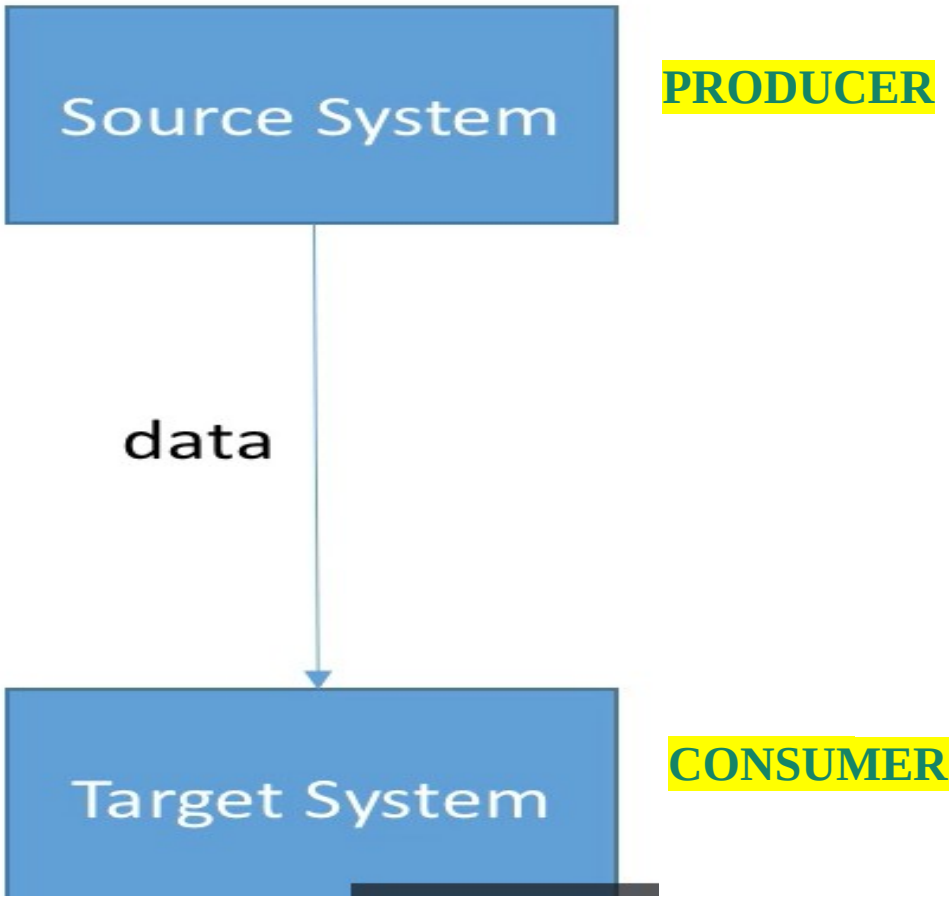
Apache Kafka Beginner

1.0 Introduzione

Supponiamo di avere il seguente caso d’uso, ovvero 2 sistemi che interagiscono:

- **sistema sorgente**
  (che ad esempio genera/produce dei dati e li scambia con un altro e quindi svolge il ruolo di Producer di dati)

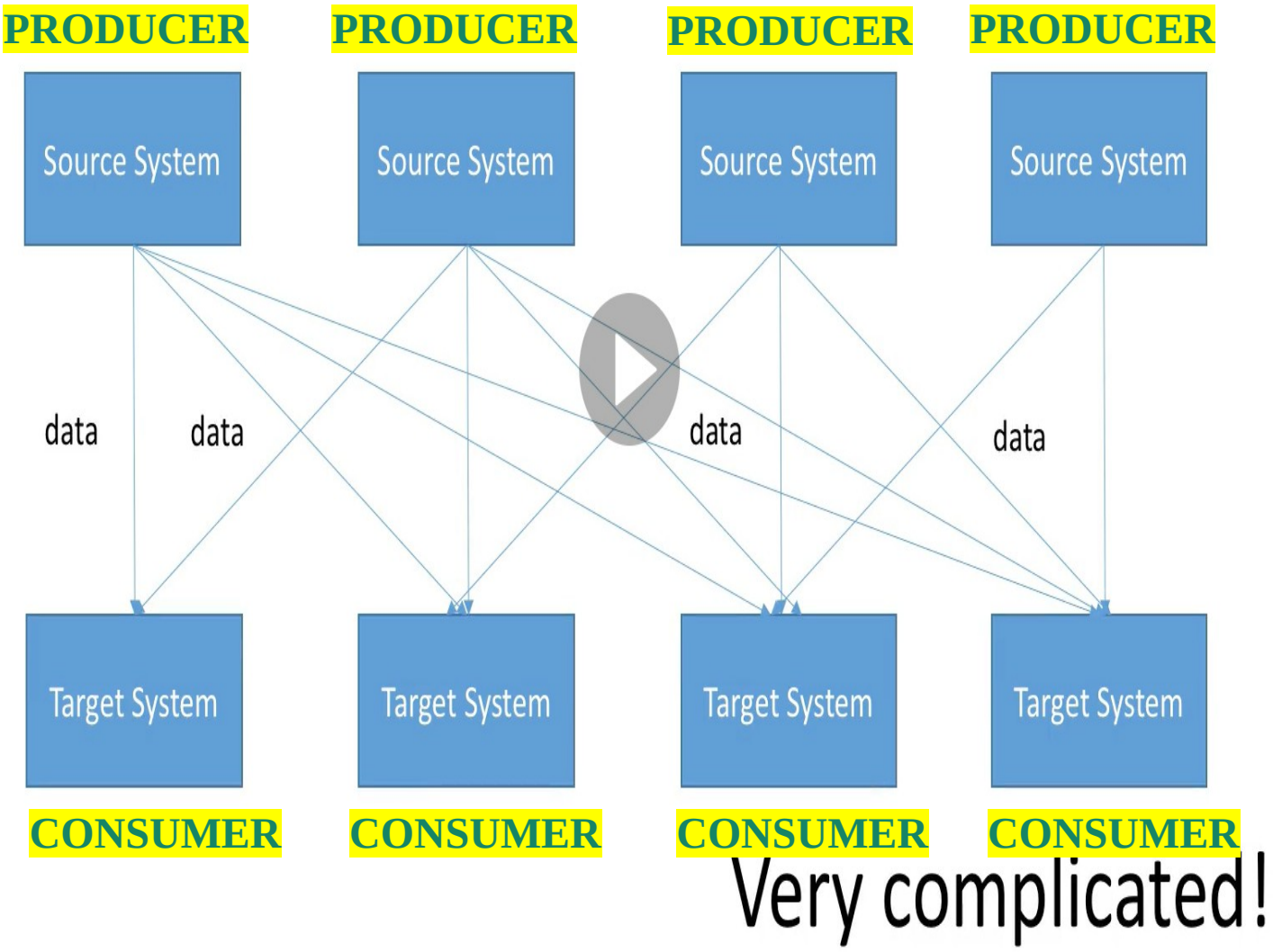
- **sistema destinazione**
  (che riceve i dati dal primo e li elabora in qualche modo e quindi svolge il ruolo di Consumer di dati)



Questo schema sembra ed è abbastanza semplice.

Ma vediamo quello che succede nella vita reale...

Supponiamo che gli affari della nostra azienda crescono e abbiamo quindi bisogno di avere maggiori risorse che generano dei dati e che li elaborano.
Trovandoci quindi in uno schema simile a questo:



In questo schema possiamo notare:

- Diversi sistemi sorgenti

- Diversi sistemi di destinazione

oggetto fondamentale è che tutti quanti i sistemi devono poter scambiare dati tra loro.

Possiamo subito notare che questo schema di scambio dei dati è molto più complesso rispetto a quello precedente dove avevamo solo 2 sistemi.

Le cose potrebbero diventare ancora più complicate se avessimo in generale:

- N sistemi sorgenti

- M sistemi di destinazione

a quel punto per fare in modo che tutti i sorgenti comunicano con tutte le destinazione dovremmo avere N*M integrazioni, dove per singola integrazione intendiamo il singolo collegamento che esiste tra sorgente e destinazione e che permette loro di scambiare dati.

Inoltre per ogni integrazione si possono avere difficoltà diverse che possono riguardare:

1. il protocollo da usare per la comunicazione
(TCP, HTTP, REST, FTP, ...)

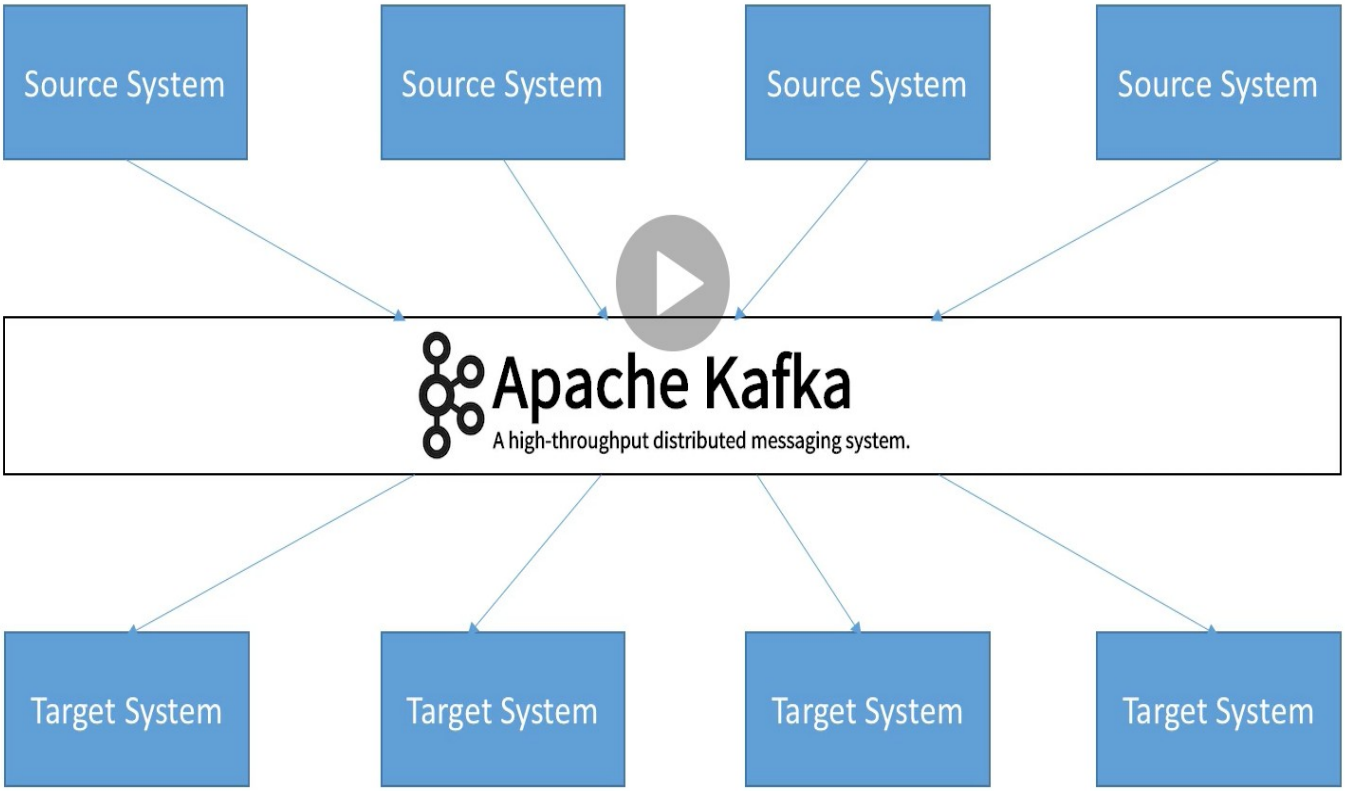
2. il formato dei dati di scambio
(JSON, XML, CSV, Binary, ...)

3. Lo schema dei dati (db) e la loro evoluzione futura

Inoltre ancora, ogni volta che si integra un sistema sorgente con un sistema destinazione, ci sarà un incremento del carico delle connessioni in quanto stiamo aggiungendo un altro flusso.

Abbiamo quindi capito che integrare diversi sistemi non è banale e che in fase di integrazione possono sorgere diverse problematiche come quelle accennate sopra.

Occore quindi trovare una soluzioni a tale problematiche, ed ecco qui che entra in gioco Apache Kafka.



