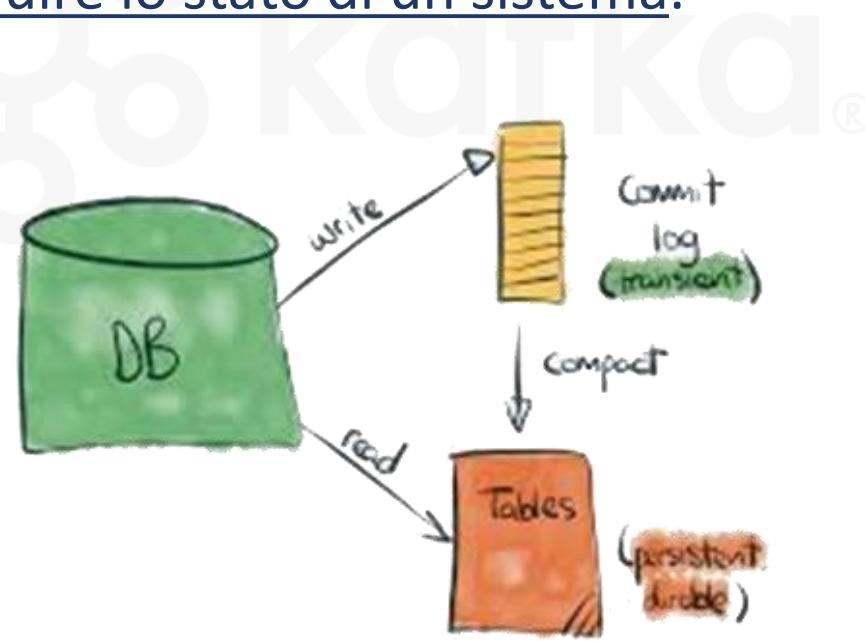


APACHE
kafka[®]

UNIT

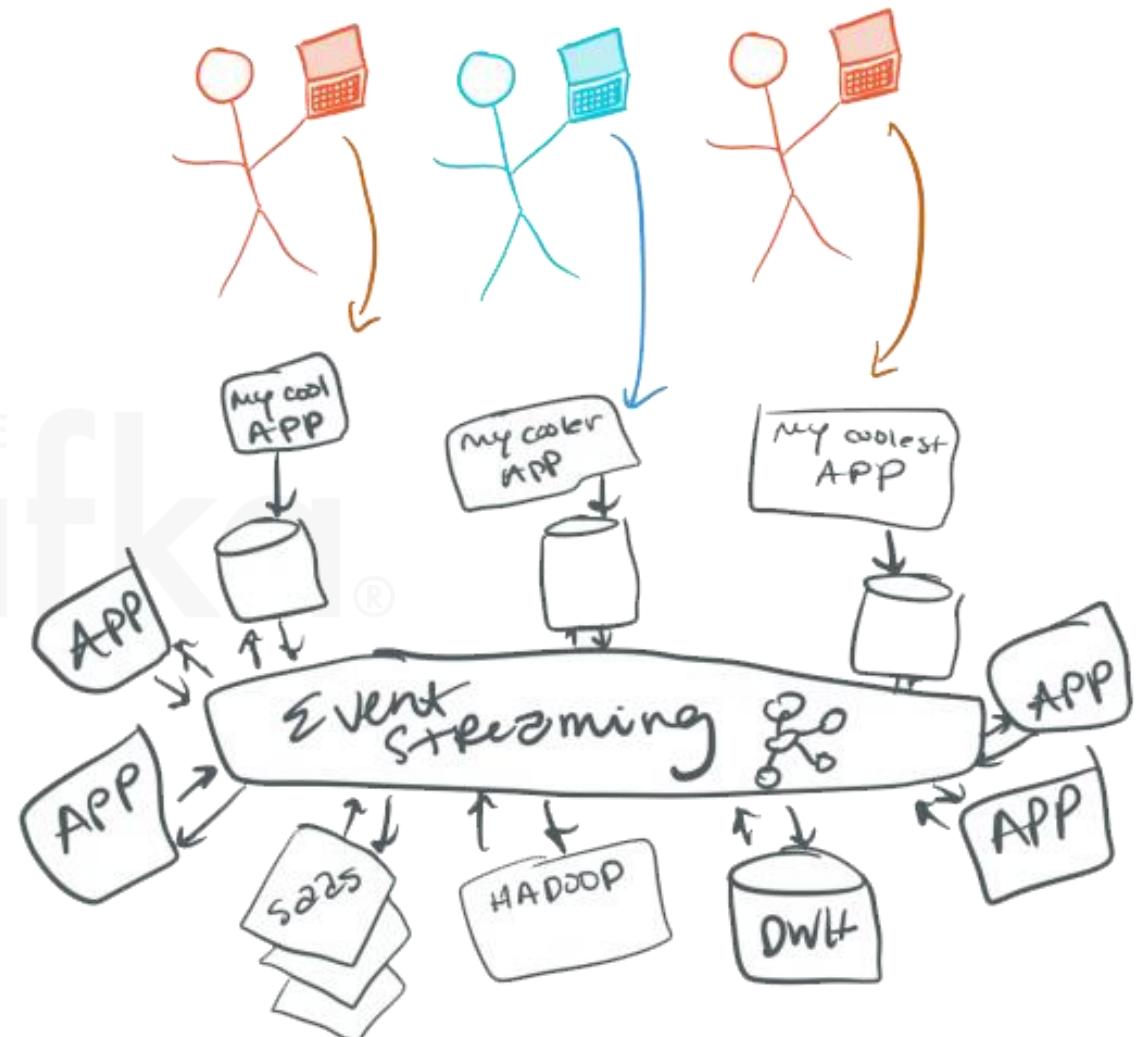
Kafka

- Viene spesso descritto come un «**distributed commit log**» (registro di commit distribuito) o più recentemente come «**distributing streaming platform**» (piattaforma di streaming distribuita).
- Ad esempio, un **log di commit** del filesystem o del database è progettato per fornire una registrazione duratura di tutte le transazioni in modo che possano essere riprodotte in modo coerente al fine di costruire lo stato di un sistema.



Distributing streaming platform

- Allo stesso modo, i dati all'interno di Kafka vengono memorizzati in modo durevole, in ordine e possono essere letti in modo deterministico.
- I dati possono essere distribuiti all'interno del sistema per fornire ulteriori protezioni contro i «**failures**» e per abbattere i tempi di esecuzione senza perdite delle prestazioni.



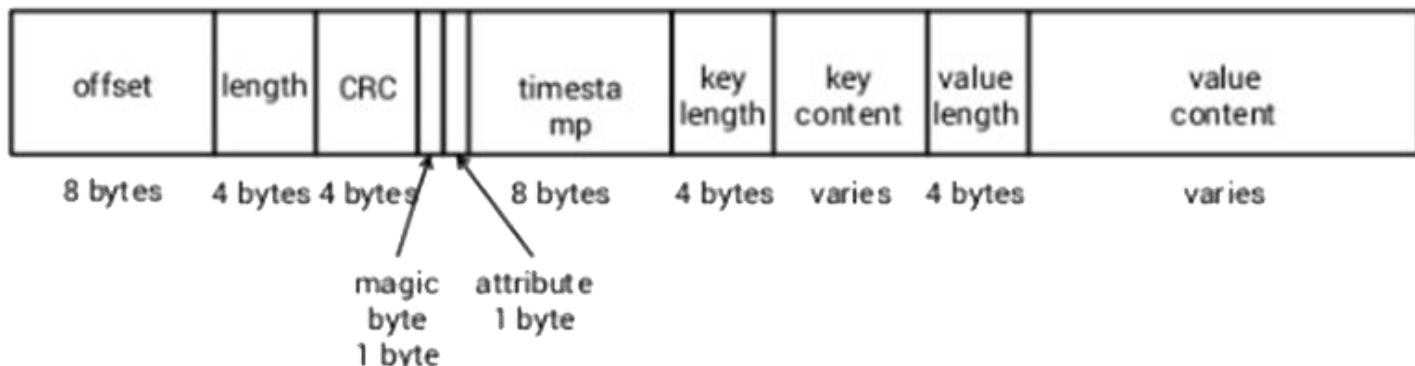
UNIT

Kafka Message

- L'unità di dati all'interno di Kafka è chiamata **messaggio**; in un parallello con un database si può considerare un messaggio simile a un record.
- Per Kafka un messaggio è semplicemente **un array di byte**, motivo per cui i dati in esso contenuti non hanno un formato o un significato specifico.
- Un messaggio può avere un **bit opzionale di metadati**, che viene definito **chiave**.
- Anche la chiave è una matrice di byte e, come nel caso del messaggio, **non ha alcun significato specifico per Kafka**.
- I messaggi sono costituiti da un'intestazione a lunghezza variabile, una matrice di byte (opaque value) di lunghezza variabile e una matrice di byte (opaque value) di lunghezza variabile.

Messages Format - On-disk format of a RecordBatch

```
1 baseOffset: int64
2 batchLength: int32
3 partitionLeaderEpoch: int32
4 magic: int8 (current magic value is 2)
5 crc: int32
6 attributes: int16
7     bit 0~2:
8         0: no compression
9         1: gzip
10        2: snappy
11        3: lz4
12        4: zstd
13     bit 3: timestampType
14     bit 4: isTransactional (0 means not transactional)
15     bit 5: isControlBatch (0 means not a control batch)
16     bit 6~15: unused
17 lastOffsetDelta: int32
18 firstTimestamp: int64
19 maxTimestamp: int64
20 producerId: int64
21 producerEpoch: int16
22 baseSequence: int32
23 records: [Record]
```



- Un «Control Batch» contiene un singolo record chiamato «Control Record».
- I «Control Record» non devono essere passati alle applicazioni; al contrario, vengono utilizzati dai consumer per filtrare i messaggi transazionali «aborted».
- Lo schema per il valore di un «Control Record» dipende dal valore «type» che è di tipo byte array (opaque value).
- Lo schema è:

```
1 version: int16 (current version is 0)
2 type: int16 (0 indicates an abort marker, 1 indicates a commit)
```

- Le intestazioni (Headers) a livello record sono state introdotte in Kafka 0.11.0.
- Di seguito viene delineato il formato su disco di un record con intestazioni.

Record

```
1 length: varint  
2 attributes: int8  
3     bit 0~7: unused  
4 timestampDelta: varint  
5 offsetDelta: varint  
6 keyLength: varint  
7 key: byte[]  
8 valueLen: varint  
9 value: byte[]  
10 Headers => [Header]  
--
```

Record Header

```
1 headerKeyLength: varint  
2 headerKey: String  
3 headerValueLength: varint  
4 Value: byte[]
```

- Le chiavi vengono utilizzate quando i messaggi devono essere **scritti su partizioni in modo più controllato**.
- Lo schema più semplice è generare un **hash coerente della chiave**, quindi concatenare il numero di partizione per quel messaggio con il risultato del modulo hash e il numero totale di partizioni nell'argomento.
- Ciò assicura che i messaggi con la stessa chiave vengano sempre scritti nella stessa partizione.

- Per efficienza, i messaggi vengono scritti in **modalità batch**.
- Un batch è solo una raccolta di messaggi, tutti prodotti nello stesso argomento e medesima partizione.
- Un singolo «**roundtrip**» attraverso la rete per ciascun messaggio comporterebbe un sovraccarico eccessivo; l'uso del batch lo riduce.
- Naturalmente si tratta di un compromesso tra **latenza e velocità effettiva**: più grandi sono i batch, più messaggi possono essere gestiti per unità di tempo, ma più tempo impiega un singolo messaggio a propagarsi.
- Anche i batch sono generalmente compressi, fornendo un trasferimento e una memorizzazione dei dati più efficienti a scapito della potenza di elaborazione.

UNIT

Message Schemas

- Poiché i messaggi sono solo array di byte per Kafka, si consiglia di imporre una struttura aggiuntiva o uno schema al contenuto del messaggio in modo che possa essere facilmente compreso.
- Esistono molte opzioni disponibili per lo schema dei messaggi, a seconda delle esigenze individuali delle applicazioni coinvolte.
- Schemi come **JSON** (Javascript Object Notation) e **XML** (Extensible Markup Language), sono facili da usare e leggibili per l'uomo ma mancano di funzionalità come la gestione affidabile dei tipi e la compatibilità tra le versioni dello schema.

- Molti sviluppatori di Kafka preferiscono l'uso di **Apache Avro**;
- **Avro** è un framework di serializzazione originariamente sviluppato per Hadoop.
- **Avro** fornisce:
 - un formato di serializzazione compatto
 - schemi separati dai payload dei messaggi che non richiedono la generazione di codice quando cambiano
 - forte tipizzazione dei dati
 - evoluzione dello schema, con compatibilità sia all'indietro che in avanti.



