Kafka
Introducción

Historia

Apache Kafka es un proyecto open-source orientado a las plataformas de software stream-processing desarrollado por Apache Software Foundation.

Principalmente fue desarrollado con el lenguaje de programación Scala, aunque más tarde también con Java.

Creado por Linkedin, **Kafka** es un proyecto Open Source mantenido principalmente por Confluent.

Proporciona una plataforma unificada, de alto rendimiento y baja latencia para manejar los datos en tiempo real.

Ventajas

High-throughput

Con un hardware pequeño, **Kafka** es capaz de manejar datos de alta velocidad y gran volumen.

Low-Latency

Rango de milisegundos, alrededor de 10ms.

Fault-Tolerant

Incluso fallando algún cluster o nodo, **Kafka** tiene capacidades inherentes para seguir funcionando.

Data-Durability

Los mensajes nunca se pierden.

Ventajas

Scalability

Agregación de nodos sin costes.
Manejo de mensajes totalmente transparentes .

Distributed

Replicación.
Particionamiento.

High Concurrency

Manejo de miles de mensajes al segundo.
Permite lectura/escritura de mensajes con alta concurrencia

Message Broker

Agente intermedio que maneja/traduce los mensajes.
Gestiona los productores/consumidores

Ventajas

By Default Persistent

Durabilidad de los mensajes.
Confiable.

Consumer Friendly

Consumidores políglotas.
Diferentes lenguajes de programación.
Diferentes comportamientos.

Batch Handling Capable

Permite el uso de funcionalidad ETL.

Real-Time Handling

Canaliza en tiempo real los datos.

Desventajas

Sin Tooling para el monitoreo

Carece de un conjunto de herramientas de gestión.
El uso empresarial tuvo rechazo.

No soporta ‘wildcard’ topics

Los tópicos solo pueden ser interpretados con una nombre absoluto.
No acepta nombres con patrones.

Poor Performance

En el caso de usar ‘pipelines’ en la entrega de mensajes, se obtiene una performance empobrecida.

Usos Populares

Linkedin

Movimiento de 7 trillones de mensajes al día.

UBER

Monitorización de +68 países y +350 ciudades.

NETFLIX

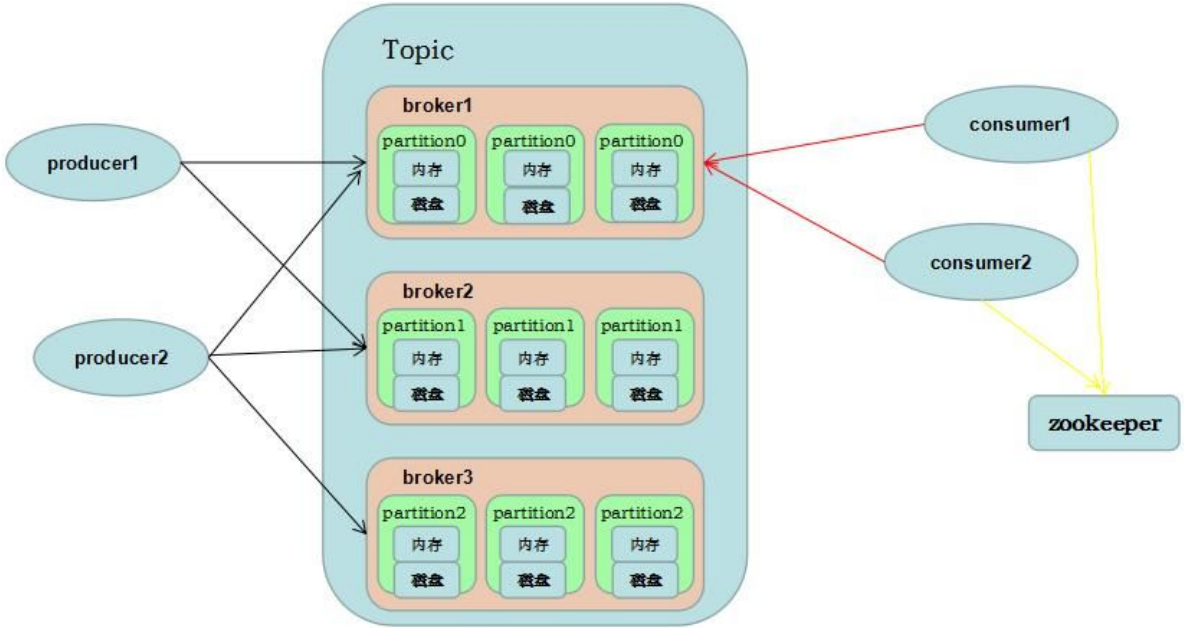
Movimiento de + de 1 trillón de mensajes.
6 Petabytes de información.

airbnb

Monitorización de logs.

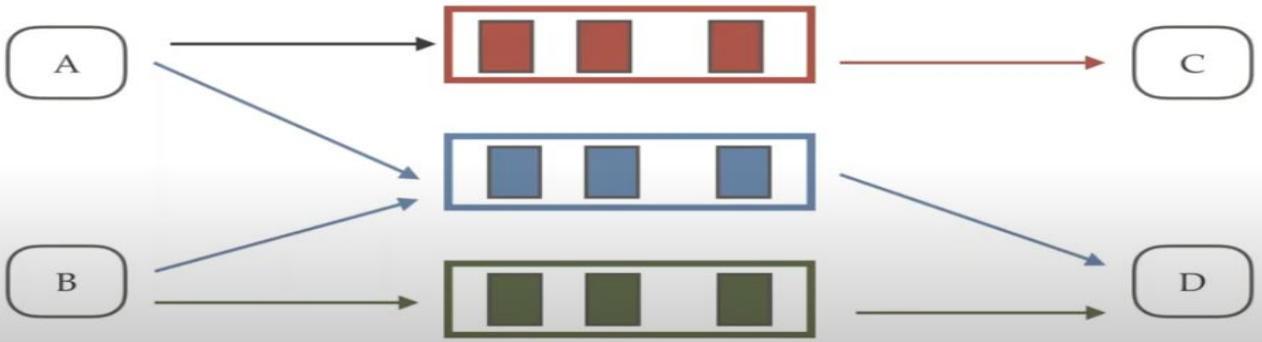
Estructura de un cluster Kafka

Estructura


Topic
broker1
partition0
partition0
partition0
内存
磁盘
broker2
partition1
partition1
partition1
broker3
partition2
partition2
partition2
producer1
producer2
consumer1
consumer2
zookeeper

Kafka I

- Existen productores y consumidores.



