Una piattaforma di streaming distribuito: analisi della tecnologia Apache Kafka

Andrea Betti
andrea.betti10@studio.unibo.it

Alessandro Oliva
alessandro.oliva4@studio.unibo.it

Febbraio 2020

La crescita dei servizi che fanno uso di comunicazione real-time ha portato all'introduzione di architetture che permettono la trasmissione di informazioni affidabili in tempi ragionevoli. La complessità di sistemi point-to-point, essendo determinata dalla quantità dei canali di comunicazione tra i nodi, risulterebbe eccessivamente alta in presenza di enormi moli di dati e processi. Per questo motivo, sono state introdotte architetture che fanno uso di broker, componenti che coordinano la comunicazione tra chi invia e riceve messaggi.

Apache Kafka, implementando una logica publish/subscribe, si propone come mezzo di comunicazione tra client, che può essere visto come un event log distribuito dove i messaggi vengono memorizzati su disco, rendendolo un ibrido tra un sistema di messaggistica istantanea e un database.

Gli elementi primari di questa tecnologia sono il cluster, i producer e i consumer. Il cluster contiene diversi broker, che hanno il compito di ricevere messaggi dai producer, e rispondere alle fetch request dei consumer con i messaggi di cui dispone, secondo determinate logiche. Questa architettura permette un'alta versatilità del sistema, che può avere molteplici applicazioni diverse.

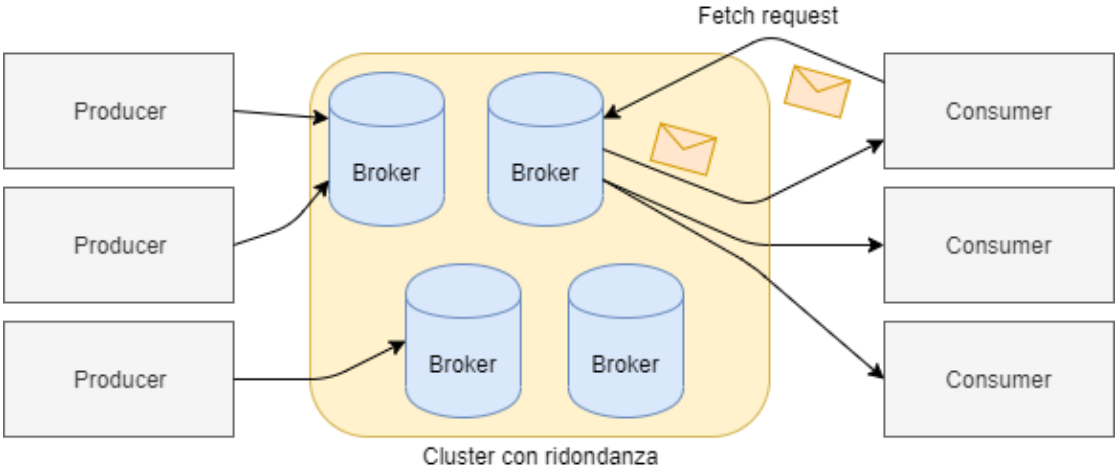

Figura 1: Architettura di Apache Kafka

Indice

1 Terminologia

Publish/subscribe: design pattern di comunicazione nel quale un mittente detto *publisher* non invia messaggi direttamente a un destinatario specifico, ma li pubblica all'interno di un tramite che non è a conoscenza della presenza di destinatari interessati a tali messaggi. Analogamente, un destinatario *receiver* si sottoscrive per ricevere i messaggi, senza sapere se vi sono mittenti che effettivamente li hanno inviati. Il tramite dei sistemi publish/subscribe è detto *broker*, *dispatcher* o in altro modo a seconda del contesto, ed è dove vengono pubblicati tutti i messaggi; separa i publisher dai subscriber, permettendo maggiore flessibilità del tipo di dati che il subscriber vuole ricevere e riducendo il numero di possibili connessioni tra i publisher e i subscriber;

Offset (Kafka): posizione all'interno di una partizione che indica il messaggio da inviare ad uno o più consumer. *Current offset*: indica l'ultimo messaggio inviato, evita l'invio multiplo dello stesso messaggio. *Committed offset*: indica l'ultimo messaggio processato correttamente da un consumer, evita di mandare messaggi

già processati ad un nuovo consumer quando avviene un ribilanciamento delle partizioni;

