

UD 7 - Apache Kafka

En los últimos años el número de aplicaciones a desarrollar por las empresas ha aumentado considerablemente con la llegada de las arquitecturas basadas en microservicios y sistemas de Big Data.

Uno de los aspectos más relevantes es la comunicación entre ellos, o la necesidad de tener que integrarse con otros sistemas enviando o recibiendo información. En estos casos, **estas comunicaciones deberán ser rápidas, seguras y fiables con una alta disponibilidad.**

Una de las soluciones para solventar este tipo de casos han supuesto el **uso de tecnologías basadas en colas de mensajes**, las cuales permiten la **comunicación asíncrona**, lo que significa que los puntos de conexión que producen y consumen los mensajes interactúan con la cola, no entre sí. Además, ayudan a simplificar de forma significativa la escritura de código para aplicaciones desacopladas, mejorando el rendimiento, la fiabilidad y la escalabilidad.

A la hora de utilizar un sistema de colas de mensajes que requieren **la transmisión de datos a tiempo real** encontramos **Apache Kafka** como una de nuestras mejores soluciones.

Según Apache Kafka, más del 80% de todas las empresas del 100 Fortune confían y usan Kafka. Puedes ver la lista en su [página oficial](#)

1. ¿Qué es Apache Kafka?

1.1 Definición

Apache Kafka es un **sistema de transmisión de datos distribuido con capacidad de escalado horizontal y tolerante a fallos**. Gracias a su alto rendimiento nos permite transmitir datos en tiempo real utilizando el patrón de mensajería **publish/subscribe**.



Figura 7.1_Kafka: Logo Kafka. (Fuente: kafka.apache.org)

Kafka fue creado por LinkedIn y actualmente es un proyecto **open source** mantenido por **Confluent**, empresa que está bajo la administración de Apache. Apache Kafka está escrito en Java y Scala y se distribuye con licencia Apache 2.0

Proporciona la plataforma para **publicar y suscribirse a flujos de eventos y permite almacenar estos eventos de una forma tolerante a fallos, escalable, persistente y con capacidad de replicación.**

Además de almacenar estos eventos, la funcionalidad se extiende **con la capacidad de procesarlos en tiempo real, a medida que se reciben de múltiples fuentes de datos**. Kafka se integra con numerosas tecnologías, de esta forma, permite construir flujos de datos en tiempo real entre distintos sistemas y aplicaciones de manera desacoplada.

Por estas razones, Apache Kafka es muy popular en la arquitectura Big Data de muchas empresas en la actualidad. Debido a su arquitectura, consigue obtener una **baja latencia, escalabilidad horizontal y absorber los picos de carga que pueden ocurrir en el sistema**.

Hoy día, existe una gran variedad de fuentes generadoras de datos, tales como microservicios, bases de datos, todo tipo de IoTs, servidores web, registros de logs, análisis de rendimiento y monitoreo, etc. **Kafka interviene en este punto para simplificar la arquitectura: vincula a todos los productores de datos con Kafka y, a su vez, Kafka se encarga de distribuir estos datos a los consumidores pertinentes, facilitando así un ecosistema de datos más ágil y desacoplado. Todo ello de una forma tolerante a fallos, escalable, persistente, con capacidad de replicación, baja latencia, escalabilidad horizontal y con capacidad de absorción de los picos de carga.**

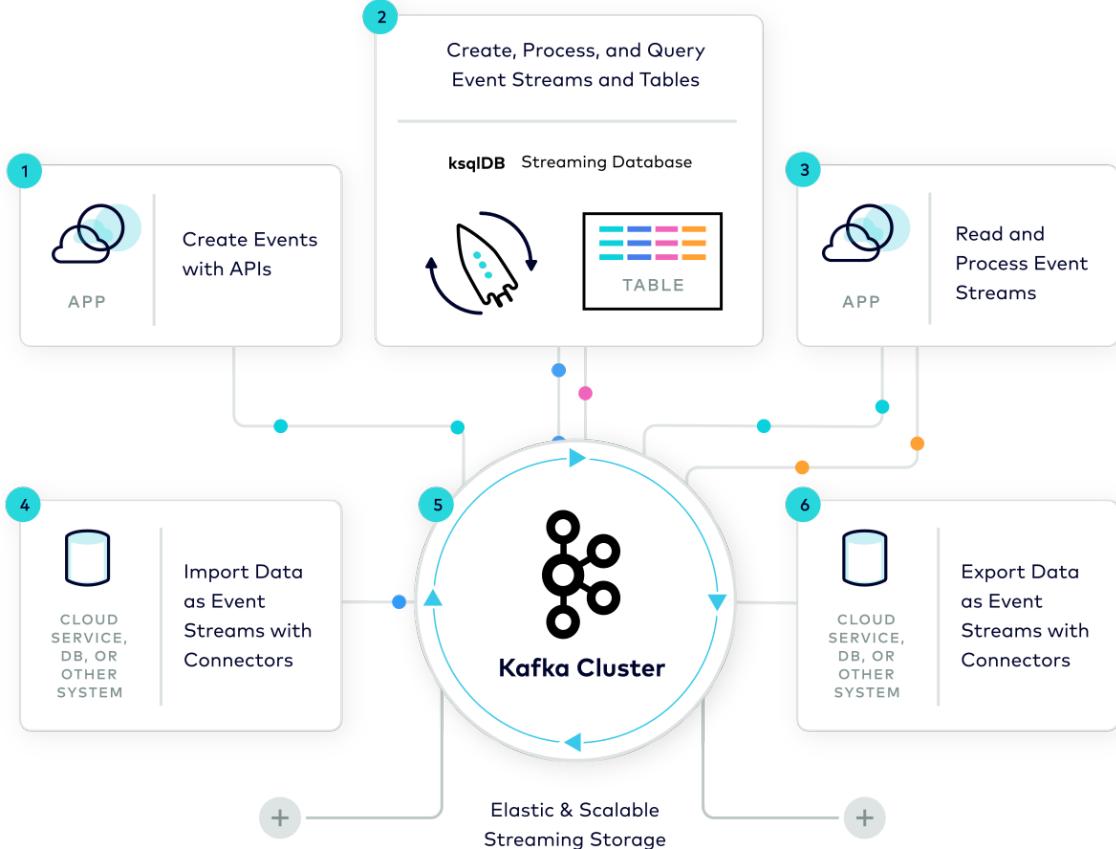


Figura 7.2_Kafka: Qué hace Apache Kafka (Fuente: developer.confluent.io)

1.2 Plataforma de transmisión de eventos

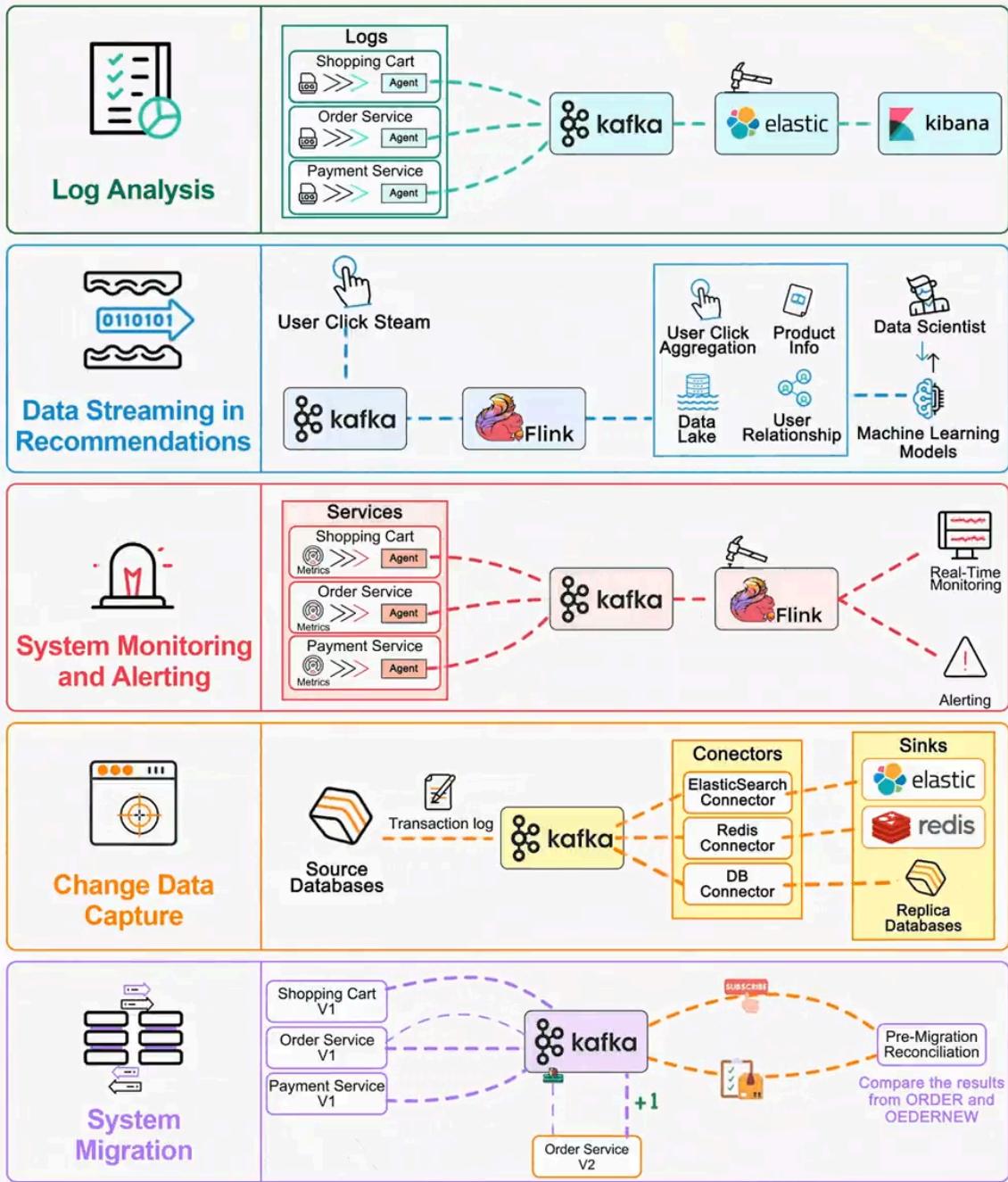
Apache Kafka® es una plataforma de transmisión de eventos. ¿Qué significa eso?