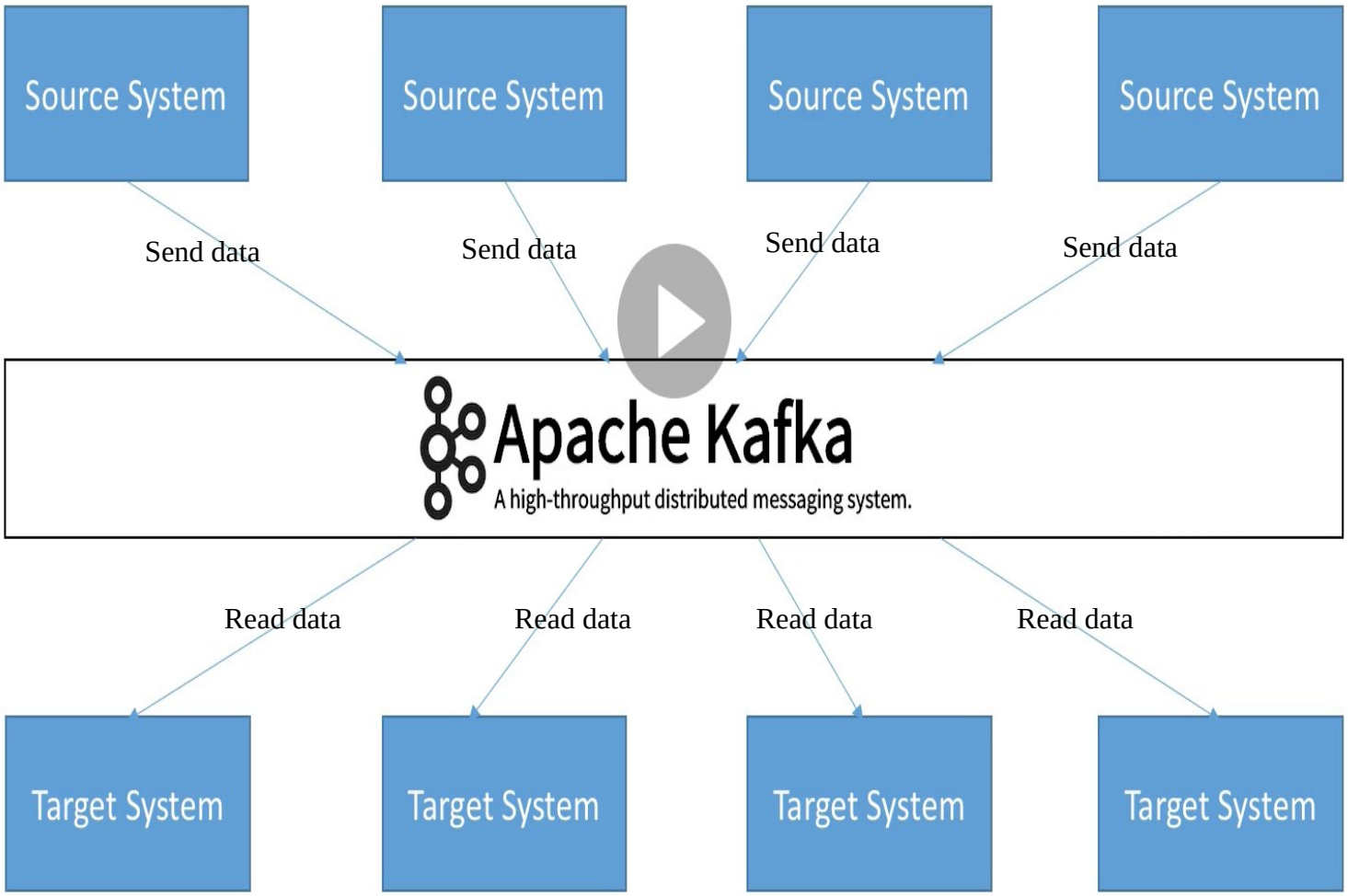
Grazie ad Apache Kafka possiamo:

→ **DISACCOPIARE** i flussi dei dati dai sistemi

Questo significa che un sistema sorgente non deve conoscere un sistema di destinazione per poter scambiare dei dati e quindi sono 2 sistemi indipendenti l'uno dall'altro.
Questo non era assolutamente vero nello schema precedente, infatti la freccia che partiva dalla sorgente e arriva alla destinazione rappresenta bene l'accoppiamento tra i 2, cioè i 2 sistemi per scambiare dati devono conoscersi.

In questo nuovo scenario Apache Kafka si posiziona nel mezzo e fa da ponte tra i sistemi sorgenti e quelli di destinazione.
Quindi un sistema sorgente non invierà più dei dati al diretto interessato sistema di destinazione, ma lo invierà a Kafka che lo memorizzerà in qualche modo.
D'altra parte i sistemi di destinazione prenderanno i dati di interesse prodotti dai sistemi sorgenti da Kafka.

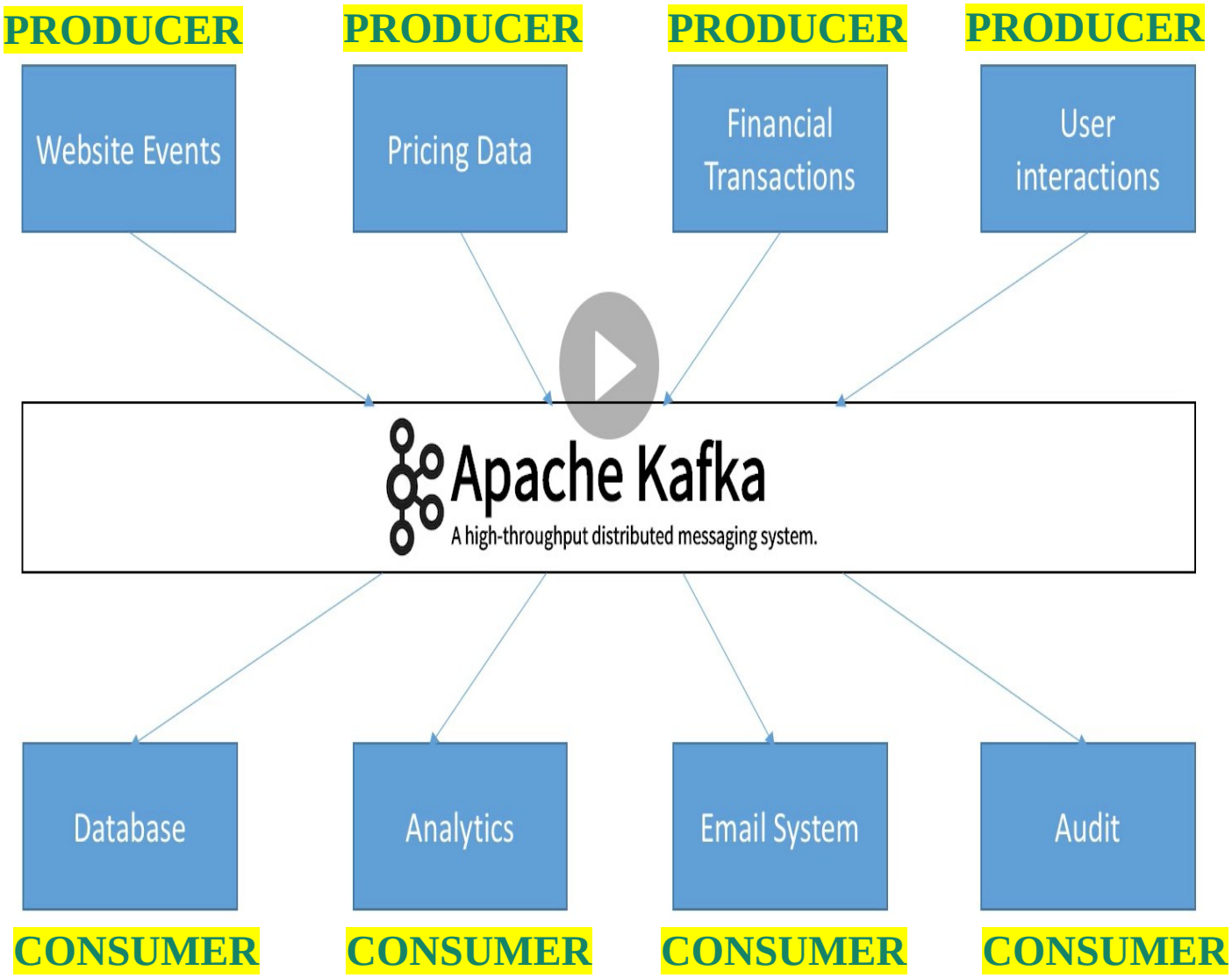
Sistemi sorgenti non sono a conoscenza dei sistemi di destinazione, loro solo producono dati e li inviano a Kafka,
Per loro il target dei dati è Kafka e non altri sistemi



Sistemi destinazione non sono a conoscenza dei sistemi sorgente, loro solo leggono dati da kafka,
per loro la fonte dati è kafka e non altri sistemi che producono dati.

In questo modo è possibile sostituire un qualsiasi sistema di sorgente o destinazione senza avere nessun impatto in quanto i sistemi sono tutti totalmente disaccoppiati, grazie a Kafka.

Schema reale di sistemi sorgente e destinazione che scambiano dati tra loro:



Ma perché usare Kafka come soluzione di integrazione?
Perchè ha i seguenti vantaggi:

1. Architettura distribuita e fault-tollerance (tolleranza ai guasti)

2. Scalabilità orizzontale
   → possiamo scalare più di 100 broker che lavorano in parallelo
   → possiamo scalare milioni di messaggi per sec.
   E questo è dimostrato da società che usano Kafka come:
   Linkedin, Netflix, Airbnb, Uber, ...

3. Estremamente performante con una latenza di meno di 10ms per questo è considerato un sistema di trasmissione di mssaggi real-time

4. Usato da moltissime aziende e quindi è un prodotto affermato

Casi d’uso di apache Kafka sono i segeunti:

1. Sistema di messaggistica
2. Tracciamento delle attività
3. Raccolta di metriche da molti luoghi diversi per dispositivi IoT
4. Logging di applicazioni
5. Stream processing (con Kafka stream API o SPARK)
6. Disaccoppiamento di sistemi dipendenti

.....
.....

Storie d’uso reali di Kafka:

1. Netflix usa Kafka per applicare consigli in tempo reale mentre stai guardando i suoi programmi TV.
   Ecco spiegato il motivo per cui, quando lasciamo una serie o un film riceviamo subito nuove raccomandazioni sul prossimo da vedere.

2. Uber usa Kafka per raccogliere in tempo reale dati su: utenti, taxi e percorsi, in modo da aggiornare in tempo reale i prezzi.

3. Linkedin usa Kafka per raccogliere dati sulle interazioni degli utenti in modo da consigliare in tempo reale nuove interazioni

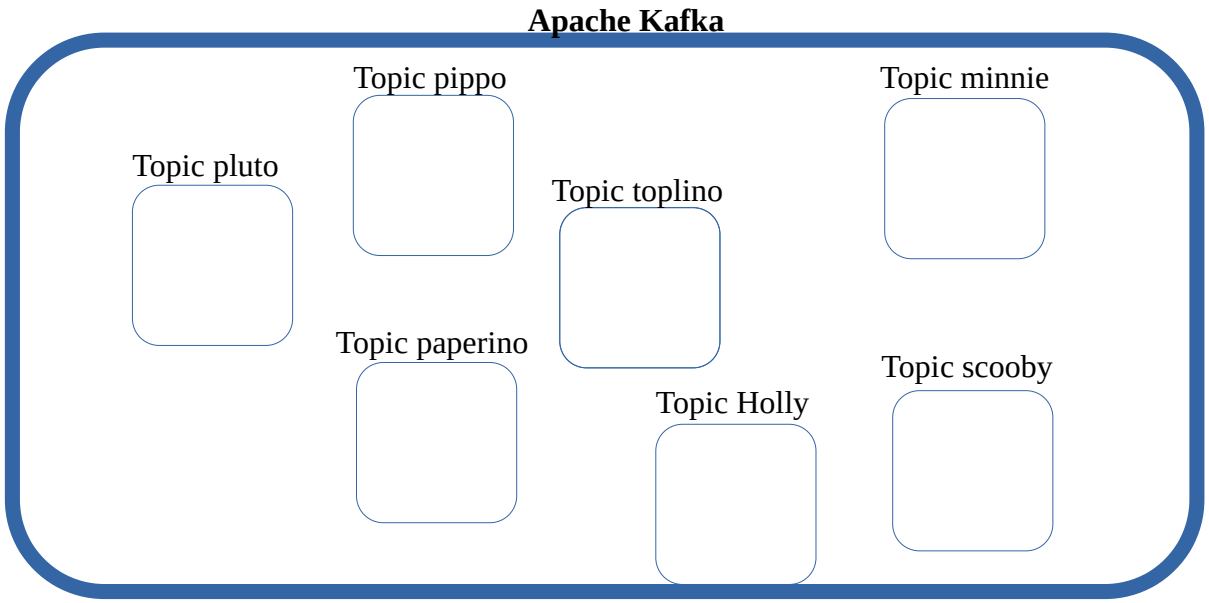
2.0 Teoria su Kafka

In questo capitolo parleremo degli argomenti core di apache kafka che sono sotanzialmente 3:

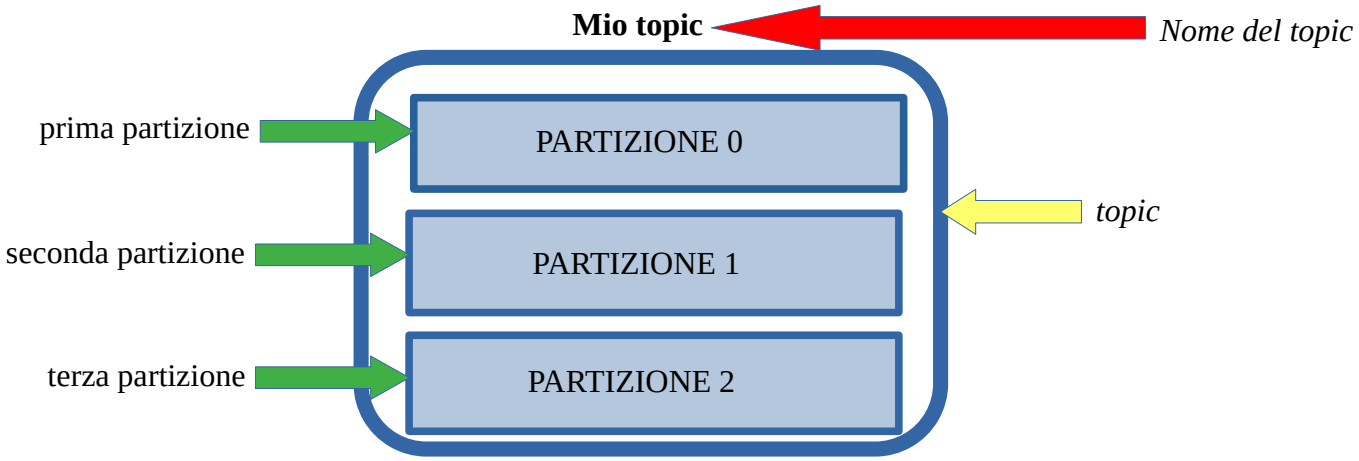
i. ***Topic***
ii. ***Partizione***
iii. ***Offset***

In apache Kafka, il topic è alla base di tutto.
Ma Cosa è un Topic ?
Un topic è uno specifico stream (flusso) di dati/messaggi.