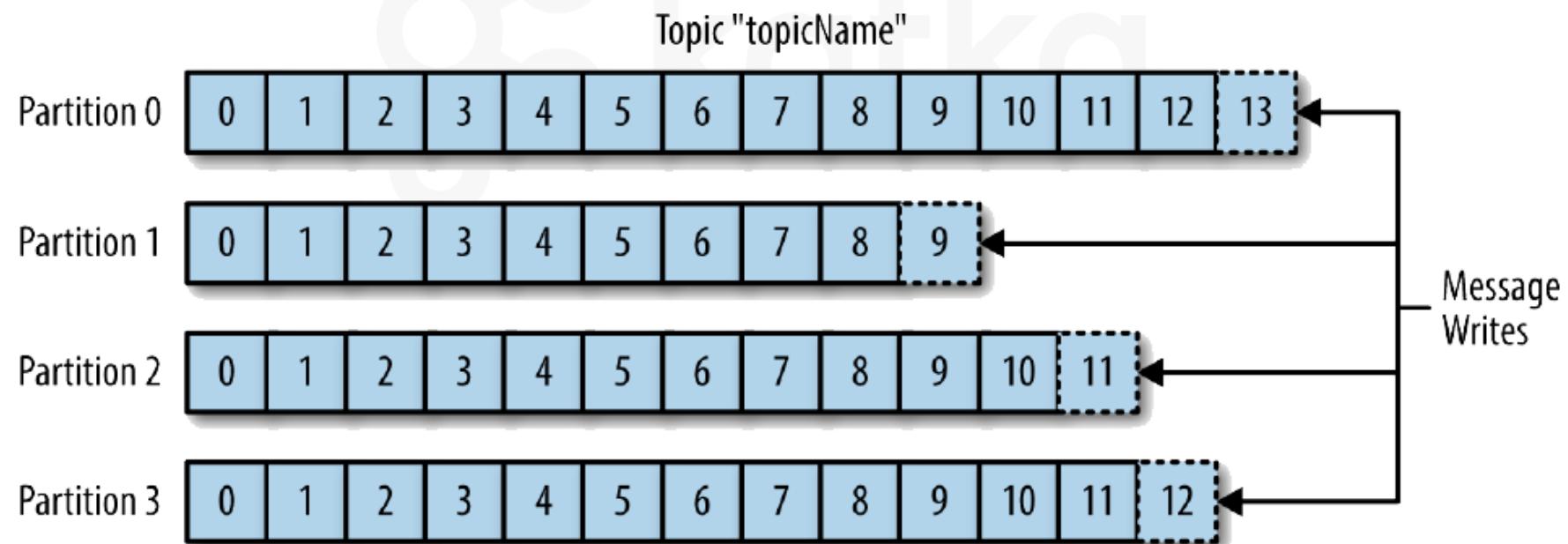
- Un formato di dati coerente è importante in Kafka, in quanto consente di **disaccoppiare la scrittura e la lettura dei messaggi**.
- Quando queste attività sono strettamente associate, le applicazioni che si «iscrivono» ai messaggi **devono essere aggiornate** per gestire il nuovo formato di dati, parallelamente al vecchio formato.
- Solo così le applicazioni che pubblicano i messaggi possono essere aggiornate per utilizzare il nuovo formato.
- Usando schemi ben definiti e memorizzandoli in un repository comune, i messaggi in Kafka possono essere compresi (da parte dei consumer) senza coordinamento riducendo al minimo o sforzo implementativo delle applicazioni.

UNIT

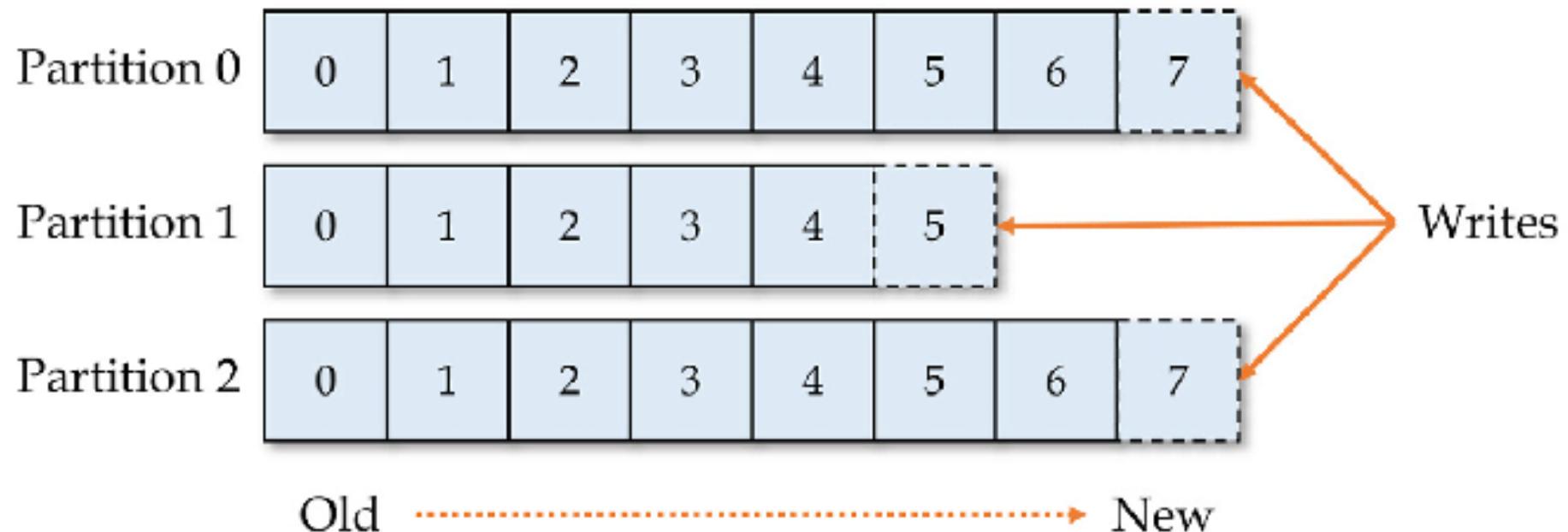
Topics And Partitions

- I messaggi in Kafka sono classificati in **Topic** (argomenti); le analogie più vicine per un argomento sono una tabella di database o una cartella in un filesystem.
- Gli argomenti sono inoltre suddivisi in diverse partizioni; tornando alla descrizione del «commit log», **una partizione è un singolo registro**.
- I messaggi vengono scritti in modalità «**append-only**» e vengono letti in ordine dall'inizio alla fine (Lista **FIFO**).
- Un argomento ha in genere più partizioni e **non esiste alcuna garanzia di ordinamento temporale dei messaggi** in tutto l'argomento all'interno di una singola partizione.

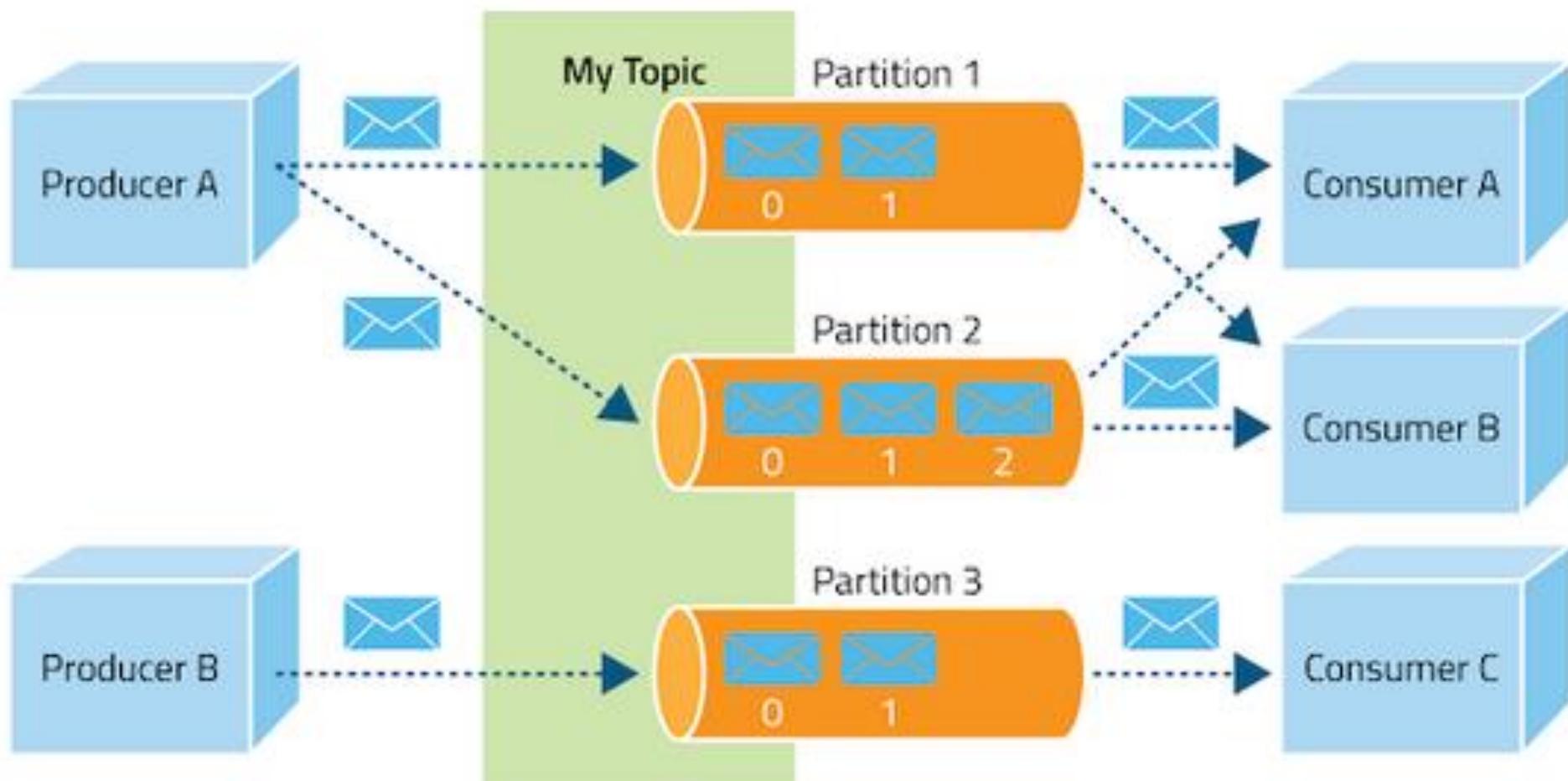
- La figura mostra un argomento con quattro partizioni, con le scritture aggiunte alla fine di ognuna. Le partizioni sono anche il modo in cui Kafka offre **ridondanza e scalabilità**.
- Ogni partizione può essere ospitata su un server diverso, il che significa che un singolo argomento può essere ridimensionato orizzontalmente su più server per fornire prestazioni ben oltre le capacità di un singolo server.



