Kafka: streaming data

Indice

1. Motivazioni

Negli ultimi anni l'avvento delle architetture a microservizi ha portato la necessità di studiare nuove soluzioni al problema della gestione di molteplici fonti di dati.

In sistemi complessi formati da più microservizi tanti componenti interdipendenti comunicano tra loro scambiandosi dati e attingendo da numerose fonti di dati comuni come database, data warehouses oppure servizi esterni.

La necessità di filtrare, standardizzare e gestire molte fonti di dati aveva portato alla nascita del processo di **Extract, Transform, Load** (ETL) per l'estrazione, trasformazione e caricamento di dati in sistemi di sintesi come data warehouse o data mart, questo processo si sta però rivelando complicato ed impegnativo in un mondo dove la mole di dati prodotta dal logging di eventi critici ad un qualsiasi business è in continua crescita: semplici esempi sono la gestione degli eventi in un sistema **Internet of things** (IoT) oppure lo studio delle abitudini dei propri clienti per un servizio di e-commerce.

Lo stream processing tra microservizi propone un nuovo approccio per la gestione di questi problemi, fornendo una soluzione adatta alla gestione di dati in real-time altamente scalabile e ad alto throughput.

Apache Kafka è una piattaforma di streaming dati nata in un contesto aziendale importante che mira a rivoluzionare il modo con cui i microservizi di un business comunicano tra loro, favorendo un approccio improntato sulla gestione di eventi legati al comportamento dei dati, più che i dati in se.
Kafka nasce per sfruttare a pieno lo stream processing e favorire una gestione intelligente di grosse moli di dati, abbandonando il classico processo "batch" ETL per una soluzione, appunto, basata sullo streaming dei dati tra microservizi.

2. Introduzione

Prima di poter discutere della soluzione architetturale fornita da Apache Kafka e quali vantaggi propone rispetto alle soluzioni di batch ETL è necessario approfondire alcuni temi, tra cui il più importante è sicurante event sourcing (ES).

Il primo capitolo presenta una veloce descrizione di un processo ETL in modo da dare visione del oggetto di cui si vuole dibattere l'effetiva praticità.

Il secondo capitolo è utilizzato per illustrare uno dei concetti chiave della tesi: l'importanza di gestire e vedere le basi di dati come eventi, perchè utilizzare il concetto di eventi per modelizzare i dati ed infine, quale strumento è possibile utilizzarlo per farlo.

Il terzo capitolo è dedicato ad esaminare la piattaforma Apache Kafka, sia da un punto di vista tecnico-architetturale, esaminando le singole parti che compongono la piattaforma, sia l'ecosistema che si è venuto a creare intorno alla piattaforma, principalmente l'utilizzio delle librerie Kafka Connect, Kafka Streams e il recente sviluppo di KSQL, un linguaggio SQL-like per ricerche su stream di dati in tempo reale; Viene inoltre presentato come event sourcing si collega perfettamente a Kafka.

Infine vengono presentati degli esempi di utilizzo di Kafka e, nelle conclusioni, vengono messi a confronto i processi ETL e soluzioni di streaming come Kafka nel contesto di gestione di grosse moli di dati tra microservizi.

3. ETL

Un processo di Extract, Transform, Load (ETL) è un processo mirato alla trasformazione di dati contenuti su più database per ottenere un nuovo insieme di dati, filtrato e traformato secondo una particolare logica, destinato ad essere salvato in una data warehouse.

Verso la fine degli anni '70 molte aziende iniziarono ad utilizzare molteplici database per salvare e gestire informazioni, è proprio in questo contesto che nascono i processi di ETL: con l'avanzare del tempo è stato necessario studiare un metodo per l'aggregazione e gestione delle varie fonti di dati.

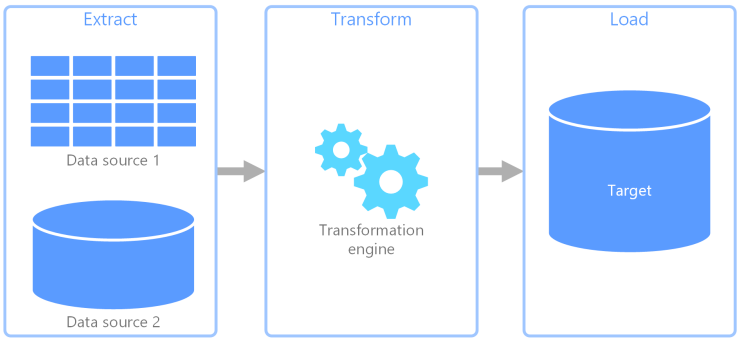

Figure 1: etl_process

Un processo di ETL si compone di tre parti:

La prima parte del processo di ETL è la fase di **Extract** ed involve l'estrazione dei dati da più data sources come database relazionali o non-relazionali, file JSON od XML ma anche risorse "adhoc" come, ad esempio, dei dati generati da programmi di web analytics.

L'obiettivo di questa fase è estrarre tutti i dati necessari dalle possibili sorgenti e prepararli alla fase di Transform.

Un importante problema legato a questa fase è il processo di **validazione delle sorgenti dati**: con più sorgenti dati spesso ci si ritrova a dover gestire più *formati* non necessariamente compatibili tra loro.
Per poter garantire alla fase di Transform dei dati comprensibili, durante la fase di Extract vengono definite delle *regole di validazione* per filtrare i dati provenienti dalle varie sorgenti, un esempio di regola di validazione è il controllo dei tipi di dati presenti nella fonte.

Nella fase di **Transform** una serie di regole e funzioni vengono applicate ai dati generati dalla fase di Extract per prepararli alla fase di Load nella data warehouse.

Il primo compito della fase di Transform è la **pulizia dei dati**: spesso le varie fonti di dati, nonostante siano state validate, possono presentare incongruenze tra loro come caratteri speciali legati all'encoding della propria sorgente oppure formati dei dati diversi ma compatibili (un esempio può essere la differneza di formattazione tra date americane ed europee).
Per garantire un corretto funzionamento delle operazioni di trasformazione è quindi necessario pulire i dati ed adattarli ad un formato comune.

Il secondo compito della fase di Transform è la **trasformazione dei dati** in nuovi dati richiesti dal business, esempi di trasformazioni sono:

- Joining di tabelle da più sorgenti
- Mapping e trasformazione di dati (esempio: "Maschio" in "M")
- Aggregazione di dati
- Generazione/calcolo di nuovi dati
- Selezione di insiemi di dati
- Validazione del nuovo formato di dati prodotto

Nella fase di **Load** l'insieme di dati generati dalla fase di Transform vengono inseriti in un target, il quale potrebbe essere una data warehouse ma anche più semplicemente un file in un formato utile.
Business diversi hanno necessità diverse, per questo l'implementazione della fase di load può avere più modalità implementative, il punto focale di questa fase è proprio stabilire la frequenza e le modalità di aggiornamento dei dati presenti nel target.
Decidere la frequenza (giornaliera, mensile, ecc.) e le modalità (sovrascrizione dei vecchi dati o meno) del target possono portare ad un processo di ETL più o meno utile ad una azienda.

Per generare un buon target è buona norma definire uno schema *preciso e chiaro* della tipologia di dati a cui il target deve aderire.
Come detto in precedenza un processo di ETL è utilizzato per aggregare più fonti di informazioni comuni ad un processo aziendale, questo suppone che le informazioni presenti nella data warehouse potrebbero venire usate da più parti di una azienda, le quali potrebbero essere abituate a particolari formati dei dati. Senza definire uno schema dei dati chiaro e preciso, si correrebbe il rischio di generare un insieme di dati inutilizzabile da determinati reparti in quanto non conforme al formato di dati da loro conosciuto.

