E.T.S. de Ingeniería Industrial, Informática
y de Telecomunicación

Desarrollo de una aplicación Big Data sobre AWS para la monitorización de tweets sobre el COVID-19

Máster en Ingeniería Informática

Trabajo Fin de Máster

Autor: Javier Díez Zunzarren

Director: Mikel Galar Idoate

Pamplona, 9 de Febrero de 2021

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Resumen

El objetivo de este trabajo es el desarrollo de una aplicación para la monitorización de tweets sobre el COVID-19.

La aplicación estará dividida en dos partes principales: Una de extracción, tratamiento y almacenamiento de datos (tweets), en la que se utilizarán herramientas como la API de Twitter o los clústers de Hadoop de AWS (EMR), y otra que será un sitio web en el que los usuarios podrán visualizar los resultados del tratamiento de datos de la primera parte.

La red y las máquinas virtuales sobre las que se ejecutaría la aplicación estarían alojadas en AWS. Para incrementar la disponibilidad y la tolerancia a fallos de la aplicación se hará uso de herramientas de AWS como los grupos de auto-escalado y los balanceadores de carga.

Palabras clave:

- Big Data.
- AWS.
- Hadoop.
- Twitter.
- COVID-19.

Abstract

The aim of this work is the development of an application for monitoring tweets about COVID-19.

The application will be separated into two main parts: One of data (tweets) extraction, treatment and storage, where we will use tools such as the Twitter API or AWS Hadoop clusters (EMR), and another that will be a website where the users will view the results of the data processinf of the first part.

The network an virtual machines where the application wil be running will be hoste don AWS. To increase the availability and fault tolerance of the application, we will used AWS tools such as auto-scaling groups and load bañancers.

Keywords:

- Big Data.
- AWS.
- Hadoop.
- Twitter.
- COVID-19.

Índice

Índice de tablas

Índice de figuras

Introducción

El 11 de Febrero de 2020 la OMS acuñaba el término COVID-19 para referirse a una nueva enfermedad infecciosa producida por el virus SARS-CoV-2 que se estaba extendiendo a un ritmo vertiginoso a nivel mundial. La enfermedad fue detectada por primera vez en la ciudad china de Wuhan y en un principio, debido a la novedad del virus, fue tomada por una enfermedad más leve y menos peligrosa que la gripe común, sin embargo, a día de hoy ha demostrado ser mucho más peligrosa cobrándose más de 1,7 millones de muertos.

Como es natural, las instituciones de diferentes estados han creado herramientas para que la población puede seguir la evolución de la enfermedad a partir de los datos oficiales de contagios, como es el caso del CNE (el Centro Nacional de Epidemiología) en España [1], que proporciona un mapa en el que se puede visualizar la distribución geográfica de la pandemia. La desventaja de este tipo de herramientas es que, al trabajar a partir de los datos confirmados, son reactivos y no preventivos.

En la actualidad existe una gran cantidad de redes sociales de las que se puede extraer información en tiempo real. Entre estas redes destaca Twitter, que resulta perfecta para el análisis de datos ya que su uso se basa en la publicación de textos de pequeño tamaño por parte del usuario (mucho más fáciles de tratar que otro tipo de datos como, por ejemplo, imágenes). Twitter ha sido ya utilizado con éxito para la predicción de acontecimientos como el vencedor de las elecciones Estado-unidenses de 2016.

El objetivo de este trabajo es desarrollar un sitio web en el que los usuarios puedan visualizar un mapa mundial que refleje, en tiempo real, la cantidad de tweets relacionados con el COVID-19 publicados en la última hora en cada país. Este sitio web podría ser utilizado posteriormente como herramienta para el estudio de la evolución geográfica del impacto del virus en las redes.

Como fuente de datos se utilizará la API de Twitter que, como se explicará posteriormente, permite crear un flujo de datos en tiempo real con reglas de filtrados. De esta forma podremos recibir únicamente los tweets que contengan hashtags como #COVID-19 o #coronavirus. La licencia de pago de la API de Twitter más barata de 150$ mensuales, de modo que utilizaremos la licencia gratuita para desarrolladores que, aunque cuenta con limitaciones, nos servirá para testear la aplicación.

La red y las máquinas sobre las que se ejecutará la aplicación estarán alojadas en AWS. Amazon Web Services (AWS) es un conjunto de herramientas y servicios de computación en la nube proporcionados por la empresa Amazon. Entre los servicios que usaremos para construir la aplicación están Amazon EC2 (máquinas virtuales), VPC (redes y subredes virtuales), Amazon S3 (almacenamiento de ficheros) y Amazon EMR (clústers de Hadoop elásticos).

Evidentemente AWS cobra por todos estos servicios, sin embargo, podremos acceder a todos ellos de forma gratuita gracias a AWS Educate. AWS Educate ofrece sin costo acceso a contenido, formaciones, itinerarios y servicios de AWS a estudiantes de una lista de centros educativos. En esta lista se encuentra la Universidad Pública de Navarra (UPNa).

Para tratar la gran cantidad de datos que entrarán a la aplicación desde la API de Twitter deberemos recurrir a técnicas de Big Data. Utilizaremos la herramienta Spark, más concretamente Spark Streaming que nos permitirá tratar un flujo constante con gran cantidad

de datos (los tweets). Spark funcionará sobre un clúster de Hadoop que nos proporcionará el servicio de AWS EMR (Elastic Map Reduce).

Para la creación del sitio web utilizaremos Angular. Angular es un framework opensource que permite crear de forma sencilla y rápida aplicaciones web de una sola página.

Capítulo 1 - Preliminares

Antes de lanzarnos a comentar las partes y la arquitectura del proyecto es conveniente dejar claros algunos conceptos que son necesarios para entender todos los componentes de este.

Cuando se trabaja con una cantidad de datos tan grande como la que emplea en este proyecto (sobre todo cuando no puede permitirse una alta latencia en los cálculos) es necesario utilizar unas técnicas diferentes a las convencionales. En la sección 1.1 se explicará qué es el Big Data, cuáles son las arquitecturas más comunes para aplicaciones Big Data y cuáles son las herramientas más utilizadas para este tipo de aplicaciones, centrándonos en Apache Spark y Spark Streaming (Sección 1.1.2.7).

A continuación, en la sección 1.2, se explicará por encima la red social Twitter, ya que será nuestro proveedor de datos para la aplicación. Se explicarán también las funciones de la API de Twitter, del tipo de operaciones que proporciona al usuario y de los tipos de cuentas, precios y limitaciones de unas y otras.

Por último, en la sección 1.3 se explicará la computación en la nube, los diferentes tipos de nubes y las ventajas y desventajas de utilizar un IaaS para alojar un servicio (nosotros utilizaremos AWS) frente a utilizar software 'On-Premises'.

1.1 Big Data

La cantidad de datos generados en todo el mundo ha crecido exponencialmente desde el nacimiento de internet. En 2018 se alcanzó de cifra de 33 zettabytes (que equivalen a mil millones de terabytes) y se espera que para 2035 la cantidad sea mayor a dos mil zettabytes [2].

Este aumento hace que se generen volúmenes de datos tan grandes para cuyo análisis se requieren nuevas técnicas de tratamiento de datos, ya que con los métodos tradicionales podría requerirse una cantidad de tiempo inasumible. El análisis de estas cantidades ingentes de datos es lo que se conoce actualmente como Big Data.

Las características principales del Big Data suelen resumirse en los que se conoce comúnmente como las tres V [3]:

- Velocidad: La velocidad es el factor clave que impulsó el desarrollo de las técnicas de Big Data. Cada día surgen más aplicaciones que requieren una respuesta casi inmediata o incluso tratamiento de datos en tiempo real, como puede ser un software que monitorice las constantes vitales de los ingresados de un hospital.
- Volumen: Como hemos dicho la cantidad de datos a aumentado exponencialmente en los últimos años. En parte se debe a que el costo del almacenamiento de datos es cada vez más barato.
- Variedad: Los datos sobre los que trabajar pueden venir de fuentes muy diferentes y, por lo tanto, pueden estar estructurados de formas diferentes. Pueden ser datos perfectamente estructurados, datos semiestructurados (archivos XML, HTML, JSON, etc) o datos no estructurados (Word, imágenes, PDF, etc).

Hay quienes añaden a esta una cuarta V:

- Veracidad: Es importante tener presente hasta qué punto podemos confiar en la información que procesamos para saber cuánto podemos confiar en la información que

sacamos de ellos. Por esto es necesario invertir tiempo en asegurar la calidad de los datos obtenidos y realizar un proceso de tratamiento previo para, por ejemplo, visualizar los outliers y determinar si estos son veraces.

O incluso se llega a hablar de las siete V:

- Valor: El valor de un dato depende de la utilidad que se pueda sacar de este y de su rareza. Si, por ejemplo, estamos estudiando los datos de diferentes personas para determinar si una enfermedad muy rara afecta más a un sector de la población que a otro, los datos de las personas afectadas por la enfermedad tendrán más valor que el resto.
- Visualización: Es importante tener una forma de transformar la información obtenida en un contenido gráfico que aporte información de forma rápida y clara. Estas visualizaciones son muy relevantes a la hora de encontrar patrones y diferencias en los datos.
- Variabilidad: Los datos pueden variar en el tiempo o según la fuente. Por eso es necesario adaptar los modelos al entorno donde se vayan a utilizar y revisarlos periódicamente.

Como vemos es un tema que no está muy formalizado y cualquiera puede escribir un artículo en que hable de cuáles son las V del Big Data para él (esto se ha convertido en un cliché en el mundo del Big Data y existen posts satíricos en los que se llega a hablar de las 42 V [4]).

Antes de profundizar en las herramientas y metodologías que nos permiten hacer uso de técnicas de Big Data es importante tener claro cuáles son los dominios reales en los que se utiliza. La empresa Oracle nos proporciona un listado de los casos de uso más relevantes entre los que encontramos la industria, la venta de productos al por menor, la salud, los combustibles como el petróleo y el gas, las telecomunicaciones y los servicios financieros. A estos campos podríamos añadir otros como el uso gubernamental o en defensa, el uso en Deep-Learning, en astronomía y en medio ambiente [5].

1.1.1 Arquitecturas Big Data

Una arquitectura de referencia es un documento o conjunto de documentos a los que un director de proyecto u otra parte interesada puede consultar las mejores prácticas. El objetivo de una arquitectura de referencia es crear un estándar abierto, uno que todas las organizaciones puedan utilizar para su beneficio.

Utilizar una arquitectura de referencia de Big Data abierta presenta múltiples beneficios:

- Proporciona un lenguaje común para las distintas partes.
- Fomenta el cumplimiento de estándares.
- Proporciona métodos para la implementación de tecnología y para la resolución de conjuntos de problemas similares.
- Ilustra y mejora la comprensión de los diversos componentes, procesos y sistemas de Big Data.
- Facilita el análisis de los estándares candidatos para la interoperabilidad, portabilidad, reutilización y extensibilidad.

1.1.1.1 Arquitectura NIST

El Instituto Nacional de Estándares y Tecnología (NIST) ofrece una arquitectura de referencia para aplicaciones Big Data llamado NIST Big Data Reference Architecture (NBDRA). El Instituto Nacional de Estándares y Tecnología (NIST) es una de las organizaciones líderes en el desarrollo y publicación de estándares. La Arquitectura de referencia de Big Data de NIST, que se presenta en la Figura 1, es un enfoque neutral del proveedor y puede ser utilizada por cualquier organización que tenga como objetivo desarrollar una arquitectura de Big Data [6].

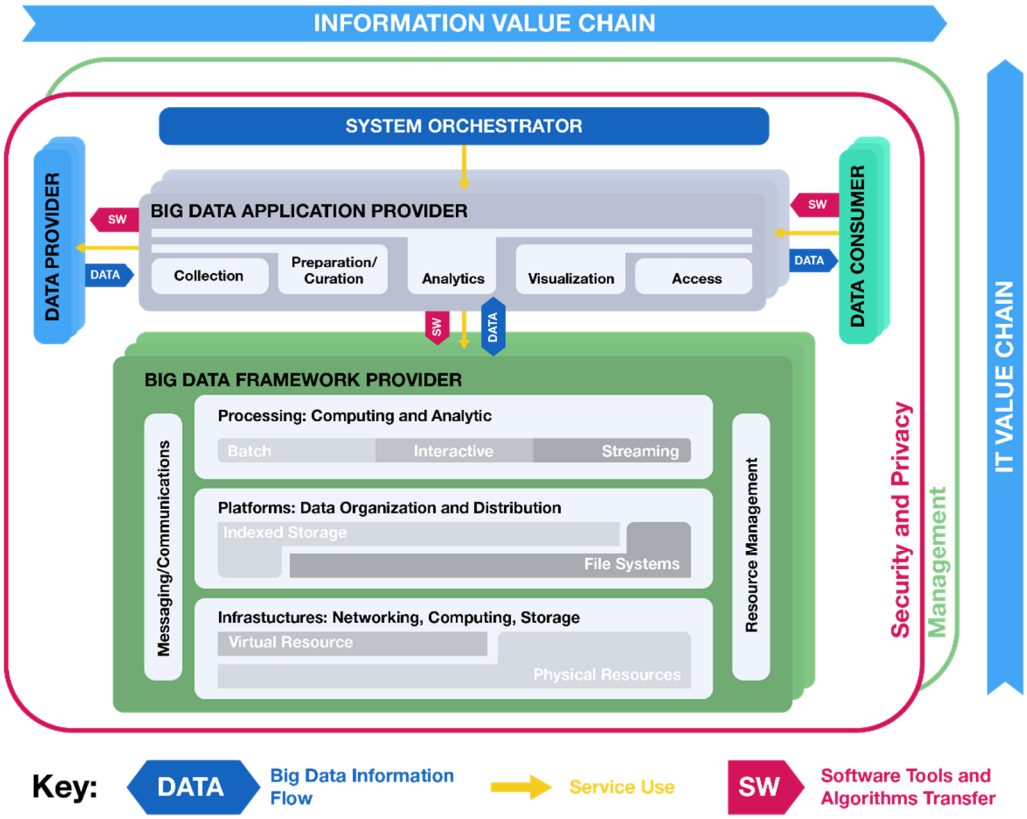


Figura 1 - Arquitectura NIST

Los componentes de esta arquitectura se organizan en torno a dos ejes, el vertical representa el valor de las tecnologías de la información (IT) y el horizontal el valor de la información. Si la recorremos de izquierda a derecha vamos que en un principio el proveedor de datos proporciona esto, datos, que es el nivel más bajo de la pirámide DIKW (jerarquía que representa la relación entre Dato, Información, Conocimiento y Sabiduría) [7]. Los datos son tratador y convertidos en información que es consumida por el usuario a la derecha de la arquitectura (con lo que han ganado 'valor'), que a su vez procesa esa información y la convierte en conocimiento. A lo largo del eje de TI, el valor se crea al proporcionar redes, infraestructura, plataformas, herramientas de aplicación y otros servicios de TI.

Los cinco componentes principales de esta arquitectura son [8]:

- Orquestador del sistema: El orquestador del sistema organiza, coordina y gestiona los sistemas informáticos del resto de componentes. Es el componente que se ocupa de garantizar que el resto de los componentes trabajan conjuntamente de manera correcta

y eficiente. Para lograr esto el orquestador del sistema utiliza flujos de trabajo, automatización y procesos de gestión de cambios. Una metáfora muy utilizada a la hora de explicar el rol del orquestador del sistema es, como su propio nombre indica, el del director de una orquesta. En esta metáfora los instrumentos de la orquesta serían los diferentes componentes de la arquitectura donde, si no fuera por el director de orquesta, cada uno tocaría a su propio ritmo en vez de estar sincronizados.

- Proveedor de datos: El proveedor de datos es el encargado de proporcionar datos de entrada al sistema. Estos datos pueden provenir de los humanos (de su actividad en redes sociales, por ejemplo), pueden ser generados automáticamente por sensores o pueden venir desde otros sistemas (como las transacciones de un sistema bancario), por citar algunos ejemplos. Estos datos pueden estar estructurados o no y venir de fuentes muy diferentes, el proveedor de datos no es el responsable de la preparación de estos.
- Proveedor de aplicaciones Big Data: El proveedor de aplicaciones Big Data es el que contiene las funcionalidades necesarias para transformar los datos obtenidos del proveedor en información valiosa. El alcance y los tipos de aplicaciones que emplea este componente varían mucho y dependen de la naturaleza de la empresa. Para las empresas financieras, las aplicaciones pueden incluir software de detección de fraude, aplicaciones de puntaje crediticio o software de autenticación. En las empresas de producción, los componentes del proveedor de aplicaciones de Big Data pueden ser software de gestión de inventario, optimización de la cadena de suministro u optimización de rutas. Es el componente responsable de las siguientes tareas:
    - Recolección: En general, la actividad de recopilación de datos se lleva a cabo utilizando el interfaz del proveedor de datos. Este puede ser un servicio general, como un servidor de archivos o un servidor web configurado por el orquestador del sistema para aceptar o realizar recopilaciones de datos específicas, o puede ser un servicio específico de la aplicación diseñado para extraer datos o recibir envíos de datos del proveedor.
    - Preparación: La preparación de los datos incluye la selección de los datos, la limpieza de éstos, la construcción de nuevas variables derivadas otras, la integración de datos de diferentes fuentes y su formateo. También se llevan a cabo tareas más específicas como la eliminación de outliers y la normalización de datos.
    - Análisis: En esta tares se aplican las técnicas de extracción de conocimiento de los datos. El código y las técnicas utilizadas dependerán de la lógica de nuestra aplicación.
    - Visualización: Es la tares que se encarga de utilizar los datos procesados para preparar un output que pueda ser entendido por un usuario consumidor. Este output puede ser simplemente cadenas de texto o pueden ser visualizaciones gráficas o incluso interactivas con el usuario.
    - Acceso: Esta tarea realiza la misma tarea que el módulo de recolección, pero en el extremo opuesto de la arquitectura. Se encarga de proporcionar una interfaz que permita al consumidor de datos acceder a los outputs generados en la capa de visualización.
- Proveedor de marco de trabajo Big Data: Este componente se ocupa de proporcionar recursos y servicios que serán utilizados por el proveedor de aplicación Big Data y provee el núcleo de la arquitectura de la aplicación. La mayoría de los entornos de Big Data utilizan almacenamiento y procesamiento distribuidos y el marco de software de código

abierto Hadoop para diseñar estos roles. Este módulo puede ser subdividido en tres submódulos:

- Capa de infraestructura: Esta capa se ocupa de cubrir y gestionar las necesidades de redes, computación y almacenamiento para garantizar que la gran cantidad de datos pueda almacenarse y transferirse eficientemente, de forma segura y escalable. El requisito clave de este tipo de almacenamiento es que sea capaz de manejar cantidades muy masivas de datos y que siga escalando con el crecimiento de la organización, y que pueda proporcionar las operaciones de entrada/salida por segundo necesarias.
- Capa de plataforma: Esta capa proporciona una colección de funciones que facilita el procesamiento de datos con un alto rendimiento. La plataforma realiza las tareas de integrar, administrar y aplicar trabajos de procesamiento a los datos. Es la capa que se encarga, por ejemplo, de distribuir los datos en diferentes nodos. Una de las infraestructuras más utilizadas para las soluciones de Big Data es el framework Hadoop.
- Capa de procesamiento: Esta es la capa en la que se ejecuta el código sobre los datos. Ejecuta comandos que realizan operaciones en tiempo de ejecución en los conjuntos de datos. En esta capa, tiene lugar el análisis real.

- Consumidor de datos: Realiza el rol contrario al proveedor de datos. Este componente accede al interfaz de acceso del proveedor de aplicaciones Big Data y visualiza los resultados de la extracción de información sobre los datos de entrada.

1.1.1.2 Arquitecturas Lambda y Kappa

Sin embargo, la arquitectura NIST no tiene en cuenta que en los últimos años se han desarrollado dos paradigmas para el tratamiento de grandes cantidades de datos. Cuando nos enfrentamos a un problema que deseamos resolver empleando técnicas Big Data, debemos tener en cuenta cuales son estos dos paradigmas, como encajan en nuestro problema y que herramientas se han desarrollado para cada uno. Estos dos paradigmas son:

- Paradigma Batch [9]: Este paradigma se apoya principalmente en el paradigma Map Reduce (que se expondrá más adelante) y consiste en procesar grandes cantidades de datos por 'lotes'. Los datos no se procesan en tiempo real conforme se recopilan, sino que se lleva a cabo un proceso de recopilación y una vez este ha terminado es cuan empiezan a procesarse. Es una buena solución cuando no se quiere que los cálculos se realicen con una periodicidad alta, por ejemplo, para generar informes y analizar las cuentas de una gran empresa a finales de mes. Este paradigma puede dividirse en cuatro etapas:
  - Ingesta de datos: Como hemos dicho es un proceso que se realiza antes del procesamiento de datos, nunca al mismo tiempo. Se pueden destacar herramientas como Apache Flume o Apache Sqoop.
  - Procesamiento de datos: El paradigma más utilizado es Map Reduce y destacan tecnologías como Apache Pig, Hive y Apache Spark.
  - Almacenamiento de datos: Los datos suelen almacenarse utilizando el sistema de archivos de los clústers de Hadoop (HDFS) y otras herramientas como Apache HBase.

  - Servicio de datos: En esta etapa los datos son proporcionados al consumidor. No destaca especialmente ninguna tecnología.
- Paradigma Streaming [10]: A diferencia del paradigma anterior, en el paradigma Streaming (o paradigma tiempo real) los datos se procesan conforme entran al sistema. Este paradigma es necesario en aplicaciones que necesitan una respuesta inmediata como, por ejemplo, un sistema de monitorización de constantes vitales de un hospital. Existen herramientas de mensajería especializadas para integrarse con este tipo de sistemas como Apache Kafka. Entre las tecnologías que soportan este paradigma pueden distinguirse dos familias:
  - Full-streaming: La ingesta de datos se produce de forma continua conforme llegan. Pueden destacarse Apache Flink, Apache Storm o Apache Samza.
  - Microbatch: La ingesta de datos de produce en pequeños lotes que, a efectos prácticos no se diferencian demasiado a ingestas en tiempo real. Destacan Spark Steaming y Storm Trident.

Es posible definir una arquitectura que no renuncie a ninguno de los dos paradigmas anteriores. La arquitectura Lambda fue propuesta por primera vez en 2012 por Nathan Marz y más tarde fue publicada el libro 'Big Data: Principles and best practices of scalable realtime data systems' [11].

Podemos ver en la Figura 2 que los datos de entrada son servidos a dos capas diferentes a la vez: la capa Batch y la capa Stream (speed layer en la figura). La capa Batch tratará los datos como cualquier sistema que siga el paradigma Batch y pasará los resultados a la capa de servicio. La capa de servicio se encargará de indexar los resultados de la capa Batch y servirlo a los consumidores. La capa Stream compensa la alta latencia de la capa Batch empleando únicamente los datos nuevos [12].

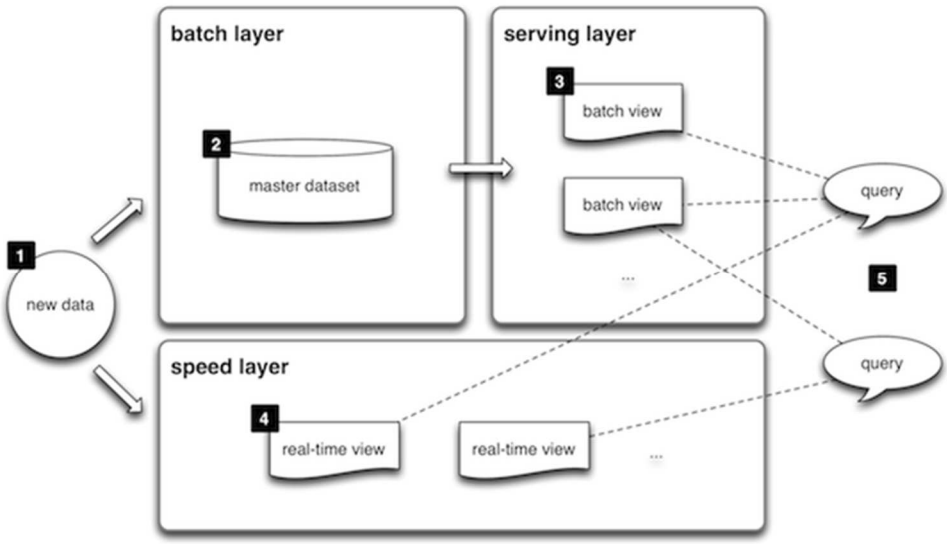


Figura 2 - Arquitectura Lambda

En 2014 Jay Kreps cuestiona en un artículo los puntos débiles de la arquitectura Lambda y propone eliminar la capa Batch y dejar únicamente la capa Stream. La capa Stream, a diferencia de la capa Batch, no tiene principio ni fin, sino que es un proceso continuo. Esto hace que podamos plantear un proceso de Batch como un proceso de Streaming acotado en el tiempo, lo que convertiría a los primeros en un subconjunto de los segundos. De esta forma nace la

arquitectura Kappa, que es una simplificación de la arquitectura Lambda en la que todo el procesamiento se realiza en una capa llamada capa de tiempo real que da soporte a procesos de Batch y de Stream [13].

La arquitectura Kappa, que podemos ver en la Figura 3, se apoya en cuatro pilares:

- Todos los procesos son un Stream: como hemos dicho los procesos de Batch pueden ser tratados como un subconjunto de los de Stream.
- Los datos iniciales no se modifican: Puesto que la información inicial no se modifica, un estado concreto puede ser recalculado.
- Solo existe un flujo de procesamiento: Por lo que la sincronización se simplifica y no hay que crear dos tipos de código (uno para cada capa de la arquitectura anterior).
- Es posible volver a lanzar un procesamiento: Es posible modificar un proceso concreto y su configuración para variar los resultados obtenidos partiendo de los mismos datos de entrada.

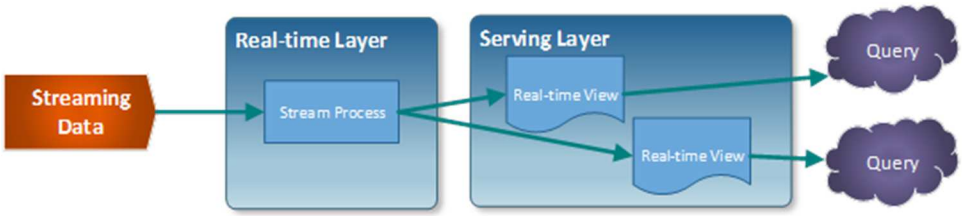


Figura 3 - Arquitectura Kappa

1.1.2 Hadoop

Uno de los términos que más se ha ido repitiendo conforme hablábamos de arquitecturas Big Data es Hadoop. Hadoop es un proyecto Open Source de la fundación Apache producido en 2006 y desarrollado originalmente en Java. Es un entorno de trabajo que proporciona herramientas para almacenamiento masivo de datos y ejecución de estos, acorde con las necesidades del Big Data.

El éxito de Hadoop radica en que puede distribuir tanto el almacenamiento de los datos como el tratamiento de estos entre el número deseado de equipos. Estos equipos pueden ser perfectamente ordenadores de sobremesa (o incluso dispositivos de menos potencia). Lo ideal es que estos sean grandes equipos dedicados únicamente al procesamiento de datos, pero se elimina la necesidad de comprar hardware para computación distribuida.

En 2006 los dos componentes principales de Hadoop, HDFS y MapReduce, fueron cedidos a Apache Software Foundation como Open Source, lo que impulsó que este fuera adoptado como herramienta por muchas empresas de la industria.

En la Figura 4 puede verse como en una primera versión Hadoop contaba con dos componentes principales: HDFS (el sistema de almacenamiento) y MapReduce (que se encarga del procesamiento de los datos). Sin embargo, esta organización hacía que el único motor de ejecución que se pudiese utilizar en Hadoop fuera MapReduce. Por esto, en una segunda versión se introdujo YARN, un software intermedio que permite la integración de otros motores de ejecución y otras herramientas desarrolladas por terceros [14].

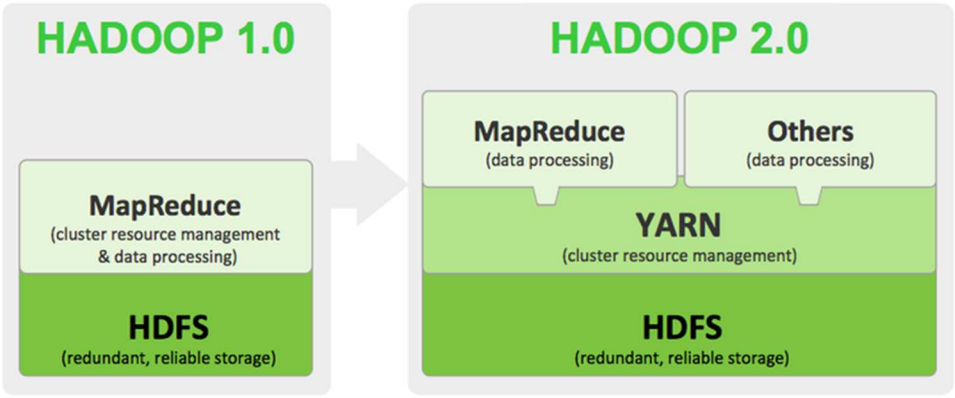


Figura 4 - Componentes Hadoop 1.0 vs Hadoop 2.0

1.1.2.1 HDFS

Hadoop cuenta con un sistema de almacenamiento de archivos propio, Hadoop Distributed File System (HDFS), que le permite almacenar los datos de forma distribuida. HDFS hace una división por bloques de cada fichero y lo distribuye entre los nodos esclavos (Datanodes) de forma replicada. Como vemos en la Figura 5 el nodo maestro que distribuye estos datos se llama Namenode [15].

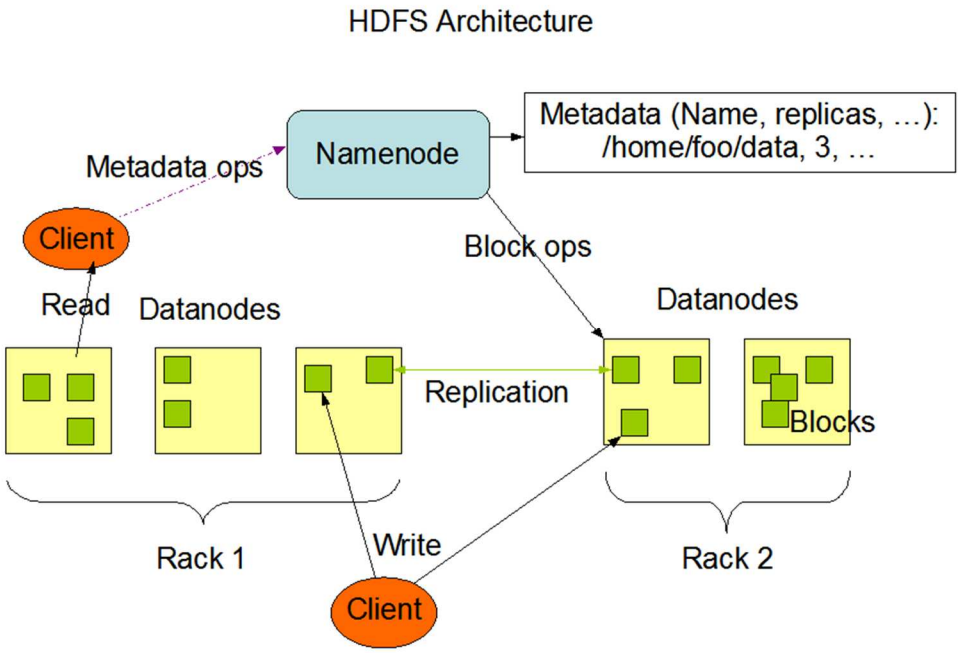


Figura 5 - Arquitectura HDFS

El nodo maestro se encarga de manejar los metadatos de los ficheros almacenados en los nodos esclavos, mapea la ubicación de cada uno de los bloques y también regula el acceso del cliente a los archivos. Los nodos esclavos se encargan de crear, eliminar o replicar bloques según demande el nodo maestro.

Puede verse en la Figura 6 como HDFS está diseñado para almacenar de manera confiable archivos muy grandes en máquinas en un clúster grande. Almacena cada archivo como una secuencia de bloques. Los bloques de un archivo están replicados para incrementar la

tolerancia a errores. Todos los bloques de un archivo, excepto el último bloque, tienen el mismo tamaño.