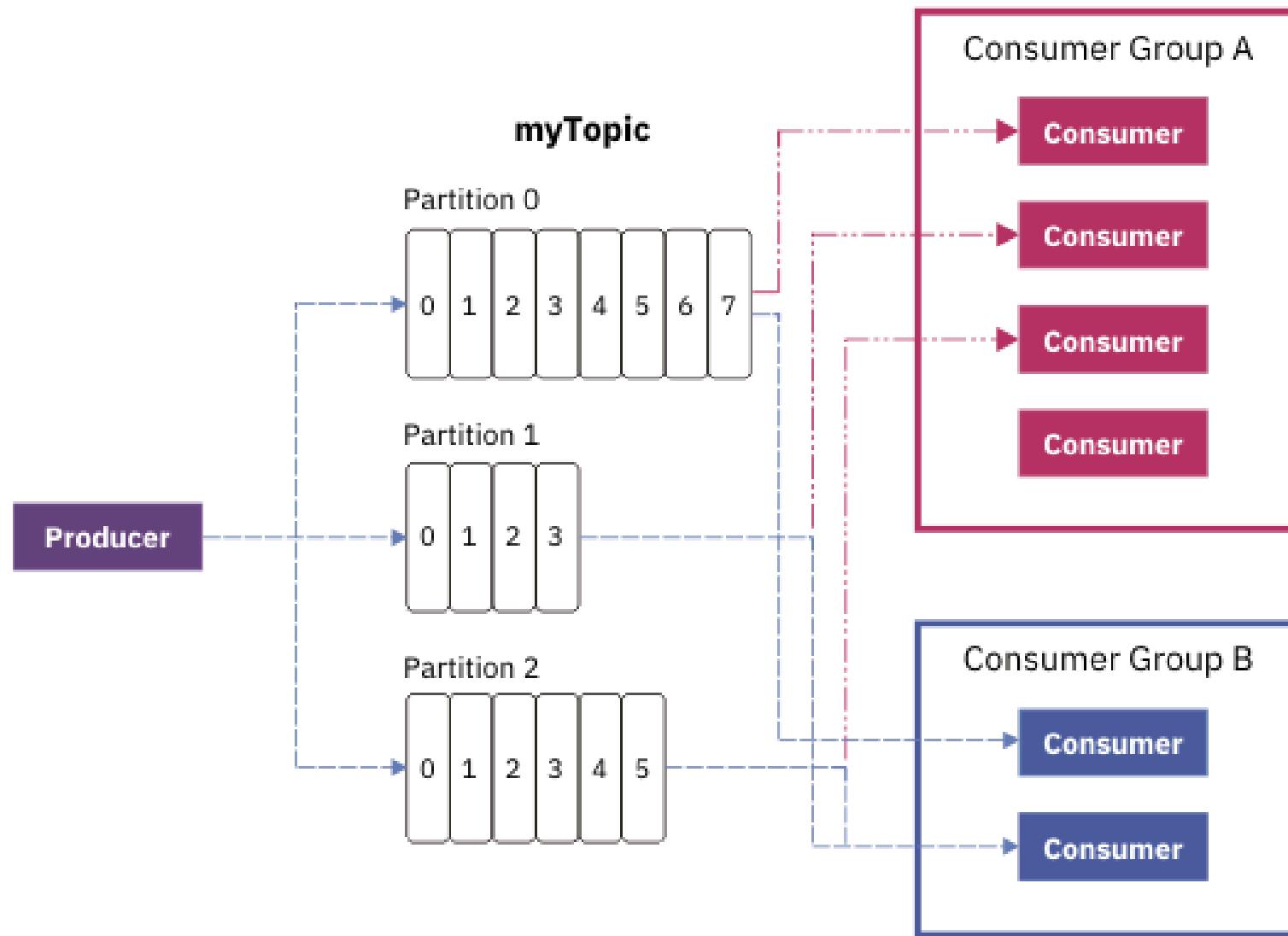
Topics and Partitions



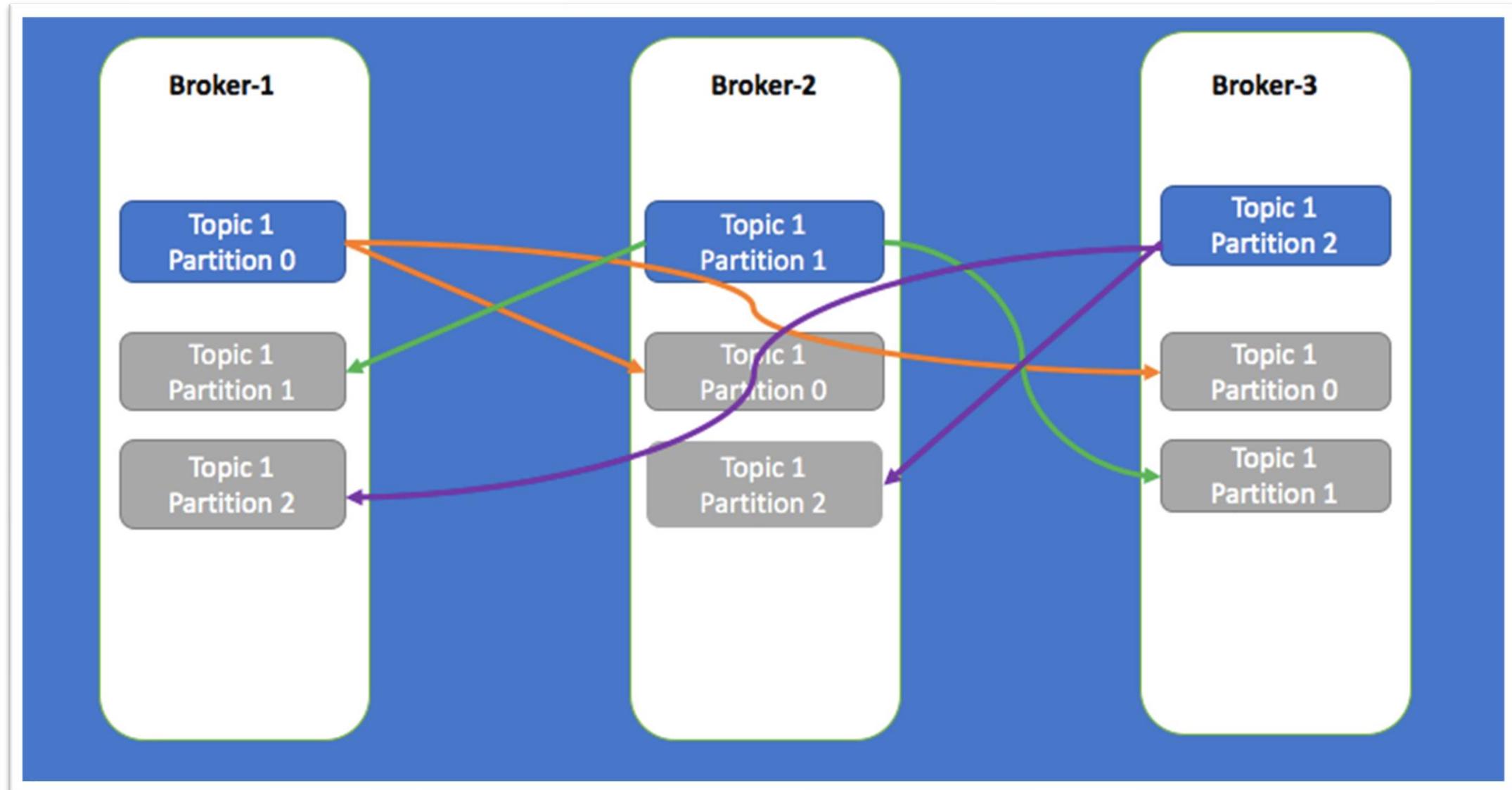
Topics and Partitions



Topics and Partitions

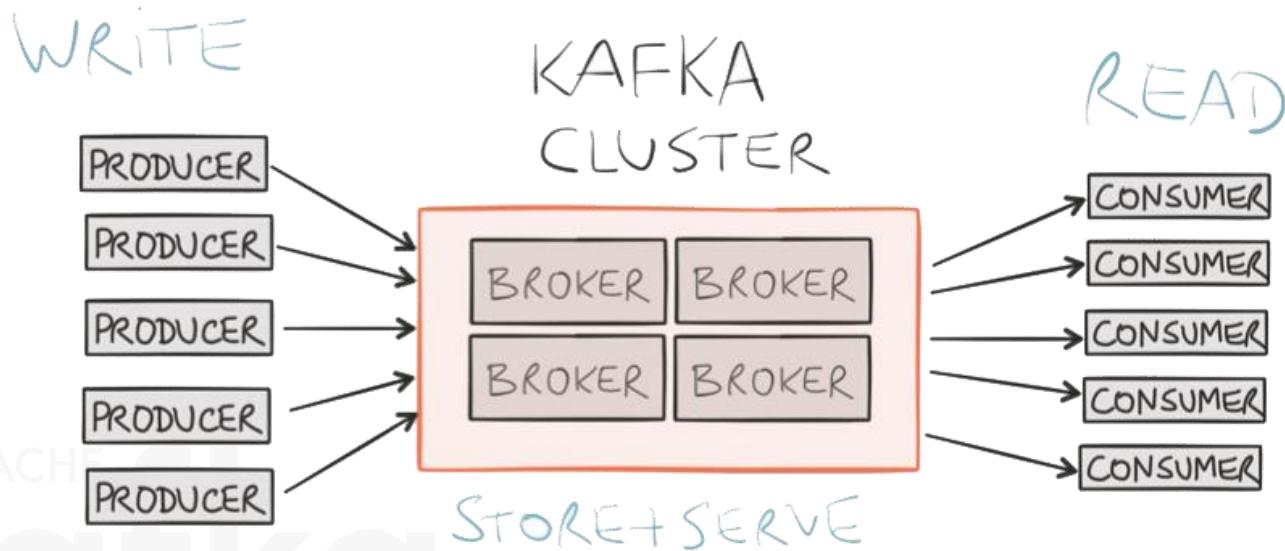


Topics and Partitions Cluster



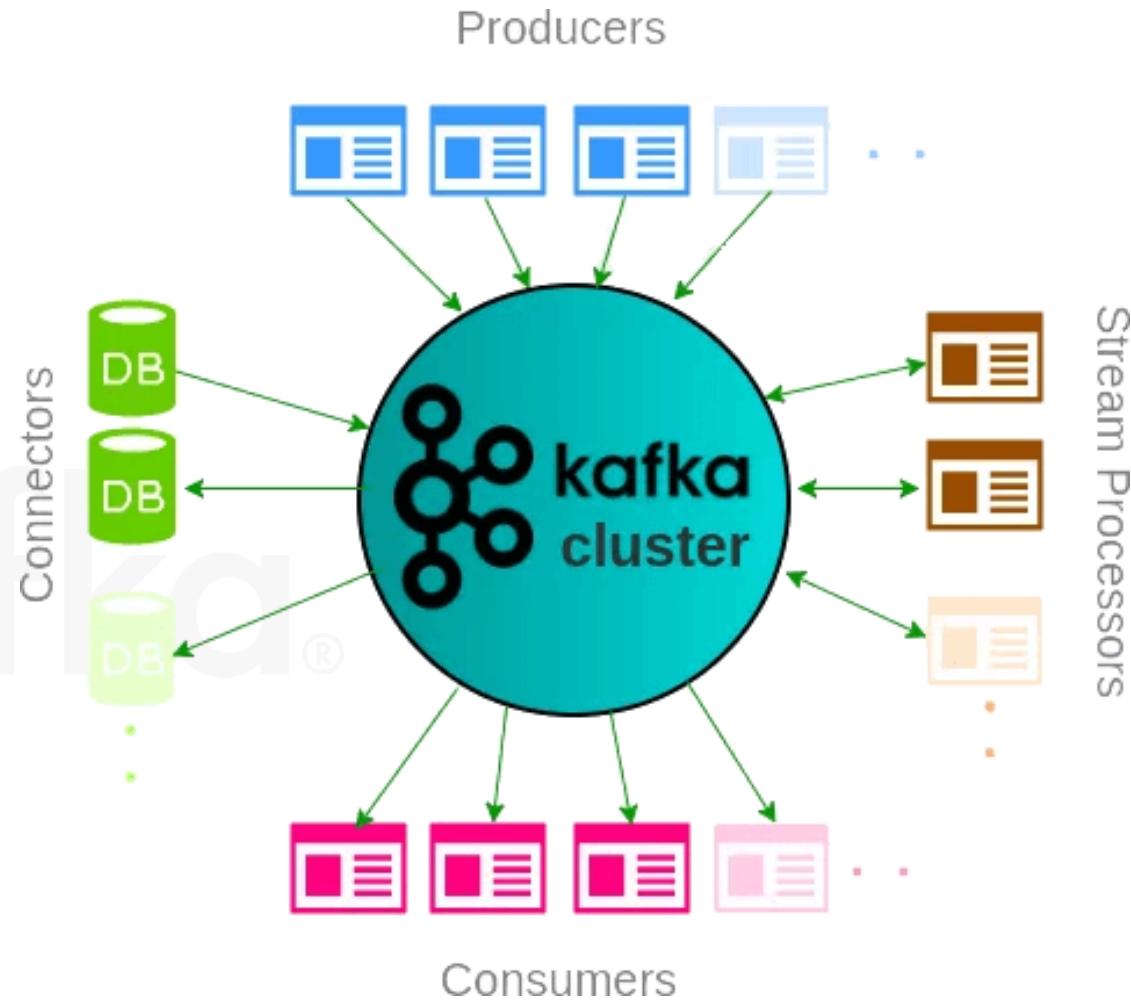
Stream

- Il termine «**Stream**» (flusso) viene spesso utilizzato quando si discute di dati all'interno di sistemi come Kafka.
- Molto spesso, uno stream è considerato un singolo argomento di dati, indipendentemente dal numero di partizioni ovvero un unico flusso di dati che si sposta dai producer ai consumer.
- In questo scenario si parla di «**elaborazione del flusso**» ove il contesto di elaborazione dei messaggi (nel nostro caso il sistema Kafka Streams) avviene in real-time.

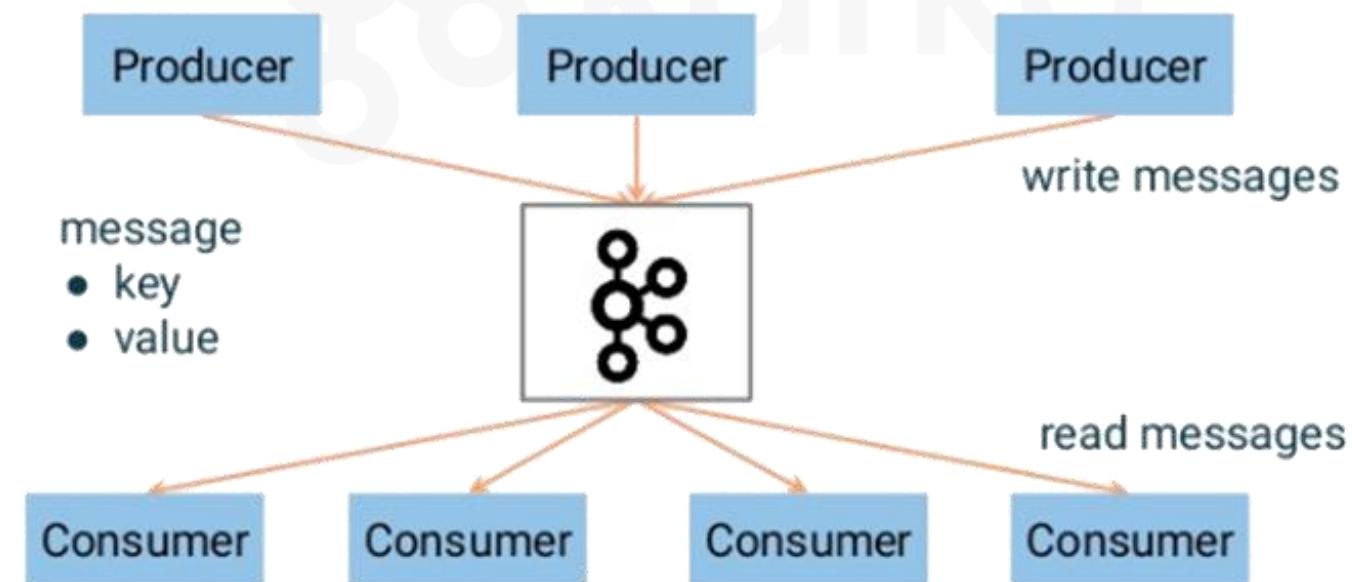


Producers and Consumers

- I client Kafka sono «utenti del sistema» e sono di due tipi:
 - Producer
 - Consumer.
- Esistono anche API client avanzate:
 - Kafka Connect API: per l'integrazione dei dati
 - Kafka Streams API: per l'elaborazione dei flussi.
- I client avanzati utilizzano producer e consumer come elementi costitutivi e forniscono funzionalità di livello superiore.

