UD 7 - Apache Kafka y Spark

1. Introducción

Para entender y aprender como podemos usar Kafka y Spark en un sistema de Big Data, lo vamos a realizar mediante un ejemplo

2. Ejemplo 6. Kafka y Spark

Imagina que vas a trabajar para un gran equipo de futbol en el análisis de datos para obtener business intelligence para que durante los partidos, podamos servir y dar soporte al stack técnico.

Vamos a realizar un pequeño ejemplo de concepto, donde vamos a generar datos sintéticos de las posiciones de los jugadores durante el partido, para seguidamente obtener información como: distancia recorrida, velocidades, sprints, etc; en tiempo real, para que ayude al stack técnico en su toma de decisiones.

Vamos a establecer un cluster de Kafka para recoger los eventos en tiempo real, que dispondrá los datos para que Spark los recoja en tiempo real en un dataset y pueda aplicarle Pandas para obtener estos datos.

> ✓ **Configuración cluster Kafka**
>
> Vamos a **configurar el cluster de Kafka usando mode KRaft** teniendo en cuenta que estamos en un **entorno de pruebas y no de producción**, para este ejemplo de concepto, vamos a configurar un *cluster con 3 servidores (en un sólo nodo o máquina) con **1 controller y 2 brokers***

2.1 Requisitos

Vamos a configurar un cluster las siguientes características:

- 1 topic jugadores con 2 particiones
- 1 factor de replica de 2
- 1 nodo con 2 brokers y 1 controller

2.2 Configuración del Clúster de Kafka

- Consideraciones previas:

1. Vamos a establecer todos los archivos de configuración en una carpeta de ejemplo llamada ejemplo6, que en mi caso estará alojada en `/opt/kafka/ejemplo6`
2. Dado que todas las instancias se ejecutarán en el mismo nodo, es crucial asignar puertos únicos y directorios de log para cada broker y el controller.

- Configuración:

1. Para el controller, debemos usar como base la configuración de propiedades de controller de kafka (KRaft mode) que se encuentran `config/kraft/controller.properties`
2. Para cada broker, necesitaremos crear un archivo de configuración por separado. Para ello debemos usar como base la configuración de propiedades de brokers de kafka que se encuentran `config/kraft/broker.properties`
3. Creamos los directorios necesarios para nuestro `ejemplo6`

```
mkdir -p /opt/kafka/ejemplo6/config
mkdir -p /opt/kafka/ejemplo6/logs
```

4. Hacemos 2 y 1 copia de los ficheros correspondientes de configuración para cada uno

```
cp config/kraft/controller.properties /opt/kafka/ejemplo6/config/controller1.properties
cp config/kraft/broker.properties /opt/kafka/ejemplo6/config/broker1.properties
cp config/kraft/broker.properties /opt/kafka/ejemplo6/config/broker2.properties
```

5. Asignamos la configuración al controller

controller1.properties

```
# Server Basics
process.roles=controller
node.id=1
controller.quorum.voters=1@localhost:9093
# Socket Server Settings
listeners=CONTROLLER://localhost:9093
controller.listener.names=CONTROLLER
# Log Basics
log.dirs=/opt/kafka/ejemplo6/logs/controller1
```

6. Asignamos la siguiente configuración para cada broker

Spark como consumer (nodo master de Spark)

Hay que tener en cuenta:

- Vamos a conectarnos con un cliente(consumer) que va a consumir datos en streaming con SPARK
- Por seguridad, kafka no permite esa conexión del cliente(consumer) desde localhost
- Teniendo en cuenta nuestra configuración del cluster (ip del master de Spark es 192.168.11.10), debemos configurarla correctamente en los brokers para que tengan acceso.
- La conexión se hace a través del master de spark, y la ejecución de la aplicación en Spark se reparte entre los workers del cluster, aunque estén en diferentes ips

broker1

broker1.properties

```
# Server Basics
process.roles=broker
node.id=2
controller.quorum.voters=1@localhost:9093
# Socket Server Settings
listeners=PLAINTEXT://192.168.11.10:9094
inter.broker.listener.name=PLAINTEXT
advertised.listeners=PLAINTEXT://192.168.11.10:9094
listener.security.protocol.map=CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SAS
L_SSL
# Log Basics
log.dirs=/opt/kafka/ejemplo6/logs/broker1
```

broker2

broker2.properties

```
# Server Basics
process.roles=broker
node.id=3
controller.quorum.voters=1@localhost:9093
# Socket Server Settings
listeners=PLAINTEXT://192.168.11.10:9095
inter.broker.listener.name=PLAINTEXT
advertised.listeners=PLAINTEXT://192.168.11.10:9095
# Log Basics
log.dirs=/opt/kafka/ejemplo6/logs/broker2
```

