientes externos que solo producen y consumen, las operaciones de creación o eliminación de tópicos no cobran mucho sentido (si bien estas sí que deben de estar permitidas para los administradores). En el caso de Druid puede darse permiso de creación por si el tópico no existiera.

Sobre la base de las necesidades de cada productor o consumidor, se han elaborado las reglas recogidas en el Cuadro 4, utilizando como sujeto el *Common Name* (CN) de los certificados SSL/TLS.

Para realizar el mapeo de CN a sujeto, es posible definir una propiedad en la configuración de Kafka a partir de expresiones regulares:

```
ssl.principal.mapping.rules=RULE:^.*[Cc][Nn]=([a-zA-Z0-9.]*).*$/$1/L,DEFAULT
```

Recurso	Patrón	Nombre del recurso	Operación	Permiso	Sujeto	Host
Tópico	Literal	lecturas	Read	Allow	Spark	localhost
Tópico	Literal	lecturas	Write	Allow	Productor	localhost
Tópico	Literal	lecturas_anomalas_alert	Read	Allow	Druid	localhost
Tópico	Literal	lecturas_anomalas_alert	Write	Allow	Spark	localhost
Tópico	Literal	lecturas_gps_alert	Read	Allow	Druid	localhost
Tópico	Literal	lecturas_gps_alert	Write	Allow	Spark	localhost
Tópico	Literal	lecturas_filtradas	Read	Allow	Druid	localhost
Tópico	Literal	lecturas_filtradas	Write	Allow	Spark	localhost

Cuadro 4: Reglas de autorización definidas.

De esta forma, se ejecutan los siguientes comandos para escribir las reglas en el autorizador de Kafka:

- **lecturas**:
  - Permite escrituras por parte del productor que lee el fichero CSV.

```
bin/kafka-acls.sh --bootstrap-server mario:9093 --add --allow-principal User:
    ↪productor --allow-host 127.0.0.1 --operation Write --topic lecturas --
    ↪command-config config/mario.properties
```

  - Permite lecturas por parte del programa Spark encargado de asegurar la calidad de los datos.

```
bin/kafka-acls.sh --bootstrap-server mario:9093 --add --allow-principal User:spark
    ↪ --allow-host 127.0.0.1 --operation Read --topic lecturas --command-config
    ↪config/mario.properties
```

- **lecturas_anomalas_alert**:
  - Permite escrituras por parte del programa Spark.

```
bin/kafka-acls.sh --bootstrap-server mario:9093 --add --allow-principal User:spark
    ↪ --allow-host 127.0.0.1 --operation Write --topic lecturas_anomalas_alert --
    ↪command-config config/mario.properties
```

  - Permite lecturas por parte de Druid, para realizar la ingesta de datos.

```
bin/kafka-acls.sh --bootstrap-server mario:9093 --add --allow-principal User:druid
    ↪ --allow-host 127.0.0.1 --operation Read --topic lecturas_anomalas_alert --
    ↪command-config config/mario.properties
```

- **lecturas_gps_alert**:
  - Permite escrituras por parte del programa Spark.

```
bin/kafka-acls.sh --bootstrap-server mario:9093 --add --allow-principal User:spark
    ↪ --allow-host 127.0.0.1 --operation Write --topic lecturas_gps_alert --
    ↪command-config config/mario.properties
```

  - Permite lecturas por parte de Druid, para realizar la ingesta de datos.

```
bin/kafka-acls.sh --bootstrap-server mario:9093 --add --allow-principal User:druid
    ↪ --allow-host 127.0.0.1 --operation Read --topic lecturas_gps_alert --
    ↪command-config config/mario.properties
```

- **lecturas_filtradas**:
  - Permite escrituras por parte del programa Spark.

```
bin/kafka-acls.sh --bootstrap-server mario:9093 --add --allow-principal User:spark
    ↪ --allow-host 127.0.0.1 --operation Write --topic lecturas_filtradas --
    ↪command-config config/mario.properties
```

  - Permite lecturas por parte de Druid, para realizar la ingesta de datos.

```
bin/kafka-acls.sh --bootstrap-server mario:9093 --add --allow-principal User:druid
    ↪ --allow-host 127.0.0.1 --operation Read --topic lecturas_filtradas --
    ↪command-config config/mario.properties
```

La ejecución de estos comandos, permite que Kafka agregue las reglas en Zookeeper. Concretamente, los ejemplos propuestos se han realizado después de activar la autenticación SSL, lo cual requiere el uso del puerto 9093 y adjuntar una configuración con la que se debe ejecutar el comando. Esta configuración consiste en indicar las ubicaciones del certificado empleado, de forma que pueda ser autenticado.

Si además, esto se realiza después de habilitar el autorizador, es recomendable que el sujeto posea permisos de superusuario. Ambas acciones pueden ajustarse de la siguiente forma en el fichero con la configuración de Kafka:

```
authorizer.class.name=kafka.security.authorizer.AclAuthorizer
super.users=User:admin
```

Finalmente, cabe destacar que a la hora de permitir la lectura de un tópico, también es necesario permitir el acceso a la acción *Describe* en el respectivo tópico. Esto es así, ya que es la acción encargada de permitir el acceso a los *offsets* utilizados para conocer el progreso de la lectura de datos en el tópico.

5. Monitorización del rendimiento

Uno de los aspectos fundamentales de este proyecto, además del procesado de los datos, consiste en la monitorización, no solo de estos últimos, sino también del rendimiento de la plataforma encargada de su distribución, Kafka. Si bien su interés de cara a los procesos de negocio y la visualización de datos puede no resultar relevante, la posterior construcción de cuadros de mando de monitorización de rendimiento puede facilitar al administrador del sistema la supervisión del mismo, detectar posibles caídas o plantear mejoras si el rendimiento observado no es el óptimo.

Para poder llevar a cabo esto, es necesario recurrir a dos tecnologías, concentrándose esta sección en la siguiente parte del despliegue (ver Figura 12):

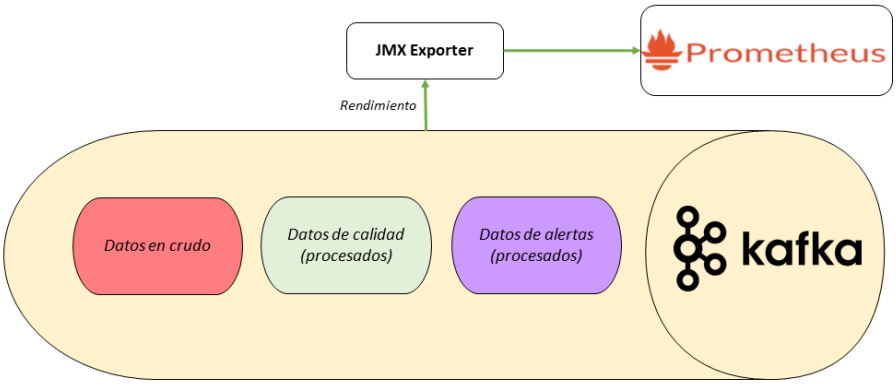

Figura 12: Componentes implicados en la monitorización del rendimiento.

- **JMX Exporter**: Herramienta puente entre Kafka y Prometheus encargada de obtener métricas de Kafka a través del protocolo JMX.
- **Prometheus**: Sistema de monitorización empleado para la adquisición y almacenamiento de métricas obtenidas a través de una ruta HTTP especificada, para su posterior uso en la generación de *dashboards*.

5.1. Configuración de JMX Exporter

El *exporter* utilizado consiste en un ejecutable Java, soportado por Prometheus de forma oficial y accesible a través de GitHub [23] o Maven. Este programa, utiliza el protocolo Java Management Extensions (JMX), estándar de Java para la monitorización de recursos de aplicaciones, los cuales son descritos mediante el uso de interfaces, y se conocen como MBeans [13].