Envío de mensajes

- El cliente **Producer** es responsable de la distribución de mensajes. Cualquier intermediario del clúster de Kafka puede proporcionar información de metadatos al productor.
- Estos metadatos incluyen información como "lista de servidores supervivientes en el clúster" / "lista de líderes de particiones".
- Cuándo el productor obtiene la información de metadatos.
- Después de eso, el productor mantendrá las conexiones de socket con todos los líderes de partición en el tema.
- El productor envía el mensaje directamente al corredor a través del socket, sin ninguna "capa de enrutamiento" en el medio.
- De hecho, la partición a la que se enruta el mensaje la determina el cliente productor, por ejemplo, se puede utilizar "aleatoria", "key-hash", "polling"
- Si hay varias particiones en un topic, es necesario implementar una" distribución de mensajes equilibrada "en el lado del productor.
- En el archivo de configuración del lado del productor, el desarrollador puede especificar el método de enrutamiento de la partición.

Envío de mensajes

Puede especificar el **mecanismo de respuesta** correspondiente al enviar el mensaje Producer y establecer si los datos de envío requieren comentarios del servidor.

Hay tres valores: 0, 1, -1

- 0: El productor no esperará a que el corredor envíe un ack
- 1: Cuando el líder recibe el mensaje, envía un ack
- -1: envía un ack cuando todos los seguidores hayan sincronizado correctamente los mensajes

Forma de parámetro: *request.required.acks = 0*

Consumo de mensajes

Kafka sólo admite Topic, puede haber varios consumidores en cada grupo y cada consumidor pertenece a un grupo de consumidores.

En circunstancias normales, un grupo contendrá varios consumidores, lo que no solo puede mejorar el consumo simultáneo de mensajes en el topic, sino también mejorar la "tolerancia a fallos".

Si un consumidor en el grupo falla, las particiones consumidas serán otros consumidores se hacen cargo automáticamente.

Para un mensaje específico en un topic, solo será consumido por uno de los consumidores de cada grupo suscrito a este tema. Este mensaje no se enviará a varios consumidores en un grupo, y todos los consumidores en un grupo serán intercalados.

Para todo el topic, el consumo de mensajes del consumidor en cada grupo es independiente entre sí, y podemos considerar que un grupo es un "suscriptor".

Consumo de mensajes

En Kafka, el mensaje de una partición solo será consumido por un consumidor del grupo (al mismo tiempo).

Cada partición de un tema solo será consumida por un consumidor en un "suscriptor", pero un consumidor puede consumir mensajes de varias particiones al mismo tiempo.

El principio de diseño de kafka determina que para un tema, no puede haber más consumidores que particiones en el mismo grupo para consumir al mismo tiempo, de lo contrario significará que algunos consumidores no podrán captar el mensaje.

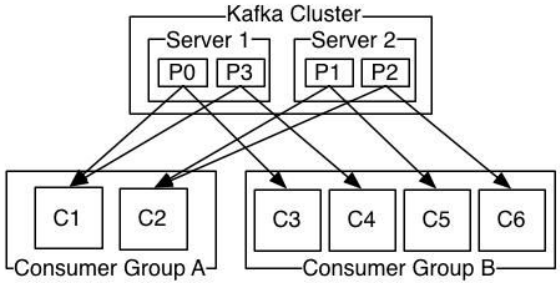
Kafka solo puede garantizar que los mensajes de una partición sean consumidos secuencialmente por un determinado consumidor; de hecho, desde la perspectiva de Topic, cuando hay varias particiones, los mensajes aún no están ordenados globalmente.

Equilibrio de carga

Cuando un consumidor se une o sale de un grupo, se activará el equilibrio de particiones. El objetivo final del equilibrio es mejorar la capacidad de consumo concurrente del topic.