Configuración con varios consumers

En el caso de que tuviéramos que configurar más de un cliente en diferentes máquinas, tendríamos que cambiar la configuración de nuestros brokers. Recordando para que sirven las siguiente propiedades:

- `listeners`: son las interfaces a las que se conecta Kafka.
- `advertised.listeners`: es la forma en que los clientes pueden conectarse.
- `listener.security.protocol.map`: define pares clave/valor para el protocolo de seguridad a utilizar por nombre de listener.

1) Opción 1

Abrir el acceso en `listeners` configurando la ip como `0.0.0.0:puerto`. Esto no sería la mejor solución, ya que debemos filtrar el acceso desde `advertised.listeners` y este valor, por configuración propia de Kafka, **debe ser un subconjunto de** `listeners`

2) Opción 2

Indicamos por ip cada uno de los interfaces que damos acceso (**solo un interface por puerto**) a los brokers de Kafka (`listeners`), y filtramos (`advertised.listeners`). Pero no hay que olvidar indicar el protocolo por el cual te conectas, que hay que indicarlo en `listener.security.protocol.map`, que también debe ser único. Imaginando que tenemos varios clientes de acceso, la configuración debería ser en ese caso

broker1

broker1.properties

```
# Server Basics
process.roles=broker
node.id=2
controller.quorum.voters=1@localhost:9093
# Socket Server Settings
listeners=BROKER://localhost:9094,LISTENER_CLIENTE1://nodo1:11094,LISTENER_CLIENTE2://nodo2:12094,LISTENER_CLIENTE3://nodo3:1309
inter.broker.listener.name=BROKER
advertised.listeners=BROKER://localhost:9094,LISTENER_CLIENTE1://nodo1:11094,LISTENER_CLIENTE2://nodo2:12094,LISTENER_CLIENTE3:/
listener.security.protocol.map=CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL,
NTEXT
# Log Basics
log.dirs=/opt/kafka/ejemplo6/logs/broker1
```

broker2

broker2.properties

```
# Server Basics
process.roles=broker
node.id=3
controller.quorum.voters=1@localhost:9093
# Socket Server Settings
listeners=BROKER://localhost:9095,LISTENER_CLIENTE1://nodo1:11095,LISTENER_CLIENTE2://nodo2:12095,LISTENER_CLIENTE3://nodo3:1309
inter.broker.listener.name=BROKER
advertised.listeners=BROKER://localhost:9095,LISTENER_CLIENTE1://nodo1:11095,LISTENER_CLIENTE2://nodo2:12095,LISTENER_CLIENTE3:/
listener.security.protocol.map=CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL,
NTEXT
# Log Basics
log.dirs=/opt/kafka/ejemplo6/logs/broker2
```

7. Iniciamos Kafka con KRaft

```
#Genera un cluster UUID
KAFKA_CLUSTER_ID="$(bin/kafka-storage.sh random-uuid)"
echo $KAFKA_CLUSTER_ID

#Formateamos los directorios de log
bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c /opt/kafka/ejemplo6/config/controller1.properties
bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c /opt/kafka/ejemplo6/config/broker1.properties
bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c /opt/kafka/ejemplo6/config/broker2.properties
```

8. Iniciamos los server(1 controller y 2 brokers) cada uno en una terminal

```
#Ejecuta el servidor Kafka
bin/kafka-server-start.sh /opt/kafka/ejemplo6/config/controller1.properties
bin/kafka-server-start.sh /opt/kafka/ejemplo6/config/broker1.properties
bin/kafka-server-start.sh /opt/kafka/ejemplo6/config/broker2.properties
```

2.3 Creación del Topic

1. Creamos el topic con factor de replica 2 y 2 particiones. El topic debe conectarse a un broker.

```
bin/kafka-topics.sh --create --topic player-position --bootstrap-server 192.168.11.10:9094 --replication-factor 2 --partitions 2
```

2. Podemos ver la descripción del topic creado

```
bin/kafka-topics.sh --describe --topic player-position --bootstrap-server 192.168.11.10:9094
```

```
Topic: player-position  TopicId: DDEeIaCeQMaewIuwSBNQtw PartitionCount: 2   ReplicationFactor: 2    Configs: segment.bytes=1073741824
    Topic: player-position  Partition: 0    Leader: 3   Replicas: 3,2   Isr: 3,2    Elr:    LastKnownElr:
    Topic: player-position  Partition: 1    Leader: 2   Replicas: 2,3   Isr: 2,3    Elr:    LastKnownElr:
```

