

Monitoreo del Tráfico Aéreo: Análisis en Tiempo Real



Autores:
Marian Aguilar Tavier.
Jennifer de la C. Sánchez.

Enero, 2025

Contents

Introducción	2
Ingesta de datos	3
API utilizada	3
Apache Kafka para la ingesta	3
¿Qué es Apache Kafka?	3
¿Por qué utilizarlo en este caso?	4
Despliegue de Kafka con Docker y Docker Compose	5
Captura de los datos	5
Almacenamiento de datos	6
Hadoop Distributed File System(HDFS)	6
HDFS y Docker	7
Almacenamiento	7
Monitoreo	8

Introducción

Hoy en día, la cantidad de datos generados en un instante de tiempo muy pequeño es prácticamente innumerable, provenientes de múltiples fuentes. De esta forma surge lo que se conoce como *Big Data*. El procesamiento y análisis de grandes volúmenes de datos de manera eficiente permite obtener información valiosa que optimiza procesos, predice tendencias y ofrece soluciones a problemas complejos, independientemente del ámbito de aplicación.

De esta forma, una de las aplicaciones del Big Data radica en el análisis de datos referentes al tráfico aéreo. Este enfoque no solo permite el monitoreo en tiempo real de las operaciones aéreas, sino que también facilita el desarrollo de análisis más complejos. De esta manera, es posible predecir retrasos, identificar picos de actividad, detectar temporadas con mayor tráfico aéreo, etcétera.

Debido a la relevancia y utilidad que tiene este tema, este proyecto tiene como objetivo desarrollar un sistema para la captura, análisis y almacenamiento de datos en tiempo real sobre el tráfico aéreo a nivel global. El sistema brindará información actualizada sobre las operaciones aéreas, posibles retrasos en los vuelos y condiciones meteorológicas que puedan afectar la trayectoria de las aeronaves.

Para ello, se emplearán tecnologías propias de BigData como Apache Kafka, Hadoop Distributed File System(HDFS), junto con Python y Docker para garantizar un entorno aislado y bien estructurado que optimice la captura, el almacenamiento y el análisis de datos.

Ingesta de datos

API utilizada

Para la ingesta de los datos en este caso se ha empleado la API de OpenSky Network [1]. La misma proporciona datos de aviación en tiempo real de Automatic Dependent Surveillance-Broadcast (ADS-B) que se recopilan a partir de una red de sensores que se encuentran ubicados en todo el mundo.

El acceso a estos datos es gratuito. La API genera vectores de estado para cada aeronave, que incluyen detalles como el identificador ICAO, el llamado (callsign), el país de origen, la posición en tiempo, la velocidad de la aeronave y la última hora de contacto.

Se pueden realizar consultas para obtener información sobre vuelos específicos o todos los vuelos dentro de un intervalo de tiempo determinado. Esto incluye la capacidad de filtrar por aeronave específica. Además permite acceder a un historial de vuelos hasta 30 días atrás, lo que es útil para análisis posteriores y seguimiento de aeronaves.

Es importante destacar que no proporciona datos comerciales de vuelo como horarios de aeropuertos, retrasos o información adicional que no pueda derivarse del contenido de los datos ADS-B.

Para acceder a la API y a todos los vuelos en tiempo real se ha utilizado la url: <https://opensky-network.org/api/states/all>.

Apache Kafka para la ingesta

¿Qué es Apache Kafka?

Apache Kafka es una plataforma de transmisión distribuida diseñada para manejar flujos de datos en tiempo real de manera eficiente, escalable y confiable [2]. Gracias a su arquitectura basada en discos y su enfoque en operaciones secuenciales, Kafka puede manejar millones de mensajes por segundo con una latencia extremadamente baja, lo que lo convierte en una solución ideal para aplicaciones de alto rendimiento. Tiene numerosos casos de uso, como la transmisión distribuida, el procesamiento de flujos, la integración de datos y la mensajería de publicación/suscripción.

Para comprender como funciona Apache Kafka es necesario comprender algunos conceptos fundamentales:

- **Topic:** es básicamente una categoría o un nombre de fuente en el que se almacenan y publican mensajes durante las operaciones. Los mensajes son en su mayoría matrices de bytes que pueden almacenar cualquier objeto en cualquier formato. Representa una unidad de datos que encapsula un evento. Cada tema en Kafka está dividido en particiones, que son subconjuntos del tema distribuidos entre varios nodos del clúster, permitiendo el manejo simultáneo de múltiples flujos de datos y garantizando la escalabilidad.

