Universidad de Murcia

Máster Universitario en Tecnologías de Análisis de Datos Masivos

Trabajo Fin de Máster

Una arquitectura Big Data para el procesamiento en tiempo real de datos en aplicaciones de gestión de flotas

Alumno:
Rubén Garrido Montesinos

Directores:
Jesús J. García Molina
Diego Sevilla Ruiz

Septiembre de 2018

Resumen

Abstract

Índice

1 Introducción

1.1 Motivación

Internet-of-Things (IoT) y las tecnologías Big Data han producido avances significativos en el dominio de los sistemas de gestión de flotas de vehículos. El paradigma IoT ha permitido mejorar el proceso de seguimiento y monitorización de vehículos y las técnicas Big Data son muy apropiadas para el análisis en tiempo real de la gran cantidad de datos obtenidos en este proceso(Loukides and Bruner, 2015). De este modo, han surgido nuevas aplicaciones de gestión de flotas que gestionan mejor los recursos de la empresa y ofrecen un procesamiento más sofisticado y escalable en sus diferentes escenarios.

Las aplicaciones de gestión de flotas son, por tanto, un dominio apropiado para la aplicación de arquitecturas Big Data. El seguimiento de los vehículos genera un gran volumen de datos que, a través de diferentes técnicas de análisis de datos, podemos extraer información de gran interés para las empresas dedicadas a este sector. Dado que las aplicaciones de gestión de flotas, deben procesar esta gran cantidad de datos y proporcionar diferentes beneficios, las tecnologías Big Data son muy apropiadas para tratarlos. Algunos de los beneficios que podemos obtener con este tipo de aplicaciones es reducir el coste de gestionar la flota, ser más responsable con el medio ambiente y poder controlar cualquier tipo de robo o mal uso de los vehículos de la empresa propietaria de la flota (Gómez, 2018) (de Aledo, 2018).

Movildata[1] es una empresa con sede en Murcia dedicada a ofrecer soluciones para la gestión de flotas. Esta empresa se ha integrado recientemente en Verizon Connect. Antes de llevarse a cabo esta integración, los tutores y alumno de este proyecto acordaron con Movildata desarrollar un proyecto piloto destinado a diseñar e implementar una arquitectura Big Data que se aplicase para ofrecer alternativas y nuevas funcionalidades a su solución de gestión de flotas. La empresa disponía de aplicaciones basadas en arquitecturas tradicionales con lo que el proyecto serviría como prueba de concepto de aplicación de tecnologías Big Data.

1.2 Objetivos

1.2.1 Objetivo principal

El objetivo principal de esta tesis de máster ha sido la elección de una arquitectura big data para aplicaciones de gestión de flotas y su evaluación en un caso de estudio definido a partir de la información proporcionada por Movildata. Se realizará una prueba de concepto de la arquitectura que ayude a la empresa a conocer las nuevas tecnologías Big Data y cómo se podría beneficiar de su aplicación.

[1] https://movildata.com/sobre-nosotros/

1.2.2 Objetivos secundarios

Entre los objetivos secundarios encontramos los requerimientos típicos de las tecnologías Big Data. Por un lado, dicha arquitectura debe ser fácil de administrar y ampliar es decir, debe ser fácilmente escalable. En nuestro caso, buscamos reconocer la dificultad y capacidad de mantener estas tecnologías y la capacidad a tolerar fallos.

A pesar de ser una prueba de concepto, tendremos que enfrentarnos a las dificultades de implementar y mantener la arquitectura. Además, veremos la capacidad de realizar nuevos desarrollos sobre la misma, comprobar la facilidad de reemplazar cualquiera de las herramientas que la componen y comprobar cómo funcionan. Por último tendremos valorar la capacidad de reemplazar a las tecnologías propuestas con las que se usaban tradicionalmente.

Dado que el objetivo principal es Investigar las diferentes arquitecturas y tecnologías aplicables para el problema abordado, se deberán justificar las razones por las que hemos seleccionado determinadas herramientas. Por tanto, será necesario evaluar las distintas herramientas que nos ofrece el mercado.

Tras esto, decir que el hardware de desarrollo es limitado por lo que se debe encontrar la forma de exportar fácilmente las diferentes configuraciones. Por otro lado, nos ayudará a valorar si cumple los requisitos mínimos de rendimiento de las diferentes herramientas.

Por último, se debe comprobar que la solución es escalable horizontalmente es decir, se escalará añadiendo más máquinas y no añadiendo más hardware al servidor. Dado esto se deberá usar una tecnología de virtualización suficientemente potente y ligera para poder añadir y quitar máquinas que proporcionen capacidad a la estructura. Por otro lado, la escalabilidad debe darse tanto en el almacenamiento como en procesamiento.

1.3 Metodología

1.3.1 Definición del trabajo

El primer paso ha consistido en definir los límites del trabajo. Se pretende desarrollar y probar los cimientos de arquitectura destinada al procesamiento de una gran cantidad de datos en streaming. A la misma vez debe tener la capacidad de almacenar los datos que recibe para realizar futuros informes. Dicha arquitectura debe ser usada para crear una plataforma que procese los datos y genere diferentes métricas que sean mostradas en tiempo real en un dashboard. Dado que la empresa Movildata está en pleno crecimiento, debido a que cada vez más son los clientes que se suscriben a sus servicios, necesitamos una estructura que sea fácilmente escalable. Por otro lado, hay información realmente importante que no se debe perder, por lo que se necesitará que los diferentes servicios puedan estar disponibles todo el día. Esto nos hace llegar a la conclusión de que debemos orientarnos hacia tecnologías horizontalmente escalables. De este modo, los diferentes servidores serán capaces de coordinarse y realizar el trabajo en equipo aunque alguno de ellos quede fuera de servicio. Por último, valoraremos únicamente herramientas gratuitas dada la naturaleza de este trabajo. Dicho esto, en este

trabajo nos centraremos en los datos que se necesitan mostrar en tiempo real. Esto es debido a que es la parte más crítica de una empresa de gestión de flotas.

La segunda etapa ha consistido en la elección de una arquitectura. Se han estudiado los dos tipos de arquitecturas más extendidos: las arquitecturas Kappa y Lambda(Careaga, 2017). Una vez realizado el estudio se seleccionará una de ellas para implementarla en nuestra solución. Dicha arquitectura debe ser lo más flexible posible para permitir cambios en las herramientas utilizadas. Es preciso tener en cuenta que el mercado de herramientas big data está cambiando continuamente.

En el tercer paso se han estudiado las herramientas candidatas a ser usadas. Tendremos que seleccionar las herramientas que más se adapten al problema de entre las que nos existentes en el mercado actual. Exigiremos que estas herramientas tengan una cierta madurez, sean robustas, haya documentación disponible y exista una gran comunidad que la avale y nos permita resolver los diferentes problemas que puedan surgir al usarla. Además, las herramientas deben ser suficientemente flexibles, tolerantes a fallos y escalables. Dada la naturaleza del trabajo, no hace falta tener experiencia sobre dichas herramientas sino que, se trata de estudiarlas y ver cual nos puede ser más útil a corto y largo plazo.

En cuarto lugar nos dedicaremos a probar las distintas herramientas y a implementar la arquitectura seleccionada con las mismas. Dicho esto, comprobaremos la interoperabilidad que existe entre las distintas herramientas y la capacidad de la arquitectura a cambiarlas.

Por último, dedicaremos el tiempo necesario para validar las herramienta y la arquitectura según los requisitos que se requieren. También tendremos que comprobar que la implementación se puede exportar a otra máquina sin problemas. Para terminar, se valorará el trabajo realizado y la posibilidad de llevarlo a un sistema en producción.

Para planificar el trabajo se ha decidido hacer uso de la herramienta GitHub, de forma que quede constancia del trabajo realizado. Gracias a GitHub también obtendremos un repositorio del código al que haremos referencia más adelante. Por otra parte, necesitaremos un repositorio de Docker, para ello usaremos Docker Hub.

Por último, la planificación del trabajo se hizo para obtener la prueba de la plataforma a finales de Junio pero se tuvo que aplazar la presentación para finales de agosto. Podemos ver como se ha ejecutado las diferentes etapas del trabajo en el diagrama de Gantt de la figura 1.

1.4 Organización del documento

Este trabajo se han organizado de la siguiente forma.En esta primera sección, se ha motivado el trabajo y se han presentado los objetivos y metodología. En la siguiente sección hablaremos del estado del arte, en el que haremos un pequeño recorrido de cómo han evolucionado las plataformas de gestión de flotas. El tercer capítulo de este trabajo se focalizará en entender qué son y que soluciona las tecnologías que engloban el Big Data. Tras esto,

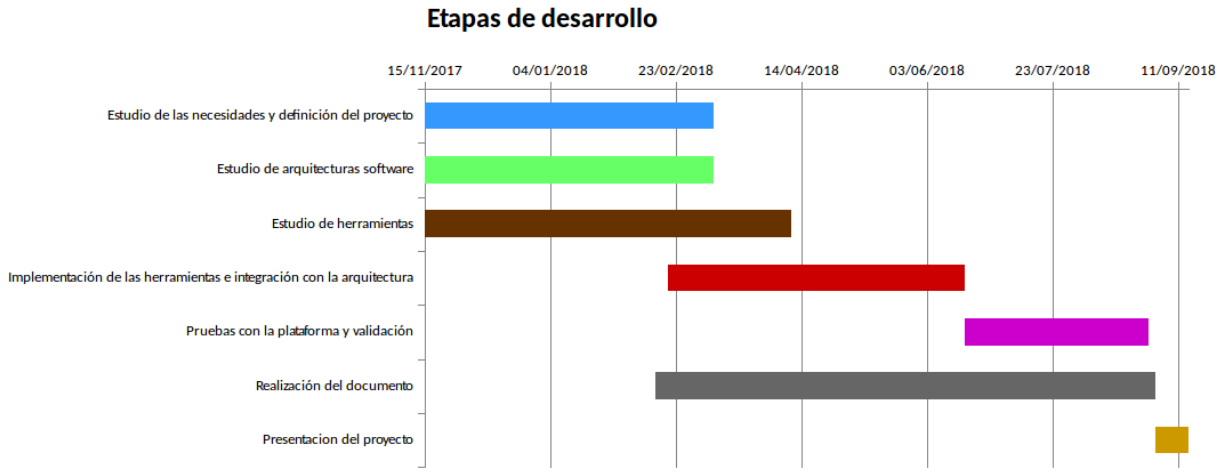

Figura 1: Planificación del trabajo

mostraremos las herramientas seleccionadas en primera instancia y mostraremos un poco la historia de cómo surgieron. Esto, es interesante para entender por qué se han seleccionado debido a que, evidentemente, su historia nos muestra que problemas tenían y cómo llegaron a solucionarlo con estas herramientas. En la cuarta parte de este trabajo, encontraremos los distintos requisitos que debe cumplir esta prueba de concepto. Cómo grueso de este trabajo, encontraremos el apartado de desarrollo en el cual, encontraremos las casuísticas de montar dichas herramientas sobre la arquitectura seleccionada. En el también encontraremos el análisis de los datos que nos ha proporcionado la empresa Movildata y los detalles de la obtención de otros datos para el desarrollo de una pequeña aplicación de monitorización del tráfico de nuestros vehículos. Para terminar con este apartado, se explicará cómo lanzar la plataforma al detalle. Por último, encontraremos los capítulos valoración y conclusión. En la valoración mostraremos el grado de aceptación que ha tenido en la empresa dicho trabajo. Por último, se presentarán las conclusiones que hemos obtenido tras realizar dicha prueba de concepto.

2 Estado del arte

En este capítulo se analizará cómo han evolucionado las arquitecturas software en las que se han basado en los sistemas de gestión de flotas. Nuestro estudio se ha centrado, principalmente, en la evolución que ha tenido lugar y en exponer las razones por la que es interesante cambiar a una arquitectura basada en Big Data.

A finales de la década de los 90, gracias al desarrollo de las telecomunicaciones móviles se hizo posible un diagnóstico más preciso a la hora del seguimiento por satélite y la monitorización de los vehículos en los sistemas de gestión de flotas. Gracias al aumento de la velocidad de comunicación y la bajada de tarifas en telecomunicaciones, los sistemas de gestión de flotas se hicieron muy populares en la década pasada (Gáspár et al., 2014).

En sus inicios, los sistemas de gestión de flotas recogían los datos directamente de los dispositivos mandando un SMS en caso de que existiera alguna infracción. Conforme fue avanzando la tecnología, estos dispositivos eran capaces de enviar más datos a un servidor central que los almacenaba. Este sistema central era también el encargado de avisar al destinatario de las infracciones y de los diferentes parámetros que se monitorizaban del vehículo. Las tareas que debía realizar el sistema de gestión de flotas eran complicadas: se debían recoger los datos de los vehículos de una forma segura y confiable evitando la pérdida de datos y asegurando que eran correctos, al mismo tiempo que se debían manejar diferentes alertas y realizar diferentes tareas relacionadas con información la geográfica (Gáspár et al., 2014).

