# Kafka: implementación con Docker y Python

Pablo Carrera Flórez de Quiñones
José Llanes Jurado

2019-05-27

## Contenidos





# 1 Introducción

Apache Kafka es un proyecto de intermediación de mensajes de código abierto desarrollado por la Apache Software Foundation escrito en Java y Scala. El proyecto tiene como objetivo proporcionar una plataforma unificada, de alto rendimiento y de baja latencia para la manipulación en tiempo real de fuentes de datos. Puede verse como una cola de mensajes, bajo el patrón publicación-suscripción, masivamente escalable concebida como un registro de transacciones distribuidas, lo que la vuelve atractiva para las infraestructuras de aplicaciones empresariales.

En este trabajo se presenta una aplicación basada en Apache Kafka que consume datos en tiempo real de Twitter y produce una serie de analíticas sobre ellos. La aplicación de Kafka se ha realizado en Docker para asegurar su reproducibilidad y portabilidad y las funcionalidades de los productores y consumidores externos se ha programado en python. La organización del trabajo es la siguiente: en la sección 2 se explica el proceso que se ha seguido con Docker para obtener el entorno de desarrollo usado; en la sección 3 se explica la configuración de Kafka utilizada; en la sección 4 se explica el código utilizado para los productores y los consumidores de Kafka; en la sección 5 se estudia un ejemplo de uso de esta aplicación; y finalmente en la sección 6 se realizan algunos comentarios sobre la aplicación y algunas propuestas adicionales.

# 2 Docker: preparación

Para desarrollar el trabajo de forma conjunta y asegurar su reproducibilidad vamos a usar Docker, esta herramienta nos permite montar aplicaciones sobre contenedores independientes que pueden ser exportados e importados fácilmente, lo que supone una gran ventaja sobre las instalaciones locales ya que aísla las dependencias internas y elimina las barreras entre sistemas operativos.

Para nuestro trabajo necesitamos dos aplicaciones:

- Kafka: la aplicación que constituye el grueso de nuestro proyecto, como ya hemos dicho, se basa en un esquema de publicación-subscripción para el manejo de datos en tiempo real. Vamos a usar la imagen `wurstmeister/kafka`, ya que es la más utilizada por la comunidad.
- Zookeeper: actúa como servicio centralizado y se usa para mantener la configuración y la sincronización de los servicios distribuidos, controla el estado de los nodos de Kafka, los topics y las particiones. Vamos a usar la imagen `wurstmeister/zookeeper`, ya que nos asegura una compatibilidad total con la imagen de Kafka que vamos utilizar.

Lo primero que hacemos es abrir un terminal de Docker y clonar el repositorio donde se encuentran estas aplicaciones,

```
dinop@LAPTOP-C3SF38PP MINGW64 /c/Program Files/Docker Toolbox
$ cd ~/Desktop/BigData_Kafka

dinop@LAPTOP-C3SF38PP MINGW64 ~/Desktop/BigData_Kafka
$ git clone https://github.com/wurstmeister/kafka-docker.git

dinop@LAPTOP-C3SF38PP MINGW64 ~/Desktop/BigData_Kafka
$ cd ~/Desktop/BigData_Kafka/kafka-docker
```

si obtenemos algun error en esto proceso quizá sea necesario ejecutar el comando `git config --global core.autocrlf false` para habilitar la clonación. Cargamos las imágenes de las aplicaciones,

```
dinop@LAPTOP-C3SF38PP MINGW64 ~/Desktop/BigData_Kafka/kafka-docker
$ docker pull wurstmeister/zookeeper

dinop@LAPTOP-C3SF38PP MINGW64 ~/Desktop/BigData_Kafka/kafka-docker
$ docker pull wurstmeister/kafka
```

de forma que ya tenemos todo preparado para establecer la configuración de Kafka.

# 3 Kafka: configuración

Para el desarrollo de nuestra aplicación debemos preparar la conexión entre los diferentes componentes de esta de forma adecuada, esto se realiza mediante la creación de un archivo *docker-compose.yml*, que contiene las definiciones de las variables de entorno de las aplicaciones.