- **Producer:** Los productores son componentes del sistema que generan datos y los publican en uno o más temas en Kafka. Estos productores tienen la capacidad de etiquetar los mensajes con claves específicas, lo que facilita la organización y distribución de los datos dentro de las particiones del tema.
- **Consumer:** los consumidores son aplicaciones que se suscriben a los temas para procesar los mensajes a medida que se publican. Los consumidores pueden trabajar de forma independiente o como parte de un grupo, donde la carga de trabajo se distribuye entre múltiples instancias para maximizar la eficiencia.
- **Broker:** El clúster de Kafka está compuesto por múltiples brokers, que son nodos responsables de almacenar y administrar los datos de los temas. Cada partición dentro de un tema tiene un líder, que es el broker encargado de manejar las solicitudes de lectura y escritura, mientras que las réplicas de las particiones se almacenan en otros brokers para garantizar la disponibilidad en caso de fallos. Esta replicación de particiones es una de las principales características que hacen de Kafka un sistema tolerante a fallos, ya que permite que el sistema continúe operando incluso si uno o varios brokers fallan.

Al dividir un registro en particiones, Kafka puede escalar horizontalmente los sistemas. Como tal, Kafka modela los eventos como pares clave-valor. Internamente, las claves y los valores son solo secuencias de bytes, pero externamente en el lenguaje de programación de elección, a menudo son objetos estructurados representados en el sistema de tipos de su lenguaje. Kafka llama a la traducción entre tipos de lenguaje y serialización y deserialización de bytes internos. El formato serializado suele ser JSON, JSON Schema, Avro o Protobuf.

¿Por qué utilizarlo en este caso?

En el caso del tráfico aéreo y el monitoreo de aviones, estamos en presencia de datos altamente dinámicos, por lo que es necesario tener acceso en tiempo real al estado de cada una de las aeronaves en todo el mundo, y Kafka es una herramienta ideal para lograrlo.

El número de aviones en vuelo puede variar significativamente dependiendo de la hora del día, el día de la semana o la temporada. Kafka permite escalar horizontalmente el sistema al agregar más brokers al clúster, garantizando que pueda manejar un incremento en la cantidad de datos sin problemas, en especial durante los horarios o temporadas picos de vuelo. Además se evita la pérdida de datos debido a su fuerte tolerancia a fallos en caso de interrupciones en la red u otro tipo de fallo en particular.

El uso de Kafka en este caso también facilita la integración con otras herramientas y tecnologías del ecosistema de big data, como Apache Spark, para el análisis en tiempo real, o HDFS, para el almacenamiento a largo plazo de datos históricos.

Despliegue de Kafka con Docker y Docker Compose

Luego de explicar los conceptos fundamentales que permiten comprender el funcionamiento de Apache Kafka, procedemos a explicar cómo se realizó finalmente este proceso de ingesta. Para ello, se utilizó Docker y Docker Compose, que permitieron una implementación eficiente y escalable del sistema.

Utilizamos Docker para crear contenedores que encapsulan Apache Kafka y Zookeeper, asegurando un entorno de ejecución consistente y simplificando la configuración. Con Docker Compose, pudimos definir y administrar estos contenedores de manera más sencilla y automatizada a través del archivo ‘docker-compose.yml’.

Este archivo incluye servicios para Apache Kafka, Zookeeper y Hadoop, especificando detalles como las imágenes de Docker a utilizar, los puertos expuestos, las variables de entorno necesarias y los volúmenes para persistencia de datos. Esta configuración permite crear un entorno de ejecución aislado y replicable, simplificando la gestión de la infraestructura y reduciendo los riesgos asociados a la configuración manual.

Una vez configurado correctamente el archivo de configuración de Docker, y cargada todas sus imágenes se inicializó el contenedor que se encargaría de llevar a cabo este proceso mediante el comando ‘docker compose up’ y dentro de este se ejecuta todo lo que se explicará en la siguiente sección.

Captura de los datos

Para el proceso de captura de datos se utilizó Apache Kafka junto con la biblioteca *kafka-python* [4].