4. Event sourcing

4.1. L'importanza dei dati e degli eventi

Lo status quo delle moderne applicazioni web è basato sul utilizzo di database per rappresentare le specifiche di dominio, spesso espresse da un cliente e/o da un esperto del dominio esterno all'ambiente di sviluppo.

Durante la fase di analisi dei requisiti (supponendo un modello di sviluppo del software agile) cliente e team di sviluppo si confrontano, cercando di trovare un linguaggio comune per definire la logica e l'utilizzo del software richiesto; Una volta stabiliti i requisti, il team di sviluppo generalmente inizia uno studio interno atto a produrre un **modello dei dati** che verrà usato come base per definire lo schema dei database utilizzati dal sistema.
Un cliente comune molto spesso non ha padronanza del concetto di 'stato di una applicazione', ma piuttosto si limita ad esporre i propri requisiti descrivendo i possibili **eventi** che, traslati sul modello di sviluppo incentrato su i database, portano il team di sviluppo a ragionare sui possibili stati di un database in risposta a questi eventi.

Lo stato di un database di una applicazione è strettamente legato all'insieme degli eventi del dominio applicativo; L'unico modo per modificare o interagire con questo database è tramite i comandi di inserimento, cancellazione o lettura, tutti comandi che vengono eseguiti solamente all'avvenire di un particolare evento.

Un database mantiene solo lo stato corrente di una applicazione; Non esiste il concetto di cronologia del database a meno di utilizzare soluzioni basate su **Change Data Capture** (CDC), generalmente utilizzate per generare un transactional log contenente tutte le operazioni eseguite sul suddetto database. In questo modello database-driven, un evento genera un cambiamento su una base di dati; Gli eventi e lo stato di un database sono però concetti diversi e slegati tra loro, l'esecuzione di un evento a volte può portare ad una asincronia tra l'esecuzione di un evento e lo stato di un database, tanto più se questo database è utilizzato da tutti i microservizi di una applicazione.

Una soluzione al problema di più microservizi che utilizzano lo stesso database è di utilizzare delle views del database locali ad ogni microservizio: ogni servizio lavorerà su una copia locale del database ed un job esterno si occuperà di compattare le views e mantenere il database aggiornato rispetto a tutti i cambiamenti.
Questa soluzione ha un enorme problema: supponiamo di notare un errore sul database e di doverlo correggere, come possiamo decidere quale delle views è "più corretta" delle altre? Per aiutarci nella ricerca dell'errore potremmo utilizzare il transactional log di ogni views, ma su database di grandezze importanti esaminare il log di ogni views che lo compone potrebbe essere un problema complesso e dispendioso in termini di tempo.

Event sourcing propone di risolvere questo genere di problemi allontanandosi da

una progettazione state-driven elevando gli eventi a elementi chiavi del modello dei dati di una applicazione.

4.2. Descrizione

Event sourcing (ES) è un design pattern che si contrappone ad una visione del mondo basata sullo stato di una applicazione fornendo come alternativa l'uso degli eventi, ovvero delle azioni o accadimenti che l'applicazione è in grado di riconoscere e gestire.

Durante l'analisi dei requisiti di una applicazione, spesso ci si trova a confronto con esperti di un dominio applicativo che non hanno particolare conoscenza delle tecnologie necessarie per implementare le loro richieste, è compito del programmatore (o del team di programmatore) analizzare le sue richieste e trasformarle in idee gestibili.
In genere questi esperti spiegheranno al programmatore le loro necessità illustrando il funzionamento del dominio utilizzando concetti molto più vicini a degli *eventi* piuttosto che *sequenze di richieste/risposte a/da un database*; Supponendo di dover sviluppare una piattaforma di e-commerce, è molto più probabile che l'esperto di dominio richieda di gestire eventi come "aggiungere un oggetto al carrello" oppure "comprare un oggetto" piuttosto che "creare dei database per gestire carrello, stock oggetti rimamenti, oggetti comprati".

La struttura dati fondamentale alla base di ES è l'**event store**, una tipologia di database ottimizzata per la gestione di eventi.
In un event store, gli eventi vengono inseriti in fondo alla struttura in ordine di avvenimento e non possono essere modificati o cancellati; Nel caso venga pubblicato per errore un evento sbagliato o inesatto per annularlo basterà pubblicare un evento contrario. Questo meccanismo garantisce che la ripetizione della storia degli eventi *porterà sempre allo stesso stato, errore compreso.*

Un event store è comunemente implementato utilizzando un **log**, una sequenza di record append-only e totalmente ordinata in base al tempo di scrittura del record.

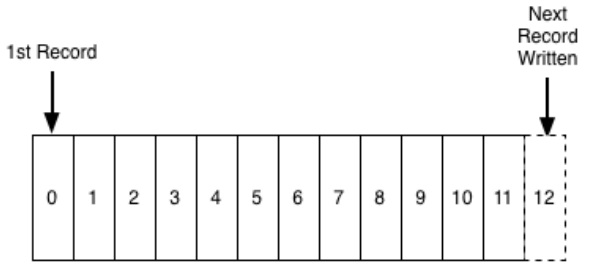

Figure 2: log_data_structure

I record sono inseriti in fondo al log e il processo di lettura è eseguito partendo dall'inizio del log.

Generalmente in un processo di sviluppo basato su ES, si tende a nominare gli eventi con il tempo passato per esplicitare il concetto che un evento è un avvenimento passato, un esempio nome per un evento potrebbe essere `item_created` oppure `item_bought`.

L'ordine di pubblicazione degli eventi è di estrema importanza in quanto è ciò che permette al pattern di rappresentare correttamente lo stato di una applicazione.

E' possibile vedere lo **stato corrente di una applicazione** come una **sequenza di operazioni di modifica dello stato eseguite partendo da uno stato iniziale**, questo implica che è possibile vedere un **evento** come il **delta tra lo stato iniziale di una applicazione e lo stato corrente dell'applicazione dopo l'esecuzione dell'evento**.
La possibilità di trasformare lo stato corrente di una applicazione in una funzione dello stato iniziale dell'applicazione e una sequenza di eventi è il meccanismo che permette ad event sourcing di avere una validità tecnica per la gestione dei dati di una applicazione.

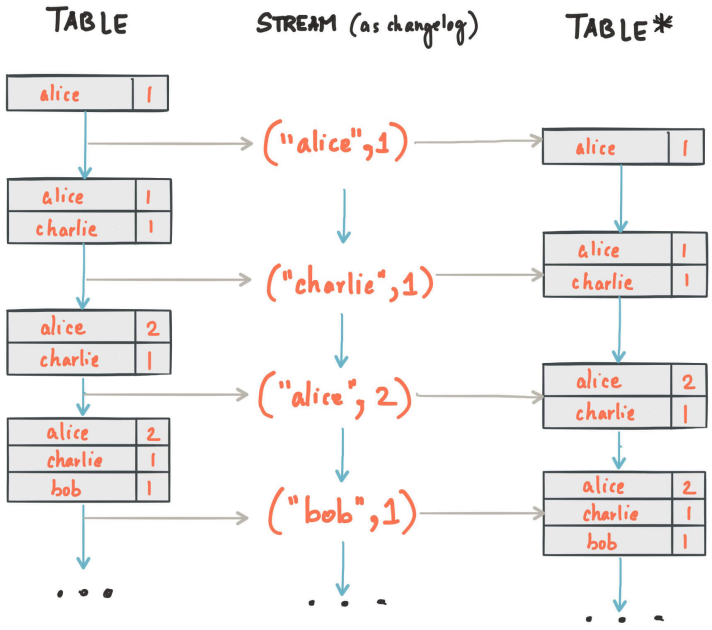

Figure 3: event_log_database_duality

4.3. Vantaggi

Event sourcing è un pattern estremamente utile per tutti quegli use-case dove è assolutamente necessario mantenere una storia dello sviluppo dello stato dell'applicazione del tempo. Tipici esempi sono gli strumenti di versioning del codice oppure la gestione di un storico bancario.

