DOCUMENTO DE DISEÑO

Francisco Javier Fernández

1. INTRODUCCIÓN

El documento proporciona una visión detallada del diseño y de la arquitectura del sistema que va a dar solución a la necesidad de implementar un sistema de análisis de sentimientos de comentarios en Twitter, que serán procesados en tiempo real utilizando Kafka.

El sistema, además de diseñarse para ser robusto y escalable, está pensado para que todos sus componentes sean desplegados como servicios mediante contenedores Docker utilizando Docker Compose, lo que garantiza una configuración consistente y una fácil reproducibilidad del entorno en diferentes plataformas. Además, de esta manera a la vez se da solución a la necesidad de desplegar todo el escenario con un solo comando.

2. OBJETIVO

El objetivo principal del sistema es dar solución a la necesidad de procesar de manera eficiente tweets en tiempo real utilizando Kafka como infraestructura central y realizar un análisis de sentimientos a partir del contenido de cada tweet. Gracias al uso de Kafka, se dispone de un sistema altamente disponible, tolerante a fallos y fácilmente escalable para satisfacer las necesidades de procesamiento de datos en tiempo real para este caso de uso.

Además, se ha intentado proporcionar una interfaz de usuario intuitiva para consultar el resultado del análisis de sentimientos y diferentes herramientas de gestión de la plataforma para poder administrar, supervisar y optimizar el rendimiento del sistema de manera eficaz.

3. ALCANCE

El alcance de este sistema abarca desde la implementación de los servicios básicos de Kafka, como Zookeeper y Broker, hasta componentes adicionales como Schema Registry, Connect o Control Center. Asimismo, se han desplegado contenedores docker para la generación y consumo de datos, con funciones añadidas como la de procesar el contenido de los datos mediante algoritmos de machine learning con la finalidad de realizar un análisis de sentimientos.

Además de los servicios esenciales para el funcionamiento de la infraestructura de Kafka, se han realizado integraciones con otros componentes, entre los que se incluyen KSQLDB Server para el análisis de datos en tiempo real, una interfaz de usuario web para la gestión y la monitorización del clúster Kafka, una base de datos MongoDB para el almacenamiento persistente de los tweets y un servidor PHP para visualizar su contenido.

4. ARQUITECTURA DEL SISTEMA

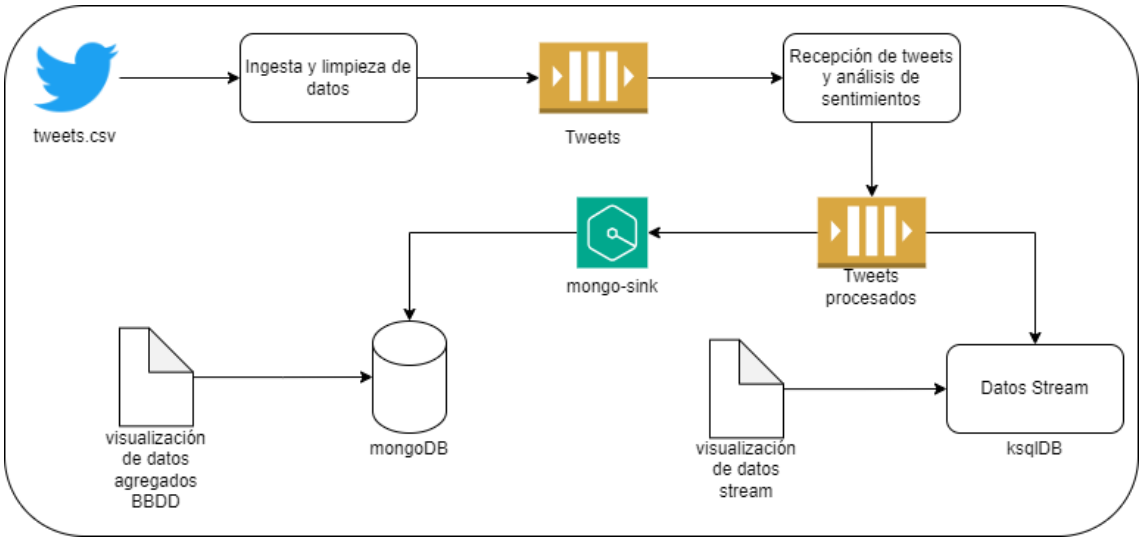
A continuación, se detalla la arquitectura del sistema con los diferentes componentes y los flujos de información necesarios para resolver el ejercicio propuesto.