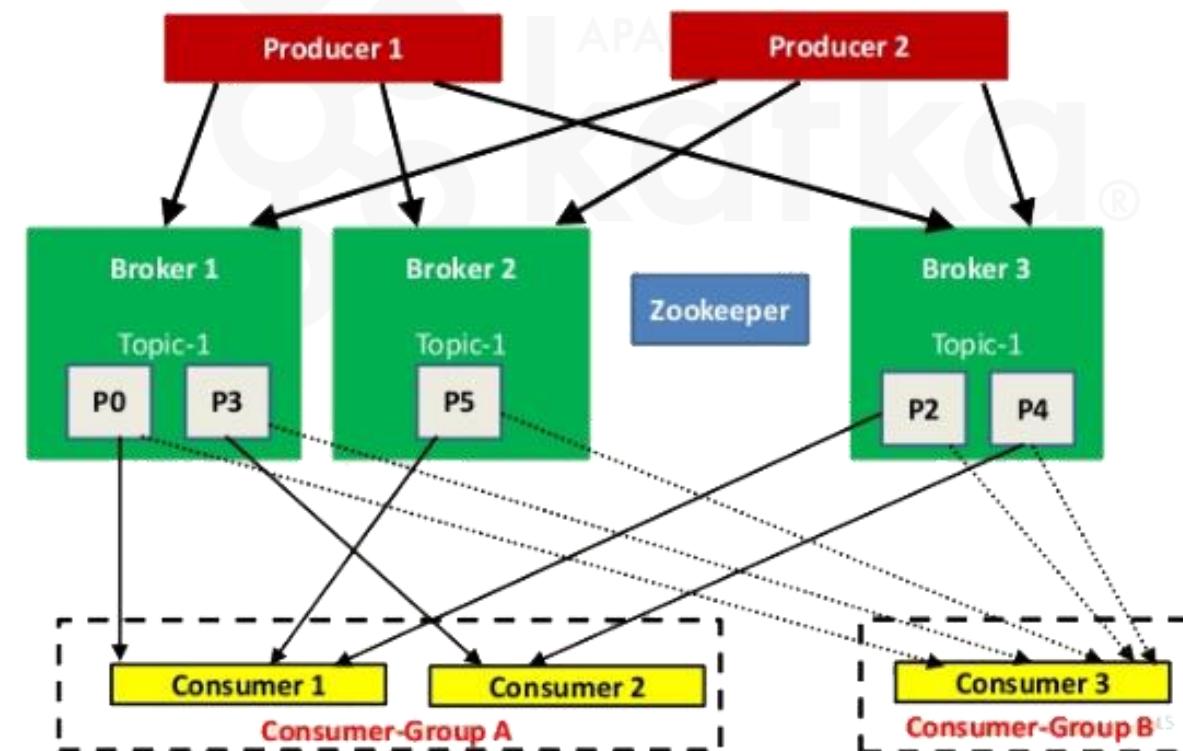


- I producer creano nuovi messaggi; in altri sistemi di Pub/Sub, questi possono essere chiamati **publishers** o **writers** (editori o scrittori).
- In generale, vengono prodotti messaggi su **argomenti specifici**.
- Per impostazione predefinita, al producer non importa in quale partizione viene scritto un messaggio specifico e bilancerà i messaggi su tutte le partizioni di un argomento in modo uniforme.



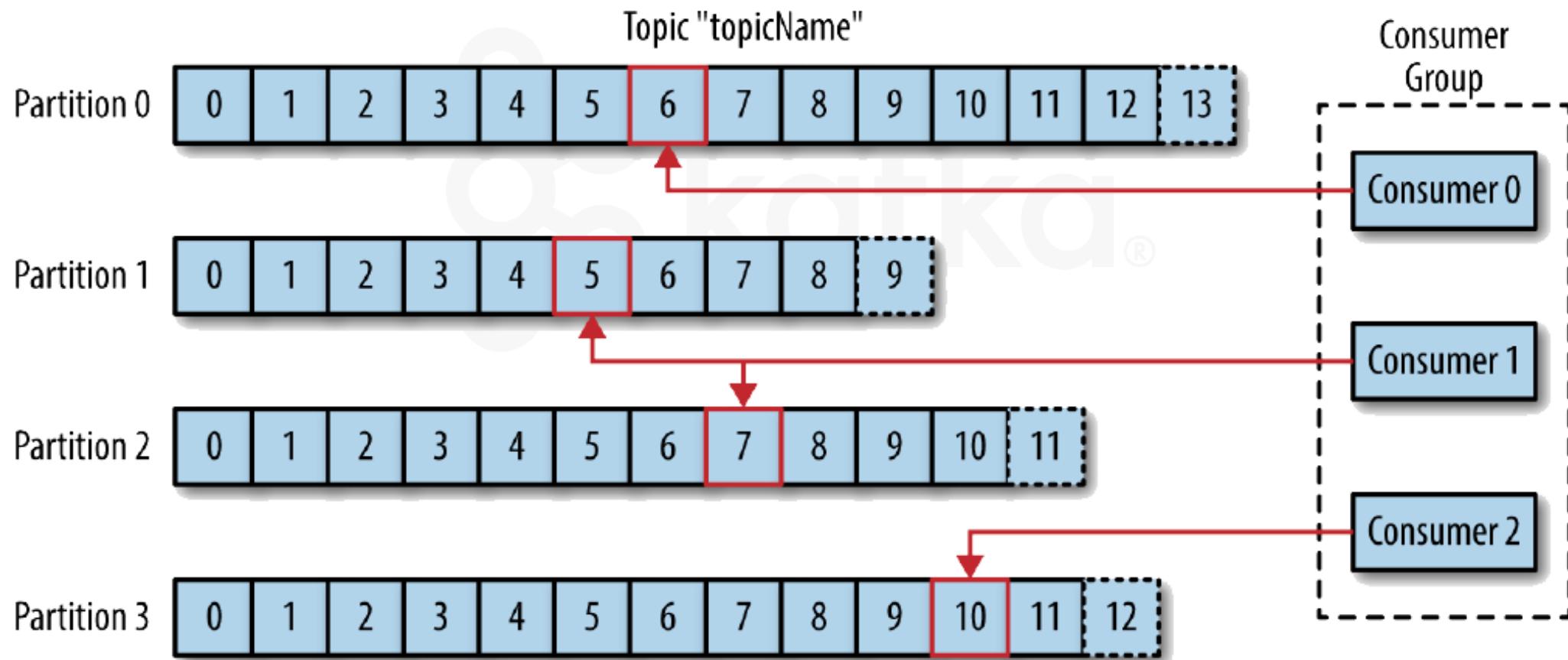
- In alcuni casi, il **producer** potrebbe indirizzare i messaggi a partizioni specifiche **utilizzando la chiave** del messaggio e un **partitioner** (partizionatore) il quale genererà un hash della chiave per poi mapparlo su di una partizione specifica.
- Questo assicura che tutti i messaggi prodotti con una determinata chiave verranno scritti nella stessa partizione.
- Il producer potrebbe anche utilizzare un **partitioner custom** che segue altre regole aziendali per **mappare i messaggi** sulle partizioni.

- I **consumer** leggono i messaggi; In altri sistemi di Pub/Sub, questi client possono essere chiamati **subscriber** o **reader** (abbonati o lettori).
- Il consumer sottoscrive **uno o più argomenti** e legge i messaggi nell'ordine in cui sono stati prodotti e **tiene traccia** di quali messaggi ha già consumato tramite l'**offset** dei messaggi stessi.



- L'offset è un altro bit di metadati, un valore intero che aumenta continuamente, che Kafka aggiunge a ciascun messaggio man mano che viene prodotto.
- Ogni messaggio in una determinata partizione ha un offset univoco; memorizzando l'offset dell'ultimo messaggio consumato per ogni partizione (in Zookeeper o in Kafka stesso), un consumer può essere stoppato e riavviato senza perdere il suo posto ovvero senza perdita di messaggi.

- I consumer lavorano come parte di un gruppo il cui scopo è **consumare un argomento**.
- Il gruppo assicura che **ogni partizione venga consumata da un solo** membro.



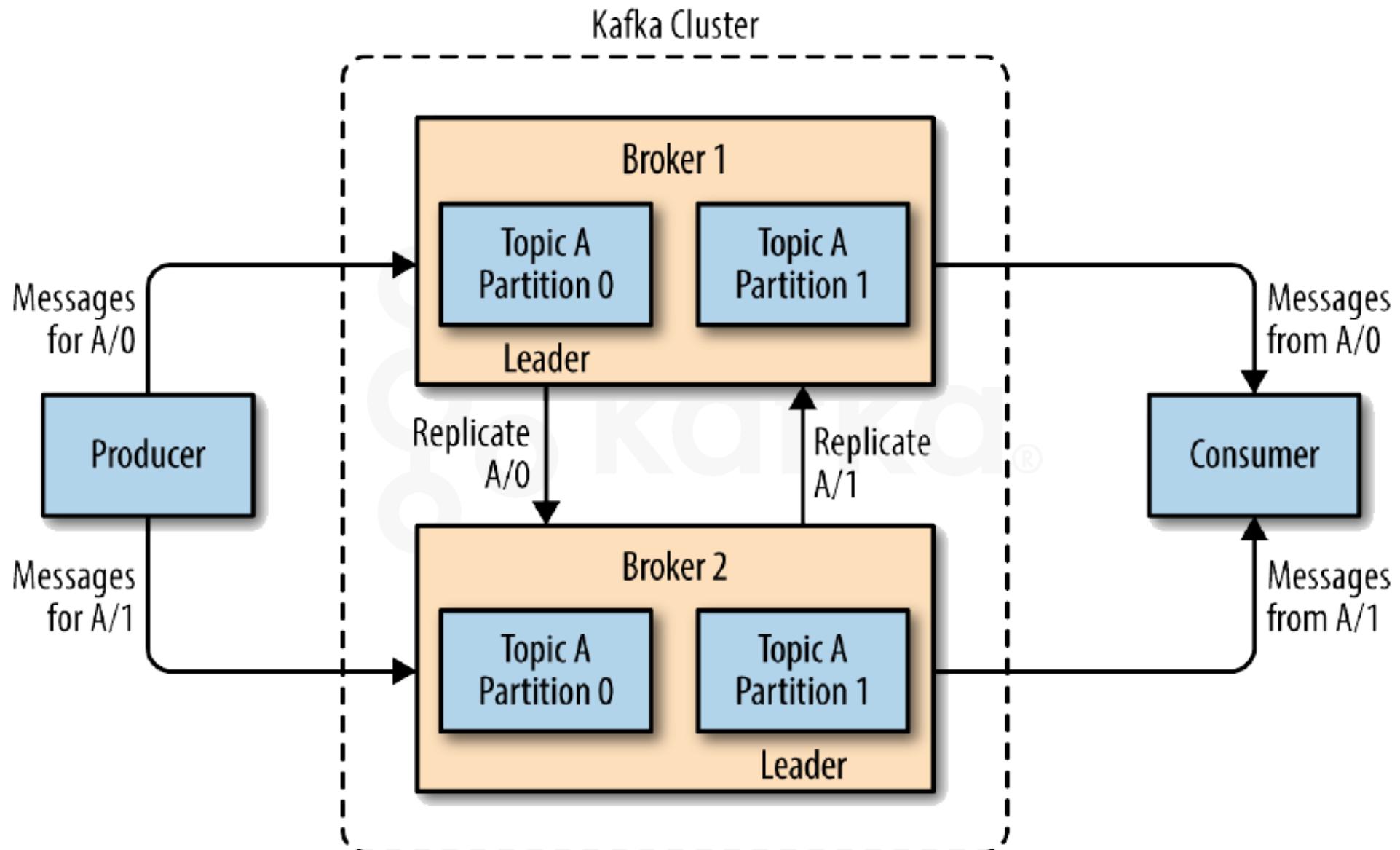
- In questo modo, i Consumer possono scalare in orizzontale per consumare argomenti con un gran numero di messaggi.
- Inoltre, se un singolo Consumer fallisce, i restanti membri del gruppo riequilibreranno le **partizioni** che vengono consumate per subentrare al membro mancante.
- La mappatura di un Consumer su di una partizione viene spesso definita **proprietà della partizione** da parte del Consumer.