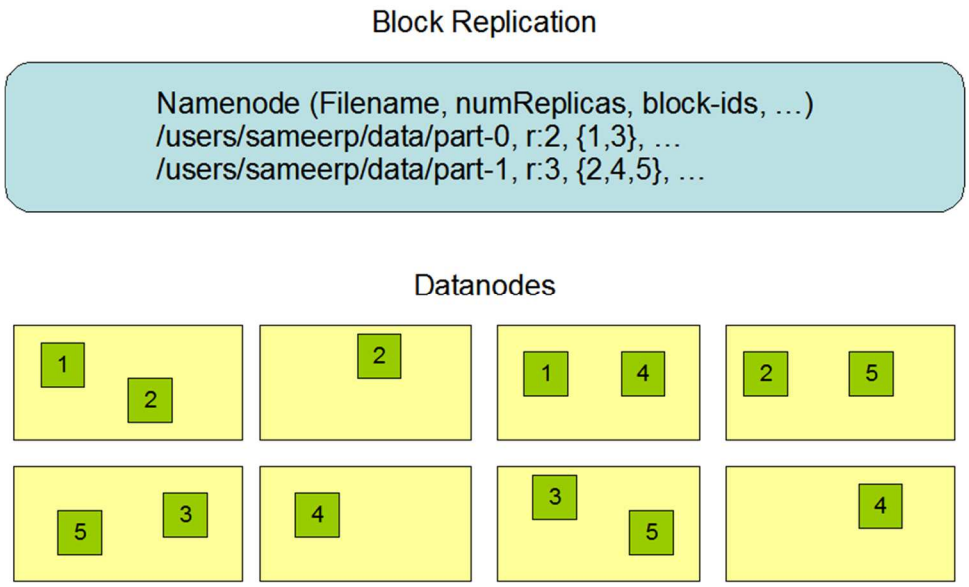


Figura 6 - Ejemplo de replicación de bloques en Hadoop

HDFS sigue el modelo Write Once Read Many, lo que significa que no se pueden modificar los archivos almacenados en este, pero si se pueden añadir nuevos datos. Cuando un cliente quiere realizar una operación de escritura, este debe comunicárselo primero al nodo maestro. El nodo maestro comprobará los permisos y responderá al cliente con la dirección de los Datanodes en los que podrá escribir el cliente. El primer Datanode entonces copiará los datos en otro Datanode, que después los copiará a un tercero, con el objetivo de tener los datos replicados. Para realizar una operación de lectura, el cliente consultará al nodo maestro la localización de los datos. Cuando este haya comprobado los permisos del cliente, enviará la localización de los Datanodes que contienen los bloques que componen el fichero al cliente.

1.1.2.2 MapReduce

MapReduce es el paradigma de programación y de procesamiento de Hadoop. Consiste en dividir el trabajo en múltiples tareas independientes que se pueden paralelizar para procesar cantidades masivas de datos en un clúster. En los sistemas tradicionales, las técnicas se centraban en mover los datos a los sistemas de almacenamiento. En cambio, lo que hace Hadoop es acercar el procesamiento al lugar en donde están almacenados los datos con el objetivo de paralelizar los procesos, haciendo el sistema más escalable y aumentando el rendimiento [16].

Como indica su nombre, MapReduce cuenta con dos procesos principales (que pueden verse en la Figura 7):

- Map: Consiste en transforma un conjunto de datos de entrada en forma de par clave/valor a otro conjunto de datos de salida también en forma de par clave/valor. De un par de entrada puede generarse un par de salida, ninguno o varios. Que los datos de salida tengan una forma clave/valor facilita el proceso intermedio entre los procesos Map y Reduce llamado Shuffle, que se encargará de juntar los pares clave/valor

generados en los diferentes nodos y ordenarlos. Este proceso se realiza en los nodos esclavos.

- Reduce: Recoge todos pares producidos en el proceso anterior, los agrupa por claves y los transforma en un producto final. Por cada clave se realiza una operación Reduce que da un solo resultado, por lo que tendremos tantos resultados como claves diferentes se hayan generado. También es posible, dependiendo del problema, realizar un Reduce previo a la fase se Shuffle (llamado combiner) con el fin aligerar su carga y reducir el tráfico de red.

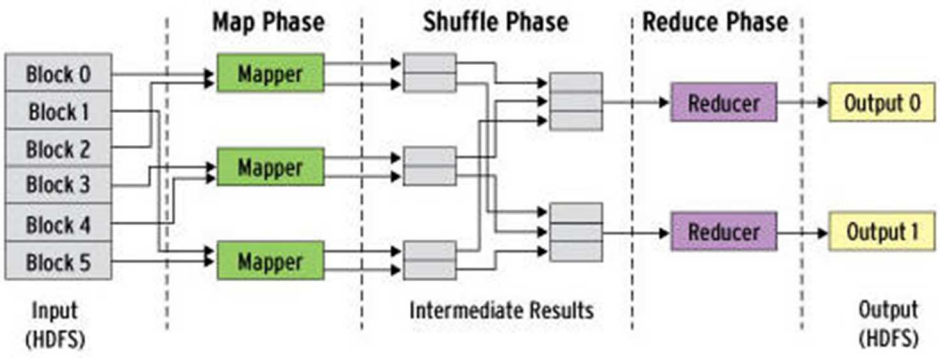


Figura 7 - Diagrama con fases de MapReduce (Fuente: https://www.admin-magazine.com/HPC/Articles/MapReduce-and-Hadoop)

1.1.2.3 YARN

Como ya hemos comentado YARN (Yet Another Resouce Negotiator) es el framework que permite que Hadoop pueda soportar varios motores de ejecución aparte de MapReduce. También realiza el rol de planificador de los trabajos que se encuentran en ejecución.

Como vemos en la Figura 8, YARN está dividido en tres componentes principales:

- Resource Manager: Es el que se encarga de gestionar y distribuir los recursos del clúster de YARN. Tiene unos criterios de planificación en función de los cuales asigna recursos a las aplicaciones. No se encarga de monitorizar el estado de ninguna aplicación ni ofrece garantías de ejecución. También es el responsable de atender las peticiones de jobs, negociar que contenedor va a utilizar una aplicación y reiniciar jobs en caso de errores. Cuenta con una lista de los Node Manager activos y de los recursos disponibles.
- Node Manager: Se encarga de gestionar los jobs siguiendo las instrucciones del Resource Manager y proporciona a las aplicaciones los recursos necesarios (en cuanto a computación) en forma de contenedores. También se encarga de monitorizar el uso de recursos (CPU, red, memoria, etc) de los contenedores.
- Application Master: Es el responsable de negociar con el Resource Manager los recursos necesarios para su aplicación y monitorizar su estado y progreso. También coordina la ejecución de los jobs en los que está dividida su aplicación.

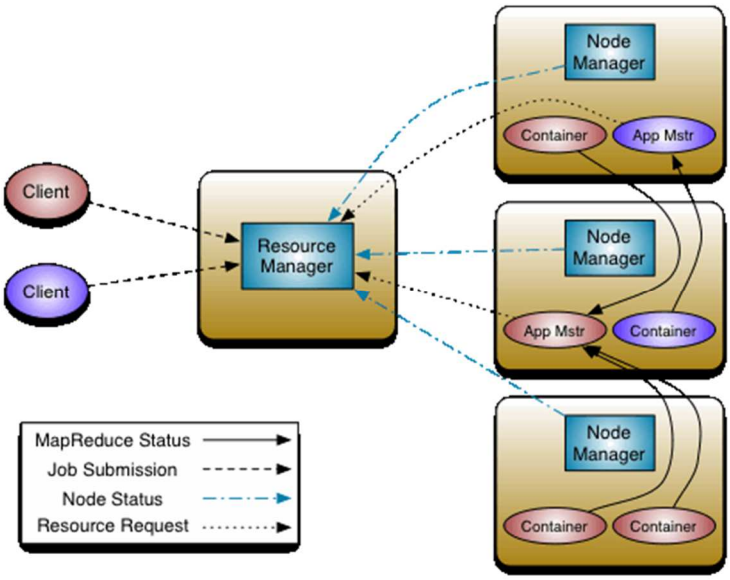


Figura 8 - Arquitectura YARN

1.1.2.4 Ecosistema Hadoop

Hadoop no es un sistema Open Source independiente, sino que es un complejos sistema de diferentes proyectos muy diversos que funcionan conjuntamente. Aunque comenzó como un proyecto individual, con el tiempo se fueron creando nuevos proyectos para cubrir las debilidades de este y añadir nuevas funcionalidades. Estos proyectos abarcan áreas como el almacenaje de los datos, el procesamiento de los datos, lenguajes de scripting, bases de datos, herramientas analíticas, lenguajes de query, etc.

Muchos de estos componentes son proyectos Open Source de la Fundación Apache, sin embargo, otros han sido creado de manera propietaria por empresas que han comercializado diferentes versiones empaquetadas de Hadoop, las llamadas distribuciones que explicaremos más adelante.

Algunos de los proyectos más importantes entrono a Hadoop (Figura 9) son:

- Pig: Pig proporciona un lenguaje de alto nivel (Pig Latin) con una sintaxis parecida a SQL que traduce a MapReduce. Convierte una descripción de alto nivel en la que se describe como deben ser las operaciones a jobs de MapReduce, mejorando la productividad de los desarrolladores.
- Hive: Proporciona un interfaz que permite transformar sentencias SQL a sentencias Pig o, directamente, a MapReduce. No es la solución más aconsejable cuando se desea realizar consultas de baja latencia. Facebook utiliza Hive hasta en un 90% de las operaciones.
- Zookeper: Es un servicio centralizado que se ocupa del mantenimiento de la configuración, de proveer sincronización distribuida y de proveer servicios de agrupación. Es tolerante a fallos y suele desplegarse en 3 nodos.
- HBase: Proporciona una base de datos de columnas distribuida totalmente sobre HDFS que puede escalar horizontalmente. Emplea un modelo de datos muy similar al de Google Big Tabla, que fue diseñado para soportar acceso aleatorio a una gran cantidad de datos estructurados.

- Spark: Es un framework de computación muy similar a su predecesor, MapReduce, pero que hace uso de unas estructuras de datos llamada RDD. Consigue una velocidad de procesamiento cien veces superior a MapReduce.
- Mahout: Proporciona un entorno y herramientas para desarrollar aplicaciones de Machine Learning que sean escalables. Ofrece herramientas para resolver, por ejemplo, problemas de clustering o de clasificación.
- Sqoop: Es una herramienta de ingesta de datos que permite exportar e importar datos estructurados entre Hadoop y bases de datos relacionales. También permite transformar directamente tablas de bases de datos a tablas Hive.
- Storm: Es una herramienta para la ingesta de datos en tiempo real. En general suele ir acompañado de Apache Kafka.
- Flume: Es una herramienta de ingesta de datos que permite incorporar grandes cantidades de estos desde diferentes fuentes (tanto estructurados como no estructurados). Se caracteriza por ser muy flexible.
- Oozie: Es el planificador de tareas para administrar los jobs de Hadoop. Se ocupa de gestionar los jobs y permite tratarlos como una sola unidad lógica.
- Kafka: Es un sistema centralizado de intercambio de mensajes que sigue el modelo publicador/suscriptor. Es muy utilizado para la ingesta de datos en streaming.

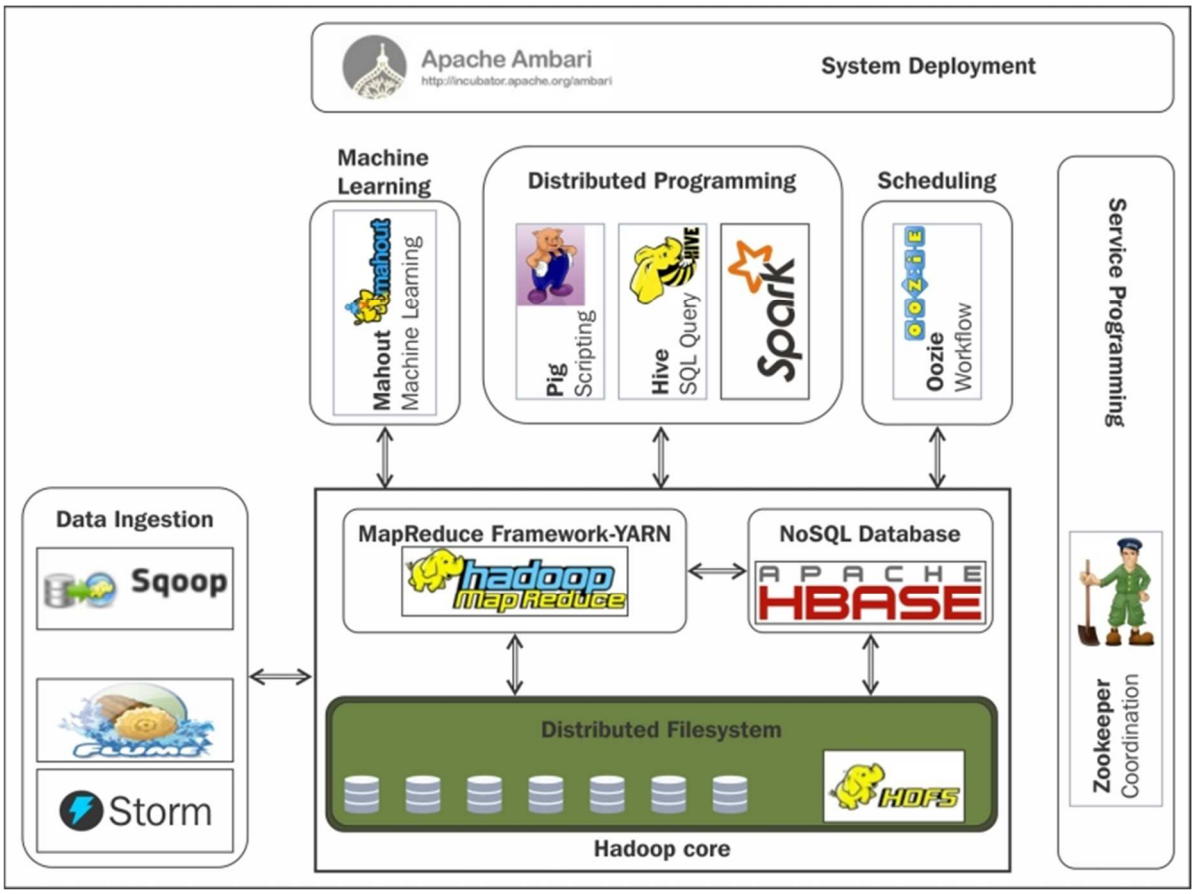


Figura 9 - Ecosistema Hadoop (Fuente: https://empresas.blogthinkbig.com/el-ecosistema-hadoop-iii-una-gran/)

1.1.2.5 Distribuciones

Es posible que, al haberse desarrollado como proyectos independientes, algunos de los componentes del ecosistema Hadoop presenten incompatibilidades entre versiones. Para suplir este problema algunas empresas han empaqueta selecciones estables de componentes e incluso han desarrollado componentes nuevos metiéndolos en estos paquetes. La venta de estas distribuciones suele venir acompañada también de garantías de soporte. Algunas de las distribuciones más importantes del mercado actualmente son [17]:

- Cloudera: Fue la primera empresa que, en 2013, oriento su estrategia empresarial a empaquetar componentes de Hadoop y comercializarlos. Su arquitecto jefe es Dough Cutting, uno de los creadores originales de Hadoop. Uno de los proyectos desarrollados por Cloudera que más popularidad ha adquirido es Apache Impala, un motor de búsqueda para consultas SLQ para Hadoop.
- MapR Technologies: Se diferencia principalmente del resto de las distribuciones por ofreces una mayor tolerancia a fallos, una alta disponibilidad y un rendimiento máximo. Uno de sus proyectos más populares es Apache Drill, que se utiliza para procesamiento de datos en tiempo real.
- HortonWorks: Todas las tecnologías que emplea esta distribución son tecnologías open source de Apache. Cuenta con una herramienta propia para administrar y gestionar los clústers llamada Apache Ambari.

1.1.2.6 Apache Kafka

Apache Kafka es un proyecto open source que consiste en un sistema intercambiador de mensajes que sigue el patrón publicador/suscriptor. Esto quiere decir que existe un nodo central en el que los consumidores se suscriben a diferentes 'topics' y al que los productores envían mensajes, y este nodo central se encarga de distribuirlos entre los diferentes suscriptores. Con el tiempo el producto se ha ido convirtiendo en una de las principales plataformas de streaming para la ingesta de datos en aplicaciones de Hadoop. Algunas de sus principales características son [18]:

- Alto rendimiento: Características como el cumplimiento del principio 'zero copy', la capacidad de escalado horizontal o la escritura secuencial en disco hacen de Apache Kafka una herramienta de intercambio de mensajes extremadamente rápida.
- Distribuido: El despliegue puede distribuirse en varios nodos de forma transparente al usuario consiguiendo una mayor tolerancia a fallos y una alta capacidad de escalado horizontal.
- Durabilidad: Los mensajes recibidos persisten en el sistema de ficheros, con lo que, si un cliente consumidor se conecta, puede recibir mensajes que se hayan enviado antes de esta conexión.
- Replicación: La información se replica entre los nodos del clúster consiguiendo mayor tolerancia a fallos y la capacidad de recuperar la información perdida cuando se cae un nodo.
- Otras: Utiliza mensajes TCP para la comunicación, por lo que puede ser utilizado desde cualquier lenguaje de programación. Cubre varios aspectos relacionados con la seguridad como el soporte de conexiones SSL o el cifrado de datos entre los brokers y los clientes. Se ejecuta en máquina virtual de Java (JVM).

Los componentes principales de la arquitectura de un clúster de Kafka, que vemos en la Figura 10, son los siguientes:

- Broker: A cada uno de los nodos que sirven de intermediarios entre los productores y los consumidores se les llama Brokers. Cada Broker contiene una o varias particiones (explicaremos este concepto más adelante) de uno o varios 'topics'. Están en comunicación constante con el nodo de Zookeper. Cada Broker puede estar instalado sobre una máquina con diferentes características de hardware en sí, por lo que puede haber nodos más eficientes que otros.
- Productor: Al cliente de Kafka encargado de publicar mensajes se le denomina productor. Cada mensaje que genera debe estar asignador a uno de los 'topics' existentes. Diferentes productores pueden estar escribiendo sobre diferentes particiones del mismo 'topic'.
- Consumidor: Al cliente de Kafka encargado de consumir mensajes se le denomina consumidor. Un consumidor puede estar suscrito a muchos 'topics' y muchos consumidores pueden estar suscritos a un mismo 'topic'.
- ZooKeeper: Como hemos comentados anteriormente, Apache Kafka es un software que se emplea para coordinar aplicaciones distribuidas. El lanzamiento de un clúster de Kafka siempre debe de estar acompañado de Zookeeper (al menos hasta hace unos meses [19]). Es el encargado de seleccionar entre los brokers donde se guarda la partición líder de un 'topic' y de decidir cuál será esa partición líder.

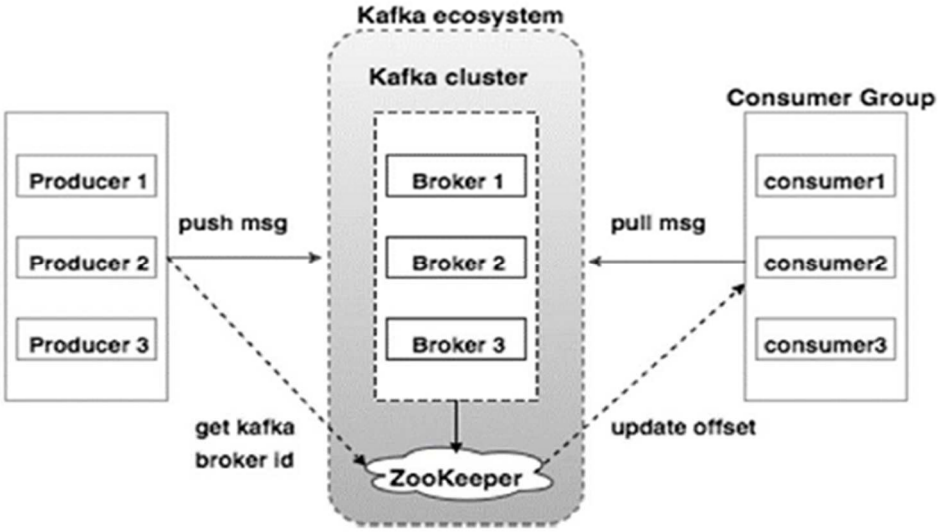


Figura 10 - Arquitectura Apache Kafka (Fuente: https://www.tutorialspoint.com/apache_kafka/apache_kafka_cluster_architecture.htm)

Se le llama partición de un 'topic' a cada una de las divisiones en las que este está fragmentado. Cada partición está replicada en uno o más nodos con el objetivo de aumentar la tolerancia a fallos y la disponibilidad. Zookeeper se encarga de asignar líder a una de las particiones y el resto de las réplicas serán las seguidoras y se encargarán de estar sincronizadas con esta. Cuando se cae un broker que contiene una partición líder, una de las otras particiones es elegida para sustituirla. El factor de replicación de un 'topic' es la cantidad de brokers en los que va a estar replicado.

En la Figura 11 se ve un clúster de Kafka que gestiona tres 'topics': Topic_0 tiene un factor de replicación de 3 y su partición líder está almacenad en Broker_0, Topic_1 tiene un factor de replicación de 2 y su partición líder está almacenad en Broker_1 y Topic_2 tiene un factor de replicación de 4 y su partición líder está almacenad en Broker_X-1.

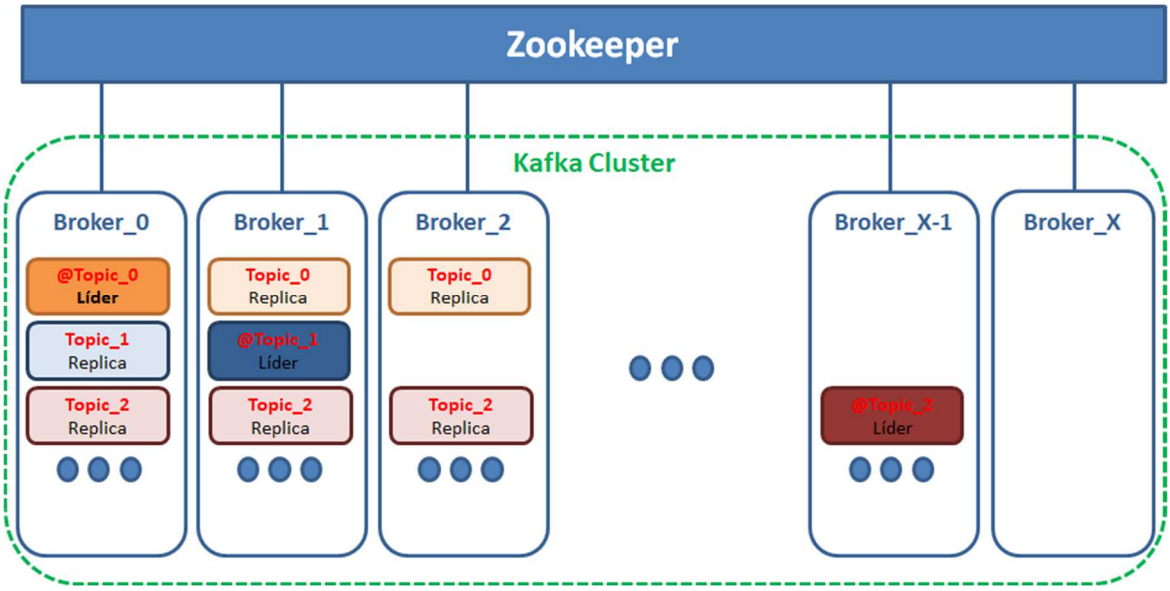


Figura 11 - Ejemplo replicación de bloques de Kafka (Fuente: https://enmilocalfunciona.io/aprendiendo-apache-kafka-parte-2-2/)

1.1.2.7 Apache Spark

Apache Spark es un motor de procesamiento que sustituye al motor clásico de Hadoop MapReduce. Ha obtenido una gran popularidad principalmente porque, al trabajar en memoria, consigue un rendimiento hasta cien veces superior que su antecesor. Es posible utilizar Spark desde Python, Scala, Java y R.

Para operar con los datos a través de Spark se emplean un tipo de estructuras llamadas RDDs (Resilent Distributed Dataset), que son una abstracción que nos permite aperar con un conjunto de datos inmutable y distribuido en memoria. Las evaluaciones sobre los RDDs se realizan de manera 'perezosa', es decir, que todas las transformaciones a las que sometamos los datos se encolan y no se ejecutan hasta que no es necesario. Esto presenta la ventaja de que, al no estar realizando continuamente transformaciones, se gana velocidad. El lado negativo es que, si hay algún error en las operaciones, no se detectará hasta que se realicen las operaciones, con lo que se dificulta el proceso de debuggeo.

Apache Spark cuenta con cinco componentes principales (Figura 12):

- Spark Core: Es el núcleo de Spark en donde se encuentra las librerías que utilizan el resto de los módulos.
- Spark SQL: Permite realizar consultas SQL y manejar tablas para el tratamiento de datos estructurados y no-estructurados.
- Spark Streaming: Permite la ingesta y tratamiento de datos en tiempo real.

- Spark MLLib: Es la librería que contiene algoritmos de Machine Learning. Contiene algoritmos de clasificación, de regresión, de árboles de decisión, de sistemas de recomendación y de clustering entre otros.
- Spark Graph: Es la herramienta de Spark para procesamiento de grafos (DAG).

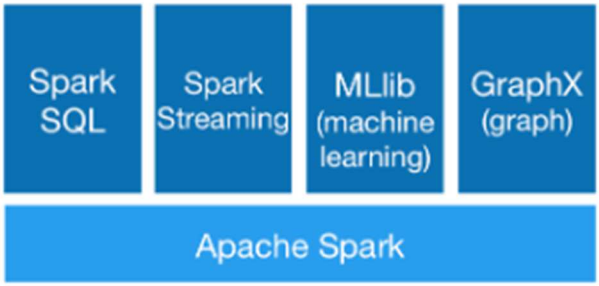


Figura 12 - Componentes de Apache Spark (Fuente: http://spark.apache.org/)

El punto de entrada en el código a todas las funcionalidades anteriores es el SparkContext. Este elemento nos permite realizar tareas como crear un RDD a partir de un fichero, crear acumuladores o variables de broadcast.

Existen dos tipos de operaciones que se pueden hacer sobre un RDD:

- Transformaciones: Una transformación es una operación que genera un nuevo RDD a partir de uno o más ya existentes. Estas operaciones se ejecutan siempre de forma perezosa, no se ponen en marcha hasta que se realiza una operación de tipo acción. Podemos distinguir dos tipos de transformaciones:
  - Narrow: En este tipo de transformaciones todos los datos necesarios para calcular los registros de una partición del RDD están en esa misma partición, con lo que no se genera tráfico de red entre los nodos. Dentro de este tipo encontramos las operaciones Map (que ejecuta un código sobre un registro y genera otro registro), FlatMap (que ejecuta también un código sobre un registro, pero generando 0 o más registros), MapPartition, Filter, Sample y Union.
  - Wide: En este tipo de transformaciones los elementos necesarios para calcular los registros de una partición están distribuidos por todas las particiones. Esto hace que este tipo de operación genere tráfico de red entre los nodos cuando se ejecuta. Las operaciones de este tipo que podemos encontrar son Intersection, Distinct, ReduceByKey, GroupByKey, Join, Cartesian, Repartition y Coalesce.
- Acciones: Una acción es una operación que genera un resultado, y por tanto obliga a que todas las transformaciones que se han acumulado de forma perezosa se ejecuten. Esto quiere decir que no generan otro RDD como en el caso anterior, sino que generan otro tipo de dato. Las acciones que podemos utilizar en Spark son Count (que devuelve el número de registros de un RDD), Collect (que simplemente hace que se ejecuten las transformaciones y devuelve el resultado de estas), Take (que devuelve un número de elementos del RDD), Top (que devuelve los valores del topo del RDD), CountByValue (que devuelve, para cada valor diferente del RDD, un conteo), Reduce (que realiza una misma operación con todos los elementos del RDD para devolver un solo valor), Fold, Aggregate y Foreach.

A continuación, nos vamos a centrar en el componente Spark Streaming. Spark Streaming es el componente de Spark que permite a los desarrolladores elaborar programas para procesar Big Data en tiempo real en los que los datos de entrada no son un conjunto de datos inmutable, sino que es un flujo de datos constante. El procesamiento no exactamente en tiempo real, sino que, como vemos en la Figura 13, va agrupando los datos que llegan en pequeños grupos llamados microbatches (otros motores como Apache Storm y Flink evalúan los datos elemento a elemento). Es totalmente compatible con YARN [20].

Figura 13 - Flujo de datos en Spark Streaming (Fuente: https://spark.apache.org/docs/latest/streaming-programming-guide.html)

La forma de operar sobre los streams de datos en Spark Streaming es utilizando un tipo de estructuras llamadas DStream (Discretized Stream) que son una abstracción de estos. Un DStream puede recibir datos de varias fuentes (Kafka, Flume, Kinesis o conexiones TCP) al mismo tiempo. Podemos considerar un DStream como un flujo constante de RDDs, donde cada RDD corresponde a los datos ingeridos en un intervalo de tiempo concreto. Todas las operaciones que se realizan sobre un DStream se aplican a todos los RDDs subyacentes.

Los DStreams soportan las mismas operaciones que los RDDs añadiendo unas cuantas novedades. Para empezar, tenemos dos tipos de operaciones: las operaciones stateless y las operaciones stateful. Las operaciones stateless se realizan sobre todos los microbatches del flujo individualmente sin tener en cuenta los demás, mientras que las operaciones stateful se pueden ejecutar sobre varios microbatches a la vez. Dentro de las operaciones stateful tenemos la de tipo ventana (Figura 14), que actúan sobre los datos que han llegado en una ventana de tiempo periódicamente. Con las operaciones de tipo ventana podemos conseguir, por ejemplo, que nuestro código evalúe cada cinco minutos los datos que han llegado en las últimas dos horas.

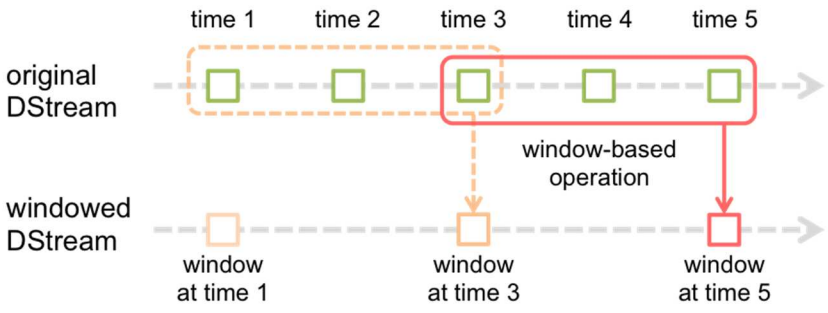


Figura 14 - Procesamiento por ventanas en Spark Streaming (Fuente: https://spark.apache.org/docs/latest/streaming-programming-guide.html)

1.2 Twitter

Twitter es una red social fundada en marzo de 2006 que permite a sus usuarios compartir información y opiniones de forma rápida y gratuita. Es una de las 15 redes sociales más utilizadas del planeta en la actualidad con más de 340 millones de usuarios activos [21]. Actualmente Twitter factura más de 2.500 millones de dólares al año y tiene un valor en bolsa superior a los 10.000 millones de dólares. No solo es utilizada por cientos de millos de usuarios anónimos, sino que también es utilizados por la mayoría de las grandes empresas del mundo, políticos y hasta presidentes del gobierno.

Antes de hablar de Twitter como fuente de datos para aplicaciones Big Data es necesario dejar clara la terminología propia de este:

- Tweet: Es el elemento básico de Twitter. Se trata de un mensaje de texto de 280 caracteres como máximo (antes de 2018 el máximo eran 140) que publica uno de los usuarios. Estos mensajes pueden ser visualizados tanto por otros usuarios de Twitter como por personas que no estén registradas en la red. Un tweet puede tener contenido multimedia como video o imágenes, puede mencionar a otros usuarios de la red y puede contener lo que se conoce como hashtags. Además de esto un tweet contiene una serie de metadatos como la fecha de publicación, la localización desde la que se publicó, cantidad de 'me gusta' obtenidos, información sobre la cuenta del creador, etc.
- Hashtag: Son palabra clave que vienen precedidas por el carácter '#' que se utilizan para hacer referencia al contenido del tweet. Por ejemplo, si estuviéramos escribiendo un tweet en el que se habla sobre el Covid 19 podríamos incluir el hashtag '#COVID_19'. Es un elemento muy utilizado por las empresas a la hora de promocionar servicios y productos y también por los partidos políticos como canal para promocionar sus campañas. Un usuario puede utilizar el buscador de Twitter para visualizar tweets relacionados con un hashtag concreto.
- ReTweet: Un ReTweet (RT) es publicar un tweet realizado por otro usuario desde tu cuenta. Se utiliza cuando se quiere los seguidores de tu cuenta visualicen también ese tweet, aunque no sigan a la cuenta que lo ha publicado originalmente. Una forma de medir la relevancia y la difusión de un tweet es a través de la cantidad de retweets que tiene.
- Respuesta: Tiene las mismas características que un tweet normal con la diferencia de que se publica como respuesta a otro tweet (propio o de otro usuario) y está necesariamente ligado a este otro tweet.
- Trending Topic: Es una lista de los hashtags y palabras más mencionados del momento en Twitter en todo el mundo o en un país concreto. No se mide únicamente la cantidad de menciones, sino que también se tiene en cuenta la novedad, de esta forma se evita que un personaje famoso con una gran cantidad de seguidores esté constantemente en la lista.