En un primero momento, puede resultar complejo conocer todos los recursos que expondrá la interfaz de Kafka, así como sus diferentes propiedades. Para facilitar este proceso, resulta útil la herramienta de monitoreo JConsole, destinada a aplicaciones Java. Además de ofrecer algunas métricas de rendimiento, es capaz también de mostrar todos los MBeans que Kafka hace accesibles para su obtención desde el *exporter* (ver Figura 13):

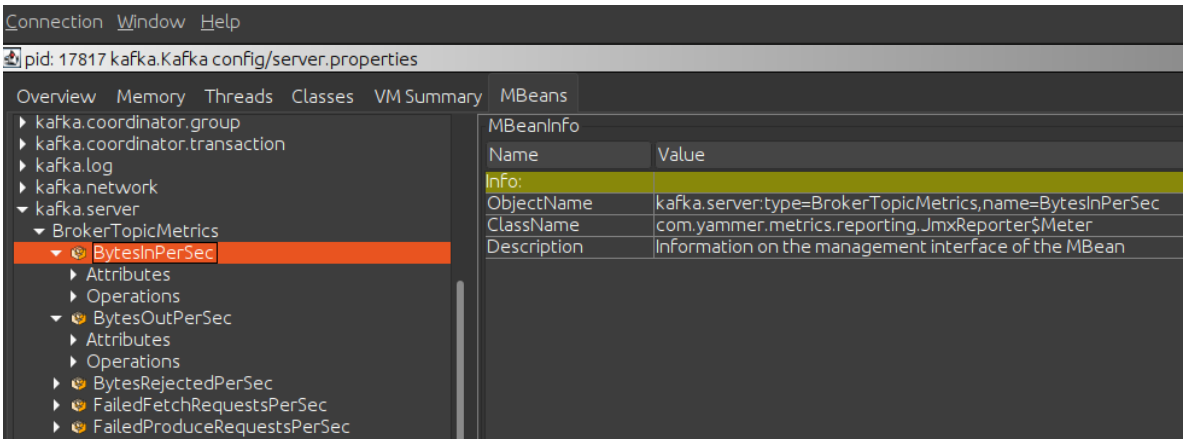

Figura 13: Visualización de MBeans a través de JConsole.

Una vez conocidas las métricas que pueden obtenerse, seleccionamos las de mayor interés, que son la media de bytes entrantes y salientes de cada tópico. En el caso de los datos entrantes, dado que los envíos se hacen mediante mensajes de forma programática, también es posible obtener una media del número de mensajes por segundo que entran en un tópico. Esto no aplica de la misma forma para la salida o lectura, pues el conjunto de mensajes retornado puede estar comprimido y no conocerse su número exacto.

Con el fin de determinar qué datos concretamente se quieren hacer públicos a Prometheus, es necesario crear un fichero YAML que contenga el conjunto de reglas que determinen las métricas a obtener. Cada regla vendrá principalmente marcada por los siguientes campos:

- **pattern**: Debe contener el nombre completo del MBean de interés.
- **name**: Nombre asociado a la métrica.
- **help**: Pequeño mensaje descriptivo relativo a la métrica.
- **type**: Tipo de métrica recogida. Generalmente, serán de tipo *GAUGE*, lo cual significa que representan un valor determinado en un momento dado.
- **labels**: Etiquetas que pueden asociarse a la métrica. Para este desarrollo, se ha adjuntado el nombre del tópico implicado, de forma que la identificación de la métrica sea más sencilla.

Y podrán agruparse en tres grupos:

- **Bytes entrantes por segundo**: Se obtiene una media de bytes entrantes por segundo para cada tópico. Para calcular la media, Kafka ofrece varios intervalos. Se ha escogido el de 1 minuto (el más breve), para obtener datos más precisos.
- **Mensajes entrantes por segundo**: Se obtiene una media de mensajes entrantes por segundo para cada tópico. De nuevo, se utiliza 1 minuto como intervalo.
- **Bytes salientes por segundo**: Se obtiene una media de bytes salientes por segundo para cada tópico. Una vez más, el intervalo de referencia es de 1 minuto.

Este tipo de reglas quedan aplicadas a los cuatro tópicos existentes. A continuación, se adjunta un ejemplo de cada grupo, con el fin de permitir observar cómo varía el campo *pattern* que se corresponde con el nombre completo del MBean de interés:

- **Bytes entrantes por segundo en *lecturas***:

```
- pattern : kafka.server<type=BrokerTopicMetrics, name=BytesInPerSec, topic=lecturas><>
    ↪OneMinuteRate
  name: LecturasBytesInPerSec_total
  help: Bytes entrantes por segundo del topico correspondiente a lecturas.
  labels:
    topic: "lecturas"
```

- **Mensajes entrantes por segundo en *lecturas_filtradas***:

```
- pattern: kafka.server<type=BrokerTopicMetrics, name=MessagesInPerSec, topic=
    ↪lecturas_filtradas><>OneMinuteRate
  name: LecturasFiltradasMessagesInPerSecond
  help: Mensajes entrantes por segundo del topico correspondiente a lecturas filtradas.
  type: GAUGE
  labels:
    topic: "lecturas_filtradas"
```

- **Bytes salientes por segundo en *lecturas_anomalas_alert***:

```
- pattern: kafka.server<type=BrokerTopicMetrics, name=BytesOutPerSec, topic=
    ↪lecturas_anomalas_alert><>OneMinuteRate
  name: LecturasAnomalasAlertBytesOutPerSecond
  help: Bytes salientes por segundo en el topico correspondiente a lecturas anomalas
      ↪que puedan generar alertas.
  type: GAUGE
  labels:
    topic: "lecturas_anomalas_alert"
```

Por último, para activar el *exporter* en el momento en el que se inicializa la instancia de Kafka, se puede modificar su *script* de arranque, añadiendo el uso de la variable de entorno *KAFKA_OPTS*, que añade nuevas opciones de configuración a la hora de ejecutar Kafka y, en este caso concreto, se permite la exposición de métricas a través del *exporter*:

```
export KAFKA_OPTS='-javaagent:/home/mario/Descargas/kafka/kafka_2.13-3.3.1/libs/
    ↪jmx_prometheus_javaagent-0.18.0.jar=7075:/home/mario/Descargas/kafka/kafka_2
    ↪.13-3.3.1/config/jmx_exporter.yml'
```

Concretamente, se indica el *exporter*, el puerto a utilizar y el fichero YAML que contiene las reglas con las métricas de interés a obtener. Por defecto, también se obtendrán algunas otras de interés que se mostrarán próximamente, como el uso de CPU o el consumo de memoria.

5.2. Configuración de Prometheus

Prometheus es el sistema utilizado para el almacenamiento de métricas de rendimiento obtenidas desde Kafka, no solo únicamente por su diseño orientado a la eficiencia con series temporales, sino también por el uso de algunas de sus funciones que permiten evaluar ciertos conjuntos de métricas a lo largo de un intervalo, algo que resultará de gran utilidad a la hora de elaborar cuadros de mando con Grafana.