La adquisición de los datos del vehículo era una tarea compleja. En 1983, Robert Bosch diseñó la tecnología CAN bus (*Controller Area Network*), que se trataba de un sistema central con el cual se podría manejar las diferentes partes electrónicas del vehículo y era, dicho sistema, del que se debían leer los datos. El problema de este sistema era que cada fabricante lo diseñaba según sus necesidades por lo que no existía un estándar que facilitase su lectura. En 2002, varios fabricantes decidieron crear una interfaz estándar que permitiera el sistema de seguimiento GPS. Dicho sistema se bautizó como Estándar FMS (*Float Management System*). Esto supuso un gran avance, ya que era mucho más fácil leer la posición de los vehículos. En 2010, se diseñó el Estándar FMS 2.0 que ya recogía algunos datos importantes del CAN bus y ya, en 2012, se desarrolló el Estándar FMS 3.0, diseñada especialmente para algunos parámetros importantes para vehículos pesados, como son los autobuses o los camiones. El desarrollo de este estándar supervisado por la ACEA (*European Automobile Manufacturers' Association*), la cual se reúne para analizar las diferentes necesidades que están surgiendo (Gáspár et al., 2014). Esto ha hecho que disponer de soluciones para terceros sea una tarea más fácil, ya que no hay que realizar un desarrollo independiente por cada tipo de vehículo. Aun con estos estándares, encontramos que hay diferentes parámetros que las empresas desean monitorizar dependiendo de las necesidades de cada una por lo que, aun teniendo los principales parámetros monitorizados, se desean obtener diferentes parámetros que se deben leer del CAN bus. Por otro lado, los vehículos ligeros, leen algunos estándares del CAN bus con el OBD (*On-Board Diagnostic*), pero sigue siendo una tarea compleja de

monitorizar. Este tipo de vehículos, como son furgonetas o coches, se desean monitorizar principalmente para comprobar que el conductor cumple con sus diferentes pedidos y realiza su trabajo de forma eficiente sin perder de vista la ley, por lo que son menos parámetros los que habría que recoger. Sin embargo, aun siendo las partes más importantes, las que forman parte del estándar, y se pudieran recoger a través del CAN bus o del OBD, son muchos los empresarios que desean recoger muchos más parámetros para asegurarse del estado del vehículo. Esto hace que también sea una tarea difícil para las empresas que quieran realizar un sistema de gestión de flotas a terceros (Karimi et al., 2004).

Existen diversas investigaciones sobre los beneficios que se obtienen al tener un sistema de seguimiento de vehículos en tiempo real y lo crítico que puede resultar este aspecto para una empresa. Además de estas, existen varios algoritmos para obtener beneficios en el enrutamiento de los vehículos dinámicamente para lo que es esencial obtener dicha información lo antes posible (Schorpp, 2010). Por otro lado, las multas por exceso de velocidad (DGT, 2010) o de tiempo de conducción (Fomento, 2018) son realmente elevadas y peligrosas, por lo que el aviso de horas de conducción y descanso a los conductores también sería de gran utilidad en estos casos.

Dada la cantidad de campos que hay que recoger, usualmente se ha optado por recoger los datos en XML o en JSON (Gáspár et al., 2014). Sin embargo, al existir únicamente conocimiento sobre bases de datos relacionales, dichos ficheros usualmente se encuentran en la base de datos para, posteriormente, realizar diferentes procesamientos y obtener los datos necesarios. Esto hizo que el proceso de monitorización fuera realmente complicado de gestionar e implicaba que la escalabilidad fuera limitada (Karimi et al., 2004). Algunos ejemplos de software libre que usa bases de datos relacionales son: OpenGTS (Ope, 2017) que usa MySQL para almacenar las tramas. También encontramos algunos documentos de investigación en el que explican cómo usar bases de datos relacionales con este propósito (Saghaei, 2016). Por otro lado, podemos encontrar algunas empresas privadas como Sateltrack que usaron XML para almacenar los datos (Sateltrack, 2009), además de tener servidores separados para los procesamientos más complejos, o Fleematics que usaban SQL Server y XML para almacenar los datos (Verizon, 2015).

Actualmente, con la evolución de la tecnología y gracias a los paradigmas Big Data e IoT, encontramos otros tipos de arquitecturas software como es la arquitectura Lambda (Marz and Warren, 2015)(Marz, 2017). Dicha arquitectura software nos ofrecerá diferentes lineas de procesamiento: una para procesar los datos en tiempo real y otra para realizar los procesamientos de históricos. Dicho esto, encontramos que los proveedores PaaS líderes, como son Azure o AWS (*Amazon Web Services*) nos ofertan diferentes arquitecturas basadas en Big Data como solución para recolectar los datos de vehículos. AWS ofrece una arquitectura que nos permite conectar diferentes vehículos a una plataforma que almacena los datos en bruto recibidos y muestra los resultados de su análisis en tiempo real. También obtiene diferentes métricas en diferentes bases de datos según el propósito el proposito para el que se contrate. Visto esto, podemos comprobar como la solución que ofrece, propone una arquitectura Lambda (AWS, 2017). Por otro lado, Azure también nos ofrece un servicio muy parecido con una arquitectura Lambda para obtener los datos de los vehículos (Azure, 2017).

En nuestro estudio hemos encontrado algunas soluciones de otros líderes en el mercado como Oracle, que ofrece una solución basada en sus tecnologías NoSQL, Datawarehouse, SQL y OLAP, así como procesamiento en tiempo real con reportes a través de sus herramientas (Oracle, 2015). También existen algunas recomendaciones de empresas menos conocidas como YugaByte DB, en la que se aplica un ejemplo de arquitectura para la obtención de datos de gestión de flotas, realizando el procesamiento en tiempo real. En su página oficial muestran cómo desarrollar este tipo de arquitecturas con Apache Kafka, Apache Spark y su solución de base de datos (YugaByte, 2017). Por otra parte, en algunos artículos en revistas online, como es InfoQ, relacionados con IoT se propone usar Apache Kafka, Apache Spark, Cassandra DB, entre otras tecnologías big datas (Baghel, 2016). También hemos encontrado algunas investigaciones como la de la Universidad de Seúl, Corea, en la que se propone un modelo en tiempo real con MongoDB y con un Datawarehouse con Hadoop (Jeon et al., 2015). A su vez y más reciente, una investigación de 2017 de la misma universidad, ha propuesto Apache Kafka, Apache Storm y MongoDB para procesar y manejar más eficientemente los datos (Syafrudin et al., 2017).

Para terminar con este análisis y evolución de los sistemas de gestión de flotas, podemos decir que normalmente se orientan a la web, dado que a la hora de representar las diferentes posiciones en mapas, los servicios que ofrecen tanto Google, como Bing o Here Go, entre otros, ofrecen una gran calidad sin la necesidad de tener que almacenar dichos mapas. Además, al mostrar los sistemas a través de una web nos aseguramos que sea multiplataforma (Gáspár et al., 2014).

Por último, y para finalizar con este apartado, diremos que la decisión de Movildata para estudiar este tipo de arquitecturas y no contratar una es tener sus propios servicios para tener mayor control con la arquitectura software que proponemos en este trabajo. De esta forma, a la hora de realizar diferentes propuestas o mejoras, irá de parte de la propia empresa Movildata diferenciándose del resto (AgileFleet, 2017).

3 Fundamentos y herramientas

3.1 Qué es el Big data

3.2 Docker

3.3 Apache Hadoop

3.4 Apache Spark

3.5 Apache Kafka

3.6 Elastic

3.7 MongoDB

4 Requisitos

4.1 Requisitos funcionales

1. Debe existir una cola que reciba los mensajes en bruto y de la cual, varios procesos puedan estar leyendo a la vez de dicha cola.
2. Deben existir varias réplicas de la cola de forma que si uno de sus nodos cae, se puedan seguir obteniendo los datos de forma transparente al servicio de streaming o batching.
3. La plataforma debe ser capaz de tener varios servidores coordinados usando la herramienta de streaming y distribuyendo el trabajo entre los diferentes nodos.
4. Cuando uno de los nodos que aloja a la herramienta de streaming caiga, los trabajos deben seguir ejecutándose en los demás nodos, siendo capaces de procesar la información del servidor que ha caído.
5. El servicio de streaming debe devolver a otra cola todos los datos procesados para que diferentes procesos puedan obtener dichos datos.
6. El sistema debe ser capaz de obtener los vehículos que circulan con exceso de velocidad.
7. El sistema debe ser capaz de identificar a qué usuario pertenece cada vehículo para poder filtrarlos posteriormente.
8. El sistema debe ser capaz de asociar los vehículos con los usuarios a los que pertenecen. El sistema debe ser capaz de detectar proximidad a puntos geográficos en tiempo real.

4.2 Requisitos no funcionales

1. Debe existir la posibilidad de mostrar los datos de tráfico sobre un mapa.
2. Se deben mostrar gráficas de temperatura de los vehículos.
3. Se debe mostrar en tiempo real si hay vehículos que se encuentran cerca de un punto negro.
4. Existen un dashboard con las diferentes gráficas y mapas en el cual se puede filtrar por usuario.
5. Las herramientas usadas deben ser gratuitas.
6. Se debe tener en cuenta que se deben almacenar los datos históricos para realizar futuros desarrollos.
7. Se debe tener en cuenta que se podrán procesar los datos históricos posteriormente de una forma ágil.

5 Arquitectura propuesta

5.1 Selección del modelo de arquitectura

A la hora de seleccionar la arquitectura, seleccionaremos la que mejor se adapte al casuística de nuestro problema. Por un lado, tenemos la arquitectura Kappa, la cual se reduce a una sola capa donde se procesan los datos y se almacenan para posteriormente mostrarlos en la aplicación. Esto lo podemos ver en la figura 2 (Careaga, 2017). Por otro lado, la arquitectura Lambda lanza dos líneas, una para los datos que se han de mostrar rápidamente y otra para los datos de los cuales necesitamos un procesamiento más sofisticado. Esto podemos verlo en la figura 3 (Kreps, 2014). Vista la casuística del problema se propone, una arquitectura Lambda frente a una arquitectura Kappa.

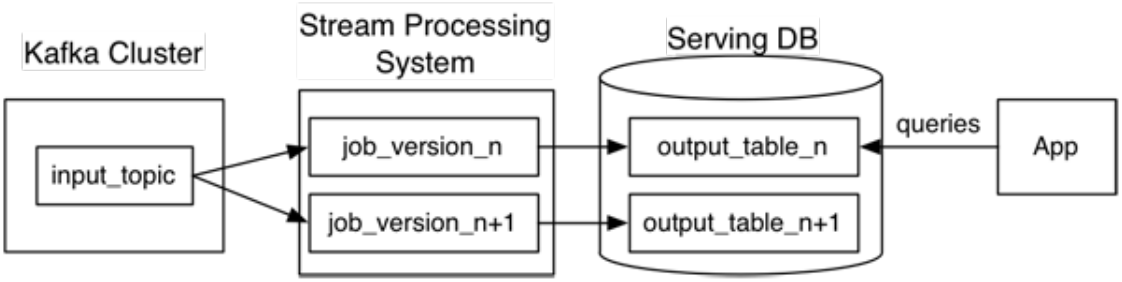

Figura 2: Arquitectura Kappa.

5.2 Selección de las herramientas elegidas

Debido a la casuística del hardware en el que estamos desarrollando y para conseguir un sistema totalmente portable se ha decidido usar Docker como herramienta para virtualizar los diferentes servicios. De esta forma usaremos menos recursos a la hora de lanzar los diferentes servicios y siendo, aún así, totalmente independientes. Por otra parte con Docker se nos va a permitir encapsular los diferentes servicios para que puedan correrse en cualquier máquina. Aun habiendo elegido este sistema también se ha valorado el uso de un Datacenter Manager como Mesos o Ambari (Siftery, 2018), pero resulta ser muy pesado para el hardware que disponemos.

Dada la arquitectura seleccionada necesitaremos seleccionar varios tipos de herramientas. Para repartir la carga de los mensajes enviados/recibidos necesitaremos un distribuidor de los diferentes datos que recibamos (broker). Tras recibir el dato necesitamos que la speed layer realice el procesamiento en tiempo real necesitaremos con una tecnología de Streaming que nos permita obtener del dato en bruto, la parte que nos interesa. Por otro lado, la batch layer nos debe permitir almacenar y consultar cualquier dato en cualquier momento ya sea en

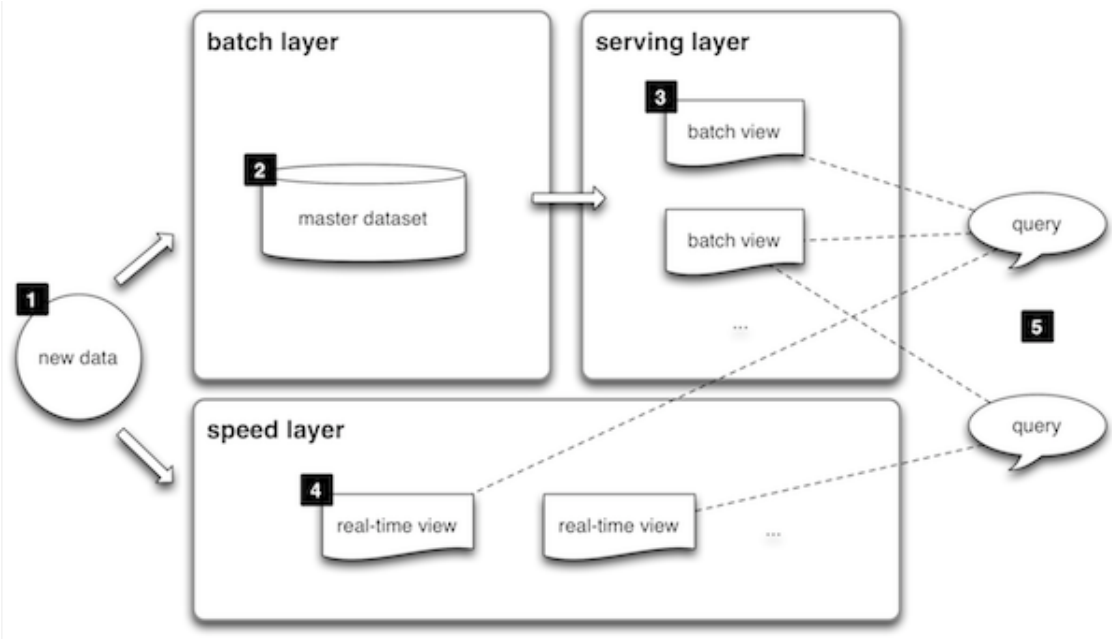

Figura 3: Arquitectura Lambda.

una o varias bases de datos. Para procesar los datos posteriormente necesitamos un sistema de batching que vaya haciendo resúmenes y obteniendo estadísticas, esto se almacenará en la serving layer. Finalmente, para poder mostrar dichos datos, necesitamos una plataforma de dashboard que nos ayude a mostrar los datos en tiempo real y los post-procesados a la cual, podamos añadir futuros informes.