La sencillez y lo bien estructurada que es la información generada por los usuarios de Twitter (únicamente mensajes de texto con contenido multimedia opcional) y la cantidad de estos la convierte en una red social perfecta para servir como fuente de datos en proyectos de Social Big Data. Podemos encontrar otras redes sociales, como Facebook, que ofrecen una variedad de contenido mucho más alta (muros, grupos, eventos, etc), lo que hace más complicado su uso para este tipo de aplicaciones. También existen otras redes sociales, como

Instagram que, a pesar de contar con una estructura para su información mucho más sencilla, se centra en un tipo de contenido mucho más complicado de procesar (como son las imágenes).

1.2.1 API de Twitter

Actualmente muchas redes sociales cuentan con una API que permite a los desarrolladores que sus aplicaciones accedan a diferentes servicios de la red a través de código. Algunas redes sociales como Facebook o Instagram ofrecen unas APIs que están muy orientadas a automatizar tareas como puede ser publicar una imagen a una determinada hora desde una cuenta concreta o moderar los comentarios. La API de Twitter, sin embargo, está más orientada a permitir a los desarrolladores utilizar la red como fuente de datos para estos puedan ser procesados.

En un principio la API de Twitter presentaba tres servicios principales:

- REST API: Ofrece acceso a las operaciones que normalmente se llevarían a cabo a través de la web. Permite realizar acciones como acceder a la información de un usuario concreto, listar sus seguidores, listar todos sus tweets, etc. Los datos accedidos en este servicio no tienen un límite de antigüedad.
- Search API: Se utiliza para realizar búsquedas de tweets en función de una query determinada. Pueden filtrarse los tweets obtenidos en función de la localización desde donde se han publicado y el lenguaje. La capa gratuita permite buscar tweets publicados únicamente publicados en los últimos siete días. La capa 'premium' y la capa 'entrerprise' ofrecen dos endpoints, uno donde se permite realizar búsquedas de tweets publicados en los últimos 30 días y otro donde se permite realizar búsquedas de tweets publicados desde que se creó Twitter en 2006.
- Streaming API: Permite recibir un flujo de tweets en tiempo real a partir de unas normas de filtrado (por ejemplo, que los tweets incluyan una serie de hashtags o que tengan una localización determinada). También ofrece un flujo de datos de ejemplo aleatorio.

Para tener acceso a la API de Twitter es necesario crear primero una cuenta de desarrollador, para lo cual deberemos especificar qué tipo de usuario somos (estudiante, programador independiente, negocio, etc), qué uso vamos a dar a la API y deberemos aceptar las condiciones de uso. En el momento en el que hayamos creado la cuenta se nos proporcionará nuestra API Key, llamada en Twitter 'Consumer Key', que identifica unívocamente nuestro usuario, y la 'Consumer secret', que es la contraseña que usaremos para acceder a la cuenta. También es necesario crear una aplicación en nuestra cuenta de Twitter (darle un nombre, una descripción y especificar nuestra página web) para obtener un 'Access Token' y un 'Access Token Secret'.

Este mismo año los desarrolladores de Twitter presentaron la API de Twitter v2 construida sobre una base completamente nueva (no había sido reconstruida desde 2012) [22]. Está nueva versión presenta una API más limpia y mucho más sencilla de usar con características nuevas como especificar los campos de retorno para una búsqueda o recuperar varios tweets dentro de un mismo hilo. También permite el acceso a funcionalidades de Twitter que no existían hace unos pocos años como las encuestas, los tweets fijados en los muros de los perfiles o los filtros de spam.

Otra nueva característica añadida en esta versión es que, además de estar separada en tres capas, que son la capa estándar, la capa premium y la capa empresarial, se hace una distinción entre tres niveles de acceso que podemos ver en la Figura 15:

- Estándar: El nivel de acceso para la mayoría de los desarrolladores, incluidos los que están construyendo una aplicación por diversión o para aprender a usarla.
- Investigación académica: Un nivel de acceso más elevado que el anterior en el que entrarían investigares cualificados de diferentes universidades que estén desarrollando un proyecto de interés público.
- Negocios: El nivel de acceso para desarrolladores de una empresa, ya sea una empresa socia de Twitter o un cliente.

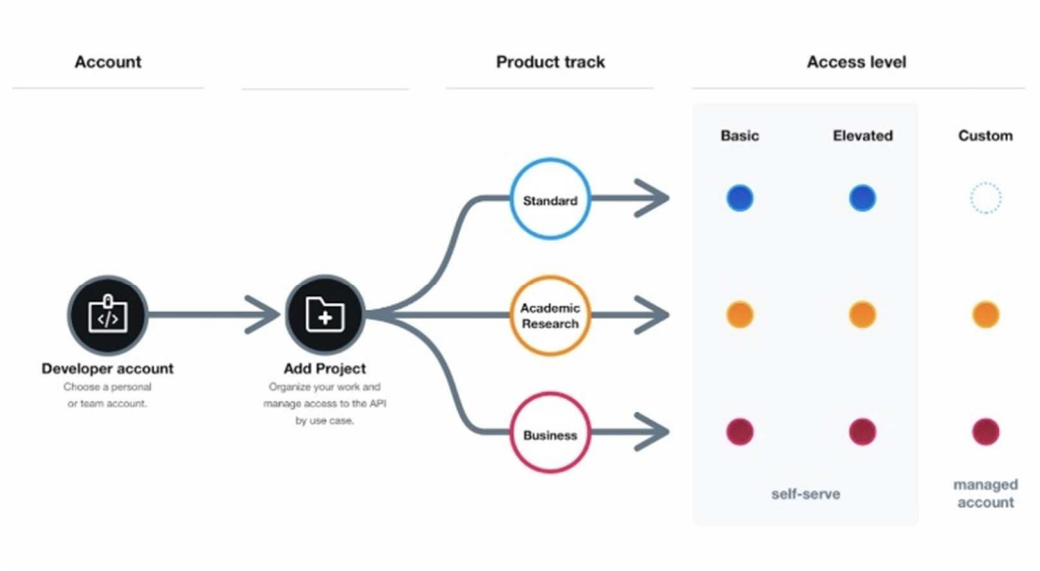


Figura 15 - Niveles de acceso en la API de Twitter (Fuente: https://blog.twitter.com/es_la/topics/product/2020/Presentamos-una-nueva-y-mejorada-API-de-Twitter.html)

Además de todas estas características se incluye una muy útil para nuestro caso de uso, que es la creación de un endpoint de streaming sobre el Covid-19. Este canal es totalmente gratuito y o presenta ninguna limitación en cuanto cantidad de tweets mensuales ni por segundo.

1.3 Cloud Computing

La computación en la nube (cloud computing) es un término que se usa para referirse a el procesamiento y almacenamiento de gran cantidad de datos a través de la red. Es una forma barata de almacenar los datos y aplicaciones de una empresa y acceder a ellos sin necesidad de montar una infraestructura física en el edificio y mantenerla.

Es un mercado que actualmente está creciendo a un ritmo exponencial (superior al 50% cada año) y, como es lógico, han surgido una gran cantidad de empresas dispuestas a aprovecharse de este auge, entre ellas algunas de las empresas más grandes del mundo. Entre estas podemos encontrar a IBM, Oracle o Alibaba Cloud (Alibaba Group) [23]. Google, que desde su nacimiento ofrece servicios íntegramente en la nube, cuenta también con su plataforma de Cloud Computing llamada Google Cloud. Microsoft, que ha tenido que reinventarse en los últimos años, ha renovado sus viejos productos como Windows y Microsoft Word a la nube

creando productos como Microsoft 365 y Office 365. Como era de esperar Microsoft también ha lanzado su servicio de cloud computing llamado Microsoft Azure. La última empresa que comentaremos que haya lanzado su propio servicio de computación en la nube es Amazon, con su famoso Amazon Web Services (AWS), pero esto será más adelante.

En función de quién sea el dueño de la infraestructura de una nube quiénes sean los usuarios, podemos encontrarnos varios tipos de nube [24]:

- Nube pública: Una nube pública es el tipo más común de computación en la nube. Todos los recursos de esta nube (servidores, almacenamiento, etc) son propiedad de la empresa que provee el servicio y es ella la que se compromete a su administración y mantenimiento. Los usuarios que contratan el servicio de una nube pública comparten los recursos de esta con otros usuarios, lo que puede provocar subidas y bajas de rendimiento aleatorias. Las ventajas que ofrece frente a otros tipos de nube son el precio, la flexibilidad (las empresas proveedoras de este tipo de nube suelen ofrecer una gran cantidad de tipos de productos y capacidad para escalar horizontalmente los recursos contratados) y el poder delegar el mantenimiento de la infraestructura a otra empresa.
- Nube privada: Una nube privada es aquella en la que todos los recursos son utilizados exclusivamente por una empresa. Puede estar localizada físicamente en las instalaciones de la propia empresa o en las de la empresa que provee el servicio. Dado que todo el hardware se dedica a una sola empresa, se evitan fluctuaciones en el rendimiento debido a la actividad de otras empresas. El tipo de empresas que suelen hacer uso de este tipo de nube son grandes empresas o incluso entidades gubernamentales, dado que se puede personalizar de forma más precisa una nube privada que una pública. La seguridad también se ve aumentada ya que permite a la empresa establecer un mayor control sobre sus recursos.
- Nube híbrida: Una nube privada ofrece las ventajas de una privada y una pública. Permite que los recursos se muevan entre nubes públicas y privadas permitiendo así una mayor flexibilidad. Por ejemplo, es posible que una empresa almacene un tipo de archivos que no requieran ninguna seguridad en una nube pública y los documentos confidenciales en una privada.
- Nube comunitaria: Una nube privada es similar una nube privada, pero en vez de pertenecer a una solo empresa, es un grupo de empresas quien posee y hace uso de esta. Ofrece la ventaja de que los costos se comparten entre las empresas, sin embargo, es posible que surjan conflictos y desacuerdo entre estas que afecten a la rentabilidad de la nube a largo plazo.

También pueden establecerse tres grupos en fusión de cuál sea el tipo de servicio en línea que se ofrezca [25]:

- Software-as-a-Service (SaaS): Consiste en ofrecer como servicio una aplición en línea al usuario. Ejemplos de esto pueden ser un servicio de correo electrónico, una plataforma de streaming o el Office 365 que se ha mencionado antes. Los usuarios acceden a la aplicación sin preocuparse de cuál es la infraestructura ni el software que lo soportan. La responsabilidad de administración y mantenimiento del servicio es únicamente del proveedor. Los usuarios finales tienen un control mínimo sobre el servicio, colocándose en la capa más superficial de este.

- Platform-as-a-Service (PaaS): Ofrece a los desarrolladores de código la posibilidad de centrarse únicamente el código de la aplicación sin preocuparse de la infraestructura. La responsabilidad de mantener la infraestructura y gestionar aspectos como la escalabilidad es del proveedor de la plataforma. Sin embargo, los desarrolladores se hacen responsable de la optimización del código para evitar que esta consuma demasiados recursos, ya que una de las formas de pago de estos servicios es por uso de recursos. Ejemplos de PaaS son Jelastic y Google Collab.
- Infraestructure-as-a-Service (IaaS): Ofrece a los desarrolladores la posibilidad de ocuparse también de la infraestructura de la aplicación. Ofrece un control sobre los recursos mucho mayor que en los casos anteriores, pero también requiere un mayor esfuerzo. Son los desarrolladores los que se ocupan de aspectos como el sistema operativo sobre el que correrá el código o las redes y subredes virtuales en la que estarán los recursos, así como la comunicación entre estos. Ejemplos de IaaS son Microsoft Azure o Amazon Web Services.

En la Figura 16 se muestran los aspectos sobre los que tienen que hacerse cargo el usuario y el proveedor de servicio en función de si se trata de un SaaS, un PaaS, un IaaS o un equipo comprado por y para el servicio (On-Premises).

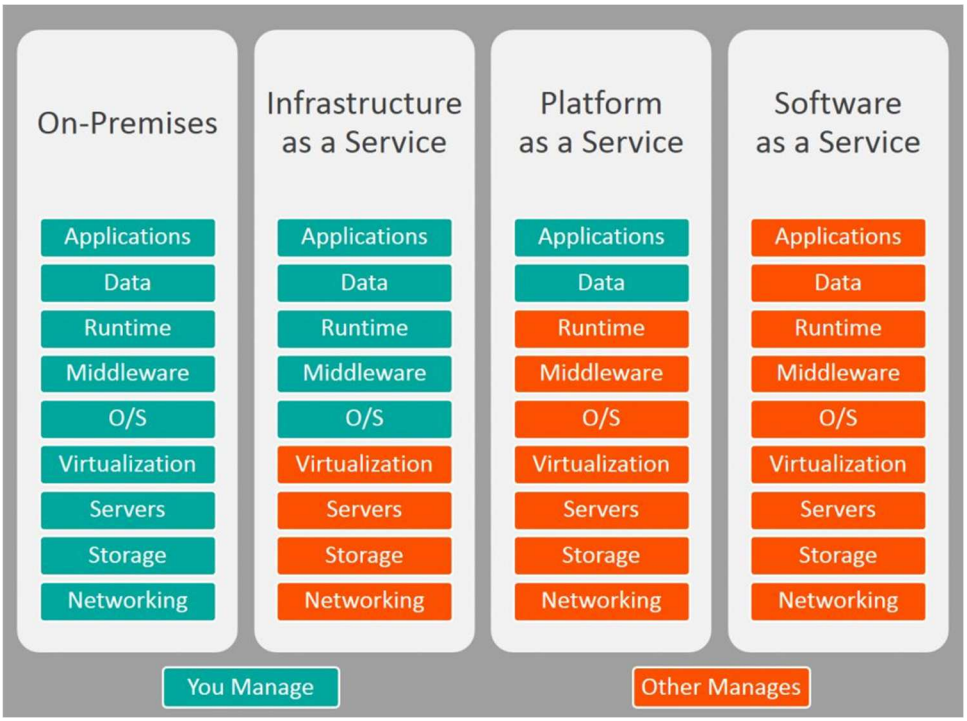


Figura 16 - Responsabilidades del usuario en servicios SaaS, PaaS, IaaS y On-Premises

Capítulo 2 - Diseño y funcionalidades de la aplicación

En este apartado se van a definir las funcionalidades que tendrá la aplicación y cómo será la interacción de los usuarios con esta. También se hará un prototipo visual de la web de la aplicación en la que se podrán ver todos sus componentes y la disposición estimada de estos en la pantalla.

La primera funcionalidad será la de mostrar un mapa mundial con el que el usuario podrá interactuar. La primera opción que se nos vino a la cabeza fue la de utilizar los mapas de Google Maps, ya que cuenta con una API para JavaScript. El problema es que la licencia de esta API es de pago y si queremos utilizar la versión gratuita para desarrolladores el mapa se ve ensombrecido por una marca de agua que es muy difícil de ignorar. Por esto decidimos utilizar el proveedor de mapas en línea Mapbox que cuenta con una capa gratuita bastante generosa y es más que suficiente para nuestras necesidades.

No solo será necesario que se muestre un mapa mundial interactivo, sino que además los países del mapa se deberán colorear en función del conteo de tweets sobre el COVID-19 de cada país. Para que el usuario pueda saber el significado de los colores del mapa será necesaria, evidentemente, una leyenda con los rangos de tweets de cada color.

También mostraremos una tabla a la derecha del mapa que muestre el top-10 de los hashtags que más veces han aparecido en los tweets evaluados. Esta tabla tendrá tres columnas: la primera con la posición del hashtag en el top, la segunda con el texto del hashtag y la tercera con el conteo de apariciones de este.

No solo queremos mostrar el estado del mapa y de la tabla en un momento concreto, sino que queremos que el usuario pueda ver su evolución a lo largo de las últimas veinticuatro horas. Como veremos más adelante, la parte de la aplicación que se ocupa de la recogida y el tratamiento de datos archivará cada diez minutos en un repositorio un informe con el conteo de tweets y hashtags por país. Para permitir al usuario cambiar la hora de los datos que se visualizan en el mapa se utilizará un slider (una barra deslizadora). La cantidad de valores que podrá tomar el slider será de 144 ((24 h * 60 min / h) / 10 min). Cada vez que el usuario deslice la barra los valores de la tabla, los colores del mapa y los valores de la leyenda deberá actualizarse para correspondes a la hora señalada. En caso de que, por un fallo, los datos esa hora no estén disponibles, el mapa se deberá quedar en blanco y la tabla sin valores.

Cuando el usuario clique sobre un país del mapa coloreado se le mostrará un pop-up con información sobre el país y el contenido de la tabla pasará de mostrar el conteo de hashtags mundiales a mostrar el conteo de hashtags en ese país en concreto en la última hora.

Podemos ver un prototipo visual de la web en la Figura 17. Para la creación del prototipo se ha utilizado la plataforma web de creación de prototipos de aplicaciones Proto.io [26].

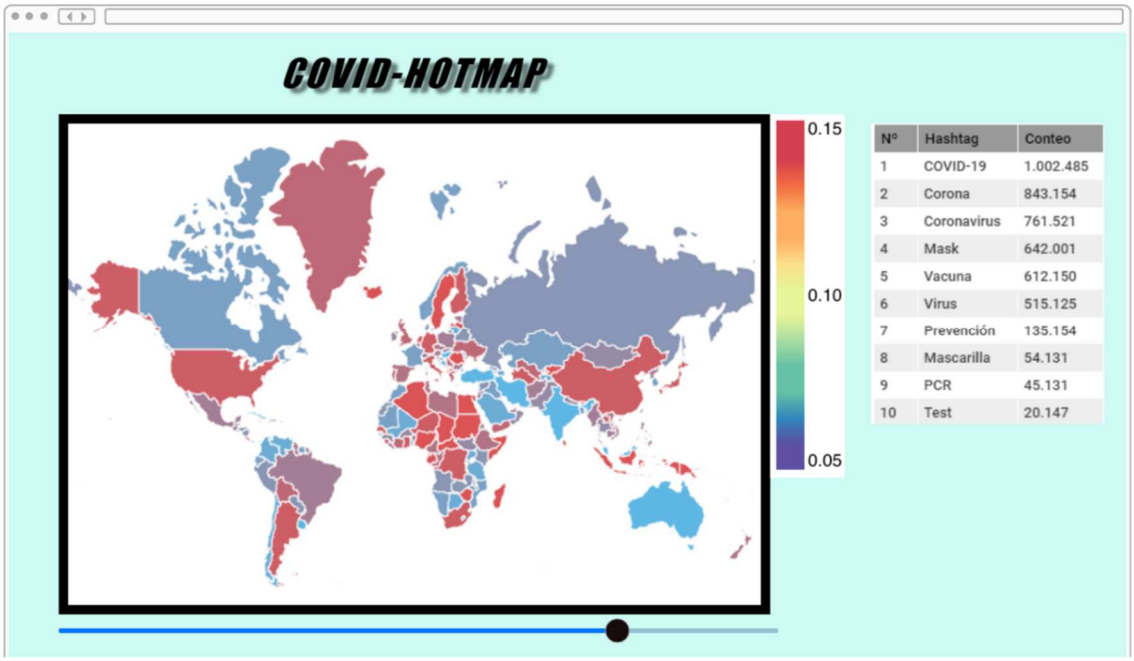


Figura 17 - Prototipo de la web de la aplicación

La recolección de los tweets y su procesamiento para conseguir los resultados que usará la web serán transparentes para el usuario y se realizará de forma independiente. Como explicaremos en el siguiente capítulo, esto se llevará a cabo utilizando servicios de AWS como EC2 y EMR y herramientas como la API de Twitter y Apache Spark.

Capítulo 3- Arquitectura de la aplicación

En este apartado nos disponemos a definir la estructura de la aplicación, así como cada uno de sus componentes y explicar la función y las características de estos. Como la totalidad de la aplicación del proyecto se va a desplegar sobre AWS, se va a intentar maximizar el uso de los beneficios de la plataforma, como son la alta disponibilidad, la tolerancia a fallos y la elasticidad.

Para realizar el diagrama de la arquitectura vamos a utilizar el software CloudCraft. CloudCraft es una herramienta online que permite modelar los componentes de una arquitectura montada sobre AWS y las relaciones entre estos al tiempo que se visualiza un diagrama de la aplicación y se calcula automáticamente su precio mensual. CloudCraft ofrece tres planes de pago con diferentes características: el gratuito, el profesional (que cuesta 50$/mes) y el empresarial. Nosotros vamos a utilizar una cuenta gratuita desde la que podemos acceder a todos los componentes del constructor de diagramas, pero el tamaño del grid sobre el que lo construiremos será limitado, sin embargo, será suficiente.

Todos los elementos de la aplicación estarán dentro de una red virtual que se creará dentro de AWS utilizando una VPC (a excepción del Bucket S3, que es independiente de las VPCs). Esta red estará dividida en tres subredes con diferentes propósitos: una subred pública que dará acceso a los usuarios desde internet y a nosotros mismos a través de una máquina bastión, una subred privada donde estarán las instancias de EC2 que servirán la página web y otra subred privada donde se realizarán las tares de extracción de datos y su tratamiento.

Para proporcionar a las instancias de las subredes privadas acceso a internet (ya que es necesario instalar algunos programas y tener acceso a la API de Twitter) utilizaremos una NAT Gateway que nos permitirá darles este acceso sin exponer sus direcciones a internet. Este componente tiene un precio de 0,045$/hora (alrededor de 30$/mes) aunque no se esté haciendo uso de él, por lo que, durante el desarrollo, crearemos uno nuevo cada vez que lo necesitemos y lo borraremos cuando ya no nos haga falta (recordemos que con AWS Educate tenemos un saldo inicial de 100$ que no se renueva).

Aunque el Bucket de S3 no esté dentro de la subred de datos (porque estos no están ligados a VPCs, sino a regiones y zonas de disponibilidad), podemos considerarlo parte de esta por sus funciones.

Como vemos en la Figura 18 - Arquitectura de la aplicación, las subredes están divididas en dos zonas de disponibilidad para aumentar la disponibilidad y la tolerancia a fallos. Esto querrá decir que, en la práctica, no tendremos realmente tres subredes sino seis, ya que cada una deberá estar replicada en las dos zonas de disponibilidad.

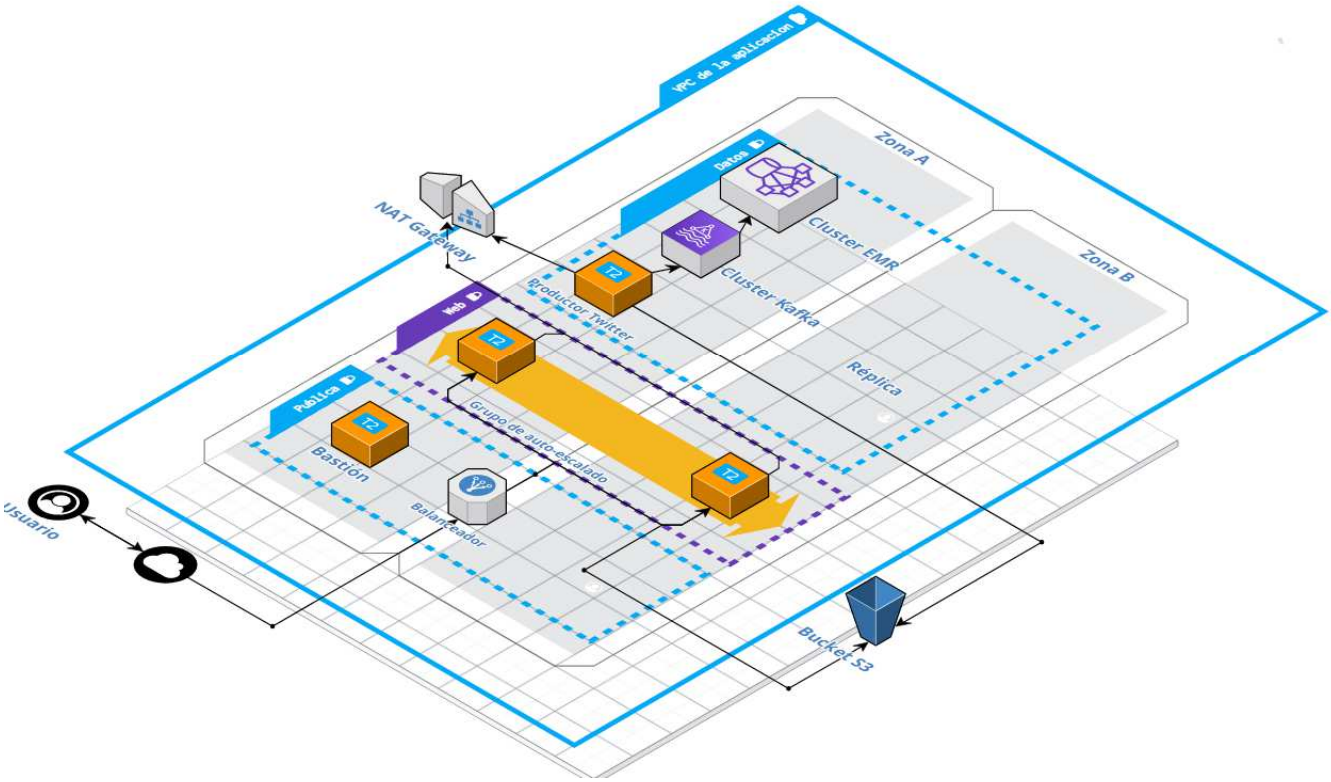


Figura 18 - Arquitectura de la aplicación

3.1 Subred Pública

La subred pública tendrá dos funciones principales:

- La primera y más importante es dar acceso a la aplicación a los usuarios desde internet. Como veremos a continuación, la subred de web utiliza un grupo de auto-escalado, lo que quiere decir que tendremos varias instancias de EC2 con diferentes direcciones IP. Para solucionar este problema utilizaremos un balanceador de carga que nos proporciona un punto de acceso único para todas las instancias del grupo de auto-escalado y un dominio DNS. Además de esto, como su propio nombre indica, el balanceador de carga se encargará de distribuir las peticiones HTTP entrantes entre todas las instancias de una forma equilibrada para que todas tengan la misma carga de trabajo. Al crearse, el balanceador se levantará en las dos zonas de disponibilidad. Al igual que con la NAT Gateway, durante el desarrollo, el balanceador se eliminará cuando no se esté utilizando ya que conlleva unos gastos mensuales de alrededor de 20$ y el saldo de la cuenta de AWS Educate es limitado.
- La segunda función es albergar una instancia de EC2 que llamaremos bastión y que utilizaremos para tener acceso al resto de instancias de la VPC de forma segura. El resto de las instancias tendrán una regla en sus Security Groups que únicamente permitirá el acceso mediante SSH desde la máquina bastión, y está máquina únicamente se encenderá cuando se quiera realizar algún tipo de mantenimiento y el resto del tiempo estará apagada, con lo que se conseguirá un nivel de seguridad bastante elevado. Además, para acceder a la máquina bastión por SSH será necesario tener el Key Pair.

3.2 Subred de Web

Esta subred privada tendrá alojado un grupo de auto-escalado de máquinas EC2. Como estas máquinas se lanzan y terminan automáticamente, sin la supervisión del administrador, deberán tener un bash script en los datos de usuario en el que se especifiquen los comando que se ejecutarán en cada máquina cuando se lancen. En este script se instalarán los programas necesarios, se descargará del Bucket S3 un proyecto de angular con la aplicación web y la ejecutará.

El acceso a las máquinas del grupo de auto-escalado por parte del usuario se llevará a cabo a través del balanceador de carga. Si la carga media de las máquinas del grupo supera el 80% se lanzará automáticamente otra máquina (hasta un máximo de 10 máquinas) para ayudar a reducir la carga media. Si la carga media de las máquinas del grupo desciende del 30% se terminará automáticamente una de las máquinas (hasta un mínimo de 2 máquinas) para reducir el coste. El grupo de auto-escalado no conlleva cargos adicionales, AWS únicamente nos cobra los gastos de cada una de las máquinas por lo que podremos reducir el número de instancias a 0 cuando no estemos desarrollo sin tener que eliminarlo.

Estas máquinas se conectarán a internet para descargar programas y dependencias a través de NAT Gateway de la subred pública. También deberán tener un rol de IAM que les dé acceso a S3.

3.3 Subred de Datos

En esta subred se encontrarán los componentes que harán las tareas de recolección de datos, preparación, tratamiento de estos y almacenaje de resultados. Estos componentes serán:

- Productor de datos: Se trata de una instancia de EC2 con acceso a internet a través de la NAT Gateway (ya que no necesitará acceso desde el exterior) que recolectará los tweets relacionados con el Covid-19, se quedará únicamente con los campos que necesitaremos para el análisis (es decir, que no solo se encargará de la recolección de datos, sino que también se ocupará de su preparación) y los enviará al siguiente componente, el clúster de Kafka. Para ello ejecutará un script escrito en python que recogerá del Bucket S3. El script utilizará la librería 'tweepy' y nuestras credenciales de Twitter para acceder a la API de Twitter y solicitar un stream continuo de Tweets. Después, hará uso de la librería de Kafka para python para enviar los datos preparados el clúster. También se encargará de recoger los resultados del procesamiento de clúster para guardarlos en el Bucket S3.
- Clúster Kafka: Es el componente que hará de intermediario entre el productor de Twitter y el clúster de EMR. Se trata de un intercambiador de mensajes que sigue el patrón publicador/suscriptor y que se integra con Spark Streaming para proporcionarle un flujo de datos constante. Las ventajas principales que obtendremos al utilizar Apache Kafka como middleware de intercambio de mensajes son:
    - Si en cualquier momento se cae el clúster EMR y estamos enviándole los datos directamente desde el productor, todos los tweets enviados desde que se cae hasta que se levanta de nuevo se pierden. Si utilizamos Kafka los mensajes se almacenan hasta que se vuelva a levantar y se entreguen.
    - Se puede levantar en varios nodos, cada uno en uno zona de disponibilidad, lo que nos aporta una tolerancia a fallos todavía mayor.

AWS cuenta con una herramienta llamada 'Amazon Managed Streaming for Apache Kafka' (Amazon MKS) que nos permite levantar clústers de Kafka de una manera sencilla y completamente administrada. El problema es que, al estar utilizando AWS Educate no tenemos una cuenta root, sino que tenemos acceso a un usuario de la cuenta con unos permisos de IAM limitados, y entre estos permisos no está incluido el uso de Amazon MSK. En vez de hacer uso de esa herramienta, levantaremos el clúster utilizando una instancia de EC2 e instalando en ella manualmente los programas y paquetes necesarios y configurando el clúster y lanzándolo también a mano (o mediante un bash script que se ejecute automáticamente).

- Clúster EMR: AWS cuenta con el servicio EMR que proporciona clústers de Hadoop elásticos, es decir, que el número de máquinas esclavas puede aumentar o disminuir en función de la carga igual que un grupo de auto-escalado (de hecho, utiliza grupos de auto-escalado internamente). Utilizaremos una de estos clústers para lanzar un trabajo de Spark Streaming que se suscriba al topic del clúster de Kafka al que el productor envía los tweets y se encargará de realizar el conteo de tres tipos de datos:
    - Contará cuantas veces aparece cada hashtag a nivel mundial.
    - Contará, por cada país, el número de tweets.
    - Contará, también por cada país, cuantas veces aparece cada hashtag.

  Cada diez minutos realizará el conteo de los datos anteriores para los tweets que han llegado en la última hora (utilizando las funciones de ventana de Spark Streaming). Estos resultados serán enviados otra vez al clúster de Kafka para que sean almacenados.
- Bucket S3: Este componente no se encontrará realmente dentro de la VPC ni de ninguna de las subredes. Los datos que almacenará serán:
    - El proyecto de Angular con la aplicación web que se servirá a los usuarios.
    - El script de python que ejecutará la instancia del productor de datos.
    - El programa de Spark Streaming que se ejecutará en el clúster EMR.
    - Los resultados obtenidos del análisis en el clúster EMR. Estos datos serán enviados por el clúster EMR al productor cada diez minutos y contendrá la información de los datos procesados recogidos en la última hora. Son los datos que serán recogidos por la web para ser visualizados en el mapa.