La capacità dell'event store di essere sia una struttura dati performante per la scrittura dei dati (è una semplice operazione di scrittura in fondo ad una sequenza di complessità O(1)) che una cronologia di tutti gli avvenimenti del sistema, permette una gestione degli errori estremamente semplice.

In un qualsiasi database relazionale se durante il normale utilizzo dell'applicazione avviene un errore logico che porta il database ad uno stato non corretto, è sempre necessario un rollback dell'intero database ad un data antecedente l'errore per poter sperare di correggiare l'errore.
Tale processo è dispendioso in termini di tempo e non è di facile esecuzione in quanto spesso il processo di backup di un database non viene eseguito dopo ogni inserimento o update di un record; Per poter correggere l'errore sarà quindi necessario calcolare partire dal backup più recente e applicare nuovamente tutte le transformazioni del database, meno l'errore.

L'analisi del motivo dell'errore può inoltre non essere di facile realizzazione con un database relazionale a meno che non siano in uso i meccanismi di CDC: senza una cronologia delle transazioni può essere molto difficile risalire al motivo dell'errore.

Diversamente nel caso dell'utilizzo di Event Sourcing, la gestione e l'analisi di un errore è estremamente semplice. Nel caso in cui l'errore, che sarà sempre un evento, non ha generato un effetto "domino" sul sistema (ovvero l'errore non ha portato all'esecuzione di uan catena di errori), una volta individuato è possibile pubblicare un evento "contrario" alla causa dell'errore in modo tale da cancellare l'apporto dell'errore sul sistema.
Nel caso contrario, ovvero il caso in cui l'errore ha generato una sequenza di errori, per ottenere lo stato corretto del sistema basterà ripetere l'esecuzione di tutti gli eventi del sistema escludendo quello che ha generato l'errore sulla base di dati.

E' bene notare che con ES la gestione degli errori nello stato del sistema è strettamente legata all'atomicità e definizione degli eventi del sistema: una corretta (semplice) definizione degli eventi del sistema porterà ad una cronologia del sistema più chiara e comprensibile.

4.4. Svantaggi

Event sourcing potrebbe non essere utile per una applicazione che richiede frequenti e continue query di richiesta sullo stato del sistema.
Come descritto in precedenza, per ottenere lo stato corrente del sistema è necessario eseguire tutti gli eventi pubblicati sull'event store partendo da uno stato iniziale; Se la nostra applicazione richiede di eseguire molte query di ricerca sullo stato corrente del database sarà quindi necessario calcolare lo stato del sistema *ogni volta che viene eseguita una nuova richiesta* (un esempio di richiesta sullo stato è la ricerca di tutti i record che presentano una particolare caratteristica).

Le modalità per risolvere questo problema sono determinate dal dominio e uso dell'applicazione che utlizza ES, ma generalmente per ovviare a questa debolezza vengono realizzati degli snapshot dello stato dell'applicazione da utilizzare per l'esecuzione delle query di ricerca. La frequenza di generazione ed aggiornamento di questi snapshot è strettamente legata al dominio applicativo dell'applicazione.

5. Apache Kafka e l'ecosistema

Publish/Subscribe è un pattern architetturale utilizzato per la comunicazione asincrona tra diversi processi od oggetti.

In questo schema mittenti e destinatari dialogano tra loro per mezzo di un *broker*, un processo incaricato, da una parte, di ricevere messaggi da dei mittenti e dall'altra di consegnare gli stessi messaggi a dei destinatari.

I destinatari non conoscono i mittenti, ed i mittenti non si interessano di chi sono i destinatari: l'unico compito del mittente è quello di pubblicare dei messaggi sul broker, starà poi al destinario il compito di abbonarsi (dall'inglese *subscribe*) al broker in modo da ricevere tutti i nuovi messaggi.

Questo pattern viene spesso utilizzato quando ci si trova ad avere più processi o servizi che generano delle metriche o dei dati, i quali sono di vitale importanza per altrettanti servizi; Una soluzione alternativa sarebbe creare dei canali dedicati tra produttori e consumatori ma questo non permetterebbe alla struttura di supportare un numero sempre più elevato di servizi od oggetti, ed in un mondo dove è sempre più frequente l'utilizzo di microservizi e il logging di eventi e dati (Big Data) porterebbe ad un debito tecnologico elevato e difficile da correggere.

E' in questo contesto che nasce Apache Kafka, una *streaming platform* basata su un append-only log utilizzato da dei *producer* per pubblicare dei messaggi utilizzati da dei *consumer*.
I messaggi pubblicati vengono persistiti nel tempo, sono leggibili deterministicamente da qualsiasi consumer ed distributi all'interno del sistema secondo particolari logiche in modo da garantire protezione da crash e scalabità del sistema.

Messaggi

L'unità dati fondamentali in Kafka è chiamata *messaggio* o *record*.

Ogni messaggio è suddiviso in *key* (opzionale) e *value* e possono essere di qualsiasi formato; Kafka non impone particolari standard riguardo i formati dei dati utilizzabili all'interno del sistema ma con lo scorrere del tempo *Avro* è diventato lo standard de facto.
Il campo `key`, quando definito, è un byte array utilizzato come metadata per garantire un particolare ordinamento all'interno di un *topic*, un altro elemento fondamentale dell'architettura.

Nonostante Kafka sia una streaming platform, la scrittura e propagazione dei messaggi all'interno della rete non avviene necessariamente per messaggio, invece, piccoli gruppi di messaggi diretti verso la stesso topic vengono raggruppati in *batches*.
La gestione dei messaggi in batch nasce per motivi di efficienza per bilanciare throughput e latenza: a fronte di una latenza più alta per la consegna di un batch, vengono sprecate meno le risorse del sistema che altrimenti si ritroverebbe costretto a gestire l'overhead di conoscegna di un batch per ogni singolo messaggio.

Topic e partizioni

Un *topic* è un elemento utilizzato in Kafka per categorizzare una collezione di messaggi, e consiste in un unico stream di dati.

Un topic è suddiviso in *partizioni*, append-only logs sui quali vengono persisti i messaggi generati dai *producers*.

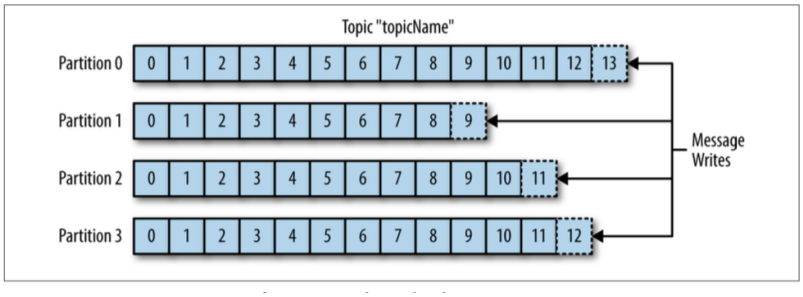

Figure 4: Un topic suddiviso in più partizioni

I messaggi sono inseriti in una partizione da un producer nell'ordine in cui sono stati inviati posizionandoli in fondo al log, non sono modificabili o cancellabili e sono contradistinti da un *offset*, un indice numerico che funziona da timestamp del messaggio. Un consumer legge i messaggi di un topic partendo dalla testa (o da uno specifico offset) del log proseguendo fino alla coda.