Con el fin de explicar brevemente su funcionamiento, se muestra en la Figura 14 su arquitectura:

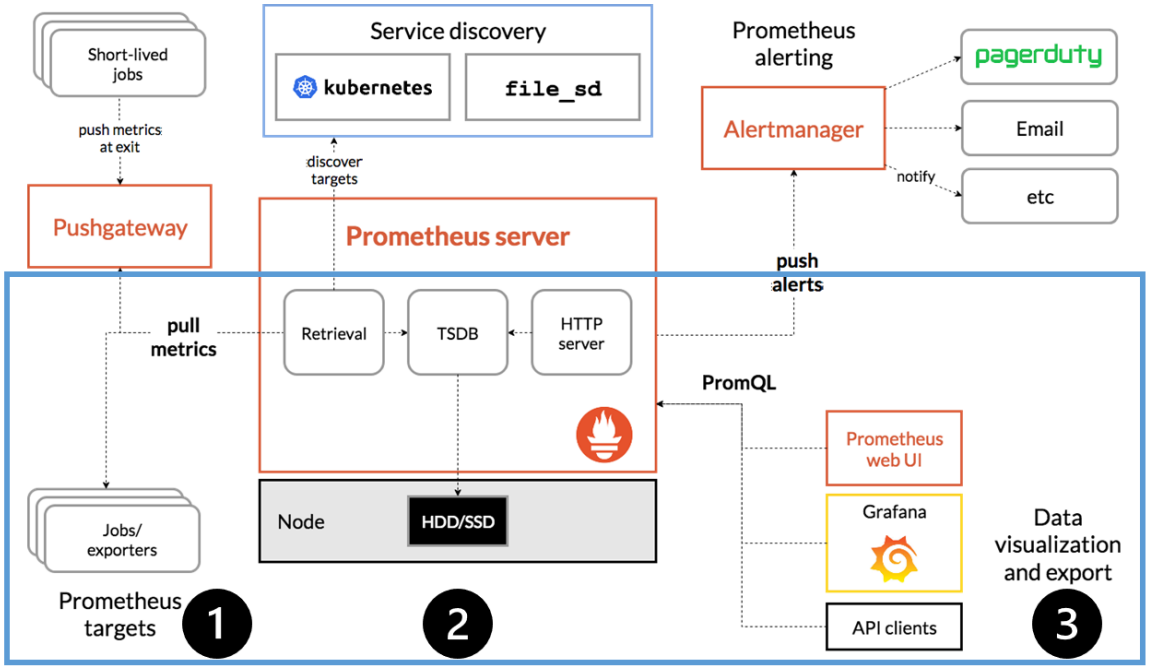

Figura 14: Arquitectura de Prometheus [21].

El estudio de esta herramienta se basa especialmente en la zona remarcada en azul, llevándose a cabo los siguientes pasos:

1. **Captura de métricas**: Se configura un *target* dentro de Prometheus, el cual se corresponde con el servidor HTTP donde el *exporter* aloja las métricas de Kafka. Para ello, es necesario modificar su fichero YAML de configuración:

```
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from
      ↪this config.
  - job_name: 'kafka'
    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.
    static_configs:
      - targets: ['localhost:7075']
```

2. **Almacenamiento**: Las métricas recogidas del servidor, son almacenadas en memoria por Prometheus, con opción también de persistencia en disco.

3. **Consulta desde Grafana**: Con el fin de elaborar los cuadros de mando, Prometheus será configurado en Grafana como una fuente de datos. Una vez hecho esto, puede accederse al mismo a través de PromQL, su lenguaje de consulta, y obtener o evaluar las métricas de interés, siendo este aspecto el que se tratará en la siguiente sección.

Cabe destacar que antes de realizar todos estos pasos, ha sido necesaria una revisión del *firewall*, con el fin de evitar posibles errores en las conexiones.

6. Visualización de datos y métricas

Una vez completado todo el desarrollo de las secciones previas, es posible comenzar a ver los resultados de los diversos programas, conexiones y configuraciones elaborados. El último paso consiste en la creación de cuadros de mando para poder visualizar los datos que se han obtenido o han sido procesados hasta el momento. Por una parte, será interesante analizar la información que se corresponde con el conjunto de datos, mientras que por otra se pretende mostrar información de utilidad al administrador del sistema.

Para ello, los elementos implicados en este último paso, serán los siguientes (ver Figura 15):

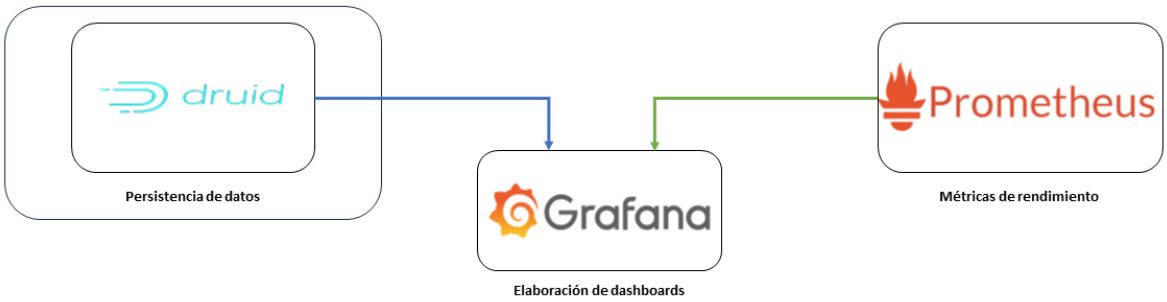

Figura 15: Componentes implicados en la generación de cuadros de mando.

Druid y Prometheus serán usados como fuentes de datos para la elaboración de los respectivos cuadros de mando, los cuales tendrán diferentes finalidades, como podrá comprobarse a lo largo de la sección, la cual está prácticamente centrada en el uso de Grafana.

6.1. Introducción a Grafana

Concretamente, Grafana es una plataforma de código abierto, la cual está enfocada a la creación de cuadros de mando y la visualización de datos y métricas. El hecho de que sea de código abierto, ha facilitado que exista una amplia colección de complementos que facilitan la conexión con un gran número de fuentes de datos, entre los que se encuentran las dos empleadas y, si se deseara, también otras alternativas populares como Cassandra, MongoDB o MySQL entre otras.

Además, son varias las alternativas que ofrece para trabajar, desde versiones en la nube a otras especializadas para empresas, si bien en este proyecto se ha utilizado la alternativa gratuita de código abierto, en la que se debe aportar la infraestructura para realizar el despliegue, utilizando en este caso el propio equipo personal. Para facilitar su accesibilidad, tanto Prometheus como Grafana se han configurado como servicios que pueden arrancarse en el momento deseado de forma manual a través de una terminal.

Finalmente, Grafana permite operar a través de CLI (por línea de comandos), lo cual es útil y de interés a la hora de querer importar posibles cuadros de mando a un servidor o controlar su configuración. A lo largo de esta sección, se mostrarán resultados a través de la otra alternativa de manejo, su interfaz gráfica, accesible por navegador a través del puerto 3000 designado en *localhost*. Al igual que pueden importarse cuadros de mando, estos también pueden ser exportados como ficheros JSON para su utilización por otros usuarios o equipos.

6.2. Establecimiento de fuentes de datos

Para la elaboración de los cuadros de mando, es necesario establecer las fuentes de datos sobre las que deben realizarse las consultas:

- **Druid**: Puede usarse como fuente de datos realizando la instalación de su complemento a través de las opciones de administración. Una vez hecho esto, para establecer una conexión válida, se debe establecer la URL donde se aloja el servidor, en este caso: *http://localhost:8888*.

  Otros parámetros, como los tiempos de espera o el número de reintentos, pueden configurarse. La conexión realizada para el proyecto, utiliza un valor por defecto de, como máximo, 5 reintentos de conexión.

- **Prometheus**: Su uso no requiere de la instalación de complementos externos, pues Grafana cuenta de forma oficial con una extensión que da soporte a la realización de conexiones con Prometheus. De igual forma, es necesario establecer la URL de conexión: *http://localhost:9090*.

  Además, debido al sólido soporte ofrecido por Grafana para esta extensión, la conexión podría asegurar autenticación por certificados, aunque no se ha contemplado, pues la dimensión de seguridad por limitaciones de extensión del proyecto, se encuentra enfocada en torno al aseguramiento de esta en el clúster de Kafka.