En primer lugar y debido a la imposibilidad de capturar tweets de X utilizando una cuenta de desarrollador gratuita, se ha utilizado un fichero .csv con más de 12.000 tweets con contenido sobre videojuegos y otros servicios digitales. Este fichero será cargado en el host que realiza el rol de productor y con un pequeño programa desarrollado en Python se seleccionará el contenido del tweet y lo enviará al topic Tweets. El consumidor, que también ejecuta un programa Python para implementar la lógica de sus operaciones, estará suscrito al topic Tweets y mediante un pipeline ejecutará un análisis de sentimientos utilizando el algoritmo de Machine Learning basado en BERT. Una vez etiquetados los tweets con el sentimiento, el consumer cumple también el rol de producer, ya que envía los tweets analizados a una segunda cola (topic tweets procesados) al

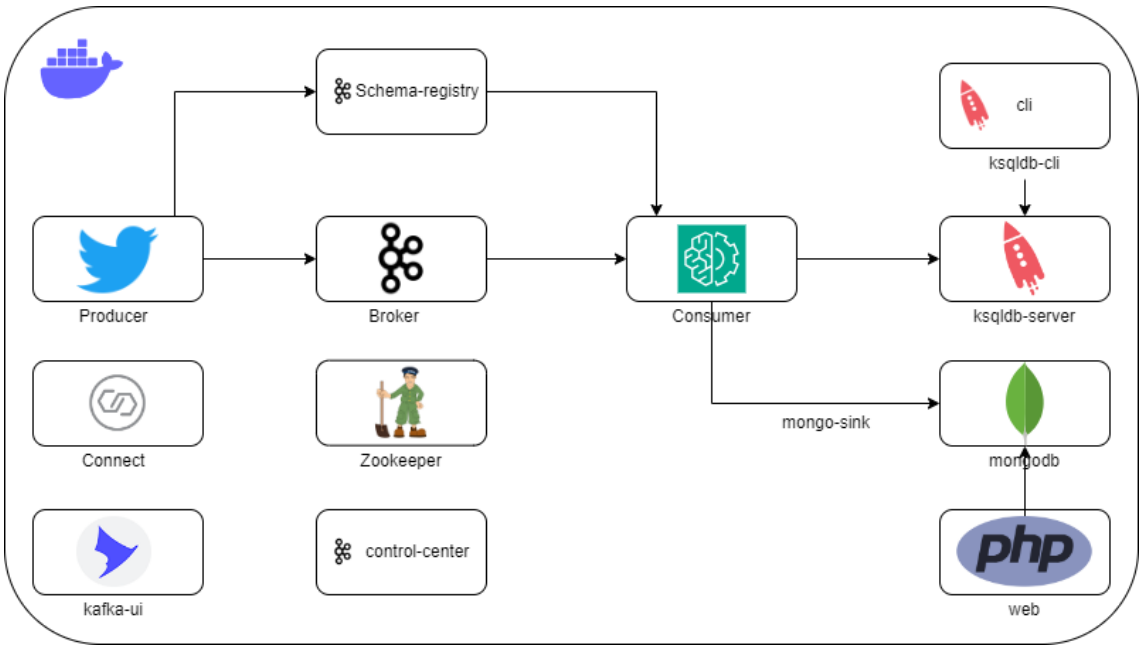
que estarán suscritos un servidor ksqlDB que permitirá trabajar con los streams de datos en tiempo real, y por otro lado una base de datos MongoDB para que los datos sean persistentes.

La visualización de los tweets y las consultas agregadas de MongoDB se consultarán a partir de la interfaz web de un servidor PHP, mientras que para la visualización de los datos de ksqlDB será necesario utilizar ksqlDB-cli.

Además, también es importante señalar que para poder conectar Kafka a un sistema externo como la base de datos MongoDB ha sido necesario utilizar un conector, permitiendo así almacenar los datos de los tweets procesados en un componente externo.



Como la solución planteada es totalmente modular y consiste en implementar un contenedor Docker para cada servicio, a continuación se incluye un diagrama con los diferentes componentes que se despliegan mediante Docker Compose, teniendo en cuenta tanto los servicios básicos que forman parte de Kafka, como los adicionales que proporcionan al sistema el resto de funcionalidades.



Servicios incluidos:

- **Broker:** es uno de los componentes críticos dentro de la arquitectura Kafka. Controla y almacena los flujos de datos, procesando los mensajes entrantes y gestionando las solicitudes de lectura o el mantenimiento de los offsets de los consumidores. También es el responsable de la gestión de los topics y sus particiones, las réplicas distribuidas y otros aspectos relacionados con la seguridad y la alta disponibilidad, razón por la que es habitual disponer de un mínimo de 3 brokers.