Stream processing: tecnologia che consente agli utenti di interrogare stream di dati e individuare velocemente informazioni in un periodo di tempo breve dal momento di ricezione del dato, che può variare da pochi millisecondi a minuti. È utilizzato nei contesti dove l'utilità dei dati processati diminuisce molto velocemente col passare del tempo[1];

Log aggregation: è il procedimento con cui log provenienti da diverse sorgenti vengono raccolti in un unico spazio. In un sistema distribuito, avere la possibilità di accedere a informazioni mediante una singola interfaccia riduce considerevolmente la complessità delle operazioni[2]. Questa nozione non è da confondere con i log secondo Kafka, che invece si riferiscono ai messaggi contenuti nei broker tramite ritenzione;

Immagine Kafka: diverse implementazioni di Kafka per Docker. Sviluppate da diversi vendor o comunità, ognuna con diverse feature esclusive. Nel progetto sono state usate le immagini *Bitnami* e *Confluent*;

Secure Sockets Layer (SSL): protocollo crittografico di presentazione che permette una comunicazione sicura su reti TCP/IP tramite autenticazione, integrità dei dati e confidenzialità;

Simple Authentication and Security Layer (SASL): framework per autenticazione e sicurezza nei protocolli Internet. Disaccoppia i meccanismi di autenticazione dai protocolli applicativi, consentendo di utilizzare qualunque meccanismo di autenticazione supportato da SASL su qualunque protocollo applicativo che utilizza SASL. Sono forniti anche layer di sicurezza che offrono servizi di integrità e confidenzialità dei dati;

Plaintext: protocollo di (non) sicurezza di Kafka secondo il quale le comunicazioni avvengono completamente in chiaro. Solitamente usato solo per casi di test o per connessioni interne su cui girano dati di bassa importanza.

2 Apache Kafka

Apache Kafka è un sistema open source di comunicazione basato su messaggi di tipo publish/subscribe, spesso descritto come un registro di eventi distribuiti dove tutti i nuovi record vengono aggiunti ad una coda. La differenza principale tra Kafka e altri servizi di comunicazione tramite messaggi è la politica di ritenzione, ovvero la conservazione dei messaggi su disco per un determinato periodo di tempo, che rende Kafka un ibrido tra un sistema di comunicazione e un database.

I concetti principali riguardanti la logica di Kafka sono la produzione di messaggi da parte dei *producer*, indirizzati a uno o più topic, e il consumo dei messaggi da parte

dei *consumer*, i quali mantengono in memoria degli *offset* per identificare il successivo messaggio di cui sono in attesa. I producer e i consumer rappresentano quindi, rispettivamente, le figure di publisher e subscriber.