6.3. Cuadro de mando de monitorización de rendimiento

Con base en los datos que se han mencionado que se extraerían en la sección de monitorización, el cuadro de mando cuenta con 4 agrupaciones que reúnen diferente tipo de información y se actualizan cada 5 segundos, mediante consultas realizadas con PromQL, pudiéndose ver los valores concretos en caso de seleccionar los puntos correspondientes a las medidas.

6.3.1. Rendimiento global

Agrupa dos tipos de gráficas diferentes, como se puede observar en la Figura 16:

- **Consumo de CPU**: Se evalúa en intervalos de un minuto sobre las métricas que aporta Kafka señalando el consumo de CPU en segundos.
- **Consumo de memoria**: Memoria que consume la máquina virtual de Java empleada por Kafka.

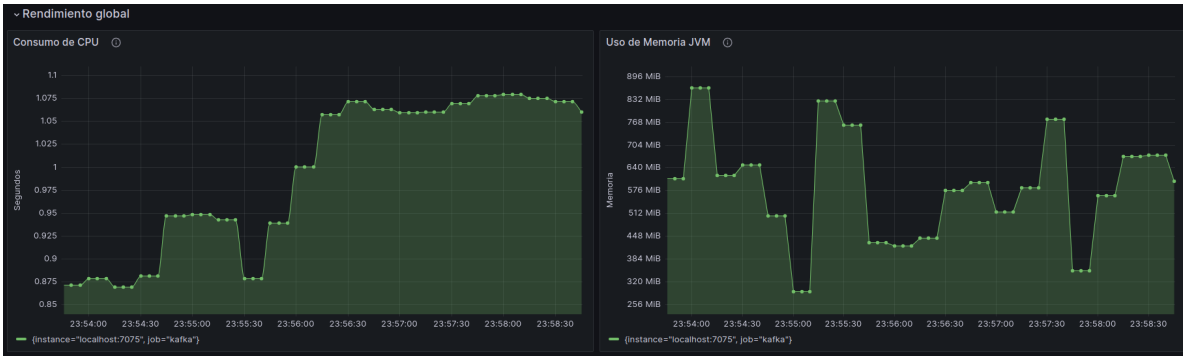

Figura 16: Gráficas de rendimiento global.

6.3.2. Tasa de escritura por tópico

Agrupa cuatro tipos de gráficas diferentes, una para cada tópico, registrando los bytes por segundo que son escritos en cada uno de ellos, es decir, la cantidad de datos "entrantes" en el tópico (ver Figura 17). Con el fin de seguir un esquema de colores fácilmente identificable y que sea común con otras tecnologías empleadas a día de hoy, se ha decidido hacer uso de un tono azul para simbolizar la entrada de datos.

Figura 17: Gráficas de escritura de datos.

6.3.3. Tasa de lectura por tópico

Sigue una distribución similar al caso anterior, pero registrando los bytes por segundo que son leídos de cada tópico, es decir, la cantidad de datos "salientes" (ver Figura 18). De igual forma que antes, para seguir un esquema de colores fácilmente identificable y común a otras tecnologías, se ha decidido hacer uso de un tono naranja para simbolizar los datos enviados o salientes.

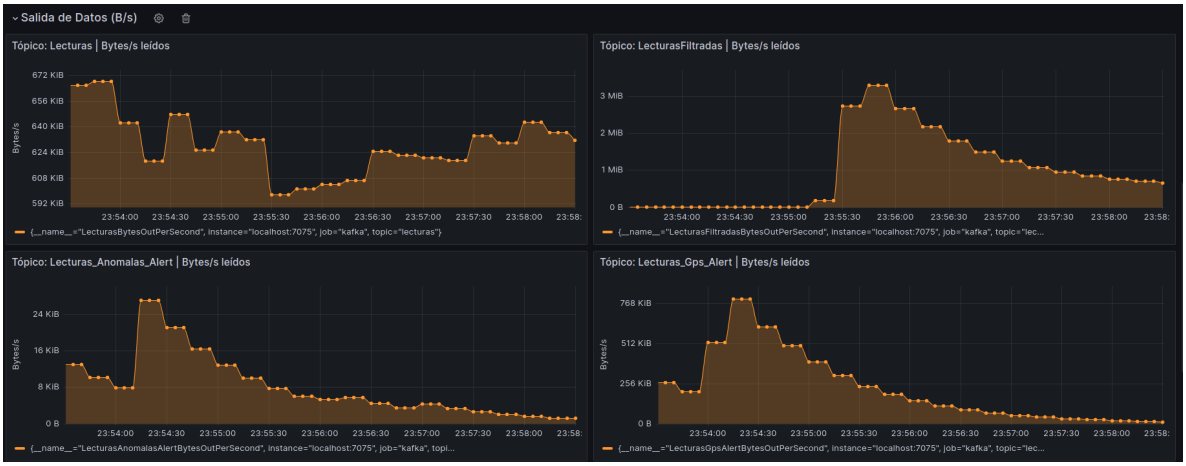

Figura 18: Gráficas de lectura de datos.

6.3.4. Tasa de escritura de mensajes por tópico

De forma complementaria a las gráficas que representan la escritura de datos en bytes por segundo, se han realizado gráficas análogas con el fin de mostrar la misma información, aunque teniendo en cuenta el número de mensajes enviados (ver Figura 19), en lugar de los bytes.

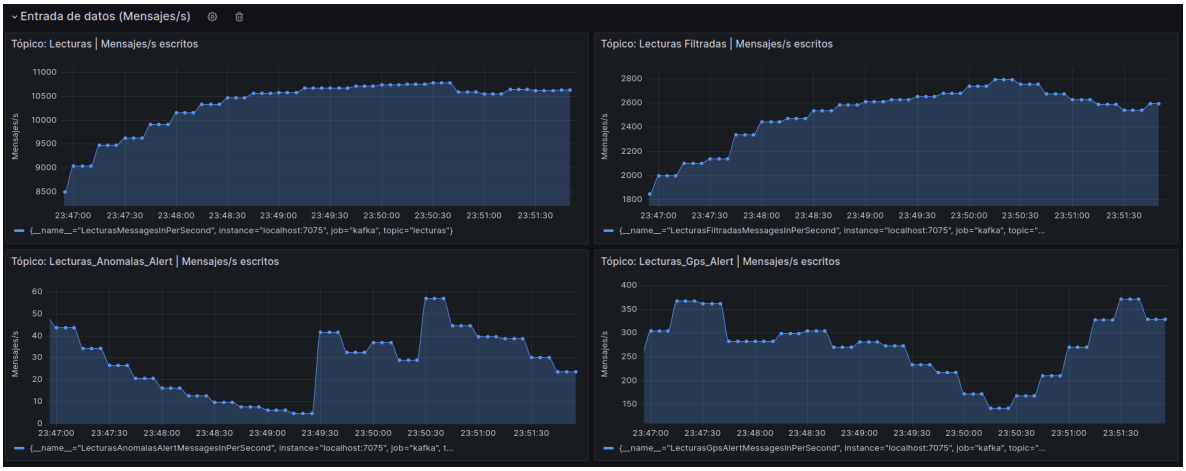

Figura 19: Gráficas de escritura de datos (en mensajes).

6.4. Cuadro de mando de visualización de datos

Partiendo de los datos almacenados en Druid, gracias al procesado previo con Spark, ahora es posible generar un cuadro de mando estableciendo una conexión y consultas contra la base de datos. Si bien, el caso más interesante resultaría uno en el que se apreciaran datos en tiempo real, al tratarse de un *dataset*, los ejemplos mostrados con gráficas se obtienen de intervalos de tiempos que ya han ocurrido. Se trabaja con el conjunto de datos del metro de Oporto.