Dicho esto, seleccionamos dos candidatos a brokers que son Kafka y RabbitMQ. RabbitMQ consiste en un sistema de colas tradicional donde encontramos un productor del mensaje y un consumidor del mensaje y diferentes colas a las que se pueden suscribir diferentes productores y consumidores. Por el contrario, Kafka, encontramos que hay un productor del mensaje y varios consumidores del mismo, lo que hace el mensaje persistente hasta un cierto tiempo. Dado que podemos usar también de caché a Kafka para recuperar trabajos y podemos operar fácilmente con las diferentes capas lo seleccionaremos como broker (Humphrey, 2017).

En cuanto a la speed layer encontramos diferentes tecnologías como Spark, Flink o Apex. Spark es un sistema de microbaching, Flink es un sistema de streaming puro y Apex es una nueva tecnología de streaming que está surgiendo ahora en la Apache Foundation. Dado que Apex es una tecnología que aún no está madura la descartamos y nos tendremos que decidir entre Spark y Flink. Spark trabaja en órdenes de segundos, ya que es microbaching, mientras que Flink trabaja en el orden de microsegundos. Aunque Flink es trabaja con tiempos más pequeños (Däschinger, 2017), realmente no necesitamos tanta precisión en cuanto al tiempo en streaming lo que nos hace, finalmente, decidirnos por Spark dada la comunidad tan grande que existe entorno a esta tecnología. Por otro lado, todas las librerías que lo componen tienen

una variedad más rica en funciones y sus desarrolladores más experiencia, por lo que lo hace la más versátil para nuestro problema (Mushketyk, 2017).

En cuanto a la parte de almacenamiento a gran escala de los datos nos decantamos por Hadoop, dada la potencia que tiene y la gran cantidad de usuarios que la usan. Por otro lado, podemos seleccionar varios sistemas para realizar el batching, como son Logstash, que pertenece al stack de elastic, Spark o Hadoop.

En cuanto a las bases de datos tenemos la opción de usar Postgre, pero dado que queremos más flexibilidad nos decantamos por ver cómo usar bases de datos NoSQL. Aun así, Postgre tiene una potencia más que suficiente y se puede escalar fácilmente. Por otro lado tenemos MongoDB, la base de datos NoSQL más usada a dia de hoy, lo que la hace una muy buena opción para seleccionarla (Melé and Valseca, 2018). Después de esta, podemos usar Elasticsearch para almacenar datos, aunque sea un motor de búsquedas. Otras alternativas serían InfluxDB que está optimizada para series temporales y Redis, que nos resultará muy útil en cuanto a velocidad de consulta ya que es una base de datos clave valor que almacena sus datos en memoria y persiste a disco cada cierto tiempo. Evidentemente, la empresa tiene ya sus bases de datos que habrá que integrar con esta estructura por lo que la elección tiene que ser especialmente enfocada para el tiempo real. Aunque Redis podría parecer a voz de pronto la mejor herramienta para estos casos (Shekhar, 2016), consume mucha memoria de la que actualmente, para realizar esta prueba de concepto, no disponemos. Dado esto, nos enfocaremos en usar Elasticsearch para este caso con su stack de tecnologías de forma que nos permitan mostrar un dashboard de una forma mas rapida.

En cuanto a la parte de visualización en tiempo real tenemos varias posibilidades como son Kibana con el stack de Elastic, una web realizada a mano con D3 o Grafana que se integra muy bien con InfluxDB. Dado que hemos elegido Elasticsearch y dada la facilidad de usar Kibana, es la herramienta elegida.

6 Aplicación desarrollada

Este capítulo tratará de hacer entender al lector el trabajo que se ha realizado y como funciona. Para ello, mostraremos los pasos que hemos seguido para montar cada una de las herramientas sobre la arquitectura seleccionada.

6.1 Hardware utilizado

La máquina que se ha usado para el desarrollo de este trabajo ha sido mi ordenador portátil por motivos de disponibilidad. Dicha máquina es un MSI GP60 2PE Leopard con las siguientes características:

- MSI GP60 2PE Leopard
  - Disco duro:
    - SSD mSATA de 250 GB
    - HDD SATA de 750GB
  - RAM: 16 GB
  - Procesador: Intel Core i7-4710HQ
  - Tarjeta gráfica: NVIDIA GeForce 840M
  - Sistema operativo: Lubuntu 16.04
  - Propietario: Rubén Garrido
- Disco duro externo WD Elements Basic Storage:
  - Capacidad 2 TB
  - Formateado en ext4
  - Propietario: Rubén Garrido

6.2 Datos obtenidos y preprocesado

Para realizar las pruebas se han obtenido datos de diferentes fuentes. Por una parte, Movildata nos ha ofrecido los datos anonimizados de varios vehículos obtenidos durante tres días, en concreto, durante los días 28, 29 y 30 de Mayo. Dichos datos se almacenan en un JSON de 1,83 GB en el siguiente formato:

```
{
  {
        "_id": {
                    "$oid": "IDENTIFICADOR DE OBJETO"
                },
        "metaData": [
                {
                  "keyName": "Type",
                  "units": "VehicleType",
                  "unitType": "string"
                }
        ],
        "reports": [
                {
                   "identity": {
                   "sensorId": "IDENTIFICACION DIARIA QUE SE LE DA
                                AL VEHICULO"
                },
                  "measurements": [
                     {
                         "location": {
                                      "coordinates": [
                                                      LONGITUD,
                                                      LATITUD,
                                                      ALTURA
                                                    ],
                                      "heading": "166",
                                      "temp": "TEMPERATURA",
                                      "speed": "VELOCIDAD",
                                      "speedmetric":"METRICA PARA
                                                     LA VELOCIDAD"
                                    },
                           "observationTime": "FECHA",
                  "Type": "TIPO DE VEHICULO (camion, coche o bus)"
                          }
                  ]
                },
                ....]
  }
}
```

Aquí podemos ver un ejemplo:

```
{
        "_id":    {
                "$oid":    "5afa9ec2d815bd6e3c12d0cf"
        },
        "metaData":    [
                {
                        "keyName":    "Type",
                        "units":    "VehicleType",
                        "unitType":    "string"
                }
        ],
        "reports":    [
                {
                        "identity":    {
                                "sensorId":    "651513"
                        },
                        "measurements":    [
                                {
                                        "location":    {
                                                "coordinates":    [
                                                        -1.317187,
                                                        40.648389,
                                                        994
                                                ],
                                                "heading":    "166",
                                                "temp":    "No",
                                                "speed":    "88",
                                                "speedmetric":
                                                    "KilometersPerHour"
                                        },
                                        "observationTime":
                                                "2018-05-15T08:45:51.0000000Z",
                                        "Type":    "truck"
                                }
                        ]
                },
       ....
        ]
}
```


Dado el formato de estos datos hemos tenido que preprocesarlos con Spark para obtener un fichero CSV equivalente, en el que cada trama se almacene en una sola línea. Esto se ha

realizado para facilitar la simulación del envío de las tramas de los vehículos.

Una vez obtenido esto, para simular los usuarios, he creado 180 usuarios y he dividido, con una distribución aleatoria, los vehículos con los usuarios que hemos creado. De esta forma, simularemos la cantidad de vehículos que tienen los clientes reales de una empresa de este tipo, donde cada empresa tiene una cantidad determinada, según su necesidad. Esta distribución podemos verla en la figura 4.

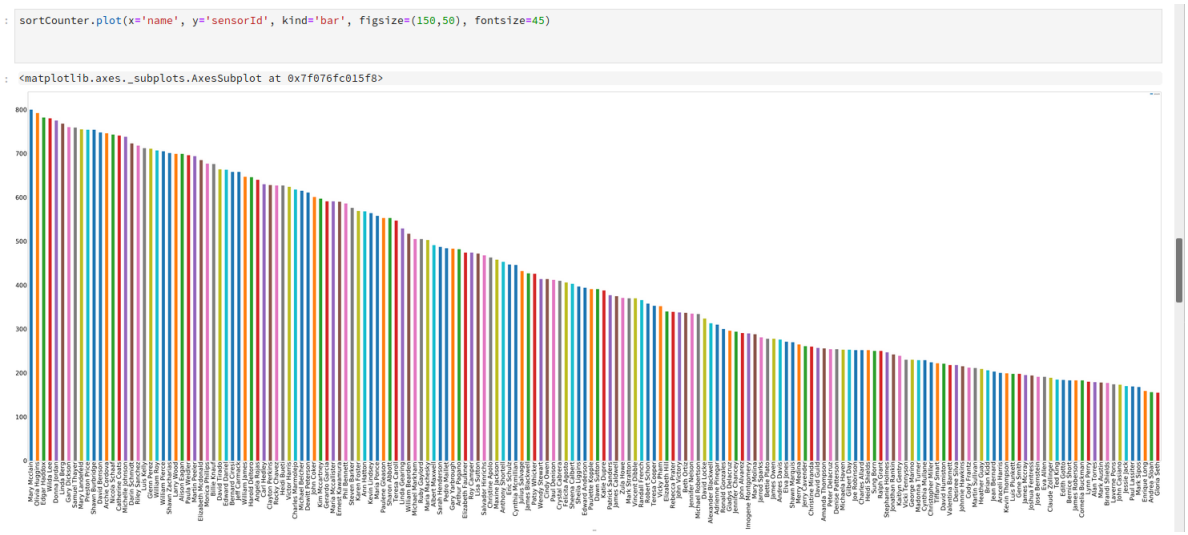

Figura 4: Cantidad de vehiculos para cada usuario.

A partir de esto, crearemos dos CSV que almacenaremos en Hadoop, en la que aparezcan los nombres de los usuarios y la asociación de los vehículos con los usuarios. Con esto obtenemos 876 vehículos en el usuario que más vehículos y 168 en el que menos.

Por último, con los datos cedidos por Movildata, hemos obtenido dos gráficas que nos mostrarán la frecuencia de los datos cada 60 segundos y cada segundo en un día concreto. Esto lo podemos ver en las siguientes figuras (5 y 6), en las que vemos como el máximo número de tramas en un minuto es de 8000 y el máximo de tramas durante 1 segundo es de 250.

Por otro lado, hemos obtenido de la DGT los puntos negros de España en 2014, que es el último documento público actualizado que hemos encontrado. Dicho documento es un excel con varias hojas, una por cada provincia o región de España, y en cada hoja se detalla la dirección del punto donde se han eventuado accidentes y se consideran puntos negros. Debido a la dificultad del formato y que no tenemos los puntos geográficos, se ha realizado un preprocesamiento para obtener un CSV más fácil de procesar por los servicios que vamos a crear. Dicho preprocesamiento aplana el excel en un solo csv y, además, obtiene los puntos geográficos a través del API de Google Maps usando la función de georreferenciación inversa. Finalmente, se han obtenido los datos de los mapas de OSM, concretamente se han obtenido los datos del mapa de España en un fichero .osm para, posteriormente, poder procesarlo e insertarlo en la base de datos seleccionada.

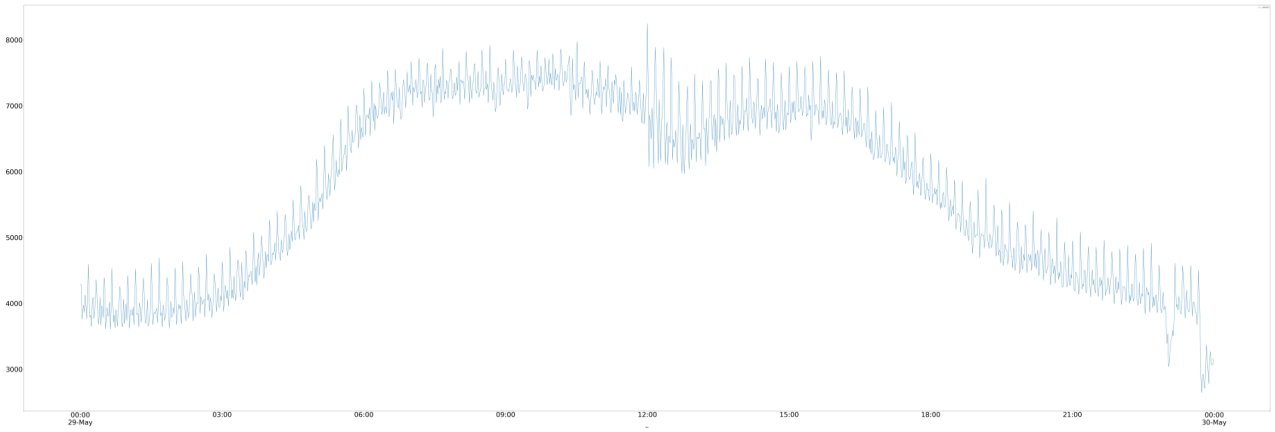

Figura 5: Frecuencia de las tramas durante un día cada 60 segundos.

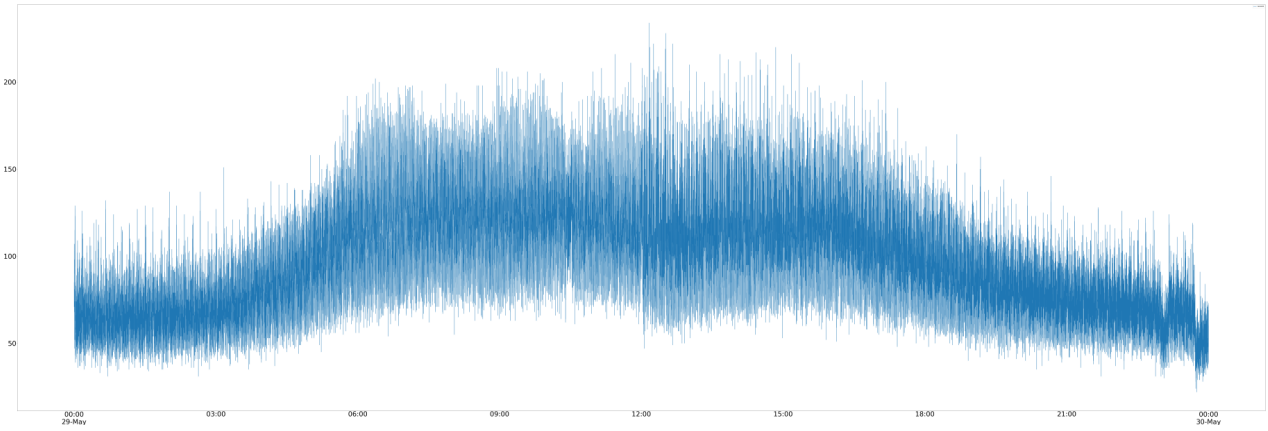

Figura 6: Frecuencia de las tramas durante un día cada segundo.