3.4 Estructura de los datos de entrada y de salida

La estructura de los datos recogidos de la API de Twitter se puede consultar directamente en la documentación de esta. En ella se explica como un tweet es un objeto definido en formato JSON con una serie de campos 'raiz' y cuatro hijos objetos más: User, Media, Poll y Place [27].

Entre los campos 'raiz' del tweet podemos encontrar los que vemos en la Tabla 1.

Nombre del campo	Descripción	Tipo de dato
Id	Identificador único del tweet. Se puede utilizar cuando queremos recuperar un tweet específico.	String
Text	El texto del tweet en formato UTF-8. Puede utilizarse para extracción de palabras clave y 'sentiment analysis'.	String
Author_id	Identificador único del autor del tweet.	String
Conversation_id	Identificador único de la conversación a la que pertenece el tweet. Se puede utilizar, por ejemplo, si	String

	se quieren recoger todas las respuestas de un tweet y las respuestas a esas respuestas, etc.	
Created_at	La fecha y hora de creación del tweet en formato ISO8601.	String
Entities	Contiene información sobre diferentes elementos que pueden formar parte del texto del tweet, pero que no son texto como tal, como son los hashtags, las menciones y los links.	Object
Geo	Contiene información sobre la localización etiquetada por el usuario en el tweet, si es que ha etiquetado alguna. Está en formato latitud/longitud.	Object
In_reply_to_user_id	El identificador único del usuario al que responde el tweet, si es que se trata de una respuesta.	String
Non_public_metrics	Métricas del tweet a las que solo podemos acceder si somos los dueños de este. Contiene la cantidad de veces que se ha accedido al perfil desde el tweet, la cantidad de impresiones generadas y los clics a los URLs del texto.	Object
Public_metrics	Métricas del tweet que son públicas, como la cantidad de retweets, la cantidad de likes o la cantidad de respuestas.	Object

Tabla 1 - Campos raíz del tweet

Entre los campos del objeto User encontramos los que vemos en la Tabla 2.

Nombre del campo	Descripción	Tipo de dato
Id	Identificador único del usuario.	String
Name	Nombre del usuario. No tiene por qué ser único y tiene un máximo de cincuenta caracteres.	String
Username	Alias del usuario. Es único en todo Twitter y tiene un máximo de quince caracteres.	String
Created_at	Fecha y hora de creación del usuario en formato ISO8601.	String
Description	Descripción del perfil del usuario.	String
Location	Localización del usuario, si es que este ha proporcionado una	Object
Pinned_tweet_id	Identificador del tweet anclado en el muro del perfil del usuario	String
Profile_image_url	Dirección URL de la imagen de perfil del usuario.	String
Public_metrics	Métricas públicas del usuario como su cantidad de seguidores, cantidad de cuentas a las que sigue o la cantidad de tweets publicados. Puede utilizarse para estimar la relevancia del autor de un tweet	Object

Tabla 2 - Campos del objeto User del tweet

En el objeto Media podemos encontrar información sobre las imágenes y videos del tweet como el peso de las imágenes o la duración de los videos. Si el tweet no tiene contenido multimedia este objeto contiene el valor 'null'.

El objeto Poll solamente se utiliza si el tweet contiene una encuesta. En él se guarda información como los votos de cada opción de la encuesta o la duración de esta.

Por último, tenemos el objeto Place que guarda la información sobre el lugar etiquetado por el creador del tweet. Sus campos pueden verse en la Tabla 3:

Nombre del campo	Descripción	Tipo de dato
Full_name	Versión larga del nombre completo del lugar. Por ejemplo, 'Manhattan, NY'.	String
Id	Identificador único del lugar.	String
Contained_within	Es posible que un lugar esté contenido dentro de otro. En este campo se guardan los identificadores de los lugares que incluyen este lugar concreto.	Array
Country	Nombre del país en el que se localiza el lugar.	String
Country_code	Código del país en el que se localiza el lugar en formato ISO Alpha-2. Por ejemplo, si el país es Estados Unidos, su código sería US.	String
Geo	Localización del lugar en formato GeoJSON.	GeoJSON
Name	Versión corta del nombre completo del lugar. Por ejemplo, 'Manhattan'.	String
Type	Tipo del lugar, por ejemplo, 'city',	String

Tabla 3 - Campos del objeto Place del tweet

De esta gran cantidad de campos, nos vamos a quedar con un subconjunto que será enviado al clúster EMR para ser analizado. El formato de este subconjunto será:

- Del campo entities, nos quedaremos únicamente con los hashtags.
- Del objeto Place, nos quedaremos con el campo country_code.

En la Figura 19 podemos ver un ejemplo de mensaje en formato JSON que se enviaría desde el productor de Twitter al clúster de Kafka, y de ahí al clúster EMR.

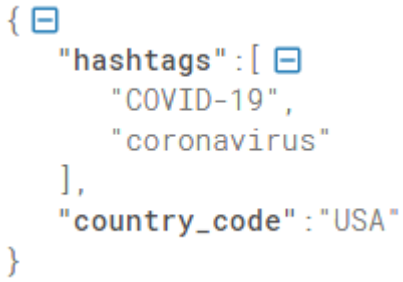


Figura 19 - Ejemplo de mensaje del productor

El formato de los mensajes con los resultados que enviará el clúster EMR de vuelta al producto será el siguiente:

- Si el mensaje representa la cantidad de tweets que se han contado para un país en concreto, el formato será un Array con dos posiciones. La primera posición será un String que comenzará por "cc " e irá seguido del código del país en formato ISO Alpha-2. La segunda posición será un número con el conteo total de tweets de ese país en la última hora.

- Si el mensaje representa la cantidad de veces que se ha contado un hashtag a nivel mundial, el formato será un Array con dos posiciones. La primera posición será un String que comenzará por “ht ” e irá seguido del hashtag. La segunda posición será un número con el conteo total de veces que se ha contado ese hashtag a nivel mundial en la última hora.
- Si el mensaje representa la cantidad de veces que se ha contado un hashtag para un determinado país, el formato será un Array con dos posiciones. La primera posición será un String que comenzará por “htcc ” e irá seguido del código del país en formato ISO Alpha-2 y del hashtag. La segunda posición será un número con el conteo total de veces que se ha contado ese hashtag en ese país en la última hora.

Para terminar, vamos a especificar el formato con el que el producto almacenará los resultados en el Bucket S3. Los resultados se guardarán cada diez minutos en un archivo en formato JSON que tendrá por título ‘HORA_MINUTO.json’, donde ‘HORA’ y ‘MINUTO’ corresponden al momento en el que se sube el archivo (para simplificar la tarea de la web, los minutos se contarán de diez en diez, es decir, que si se sube un archivo a las 10:43, se guardará como ‘10_40.json’).

Los campos que tendrá este JSON son:

- Country_count: Un Array en el que cada posición el un código de país en formato ISO Alpha-2 y el conteo de tweets para él. Estará ordenado de mayor a menor para facilitar las tareas de la web.
- Hashtag_world_count: Un Array de diez posiciones con el conteo de los diez hashtags que más veces se han contado a nivel mundial. Estará ordenado de mayor a menor para facilitar las tareas de la web.
- Hashtag_countries_count: Un diccionario en el que las claves son el código de un país en formato ISO Alpha-2 y el valor de cada uno es un Array de diez posiciones con el conteo de los diez hashtags que más veces se han contado para ese país. Los valores para cada país estarán ordenados de mayor a menor para facilitar las tareas de la web.

En la Figura 20 Podemos ver un ejemplo de los archivos que acabamos de describir. Se trata de un fichero creado a mano ya que los archivos reales que produce son demasiado grandes para mostrarse en una imagen de esta memoria. Además, algunos campos han sido replegados por esta misma razón.

```
{
  "Country_count":[
    [
      "GB",
      37
    ],
    [
      "US",
      22
    ],
    [
      "MY",
      18
    ],
    [ ],
    [ ]
  ],
  "Hashtag_world_count":[
    [
      "CovidVaccines",
      2
    ],
    [
      "IdiotPM",
      2
    ],
    [ ]
  ],
  "Hashtag_countries_count":{
    "GB":[ ],
    "JP":[ ],
    "IN":[
      [
        "COVID",
        1
      ],
      [
        "COVIDVaccination",
        1
      ],
```


Figura 20 - Ejemplo de archivo con los resultados del conteo

Capítulo 4 - Implementación

En este apartado se van a describir los procesos seguidos para implementar la arquitectura descrita en el apartado anterior y cada uno de sus pasos. Estos procesos serán: la obtención de una cuenta de AWS Educate, la obtención de la cuenta de desarrollador de Twitter para poder utilizar la API, la implementación de cada uno de los componentes de AWS, el desarrollo del código que ejecutará cada uno y la integración de todos los componentes.

4.1 Obtención de una cuenta AWS Educate

Como ya comentamos, AWS cuenta con una plataforma, llamada AWS Educate, orientada a proporcionar a estudiantes los medios necesarios para desarrollar habilidades relacionadas con la nube, más concretamente con AWS. La plataforma es utilizada en más de 200 países por 2400 instituciones, más de 10 000 docentes y cientos de miles de estudiantes.

Además de ofrecer herramientas como talleres online, seminarios en video y otros contenidos multimedia, proporciona los estudiantes una cuenta gratuita con un saldo de 100$ para practicar con algunos de los servicios más famosos de AWS (realmente no se trata de una cuenta como tal, sino que proporciona las claves de un usuario dentro de una cuenta con permisos de AWS IAM limitados).

Para obtener una cuenta lo primero que hicimos fue acceder a la web https://aws.amazon.com/es/education/awseducate/ y seleccionar la opción “Unirme a AWS Educate”. A continuación, nos pudieron que especificáramos que tipo de usuario somos: estudiante, educador, empresa, institución o veterano del ejército de EEUU, elegimos la primera opción. Después rellenamos un formulario con nuestro nombre, fecha de nacimiento, país, nuestro centro de estudio (existe una lista de centros de estudio que pueden acceder a AWS Educate entre los que se encuentra la UPNA) y nuestra dirección de e-mail del centro. Después de enviar el formulario y esperar un tiempo, recibimos una confirmación en nuestro correo y pudimos acceder a la cuenta.

A través de la cuenta de AWS Educate se puede acceder a la consola de administración de AWS. Esta consola nos proporciona un listado de todos los servicios de AWS (tengamos acceso a ellos o no) organizados por temática y un listado con un subconjunto de estos servicios visitados recientemente. En la Figura 21 podemos ver la consola de AWS después de haber accedido recientemente a algunos de sus servicios.

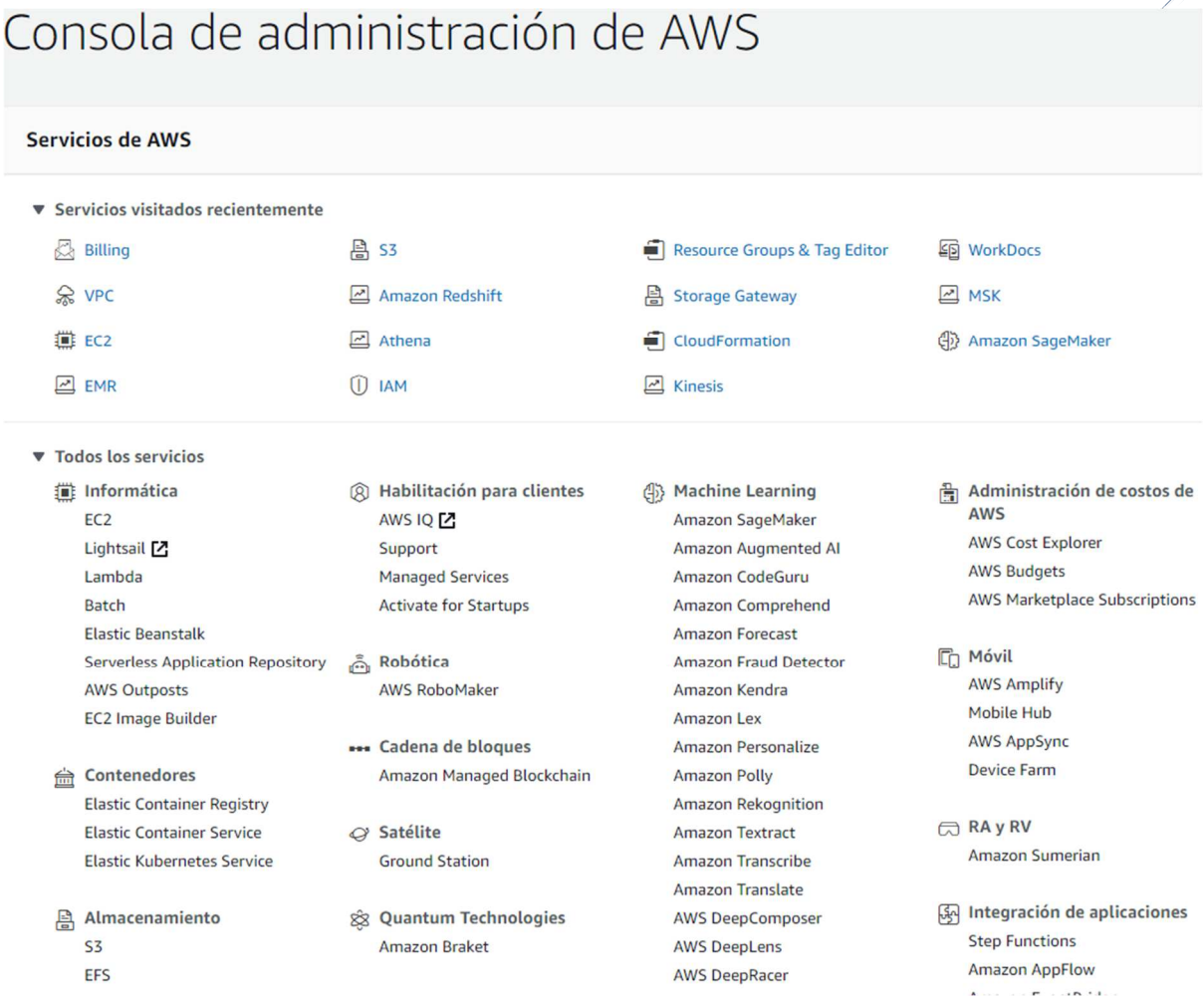


Figura 21 - Consola de AWS

4.2 Obtención de una cuenta de Twitter de desarrollador

Para convertir una cuenta de Twitter normal en una cuenta de desarrollador es necesario solicitarlo a través de la web developer.twitter.com/en/apply-for-access registrados con la cuenta que queramos transformar. En ella se nos pidió rellenar un formulario en el que dábamos información sobre qué tipo de usuario somos (profesional, estudiante, profesor, alguien que simplemente quiere explorar la API, etc), en nuestro caso estudiante, el nombre de nuestra entidad legal, en nuestro caso la Universidad Pública de Navarra, y si íbamos a utilizar la API con fines lucrativos, en nuestro caso no. Después se nos pidió que redactáramos un texto de, al menos, 200 caracteres en el que expliquemos qué usos vamos a dar a la API. Una vez terminado el formulario se nos pidió que aceptásemos los términos y condiciones de uso y después de esperar unos minutos recibimos un e-mail informándonos de que nuestra solicitud había sido aceptada.

Una vez creada la cuenta tenemos acceso al portal de desarrollador de Twitter en el que vemos un listado de los proyectos y las aplicaciones creadas. A la hora de crear una aplicación debemos darle un nombre único a nivel mundial, la nuestra se llamará COVIDHotMap. Después de crear la aplicación se nos proporcionan la API Key, la API Secret Key y el Bearer Token, que debemos guardar porque será la última vez que se nos mostrará. En la Figura 22 se puede ver

como Twitter muestra estos valores al usuario y le avisa de que, por seguridad, será la última vez que se muestren (por seguridad se ha tapado parte de estos valores).

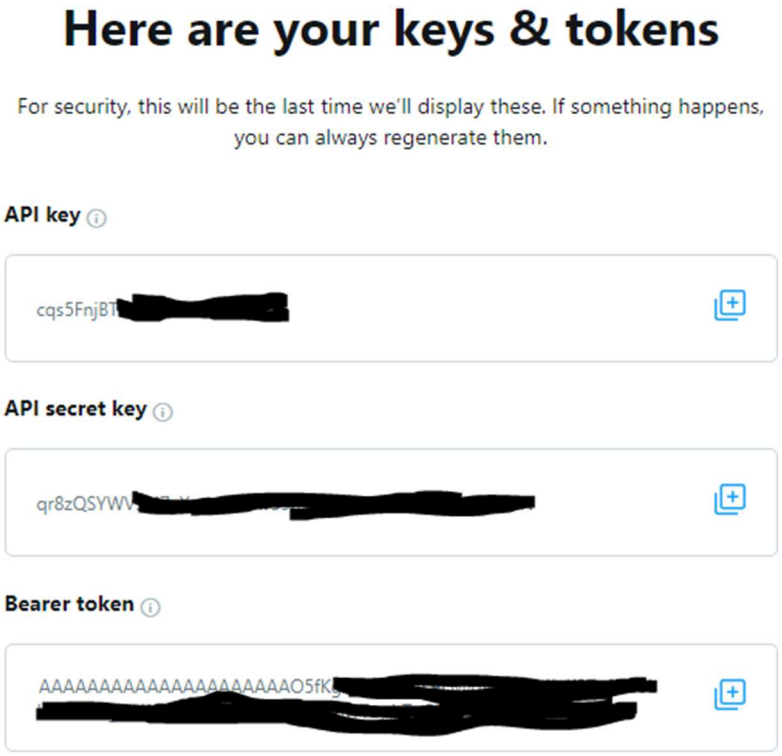


Figura 22 - API key, API secret key y Bearer token de Twitter

4.3 Creación de la VPC

Para crear la VPC debemos acceder al servicio seleccionando 'VPC' en la consola de administración de AWS. Seleccionando la opción 'Crear VPC' accedemos al formulario de creación en el que debemos especificar el nombre de la VPC único en la cuenta (VPC-aplicacion), el rango de direcciones IP (10.0.0.0/16), si la red utilizará o no direccionamiento IPv6 (no) y si utilizará hardware dedicado (no). Dando un rango /16 a la red tendremos margen para crear varias subredes /24 con 254 direcciones IP libres cada una (bastantes más de las que necesitamos). No necesitamos direccionamiento IPv6 ni hardware dedicado para nuestra aplicación, además esta última opción conlleva gastos adicionales. Una vez creada la VPC, debemos activar la resolución DNS en nuestra red, esto es necesario para acceder más tarde al clúster EMR que alojará. Los detalles de nuestra VPC pueden verse en la Tabla 4.

VPC-ID	vpc-00236ed41cde740f2	
Nombre	VPC-aplicacion	
Opciones DNS	DNS Hostnames	Enabled
	DNS Resolution	Enabled
Tenancy	Default	
DHCP options set	Set de opciones DHCP por defecto	
Route table	Tabla de rutas por defecto	
Network ACL	Network ACL por defecto	
Default VPC	NO	

CIDRs	IPv4 CIDR	10.0.0.0/16
	IPv6 pool	-
	IPv6 CIDR	-

Tabla 4 - Propiedades de VPC-Aplicacion

El siguiente paso es dividir la VPC en seis subredes: dos públicas (una en cada zona de disponibilidad), dos para la web y dos para los datos. Los CIDRs de cada subred serán 10.0.0.0/24, 10.0.1.0/24, 10.0.2.0/24, 10.0.3.0/24, 10.0.4.0/24 y 10.0.5.0/24 respectivamente. Para crear una subred seleccionamos la opción 'Create Subnet' en la pestaña 'Subnets' del servicio de VPC y seleccionamos la VCP en la que se va a crear (VPC-aplicacion), el nombre de la subred, la zona de disponibilidad (us-east-1a o us-east.1b) y el CIDR. Una vez creadas, podemos visualizarlas en el panel de subnets como vemos en la Figura 23.

Name	Subnet ID	State	VPC	IPv4 CIDR
public-a	subnet-08fbcef69e026b30b	Available	vpc-00236ed41cde740f2 | VPC-aplicacion	10.0.0.0/24
public-b	subnet-089a0ce84fd861293	Available	vpc-00236ed41cde740f2 | VPC-aplicacion	10.0.1.0/24
Web-a	subnet-0fff9e002805c5bee	Available	vpc-00236ed41cde740f2 | VPC-aplicacion	10.0.2.0/24
Web-b	subnet-0122b2a6508438bd3	Available	vpc-00236ed41cde740f2 | VPC-aplicacion	10.0.3.0/24
Datos-a	subnet-00a552c774990b9ef	Available	vpc-00236ed41cde740f2 | VPC-aplicacion	10.0.4.0/24
Datos-b	subnet-0dda8bf6ae0c2a68e	Available	vpc-00236ed41cde740f2 | VPC-aplicacion	10.0.5.0/24

Figura 23 - Subredes de la aplicación

Es necesario activar la opción 'Auto-assign public IPv4 address' para poder conectarnos a la máquina bastión mediante SSH desde el exterior de AWS. Los detalles de las subnets son los que se ven en la Tabla 5.

Subnet-ID	[subnet-0dda8bf6ae0c2a68e, subnet-00a552c774990b9ef, subnet-0122b2a6508438bd3, subnet-0fff9e002805c5bee, subnet-089a0ce84fd861293, subnet-08fbcef69e026b30b]	
Nombre	[Public-a. Public-b, Web-a, Web-b, Datos-a, Datos-b]	
VPC	VPC-aplicacion	
CIDRs	IPv4 CIDR	[10.0.0.0/24, 10.0.1.0/24, 10.0.2.0/24, 10.0.3.0/24, 10.0.4.0/24, 10.0.5.0/24]
	IPv6 CIDR	[-, -, -, -, -, -]
Availability Zone	[us-east-1a, us-east-1b, us-east-1a, us-east-1b, us-east-1a, us-east-1b]	
Network border group	us-east-1	
Auto-assign IP	Auto-assign public IPv4 address	[Yes, Yes, No, No, No, No]
	Auto-assign IPv6 address	No

Tabla 5 - Propiedades de las subredes

A continuación, creamos el Internet Gateway que utilizarán las máquinas de la subred pública (únicamente la máquina bastión). Recordemos que es un componente gratuito. Para crearlo solamente accedemos a la pestaña 'Internet Gateways' del servicio VPC, seleccionamos 'Create Internet Gateway', le damos un nombre (aplicación-ig) y seleccionamos la VPC en la que estará alojada (VPC-aplicacion). Podemos ver como se ha creado en el panel de Internet Gateway, como vemos en la Figura 24.

Name	Internet gateway ID	State	VPC ID	Owner
Aplicacion-ig	igw-0ca52d3ad905264d1	Attached	vpc-00236ed41cde740f2 | VPC-aplicacion	918498776862

Figura 24 - Internet Gateway de la aplicación

Después creamos la NAT Gateway que dará acceso a internet a las máquinas de las subredes privadas sin exponer sus direcciones IP. Para ello accedemos a la pestaña 'NAT Gateways' en el servicio de VPC y seleccionamos 'Create NAT Gateway'. En el formulario que nos aparece debemos dar nombre a la NAT Gateway (aplicación-NAT), seleccionar la subred en la que estará alojada (public-a, ya que necesario que se encuentre en una subred pública) y proporcionarle una Elastic-IP. Como no tenemos creada ninguna Elastic-IP seleccionamos la opción 'Allocate Elastic IP' para que se cree una y se le vincule automáticamente (como hemos dicho antes, eliminaremos la NAT Gateway cuando no la necesitemos durante el desarrollo para reducir gastos, también es necesario liberar la Elastic-IP que se autoasigna, ya que una Elastic-IP sin asociar a ningún componente conlleva cargos extra). Podemos ver como se ha creado en el panel de NAT Gateway, como vemos en la Figura 25.

Name	NAT gateway ID	State	VPC	Subnet
aplicacion-NAT	nat-05cb9b6124...	Available	vpc-00236ed41cde740f2 / VPC-aplicacion	subnet-08fbcef69e026b30b / public-a

Figura 25 - NAT Gateway de la aplicación

El último paso es crear las tablas de rutas que utilizarán las subredes. Crearemos dos tablas de rutas: una para las subredes públicas y otra para las subredes privadas. Para ello seleccionamos la opción 'Create Route Table' en la pestaña 'Route Tables' del servicio de VPC y asignamos a cada tabla un nombre y una VPC (VPC-aplicacion). Una vez creadas, les asignaremos las reglas de enrutamiento seleccionando en cada tabla la opción 'edit routes'. Ambas tablas tendrán una regla por defecto que redirigirá todo el tráfico con destino 10.0.0.0/16 de forma local. El resto del tráfico se redirigirá, en el caso de las subredes públicas al Internet Gateway y, en el caso de las subredes privadas, al NAT Gateway. Podemos ver las reglas de las subredes públicas en la Figura 26 y las reglas de las subredes privadas en la Figura 27.

Destination	Target	Status
10.0.0.0/16	local	active
0.0.0.0/0	igw-0ca52d3ad905264d1	active

Figura 26 – Reglas de la tabla de rutas de las subredes públicas

Destination	Target	Status
10.0.0.0/16	local	active
0.0.0.0/0	nat-05cb9b612474f7ae6	active

Figura 27 – Reglas de la tabla de rutas de las subredes privadas

Cuando tengamos las tablas creadas, volvemos al panel de Subnets a asignamos a cada una su correspondiente tabla de rutas. Podemos ver en la Figura 28 como cada tabla está asignada al número correcto de subredes.

Name	Route Table ID	Explicit subnet associations	VPC ID
Public	rtb-08565907ea6f83dc1	2 subnets	vpc-00236ed41cde740f2 | VPC-aplicacion
Private	rtb-07633ed58e9556a47	4 subnets	vpc-00236ed41cde740f2 | VPC-aplicacion

Figura 28 - Tablas de rutas

4.4 Creación del Bucket S3

Dentro del Bucket S3 se guardará tanto los ficheros con el código necesario para el funcionamiento de cada uno de los componentes como los resultados del procesamiento del clúster EMR. Así que tendremos cuatro carpetas: productor (con el código del productor de Twitter), cluster-EMR (con el script de Spark que ejecutará el clúster), web (con la página web que servirá el grupo de auto-escalado) y resultados (con los resultados del procesamiento del clúster EMR).

Para crear un Bucket accedemos al servicio de S3 desde la consola de AWS y seleccionamos la opción 'Create Bucket'. Debemos dar un nombre único a nivel mundial al Bucket (bucket-aplicación-upna) y seleccionar la región. En la configuración de acceso público seleccionamos 'Bloquear todo el acceso público', de forma que, si queremos que algún archivo sea público (contenido de la web), lo configuraremos manualmente. Habilitamos el control de versiones, por si necesitamos recuperar un archivo perdido y creamos el bucket.

Una vez creado el Bucket, en la pestaña objetos, seleccionamos la opción 'crear carpeta' y creamos cada una de las carpetas nombradas. Podemos ver el árbol de directorios en la Figura 29.

Nombre	Tipo
cluster-EMR/	Carpeta
productor/	Carpeta
resultados/	Carpeta
web/	Carpeta

Figura 29 - Directorios del Bucket de S3

A continuación, creamos una regla para eliminar de forma automática los archivos del directorio de resultados que lleven creados más de un día, de esta forma evitamos el riesgo de que el tamaño del Bucket crezca en exceso. En la pestaña de administración del Bucket seleccionamos la opción 'Crear la regla del ciclo de vida', le damos un nombre (delete-after-1-day), en el tipo de filtro especificamos el prefijo 'resultados/' y, en acciones, seleccionamos la acción 'Hacer que venzan las versiones actuales de los objetos' y en número de días escribimos 1.

Por último, dado que la web tendrá que realizar peticiones HTTP al Bucket, es necesario activar el uso compartido de recursos entre orígenes, o CORS. Para activarlos accedemos a la pestaña 'permisos' del Bucket y en la sección 'CORS' le pasamos un JSON con las propiedades como el que aparece en la Figura 30 - Activación de los CORS para el Bucket S3.

Figura 30 - Activación de los CORS para el Bucket S3

4.5 Creación de la máquina Bastión

Antes de lanzar cualquier instancia de Amazon EC2 es necesario crear el par de claves que nos permitirá conectarnos mediante SSH a la máquina. Lo normal en un equipo de desarrollo es crear un par de clave por instancia y proporcionar cada clave únicamente a los desarrolladores que deban tener acceso a esas máquinas, pero como nosotros solo somos una persona, crearemos un solo par de claves que utilizaremos para todas las instancias.

Para crear un par de claves accedemos a la pestaña 'Key Pairs' en el servicio de Amazon EC2 y seleccionamos la opción 'Create Key Pair'. A continuación, debemos darle un nombre (aplicación-kp) y seleccionar el formato del archivo de la clave privada: pem (para usar desde OpenSSH) o ppk (para usar desde PuTTY), nosotros accederemos desde PuTTY, así que seleccionamos la opción ppk. Una vez creado el par de claves se nos descarga automáticamente el archivo con extensión '.ppk' que deberemos guardar de forma segura ya que, si lo perdemos, no podremos acceder más a ninguna de las instancias levantadas con esa clave. En el panel de Key Pairs podemos ver que se ha creado correctamente (Figura 31).

Nombre	Huella digital	ID
aplicacion-kp	0b:fe:a8:80:cc:4f:11:3c:ea:ce:9e:7f:0a:4c:1c:54:02:e5:7b:6b	key-05e658d6134923ea4

Figura 31 - Par de claves de la aplicación

El siguiente paso es crear un grupo de seguridad para la instancia. Un grupo de seguridad es un conjunto de reglas de entrada y salida que hacen el papel del firewall. Para crear un grupo de seguridad seleccionamos la opción 'Create Security Group' en la pestaña 'Security Groups' del servicio de Amazon EC2. Debemos proporcionarle un nombre (seg-bastion), una descripción

y la VPC que alojará las instancias del grupo de seguridad (VPC-aplicacion). Únicamente le daremos una regla de entrada, que es la que permitirá el acceso al puerto 22 por SSH desde cualquier dirección IP (0.0.0.0/0). Las reglas de entrada permitirán todo el tipo de tráfico con cualquier destino (Figura 32).

Reglas de entrada

Tipo	Protocolo	Intervalo de puertos	Origen
SSH	TCP	22	0.0.0.0/0

Figura 32 - Reglas de entrada para el grupo de seguridad del bastión

Con el par de claves y el grupo de seguridad creados podemos proceder a lanzar la instancia. En la pestaña 'instances' del servicio de EC2 seleccionamos la opción 'launch instance':

- El primer paso es la selección de la AMI: tenemos imágenes de Ubuntu, de Windows, de macOS (desde hace unos pocos meses) y de Amazon Linux entre otras, seleccionamos una imagen de Ubuntu de propósito general.
- El segundo paso es seleccionar el tipo de instancia: como no vamos a necesitar que la máquina bastión tenga unas capacidades muy altas, seleccionamos el tipo 't2.micro', con una CPU y 1 GiB de memoria.
- El tercer paso es la configuración de la instancia: En las opciones 'Network' y 'Subnet' seleccionamos la VPC de la aplicación y la subred 'public-a' y dejamos el resto de las opciones como están por defecto.
- El cuarto paso es seleccionar el tipo de almacenamiento del EBS: elegimos un EBS de propósito general (gp2) y le damos 8 GiB de tamaño.
- El quinto paso es añadir etiquetad: únicamente vamos a añadir una etiquita llamada 'name' y la vamos a dar el valor 'bastion'.
- El sexto y último paso es asignarle el grupo de seguridad que hemos creado para él.

Los detalles de la instancia creada pueden verse en la Tabla 6.