Nella pratica un topic può essere considerato molto simile ad una tabella di un database.
Così come le tabelle di un database hanno un nome, lo stesso vale per i topic e così come per i database possiamo creare diverse tabelle, lo stesso vale per i topic.
Quindi in kafka possiamo avere un insieme di topic e ciascuno viene identificato dal suo nome.



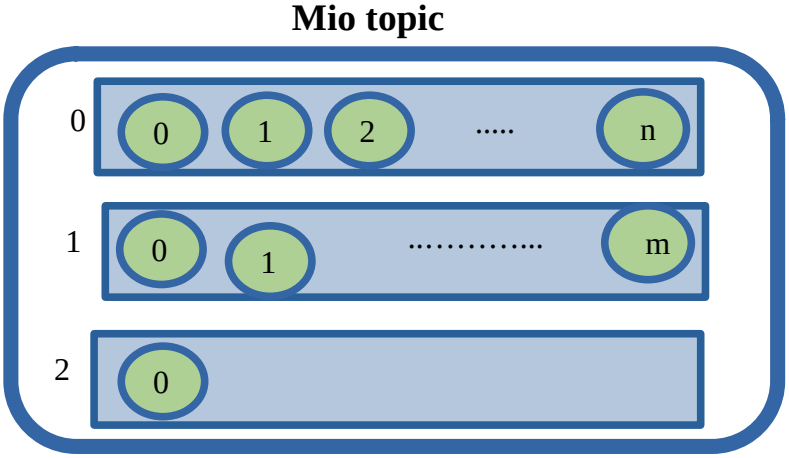
Come per i database le tabelle sono divise in più record, in kafka un topic è diviso in più partizioni.
Di seguito abbiamo una raffigurazione di un topic con 3 partizioni, ma in generale un topic può avere n partizioni.



L'immagine mostra chiaramente come un topic è diviso in più partizioni (nel nostro esempio 3).

NOTA BENE) Le partizioni sono numerate e partono sempre dal numero 0.
Quindi se in generale abbiamo un topic con all'interno n partizioni, la prima avrà sempre numero 0 e l'ultima sempre n-1.

Ogni partizione contiene al suo interno un numero variabile di messaggi e ogni messaggio è identificato da un id che è un numero incrementale che viene molto comunemente chiamato offset.
I messaggi situati all'interno di una specifica partizione non sono disposti in modo casuale ma sono ordinati.



NOTA BENE 1) All’interno di ciascuna partizione, i messaggi sono ordinati nel senso che verranno depositati/scritti nella partizione in ordine crescente di id, ovvero prima arriva nella partizione messaggio con id=1, poi arriva messaggio con id=2, poi arriva messaggio con id=3, e così via…
Non capiterà mai in una partizione che arrivi ad esempio il messaggio con id=4 e dopo quello con id=3 in quanto kafka ci garantisce che i messaggi si depositeranno/scritti in ordine crescente di id

NOTA BENE 2) Ogni partizione può avere un numero di messaggi che è “potenzialmente infinito”

NOTA BENE 3) Ogni partizione può avere un numero di messaggi diverso dalle altre partizioni

NOTA BENE 4) l’offset o id serve per individuare un particolare messaggio all’interno di una specifica partizione.

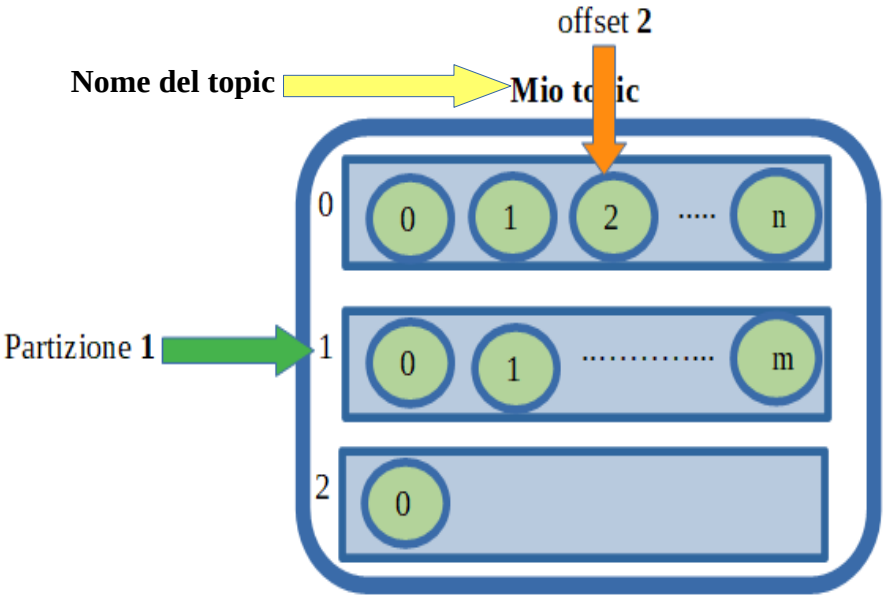
NOTA BENE 5) l’offset ha significato solo per una specifica partizione
Questo implica che partizione 1 offset 2 individua un messaggio completamente diverso da partizione 2 offset 2

Quindi se io volessi individuare un qualsiasi messaggio all’interno di una partizione di un particolare topi mi basterebbero solamente 3 informazioni:

→ nome del topic
→ il numero della partizione
→ l’offset (o id del messaggio)

Ad esempio il 3° messaggio della 2° partizione del topic “mio topic” e identificato dalla terna (“mio topic”, 1, 2) dove:

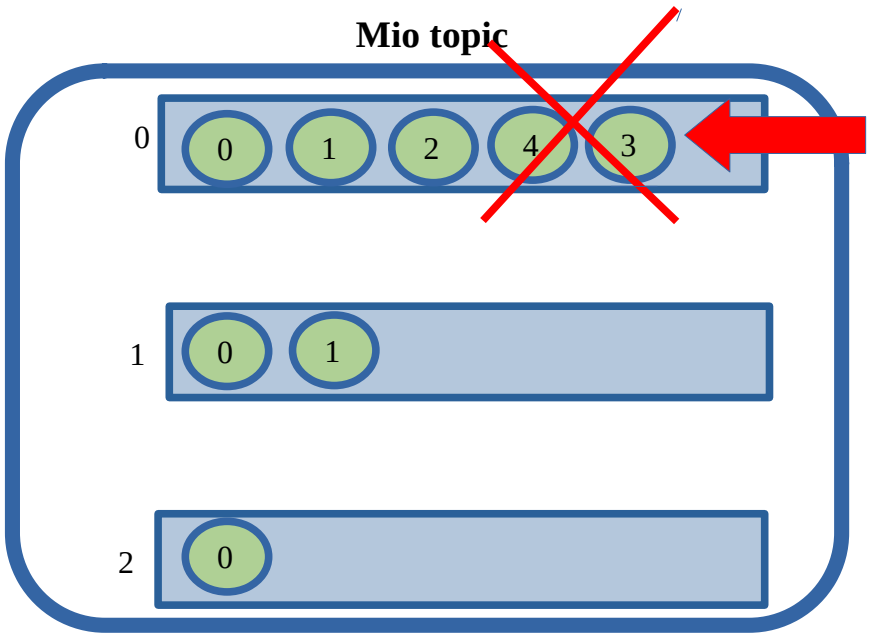
- “mio topic” individua un particoalare topic
- 1 individua la partizione
- 2 individua il messaggio all’interno della partizione 1, che poi coincide anche con lo scostamento che ho fatto all’interno della partizione 1 per trovare il messaggio e infatti offset vuol dire proprio scostamento.



Abbiamo detto prima che i messaggi all'interno di una partizione sono ordinati.
Ma in che senso sono ordinati?
Nel senso che kafka ci garantisce che il messaggio con id=0 arriverà nella partizione prima del messaggio con id=1, e che il messaggio con id=1 arriverà nella partizione prima del messaggio con id=2, e che il messaggio con id=2 arriverà nella partizione prima del messaggio con id=3, ecc...
Così facendo i messaggi si depositeranno nella partizione rispettando sempre l'ordinamento crescente per id.

Abbiamo quindi capito che l'ordine di arrivo dei messaggi nella singola partizione è garantito da kafka.
Tuttavia l'ordine di arrivo dei messaggi attraverso le partizioni di un topic non è garantito da kafka. Questo significa che: potrebbe ad esempio succedere di ricevere prima il messaggio con id=5 nella partizione 1 e dopo ricevere il messaggio con id=4 nella partizione 2.

RICORDA L'ORDINE DI ARRIVO DEI MESSAGGI E' GARANTITO DA KAFKA SOLO A LIVELLO DI PARTIZIONE E NON CROSS PARTIZIONE.

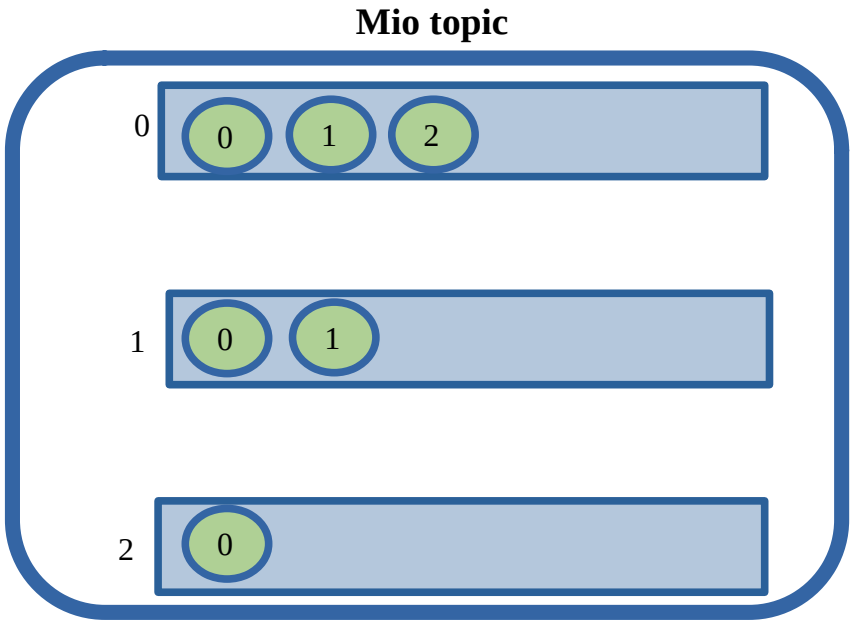


Questa situazione in una partizione non potrà mai verificarsi: ovvero che venga depositato nella partizione prima il messaggio con id=4 e poi quello con id=3

Vedendo l'immagine nella partizione 0 possiamo notare che il messaggio con id=4 è arrivato nella partizione prima del messaggio

con id=3 non rispettando quindi l'ordine di arrivo dei messaggi.
Ecco questo scenario non potrà mai capitare all'interno di una partizione in quanto Kafka ci garantisce che i messaggi arriveranno nella specifica partizione in modo ordinato.
Ovvero che il messaggio con id=3 arrivi prima del messaggio con id=4.

