Facultad de Ciencias

Monitorización en tiempo real del comportamiento en la conducción de vehículos

Real-time monitoring of vehicle driving behavior

Trabajo de Fin de Máster
para acceder al

MÁSTER EN INGENIERÍA INFORMÁTICA

Autor: Sebastián García Bustamante

Directores: Marta Zorrilla y Ricardo Dintén

Resumen

En la actualidad, las empresas tienen la necesidad de procesar cantidades ingentes de datos que proceden en gran medida de dispositivos Internet of Things (IoT), por lo cual requieren el uso de tecnologías Big Data para realizar un procesamiento en un tiempo acotado. Adicionalmente a procesar grandes volúmenes de datos, estos deben ser procesados en tiempo real no estricto, por lo que se requiere que la arquitectura sea flexible, distribuida, y escalable. Este es el caso del sector logístico, donde mediante dispositivos IoT se recogen una serie de parámetros que para el caso de estudio que aquí se aborda ayuda a la empresa a medir el estado y determinar el modo de conducción de los conductores. El objetivo de este Trabajo Fin de Máster es diseñar, desarrollar y desplegar un sistema distribuido y escalable que realice el cálculo para determinar el comportamiento de conducción de los conductores en tiempo real no estricto. Para ello se utilizará el gestor de colas utilizando RabbitMQ, al cual los camiones envían datos del ordenador de viaje con cierta frecuencia y Apache Flink, como procesador de *streaming* que se encargará de limpiar, procesar, y agregar los datos para generar tramos y computar los patrones de conducción. Estos se almacenarán en el gestor NoSQL Cassandra.

Palabras clave: IoT, Procesador Complejo de Eventos, Arquitectura Kappa, Aplicaciones de Datos Intensivos

Abstract

Currently, companies have the need to process huge amounts of data that come largely from Internet of Things (IoT) devices, that is why they require the use of Big Data technologies to carry out processing in a limited amount time. In addition to managing large volumes of data, this data must be processed in real time, that is why the architecture must be flexible, distributed, and scalable. This case applies to the logistics sector, where IoT devices are used to monitor a series of indicators that help companies to assess the driver's driving behavior for processing and analysis. The objective of this dissertation is to design, develop, and deploy a system that is capable of solving these challenges. The objective of this work is to design, develop and deploy a distributed and scalable system that performs the computation to determine the driving behavior of drivers in non-strict real time. This will be done using the queue manager using RabbitMQ, to which the trucks send data from the trip computer with a certain frequency and Apache Flink, as a streaming processor that will be in charge of cleaning, processing, and aggregating the data to generate trips and compute the driving patterns. These will be stored in the Cassandra NoSQL manager.

Keywords: IoT, Complex Event Processor, Kappa Architecture, Data Intensive Applications

Índice de contenido

Índice de Figuras

Índice de Tablas

Contexto y Objeto del Proyecto

Este Trabajo Fin de Máster (TFM) se ha desarrollado dentro del marco del proyecto "rut-IA: Investigación industrial de tecnologías de analítica de datos para el tratamiento de información de explotación logística georreferenciada" de la empresa Fagor Telecom, S.L.U. en colaboración con el grupo de investigación Ingeniería de Software y Tiempo Real de la Universidad de Cantabria y cofinanciado por la consejería de Industria, Turismo, Innovación, Transporte y Comercio del Gobierno de Cantabria.

Fagor Telecom, S.L.U. es una empresa de electrónica que se dedica a vender ordenadores de abordo para su instalación en flotas de vehículos, en particular, camiones y vehículos de reparto. Junto con estos dispositivos suministra un software de gestión de flotas, denominado FlotasNet. Este software facilita a empresas de logística información crucial sobre la actividad y telemetría de la flota con el fin de mejorar la rentabilidad, seguridad, toma de decisiones y el servicio a sus clientes. El software proporciona un sistema de visualización y análisis de indicadores clave de rendimiento (KPI), y entre otros, ofrece el perfil de conducción de los conductores. Los datos que son analizados se capturan por los sensores alojados en los camiones y son enviados periódicamente a una base de datos centralizada. Estos datos de telemetría y geoposición se utilizan para calcular el perfil de conducción de los conductores el cual se realiza por lotes cada noche recalculando los tramos que han realizado los camiones en los últimos tres días. Fagor tiene como objetivo cambiar este sistema de procesamiento centralizado y por lotes a uno distribuido y en tiempo real para enviar a los conductores notificaciones que les permita modificar su conducción, y consecuentemente reducir costes y la emisión de CO2. Además, una solución distribuida y escalable les permitiría adaptar mejor su solución al crecimiento de su mercado que tienen previsto para los próximos dos años.

Para satisfacer ambos requisitos en este trabajo se propone una solución con una arquitectura que implementa un gestor de colas, un software de procesamiento distribuido y paralelizado, y que es capaz de persistir estos indicadores en una base de datos NoSQL.

El diseño, desarrollo y despliegue de aplicaciones intensivas en datos (DIA), como la que se implementa en este TFM es complejo debido principalmente a la gran cantidad de diferentes tecnologías que deben organizarse, configurarse, e implementarse en un entorno mixto de *cloud*, *fog*, y *edge computing* para cumplir con su funcionalidad. Para ello se hará uso del metamodelo de referencia RAI4.0 desarrollado por el grupo de Ingeniería de Software y Tiempo Real (ISTR) de la Universidad de Cantabria. RAI4.0 es un metamodelo que recoge la descripción de todos los elementos involucrados en una plataforma digital Big Data (datos, recursos, cargas de trabajo y métricas), así como la información necesaria para configurar, desplegar y ejecutar cargas de trabajo (aplicaciones) en él (López Martínez et al., 2021).

En relación con el cálculo del perfil de conducción se implementará de modo que se combinen y procesen los datos procedentes de los camiones cada parada, o fin de tramo de modo que al arrancar de nuevo se conozca la evaluación del tramo anterior.

En resumen, el objeto de este trabajo es presentar una solución que permita que Fagor supere los desafíos del crecimiento de la flota que habilite el procesamiento en tiempo real. Por lo tanto, la solución procesa los datos en un flujo continuo en vez de por lotes para permitir la notificación a los conductores al finalizar cada tramo. Además, es escalable y distribuida para poder paralelizar la carga de trabajo.

Planificación del Proyecto

El desarrollo de la solución propuesta se ha llevado a cabo de acuerdo con las siguientes fases:

1. Análisis del problema y diseño de la solución a implementar
2. Instalación, configuración, y pruebas de comunicación de la plataforma digital
3. Procesamiento y cálculo de tramos a partir del fichero de eventos históricos
4. Procesamiento y cálculo de tramos en base a cantrama y eventos históricos
5. Despliegue de solución en entorno distribuido (nube de Google)
6. Monitorización y evaluación de la configuración on premise y distribuida
7. Redacción de la memoria

En la Figura 1 Diagrama de Gantt se recoge un diagrama de Gantt que muestra la distribución temporal de las tareas realizadas.

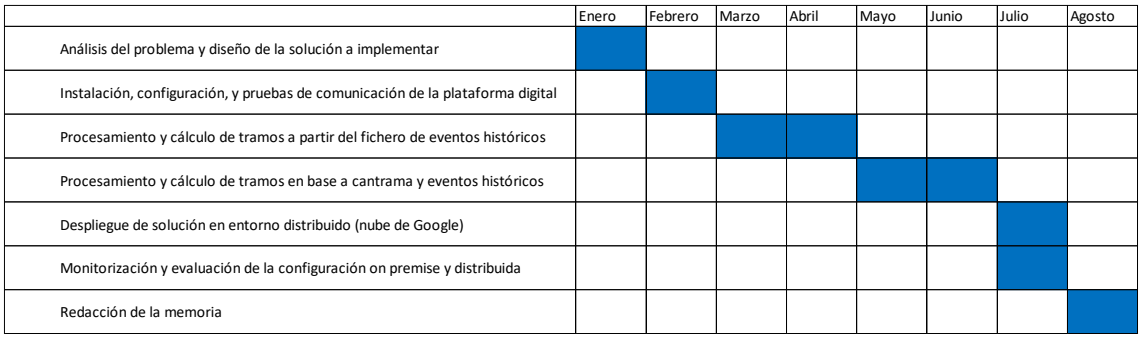

Figura 1 Diagrama de Gantt

Análisis del problema y diseño de la solución a implementar

En esta fase se estudia y comprende el problema a resolver. Por un lado, se analiza el algoritmo para la generación de tramos y el cálculo de los parámetros necesarios para generar el perfil de conducción que utiliza Fagor; y por otro, se estudia la arquitectura kappa y la documentación relativa al modelo de referencia RAI4.0 para el diseño de la plataforma digital. Adicionalmente se estudian las tecnologías big data seleccionadas para desarrollar el proyecto. Estas tecnologías son: el gestor de colas RabbitMQ, procesamiento de datos con Apache Flink y la capa de persistencia utilizando Apache Cassandra.

Instalación, configuración, y pruebas de comunicación de la plataforma digital

En esta segunda fase se instalan las herramientas en mi computador personal para la realización de pruebas de comunicación de la plataforma digital. Estas pruebas consisten en:

1. Crear un modelo Productor Consumidor usando RabbitMQ para simular el envío y consumo de eventos de una cola.
2. Desarrollar una simple aplicación Java para Apache Flink que lea datos de una cola de RabbitMQ envíe a la salida estándar, para integrar ambas tecnologías.
3. Finalmente persistir los datos obtenidos con Flink en Apache Cassandra.

Procesamiento y cálculo de tramos a partir del fichero de eventos históricos

En esta tercera fase se estudia un fichero que contiene los parámetros generados por los ordenadores de viaje de los camiones. A continuación, se implementan scripts de Python que permitan enviar los parámetros de los ordenadores de viaje a la capa de procesamiento con Apache Flink. Para agilizar el proceso de desarrollo y pruebas se utiliza un subconjunto reducido

de los datos disponibles. Al finalizar el desarrollo en Flink se procede a persistir los tramos en una base de datos NoSQL Cassandra.

Procesamiento y cálculo de tramos en base a cantrama y eventos históricos

Se estudia y comprende un segundo fichero de datos con parámetros adicionales a los eventos procesados anteriormente. Para incluirlos y procesar todos los parámetros que un camión genera se desarrolla una operación de unión (*Join*) en Flink. Además, se realizan las modificaciones pertinentes a la base de datos para incluir los nuevos parámetros.

Despliegue de la solución en un entorno distribuido (nube de Google)

Durante el mes de julio, se monta una arquitectura distribuida y escalable en base a máquinas virtuales en la nube de Google Cloud Engine (GCE) y se despliega la solución utilizando diferentes configuraciones para las máquinas virtuales.

Monitorización y evaluación de solución en local y distribuido

A finales de julio y comienzos de agosto se crean subconjuntos de los dos ficheros proporcionados por Fagor para realizar el prototipo. Con estos subconjuntos de datos se realizan tres pruebas, una para analizar el funcionamiento del sistema bajo carga, una para analizar la tolerancia a fallos del sistema, y una tercera para analizar el comportamiento del sistema al realizar escalada en caliente.

Redacción de la memoria

El mes de agosto se dedica a la redacción de la memoria y se realizan las revisiones pertinentes con los tutores de este TFM.

Diseño de la Plataforma Digital y Tecnologías Utilizadas

El sistema que debe procesar los eventos de los camiones de una flota logística en crecimiento necesita ser escalable y distribuida. Debido a la cantidad y variedad de datos que generan los camiones y la velocidad de creación de estos se requieren plataformas basadas en tecnología big data para ingerir, procesar y entregar conocimiento útil en tiempo real. Las arquitecturas distribuidas y escalables tipo Kappa son actualmente la solución de software que se adapta a estas necesidades y desafíos que se pueden ver dentro del sector del transporte (Philip Chen & Zhang, 2014).

Las arquitecturas Kappa (ver Figura 2 Arquitectura Kappa) se componen de dos capas: una capa de procesamiento de flujo y una capa de servicio. La primera capa se encarga del procesamiento de los flujos de datos que son introducidos al sistema, para así poder realizar el procesamiento de datos en tiempo real. La segunda se utiliza para consultar los resultados.

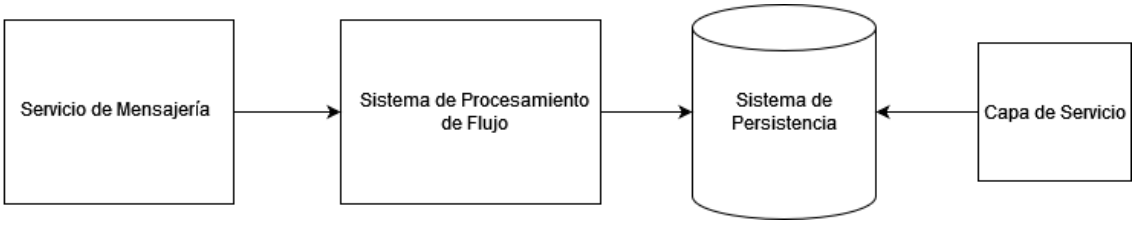

Figura 2 Arquitectura Kappa

Protocolo MQTT

MQTT es un protocolo de mensajería que se ejecuta sobre TCP/IP y permite la publicación y suscripción a colas entre servidores y clientes. Las funciones de MQTT incluyen: distribución de mensajes de uno a muchos nodos, agnóstico al contenido de la carga útil (*payload*) del mensaje, y también tiene tres tipos de funcionamiento de calidad de servicio para garantizar la entrega de mensajes (Soni & Makwana, 2017):

- At most once: donde los mensajes se publican, pero la pérdida del mensaje puede ocurrir.
- At least once: donde se asegura que el mensaje llegará, pero pueden ocurrir mensajes duplicados.
- Exactly once: donde se asegura que el mensaje llegue exactamente una vez.