ID de la instancia	i-0d33039170ee18610	
Nombre	bastion	
ID de VPC	vpc-00236ed41cde740f2 (VPC-aplicacion)	
ID de subred	subnet-08fbcef69e026b30b (public-a)	
Direcciones IP	Dirección IPv4 pública	Variable (se reasigna aleatoriamente cada vez que se inicia)
	Dirección IPv4 privada	10.0.0.136
	Dirección IPv6	-
Tipo de instancia	t2.micro	
Características hardware	CPUs virtuales	1
	Memoria (Gib)	1
	Tamaño del volumen (GiB)	8
	Rendimiento de red	Hasta 1 Gbps
Plataforma	Ubuntu	
AMI	ID de AMI	ami-0885b1f6bd170450c

	Nombre de AMI	ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-20201026
	Ubivación de AMI	099720109477/ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-20201026
Rendimiento ampliable	Créditos de CPU obtenidos por hora	6
	Créditos máximos	144
	Utilización de referencia por CPU	10%
Par de claves	aplicacion-kp	
Grupo de seguridad	ID de grupo de seguridad	sg-06978555701ceac3a (seg-bastion)
	Reglas de entrada	Puertos: 22, Protocolo: TCP, Origen: 0.0.0.0/0
	Reglas de salida	Puertos: Todo, Protocolo: Todo, Origen: 0.0.0.0/0

Tabla 6 - Propiedades de la máquina bastion

Nos conectamos a la máquina mediante SSH siguiendo los pasos del Apéndice 1 y comprobamos que tenemos conexión a internet (a través del Internet Gateway, ya que estamos en una de las subredes públicas) haciendo un ping a www.google.com (Figura 33).

```
Using username "ubuntu".
Authenticating with public key "Claves_de_test"
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-1029-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Wed Dec 16 15:26:11 UTC 2020

  System load:  0.0               Processes:             104
  Usage of /:   29.3% of 7.69GB   Users logged in:       0
  Memory usage: 21%               IPv4 address for eth0: 10.0.0.136
  Swap usage:   0%

 * Introducing self-healing high availability clusters in MicroK8s.
   Simple, hardened, Kubernetes for production, from RaspberryPi to DC.

     https://microk8s.io/high-availability

22 updates can be installed immediately.
0 of these updates are security updates.
To see these additional updates run: apt list --upgradable


The list of available updates is more than a week old.
To check for new updates run: sudo apt update

Last login: Wed Dec  9 15:09:11 2020 from 88.9.64.150
ubuntu@ip-10-0-0-136:~$ ping www.google.com
PING www.google.com (172.217.5.228) 56(84) bytes of data.
64 bytes from iad30s07-in-f228.1e100.net (172.217.5.228): icmp_seq=1 ttl=112 tim
e=0.743 ms
64 bytes from iad30s07-in-f228.1e100.net (172.217.5.228): icmp_seq=2 ttl=112 tim
e=0.785 ms
64 bytes from iad30s07-in-f228.1e100.net (172.217.5.228): icmp_seq=3 ttl=112 tim
e=0.789 ms
^C
--- www.google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2015ms
rtt min/avg/max/mdev = 0.743/0.772/0.789/0.020 ms
ubuntu@ip-10-0-0-136:~$
```

Figura 33 - Comprobación de la conexión a Internet del bastión

4.6 Creación del productor de Twitter

Como con la instancia anterior, necesitamos tener dos componentes: el par de claves y el grupo de seguridad. El par de claves que utilizaremos será el mismo (aplicación-kp). El grupo de seguridad se llamará 'seg-productor' y contendrá las mismas reglas que el grupo de seguridad de la máquina bastión excepto qué, en vez de permitir el tráfico de entrada de SSH desde cualquier dirección IP, solo se permitirá desde instancias que pertenezcan al grupo de seguridad del bastión (es decir, solo desde el bastión). Estas reglas pueden verse en la Figura 34.

Reglas de entrada

Tipo	Protocolo	Intervalo de puertos	Origen
SSH	TCP	22	sg-06978555701ceac3a (seg-bastion)

Figura 34 - Reglas de entrada del grupo de seguridad del productor

Para el lanzamiento de la instancia, seguimos los mismos pasos que con la instancia anterior con los siguientes cambios:

- En vez de seleccionar la subred 'public-a', seleccionamos la subred 'Datos-a'.
- En vez del nombre de 'bastión', le damos el nombre de 'Productor-Twitter'.
- Le asignamos el grupo de seguridad 'seg-productor'.

Los detalles de la instancia creada pueden verse en la Tabla 7.

ID de la instancia	i-09878cb6d220d49c4	
Nombre	Productor-Twitter	
ID de VPC	vpc-00236ed41cde740f2 (VPC-aplicacion)	
ID de subred	subnet-00a552c774990b9ef (Datos-a)	
Direcciones IP	Dirección IPv4 pública	Variable (se reasigna aleatoriamente cada vez que se inicia)
	Dirección IPv4 privada	10.0.4.36
	Dirección IPv6	-
Tipo de instancia	t2.micro	
Características hardware	CPUs virtuales	1
	Memoria (Gib)	1
	Tamaño del volumen (GiB)	8
	Rendimiento de red	Hasta 1 Gbps
Plataforma	Ubuntu	
AMI	ID de AMI	ami-0885b1f6bd170450c
	Nombre de AMI	ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-20201026
	Ubicación de AMI	099720109477/ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-20201026
Rendimiento ampliable	Créditos de CPU obtenidos por hora	6
	Créditos máximos	144
	Utilización de referencia por CPU	10%
Nombre del par de claves	aplicacion-kp	
Grupo de seguridad	ID de grupo de seguridad	sg-097efaa827bc14f51 (seg-productor)
	Reglas de entrada	Puertos: 22, Protocolo: TCP, Origen: sg-06978555701ceac3a (seg-bastion)
	Reglas de salida	Puertos: Todo, Protocolo: Todo, Origen: 0.0.0.0/0

Tabla 7 - Propiedades de la máquina Productor-Twitter

Una vez que hemos creado la máquina y que hemos comprobado que tiene conexión a internet (esta vez a través de la NAT Gateway) debemos instalar una serie de programas que nos permitirán descargar el script del Bucket S3 y ejecutarlo. El primer paso es instalar el sistema gestor de paquetes de python con el comando que vemos en la Figura 35.

```
ubuntu@ip-10-0-4-36:~$ sudo apt-get install python3-pip
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

Figura 35 - Instalación de pip

Una vez tenemos pip instalado debemos utilizarlo para instalar el cliente de línea de comandos de AWS. Este cliente nos permitirá, entre otras muchas cosas, descargar archivos de nuestros Buckets S3. Para instalarlo ejecutamos el comando que vemos en la Figura 36.

```
ubuntu@ip-10-0-4-36:~$ pip3 install awscli
Collecting awscli
```

Figura 36 - Instalación de awscli

Con el cliente de AWS instalado podemos navegar por los archivos y directorios de nuestros Buckets utilizando el comando 'aws s3 ls'. En la Figura 37 podemos ver como llegamos al archivo main.py (que es el que contiene el código del productor) utilizando este comando (cuando listamos los directorios raíz no aparecen las carpetas 'resultados', 'cluster-EMR' y 'web' porque todavía no se habían creado).

```
ubuntu@ip-10-0-4-36:~$ aws s3 ls
2020-11-11 10:55:23 aws-logs-918498776862-us-east-1
2020-11-15 17:26:32 bucket-upna-aplicacion
ubuntu@ip-10-0-4-36:~$ aws s3 ls s3://bucket-upna-aplicacion
                           PRE productor/
                           PRE pruebas/
ubuntu@ip-10-0-4-36:~$ aws s3 ls s3://bucket-upna-aplicacion/productor/
2020-11-24 15:02:25          0
2020-11-24 15:06:34       1699 main.py
```

Figura 37 - Navegando por los directorios del Bucket S3

Ahora que tenemos la dirección del archivo que buscamos debemos crear directorio de la máquina en el que se copiará el directorio del Bucket S3. Siguiendo los comandos que aparecen en la Figura 38 utilizamos el comando 'aws s3 cp' para copiar el contenido de la carpeta 'productor' de nuestro Bucket y hacemos un cat al archivo 'main.py' para comprobar que se ha descargado correctamente.

```
ubuntu@ip-10-0-4-36:~$ mkdir productor
ubuntu@ip-10-0-4-36:~$ aws s3 cp s3://bucket-upna-aplicacion/productor productor --recursive
download: s3://bucket-upna-aplicacion/productor/main.py to productor/main.py
ubuntu@ip-10-0-4-36:~$ cd productor/
ubuntu@ip-10-0-4-36:~/productor$ ls
main.py
ubuntu@ip-10-0-4-36:~/productor$ cat main.py
import tweepy
from tweepy import OAuthHandler
from tweepy import Stream
from tweepy.streaming import StreamListener
import socket
import json
```

Figura 38 - Descargando el código del productor

El contenido del archivo 'main.py' puede verse en Apéndice 2 – Código del productor de Twitter. Lo primero que hace este script es hacer la importaciones: de la librería 'tweepy' importamos la clase 'OAuthHandler' (que nos permite autenticarnos en la API), la clase 'Stream' (que nos permite crear un flujo de tweets y filtrarlo por palabras clave o hashtags) y la clase 'StreamListener' (que nos permite crear un escuchador para establecer que debe hacer el programa cuando reciba un tweet), de la librería 'kafka' importamos la clase 'KafkaProducer' (que nos permite conectarnos a un clúster de kafka y publicar mensajes en un topic) y la librería 'json'.

A continuación, el script inicializa las credenciales necesarias para acceder a la API de Twitter (por motivos de seguridad se han ocultado). Después se define la clase 'TweetsListener' que recibe como parámetro de entrada un 'StreamListener' de 'tweepy' y en el que definimos que hacer con los tweets entrantes. Como vemos en la Figura 39, el primer paso es transformar en tweet a un objeto JSON utilizando la función 'json.load()' e inicializar el mensaje que e enviará a kafka como un diccionario vacío. Si el tweet contiene el campo 'place' y este no está vacío, añadimos al mensaje de kafka el campo 'country_code' con el código del país del tweet en formato ISO 3166-1 alpha-3. A su vez, si el tweet contiene el campo 'entities' y su campo de hashtags no está vacío, añadimos los hashtags uno a uno en el campo del mensaje de kafka 'hashtags'. Por último, si el mensaje de kafka no está vacío (es posible que el tweet no tuviera ubicación ni hashtags), se envía al clúster de kafka con el topic 'tweets'.

```
def on_data(self, data):
    try:
        msg = json.loads( data )
        kafka_msg = {}
        if 'place' in msg:
            if msg['place'] != None:
                kafka_msg['country_code'] = msg['place']['country_code']
        if 'entities' in msg:
            if msg['entities']['hashtags'] != []:
                kafka_msg['hashtags'] = []
                for hash in msg['entities']['hashtags']:
                    kafka_msg['hashtags'].append(hash['text'])
        if kafka_msg != {}:
            self.productor.send('tweets', json.dumps(kafka_msg).encode())
        return True
    except BaseException as e:
        print("Error: %s" % str(e))
    return True
```

Figura 39 - Preparación de los tweets

Lo siguiente que hace el script es definir la función 'sendData' (Figura 40). Esta función crea el objeto 'OAuthHandler' utilizando nuestras credenciales, inicializa el objeto 'Stream' pasándole como argumento las credenciales y un objeto 'TweetListener' y añade al stream un filtro que consiste en una lista de palabras clave que debe contener el tweet (al menos una de la lista).

```
def sendData(prod):
  auth = OAuthHandler(consumer_key, consumer_secret)
  auth.set_access_token(access_token, access_secret)

  twitter_stream = Stream(auth, TweetsListener(prod))
  twitter_stream.filter(track=['加油武汉','covid','#COVID-19',
```

Figura 40 - Función sendData

Por último, lo que hace el script es inicializar el productor de kafka pasándole como argumento la dirección del clúster (10.0.4.192:9092) y llamar a la función 'sendData'.

Para ejecutar el script lanzamos el comando 'python3 productor/main.py'.

Ahora vamos a explicar el script que de python que recibe los resultados de los conteos a través de Kafka y los guarda en el Bucket S3. Este script lo hemos descargado a la vez que el anterior ya que hemos copiado el directorio entero. Podemos ver el código completo del script en el apéndice 4.

Lo primero que hace el script es importar las librerías y clases necesarias. De la librería 'pykafka' importamos la clase 'KafkaClient' (que nos permite crear un cliente para conectarnos al clúster de Kafka) y la clase OffsetType (que utilizaremos para que el cliente no empiece a leer los resultados desde el principio, sino desde que lo lanzamos). Importamos la librería 'datetime' que nos permite obtener la fecha y la hora actual (para dar el nombre al archivo de S3). Importamos también la librería 'json' (para tratar con objetos en formato JSON). Importamos por último la librería 'os' que sirve para realizar llamadas al sistema operativo.

En la Figura 41 vemos lo siguiente que hace el script que es inicializar la variable. En primer lugar, crea el cliente utilizando 'KafkaClient' y pasándole la dirección del clúster de Kafka ("10.0.4.192.9092"), después especificamos el topic ('resultados') y utilizamos la función 'get_simple_consumer' para crear un consumidor para el topic. A esta última función le debemos pasar dos parámetros (utilizando el objeto 'OffsetType') para decirle que no debe empezar a leer del topic desde que se creó el topic, sino desde que lanzamos el cliente. A continuación, inicializamos el diccionario 'data' que contendrá los campos con los valores que guardaremos en el archivo de resultados. Los nombres de los campos son los que hemos explicado en la sección 3.4. Por último, inicializamos las variables 'last_hour' y 'last_minute' que contienen la hora y el minuto de la última vez que se ha guardado un archivo en el Bucket S3.

```
client = KafkaClient(hosts = "10.0.4.192:9092")
topic = client.topics['resultados']

consumer = topic.get_simple_consumer(auto_offset_reset = OffsetType.LATEST, reset_offset_on_start=True)

data = {}
data["Country_count"] = []
data["Hashtag_world_count"] = []
data["Hashtag_countries_count"] = {}

now = datetime.datetime.now()
last_hour = now.hour
last_minute = now.minute
```

Figura 41 - Inicialización del script de resultados

En la Figura 42 vemos el código que se ejecuta cada 10 minutos y se encargar de crear y guardar el archivo en S3. Lo primero que hace es ordenar los resultados de cada uno de los campos según del conteo de mayor a menor (Esto se hace también en el clúster EMR pero, al

tener varias particiones, es posible que los mensajes no lleguen del todo ordenados) y limitar el número de los hashtags mundiales y de cada país a 10 (ya que las tablas de la web tendrán 10 elementos). Después, creamos el archivo que enviaremos al bucket (serializando el diccionario a través de la función 'json.dump') y hacemos una llamada al sistema para que copie la carpeta de resultados en el directorio del Bucket S3 con la CLI de AWS. Es importante que en el nombre del archivo los minutos sean múltiplos de diez para facilitar las tareas de la web. Para acabar, volvemos a inicializar la hora, los minutos y el objeto 'data'.

```
now = datetime.datetime.now()
if now.hour != last_hour or now.minute - last_minute >= 10:

    data["Country_count"].sort(key = lambda x: -x[1])
    data["Hashtag_world_count"].sort(key = lambda x: -x[1])
    data["Hashtag_world_count"] = data["Hashtag_world_count"][:10]
    for cc in data["Hashtag_countries_count"]:
        data["Hashtag_countries_count"][cc].sort(key = lambda x: -x[1])
        data["Hashtag_countries_count"][cc] = data["Hashtag_countries_count"][cc][:10]

    path = "resultados/" + str(last_hour) + "_" + str(last_minute - (last_minute % 10)) + ".txt"
    with open(path, 'w') as outfile:
        json.dump(data, outfile)
    os.system('aws s3 cp resultados s3://bucket-upna-aplicacion/resultados --recursive')
    os.remove(path)
    print('Conteo enviado al Bucket')

    last_hour = now.hour
    last_minute = now.minute

    data["Country_count"] = []
    data["Hashtag_world_count"] = []
    data["Hashtag_countries_count"] = {}
```

Figura 42 - Guardando los resultados en el Bucket S3

En la Figura 43 se muestra cómo se trata cada mensaje de entrada para poblar el objeto 'data'.

```
msg = json.loads(msg.value)
tipo = msg[0].split(' ')[0]
conteo = msg[1]

if tipo == 'cc':
    cc = msg[0].split(' ')[1]
    data["Country_count"].append([cc,conteo])

if tipo == 'ht':
    ht = msg[0].split(' ')[1]
    data["Hashtag_world_count"].append([ht,conteo])

if tipo == 'htcc':
    cc = msg[0].split(' ')[1]
    ht = msg[0].split(' ')[2]
    if not cc in data["Hashtag_countries_count"]:
        data["Hashtag_countries_count"][cc] = []
    data["Hashtag_countries_count"][cc].append([ht,conteo])
```

Figura 43 - Tratamiento de mensaje de resultados

4.7 Creación del clúster de kafka

Como siempre, utilizaremos el par de claves 'aplicacion-kp'. El grupo de seguridad se llamará 'seg-cluster-kafka' contendrá la misma regla para el acceso desde el puerto 22 del grupo de seguridad del productor de Twitter (la que permite únicamente la conexión desde el bastión), pero también contendrá dos reglas más: una que habrá el puerto 9092 (Apache Kafka) y otra que habrá el puerto 2181 (Apache ZooKeeper). Estas reglas pueden verse en la Figura 44 (si vemos que por defecto se crean dos reglas, una con origen 0.0.0.0/0 y otra con origen ::/0, es porque acepta tráfico de IPv4 e IPv6).

Reglas de entrada

Tipo	Protocolo	Intervalo de puertos	Origen
SSH	TCP	22	sg-06978555701ceac3a (seg-bastion)
TCP personalizado	TCP	2181	0.0.0.0/0
TCP personalizado	TCP	2181	::/0
TCP personalizado	TCP	9092	0.0.0.0/0
TCP personalizado	TCP	9092	::/0

Figura 44 - Reglas de entrada del grupo de seguridad de kafka

Para el lanzamiento de la instancia, seguimos los mismos pasos que con la instancia anterior con los siguientes cambios:

- En vez del nombre de 'Productor-Twitter', le damos el nombre de 'cluster-kafka'.
- Le asignamos el grupo de seguridad 'seg-cluster-kafka'.
- En vez de seleccionar una instancia de tipo 't2.micro', seleccionamos una del tipo 't2.small'.

Los detalles de la instancia creada pueden verse en la Tabla 8.

ID de la instancia	i-0fb5f16253e42abd8	
Nombre	cluster-kafka	
ID de VPC	vpc-00236ed41cde740f2 (VPC-aplicacion)	
ID de subred	subnet-00a552c774990b9ef (Datos-a)	
Direcciones IP	Dirección IPv4 pública	Variable (se reasigna aleatoriamente cada vez que se inicia)
	Dirección IPv4 privada	10.0.4.192
	Dirección IPv6	-
Tipo de instancia	t2.small	
Características hardware	CPUs virtuales	1
	Memoria (Gib)	1
	Tamaño del volumen(GiB)	8
	Rendimiento de red	Hasta 1 Gbps
Plataforma	Ubuntu	
AMI	ID de AMI	ami-0885b1f6bd170450c
	Nombre de AMI	ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-20201026

	Ubivación de AMI	099720109477/ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-20201026
Rendimiento ampliable	Créditos de CPU obtenidos por hora	12
	Créditos máximos	288
	Utilización de referencia por CPU	20%
Nombre del par de claves	aplicacion-kp	
Grupo de seguridad	ID de grupo de seguridad	sg-0bc8f82acd14525df (seg-cluster-kafka)
	Reglas de entrada	Puertos: 22, Protocolo: TCP, Origen: sg-06978555701ceac3a (seg-bastion) Puertos: 2181, Protocolo: TCP, Origen: Todo Puertos: 9092, Protocolo: TCP, Origen: Todo
	Reglas de salida	Puertos: Todo, Protocolo: Todo, Origen: 0.0.0.0/0

Tabla 8 - Propiedades de la máquina cluster-kafka

Una vez que hemos creado la máquina y que hemos comprobado que tiene conexión a internet vamos a proceder a la instalación de Apache Kafka y Apache ZooKeeper. Primero debemos actualizar la librería de paquetes e instalar los programas 'nano' (para editar archivos de texto), 'openjdk-8-jdk' (con el conjunto de bibliotecas de Java) y 'wget' (para descargar mediante una petición HTTP) como vemos en la Figura 45. Podemos utilizar la opción '-y' si queremos que no se nos pida confirmación para la instalación de cada uno de los programas. Este proceso durará alrededor de un minuto.

```
ubuntu@ip-10-0-4-192:~$ sudo apt update && sudo apt install -y nano openjdk-8-jdk wget
```

Figura 45 - Instalación de paquetes para máquina kafka

Es siguiente paso, después de instalar las dependencias, es descargarse el programa de Apache Kafka de la página oficial utilizando el programa 'wget' y descomprimirlo utilizando el programa 'tar' como vemos en la Figura 46. Este proceso durará también alrededor de un minuto.

```
ubuntu@ip-10-0-4-192:~$ wget http://apache.uvigo.es/kafka/2.6.0/kafka_2.13-2.6.0.tgz && tar zxvf kafka_2.13-2.6.0
.tgz
--2020-11-20 15:49:06--  http://apache.uvigo.es/kafka/2.6.0/kafka_2.13-2.6.0.tgz
Resolving apache.uvigo.es (apache.uvigo.es)... 193.146.32.74, 2001:720:1214:4200::74
Connecting to apache.uvigo.es (apache.uvigo.es)|193.146.32.74|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 65537909 (63M) [application/x-gzip]
Saving to: 'kafka_2.13-2.6.0.tgz'

kafka_2.13-2.6.0.tgz         100%[=========================================>]  62.50M  2.34MB/s    in 12s

2020-11-20 15:49:19 (5.13 MB/s) - 'kafka_2.13-2.6.0.tgz' saved [65537909/65537909]
```

Figura 46 - Descargando y descomprimiendo Apache Kafka

Antes de lanzar el servidor de Kafka es necesario haber lanzado primero el servidor de ZooKeeper. Para ello debemos realizar un paso previo que es acceder al fichero de configuración de Zookeeper (que está en la ruta 'kafka_2.13-2.6.0/config/zookeeper.properties'). Una vez en el fichero de configuración, que vemos en la Figura 47, modificamos la variable dataDir, que tiene el valor por defecto '/tmp/zookeeper' (una carpeta temporal), por el valor '/zookeeper-data', de esta forma podemos revisar los logs creados por el programa para depurar errores sin que se borren periódicamente. El resto de los parámetros (como el del puerto del cliente en el número 2181) los dejaremos con su valor por defecto.

```
  GNU nano 4.8                    kafka_2.13-2.6.0/config/zookeeper.properties
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements.  See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License.  You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# the directory where the snapshot is stored.
dataDir=/zookeeper-data
# the port at which the clients will connect
clientPort=2181
# disable the per-ip limit on the number of connections since this is a non-production config
maxClientCnxns=0
# Disable the adminserver by default to avoid port conflicts.
# Set the port to something non-conflicting if choosing to enable this
admin.enableServer=false
# admin.serverPort=8080
```

Figura 47 - Fichero de configuración de Apache ZooKeeper

Podemos crear la carpeta zookeeper-data manualmente, pero si no lo hacemos se creará automáticamente. Para lanzar el proceso de ZooKeeper debemos ejecutar el archivo que se encuentra en la ruta 'kafka_2.13-2.6.0/bin/zookeeper-server-start.sh' pasándole como argumento la ruta del archivo de configuración que acabamos de modificar.

Ahora que se está ejecutando ZooKeeper, vamos a proceder a editar el archivo de configuración de Kafka (que se encuentra en la ruta 'kafka_2.13-2.6.0/config/server.properties'). El primer parámetro que nos encontramos es el 'broker.id', si estuviésemos levantando Kafka en varios nodos deberíamos darle un valor diferente en cada uno de ello, pero como vamos a ejecutarlo en solo nodo lo dejaremos en su valor por defecto que es 0. Al igual que con ZooKeeper, vamos a cambiar el valor de parámetro 'log.dirs', que es '/tmp/kafka-log' por defecto (una carpeta temporal), y lo vamos a dejar en solo '/kafka-logs' (como vemos en la Figura 48) para poder consultar los logs sin que se borren. Si estuviéramos levantando un nodo de Kafka en un nodo diferente al de ZooKeeper deberíamos cambiar el parámetro 'zookeeper.connect', que por defecto está como 'localhost:2181', a la dirección de su nodo.

```
# A comma separated list of directories under which to store log files
log.dirs=/kafka-logs
```

Figura 48 - Editando el fichero de configuración de Apache Kafka

Al igual que en el caso anterior, podemos crear la carpeta 'kafka-logs' manualmente o podemos dejar que se cree automáticamente. Para lanzar el proceso de Kafka debemos ejecutar el archivo que se encuentra en la ruta 'kafka_2.13-2.6.0/bin/kafka-server-start.sh' pasándole como argumento la ruta del archivo de configuración que acabamos de modificar.

A continuación, vamos a crear el topic 'tweets', que es el topic que se utilizará para enviar los tweets del productor al clúster de Hadoop. Para ello debemos ejecutar el archivo que se encuentra en la ruta 'kafka_2.13-2.6.0/bin/kafka-topics.sh' (como aparece en la) pasándole la opción '--create'. También demos pasarle el parámetro '—bootstrap-server' con el valor

’10.0.1.192:9092’ (ya que es posible crear los topics desde otras máquinas) y el parámetro ‘--topic’ con el valor ‘tweets’ (que es el nombre del topic).

Para comprobar que el topic se ha creado correctamente ejecutamos el mismo archivo como se ven en la Figura 49, pero en vez de pasarle el argumento ‘--create’, le pasamos el argumento ‘--list’. De esta forma se nos devolverá una lista con todos los topics del clúster y podremos comprobar que está el nuevo (si estamos trabajando con varios nodos y creamos el topic en uno, automáticamente se crea en todos los nodos).

```
ubuntu@ip-10-0-0-136:~$ kafka_2.13-2.6.0/bin/kafka-topics.sh --create --bootstrap-server 10.0.4.1
92:9092 --topic tweets
Created topic tweets.
ubuntu@ip-10-0-0-136:~$ kafka_2.13-2.6.0/bin/kafka-topics.sh --list --bootstrap-server 10.0.4.192
:9092
tweets
```

Figura 49 - Creando los topics de Kafka

Para crear el otro topic, el de los resultados del clúster EMR, seguimos el mismo proceso cambiando la palabra ‘tweets’ por ‘resultados’.

Para comprobar que nuestro clúster de Kafka funciona correctamente vamos a crear un cliente en la máquina bastión que produzca mensajes en el topic ‘tweets’ y otro, también en la misma máquina, que esté suscrito al mismo topic. Una vez hemos descargado Kafka en la máquina bastión igual que antes, para lanzar el productor, ejecutamos el archivo que está en la ruta ‘kafka_2.13-2.6.0/bin/kafka-console-producer.sh’ pasándole como argumentos la dirección del nodo de Kafka y el nombre del topic. Como vemos en la Figura 50 podemos escribir cadenas de texto que se enviarán al topic seleccionado. Para lanzar el consumidor creamos una nueva sesión y ejecutamos el archivo que está en ruta ‘kafka_2.13-2.6.0/bin/kafka-console-consumer.sh’ pasándole como argumentos la dirección del nodo de Kafka, el nombre del topic y la opción ‘—from-beginning’. Como vemos en la Figura 51 podemos leer los mensajes de ese topic, incluso los que se han enviado antes de conectarnos ya que hemos utilizado la opción ‘—from-beginning’.

```
ubuntu@ip-10-0-0-136:~$ kafka_2.13-2.6.0/bin/kafka-console-producer.sh --broker-list 10.0.4.192:9
092 --topic tweets
>hola
>34 532 2342 sffs 32
>......
>´5·645
```

Figura 50 - Productor de mensajes Kafka