2.4 Productor

1. Creamos el programa que va a generar los datos sintéticos.
2. Vamos a intentar simular datos lo más reales posibles a los movimientos de un jugador de futbol.
3. Suponemos que el campo es de 100x65 metros
4. El punto (0,0) será la esquina inferior izquierda y el punto (100,65) será la esquina superior derecha.
5. Generaremos 10 eventos por segundo

generate_synthetic_data.py

```python
import random
import time
from datetime import datetime, timedelta
import math

# Parámetros del campo y jugadores
FIELD_LENGTH = 100
FIELD_WIDTH = 65
NUM_PLAYERS = 11
TIMESTAMPS_PER_SECOND = 10
DURATION_MINUTES = 45

# Velocidades en metros por segundo
WALK_SPEED = 1.4
RUN_SPEED = 5.0
SPRINT_SPEED = 8.0

def generate_initial_positions():
    """Genera posiciones iniciales razonables para los jugadores."""
    positions = {}
    for player_id in range(1, NUM_PLAYERS + 1):
        if player_id == 1:
            # Portero
            positions[player_id] = (random.uniform(0, 5), random.uniform(FIELD_WIDTH / 2 - 5, FIELD_WIDTH / 2 + 5))
        else:
            # Jugadores de campo
            positions[player_id] = (random.uniform(0, FIELD_LENGTH), random.uniform(0, FIELD_WIDTH))
    return positions

def move_player(position, speed):
    """Mueve al jugador a una nueva posición basada en la velocidad y dirección."""
    angle = random.uniform(0, 360)  # Dirección aleatoria
    distance = speed / TIMESTAMPS_PER_SECOND
    delta_x = distance * math.cos(math.radians(angle))
    delta_y = distance * math.sin(math.radians(angle))

    new_x = max(0, min(FIELD_LENGTH, position[0] + delta_x))
    new_y = max(0, min(FIELD_WIDTH, position[1] + delta_y))

    return (new_x, new_y)

def generate_player_position_data(match_id, player_id, position):
    timestamp = datetime.utcnow().isoformat()
    return {
        "match_id": match_id,
        "player_id": player_id,
        "timestamp": timestamp,
        "position_x": position[0],
        "position_y": position[1]
    }

def simulate_player_movement(positions):
    """Simula el movimiento de los jugadores en el campo."""
    new_positions = {}
    for player_id, position in positions.items():
        if player_id == 1:
```

```
            # Portero: se mueve dentro del área de gol
            new_positions[player_id] = move_player(position, random.uniform(0, WALK_SPEED))
        else:
            # Jugadores de campo: pueden caminar, correr o sprintar
            speed = random.choice([WALK_SPEED, RUN_SPEED, SPRINT_SPEED])
            new_positions[player_id] = move_player(position, speed)
    return new_positions

def main():
    positions = generate_initial_positions()
    end_time = datetime.utcnow() + timedelta(minutes=DURATION_MINUTES)

    while datetime.utcnow() < end_time:
        for _ in range(TIMESTAMPS_PER_SECOND):
            for player_id in range(1, NUM_PLAYERS + 1):
                data = generate_player_position_data("match_1", player_id, positions[player_id])
                print(data)  # Aquí enviaríamos los datos a Kafka en lugar de imprimirlos
            positions = simulate_player_movement(positions)
            time.sleep(1 / TIMESTAMPS_PER_SECOND)

if __name__ == "__main__":
    main()
```

2. Creamos el productor con KafkaProducer

producer_positions.py

```
from kafka import KafkaProducer
import json
from generate_synthetic_data import generate_player_position_data, generate_initial_positions,
simulate_player_movement
import time
from datetime import datetime, timedelta

# Parámetros del campo y jugadores
NUM_PLAYERS = 11
TIMESTAMPS_PER_SECOND = 10
DURATION_MINUTES = 45

def main():
    match_id = "match_" + datetime.utcnow().strftime("%Y%m%d%H%M%S")
    producer = KafkaProducer(bootstrap_servers='192.168.11.10:9094',
                             value_serializer=lambda v: json.dumps(v).encode('utf-8'))

    positions = generate_initial_positions()
    end_time = datetime.utcnow() + timedelta(minutes=DURATION_MINUTES)

    while datetime.utcnow() < end_time:
        for _ in range(TIMESTAMPS_PER_SECOND):
            for player_id in range(1, NUM_PLAYERS + 1):
                data = generate_player_position_data(match_id, player_id, positions[player_id])
                producer.send('player-position', value=data)
                print(f"Sent: {data}")
            positions = simulate_player_movement(positions)
            time.sleep(1 / TIMESTAMPS_PER_SECOND)

if __name__ == "__main__":
    main()
```