MQTT es un protocolo que en particular se utiliza con dispositivos IoT porque al ser un protocolo que no demanda de una carga computacional grande para su funcionamiento puede ser usado en computadores embebidos (Hwang et al., 2016).

RabbitMQ

RabbitMQ es un *broker* de mensajería de código abierto que fue lanzado en el 2007 y desarrollado y actualizado por la empresa Pivotal Software. RabbitMQ implementa el estándar AMQP (Advanced Message Queuing Protocol), está escrito en el lenguaje de programación Erlang y basado en el framework Open Telecom Platform.(RabbitMQ, 2007) RabbitMQ tiene ciertas características clave que han contribuido a su uso en la industria. En primer lugar, se pueden agrupar varios servidores RabbitMQ de una red local, formando un *cluster* que trabaja en conjunto con el fin del obtener balanceo de carga y la tolerancia a fallos. Otra característica importante es el protocolo AMQP que utiliza RabbitMQ que acepta conexiones entre diferentes plataformas. Por ejemplo, se aceptan tanto el protocolo de cola de mensajes MSMQ que usa C# como STOMP que usa Ruby (Manuel Ionescu, 2015).

La estructura de RabbitMQ se ve en la Figura 3. Arquitectura RabbitMQ:

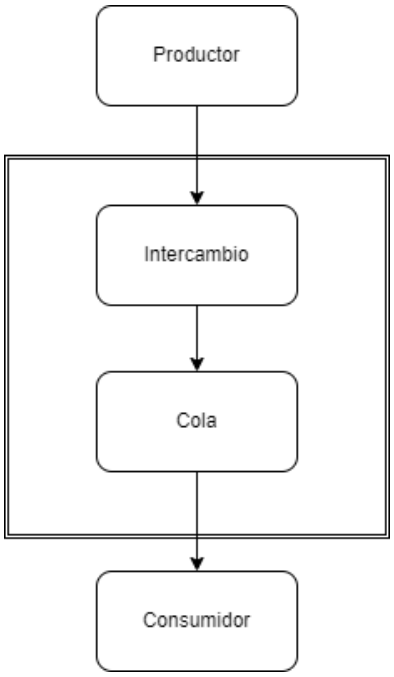

Figura 3. Arquitectura RabbitMQ

El productor realiza una operación llamada intercambio lo cual significa una publicación de un mensaje. El intercambio lleva consigo también una regla de enrutamiento para los mensajes, esto se usa para que el *broker* de RabbitMQ pueda enrutar los mensajes a la cola correcta. Los principales tipos de intercambios que se tienen en RabbitMQ son (*AMQP 0-9-1 Model Explained — RabbitMQ*, n.d.):

1. Intercambio predeterminado: un intercambio directo sin nombre donde la clave de enrutamiento es simplemente el nombre de la cola.
2. Intercambio Directo: En este tipo se entregan mensajes a las colas en función de la clave de enrutamiento. Los intercambios directos se utilizan a menudo para distribuir tareas entre varios trabajadores (instancias de la misma aplicación) de forma round-robin.
3. Intercambio *Fanout*: Se enrutan los mensajes a todas las colas que están vinculadas al productor y se ignora la clave de enrutamiento. Los intercambios *fanout* son ideales para el enrutamiento de mensajes de difusión.

La cola en RabbitMQ es un buffer para el almacenamiento de mensajes y estos buffers se nombran para que la referencia de las aplicaciones sea más fácil. Una cola se define por las siguientes propiedades:

- Nombre
- *Durable*: hace que los mensajes en la cola se conserven en disco y que no desaparezcan en caso de que se reinicie un nodo.
- *Exclusive*: solo puede ser usado en una conexión en concreto y al cerrar esta conexión se borra la información.
- *Auto-Delete*: se elimina automáticamente cuando el último consumidor que se suscribió cancela la suscripción.

Finalmente, el consumidor es configurado para leer mensajes de las colas y con el nivel de calidad de servicio de confirmación de los mensajes que se desee. Debido a que RabbitMQ es

un *broker* de mensajería optimizado para ser usado en dispositivos IoT y que proporciona una abundante cantidad de funcionalidades que permiten encolar mensajes para ser procesados, es una tecnología ideal para el envío de datos que son procesados por un sistema con la arquitectura Kappa.

Apache Flink

Apache Flink es un software que proporciona librerías y un motor de procesamiento distribuido escrito en Java y Scala para el procesamiento de flujos de datos. Flink es capaz de procesar datos por lotes o en un flujo continuo. Las aplicaciones de Flink son tolerantes a fallos e incluyen la posibilidad de establecer un nivel de calidad de servicio con confirmación de los mensajes (Katsifodimos & Schelter, 2016). Flink no proporciona su propio sistema de almacenamiento de datos para persistir los resultados de las operaciones sobre el flujo de datos, pero proporciona conectores para fuentes o sumidero de datos a sistemas como Amazon Kinesis, Apache Kafka, HDFS, Apache Cassandra, ElasticSearch, RabbitMQ, y JDBC (Apache Foundation, n.d.). Todas estas funcionalidades hacen que Apache Flink sea la herramienta ideal para la capa de procesamiento de un sistema con una arquitectura Kappa.

La arquitectura de Flink (ver Figura 4 Comunicación *JobManager* y *TaskManagers* en Flink) consiste en un *cluster* donde un nodo maestro o *JobManager* se encarga de la planificación de las tareas y de la gestión de los recursos de los nodos esclavos o *TaskManager*. Los nodos esclavos le comunican el estado vía métricas que contienen el rendimiento actual del nodo al nodo maestro para que sea capaz de realizar la gestión de recursos y la planificación de tareas.

Figura 4 Comunicación JobManager y TaskManagers en Flink

Un programa que utiliza la API DataStream de Apache Flink sigue la siguiente estructura:

Configuración de una fuente de datos:

Como fue mencionado previamente, Flink proporciona una variedad de conectores para poder configurar tanto la fuente como el sumidero.

Transformaciones sobre el flujo de datos:

Tras la configuración de la fuente de datos, Flink proporciona una serie de transformaciones que se pueden realizar sobre el flujo de datos. Estas transformaciones se pueden realizar a nivel de instancia o mediante procesamiento de ventanas de datos.

Ventanas: Es posible asignar un conjunto de datos a una ventana temporal. Esta asignación se puede realizar en base a una clave (llamada *key* en Flink) que consiste en una propiedad del conjunto de datos que identifica inequívocamente a un conjunto de datos, en el caso de este trabajo la clave utilizada es el identificador del vehículo que sirve para asignar todos los datos que provienen de un vehículo a una ventana. Flink proporciona tres tipos de ventanas:

- *Tumbling Window*: donde periódicamente se asignan datos a una ventana de un tiempo determinado por el usuario,
- *Sliding Windows*: donde de forma similar a las *Tumbling Windows*, el tamaño de las ventanas se configura por el usuario, pero además se debe configurar el parámetro de deslizamiento de ventana que controla la frecuencia con la que se inicia una ventana deslizante. Por lo tanto, las ventanas deslizantes pueden superponerse en el caso que el parámetro de deslizamiento es más pequeño que el tamaño de la ventana.
- *Global Windows*: donde se asignan todos los elementos con la misma clave a la misma ventana global única. Este tipo de ventanas solo pueden funcionar si el usuario implementa un *Trigger* personalizado. De lo contrario, no se realizará ningún cálculo, ya que la ventana global no tiene un final en el que podamos agregar y procesar los elementos.

Triggers: Un *Trigger* determina cuándo los datos que componen una ventana están listos para ser procesados. En el caso del desarrollo de este trabajo se implementó un *Trigger* personalizado donde se iba registrando el identificador del estado o "IdEstado" de cada evento que enviaba el vehículo y si este identificador era "37+", "38+", "50+", o "243" entonces se asignaban todos los eventos hasta ese punto a una ventana y se procesaban.

Operaciones sobre el conjunto de datos: una vez los datos se han agrupado con el uso de una ventana y un *Trigger* es posible realizar operaciones de agregación sobre ellos utilizando funciones como la *ReduceFunction*, por ejemplo, que permite que dos elementos de la fuente de datos se combinen para producir un elemento de salida del mismo tipo.

Configuración de sumidero:

Envío de los datos a un sumidero que puede consistir en una redirección a Standard Output, escritura en un fichero, o persistir los datos con alguna otra tecnología usando un conector. En cuanto a la solución propuesta en este trabajo se usa el conector de Cassandra para persistir los tramos calculados.

Apache Cassandra

Apache Cassandra es una base de datos NoSQL, distribuida, escalable y de alta disponibilidad. Es una tecnología líder en su ámbito y es utilizada para administrar algunos de los conjuntos de datos más grandes del mundo en *cluster*s e implementados en múltiples centros de datos (Avinash Lakshman, n.d.). Los casos de uso más comunes para la base de datos Cassandra consisten en: catálogos de productos, datos de sensores IoT, mensajería y redes sociales, recomendaciones, personalización, detección de fraudes y otras aplicaciones como las de datos de series temporales (Chebotko et al., 2015).

Cassandra utiliza el Cassandra Query Language (CQL) que permite a los usuarios una manera de consultar los datos almacenados dentro de un grupo de nodos Cassandra. Con este lenguaje es posible crear y actualizar el esquema de la base de datos y acceder a los datos mediante una sintaxis similar a la de SQL. En Cassandra los *keyspace* contienen las tablas con los conjuntos de datos y además son conjunto de datos que pueden ser replicados entre los diferentes nodos. La

replicación es el número de copias guardadas por *cluster* (Features | Apache Cassandra Documentation, n.d.).

Una de las principales características que hacen que Cassandra se destaque dentro del mundo de las bases de datos es la manera en la que los datos se almacenan y ordenan en base a su clave primaria. La clave primaria en Cassandra consta de una o más claves de partición y cero o más claves de agrupación. El orden de estos componentes siempre pone primero la clave de partición y luego la clave de agrupación. La clave de partición distribuye o particiona los datos y la de agrupamiento los ordena dentro de cada partición.

Cassandra tiene una arquitectura de tipo token ring donde la información se agrupa en base a la clave de partición. Los datos se colocan en cada nodo según el valor de la clave de partición y el rango del que es responsable el nodo. Este rango que cada nodo debe almacenar se obtiene a través de una técnica de *hashing* llamada *hash* consistente que consiste en generar un *hash* independientemente de la cantidad de servidores con el fin de distribuir el almacenamiento entre los nodos de un sistema distribuido (Datastax, n.d.).

La arquitectura de un *cluster* de Cassandra consiste en un grupo de nodos (ver Figura 5 Replicación de datos en nodos) que almacenan datos de tal manera que las lecturas y escrituras estén optimizadas y los datos estén repartidos de manera uniforme entre los nodos.

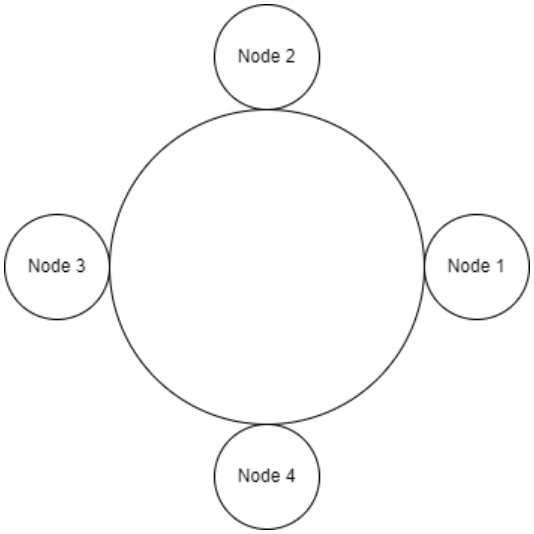

Figura 5 Replicación de datos en nodos

La manera en que Cassandra gestiona la repartición del almacenamiento de datos y el agrupamiento de estos lo convierten en una tecnología ideal para almacenar datos que provienen de dispositivos IoT. En la solución implementada en este trabajo esto es necesario ya que podría agilizar las consultas en base a tramos que ocurren en rangos de fecha determinados y para determinar tramos de ciertos vehículos y conductores.

Pandas

Esta librería de Python ha sido utilizada para poder crear los subconjuntos de datos usados para el desarrollo y las pruebas de rendimiento. Pandas permite la gestión de grandes volúmenes de datos mediante los objetos DataFrame y diversas operaciones que se pueden realizar sobre los conjuntos y subconjuntos de datos.

Prometheus y Grafana

Prometheus es una solución de monitorización para registrar, almacenar, y visualizar datos en forma de series temporales. Recopila, organiza y almacena métricas junto con identificadores únicos y marcas de tiempo. Prometheus es un software de código abierto que recopila métricas de los objetivos mediante el método Pull HTTP de métricas. Las ventajas de Prometheus incluyen:

- Integración con Grafana. Esto brinda la sinergia entre la recopilación de métricas las visualizaciones con cuadros de mando dinámicos.
- PromQL. Un lenguaje de consulta para obtener y analizar datos de métricas de Prometheus.

Grafana permite a un usuario consultar, visualizar, alertar y comprender sus métricas dónde estén almacenadas. Permite la conexión con Prometheus para poder así visualizar con cuadros de mando personalizados o predefinidos los datos que Prometheus recopila. Adicionalmente, proporciona la funcionalidad de guardar cuadros de mando juntos con sus resultados en sus servidores para poder visualizarlos en el futuro ya que Prometheus no persiste los datos que recopila.