```
ubuntu@ip-10-0-0-136:~$ kafka_2.13-2.6.0/bin/kafka-console-consumer.sh --bootstrap-server 10.0.4.
192:9092 --topic tweets --from-beginning
hola
34 532 2342 sffs 32
......
´5·645
```

Figura 51 - Consumidor de mensajes Kafka

4.8 Creación del clúster EMR

Para crear un clúster seleccionamos la opción ‘crear clúster’ en el servicio de EMR de la consola de AWS. Por defecto se nos mostrarán las opciones rápidas de configuración, pero nosotros seleccionaremos las opciones avanzadas.

Las opciones avanzadas están divididas en cuatro pasos. El primero es seleccionar el software y los pasos. Se nos mostrará un panel de selección como el la Figura 52 en el que debemos seleccionar la versión 'emr-5.32.0' (las versiones de EMR son conjuntos de versiones de los diferentes programas de Hadoop estables y compatibles entre sí) y, de entre los programas, seleccionaremos únicamente Hadoop 2.10.1 y Spark 2.4.7, ya que no necesitaremos ninguno de los otros programas.

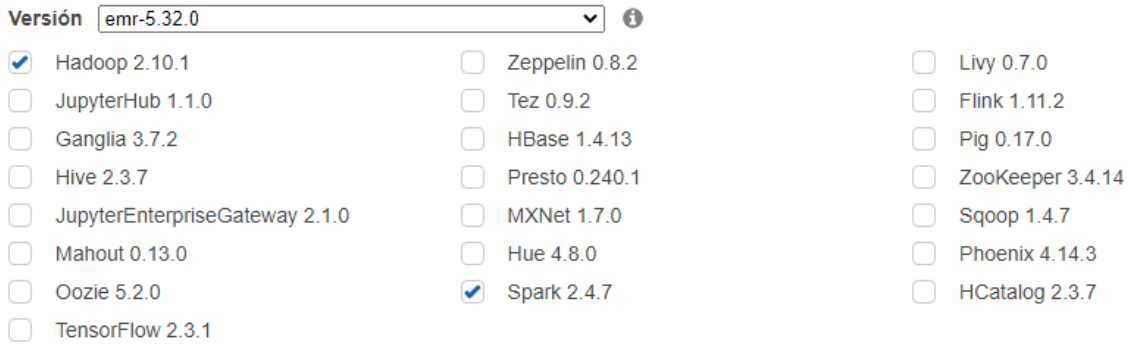


Figura 52 - Panel de selección de programas de EMR

El resto de las opciones que se nos muestran en este paso son, en primer lugar, la opción de activar la compatibilidad con instancias maestra múltiples que no seleccionaremos porque trabajaremos con un solo nodo maestro. La siguiente opción es la de activar el catálogo de datos de AWS Glue para los metadatos de Spark, tampoco la activaremos. La siguiente opción es la posibilidad de pasarle un archivo de configuración de software que tampoco utilizaremos. Por último, se nos da la opción de añadir pasos. Los pasos son unidades de trabajo que podemos enviar al clúster antes de su ejecución y que pueden contener, por ejemplo, tareas de Hadoop o de Spark. Nosotros no añadiremos ningún paso ya que lanzaremos la tarea después de la ejecución del clúster utilizando el comando 'spark-submit'.

El segundo paso de la configuración avanzada es el hardware. En primer lugar, deberemos seleccionar si emplearemos grupos de instancias uniformes (es decir, que utilizaremos el mismo tipo de instancia para todos los nodos) o flotas de instancias (que permite seleccionar diferentes tipos de instancias y subredes para los nodos), seleccionamos la primera opción. A continuación, debemos seleccionar la VPC y la subred en la que estará alojado el clúster, seleccionamos la VPC 'VPC-aplicacion' y la subred 'Datos-a'. Después debemos seleccionar los tipos de instancias del nodo maestro y de los nodos esclavos. Como vemos en la Figura 53 seleccionamos el tipo 'm5.xlarge', una instancia para el nodo maestro (es la única opción) y dos instancias iniciales para los nodos esclavos. Utilizaremos instancias bajo demanda en vez de instancias spot. En la siguiente opción, activaremos el 'Cluster scaling', que quiere decir que el número de instancias esclavas podrá incrementarse y reducirse automáticamente en función de la carga media de estas. Seleccionamos como número de instancias mínimas del grupo dos y como máximo cinco (estos valores pueden cambiarse si vemos que se necesitan más instancias). La última opción el tamaño del volumen de EBS de las instancias (con un máximo de 100 GiB), que dejaremos con su valor por defecto a 10 GiB.

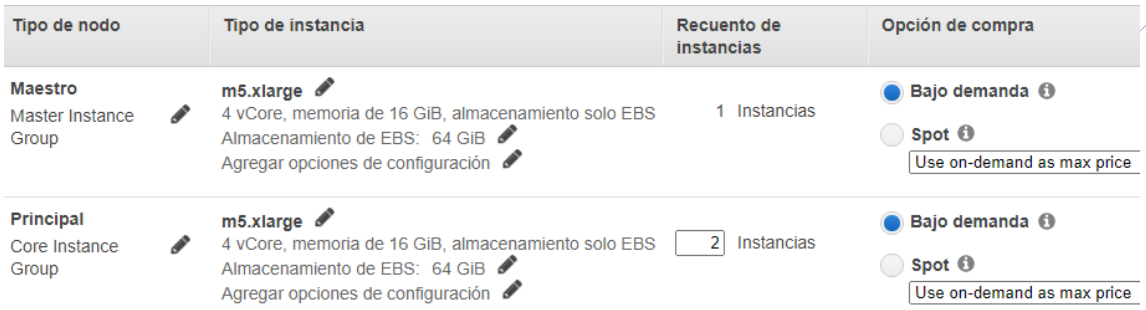


Figura 53 - Tipos de instancias del clúster EMR

El tercer paso de la configuración avanzada es la configuración general del clúster. En este paso únicamente seleccionaremos la opción de que cree un registro en un Bucket S3 en el que se guarden los logs del clúster. El resto de las opciones, como las etiquetas, no las vamos a utilizar.

El cuarto y último paso de la configuración avanzada es la seguridad. Aquí debemos seleccionar el par de claves que hemos estado utilizando en el resto de las máquinas, 'aplicación-kp'. En la sección de permisos debemos seleccionar el rol de IAM que tendrá el clúster, los nodos y el grupo de auto-escalado:

- Para el clúster seleccionamos el rol de IAM 'EMR_DefaultRole'. Este rol permite a EMR llamar a otros servicios de AWS como CloudWatch, EC2, IAM, S3 o SQS (siempre con una lista de permisos parciales sobre cada servicio).
- Para los nodos seleccionamos el rol de IAM 'EMR_EC2_DefaultRole'. Este rol proporciona a las instancias de EC2 del clúster acceso completo a algunos servicios de AWS necesarios para su funcionamiento como S3, RDS, DynamoDB o CloudWatch y acceso limitado a otros servicios como EC2, Glue, EMR o Kinesis.
- Para el grupo de auto-escalado seleccionamos el rol de IAM 'EMR_AutoScaling_DefaultRole'. Este rol permite al grupo de auto-escalado levantar y terminar instancias de EC2 en función de su carga. Para ello únicamente necesitas permisos parciales sobre CloudWatch y EMR.

Para terminar el cuarto paso debemos seleccionar los grupos de seguridad del nodo maestro y de los nodos esclavos. Como en caso de los roles de IAM utilizaremos los grupos de seguridad predeterminados de EMR:

- Para el nodo maestro seleccionamos el grupo de seguridad 'ElasticMapReduce-master' con el ID 'sg-0d8f82e32b59b2d34'. Este grupo de seguridad permite el acceso por TCP, UDP o ICMP a cualquier puerto desde cualquier máquina que tenga asignado el grupo de seguridad por defecto de nodo maestro o de nodo esclavo. También permite el acceso TCP al puerto 8443 a una serie de direcciones IP relacionadas con el funcionamiento de AWS. Como este grupo de seguridad no permite explícitamente el acceso mediante SSH debemos añadirlo una vez está creado el clúster.
- Para los nodos esclavos seleccionamos el grupo de seguridad 'ElasticMapReduce-slave' con el ID 'sg-0a81c1f0e37b3abb0'. Este grupo de seguridad permite el acceso por TCP, UDP o ICMP a cualquier puerto desde cualquier máquina que tenga asignado el grupo de seguridad por defecto de nodo maestro o de nodo esclavo.

Una vez que hemos configurado correctamente el clúster y que hemos modificado el grupo de seguridad para permitir las conexiones SSH vamos a conectarnos. Para ello primero necesitamos conocer la dirección de DNS pública que se le asigna automáticamente al nodo maestro. Como vemos en la Figura 54, esta dirección puede encontrarse en el apartado resumen de los detalles del clúster.

Figura 54 - DNS público de EMR

Cuando nos conectamos al nodo maestro (igual que hemos anteriormente hecho con el resto de las instancias de EC2) se nos pide que introduzcamos el nombre de usuario, que debe de ser 'hadoop'. El clúster viene con todos los programas que necesitamos, incluidos la interfaz de línea de comandos de AWS y el sistema de gestión de paquetes de python, 'pip'. Únicamente tenemos que instalar una dependencia que es la librería de python 'pykafka', que nos permitirá crear un cliente de Kafka para tener un flujo de entrada de datos continuo. Para instalarlo lanzamos el comando 'pip3 install pykafka' y lo tendremos instalado en unos segundos.

Para descargar el script con el programa de Spark Streaming del Bucket S3 seguiremos el mismo proceso que con el productor de Twitter. En primer lugar, creamos una carpeta (llamada 'script') y, una vez tenemos la URL de la carpeta que vamos a copiar, ejecutamos el comando 'aws s3 cp s3://bucket-upna-aplicación/cluster-EMR/ script'. Recordemos que este comando copia el contenido del directorio 'cluster-EMR' de nuestro Bucket a nuestra carpeta 'script'.

El último paso es enviar la tarea al clúster. Para ello utilizaremos el comando 'spark-submit'. El primer parámetro que debemos pasar a este comando son los paquetes. Como vamos a usar Kafka como fuente de entrada de datos tenemos que usar el paquete que se encuentra en la URL 'org.apache.spark:spark-streaming-kafka-0-8_2.11:2.4.3'. El siguiente parámetro es el nombre del script (que hemos llamado 'script.py'). El tercer parámetro es la dirección del nodo de ZooKeeper del clúster de Kafka, que es '10.0.4.192:2181'. Por último, le pasamos el nombre del topic al que se debe suscribir, que es 'tweets'. El comando final queda como 'spark-submit –packages org.apache.spark:spark-streaming-kafka-0-8_2.11:2.4.3 script.py, 10.0.4.192:2181 tweets'.

A continuación, vamos a explicar el contenido del código del script de Spark que se ejecuta en el clúster. Como siempre, primero importamos las librerías necesarias. De la librería 'pyspark' (que venía instalado por defecto en el clúster) importamos el objeto 'SparkContext' (es el punto de entrada a las funcionalidades de Spark y puede ser utilizado para tareas como crear un RDD), el objeto 'StreamingContext' (es el equivalente al 'SparkContext' para las

funcionalidades de Spark Streaming. Se crea a partir de un objeto 'SparkContext' y permite realizar tareas como crear un DStream) y el objeto KafkaUtils (permite a Spark Streaming crear un flujo de entrada de datos desde Kafka). De la librería 'pykafka' importamos el objeto 'KafkaClient' que nos permite crear un cliente de Kafka (lo utilizaremos para crear un productor que envíe los resultados al clúster de Kafka). Además, importamos la librería 'json' para realizar transformaciones entre objetos y Strings en formato JSON y la librería 'pprint' para tareas de debuggeo.

En la Figura 55 aparece la función que se ejecutara por cada tweet de entrada antes de hacer el conteo a través de una función 'map'. Está función se ejecutará dentro de un 'flatMap', por lo que se podrá crear más de un registro de salida por cada registro de entrada, por eso la salida de la función es un Array. Lo primero que hace es cargar el texto del mensaje (en formato JSON) a un objeto JSON e inicializa la variable 'contenido' que contiene el Array que será devuelto a final de la función. Si el mensaje de entrada contiene el campo 'country_code', entonces añadimos al Array de salida el string "cc " más el código del país(recordemos la estructura de los datos de salida del capítulo 3.4). Si el mensaje contiene el campo 'hashtags', entonces añadimos un al Array de salida el string "ht " más el hashtag por cada uno de los hashtags. Por último, si el mensaje contiene los dos campos anteriores, por cada hashtags, añadimos al Array de salida el string "htcc " más el código del país más cada hashtag.

```
def preparar_tweets(msg):
    tweet = json.loads(msg)
    contenido = []
    if 'country_code' in tweet:
        contenido.append("cc " + tweet['country_code'])
    if 'hashtags' in tweet:
        for hash in tweet['hashtags']:
            contenido.append("ht " + hash)
            if 'country_code' in tweet:
                contenido.append("htcc " + tweet['country_code'] + " " + hash)
    return contenido
```

Figura 55 - Función preparar_tweets del clúster EMR

Para que quede más claro el resultado de la función 'preparar_tweets' podemos ver un ejemplo en la Figura 56. En el ejemplo un mensaje que contiene dos hashtags y un código de país se descompone en cinco registros: uno para el código del país, uno para cada hashtag solo y uno para cada hashtag con el código del país. Podemos deducir que la cantidad de registros creados por la función viene dada por la siguiente fórmula:

$$nº\ de\ registros = \begin{cases} nº\ de\ hastags & si\ hay\ código\ de\ país \\ (nº\ de\ hashtags * 2) + 1 & si\ no\ hay\ código\ de\ país \end{cases}$$

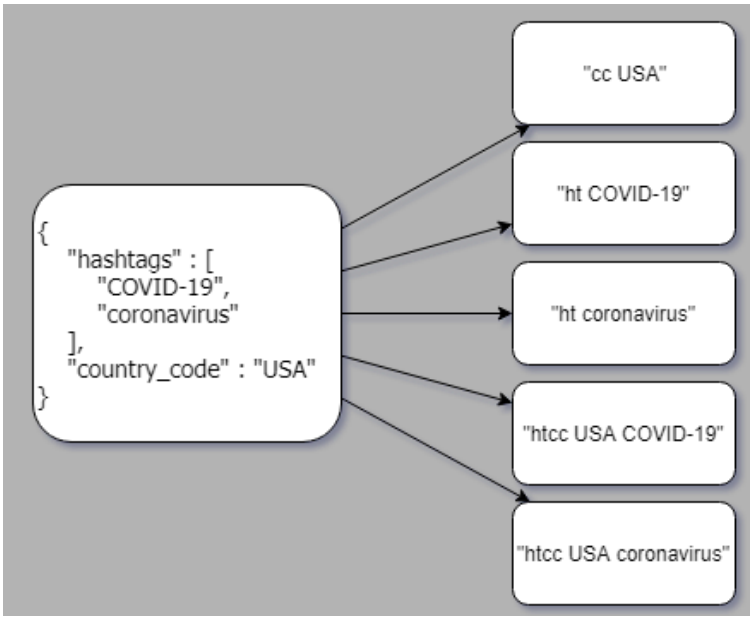


Figura 56 - Ejemplo de la función preparar_tweets del clúster EMR

La siguiente función que se define en el script es 'sendCount'. Esta función toma como parámetro de entrada una partición de Hadoop y, por cada registro, envía un mensaje al topic 'resultados' del clúster de Kafka. Como podemos ver en la Figura 57, el primer paso es crear el objeto 'KafkaClient' al que le pasamos la dirección del host de Kafka (10.0.4.192:9092). A continuación, especificamos el topic al que queremos enviar los conteos (resultados). Por último, creamos un productor para el topic y enviamos cada registro de salida después de serializarlos con la función 'json.dumps'. Es importante que la creación del productor esté fuera del bucle ya que la función 'get_producer' puede tardar varios segundos en ejecutarse y de estar dentro del bucle se ralentizaría bastante el proceso.

```
def sendCount(partition):
    cliente = KafkaClient(hosts = "10.0.4.192:9092")
    topic = cliente.topics['resultados']
    with topic.get_producer() as producer:
        for count in partition:
            producer.produce(bytes(json.dumps(count), encoding = 'utf-8'))
```

Figura 57 - Función sendCount del clúster EMR

Para terminar, vamos a explicar el resto del código del script, que aparece en la Figura 58. En primer lugar, recoge los parámetros de entrada que son la dirección de zookeeper del clúster de Kafka ('zkQuorum' en el código) y el nombre del topic para los datos de entrada. En la siguiente línea inicializamos la variable 'SparkContext' (sus funcionalidades se han explicado anteriormente) pasándole como argumento el nombre de la aplicación ('KafkaTweetsCount'). Después establecemos el nivel de los logs a "WARN" para tareas de debuggeo e inicializamos el objeto 'StreamingContext' pasándole como parámetros el objeto 'SparkContext' y los segundos de los microbatches (cinco segundos). A continuación, creamos el stream de datos de entrada utilizando la librería 'KafkaUtils' y pasándole como argumentos el objeto 'StreamingContext', la dirección de ZooKeeper, el nombre del stream ("spark-streaming-consumer") y la partición del topic (solo hay una). A partir del stream de kafka, utilizamos la función 'window' para establecer

el tamaño y la frecuencia de la ventana de procesamiento (cada diez minutos analizamos la última hora de tweets). Después aplicamos en un 'flatMap' la función 'preparar_tweets' que hemos explicado antes y a cada registro de salida le aplicamos una función 'map' que crea un par clave-valor con el texto del registro como clave y 1 como valor. Para acabar con los cálculos aplicamos una función 'reduceByKey' que sume los valores de todos los registros de una clave. Empleamos la función 'foreachRDD' para ordenar de mayor a menor (utilizando el valor, no la clave) los resultados del conteo y aplicamos la función 'sendCount' con cada partición del RDD para enviar los resultados el clúster de Kafka. Por último, utilizamos las funciones 'start' y 'awaitTermination' para comenzar el 'StreamContext'.

```
zkQuorum, topic = sys.argv[1:]
sc = SparkContext(appName = "KafkaTweetsCount")
sc.setLogLevel("WARN")
ssc = StreamingContext(sc, 5)
ssc.checkpoint('checkpoint')
kvs = KafkaUtils.createStream(ssc, zkQuorum, "spark-streaming-consumer", {topic: 1})
words = kvs.window(3600, 600) \
            .flatMap(lambda x: preparar_tweets(x[1])) \
            .map(lambda a: (a, 1)) \
            .reduceByKey(lambda a, b: a+b)

words.pprint()

words.foreachRDD(lambda rdd: rdd.sortBy(lambda x: x[1], ascending=False).foreachPartition(sendCount))

ssc.start()
ssc.awaitTermination()
```

Figura 58 - Inicialización del programa de Spark

4.9 Creación de la web

La última sección de la implementación es la creación de la web. Para ello hemos utilizado el framework para desarrollo de aplicaciones web Angular. Como se explicó en el Capítulo 2, para el mapa de la aplicación utilizaremos el proveedor de mapa online Mapbox.

Cuando creamos un nuevo proyecto de Angular mediante el comando 'ng new nombre_del_proyecto' se nos genera una carpeta con una gran cantidad de archivos diferentes. Para el desarrollo de nuestra web nos vamos a centrar en los siguientes (el contenido de estos archivos se encuentra en el apéndice 5):

- En la carpeta 'src' encontramos el archivo 'index.html', que contiene el código HTML de la página que se cargará en el equipo del usuario. Sin embargo, si nos fijamos en el cuerpo, únicamente hay un componente llamado 'app-root'. Este componente contendrá toda la funcionalidad de nuestra aplicación.
- En esta carpeta encontramos el archivo 'styles.css' que contiene los estilos generales de la aplicación web.
- En la carpeta 'app' tenemos el archivo 'app.module.ts' que contiene las importaciones de algunos de los componentes que vamos a necesitar. Entre estos encontramos los objetos MatSliderModule (para crear un slider con el que podemos cambiar la hora de los datos que se muestran en el mapa) y el objeto MatTableModule (para crear la tabla con el top 10 de hashtags).
- En esta misma carpeta está el archivo 'app.component.html' que contiene la disposición de los diferentes componentes de la aplicación. Como vemos en el código está compuesto por dos 'div' principales, uno contiene los componentes que se mostrará a la izquierda y el otro los que se mostrará a la derecha. Los componentes que se

mostrarán a la izquierda son tres: el mapa, la leyenda (que está preparada para que los valores cambien según la posición del slider) y el slider. En el 'div' de la derecha tenemos un único componente que es la tabla.

- También tenemos el archivo 'app.component.css' que contiene los estilos de los componentes de la aplicación.
- El último componente es el más importante y el más complejo. Se trata del archivo 'app.component.ts' que contiene la lógica de la aplicación web en lenguaje typescript. Las funciones más importantes de este archivo son:
    - getResultados: Se encarga de descargar los resultados del Bucket de S3 (necesario haber activado los CORS para el Bucket S3 antes).
    - creaLayer: Se encarga de crear las capas del mapa de Mapbox. La forma de colorear los mapas de Mapbox es a través capas. Esta función crea diez capas (una por cada color de la leyenda) y aplica un filtro para que no se muestre ninguna capa en ningún país (más tarde añadiremos países a cada filtro para que muestren). Además de esto, se ocupa de que cuando el usuario haga clic sobre un país se muestre una ventana con sus datos (bandera, capital, población, etc) y el contenido de la tabla (que hasta ahora mostraba los hashtags más repetido a nivel mundial) muestre los hashtags más repetidos para ese país.
    - onInputChange: Es la función que se encarga de cambiar los colores de los países del mapa, la leyenda y el contenido de la tabla en función de la posición del slider. Si no existe información para la hora que marca el slider (por algún fallo del clúster), la función dejará el mapa sin colorear y la tabla vacía.
    - ngOnInit: Es la función que se ejecuta cuando se carga la página y se ocupa de inicializar el mapa de Mapbox y llamar a las funciones anteriores. También es donde se definen los colores de la leyenda.

Podemos ver un ejemplo de cómo se mostrará la web en funcionamiento en la Figura 59 - Web de la aplicación

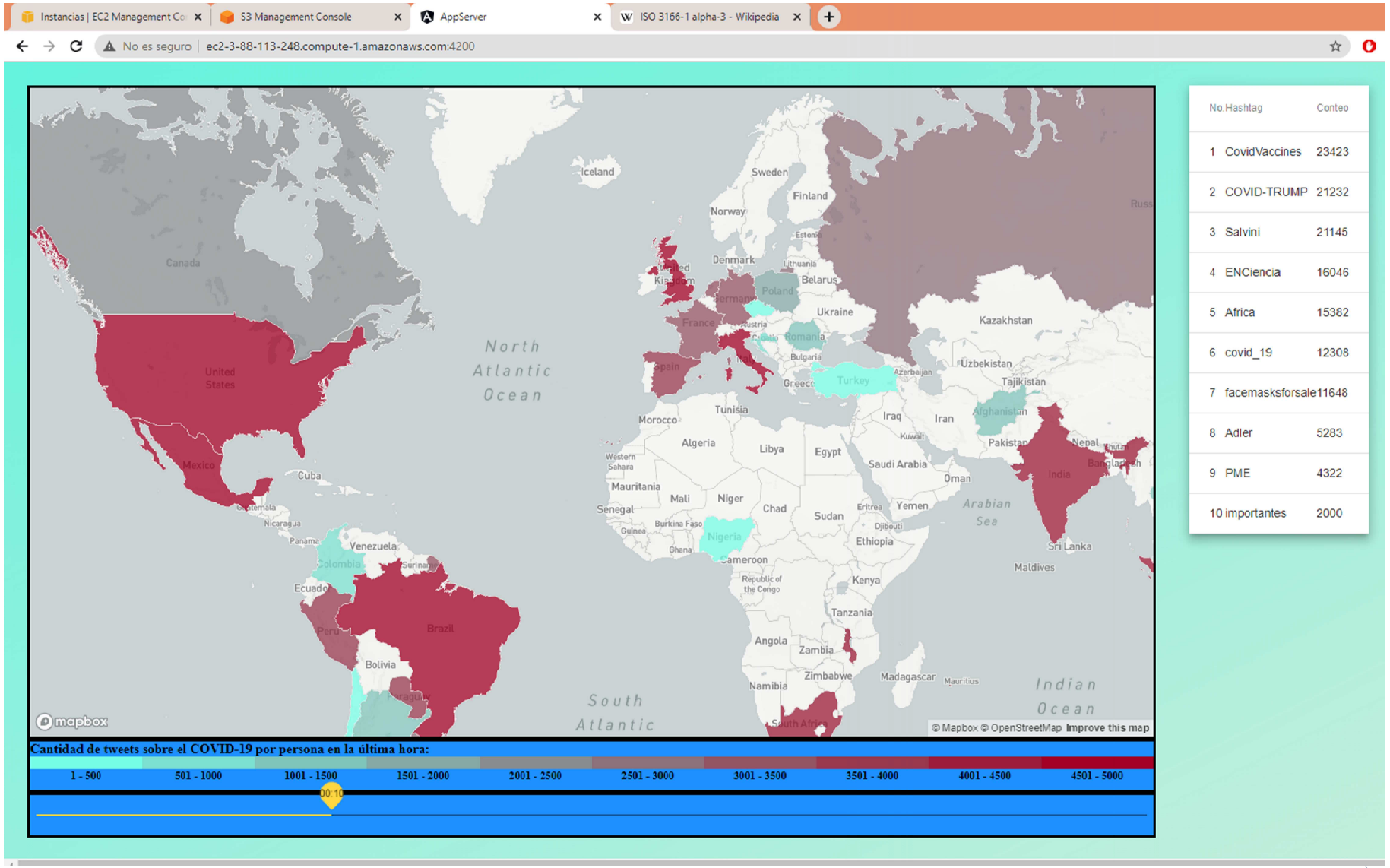

Instancias | EC2 Management Co
S3 Management Console
AppServer
ISO 3166-1 alpha-3 - Wikipedia
No es seguro | ec2-3-88-113-248.compute-1.amazonaws.com:4200
North Atlantic Ocean
South Atlantic
Indian Ocean
Arabian Sea
Canada
United States
Mexico
Cuba
Venezuela
Brazil
Bolivia
Peru
Ecuador
Colombia
Iceland
Sweden
Finland
Norway
Denmark
Poland
Belarus
Ukraine
France
Spain
Germany
Romania
Bulgaria
Greece
Turkey
Kazakhstan
Uzbekistan
Tajikistan
Afghanistan
Iran
Iraq
Pakistan
India
Nepal
Morocco
Algeria
Libya
Egypt
Tunisia
Saudi Arabia
Oman
Yemen
Sudan
Chad
Niger
Mali
Mauritania
Senegal
Nigeria
Ethiopia
Kenya
Tanzania
Angola
Zambia
Zimbabwe
Namibia
Madagascar
Sri Lanka
Maldives
mapbox
© Mapbox © OpenStreetMap Improve this map
Cantidad de tweets sobre el COVID-19 por persona en la última hora:
1 - 500
501 - 1000
1001 - 1500
1501 - 2000
2001 - 2500
2501 - 3000
3001 - 3500
3501 - 4000
4001 - 4500
4501 - 5000
00:10
No.Hashtag Conteo
1 CovidVaccines 23423
2 COVID-TRUMP 21232
3 Salvini 21145
4 ENCiencia 16046
5 Africa 15382
6 covid_19 12308
7 facemasksforsale 11648
8 Adler 5283
9 PME 4322
10 importantes 2000

Figura 59 - Web de la aplicación

Una vez hemos creado el proyecto y lo hemos almacenado en el directorio '/web' del Bucket S3 creamos el grupo de auto-escalado que contendrá las instancias que servirán la web. Lo primero es crear el grupo de seguridad con las reglas de entrada y salida del grupo. Creamos un grupo de seguridad llamado 'web-sg' que deje entrar el tráfico SSH por el puerto 22 y permita conexiones TCP por el puerto 4200 (el puerto por defecto de Angular). Podemos ver las reglas de entrada en la Figura 60.

Reglas de entrada | Editar r

Tipo	Protocolo	Intervalo de puertos	Origen
SSH	TCP	22	0.0.0.0/0
TCP personalizado	TCP	4200	0.0.0.0/0
TCP personalizado	TCP	4200	::/0

Figura 60 - Reglas de entrada del grupo de seguridad de la web

El balanceador de carga debe crearse antes que el grupo de auto-escalado. Para crearlo seleccionamos 'Crear balanceador de carga' en la pestaña 'Balanceadores de carga' del servicio de EC2. Debemos seleccionar el tipo de balanceador que queremos (los diferentes tipos y las funcionalidades de cada uno se explican en el trabajo de investigación complementario), seleccionamos el tipo aplicación. Para configurar el balanceador debemos seguir siete pasos:

- Definir el balanceador: En este paso debemos darle un nombre ('balanceador-aplicacion') y seleccionar la VPC y las subredes en las que se desplegará ('VPC-aplicacion' y subredes 'public-a' y 'public-b'). Es importante que el balanceador se despliegue en subredes en diferentes zonas de disponibilidad porque si no perdemos la ventaja de desplegar el grupo de auto-escalado también en varias zonas. También debemos especificar que el tráfico que llegue al puerto 80 (HTTP) del balanceador se redirija al puerto 4200 (Angular) del grupo de auto-escalado.
- Asignar el grupo de seguridad: Aquí creamos un grupo de seguridad que permita el acceso al puerto 80 desde cualquier dirección IP.
- Configurar la seguridad: Aquí podemos configurar la seguridad del balanceador. Por estar utilizando la cuenta gratuita de AWS Educate no podemos modificar algunas opciones, por lo que las dejaremos con sus valores por defecto.
- Configurar el 'Health Check': En este paso debemos configurar las comprobaciones de estado del balanceador de carga hacia las máquinas del grupo. Realizará un ping cada 30 segundos a cada una de las máquinas para comprobar si funcionan correctamente.
- Añadir instancias de EC2: No añadiremos ninguna ya que lo asignaremos al grupo de auto-escalado.
- Añadir etiquetas: No añadiremos ninguna.
- Revisar: Revisamos que la configuración es correcta.

En la Figura 61 podemos ver el panel de balanceadores de carga del servicio de EC2 con el nuevo creado.

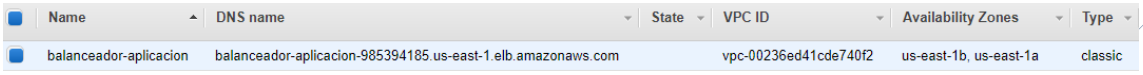


Figura 61 - Balanceador de carga

Antes de crear el grupo tenemos que crear la plantilla de lanzamiento con que se lanzarán las nuevas instancias. Para ello seleccionamos la pestaña 'Plantillas de lanzamiento' ene l panel del servicio de EC2 y seleccionamos 'Crear plantilla de lanzamiento'. Le damos el nombre 'plantilla-web' y seleccionamos la AMI 'Ubuntu Server 20.04 LTS'. Para el tipo de instancia seleccionamos la 't2.small' (ya que la 't2.micro' podría quedarse corta) y si vemos que no es suficiente siempre puede modificarse la plantilla para que las nuevas instancias se lancen con otra configuración. No es necesario proporcionar a estas instancias un par de claves ya que se deben lanzar automáticamente, sin embargo, le vamos a pasar nuestro par de claves 'kp-aplicacion' para depurar posibles errores durante el desarrollo.

Dejamos el resto de las opciones de la plantilla con su valor por defecto excepto el último, los datos de usuario. Aquí deberemos introducir los comandos que deberán ejecutarse cuando se lance una nueva máquina. Los comandos son los siguientes:

- sudo apt update; sudo apt upgrade . (Con estos comandos nos aseguramos de que la lista de paquetes disponible esté actualizada. Para este y para algunos de los siguientes comandos será necesario que la instancia tenga acceso a internet, por lo que la tabla de rutas de las redes debe apuntar al NAT Gateway).
- sudo apt-get install -y nodejs python3-pip (Con este comando instalamos Nodejs y pip. Nodejs es necesario para instalar Angular y pip para instalar el cliente de línea de comandos de AWS para descargar el proyecto de Angular del Bucket S3. Es importante la opción '-y' para que no se nos pida confirmación para la instalación, ya que las máquinas se lanzarán automáticamente sin supervisión).
- sudo npm install npm@latest -g; sudo npm install @angular/cli (Con estos comandos instalamos el gestor de librerías de Nodejs y el cliente de Angular).
- pip3 install awscli (Para instalar el cliente de línea de comandos de AWS).
- mkdir app-server (Antes de descargar el proyecto de Angular del Bucket S3 hay que crear el directorio que lo contendrá).
- aws s3 cp s3://bucket-upna-aplicacion/web app-server –recursive (copiamos el proyecto de Angular del Bucket de S3 a la nueva carpeta. La opción '--recursive' es necesaria porque estamos copiando una carpeta que contiene varias carpetas. Este proceso puede tardar vario minutos).
- cd app-server; ng serve –host=0.0.0.0 –disable-host-check (El comando que lanza el proyecto de Angular).

Para crear el grupo de auto-escalado vamos a la pestaña 'Grupos de auto-escalado' del panel del servicio de EC2 y seleccionamos 'Crear grupo de auto-escalado'. El proceso de creación del grupo está dividido en siete pasos:

- Elegir la plantilla de lanzamiento o la configuración: Aquí deberemos pasarle la plantilla de lanzamiento creada anteriormente y darle un nombre al grupo ('grupo-web').
- Configurar ajustes: Aquí tenemos que seleccionar la VPC y las subredes en las que se lanzarán las instancias. Seleccionamos la VPC 'VPC-aplicacion' y las subredes 'web-a' y 'web-b' (que se encuentran en diferentes zonas de disponibilidad, con lo que ganaremos disponibilidad y tolerancia a fallos).

- Configuración de las opciones avanzadas: En este paso deberemos asignar el balanceador de carga que hemos creado anteriormente (o crearlo si no lo hubiésemos hecho) y especificar que las comprobaciones de estado las realizará el balanceador.
- Configurar políticas de escalado y tamaño de grupo: Seleccionamos que la cantidad deseada de instancias en el grupo debe ser 2, siempre entre un mínimo de 2 máquinas y un máximo de 4. También seleccionamos que la política de escalado de máquinas se guíe según la utilización de CPU promedio de las máquinas (si esta supera el 80%, se lanza una nueva máquina).
- Añadir notificación: Podemos configurar que se nos envíe una notificación si, por ejemplo, la cantidad de instancias llega al máximo.
- Añadir etiquetas: No añadimos ninguna etiqueta.
- Revisar: Revisamos que la configuración es correcta.

En la Figura 62 podemos ver el panel de grupos de auto-escalado del servicio de EC2 con el nuevo creado.

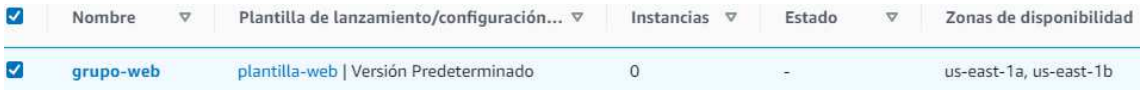


Figura 62 - Grupo de auto-escalado

Capítulo 5 – Experimentos

En este capítulo vamos a realizar un experimento para comprobar que la aplicación funciona correctamente y para establecer los tipos finales de las instancias de EC2 de los componentes. Para ello tendremos el sistema en marcha durante 24 horas y monitorizaremos el rendimiento de cada componente utilizando CloudWatch. A partir de las gráficas de CloudWatch podremos determinar si una instancia de EC2 tiende a perder más créditos que a ganarlos (ver el experimento del capítulo 1.4.6.1 del trabajo de investigación complementario) y reestablecer su tamaño en consecuencia.

El principal problema que tenemos son las limitaciones de la cuenta gratuita de la API de Twitter. Como explicamos en la sección 1.2.1, la API de Twitter cuenta con un punto de acceso gratuito e ilimitado a un flujo de tweets mundiales sobre el COVID-19. Por desgracia es necesario solicitar el acceso a este punto y Twitter ya no acepta solicitudes. Por suerte un laboratorio de la Universidad Estatal de Georgia (Georgia State University) se encarga de publicar diariamente una Dataset con todos los tweets recogidos de este punto de acceso en un repositorio público de GitHub (https://github.com/thepanacealab/covid19_twitter).

La ventaja de haber utilizado Apache Kafka para la ingesta de datos del clúster EMR es que podemos sustituir el productor original (que recogía tweets directamente de la API de Twitter) por un nuevo productor de pruebas que, a partir uno de los datasets anteriores, simule durante 24 hora que está enviando los tweets en tiempo real. Para el clúster de EMR esto será transparente, es más, podríamos tener los dos productores o más funcionando al mismo tiempo y seguiría siendo transparente.

Este experimento servirá también para hacer una estimación del precio que supondría tener la aplicación en funcionamiento mensualmente. Por estar utilizando la cuenta gratuita de AWS Educate no tenemos acceso a la facturación, es decir, que no podremos ver el gasto de cada componente en particular. Sin embargo, podremos comparar el saldo total del usuario antes y después del experimento.

Para realizar la prueba hemos seleccionado el Dataset del repositorio llamado “2021-01-11-dataset.tsv”, que contiene todos los tweets recogidos entre las 5:00 del once de Enero de 2021 y las 5:00 del día siguiente. Como vemos en la Figura 63 el Dataset cuenta con 1.882.269 entradas y cada entrada tiene cinco propiedades: el identificador del tweet, la fecha, la hora, el lenguaje en el que está escrito y el código del país (si el tweet no tenía activada esta opción aparece como “NaN”).

	tweet_id	date	time	lang	country_code
0	1348495783029567491	2021-01-11	05:03:43	de	NaN
1	1348495783260250112	2021-01-11	05:03:43	en	NaN
2	1348495784379969539	2021-01-11	05:03:43	th	NaN
3	1348495784745062401	2021-01-11	05:03:43	fr	NaN
4	1348495784778481665	2021-01-11	05:03:43	en	NaN
...	...	...	...	...	...
1882264	1348858232438202368	2021-01-12	05:03:57	es	NaN
1882265	1348858232597475328	2021-01-12	05:03:57	en	NaN
1882266	1348858233604038657	2021-01-12	05:03:58	en	NaN
1882267	1348858233922801669	2021-01-12	05:03:58	en	NaN
1882268	1348858234191298561	2021-01-12	05:03:58	en	NaN

Figura 63 - Dataset de pruebas

De las propiedades de los tweets no quedaremos tan solo con la hora y código del país. Este Dataset no incluye los hashtags asociado a cada tweet, podríamos hacer una petición a la API de Twitter por cada tweet ya que tenemos su identificador, pero seguimos teniendo las limitaciones de la cuenta gratuita de la API. En lugar de esto, asumiremos que cada tweet tiene de media un hashtag y lo enviaremos al clúster de EMR con un hashtag aleatorio y su código de país (si tiene) a su hora.

En el Apéndice 5 podemos ver el código que se ha utilizado para el productor en la prueba. En primer lugar, tenemos las importaciones, que vienen a ser las misma que en el productor original, pero sin importar la librería 'tweepy' (porque no nos conectaremos a la API de Twitter) e importando en su lugar las librerías 'datetime' y 'time' (para operar con las horas de los tweets). Después de las importaciones leemos el Dataset, eliminamos las columnas que no vamos a necesitar (identificador, fecha e idioma) e inicializamos las variables necesarias (los hashtags de prueba, el productor de Kafka y la variable 'old_time', que contiene la hora del último tweet procesado).

Para cada tweet del Dataset ejecutamos el código del código de la Figura 64. En primer lugar, creamos dos variables del tipo 'datetime', la primera con la hora del último tweet enviado y la segunda con la hora del actual y esperamos la cantidad de segundos que hay entre uno y otro (de esta forma simulamos que se envían en tiempo real). El siguiente paso es enviar el tweet al clúster de Kafka con su código de país (si tiene) y uno de los diez hashtags elegido de forma aleatoria. Por último, actualizamos la variable 'old_time' con la hora del tweet.