L'ordine di scrittura è garantito solo per ogni singola partizione: non è detto che messaggi appartenenti al medesimo topic siano in ordine cronologico se inseriti su partizioni diverse.

Per dare un esempio pratico di topic, supponiamo di utilizzare Kafka per creare uno storage di eventi ricevuti dal front-end di una applicazione: tipici eventi che vengono spesso loggati da un front-end possono essere *i link cliccati in una pagina*, *quali pagine sono state visualizzate in una sessione* oppure *se è stato visualizzato un particolare video embdeed*.
Per ogniuno di questi eventi verrà creato un singolo `topic` per raggruppare tutte le notifiche e dati generati da uno di quei particolari eventi a front-end: ad esempio avremo il topic `views-video-embdeed` sul quale verranno registrati dei semplici `yes` o `no` con magari l'aggiunta di un `timestamp` (l'ora di visualizzazione), in questo modo il topic ci permetterà di calcolare la frequenza di visualizzazione del video.

Producers

Kafka è utilizzata da due tipologie di client: *producers* e *consumers*.

I *producers* hanno il compito di creare messaggi indirizzati a specifici topic (indipendentemente dal numero di partizioni da cui sono formati).
Come illustrato in precedenza, un topic è formato da un numero variabile di partizioni utilizzate come meccanismo di replica e gestione dei messaggi; Alla creazione di un messaggio è possibile indicare al producer su quale partizione andare a scrivere il record specificando l'identificativo di una partizione specifica.

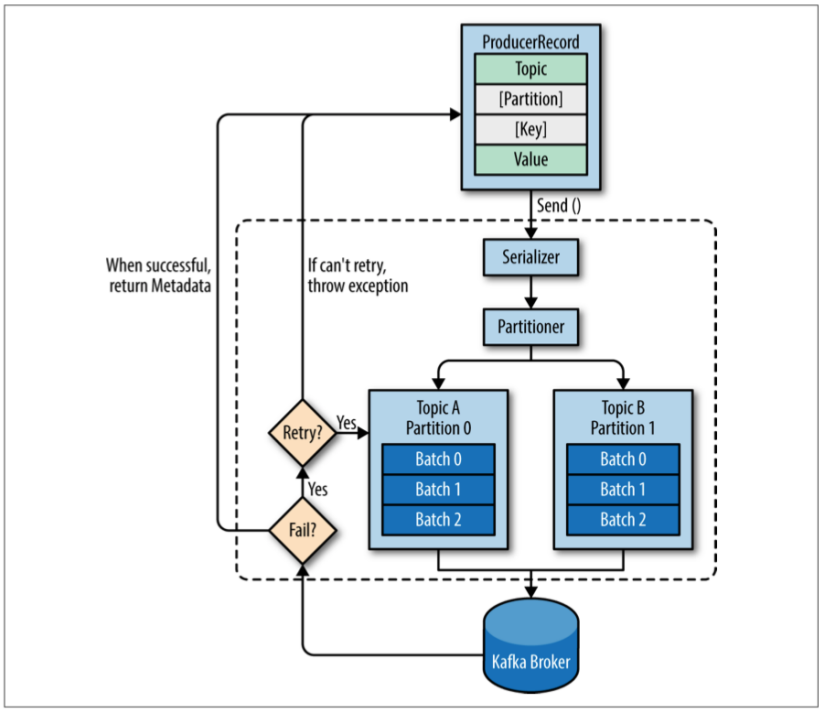

Figure 5: Processo di pubblicazione di un messaggio

Il processo di pubblicazione di un messaggio inizia con la produzione di un `ProducerRecord` il quale deve contenere il topic sul quale vuole pubblicare il messaggio ad una value, ovvero il contenuto del messaggio; Opzionalmente è possibile specificare una chiave o una partizione specifica.
Nella maggior parte dei casi d'uso di Kafka, il producer non si pone mai il problema di decidere su quale partizione andare a scrivere un particolare messaggio ma piuttosto vengono utilizzati dei meccanismi di load-balancing per spartire correttamente i messaggi su tutte le partizioni disponibili presenti nel topic.
Tipici esempi di algoritmi di load-balancing sono il calcolo della partizione in base ad una hash key derivata dall'offset del messaggio oppure utilizzando un

algoritmo round robin, se necessario è presente la possibilità di specificare un *partioneer* creato su misura al caso d'uso.

Una volta creato il `ProducerRecord` il producer serializza chiave e value del messaggio in `ByteArray` in modo da effettivamente trasmetterli sulla rete. I dati sono quindi recapitati ad un partitioner che andrà a deciderà su quale partizione pubblicare il messaggio (solo nel caso in cui la partizione non era già stata specificata nel record). Il record viene quindi aggiunto ad un batch di record e il producer resta in attesa di avere abbastanza record (o alternativamente, in attesa della scadenza di un timeout) prima di inviare il batch di messaggi ad un broker.

Una volta che il broker riceve il batch di messaggi verranno effettuati una serie di controlli per garantire la validità dei messaggi del batch rispetto al topic su cui si sta cercando di pubblicare questi messaggi; In caso positivo il broker invia al producer un `RecordMedatada` con topic, partizione e offset dei messaggi dei pubblicati, altrimenti ritornerà un errore. In caso di errore, il producer può provare a rinviare il batch di messaggi.

Ogni partizione in un cluster ha un *leader* ed un insieme di sue *repliche* distribuite sui vari brokers.
Tutte le pubblicazioni dirette ad una particolare partizione devono prima essere pubblicate sul leader, ed in un secondo momento devo essereno replicate sulle repliche (o followers), questo meccanismo è utilizzato per garantire la durabilità dei dati in un cluster: se uno dei leader muore, viene eletta una delle repliche a nuovo leader della partizione e viene creata una nuova replica.
E' possibile garantire diversi livelli di durabilità a seconda di come viene configurato il producer, favorendo od evitando problemi di scrittura e lettura dei messaggi a scapito del throughput.

Per poter creare un producer sono necessari tre parametri:

- `bootstrap.servers`: lista degli indirizzi (`host:port`) dei brokers del cluster Kafka che vogliamo utilizzare.
- `key.serializer`: nome della classe che verrà utilizzata per serializzare in byte array le chiavi dei record che vogliamo pubblicare con kafka. E' possibile crearne di nuovi implementando `org.apache.kafka.common.serialization.Serializer`.
- `value.serializer`: nome della classe che verrà utilizzata per serializzare in byte array il record da pubblicare.

Un esempio di producer è dato dal seguente codice Scala:

```
import java.util.Properties
import org.apache.kafka.clients.producer.{Callback, RecordMetadata}
import org.apache.kafka.clients.producer.{ProducerRecord, KafkaProducer}
import scala.concurrent.Promise

case class Producer(topic: String){
    val props = new Properties()
    props.put("bootstrap.servers", localhost:9092)
    props.put("key.serializer",
        "org.apache.kafka.common.serialization.StringSerializer")
    props.put("value.serializer",
        "org.apache.kafka.common.serialization.StringSerializer")

    private val producer = new KafkaProducer[String, String](props)

    def send(value: String) = {
      val record = new ProducerRecord[String, String](topic, value)

      try {
        producer.send(record).get()
      } catch {
        case e: Exception => e.printStackTrace
      }
    }

    def sendAsync(value: String) = {
        val record = new ProducerRecord[String, String](topic, value)
        val promise = Promise[(RecordMetadata, Exception)]()

        producer.send(record, new Callback {
            override def onCompletion(metadata: RecordMetadata,
                                      exception: Exception) = {
                promise.success((metadata, exception))
            }
        })
    }
}
```

Creato un oggetto `Properties` con le configurazioni di base, viene creato un nuovo producer capace di pubblicare dei record (di tipo `String`) ad un topic (passato come parametro alla case class) con il metodo `send(value: String)`.