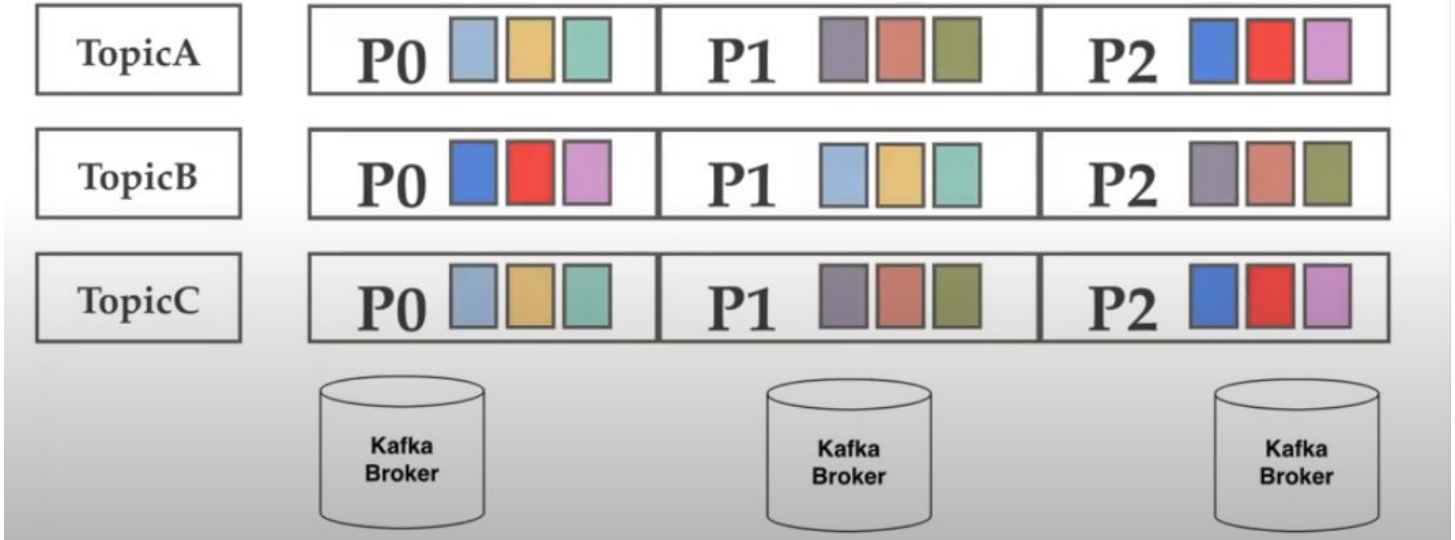
Los pasos son los siguientes:

1. Si el tema1 tiene las siguientes particiones: P0, P1, P2, P3
2. Únase al grupo, existen los siguientes consumidores: C1, C2
3. Primero clasifique las particiones según el número de índice de la partición: P0, P1, P2, P3
4. Ordenar según el consumidor id: C0, C1
5. Calcule el múltiplo: M = [P0, P1, P2, P3] .size / [C0, C1] .size, el valor de este ejemplo es M = 2 (redondeado hacia arriba)
6. Luego asigne particiones en secuencia: C0 = [P0, P1], C1 = [P2, P3], es decir, Ci = [P (i * M), P ((i + 1) * M -1)]

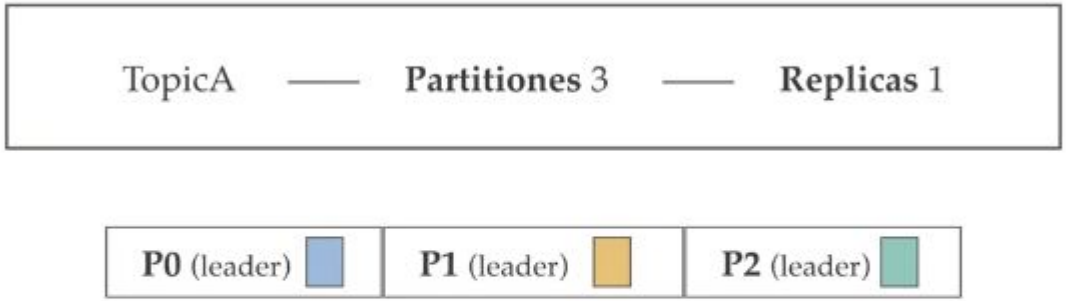


Topics: Particiones I

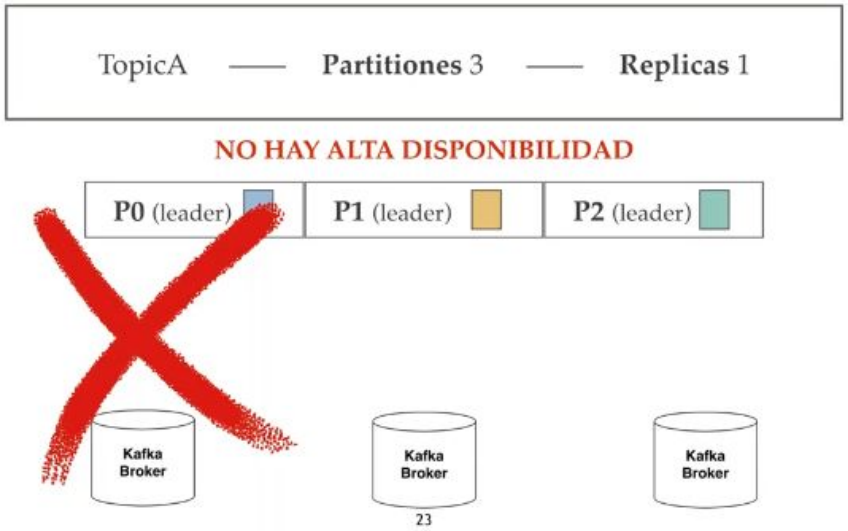
- Las particiones contienen los mensajes.