Kafka combina tres capacidades clave para que pueda implementar sus **casos de uso** para la transmisión de eventos de un extremo a otro con una única solución:

1. Para **publicar** (escribir) y **suscribirse** (leer) a transmisiones de eventos, incluida la importación/exportación continua de sus datos desde otros sistemas.
2. Para **almacenar** transmisiones de eventos de forma duradera y confiable durante el tiempo que desee.
3. Para **procesar** flujos de eventos a medida que ocurren o de forma retrospectiva.

Y toda esta funcionalidad se proporciona de forma distribuida, altamente escalable, elástica, tolerante a fallos y segura. Kafka se puede implementar en hardware básico, máquinas virtuales y contenedores, tanto en las instalaciones como en la nube. Puede elegir entre autoadministrar sus entornos Kafka y utilizar servicios totalmente administrados ofrecidos por una variedad de proveedores.

Top 5 Kafka Use Cases



Animación 7.1_Kafka: Top 5 de casos de uso de Kafka (Fuente: bytebytogo.com)

2. Elementos Kafka

Apache Kafka está formado por una serie de conceptos y elementos que describiremos a continuación

2.1 Eventos

Un **evento** (También llamados *registros* o *mensajes*) registra el hecho de que "algo sucedió" en el mundo o en su negocio. Cuando lees o escribes datos en Kafka, lo haces en forma de eventos. Conceptualmente, un evento tiene una **clave**, un **valor**, una **marca de tiempo(timestamp)** y encabezados de metadatos opcionales. Aquí hay un evento de ejemplo:

- **Clave del evento:** "Alicia"
 - **Valor del evento:** "Se realizó un pago de \$200 a Bob"
 - **Marca de tiempo del evento:** "25 de marzo de 2024 a las 14:06"

Eventos de ejemplo son transacciones de pago, actualizaciones de geolocalización desde teléfonos móviles, pedidos de envío, mediciones de sensores desde dispositivos IoT o equipos médicos, y mucho más.

2.2 Producers/Consumers

Los **producers**(productores) son aquellas aplicaciones cliente que publican (escriben) eventos en Kafka, y los **consumers**(consumidores) son aquellos que se suscriben (leen y procesan) estos eventos. En Kafka, los productores y los consumidores están completamente desacoplados y son agnósticos entre sí, lo cual es un elemento de diseño clave para lograr la alta escalabilidad por la que Kafka es conocido. Por ejemplo, los productores nunca necesitan esperar a los consumidores. Kafka ofrece varias garantías, como la capacidad de **procesar eventos exactamente una vez (events exactly-once)**.

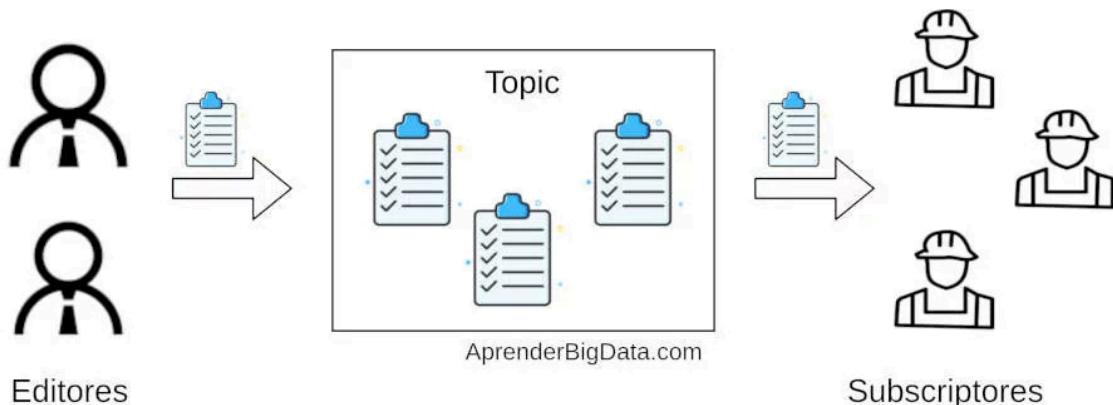


Figura 7.3_Kafka: producers/consumers. (Fuente: aprenderbigdata.com)

2.3 Topics

Los **eventos** se organizan y almacenan de forma duradera en **topics** (temas). De manera muy simplificada, un topic es similar a una carpeta en un sistema de archivos y los eventos son los archivos en esa carpeta. Un nombre de topic de ejemplo podría ser "pagos/payments". Los topics en Kafka son siempre multiproductor y multisuscriptor: un topic puede tener cero, uno o

muchos productores que le escriban eventos, así como cero, uno o muchos consumidores que se suscriban a estos eventos.

Los eventos de un topic se pueden leer tantas veces como sea necesario; a diferencia de los sistemas de mensajería tradicionales, los eventos no se eliminan después de su consumo. En su lugar, se define durante cuánto tiempo Kafka debe retener sus eventos a través de una configuración por topic, después del cual se descartarán los eventos antiguos. El rendimiento de Kafka es efectivamente constante con respecto al tamaño de los datos, por lo que almacenar datos durante un período prolongado también podría ser correcto.

2.4 Partitions

Los topics están divididos, lo que significa que un topic se distribuye en varios "bucket(depósitos)" ubicados en diferentes **brokers** (veremos más adelante) de Kafka.

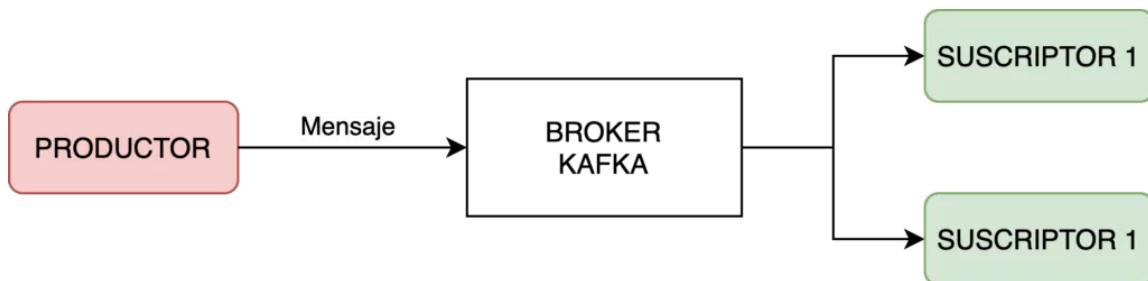


Figura 7.4_Kafka: Broker (Fuente: aprenderbigdata.com)

Esta ubicación distribuida de sus datos es muy importante para la escalabilidad porque permite que las aplicaciones cliente lean y escriban datos desde/hacia muchos brokers al mismo tiempo. Cuando se publica un nuevo evento en un topic, en realidad se agrega a una de las particiones del topic. Los eventos con la misma clave de evento (por ejemplo, un ID de cliente o de vehículo) se escriben en la misma partición, y Kafka garantiza que cualquier consumidor de una partición de topic determinada siempre leerá los eventos de esa partición exactamente en el mismo orden en que fueron escritos.

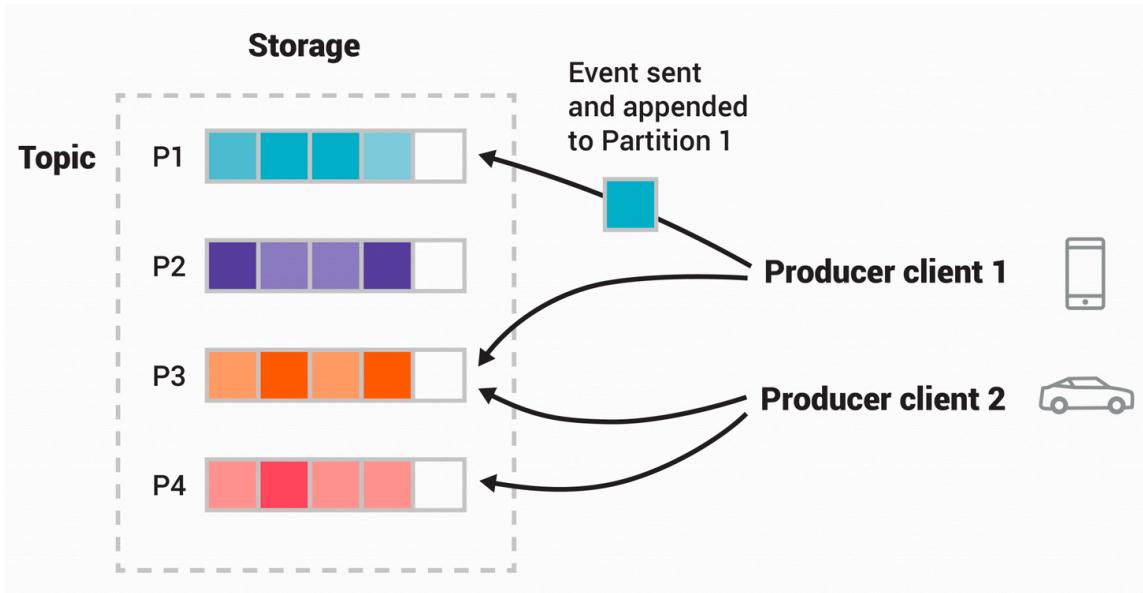


Figura 7.5_Kafka: Ejemplo de un topic de 4 particiones (Fuente: kafka.apache.org)

Este topic de ejemplo tiene cuatro particiones P1–P4. Dos clientes productores diferentes publican, independientemente uno del otro, nuevos eventos para el topic escribiendo eventos a través de la red en las particiones del topic. Los eventos con la misma clave (indicados por su color en la figura) se escriben en la misma partición. Tengamos en cuenta que ambos productores pueden escribir en la misma partición si corresponde.

2.5 Replication

Para que los datos sean tolerantes a fallos y tengan alta disponibilidad, **cada topic se puede replicar**, incluso entre regiones geográficas o centros de datos, de modo que siempre haya varios intermediarios que tengan una copia de los datos en caso de que algo salga mal, hacer mantenimiento a los brokers, etc. Una configuración de producción común es un factor de replica 3, es decir, siempre habrá tres copias de sus datos. Esta replica se realiza a nivel de particiones de topics.