6.3 Detalles de implementación

Dedicaremos esta sección a explicar en detalle el desarrollo realizado. Dicho esto comenzaremos explicando cómo se va a ver la arquitectura a grandes rasgos hasta llegar al detalle de implementación de cada herramienta.

6.3.1 Diseño de la arquitectura con las herramientas seleccionadas

En cuanto a la arquitectura desarrollada, planificamos una estructura como la que se muestra en la figura 7. En esta arquitectura encontramos los datos ofrecidos por Movildata como emisor de los datos, los cuales llegarán a una cola de Kafka. Dichos datos serán recogidos por Hadoop para almacenarlos en bruto en HDFS. Por otro lado, Spark recogerá los datos de la cola para realizar el procesamiento en tiempo real, es decir, para asociarlos con el usuario y detectar anomalías. Tras realizar el procesamiento, Spark insertará en otra cola de Kafka los datos, los cuales serán leídos por Logstash para insertarlos en Elasticsearch. Una vez insertados en Elasticsearch seremos capaces de mostrarlos en tiempo real a través de Kibana, donde configuraremos diferentes gráficos y dashboard.

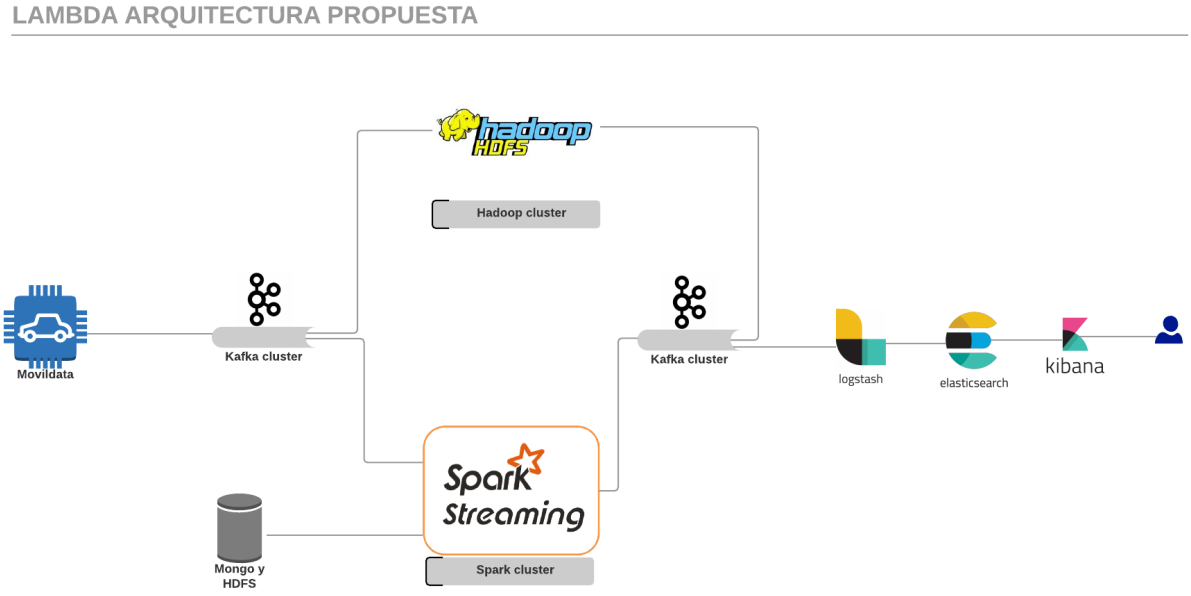

Figura 7: Lambda arquitectura propuesta.

Para realizar esto, lo primero que se ha realizado es configurar los diferentes Dockerfiles para establecer las imagenes de nuestros container. Para ello, usaremos la herencia que nos proporciona Docker y crearemos un esquema como el que aparece en la figura 8. Como padre tendremos una imagen de Ubuntu que contendrá todas las librerías comunes para todos. Por consiguiente, crearemos una imagen padre para Hadoop, Zookeeper, Kafka, Spark, las tecnologías de Elastic y MongoDB, teniendo en cuenta que Zookeeper estará por encima de

Kafka, ya que es necesario para que Kafka funcione. A partir de este diseño, comenzaremos a explicar los entresijos de cada una de estas imágenes.

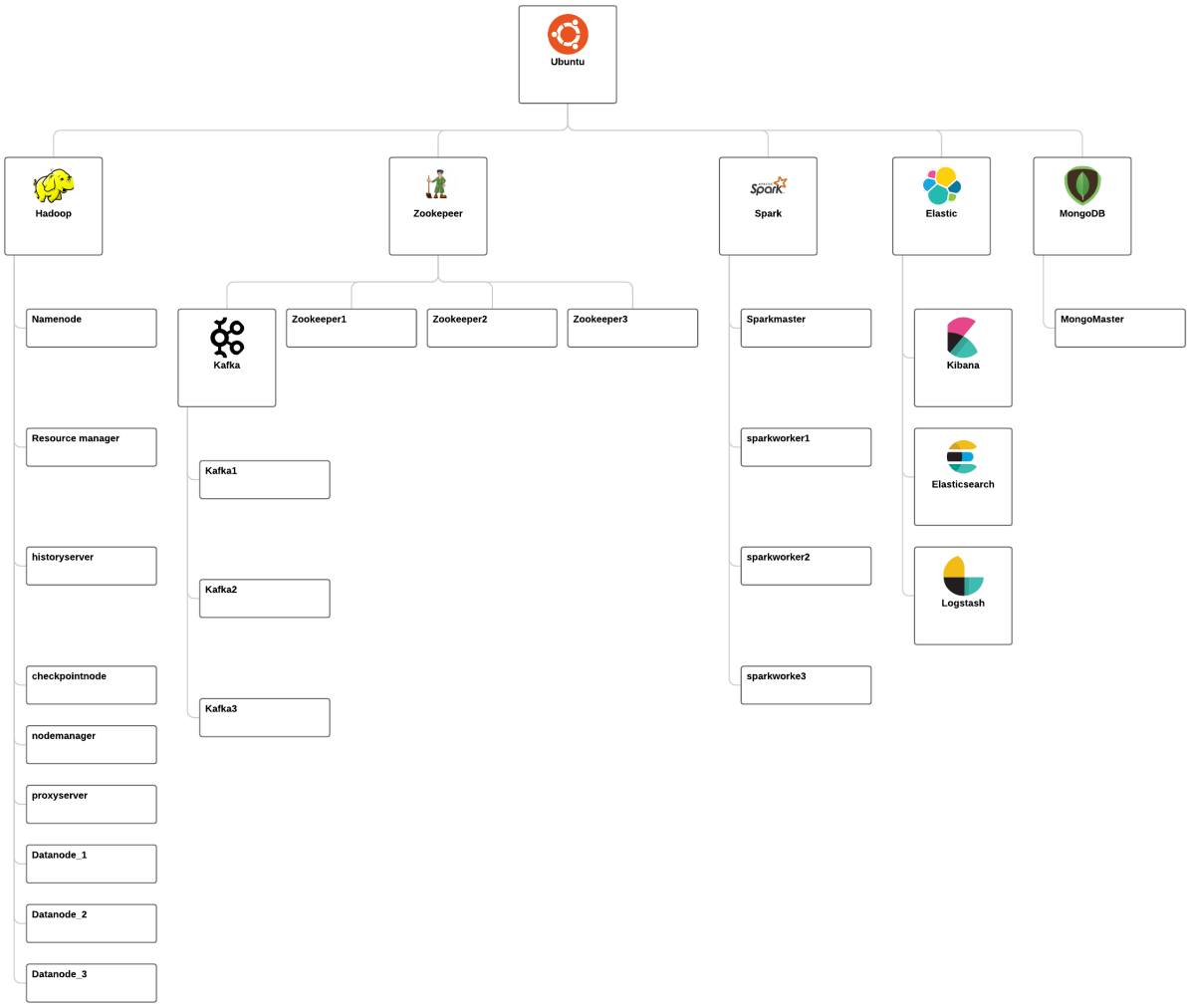

Figura 8: Esquema jerárquico docker.

6.3.2 Montaje de la imagen de Ubuntu

Para crear esta imagen, hemos usado la imagen oficial de Ubuntu que encontramos en Docker Hub. A partir de esta imagen, hemos instalado SSH para tener interconectadas todas las máquinas que creemos. Para que la conexión sea segura y no necesitemos acceder a través de contraseña, hemos generado un certificado RSA y hemos establecido en el fichero de configuración de SSH "ssh_config", que se encuentra en el directorio /etc/ssh/", el parámetro "SET StrictHostKeyChecking.ª "no"para que no pregunte si desea conectarse y se conecte automáticamente.

Para finalizar, hemos instalado en esta máquina Java 8, Scala 2.12 y Python 3. Java es necesario para que puedan ejecutarse Hadoop, Zookeeper y las herramientas de Elastic. Scala

será necesario para Spark y Kafka. Por último, nuestros desarrollos se realizarán en Python 3. Dado que es un lenguaje muy fácil de usar no necesitamos compilar, esto hará que el desarrollo de esta prueba de concepto sea más ágil.

6.3.3 Pasos a seguir para añadir aplicaciones sobre la imagen de Ubuntu

En este subapartado, explicaremos los pasos a seguir para añadir aplicaciones y sea más fácil actualizarlas en un futuro, se debe seguir la siguiente estructura:

1. Se debe añadir un directorio, cuyo nombre sea el propósito de la aplicación, al directorio "/opt". Un ejemplo de esto serían Hadoop, cuyo propósito es almacenar datos sobre su HDFS y procesarlos, por lo que entrará dentro de la categoría de base de datos y crearemos el directorio "/opt/bd".
2. Sobre el directorio creado se debe descargar la aplicación con el nombre y la versión que le corresponde. Siguiendo el ejemplo anterior, para Hadoop, añadiremos el directorio de la aplicación como "/opt/bd/hadoop-3.1.0".
3. En ese mismo directorio se debe crear un link al directorio de la aplicación con el nombre de la misma, de forma que, cuando queramos actualizar la aplicación, simplemente tendremos que cambiar el link del directorio.
4. Por último, como norma más importante, las variables de entorno de la aplicación deben apuntar al directorio creado como link y no al directorio creado en el paso 2 de la aplicación.

Por otro lado, para configurar los ficheros donde las aplicaciones almacenarán los datos y los logs de las aplicaciones deberán seguir los siguientes pasos:

Se creará un directorio en "/var/data" en el caso de los datos de aplicación y un directorio en /var/log para el caso de los log con el nombre de la aplicación. En el caso de Hadoop, por ejemplo, sería /var/data/hadoop en el caso de los datos y "/var/log/hadoop" en el caso de los logs.

1. Dadas las características de Docker, para añadir la persistencia, dichos directorios se deben establecer como volúmenes o como enlaces a directorios propios del sistema. En nuestro caso, estableceremos enlaces a los directorios del sistema y los enlazaremos a un directorio que seguirá la siguiente nomenclatura "nombre de la aplicación" + "_resources". Siguiendo con el ejemplo de Hadoop, crearemos un directorio cuyo nombre será "hadoop_resources".
2. Dado que podemos tener varios servicios por aplicación, tendremos que crear, dentro de este directorio, otro directorio con el nombre del servicio. Siguiendo el ejemplo de Hadoop y haciendo referencia al servicio datanode, crearemos el directorio "hadoop_resources/datanode/"

3. Para terminar, dentro de este directorio debemos crear los directorios data y log que serán los que realmente enlazarán con el directorio que corresponde a los del paso 1. Para seguir con el ejemplo, tendremos que crear los directorios "./hadoop_resources/datanode/data" para los datos, y "./hadoop_resources/datanode/log" para los logs. Esto lo podremos enlazar a través del comando correspondiente de Docker o en el fichero "docker-compose" que define cómo se lanzarán los servicios.

Por último, decir que en cada una de las aplicaciones se debe definir un fichero Host que contendrá las rutas DNS que queramos añadir al fichero "/etc/hosts". Al inicio, cada servicio se deben añadir estas rutas a dicho fichero. Los nombres que se establezcan únicamente pueden tener caracteres alfanuméricos debido a algunas incompatibilidades con algunas aplicaciones. Gracias a esto, los nombres de los servicios serán más amigables, por lo que será más sencillo acceder a los mismos.

6.3.4 Montaje de las imágenes de Hadoop

Para la realización de este apartado, hemos usado Hadoop 3.1, aunque también hemos probado la versión 2.7, que es la que más se usa actualmente. Siendo así, ambas mantienen la misma estructura, a excepción de algunos parámetros de configuración que varían de una versión a otra o que quedan obsoletas. Las dos versiones están disponibles y funcionales tanto en GitHub como en Docker Hub. Dado que es una prueba de concepto, usaremos la versión superior por defecto, pudiendo comprobar de esta forma que el funcionamiento de la última versión es correcto y, cuando se desee integrar, podamos usar las últimas versiones de Hadoop.

Las imágenes de Hadoop constan de varias partes. Por un lado, encontraremos la configuración básica que dará lugar al cluster y por otro lado, las ejecuciones de los servicios en máquinas independientes como recomienda la filosofía de Docker. La imagen base consta de la instalación básica de Hadoop para el usuario "hdmaster", que se almacenará en "/opt/bd/hadoop" (siguiendo los pasos explicados en la sección anterior), y el fichero de redirección DNS de todos los componentes del cluster. Además de esto, añadiremos las variables de entorno que Hadoop requiere para su ejecución: HADOOP_HOME, HADOOP_COMMON_HOME, HADOOP_HDFS_HOME, HADOOP_MAPRED_HOME y HADOOP_YARN_HOME que apuntan al directorio de de instalación de Hadoop que hemos comentado y HADOOP_CONF_DIR y YARN_CONF_DIR que apuntan al directorio donde se encuentran los ficheros de configuración de Hadoop ("/opt/bd/hadoop/etc/hadoop"). Por último, respecto a la configuración que viene por defecto en Hadoop, hemos modificado los siguientes ficheros de configuración:

- core-site.xml
  - Definimos la referencia al namenode como "hdfs://namenode:9000".
  - Usuario estático por defecto como "hdfs" .
  - El directorio temporal en "/var/tmp" ya que no necesitamos que sea persistente.
- hadoop-env.sh

- Definimos la ruta a Java a JAVA_HOME.