## 3.1 Configuración básica

Antes de proceder con la configuración de final de nuestra aplicación creemos que puede ser ilustrativo ver como generar aplicaciones más sencillas. De forma que como primer ejemplo podemos probar a construir una aplicación de Kafka con un solo broker, que tome datos desde dentro nuestro contenedor de docker. Con esta idea en mente creamos el siguiente archivo *docker-compose.yml*:

```
services:
  zookeeper:
    image: wurstmeister/zookeeper
    ports:
      - "2181:2181"
  kafka:
    image: wurstmeister/kafka
    build: .
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: 192.168.99.100
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

donde vemos que basta con especificar `KAFKA_ADVERTISED_HOST_NAME`, el nombre del equipo, y `KAFKA_ZOOKEEPER_CONNECT`, el puerto de conexión de Kafka con Zookeper. Para componer la aplicación hacemos

```
dinop@LAPTOP-C3SF38PP MINGW64 ~/Desktop/Tarea_Kafka/kafka-docker (master)
$ docker-compose up -d --build
```

Una vez hecho esto podemos crear un topic y asociarle un productor y un consumidor local. Para ello abrimos dos nuevas terminales de Docker y usamos los comandos habituales de Kafka para estas labores, en la primera hacemos

```
dinop@LAPTOP-C3SF38PP MINGW64 /c/Program Files/Docker Toolbox
$ cd ~/Desktop/BigData_Kafka/kafka-docker

dinop@LAPTOP-C3SF38PP MINGW64 ~/Desktop/BigData_Kafka/kafka-docker (master)
$ docker exec -it kafkadocker_kafka_1 bash
bash-4.4# kafka-topics.sh \
--create \
--zookeeper zookeeper:2181 \
--replication-factor 1 \
--partitions 1 \
--topic example0

bash-4.4# kafka-console-producer.sh \
--broker-list 192.168.99.100:9092 \
--topic example0
```

y en la segunda

```
dinop@LAPTOP-C3SF38PP MINGW64 /c/Program Files/Docker Toolbox
$ cd ~/Desktop/BigData_Kafka/kafka-docker

dinop@LAPTOP-C3SF38PP MINGW64 ~/Desktop/BigData_Kafka/kafka-docker (master)
$ docker exec -it kafkadocker_kafka_1 bash
bash-4.4# kafka-console-consumer.sh \
--bootstrap-server 192.168.99.100:9092 \
--topic example0 \
--from-beginning
```

de forma que podemos introducir mensajes a mano en la primera, el productor, y ver como van siendo mostrados en la segunda, el consumidor.

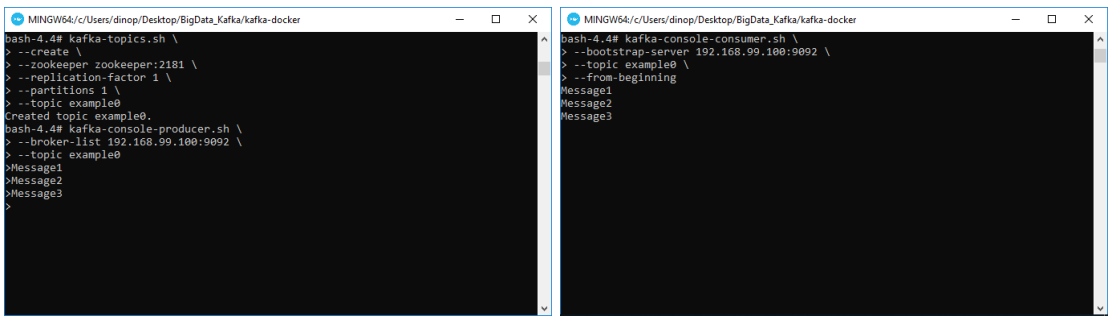


Figura 1: Ejemplo de consumidor y productor mediante terminal.

## 3.2 Configuración avanzada

Como hemos comentado anteriormente, la idea es que nuestra aplicación tome datos en tiempo real de Twitter mediante un script de Python, esto se ejecutará en local, no en Docker, de forma que debemos hacer que Kafka pueda comunicarse con nuestro ordenador. Trabajar de esta forma requiere de la siguiente configuración

```
version: '3'
services:
  zookeeper:
    image: wurstmeister/zookeeper
    ports:
      - "2181:2181"
  kafka:
    image: wurstmeister/kafka
    build: .
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: 192.168.99.100
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://192.168.99.100:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

donde hemos especificado una serie de variables adicionales, que comunican el contenedor de Kafka en Docker con nuestra máquina local.