Diseño e Implementación de la Solución

Fagor S.L.U. ha proporcionado datos recopilados por los ordenadores de viaje utilizados en los camiones en forma de dos ficheros que recopilan datos de una flota de logística por el periodo de enero a marzo del año 2021.

Requisitos

Fagor solicita desarrollar un prototipo para realizar el cómputo de los tramos y de los perfiles de conducción en cuasi tiempo real de manera distribuida con el fin de poder atender a la demanda creciente que estima. El algoritmo actual que realiza la generación de estos perfiles es:

1. Determinar cada tramo recorrido, esto es, identificar el vehículo, conductor, fecha, distancia recorrida y velocidad media desde el arranque del motor hasta la parada o fin del trayecto.
2. Asociar las medidas del ordenador de viaje para computar los parámetros de comportamiento en la conducción.
3. Generar el perfil de conducción en base al modelo del vehículo.

Para ello Fagor proporciona dos ficheros que recopilan las medidas de los ordenadores de viaje de la flota de uno de sus clientes entre los meses de enero y marzo de 2021.

Fuentes de Datos

La Tabla 1 Resumen del tamaño de los ficheros muestra el volumen de los datos que se contienen en los dos ficheros de tipo CSV que proporciona Fagor.

Fichero	Número de filas	Número de vehículos
EventosHistorico	155.680.01	4.347
Cantrama	8.061.544	4.317

Tabla 1 Resumen del tamaño de los ficheros

CSV 1: EventosHistorico

Los datos proporcionados consisten en un fichero titulado “EventosHistorico” que recopila los eventos de los camiones de la flota con una frecuencia de cinco minutos. Los eventos recopilados consisten en identificadores únicos para el evento, el conductor, y el vehículo además de la distancia recorrida, la fecha del evento con granularidad hasta el segundo, velocidad, longitud, latitud, y un identificador del estado del evento. El identificador del estado del evento, denominado “IdEstado” en el histórico de datos es uno de los datos que será el más importante para calcular el tramo y necesario para poder delimitar el tramo y así evaluar el comportamiento del conductor en el tramo. Fagor utiliza cuatro estados distintos para señalizar el final de un tramo del recorrido del vehículo repartidor, siendo estos los siguientes:

- “37+”: identificador que marca la entrada a una zona.
- “38+”: identificador que marca la salida de una zona.
- “50+”: identificador que marca el cambio de conductor.
- “243”: identificador que marca el cambio de país.

Estos identificadores son utilizados por Fagor para marcar el cierre de un tramo y, por lo tanto, con este identificador se cierra un tramo para este vehículo y conductor y se procede a realizar los cálculos sobre la diferencia entre la distancia, velocidad, posición geográfica, y las fechas de inicio y fin del trayecto o tramo.

CSV 2: Cantrama

Para la segunda parte del cálculo se ha introducido un nuevo conjunto de datos llamado "Cantrama" que proporcionan información acerca de 21 medidas de sensores adicionales. Estas medidas adicionales vienen de sensores en los vehículos y además están relacionadas con conjunto de datos previo de "EventosHistorico" a través del identificador del vehículo y conductor. Las medidas adicionales y sus descripciones son las siguientes:

- **CruiseActive**: Control de crucero usado en los vehículos para mantener una velocidad constante.
- **RPMExcesivas**: Contador que detecta las veces que el vehículo se ha excedido en las revoluciones por minuto recomendadas por el fabricante.
- **FrenadasBruscas**: Número de veces que se detecta una deceleración superior a 1,5m/s². Muchas frenadas bruscas denotan generalmente una conducción agresiva y puede ser peligrosa. El parámetro debe tender a cero aunque existen situaciones inevitables en las que se debe realizar una frenada brusca.
- **AceleracionesBruscas**: Tiempo en segundos que se detecta una aceleración superior a un valor fijado en 1.5m/s². Es un parámetro que debe ser lo más bajo posible, aunque existen circunstancias en las que puede ser necesaria por motivos de seguridad.
- **CNoPredictiva2**: Número de veces que se suelta el acelerador y se pisa el freno en un tiempo prefijado. Con este indicador se mide la falta de anticipación en una conducción eficiente, debido a que cuando se prevé la necesidad de reducir velocidad, se requiere soltar el acelerador y dejar circular el vehículo con su inercia (consumo 0) y luego utilizar el freno según necesidad.
- **ZRoja2**: Tiempo en segundos que el vehículo circula en un régimen de revoluciones por minuto superior al límite marcado por cada modelo de vehículo.
- **ZMasVerde2**: Tiempo en segundos que el vehículo circula en un régimen de revoluciones por minuto superior a la zona que se considera óptima (zona verde), pero sin llegar al límite marcado por cada modelo de vehículo (zona roja). Este parámetro se considera negativo para una conducción eficiente, pero sin llegar al tener el límite configurado por cada modelo.
- **RalInec2**: Número de veces que se detecta que el vehículo permanece a ralentí (motor arrancado y velocidad 0) durante un periodo superior a 300 segundos. Desde el punto de vista de la eficiencia, un tiempo superior se considera que no tiene sentido, ya que en 300 segundos, el motor estará en una temperatura óptima para su utilización.
- **TiempoConduccionCrucero2** Tiempo en segundos utilizando el control de crucero.
- **MetrosAscendidos2**: Ascenso acumulado.
- **MetrosDescendidos2**: Descenso acumulado.
- **Odometro2**: Contador de kilómetros del camión que calcula distancia total.
- **TotalFuel2**: Cantidad de combustible total consumido.
- **TiempoRal2**: Tiempo en ralentí.
- **ConsumoRal2**: Consumo de combustible en estado de ralentí.
- **TiempoConduccion2**: Tiempo que se ha acumulado en conducción.
- **NFreno3**: Cantidad de usos del freno.
- **NEmbrague3**: Cantidad de usos del embrague.
- **TiempoMotor3**: Tiempo que ha estado en funcionamiento el motor.

Diseño arquitectural

La arquitectura de la solución está compuesta por RabbitMQ como el gestor de colas, Apache Flink como software de procesamiento distribuido y paralelizado, y Apache Cassandra para persistir los indicadores asociados con la de conducción como se puede ver en la Figura 6 Diseño de la solución.

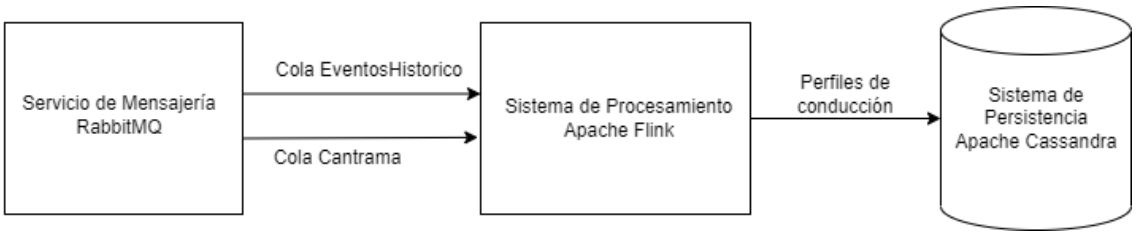

Figura 6 Diseño de la solución

Servicio de Mensajería: RabbitMQ

RabbitMQ se utilizará en este modelo de la plataforma como el *broker* de mensajería que se encarga de implementar colas de mensajería para el envío de los datos de los dispositivos IoT de una flota de vehículos. A través de ficheros formato CSV que han sido proporcionados es posible realizar un envío de los datos con la estructura real a ser procesados Estos datos serán enviados a través de dos colas de mensajería, una para los datos del CSV "EventosHistorico" y otra para los datos de "Cantrama".

La solución distribuida a desplegar en la nube consistirá en un *broker* de mensajería como parte del sistema en la nube de Google Compute Engine. Se realizará un envío de los datos desde el computador personal que tiene el contenido de los CSVs y, por lo tanto, tiene como rol simular el envío simultaneo de datos por parte de los camiones a través de scripts de Python. El segundo *broker* de RabbitMQ recibe los datos y actúa como un buffer de datos donde llegan los parámetros de ambas colas para ser ingeridas por el sistema de procesamiento.

Sistema de Procesamiento: Apache Flink

La parte de procesamiento de datos utilizará Apache Flink para ingerir datos de RabbitMQ, procesarlos de manera que delimiten los tramos de vehículos con los perfiles de conducción, y enviarlos a la base de datos Cassandra para persistirlos. El flujo de procesamiento consiste en los cuatro pasos representados en la Figura 7 Funcionamiento interno de Flink:

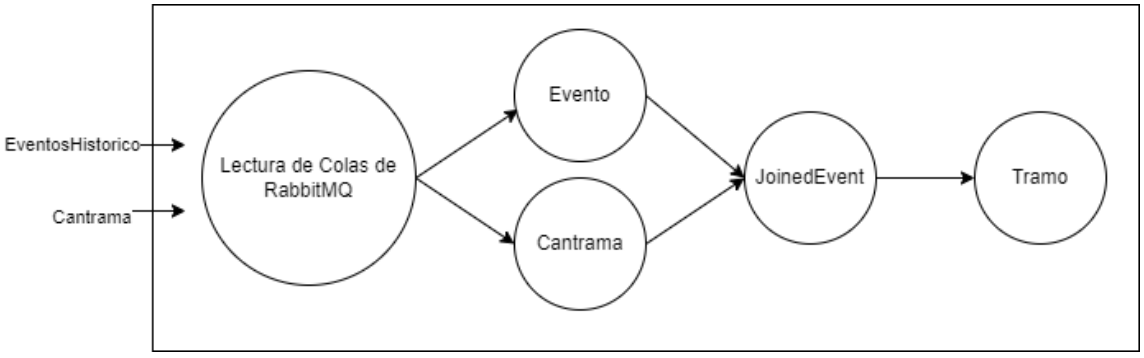

Figura 7 Funcionamiento interno de Flink

El primer paso en este diseño de Flink será ingerir los datos de las colas de RabbitMQ a través del conector que proporciona la librería de conectores de Apache Flink. Este nodo recibirá los datos en dos colas y al configurar los conectores de RabbitMQ para Apache Flink con información

sobre la dirección IP, puerto, y colas de datos este es capaz de consumir los parámetros que llegan al *broker* de RabbitMQ en la nube. La configuración por defecto del conector RabbitMQ y Apache Flink tiene los *acknowledgments* de la calidad de servicio como *at-least once*. Se utiliza una calidad de servicio de *at-least once* en este caso debido a que el volumen y ritmo de envío de los datos no causa problemas de mensajes duplicados.

Una vez los datos de RabbitMQ sean ingeridos, se crearán objetos en base a los parámetros que provienen de las dos colas con los datos de "EventosHistorico" y "Cantrama" como se puede ver en el pseudocódigo de la Figura 8 Pseudocódigo *Join*. Si el objeto Cantrama es distinto a nulo y ambos registros han ocurrido en la misma fecha, hora, y minuto se procederá a hacer un *Join* para los datos que provienen de ese vehículo.

```
cantrama = subscribe to CANTRAMA RABBITMQ datastream
events = subscribe to EVENTS RABBITMQ datastream
joinedStream  =
    events.join(cantrama)
    .where(idVehicle)
    .equalsTo(IdVehicle)
    .window(TumblingWindow(1.9s)
    .joinFunction(joinFn)

def joinFn(event, cantrama):
    if cantrama != null && compareDates(event, cantrama) then
        joined = new JoinedEvent(event)
        for each parameter p in event do
            joined.p = p
        end
        for each parameter p in cantrama do
            joined.p = p
        end
    end
    return joined

def compareDates(event, cantrama):
    dateEvent = getattr(event,'date')
    dateCantrama = getattr(cantrama,'date')
    dateEvent = dateEvent.format("yyyy-MM-dd HH:mm")
    dateCantrama = dateCantrama.format("yyyy-MM-dd HH:mm")
    minuteMatch = True if dateEvent.equals(dateCantrama) else False
```

Figura 8 Pseudocódigo Join

Al tener los parámetros que provienen de ambas colas en un solo objeto se procederá a particionar el flujo de datos en base al identificador del vehículo para posteriormente analizar los tramos de cada vehículo. Para agrupar los parámetros de cada vehículo para ser analizados se utilizarán ventanas globales debido a que el final de un tramo es dictado por los estados que se explicaron en el apartado de Fuentes de Datos CSV 1: EventosHistorico. Al detectar el final de un tramo a través de uno de estos estados, se procederá a procesar la información de esta ventana para generar un tramo y persistir esta información en una base de datos.

Sistema de Persistencia: Apache Cassandra

Para persistir los datos procesados se utiliza Cassandra como base de datos NoSQL. Para almacenar los datos se utiliza una clave compuesta y una clave de agrupación para ordenar los datos. La clave compuesta consiste en el identificador de vehículo y el del conductor para almacenar los tramos que pertenecen a un vehículo y conductor en concreto. Adicionalmente, la clave de agrupación consiste en la fecha de inicio del tramo. La fecha de inicio es la fecha del primer JoinedEvent que entró a la ventana donde se convierten los JoinedEvents a un tramo. Con esta estructura se almacenan los datos de un vehículo y conductor y además se almacenarán en la misma partición de datos para que las consultas sobre estos datos sean más eficientes.