- hdfs-site.xml

  - Establecemos el factor de replicación de bloques a 2. Por defecto está a 3, que son demasiados para este proyecto.
  - El tamaño de bloque se establecerá a 64 megas. Por defecto está a 256 megas, que es demasiado grande para nuestro propósito.
  - Establecemos la interfaz web para que esté disponible.
  - Estableceremos el directorio donde el namenode guardará los metadatos a "/var/data/hadoop/hdfs/nn".
  - Establecemos el directorio donde el nodo checkpoint guarda los checkpoint y los los edits temporales a "/var/data/hadoop/hdfs/cpn".
  - Establecemos el directorio donde los datanodes guardan los datos a "/var/data/hadoop/hdfs/dn".
  - Establecemos la interfaz de acceso al datanode a "namenode:50070".
  - Establecemos la interfaz del nodo checkpoint a "checkpointnode:50090".
  - Establecemos que el usuario y grupo hdfs, que será el que aparezca en la interfaz web, tenga permisos para subir y borrar ficheros del HDFS.

- mapred-site.xml

  - Definimos que usaremos el framework yarn, que será el que realiza el MapReduce.
  - Definimos el JobHistory a historyserver:10020 y la interfaz web a historyserver:19888.
  - Establecemos el uso máximo de memoria a 1GB tanto para Map como para Reduce.*

- yarn-site.xml

  - Establecemos el nombre del resourcemanager y la dirección a "resourcemanager:8032".
  - Establecemos los logs de aplicaciones sobre el historyserver.
  - Establecemos el sistema de map como ShuffleHandler.
  - Establecemos el directorio de logs a "var/log/hadoop/yarn"
  - Establecemos el uso de cores a 4 por nodo.*
  - Establecemos el máximo de uso de memoria por contenedor a 4GB.*
  - Establecemos el ratio de memoria física y virtual a 4.
  - Establecemos el mínimo de memoria reservada a 1GB.*
  - Establecemos el proxy a "proxyserver:50070"

- workers

- Dicho fichero tiene los nombres de los nodos que se desean establecer como "workers" o trabajadores. Aquí, estableceremos como workers a los nodos nodemanager, el namenode, el datanode1, el datanode2 y el datanode3. Posteriormente, dichos nodos serán establecidos como workers a través del nodemanager.

Una vez hecho esto, definiremos los diferentes servicios de Hadoop especificados en la sección 2.5. Para ello, crearemos diferentes imágenes que lanzarán un shell llamado "run.sh" que simplemente se asegurará de que los puertos que necesita cada uno esté abierto y lance su servicios, a excepción del namenode. El servicio de namenode se encargará de montar el sistema de ficheros HDFS y, para que en posteriores ejecuciones no se vuelva a lanzar, en su directorio persistente guardará que ya está creado.

Para terminar con la configuración, crearemos un fichero docker-compose.yml con el que podemos probar el cluster compuesto por tres datanodes.

Por último, decir que la versión 2.7 de Hadoop, al realizar los diferentes test que vienen con la aplicación, consume más RAM que la versión 3.1. El haber realizado las pruebas con Hadoop 2.7 también es porque es la versión oficial compatible tanto para Spark como para Flink (solo usaremos Flink para comprobar que Spark es una herramienta que podemos reemplazar fácilmente) y, tras haber comprobado las diferencias, hemos llegado a la conclusión de que la única parte de Hadoop 3.1 que no es compatible con estos es la de asignar la memoria y los cores que usa de forma dinámica. Dado esto, hemos establecido en los ficheros de configuración la asignación de memoria y los cores de forma fija (aparecen con un asterisco).

6.3.5 Montaje de las imágenes de Zookeeper y Kafka

Apache Zookeeper es un orquestador que nos ayudará a volver a seleccionar líderes en los cluster de Apache Kafka y, posteriormente, de Apache Spark, por lo que lo añadiremos en la ruta "/opt/orchestrator". Para configurar esta herramienta, crearemos un cluster de Zookeeper que se dedique a esto. Para ello, en el fichero de configuración "zoo.cfg" estableceremos las direcciones de los nodos del cluster y le especificaremos que trabajará como tal.

En cuanto a Apache Kafka, dado que es un gestor de colas, lo añadiremos en la ruta "/opt/queuesmanager". En cuanto a la configuración, se encuentra en el fichero "server.properties", en el cual añadiremos las direcciones del cluster de Zookeeper. Por otro lado, dado que hay que asignarle un identificador a cada uno de los nodos, asignaremos una forma de lanzarlo dinámicamente. Para ello, estableceremos con "@id" como marca para posteriormente establecer el identificador. Aprovechando esta tarea, estableceremos el nombre del host de la misma forma como "@hostname". Por último, asignaremos el número de particiones por defecto a 1, al igual que el factor de replicación.

Para lanzar los diferentes contenedores de Kafka, lanzaremos un shell que modifique el fichero de configuración estableciendo los valores que haya que asignar y, posteriormente, lance el proceso de Kafka.

6.3.6 Montaje de las imágenes de Spark

Para montar la imagen de Apache Spark, añadiremos la aplicación en "/opt/bd/streaming", debido a que se conectará con la base de datos para realizar el streaming. Por otro lado, estableceremos en el fichero "spark-defaults.conf" la siguiente configuración:

- Asignamos, por defecto, user "yarn" es decir, el cluster de Hadoop.
- Establecemos que reserve como máximo de memoria 512MB, tanto para el yarn como para el cluster de Spark.
- Establecemos los logs.
- Asignamos los ficheros de logs si usa el yarn en el directorio "hdfs://namenode:9000/user/hdmaster/spark-logs".
- Establecemos el history server de Spark.
- Establecemos el puerto para la web de Spark a "18080".
- Seleccionamos dónde se albergan las librerías de Spark en el hdfs a "hdfs://namenode:9000/user/hdmaster/spark-archives.zip"
- Le confirmamos que la replicación de bloques del yarn está a 2.
- Le especificamos dónde está el cluster de Hadoop y con los puertos por los que se debe comunicar con: "hdfs://namenode:9000, hdfs://datanode1:9866, hdfs://datanode2:9866, hdfs://datanode3:9866"
- Establecemos el acceso a datos de HDFS a "hdfs://namenode:9000"
- Especificamos que la forma de recuperación, en caso de caÍdas, para seleccionar un nuevo líder, lo haremos con Zookeeper. Esta integración se hará automática añadiendo las direcciones del cluster de Zookeeper "zoo1:2181, zoo2:2181, zoo3:2181"

Por otro lado, añadiremos como esclavos en el fichero "slaves", que serán los procesos de Spark que ejecutarán trabajos, a los nodos sparkmaster, sparkworker1, sparkworker2 y sparkworker3, para el caso que queramos ejecutar el cluster de Spark.

Para terminar con la configuración de la imagen base de Spark, decir que para poder usar pyspark con Python 3, hemos establecido la variable de entorno "PYSPARK_PYTHON" a "python3", ya que, de no ser así, usará por defecto Python2. Por otra parte, también hemos establecido que el código de Python será introducido con codificación UTF-8, de manera que minimizaremos los problemas con caracteres españoles. Para ello, estableceremos la variable de entorno "PYTHONIOENCODING" a "UTF-8".

Con esta configuración, seleccionará por defecto los workers de Hadoop para ejecutar los trabajos. Hadoop debe contener las librerías de Spark para poder hacer uso de ellas, por lo que será necesario que estén introducidas en la ruta del HDFS especificadas. Para esto, cuando definamos el nodo spark-master, llevará a cabo su ejecución de la siguiente forma:

1. Cuando esté lanzado el contenedor por primera vez, esperará 14 segundos, dado que no hay forma de saber si el HDFS está montado.
2. Cuando hayan pasado los 14 segundos correspondientes, tendrá que comprobar de un fichero que esté guardado de forma persistente, si se han añadido ya las librerías de Spark al HDFS. Esto lo sabremos si se ha ejecutado anteriormente el script que las añade.
3. Una vez hecho esto, los slaves y el history-server de Spark estarán esperando a que el spark-master lance sus demonios.

Para cumplir con los requisitos, hemos hecho también pruebas con Apache Flink, comprobando que podemos cambiar la herramienta si en algún momento se requiere valorando esta alternativa cuando esté más madura en el mercado. Dado que usamos Kafka, también hemos comprobado que podemos usar ambas herramientas a la vez sin ningún problema, ya que pueden existir varios suscriptores en las colas.

6.3.7 Montaje de las imágenes Elastic

Para el montaje del Stack de Elastic introduciremos Elasticsearch, Logstash y Kibana en /opt/bd, en la imagen base. En la configuración de Elasticsearch, en el fichero "elastic-serhc.yml", estableceremos el nombre del cluster a "elastic-cluster", como nombre descriptivo lo estableceremos a "elastic-1-master", la ruta del host a "elasticserach" (está en la ruta del DNS), el puerto a 9200 y la ruta a datos y al log como se establece en el apartado 6.3.3. Por la parte de Logstash, estableceremos en el fichero "logstash.yml", el nombre del nodo a logstash-1, que puede usar dos núcleos de la CPU y las rutas de datos y log como se establece en el apartado 6.3.3 Por último, para configurar Kibana ejecutaremos en el fichero "kibana.yml" la ruta al host como Kibana (está en la ruta del DNS), en el puerto 5601, la ruta a Elasticsearch a "http://elasticsearch:9200ƶ el fichero de log como se establece en el apartado 6.3.3.

Para hacer uso de las aplicaciones de una forma más fácil, estableceremos las variables de entorno ES_HOME, que contendrá la ruta a Elasticsearch, ES_PATH_CONF, que contiene la ruta al fichero de configuración a Elasticsearch, LOGSTASH_HOME, que contendrá la ruta a Logstash, y KIBANA_HOME, que contendrá la ruta de Kibana.

Para lanzar los servicios, crearemos una imagen para Elasticsearch, que lanzará Elasticsearch, la de Kibana lanzará Kibana y la de Logstash no lanzará nada para poder realizar las diferentes pruebas.

6.3.8 Montaje de las imágenes de MongoDB en integración de OSM

Para montar Mongo DB, crearemos una imagen en la que añadiremos el repositorio oficial que contiene la versión 3.6 de MongoDB y lo instalamos con "apt-get", por lo que no seguirá la configuración del apartado 6.3.3. En el fichero mongodb.conf asignaremos la ruta a los

ficheros de datos y logs como se especifica en el apartado 6.3.3, la ruta como mongomaster y el puerto 27017. Por último, crearemos la imagen que lanza MongoDB.

Para integrar los datos de OSM, usaremos esta misma imagen añadiendo el mapa de España. Para subir los datos nos basaremos en un proyecto de GitHub que recomiendan en la página oficial de OSM. Dicho proyecto contiene unos scripts de Python que hemos actualizado a Python 3 para que sea compatible con nuestros proyectos. Una vez hecho esto, usando estos los scripts, hemos usado el fichero especificado en el apartado 6.2 del mapa de España y lo hemos subido a nuestra base de datos de MongoDB. Dado que los datos se almacenan en "/var/data/mongodb", podremos transportarlos entre los container que queramos.

Dado esto, crearemos una API web para poder consultar la información de un punto, de forma que no tengamos que abrir las conexiones a la base de datos cada vez que se realiza una consulta. Se ha de tener en cuenta esto, ya que, si trabajamos con programas que se ejecutan en un cluster, cada una de las máquinas del cluster ejecutará diferentes partes del código y no podemos asegurar que una sola máquina realice la ejecución de las consultas a la base de datos. Dado esto, crearemos dicha API, con Python y la librería werkzeug, con la que podremos consultar un punto geográfico. Esta API nos devolverá la información de la carretera de la base de datos que más se aproxime al punto que pidamos, en un radio de 1,11 km. El código se puede ver en el anexo 9.1, y podremos realizar consultas a través de la url "http://mongomaster:5000/getway/¡latitud¿/¡longitud¿" que nos devolverá un JSON con los datos que se encuentran en la base de datos.

6.3.9 Integración de los datos de pruebas

Para que los datos se introduzcan en Kafka simularemos con un script de Python los mensajes que se recibirán de los vehículos. Este script introducirá un JSON por cada trama en el topic "streamKafka" de Kafka.

Tras esto, haremos el preprocesamiento de Spark Streaming. Para ello, debemos introducir, en el HDFS, los CSV creados de los usuarios y la asociación de los vehículos para, posteriormente, cargarlos. Por otro lado, tendremos que integrar los datos de los puntos negros, por lo que subiremos también el CSV al HDFS.

Una vez subidos los datos, probaremos Spark Streaming y SparkSQL Streaming para ver cómo se comportan. Esto lo encontraremos en el ANEXO 9.3. y 9.4. Dicho esto, comprobamos que los conectores a Kafka son diferentes al igual que la configuración de tiempo del microbatching, además de comportarse de forma distinta. En dichas pruebas, que consisten en recoger las tramas y asociar los vehículos con sus usuarios, comprobamos que Spark Streaming se comporta mejor que SparkSQL Streaming. Dado esto, seleccionamos Spark Streaming para realizar las pruebas. Por otro lado, para la asociación de los vehículos y los usuarios, probamos si funciona mejor con SparkSQL cambiando de contexto o con un rdd de Spark y comprobamos que SparkSQL funciona mejor, ya que, además de que es más fácil de usar, crea un batch con la concatenación de las diferentes aplicaciones que realizamos sobre los conjuntos de datos.