## 3.3 Configuración utilizada

Este archivo todavía puede modificarse un poco más para incluir varios brokers y crear automáticamente los topics que vamos a utilizar. Esto ayuda a que la aplicación de Kafka este autocontenida y solo tengamos actuar con los scritps de Python, sin tener que acceder a Kafka para personalizar la configuración. Vamos a crear dos topics:

- *tweetsraw*: contendrá los tweets consumidos directamente de Twitter, sin ningún procesamiento. Vamos a usar 5 particiones y un factor de replicación 2 para mejorar la tolerancia a fallos del sistema.
- *tweetscurated*: contendrá los tweets limpios, preparados para que los consumidores finales puedan usarlos con un procesamiento mínimo. Vamos a usar también 5 particiones y un factor de replicación 2 para mejorar la tolerancia a fallos del sistema.

de forma que la configuración final queda como sigue:

```
version: '3'
```

```
services:
  zookeeper:
    image: wurstmeister/zookeeper
    ports:
      - "2181:2181"
  kafka:
    image: wurstmeister/kafka
    build: .
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: 192.168.99.100
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://192.168.99.100:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_CREATE_TOPICS: "tweetsraw:5:1,tweetscurated:5:1"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

# 4 Python: productores y consumidores

## 4.1 Productor de tweets crudos

El primer script que hemos desarrollado, *producer.py*, tiene como función obtener datos en tiempo real de Twitter. Para ello se ha utilizado Tweepy, la API de Twitter para Python, que proporciona funciones que permiten capturar tweets en Streaming una vez fijadas unas palabras clave para filtrar. Para hacer que este script esté operativo basta con introducir nuestras credenciales de desarrollador en Twitter, a saber `consumer_key`, `consumer_secret`, `access_token` y `access_token_secret`, y las palabras que queremos usar como filtros a la hora de capturar tweets, `to_track`.

```
# Import libraries
from tweepy.streaming import StreamListener
from tweepy import OAuthHandler, Stream
from kafka import KafkaProducer
import json

# Set API keys for Twitter
consumer_key =  ""
consumer_secret =  ""
access_token = ""
access_token_secret =  ""

auth = OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)

# Create the Kafka producer
producer = KafkaProducer(bootstrap_servers = ["192.168.99.100:9092"],
                         value_serializer=lambda v: json.dumps(v).encode("utf-8"))
```

```
# Create the stream object
class StdOutListener(StreamListener):
    def on_data(self, data):
        print(data)
        producer.send("tweetsraw", data)
        return True
    def on_error(self, status):
        print (status)

# Make it work
to_track = [""]

l = StdOutListener()
stream = Stream(auth, l)
stream.filter(track = to_track)
```

Vemos que el objeto `StdOutListener` que hemos creado envía los tweets directamente a Kafka a través del productor `producer` mediante el comando `porducer.send(topic, message)`. Es importante notar que todo esto ocurre en tiempo real, y que el productor no empieza a enviar mensajes hasta que con se corre el script.

## 4.2 Productor-consumidor de tweets limpios

El segundo script que hemos desarrollado, *consumer-producer.py*, tiene la función de consumir los tweets crudos almacenados en el topic *tweetsraw*, obtener de ellos una serie de campos como el texto, la fecha o el usuario, y introducirlos de nuevo en Kafka en el topic *tweetscurated*. Obtendremos además una serie de datos técnicos de los mensajes almacenados, el topic, la partición y el offset, para poder analizar como se distribuye el trabajo dentro de Kafka.