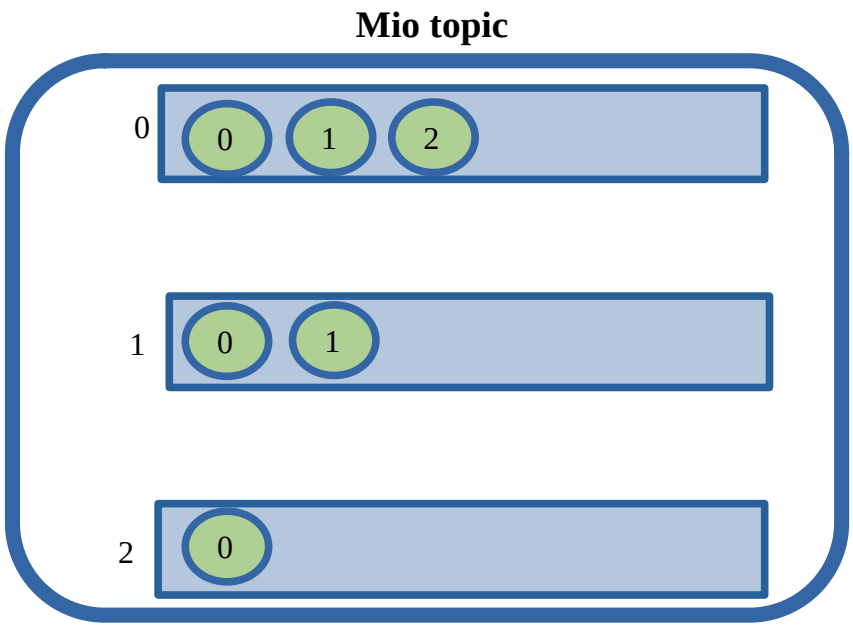
Diversa è invece la situazione della figura seguente:



Potrebbe invece verificarsi il caso che:
all'istante t_0 arrivi il messaggio con id=2 nella partizione 0 e successivamente all'istante t_1 arrivi il messaggio con id=0 nella partizione 2.
Questo è lecito perché kafka non garantisce invece l'arrivo ordinato dei messaggi cross partizioni.

I dati all’interno di Kafka non sono memorizzati in eterno ma sono memorizzati solo per un certo periodo di tempo.
Questo vuol dire che ad un certo punto questi dati verranno cancellati.
Per default Kafka memorizza i dati per un periodo che equivale ad una settimana dopo di che verranno cancellati.
Tuttavia gli id (offset) dei messaggi all’interno di una partizione non verranno ripristinati a 0 ma continueranno ad incrementare.

Altra cosa molto importante è che una volta che un dato/messaggio viene scritto su una partizione, questo è immutabile, che vuol dire che non può più essere modificabile.



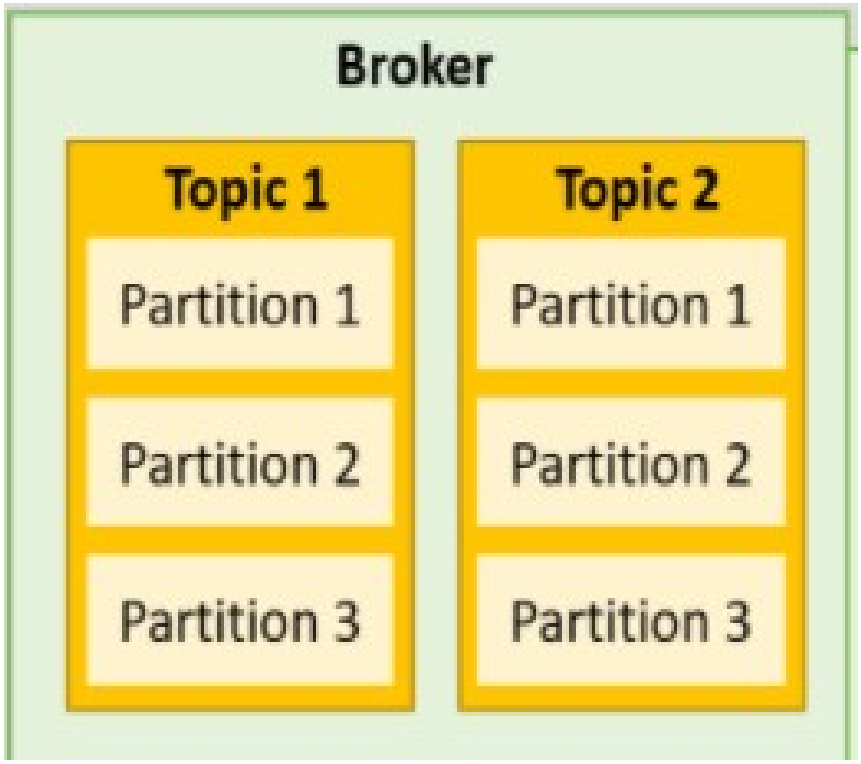
Ogni messaggio/dato una volta scritto sulla partizione di un topic, non può più essere modificato

Ma con quale logica kafka decide se un messaggio è destinato ad andare in una partizione piuttosto che in un’altra?
Se il messaggio non possiede una chiave, questo verrà memorizzato in una partizione casuale che verrà scelto secondo l’algoritmo Round-Robin
Round-Robin vuol dire che se abbiamo 5 partizioni, i messaggi che arrivano vengono memorizzati una volta nella partizione 0, poi nella partizione 1, poi nella partizione 2, poi nella partizione 3, poi nella partizione 4, poi nella partizione 5, e successivamente si comincia a depositare/scrivere il messaggio nella partizione 0, poi nella partizione 1 e così via.

3.0 Brokers e topics

Abbiamo parlato prima dei topic e abbiamo visto come sono strutturati

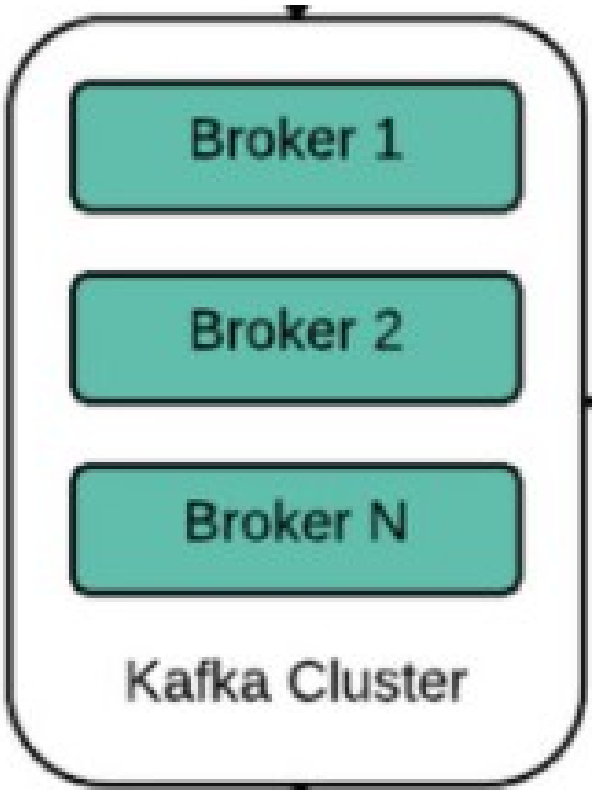
Ma una domanda che ci sorge spontanea adesso è:
Chi è che si occupa, contiene, gestisce i topics?
La risposta è i Brokers, anche conosciuti come Kafka server o nodi Kafka.



Nell'immagine sopra è raffigurato un broker con all'interno due topic (topic 1, topic 2) con 3 partizioni ciascuno, ma in generale un broker può contenere n topic.

La cosa importante da capire è che un broker è un'applicazione server deployata in una macchina fisica che permette di gestire 1 oppure n topics.

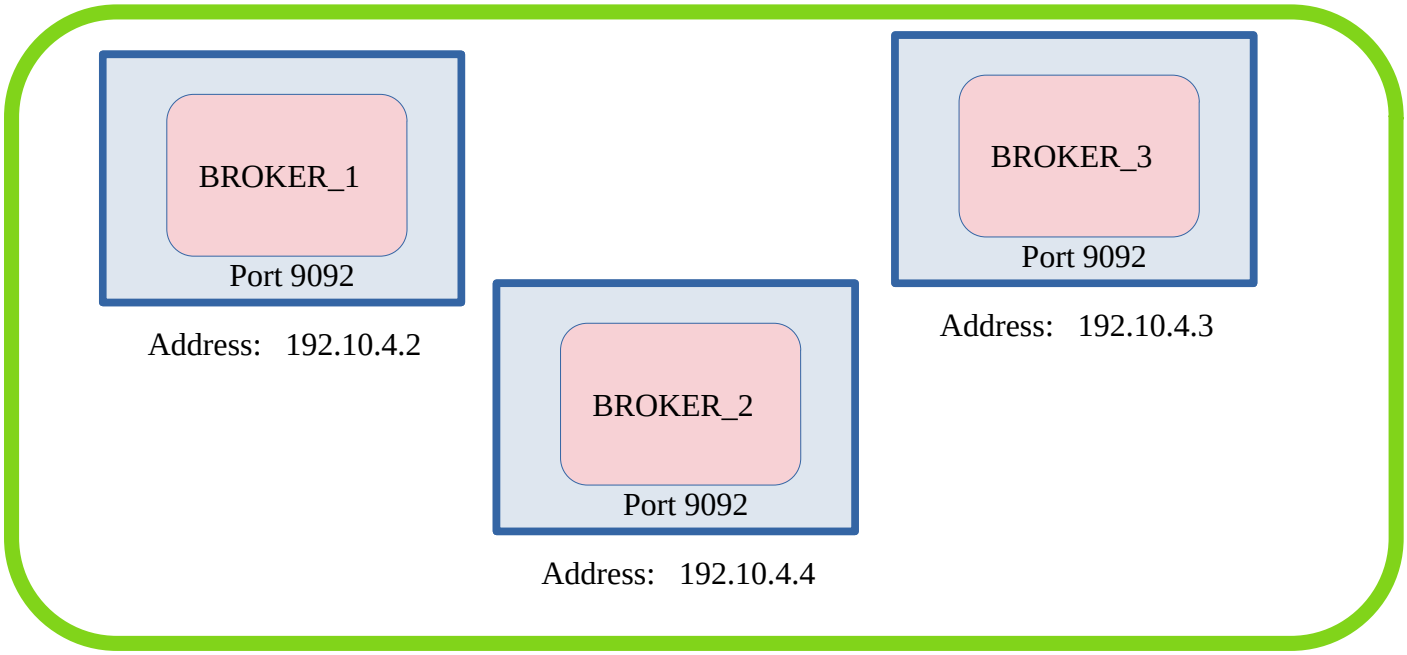
Nella realtà di solito non abbiamo solamente un broker ma abbiamo diversi brokers e quindi in questi casi si parla di cluster di broker (o cluster di nodi kafka).



Nell’immagine è raffigurato un Kafka Cluster con N Broker

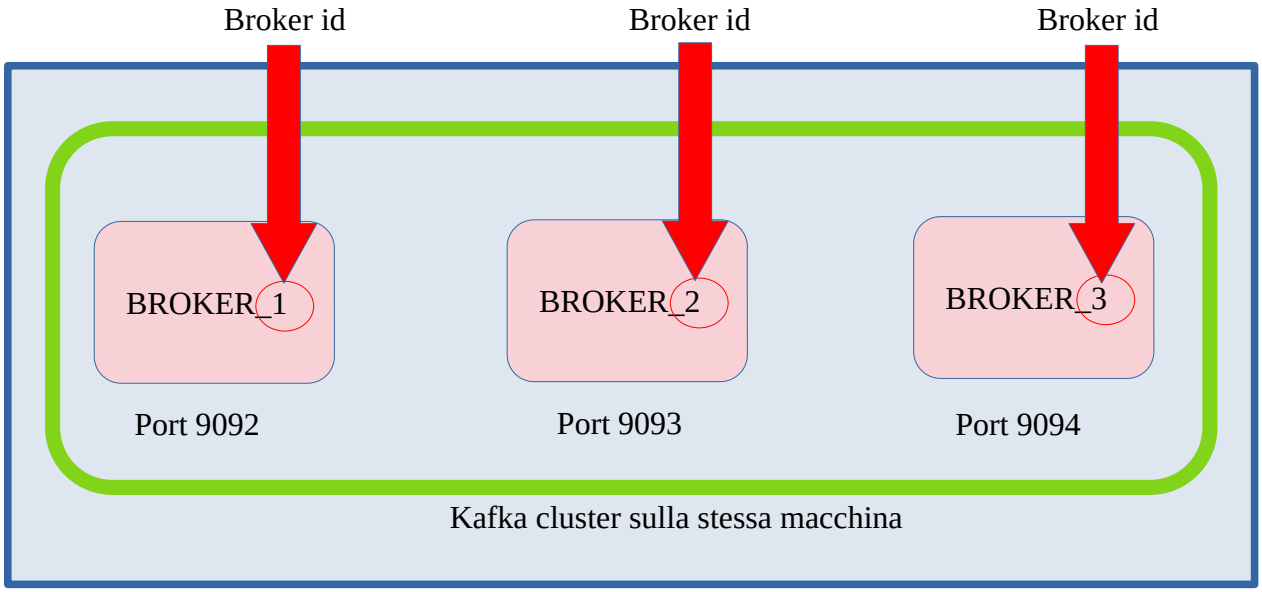
Abbiamo prima detto che un Broker Kafka di fatto è un’applicazione server che gestisce 1 o n topic.
Noi potremmo avere ciascun broker deployato su una macchina diversa oppure sulla stessa assegnando a ciascuno una porta diversa.