El script de Spark Streaming realizará lo siguiente:

- Cada 10 segundos se añade un trabajo a la cola de trabajos de Spark que se ejecutarán si el cluster no está ejecutando ningún otro proceso de esa misma cola.
- Cada uno de esos trabajos leerá de Kafka los 10 segundos que le corresponden indiferentemente de si corresponden en el tiempo o no. Esto es porque puede ser que algún trabajo se retrase, en ese caso, los trabajos retrasados se ejecutarán tras haber terminado el anterior.
- El proceso convertirá los JSON leídos de Kafka en una Dataframe de SparkSQL.
- Posteriormente, filtrará los datos erróneos, es decir, los que no son del día actual. En el caso de la simulación, los que no sean de los días que hemos recogido los datos.
- Se realizará los JOINs correspondientes para obtener el usuario.
- Analizamos la temperatura
- A continuación, buscaremos trama por trama, la que se acerque a un punto negro y la distancia hasta el mismo. Para ello, haremos lo siguiente:
  - Eliminamos todas las tramas que contengan velocidad 0 km/h.
  - Realizaremos la combinación completa de una trama a todos los puntos.
  - Filtraremos todos aquellos que se encuentren por encima de puntos que tengan una distancia por encima de 1.11 Km a través de la latitud y la longitud, es decir, truncando por el tercer decimal.
  - Calcularemos la distancia a los puntos negros con el algoritmo de Haversine[2] y nos quedaremos con el que menos distancia tenga.
  - Añadiremos los puntos negros que corresponden a cada trama y la distancia que hay hasta los mismos al Dataframe obtenido en el paso anterior.
- Por último, introduciremos cada fila del Dataframe (cada trama procesada), convertidas en un JSON, en otro topic de Kafka, diferente al que usábamos para recibir las tramas.

Dado esto, también vamos a intentar obtener la dirección exacta de por dónde van los vehículos para ver si van en exceso de velocidad. Para ello, haremos uso de la base de datos que hemos creado de OSM en MongoDB. Hemos realizado pruebas abriendo y cerrando la conexión sobre la base de datos pero no era eficiente, por lo que se ha decidido realizarlo a través de la API web que hemos creado. Para poder tratar esto, se ha de realizar una petición trama por trama. A continuación, explicaremos el proceso que sigue para saber si va en exceso de velocidad y la dirección en la que se encuentra. Los pasos son los siguientes:

- Haciendo uso de la función "udf" de Spark, que nos permitirá aplicar una función a cada una de las filas del Dataframe y devolver una estructura de datos, crearemos una función que:

[2] https://en.wikipedia.org/wiki/Haversine_formula

    - Obtenga la ubicación y la velocidad a la que va el vehículo.
    - Consulte en nuestra API la dirección, el tipo de vía y si tiene la velocidad máxima a la que se puede transitar en la misma.
    - Si tiene marcado la velocidad máxima, devolveremos ese límite de velocidad aplicando las reglas que se aplican según el tipo de vehículo. En el caso de no ser así, dependiendo del tipo de vía y de vehículo, distinguiremos la velocidad máxima de la vía para dicho vehículo para devolverla.
    - La función devolverá la dirección y la velocidad máxima a la que puede transitar el vehículo.

- Aplicamos la función definida con "udf", se la pasamos a cada fila del Dataframe usando la ubicación y el tipo de vehículo que es y obteniendo la dirección y la velocidad máxima para añadirlos al Dataframe que, posteriormente, se enviará a la siguiente cola de Kafa.

Dicho esto, también se ha comprobado que eliminando cualquier esclavo de Spark o de Hadoop cuando se está ejecutando, este es capaz de recuperarse del fallo.

Para introducir los datos producidos por Spark en Elasticsearch, usaremos Logstash. Dicha herramienta, es capaz de leer los JSON del Topic donde Spark deposita su salida para, posteriormente, introducirlos en la Elasticsearch. Para ello, lanzaremos nuestra pipe de Logstash e introducirá dichos datos en un índice de Elasticsearch previamente creado. A partir de aquí, podemos hacer uso de Kibana, obteniendo los datos del índice. Con esta herramienta, somos capaces de ver, gráficamente, el tráfico de los vehículos, filtrar por usuario, o saber que vehículos van en exceso de velocidad, si hemos activamos la parte de consultas a OSM.

6.3.10 Uso de la arquitectura

En este apartado explicaremos como montar en las diferentes imágenes de la arquitectura de forma propia a partir de descargar el código que se encuentra en el repositorio de GitHub[3].

Para hacer uso de la arquitectura, tenemos dos opciones. En la primera opción habrá que montar las imagenes de Docker nosotros mismos a través de los Dockerfiles, de forma que reemplacemos los certificados de acceso. La segunda forma es más sencilla, pero no reemplazamos los certificados, por lo que, cualquiera, podrá acceder a las máquinas con el certificado que hay disponible en la imagen de Docker Hub. Aún así, para ambas opciones necesitaremos descargarnos el proyecto del repositorio.

En cuanto a la primera opción, para montar las imagenes de Docker en nuestro host, se ha creado un script en la carpeta principal del proyecto con nombre "rebuildAllDocker.sh". Dicho Script creará todas las imágenes a partir de los Dockerfiles mencionados anteriormente, sin embargo, antes de lanzar estos scripts, debemos descargarnos las imágenes de cada aplicación y añadirlas en su directorio correspondiente, cuyo nombre estará compuesto por el nombre

[3] https://github.com/Kartonatic/tfm

de la aplicación + "_base". En el README que se encuentra en estos directorios aparece la URL con la que podemos descargar las aplicaciones.

Si vamos a usar la segunda opción, simplemente haremos uso del comando "docker-compose" y descarga las imágenes del repositorio.

Antes de usar la arquitectura debemos cargar los datos de OSM. Para ello, en el directorio "MongoDB_to_OSM", seguiremos las instrucciones del README. Cuando se empiecen a cargar los datos puede llevar varias horas, según el hardware utilizado. Una vez hecho esto, tendremos los datos de la base de datos en "mongodb_resources". Este directorio lo podremos copiar en cualquier otro lado y seguirá siendo funcional con la imagen de MongoDB que hemos creado.

Para lanzar la arquitectura vamos a seguir los siguientes pasos, que serán idénticos para ambos métodos. La arquitectura final se encontrará en el directorio "00_final" y habrá que seguir los siguientes pasos:

1. Tendremos que crear los directorios de datos, para lo que haremos uso del script "createFoldersFromGit.sh" que se encuentra en el directorio "ToGenerateFolders".
2. Reemplazaremos el directorio "mongodb_resources" por el directorio obtenido en la integración de MongoDB con OSM.
3. Estableceremos con el comando "sysctl -w vm.max_map_count=262144" mas memoria virtual a las máquinas. Esto es porque Elasticsearch la necesita.
4. Añadiremos a "spark_resources", al directorio de "sparkmaster" los csv de los usuarios y los puntos negros que se encuentran en el directorio "Data".
5. Lanzaremos con "docker-compose" el yaml del directorio "00_Final", que lanzará los diferentes contenedores que hemos creado.
6. Accederemos al container de Spark, "sparkmaster" y usuario "hdmaster", y haremos lo siguiente:
   a) Añadiremos, desde este contenedor, los ficheros de datos que hemos introducido en "spark_resources" en el HDFS.
   b) Introduciremos el script "sparkstreaming.py, que se encuentra en el directorio "Code" y lo lanzamos con "spark-submit".
7. Creamos los índices de Elasticsearch con el script "CreateIndexForElastic.py".
8. Accedemos al container de Logstash y copiamos el fichero "logstashkafkaUpdates.conf" que define una pipe de Logstash que actualiza los datos actualizando cada vehículo. Tras realizar esto lanzamos la pipe de Logstash.
9. Abriremos Kibana, a través de un navegador web, y añadimos el índice que hemos creado en Elasticsearch.
10. Cargamos el fichero de dashboards que se encuentran en el directorio "DASHBOARD_KIBANA". Este fichero es un JSON que te deja importar y exportar Kibana con la configuración de diferentes gráficos y dashboards.

Una vez hecho esto, nos debe aparecer un dashboard en Kibana como el que aparece en las figura 9.

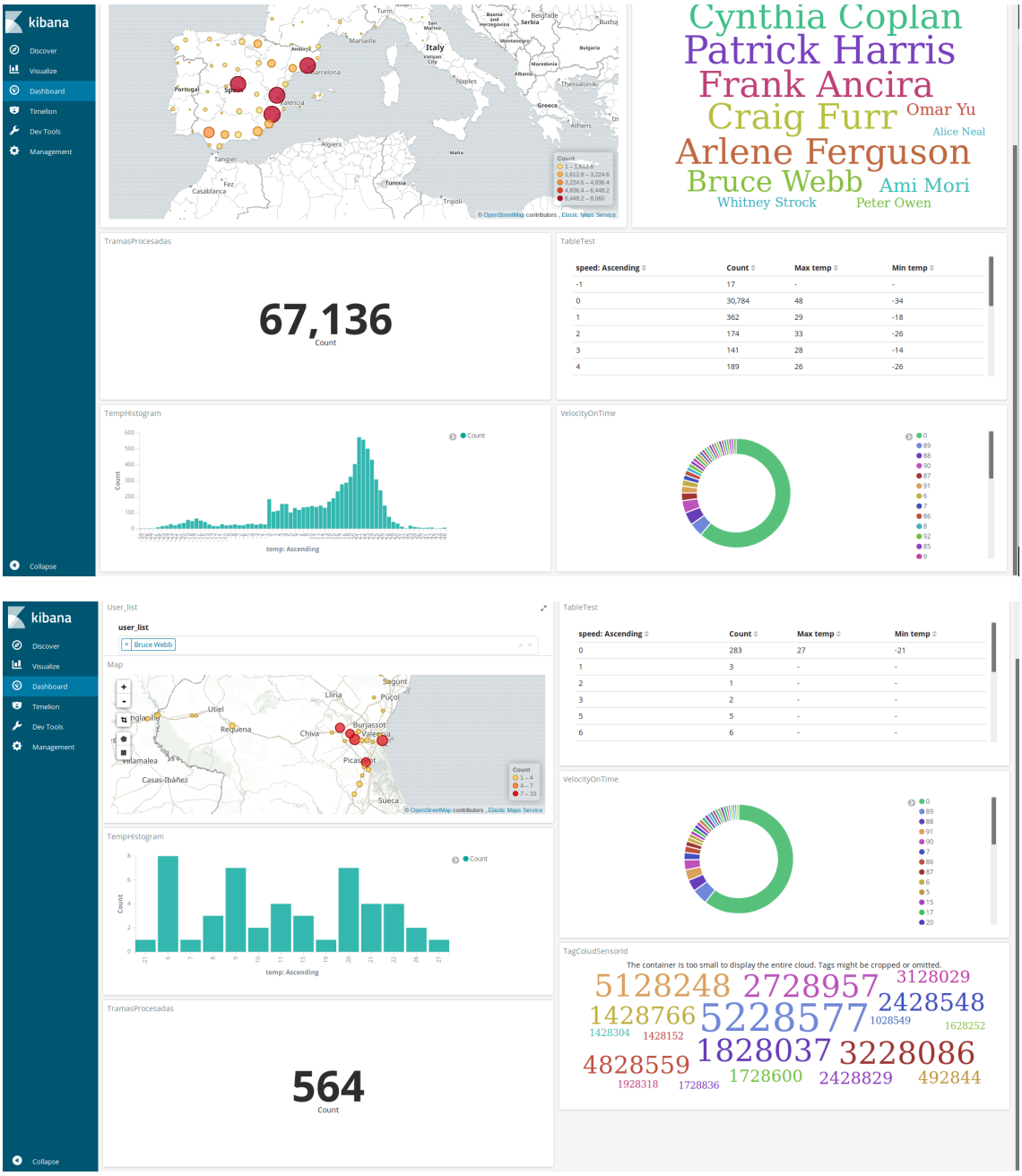

Figura 9: Dashboards de Kibana

7 Validación

En este apartado vamos a analizar como se ha comportado la arquitectura propuesta sobre el hardware que hemos usado. Evidentemente, los resultados de tiempo de ejecución no corresponden con los de un cluster real pero, por motivos de disponibilidad, no se ha podido realizar el estudio sobre uno de estos.

La arquitectura propuesta nos a aportado una flexibilidad que, con una arquitectura tradicional no tendríamos. En el caso de una arquitectura Lambda tenemos dos vías para las cuales realizar procesamientos. Por un lado tenemos la batch layer para los procedimientos más costosos y que requieren obtener datos de histórico y por otro la Speed layer que nos permitirá obtener los procesamientos de tiempo real que nos sean necesarios. También, dado que la funcionalidad no está certificada en una sola aplicación sino que, distinguimos varias partes, hace que reemplazar cualquier herramienta sea una tarea más sencilla. Dado esto y, principalmente por poder separar los procesamientos en tiempo real y de histórico, la arquitectura Lambda es más que recomendable en sistemas de IoT que requieren gran cantidad de procesamientos y, más aún, en las aplicaciones dirigidas al transporte y gestión de flotas.

En cuanto a las herramientas utilizadas, comenzaremos analizando el uso de Docker sobre esta arquitectura. Docker, nos a aportado la capacidad de mover las diferentes herramientas de un sitio a otro sin ningun problema gracias a la encapsulación, lo que hará que sea más fácil poder probarlo en un cluster en un futuro. Por otro lado, aunque la curva de aprendizaje es dura al principio, luego se hace realmente fácil de manejar, aportando las ventajas vistas en el apartado 3.2.

En cuanto a Apache Kafka, hemos encontrado un rendimiento más que notable, siendo capaces de leer con varios procesos sobre un mismo Topic cuando hemos realizado las diferentes pruebas. Además de esto, el hecho de que sea tan fácil de configurar lo hace una herramienta muy recomendable. Una de las propiedades que más llama la atención de Kafka y es lo que lo hace tan potente, es que podemos almacenar los mensajes durante un tiempo, lo que hace que podamos leer gran cantidad de tramas en el momentos determinados. Esto hace que en aplicaciones de gestión de flotas sea muy recomendable ya que podemos buscar tramas erróneas en diferentes procesos de análisis que se pueden ir ejecutando por otra parte.