Un producer può inviare record con tre diverse modalità:

- Fire-and-forget: il messaggio viene mandato al server e viene ignorata la possibilità che questo messaggio non venga ricevuto.
- Sincrona: il messaggio viene mandato con un metodo `send()` il quale restituisce un `Future`, viene quindi utilizzato il metodo `.get()` per attendere una risposta dal server per capire se il messaggio è effettivamente stato ricevuto.
- Asincrona: il metodo `send()` restituisce una `callback` che verrà eseguita quando (e se) riceverà una risposta dal broker Kafka.

E' possibile configurare un producer secondo una moltitudine di campi di configurazione, ma sicuramente uno dei più importanti è il valore attribuito al campo `acks` il quale è direttamente collegato alla durabilità dei messaggi e definisce quante repliche devono ricevere il messaggio pubblicato sul leader prima di poter garantire al producer la corretta pubblicazione del messaggio.

Esistono tre possibili valori per `acks`:

- con `acks=0`, il producer non si aspetterà di ricevere un messaggio di conferma da parte del broker. Questo implica che nel caso di un malfunzionamento, il producer non sarà a conoscenza del fallimento ed il messaggio verrà perso. E' l'opzione che garantisce il più alto valore di throughput.
- con `acks=1`, il producer riceverà un messaggio di conferma di pubblicazione del messaggio appena *almeno* una replica confermerà di aver ricevuto il messaggio pubblicato sul leader. Con questa opzione è comunque possibile perdere il messaggio se, a seguito della morte del leader, viene eletto come nuovo leader non la replica che aveva confermato la ricezione del messaggio ma piuttosto una delle repliche che non lo avevano ancora ricevuto.
- con `acks=all`, il producer riceverà conferma della pubblicazione del messaggio solo dopo che tutte le repliche hanno ricevuto il messaggio pubblicato sul leader. Questa opzione garantisce la pubblicazione di un messaggio a scapito di un minor throughput.

Consumers

Un *consumer* è un client Kafka utilizzato per leggere dati da un topic e tipicamente è parte di un *consumer group*.
Un consumer group è definito da uno specifico `group.id` ed un topic che tutti i membri del gruppo hanno in comune; Ogni partizione del topic è letta da un solo consumer del gruppo e più consumer del gruppo non possono leggere dalla stessa partizione.
Supponiamo di avere un topic T1 con quattro partizioni, nel caso in cui il nostro gruppo è formato da un solo consumer C1 sarà solo lui a consumare l'intero topic.

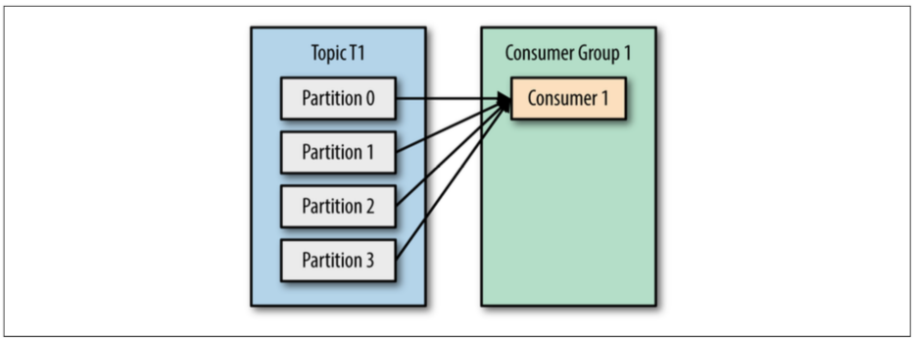

Figure 6: Un consumer con quattro partizioni

Se aggiungiamo un nuovo consumer C2 al gruppo, ogni consumer riceverà dati da solo due partizioni disponibili.

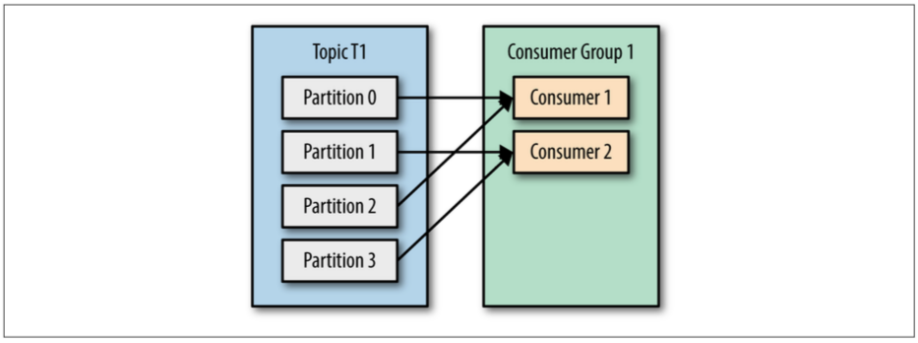

Figure 7: Due consumer consumano le quattro partizioni

Con quattro consumer, ogni consumer leggerà da una sola partizione.

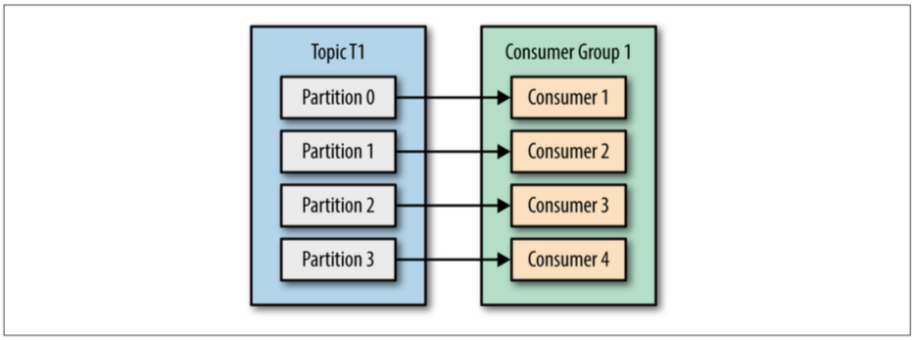

Figure 8: Quattro consumer consumano l'intero topic individualmente

Ed infine, nel caso in cui il gruppo contenga più consumer del numero di partizioni del topic vi saranno sicuramente dei consumer che non leggeranno dal topic.

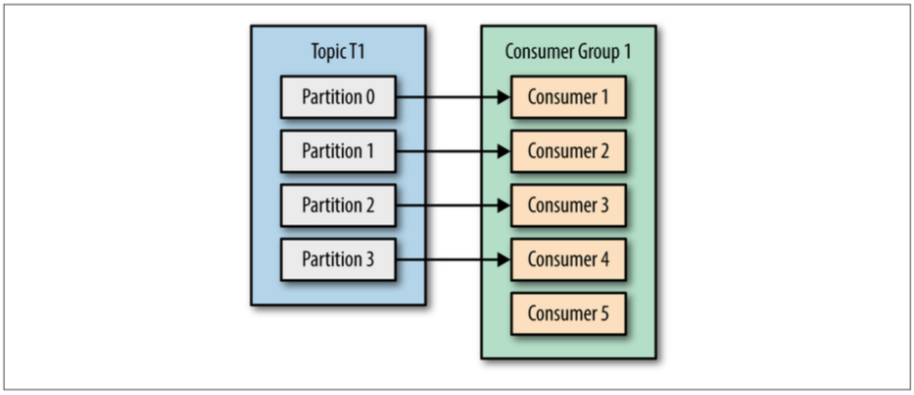

Figure 9: Un numero troppo elevato di consumer rispetto alle partizioni di un topic

Questo meccanismo di bilanciamento dei consumer rispetto alle partizioni di un topic permette di scalare l'architettura orizzontalmente nel caso di topic con grossi moli di dati sensibili che necessitano di essere consumati da applicazioni con operazioni ad alta latenza come la scrittura su un database o l'esecuzione di calcoli: con Kafka, per evitare situazioni a "collo di bottiglia", basta aumentare il numero di partizioni di un topic ed il numero di consumer di quel particolare topic.