- **Zookeeper**: se trata de un servicio de coordinación distribuida para Kafka, que implementa diferentes primitivas para gestionar y sincronizar la configuración y los metadatos del clúster.

- **Producer**: su función principal es enviar datos a un topic de Kafka para que posteriormente puedan ser consumidos. Cuando un productor envía un mensaje, puede especificar un "key" (clave) y un "value" (valor) para el mensaje. El valor contiene los datos enviados y puede ser cualquier objeto serializable, mientras que la clave es opcional y se utiliza para determinar a qué partición específica de un topic se enviará el mensaje, por lo que si se envían los datos sin una clave determinada Kafka alternará los mensajes entre los diferentes brokers mediante el algoritmo Round Robin.

- **Consumer**: este componente se suscribe a uno o más topics para leer los mensajes que han sido publicados en ellos, pudiendo trabajar solo o como parte de un grupo de consumidores.
  Cuando un consumidor lee mensajes de un topic, también mantiene un "offset" para hacer seguimiento de qué mensajes ha consumido. Estos mensajes consumidos no son eliminados del topic de inmediato, ya que a partir de los offsets es posible rastrear qué mensajes han sido leídos y cuáles no, permitiendo recuperar mensajes previamente consumidos en caso de que sea necesario.
  Una característica importante a tener en cuenta relacionada con el paralelismo de consumo cuando los consumidores pertenecen a un grupo de consumidores, es que una partición de un topic solo puede ser consumida por un único consumidor de un consumer group, aunque sí puede ser consumida por consumidores que se encuentren en diferentes consumer groups.

- **Schema Registry:** este componente permite gestionar y almacenar los esquemas de datos utilizados en los mensajes, proporcionando un formato común que garantice la compatibilidad de los datos entre productores y consumidores de Kafka.

- **Connect**: es un componente que permite que la infraestructura de Kafka pueda ser integrada con diferentes sistemas externos, tanto de entrada como de salida.

- **Control Center:** se trata de una aplicación web de la empresa Confluent que facilita la supervisión y operación de un clúster Kafka, la configuración de componentes, la monitorización de rendimiento, la visualización de métricas, supervisión de topics, producción de mensajes e incluso la integración con ksqldDB.

- **UI for Apache Kafka:** similar al servicio anterior, consiste en una interfaz web de código abierto para gestionar y operar clústers de Kafka. Permite visualizar diferentes clústers Kafka, los brokers, gestión de topics, gestión de consumers, producción de mensajes,

visualización de estadísticas, realizar configuraciones de esquema o incluso establecer ACL.

- **KsqlDB Server:** es un motor de procesamiento de datos en tiempo real desarrollada por Confluent que proporciona un lenguaje de consultas basado en SQL sobre flujos de datos de Kafka.

- **KsqlDB Client:** consiste en una interfaz de usuario que proporciona una forma intuitiva de interactuar con ksqlDB Server y realizar análisis de datos en tiempo real sobre flujos de datos en Kafka, permitiendo escribir y ejecutar consultas KSQL, visualizar resultados o monitorizar consultas activas para análisis de datos.

- **MongoDB:** es un DBMS NoSQL de código abierto orientado a documentos, que presenta un alto rendimiento y es altamente escalable. En este sistema se utilizará para almacenar los mensajes procesados por el analizador de sentimientos.

- **PHP:** este servicio ejecuta una aplicación web para acceder a la base de datos de mongoDB y mostrar los registros almacenados.

5. DESARROLLO DEL SISTEMA

A continuación, se detalla cómo se ha construido la lógica del sistema y las decisiones de diseño que se han tomado de manera global y para cada uno de los componentes.

Docker Compose

Para lograr un sistema modular, escalable, portable, eficiente, tolerante a fallos y que permita levantar todos sus componentes de manera automática se ha decidido utilizar Docker Compose como base.

Se ha partido del fichero .yml del repositorio de github utilizado en la asignatura, modificando y añadiendo nuevos servicios tal como se puede apreciar en el directorio de código que se adjunta. Para cada uno de los servicios, se indican las características del contenedor, la imagen que utiliza, los servicios dependientes, la definición de alias, el reinicio automático en caso de fallo o los comandos a ejecutar.