3. Estructura

3.1 Topics, mensajes y particiones

Podemos crear tantos topics como queramos y estos serán identificados por su nombre. Los topics pueden dividirse en particiones. Cada elemento que se almacena en un topic se denomina evento. Éstos son inmutables y son añadidos a una partición determinada (específica definida por la clave del mensaje o mediante round-robin en el caso de ser nula) en el orden el que fueron enviados, es decir, se garantiza el orden dentro de una partición pero no entre ellas.

Cada **evento** dentro de una **partición** tiene un **identificador numérico incremental** llamado **offset**. Con este mecanismo, se puede identificar un evento con el nombre del topic que lo contiene, la partición y el offset.

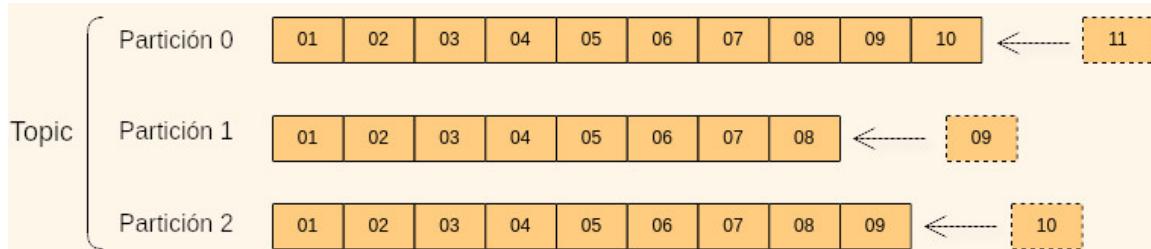


Figura 7.6_Kafka: Topic Kafka (partición y offset) (Fuente: paradigmadigital.com)

3.2 Brokers y Topics

Un clúster de Kafka consiste en uno o más servidores denominados Kafka **brokers**. Cada broker es identificado por un ID (integer) y contiene ciertas particiones de un topic, no necesariamente todas.

Además, permite replicar y particionar dichos topics balanceando la carga de almacenamiento entre los brokers. **Esta característica permite que Kafka sea tolerante a fallos y escalable.**

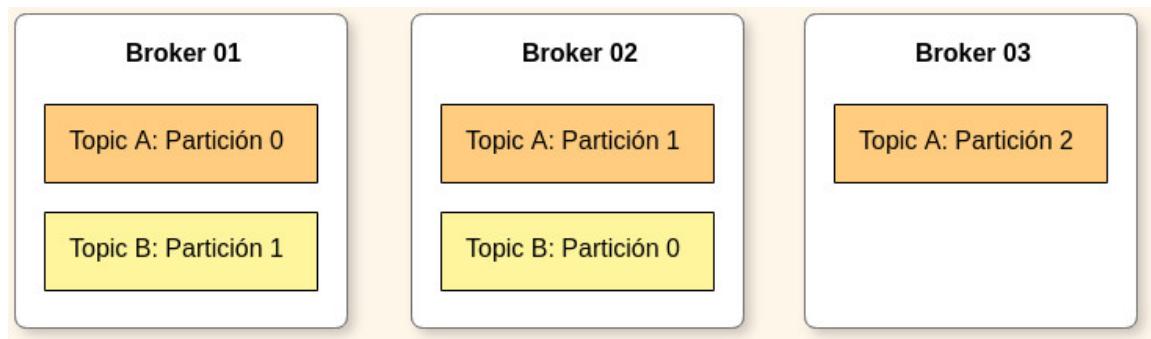


Figura 7.7_Kafka: Brokers (Fuente: paradigmadigital.com)

El número de particiones debería ser siempre igual o mayor que el número de brokers. Si no, habrá brokers que no estén consumiendo mensajes.

3.3 Topic replication

Los topics deberán tener un factor de replica mayor a 1 (normalmente 2 y 3), de esta forma si un broker se cae, otro broker puede servir los datos.

En cada momento **sólo puede haber un broker líder** para cada partición de un topic. Sólo el líder puede recibir y servir datos de una partición, mientras tanto los otros brokers sincronizarán sus datos como seguidores. Si este se cae, se cambia el líder.

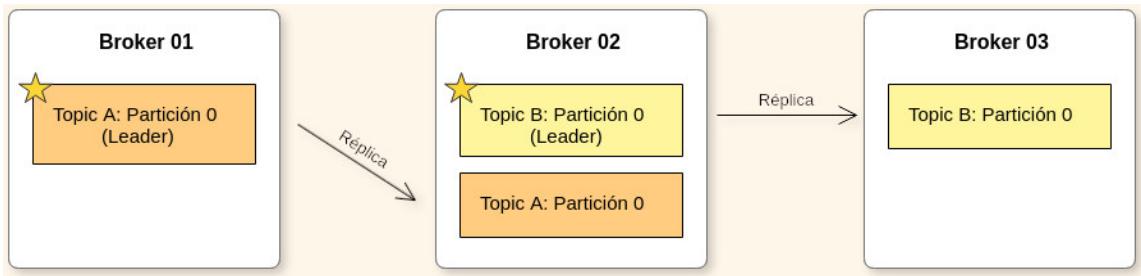


Figura 7.8_Kafka: Topic replication (Fuente: paradigmadigital.com)

3.4 Grupos de consumidores

Los consumidores de Kafka son los clientes conectados suscritos a los topics que consumen los mensajes. Cada consumidor tiene asociado un **grupo de consumidores**. Kafka garantiza que cada mensaje sólo es leído por un consumidor de cada grupo.

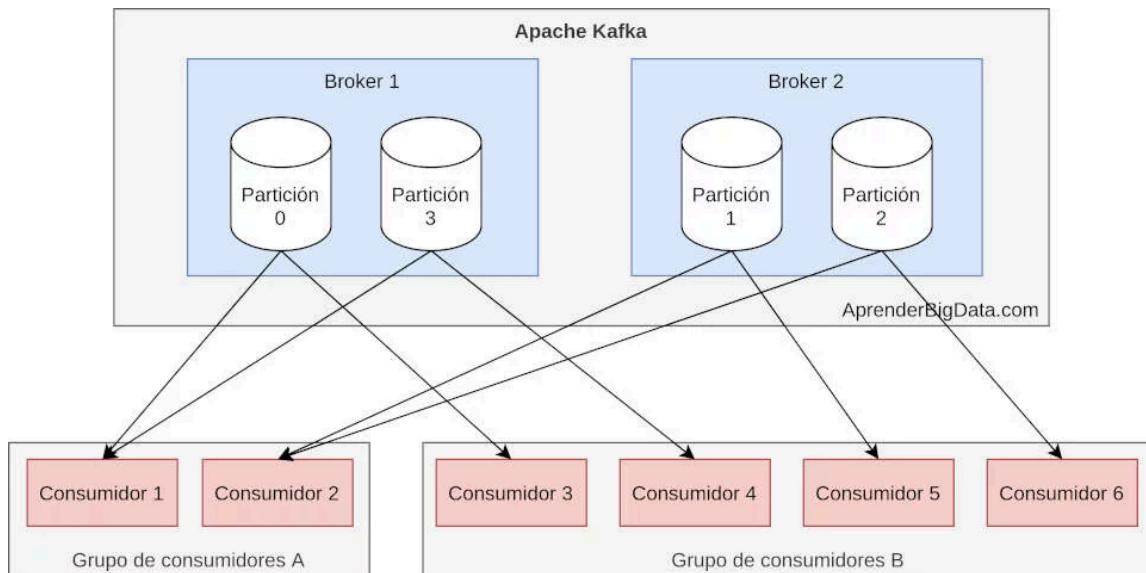


Figura 7.9_Kafka: Grupos de consumidores (Fuente: aprenderbigdata.com)

En la imagen, un cluster de kafka está formado por dos brokers con 4 particiones con dos grupos de consumidores. En el grupo de consumidores A, Kafka asigna dos particiones a cada consumidor, mientras que en el grupo B, al haber 4 consumidores para 4 particiones, es posible asignar una partición a cada consumidor.



Video resumen

Si no te ha quedado claro, puedes ver el [siguiente video](#) que explica brevemente estos conceptos

3.5 Gestión de Sistemas distribuidos

Para ayudar en la gestión del cluster de Kafka en este sistema distribuido, existen 2 posibles soluciones:

Zookeeper

Entre los nodos de Zookeeper, se elige uno como líder. El resto de nodos del clúster se denominan seguidores, y uno de ellos es elegido como nuevo líder en el caso de que el líder actual tenga un fallo. Generalmente, los clusters de Zookeeper se despliegan con 3 ó 5 nodos.

Para gestionar un clúster de Kafka, **Zookeeper almacena información del estado del clúster**: detalles de los topics como el nombre, las particiones, las réplicas y los grupos de consumidores.



Figura 7.10_Kafka: Logo Zookeeper (Fuente: zookeeper.apache.org)

En el momento en el que Zookeeper detecta que uno de los brokers de Kafka está caído, realiza las siguientes acciones:

1. Elige un nuevo broker para tomar el lugar del broker caído.
2. Actualiza los metadatos para la distribución de carga de los productores y los consumidores para que no exista pérdida de servicio.

Tras estas acciones, se pueden volver a escribir y leer mensajes con normalidad.

KRaft

Se propone [eliminar la dependencia de Kafka con Zookeeper](#) reemplazándola con un mecanismo interno. KRaft funciona como un mecanismo de consenso para el protocolo de quorum. Kafka 3.3 marca este mecanismo como listo para producción. A partir de la versión 3.5 se proporciona un mecanismo de migración y se marca el soporte a Zookeeper como deprecated.