Un altro importante vantaggio dell'utilizzo dei consumer group è dato dalla possibilità di ribilanciare il gruppo nel caso di crash, morte o aggiunta di un consumer.

Per *ribilanciare il consumer group* si intende il processo di cambio di proprietà di una partizione da un consumer A ad un consumer B; Questo procedimento, insieme alla possibilità di aggiungere e rimuovere consumer per ogni partizione, è ciò permette a Kafka di scalare la propria architettura su grossi numeri di record e topic. E' importante notare che questa funzionalità non è comunque desiderabile: durante un rebalance il consumer group non può consumare i dati di un topic, comportando quindi un rallentamento nella lettura dei dati.

Un consumer per non risultare morto deve inviare degli *heartbeats* al broker eletto a *cordinator* del consumer group. Questi "segni di vita" sono inviati al broker ogni volta che il consumer tenta di leggere dal topic.

Nel caso in cui il broker non riceva un heartbeat da un consumer entro un particolare lasso di tempo, verrà subito scatenato un ribilanciamento del gruppo a cui appartiene il consumer. Allo stesso modo, nel caso in cui un consumer venga rimosso dal gruppo, sarà lo stesso consumer ad inviare un messaggio di "uscita" dal gruppo al broker che anche in questo caso forzerà un ribilanciamento del consumer group.

Un esempio di consumer sviluppato in Scala è dato dal seguente codice:

```
import java.util.Properties
import org.apache.kafka.clients.consumer.KafkaConsumer

case class Consumer(topic: String){
    val props = new Properties()
    props.put("bootstrap.servers", localhost:9092)
    props.put("key.deserializer",
        "org.apache.kafka.common.serialization.StringSerializer")
    props.put("value.deserializer",
        "org.apache.kafka.common.serialization.StringSerializer")
    props.put("group.id", "example")

    private val consumer = new KafkaConsumer[String, String](props)
    consumer.subscribe(util.Collections.singletonList(topic))

    def readTopic(){
        while(true){
            val records = consumer.poll(100)
            for (r <- records.asScala){
                println(s"${r.offset} ${r.key} ${r.value}")
            }
        }
    }
}
```

Come in precedenza con l'implementazione del producer, prima di poter creare un producer è necessario definire alcune configurazioni minime:

- `bootstrap.servers`: l'indirizzo del broker
- `key.deserializer`: il tipo di classe da utilizzare per deserializzare la chiave dei dati pubblicati su un topic
- `value.deserializer`: il tipo di classe da utilizzare per deserializzare i dati pubblicati su un topic
- `group.id`: l'identificativo del consumer group di cui il consumer fa parte

I consumer leggono i dati di un topic attraverso un meccanismo di "pull", ovvero sono loro stessi a richiedere al broker i dati.
Ogni volta che un consumer decide di leggere da un topic è lui stesso a tenere traccia dell'ultimo messaggio che è stato letto e per tenerne traccia utilizzerà un *offset*: un identificativo corrispondente alla posizione del messaggio nella partizione (il primo messaggio avrà offset pari a 0, l'n-esimo messaggio offset n, etc.).

Dato che sono gli stessi consumer a tenere traccia dell'offset dei messaggi letti e che sono sempre loro a richiedere i dati al broker consumer group diversi possono leggere lo stesso topic senza perdita di messaggi dal topic: i record presenti in un topic sono immutabili.

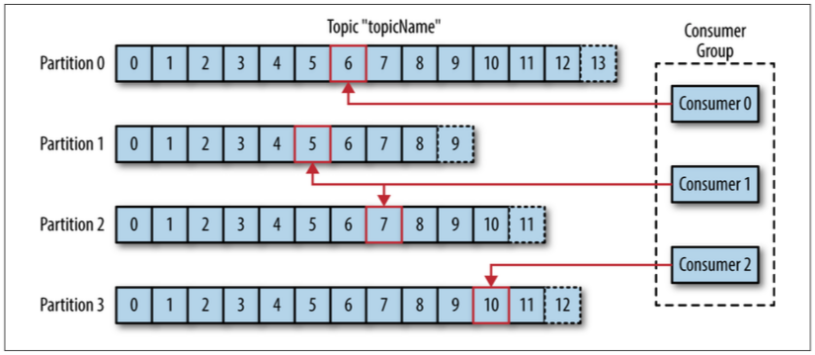

Figure 10: Un topic con più gruppi di consumer

La lettura di una partizione può partire o dal primo messaggio esistente oppure specificando un particolare offset.

Come si può vedere dalla funziona `readTopic()` definita nell'esempio, un consumer utilizza un loop infinito di chiamate a `.poll()` definito su un intervallo di tempo variabile (ovvero il metodo messo a disposizione dall'interfaccia di Kafka) per richiedere al broker tutti i messaggi che quel consumer non ha ancora letto. Per sapere quali messaggi inviare il broker deve riceve dal consumer l'offset dell'ultimo messaggio letto attraverso una operazione di _commit; Gli offset vengono pubblicati dai consumer su di uno speciale topic Kafka chiamato

`__consumer_offset` al quale tutti i broker hanno accesso e a cui fanno riferimento per calcolare quali messaggi inviare.

Il topic `__consumer_offset` è inoltre utilizzato nel caso di ribilanciamento di un gruppo.
A seguito di un ribilanciamento un o più consumer del gruppo possono ricevere un nuovo insieme di partizioni, e per capire da dove iniziare il nuovo processo di lettura, devono interrogare il topic `__consumer_offset`.

Se l'offset pubblicato su `__consumer_offset` è *minore* dell'offset dell'ultimo messaggio che il consumer client ha in memoria, tutti i messaggi compresi tra i due offset verranno riprocessati.

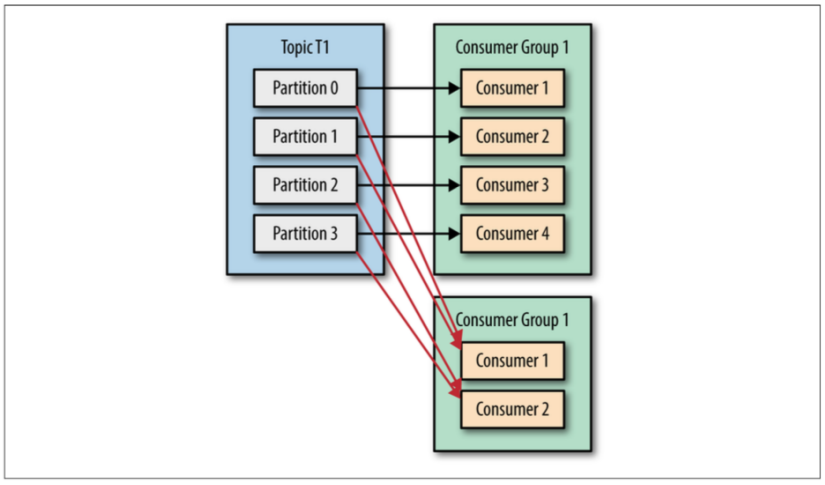

Figure 11: Replay dei messaggi in base all'offset

Se l'offset pubblicato su `__consumer_offset` è *maggiore* dell'offset dell'ultimo messaggio che il consumer client ha in memoria, tutti i messaggi compresi tra i due offset verranno *persi*.