Se creó un archivo “producer.py” encargado de obtener los datos en tiempo real de la API y enviarlos al clúster de Kafka. Estos datos se envían de forma continua a un topic en específico, para el cual se creó el topic ‘Air_Traffic’, permitiendo una ingesta constante y actualizada de información sobre el estado de los vuelos. Primero, se configura una instancia de *KafkaProducer* que se conecta al servidor de Kafka en localhost:9092. Este producer se encarga de serializar los datos en formato JSON antes de enviarlos. La configuración se realiza definiendo las propiedades de conexión y el serializador de valores.

```
1 from kafka import KafkaProducer
2 import json
3
4 producer = KafkaProducer(
5     bootstrap_servers=['localhost:9092'],
6     value_serializer=lambda x: json.dumps(x).encode('
7         utf-8')
```

Luego se implementó la función ‘fetch_data’, que realiza solicitudes periódicas a la API de OpenSky-Network para obtener información sobre el estado de las aeronaves. Cada solicitud utiliza el protocolo HTTP, y los datos recibidos se

procesan en formato JSON, lo que permite una integración directa con Kafka y otros componentes del sistema. Una vez capturados, los datos se publican en el tema 'Air_Traffic' utilizando el método 'send', y se garantiza su envío inmediato mediante el uso del método 'flush'. Este procedimiento asegura que no haya retrasos significativos en la transmisión de información, lo cual es crucial para aplicaciones en tiempo real como esta.

Luego de implementar el producer, se configuró el consumer que será el encargado de suscribirse al topic creado y leer los mensajes en tiempo real. Una vez que el consumer se suscribe a "Air Traffic" (topic) este deserializa los datos recibidos en formato json, asegurando que sean legibles y puedan ser procesados de manera eficiente.

```
1 from kafka import KafkaConsumer
2 import json
3
4 consumer = KafkaConsumer(
5     'Air_Traffic',
6     bootstrap_servers=['localhost:9092'],
7     auto_offset_reset='earliest',
8     value_deserializer=lambda x: json.loads(x.decode('
9         utf-8'))
10 )
```

Una de las principales ventajas de utilizar consumidores de Kafka es su capacidad para operar de manera distribuida, lo que permite procesar grandes volúmenes de datos dividiendo la carga entre múltiples instancias. Esto garantiza que el sistema pueda escalar de forma eficiente según las necesidades. Además por su naturaleza desacoplada los productores no dependen de los consumidores para operar, y los datos permanecen almacenados temporalmente en Kafka hasta que los consumidores estén listos para procesarlos, lo que no solo mejora la resiliencia del sistema, sino que también permite una mayor flexibilidad al integrar nuevos componentes o funcionalidades sin interrumpir el flujo de datos existente.

Almacenamiento de datos

Hadoop Distributed File System(HDFS)

HDFS es un sistema de archivos distribuido diseñado para almacenar grandes volúmenes de datos en un entorno distribuido y manejar de manera eficiente aplicaciones que requieren un procesamiento intensivo de datos.[3]

Utiliza una arquitectura distribuida que divide los datos en bloques de tamaño fijo y los distribuye entre varios nodos en un clúster. Una de sus características más importantes es su tolerancia a fallos, ya que cada bloque de datos se replica en múltiples nodos para garantizar la disponibilidad de los datos incluso si uno o varios nodos fallan. Por defecto, cada bloque tiene tres réplicas, aunque este número es configurable según las necesidades del sistema.

La arquitectura de HDFS sigue un modelo maestro-esclavo compuesto por el Namenode, que actúa como maestro, y los Datanodes, que son los nodos esclavos. El Namenode administra el sistema de archivos, gestionando metadatos como la estructura de directorios, la ubicación de los bloques y las políticas de replicación, mientras que los Datanodes almacenan físicamente los bloques de datos y ejecutan las operaciones de lectura y escritura según las instrucciones del Namenode.

Cuando un cliente escribe un archivo, este se divide en bloques que se envían a diferentes Datanodes, siguiendo las políticas de replicación establecidas. El Namenode registra las ubicaciones de estos bloques. En el caso de una lectura, el cliente consulta al Namenode para obtener la ubicación de los bloques y accede directamente a los Datanodes para leerlos. Si un nodo falla, el sistema replica automáticamente los bloques desde las copias disponibles en otros nodos para garantizar la continuidad del servicio.