La Figura 9 Modelo RAI4.0 correspondiente al despliegue muestra la manera en que las diferentes partes se conectan y dependen entre sí. En este diagrama cada instancia de un servicio es alojado en un *NodeCluster* que apunta al nodo donde se hará su despliegue y al ser un despliegue en un solo nodo todas apuntan al mismo nodo de procesamiento. Adicionalmente, las instancias de Rabbit, Flink, y Cassandra tienen sus propias configuraciones (Rabbit conf, Cassandra conf, Flink conf) dentro de este mismo nodo.

Las tareas de Flink de recibir los datos de las colas de mensajería y el cálculo de los tramos con el perfil de conducción (*Decode and Join* y *Trip Calculation*) están vinculadas al *workflow* principal de Flink que se ejecuta en un solo nodo físico de procesamiento. El proceso de decodificar consiste en interpretar como un String cada línea de CSV que envía Rabbit. Por lo tanto, desde Flink se realiza la operación de los datos que provienen de los ordenadores de viaje (e.g. identificador del vehículo, conductor, velocidad, distancia, etc.) para almacenarlos en un objeto Evento o Cantrama dependiendo de la cola de donde provengan estos datos. Al tenerlos decodificados y almacenados dentro de objetos Java, se procede a realizar la operación de unión entre los objetos y así crear un objeto JoinedEvent que contiene todos los datos. Finalmente, Flink realiza la verificación del identificador de estados dentro de la ventana para detectar un identificador que indique uno de los estados que señalizan el final de un tramo. Al tener este tramo, se envían sus datos a Cassandra para ser almacenados por identificador del vehículo y conductor y agrupados en la misma partición en base a la fecha de inicio del tramo.

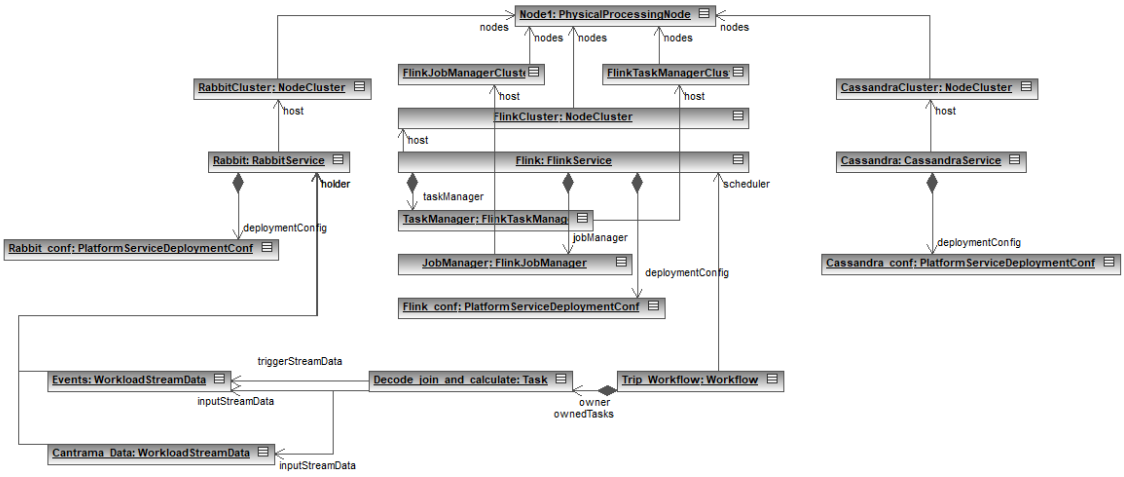

Figura 9 Modelo RAI4.0 correspondiente al despliegue en un solo nodo

La Figura 10 Modelo RAI 4.0 correspondiente al despliegue distribuido también contiene un modelo diseñado a partir del metamodeo RAI4.0, pero adaptado a un despliegue distribuido al

contener las instancias y configuraciones de los sistemas dentro de diferentes nodos de procesamiento virtuales. Los *cluster* de RabbitMQ, Flink, y Cassandra están alojados en distintos *VirtualProcessingNodes* en vez de estar dentro de un solo nodo. En el caso de Flink se tienen diferentes nodos virtuales de procesamiento debido a que se han distribuido en ellas los *JobManager* y los *TaskManagers.* El *JobManager* forma parte del *JobManager Cluster* y los *TaskManagers* pertenecen al *TaskManager Cluster* que en si forman parte todos del mismo *Flink Cluster.* Los elementos *VirtualProcessingNode* del diagrama serán máquinas virtuales que ejecutan una instancia de RabbitMQ, Apache Flink (JobManager o *TaskManagers*) o Cassandra. El diseño del *workflow* a procesar por el sistema es el mismo al despliegue de un nodo pero adaptado a un entorno distribuido como el que se puede implementar con las máquinas virtuales de un servicio como GCE.

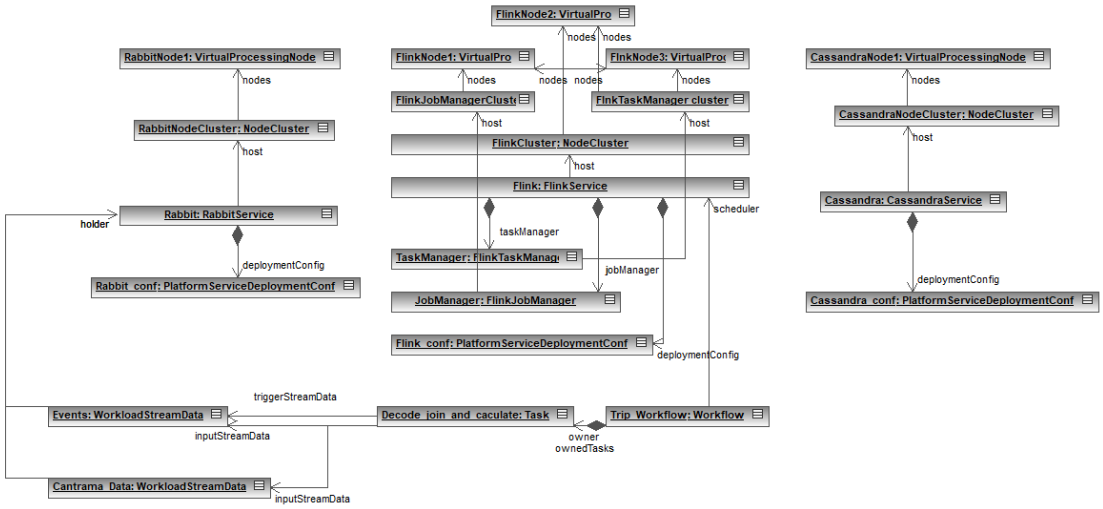

Figura 10 Modelo RAI 4.0 correspondiente al despliegue distribuido

Implementación

RabbitMQ

Para realizar el envío de los datos de los vehículos que se encuentran en los CSVs "EventosHistorico" y "Cantrama" se han realizado scripts en Python que permiten leer los ficheros, crear colas, y publicar los datos de manera continua y planificada. Para realizar esto se ha utilizado la liberaría Pika de Python que proporciona una interfaz para poder interactuar con el servidor de RabbitMQ ejecutando en local tanto como en la nube. Las diferencias implementadas en el desarrollo para las pruebas en local y en la nube consisten en distintas configuraciones del usuario, contraseña, y dirección IP. Para ejecutar las pruebas en local se ha utilizado un usuario llamado *guest* que es el usuario que por defecto está configurado en RabbitMQ para envíos a la interfaz de *loopback* de las máquinas y por otra parte usando la dirección de la interfaz de *loopback* como el parámetro de host. Para las pruebas de rendimiento sobre el sistema distribuido se han implementado distintos scripts con el fin de simular el envío de 30 vehículos distintos que envían simultáneamente sus datos al servidor RabbitMQ en la nube para ser consumidos por los *TaskManager* de Flink. La Figura 11 Script RabbitMQ EventosHistorico y la Figura 12 Script RabbitMQ Cantrama muestran los scripts de Python que son responsables de leer los ficheros CSV, crear las colas, publicar los datos, y adicionalmente se ha incluido una parte del script de Bash que levantaba 60 procesos distintos para simular el envío de datos de "EventosHistorico" y "Cantrama" por cada vehículo.

```
#!/usr/bin/python3
import pika, sys, os, time
from datetime import datetime

filename =
'/home/sebastian/Desktop/TestingCSVs/event_df'+str(sys.argv[1])+'.csv'
rabbit_ip = str(sys.argv[2])
counter = 0

credentials_rabbit = pika.PlainCredentials('admin', 'admin')
connection = pika.BlockingConnection(pika.ConnectionParameters(host =
rabbit_ip, port = 5672, virtual_host = '/', credentials =
credentials_rabbit))
channel = connection.channel()

channel.queue_declare(queue='historico', durable=True)

with open(filename, 'r') as file:
    next(file)

    for line in file:
        row_content = line.strip()
        time.sleep(1.73)
        channel.basic_publish(exchange='', routing_key='historico',
body=row_content)
        counter+=1

now = datetime.now()
current_time = now.strftime("%H:%M:%S")

print(counter)
print("[x] Sent all rows in CSV")
print("Current Time =", current_time)
connection.close()
```

Figura 11 Script RabbitMQ EventosHistorico

```
#!/usr/bin/python3
import pika, sys, os, time
from datetime import datetime

filename =
'/home/sebastian/Desktop/TestingCSVs/cantrama_df'+str(sys.argv[1])+'.csv'
rabbit_ip = str(sys.argv[2])
counter = 0

credentials_rabbit = pika.PlainCredentials('admin', 'admin')
connection = pika.BlockingConnection(pika.ConnectionParameters(host =
rabbit_ip, port = 5672, virtual_host = '/', credentials =
credentials_rabbit))
channel = connection.channel()

channel.queue_declare(queue='cantrama', durable=True)

with open(filename, 'r') as file:
    next(file)

    for line in file:
        row_content = line.strip()
        time.sleep(1.73)
        channel.basic_publish(exchange='', routing_key='cantrama',
body=row_content)
        counter+=1

now = datetime.now()
current_time = now.strftime("%H:%M:%S")

print(counter)
print("[x] Sent all rows in CSV")
print("Current Time =", current_time)
connection.close()
```

Figura 12 Script RabbitMQ Cantrama

Apache Flink

Para poder procesar estos datos con Flink se han creado 7 clases Java, unas para encapsular la lógica de los datos que provienen de los CSVs y otras que proporcionan métodos para el procesamiento de estos datos. Cada una de estas clases tiene como propósito facilitar el procesamiento de los datos al ser instanciada e inicializada con datos que provienen de RabbitMQ y el código completo se encuentra disponible en github (*GitHub - Sgb597/Rabbit*, n.d.). Para el procesamiento de los datos se han creado 4 clases:

1. Event: Esta clase contiene como atributos los campos de interés de la cola de RabbitMQ que envía los datos del CSV EventosHistorico.
2. Cantrama: Esta clase contiene como atributos los campos de interés de la cola de RabbitMQ que envía los datos del CSV Cantrama.

3. JoinedEvent: Después de realizar un *Join* entre los datos de EventosHistorico y Cantrama se crea un JoinedEvent que tiene como atributos todos los atributos de las clases Event y Cantrama.
4. Tramo: Clase utilizada dentro de la TramoCalculationFunction para almacenar los resultados del procesamiento de los JoinedEvent.

Las clases que contienen métodos que son utilizados para el procesamiento de los datos son:

TramoTrigger: Esta clase extiende la clase *Trigger* de la librería de *streaming* de Apache Flink para poder *Override* el método onElement. Con el uso del método onElement esta clase proporciona la funcionalidad de poder obtener el JoinedEvent más reciente que ha entrado en la ventana global y el anterior a este y realizar una operación sobre ellos. El primer paso de la lógica implementada consiste en revisar si el idEstado del JoinedEvent anterior al actual es distinto de null. Si el JoinedEvent anterior no tiene idEstado significa que el JoinedEvent que se está procesando es el primero de la ventana y, por lo tanto, ningún calculo se debería realizar sobre él. Si el JoinedEvent anterior sí tiene un idEstado entonces se procede a revisar el valor del idEstado del JoinedEvent más reciente en la ventana y en el caso que este identificador sea uno de los cuatro que indican la finalización de un tramo entonces el *Trigger* se dispara, enviando todos los JoinedEvent de la ventana actual a ser procesados por TramoCalculationFunction y elimina los JoinedEvent de la ventana actual para poder iniciar una nueva.

TramoCalculationFunction: Se encarga de procesar los JoinedEvent de la ventana cuando TramoTrigger detecta un tipo de evento que marca el final de un tramo. El primer paso dentro de la clase es convertir el tipo del objeto de la ventana que ha enviado el *Trigger* de un Iterable a un ArrayList para poder extraer el primer y último elemento. Al extraer el primer y último JoinedEvent que estaban dentro de esta ventana se procede a obtener la diferencia en el tiempo entre ambos, la diferencia en su distancia, la velocidad, fecha de inicio, fecha final, identificador de conductor y vehículo, y obtener la resta de todos los campos de Cantrama que se encontraban en ambos. Al obtener todos estos resultados se crea un objeto Tramo donde se almacenan todas estas propiedades y se procede a recolectar este objeto mediante un *Collector* que emite estos datos al sumidero de Cassandra.