Topics: Replicas I

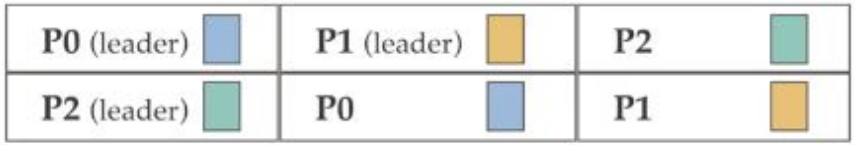


Topics: Replicas II

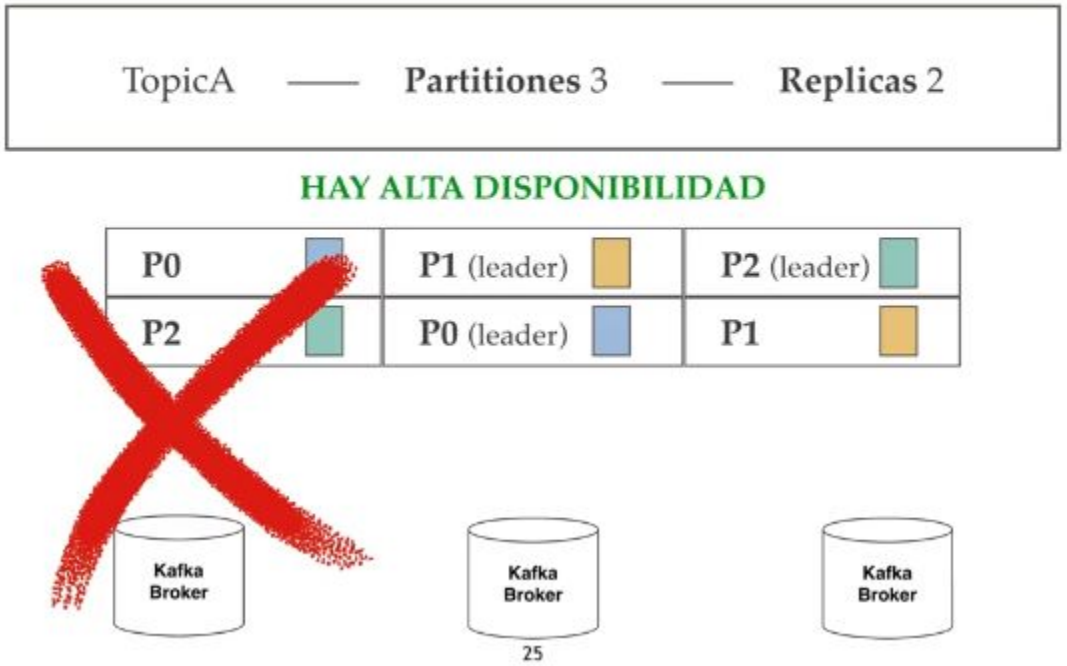


Topics: Replicas III

TopicA —— **Partitiones** 3 —— **Replicas** 2



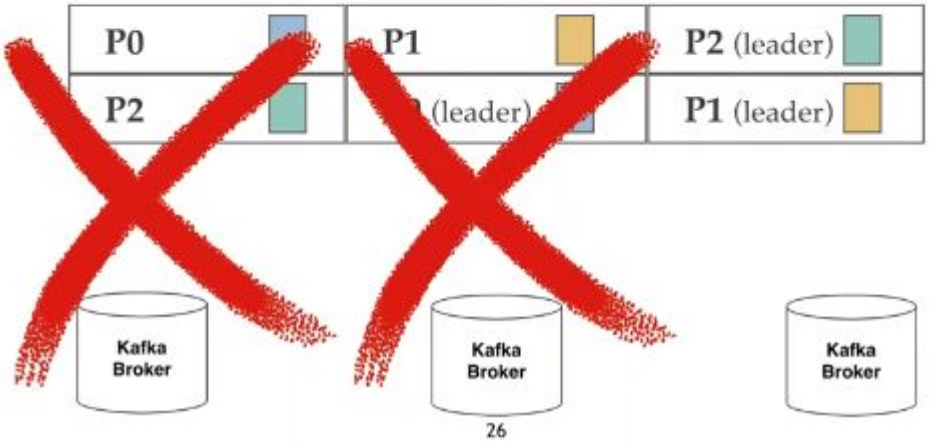
Topics: Replicas IV



Topics: Replicas V

TopicA —— **Partitiones** 3 —— **Replicas** 2

NO HAY ALTA DISPONIBILIDAD



Topics: Replicas VIII

Máximas replicas == Numero de brokers.

- Al incrementar las replicas hacemos el sistema mas robusto ante caídas.
- Las replicas implican un incremento del uso del ancho de banda entre los brokers.
- Las replicas provocan la disminución de la tasa de producción de mensajes por segundos a un topic, si se activa el asentimiento por replica.

Topics: Particiones II

- Cada partición es un fichero que se encuentra en el disco.
- Los ficheros son denominados **logs**.
- Internamente cada mensaje dentro de log es identificado por un **offset**.

```
root@kschool:~# ls -lR /tmp/kafka-logs/topic1-*

/tmp/kafka-logs/topic1-0:
-rw-r--r-- 1 root root 10485760 dic  7 17:01 00000000000000000000.index
-rw-r--r-- 1 root root        0 dic  7 17:01 00000000000000000000.log

/tmp/kafka-logs/topic1-1:
-rw-r--r-- 1 root root 10485760 dic  7 17:01 00000000000000000000.index
-rw-r--r-- 1 root root      175 dic  7 17:10 00000000000000000000.log
```

Teorema CAP y posición de Kafka

Teorema CAP

El teorema CAP (también llamado Conjetura de Brewer) enuncia que es imposible para un sistema de cómputo distribuido garantizar simultáneamente,

- La consistencia (Consistency), es decir, cualquier lectura recibe como respuesta la escritura más reciente o un error.
- La disponibilidad (Availability), es decir, cualquier petición recibe una respuesta no errónea, pero sin la garantía de que contenga la escritura más reciente.
- La tolerancia al particionado (Partition Tolerance), es decir, el sistema sigue funcionando incluso si un número arbitrario de mensajes son descartados (o retrasados) entre nodos de la red.

Posición de Kafka frente al teorema CAP

Kafka es un sistema de CA

Según los ingenieros en LinkedIn (donde Kafka se fundó inicialmente).