```
while True:
  row = df.iloc[i]
  a = dt.datetime(2021,1,15,int(old_time.split(":")[0]),int(old_time.split(":")[1]),int(old_time.split(":")[2]))
  b = dt.datetime(2021,1,15,int(row.time.split(":")[0]),int(row.time.split(":")[1]),int(row.time.split(":")[2]))
  time.sleep((b-a).seconds)

  kafka_msg = {}
  if type(row.country_code) is not float:
    kafka_msg['country_code'] = countries.get(row.country_code).alpha3
  kafka_msg['hashtags'] = [hashtags[random.randint(0, 9)]]
  producer.send('tweets', json.dumps(kafka_msg).encode())

  old_time = row.time
  i += 1
```

Figura 64 - Tratamiento tweets de pruebas

Después de haber tenido toda la arquitectura en funcionamiento durante 24 horas y haber consumido todos los tweets del Dataset es el momento de utilizar CloudWatch para visualizar las gráficas con las métricas de los componentes. El primer paso es crear un panel en el servicio de CloudWatch de la consola de AWS seleccionando la opción 'Crear panel'. Un panel es un conjunto de vistas de gráficas de métricas de diferentes componentes. Para añadir una de estas vistas de, por ejemplo, una instancia de EC2 debemos añadirlas directamente desde el servicio de EC2 en la pestaña 'monitorización' de la instancia (las instancias de EC2 del clúster EMR aparecen también en el servicio EC2, las métricas que aparecen en el servicio EMR son más generales). Una vez hemos añadido todas las métricas podemos ver un panel de CloudWatch como el la Figura 65.

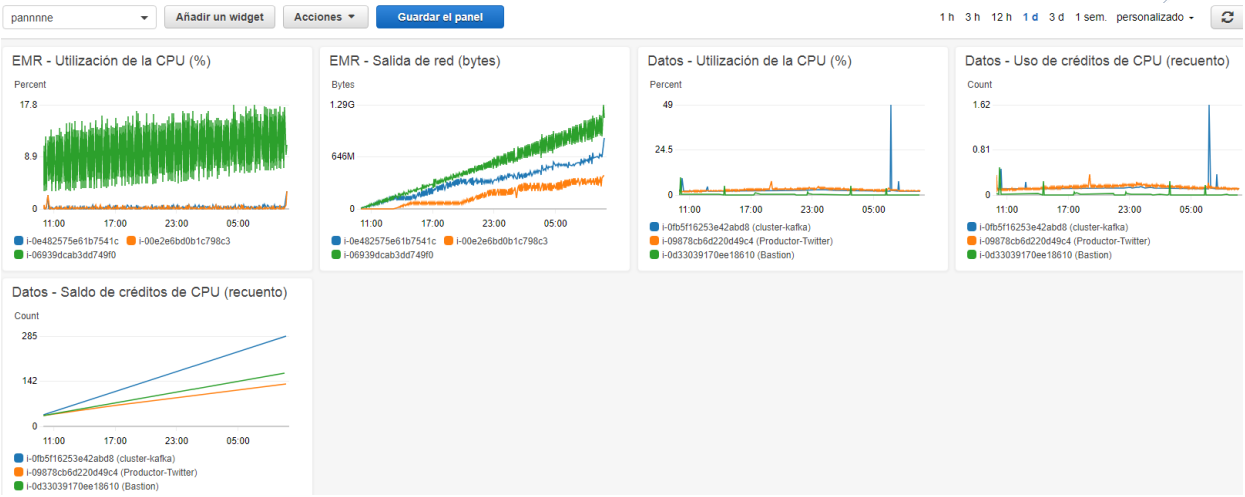


Figura 65 - Panel de AWS CloudWatch

En la Figura 66 podemos ver la utilización de CPU de las instancias de los nodos del clúster EMR, donde la línea verde corresponde al nodo maestro y la naranja y la azul a los nodos esclavos. La línea del nodo maestro oscila constantemente entre un número alrededor del 16% y otro alrededor del 4%. Esto se debe a que, por defecto, CloudWatch toma medidas de las métricas cada cinco minutos y el código de Spark Streaming está escrito para que la ventana de operaciones se realice cada diez minutos. Las líneas de los nodos esclavos se mantienen en todo momento entre el 0.5% y el 1% de utilización de CPU. Esto se debe a que la carga de tweets que tiene que analizar el clúster en esos 5 minutos es de alrededor de 6.500 de media (1.882.269 el día), que son uno 22 tweets por segundo. Estamos utilizando instancias m5.xlarge que están preparadas para una carga de trabajo mayor.

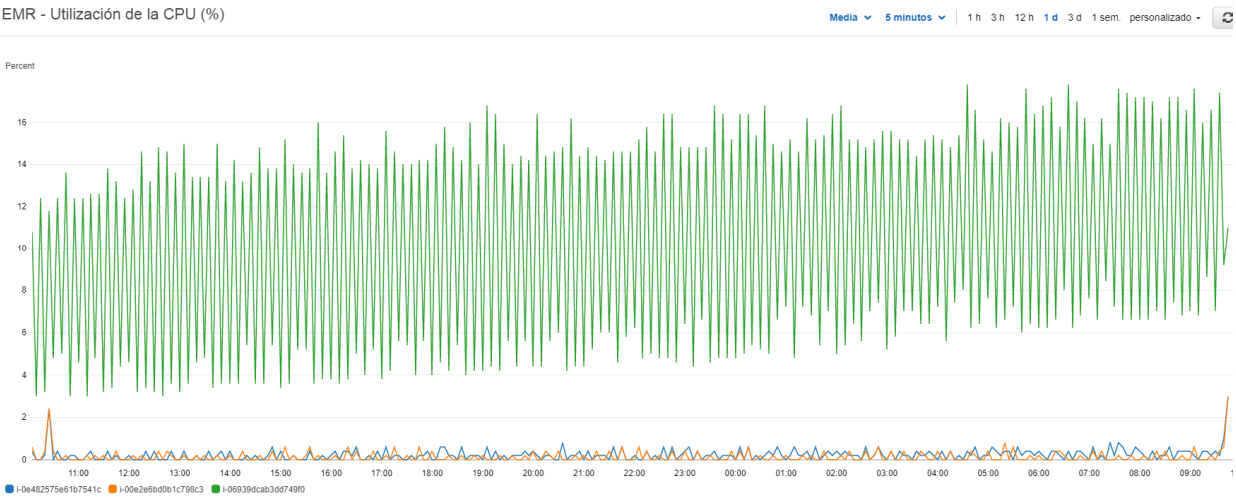


Figura 66 - Utilización de CPU del clúster EMR

En la Figura 67 podemos ver la utilización de CPU de los otros componentes de la aplicación, la línea naranja corresponde al productor de Twitter, la azul al clúster de Apache Kafka y la verde a la máquina bastión. La utilización de CPU del productor y el clúster de Kafka se mantiene entre el 2% y el 5%, pero la línea del productor hace picos cada cinco minutos igual que el nodo central del clúster EMR. Esto se debe a que, además de enviar los datos al clúster, cada diez minutos recibe los resultados que debe almacenar en el Bucket S3. La utilización de la

máquina bastión está siempre cercana al 0% como cabía esperar. A las 6:30 la utilización del clúster de Kafka tiene un pico que llega al 40%, esto puede deberse a una rutina de limpieza de datos por parte de ZooKeeper.

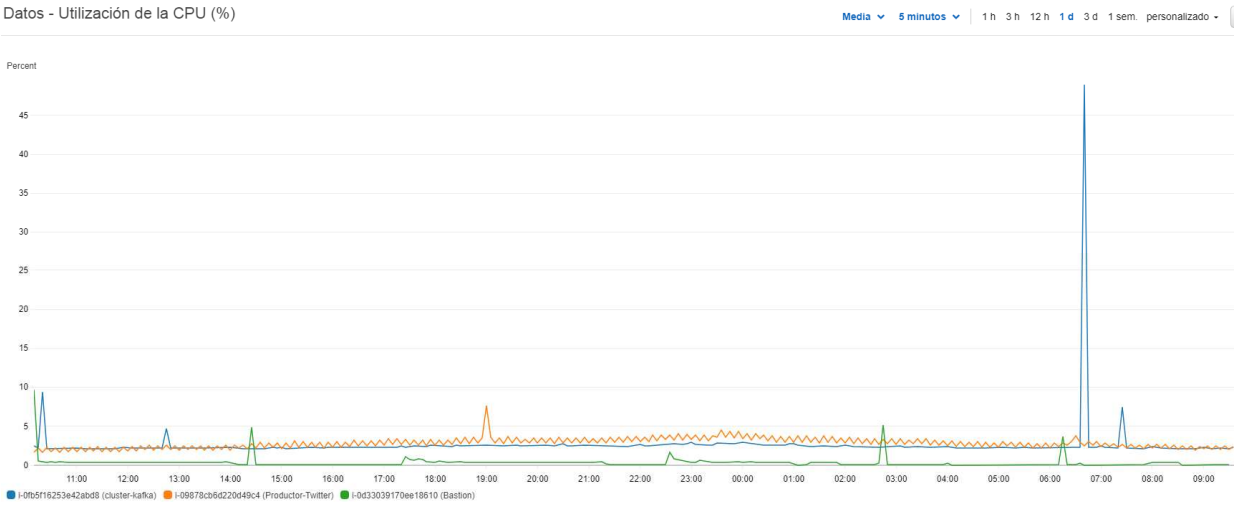


Figura 67 - Utilización de CPU de los componentes

En la Figura 68 podemos ver el uso de créditos de CPU de estos tres componentes y la gráfica es prácticamente la misma que la anterior, esto es porque el uso de créditos depende únicamente de la utilización de CPU. El uso de créditos está en todo momento entre 0.1 y 0.2 que es un ritmo de pérdida de créditos menor al ritmo de ganancia de estos, que es de seis créditos la hora en el caso de la máquina bastión y el productor y 12 en el caso del clúster de Kafka.

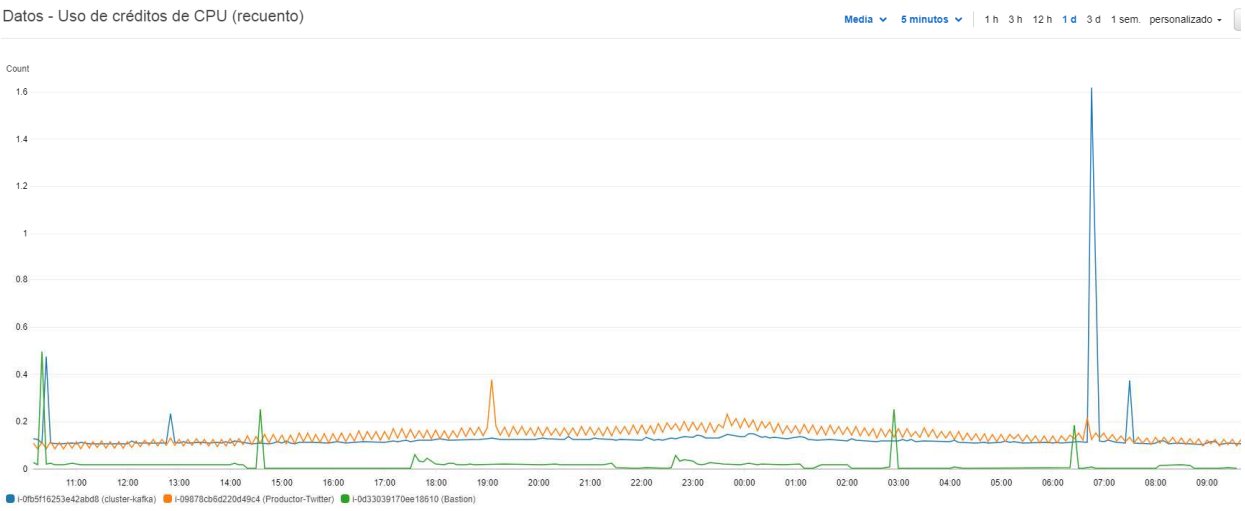


Figura 68 - Uso de créditos de CPU de los componentes

En la Figura 69 podemos ver el saldo de créditos de los componentes anteriores, que es el resultado directo de la gráfica del uso de créditos. Como acabamos de explicar el ritmo de gasto de créditos es menor que el ritmo de ganancia de estos, por lo que la cantidad de estos aumenta en el tiempo. Aunque la ganancia de créditos de la máquina bastión y del productor es la misma, los créditos del primero aumentan a mayor velocidad porque la utilización CPU de este en menor. Si nos fijamos en el momento en el que ocurre el pico de utilización de CPU del

clúster de Kafka (6:30), el saldo de crédito disminuye porque el gasto de estos supera por primera vez a la ganancia.

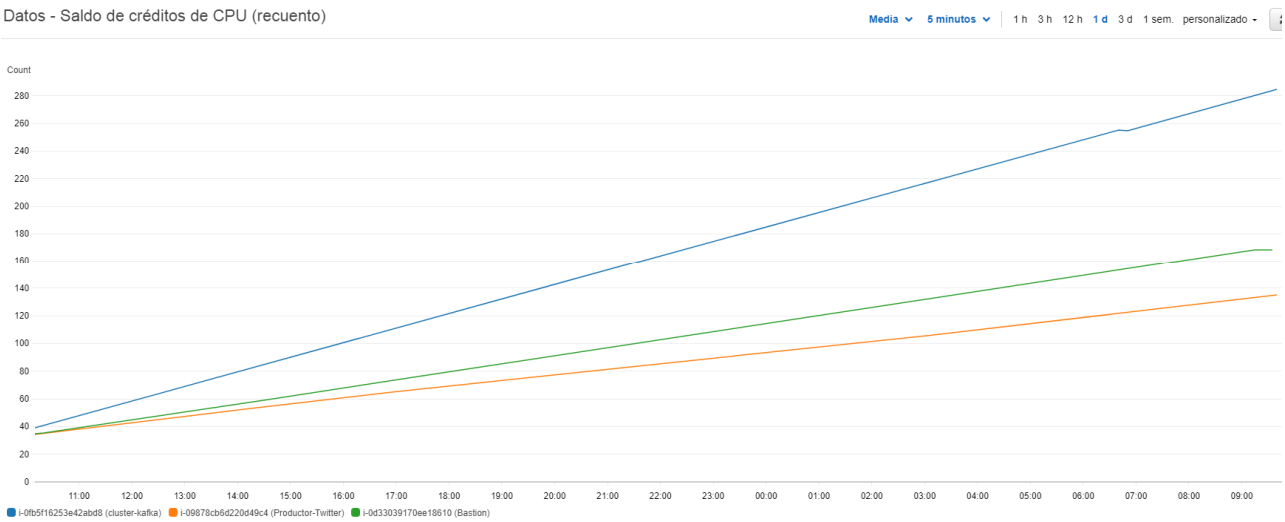


Figura 69 - Saldo de créditos de CPU de los componentes

El saldo de la cuenta de AWS Educate antes de realizar el experimento era de 67.66$ y al terminar este era de 62.67$, esto hace un gasto aproximado de 5$ al día o 150$ al mes.

En la Figura 70 podemos ver una captura en la que se muestra cómo queda la web después del experimento utilizando los datos generados en él. Concretamente, muestra los datos correspondientes a las 6:10 (con los tweets recopilados desde las 5:10). Vemos que la mayor parte de los países de África y de Asia aparecen en blanco, es decir, que no sea recogido ningún tweet correspondiente esos países en la última hora. Esto puede deberse a muchos factores:

- La hora del día. Podría ser que en muchos de estos países todavía fuera de noche.
- El uso de Twitter. Hay países en los que uso de Twitter está mucho más extendido que en otros.
- Tener activada la geolocalización. Hay países en los que hay una mayor cultura de la privacidad y es menos común encontrar tweets con código de país.
- Los hashtags. La selección de hashtags y palabras clave del punto de acceso del cOVID-19 de la API de Twitter puede no ser completo. Hay muchos idiomas que no se ven representados en esa selección.

También nos hemos podido fijar en que, en algunos, en países con una población muy pequeña (como por ejemplo la Isla de Man, con 80.000 habitantes) basta con que aparezca un solo tweet para eclipsar al resto de países, haciendo que el mapa mundial que prácticamente azul con un solo punto rojo. Sería posible arreglar este problema penalizando a los países muy pequeño o, simplemente, eligiendo los umbrales de la leyenda de forma una más inteligente (por ejemplo, intentado igualar el número de países que corresponden a cada color).

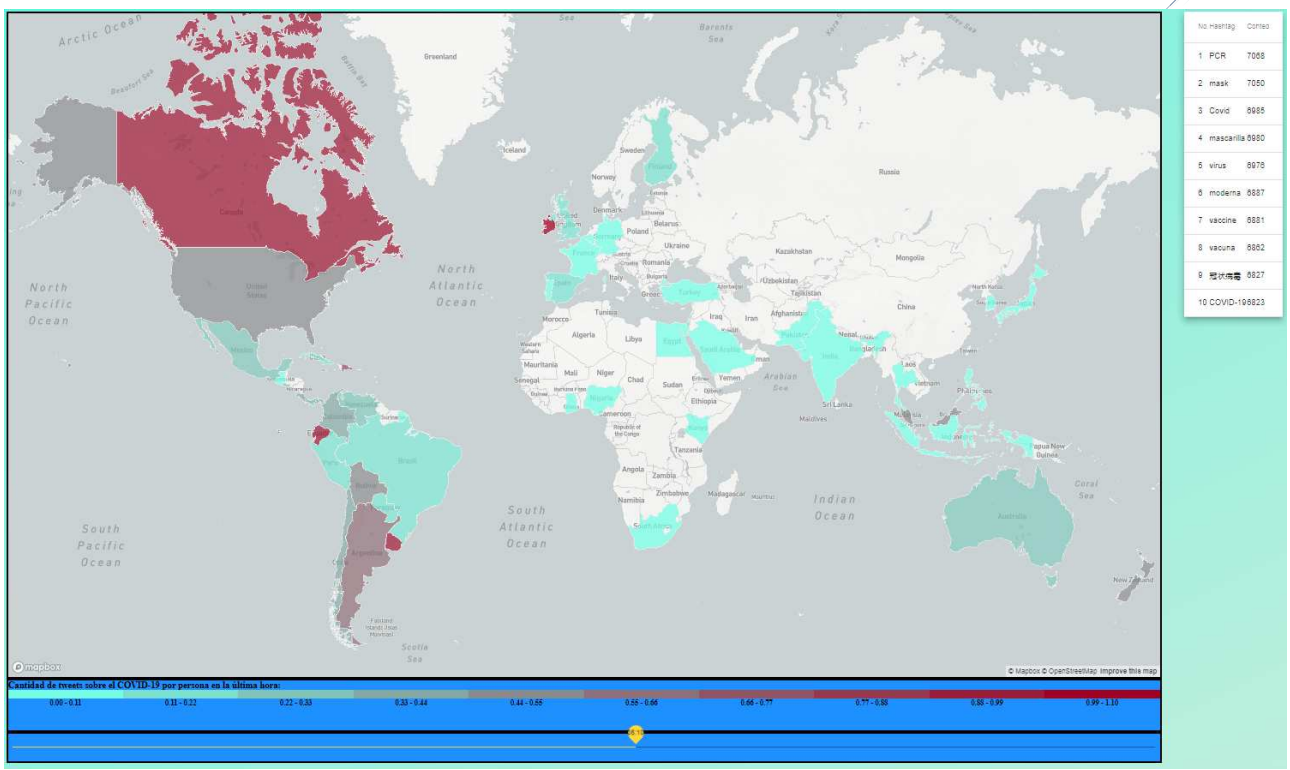


Figura 70 - Resultado de la web después del experimento

Capítulo 6 – Conclusiones y líneas futuras

6.1 Conclusiones

Durante el desarrollo del proyecto hemos podido comprobar que a la hora de implementar una aplicación Big Data no existe una arquitectura de referencia única que seguir. En el capítulo 1.1.1 explicamos algunas de estas arquitecturas y podríamos argumentar como la arquitectura de este proyecto se adapta a cada una de ellas, ya que estas son bastante generales. Si nos fijamos en la arquitectura NIST podemos identificar cada uno de sus componentes con elementos de nuestro proyecto: el proveedor y el consumidor de datos serían Twitter y la web respectivamente, la recolección y la preparación de datos se realizarían en la máquina del productor de Twitter y el clúster de Kafka, el análisis en el clúster EMR, la visualización y el acceso en la web y como proveedores de herramientas de procesamiento, plataforma e infraestructura tendríamos a Spark Streaming, a HDFS y a AWS y sus servicios. También podríamos encajar el proyecto dentro de las arquitecturas Lambda y Kappa teniendo en cuenta que solo tendríamos la capa 'Speed'.

En cuanto a la elección de Twitter como red social para la extracción de los datos podemos afirmar que ha sido una decisión acertada. La naturaleza de la red social hace que sea perfecta para este tipo de aplicaciones no solo por el formato de la información que genera (fragmentos de texto muy pequeños) sino también por el hecho de que cualquier usuario puede acceder a toda esta información ya que los tweets son públicos.

Se ha comprobado que es posible alojar todos los componentes de una aplicación Big Data, desde la extracción y la preparación de los datos hasta la visualización de los resultados, completamente en la nube. Esto se debe a que AWS provee una variedad de servicios muy amplia con el objetivo de que el cliente no tenga que dejar la plataforma a la hora de realizar diferentes tareas (este tema se explica más en profundidad en el trabajo de investigación complementario).

Ha sido posible desarrollar satisfactoriamente la aplicación a pesar de las limitaciones que se han presentado por estar utilizando las cuentas gratuitas de la API de Twitter y de AWS. Las limitaciones de la API de Twitter pudieron suplirse en parte gracias al repositorio publicado en GitHub por la Universidad Estatal de Georgia en el que recogen los tweets diarios sobre el COVID-19. La limitación de la cuenta de AWS Educate por la que no se nos permitía el acceso a algunos servicios como el servicio de clústers de Apache Kafka gestionados (MSK) pudo superarse levantando el clúster manualmente con instancias de EC2. También hubo que ser especialmente cautos a la hora de utilizar cualquier servicio ya que se contaba con un saldo inicial de 100$ que no podía reponerse.

Es necesario poner en contexto la información que se muestra a los usuarios en la web. Para ello hay que dejar claro que la información ha sido sacada exclusivamente de Twitter, esto quiere decir que en los países en donde Twitter es más popular se tenderá a recoger más tweets sobre el COVID-19 (y sobre cualquier tema) y esto no quiere decir que se está hablando más o que haya mayor cantidad de casos de contagiados que otros países. También es necesario tener en cuenta otro factor que es la cultura que tiene un país con respecto a la privacidad en internet. En un país en el que exista mayor reticencia a la hora de activar la geolocalización en las redes sociales encontraremos menor cantidad de tweets sobre cualquier tema. Por último, hay que saber que los tweets que recoge Twitter en su punto de acceso sobre el COVIS-19 corresponden a un conjunto de hashtags y palabras clave elegido por Twitter con el objetivo de cubrir la mayor

parte posible de la conversación sobre el tema, sin embargo, es prácticamente imposible cubrirlo completamente.

6.2 Líneas futuras

Si el proyector se volviera a levantar, esta vez sin limitaciones de fondos, el cambio fundamental sería que se podrían utilizar todos los servicios de AWS que a los que no hemos tenido acceso por utilizar la cuenta de AWS Educate. La instancia de EC2 que hace el papel de clúster de Apache Kafka podría sustituirse por el servicio de clústers de Apache Kafka gestionados (MSK). Después de finalizar el trabajo de investigación complementario a este nos dimos cuenta de que algunas tareas que realizamos a mano se podrían haber sustituido por servicio de AWS. A la hora de almacenar los resultados del procesamiento del clúster EMR en el Bucket de S3 podría haberse utilizado el servicio Kinesis Data Firehose (que se explica en el trabajo complementario) y para alojar la web podría haberse empleado AWS Elastic Beanstalk.

Al estar utilizando Amazon MSK para el clúster de Kafka se podría aprovechar para adaptar todos los componentes de la red de extracción y tratamiento de datos a la alta disponibilidad de AWS utilizando dos o más zonas de disponibilidad (actualmente solo la parte de la web lo aprovecha). Para adaptar la instancia del productor de Twitter a este nuevo modelo una de las prácticas más utilizadas es crear un grupo de auto-escalado en las dos zonas de disponibilidad para alojarla estableciendo el máximo y el mínimo de instancias a uno. De esta forma si se cayera una de las zonas la instancia se levantaría automáticamente en la otra. El clúster de MSK se levantaría por defecto en más de una zona de disponibilidad. Como se explica en el trabajo complementario es posible levantar un clúster de EMR con más de un nodo maestro en diferentes zonas de disponibilidad y los nodos de trabajo también en diferentes zonas. De esta forma tendríamos todos los componentes de la aplicación en alta disponibilidad y con tolerancia a fallos.

Otro de los aspectos a cambiar sería el de poder acceder al punto de acceso de la API de Twitter sobre el COVID-19. En caso de no poder acceder a este sería posible utilizar la API con nuestro propio conjunto de hashtags y palabras clave si estamos dispuestos a afrontar los cargos mensuales por el uso de la API.

Por último, sería interesante explorar la posibilidad de utilizar otras redes sociales además de Twitter para la ingesta de datos. La integración de otra red social al flujo de datos de entrada sería sencilla gracias al clúster de Apache Kafka.

Referencias

[1] CNE, «cnecovid.isciii.es,» [En línea]. Available: cnecovid.isciii.es/covid19/. [Último acceso: 4 Enero 2021].

[2] «es.statista.com,» 17 Abril 2019. [En línea]. Available: es.statista.com/grafico/17734/cantidad-real-y-prevista-de-datos-generados-en-todo-el-mundo/. [Último acceso: 6 Noviembre 2020].

[3] «www.masterbigdataucm.com,» [En línea]. Available: www.masterbigdataucm.com/que-es-big-data/. [Último acceso: 6 Noviembre 2020].

[4] T. Shafer, «www.kdnuggets.com,» Abril 2017. [En línea]. Available: www.kdnuggets.com/2017/04/42-vs-big-data-data-science.html. [Último acceso: 6 Noviembre 2020].

[5] «www.oracle.com,» [En línea]. Available: www.oracle.com/big-data/guide/big-data-use-cases.html. [Último acceso: 7 Novienbre 2020].

[6] «www.bigdataframework.org,» 7 Mayo 2017. [En línea]. Available: www.bigdataframework.org/big-data-architecture/. [Último acceso: 7 Noviembre 2020].

[7] «en.wikipedia.org,» [En línea]. Available: en.wikipedia.org/wiki/DIKW_pyramid. [Último acceso: 7 Noviembre 2020].

[8] W. L. Chang, D. Boyd y O. Levin, «www.nist.gov,» 21 Octubre 2019. [En línea]. Available: nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.1500-6r2.pdf. [Último acceso: 8 Noviembre 2020].

[9] A. Rayón, «blogs.deusto.es,» 2 Septiembre 2016. [En línea]. Available: blogs.deusto.es/bigdata/paradigma-batch-para-sistemas-big-data-i/. [Último acceso: 9 Noviembre 2020].

[10] A. Rayón, «blogs.deusto.es,» 12 Septiembre 2016. [En línea]. Available: blogs.deusto.es/bigdata/paradigma-tiempo-real-para-sistemas-big-data-ii/. [Último acceso: 9 Noviembre 2020].

[11] J. Dominguez, «www.paradigmadigital.com,» 2018. [En línea]. Available: www.paradigmadigital.com/techbiz/de-lambda-a-kappa-evolucion-de-las-arquitecturas-big-data/. [Último acceso: 7 Noviembre 2020].

[12] A. Rayón, «blogs.deusto.e,» 9 Otubre 2019. [En línea]. Available: blogs.deusto.es/bigdata/arquitectura-lambda-para-sistemas-big-data-y-iii/. [Último acceso: 9 Noviembre 2020].

[13] J. Kreps, «www.oreilly.com,» 2 Julio 2014. [En línea]. Available: www.oreilly.com/radar/questioning-the-lambda-architecture/. [Último acceso: 9 Noviembre 2020].

[14] «www.hdfstutorial.com,» [En línea]. Available: www.hdfstutorial.com/blog/hadoop-1-vs-hadoop-2-differences/. [Último acceso: 12 Noviemrbe 2020].

[15] «hadoop.apache.org,» [En línea]. Available: hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. [Último acceso: 11 Noviembre 2020].

[16] A. Rayón, «blogs.deusto.es,» 19 Agosto 2016. [En línea]. Available: blogs.deusto.es/bigdata/cuando-empieza-esta-era-del-big-data-mapreduce/. [Último acceso: 9 Noviembre 2020].

[17] P. R. d. l. Santos, «empresas.blogthinkbig.com,» 10 Octubre 2017. [En línea]. Available: empresas.blogthinkbig.com/cloudera-mapr-hortonworksque/. [Último acceso: 26 Noviembre 2020].

[18] V. Madrid, «enmilocalfunciona.io,» 27 Septiembre 2018. [En línea]. Available: enmilocalfunciona.io/aprendiendo-apache-kafka-parte-1/. [Último acceso: 25 Noviembre 2020].

[19] C. McCabe, «www.confluent.io,» 15 Msyo 2020. [En línea]. Available: www.confluent.io/blog/removing-zookeeper-dependency-in-kafka/. [Último acceso: 25 Noviembre 2020].

[20] «spark.apache.org,» [En línea]. Available: spark.apache.org/docs/latest/streaming-programming-guide.html. [Último acceso: 27 Noviembre 2020].

[21] S. Martín, «mkparadise.com,» 30 Junio 2020. [En línea]. Available: mkparadise.com/redes-sociales-mas-utilizadas. [Último acceso: 23 Noviembre 2020].

[22] I. Cairns y P. Shetty, «blog.twitter.com,» 16 Julio 2020. [En línea]. Available: blog.twitter.com/es_la/topics/product/2020/Presentamos-una-nueva-y-mejorada-API-de-Twitter.html. [Último acceso: 24 Noviembre 2020].

[23] P. Fuló, «www.cloudmasters.es,» 14 Septiembre 2019. [En línea]. Available: www.cloudmasters.es/sabes-quienes-son-las-10-grandes-en-cloud-computing/. [Último acceso: 6 Noviembre 2020].

[24] «axarnet.es,» 18 Diciembre 2018. [En línea]. Available: axarnet.es/blog/nube-privada-publica-hibrida. [Último acceso: 6 Noviembre 2020].

[25] «axarnet.es,» 4 Diciembre 2018. [En línea]. Available: axarnet.es/blog/saas-paas-iaas. [Último acceso: 6 Noviembre 2020].

[26] «proto.io,» [En línea]. Available: proto.io. [Último acceso: 9 Enero 2021].

[27] «developer.twitter.com,» [En línea]. Available: developer.twitter.com/en/docs/twitter-api/data-dictionary/object-model/tweet. [Último acceso: 2 Diciembre 2020].

Apéndices

Apéndice 1 - Conexión SSH a una instancia de EC2

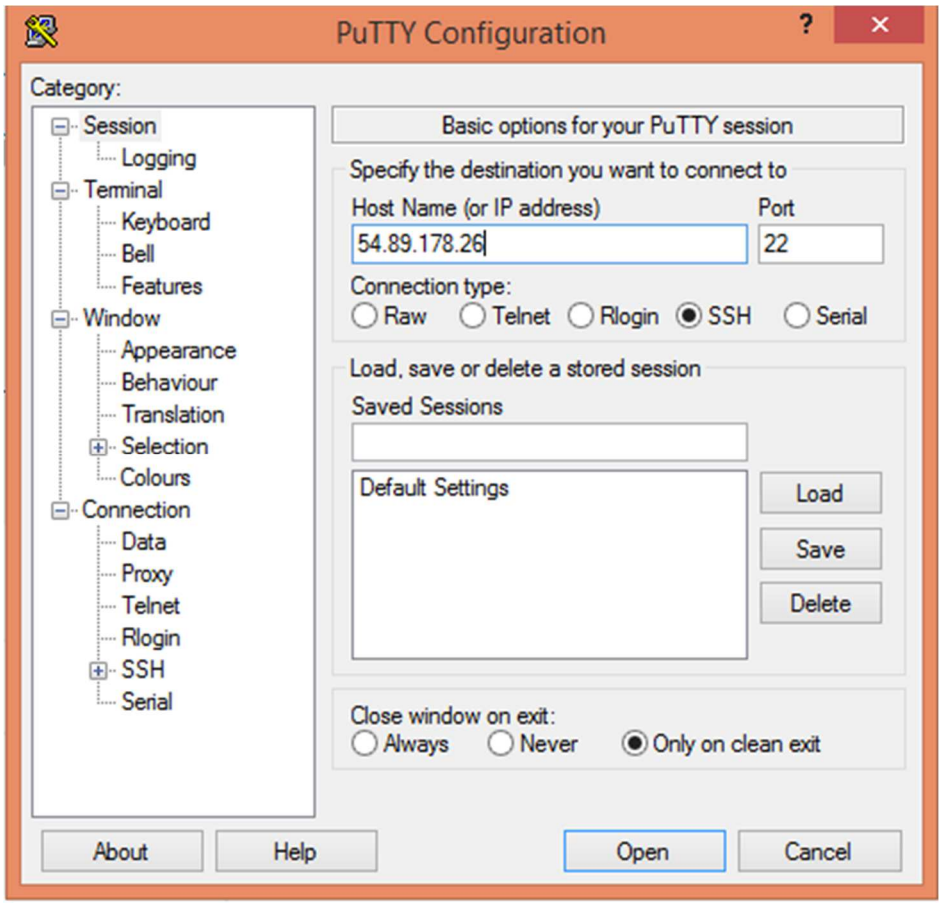
Para conectarse a una instancia EC2 mediante SSH primero hay que asegurarse de que la instancia está en estado Running, de que tiene una dirección IP pública asociada y de que su Security Group permite el acceso en el puerto 22. Si nos quisiéramos conectar desde una terminal Linux sería tan sencillo como escribir el comando:

$ ssh “dirIP” -l ubuntu –i “dirKey”

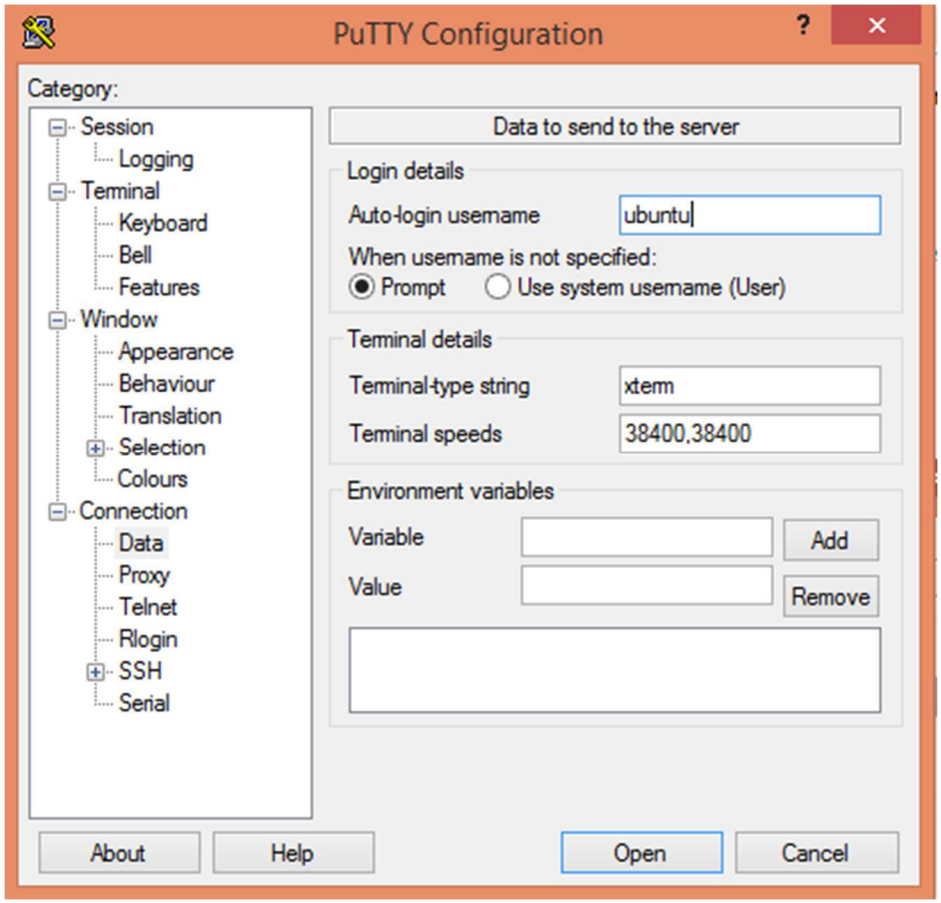
Donde “dirIP” es la dirección IP de la instancia y “dirKey” es la dirección del fichero en el que hemos guardado la clave pública del Key Pair asociado a la instancia EC2.

Si queremos conectarnos desde una terminal de Windows, como es nuestro caso, entonces podemos utilizar el programa PuTTY. PuTTY es un cliente para conexiones SSH y Telnet desarrollado originalmente por Simon Tatham pala Windows. PuTTY es un software ‘open source’ y actualmente su soporte es suministrado por un grupo de voluntarios.

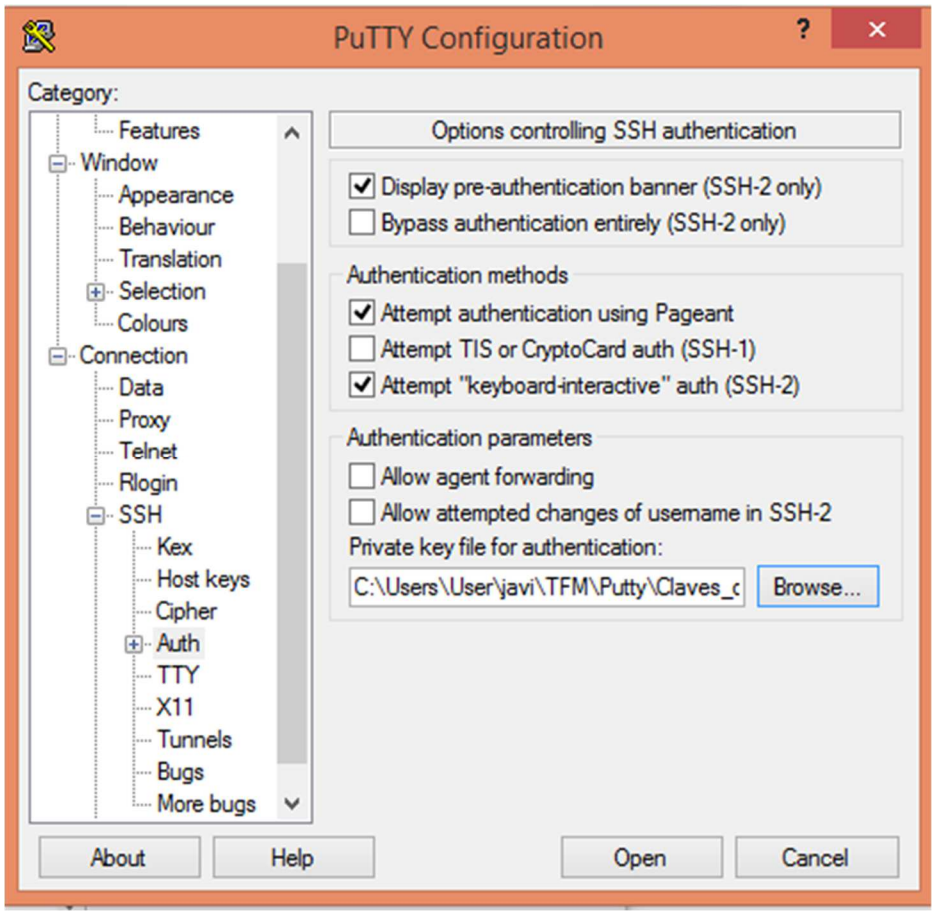
El primer paso es rellenar el campo Host Name (or IP address) con la dirección IP de nuestra instancia EC2 t en el tipo de conexión seleccionar la opción SSH:



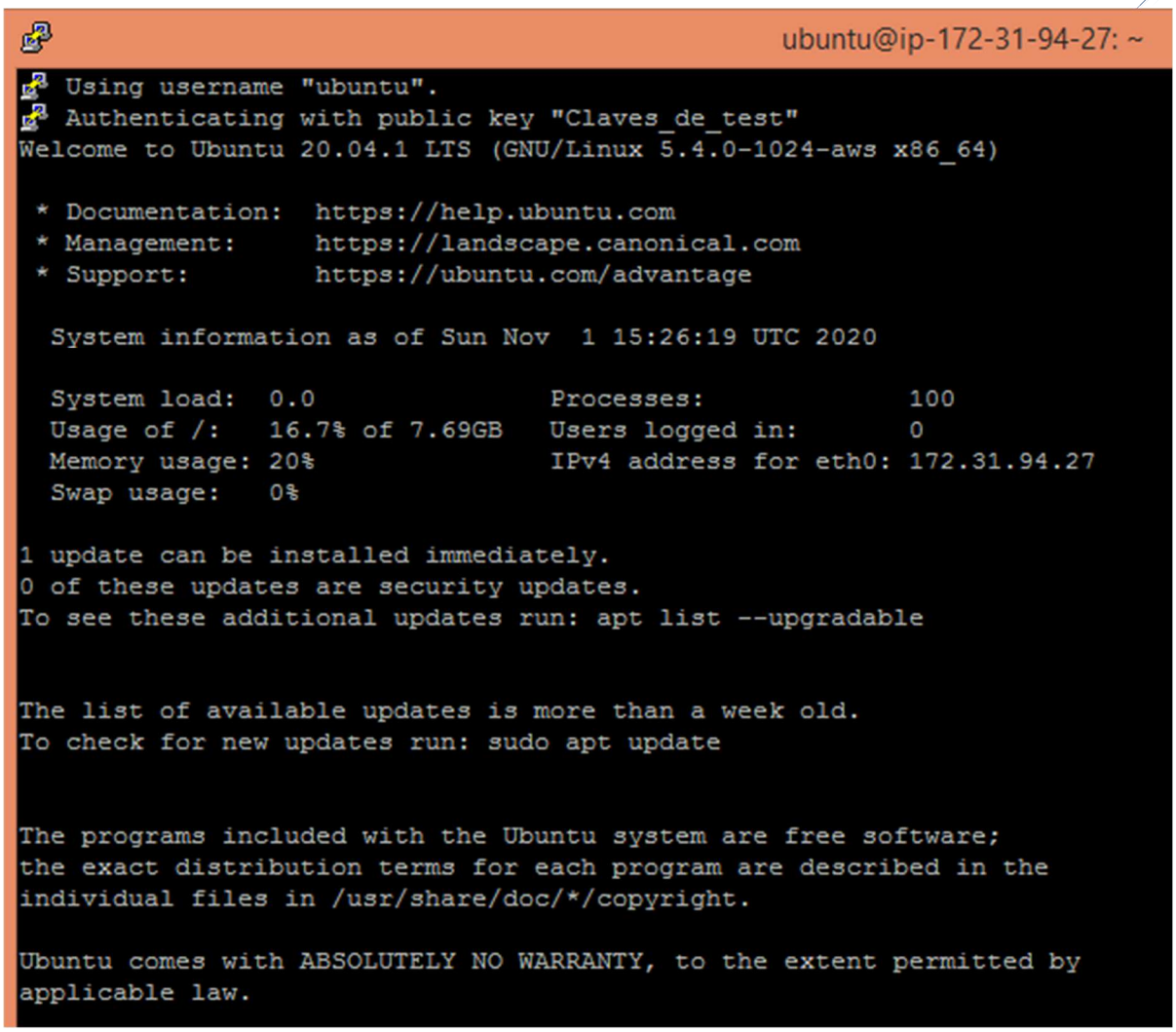
A continuación, deberemos seleccionar en el panel de la izquierda la categoría Connection -> Data y rellanar el campo Auto-login username con ‘ubuntu’. Este será el nombre de usuario por defecto con el debemos iniciar sesión en las instancias EC2 de Ubuntu de AWS (que son las que utilizaremos para ejecutar nuestra aplicación:



Por último, tenemos que seleccionar la categoría Connection -> SSH -> Auth en el panel de la izquierda y seleccionar la dirección del fichero en el que hemos guardado la clave pública del Key Pair asociado a la instancia EC2:



Hecho todo lo anterior, solo queda hacer click en la opción 'Open' y estaremos conectados a la instancia de EC2:



Apéndice 2 – Código del productor de Twitter

El contenido de las variables 'consumer_key', 'consumer_secret', 'access_token' y 'access_secret' ha sido ocultado por motivos de seguridad.