UNIT

Brokers and Clusters

- Un singolo server Kafka è chiamato «**Broker**».
- Il Broker **riceve** i messaggi dai Producer, assegna loro gli offset e li **memorizza** sul disco.
- Fornisce inoltre assistenza ai Consumer, **rispondendo** alle richieste di recupero delle partizioni e rispondendo ai messaggi che sono stati **commitati** su disco.
- A seconda dell'**hardware** specifico e delle sue caratteristiche prestazionali, un singolo broker può gestire facilmente migliaia di partizioni e milioni di messaggi al secondo.

Replication di partitions in un cluster



- Una caratteristica chiave di Kafka è quella della «**Retention**» (conservazione), ovvero della persistenza dei messaggi per un certo periodo di tempo.
- I broker Kafka sono configurati con valori di default per la retention degli argomenti, al fine di conservare i messaggi per un certo periodo di tempo (ad es. 7 giorni) o fino a quando l'argomento non raggiunge una certa dimensione in byte (ad es. 1 GB).
- Una volta raggiunti questi limiti, **i messaggi scadono e vengono eliminati** in modo che la retention di default garantisca una quantità minima di dati disponibili in qualsiasi momento.

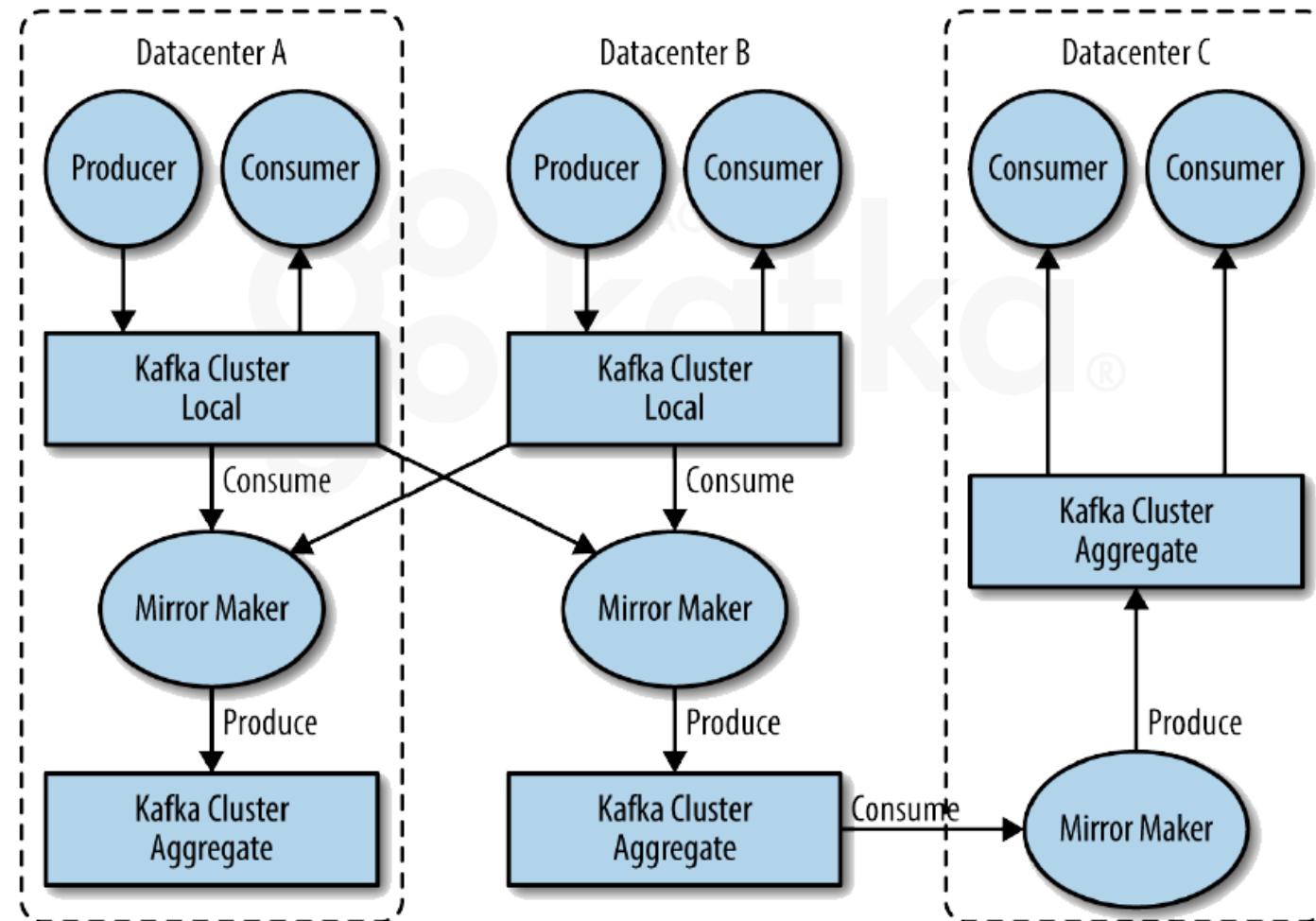
- E' possibile prevedere per ogni singolo argomento una propria **configurazione di default per la retention** in modo che i messaggi vengano archiviati solo finché sono utili in funzione dell'argomento stesso.
- Ad esempio: un argomento di tracciamento potrebbe essere conservato per diversi giorni, mentre le metriche dell'applicazione potrebbero essere conservate solo per alcune ore.
- Gli argomenti possono anche essere configurati come **log compattati**, il che significa che Kafka manterrà solo l'ultimo messaggio prodotto con una chiave specifica.
- Questo può essere utile per i dati di tipo «**changelog**», dove solo l'ultimo aggiornamento è di interesse.

- In ambienti «Enterprise» è vantaggioso disporre di più cluster; alcuni casi sono, ad esempio:
 - Segregazione di tipi di dati (letteralmente l'impossibilità per l'utente X di accedere ai dati dell'utente Y quando $X \neq Y$)
 - Isolamento per requisiti di sicurezza
 - Più datacenter (disaster recovery)
- L'obiettivo è che i messaggi vengano copiati tra i vari data center in modo tale che le applicazioni possono avere «sempre» accesso all'attività dell'utente sugli stessi data center.

- Ad esempio: se un utente modifica le informazioni pubbliche nel proprio profilo, tale modifica dovrà essere visibile indipendentemente dal centro dati in cui vengono visualizzati i risultati della ricerca.
- Ad Esempio: i dati di monitoraggio possono essere raccolti da molti siti in un'unica posizione centrale in cui sono ospitati i sistemi di analisi e di allerta.
- I meccanismi di replica all'interno dei cluster Kafka sono progettati solo per funzionare all'interno di un singolo cluster, non tra più cluster.

- Kafka include uno strumento chiamato **MirrorMaker**, utilizzato per questo scopo.
- MirrorMaker è semplicemente un consumer e producer Kafka, collegato con una **coda**.
- I messaggi vengono consumati da un cluster Kafka e prodotti per un altro.
- La natura semplice dell'applicazione nasconde la sua forza nel creare sofisticate **pipeline di dati**.

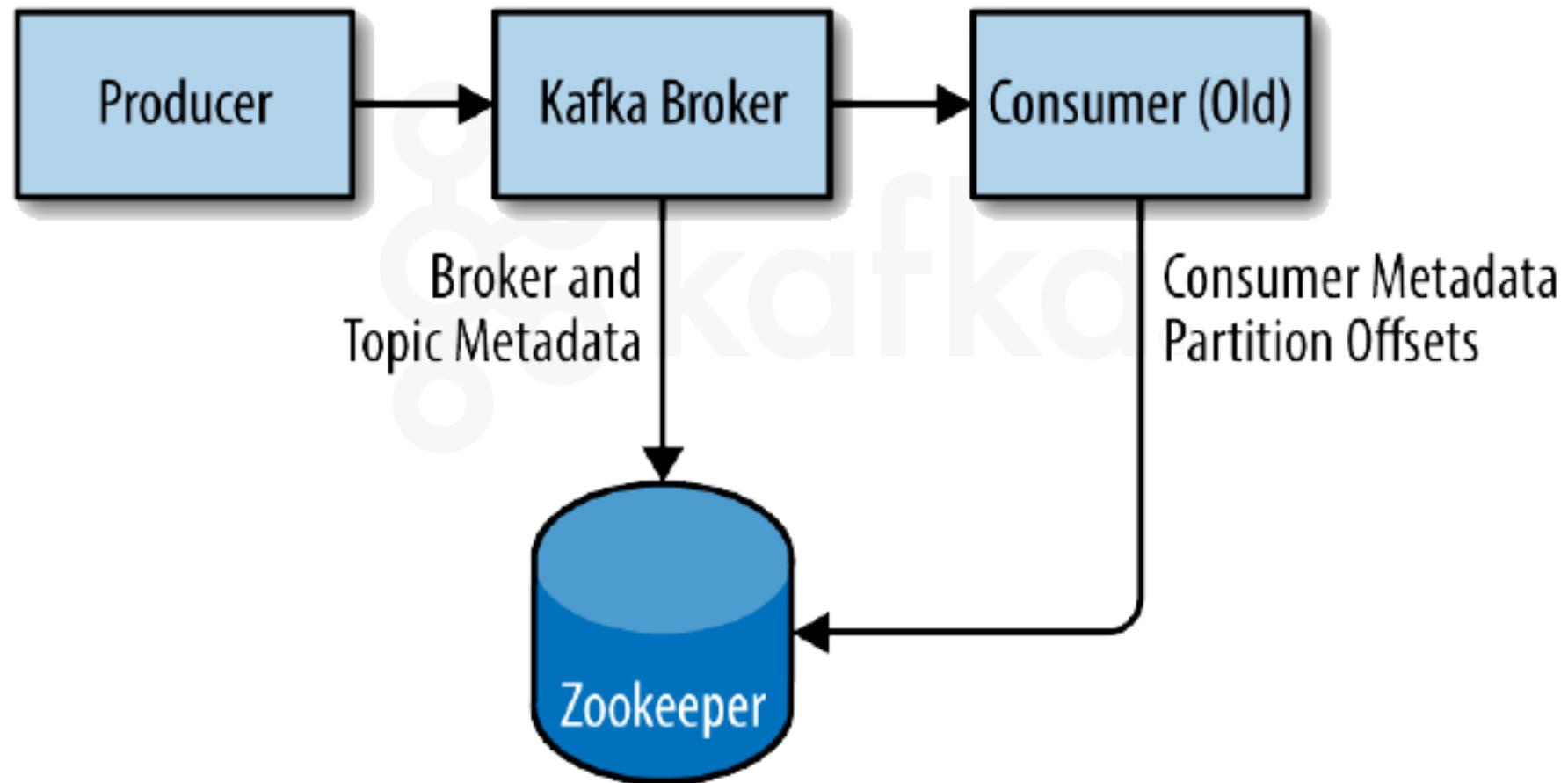
- La figura mostra un esempio di un'architettura che utilizza **MirrorMaker**, aggregando i messaggi da due cluster locali in un cluster aggregato e quindi copiando quel cluster in altri datacenter.



UNIT

Zookeeper

- Apache Kafka utilizza **Zookeeper** per archiviare i metadati relativi al cluster Kafka, nonché i dettagli dei consumer client.