Esempio di un cluster con 3 broker ciascuno deployato su una macchina fisica diversa:



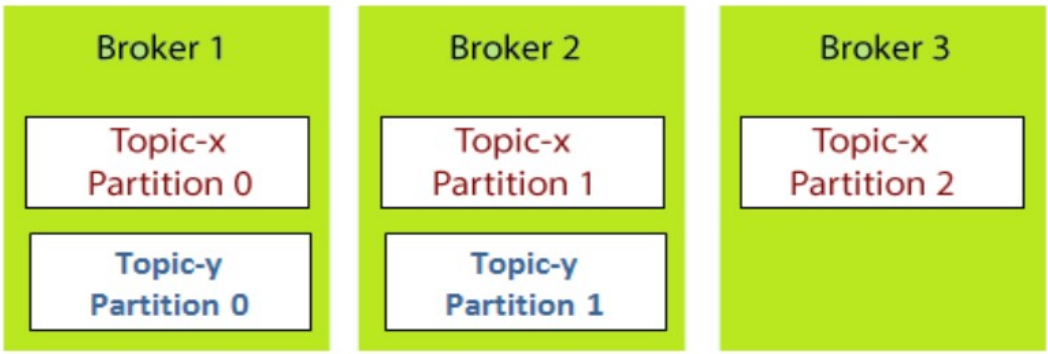
Kafka cluster distribuito su più macchine fisiche

Esempio di un cluster con 3 broker deployati sulla stessa macchina fisica:



NOTA BENE) Ogni Broker è identificato da un id univoco che deve essere necessariamente un numero intero.

NOTA BENE BENE) Abbiamo prima detto che ogni broker contiene uno o più topic nella sua interezza.
In realtà non è proprio così in quanto un broker non contiene un topic nella sua interezza, ma contiene solo una parte di tutte le sue partizioni.
Questo significa che in un determinato broker possiamo avere 3 partizioni su 5 esistenti del topic “Pippo” e 2 partizioni su 4 esistenti del topic “pluto”, ma non conterrà mai tutte le partizioni di un dato topic in quanto Kafka è distribuito.



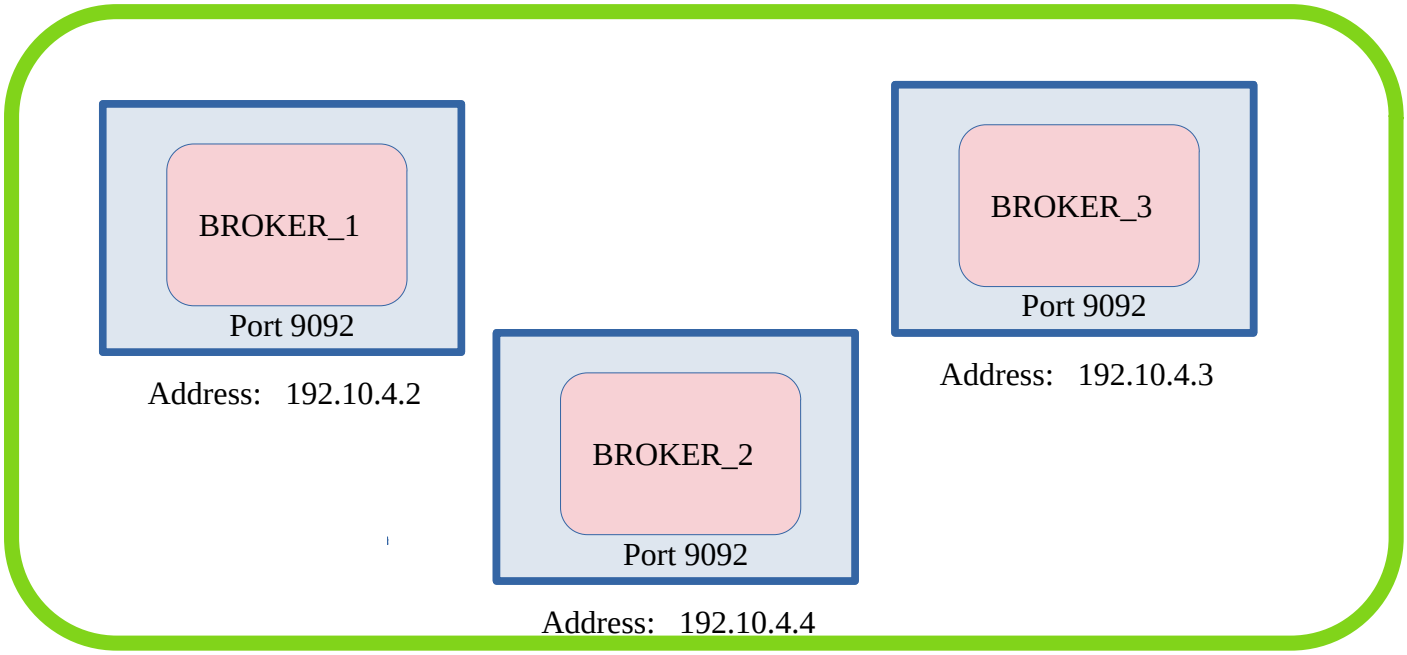
L’immagine sopra mostra tre broker:
- Broker 1
- Broker 2
- Broker 3

e due Topic:
- Topic X con 3 partizioni (p0, p1, p2)
- Topic Y con 2 partizioni (p0, p1)

Osserva come ciascuno dei tre broker non contiene tutte le partizioni di un topic ma solo una parte.
Infatti il broker 1 contiene solo 1 partizione su 3 del topic X e 1 partizione su 2 e del topic Y, stessa cosa broker 2, mentre broker 3 contiene solo 1 partizione su 3 del topic X.

Fatto molto importante è che ogni broker all'interno di un cluster conosce tutti gli altri.
Ad esempio supponenendo di avere un cluster kafka con 3 nodi kafka (o server kafka) come nell'immagine seguente, si ha che:

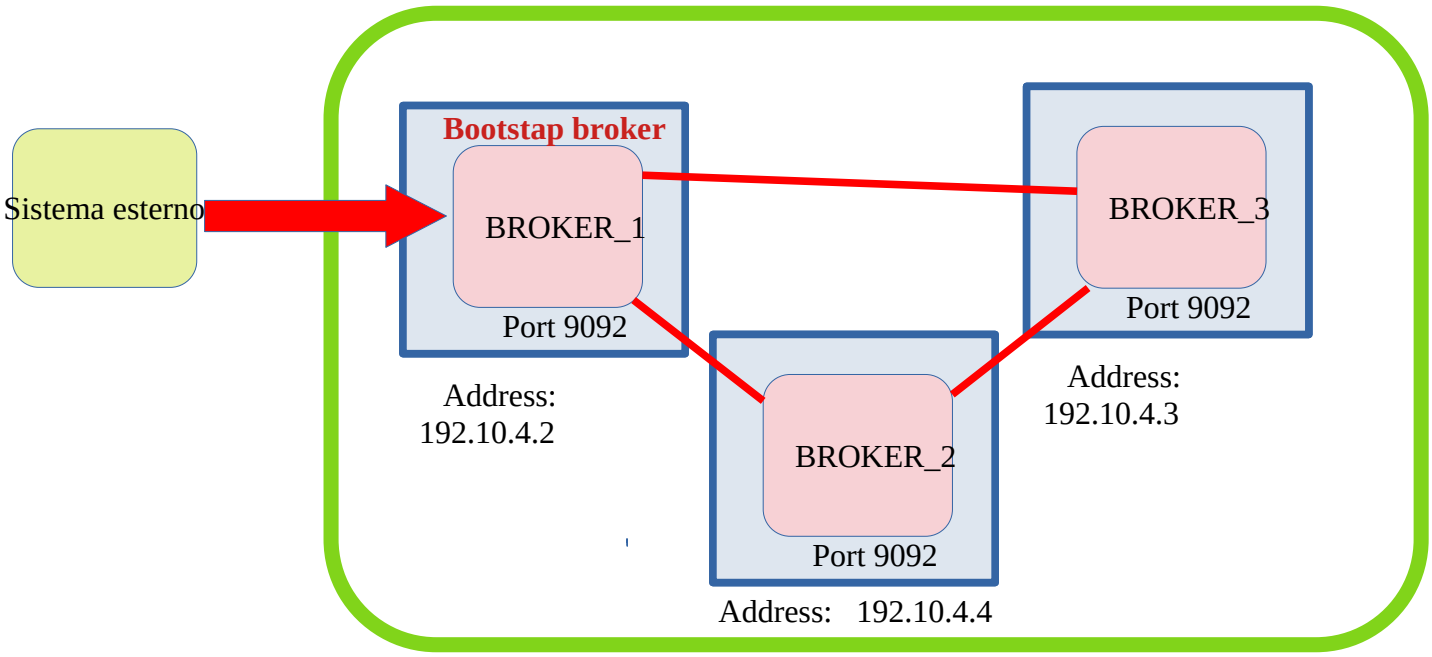
→ Broker_1 conosce Broker_2 e Broker_3
→ Broker_2 conosce Broker_1 e Broker_3
→ Broker_3 conosce Broker_1 e Broker_2



Kafka cluster distribuito su più macchine fisiche

Questo vuol dire che basta connettersi a un broker qualsiasi per essere connessi all'intero cluster e questo succede perché tutti i broker all'interno del cluster si conoscono.

Quando un sistema esterno si connette ad un broker kafka (o server kafka) del cluster, quel broker prende il nome di **bootstrap broker** (o bootstrap server).
Una volta connessi con il bootstrap broker, siamo connessi con l'intero cluster, e questo è vero sempre per il motivo che tutti i broker del cluster si conoscono (sono connessi tra loro), quindi basta connettersi al bootstrap broker per connettersi all'intero cluster.



Kafka cluster distribuito su più macchine fisiche

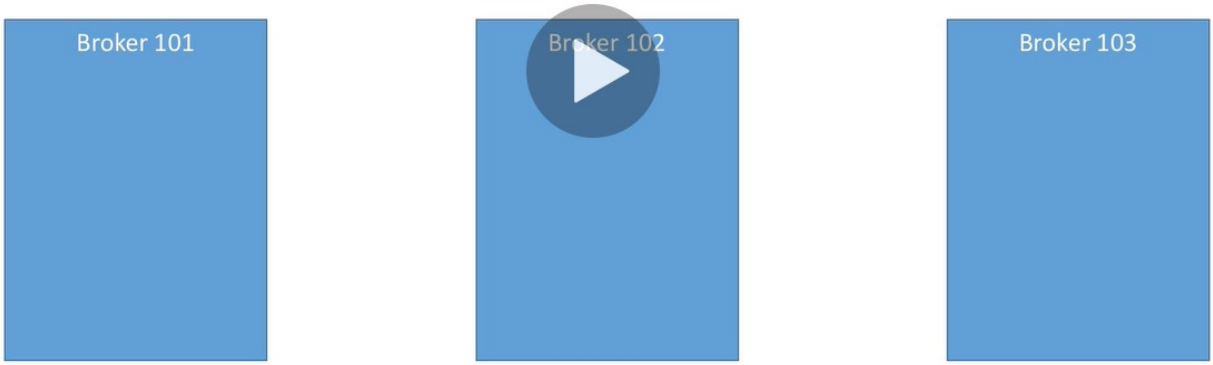
La freccia rossa a sinistra dell'immagine sopra vuole rappresentare un tipico scenario di un sistema esterno che si connette ad un broker di un cluster kafka (che prende quindi il nome di bootsrap broker).

Le linee rosse tra i 3 broker vogliono sottolineare la connessione (conoscenza tra i broker). Si può ben notare come tutti i broker sono connessi tra di loro, quindi appena si è connessi al bootstrap broker in automatico si è connessi all'intero cluster.
Quanto detto vale non sono per cluster con 3 broker ma anche per cluster che hanno in generale n broker.

Vediamo adesso di approfondire meglio il legame tra broker e topic.

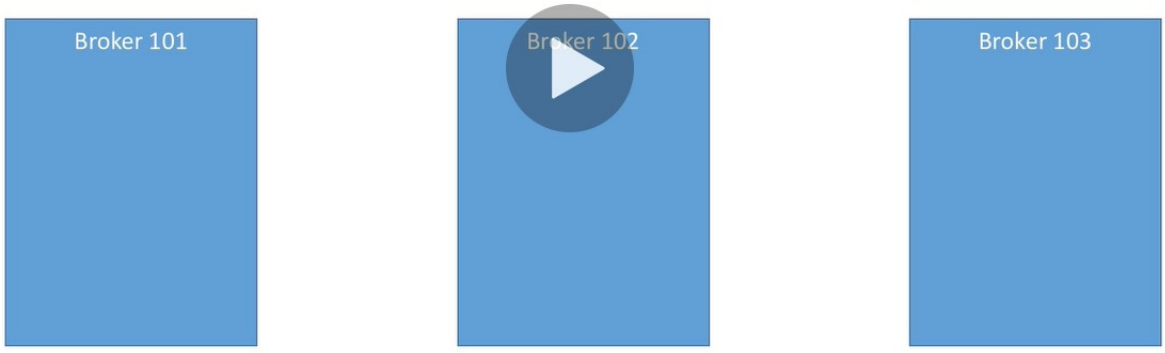
Supponiamo di avere tre broker come nella figura di sotto:

→ Broker con id=101
→ Broker con id=102
→ Broker con id=103



Creiamo adesso un topic che chiamiamo topic A con 3 partizioni p0, p1, p2.

• Example of **Topic-A** with **3 partitions**



Kafka quello che farà è distribuire le partizioni del topic su tutti broker come si può vedere nella figura sotto.