***Streaming*Job**: La clase *Streaming*Job contiene el método main donde se ejecuta la lógica de la aplicación Java. La lógica está dividida en tres partes:

Conexión y procesamiento de datos de RabbitMQ: Para poder conectarse a las colas de RabbitMQ, Flink necesita configurar los parámetros de la IP donde se ejecuta el servidor RabbitMQ, el virtualHost, el usuario y contraseña que ejecuta este proceso dentro de RabbitMQ, el nombre de la cola, y el puerto de envío. Con esta configuración se puede crear dos objetos de tipo DataStream que tienen como fuente de datos un RMQSource (una fuente de datos que proviene de RabbitMQ) con la configuración adecuada para consumir los datos (ver Figura 13 Lectura de RabbitMQ).

Se han creado dos DataStream de este tipo, uno para los datos de EventosHistorico y otro para los datos de Cantrama. Estos objetos DataStream tienen configurado a String el tipo de dato de la fuente de RabbitMQ debido a que Rabbit envía cada fila del CSV como un solo String. Una vez se consumen estos datos de Rabbit, se procede a ejecutar una operación sobre cada uno de estos DataStreams. Debido a que los datos de Rabbit son de tipo String, es necesario convertirlo a objetos donde persistir su información para procesarlos luego. Por lo tanto, a cada DataStream con datos de Rabbit se les aplica una función de flatMap() que lo que hace es convertir estos

datos de String a objetos Event o Cantrama. Estas operaciones hacen uso de dos clases llamadas EventoParser y CantramaParser cuyo código se puede ver en el Anexo en la Figura 25 Clases Parser para creación de objetos. Con estos objetos se procederá a hacer un *Join* entre ellos si son del mismo vehículo y si ocurren dentro de la misma fecha con una granularidad hasta el minuto Figura 14 *Join* de Objeto Event y Cantrama.

```
// the host, port, and queue name to connect to
final String hostname = "10.204.0.7";
final String virtualHost = "/";
final String userName = "admin";
final String password = "admin";
final String historicoQueue = "historico";
final String cantramaQueue = "cantrama";
final int port = 5672;

// creation of streaming execution environment
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

// checkpointing is required for exactly-once or at-least-once guarantees
env.enableCheckpointing(1000);

final RMQConnectionConfig connectionConfig = new
RMQConnectionConfig.Builder()
    .setHost(hostname)
    .setPort(port)
    .setVirtualHost(virtualHost)
    .setUserName(userName)
    .setPassword(password)
    .build();

// read from historico queue
final DataStream<String> historicoTextStream = env
    .addSource(new RMQSource<String>(
        connectionConfig,
        historicoQueue,
        false,
        new SimpleStringSchema()));

// read from cantrama queue
final DataStream<String> cantramaTextStream = env
    .addSource(new RMQSource<String>(
        connectionConfig,
        cantramaQueue,
        false,
        new SimpleStringSchema()));

// create Event object
DataStream<Tuple2<Event, String>> eventStream = historicoTextStream
    .flatMap(new EventoParser());

// create Cantrama object
DataStream<Tuple2<Cantrama, String>> cantramaStream = cantramaTextStream
    .flatMap(new CantramaParser());
```

Figura 13 Lectura de RabbitMQ

***Join* entre objetos Evento y Cantrama**: Una vez se tienen objetos Evento y Cantrama se puede proceder a realizar el agrupamiento o *Join* sobre estos datos. La operación del *Join* consiste en revisar si tanto el objeto Event como el objeto Cantrama tienen el mismo identificador de Vehículo, si caen dentro de la misma ventana, y si ambos objetos tienen la misma fecha. La comparación de fechas tiene una granularidad hasta el minuto, por lo tanto, si cumplen todas estas condiciones se crea un objeto JoinedEvent en base a los datos de ambos.

Esto se ha llevado a cabo de nuevo utilizando la API DataStream de Apache Flink que proporciona una funcionalidad que permite realizar un *Join* sobre un conjunto de datos en base a una *key*, condición, y en una ventana temporal (ver Figura 14 Join de Objeto Event y Cantrama). En el caso de este sistema se ha elegido como criterio del *Join* el identificador de vehículo, con la condición de que ambos registros ocurran en el mismo día, hora, y minuto, y adicionalmente se ha establecido una ventana temporal para la realización del *Join* entre los datos que se reciben de ambas colas. El funcionamiento óptimo de las pruebas que se realizaron para las ventanas de envío y para la operación de *Join* fueron 1.9 segundos para la ventana temporal de tipo *Tumbling Window* de Flink y 1.7 segundos de latencia en cada publicación a las colas de RabbitMQ como es ilustrado en la Figura 15 Ventana temporal para el funcionamiento de la operación de *Join*.

```
DataStream<String> joinStream = eventStream.join(cantramaStream)
    .where(value -> value.f1)
    .equalTo(value -> value.f1)
    .window(TumblingProcessingTimeWindows.of(Time.milliseconds(1900)))
    .apply (new JoinFunction<Tuple2<Event, String>, Tuple2<Cantrama,
String>, String> (){

        private static final long serialVersionUID = 1L;

        @Override
        public String join(Tuple2<Event, String> event, Tuple2<Cantrama,
String> cantrama) {
            String output = "";
            Boolean joinStream = compareDates(event.f0.getFecha()
,cantrama.f0.getFecha());

            if (joinStream) {
                output = outputEventString(event.f0) + "," +
outputCantramaString(cantrama.f0);
            } else {
                output = outputEventString(event.f0);
            }
            output = outputEventString(event.f0) + "," +
outputCantramaString(cantrama.f0);
            return output;
        }
    });

DataStream<Tuple2<JoinedEvent, String>> parsedStream = joinStream
        .flatMap(new Parser());
```

Figura 14 Join de Objeto Event y Cantrama

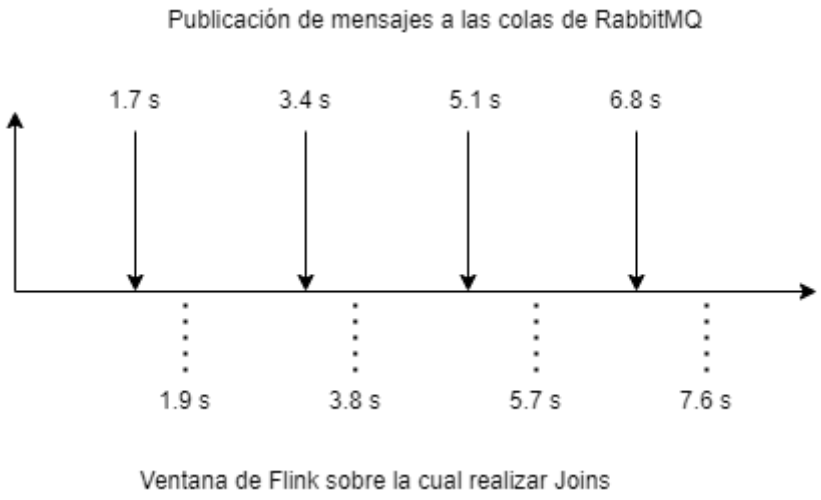

Figura 15 Ventana temporal para el funcionamiento de la operación de Join

Al lograr realizar los *Joins* en base a los criterios previamente mencionados, Flink convierte los dos flujos de datos (Cantrama y Eventos Históricos) en un único flujo de datos que contiene toda la información que proviene de los vehículos repartidores. Una vez se tiene un flujo de datos que contienen toda la información se procede a crear una nueva ventana global en Flink con la funcionalidad de *Global Windows* que es descrita en el apartado de Arquitectura y Tecnologías. Al definir una ventana global estamos creando una ventana que no tiene un límite temporal como era en el caso de la ventana usada para los *Joins* ya que esa era una ventana delimitada por tiempo (una ventana nueva cada 1.9s). Con esta ventana global lo que se necesita es definir un *Trigger* personalizado que le indica a Flink en qué punto cerrar la ventana y proceder a procesar los datos que están dentro de esta ventana. Este *Trigger* personalizado cierra las ventanas cuando se encuentra un conjunto de datos agrupados que tienen un identificador de estado de: 37+, 38+, 50+, y 243. Estos indicadores son usados por Fagor para indicar el fin de un tramo recorrido por este vehículo y conductor.

Creación de tramos: Para la creación de tramos se utiliza la lógica empleada por las clases TramoTrigger y TramoCalculationFunction. Dentro de la ventana global se contienen objetos de tipo JoinedEvent de un vehículo en concreto debido al método keyBy() que permite particionar lógicamente el flujo de datos por vehículo. La instancia de la clase TramoTrigger permite delimitar una ventana de objetos una vez se detecta uno de los estados que marcan el final de un tramo. Al delimitar la ventana, todos los objetos JoinedEvent que componen esta ventana son enviados a procesar a través de la instancia de la clase TramoCalculationFunction (ver Figura 16 Creación de Tramo)

```
DataStream<Tuple7<String, String, Timestamp, Timestamp, Double, Double,
HashMap<String, Double>>> tramoStream = parsedStream
    .keyBy(value -> value.f1)
    .window(GlobalWindows.create())
    .trigger(new TramoTrigger())
    .process(new TramoCalculationFunction());
```

Figura 16 Creación de Tramo

Las operaciones sobre los objetos de la ventana que se realizan en TramoCalculationFunction tienen como resultado los siguientes parámetros:

- Id del Vehículo
- Id del Conductor
- La fecha de inicio del tramo
- La fecha final del tramo
- La distancia recorrida en el tramo por el vehículo
- La velocidad media
- Todas las diferencias de los datos que provienen de Cantrama almacenadas en un HashMap que tiene como claves los nombres (con tipo de dato String) de los campos del CSV de Cantrama y (con tipo de dato Double) el resultado de hacer la resta de estos campos en el ultimo y el primer JoinedEvent de la Ventana.

Persistencia de los resultados: Todos estos datos son luego almacenados en Cassandra usando una clave compuesta que usa el Id del vehículo y del conductor y además se utiliza la fecha final como clave de agrupación para ordenar en base a esta fecha los datos en Cassandra (ver Figura 17 Persistencia de Tramo).

```
CassandraSink.addSink(tramoStream)
.setQuery("INSERT INTO tfm.tramos(idVehiculo , IdConductor , FechaInicio
, FechaFinal , Distancia , Velocidad, tramoData) values (?, ?, ? , ?, ?,
?, ?);")
.setHost("10.204.0.6")
.build();
```

Figura 17 Persistencia de Tramo

Apache Cassandra

La implementación de Cassandra ha variado en las pruebas en un solo nodo y en el despliegue en la nube. Para las pruebas en un solo nodo se ha configurado Cassandra en Docker. Al tener Cassandra siendo ejecutado como un contenedor se facilita probar diferentes configuraciones que son almacenadas como distintos contenedores e imágenes. Para poder ejecutar apropiadamente Cassandra en Docker y poder almacenar datos desde Flink se ha hecho uso de la funcionalidad de redirección de puertos de Docker. Debido a que el puerto donde Cassandra espera recibir datos es el 9042, en las pruebas en un solo nodo se realiza una redirección de puerto del 9042 de la interfaz de *loopback* del computador y el puerto 9042 del contenedor.

Para las pruebas de rendimiento del sistema distribuido en la nube se ha realizado una instalación directamente sobre una máquina virtual para funcionar como un servidor que ejecuta Cassandra. En ambos casos, un solo nodo y en la nube, la configuración con la que se han creado las tablas para almacenar los datos de los tramos que provienen de Flink es la de la Figura 18. Configuración Cassandra.

```
CREATE  KEYSPACE [IF NOT EXISTS] tfm
   WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor'
: 1 }

CREATE TABLE tfm.tramos (
IdConductor text,
IdVehiculo text,
FechaInicio timestamp,
FechaFinal timestamp,
Distancia double,
Velocidad double,
TramoData map<text,double>,
PRIMARY KEY ((IdVehiculo, IdConductor),FechaInicio));
```

Figura 18. Configuración Cassandra

Dentro de Google Compute Engine se han configurado adicionalmente las *Firewall Rules* para añadir una regla de tipo *Ingress* al puerto TCP 9042 para que las máquinas virtuales que funcionan como *TaskManagers* y ejecutan el procesamiento de los datos puedan almacenar los datos en el servidor que ejecuta Cassandra.

Despliegue en un entorno distribuido

La Tabla 2 Especificaciones Instancia GCE contiene la información sobre las máquinas virtuales que han sido contratadas en GCE. Cada uno de los nodos que se han usado en la solución utilizan este tipo de máquina virtual.

	Tipo de CPU	Detalles de CPU	Memoria RAM	Disco	OS
E2-medium	Intel(R) Xeon(R) CPU @ 2.30GHz	Frequency: 2.3 GHz Cache size: 46.1 MB	4GB	10GB	Ubuntu 18.04.4 LTS

Tabla 2 Especificaciones Instancia GCE

Con estas especificaciones se han creado nodos para RabbitMQ, Apache Flink, y Cassandra. Adicionalmente se han utilizado máquinas virtuales que ejecutan Prometheus y Grafana para poder realizar mediciones sobre el rendimiento de Apache Flink y RabbitMQ. Para el nodo encargado de realizar el servicio de mensajería se ha configurado una máquina virtual con RabbitMQ que recibe los mensajes enviados a las colas realizado en el código de la Figura 11 Script RabbitMQ EventosHistorico y la Figura 12 Script RabbitMQ Cantrama. Se han configurado cuatro diferentes máquinas virtuales con Apache Flink. Tres de ellas se han configurado para ejecutar un *TaskManager* de Flink en cada una para realizar la tarea de procesamiento de datos de manera distribuida y la cuarta se ha configurado para ser el *JobManager* que está a cargo de planificar los Jobs para los *TaskManagers*. Adicionalmente se ha configurado una máquina virtual que ejecuta RabbitMQ, Apache Flink, y Cassandra todo en un solo nodo para poder comparar luego las métricas de rendimiento de este nodo con una solución distribuida. El nodo encargado de la persistencia ejecuta Cassandra con el fin de almacenar los tramos con los perfiles de conducción.