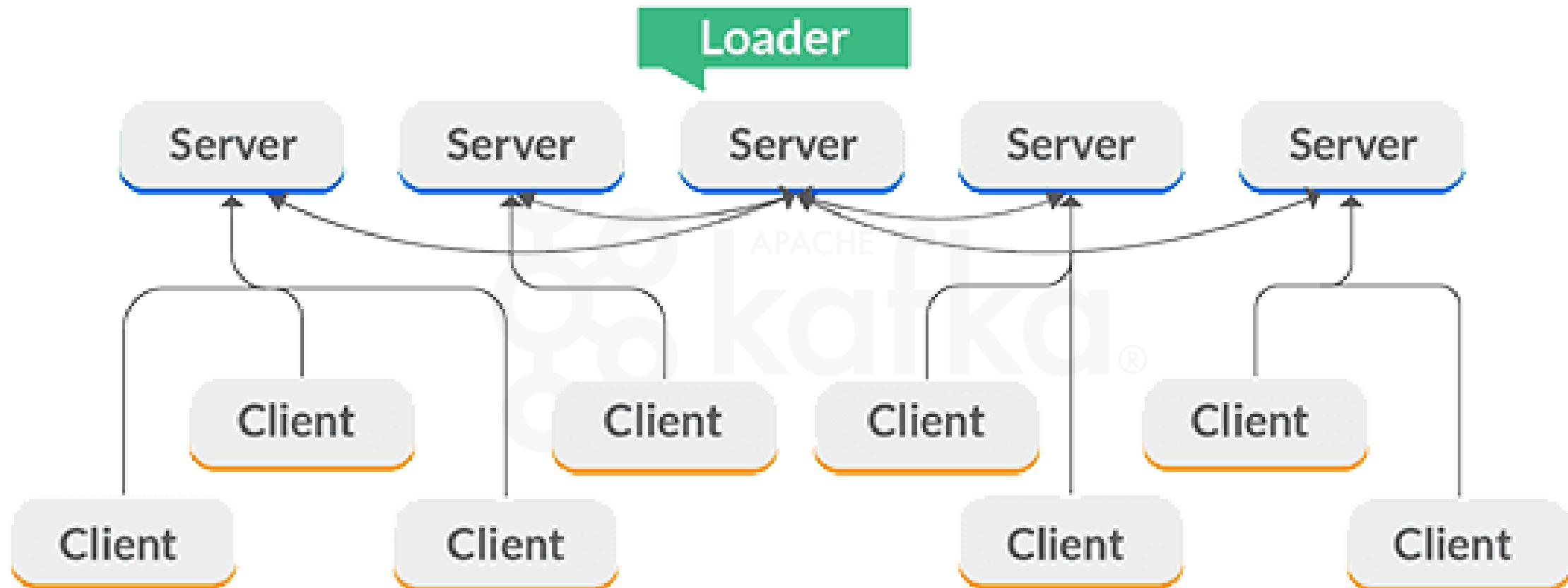


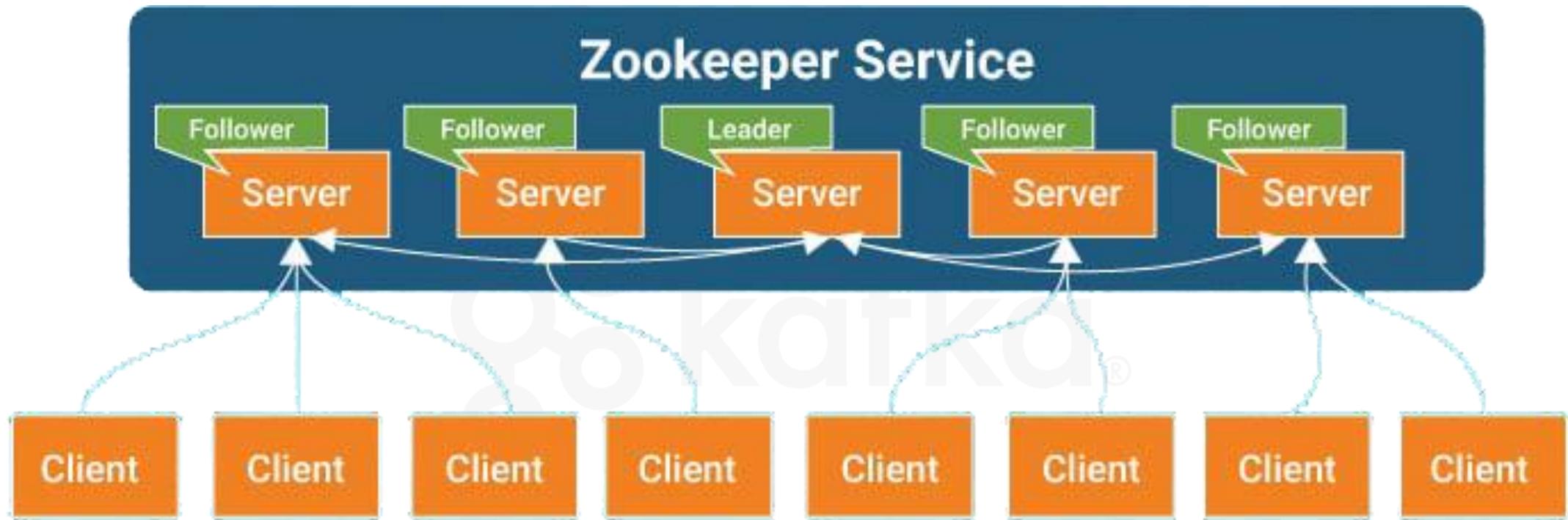
- «Apache ZooKeeper» è un progetto software della Apache Software Foundation
- E' stato pensato col fine di fornire:
 - un **servizio di configurazione distribuito** open source,
 - un **servizio di sincronizzazione per grandi sistemi distribuiti**
- Era un sotto-progetto di «Apache Hadoop» ora diventato indipendente.

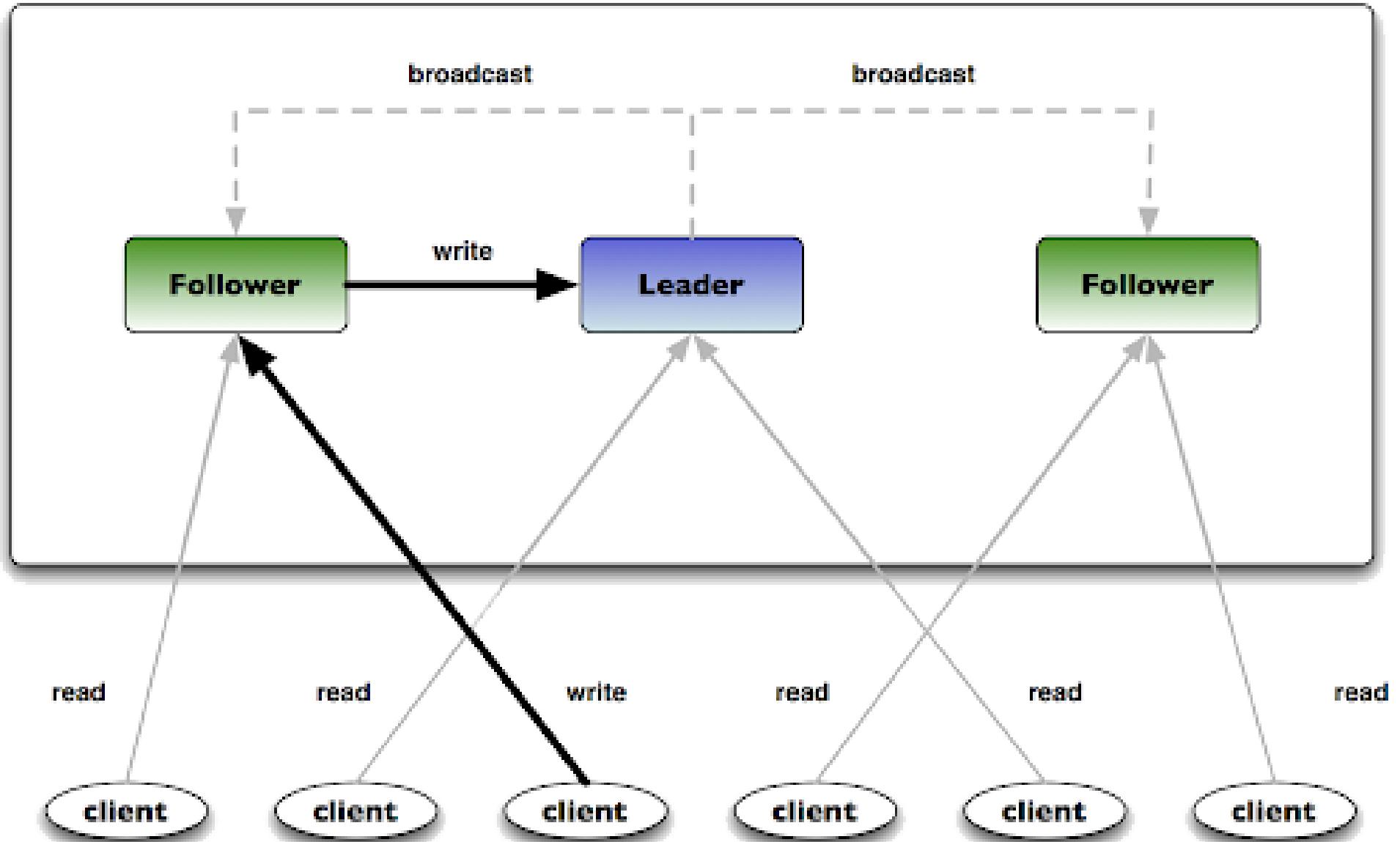


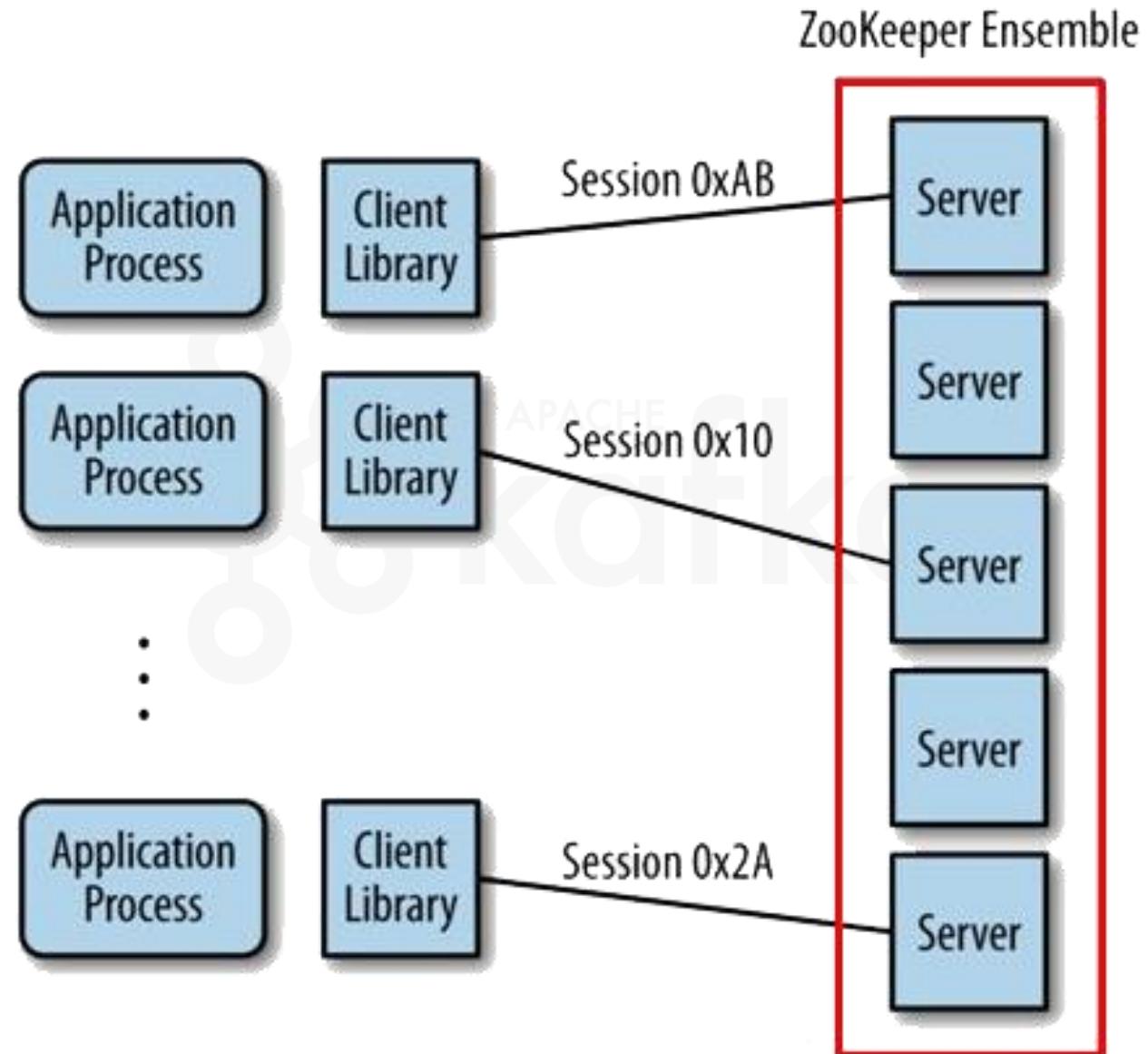
APACHE
ZooKeeperTM

- E' un servizio centralizzato per mantenere le informazioni di **configurazione**, di **naming**, fornendo sincronizzazione distribuita, e la fornitura di servizi di gruppo.
- In breve è un servizio di **coordinamento ad alto rendimento** per le applicazioni distribuite, come quelli gestiti in un cluster Hadoop.
- Permette ai processi distribuiti di coordinarsi attraverso una **struttura dati gerarchica condivisa** simile a un file system ma mantenuta in memoria, garantendo alte performance.