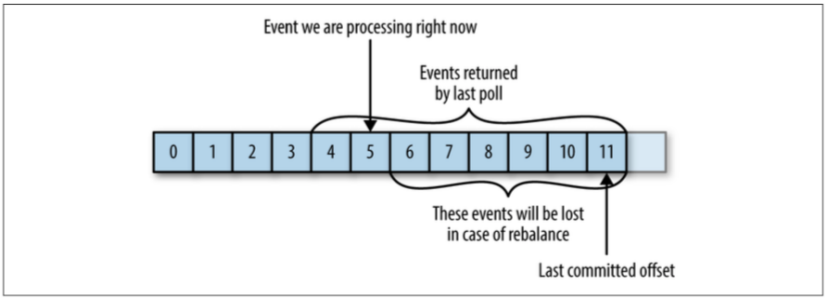

Figure 12: Perdita di messaggi in base all'offset

Per una corretta gestione dei messaggi è quindi di assoluta importanza la capacità di gestire gli offset in modo adeguato alle necessità di ogni progetto, per questo motivo esistono varie possibile configurazione del processo di commit:

- automatic commit: ogni cinque secondi il consumer genera un commit dell'ultimo offset letto con `.poll()`, è la configurazione di default di ogni consumer.
- commit current offset: è lo stesso consumer a decidere quando inviare l'ultimo offset letto attraverso l'uso della funzione `commitSync()`: alla chiamata della funzione viene inviato al broker l'ultimo offset letto da `.poll()` ed il consumer rimane in attesa di un segnale di `acknowledgment` da parte del broker, in caso positivo il consumer continuerà nel processo di lettura, altrimenti verrà lanciata un'eccezione. E' prevista la possibilità di riprovare un numero di volte il processo di commit.
- commit current offset in modalità asincrona: come la modalità precedente ma non bloccante per il consumer, l'offset viene inviato chiamando .commitAsync().

Brokers e clusters

Un *broker* è un server Kafka con svariati compiti quali ricevere, indicizzare e salvare i messaggi inviati dai producers ed inviare i messaggi richiesti dai consumers; Un singolo broker è capace di gestire migliaia di partizioni e millioni di messaggi al secondo.

I broker sono stati creati per lavorare in *clusters* ovvero gruppi di brokers ordinanti secondo una particolare gerarchia.

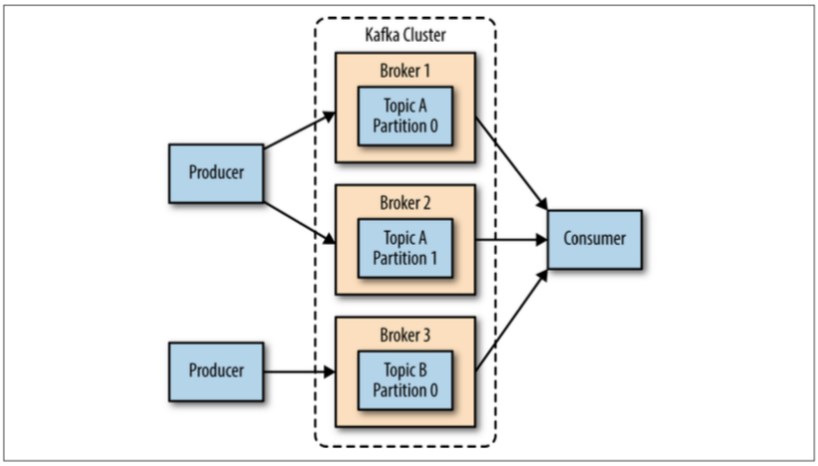

Figure 13: Esempio di cluster

A capo di un cluster troviamo un broker *leader* al quale tutti gli altri broker del cluster devono far riferimento per permette ai meccanismi di replicazioni dei messaggi di funzionare correttamente: una partizione può essere assegnata a più broker, questo permette al cluster di poter gestire fallimenti dei brokers. In ogni cluster un particolare broker viene eletto a *controller*, ovvero un broker con l'incarico di gestire la suddivisione delle partizioni sull'intero cluster e di monitorare l'andamento del cluster.

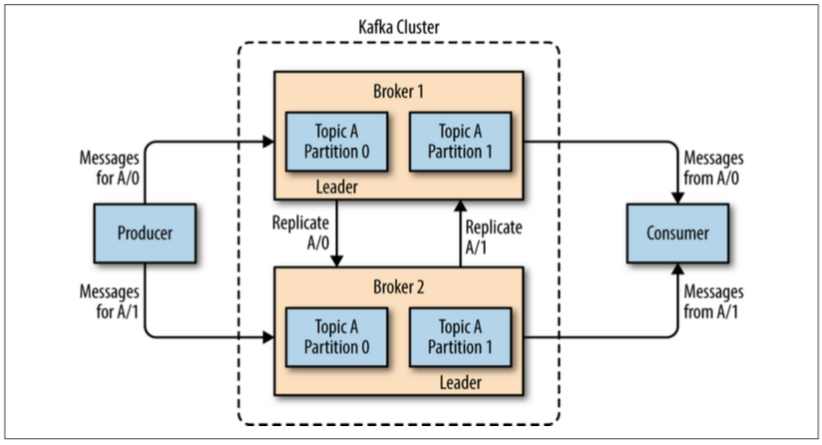

Figure 14: Gestione delle repliche

Una funzionalità importante di Kafka è la possibilità di utilizzare i topic come database di messaggi persistenti.
I messaggi vengono tenuti in memoria per un particolare periodo di tempo oppure in base allo spazio di memoria di occupato, entrambe le opzioni sono configurabili alla creazione di un broker, vi è poi la possibilità di abilitare la *log compaction* ovvero un meccanismo che permette a Kafka di mantenere in memoria *solo gli ultimi messaggi indicizzati su un indicativo specifico*.

Schema

Nonostante i messaggi in Kafka non siano altro che degli array di byte è fortemente consigliato l'uso di *schema* per la gestione e l'uso della struttura dei record.

Lo *schema* è la struttura o organizzazione logicati dei dati contenuti in un topic e nel caso specifico di Kafka, la scelta dei formati disponibili ricade spesso su di un singolo formato: Apache Avro. Esistono altre scelte possibili come Javascript Object Notation (JSON) oppure Extensible Markup Language (XML), ma Avro offre una serie di vantaggi rispetto a questo genere di schemi oltre ad avere alcune implementazioni ad-hoc in Kafka. Avro è diventato nel tempo lo standard per gli schema nelle applicazioni basate su Kafka, gli stessi sviluppatori di Kafka ne promuovono l'uso citando una serie di motivi:

- Avro è mappabile su JSON
- Al contrario di JSON, è possibile scomporre lo schema dei dati dalla definizione dell'oggetto
- E' un linguaggio maturo ben supportato dalla community; Esistono molte librerie che permettono di creare automaticamente oggetti Java o case classes in Scala partendo da uno schema Avro

Apache Avro è un formato per la serializzazione di dati, ogni messaggio Avro si divide in due parti: i *dati* e lo *schema dei dati*.