3. Aprovecharemos la *Consumer API* de Kafka para ver está consumiendo los datos correctamente una vez lanzada la aplicación

```
bin/kafka-console-consumer.sh --topic player-position --from-beginning --bootstrap-server 192.168.11.10:9094
```

2.5 Configuración del Clúster de Spark

Aprovecharemos las máquinas master y los 3 nodos que ya tenemos preparadas para lanzar el cluster de Spark. Consulta la documentación de Spark para cualquier duda

1. Código del programa
    a. Consumirá los eventos en tiempo real de kafka
    b. Lo almacena en un dataset
    c. Obtenemos ETL a través de Pandas
    d. Mostramos el resultado por consola:
        i. El calculo de cada microBatch
        ii. El acumulado del (último) partido.

2. Para una correcta configuración, nos basaremos en la documentación oficial de Apache Spark y su integración con Kafka: Structured Streaming + Kafka Integration Guide

Spark_PlayerPositionProcessing.py

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, from_json
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, FloatType
import pandas as pd
import numpy as np

# Definir el esquema de los datos
schema = StructType([
    StructField("match_id", StringType(), True),
    StructField("player_id", IntegerType(), True),
    StructField("timestamp", StringType(), True),
    StructField("position_x", FloatType(), True),
    StructField("position_y", FloatType(), True)
])

# Crear la sesión de Spark
spark = SparkSession.builder \
    .appName("KafkaDataConsumer") \
    .getOrCreate()

# Set Spark logging level to ERROR to avoid varios otros logs en consola.
spark.sparkContext.setLogLevel("ERROR")

# Leer los datos de Kafka
df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "192.168.11.10:9094") \
    .option("subscribe", "player-position") \
    .option("startingOffsets", "earliest") \
    .load()

player_position_df = df.selectExpr("CAST(value AS STRING)") \
    .select(from_json(col("value"), schema).alias("data")) \
    .select("data.*")

# Crear un DataFrame acumulativo vacío
accumulated_schema = StructType([
    StructField("match_id", StringType(), True),
    StructField("player_id", IntegerType(), True),
    StructField("total_distance", FloatType(), True)
])
accumulated_df = spark.createDataFrame([], accumulated_schema)

# Variable global para el `match_id` actual
current_match_id = None

# Definir una función para procesar cada microbatch
def process_batch(df, epoch_id):
    global accumulated_df, current_match_id

    # Convertir a Pandas DataFrame
    pd_df = df.toPandas()

    # Obtener el `match_id` del batch actual
    if not pd_df.empty:
        match_id = pd_df['match_id'].iloc[0]
        current_match_id = match_id

    # Calcular la distancia recorrida por cada jugador
    pd_df['prev_position_x'] = pd_df.groupby('player_id')['position_x'].shift(1)
    pd_df['prev_position_y'] = pd_df.groupby('player_id')['position_y'].shift(1)

    pd_df['distance'] = np.sqrt((pd_df['position_x'] - pd_df['prev_position_x']) ** 2 +
                                (pd_df['position_y'] - pd_df['prev_position_y']) ** 2)

    pd_df['distance'] = pd_df['distance'].fillna(0)

    # Sumar la distancia total por jugador
    total_distance_df = pd_df.groupby(['match_id', 'player_id'])['distance'].sum().reset_index()

    # Convertir de nuevo a DataFrame de Spark
    spark_df = spark.createDataFrame(total_distance_df)

    # Unir con el DataFrame acumulativo
    accumulated_df = accumulated_df.union(spark_df).groupBy("match_id", "player_id").agg({"total_distance":
"sum"}).withColumnRenamed("sum(total_distance)", "total_distance")

    # Filtrar el acumulado para el `match_id` actual
```

```
    current_match_df = accumulated_df.filter(col("match_id") == current_match_id)

    # Mostrar el resultado del batch actual y el acumulado del último partido
    print("Batch Result:")
    spark_df.show()
    print(f"Resultado acumulado para match_id = {current_match_id}:")
    current_match_df.show()

# Escribir los datos procesados usando el método foreachBatch
query = player_position_df \
    .writeStream \
    .foreachBatch(process_batch) \
    .outputMode("append") \
    .start()

query.awaitTermination()
```

2. Lanzamos spark master y los workers del **cluster**.