Notiamo come le 3 partizioni sono state distribuite su tutti e 3 i broker.
Se le partizioni erano 2, allora ci sarebbe stato un broker con nessuna partizione al suo interno.
Se le partizioni invece erano più di 3 (ad esempio 4), allora un broker avrebbe avuto più di una partizione.

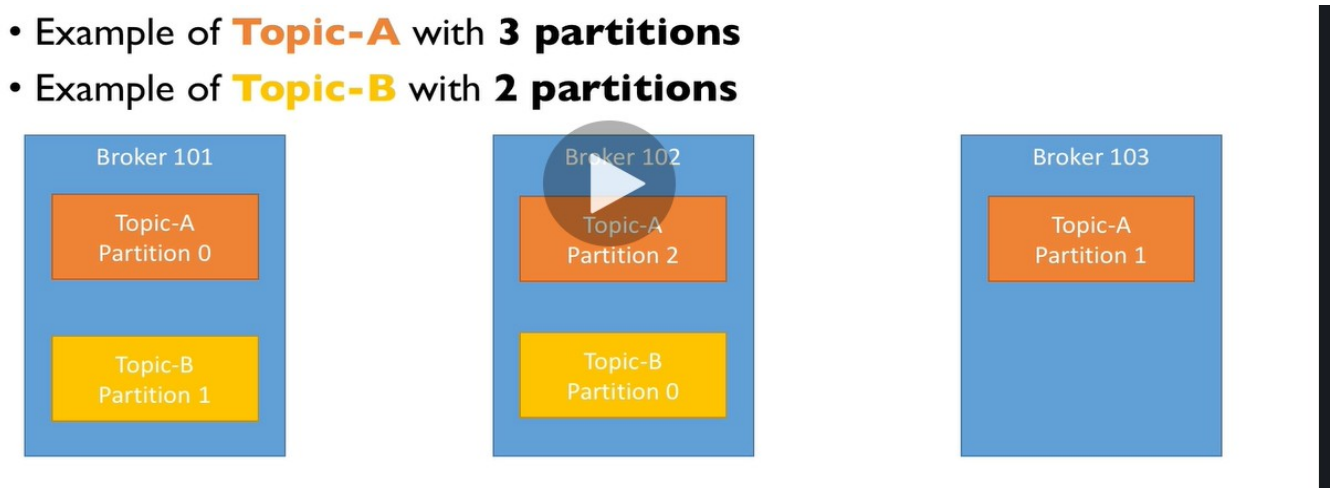
Ora cosa succede se creiam oun topic B con 2 partizioni?

Poichè il numero di partizioni del topic B sono 2 che è minore del numero di broker, allora un broker non avrà nessuna partizione del topic B.

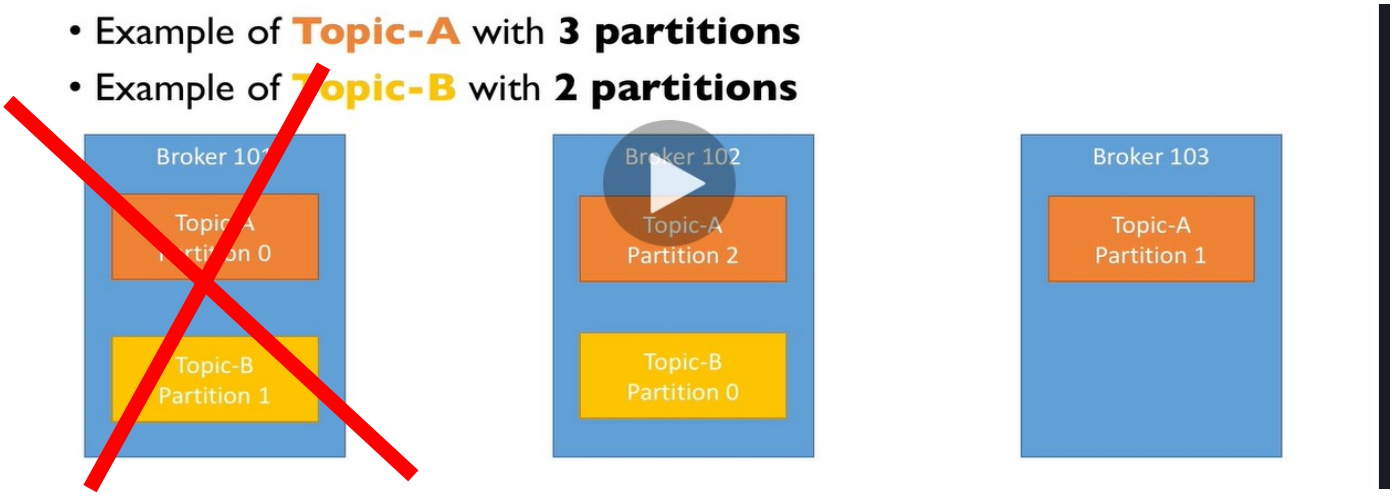
4.0 Repliche

Abbiamo visto precedentemente come un cluster kafka è costituito da diversi broker (minimo 2) installati (deployati) su diverse macchine. Questo rende Kafka un software distribuito.

Abbiamo dopo visto che ogni broker contiene più partizioni al suo interno che appartengono a diversi topic, notando in particolare che un broker non contiene tutte le partizioni di un topic ma un sottoinsieme.



Supponiamo che ad un certo instante t_0, il broker con id=101 deployato sulla macchina fisica con ip x.y.z.w sia giù per qualche motivo.



Di conseguenza avremo come risultato che, partizione 0 del topic A e partizione 1 del topic B non saranno più accessibili.

Per evitare questo inconveniente allora Kafka ci viene in aiuto con le repliche, che altro non sono che delle partizioni clone che risiedono in broker diversi. Così se un broker del cluster cade per qualche motivo, le partizioni saranno ugualmente accessibili in quanto sono duplicate in altri broker e potranno continuare a servire dati.

Un fattore molto importante è quindi scegliere il numero di repliche per ogni partizione all'interno del cluster kafka.
Naturalmente maggiori sono le repliche di una partizione all'interno di un cluster, maggiore sarà la tolleranza ai guasti (fault tollerance) in quanto abbiamo diverse repliche sparse su più nodi all'interno del cluster.

Definiamo quindi **FATTORE DI REPLICA** o **REPLICATION FATCOR** il numero di partizioni uguali che una partizione possiede all'interno del cluster.

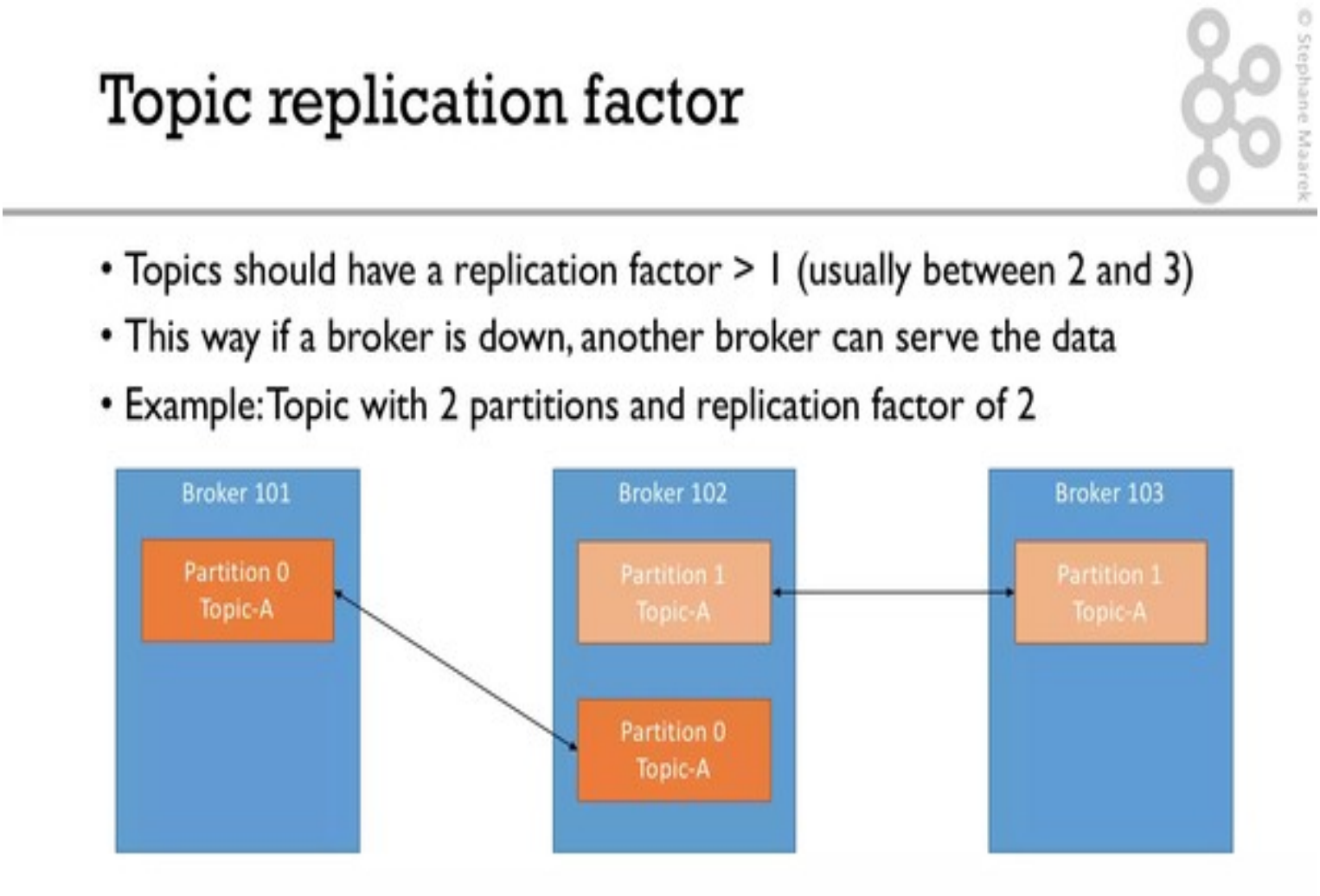
Ad esempio:
* se in un cluster kafka, esistono per ogni partizione altri 2 cloni di quella, allora diremo che quel cluster ha un replication factor = 3
* se in un cluster kafka, esistono per ogni partizione altri 3 cloni di quella, allora diremo che quel cluster ha un replication factor = 4

NOTA BENE 1) Replication factor = 1 equivale a dire che per ogni partizione all'interno del cluster non esistono repliche (cloni) di questa.

NOTA BENE 2) Per avere un minimo di fault tollerance è necessario avere all'interno di un cluster un replication factor > 1.

NOTA BENE 3) Di solito in un cluster viene scelto un Replication factor che è uguale a 2 oppure 3.
Replication factor = 3 è la scelta migliore,
Replication factor = 2 è un po' rischioso.

La seguente immagine mostra un cluster kafka con 3 Broker (o nodi Kafka) avente un fattore di replica = 2.



All’interno del cluster abbiamo 2 partizioni differenti:

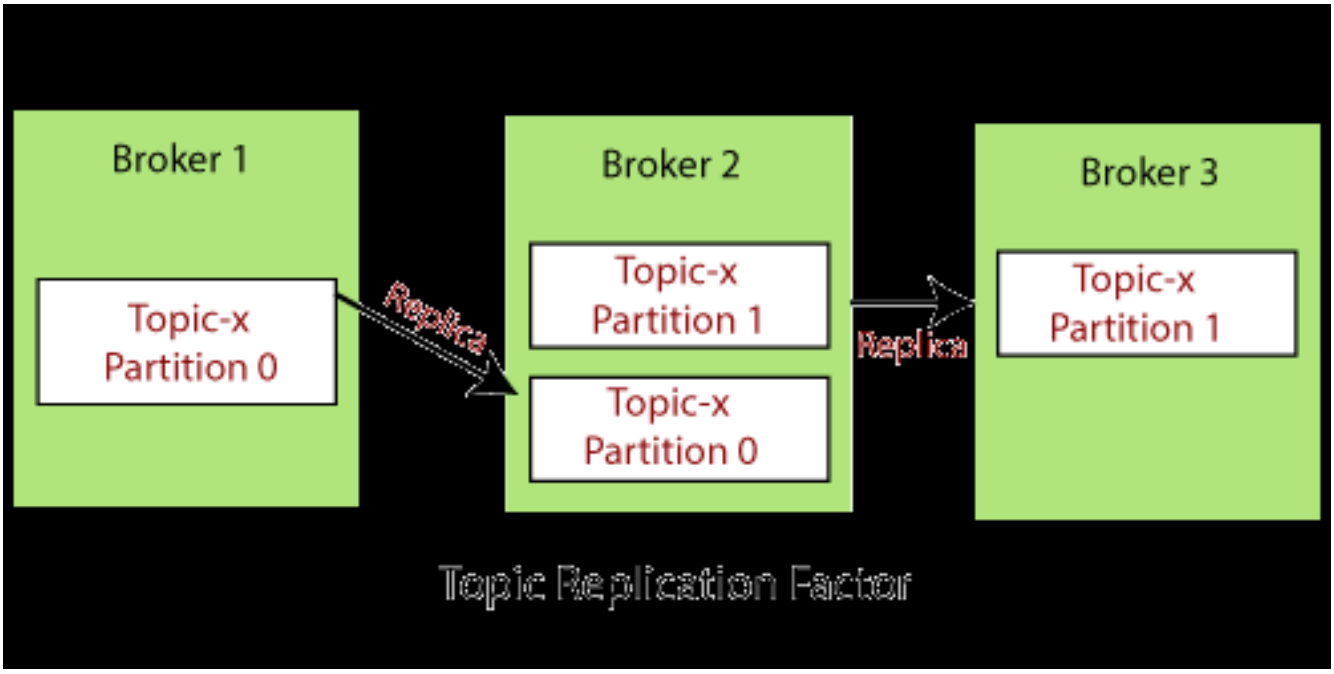
1. Partizione 0 del topic A
2. Partizione 1 del topic A

Per ognuna delle partizioni vi è una copia (clone) che definiamo Replica.
In questo modo se il nodo con id=101 cade, i dati all’interno del partizione 0 del topic A sono ancora disponibile in quanto vi è una replica di questa in un altro nodo (quello con id = 102).