6.4.1. Datos analógicos y alertas

Si bien, sería posible crear gráficas para cada atributo, debido a los recursos limitados de memoria del equipo se han elaborado dos gráficas de los atributos más representativos:

- **Temperatura del aceite**: Cada valor recogido da lugar a un punto en la gráfica, generando el resultado visible en la Figura 20. Los valores pueden aparecer representados en tres colores:
  - Verde: No presentan problemas de presión o GPS.
  - Amarillo: No se ha detectado señal GPS al obtener el dato.
  - Naranja: El dato se ha obtenido con la señal LPS activada y existe un posible problema de presión en consecuencia.
- **Presión en válvula TP3**: La gráfica generada, visible también en la Figura 20, sigue un patrón de diseño similar. No obstante, en caso de existir una alerta de presión, que se origina cuando esta cae por debajo de los 7 bares, ahora se marca en rojo en lugar de naranja, pues dentro del contexto de la gráfica, que refleja concretamente la presión, es una alerta más severa.

Cabe destacar, que es posible que un registro genere tanto una alerta de GPS como de presión. En este caso, se priorizan las alertas de presión al considerarse de mayor severidad, mostrándose, por lo tanto, el color naranja o rojo por encima del amarillo.

Figura 20: Gráficas de valores analógicos y sus alertas.

6.4.2. Señales digitales

De igual forma que con los valores analógicos, también se han creado dos gráficas que permiten monitorizar la evolución en el tiempo de dos señales digitales:

- **LPS**: Señal que se activa cuando la presión del sistema cae por debajo de los 7 bares, siguiendo de nuevo un esquema de colores y severidad de las alarmas similar al descrito para las gráficas de valores analógicos, como se puede observar en la Figura 21.
- **GPS_QUALITY**: Señal que indica si se obtienen valores de GPS (valor igual a 1) o no (valor igual a 0). El esquema de colores para reflejar las alertas está permutado, mostrando en rojo el valor 0.

Figura 21: Gráficas de señales digitales.

6.4.3. Mapas

En base a las coordenadas obtenidas de cada medición, es posible pintar en un mapa el punto en el que se ha generado el dato. Teniendo en cuenta las alertas producidas por valores de presión por debajo de 7 bares, es posible mostrar un mapa al operador, donde pueda ver en qué zona se encuentra el tren cuando se produce la bajada de presión, como puede observarse en la Figura 22.

Debido de nuevo a problemas de memoria en el equipo, la consulta para generar el mapa se ha limitado únicamente a una extensión de cinco días en los que se obtuvo un registro de una avería durante los mismos, ya que representar todos los puntos conlleva un consumo excesivo de recursos. Esta medida se ha tomado también en las gráficas previas, con un rango similar, con el fin de asegurar

el buen funcionamiento de la plataforma. En un clúster real, esto no sería tan problemático, ya que no todas las tecnologías estarían desplegadas sobre un único equipo.

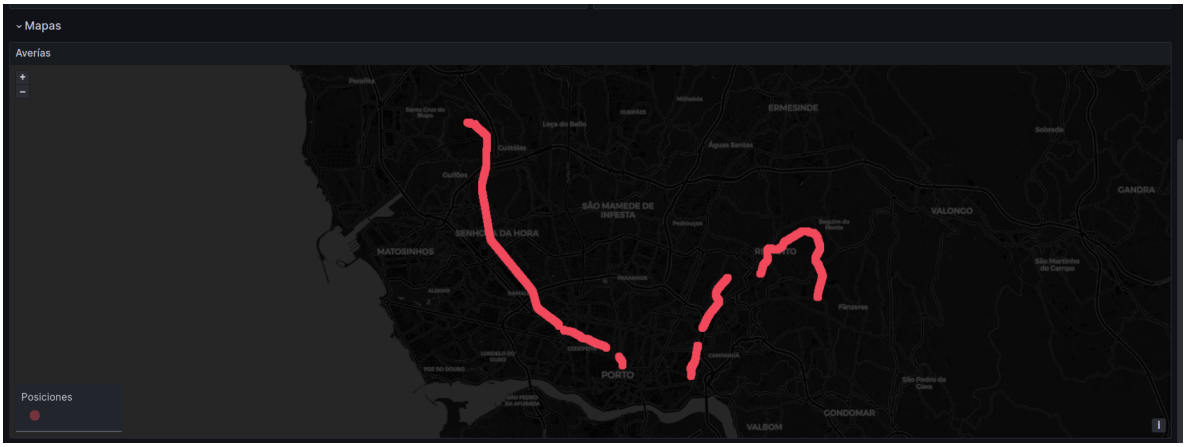

Figura 22: Mapa con mediciones que generan alertas por baja presión.

Cabe destacar que este último cuadro de mando, se ha diseñado para el análisis y la visualización de métricas de un único tren. En caso de existir varios, sería necesario recurrir a soluciones alternativas, tales como generar *dashboards* de monitorización para cada vehículo o realizar diversas agrupaciones en función de los intereses de los involucrados.

7. Conclusiones y líneas futuras de aplicación

El proceso de digitalización en el que estamos inversos lleva a la generación de ingentes cantidades de datos, lo que requiere de la adaptación de las arquitecturas y plataformas utilizadas para la gestión de datos, así como para su posterior análisis e interpretación.

Este trabajo, desarrollado dentro del contexto de un proyecto de investigación, demuestra, mediante un caso de IoT industrial, cómo es posible construir, mediante diversas tecnologías interconectadas surgidas a lo largo de los últimos años, un servicio que controle la calidad y seguridad de los datos. Todo ello, empleando una plataforma digital centrada en el dato, distribuible y escalable, que facilita la construcción de servicios como el de monitorización y visualización de los datos desarrollado en este TFG. Para ello, se ha expuesto un conjunto de herramientas, cada una de ellas dedicada a una tarea específica: gestión de eventos, procesado, persistencia de datos, obtención de lecturas, gestión de certificados, monitorización y visualización de la información procesada.

Una vez explicadas las herramientas a utilizar, se ha recopilado el proceso seguido para construir, a partir de un caso de uso, los diferentes elementos necesarios para llegar al almacenamiento de datos y su monitorización. Además, en el caso de existir alternativas relevantes, como ha sido en la sección de persistencia, se ha ofrecido un estudio de las ventajas, inconvenientes y rendimiento de los posibles sistemas a plantear.

Los ficheros resultantes de algunas configuraciones, así como los cuadros de mando y programas en Java elaborados, se encuentran accesibles públicamente en GitHub [15].

Cabe señalar que este proyecto de investigación, aún dispone de varias líneas de trabajo en las que avanzar:

- **Despliegue de la solución en una plataforma distribuida**: La investigación realizada se ha llevado a cabo en un equipo personal, que si bien permite experimentar con las tecnologías utilizadas y observar sus virtudes, no permite explotar todo su potencial, al encontrarse estas orientadas a entornos distribuidos. Dicho despliegue podría llevarse a cabo en un clúster de nodos *on-premise* o en la nube.
- **Incorporación de elementos a la arquitectura RAI4.0**: Las secciones relativas a seguridad y monitorización a través de Prometheus/Grafana, presentan un potencial interés para extender el metamodelo RAI4.0 [10].
- **Establecer patrones de predicción de averías**: El caso utilizado a lo largo del trabajo, podría evolucionar y completarse con la extracción de patrones con el fin de predecir averías, en lugar de mostrar únicamente cuando estas se producen.
- **Desarrollo de aplicaciones**: Una vez establecido todo el despliegue planteado, resulta posible la construcción de aplicaciones web, móviles o de escritorio, capaces de acceder a los datos o los cuadros de mando de Grafana mediante su API.
- **Aumentar el número de plataformas monitorizadas**: Es posible utilizar JMX Exporter con Prometheus para obtener métricas de Cassandra o la extensión Prometheus Emitter de Druid. No obstante, no se ha contemplado su desarrollo en el presente proyecto, debido a la amplitud que se recoge y con el fin de adaptar el trabajo realizado al número de horas disponibles.