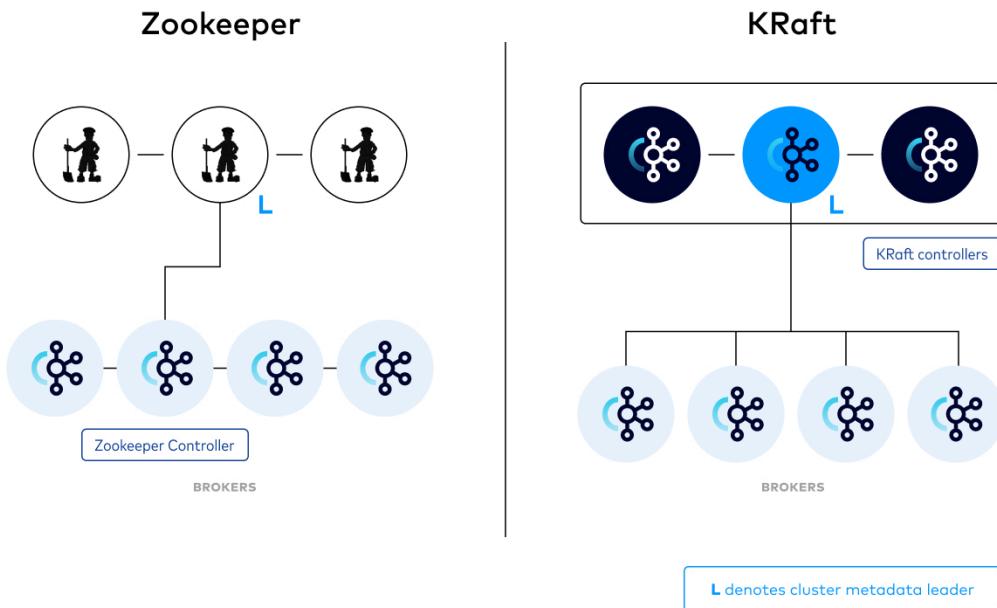


Figura 7.11_Kafka: Diagrama Kraft (Fuente: kafka.apache.org)

4. Kafka APIs

Una vez comentada la estructura de Apache Kafka, vamos a ver cómo se interactúa con él mediante APIs

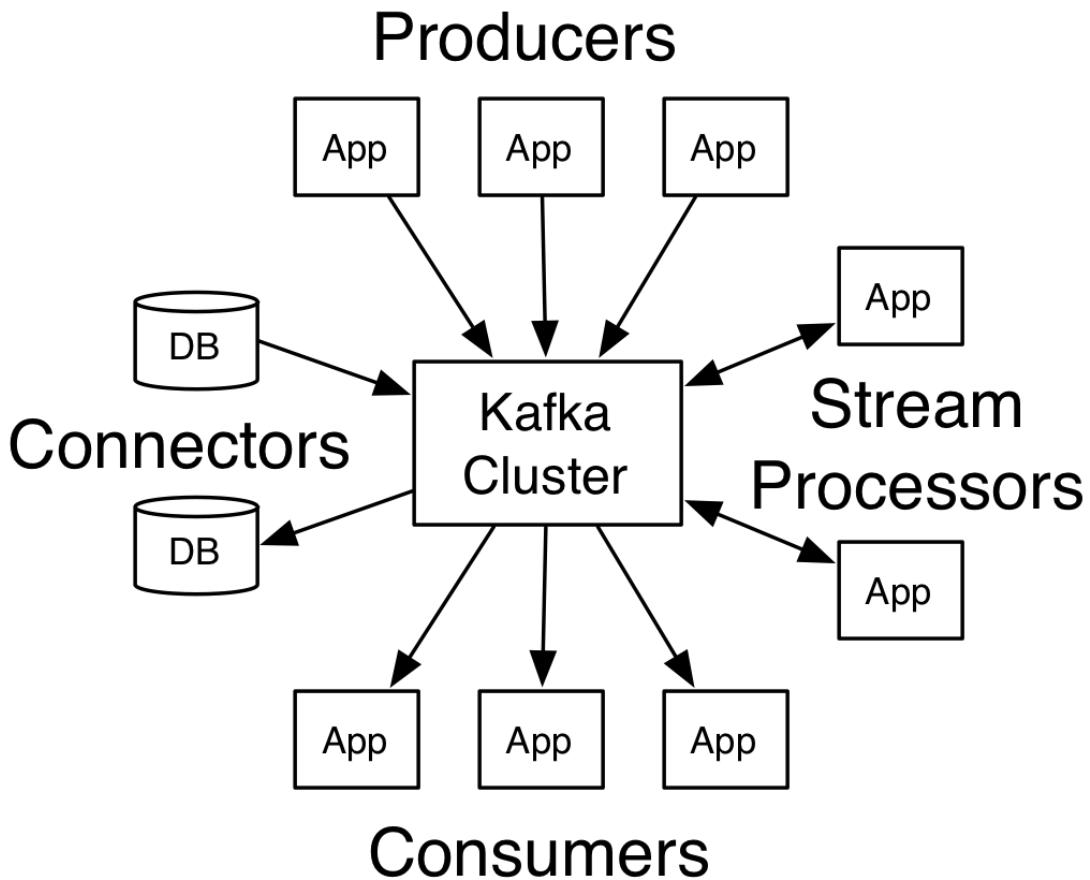


Figura 7.12_Kafka: APIs(Fuente: paradigmadigital.com)

4.1 Producer API

Producer API permite a las aplicaciones que pueda **publicar/enviar** mensajes a **un topic de Kafka** de forma asíncrona. Los productores automáticamente saben a qué broker y a qué partición deben escribir.

En el caso de que un broker se caiga, el productor sabe cómo recuperarse y seguirá escribiendo en el resto. Los productores envían los mensajes con clave (string, número, etc) o sin ella.

Si la clave es nula se enviarán en round robin entre los brokers. Si no es nula, todos los mensajes con esa clave se enviarán siempre a la misma partición.

Además, para confirmar que los mensajes han sido correctamente escritos en Kafka se podrá configurar la recepción de un ACK, ya sea por la recepción del mensaje por parte broker líder o por todos los brokers réplica:

- ACKs=0 (El productor no espera el acuse de recibo; posible pérdida de datos)

- ACKs=1 (El productor espera el reconocimiento del líder: pérdida de datos limitada)
- ACKs=all (El productor espera al líder + reconocimiento de réplicas; sin pérdida de datos)

La carga se balancea entre los brokers

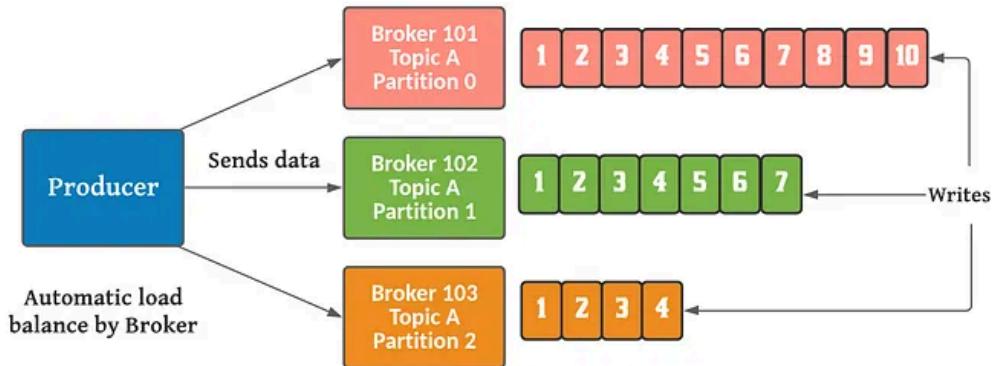


Figura 7.13_Kafka: Producers y Brokers (Fuente: medium.com/geekculture)

4.2 Consumer API

Consumer API permite a las aplicaciones que puedan **leer** los mensajes **desde un topic de Kafka** y tratarlos desde nuestra aplicación. Podemos crear un consumidor o un grupo de consumidores.

La diferencia entre ellos es que el grupo de consumidores permite el consumo de mensaje de forma paralela, es decir, si un nodo de ese grupo consume un mensaje el resto no lo hará.

Esto es útil a la hora de tener más de una instancia de un microservicio corriendo en nuestro sistema. Cada consumidor del grupo de consumidores leerá de una partición exclusiva.

Si hay más consumidores que particiones, algunos de los consumidores estarán inactivos, para solucionar esto es recomendable tener el mismo número de particiones que de consumidores dentro de un grupo.

En el caso de que un broker de los que está leyendo se caiga, los consumidores saben cómo recuperarse. Los datos son leídos en orden dentro de cada partición pero no entre ellas. Kafka almacena los offsets de los grupos de consumidores cuando estos leen los datos.

Consumer offset

Los offsets son almacenados en un topic de Kafka denominado "consumer_offsets". Cuando un consumidor de un grupo lee datos de Kafka, se actualiza el offset. Kafka almacena los offsets por el que va leyendo un grupo de consumidores, a modo de checkpoint. Si un consumidor se cae, cuando vuelva a ser levantado seguirá leyendo datos desde donde se quedó anteriormente.

Cuando un consumidor de un grupo ha procesado los datos que ha leído de Kafka, realizará un commit de sus offsets. Si el consumidor se cae, podrá volver a leer los mensajes desde el último offset sobre el que se realizó commit.

Garantías de Entrega

Como en otros sistemas distribuidos, Kafka tiene tres maneras de gestionar las garantías de entrega de los mensajes en sus protocolos:

- **At-least-once:** Garantiza que el mensaje siempre se entregará. Es posible que en caso de fallo se entregue varias veces, pero no se perderá ningún mensaje en el sistema.
- **At-most-once:** Garantiza que el mensaje se entregará una vez o no se entregará. Un mensaje nunca se entregará más de una vez.
- **Exactly-once:** Garantiza que todos los mensajes se van a entregar exactamente una vez, realizando el sistema las comprobaciones necesarias para que esto suceda.

4.3 Streams API

Streams API permiten transformar los streams de datos **desde** input topics **hacia** output topics. Podemos realizar modificaciones (ETL) sobre los mensajes (input topics) y escribir en otro topic (output topics) actuando como productor.

Combina la "simplicidad" del desarrollo de aplicaciones en lenguaje Java o Scala con los beneficios de la integración con el cluster de Kafka.

Los veremos con más profundidad más adelante.

4.4 Connect API

Se tratan de componentes listos para usar que **nos permiten simplificar la integración entre sistemas externos y el cluster de Kafka**. Podemos crear y ejecutar productores o consumidores reutilizables que conectan los topics de Kafka a las aplicaciones o sistemas externos, como por ejemplo una base de datos.