Pruebas de Verificación y Validación

Al tener configurada y desplegada la arquitectura de Flink en la nube de Google se procedió a realizar una serie de pruebas con el objetivo de verificar que la arquitectura distribuida y desplegada en la nube es capaz de crear correctamente procesar los tramos de tres vehículos. Se diseñan las siguientes pruebas:

1. Prueba a arquitectura distribuida con 2 *TaskManagers*.
2. Prueba a arquitectura distribuida con 2 *TaskManagers* y simulando un fallo durante el proceso de ejecución al apagar una de las VMs donde se está ejecutando uno de los *TaskManagers*.
3. Prueba a arquitectura distribuida con 2 *TaskManagers*, pero iniciando con 1 *TaskManager* y añadiendo otro nodo en caliente.

Los resultados se almacenan en una tabla distinta de Cassandra dentro del mismo *keyspace* con el fin de poder comparar los resultados. El resultado de las tres pruebas fue que los datos en Cassandra eran idénticos y, por lo tanto, podíamos concluir que tanto en el caso de un fallo y en escalada en caliente Flink es capaz de distribuirse el trabajo de tal manera que se obtenga el mismo resultado. Los resultados de la base de datos se pueden encontrar en el Anexo.

Para poder saber exactamente la cantidad de tramos que debían generar estas pruebas se han utilizado datos de 3 vehículos donde se habían cambiado los datos del identificador de estado de tal manera que se supiera con antelación los resultados que debería dar el algoritmo.

Para estas pruebas y también las pruebas de rendimiento que se analizan en el apartado siguiente se particionaron los datos de tal manera que se tuvieran los datos de cada vehículo en CSVs por separado para así enviar simultáneamente datos de cada CSV/vehículo via RabbitMQ y así simular el comportamiento real de múltiples vehículos enviando información al mismo tiempo.

Para la creación de los ficheros de estas pruebas se ha utilizado Python y la librería Pandas. Uno de los criterios que se ha utilizado para generar estos subconjuntos de datos de los vehículos ha sido que el vehículo haya emitido por lo menos 1000 eventos. El motivo detrás de seleccionar los vehículos que tuvieran más de mil registros en tanto EventosHistorico como en Cantrama ha sido debido a que hay vehículos que aparecen en ambos ficheros que tienen muy pocos eventos emitidos y por lo tanto no son de utilidad en cuanto al análisis. Para lograr particionar los datos de esta manera se ha utilizado el script de Python mostrado en la Figura 19 Generador de Subconjuntos.

```
#!/bin/bash

import pandas as pd
pd.options.display.max_columns = None

event_file = '/home/sebastian/Documents/TFM/data/EventosHistorico.csv'
event_df = pd.read_csv(event_file, sep=";")
event_id_set = set(event_df['Id_Vehiculo'].unique().tolist())

cantrama_file = '/home/sebastian/Documents/TFM/data/Cantrama3.csv'
cantrama_df = pd.read_csv(cantrama_file, sep=";")
cantrama_id_set = set(cantrama_df['Id_Vehiculo'].unique().tolist())

matching_ids = list(event_id_set.intersection(cantrama_id_set))
event_dict =
event_df[event_df['Id_Vehiculo'].isin(matching_ids)]['Id_Vehiculo'].value
_counts().to_dict()
cantrama_dict =
cantrama_df[cantrama_df['Id_Vehiculo'].isin(matching_ids)]['Id_Vehiculo']
.value_counts().to_dict()

ideal_Ids = [key for key in event_dict if event_dict[key] > 1000 and
cantrama_dict[key] > 1000]

#EVENTO HISTORICO CSVs
for i in ideal_Ids:
    _ = event_df[event_df['Id_Vehiculo'] == i]
    path = '/home/sebastian/Desktop/TestingCSVs/event_df'+str(i)+'.csv'
    _.to_csv(path, sep=';', index=False)

#CANTRAMA CSVs
for i in ideal_Ids:
    _ = cantrama_df[cantrama_df['Id_Vehiculo'] == i]
    path =
'/home/sebastian/Desktop/TestingCSVs/cantrama_df'+str(i)+'.csv'
    _.to_csv(path, sep=';', index=False)
```

Figura 19 Generador de Subconjuntos

Este script crea un Set de los identificadores de Vehículo y a través del método intersection se pueden obtener los identificadores que existen en ambos Set de “EventosHistorico” y “Cantrama”. Luego con estos identificadores que aparecen en ambos CSVs se crean diccionarios que contienen como clave el identificador del vehículo y como valor el número de veces que aparece en el CSV. Con este diccionario se procede a crear CSVs que contienen solo los datos de un vehículo en concreto con el fin de luego simular sus envíos.

Para el envío simultaneo se ha desarrollado un script de Bash donde se ejecutan en paralelo múltiples procesos de RabbitMQ que leen de distintos CSVs y así se realizan envíos de datos de una manera que se asemeja a la realidad (ver Figura 20 Script Bash Simulación de Vehículos.

```
#!/bin/bash

RABBIT_IP='34.175.101.210'
(python3 dynamic_event_stream_cloud.py 2416 ${RABBIT_IP}) &
(python3 dynamic_event_stream_cloud.py 3573 ${RABBIT_IP}) &
(python3 dynamic_event_stream_cloud.py 3866 ${RABBIT_IP}) &
(python3 dynamic_event_stream_cloud.py 3569 ${RABBIT_IP}) &
(python3 dynamic_event_stream_cloud.py 3645 ${RABBIT_IP}) &
(python3 dynamic_event_stream_cloud.py 2551 ${RABBIT_IP}) &
(python3 dynamic_event_stream_cloud.py 623 ${RABBIT_IP}) &
(python3 dynamic_event_stream_cloud.py 2883 ${RABBIT_IP}) &
(python3 dynamic_event_stream_cloud.py 2661 ${RABBIT_IP}) &
(python3 dynamic_event_stream_cloud.py 2449 ${RABBIT_IP}) &
(python3 dynamic_event_stream_cloud.py 3784 ${RABBIT_IP}) &
(python3 dynamic_event_stream_cloud.py 1889 ${RABBIT_IP}) &

(python3 dynamic_cantrama_stream_cloud.py 2416 ${RABBIT_IP}) &
(python3 dynamic_cantrama_stream_cloud.py 3573 ${RABBIT_IP}) &
(python3 dynamic_cantrama_stream_cloud.py 3866 ${RABBIT_IP}) &
(python3 dynamic_cantrama_stream_cloud.py 3569 ${RABBIT_IP}) &
(python3 dynamic_cantrama_stream_cloud.py 3645 ${RABBIT_IP}) &
(python3 dynamic_cantrama_stream_cloud.py 2551 ${RABBIT_IP}) &
(python3 dynamic_cantrama_stream_cloud.py 623 ${RABBIT_IP}) &
(python3 dynamic_cantrama_stream_cloud.py 2883 ${RABBIT_IP}) &
(python3 dynamic_cantrama_stream_cloud.py 2661 ${RABBIT_IP}) &
(python3 dynamic_cantrama_stream_cloud.py 2449 ${RABBIT_IP}) &
(python3 dynamic_cantrama_stream_cloud.py 3784 ${RABBIT_IP}) &
(python3 dynamic_cantrama_stream_cloud.py 1889 ${RABBIT_IP}) &
```

Figura 20 Script Bash Simulación de Vehículos

Pruebas de Rendimiento de Tolerancia a Fallo y Escalado en Caliente

Las pruebas de rendimiento realizadas tenían como objetivo analizar el comportamiento de Flink ante la caída de un nodo y la incorporación de un nodo. Se diseñan cuatro pruebas sobre una implementación del sistema en la nube de GCE siguiendo el modelo de la Figura 10 Modelo RAI 4.0 correspondiente al despliegue distribuido. Para las pruebas se utilizaron los datos de los 30 vehículos con más datos de ambos CSVs con el fin de transmitir una carga que simulara el comportamiento de los vehículos reales. Para analizar la carga de Flink durante las pruebas se configuró una instancia con Prometheus y Grafana para recoger métricas y analizarlas posteriormente. Las pruebas fueron las siguientes:

1. Prueba en un nodo: Se realizó la prueba sobre un único nodo en GCE donde ejecutaba en el mismo RabbitMQ, Apache Flink, y Cassandra.
2. Prueba de 4 nodos de Flink: Se ejecutó el *JobManager* de Flink y tres *TaskManagers* por separado y se dejó ejecutar para medir el rendimiento.
3. Prueba de tolerancia a fallos: Se ejecutó el *JobManager* de Flink y tres *TaskManagers* por separado y se apagó manualmente una máquina virtual para analizar el rendimiento del sistema ante este fallo.
4. Prueba de escalado en caliente: Se ejecutó el *JobManager* de Flink y dos *TaskManagers* por separado y mientras los otros *TaskManager* se ejecutaban se ejecutó Flink en otra máquina para añadir otro *TaskManager* al cluster de Flink.

Métricas

A través de Prometheus se obtuvieron datos sobre las siguientes métricas de rendimiento que Flink proporciona sobre el Java Virtual Machine (JVM) del *JobManager* y *TaskManager*:

1. CPU Load: Porcentaje ocupación de la CPU.
2. CPU Time: La media del tiempo de CPU utilizado por la JVM.
3. CPU Threads: Número de threads en media.
4. Heap: Cantidad de memoria de heap utilizada en media (en bytes).
5. Non-Heap: Cantidad de memoria de non-heap utilizada (en bytes). Compuesta por la memoria metaspace, la memoria stack, la cache de bytecode, y datos del recolector de basura.

A través de Prometheus también se obtuvieron las siguientes métricas de RabbitMQ:

1. Channel Acks Uncommited: Número de mensajes que no recibieron *acknowledgement*.
2. Channel Messages Acked Total: Número total de *acknowledgments* que recibe el *broker*.
3. Channel Messages Uncomfirmed: Número de mensajes no confirmados cuando se tiene una configuración de tipo durable en una cola de RabbitMQ hace falta persistir estos mensajes en disco. Las confirmaciones en RabbitMQ son confirmaciones de que el mensaje ha podido ser almacenado en disco y es por lo tanto durable.
4. Incoming Total Bytes: Número de bytes totales que recibe el *broker*.

Pruebas sobre Apache Flink

La Tabla 3 Cantidad de filas e identificadores de finalización de tramo por vehículo contiene la información sobre los ficheros utilizados durante las pruebas de rendimiento donde cada uno de los ficheros contiene datos de un vehículo en concreto. Estos ficheros contienen los datos de los 30 vehículos con más datos en los ficheros de EventosHistorico y Cantrama y se han generado utilizando el script de la Figura 19 Generador de Subconjuntos.

A pesar de disponer con tres meses de datos en los dos ficheros proporcionados por Fagor, se trabajó con estos subconjuntos para obtener métricas de rendimiento en tiempos razonables, pues emular el comportamiento real del sistema no era factible por problemas técnicos y temporales. Los problemas técnicos que llevaron a utilizar un subconjunto de treinta vehículos fueron debido a limitaciones en el equipo personal que se congelaba al intentar realizar un envío de más de treinta ficheros. Los problemas temporales que hacían que el tiempo de cada prueba fuera acotado a una hora y no se pudiera realizar el envío de todos los datos de los 30 vehículos fue debido al tiempo de gratuidad de GCE. Para ejecutar una sola prueba se tendría que esperar alrededor de 30 horas para que se enviaran todos los datos de los ficheros CSVs y al tener diseñadas cuatro pruebas esto supondría que solamente la ejecución de todas tardaría cinco días sin tomar en cuenta que entre cada prueba se recogen y analizan los resultados. Para cuando se finalizó la implementación del despliegue en la nube no se tenía suficiente tiempo de acceso a GCE como para realizar esto y por lo tanto el enfoque ha sido corroborar el funcionamiento correcto del sistema completo y analizar su rendimiento a través de las métricas recopiladas por Prometheus.

Las columnas Tramos Estado 37+ y Tramos Estado 38+ contienen la cantidad de tramos que hay por vehículo utilizado en estas pruebas. Los números de tramos por vehículo son ligeramente distintos a los de las columnas Estado 37+ y Estado 38+ debido a que en el algoritmo desarrollado se toma en cuenta que si la finalización de un tramo es directamente seguido por otro evento que indica un final de tramo no se debe tomar en cuenta ese último evento asociado al identificador repetido debido a que esto generaría una ventana de solamente un evento donde no se tiene un conjunto de eventos sobre los cuales realizar el procesamiento. Es importante tomar en cuenta que en estos subconjuntos de datos no aparecían los estados 50+ y 243 debido a que los 30 vehículos con más datos nunca emitieron estos dos identificadores y por lo tanto no se ven reflejados en la tabla.