HDFS y Docker

Como en el caso de la ingesta, se utilizó Docker con HDFS para crear un entorno aislado, portable y escalable que facilite tanto el desarrollo como la implementación en distintos entornos. Luego se añadió al archivo ‘docker-compose.yml’ mencionado anteriormente toda la configuración necesaria para gestionar los servicios y contenedores de HDFS. Cada uno de los servicios, como el NameNode y los DataNodes, se define como un contenedor independiente dentro del archivo Compose. Estos servicios se coordinan entre sí mediante una red interna, lo que permite la comunicación eficiente entre los nodos del clúster de HDFS.

Almacenamiento

Para el almacenamiento eficiente en HDFS se ha utilizado la biblioteca *hdfs* de Python.

Su configuración comienza estableciendo la conexión con el sistema de archivos distribuido mediante un cliente inseguro (*InsecureClient*). Este cliente se conecta a una URL de HDFS específica (*HDFS_URL*) y realiza operaciones utilizando el usuario ‘root’.

Luego se verifica que el directorio designado para almacenar los datos en HDFS exista. Si el directorio no se encuentra presente, el sistema lo crea. Este proceso implica la verificación del directorio mediante una solicitud de estado (*client.status*). En caso de no existir, se crea el directorio y todos sus directorios padre necesarios (*client.makedirs*).

Una vez configurada la conexión y verificados los permisos, los datos capturados en tiempo real desde Kafka se procesan y se guardan en archivos JSON dentro de HDFS. Cada mensaje recibido del tópico de Kafka se convierte en un archivo de datos que se guarda en el directorio previamente definido. Los archivos son nombrados de manera dinámica usando un timestamp, lo que asegura que cada archivo tenga un nombre único, facilitando la gestión de los

mismos.


Si la escritura es exitosa, se notifica al usuario mediante mensajes de consola. En caso de que HDFS no esté disponible o ocurra un error durante la escritura, los datos se guardan localmente en un directorio especificado (*LOCAL_DIR*), asegurando que ningún dato se pierda incluso si HDFS no está accesible en ese momento.

Es importante destacar que en caso de que la conexión con HDFS falle o si se presentan problemas durante la escritura, el sistema gestiona los errores y proporciona mensajes claros de lo que ha ocurrido, lo que permite una rápida identificación de problemas.

Monitoreo

Al trabajar con datos en streaming es fundamental tener sistema robusto de monitoreo que permita saber en cada momento el estado actual del clúster.

En este caso, se utilizó la biblioteca *rich* para crear una interfaz interactiva en la terminal que facilita la visualización del flujo de datos y el estado del sistema. Se muestra en tiempo real el estado del procesamiento y almacenamiento de los datos. A medida que los datos se reciben desde Kafka, la aplicación actualiza una tabla en la terminal con información relevante como el *timestamp* de cada mensaje procesado y el archivo guardado. La interfaz también hace uso de colores y estilos para mejorar la legibilidad de los datos, permitiendo una rápida identificación de los eventos importantes.