Además, **algunos permiten realizar modificaciones simples sobre los mensajes que irán a los topics de Kafka**. Se configuran mediante ficheros properties o a través de su API REST y entre sus características destacan ser distribuidos y escalables.

Los veremos con más profundidad más adelante.

4.5 Admin API

Admin API admite la gestión e inspección de topics, brokers, ACL y otros objetos de Kafka.

5. Quick Start

Usando algunas de las APIs

5.1 Obtener Kafka

[Descarga la última versión de Kafka](#)

```
1 wget https://dlcdn.apache.org/kafka/3.9.0/kafka_2.13-3.9.0.tgz
2 tar -xzf kafka_2.13-3.9.0.tgz
```

5.2 Configura y arranca el Entorno de Kafka

1. Debemos tener correctamente instalado y funcionando Java 8
2. Movemos el directorio a nuestro directorio `/opt` para una correcta organización

```
1 sudo mv kafka_2.13-3.9.0 /opt/kafka_2.13-3.9.0
```

3. Accedemos al directorio de Kafka

```
1 cd /opt/kafka_2.13-3.9.0
```

4. Apache Kafka puede ser iniciado usando Zookeeper o KRaft

Kafka with Zookeeper

1. Arranca los servicios Zookeeper

```
1 # Start the ZooKeeper service
2 bin/zookeeper-server-start.sh config/zookeeper.properties
```

2. Abre otra terminal y ejecuta

```
1 # Start the Kafka broker service
2 bin/kafka-server-start.sh config/server.properties
```

Kafka con KRaft

1. Kafka se puede ejecutar usando el modo KRaft mediante scripts locales y archivos descargados o mediante la imagen de docker

2. Usando los archivos descargados

- Genera un cluster UUID

```
1 | KAFKA_CLUSTER_ID="$(bin/kafka-storage.sh random-uuid)"
```

- Formatea el directorio de log

```
1 | bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c config/kraft/server.properties
```

- Ejecuta el servidor Kafka

```
1 | bin/kafka-server-start.sh config/kraft/server.properties
```

3. Usando la imagen de docker

- Obtenemos la imagen de docker

```
1 | docker pull apache/kafka:3.9.0
```

- Iniciamos el contenedor docker de kafka

```
1 | docker run -p 9092:9092 apache/kafka:3.9.0
```

Una vez que el servidor Kafka se haya iniciado exitosamente, tendremos un entorno Kafka básico ejecutándose y listo para usar.

5.3 Crea un Topic para Almacenar tus eventos

Antes de que podamos escribir nuestros primeros eventos, debemos crear un **topic**. Abre otra sesión de terminal y ejecute:

```
1 | bin/kafka-topics.sh --create --topic quickstart-events --bootstrap-server localhost:9092
```

✓ Argumentos en comandos de Kafka

Todos los scripts de línea de comandos de Kafka tienen opciones adicionales: para saber cuales ejecútalos sin ningún argumento para mostrar información de uso, por ejemplo: `kafka-topics.sh`. También podemos mostrar los detalles y partición de un topic:

```
1 | bin/kafka-topics.sh --describe --topic quickstart-events --bootstrap-server localhost:9092
2 | Topic: quickstart-events          TopicId: NPmZHyhbR9y00wMg1MH2sg
3 | PartitionCount: 1      ReplicationFactor: 1      Configs:
4 | Topic: quickstart-events Partition: 0      Leader: 0      Replicas: 0  Isr: 0
```

5.4 Escribe algún evento en el topic

Un cliente de Kafka se comunica con los **brokers** de Kafka a través de la red para escribir (o leer) eventos. Una vez recibidos, los brokers almacenarán los eventos de forma duradera y tolerante a fallos durante el tiempo que sea necesario, incluso para siempre.

Ejecutamos el *cliente productor (Producer API)* para escribir algunos eventos en el topic. De forma predeterminada, cada línea que ingresemos dará como resultado que se escriba un evento separado en el topic.

```

1 bin/kafka-console-producer.sh --topic quickstart-events --bootstrap-server
localhost:9092
2 >Este es mi primer evento
3 >Este es mi segundo evento
4 >Módulo Big Data Aplicado
5 >IES Gran Capitán

```

Podemos parar el cliente en cualquier momento con **Ctrl-C**

5.5 Lee los eventos

Abre otra sesión de terminal y ejecute el *cliente consumidor (Consumer API)* de consola para leer los eventos que acabamos de crear:

```

1 bin/kafka-console-consumer.sh --topic quickstart-events --from-beginning -
--bootstrap-server localhost:9092
2 Este es mi primer evento
3 Este es mi segundo evento
4 Módulo Big Data Aplicado
5 IES Gran Capitán

```

The screenshot shows three terminal windows on a Linux system (hadoop@master). The top window shows the producer command being run, with five lines of text being typed: "Este es mi primer evento", "Este es mi segundo evento", "Módulo Big Data Aplicado", and "IES Gran Capitán". The middle window shows the consumer command running, displaying the same five lines of text as they appear in the producer's output. The bottom window shows the consumer command running again, but this time it only displays the first two lines: "Este es mi primer evento" and "Este es mi segundo evento". This indicates that the consumer has caught up to the end of the topic.

Figura 7.14_Kafka: Quick Start

6. Ejemplo1. Python en Kafka

En principio, Kafka sólo tiene soporte para crear aplicaciones en lenguaje Java y Scala. Pero gracias a la librería [Kafka-python](#), podemos crear aplicaciones en Python para ser usadas con Kafka. Para ello vamos a crear el productor a través de [KafkaProducer](#) y el consumidor a través de [KafkaConsumer](#)

Elaboraremos una pequeña aplicación para implementar un sistema de registro de eventos de sensores. En este escenario, el productor enviará eventos (simulados) que contienen lecturas de sensores en formato JSON, y el consumidor procesará estos eventos para imprimir la información.

1. Instalamos la librería

```
1 pip install kafka-python
```

2. Suponemos que tendremos la siguiente configuración kafka, que crearemos después

3. Servidor en `localhost:9092`

4. Topic llamado `topic-python-sensor`

5. Grupo de consumidores `group_consumer_sensor`

6. Creamos el productor con [KafkaProducer](#)

producer.py

```
1 from kafka import KafkaProducer
2 import json
3 from datetime import datetime
4 import random
5 import time
6
7 # Configuración del productor con serialización JSON
8 producer = KafkaProducer(
9     bootstrap_servers=['localhost:9092'],
10    value_serializer=lambda x: json.dumps(x).encode('utf-8')
11 )
12
13 # Simulación de envío de eventos de sensores
14 while True:
15     sensor_event = {
16         'sensor_id': random.randint(1, 100),
17         'sensor_type': random.choice(['temperature', 'humidity',
18             'pressure']),
19         'value': round(random.uniform(20.0, 30.0), 2),
20         'timestamp': datetime.now().isoformat()
```

```

20     }
21     # Enviamos un mensaje
22     producer.send('topic-python-sensor', value=sensor_event)
23     print(f"Sent: {sensor_event}")
24     time.sleep(1)

```

4. Creamos el productor con KafkaConsumer

consumer.py

```

1  from kafka import KafkaConsumer
2  import json
3
4  # Configuración del consumidor con deserialización JSON
5  consumer = KafkaConsumer(
6      'topic-python-sensor',
7      bootstrap_servers=['localhost:9092'],
8      auto_offset_reset='earliest', # Para leer mensajes desde el inicio
del topic
9      enable_auto_commit=True, # Para confirmaciones automáticas
10     group_id='group_consumer_sensor', # Identificador del grupo de
consumidores
11     value_deserializer=lambda x: json.loads(x.decode('utf-8')) # Para
deserializar los mensajes desde JSON
12 )
13
14 # Procesamiento de eventos de sensores
15 for message in consumer:
16     event = message.value
17     print(f"Received event from sensor {event['sensor_id']}:"
{event['sensor_type']} = {event['value']} at {event['timestamp']}")

```

5. Iniciamos Kafka con KRaft

```

1  #Genera un cluster UUID
2  KAFKA_CLUSTER_ID="$(bin/kafka-storage.sh random-uuid)"
3
4  #Si no te permite formatear (con el comando siguiente) es debido a que se
mantienen los logs del ejemplo anterior. debes borrarlos
5  #sudo rm -r /tmp/kraft-combined-logs/
6
7  #Formatea el directorio de log
8  bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c
config/kraft/server.properties
9
10 #Ejecuta el servidor Kafka
11 bin/kafka-server-start.sh config/kraft/server.properties

```

6. Creamos el topic sensores

```

1  bin/kafka-topics.sh --create --partitions 1 --replication-factor 1 --topic
topic-python-sensor --bootstrap-server localhost:9092