> *"Todos los sistemas distribuidos deben hacer compensaciones entre garantizar la coherencia, la disponibilidad y la tolerancia de partición (Teorema CAP). Nuestro objetivo era admitir la replicación en un clúster Kafka dentro de un solo centro de datos, donde la partición de la red es rara, por lo que nuestro diseño se enfoca en mantener réplicas altamente disponibles y altamente consistentes. Una fuerte consistencia significa que todas las réplicas son byte a byte idéntico, lo que simplifica el trabajo de un desarrollador de aplicaciones."*

"CA pero no P" significa que cuando ocurre una partición de red arbitraria, cada Kafka topic-partición deja de servir solicitudes (pierda A), o pierde algunos datos (pierda C), o ambos, dependiendo de su configuración y detalles específicos de la partición.

Posición de Kafka frente al teorema CAP

- Si una partición de red divide todos los ISR de Zookeeper, con la configuración predeterminada unclean.leader.election.enable = false, no se pueden elegir réplicas como líder **(perder A)**.

- Si al menos un ISR puede conectarse, será elegido, por lo que aún puede atender solicitudes **(preservar A)**. Pero con el valor predeterminado min.insync.replicas = 1 un ISR puede estar a la zaga del líder aproximadamente replica.lag.time.max.ms = 10000. Entonces, al elegirlo Kafka potencialmente tira las escrituras confirmadas a los productores por el ex líder **(pierde C)**.

- Kafka puede **preservar tanto A como C** para algunas particiones limitadas. P.ej. tienes min.insync.replicas = 2 y replication.factor = 3, y las 3 réplicas están sincronizadas cuando ocurre una partición de red, y se divide a lo sumo 1 ISR (fallas de un solo nodo, o una falla de CC única o una falla de enlace de CC cruzada).

- Para **conservar C** para particiones arbitrarias, debe establecer min.insync.replicas = replication.factor. De esta manera, no importa qué ISR se elija, se garantiza que tendrá los datos más recientes. Pero al mismo tiempo, no podrá atender solicitudes de escritura hasta que la partición sane **(pierda A)**.

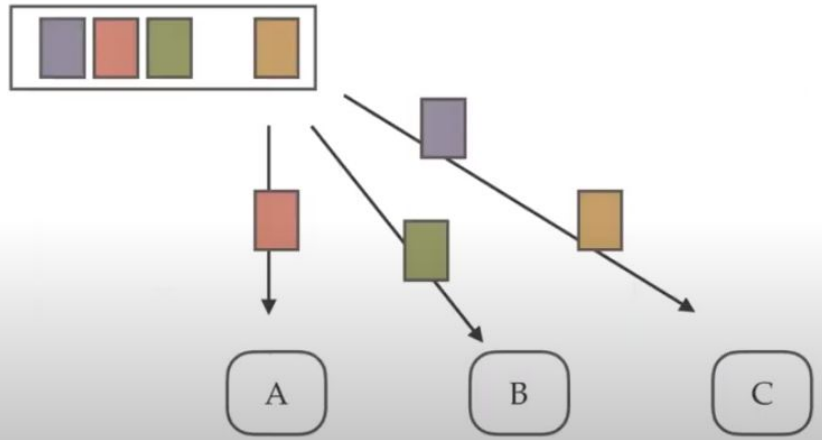
Modalidades en los sistemas de mensajes

Características de los sistemas de mensajes o colas

- Este tipo de comunicación se utiliza sobre todo para la **comunicación asíncrona.**
- Requiere de otra pieza que sirve de **intermediario (broker)** donde se publican los topics.
- Proporciona un **sistema centralizado** que permite la publicación de tipos genéricos de datos y que puede evolucionar con el tiempo.

Kafka II

- **Modelo de colas:** Los mensajes son repartidos.



Patrón "Publish / Subscribe Messaging"

Patrón de uso dentro de la tipología de arquitectura de "Cola de mensajes", utilizado para la comunicación entre aplicaciones. Por lo tanto, se engloba más en el movimiento de información, aunque también puede cumplir aspectos de preparación o modificación.

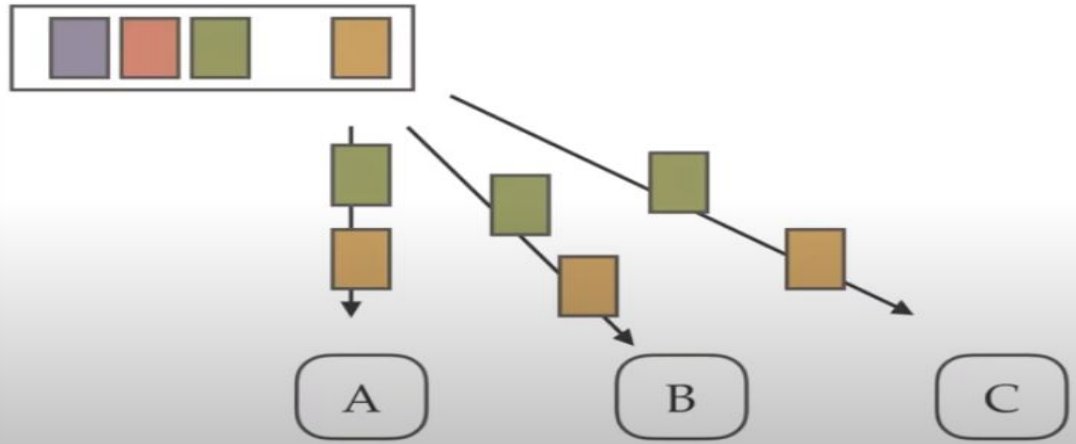
También se denomina *"publish / subscribe"*, *"publicador / suscriptor"* o *"productor / consumidor"*.

Existe un elemento **"publisher"** (publicador / remitente / emisor / productor) que al **generar un dato** (message / mensaje / record / registro) **no** lo **dirige** o referencia específicamente a un **"subscriber"** (receptor / subscriptor / suscriptor) en concreto, es decir, no lo envía de forma directa a la "dirección" del subscriber.