```
# Import libraries
from kafka import KafkaConsumer, KafkaProducer
import json
from datetime import datetime
from email.utils import parsedate_tz, mktime_tz

# Create the consumer and the producer
consumer = KafkaConsumer("tweetsraw",
                         bootstrap_servers = ["192.168.99.100:9092"],
                         value_deserializer = lambda x: json.loads(x.decode("utf-8"))
                         )

producer = KafkaProducer(bootstrap_servers = ["192.168.99.100:9092"],
                         value_serializer = lambda x: json.dumps(x).encode("utf-8"))

# Define some specific functions
def to_local_time(tweet_time_string):
    timestamp = mktime_tz(parsedate_tz(tweet_time_string))
    return datetime.fromtimestamp(timestamp).strftime('%Y-%m-%d %H:%M:%S')

# Create the curated tweet
for tweet in consumer:
```

```
    tweet_raw = json.loads(tweet.value)

    tweet_curated = {"topic" : tweet.topic,
                     "partition" : tweet.partition,
                     "offset" : tweet.offset,
                     "date" : tweet_raw["created_at"],
                     "text" : tweet_raw["text"],
                     "user" : tweet_raw["user"]["screen_name"]
                     }

    # Fix the data and round to minute
    tweet_curated["date"] = to_local_time(tweet_curated["date"])
    tweet_curated["date"] = re.sub("\d\d$", "00", tweet_curated["date"])

    print(tweet_curated)
    producer.send("tweetscurated", tweet_curated)
```

Donde vemos que el cosumidor `consumer` consume los mensajes que hemos producido en la sección anterior. De igual manera que la sección anterior, es importante notar que todo el proceso ocurre en tiempo real, el consumidor no empieza a consumir hasta que no se corre el script. Una versión alternativa de esto sería empezar a consumir los mensajes generados por el productor desde el inicio, pero dado que la idea de la aplicación es funcionar en streaming mantendremos la configuración como hemos mostrado.

## 4.3 Consumidores finales

Finalmente, podemos crear varios consumidores adicionales que consuman los mensajes del topic *tweetscurated* y nos propocionen cierta información sobre los tweets capturados. Dado que vamos a obtener un gran número de tweets podemos extraer estadísticas de uso, como pueden ser el número de tweets publicados por minuto en relación con las palabras que estamos usando para filtrar o el ranking de usuarios más activos publicando sobre ese tema.

Nuestro primer consumidor, como hemos dicho consistirá en un script que consumirá los mensajes del topic *tweetscurated*, los convierta en un DataFrame y mediante una agrupación por minutos nos proporcione el número de tweets publicados en ese minuto. Para correr este script quizá sea necesario usar el comando magic de Python `%matplotlib` en la terminal para que pueda obtenerse una representación interactiva.

```
# Import libraries
from kafka import KafkaConsumer
from datetime import datetime
import json
import re
import pandas as pd
import matplotlib.pyplot as plt

# Create the consumer
consumer = KafkaConsumer("tweetscurated",
                         bootstrap_servers = ["192.168.99.100:9092"],
                         value_deserializer = lambda x: json.loads(x.decode("utf-8"))
                         )
```

```
# Consume the tweets into a dataframe
tweets = pd.DataFrame(columns = ["topic", "partition", "offset", "date", "text", "user"])

# Create the interactive figure
fig = plt.figure()
plt.ion()

for tweet in consumer:
    # Convert the tweet to a dataframe
    tweet_curated = tweet.value
    tweet_curated = {key : [tweet_curated[key]] for key in tweet_curated.keys()}
    tweet_curated = pd.DataFrame(tweet_curated)

    # Fix the date to the minute
    tweet_curated.loc[0, "date"] = re.sub("\d\d$", "00", tweet_curated.loc[0, "date"])
    tweet_curated.loc[0, "date"] = datetime.strptime(tweet_curated.loc[0, "date"], "%Y-%m-%d %H:%M

    # Create the dataframe for visualization
    tweets = tweets.append(tweet_curated)
    tweets_grouped = tweets.groupby(by = "date").size()

    # Update the plot with the new data
    fig.clear()
    plt.plot(tweets_grouped)
    plt.xticks(rotation = 45)
    plt.title("Tweets publicated per minute", weight = "bold")
    plt.pause(0.001)
```

Nuestro segundo consumidor, consistirá, de forma análoga al anterior, en un script que consumirá los mensajes del topic *tweetscurated*, los convierta en un DataFrame y mediante una agrupación por minutos nos proporcione un ranking de los usuarios más activos. Para correr este script quizá también sea necesario usar el comando magic de Python `%matplotlib` en la terminal de Python.