Por la parte de Hadoop, encontramos que el tratamiento de datos es complicado sin herramientas como Hive. Sin embargo, también hemos encontrado la facilidad de integrarlo con diferentes herramientas. Por otro lado, también vemos que el sistema de ficheros HDFS nos da la ventaja de dejar de tener que usar RAIDs en nuestros servidores y poder acceder a los datos de una forma rápida.

Por la parte de Spark, encontramos que las operaciones para obtener los usuarios y comprobar si está cerca de un punto negro, se realizan muy eficientemente y se pueden tratar las tramas rápidamente. Con las pruebas realizadas hemos podido tratar más de 3000 tramas por debajo de los 10 segundos del microbaching. También se ha observado que la primera

vez tarda más debido a que tiene que cargar los datos de los usuarios y las primeras iteraciones van desacompasadas. También encontramos que, al mantener el índice de Kafka, si lo paramos y lo volvemos a ejecutar, procesa todas las tramas desde que se paró hasta que vuelve a ponerse en ejecución. Aún así, el proceso es capaz de recuperarse en unos minutos realizando el procesamiento streaming a su hora. Por otro lado, hemos comprobado que el rendimiento añadiendo las consultas a la base de datos de OSM, penaliza mucho el rendimiento. Realizando dichas consultas, el procesamiento no es capaz de procesar las 3000 tramas en los 10 segundos de tiempo que se usan, llegando a tardar más de 40 segundos para procesarlas. Aunque el proceso es capaz de procesar las tramas, no es capaz de acompasarse debido al acceso a datos. Aunque existe la posibilidad de integrar MongoDB con Spark, descartamos dicha opción ya que carga toda la colección para hacer uso de ella, por lo que no sería eficiente en memoria en el caso de la base de datos de OSM. Por otro lado, encontramos que es realmente sencillo de usar, ya que con un solo script de Python no hemos tenido que manejar la concurrencia entre los diferentes servidores y, además, nos ofrece gran cantidad de operaciones que podríamos encontrar en una base de datos relacional tradicional.

En cuanto a MongoDB encontramos que es una herramienta muy potente a la hora de manejar gran cantidad de datos. Encontramos una gran flexibilidad a la hora de introducir datos que no encontramos por parte de las bases de datos relacionales, que al manejar gran cantidad de filas sin indexar, se hacen mucho más lentas. Además, es muy fácil realizar APIs sobre esta ya que el mismo motor devuelve objetos JSON muy usados en el mundo web. Por tanto, calificamos esta herramienta como válida para diferentes propósitos, pero muy especialmente en el nuestro, por la flexibilidad a la hora de tratar diferentes tipos de datos.

Por último, el Stack de Elastic, nos a ayudado mucho a ver los datos en tiempo real. El coste de trabajo de ponerla estas herramientas en producción es muy bajo con el rendimiento que obtenemos, además de la facilidad de usar Kibana para analizar los datos.

Dicho esto, concluimos que se han validado los requisitos propuestos por la empresa. Por su parte, Cristóbal, CTO de Movildata nos a validado el estudio realizado valorandolo muy positivamente y, de no ser por la compra de Verizon, valoraría la posibilidad de migrar a dicha arquitectura.

8 Conclusiones y trabajo futuro

Referencias

(2017). Opengts. http://www.opengts.org/.

AgileFleet (2017). Fleet management software: Build or buy? https://cdn2.hubspot.net/hubfs/437692/E-Guides,_e-books,_etc./Build_or_Buy_E-Book_20170629-1.pdf.

AWS (2017). Presentación de la solución aws connected vehicle. https://aws.amazon.com/es/about-aws/whats-new/2017/11/introducing-the-aws-connected-vehicle-solution/.

Azure (2017). Lambda architecture for connected car fleet management. https://azure.microsoft.com/en-in/resources/videos/build-2017-lambda-architecture-for-connected-car-fleet-management/.

Baghel, A. (2016). Traffic data monitoring using iot, kafka and spark streaming. https://www.infoq.com/articles/traffic-data-monitoring-iot-kafka-and-spark-streaming.

Banker, K. (2011). *MongoDB in Action*. Manning Publications Co., Greenwich, CT, USA.

Banon, S. (2006). The birth of compass. http://thedudeabides.com/articles/the_birth_of_compass.

BBVAOPEN4U (2015). Apache spark: las ventajas de usar al nuevo 'rey' de big data. https://bbvaopen4u.com/es/actualidad/apache-spark-las-ventajas-de-usar-al-nuevo-rey-de-big-data.

BigDataDummy (2017). Apache kafka. https://bigdatadummy.com/2017/02/01/apache-kafka/.

Careaga, J. (2017). Arquitectura lambda vs arquitectura kappa. https://www.oreilly.com/ideas/questioning-the-lambda-architecture.

Carter, E. (2018). 2018 docker usage report. https://sysdig.com/blog/2018-docker-usage-report/.

Chambers, B. and Zaharia, M. (2018). *Spark - The Definitive Guide: Big Data Processing Made Simple*. O'Reilly Media, Incorporated.

Chodorow, K. (2013). *MongoDB: The Definitive Guide*. O'Reilly Media, Inc.

de Aledo, A. G. (2018). El supremo avala el gps para vigilar al trabajador. https://www.diariodecadiz.es/provincia/Supremo-avala-GPS-vigilar-trabajador_0_1276372522.html.

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplied data processing on large clusters. https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf.

DGT (2010). Cuadro para excesos de velocidad. https://sede.dgt.gob.es/Galerias/tramites-y-multas/alguna-multa/consulta-de-sanciones-por-exceso-velocidad/cuadro_velocidad.pdf.

Docker (2018a). Docker company. https://www.docker.com/company.

Docker (2018b). Dockerfile reference. https://docs.docker.com/engine/reference/builder/#format.

Docker (2018c). Overview of docker compose. https://www.bluedata.com/blog/2017/08/hadoop-spark-docker-ten-things-to-know/.

Däschinger, M. (2017). Apache flink vs apache spark. https://www.woodmark.de/blog/apache-spark-vs-apache-flink/.

Elastic (2018). Elastic stack. https://www.elastic.co/.

Esteso, M. P. Apache spark: qué es y cómo funciona. https://geekytheory.com/apache-spark-que-es-y-como-funciona.

Flair, D. (2017). Directed acyclic graph dag in apache spark. https://data-flair.training/blogs/dag-in-apache-spark/.

Fomento, M. (2018). Tiempos de conducción y descanso. https://www.fomento.gob.es/transporte-terrestre/inspeccion-y-seguridad-en-el-transporte/tiempos-de-conduccion-y-descanso/conduccion/tiempos-de-conduccion.

Frampton, M. (2016). *Big Data Made Easy, A Working Guide to the Complete Hadoop Toolset*. APRESS.

Ghavam, P. K. (2016). *BIG DATA GOVERNANCE, Modern Data Management Principles for Hadoop, NoSQL and Big Data Analytics*. CreateSpace Independent Publishing Platform.

Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf.

Golub, B. (2013). dotcloud, inc. is becoming docker, inc. https://blog.docker.com/2013/10/dotcloud-is-becoming-docker-inc/.

Gormley, C. and Tong, Z. (2015). *Elasticsearch: The Definitive Guide*. O'Reilly Media, Inc., 1st edition.

Gáspár, P., Szalay, Z., and Aradi, S. (2014). Highly automated vehicle systems. http://www.mogi.bme.hu/TAMOP/jarmurendszerek_iranyitasa_angol/.

Gómez, J. M. F. (2018). *Introducción a la gesitón de flotas de vehículos*. SafeCreative.

Hadoop, A. (2018). What is apache hadoop. http://hadoop.apache.org/#What+Is+Apache+Hadoop%3F.

Hallock, S. (2013). 10gen announces company name change to mongodb, inc. https://www.mongodb.com/press/10gen-announces-company-name-change-mongodb-inc.

Humphrey, P. (2017). Understanding when to use rabbitmq or apache kafka. https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka.

Hurwitz, J., Nugent, A., Halper, F., and Kaufman, M. (2013). *Big Data for Dummies*. John Wiley and Sons, Inc.

Hykes, S. (2013a). Docker company. https://docs.docker.com/compose/overview/.

Hykes, S. (2013b). The future of linux containers. https://www.youtube.com/watch?v=wW9CAH9nSLs.

Jayesh (2016). Retrieve timestamp based data from kafka. https://stackoverflow.com/questions/39514167/retrieve-timestamp-based-data-from-kafka.

Jeon, J., An, M., and Lee, H. (2015). Nosql database modeling for end-of-life vehicle monitoring system. *JSW*, 10(10):1160–1169.

Kafka, A. (2018). Apache kafka. https://kafka.apache.org/.

Karimi, A., Olsson, J., and Rydell, J. (2004). A Software Architecture Approach to Remote Vehicle Diagnostics. Master's thesis, IT UNIVERSITY OF GÖTEBORG.

Kreps, J. (2014). Questioning the lambda architecture. https://www.oreilly.com/ideas/questioning-the-lambda-architecture.

Kupfer, D. (2014). A brief history of elasticsearch. https://jaxenter.com/elasticsearch-founder-interview-112677.html.

Lee, J. (2018). What are the benefits using docker? https://www.quora.com/What-are-the-benefits-using-Docker.

Lohr, S. (2013). Business, innovation, technology, society the origins of 'big data': An etymological detective story. https://bits.blogs.nytimes.com/2013/02/01/the-origins-of-big-data-an-etymological-detective-story/.

Loukides, M. and Bruner, J. (2015). *What Is the Internet of Things?* O'Reilly Media.

Macías, M., Gómez, M., Tous, R., and Torres, J. (2015). *Introducción a Apache Spark*. Manuales. Oberta UOC Publishing.

Marz, N. (2017). Lambda arquitecture. http://lambda-architecture.net/.

Marz, N. and Warren, J. (2015). *Big Data, Principles and best practices of scalable real-time data systems*. MANNING.

Melé, A. and Valseca, R. (2018). Nosql y python: usando mongodb y redis en proyectos reales con python. https://www.youtube.com/watch?v=oltouxKTkFw.

Microsft (2013). Introducción a containers y docker. https://docs.microsoft.com/es-es/dotnet/standard/microservices-architecture/container-docker-introduction/.

Microsoft (2017). ¿qué es docker? https://docs.microsoft.com/es-es/dotnet/standard/microservices-architecture/container-docker-introduction/docker-defined.

Midtgård, K. (2017). Collecting massive amounts of location data in a nosql database. Master's thesis, Norwegian University of Science and Technology.

Molina, P. J. (2016). Contenedores con docker. https://www.youtube.com/watch?v=iK5XK3Ornq0&t=4809s.

MongoDB (2018). Nosql databases explained. https://www.mongodb.com/nosql-explained.

Mushketyk, I. (2017). Apache flink vs apache spark. https://brewing.codes/2017/09/25/flink-vs-spark/.

Narkhede, N., Shapira, G., and Palino, T. (2017). *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale*. O'Reilly Media, Inc., 1st edition.

Novoseltseva, E. (2018). Beneficios de utilizar docker. https://apiumhub.com/es/tech-blog-barcelona/beneficios-de-utilizar-docker/.

Oracle (2015). Improving logistics and transportation performance with big data. http://www.oracle.com/us/technologies/big-data/big-data-logistics-2398953.pdf.

Plugge, E., Hawkins, T., and Membrey, P. (2010). *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkely, CA, USA, 1st edition.

Rahul (2015). Whether key is required as part of sending message in kafka. https://stackoverflow.com/questions/29511521/whether-key-is-required-as-part-of-sending-message-in-kafka.

RedHat (2018). ¿qué es un contenedor de linux? https://www.redhat.com/es/topics/containers/whats-a-linux-container.

Saghaei, H. (2016). Design and implementation of a fleet management system using novel gps/glonass tracker and web - based software. In *International Conference on New Research Achievements in Electrical and Computer Engineering*.

Sateltrack (2009). The professional solution for web-enabled fleet management. https://www.sateltrack.com/download/sateltrack_features_en.pdf.

Schorpp, S. (2010). Dynamic fleet management for international truck transportation. Technical report.

Shekhar (2016). When to redis? when to mongodb? https://stackoverflow.com/questions/5400163/when-to-redis-when-to-mongodb.

Siftery (2018). Apache mesos vs apache ambari. https://siftery.com/product-comparison/apache-mesos-vs-apache-ambari.

Soulou (2013). Can't stack more than 42 aufs layers. https://github.com/moby/moby/issues/1171.

Spark, A. (2018). Documentación de apache spark. https://spark.apache.org/docs/latest/.

Syafrudin, M., Fitriyani, N. L., Rhee, J., Kang, Y.-S., Li, D., and Alfian, G. (2017). An open source-based real-time data processing architecture framework for manufacturing sustainability.

Verizon (2015). United states securities and exchange commission. https://www.verizon.com/about/sites/default/files/FLTX%2010K.PDF.

White, T. (2009). *Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale*. O'Reilly Media.

Yamato, Y. (2016). Performance-aware server architecture recommendation and automatic performance verification technology on iaas cloud.

YugaByte (2017). Iot fleet management - spark and kafka. https://docs.yugabyte.com/latest/develop/realworld-apps/iot-spark-kafka/.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA. USENIX Association.