Fichero	Número de filas	Estado 37+	Estado 38+	Tramos Estado 37+	Tramos Estado 38+
event_df2416.csv	62507	923	964	895	901
event_df3573.csv	34667	3328	3321	3224	3235
event_df3866.csv	34330	1080	1122	1065	1060
event_df3569.csv	22363	3354	3392	3239	3233
event_df3645.csv	19808	4320	4358	4217	4125
event_df2551.csv	19399	898	893	858	848
event_df623.csv	18772	2272	2295	2218	2211
event_df2883.csv	18112	4095	4039	3946	3809
event_df2661.csv	17148	1977	2091	1892	1870
event_df2449.csv	16883	749	753	742	743
event_df3784.csv	16694	674	692	652	648
event_df1889.csv	16539	1759	1792	1709	1709
event_df3270.csv	16295	3225	3280	3094	3082
event_df4115.csv	16070	392	401	386	381

event_df1953.csv	16061	3737	3744	3586	3487
event_df3669.csv	15779	2111	2167	2078	2073
event_df2773.csv	15772	2683	2739	2609	2607
event_df3634.csv	15714	1879	1924	1833	1810
event_df3562.csv	15446	2551	2608	2517	2506
event_df3038.csv	14923	999	1137	952	947
event_df3152.csv	14820	1418	1525	1361	1356
event_df3668.csv	14688	2337	2436	2284	2272
event_df2789.csv	14643	3349	3330	3220	3150
event_df1667.csv	14612	3579	3608	3426	3382
event_df1125.csv	14122	3008	3003	2626	2347
event_df299.csv	14067	2179	2187	2137	2129
event_df2737.csv	13898	2066	2127	2039	2035
event_df4154.csv	13895	1617	1648	1536	1453
event_df2301.csv	13743	1674	1701	1668	1662
event_df577.csv	13568	2339	2397	2271	2271

Tabla 3 Cantidad de filas e identificadores de finalización de tramo por vehículo

Resultados de la Prueba en un solo nodo

En esta prueba se ejecuta RabbitMQ, Apache Flink, y Cassandra en una sola instancia. El propósito detrás de ejecutar todo en un solo nodo es establecer una línea base para comparar con la arquitectura distribuida. En esta prueba se esperaba ver que la carga sobre el *TaskManager* fuera superior a la de los *TaskManagers* en una arquitectura distribuida, aspecto que se puede ver comparando las medidas de CPU de la carga y tiempo en la Tabla 4 Resultados Prueba un solo nodo y la Tabla 5 Resultados Prueba Flink Distribuido

	CPU Load	CPU Time	CPU Threads	Heap	Non-Heap
JobManager	0.01	0.97 s	67	286 MiB	
TaskManager	0.02	3.09 s	70	207.71 MiB	110.65 MiB

Tabla 4 Resultados Prueba un solo nodo

En esta prueba se ve una carga de CPU del 0.02, superior al resto de los *TaskManagers* en el resto de las pruebas. Adicionalmente, se ve un valor para el tiempo medio utilizado por la JVM de la CPU mayor al resto de los *TaskManagers*. Por la parte de la memoria se pueden ver valores bastante similares al resto de las pruebas.

Resultados de la Prueba 2 Flink Distribuido

Esta prueba tenía como objetivo analizar el rendimiento en cuanto a CPU y memoria de los nodos distribuidos de Apache Flink. Adicionalmente, esta prueba sirve para establecer una línea base para comparaciones con las pruebas de tolerancia a fallos y de escalado en caliente. Se esperaba ver valores reducidos para las métricas debido a que la carga de procesamiento se debería estar repartiendo entre los diferentes *TaskManagers*.

	CPU Load	CPU Time	CPU Threads	Heap	Non-Heap
JobManager	0.0174	1.61 s	70	286MiB	

TaskManager 1	0.009	1.20 s	102	191.72 MiB	128 MiB
TaskManager 2	0.0362	1.95 s	97	243.63 MiB	99.7 MiB
TaskManager 3	0.0154	2.16 s	83	205.3 MiB	93.4 MiB

Tabla 5 Resultados Prueba Flink Distribuido

Al realizar la media de las métricas de carga y tiempo de la CPU en los *TaskManagers* para esta prueba se obtiene un 0.0202 para la carga y 1.77s de tiempo. Esto indica que entre los nodos se utiliza por menos tiempo la CPU al distribuir la carga de procesamiento, pero el volumen de datos no es suficiente como para cambiar la media de la carga.

Resultados de la Prueba 3 Tolerancia a Fallos

Al realizar una prueba de tolerancia a fallos se espera ver la reacción del sistema ante un fallo repentino. Se espera ver que el trabajo realizado por un nodo sea procesado por otro nodo posterior al fallo. Además de esto también se espera detectar problemas con los *acknowledgement*s entre Flink y RabbitMQ debido a que habrá un periodo en el cual los mensajes no estén siendo consumidos por el nodo caído de Flink hasta que sean procesados por otro nodo.

	CPU Load	CPU Time	CPU Threads	Heap	Non-Heap
JobManager	0.0137	1.55 s	70	264 MiB	
TaskManager 1	0.0120	1.44 s	87	214 MiB	109 MiB
TaskManager 2	0.0162	1.23 s	104	181 MiB	141 MiB
TaskManager 3	0.00861	1.16 s	97	276 MiB	113 MiB

Tabla 6 Resultados Prueba Tolerancia a Fallos

El nodo apagado durante el procesamiento ha sido el *TaskManager* 3 que presenta el valor de carga de CPU más bajo de todas las pruebas. Las métricas del resto de los nodos se mantienen relativamente similares a las de la segunda prueba. En cuanto a los *acknowledgements* se puede ver en la Figura 21 Channel Acks Uncommited que hay un pico aproximadamente a las 20:18 cuando se apaga el nodo y no se recibe un *acknowledgment*.

Resultados de la Prueba 4 Escalado en Caliente

La prueba de escalado en caliente consiste en añadir un *TaskManager* al *cluster* de Flink mientras se están realizando los envíos por parte de RabbitMQ y se tienen dos nodos ya procesando estos eventos. Se espera que al añadir este nodo se reparta la carga de dos nodos a tres nodos.

	CPU Load	CPU Time	CPU Threads	Heap	Non-Heap
JobManager	0.0167	1.91 s	71	286 MiB	
TaskManager 1	0.0157	3.27 s	83	158 MiB	93.1 MiB

TaskManager 2	0.0173	1.20 s	102	243 MiB	128 MiB
TaskManager 3	0.0167	1.87 s	97	221 MiB	99.7 MiB

Tabla 7 Resultados Prueba Escalado en Caliente

Al añadir el nodo al *cluster* se han comenzado a repartir vehículos gracias al método de keyBy() que se utiliza para particionar el *stream* de datos. Observando los logs de los *TaskManagers* se puede comprobar que cada nodo generó los tramos para vehículos distintos. En cuanto a las métricas, se ven valores muy similares al resto de las pruebas tanto en las medidas realizadas a la CPU como la memoria.

Los resultados de las pruebas muestran una leve mejora en los tiempos de CPU que utilizan los *TaskManager* al tenerlo distribuido, pero en general los cambios no son apreciables. Esto se debe a que el sistema nunca se llega a saturar con una cantidad considerable de mensajes. Debido a que se necesita hacer un *sleep* de 1.73 segundos en el script de Python que realiza los envíos de los mensajes al servidor de RabbitMQ para que sea posible realizar la operación de *Joins* dentro de Flink y, por lo tanto, en estas pruebas el sistema solo recibe datos de 30 vehículos cada 1.73 segundos, siendo estas 60 líneas de datos debido a que cada vehículo envía datos de EventosHistorico y de Cantrama.

El hecho de que las medidas de memoria sean muy similares en todas las pruebas también se puede explicar con el hecho que los datos son publicados cada 1.73 segundos y, por lo tanto, el espacio que los objetos instanciados ocupan sea eliminado a un ritmo constante por el recolector de basura de la JVM.

Análisis de RabbitMQ

Al conectar Prometheus con el nodo que hace de *broker* RabbitMQ se han obtenido una serie de métricas acerca de los envíos de eventos al nodo de Flink.

En la Figura 21 Channel Acks Uncommited se observa cómo, durante el funcionamiento de la tolerancia a fallos, se ha tenido un caso donde un mensaje no ha recibido el *acknowledgement* en el momento de que el nodo se ha apagado. De lo contrario ha habido un funcionamiento correcto de los *acknowledgments* de los mensajes publicados a las colas.

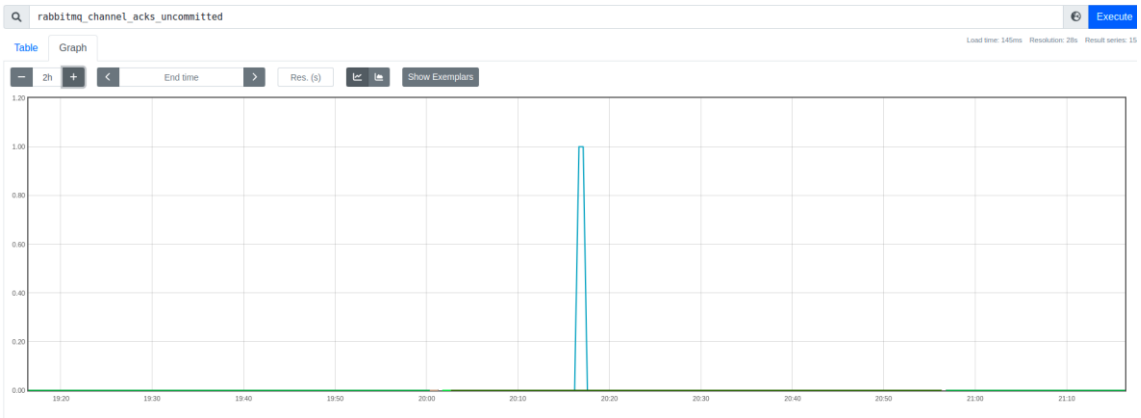

Figura 21 Channel Acks Uncommited

Adicionalmente, la Figura 22 Channel Messages Acked Total muestra un acumulado de los mensajes que han recibido un *acknowledgement* correcto por parte de Flink, excepto por el caso previo.

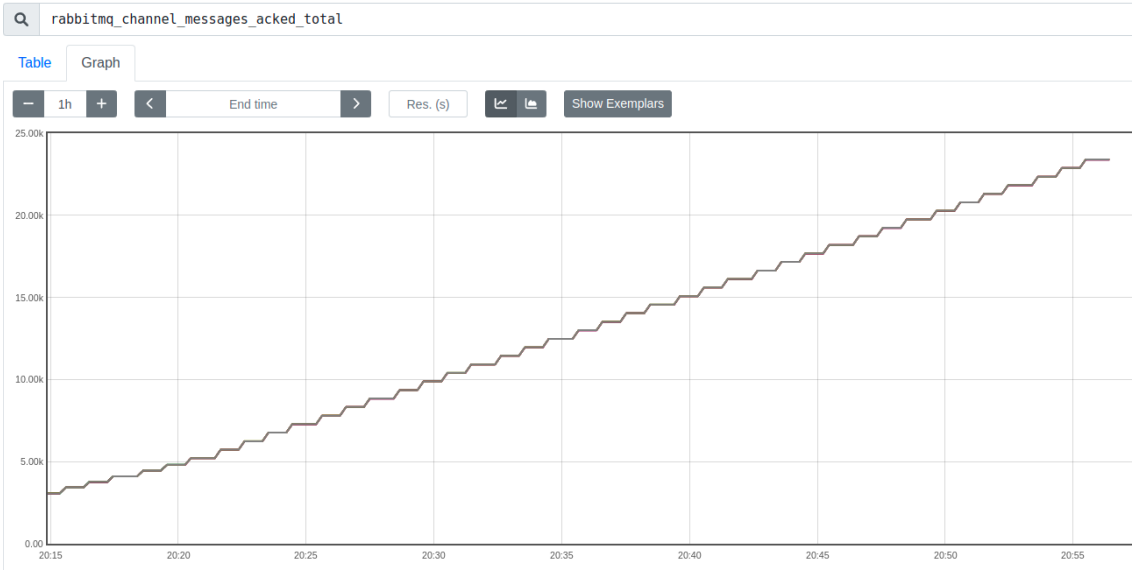

Figura 22 Channel Messages Acked Total

La Figura 23 Messages Uncomfirmed muestra cómo se ha confirmado todos los mensajes para las colas. Sin embargo, en el gráfico se puede ver como a las 20:02 y 21:58 hubo un breve instante donde Prometheus no ha pintado datos en la gráfica de mensajes no confirmados. No es posible indicar a que se debe este error ya que Prometheus no persiste los logs con estas métricas y, adicionalmente, en el resto de las gráficas no se encuentran estos puntos sin datos. Por lo tanto, este error ha sido algo puntual y solamente encontrado en una de las gráficas exportadas por Prometheus.

Figura 23 Messages Uncomfirmed

La Figura 24 Incoming Bytes Total muestra un acumulado de todos los bytes que se han enviado por las dos colas de RabbitMQ, siendo cada línea del gráfico un vehículo. En la gráfica se puede apreciar como todos los procesos de RabbitMQ que simulan el comportamiento de los ordenadores de viaje han ido progresivamente enviando datos.