Finalmente, en el plano personal, este trabajo desarrollado en el contexto de una beca de colaboración, me ha facilitado y acercado a la comprensión del funcionamiento del mundo académico y de la investigación, empleando para ello conocimientos adquiridos a lo largo del grado con el fin de aprender sobre tecnologías más específicas y aplicarlos a estas.

Bibliografía

[1] Apache Cassandra. *Main Documentation*. [Consulta: 9 febrero 2023]. Disponible en: https://cassandra.apache.org/doc/latest/

[2] Apache Druid. *Druid Docs*. [Consulta: 23 marzo 2023]. Disponible en: https://druid.apache.org/docs/latest

[3] Apache Druid. *Technology*. [Consulta: 26 marzo 2023]. Disponible en: https://druid.apache.org/technology

[4] Apache Kafka. *Downloads*. [Consulta: 23 enero 2023]. Disponible en: https://kafka.apache.org/downloads

[5] Apache Kafka. *Introduction*. [Consulta: 23 enero 2023]. Disponible en: https://kafka.apache.org/intro

[6] Apache Spark. *Spark Docs*. [Consulta: 6 febrero 2023]. Disponible en: https://spark.apache.org/docs/latest/

[7] Apache Spark. *Spark Docs, Streaming programming guide*. [Consulta: 6 febrero 2023]. Disponible en: https://spark.apache.org/docs/latest/streaming-programming-guide.html

[8] Apache Zookeeper. *Zookeeper Documentation, Overview* [Consulta: 23 enero 2023]. Disponible en: https://zookeeper.apache.org/doc/r3.8.1/zookeeperOver.html

[9] Ayuntamiento de Santander. *Datos abiertos, Sensores ambientales*. [Consulta: 23 enero 2023]. Disponible en: https://datos.santander.es/dataset/?id=sensores-ambientales

[10] Dintén, R., López Martínez, P., Zorrilla, M. 2021. Reference architecture for the designand development of applications for Industry 4.0. *Revista Iberoamericana de Automática e Informática Industrial* 18, 300-311. Disponible en: https://doi.org/10.4995/riai.2021.14532

[11] Evgeny, Milanov. 2009. *The RSA Algorithm*. Disponible en: https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf

[12] Grafana. *Grafana, Docs, Introduction*. [Consulta: 15 marzo 2023]. Disponible en: https://grafana.com/docs/grafana/latest/introduction/

[13] IBM. *IBM Web Sphere, Java Management Extensions*. [Consulta: 30 abril 2023]. Disponible en: https://www.ibm.com/docs/es/was-zos/8.5.5?topic=jmx-java-management-extensions-websphere-application-server.

[14] IBM. *Sterling External Authentication Server, BasicConstraints Extension*. [Consulta: 11 abril 2023]. Disponible en: https://www.ibm.com/docs/en/external-auth-server/2.4.3?topic=extensions-basicconstraints-extension

[15] Martín, Mario. *TFG_ Kafka, Repositorio GitHub*. Disponible en: https://github.com/mmp819/TFG_Kafka

[16] Ogievetsky, Vadim. *Koalas to the Max*. [Consulta: 24 marzo 2023]. Disponible en: https://koalastothemax.com

[17] Oracle. *Keytool*. [Consulta: 11 abril 2023]. Disponible en: https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html

[18] Oracle. *Java Cryptography Architecture Oracle Providers Documentation for JDK 8*. [Consulta: 11 abril 2023]. Disponible en: https://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html

[19] Panda Security. 2022. *Mediacenter, Seguridad, Protocolo SSL*. [Consulta: 11 abril 2023]. Disponible en: https://www.pandasecurity.com/es/mediacenter/seguridad/protocolo-ssl/

[20] Pedreschi, Rachel. 2019. *Apache Cassandra vs. Apache Druid*. [Consulta: 4 mayo 2023]. Disponible en: https://imply.io/blog/apache-cassandra-vs-apache-druid/

[21] Prometheus. *Prometheus, Docs, Introduction*. [Consulta: 10 febrero 2023]. Disponible en: https://prometheus.io/docs/introduction/overview/

[22] Prometheus. *Prometheus, Docs, Configuration*. [Consulta: 25 abril 2023]. Disponible en: https://prometheus.io/docs/prometheus/latest/configuration/configuration/#tls_config

[23] Prometheus. *Repositorio GitHub*. Disponible en: https://github.com/prometheus/jmx_exporter

[24] Shapira, G. [et al.] *Kafka: The Definitive Guide. Real-Time Data and Stream Processing at Scale*. 2nd ed. USA: O'Reilly Media. ISBN 978-1492043089

[25] Veloso, B., Ribeiro, R.P., Gama, J. et al. 2022. *MetroPT: A Benchmark dataset for predictive maintenance*. [Consulta: 24 mayo 2023]. Disponible en: https://doi.org/10.5281/zenodo.6854240

[26] Veloso, B., Ribeiro, R.P., Gama, J. et al. 2022. The MetroPT dataset for predictive maintenance. *Sci Data* 9, 764. Disponible en: https://doi.org/10.1038/s41597-022-01877-3

A. Muestra de los conjuntos de datos utilizados

- **Datos ambientales (CSV)**:

```
3005,2019-10-21T07:18:00Z,AirQualityObserved,43.4687,-3.78775,115.0,28.0,,,17.2,0.1,,,,
    ↪http://datos.santander.es/api/datos/sensores_smart_mobile/3005.csv
3032,2019-10-21T07:17:49Z,AirQualityObserved,43.4745,-3.78537,19.0,4.0,,,15.1,0.2,,,,
    ↪http://datos.santander.es/api/datos/sensores_smart_mobile/3032.csv
3006,2019-10-21T07:17:39Z,AirQualityObserved,43.4553,-3.83414,116.0,10.0,,,17.5,0.1,,,,
    ↪http://datos.santander.es/api/datos/sensores_smart_mobile/3006.csv
```

- **Juego web (JSON)**:

```
{"timestamp":"2019-08-25T00:00:00.031Z","session":"S56194838","number":"16","event":{"
    ↪type":"PercentClear","percentage":55},"agent":{"type":"Browser","category":"
    ↪Personal computer","browser":"Chrome","browser_version":"76.0.3809.100","os":"
    ↪Windows 7","platform":"Windows"},"client_ip":"181.13.41.82","geo_ip":{"continent
    ↪":"South America","country":"Argentina","region":"Santa Fe","city":"Rosario"},"
    ↪language":["es","es-419"],"adblock_list":"NoAdblock","app_version":"1.9.6","path
    ↪":"http://www.koalastothemax.com/","loaded_image":"http://www.koalastothemax.com
    ↪/img/koalas2.jpg","referrer":"Direct","referrer_host":"Direct","server_ip":"172.
    ↪31.57.89","screen":"1680x1050","window":"1680x939","session_length":76261,"
    ↪timezone":"N/A","timezone_offset":"180"}
{"timestamp":"2019-08-25T00:00:00.059Z","session":"S46093731","number":"24","event":{"
    ↪type":"PercentClear","percentage":85},"agent":{"type":"Mobile Browser","category
    ↪":"Smartphone","browser":"Chrome Mobile","browser_version":"50.0.2661.89","os":"
    ↪Android","platform":"Android"},"client_ip":"177.242.100.0","geo_ip":{"continent"
    ↪:"North America","country":"Mexico","region":"Chihuahua","city":"Nuevo Casas
    ↪Grandes"},"language":["en","es","es-419","es-MX"],"adblock_list":"NoAdblock","
    ↪app_version":"1.9.6","path":"https://koalastothemax.com/","loaded_image":"https:
    ↪//koalastothemax.com/img/koalas1.jpg","referrer":"https://www.google.com/","
    ↪referrer_host":"www.google.com","server_ip":"172.31.11.5","screen":"320x570","
    ↪window":"540x743","session_length":252689,"timezone":"CDT","timezone_offset":"30
    ↪0"}
```