```

7. Lanzamos el productor

```
1 | python3 producer.py
```

8. Lanzamos el consumidor

```
1 | python3 consumer.py
```

```

[hadoop@master:/opt/kafka/ejemplo1] hadoop@master:/opt/kafka/ejemplo1
[2025-03-12 13:03:20,125] INFO [LogLoader partition=topic-python-sensor-0, dir=/tmp/kraft-combined-logs/topic-python-sensor-0] Loading producer state till offset 0 with message format version 2 (kafka.log.UnifiedLogs)
[2025-03-12 13:03:20,126] INFO Created log for partition topic-python-sensor-0 in /tmp/kraft-combined-logs/topic-python-sensor-0 with properties {} (kafka.log.LogManager)
[2025-03-12 13:03:20,707] INFO [Partition topic-python-sensor-0 brokers=[1]] No checkpointer highwatermark is found for partition topic-python-sensor-0 (kafka.cluster.Partition)
[2025-03-12 13:03:20,712] INFO [Partition topic-python-sensor-0 brokers=[1]] Log loaded for partition topic-python-sensor-0 with initial high watermark 0 (kafka.cluster.Partition)
[2025-03-12 13:03:27,317] INFO [GroupCoordinator 1]: Preparing to rebalance group console-consumer-30894 in state PreparingRebalance with old generation 1 (_consumer_offsets-25) (reason: Removing member console-consumer-2d45c970-acae-4009-80cd-841a02c204b4 on LeaveGroup; client reason: the consumer is being closed) (kafka.coordinator.group.GroupCoordinator)
[2025-03-12 13:03:27,319] INFO [GroupCoordinator 1]: Group console-consumer-30894 with generation 2 is now empty (_consumer_offsets-25) (kafka.coordinator.group.GroupCoordinator)
[2025-03-12 13:03:27,327] INFO [GroupCoordinator 1]: Member MemberMetadata(memberId=console-consumer-2d45c970-acae-4009-80cd-841a02c204b4, groupIdInstance=None, clientId=console-consumer, clLengths=[127.0,0.1], sessionTimeoutMs=45000, rebalanceTimeoutMs=300000, supportedProtocols=list(range, cooperative-sticky)) has left group console-consumer-30894 through explicit "LeaveGroup"; client reason: the consumer is being closed (kafka.coordinator.group.GroupCoordinator)
[2025-03-12 13:03:27,329] INFO [GroupCoordinator 1]: [NodeControllerClientManager 1] Name registration for broker 1 succeeded. (org.apache.kafka.clients.NetworkClient)
[2025-03-12 13:06:33,257] INFO [GroupCoordinator 1]: Dynamic Member with id 1 joins group group_consumer_sensor in Empty state. Created a new member id kafka-python-2.0.6-1baed3ab-c09e-4a3a-afcf-e37a79262cc9 for this member and add to the group. (kafka.coordinator.group.GroupCoordinator)
[2025-03-12 13:06:33,860] INFO [GroupCoordinator 1]: Preparing to rebalance group group_consumer_sensor in state PreparingRebalance with old generation 0 (_consumer_offsets-5) (reason: Adding new member kafka-python-2.0.6-1baed3ab-c09e-4a3a-afcf-e37a79262cc9 with group instance id None; client reason: not provided) (kafka.coordinator.group.GroupCoordinator)
[2025-03-12 13:06:36,861] INFO [GroupCoordinator 1]: Stabilized group group_consumer_sensor generation 1 (_consumer_offsets-5) with 1 members (kafka.coordinator.group.GroupCoordinator)
[2025-03-12 13:06:36,865] INFO [GroupCoordinator 1]: Assignment received from leader kafka-python-2.0.6-1baed3ab-c09e-4a3a-afcf-e37a79262cc9 for group group_consumer_sensor for generation 1. The group has 1 members, 0 of which are static. (kafka.coordinator.group.GroupCoordinator)
[]

[hadoop@master:/opt/kafka/ejemplo1] hadoop@master:/opt/kafka/ejemplo1
Sent: {'sensor_id': 100, 'sensor_type': 'temperature', 'value': 27.16, 'timestamp': '2025-03-12T13:06:33.614944'}
Sent: {'sensor_id': 65, 'sensor_type': 'humidity', 'value': 22.63, 'timestamp': '2025-03-12T13:06:34.617031'}
Sent: {'sensor_id': 71, 'sensor_type': 'pressure', 'value': 20.87, 'timestamp': '2025-03-12T13:06:35.623685'}
Sent: {'sensor_id': 5, 'sensor_type': 'pressure', 'value': 24.13, 'timestamp': '2025-03-12T13:06:36.629426'}
Sent: {'sensor_id': 71, 'sensor_type': 'pressure', 'value': 28.44, 'timestamp': '2025-03-12T13:06:37.632492'}
Sent: {'sensor_id': 31, 'sensor_type': 'temperature', 'value': 22.23, 'timestamp': '2025-03-12T13:06:38.638099'}
Sent: {'sensor_id': 27, 'sensor_type': 'humidity', 'value': 26.98, 'timestamp': '2025-03-12T13:06:39.643468'}
Sent: {'sensor_id': 85, 'sensor_type': 'pressure', 'value': 24.99, 'timestamp': '2025-03-12T13:06:40.646400'}
Sent: {'sensor_id': 8, 'sensor_type': 'pressure', 'value': 23.99, 'timestamp': '2025-03-12T13:06:41.657184'}
Sent: {'sensor_id': 4, 'sensor_type': 'pressure', 'value': 29.81, 'timestamp': '2025-03-12T13:06:42.659764'}
```

Figura 7.15_Kafka: Ejemplo sensor en python

7. Kafka on KRaft(Kafka Raft Metadata mode)

Como hemos visto anteriormente, recordemos que desde la versión 3.5 se elimina la dependencia de Kafka con Zookeeper y se marca el soporte de Zookeeper como deprecated, pasándose a usar de manera estable el sistema interno KRaft para la gestión del cluster de Kafka en este sistema distribuido.

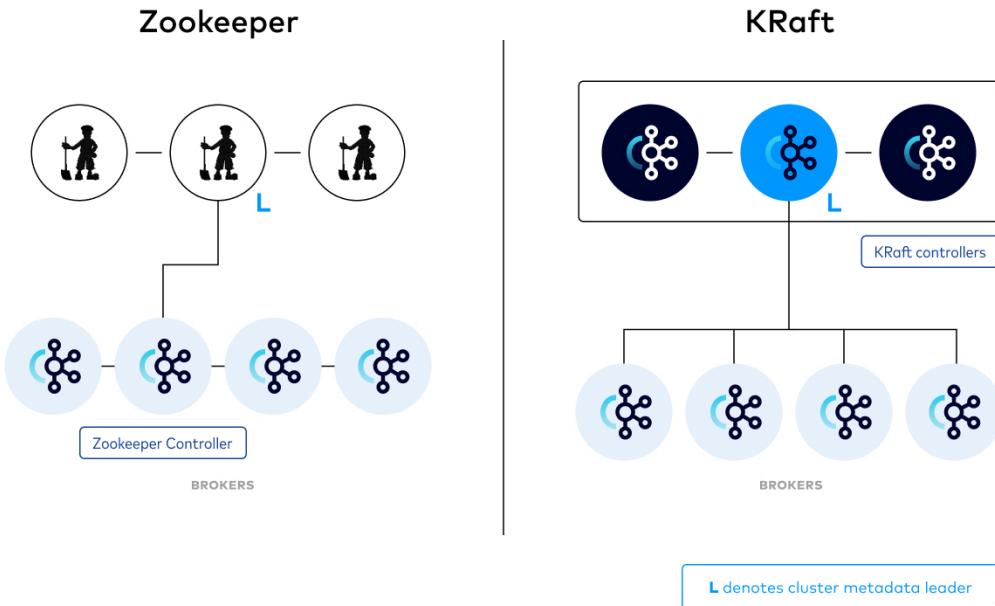


Figura 7.11_Kafka: Diagrama Kraft (Fuente: kafka.apache.org)

El modo KRaft utiliza un nuevo servicio de controlador de quorum en Kafka que reemplaza al controlador anterior y utiliza una variante basada en eventos del protocolo de consenso Raft.

Esto nos trae ciertas consideraciones a tener en cuenta (además de [otras ventajas](#)):

- Kafka no necesita un sistema de terceros para la gestión del sistema distribuido.
- **Todos los nodos del cluster son nodos Kafka, que tomarán roles diferentes dependiendo de su función en el cluster en Kraft mode:** controller , broker (como puedes ver en la figura) y server .

Por tanto, a la hora de configurar un cluster Kafka, estos roles de cada nodo hay que configurarlos para un correcto diseño de nuestro sistema.

7.1 Roles

Cuando opera Apache Kafka® en modo KRaft, debemos configurar la propiedad `process.roles` . Esta propiedad especifica si el servidor(nodo que forma parte del cluster Kafka) actúa como controller , broker (como puedes ver en la figura) o combinado, aunque actualmente el modo combinado no es compatible con cargas de trabajo de producción.

A) Controller

- **Role:** El controller es un componente crítico en Kafka que gestiona el estado del clúster. Esto incluye, pero no se limita a, la asignación de particiones a los brokers y la gestión de

los cambios en la membresía del clúster. En el modo KRaft, el controlador también gestiona el quorum y los metadatos del clúster sin depender de Zookeeper.

- **Configuración:** Cuando levantas un nodo en modo controller en KRaft, específicas `process.roles=controller` en la configuración. Esto indica que el nodo debe actuar exclusivamente como un controlador.
- **Comunicación:** El controlador se comunica con los brokers para coordinar la gestión del clúster. Se configura un conjunto de votantes del quorum (`controller.quorum.voters`) para participar en el consenso Raft.

En el modo KRaft, se seleccionan servidores Kafka específicos para que sean controladores, almacenando metadatos para el clúster en el registro de metadatos, y se seleccionan otros servidores para que sean intermediarios. Los servidores seleccionados para ser controladores participarán en el quorum de metadatos. Cada controlador es activo o de reserva activa para el controlador activo actual.

En un entorno de producción, el quorum del controlador se implementará en varios nodos. A esto se le llama **ensemble**. **Ensemble** es un conjunto de $2n + 1$ controladores donde n es cualquier número mayor que 0. El número impar de controladores permite que el quorum de controladores realice elecciones mayoritarias para el liderazgo. En cualquier momento dado, puede haber hasta n servidores fallidos en un conjunto y el clúster mantendrá el quorum. Por ejemplo, con 3 controladores, el clúster puede tolerar un fallo de controlador. Si en algún momento se pierde el quorum, el clúster dejará de funcionar. **Actualmente, no se recomiendan más de 3 controladores en entornos críticos.**

B) Broker

- **Role:** Los brokers son los nodos que realmente almacenan y sirven datos (es decir, los mensajes de Kafka) a los productores y consumidores. En un clúster de Kafka, puedes tener muchos brokers para escalar horizontalmente el rendimiento y la capacidad de almacenamiento.
- **Configuración:** Levantar un nodo en modo broker en KRaft implica configurar `process.roles=broker` en las propiedades del nodo. Esto lo designa para servir como un broker, manejando las solicitudes de los clientes y el almacenamiento de los datos.
- **Comunicación:** Los brokers se registran con el controlador y siguen sus instrucciones para la asignación de particiones y otras tareas de gestión del clúster. Los brokers no participan directamente en el consenso Raft; solo el controlador gestiona los metadatos del clúster.

C) Combinado

- **Role:** En algunos contextos, especialmente en entornos de desarrollo o pruebas, puedes querer que un nodo actúe tanto como controlador como broker. Esto simplifica la configuración y reduce la cantidad de recursos necesarios, ya que no es necesario ejecutar nodos separados para los roles de controlador y broker.