Fattore di replica = 2, vuol dire che abbiamo 2 copie (cloni/repliche) per ogni partizione del cluster.

Una domanda che può sorgere spontanea a questo punto è la seguente:
→ Come facciamo a sapere quali delle partizioni uguali sia la “partizione replica” e quale la “partizione originale/master” ?
Vi è un’entità (vedremo dopo che si tratta di zookeper) che proclama uno dei broker che contiene una copia di quella data partizione come leader/master per quella partizione, e proclama gli altri broker che contengono le altre copie come repliche/follower per quella partizione.

L’immagine seguente mostra un cluster kafka con 3 nodi e fattore di replica = 2.



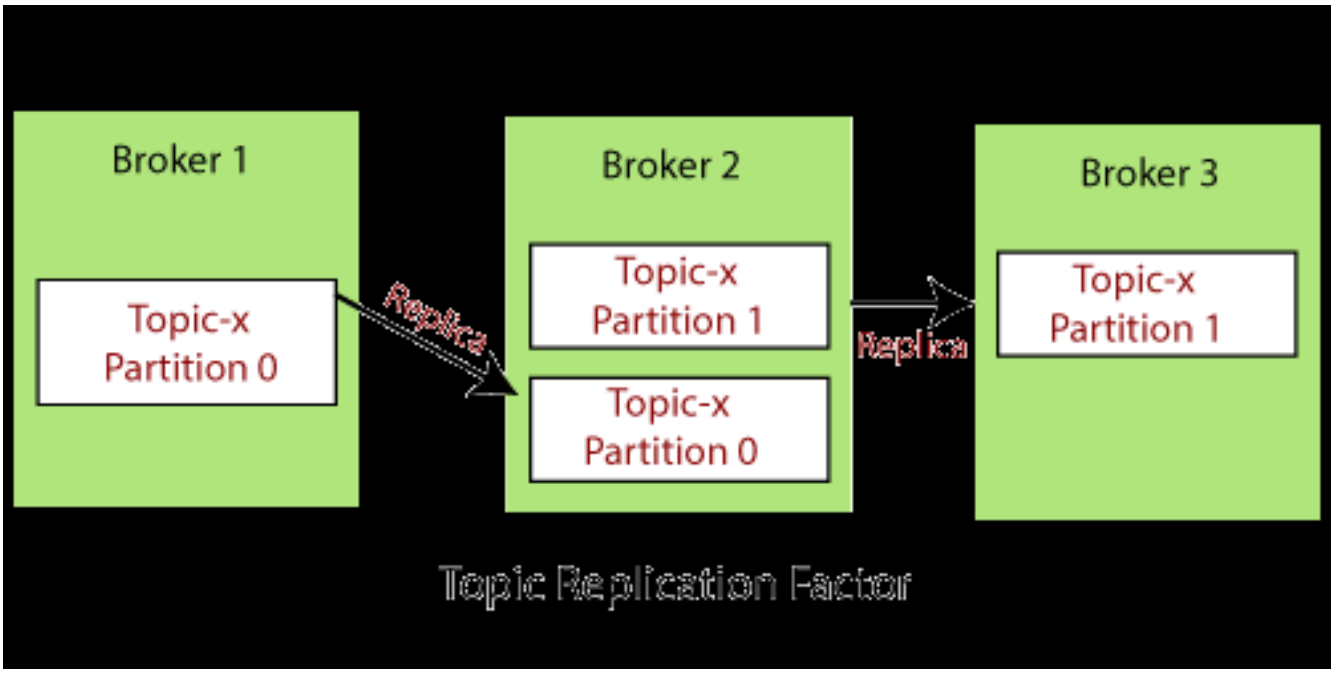
In questo caso Zookepeer ha proclamato:

- Broker con id=1:
    *) **leader/master** per la partizione 0 del topic X.
- Broker con id=2:
    *) **leader/master** per la partizione 1 del topic X
    *) **replica/follower** per la partizione 0 del topic X
- Broker con id=3:
    *) **replica/follower** per la partizione 1 del topic X

NOTA BENE 1) In qualsiasi istante t solamente un broker può essere il leader per quella data partizione. Questo vuol dire che non è possibile avere lo scenario che 2 broker sono contemporanemente leader per una stessa partizione X del topic Y.

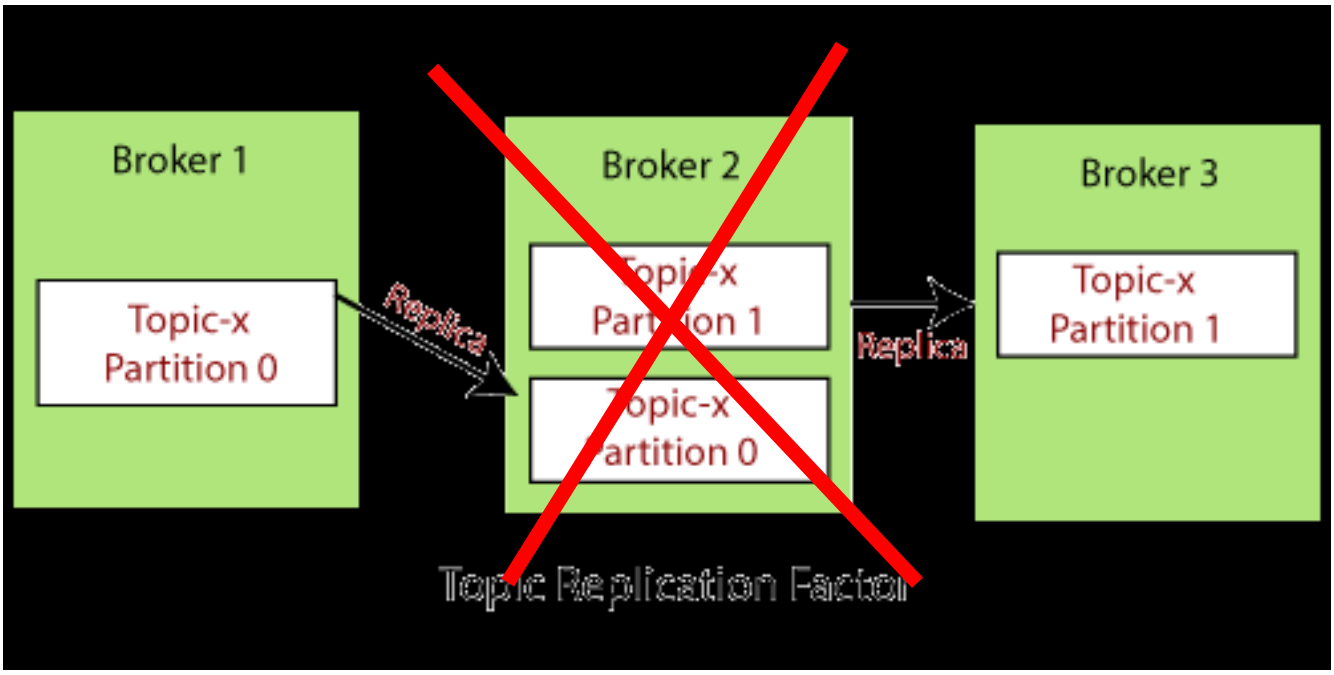
NOTA BENE 2) Solo il broker leader (che è unico per quella data partizione) può ricevere ed inviare dati.

NOTA BENE 3) I broker che sono proclamati replica per quella data partizione avranno sempre i dati sincronizzati al leader.



Una domanda che ci sorge a questo punto è la seguente:
→ Cosa succedere se un broker che è leader per quella partizione è giù?

Ad esempio, abbiamo prima detto che Broker con id=2 (dell'immagine prcedente) è leader per la partizione 1 del topic X. Ma se questo è giù cosa succede?



Quello che succede è che Zookeper in questo caso eleggerà:

*) **leader** il Broker con id=1 per la partizione 0 del topic X.

*) **leader** Il Broker con id=3 per la partizione 1 del topic X.

Non appena il broker con id=2 ritornerà disponibile verrà ripristinata la leadership dei broker come era in precedenza dopo essersi prima sincronizzato allo stato attuale.

NOTA BENE)

Tutte queste attività di:

- elezione dei broker leader e repliche per una partizione
- sincronizzazione tra leadere e repliche
- ripristino della leadership dopo che un broker guasto viene ripristinato
- ecc...

sono svolte da zookeper che verrà trattato in seguito.

5.0 Producers e message keys

In precedenza abbiamo parlato di Topic, partizioni, broker, repliche, ecc.. .
Ma ancora non abbiamo parlato di come fa Kafka ad ottenere i dati.
Bene questo è il ruolo che svolge un **Producer** (produttore)

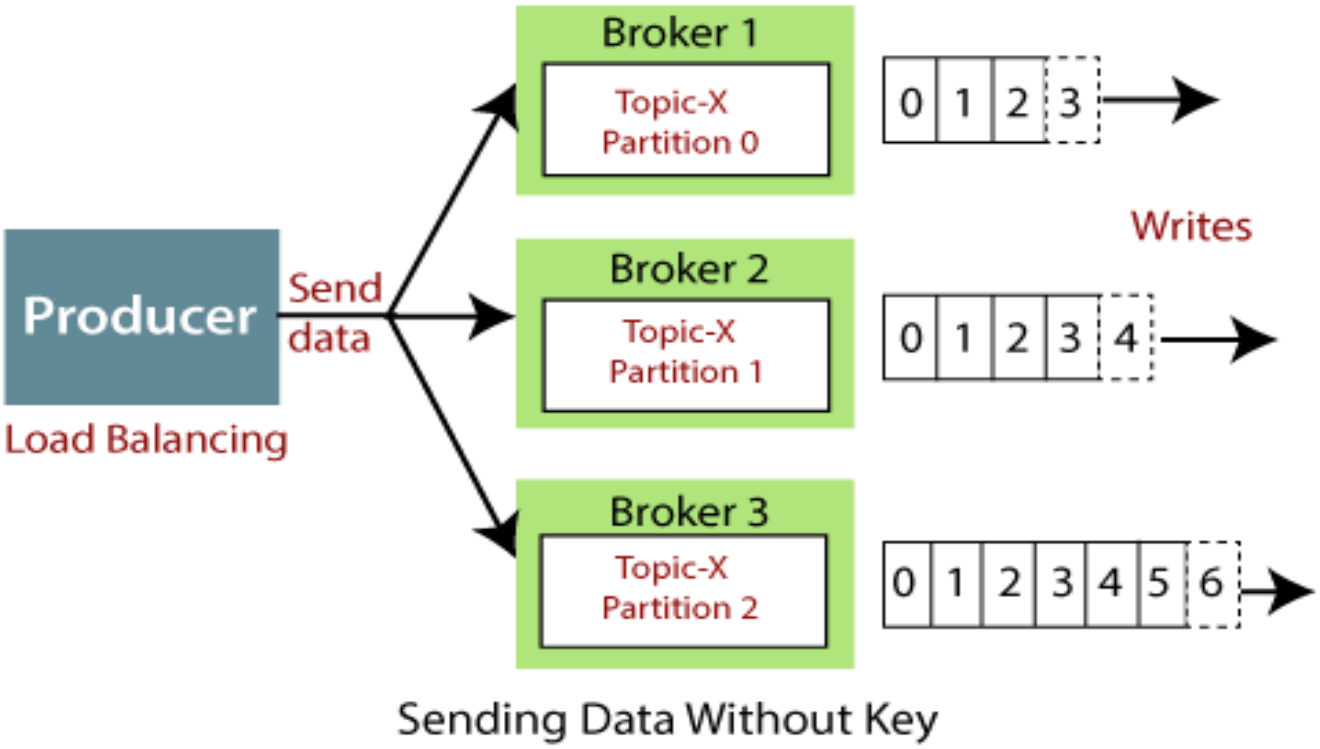
Un producer è semplicemente un applicativo esterno a Kafka che scrive dei dati all'interno di un topic Kafka (più precisamente all'interno di una partizione di un topic Kafka) e quindi poiché introduce dei dati all'interno dell'ecosistema Kafka viene etichettato come producer.

Un producer scrive dei dati in un topic (e ricordiamo i topic sono fatti da più partizioni, quindi in realtà un producer scrive in una partizione di un topic).