- **MetroPT - Datos del metro de Oporto (CSV)**:

```
timestamp,TP2,TP3,H1,DV_pressure,Reservoirs,Oil_temperature,Flowmeter,Motor_current,
    ↪COMP,DV_eletric,Towers,MPG,LPS,Pressure_switch,Oil_level,Caudal_impulses,gpsLong
    ↪,gpsLat,gpsSpeed,gpsQuality

2022-01-01 06:00:00,-0.0120000000000004,9.758,9.76,-0.0279999999999986,
    ↪1.5760000000000003, 63.35, 19.049625,3.955,1,0,1,1,0,0,0,0,-8.65934,41.2124,0,1
2022-01-01 06:00:01, -0.0120000000000004, 9.76, 9.76, -0.0279999999999986,
    ↪1.5779999999999994, 63.25,19.049625, 4.0275,1,0,1,1,0,0,0,0,-8.65934,41.2124,0,1
```

B. *Spec* Druid - *Dataset* juego web

Definición de la configuración de la conexión entre Kafka y Druid para el conjunto de datos del juego web *Koalas to the Max*.

```
{
  "type": "kafka",
  "spec": {
    "ioConfig": {
      "type": "kafka",
      "consumerProperties": {
        "bootstrap.servers": "mario:9092"
      },
      "topic": "kttm",
      "inputFormat": {
        "type": "json"
      },
      "useEarliestOffset": true
    },
    "tuningConfig": {
      "type": "kafka"
    },
    "dataSchema": {
      "dataSource": "kttm",
      "timestampSpec": {
        "column": "timestamp",
        "format": "iso"
      },
      "dimensionsSpec": {
        "dimensions": [
          "session",
          "number",
          "client_ip",
          "language",
          "adblock_list",
          "app_version",
          "path",
          "loaded_image",
          "referrer",
          "referrer_host",
          "server_ip",
          "screen",
          "window",
          {
            "type": "long",
            "name": "session_length"
          },
          "timezone",
          "timezone_offset",
          {
            "type": "json",
```

```
            "name": "event"
          },
          {
            "type": "json",
            "name": "agent"
          },
          {
            "type": "json",
            "name": "geo_ip"
          }
        ]
      },
      "granularitySpec": {
        "queryGranularity": "none",
        "rollup": false,
        "segmentGranularity": "day"
      }
    }
  }
}
```

C. *Spec* Druid - *Dataset* MetroPT

Definición de la configuración de la conexión entre Kafka y Druid para el conjunto de datos del metro de Oporto. Concretamente, se aporta la utilizada para el tópico *lecturas_filtradas*, siendo necesario cambiar los campos *dataSource* y *topic* para realizar la conexión con otros tópicos.

```
{
  "type": "kafka",
  "spec": {
    "dataSchema": {
      "dataSource": "lecturas_filtradas",
      "timestampSpec": {
        "column": "timestamp",
        "format": "auto",
        "missingValue": null
      },
      "dimensionsSpec": {
        "dimensions": [
          {
            "type": "double",
            "name": "TP2",
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
          {
            "type": "double",
            "name": "TP3",
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
          {
            "type": "double",
            "name": "H1",
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
          {
            "type": "double",
            "name": "DV_pressure",
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
          {
            "type": "double",
            "name": "Reservoirs",
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
          {
            "type": "double",
            "name": "Oil_temperature",
```

```
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
          {
            "type": "double",
            "name": "Flowmeter",
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
          {
            "type": "double",
            "name": "Motor_current",
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
          {
            "type": "long",
            "name": "COMP",
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
          {
            "type": "long",
            "name": "DV_eletric",
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
          {
            "type": "long",
            "name": "Towers",
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
          {
            "type": "long",
            "name": "MPG",
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
          {
            "type": "long",
            "name": "LPS",
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
          {
            "type": "long",
            "name": "Pressure_switch",
            "multiValueHandling": "SORTED_ARRAY",
            "createBitmapIndex": false
          },
```

```
97            {
98              "type": "long",
99              "name": "Oil_level",
100             "multiValueHandling": "SORTED_ARRAY",
101             "createBitmapIndex": false
102           },
103           {
104             "type": "long",
105             "name": "Caudal_impulses",
106             "multiValueHandling": "SORTED_ARRAY",
107             "createBitmapIndex": false
108           },
109           {
110             "type": "double",
111             "name": "gpsLong",
112             "multiValueHandling": "SORTED_ARRAY",
113             "createBitmapIndex": false
114           },
115           {
116             "type": "double",
117             "name": "gpsLat",
118             "multiValueHandling": "SORTED_ARRAY",
119             "createBitmapIndex": false
120           },
121           {
122             "type": "long",
123             "name": "gpsSpeed",
124             "multiValueHandling": "SORTED_ARRAY",
125             "createBitmapIndex": false
126           },
127           {
128             "type": "long",
129             "name": "gpsQuality",
130             "multiValueHandling": "SORTED_ARRAY",
131             "createBitmapIndex": false
132           }
133         ],
134         "dimensionExclusions": [
135           "__time",
136           "timestamp"
137         ],
138         "includeAllDimensions": false
139       },
140       "metricsSpec": [],
141       "granularitySpec": {
142         "type": "uniform",
143         "segmentGranularity": "DAY",
144         "queryGranularity": {
145           "type": "none"
146         },
147         "rollup": false,
```

```
        "intervals": []
      },
      "transformSpec": {
        "filter": null,
        "transforms": []
      }
    },
    "ioConfig": {
      "topic": "lecturas_filtradas",
      "inputFormat": {
        "type": "csv",
        "columns": [
          "timestamp",
          "TP2",
          "TP3",
          "H1",
          "DV_pressure",
          "Reservoirs",
          "Oil_temperature",
          "Flowmeter",
          "Motor_current",
          "COMP",
          "DV_eletric",
          "Towers",
          "MPG",
          "LPS",
          "Pressure_switch",
          "Oil_level",
          "Caudal_impulses",
          "gpsLong",
          "gpsLat",
          "gpsSpeed",
          "gpsQuality"
        ]
      },
      "replicas": 1,
      "taskCount": 1,
      "taskDuration": "PT3600S",
      "consumerProperties": {
        "bootstrap.servers": "mario:9093",
        "ssl.keystore.location": "/home/mario/Escritorio/TFG/CertificadosKafka/Clientes/
            ↪Druid/druidcliente.ks.p12",
        "ssl.keystore.password": "druidcliente-ks-password",
        "ssl.key.password": "druidcliente-ks-password",
        "security.protocol": "SSL",
        "ssl.truststore.location": "/home/mario/Escritorio/TFG/CertificadosKafka/Clientes/
            ↪clientes.ts.p12",
        "ssl.truststore.password": "clientes-ts-password"
      },
      "autoScalerConfig": null,
      "pollTimeout": 100,
```

```
      "startDelay": "PT5S",
      "period": "PT30S",
      "useEarliestOffset": true,
      "completionTimeout": "PT1800S",
      "lateMessageRejectionPeriod": null,
      "earlyMessageRejectionPeriod": null,
      "lateMessageRejectionStartDateTime": null,
      "configOverrides": null,
      "idleConfig": null,
      "stream": "lecturas_filtradas",
      "useEarliestSequenceNumber": true
    },
    "tuningConfig": {
      "type": "kafka",
      "appendableIndexSpec": {
        "type": "onheap",
        "preserveExistingMetrics": false
      },
      "maxRowsInMemory": 1000000,
      "maxBytesInMemory": 0,
      "skipBytesInMemoryOverheadCheck": false,
      "maxRowsPerSegment": 5000000,
      "maxTotalRows": null,
      "intermediatePersistPeriod": "PT10M",
      "maxPendingPersists": 0,
      "indexSpec": {
        "bitmap": {
          "type": "roaring",
          "compressRunOnSerialization": true
        },
        "dimensionCompression": "lz4",
        "stringDictionaryEncoding": {
          "type": "utf8"
        },
        "metricCompression": "lz4",
        "longEncoding": "longs"
      },
      "indexSpecForIntermediatePersists": {
        "bitmap": {
          "type": "roaring",
          "compressRunOnSerialization": true
        },
        "dimensionCompression": "lz4",
        "stringDictionaryEncoding": {
          "type": "utf8"
        },
        "metricCompression": "lz4",
        "longEncoding": "longs"
      },
      "reportParseExceptions": false,
      "handoffConditionTimeout": 0,
```

```
      "resetOffsetAutomatically": false,
      "segmentWriteOutMediumFactory": null,
      "workerThreads": null,
      "chatThreads": null,
      "chatRetries": 8,
      "httpTimeout": "PT10S",
      "shutdownTimeout": "PT80S",
      "offsetFetchPeriod": "PT30S",
      "intermediateHandoffPeriod": "P2147483647D",
      "logParseExceptions": false,
      "maxParseExceptions": 2147483647,
      "maxSavedParseExceptions": 0,
      "skipSequenceNumberAvailabilityCheck": false,
      "repartitionTransitionDuration": "PT120S"
    }
  },
  "context": null,
  "suspended": false
}
```