- **Configuración:** Para levantar un nodo en modo combinado (`server`), configuras `process.roles=broker,controller` en las propiedades del nodo. Esto indica que el nodo debe asumir ambos roles.
- **Comunicación:** Como este nodo combina ambos roles, gestionará tanto los metadatos del clúster como el almacenamiento y la transmisión de los datos de los mensajes.

Resumen

- **Controller:** Gestiona el estado del clúster y los metadatos, no almacena ni sirve datos de los mensajes.
- **Broker:** Almacena y sirve datos de los mensajes, pero no gestiona los metadatos del clúster directamente.
- **Combinado:** Realiza ambas funciones, tanto la gestión de metadatos del clúster como el almacenamiento y servicio de los datos de los mensajes, útil para configuraciones simplificadas.

Al configurar un clúster Kafka en modo KRaft, es importante planificar el número y tipo de nodos según los requisitos de escala, rendimiento y alta disponibilidad de tu aplicación o sistema.

A continuación, vamos a ver algunos de las configuraciones más importantes y de uso más frecuente para una correcta configuración de nuestro cluster Kafka en Kraft mode teniendo en cuenta los posibles roles de cada uno de los servers que componen el cluster.

7.2 Server Basic

7.2.1 `process.roles`

Como ya hemos visto, puede tener los siguientes valores: `controller`, `broker`, `controller,broker`.

- Type: string
- Default:
- Importance: required for KRaft mode

7.2.2 `node.id`

El identificador único de este servidor. **Cada ID de nodo debe ser único en todos los servidores de un clúster en particular.** No hay dos servidores que puedan tener el mismo ID de nodo independientemente de su valor de `process.roles`. Este identificador reemplaza a `broker.id`, que se utiliza cuando se opera en modo ZooKeeper.

- Type: int
- Default:

- Importance: required for KRaft mode

7.2.3 controller.quorum.voters

Todos los servidores (controllers y brokers) en un clúster de Kafka deben establecer la propiedad controller.quorum.voters, debido a que todos deben descubrir los votantes del quórum. Por tanto, debemos identificar a todos los controllers incluyéndolos en la lista que proporciona esta propiedad controller.quorum.voters, separados por comas.

Además, **todos los controladores deben estar enumerados**. Cada controlador se identifica con su `id`, información de `host` y `puerto` con el siguiente formato: `{id}@{host}:{port}`. Varias entradas estarán separadas por comas y pueden tener un aspecto similar al siguiente:

```
1 controller.quorum.voters=1@host1:port1,2@host2:port2,3@host3:port3
```

El ID de nodo proporcionado en la propiedad controller.quorum.voters debe coincidir con el ID correspondiente en los servidores del controlador. Por ejemplo, en el controller1, `node.id` debe establecerse en `1`. **Si un servidor es solo un broker, su ID de nodo no debe aparecer en la lista controller.quorum.voters. Tampoco puede haber dos servidores que puedan tener el mismo ID de nodo independientemente de su valor en process.roles.**

Por ejemplo, si un clúster Kafka tiene 3 controladores denominados controller1, controller2 y controller3, entonces el controller1 tendría la siguiente configuración:

```
1 process.roles=controller
2 node.id=1
3 listeners=CONTROLLER://controller1.example.com:9093
4 controller.quorum.voters=1@controller1.example.com:9093,2@controller2.example
9093
```

- Type: string
- Default:
- Importance: required for KRaft mode



Dynamic KRaft Quorums

Los controladores en KRaft son nodos Kafka que utilizan el algoritmo de consenso Raft para elegir un líder (controlador activo) y replicar los metadatos del clúster. Antes de **Kafka v3.9.0**, los clusters basados en KRaft sólo permitían configuraciones de quórum estáticas, en las que el conjunto de nodos controladores (también conocidos como votantes) es fijo y no puede cambiarse sin reiniciar.

Kafka v3.9.0 introduce soporte para quórum dinámico con KIP-853. Esta es una de las características que faltaban para alcanzar la paridad de características con los clústeres basados en ZooKeeper. A partir de ahora, el quórum dinámico debería ser la forma preferida de configurar un clúster basado en KRaft, ya que no requiere tiempo de inactividad del clúster.

El escalado dinámico del clúster es importante para una serie de casos de uso:

- Escalado: El operador quiere escalar el número de controladores añadiendo o quitando un controlador (esto es bastante raro en la práctica, ya que el quórum del controlador se establece una vez y rara vez se cambia).
- Sustituir: El operador desea sustituir un controlador debido a un fallo de disco o hardware.
- Migrar: El operador desea migrar el clúster de máquinas antiguas a nuevas, o cambiar la arquitectura de KRaft (por ejemplo, pasar de controladores dedicados a nodos de modo combinado).

Static versus Dynamic KRaft Quorums

Por tanto, ahora **hay dos formas de ejecutar KRaft**: la antigua, utilizando quórum de controladores estáticos, y la nueva, utilizando quórum de controladores dinámicos KIP-853.

Cuando se utiliza un **quórum estático**, el archivo de configuración para cada broker y controlador debe especificar los IDs, nombres de host y puertos de todos los controladores en `controller.quorum.voters`.

En cambio, cuando se utiliza un **quórum dinámico**, se debe configurar `controller.quorum.bootstrap.servers`. No es necesario que esta clave de configuración contenga todos los controladores, pero debe contener tantos como sea posible para que todos los servidores puedan localizar el quórum. En otras palabras, su función es muy parecida a la configuración `bootstrap.servers` utilizada por los clientes Kafka.

Debido a que ya se está trabajando en la versión 4.0 de Kafka, y ya ha salido alguna release candidate, seguiremos usando Static quorum a la espera de descubrir como finalmente se establece la forma de hacer quorum cuando salga la versión estable de 4.0

7.3 Socket Server Settings

7.3.1 `listeners`

Los servidores Kafka admiten la escucha de conexiones en varios puertos. Esto se configura a través de la propiedad `listeners` en la configuración del servidor, que acepta una lista separada por comas de los *listeners* habilitados. Debe definirse al menos una escucha en cada servidor. El formato de cada listener definido en `listeners` es el siguiente:

{LISTENER_NAME}://{{hostname}}:{port} . `LISTENER_NAME` suele ser un nombre descriptivo que define el propósito de la escucha.

Si no se configura, el nombre del host será igual al valor de

`java.net.InetAddress.getCanonicalHostName()` con el nombre de *listener* `PLAINTEXT`, y el puerto `9092`.

Para más información sobre la seguridad de la comunicación entre las conexiones de escucha, [consulta la documentación oficial](#)

- Type: string with the format `listener_name://host_name:port`
- Default: If not configured, the host name will be equal to the value of `java.net.InetAddress.getCanonicalHostName()`, with `PLAINTEXT` listener name, and port `9092`. Example: `listeners=PLAINTEXT://your.host.name:9092`
- Importance: high

7.3.2 `controller.listener.names`

Una lista separada por comas de entradas `listener_name` para los `listeners` utilizados por el `controller`. En un nodo con `process.roles=broker`, sólo el primer *listener* de la lista será utilizado por el broker. Los brokers basados en ZooKeeper no deben establecer este valor. Para controladores KRaft en modo aislado o combinado, el nodo escuchará como controlador KRaft en todos los *listeners* que estén listados para esta propiedad, y cada uno debe aparecer en la propiedad `listeners`.

- Type: string
- Default: null
- Importance: required for KRaft mode.

7.4 Log Basics

7.4.1 `log.dir`

DIRECTORIO en el que se guardan los datos de registro (complementario por la propiedad `log.dirs`)

- Type: string
- Default: `/tmp/kafka-logs`
- Importance: high

7.4.2 log.dirs

Lista separada por comas de los directorios donde se almacenan los datos de registro. Si no se define, se utiliza el valor de log.dir

- Type: string
- Default: null
- Importance: high

7.4.3 num.partitions

El número por defecto de particiones de registro por topic. Más particiones permiten un mayor paralelismo para el consumo, pero esto también resultará en más archivos a través de los brokers. **Establece esta configuración sólo para los brokers. Esto será ignorado por los controladores KRaft.**

- Type: int
- Default: 1
- Importance: medium

7.4.4 metadata.log.dir

Se utiliza para especificar dónde se aloja el registro de metadatos de los clusters en modo KRaft. Si no se establece, el registro de metadatos se coloca en el primer directorio de registro especificado en la propiedad `log.dirs`.

- Type: string
- Default: null
- Importance: high

7.5 Otras configuraciones

Para consultar el resto de propiedades puedes consultar la [documentación oficial](#)

7.6 Tools for debugging KRaft mode

Vamos a ver algunos de ellos, los de uso más frecuente.

7.6.1 Describe runtime status

Puede describir el estado en tiempo de ejecución de la partición de metadatos del clúster utilizando `kafka-metadata-quorum`

```
1 bin/kafka-metadata-quorum --bootstrap-server host1:9092 describe --status
```

```

1 Output might look like the following:
2
3     ClusterId:          fMCL8kv1SWm87L_Md-I2hg
4     LeaderId:           3002
5     LeaderEpoch:        2
6     HighWatermark:     10
7     MaxFollowerLag:    0
8     MaxFollowerLagTimeMs: -1
9     CurrentVoters:     [3000,3001,3002]
10    CurrentObservers:  [0,1,2]

```

7.6.2 Debug log segments

Podemos utilizar `kafka-dump-log` para depurar los segmentos de registro y las instantáneas del directorio de metadatos del clúster. La herramienta escaneará los archivos proporcionados y decodificará los registros de metadatos. Por ejemplo, el siguiente comando decodifica e imprime los registros del primer segmento de registro:

```

1 bin/kafka-dump-log --cluster-metadata-decoder --files tmp/kraft-combined-
logs/_cluster_metadata-0/00000000000000023946.log

```

7.6.3 Inspect the metadata partition

Podemos utilizar `kafka-metadata-shell` para inspeccionar de forma interactiva el clúster de metadatos. El siguiente ejemplo muestra cómo abrir el shell.

```

1 kafka-metadata-shell --directory tmp/kraft-combined-logs/_cluster-
metadata-0/

```

```

1 Loading...
2 [ Kafka Metadata Shell ]
3 >> ls
4 brokers configs features linkIds links shell topicIds topics
5 >> ls /topics
6 test
7 >> cat /topics/test/0/data
8 {
9     "partitionId" : 0,
10    "topicId" : "5zoAlv-xEh9xRANKXt1Lbg",
11    "replicas" : [ 1 ],
12    "isr" : [ 1 ],
13    "removingReplicas" : null,
14    "addingReplicas" : null,
15    "leader" : 1,
16    "leaderEpoch" : 0,
17    "partitionEpoch" : 0
18 }
19 >> exit

```

8. Ejemplo 2. Cluster Kafka

Imagina que estamos construyendo un sistema para un banco que necesita procesar eventos de transacciones financieras (por ejemplo, transferencias de dinero entre cuentas) en tiempo real. El sistema debe ser capaz de manejar un alto volumen de transacciones de forma eficiente y confiable. Utilizaremos Kafka para construir la infraestructura de mensajería que soportará este procesamiento.