Quando per qualche motivo un broker è giù, allora tutti i dati contenuti nei topic del broker sarebbero indisponibili, ma non è così, i producer recupereranno i dati contenuti in quel broker in altri broker del cluster in modo trasparente se configuriamo un fattore di replica > 1.
Questa caratteristica viene chiamata "AUTOMATICALLY RECOVER", ovvero recupero dei fallimenti automatici ed è gestita in modo trasparente da zookeper.

L’immagine seguente mostra uno scenario reale, dove abbiamo:
→ un applicativo esterno a Kafka che funge da produttore (Producer) che invia dati verso Kafka.
Nello specifico invia dati verso 3 partizioni:

- 0 del topic X
- 1 del topic X
- 2 del topic X



La domanda che ci poniamo adesso è:
→ *A quale broker invia i dati il producer ?*

In sostanza, se un produttore invia dei dati senza nessuna chiave associata, allora i dati verranno inviati in modo round robin ai vari broker.
Questo significa che ad esempio il primo dato verrà inviato nella partizione contenuta nel broker 1, poi in quella contenuta nel broker 2, poi in quella contenuta nel broker 3, e successivamente si ricomincia ad inviare alla partizione del broker 1, poi in quella del broker 2, e così via...

Questo significa che se il produttore invia dati verso Kafka senza alcuna chiave, allora per distribuire meglio il carico di lavoro sui diversi broker (load balancing), Kafka adotta la strategia round robin sui diversi broker.
In questo modo non stressiamo un broker solamente inviando solamente dati a lui, ma distribuiamo il carico di lavoro su tutti i broker.

Ok abbiamo visto la strategia di default che adotta Kafka per bilanciare il carico di lavoro.

Ma adesso sorge un altra domanda:
→ *Come fa il producer ad essere sicuro che i dati sono stati scritti memorizzati all'interno di un topic?*

In un certo senso servirebbe una risposta di Kafka che dice: "si tutto ok ho scritto il tuo dato nel topic da te richiesto".
Ovvero quello che in gergo informatico viene chiamato ack o acknowledgement che è sinonimo di conferma dell'operazione richiesta.

Ed In effetti è proprio così: il producer quando invia un dato ad un topic Kafka può dire se avere indietro un acknowledgement o meno sul dato inviato, in modo da essere assolutamente certo che quel dato è stato scritto nel topic senza essersi perso in qualche modo.

Esistono 3 tipi di acknowledgement che un producer può configurare per essere a conoscenza della fine che ha fatto il suo dato inviato a Kafka:

1) **acks = 0**

   Il producer configurando acks =0, sta dicendo che non intende aspettare nessuna conferma sul dato che ha inviato e se questo è stato scritto o meno sul topic (possibile perdita di dati perché se il producer invia un dato ma il broker non è attivo, il producer non ricevendo nessuna risposta sul dato inviato questo non verrà scritto e quindi perso per sempre)

2) **acks = 1** (comporatemento di default)

Il producer configurando acks =1, sta dicendo che intende aspettare la conferma di scrittura del dato nel topic da parte del broker leader. Quindi ad esempio il produce invia i dati nella
partizione 2 del topic X broker 3.
Il producer aspetterà la conferma di scrittura del dato solo da parte del broker leader.
(vi è una perdita di dati minore rispetto al caso precedente ma c'è sempre una percentuale di possibilità di perdita dei dati nel caso il broker leader è giù questo perché i dati sono stati si memorizzati nel leader ma potrebbero non essere stati memorizzati per un qualche motivo nelle repliche in quanto abbiamo solo atteso la conferma del broker leader)

3) **acks = 2** (oppure acks = **ALL**)

Il producer configurando acks =2, sta dicendo che intende aspettare la conferma di scrittura del dato nel topic da parte del broker leader e anche da parte di tutte le repliche.
Quindi ad esempio il producer invia i dati nella
partizione 2 del topic X broker 3.
Il producer aspetterà la conferma di scrittura del dato sia da parte del broker leader che da tutte le repliche.
Quindi in questo caso sia il broker leader che le repliche dovranno dire in un certo senso
al producer: "ok, ho memorizzato i tuoi dati"
(nessuna perdita di dati)

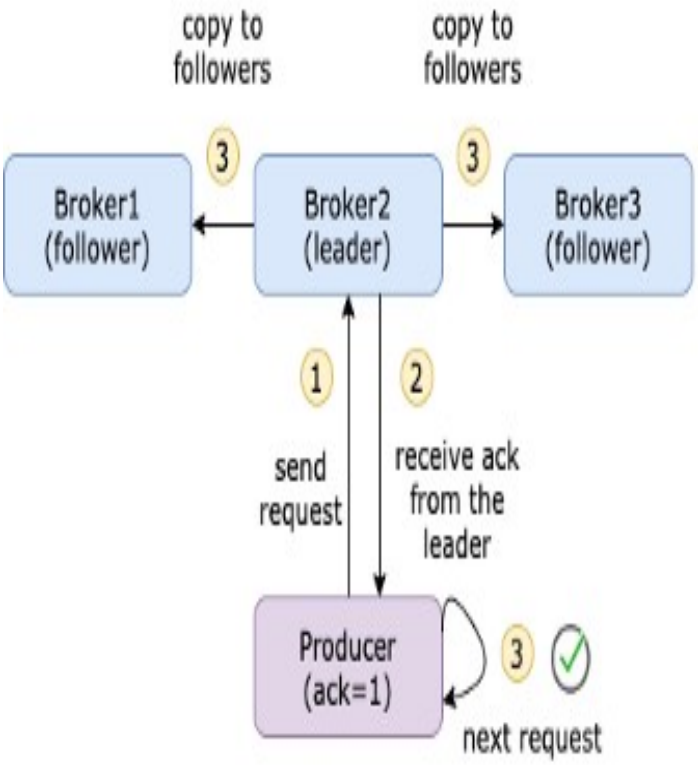
RICORDA:

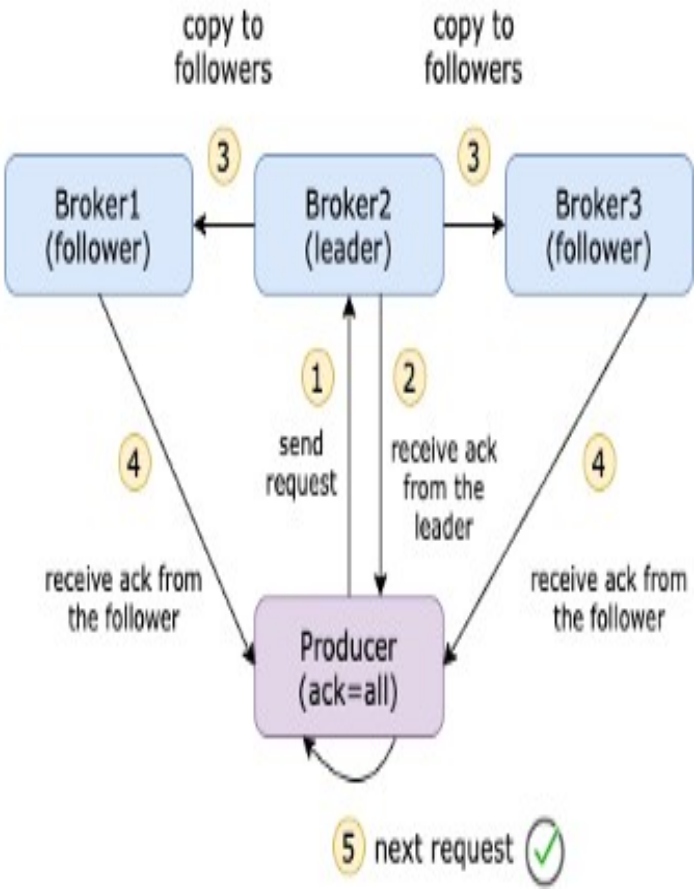
- ack = 0 → nessuna risposta di conferma di scrittura del dato da Kafka al producer.
- ack = 1 → risposta di conferma di scrittura del dato solo da parte del broker leader.
- ack = 2 → risposta di conferma di scrittura del dato da parte del broker leader e anche da parte di tutte le repliche.

ack = 0 → elevata probabilità di perdita di dati

ack = 1 → bassa probabilità di perdita di dati

ack = 2 → probabilità nulla di perdita di dati





Adesso parliamo delle chiavi dei messaggi per i producer.

Un producer quando invia un messaggio/dato ha 2 possibilità:

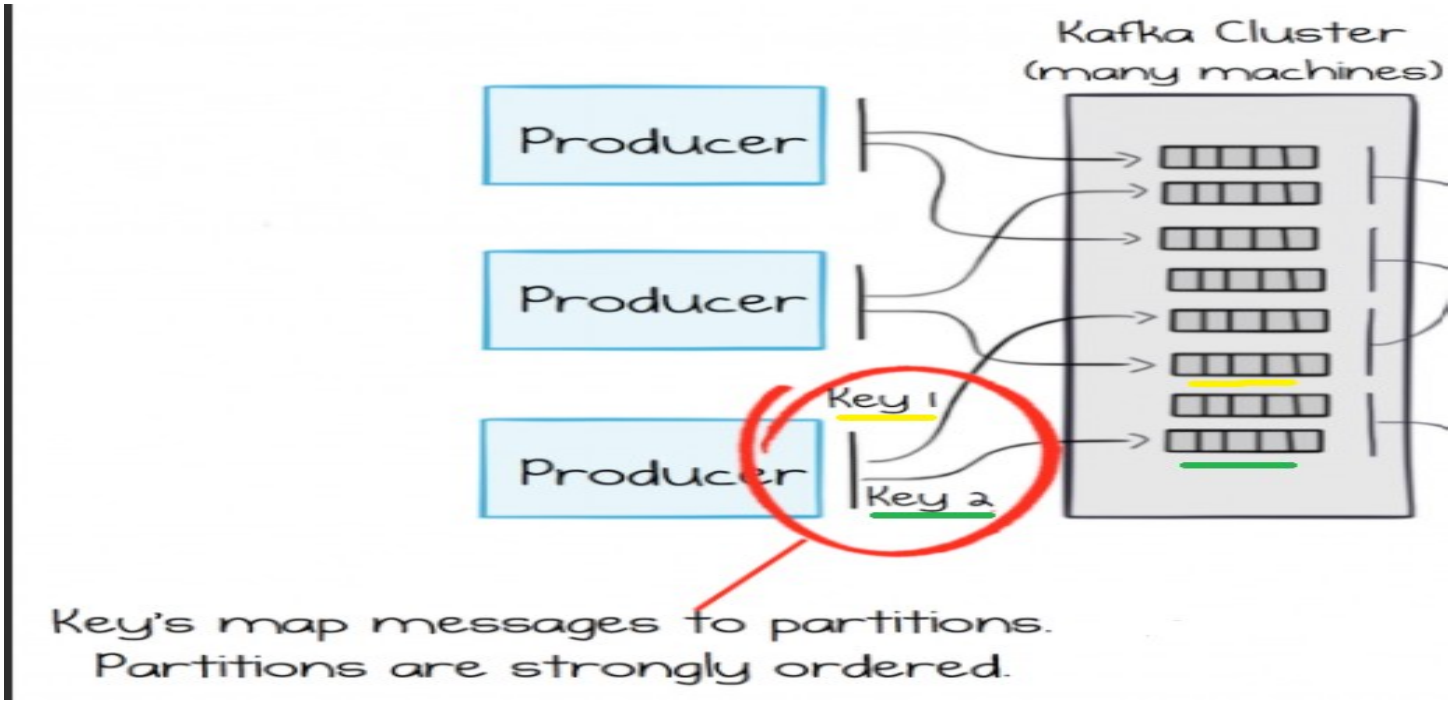
1 - inviare il messaggio/dato con una chiave associata
2- inviare il messaggio/dato senza chiave associata

Inoltre abbiamo visto in precedenza che in caso non vi è nessuna chiave associata al dato/messaggio, Kafka lo gestisce applicando l'algoritmo round robin e bilanciando il carico dei messaggi sui diversi broker che contengono quel topic.

La **chiave può essere qualsiasi cosa**:

- una stringa,
- un numero intero,
- qualsiasi cosa...

Se la chiave=null, allora è come non averla e quindi Kafka adotterà la strategia round robin per bilanciare il carico di lavoro.
Altrimenti se i messaggi verranno inviati con una chiave, allora tutti i messaggi che hanno la stessa chiave, verranno inviati sempre e comunque nella medesima partizione, questo particolarità è garantita da Kakfa (nello specifico da zookeeper).



Key's map messages to partitions.
Partitions are strongly ordered.

6.0 Consumers e conusumer group

7.0 Consumer Offsets e Delivery semantics

8.0 Kafka broker discovery

9.0 Zookeeper

10.0 Garanzie offerte da Kafka

11.0 Download Kafka

12.0 Installazione Kafka in Windows

13.0 Avvio Zookeper e Kafka

14.0 Introduzione alla CLI di Kafka

15.0 Kafka Topics CLI

16.0 Kafka Console Producer CLI

17.0 Kafka Console Consumer CLI

18.0 Kafka Consumers in group

19.0 Kafka Consumers group CLI

21.0 Intro to Kafka programming

22.0 Creazione primo progetto Kafka

23.0 Creazione primo Producer in Java

24.0 Java Producer Callback

25.0 Java Producer con chiavi

26.0 Java Consumer

26.0 Java Consumer all’interno di un un consumer group

20.0 Resetting offsets