Un esempio di **schema dei dati** di un record con cinque campi:

```
{
  "type": "record",
  "doc":"This event records the sale of a product",
  "name": "ProductSaleEvent",
  "fields" : [
    {"name":"time", "type":"long", "doc":"The time of the purchase"},
    {"name":"customer_id", "type":"long", "doc":"The customer"},
    {"name":"product_id", "type":"long", "doc":"The product"},
    {"name":"quantity", "type":"int"},
    {"name":"payment",
        "type":{"type":"enum",
            "name":"payment_types",
                "symbols":["cash","mastercard","visa"]},
        "doc":"The method of payment"}
  ]
}
```

Ed un generico record di **dati** definito in base allo schema:

```
{
  "time": 1424849130111,
  "customer_id": 1234,
  "product_id": 5678,
  "quantity": 3,
  "payment_type": "mastercard"
}
```

In Kafka ogni topic posside un particolare schema Avro im modo da :

- definire la struttura dei messaggi pubblicabili nel topic
- permettere a producers e consumers di conoscere quali sono i campi dei messaggi del topic e qual'è il loro tipo
- documentare la tipologia dei messaggi pubblicati nel topic
- evitare la presenza di dati corrotti nel topic

Questo genere di meccanismo per la gestione dei dati diventa di assoluta importanza all'aumentare delle applicazioni che dipendono dall'utilizzo dei dati prodotti e gestiti da una piattaforma Kafka, evitando problemi "effetto Domino" dove un singolo errore in un messaggio potrebbe portare a corrompere un consumer o applicazioni terze che utilizzano i dati forniti dal consumer.

Un formato dei dati consistente come Avro permette di *disacoppiare i formati utilizzati per la lettura e la scrittura dei messaggi*, ovvero viene data la possibilità alle applicazioni che si iscrivono ad un particolare topic di poter utilizzare un nuovo schema di dati compatibile con un vecchio formato, senza dover aggiornare tutte le applicazioni che utilizzano ancora il vecchio formato.

Supponiamo ad esempio di utilizzare un formato del tipo seguente per gestire le informazioni riguardando agli acquirenti di un particolare servizio/piattaforma:

```
{
    "namespace": "customerManagement.avro",
    "type": "record",
    "name": "Customer",
    "fields": [
         {"name": "id", "type": "int"},
         {"name": "name",  "type": "string"},
         {"name": "faxNumber", "type": ["null", "string"], "default": "null"}
    ]
}
```

Una applicazione che vuole utilizzare lo stream di dati di questo topic avrà probabilmente dei metodi come `getId()`, `getName()` e `getFaxNumber()` per leggere i dati del topic; Di nota è il tipo del campo `faxNumber` il quale è esplicitamente possibile che sia `null`, ovvero è lecito aspettarsi che l'applicazione che utilizza questi dati non si romperà nel caso in cui il fax non sia presente nel messaggio.

Supponiamo di aver utilizzato lo schema precedente per genere una importante mole di dati in un topic ma di voler migrare il nostro schema ad un nuovo formato che permetta agli utenti di specificare la loro `email` piuttosto che il loro fax.

```
{
    "namespace": "customerManagement.avro",
    "type": "record",
    "name": "Customer",
    "fields": [
         {"name": "id", "type": "int"},
         {"name": "name",  "type": "string"},
         {"name": "email", "type": ["null", "string"], "default": "null"}
    ]
}
```

Ancora una volta l'applicazione che utilizza questo schema avrà dei metodi `getId()`, `getName()` ma invece di `getFaxNumber()` avrà `getEmail()`; Dopo l'aggiornamento dello schema, i vecchi messaggi presenti nel topic conterranno il campo `faxNumber` mentre i nuovi messaggi avranno il campo `email`.

Dato che il tipo dei campi `faxNumber` e `email` può essere sia `string` che `null`, nessuna delle tue tipologie di applicazioni potrà fallire: la vecchia tipologia di applicazioni semplicemente registrerà i nuovi messaggi come utenti senza un numero di fax, mentre le nuove applicazioni vedranno i vecchi messaggi del topic come utenti senza una email.

Questo genere di *evoluzione* dello schema dei dati di un topic è il motivo centrale dietro all'uso della tecnologia: garantire la robustezza dei dati senza compromettere la leggibilità dello schema o il funzionamento di applicazioni che utilizzano lo stesso topic.
L'evoluzione dello schema è permessa solo secondo determinate regole di compatibilità la cui definizione esula dal contesto di questo tesi ma che possono essere visionate nella documentazione di Apache Avro.

Schema Registry

Uno dei vantaggi di Avro rispetto a JSON è la possibilità di non dover inserire lo schema dei dati "completo" in ogni record permettendo di pubblicare su un topic dei messaggi meno pesanti rispetto a JSON ed è proprio per sfruttare questo vantaggio che nasce lo *Schema Registry*.
Uno *schema registry* è un servizio di gestione degli schema Avro utilizzato da Kafka per servire ed evolvere i metadati di un topic; E' utilizzato dai producers nella fase di scrittura (serializzazione di un messaggio). dai consumer nella fase di lettura (deserializzazione di un messaggio) ed impone delle regole di forma a tutti i client che intendono utilizzare un topic specificato con formato dati Avro.

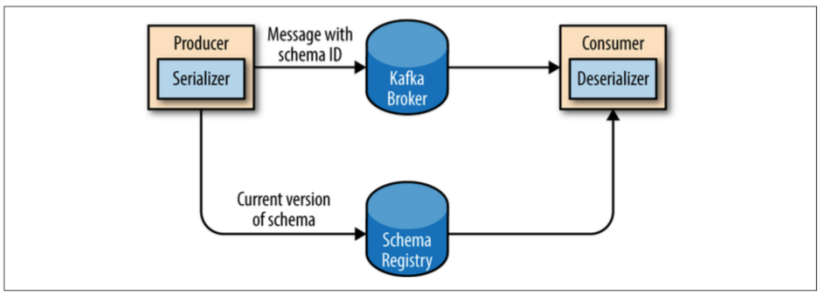

Figure 15: Serializzazione e deserializzazione con schema registry

Supponiamo di avere uno schema registry e di voler produrre dei messaggi su di un topic:

1. Il producer interrogherà il registry per sapere se esiste già uno schema dei dati per un particolare topic inviando la propria copia dello schema, in caso contrario sarà lui stesso a pubblicarlo nel registry.
2. Lo schema registry verifica se lo schema ricevuto dal producer è uguale o una evoluzione compatibile dello schema già presente, in caso negativo verrà alzata un eccezzione e vietata la scrittura al producer.
3. Se lo schema proposto dal producer è valido, nel record Avro verrà inserito un riferimento allo schema del topic.

L'utilizzo dello schema registry da parte di un consumer è speculare a quello di un producer.

Come collegare event sourcing e kafka => perchè kafka è una buona piattaforma per event sourcing

3.2 Kafka Connect

Schema

Source connectors

Sink connectors

Community involment

3.3 Kafka Streams

cos'è streams

KSQL/LSQL

8. Esempi di utilizzo di Kafka

9. Bibliografia

https://docs.confluent.io/current/clients/producer.html
https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying https://www.confluent.io/blog/stream-data-platform-1/ https://martinfowler.com/eaaDev/EventSourcing.html
https://en.wikipedia.org/wiki/Event_store
https://www.confluent.io/blog/data-dichotomy-rethinking-the-way-we-treat-data-and-services/
https://www.confluent.io/blog/build-services-backbone-events/
https://www.confluent.io/blog/apache-kafka-for-service-architectures/
https://www.confluent.io/blog/messaging-single-source-truth/
https://www.confluent.io/blog/building-a-microservices-ecosystem-with-kafka-streams-and-ksql/
https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka
https://qconsf.com/sf2016/system/files/keynotes-slides/etl_is_dead_long-live_streams.pdf <= https://www.youtube.com/watch?v=I32hmY4diFY