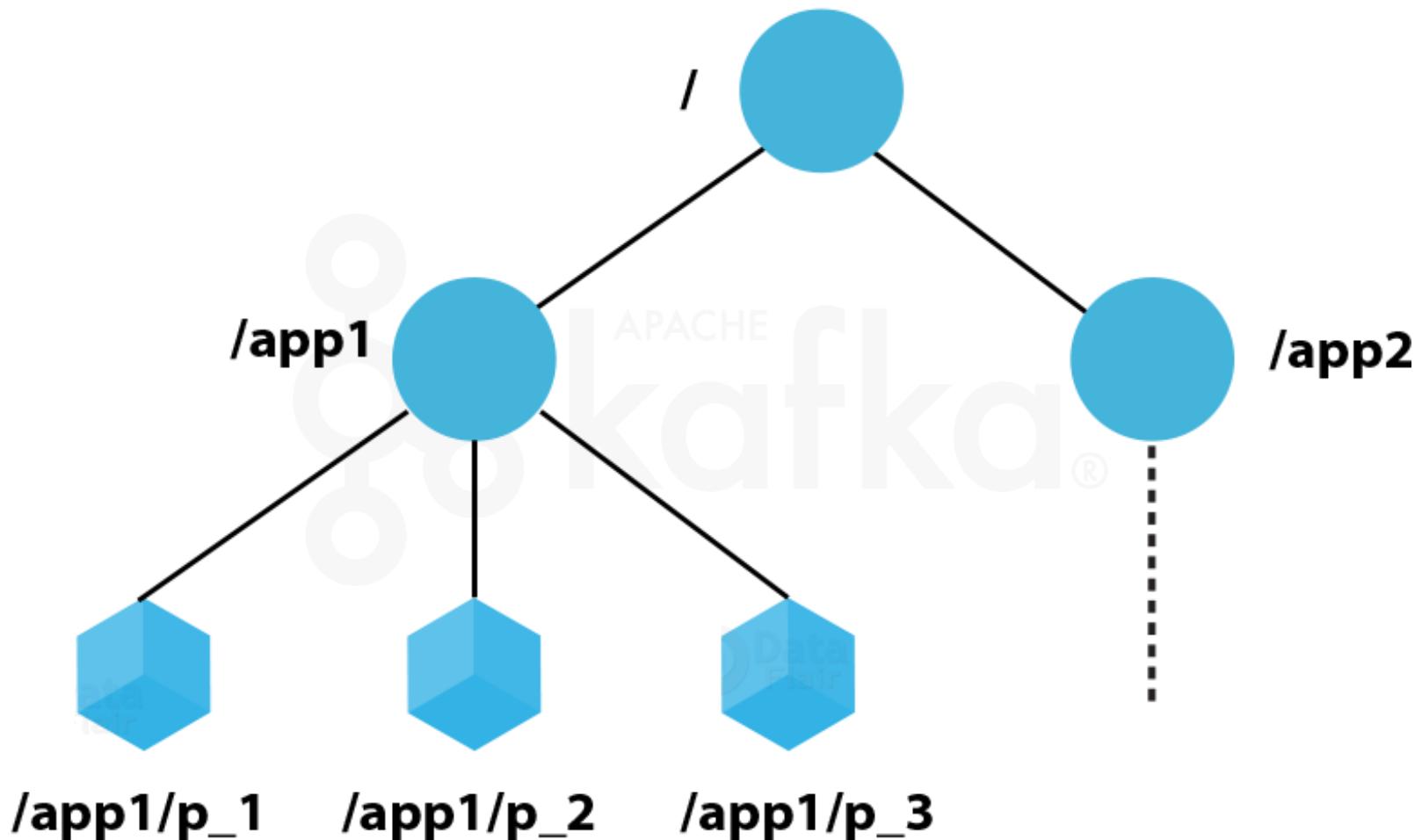






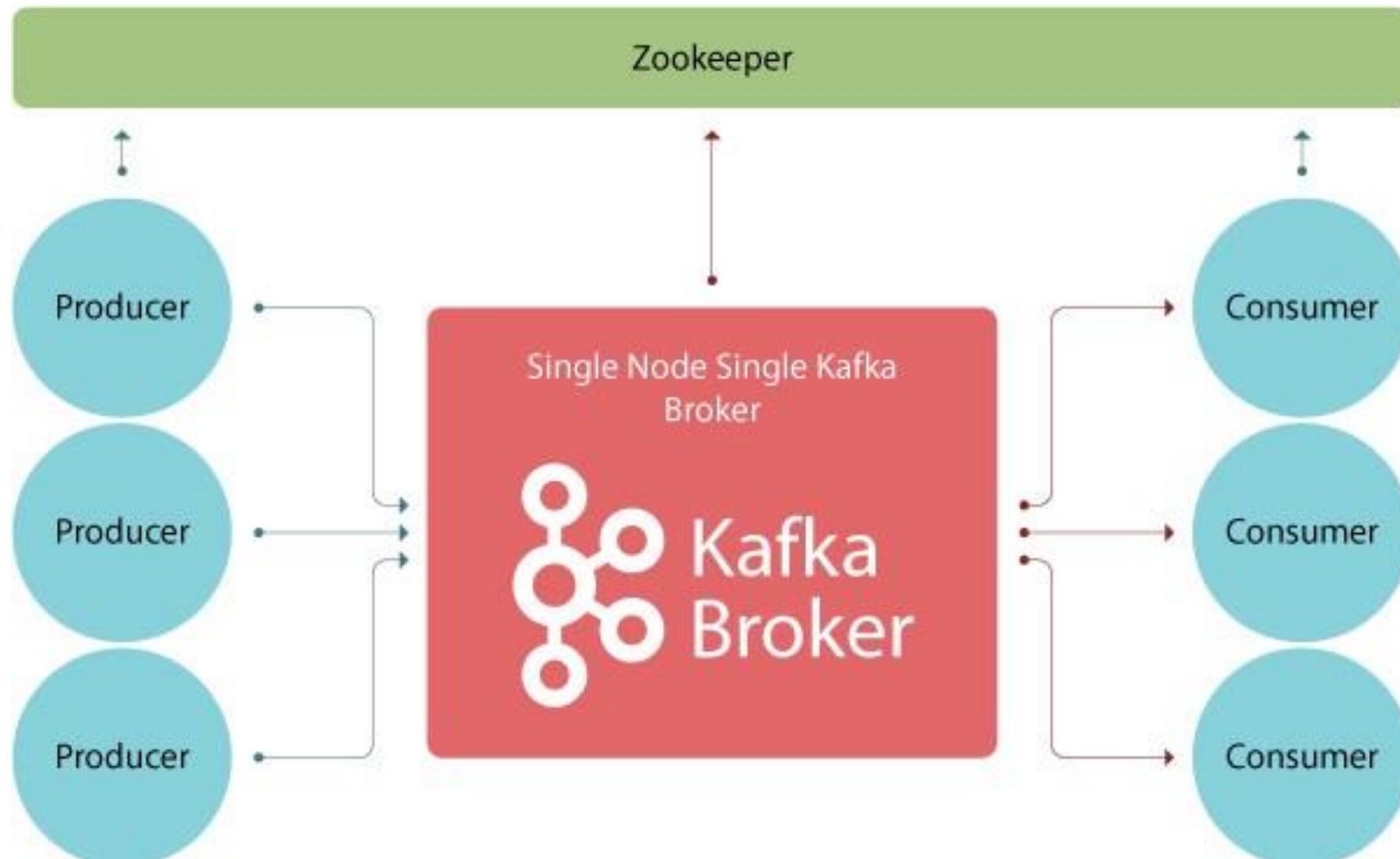


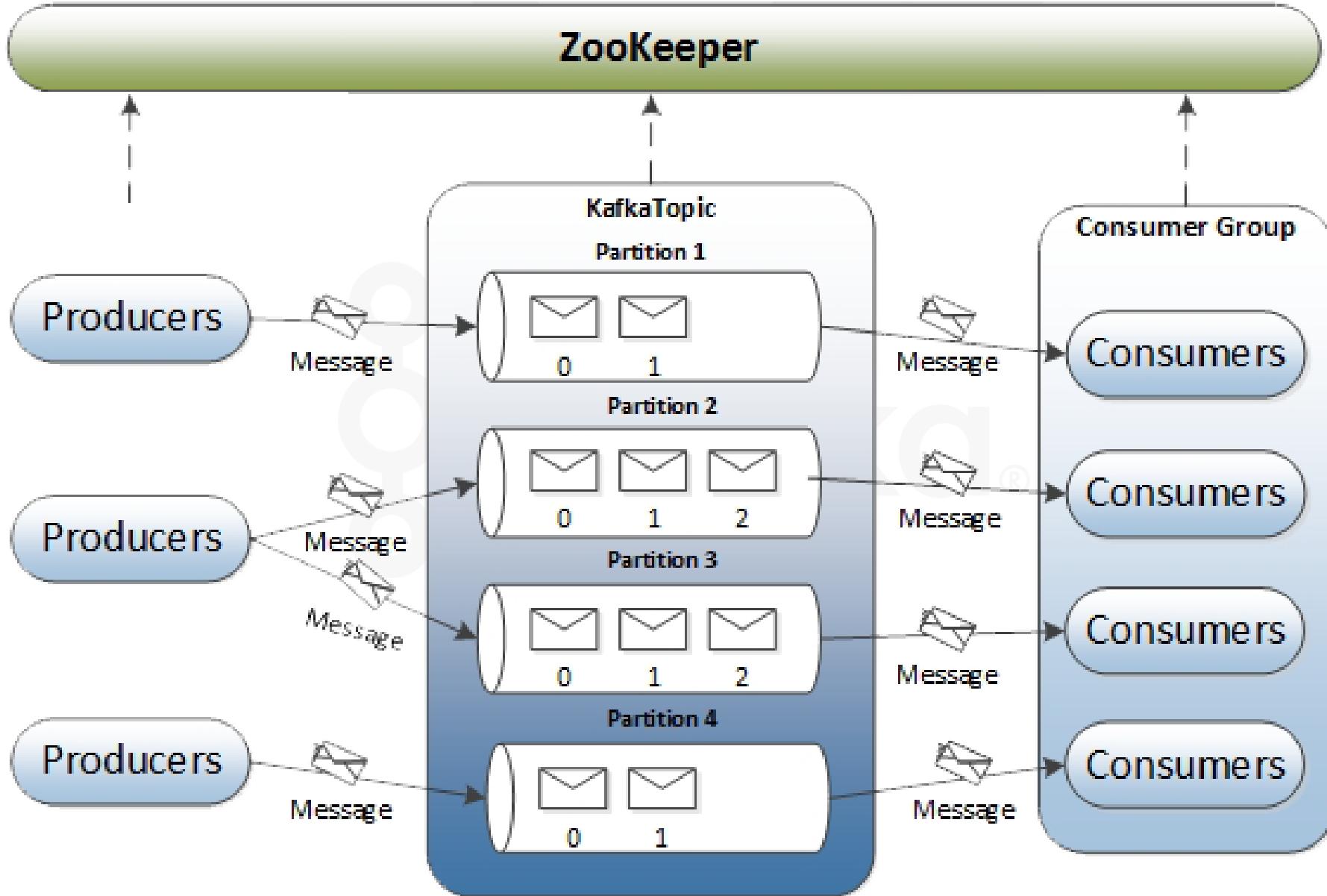
- E' un servizio che utilizza un insieme di server su cui vengono **replicati i dati mantenuti** all'interno, in modo da fornire **high availability** in caso di guasti.
- L'architettura di ZooKeeper supporta un'alta disponibilità attraverso servizi ridondanti.
- I client possono così richiedere un ulteriore «**ZooKeeper master**» se il primo fallisce nel rispondere.
- I **nodi** di ZooKeeper memorizzano i loro dati con uno spazio dei nomi gerarchico (come già accennato simile a un file system o, più in generale a una struttura dati ad albero).
- I **client** possono leggere e scrivere dai/ai nodi e in questa maniera hanno condiviso un servizio di configurazione.



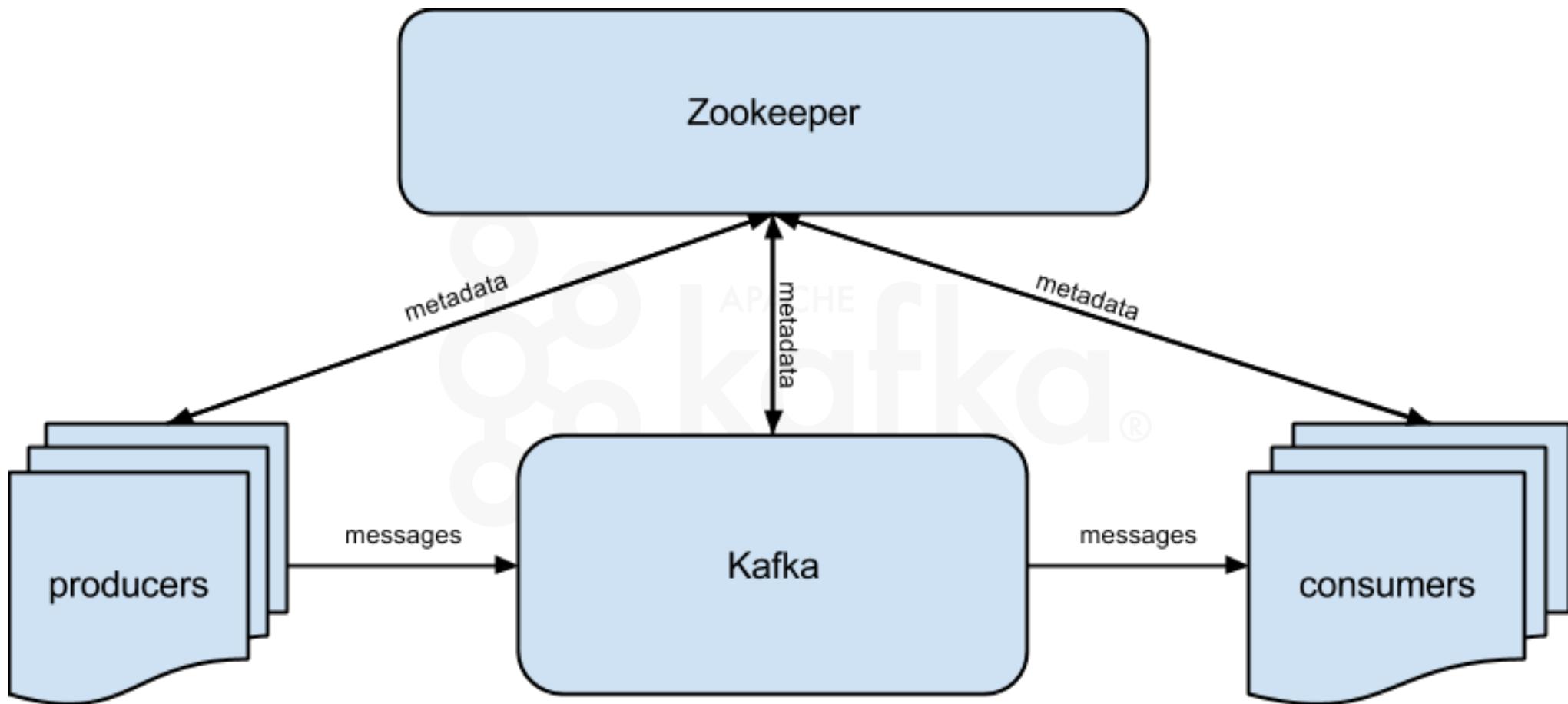
- E' uno strumento per la **gestione del cluster** che offre le seguenti garanzie di funzionamento:
 - **Consistenza sequenziale**, con gli aggiornamenti inviati da un client applicati nell'ordine in cui sono mandati;
 - **Atomicità**: gli aggiornamenti vanno a buon fine interamente oppure non sono affatto applicati;
 - Un **client** ottiene sempre la stessa **vista del servizio** indipendentemente dal server a cui si connette;
 - **Affidabilità**

- Fornisce un'**infrastruttura centralizzata** e una serie di **servizi** che consentono la sincronizzazione di oggetti comuni nel cluster.
- Tali oggetti comuni sono, per esempio, le **configurazioni**, che devono essere presenti su tutti i nodi e che devono rimanere sincronizzate.
- **Apache ZooKeeper, per quanto concerne l'utilizzo in un ecosistema Kafka, gestisce lo stato dei cluster Kafka e non solo ☺.**





Zookeeper & Kafka





- State
- Quotas
- Replicas
- Nodes and Topics Registry

- Offsets
- Registry

UNIT

Why Kafka ?