Para ello se **dispone** de **listas de temas/topics** publicados específicos y un **conjunto de suscriptores**, el productor trata de clasificar el mensaje en base a una tipología, lo pone en la lista de un tema específico y el receptor se suscribe a la listas para recibir ese tipo de mensajes.

Kafka III

- **Modelo publicador/subscriptor**: Los mensajes se difunden en broadcast.



Pretexto

En aplicaciones distribuidas, un protocolo de comunicación tiene que poder comunicarse con componentes tanto de sistema como a medida.

Un flujo síncrono entorpece y bloquea las llamadas entre componentes.

Una mensajería asíncrona permite desacoplar los escenarios de envío/recibo de información.

A veces tener colas de mensajes dedicadas para cada consumidor no es lo más óptimo.
Puede que ciertos consumidores puedan estar interesados en un subconjunto de dicha información.

¿Cómo un remitente puede enviar eventos a todos los consumidores sin conocerlos?

Solución

Introducción de un subsistema de mensajería asíncrona con la participación de los siguientes elementos :

Publisher

Es el encargado de empaquetar los eventos en mensajes.
Conoce el formato de los mensajes.
Envía los mensajes a través del canal de entrada.

Input Channel

Canal de información por donde viajan los mensajes del remitente (publisher).

Broker

Orquestador que distribuye los mensajes entrantes en los canales de comunicación saliente.

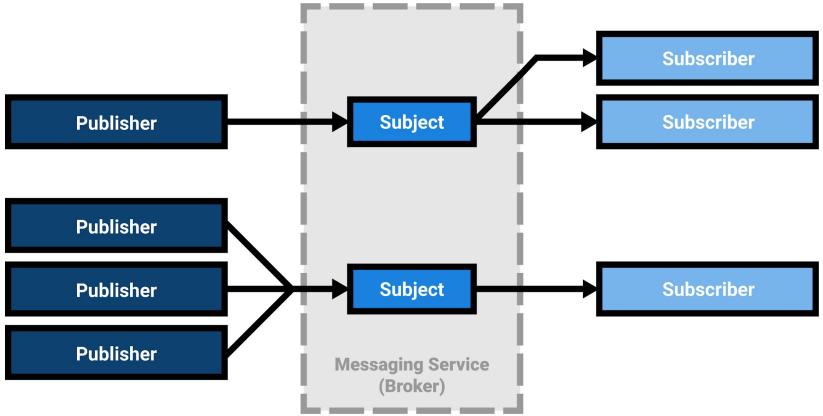
Output Channel

Canal de información por donde viajan los mensajes que van a los consumidores (subscribers).

Solución

Subcriber

Es el encargado de desempaquetar los mensajes.
Conoce el formato de los mensajes.
Recibe los mensajes a través del canal de salida.



Ventajas

Desacoplamiento

Los subsistemas se establecen de manera independiente.

Asincronicidad

Los mensajes llegan en tiempo diferido sin esperas ni bloqueos.
Consumidores esperan a procesar los mensajes sin costos.

Alta escalabilidad

Los publicadores envían rápidamente los mensajes a sus canales de entrada.
Responsabilidades isoladas.

Desventajas

Seguridad

Los mensajes no van cifrados.
Se pueden interceptar.

Wildcard

No se pueden usar tópicos con definición de patrones (*).

Orden de los mensajes

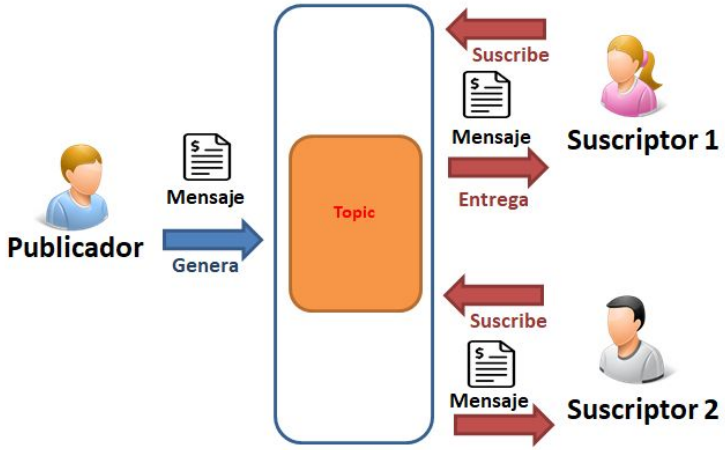
No se garantiza orden.
Necesidad de clasificar mensajes como idempotentes (se pueden repetir).

Expiración de los mensajes

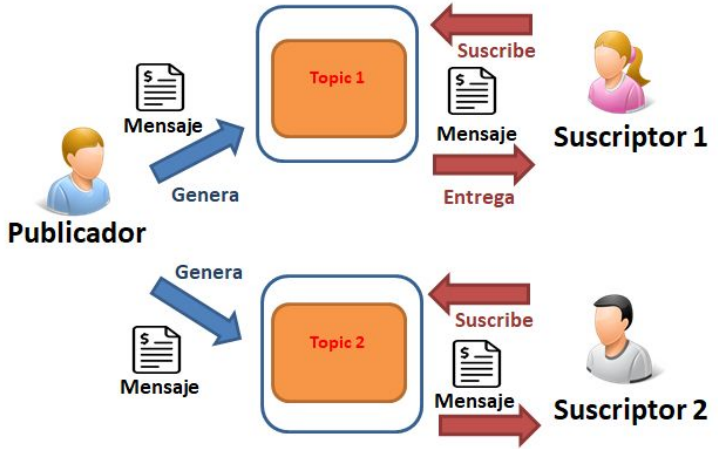
Los mensajes nunca se descartan.
Necesidad de que el Publisher pueda especificar un tiempo de vida de ese mensaje.