Esta solución, como ya hemos comentado, tiene muchas ventajas y en lo relacionado con Kafka, permitiría de una manera sencilla y siempre que se dispongan de recursos suficientes en el equipo que ejecute Docker Desktop, aumentar el número de productores, brokers o consumidores, por poner un ejemplo, añadiendo el nuevo servicio en el .yml y modificando los scripts que implementan la lógica del sistema para modificar el número de particiones, réplicas, consumidores o grupos de consumidores, etc.

Zookeper

El servicio definido en el docker es el mismo que ya se encontraba definido en el fichero .yml de partida, al que únicamente se le ha añadido un alias para poder ser referenciado por el mismo dentro de la red y configurar un reinicio automático del contenedor en caso de fallo.

Broker

Tampoco se han realizado cambios respecto a la configuración inicial, excepto las comentadas anteriormente para Zookeper. Teniendo en cuenta las limitaciones del escenario con el que se está trabajando, únicamente se va a desplegar 1 broker, algo que nos va a impedir introducir varias

réplicas de particiones, aunque es algo que se puede solventar definiendo nuevos servicios bróker y rebalanceando el número de réplicas (se puede hacer incluso en caliente). Por otro lado, teniendo en cuenta que solo tendremos una partición para el único consumidor del sistema, la latencia en este caso será mínima.

Schema-registry

Tampoco se han realizado cambios respecto a la configuración inicial, excepto las comentadas para definir un alias y establecer un reinicio automático en caso de fallo.

Connect

En este caso, para permitir que la base de datos MongoDB se pueda conectar a la infraestructura de Kafka se han especificado volúmenes para añadir al contenedor Connect el conector desarrollado por MongoDB y el script que cargará la configuración del conector de MongoDB.

Control-Center

De la misma manera que en algunos de los servicios anteriores, tampoco se han realizado cambios respecto a la configuración inicial, excepto las comentadas para definir un alias y establecer un reinicio automático.

KsqlDB-Server

Tampoco ha sido necesario realizar más cambios de los ya comentados.

KsqlDB-Cli

En este caso se ha establecido un volumen para introducir en el contenedor docker el contenido del directorio ksqldb, que contiene dos scripts que van a permitir configurar el servidor ksqldb-server y analizar sus datos.

En la definición de docker compose se ha definido un comando que desencadena la ejecución del primero de los scripts, que comprueba si ksqkDB-server está disponible y después de unos segundos de espera, ejecuta el script principal. Cuando el script principal se ejecuta, en primer lugar se verifica si ya se ha creado anteriormente en el servidor un stream o una tabla con el mismo nombre, y si no existe se crean.

MongoDB

Tampoco ha sido necesario realizar más cambios de los ya comentados para los servicios anteriores. La lógica que define la configuración de la base de datos MongoDB se establece en el fichero mongo_sink_config.json que se carga con un script en el servicio Connect para permitir que se realice la conexión de la infraestructura Kafka con este contenedor.

```
{
  "name": "sink",
  "config": {
    "connector.class": "com.mongodb.kafka.connect.MongoSinkConnector",
    "key.converter": "org.apache.kafka.connect.storage.StringConverter",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "topics": "tweets_processed",
    "connection.uri": "mongodb://admin:admin@mongodb:27017",
    "database": "my_tweets",
    "collection": "kafka_tweets",
    "value.converter.schemas.enable": "false"
  }
}
```

Producer

Para construir el contenedor del producer, se ha utilizado la imagen Python:3.9-slim sobre la que se instalan los diferentes paquetes y librerías necesarias para que pueda desarrollar la función de productor. Además, se definen volúmenes para introducir en el contenedor el programa Python que contiene la lógica del producer, y el .csv con el dataset de tweets que se utiliza como fuente de información.

El dataset utilizado dispone de cuatro columnas [identificador, servicio, sentimiento, mensaje] y cada mensaje ya se encuentra etiquetado con un sentimiento, aunque en nuestro sistema solo se utilizará el contenido del campo mensaje, siendo lo demás obviado, por lo que el sentimiento del dataset únicamente servirá como referencia para verificar de manera manual el grado de éxito del algoritmo de ML utilizado en el consumer.

```
13188,Xbox(Xseries),Neutral,What da hell this mean in english
13189,Xbox(Xseries),Negative,My iphone has more cores than this.
13190,Xbox(Xseries),Irrelevant,"Would be so cool to win this, wouldnt even find it
funny it just looks cool as!  "
13191,Xbox(Xseries),Negative,Xbox naming is unbelievably confusing.
```

Respecto al código Python que se ejecuta en el productor, se ha utilizado como punto de partida el código de simpleProducer.py, al que se le han añadido las siguientes funcionalidades:

En primer lugar y para que la creación del topic se realice de manera desatendida al ejecutarse el programa producer.py, se hace uso de la biblioteca KafkaAdminClient para crear un nuevo topic denominado 'Tweets', con una partición y un factor de replicación igual a 1. Aunque realmente estas últimas características se podrán omitir ya que estamos utilizando los valores por defecto de Kafka, se dejan indicados para facilitar el escalado, y que solo sea necesario indicar en el código Python el número de particiones deseadas si se añaden nuevos consumidores o el nuevo factor de replicación, en el caso de que se añadan más brokers, aunque también tendremos siempre la posibilidad de realizar estos cambios ejecutando el comando correspondiente. Por otro lado, y como en este ejercicio solo tendremos un consumidor por el momento, no necesitamos paralelismo y 1 con una partición es más que suficiente, algo que además nos permite disminuir la latencia, aunque en este entorno no sea un factor relevante.

Respecto a la lectura del .csv, para cada línea se utiliza el primer campo como clave y para el valor se crea un diccionario con el contenido del tweet, que es el cuarto campo de la fila. Posteriormente, se utiliza el método send para que el productor envíe cada mensaje al topic tweets. En este caso y para intentar simular un poco más la ingesta de tweets en tiempo real sin complicar mucho la lógica del programa, se ha decidido que los mensajes se envíen uno a uno en un intervalo de tiempo aleatorio entre 0 y 3 segundos, por lo que los mensajes no se están acumulando antes de ser enviados y no se distribuyen en forma de batch.

Consumer

La construcción del contenedor del consumidor de Kafka también se ha realizado a partir de la imagen Python:3.9-slim, pero en este caso para poder ejecutar el algoritmo de machine learning que se ha seleccionado ha sido necesario instalar diferentes librerías que son muy pesadas, y como no se han configurado volúmenes persistentes para cada ejecución del consumidor causaban bastantes minutos de espera hasta instalar todo el software. Para solventar esta problemática se ha publicado la imagen que incluye todas las dependencias en Docker Hub, aunque también se ha dejado el Dockerfile en el directorio docker-consumer por si se en algún momento se desea realizar todo el proceso manualmente.

Cuando el consumidor se inicia, ejecuta el programa consumer.py, que inicializa un pipeline con el modelo finiteautomata basado en BERT, ya que después de probar diferentes algoritmos este es el que mayor grado de acierto tiene. También hay que destacar que este modelo, aunque es muy potente, tiene una limitación en cuanto al número máximo de tokens que puede procesar, ya que cuando alcanzaba 128 tokens causaba la ejecución de programa Python.

Posteriormente, el consumer se suscribe al topic tweets, mediante el cual los mensajes que el productor ha procesado son analizados con el modelo descrito en el pipeline. Una vez realizado este proceso, como este contenedor tiene la doble función de consumidor-productor, envía por el topic 'tweets_processed' los mensajes etiquetados con el sentimiento y el score.

Kafka-UI

Se ha decidido implementar esta herramienta como alternativa a Control-Center, ya que se trata de una herramienta libre que con una relativa simplicidad de implementación permite gestionar varios clústers de Kafka, con una interfaz muy amigable y sobre todo con una velocidad de respuesta muy alta gracias a su ligereza. Para ello se ha creado un servicio con la imagen publicada por su creador y las variables de entorno necesarias para integrarlo en el sistema.

PHP

Por último y para proporcionar una manera visual y más amigable de ver el contenido de los documentos almacenados en MongoDB, se ha preparado un contenedor PHP que proporciona una interfaz web bastante sencilla para visualizar el número total de mensajes procesados, el número de mensajes por cada tipo de sentimiento tanto en números absolutos como en porcentajes, o el sentimiento promedio. Además, se ha decidido añadir un desplegable para poder consultar el cuerpo entero de los mensajes por tipo de sentimiento.

La lógica de la aplicación PHP anterior se incluye en el fichero index.php. Hay que destacar que para poder consultar desde PHP la base de datos MondoDB ha sido necesario instalar diferentes dependencias y utilizar composer, tal como se puede apreciar en la carpeta docker-php en la que se encuentra el archivo Dockerfile con el que se ha construido. En cualquier caso y para facilitar su distribución, se ha decidido publicar la imagen en Docker Hub.

6. ENTREGABLES

Todo el contenido, tanto el código fuente como la documentación, además de publicarse en GitHub, se entrega en formato .zip a través del Moodle del campus. En la siguiente ilustración se puede observar la estructura de directorios que se adjunta.