D. Cassandra - *Dataset* juego web

Esquema de base de datos utilizado para almacenar el conjunto de datos del juego web *Koalas to the Max*.

- **Crear *keyspace***:

```
create KEYSPACE kttm with replication={'class':'SimpleStrategy', 'replication_factor'
    ↪:1};
```

- **Crear tipos propios**:

```
create type event(type TEXT, data TEXT);
create type agent(type TEXT, category TEXT, browser TEXT, browser_version TEXT, os TEXT
    ↪, platform TEXT);
create type geo_ip(continent TEXT, country TEXT, region TEXT, city TEXT);
```

- **Crear tabla que use los tipos y que recogerá las instancias del fichero**:

```
create table kttm_kafka(timestamp TIMESTAMP, session TEXT, number INT, event FROZEN<
    ↪event>, agent FROZEN<agent>, client_ip TEXT, geo_ip FROZEN<geo_ip>, language SET
    ↪<TEXT>, adblock_list TEXT, app_version TEXT, path TEXT, loaded_image TEXT,
    ↪referrer TEXT, referrer_host TEXT, server_ip TEXT, screen TEXT, window TEXT,
    ↪session_length BIGINT, timezone TEXT, timezone_offset TEXT, PRIMARY KEY (
    ↪timestamp, session));
```

E. Cassandra - *Dataset* MetroPT

Esquema de base de datos utilizado para almacenar el conjunto de datos del metro de Oporto.

- **Crear *keyspace***:

```
create KEYSPACE train with replication={'class':'SimpleStrategy', 'replication_factor'
    ↪:1};
```

- **Crear tabla que use los tipos y que recogerá las instancias del fichero**:

```
create table lecturas(timestamp TIMESTAMP, TP2 float, TP3 float, H1 float, DV_pressure
    ↪float, reservoirs float, oil_temperature float, flowmeter float, motor_current
    ↪float, COMP int, DV_electric int, towers int, MPG int, LPS int, pressure_switch
    ↪int, oil_level int, caudal_impulses int, gpsLong float, gpsLat float, gpsSpeed
    ↪int, gpsQuality int, PRIMARY KEY(timestamp));
```

F. Fichero de configuración SSL - Cliente Kafka

Utilizado para ejecutar comandos con una autenticación como administrador.

```
security.protocol=SSL
ssl.keystore.location=/home/mario/Escritorio/TFG/CertificadosKafka/Clientes/Admin/admin.ks.p12
ssl.keystore.password=admin-ks-password
ssl.key.password=admin-ks-password
ssl.keystore.type=PKCS12
ssl.truststore.location=/home/mario/Escritorio/TFG/CertificadosKafka/Clientes/clientes.ts.p12
ssl.truststore.password=clientes-ts-password
ssl.truststore.type=PKCS12
```

G. Creación de certificado para cliente Java (Spark)

```
keytool -genkey -keyalg RSA -keysize 2048 -keystore sparkcliente.ks.p12 -storetype PKCS12 -storepass sparkcliente-ks-password -keypass sparkcliente-ks-password -alias SparkCliente -dname "CN=Spark, OU=Facultad de Ciencias, O=Universidad de Cantabria, C=ES" -validity 365
```

```
keytool -certreq -file sparkcliente.csr -keystore sparkcliente.ks.p12 -storetype PKCS12 -storepass sparkcliente-ks-password -keypass sparkcliente-ks-password -alias SparkCliente
```

```
keytool -gencert -infile sparkcliente.csr -outfile sparkcliente.crt -keystore ../../ClientesCA/clientes.ca.p12 -storetype PKCS12 -storepass clientes-ca-password -alias ClientesCA -validity 365 -rfc
```

```
cat sparkcliente.crt ../../ClientesCA/clientes.ca.crt > sparkclientechain.crt
```

```
keytool -importcert -file sparkclientechain.crt -keystore sparkcliente.ks.p12 -storepass sparkcliente-ks-password -keypass sparkcliente-ks-password -alias SparkCliente -storetype PKCS12 -noprompt
```

H. Creación de certificado para productor Java

```
keytool -genkey -keyalg RSA -keysize 2048 -keystore productor.ks.p12 -storetype PKCS12 -storepass productor-ks-password -keypass productor-ks-password -alias Productor -dname "CN=Productor, OU=Facultad de Ciencias, O=Universidad de Cantabria, C=ES" -validity 365
```

```
keytool -certreq -file productor.csr -keystore productor.ks.p12 -storetype PKCS12 -storepass productor-ks-password -keypass productor-ks-password -alias Productor
```

```
keytool -gencert -infile productor.csr -outfile productor.crt -keystore ../../ClientesCA/clientes.ca.p12 -storetype PKCS12 -storepass clientes-ca-password -alias ClientesCA -validity 365 -rfc
```

```
cat productor.crt ../../ClientesCA/clientes.ca.crt > productorchain.crt
```

```
keytool -importcert -file productorchain.crt -keystore productor.ks.p12 -storepass productor-ks-password -keypass productor-ks-password -alias Productor -storetype PKCS12 -noprompt
```

I. Creación de certificado para Druid

```
keytool -genkey -keyalg RSA -keysize 2048 -keystore druidcliente.ks.p12 -storetype PKCS12 -storepass druidcliente-ks-password -keypass druidcliente-ks-password -alias DruidCliente -dname "CN=Druid, OU=Facultad de Ciencias, O=Universidad de Cantabria, C=ES" -validity 365
```

```
keytool -certreq -file druidcliente.csr -keystore druidcliente.ks.p12 -storetype PKCS12 -storepass druidcliente-ks-password -keypass druidcliente-ks-password -alias DruidCliente
```

```
keytool -gencert -infile druidcliente.csr -outfile druidcliente.crt -keystore ../../ClientesCA/clientes.ca.p12 -storetype PKCS12 -storepass clientes-ca-password -alias ClientesCA -validity 365 -rfc
```

```
cat druidcliente.crt ../../ClientesCA/clientes.ca.crt > druidclientechain.crt
```

```
keytool -importcert -file druidclientechain.crt -keystore druidcliente.ks.p12 -storepass druidcliente-ks-password -keypass druidcliente-ks-password -alias DruidCliente -storetype PKCS12 -noprompt
```