```
import tweepy
from tweepy import OAuthHandler
from tweepy import Stream
from tweepy.streaming import StreamListener
from kafka import KafkaProducer
import socket
import json

# Inicializamos las credenciales
consumer_key = "-------------------------"
consumer_secret = "-----------------------------"
access_token = "-----------------------------"
access_secret = "-----------------------------------------"

class TweetsListener(StreamListener):

  def __init__(self, productor_kafka):
      self.productor = productor_kafka
```

```
  def on_data(self, data):
      try:

          msg = json.loads( data )
          kafka_msg = {}

           # Si el tweet tiene asignado un código de país, se añade al mensaje
          if 'place' in msg:
              if msg['place'] != None:
                  kafka_msg['country_code'] = msg['place']['country_code']

           # Si el tweet tiene hashtags, se añaden al mensaje
          if 'entities' in msg:
              if msg['entities']['hashtags'] != []:
                  kafka_msg['hashtags'] = []
                  for hash in msg['entities']['hashtags']:
                      kafka_msg['hashtags'].append(hash['text'])

          if kafka_msg != {}:
              self.productor.send('tweets', json.dumps(kafka_msg).encode())

          return True

      except BaseException as e:
          print("Error: %s" % str(e))

      return True

  def on_error(self, status):
      print(status)
      return True

def sendData(prod):
  auth = OAuthHandler(consumer_key, consumer_secret)
  auth.set_access_token(access_token, access_secret)

  twitter_stream = Stream(auth, TweetsListener(prod))
  twitter_stream.filter(track=[LISTA DE HASHTAGS])

if __name__ == "__main__":
  print('Conectando al cluster de Kafka...')
  producer = KafkaProducer(bootstrap_servers='10.0.4.192:9092')
  print('Enviando tweets')
  sendData(producer)
```

Apéndice 3 – Código del script de Spark

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
from pykafka import KafkaClient
import json
import pprint

def preparar_tweets(msg):
    tweet = json.loads(msg)
    contenido = []
    if 'country_code' in tweet:
        contenido.append("cc " + tweet['country_code'])
    if 'hashtags' in tweet:
        for hash in tweet['hashtags']:
            contenido.append("ht " + hash)
            if 'country_code' in tweet:
                contenido.append("htcc " + tweet['country_code'] + " " + hash)
```

```
    return contenido

def sendCount(partition):
    cliente = KafkaClient(hosts = "10.0.4.192:9092")
    topic = cliente.topics['resultados']
    with topic.get_producer() as producer:
        for count in partition:
            producer.produce(bytes(json.dumps(count), encoding = 'utf-8'))

zkQuorum, topic = sys.argv[1:]
sc = SparkContext("local[*]", "KafkaTweetsCount")
sc.setLogLevel("WARN")
ssc = StreamingContext(sc, 5)
ssc.checkpoint('checkpoint')
kvs = KafkaUtils.createStream(ssc, zkQuorum, "spark-streaming-consumer",
{topic: 1})
words = kvs.window(3600, 60) \
             .flatMap(lambda x: preparar_tweets(x[1])) \
             .map(lambda a: (a, 1)) \
             .reduceByKey(lambda a, b: a+b)

words.pprint()

words.foreachRDD(lambda rdd: rdd.sortBy(lambda x: x[1],
ascending=False).foreachPartition(sendCount))

ssc.start()
ssc.awaitTermination()
```

Apéndice 4 – Código del script que almacena los resultados

```
from pykafka import KafkaClient
from pykafka.common import OffsetType
import datetime
import json
import os

client = KafkaClient(hosts = "10.0.4.192:9092")
topic = client.topics['resultados']

consumer = topic.get_simple_consumer(auto_offset_reset = OffsetType.LATEST,
reset_offset_on_start=True)

data = {}
data["Country_count"] = []
data["Hashtag_world_count"] = []
data["Hashtag_countries_count"] = {}

now = datetime.datetime.now()
last_hour = now.hour
last_minute = now.minute

for msg in consumer:

    if msg is not None:

        now = datetime.datetime.now()
        if now.hour != last_hour or now.minute - last_minute >= 1:

            data["Country_count"].sort(key = lambda x: -x[1])
            data["Hashtag_world_count"].sort(key = lambda x: -x[1])
            data["Hashtag_world_count"] = data["Hashtag_world_count"][:10]
            for cc in data["Hashtag_countries_count"]:
                data["Hashtag_countries_count"][cc].sort(key = lambda x: -
x[1])
```

```
                data["Hashtag_countries_count"][cc] =
data["Hashtag_countries_count"][cc][:10]

            path = "resultados/" + str(last_hour) + "_" + str(last_minute -
(last_minute % 10)) + ".txt"
            with open(path, 'w') as outfile:
                json.dump(data, outfile)
            os.system('aws s3 cp resultados s3://bucket-upna-
aplicacion/resultados --recursive')
            os.remove(path)
            print('Conteo enviado al Bucket')

            last_hour = now.hour
            last_minute = now.minute

            data["Country_count"] = []
            data["Hashtag_world_count"] = []
            data["Hashtag_countries_count"] = {}


        msg = json.loads(msg.value)
        tipo = msg[0].split(' ')[0]
        conteo = msg[1]

        if tipo == 'cc':
            cc = msg[0].split(' ')[1]
            data["Country_count"].append([cc,conteo])

        if tipo == 'ht':
            ht = msg[0].split(' ')[1]
            data["Hashtag_world_count"].append([ht,conteo])

        if tipo == 'htcc':
            cc = msg[0].split(' ')[1]
            ht = msg[0].split(' ')[2]
            if not cc in data["Hashtag_countries_count"]:
                data["Hashtag_countries_count"][cc] = []
            data["Hashtag_countries_count"][cc].append([ht,conteo])
```

Apéndice 5 – Código de los archivos más destacados del proyecto de Angular

src/index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>AppServer</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">

  <link href='https://api.mapbox.com/mapbox-gl-js/v2.0.0/mapbox-gl.css'
rel='stylesheet' />

</head>
<body>
  <app-root></app-root>
</body>
</html>
```

src/styles.css

```
@import '@angular/material/prebuilt-themes/deeppurple-amber.css';

html, body {
  width:100%;
  height:100%;
```

```
  background-image: linear-gradient(to bottom right, #64F3DE, #CAEEDC);
}

.img-pop-up {
  float:left;
  width: 30px;
  margin-right: 10px;
}

#mapa-app .mapboxgl-popup-content {
  padding: 10px;
  max-width: 300px;
  padding-top: 20px;
}
#mapa-app .mapboxgl-popup-content ul {
  list-style: none;
  margin: 0;
  padding: 0;
  float: left;
}

#mapa-app .mapboxgl-popup-content ul h3 {
  margin: 0 0 10px 0;
}
```

src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { MatSliderModule } from '@angular/material/slider';
import { MatTableModule } from '@angular/material/table';
import {BrowserAnimationsModule} from '@angular/platform-browser/animations';


@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    MatSliderModule,
    MatTableModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

src/app/app.component.html

```
<div class='box'>
<div class='div-izda' style='padding: 20px;'>

        <div id="mapa-app"></div>

        <div id='legend'>
                <strong>Cantidad de tweets sobre el COVID-19 por persona en la
última hora:</strong>
                <nav class='legend clearfix'>
                        <span style='background:{{colores[0]}};'></span>
                        <span style='background:{{colores[1]}};'></span>
                        <span style='background:{{colores[2]}};'></span>
                        <span style='background:{{colores[3]}};'></span>
                        <span style='background:{{colores[4]}};'></span>
                        <span style='background:{{colores[5]}};'></span>
```

```
                        <span style='background:{{colores[6]}};'></span>
                        <span style='background:{{colores[7]}};'></span>
                        <span style='background:{{colores[8]}};'></span>
                        <span style='background:{{colores[9]}};'></span>
                        <label><strong>{{valoresLeyenda[0]}}</strong></label>
                        <label><strong>{{valoresLeyenda[1]}}</strong></label>
            <label><strong>{{valoresLeyenda[2]}}</strong></label>
                        <label><strong>{{valoresLeyenda[3]}}</strong></label>
                        <label><strong>{{valoresLeyenda[4]}}</strong></label>
                        <label><strong>{{valoresLeyenda[5]}}</strong></label>
                        <label><strong>{{valoresLeyenda[6]}}</strong></label>
                        <label><strong>{{valoresLeyenda[7]}}</strong></label>
                        <label><strong>{{valoresLeyenda[8]}}</strong></label>
                        <label><strong>{{valoresLeyenda[9]}}</strong></label>
                </nav>
        </div>

    <div id="slider-app-div">
                <mat-slider id="slider-app" min="0" max="143" thumbLabel
[displayWith]="formatLabel" (input)="onIn>
        </div>

</div>
<div class='div-dcha' style='padding: 20px;'>
<table mat-table [dataSource]="dataSource" class="mat-elevation-z8">
  <ng-container matColumnDef="position">
    <th mat-header-cell *matHeaderCellDef> No. </th>
    <td mat-cell *matCellDef="let element"> {{element.position}} </td>
  </ng-container>

  <ng-container matColumnDef="hashtag">
    <th mat-header-cell *matHeaderCellDef> Hashtag </th>
    <td mat-cell *matCellDef="let element"> {{element.hashtag}} </td>
  </ng-container>

  <ng-container matColumnDef="conteo">
    <th mat-header-cell *matHeaderCellDef> Conteo </th>
    <td mat-cell *matCellDef="let element"> {{element.conteo}} </td>
  </ng-container>

  <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
  <tr mat-row *matRowDef="let row; columns: displayedColumns;"></tr>
</table>
</div>
</div>
```

src/app/app.component.css

```
.box{
  display: flex;
  width : 100vw;
}

.div-izda:{
  flex: 1 1 auto;
}

.div-dcha:{
  flex: 1 1 auto;
  padding: 60 px;
}


#mapa-app{
  margin: auto;
```

```
  height: 80vh;
  width: 70vw;
  background-color: yellow;
  border: 3px solid black;
}

#slider-app{
  width: 100%;
}

#slider-app-div{
  margin: auto;
  width: 70vw;
  background-color: DodgerBlue;
  border: 3px solid black;
}

#legend{
  display: block;
  margin: auto;
  width: 70vw;
  height: 6vh;
  background-color: DodgerBlue;
  border: 3px solid black;
}

.legend label,
.legend span {
  display:block;
  float:left;
  height:15px;
  width:10%;
  text-align:center;
  font-size:13px;
  color:#000000;
  }
```

src/app/app.component.ts

```
import { HttpClient, HttpHeaders, HttpClientModule } from
'@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { environment } from '../environments/environment.prod'
import { MatTableDataSource } from '@angular/material/table'
import * as poblacion from './poblacion.json'
import * as Mapboxgl from 'mapbox-gl'

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit{

  public mapa: Mapboxgl.Map;            // El objeto Mapboxgl
  public http: HttpClient;              // El cliente HTTP
  public resultados: any;               // Los resultados con los conteos

  public colores: string[];             // Los 10 colores del gradiente
  public valoresLeyenda : string[];     // Contiene los valores que se
mostraran en la leyenda

  public ELEMENT_DATA : PeriodicElement[] = [
    { position: 1, hashtag: '-', conteo: 0},
```

```
    { position: 2, hashtag: '-', conteo: 0},
    { position: 3, hashtag: '-', conteo: 0},
    { position: 4, hashtag: '-', conteo: 0},
    { position: 5, hashtag: '-', conteo: 0},
    { position: 6, hashtag: '-', conteo: 0},
    { position: 7, hashtag: '-', conteo: 0},
    { position: 8, hashtag: '-', conteo: 0},
    { position: 9, hashtag: '-', conteo: 0},
    { position: 10, hashtag: '-', conteo: 0},
  ]

  public dataSource = new
MatTableDataSource<PeriodicElement>(this.ELEMENT_DATA);

  public displayedColumns: string[] = ['position', 'hashtag', 'conteo'];

  ngOnInit(){

    this.colores = ['#77fde6', '#7be0d0', '#80c4ba', '#85a8a5', '#898c8f',
'#8e707a', '#935464', '#97384f', '#9c1c39', '#a10024']
    this.resultados = [];
    this.valoresLeyenda = ["-", "-", "-", "-", "-", "-", "-", "-", "-", "-"]

    this.mapa = new Mapboxgl.Map({
      accessToken: environment.mapboxKey,
      container: 'mapa-app',
      style: 'mapbox://styles/mapbox/light-v9',
      center: [-74.5, 40],
      zoom: 2,
      minZoom: 2
    });

    this.getResultados();

    for(var i = 0; i < 10; i++){
      this.creaLayer(this.colores, i);
    }
  }

  onInputChange(event: any) {
    var d = new Date();
    event.value += Math.floor((d.getMinutes() + (d.getHours() * 60)) / 10);
    event.value %= 144;

    if(this.resultados[event.value] !== undefined){

      var paises = this.resultados[event.value].Country_count;
      var maximo = parseInt(paises[0][1]);
      var paisesPorCapa = [];

      for(var i = 0; i < 10; i++){
        paisesPorCapa[i] = new Array();
      }

      for(var i = 0; i < paises.length; i++){
        var pais = paises[i][0];
        var conteo = parseInt(paises[i][1]);
        var capa = Math.floor(((conteo - 1) / maximo) * 10);
        paisesPorCapa[capa].push(pais);
      }

      for(var i = 0; i < 10; i++){
        this.mapa.setFilter(
            ('capa').concat(i.toString()),
            ['in', 'ADM0_A3_IS'].concat(paisesPorCapa[i]),
          );
```

```
        this.valoresLeyenda[i] = (maximo * i / 10 + 1).toString().concat(" -
").concat((maximo * (i + 1) / 10).toString());
      }

      var hashtags = this.resultados[event.value].Hashtag_world_count;
      console.log(hashtags);
      for(var i = 0; i < 10; i++){
        this.ELEMENT_DATA[i] = {position: i + 1, hashtag: hashtags[i][0],
conteo: parseInt(hashtags[i][1])};
      }
      this.dataSource.data = this.ELEMENT_DATA;

    } else {
      for(var i = 0; i < 10; i++){
        this.mapa.setFilter(
            ('capa').concat(i.toString()),
            ['in', 'ADM0_A3_IS'].concat([]),
          );
      }
      this.valoresLeyenda = ["-", "-", "-", "-", "-", "-", "-", "-", "-", "-"]
      for(var i = 0; i < 10; i++){
        this.ELEMENT_DATA[i] = {position: i + 1, hashtag: "-", conteo: 0};
      }
      this.dataSource.data = this.ELEMENT_DATA;
    }
  }

  formatLabel(value: number) {
    var d = new Date();
    value += Math.floor((d.getMinutes() + (d.getHours() * 60)) / 10);
    value %= 144;

    var minutosTotales = value * 10;
    var horas = Math.floor(minutosTotales / 60);
    var minutos = minutosTotales % 60;

    var salida = horas.toString().concat(":").concat(minutos.toString()) ;
    if (horas < 10){salida = ("0").concat(salida)}
    if (minutos == 0){salida = salida.concat("0")}

    return salida ;
  }

  getResultados(){
    for(var i = 0; i < 144; i++){
      var minutosTotales = i * 10;
      var horas = Math.floor(minutosTotales / 60);
      var minutos = minutosTotales % 60;

      var horasTxt = horas.toString();
      var minutosTxt = minutos.toString();

      if (horas < 10){horasTxt = ("0").concat(horasTxt)}
      if (minutos == 0){minutosTxt = minutosTxt.concat("0")}

      var url = ('https://bucket-upna-
aplicacion.s3.amazonaws.com/resultados/').concat(horasTxt).concat("_").concat(
minutosTxt).concat(".txt");

      fetch(url,{headers: {"Access-Control-Allow-Origin": "*"}})
      .then( r => r.text())
      .then( t =>{
        if(t[0] !== "<"){
          var contenido = JSON.parse(t);
          var index = contenido.index;
          this.resultados[parseInt(index)] = contenido;
        }
```

```
      })

    }
  }

  creaLayer(colores: string[], i: number){
    var mapa = this.mapa;
    var ELEMENT_DATA = this.ELEMENT_DATA;
    var dataSource = this.dataSource;
    var resultados = this.resultados;

    mapa.on('load',function(){
      mapa.addLayer({
        'id': ('capa').concat(i.toString()),
        'source': {
          'type': 'vector',
          'url': 'mapbox://byfrost-articles.74qv0xp0'
        },
        'source-layer': 'ne_10m_admin_0_countries-76t9ly',
        'type': 'fill',
        'paint': {
          'fill-color': colores[i],
          'fill-opacity': 0.75,
          'fill-outline-color': '#F2F2F2'
        }
      });

      mapa.setFilter(
        ('capa').concat(i.toString()),
        ['in', 'ADM0_A3_IS'].concat([]),
      );

      mapa.on('click', ('capa').concat(i.toString()), function(mapElement){
        const cc = mapElement.features[0].properties.ADM0_A3_IS;
        console.log( resultados[0].Hashtag_countries_count);
        if (resultados[0].Hashtag_countries_count[cc] !== undefined){
          for(var i = 0; i < 10; i++){
            if(resultados[0].Hashtag_countries_count[cc][i] !== undefined){
              ELEMENT_DATA[i] = {position: i + 1, hashtag:
resultados[0].Hashtag_countries_count[cc][i][0], conteo:
resultados[0].Hashtag_countries_count[cc][i][1]};
            }else{
              ELEMENT_DATA[i] = {position: i + 1, hashtag: '-', conteo: 0};
            }
          }
        }else{
          for(var i = 0; i < 10; i++){
            ELEMENT_DATA[i] = {position: i + 1, hashtag: '-', conteo: 0};
          }
        }
        dataSource.data = ELEMENT_DATA;
        fetch(`https://restcountries.eu/rest/v2/alpha/${cc}`)
          .then((data) => data.json())
          .then((country) => {
            const html = `
              <img class="img-pop-up" src='${country.flag}' />
              <ul>
                <li><h3>${country.name}</h3></li>
                <li><strong>Spanish translation</strong>
${country.translations.es}</li>
                <li><strong>Capital:</strong> ${country.capital}</li>
                <li><strong>Population:</strong> ${country.population}</li>
                <li><strong>Demonym:</strong> ${country.demonym}</li>
              </ul>`;
            new Mapboxgl.Popup()
              .setLngLat(mapElement.lngLat)
              .setHTML(html)
```

```
                .addTo(mapa);
          });
        })

    });
  }
}


export interface PeriodicElement {
  hashtag: string;
  position: number;
  conteo: number;
}
```

Apéndice 6 – Código del productor de pruebas

```
from kafka import KafkaProducer
import math
import json
import pandas as pd
import datetime as dt
import time
from iso3166 import countries
import random

hashtags = ['冠状病毒
','mascarilla','vacuna','moderna','vaccine','mask','virus','PCR','COVID-
19','Covid']

df = pd.read_csv("2021-01-11-dataset.tsv", sep='\t')
df = df.drop(["tweet_id", "date", "lang"], axis=1)

producer = KafkaProducer(bootstrap_servers='10.0.4.192:9092')

i = 0

old_time = df.iloc[0].time

while True:
  row = df.iloc[i]
  a =
dt.datetime(2021,1,15,int(old_time.split(":")[0]),int(old_time.split(":")[1]),
int(old_time.split(":")[2]))
  b =
dt.datetime(2021,1,15,int(row.time.split(":")[0]),int(row.time.split(":")[1]),
int(row.time.split(":")[2]))
  time.sleep((b-a).seconds)

  kafka_msg = {}
  if type(row.country_code) is not float:
    kafka_msg['country_code'] = countries.get(row.country_code).alpha3
  kafka_msg['hashtags'] = [hashtags[random.randint(0, 9)]]
  producer.send('tweets', json.dumps(kafka_msg).encode())

  old_time = row.time
  i += 1
```