- Kafka è in grado di gestire senza problemi più producer, indipendentemente dal fatto che quei client utilizzino argomenti diversi o il medesimo.
- Questo rende il sistema ideale per **aggregare dati da molti sistemi frontend al fine di renderli coerenti.**
- Ad esempio: un sito che fornisce contenuti agli utenti tramite una serie di microservizi può avere un unico argomento per le visualizzazioni di pagina in cui tutti i servizi possono scrivere utilizzando un formato comune.

- Kafka è progettato per consentire a più consumer di leggere qualsiasi singolo flusso di messaggi senza interferire tra loro.
- Ciò è in contrasto con molti sistemi di accodamento in cui una volta che un messaggio viene utilizzato da un client, non è disponibile per nessun altro.
- I consumer multipli di Kafka **possono scegliere di operare come parte di un gruppo e condividere un flusso**, garantendo che l'intero gruppo elabori un determinato messaggio una sola volta.

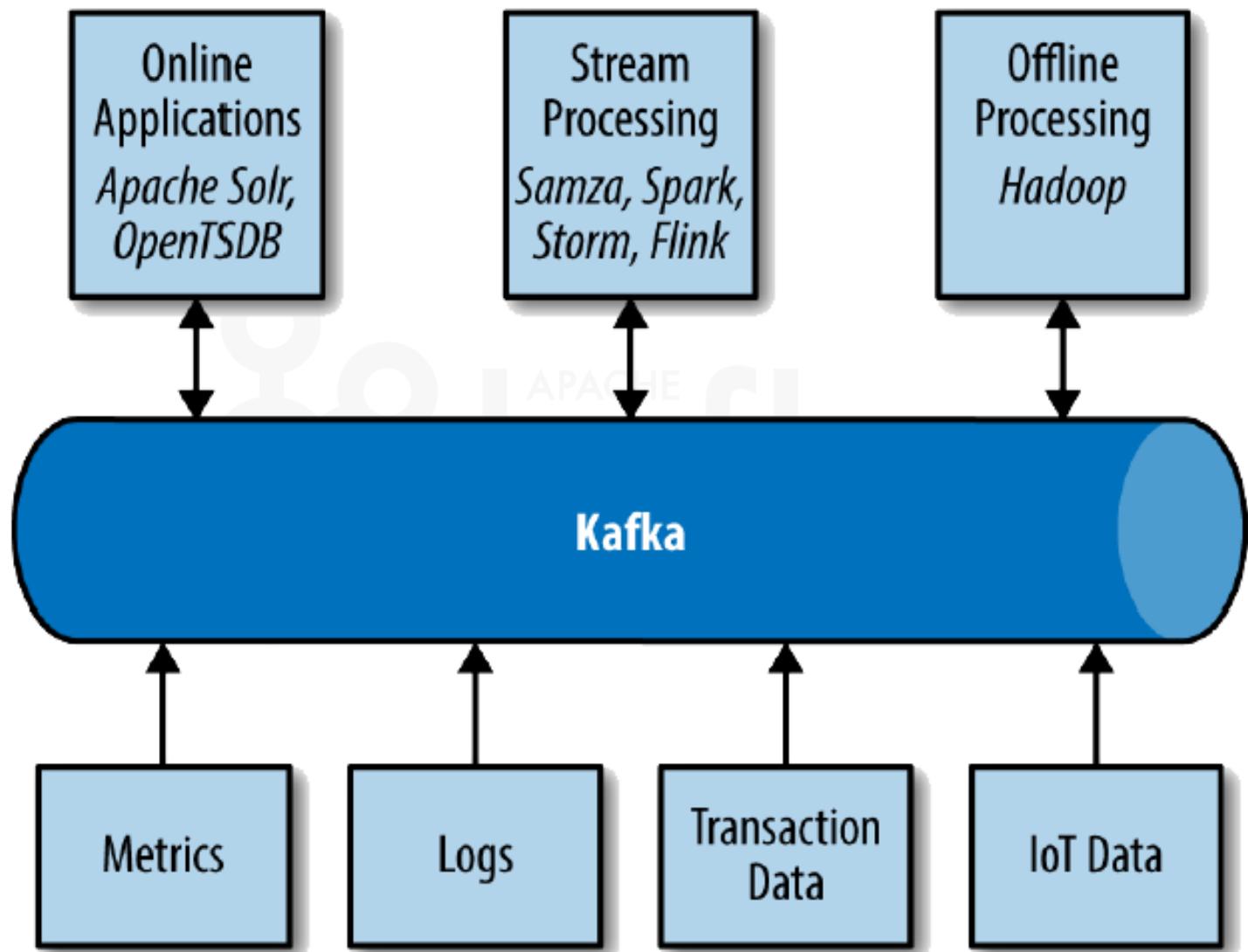
- Kafka non solo può gestire più consumer, ma anche la «**durable message retention**», il che significa che i consumer non devono sempre lavorare in tempo reale.
- I messaggi vengono committati su disco e verranno archiviati con regole di conservazione **configurabili**.
- Queste opzioni possono essere selezionate in base all'argomento, consentendo a diversi flussi di messaggi di avere diversi livelli di conservazione in base alle esigenze del consumer.
- «**durable retention**» significa che se un consumer rimane indietro, a causa della lentezza dell'elaborazione o di un'esplosione del traffico, non c'è pericolo di perdere dati.

- Significa anche che la manutenzione può essere eseguita sui consumer, **portando offline le applicazioni per un breve periodo di tempo**, senza preoccuparsi del backup dei messaggi sul producer o della perdita.
- I consumer possono quindi essere fermati e mentre i messaggi verranno **conservati** in Kafka.
- **Ciò consente di riavviare i consumer i quali così possono riprendere l'elaborazione dei messaggi da dove erano stati interrotti senza perdita di dati.**

- La flessibile **scalabilità** di Kafka semplifica la gestione di **qualsiasi quantità di dati**.
- Gli utenti possono iniziare con un **singolo broker** come POC (proof of concept), espandersi in un piccolo cluster di sviluppo di tre broker e passare alla produzione con un cluster più ampio di decine o addirittura centinaia di broker che cresce nel tempo man mano che i dati aumentano.
- Le espansioni possono essere eseguite mentre il cluster è online, senza alcun impatto sulla **disponibilità** del sistema nel suo insieme.
- Ciò significa anche che un cluster di più broker **può gestire il fallimento di un singolo broker e continuare a servire i clients**.
- I cluster che devono **tollerare più errori simultanei** possono essere configurati con fattori di replica più elevati.

- Tutte queste funzionalità si uniscono per rendere Apache Kafka un sistema di messaggistica di Pub/Sub **con prestazioni eccellenti a carico elevato.**
- Producer, consumer e broker possono essere ridimensionati per gestire facilmente flussi di messaggi molto grandi.
- Questo può essere fatto fornendo al contempo **una latenza del messaggio inferiore al secondo** (dalla produzione di un messaggio alla disponibilità per i consumer).

- Kafka fornisce il sistema di base per il «**circolo**» dei dati all'interno di infrastrutture sia semplici che complesse.
- **Trasporta** messaggi tra i vari membri dell'infrastruttura, fornendo **un'interfaccia coerente** per tutti i client.
- Se abbinati a un sistema per fornire schemi di messaggi, **producer e consumer possono essere tranquillamente disaccoppiati** per cui non sono più richieste connessioni dirette di alcun tipo.
- I componenti possono essere **aggiunti e rimossi** a seconda delle esigenze e dei casi d'uso; i producer non devono preoccuparsi di chi sta utilizzando i dati o del numero di applicazioni client.



UNIT

Altri Casi D'Uso

- In origine, il caso d'uso di Kafka è stato quello del **monitoraggio delle attività degli utenti**.
- Gli utenti di un sito Web interagiscono con le applicazioni di **frontend**, che generano messaggi relativi alle azioni intraprese dall'utente.
- Può trattarsi di **informazioni passive**, come visualizzazioni di pagina e tracciamento dei clic, oppure azioni più **complesse**, come le informazioni che un utente aggiunge al proprio profilo.
- I messaggi vengono **pubblicati** su uno o più argomenti, che vengono quindi utilizzati dalle applicazioni sul back-end.
- Queste applicazioni possono generare report, alimentare sistemi di apprendimento automatico, aggiornare i risultati della ricerca o eseguire altre operazioni necessarie per fornire un'esperienza utente completa.

- Kafka viene anche utilizzato per la **messaggistica**, in cui le applicazioni devono inviare notifiche (come le e-mail) agli utenti.
- Tali applicazioni possono produrre messaggi senza preoccuparsi della **formattazione** o del modo in cui i messaggi verranno effettivamente inviati.
- Una singola applicazione può quindi leggere tutti i messaggi da inviare e gestirli in modo coerente, tra cui:
 - Formattazione dei messaggi (noto anche come decorazione) utilizzando un «**look and feel**» comune
 - Raccolta di più messaggi in un'unica notifica da inviare
 - Applicazione delle preferenze di un utente sul come desidera ricevere messaggi

- Kafka è ideale per la raccolta di **metriche e log di applicazioni e sistemi**. Questo è un caso d'uso in cui spicca la possibilità di avere più applicazioni che producono lo stesso tipo di messaggio; le applicazioni pubblicano metriche su base regolare su un argomento Kafka e tali metriche possono essere utilizzate dai sistemi per il **monitoraggio** e gli **avvisi**.
- Possono anche essere utilizzati in un sistema offline come Hadoop per eseguire analisi a più lungo termine, come le **proiezioni di crescita**.
- I messaggi di registro possono essere pubblicati allo stesso modo e possono essere indirizzati a sistemi di ricerca dei registri dedicati come **Elasticsearch** o applicazioni di analisi della **sicurezza**.
- Un altro vantaggio di Kafka è che quando il sistema di destinazione deve cambiare (ad esempio, è tempo di aggiornare il sistema di archiviazione dei log), non è necessario modificare le applicazioni front-end o gli strumenti di aggregazione.

- Poiché Kafka poggia sul concetto di un «**commit log**», le modifiche al database possono essere pubblicate su Kafka e le applicazioni possono facilmente monitorare questo flusso per ricevere aggiornamenti in **tempo reale** mentre si verificano.
- Il «**changelog**» può anche essere utilizzato per replicare gli aggiornamenti del database su un sistema remoto o per consolidare le modifiche da più applicazioni in una singola vista del database.
- La «**durable retention**» è utile qui per fornire un **buffer** per «changelog», il che significa che può essere riprodotto in caso di guasto delle applicazioni «consumatrici».
- In alternativa, è possibile utilizzare argomenti «**log-compacted**» per fornire una retention più lunga mantenendo una sola modifica per chiave.

- Un'altra area di interesse è l'**elaborazione di flussi dati** (grandi o piccoli che siano).
- Mentre quasi tutto l'utilizzo di Kafka può essere considerato come elaborazione in streaming, quest'ultimo viene generalmente utilizzato per indicare applicazioni che **forniscono funzionalità simili per mappare/ridurre l'elaborazione in Hadoop** (di solito si basa sull'aggregazione di dati per un lungo periodo di tempo, sia ore che giorni).
- L'elaborazione del flusso opera sui dati in **tempo reale** con la stessa rapidità con cui vengono prodotti i messaggi.
- I framework di streaming consentono agli utenti di scrivere **piccole applicazioni per operare su messaggi Kafka**, eseguendo attività come il conteggio delle metriche, il partizionamento dei messaggi per un'elaborazione efficiente da parte di altre applicazioni o la trasformazione di messaggi utilizzando dati provenienti da più origini.