Modalidades

Un único tema/topic y dos suscriptores

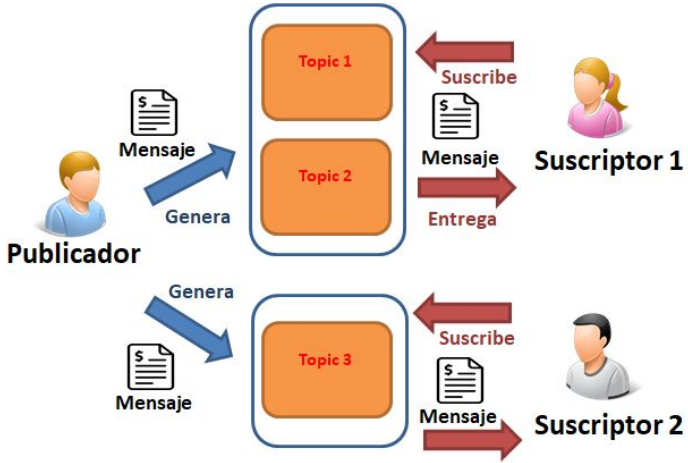


Dos temas/topics y un suscriptor por tema/topic

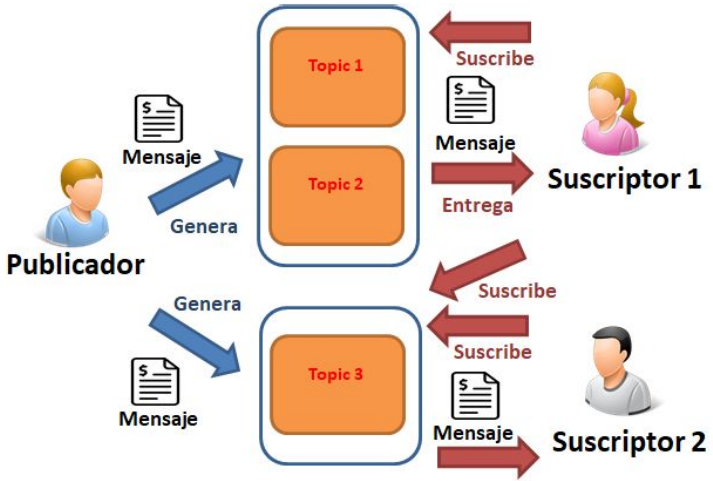


Modalidades

Varios temas/topics y un suscriptor con varios temas/topics específicos



Varios temas/topics y uno de ellos es compartido por varios suscriptores



Tipos de configuración

Orden en el que se **seleccionan los clientes** de la cola para recibir el servicio.

La reglas habituales de selección suelen ser:

- FIFO (primero en entrar, primero en salir). La suposición más habitual.
- LIFO (último en entrar, primero en salir). Aplicable a sistemas de inventarios.
- Aleatoria
- Procedimiento de prioridad. (Aplicable en servicios de emergencia.)

Estrategias de entrega

- "Como máximo una vez" (at most once): Sólo se envía una vez, garantiza que no hay duplicados pero puede perderse.
- "Al menos una vez" (at least once): Garantiza que no hay pérdida.
- "Exactamente una vez" (exactly once): Garantiza que no hay duplicados y que tampoco hay pérdidas.

Estrategias de ordenación

- No hay ordenación (No ordering): No importa el orden de recepción y por lo tanto se puede incrementar el rendimiento.
- Ordenación por partición (Partitioned ordering): Asegurar un orden por partición y por lo tanto tiene un coste extra.
- Ordenación Global (Global Order): Se requiere un orden en los datos por lo tanto necesita un mayor coste extra de los recursos y posibles implicaciones en el rendimiento.

Componentes de Kafka

Topics, mensajes y particiones

Un topic es un flujo de datos sobre un tema en particular.

Podemos crear tantos topics como queramos y estos serán identificados por su nombre.

Los topics pueden dividirse en particiones en el momento de su creación.

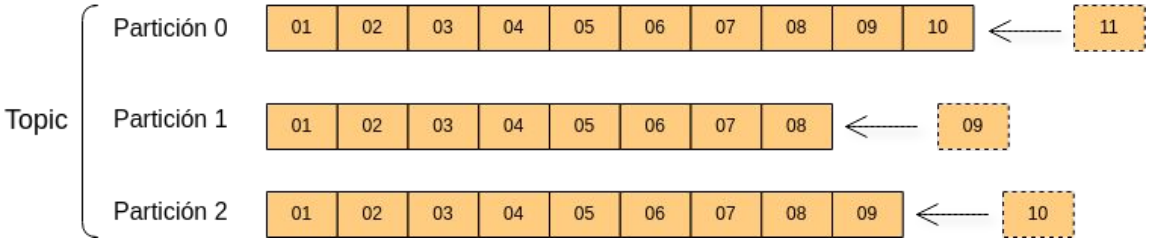
Cada elemento que se almacena en un topic se denomina mensaje.

Los mensajes 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.

Topics, mensajes y particiones

Cada mensaje dentro de una partición tiene un identificador numérico incremental llamado offset.