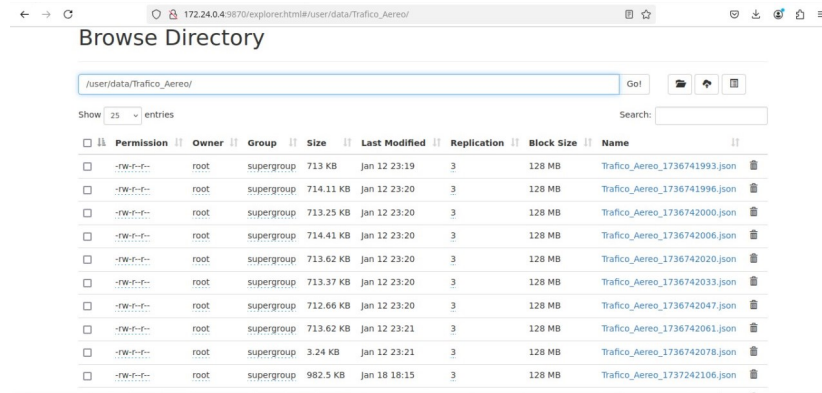


Timestamp	Last File Saved
2025-01-18 18:16:20	HDFS: Trafico_Aereo_1737242180.json
2025-01-18 18:16:25	HDFS: Trafico_Aereo_1737242185.json
2025-01-18 18:16:29	HDFS: Trafico_Aereo_1737242189.json
2025-01-18 18:16:34	HDFS: Trafico_Aereo_1737242194.json
2025-01-18 18:16:38	HDFS: Trafico_Aereo_1737242198.json
2025-01-18 18:16:43	HDFS: Trafico_Aereo_1737242203.json
2025-01-18 18:16:48	HDFS: Trafico_Aereo_1737242208.json
2025-01-18 18:16:55	HDFS: Trafico_Aereo_1737242215.json

Figure 1: Salida en consola

En paralelo a la monitorización en la terminal, el sistema también está respaldado por la web de Hadoop, que ofrece una visión detallada del estado del clúster HDFS. A través de esta interfaz web, se puede obtener información crítica sobre el estado general del sistema de archivos distribuido. La web proporciona métricas clave sobre el estado del clúster, como la cantidad de nodos activos y la distribución del almacenamiento entre los DataNodes. Esto permite a los administradores del sistema asegurarse de que todos los componentes del clúster

están operando correctamente y que hay suficiente capacidad de almacenamiento disponible para manejar el volumen de datos generado.



Browse Directory

/user/data/Trafico_Aereo/

Show 25 entries

Search:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	root	supergroup	713 KB	Jan 12 23:19	3	128 MB	Trafico_Aereo_1736741993.json
-rw-r--r--	root	supergroup	714.11 KB	Jan 12 23:20	3	128 MB	Trafico_Aereo_1736741996.json
-rw-r--r--	root	supergroup	713.25 KB	Jan 12 23:20	3	128 MB	Trafico_Aereo_1736742000.json
-rw-r--r--	root	supergroup	714.41 KB	Jan 12 23:20	3	128 MB	Trafico_Aereo_1736742006.json
-rw-r--r--	root	supergroup	713.62 KB	Jan 12 23:20	3	128 MB	Trafico_Aereo_1736742020.json
-rw-r--r--	root	supergroup	713.37 KB	Jan 12 23:20	3	128 MB	Trafico_Aereo_1736742033.json
-rw-r--r--	root	supergroup	712.66 KB	Jan 12 23:20	3	128 MB	Trafico_Aereo_1736742047.json
-rw-r--r--	root	supergroup	713.62 KB	Jan 12 23:21	3	128 MB	Trafico_Aereo_1736742061.json
-rw-r--r--	root	supergroup	3.24 KB	Jan 12 23:21	3	128 MB	Trafico_Aereo_1736742078.json
-rw-r--r--	root	supergroup	982.5 KB	Jan 18 18:15	3	128 MB	Trafico_Aereo_1737242106.json

Figure 2: Web de hadoop

Además, la web de Hadoop ofrece detalles sobre el uso del espacio de almacenamiento en HDFS, lo que permite monitorear el crecimiento de los datos y evitar posibles cuellos de botella. Otra característica importante de la interfaz web es la capacidad de visualizar estadísticas sobre el rendimiento del sistema, como la latencia de las operaciones de lectura y escritura y el número de operaciones completadas por segundo. Si ocurre algún problema en el clúster, la interfaz web de Hadoop muestra alertas y notificaciones, lo que permite a los administradores identificar rápidamente los problemas y tomar las acciones necesarias para resolverlos.

References

- [1] Matthias Schäfer, Martin Strohmeier, Vincent Lenders, Ivan Martinovic, and Matthias Wilhelm. *Bringing Up OpenSky: A Large-scale ADS-B Sensor Network for Research*. In Proceedings of the 13th IEEE/ACM International Symposium on Information Processing in Sensor Networks (IPSN), pages 83–94, April 2014.
- [2] Apache Kafka. *Apache Kafka Documentation*. Available at: <https://kafka.apache.org/documentation/>. Accessed: December 2024.
- [3] Apache Hadoop. *HDFS User Guide*. Available at: <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html>. Accessed: January 2025.
- [4] Kafka Python. *Kafka-Python Documentation*. Available at: <https://kafka-python.readthedocs.io/en/master/>. Accessed: December 2024.