```
# Import libraries
from kafka import KafkaConsumer
from datetime import datetime
import json
import re
import pandas as pd
import matplotlib.pyplot as plt

# Create the consumer
consumer = KafkaConsumer("tweetscurated",
                         bootstrap_servers = ["192.168.99.100:9092"],
                         value_deserializer = lambda x: json.loads(x.decode("utf-8"))
                         )

# Consume the tweets into a dataframe
tweets = pd.DataFrame(columns = ["topic", "partition", "offset", "date", "text", "user"])

# Create the interactive figure
fig = plt.figure()
plt.ion()
```

```
for tweet in consumer:
    # Convert the tweet to a dataframe
    tweet_curated = tweet.value
    tweet_curated = {key : [tweet_curated[key]] for key in tweet_curated.keys()}
    tweet_curated = pd.DataFrame(tweet_curated)

    # Fix the date to the minute
    tweet_curated.loc[0, "date"] = re.sub("\d\d$", "00", tweet_curated.loc[0, "date"])
    tweet_curated.loc[0, "date"] = datetime.strptime(tweet_curated.loc[0, "date"], "%Y-%m-%d %H:%M

    # Create the dataframe for visualization
    tweets = tweets.append(tweet_curated)
    tweets_grouped = tweets.groupby(by = "user").size()
    tweets_grouped = tweets_grouped.sort_values(ascending = False)
    tweets_grouped = tweets_grouped[:10][::-1]

    # Update the plot with the new data
    tweets_grouped.plot(x = "user", y = "count", kind = "barh", zorder = 2, width = 0.85, legend =
    plt.title("Most active users", weight = "bold")
    plt.pause(0.001)
```

# 5 Ejemplo de uso

Como ejemplo de uso vamos a usar la configuración anteriormente mencionada para muestrear la actividad de Twitter un día cualquiera por la mañana. Para ello primero, usando el fichero *docker-compose.yml* detallado en la seccion 3.3 abrimos una terminal de docker y ejecutamos los siguientes comando para arrancar Kafka:

```
dinop@LAPTOP-C3SF38PP MINGW64 /c/Program Files/Docker Toolbox
$ cd ~/Desktop/BigData_Kafka/kafka-docker