```
hadoop@master:/opt/hadoop-3.4.1/spark-3.5.4$ ./sbin/start-master.sh
hadoop@master:/opt/hadoop-3.4.1/spark-3.5.4$ ./sbin/start-workers.sh
```

3. Recuerda que debes tener instalado correctamente pandas. Si no es el caso, hay que instalarlo en cada uno de los nodos

```
pip3 install pandas
pip3 install pyarrow
```

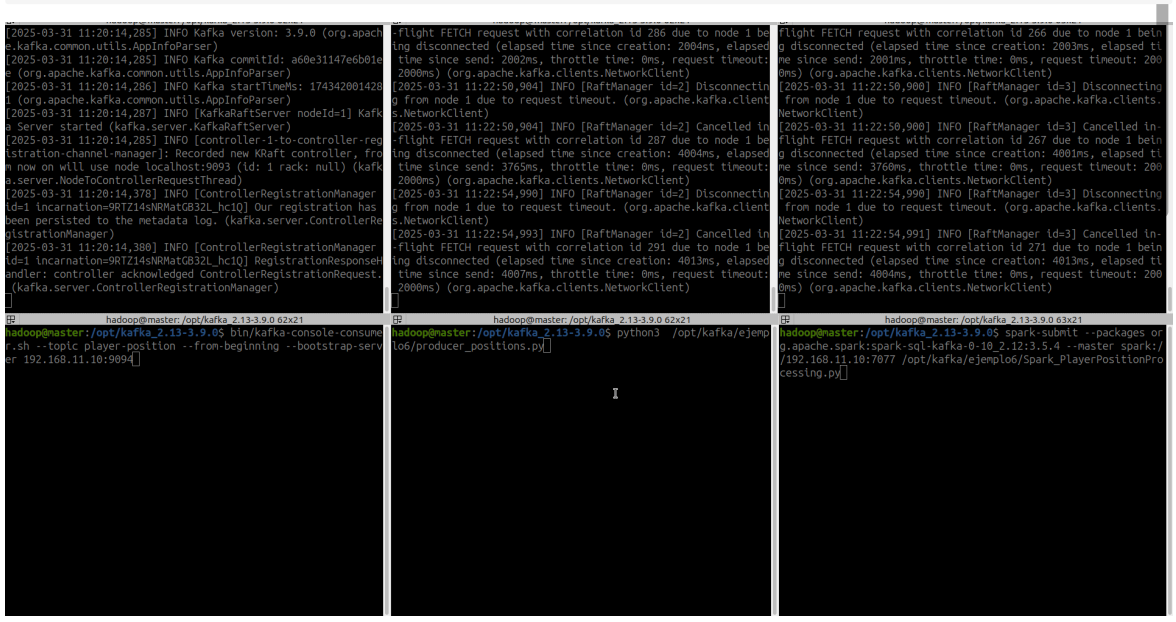
4. Lanzamos el programa. Siguiendo la documentación oficial, añadimos el paquete del conector de Kafka

```
spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.4 --master spark://192.168.11.10:7077
/opt/kafka/ejemplo6/Spark_PlayerPositionProcessing.py
```

2.6 Ejecución de la aplicación

1. Lanzamos el productor

```
python3  /opt/kafka/ejemplo6/producer_positions.py
```



Animación 7.1_Kafka_Spark: Ejemplo 6

2.7 Persistencia de datos

Si queremos realizar una persistencia de estos datos, sólo tendríamos que almacenarlos **usando algunas de las ya explicadas a lo largo del módulo**, o cualquier otra que sea conveniente para el escenario en concreto (HDFS, Sistema de ficheros, S3, MongoDB,...).

2.8 Conclusiones

A partir de esta configuración, podemos añadir ETL para obtener más conocimiento sobre los jugadores, como velocidades, sprints, mapas de calor, etc, que podría ayudar al stack técnico en diferentes tomas de decisiones (sistema de juego, rendimiento de jugadores, cansancio acumulado,...).

Podríamos añadir información adicional si pudiéramos capturar los datos del balón, que combinados con la de los jugadores, podríamos obtener otros datos valiosos de aplicación de business intelligence para ofrecérselos al stack técnico, tales como: estadísticas de pases, disparos, zonas de mayor juego,...