Anexos

A Anexo 1: tal tal tal

```
from __future__ import print_function

import sys

from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split, col, from_json
from pyspark.sql.types import *
from pyspark import SparkContext


if __name__ == "__main__":
    if len(sys.argv) != 4:
        print("""
        Usage: structured_kafka_wordcount.py <bootstrap-servers> <subscribe-t
        """, file=sys.stderr)
        exit(-1)

    bootstrapServers = sys.argv[1]
    subscribeType = sys.argv[2]
    topics = sys.argv[3]

    sc = SparkContext().getOrCreate()

    spark = SparkSession(sc)\
        .builder\
        .appName("StructuredKafkaGPS")\
        .getOrCreate()


    schema = StructType([StructField("sensorId", IntegerType()),
                         StructField("Type", StringType()),
                         StructField("coordinates_lat", FloatType()),
                         StructField("coordinates_long", FloatType()),
                         StructField("altitude", FloatType()),
                         StructField("heading", FloatType()),
                         StructField("speed", IntegerType()),
```

```
                        StructField("speedmetric", StringType()),
                        StructField("observationTime", StringType()),
                        StructField("serverTime", StringType()),
                        StructField("temp", FloatType()),
                        StructField("date", StringType()),
                        StructField("location", ArrayType(FloatType())),
                        StructField("dateSend", TimestampType())
                       ])

# Create DataSet representing the stream of input lines from kafka
lines = spark\
    .readStream\
    .format("kafka")\
    .option("kafka.bootstrap.servers", bootstrapServers)\
    .option(subscribeType, topics)\
    .option("startingoffsets", "earliest")\
    .load() \
    .select(from_json(col("value").cast("string"), schema).alias("stream"
    #.select(from_json(col("value").cast("string"), schema).alias("data")

# lines.isStreaming()    # Returns True for DataFrames that have streamin

lines.printSchema()

# Generate running word count
#carCounts = lines.groupBy('stream.Type').count()

sch_user = StructType([
    StructField("id_user", IntegerType()),
    StructField("name", StringType())
    ])

sch_sen_user = StructType([
    StructField("id_user_sensor", IntegerType()),
    StructField("sensorId", StringType()),
    StructField("user", IntegerType())
    ])


users = spark.read.csv("Users.csv", header=True, schema = sch_user)
userSensor = spark.read.csv("UserSensor.csv", header=True, schema=sch_sen
users.printSchema()
userSensor.printSchema()
```

```
#joinUserSensor = users.alias('user').join(sc.broadcast(userSensor.alias(
joinUserSensor = users.alias('user').join(userSensor.alias('sensors'), co
joinUserSensor.printSchema()

joinData = lines.alias('stream').join(joinUserSensor.alias('data'), col('
joinData.printSchema()

# Start running the query that prints the running counts to the console
query = joinData.groupBy("data.name").count().sort(col("count").desc())\
    .writeStream\
    .outputMode('complete')\
    .format('console')\
    .trigger(processingTime='2 seconds')\
    .start()

query.awaitTermination()
```

B Anexo 2: tal tal tal

```
from __future__ import print_function

import sys
import json
import time as time_py
from urllib.request import urlopen

from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
from pyspark.sql import Row, SparkSession
#spark sql function for columns
from pyspark.sql.functions import col, expr, to_json, struct, format_number,
                                  monotonically_increasing_id, udf, unix_times
                                  from_unixtime, datediff, array #current_dat
from pyspark.sql.types import *
#math spark sql function
from pyspark.sql.functions import acos, cos, sin, lit, radians
#Usamos pymongo porque no carga la collecion en memoria (la libreria que usa
import pymongo
from pymongo import MongoClient

'''
```

```
Instancia global de la sesion de spark (singleton, estancia perezosa)
'''
def getSparkSessionInstance(sparkConf):
    if ("sparkSessionSingletonInstance" not in globals()):
        globals()["sparkSessionSingletonInstance"] = SparkSession \
            .builder \
            .config(conf=sparkConf) \
            .getOrCreate()
    return globals()["sparkSessionSingletonInstance"]

'''
Obtiene los datos del usuarios con un singleton
'''
def getDataUsers(sparkContext):
    if ("UserSensorName" not in globals()):
        # Get data from csv (in hadoop)
        sch_user = StructType([
                        StructField("id_user", IntegerType()),
                        StructField("name", StringType())
                    ])

        sch_sen_user = StructType([
                        StructField("id_user_sensor", IntegerType()),
                        StructField("sensorId", StringType()),
                        StructField("user", IntegerType())
                    ])

        spark = getSparkSessionInstance(sparkContext.getConf())
        users = spark.read.csv("Users.csv", header=True, schema = sch_user)
        userSensor = spark.read.csv("UserSensor.csv", header=True, schema=sch

        #Join data form hdfs
        joinUserSensor = users.alias('user').join(userSensor.alias('sensors')

        globals()["UserSensorName"] = joinUserSensor
    return globals()["UserSensorName"]

'''
Obtiene los puntos negros de Espania
'''
def getBlackShapes(sparkContext):
    if ("BlackShapes" not in globals()):
        # Get data from csv (in hadoop)
```

```
        sch_blk_shp = StructType([
                            StructField("", IntegerType()),
                            StructField("Address", StringType()),
                            StructField("Province", StringType()),
                            StructField("Country", StringType()),
                            StructField("numAccident", FloatType()),
                            StructField("lat", FloatType()),
                            StructField("long", FloatType())
                        ])
        spark = getSparkSessionInstance(sparkContext.getConf())
        blk_shp = spark.read.csv("blackshapes.csv", header=True, schema = sch
        blk_shp = blk_shp.withColumn("lat_min", (format_number(blk_shp.lat, 3
                withColumn("lat_max", (format_number(blk_shp.lat, 3) + 0.005)
                withColumn("long_min", (format_number(blk_shp.long, 3) - 0.00
                withColumn("long_max", (format_number(blk_shp.long, 3) + 0.00

        globals()["BlackShapes"] = blk_shp
    return globals()["BlackShapes"]

'''
Distancia en km entre dos puntos (haversine)
'''
def dist(long_x, lat_x, long_y, lat_y):
    return acos(
        sin(radians(lat_x)) * sin(radians(lat_y)) +
        cos(radians(lat_x)) * cos(radians(lat_y)) *
            cos(radians(long_x) - radians(long_y))
    ) * lit(6371.0)

#   bearing = atan2(sin(long2-long1)*cos(lat2), cos(lat1)*sin(lat2)-sin(lat1)
#   bearing = degrees(bearing)
#   bearing = (bearing + 360) % 360


'''
Obtener datos de OSM
'''
def getWay(lat, long, waysConnection):
    try:
        query = waysConnection.find({ "$and": [ {"loc": { "$near": [lat, long
                                    { "$and": [ {"tg" : {'$elemMatch':{'$elemMa
                                                {"tg" : {'$elemMatch':{'$elemMatc
```

```
                                                                    }
                                                                }
                                                            }
                                                            ]
                                                    }
                                                ]}  ).limit(1)
            if (query.count(with_limit_and_skip=True))>0:
                return query[0]
            return None
        except:
            return None

def getWayUrl(lat, long):
    try:
        url = "http://mongomaster:5000/getway/"+str(lat)+"/"+str(long)
        response = urlopen(url)
        data = json.loads(response.read().decode("utf-8"))
        return data
    except:
        return None

#Datos de velocidad
carLimit = {
    'motorway' : 120,
    'trunk' : 100,
    'primary' : 50,
    'secundary' : 90,
    'tertiary' : 90,
    'unclassified' : 60,
    'residential' : 30,
    'service' : 50,
    'road' : 80
}

busLimit = {
    'motorway' : 100,
    'trunk' : 90,
    'primary' : 50,
    'secundary' : 80,
```

```
    'tertiary' : 80,
    'unclassified' : 60,
    'residential' : 30,
    'service' : 50,
    'road' : 80
}

truckLimit = {
    'motorway' : 100,
    'trunk' : 90,
    'primary' : 50,
    'secundary' : 70,
    'tertiary' : 70,
    'unclassified' : 60,
    'residential' : 30,
    'service' : 50,
    'road' : 80
}

'''
Obtain max velocity for road type and vehicle type
'''
def getSpeedLimit(road_type, vh_type):
    if (vh_type == "truck"):
        return truckLimit.get(road_type, 120)
    elif (vh_type == "bus"):
        return busLimit.get(road_type, 120)
    else:
        return carLimit.get(road_type, 120)


'''
Obtener los datos que nos interesan de mongodb
'''
def getInfo(query, vh_type):
    speedLimit = 120
    tagSpeed=False
    tafRef = False
    name = ""
    try:
        if (query is None):
            return (name, speedLimit)
        for i in query['tg']:
```

```
            if ("maxspeed" == i[0]):
                speedLimit = int(i[1])
                tagSpeed = True
            elif ("highway" == i[0]):
                if not tagSpeed:
                    speedLimit = getSpeedLimit(i[1], vh_type)
            elif ("name" in i[0]):
                if not tagSpeed:
                    name = i[1]
            elif ("ref" in i[0]):
                name = i[1]
        return (name, speedLimit)
    except:
        return (name, speedLimit)

'''
Funcion que pasaremos por el udf que obtiene los datos de mongo OSM
Hay que pasarle las columnas latitud y longitud
'''
def reference_to_dict(l):
    d = []
    if (l[3] is not None):
        if (l[3]>0):
            if ((l[0] is not None) and (l[1] is not None)):
                if (l[2] is None):
                    l[2] = "truck"
                try:
                    #client = MongoClient("mongomaster", 27017)
                    #db = client.osm
                    #ways = db.ways
                    #d = getInfo(getWay(l[0], l[1], ways), l[2])
                    #client.close()
                    d =  getInfo(getWayUrl(l[0], l[1]), l[2])
                    return [d[0], d[1]]
                except:
                    return [None, 120]
    return [None, 120]


if __name__ == "__main__":
    if len(sys.argv) != 5:
        print("Usage: sparkStreaming2.py <zkServers> <topicIn> <kServers> <to
        exit(-1)
```

```
zkQuorum, topicIn, kServer, topicOut = sys.argv[1:]
sc = SparkContext(appName="PythonStreamingKafkaJson")
ssc = StreamingContext(sc, 10)
#inicializamos los globals
getBlackShapes(sc)
getDataUsers(sc)
kvs = KafkaUtils.createStream(ssc, zkQuorum, "spark-streaming-consumer",

# Convert RDDs of the words DStream to DataFrame and run SQL query
def process(time, rdd):
    print("========= %s =========" % str(time))
    #2018-05-28T13:52:07.0000000Z,2018-05-28T13:52:35.6721175Z
    format_1 = "yyyy-MM-dd'T'HH:mm:ss.SSSSSSS'Z'"
    a = time_py.time()

    try:
        # Get the singleton instance of SparkSession
        if (not rdd.isEmpty()):
            spark = getSparkSessionInstance(rdd.context.getConf())
            #rdd.context.clearCache()
            # Get data from kafka json to df
            df = spark.read.json(rdd.map(lambda x: x[1]))
            df = df.rdd.repartition(100).toDF()
            print(df.count())

            df = df.withColumn('observationDate', from_unixtime(unix_times
                    withColumn('serverDate', from_unixtime(unix_timestam
            #Usa current_date(), col("observationDate") en produccion
            df = df.where(datediff(col("serverTime"), col("observationDat

            joinUserSensor = getDataUsers(rdd.context)

            # join data from hdfs and stream
            joinData = df.alias('stream').join(joinUserSensor.alias('data

            dataToSend = joinData.select("Type","altitude","coordinates_l
            dataToSend = dataToSend.withColumn("id", monotonically_increa

            #Convertimos la funcion en udf
            schema4udf = StructType([StructField("addrs_name", StringType
                                     StructField("max_speed", IntegerT
                                    ])
            reference_to_dict_udf = udf(reference_to_dict, schema4udf)
```

```
#Obtenemos la georeferenciacion
dataToSend = dataToSend.withColumn("data_osm", reference_to_d
dataToSend = dataToSend.select("id", "Type","altitude","coord
                    "serverTime","heading","location","stream.
                    col("data_osm.addrs_name").alias("addrs_nar

actualCoordinates = dataToSend.select("id", "coordinates_lat"
# Cargamos los puntos negros
blackShapes = getBlackShapes(rdd.context)
# Los cruzamos con las posiciones actuales de los vehiculos
nearBlkShp = actualCoordinates.crossJoin(blackShapes)
# Obtenemos solo los puntos mas cercanos a ~1km de distancia
# How to find the most near position efficient?
# https://gis.stackexchange.com/questions/8650/measuring-accu
nearBlkShp = nearBlkShp.filter((nearBlkShp.lat_min <= nearBlkS
#Obtenemos las distancias a los puntos negros
nearBlkShp = nearBlkShp.select(col("id"), col("Address"), col
                               col("lat").alias("blk_point_la
                               dist(col('coordinates_lat'),co
#Nos quedamos con la menor
minD4 = nearBlkShp.groupBy("id").min("Distance")
#Ahora obtenemos los que tienen solo menor distancia
finalNearBlkShp = minD4.alias('mins').join(nearBlkShp.alias('
    (col('mins.id')==col('dataBlkShp.id')) & (col('mins.min(D
    select(col("dataBlkShp.id").alias('id'),
    col("dataBlkShp.Address").alias('address'),
    col("dataBlkShp.Province").alias('province'), col("dataBl
    col("dataBlkShp.blk_point_lat").alias('blk_point_lat'), c

#Join de los puntos negros con los datos
dataToSend = dataToSend.alias('data').join(finalNearBlkShp.al
dataToSend = dataToSend.select(col("data.Type").alias("Type")
                               col("data.altitude").alias("al
                               col("data.observationTime").al
                               col("data.dateSend").alias("da
                               col("data.serverTime").alias("
                               col("data.heading").alias("hea
                               col("data.location").alias("lo
                               col("data.sensorId").alias("se
                               col("data.speed").alias("speed
                               col("data.speedmetric").alias(
                               col("data.temp").alias("temp")
```

```
                                                col("data.id_user").alias("id_
                                                col("data.name").alias("user")
                                                col("data.addrs_name").alias("
                                                col("data.max_speed").alias("m
                                                col("blk_shp.address").alias("
                                                col("blk_shp.province").alias(
                                                col("blk_shp.country").alias("
                                                col("blk_shp.accidents").alias
                                                array(col('blk_shp.blk_point_l
                                                col("blk_shp.dist_to_blk_shp")
                                               )

                #Send data
                #dataToSend.printSchema()
                print(dataToSend.rdd.getNumPartitions())
                dataToSend.select(to_json(struct([dataToSend[x] for x in dataT

        except Exception as e:
            print(str(e))
            pass

        b = time_py.time()
        c = (b-a)
        print(c)

kvs.foreachRDD(process)
ssc.start()
ssc.awaitTermination()
```