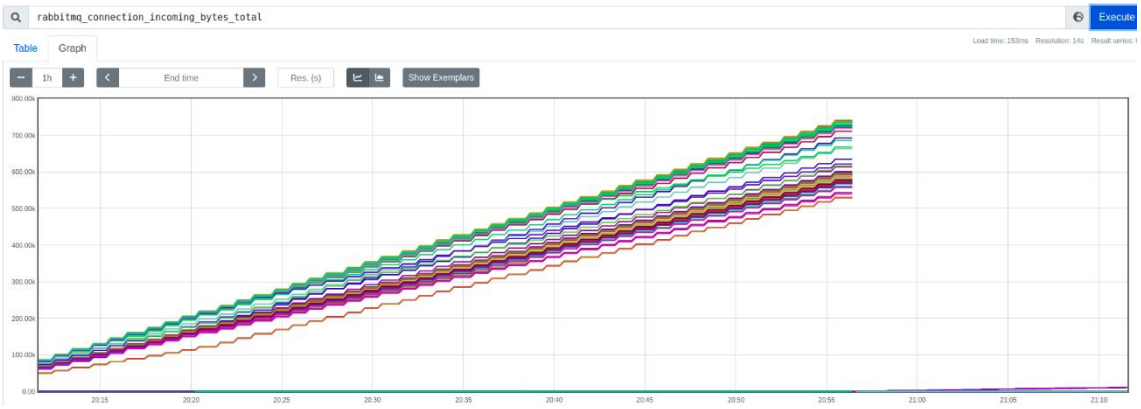

Figura 24 Incoming Bytes Total

Por parte del servidor de RabbitMQ se mantiene un flujo constante de datos y solamente se ve un punto donde no se confirma un mensaje que es en el punto de la prueba de tolerancia a fallos donde se apaga uno de los nodos. El servidor de RabbitMQ tampoco se vio saturado en el transcurso de las pruebas debido al hecho que solamente se han podido utilizar 30 vehículos para los envíos y que cada envío tenía una espera de 1.79 para poder realizar el *Join* dentro de Flink, por lo tanto, no ocurrían casos donde el servidor de RabbitMQ recibiera una cantidad de datos difícil de gestionar en un mismo momento.

Discusión

Un sistema que procesa datos en tiempo real no estricto y de forma distribuida no es comparable con uno que procesa por lotes. El sistema actual de Fagor genera los tramos de madrugada para los dos días anteriores y por ello se puede medir el tiempo total de procesamiento. Sin embargo, un sistema de procesamiento en *streaming*, como es el caso de la solución desarrollada en este trabajo, funciona a través de ventanas globales donde los parámetros de ordenadores de viaje van llegando de manera constante y solamente se procesan en el momento que se detecta un identificador de estado que finalice el tramo.

La solución desarrollada en este trabajo contiene ventajas y desventajas a comparación con el sistema que actualmente se utiliza por parte de Fagor S.L.U. (Dintén et al., 2022). Una solución que proporciona los datos sobre el perfil de conducción en cuasi tiempo real permitiría la notificación de mensajes a los conductores acerca de su comportamiento al finalizar el tramo. Adicionalmente, una solución distribuida eliminaría el punto único de fallo de un sistema centralizado y proporcionaría alta disponibilidad al poder recuperar nodos de procesamiento que fallen.

Sin embargo, los costes cumulativos de una solución desplegada en la nube se estiman mayores a los de una solución *on-premise*. Como fue visto en el paper de (Fisher & Fisher, 2018), a partir de los tres a cinco años los costes de tener servicios alojados en la nube superan a una solución *on-premise*. Por lo tanto, Fagor S.L.U. necesitaría valorar si su infraestructura fuera capaz de soportar el aumento en las flotas a procesar o si hiciera falta optar por un modelo híbrido o en la nube. Adicionalmente, se tendría que adaptar y migrar el sistema actual a uno de cuasi tiempo real donde el conjunto de tecnologías que se han utilizado para desarrollar esta solución ya que son complicados de aprender a desarrollar, gestionar, y mantener.

Conclusiones

En este Trabajo Fin de Máster se ha logrado implementar una arquitectura Kappa para el procesamiento distribuido de una flota de vehículos utilizando tecnologías punteras. La arquitectura implementada en un entorno local y en un entorno distribuido en la nube realiza las operaciones necesarias sobre los datos proporcionados por los navegadores de viaje para calcular los tramos realizados por los conductores en cada ruta. Las tecnologías han cumplido las expectativas y requisitos para poder implementar una arquitectura Kappa debido a que posibilitan el envío de mensajes, su procesamiento distribuido y paralelizado, y el almacenamiento de estos en una base de datos especializada para este tipo de datos.

Sin embargo, el análisis del rendimiento de este sistema bajo una carga intensa de datos no se ha podido realizar por dos causas: el ordenador utilizado y por los datos disponibles. El equipo personal se congelaba al intentar ejecutar un servidor RabbitMQ, Apache Flink, y Cassandra. Para poder superar estas dificultades se desplegó la arquitectura en la nube y se realizó una prueba donde se instalaban todos los elementos del proyecto en un mismo nodo para poder así simular el procesamiento en un solo nodo.

Adicionalmente a esto, el equipo personal limitó las pruebas de rendimiento en la nube debido a que a la hora de crear los procesos de Python que publican los mensajes al servidor RabbitMQ se congelaba al pasar el envío de los datos de 30 vehículos (creando 60 procesos ya que por cada vehículo se envían datos de EventosHistorico y Cantrama). A pesar de estas limitaciones se debe tomar en cuenta que el sistema está diseñado para el procesamiento de los parámetros de más de cuatro mil vehículos que proporciona Fagor en sus ficheros. Los más de cuatro mil vehículos que contienen los ficheros de Fagor corresponden a los datos de solamente uno de sus clientes. Sin embargo, Fagor estima que tienen cerca de 20.000 dispositivos embarcados y esperan crecer hasta los 80.000 en los próximos dos años

Los datos proporcionados por Fagor también supusieron una limitación considerable debido a que no todos los vehículos de un CSV aparecían en ambos y aunque aparecieran en ambos CSVs no siempre tenían una cantidad de datos suficiente para realizar una prueba. Además de esto, hay múltiples casos donde un registro de un vehículo que aparece en ambos CSVs no tiene la misma fecha, por lo tanto, probar hacer *Joins* sobre estos dos CSVs sería imposible en estos casos ya que jamás podrían coincidir.

La arquitectura, el algoritmo de *Joins* y de delimitación y cálculo de tramos de este trabajo se ha presentado en un paper titulado “Fleet management systems in Logistics 4.0 era: a real time distributed and scalable architectural proposal” (Dintén et al., 2022) a la cuarta Conferencia Internacional sobre Industria 4.0 y Fabricación Inteligente (ISM). Esta conferencia es un evento anual que explora el impacto de las tecnologías digitales de la cuarta revolución industrial en diversos sectores económicos. Este paper ha sido aceptado para presentarse en el congreso el día 05 de septiembre de 2022 y será presentado entre el 2 y el 4 de noviembre de este mismo año.

Líneas Futuras

Una de las posibles líneas futuras de esta arquitectura sería el análisis del rendimiento del sistema utilizando una mayor cantidad de datos y una mayor cantidad de vehículos para poder medir el rendimiento cuando el sistema se encuentra bajo presión. De esta manera se podría medir efectivamente los límites de RabbitMQ, Apache Flink ejecutándose en un solo nodo y también de forma distribuida, y de la base de datos Cassandra diseñada. En cuanto al nodo de

mensajería ejecutando RabbitMQ, sería interesante por parte de Fagor S.L.U. si necesitan una calidad de servicio de *exactly-once*. De ser así se debería medir en un entorno de preproducción la carga que supone esta calidad de servicio debido a que se esperaría mayor saturación del canal por asegurar que cada mensaje tendría que ser confirmado.

Adicionalmente a esto, una vez se saben los límites de los nodos se podría analizar la posible mejora de rendimiento si se implementan máquinas virtuales especializadas. En el caso de Google Compute Engine se tienen máquinas con optimización de memoria que son ideales para cargas que requieren de una gran cantidad de memoria por núcleo (*Acerca de Las Familias de Máquinas | Documentación de Compute Engine | Google Cloud*, n.d.). Las máquinas con optimización de memoria posiblemente podrían llegar a mejorar el rendimiento cuando se estén utilizando una gran cantidad de datos debido a la ventana global que hay implementada en el DataStream de creación de tramos ya que Flink tendrá que alojar una gran cantidad de eventos en el buffer de memoria en el caso de recibir una ventana grande.

Bibliografía

Acerca de las familias de máquinas | Documentación de Compute Engine | Google Cloud. (n.d.). Retrieved August 23, 2022, from https://cloud.google.com/compute/docs/machine-types

AMQP 0-9-1 Model Explained — RabbitMQ. (n.d.). Retrieved August 14, 2022, from https://www.rabbitmq.com/tutorials/amqp-concepts.html

Apache Foundation. (n.d.). *DataStream Connectors*. Retrieved August 15, 2022, from https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/connectors/datastream/overview/

Avinash Lakshman, P. M. (n.d.). *Apache Cassandra*.

Chebotko, A., Kashlev, A., & Lu, S. (2015). A Big Data Modeling Methodology for Apache Cassandra; A Big Data Modeling Methodology for Apache Cassandra. *2015 IEEE International Congress on Big Data*. https://doi.org/10.1109/BigDataCongress.2015.41

Datastax. (n.d.). *Consistent hashing | Apache Cassandra 3.0*. Retrieved August 15, 2022, from https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/architecture/archDataDistributeHashing.html

Dintén, R., García, S., & Zorrilla, M. (2022). *4th International Conference on Industry 4.0 and Smart Manufacturing Fleet management systems in Logistics 4.0 era: a real time distributed and scalable architectural proposal*. www.sciencedirect.comwww.elsevier.com/locate/procedia

Fisher, C., & Fisher, C. (2018). Cloud versus On-Premise Computing. *American Journal of Industrial and Business Management*, *8*(9), 1991–2006. https://doi.org/10.4236/AJIBM.2018.89133

GitHub - Sgb597/Rabbit. (n.d.). Retrieved September 18, 2022, from https://github.com/Sgb597/Rabbit

Hwang, H. C., Park, J. S., & Shon, J. G. (2016). Design and Implementation of a Reliable Message Transmission System Based on MQTT Protocol in IoT. *Wireless Personal Communications*, *91*(4), 1765–1777. https://doi.org/10.1007/S11277-016-3398-2/TABLES/8

Katsifodimos, A., & Schelter, S. (2016). *Apache Flink: Stream Analytics at Scale*. 193–193. https://doi.org/10.1109/IC2EW.2016.56

López Martínez, P., Dintén, R., Drake, J. M., & Zorrilla, M. (2021). A big data-centric architecture metamodel for Industry 4.0. *Future Generation Computer Systems*, *125*, 263–284. https://doi.org/10.1016/J.FUTURE.2021.06.020

Manuel Ionescu, V. (2015). *The analysis of the performance of RabbitMQ and ActiveMQ; The analysis of the performance of RabbitMQ and ActiveMQ*. https://doi.org/10.1109/RoEduNet.2015.7311982

Overview | Apache Cassandra Documentation. (n.d.). Retrieved August 15, 2022, from https://cassandra.apache.org/doc/latest/cassandra/architecture/overview.html

Philip Chen, C. L., & Zhang, C. Y. (2014). Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information Sciences*, *275*, 314–347. https://doi.org/10.1016/J.INS.2014.01.015

RabbitMQ. (2007). *Launch of RabbitMQ Open Source Enterprise Messaging*. https://www.rabbitmq.com/resources/RabbitMQ_PressRelease_080207.pdf

Soni, D., & Makwana, A. (2017). *A SURVEY ON MQTT: A PROTOCOL OF INTERNET OF THINGS(IOT) MP-Index View project Analysis and Survey on String Matching Algorithms for Ontology Matching View project A SURVEY ON MQTT: A PROTOCOL OF INTERNET OF THINGS(IOT)*. https://www.researchgate.net/publication/316018571

Anexo

Las Figura 25 Clases Parser para creación de objetos contiene el código de las clases Java que se han utilizado para convertir las líneas de los CSVs que son enviado en formato String a Apache Flink y convertir este String en un objeto Java que encapsule los datos que corresponden a esa fila de EventosHistorico, Cantrama, y el JoinedEvent que se crea en base a los dos. Las capturas de pantalla que se muestran en la Figura 26 Resultado Prueba de Verificación y Validación Flink Distribuido, Figura 27 Resultado Prueba de Verificación y Validación Tolerancia a Fallos y la Figura 28 Resultado Prueba de Verificación y Validación Escalada en Caliente son capturas de la base de datos Cassandra después de realizar las pruebas de Verificación y Validación en tres diferentes tablas donde se puede ver que los resultados a la hora de ejecutar las pruebas con los datos de los tres vehículos han producido los mismos resultados.

```
public static final class Parser implements FlatMapFunction<String,
Tuple2<JoinedEvent, String>> {
    private static final long serialVersionUID = 1L;

    @Override
    public void flatMap(String value, Collector<Tuple2<JoinedEvent,
String>> out) throws Exception {
        JoinedEvent event = new JoinedEvent(value);
        out.collect(new Tuple2<JoinedEvent, String>(event,
event.getIdVehiculo()));
    }
}

public static final class EventoParser implements FlatMapFunction<String,
Tuple2<Event, String>> {
    private static final long serialVersionUID = 1L;

    @Override
    public void flatMap(String value, Collector<Tuple2<Event, String>>
out) throws Exception {
        Event event = new Event(value);
        out.collect(new Tuple2<Event, String>(event,
event.getIdVehiculo()));
    }
}

public static final class CantramaParser implements
FlatMapFunction<String, Tuple2<Cantrama, String>> {
    private static final long serialVersionUID = 1L;

    @Override
    public void flatMap(String value, Collector<Tuple2<Cantrama, String>>
out) throws Exception {
        Cantrama cantrama = new Cantrama(value);
        out.collect(new Tuple2<Cantrama, String>(cantrama,
cantrama.getIdVehiculo()));
    }
}
```

Figura 25 Clases Parser para creación de objetos

Figura 26 Resultado Prueba de Verificación y Validación Flink Distribuido

Figura 27 Resultado Prueba de Verificación y Validación Tolerancia a Fallos

Figura 28 Resultado Prueba de Verificación y Validación Escalada en Caliente