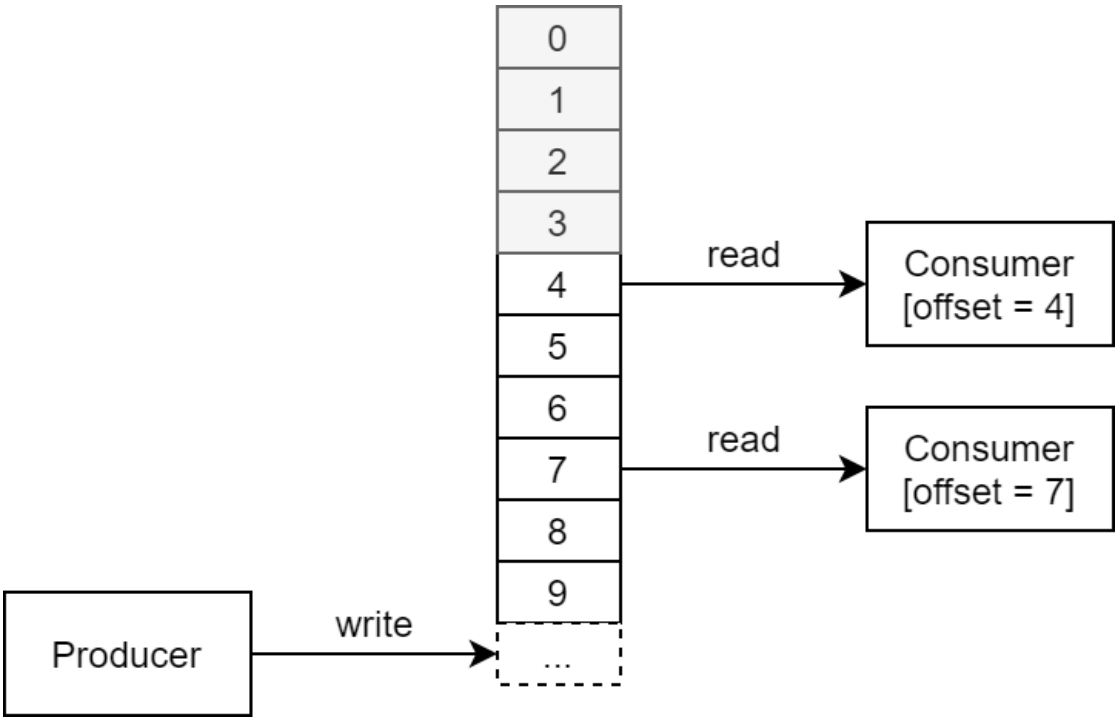

Figura 2: Rappresentazione delle operazioni svolte dai producer e dai consumer

Kafka mira a fornire una piattaforma affidabile e ad alta efficienza per gestire stream di dati in tempo reale e costruire pipeline dati. Può essere inoltre usato come sistema di event sourcing, ovvero per raccogliere eventi di varia natura e renderli disponibili a sistemi esterni.

È stato originariamente sviluppato da LinkedIn e reso open source nel 2011[3]. Viene attualmente utilizzato da diverse compagnie importanti, che arrivano a produrre 1.000.000.000 messaggi al giorno circa. Kafka viene usato principalmente per produrre flussi di dati in tempo reale e collezionare big data[4].

Tra le principali applicazioni industriali vi sono:

Twitter: i tweet degli utenti sono inviati come messaggi Kafka;

LinkedIn: per stream dei dati e metriche operazionali;

Netflix: real-time monitoring e processing;

Mozilla: per collezionare informazioni su performance e uso dei dati su browser;

Oracle: l'OSB (Oracle Service Bus) utilizza pipeline di dati staged (area intermedia per processi Extract, Transform, Load) che implementano nativamente il supporto alla connettività mediante Kafka.

2.1 Architettura

L'architettura standard di un sistema Apache Kafka è composta da producer, consumer e un cluster. Quest'ultimo consiste in un insieme di broker, che al fine di garantire un adeguato livello di ridondanza corrisponde esattamente a 3 broker.

Un broker è responsabile della ricezione dei messaggi inviati dai producer, dell'assegnamento di *offset* e della memorizzazione dei messaggi su disco, oltre che della risposta alle richieste di fetch dei consumer e ai relativi invii di messaggi.

In Kafka, l'invio di un messaggio a un broker è legato anche a uno specifico *topic*. I topic consentono di suddividere in categorie i messaggi memorizzati, e possono essere ulteriormente suddivisi in un numero specificato di *partizioni*.

Ciascuna partizione è una sequenza ordinata e immutabile di record, aggiunti all'estremo della sequenza, ed è indipendente dalle altre partizioni che fanno parte dello stesso topic. La possibilità di suddividere il topic in più partizioni migliora la scalabilità del sistema, in quanto permette l'accesso a partizioni diverse dello stesso topic da parte di più consumer. È possibile parlare di scalabilità orizzontale quando le partizioni di uno stesso topic vengono distribuite fisicamente su diversi host.