✓ Configuración cluster Kafka

Vamos a **configurar el cluster de Kafka usando mode KRaft**. Siguiendo las **consideraciones del punto anterior** y 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*

8.1 Requisitos

Vamos a configurar un cluster con las siguientes características:

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

8.2 Configuración del Clúster de Kafka

- **Consideraciones previas:**

1. Vamos a establecer todos los archivos de configuración en una carpeta de ejemplo llamada `ejemplo2`, que en mi caso estará alojada en `/opt/kafka/ejemplo2`
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 `ejemplo2`

```
1 | mkdir -p /opt/kafka/ejemplo2/config
```

```
2 | mkdir -p /opt/kafka/ejemplo2/logs
```

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

```
1 | cp config/kraft/controller.properties  
/opt/kafka/ejemplo2/config/controller1.properties  
2 | cp config/kraft/broker.properties  
/opt/kafka/ejemplo2/config/broker1.properties  
3 | cp config/kraft/broker.properties  
/opt/kafka/ejemplo2/config/broker2.properties
```

5. Asignamos la configuración al controller

controller1.properties

```
1 | # Server Basics  
2 | process.roles=controller  
3 | node.id=1  
4 | controller.quorum.voters=1@localhost:9093  
5 | # Socket Server Settings  
6 | listeners=CONTROLLER://localhost:9093  
7 | controller.listener.names=CONTROLLER  
8 | # Log Basics  
9 | log.dirs=/opt/kafka/ejemplo2/logs/controller1
```

6. Asignamos la siguiente configuración para cada broker

broker1

broker1.properties

```
1 | # Server Basics  
2 | process.roles=broker  
3 | node.id=2  
4 | controller.quorum.voters=1@localhost:9093  
5 | # Socket Server Settings  
6 | listeners=PLAINTEXT://localhost:9094  
7 | advertised.listeners=PLAINTEXT://localhost:9094  
8 | # Log Basics  
9 | log.dirs=/opt/kafka/ejemplo2/logs/broker1
```

broker2

broker2.properties

```
1 | # Server Basics  
2 | process.roles=broker  
3 | node.id=3  
4 | controller.quorum.voters=1@localhost:9093  
5 | # Socket Server Settings  
6 | listeners=PLAINTEXT://localhost:9095  
7 | advertised.listeners=PLAINTEXT://localhost:9095
```

```

8 # Log Basics
9 log.dirs=/opt/kafka/ejemplo2/logs/broker2

```

7. Iniciamos Kafka con KRaft

```

1 #Genera un cluster UUID
2 KAFKA_CLUSTER_ID=$(bin/kafka-storage.sh random-uuid)"
3 echo $KAFKA_CLUSTER_ID
4
5 #Formateamos los directorios de log
6 bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c
/opt/kafka/ejemplo2/config/controller1.properties
7 bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c
/opt/kafka/ejemplo2/config/broker1.properties
8 bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c
/opt/kafka/ejemplo2/config/broker2.properties

```

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

```

1 #Ejecuta el servidor Kafka
2 bin/kafka-server-start.sh
/opt/kafka/ejemplo2/config/controller1.properties
3 bin/kafka-server-start.sh /opt/kafka/ejemplo2/config/broker1.properties
4 bin/kafka-server-start.sh /opt/kafka/ejemplo2/config/broker2.properties

```

8.3 Creación del Topic

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

```

1 bin/kafka-topics.sh --create --topic financial-transactions --bootstrap-
server localhost:9094 --replication-factor 2 --partitions 2

```

2. Podemos ver la descripción del topic creado

```

1 bin/kafka-topics.sh --describe --topic financial-transactions --bootstrap-
server localhost:9094

```

```

1 Topic: financial-transactions TopicId: 2WLtmqr4TAiH3FwM7LB5Ng
PartitionCount: 2 ReplicationFactor: 2 Configs: segment.bytes=1073741824
2 Topic: financial-transactions Partition: 0 Leader: 3 Replicas:
3,2 Isr: 3,2
3 Topic: financial-transactions Partition: 1 Leader: 2 Replicas:
2,3 Isr: 2,3

```

8.4 Productor y Consumidor

1. Creamos el productor con [KafkaProducer](#)

producer.py

```

1  from kafka import KafkaProducer
2  import json
3  import time
4  import random
5
6  producer = KafkaProducer(bootstrap_servers=['localhost:9094',
'localhost:9095'],
7                           value_serializer=lambda x:
json.dumps(x).encode('utf-8'))
8
9  while True:
10     transaction = {
11         "source_account": random.randint(1000, 1999),
12         "destination_account": random.randint(2000, 2999),
13         "amount": round(random.uniform(10.00, 1000.00), 2),
14         "currency": "EUR",
15         "timestamp": time.time()
16     }
17     producer.send('financial-transactions', value=transaction)
18     print(f"Sent transaction: {transaction}")
19     time.sleep(random.randint(1, 5))

```

2. Creamos el consumidor con [KafkaConsumer](#)**consumer.py**

```

1  from kafka import KafkaConsumer
2  import json
3
4  consumer = KafkaConsumer('financial-transactions',
5                           bootstrap_servers=['localhost:9094',
'localhost:9095'],
6                           auto_offset_reset='earliest',
7                           value_deserializer=lambda x:
json.loads(x.decode('utf-8')))
8
9  for message in consumer:
10     transaction = message.value
11     print(f"Received transaction: {transaction}")

```

8.5 Ejecución de la aplicación

1. Lanzamos el productor

```
1  python3 producer.py
```

2. Lanzamos el consumidor

```
1  python3 consumer.py
```

```
# metadata Features(metadataVersion=3,9,1v0, finalizedFeatures=metadata.version>20, finalizedFeaturesEpoch=4). (org.apache.kafka.clients.producer.internals.ConsumerRunnable)
[2025-03-12 14:15:02,001] INFO [ControllerRegistrationManager id=1 incarnation=0MTNsV2tfeInn8vvrY_xw] sendControllerRegistrationRequestData(controllerId=1, incarnationId=0MTNsV2tfeInn8vvrY_xw, zkMigrationReady=false, listeners=[Listener{name='CONTROLLER', host='localhost', port=9093, securityProtocol=0}], features=[Feature{name='kraft.version', minSupportedVersion=0, maxSupportedVersion=1}, Feature{name='metadata.version', minSupportedVersion=1, maxSupportedVersion=21}]) (kafka.server.ControllerRegistrationManager)
[2025-03-12 14:15:02,222] INFO [ControllerRegistrationManager id=1 incarnation=0MTNsV2tfeInn8vvrY_xw] RegistrationResponseHandler: controller acknowledged ControllerRegistrationRequest. (kafka.server.ControllerRegistrationManager)
[2025-03-12 14:15:02,223] INFO [ControllerRegistrationManager id=1 incarnation=0MTNsV2tfeInn8vvrY_xw] Our registration has been persisted to the metadata log. (kafka.server.ControllerRegistrationManager)
[]

hadoop@master:/opt/kafka_2.13-3.9.0$ hadoop@master:/opt/kafka_2.13-3.9.0 94x21
a.log.UnifiedLogs)
[2025-03-12 14:15:31,841] INFO [Created log for partition financial-transactions-1 in /opt/kafka_2.13-3.9.0/logs/broker1/financial-transactions-1 with properties {}] (kafka.log.LogManager)
[2025-03-12 14:15:31,841] INFO [Partition financial-transactions-1 broker=2] No checkpointer high watermark is found for partition financial-transactions-1 (kafka.cluster.Partition)
[2025-03-12 14:15:31,941] INFO [Partition financial-transactions-1 broker=2] Log loaded for partition financial-transactions-1 with initial high watermark 0 (kafka.cluster.Partition)
[2025-03-12 14:15:31,942] INFO [ReplicaFetcherManager on broker 2] Removed fetcher for partitions Set(financial-transactions-1) (kafka.server.ReplicaFetcherManager)
[2025-03-12 14:15:31,970] INFO [ReplicaFetcherManager on broker 2] Added fetcher to broker 3 for partitions HashMap(financial-transactions-1 -> InitialFetchState(Some(Ln1Mx2B592Oujj8tLEz0g),BrokerEndPoint(id=3, host=localhost:9095),0,0)) (kafka.server.ReplicaFetcherManager)
[2025-03-12 14:15:31,872] INFO [ReplicaFetcherThread-0-3]: Starting (kafka.server.ReplicaFetcherThread)
[2025-03-12 14:15:31,877] INFO [ReplicaFetcher replicaId=2, leaderId=3, fetcherId=0] Truncating partition financial-transactions-1 with TruncationState(offset=0, completed=true) due to local high watermark 0 (kafka.server.ReplicaFetcherThread)
[2025-03-12 14:15:31,879] INFO [UnifiedLog partition=financial-transactions-1, dir=/opt/kafka_2.13-3.9.0/logs/broker1] Truncating to 0 has no effect as the largest offset in the log ts -1 (kafka.log.UnifiedLog)
[]

hadoop@master:/opt/kafka_2.13-3.9.0$ python3 /opt/kafka/ejempr02/consumer.py []
hadoop@master:/opt/kafka_2.13-3.9.0$ python3 /opt/kafka/ejempr02/consumer.py []

```

Animación 7.2_Kafka: Ejemplo 2