dinop@LAPTOP-C3SF38PP MINGW64 ~/Desktop/Tarea_Kafka/kafka-docker (master)
$ docker-compose up -d --build
```

de forma que ya tenemos Kafka activo y con los topics disponibles. El siguiente paso es ejecutar los scripts detallados anteriormente, nosotros para ello hemos usado Spyder. Abrimos una terminal de Python para cada uno de ellos:

- *producer.py*: hacemos `to_track = ["España"]` para muestrear que se comenta sobre el país a estas horas y ejecutamos. Es imprescindible añadir aquí las claves necesarias para poder acceder a Twitter.
- *consumer-producer.py*: simplemente ejecutamos, la limpieza de los tweets se produce de forma automática.
- *consumer.py*: ejecutamos el comando `%matplotlib` y ejectuamos el script para obtener una gráfica en tiempo real de los tweets publicados por minuto.

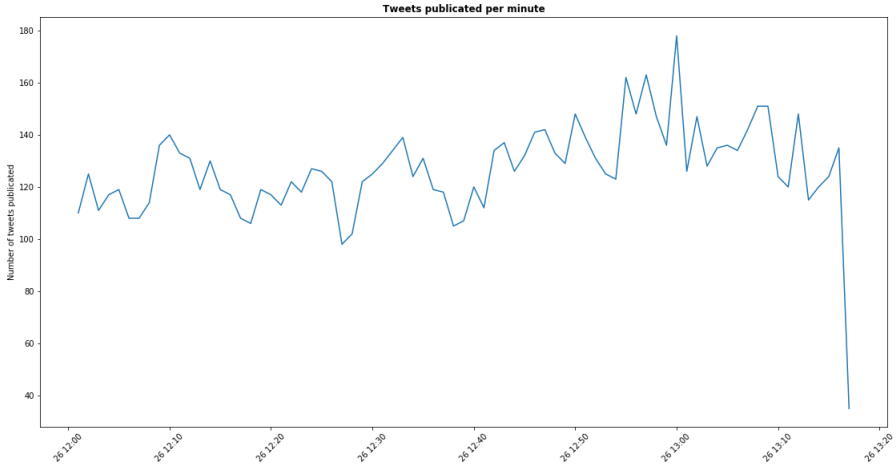


Figura 2: Ejemplo de uso de la aplicación propuesta para muestrear la palabra `"España"` en Twitter. Se muestran los tweets publicados por minuto. El periodo de muestre ha sido de aproximadamente dos horas.

- *consumer2.py*: ejecutamos el comando `%matplotlib` y ejectuamos el script para obtener una gráfica en tiempo real de los usuarios más activos desde que se ha iniciado la aplicación.

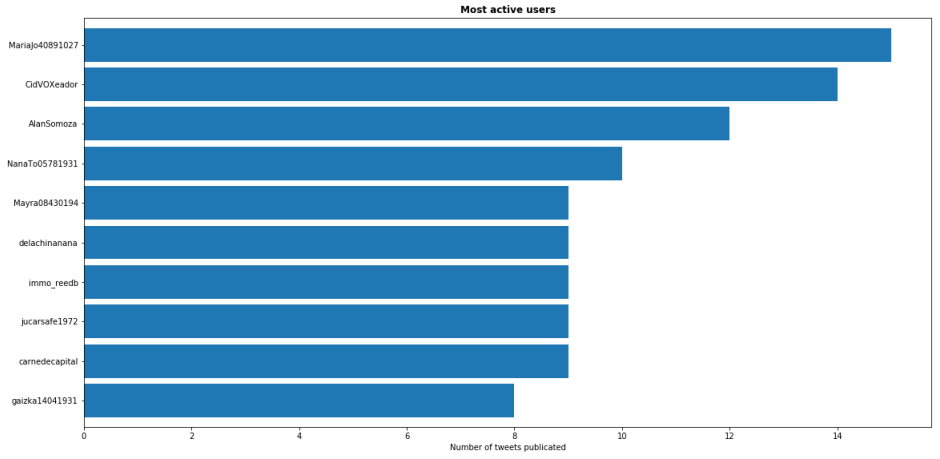


Figura 3: Ejemplo de uso de la aplicación propuesta para muestrear la palabra `"España"` en Twitter. Se muestran usuarios más activos desde que se inició la aplicación. El periodo de muestre ha sido de aproximadamente dos horas.

# 6 Comentarios adicionales

En este trabajo hemos realizado una implementación de la herramienta libre de procesado de datos en tiempo real Apache Kafka mediante Docker y Python. Primero hemos preparado la aplicación de Kafka mediante el uso de dos imágenes de Docker, una para Zookeeper y otra para Kafka, de forma totalmente reproducible, para después pasar a manejar esta únicamente mediante

Python. Esta forma de preparar nuestra aplicación hace que no sea necesario volver a interaccionar con Kafka directamente, de hecho, existen comandos adicionales dentro de la librería kafka-python que permiten crear y eliminar topics de forma directa desde dentro de Python. En esta parte creemos que es necesario destacar otra vez que Kafka no se está ejecutando en la máquina local, sino en la máquina virtual que genera Docker, lo que hace que nuestra aplicación esté totalmente preparada para ser escalada en la nube con unos cambios mínimos.

Finalmente proponemos un ejemplo de uso de esta aplicación en el que mostramos la evolución de la palabra `"España"` en Twitter, mostrando tanto el número de tweets por segundo, como el top 10 de usuarios que emplean esta palabra ordenados por frecuencia. Posibles funcionalidades adicionales de esta aplicación podrían ser el análisis de sentimiento en tiempo real de los tweets publicados o la creación de una red de usuarios en tiempo real atendiendo a las respuestas de unos a otros.

# 7 Recursos

Para la realización de este trabajo se ha hecho un gran uso de la documentación proporcionada por las diferentes herramientas que hemos utilizado:

- Docker: `https://docs.docker.com/`
- Docker images:
  - Zookeeper: `https://hub.docker.com/r/wurstmeister/zookeeper`
  - Kafka: `https://hub.docker.com/r/wurstmeister/kafka`
- Kafka: `https://kafka.apache.org/documentation/`
- Python: `https://docs.python.org/3/`
- Python libraries:
  - kafka-python: `https://kafka-python.readthedocs.io/en/master/`
  - tweepy: `https://tweepy.readthedocs.io/en/v3.5.0/`