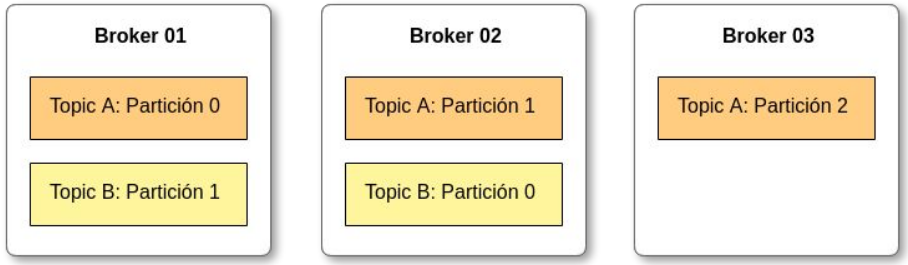
Aunque los mensajes se guarden en los topics por un tiempo limitado (una semana por defecto) y sean eliminados, el offset seguirá incrementando su valor.



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.



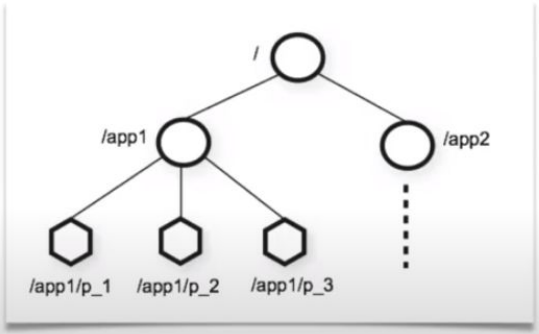
Zookeeper

Se trata de un servicio centralizado imprescindible para el funcionamiento de Kafka, al cual envía notificaciones en caso de cambios como: creación de un nuevo topic, caída de un broker, levantamiento de un broker, borrado de topics, etc.

Su labor principal es gestionar los brokers de Kafka, manteniendo una listado con sus respectivos metadatos y facilitar mecanismos para health checking. Además, ayuda en la selección del broker líder para las distintas particiones de los topics.

ZooKeeper I

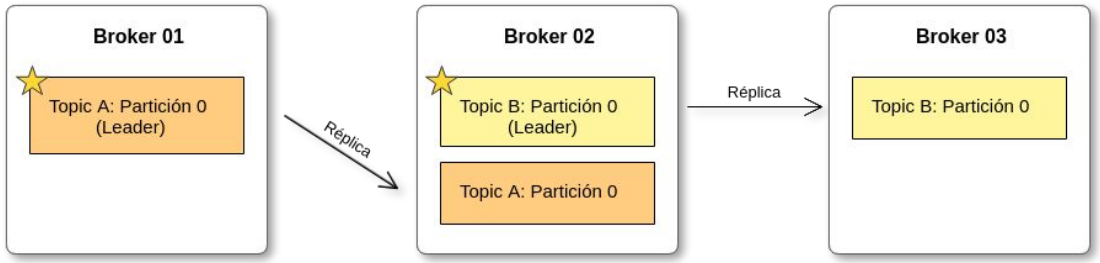
- Proporciona una estructura similar a la que se encuentra en un sistemas de ficheros.
- Cada elemento de la estructura es conocido como **ZNode**.
- Los ZNodes pueden contener información en su interior. Por ejemplo un JSON String.



Topic replication

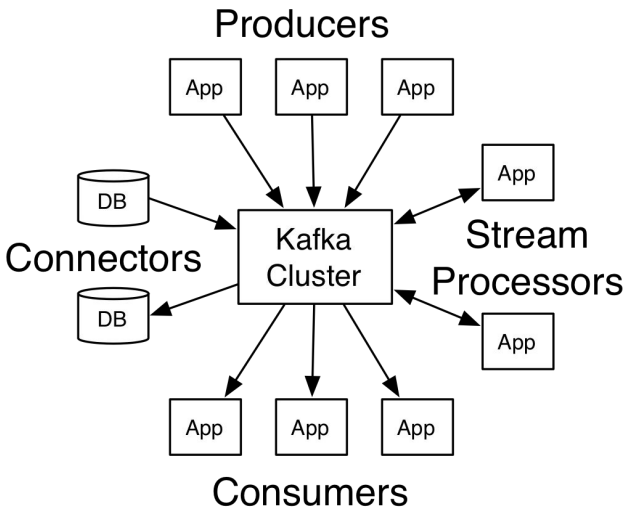
Los topics deberán tener un factor de replicación > 1 (normalmente 2 y 3), de esta forma si un broker se cae, otro broker puede servir los datos.

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. Si este se cae, se cambia el líder.



API

Una vez comentada la estructura de Apache Kafka, vamos a ver cómo se interactúa con él mediante cuatro API's



Producers

Permite que una aplicación pueda publicar mensajes de 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.

Consumers

Con su uso podemos suscribirnos a un topic de Kafka y consumir sus mensajes para poder tratarlos en 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.

Consumers

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.

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. Si un consumidor se cae, cuando vuelva a ser levantado seguirá leyendo datos desde donde se quedó anteriormente.

Consumers Groups

Es un método utilizado por Kafka para realizar la transmisión (a todos los consumidores) y unidifusión (a cualquier consumidor) de un mensaje de tema.

Un tema puede tener varios CG. Los mensajes de tema se copiarán (no se copiarán realmente, conceptualmente) a todos los CG, pero cada partición solo enviará el mensaje a un consumidor en el CG.

Si necesita implementar la transmisión, siempre que cada consumidor tenga un CG independiente. Para lograr unidifusión, siempre que todos los consumidores estén en el mismo CG.

Con CG, los consumidores también pueden agruparse libremente sin enviar mensajes a diferentes temas varias veces.

Stream Processors

Se trata de una librería para crear aplicaciones que nos permite consumir un stream de datos de un topic para poder realizar modificaciones sobre los mensajes y escribir en otro topic actuando como productor, es decir, la entrada y la salida de datos son almacenados en el cluster de Kafka.

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

Entre sus características destacan su alta capacidad de procesamiento de mensajes por segundo, su escalabilidad y una alta tolerancia a fallos.

Connectors

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 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.

Web de los conectores oficiales