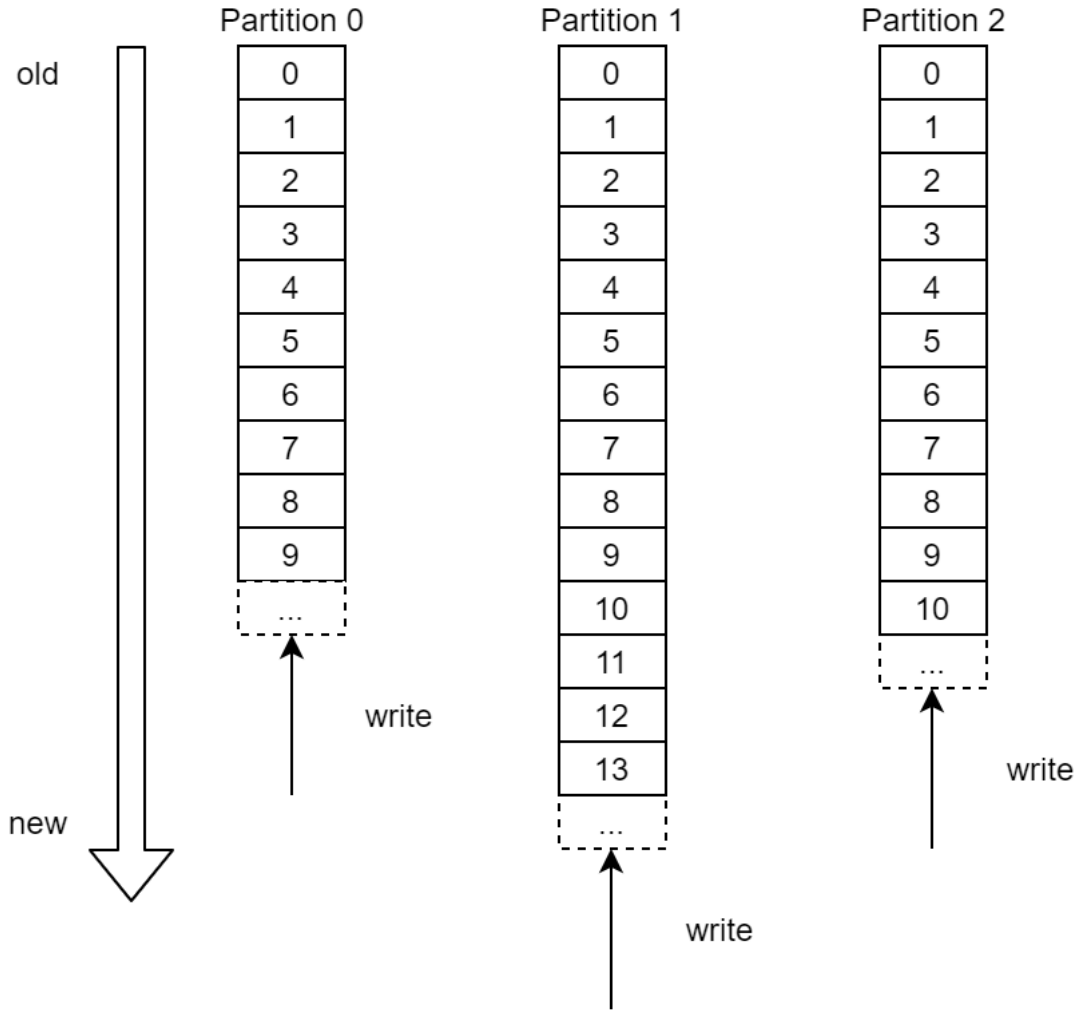

Figura 3: Rappresentazione schematica di un topic su più partizioni

Un messaggio Kafka consiste in una singola unità di dati, un array di byte serializzati che verranno poi convertiti in oggetti del tipo desiderato dalle applicazioni a messaggio ricevuto. Per ridurre la latenza di rete e incrementare il throughput del sistema, è possibile inviare messaggi in *batch*, collezioni comprimibili di messaggi indirizzati allo stesso topic e alla stessa partizione.

Avendo a disposizione i dati solo sotto forma di sequenze di 0 e 1, Kafka non ha la possibilità di verificare il tipo e il contenuto del dato che essi rappresentano, e può quindi portare a errori di decodifica da parte del consumer nel caso riceva un dato in un formato che non si aspetta.

Per l'utilizzo di formati complessi, la piattaforma Confluent fornisce l'utilizzo degli *schema registry*, interfacce con cui i consumer e i producer possono comunicare per verificare la correttezza del formato dei messaggi consultando gli *schema* memorizzati.[5]

I producer possono assegnare ai messaggi una chiave opzionale per specificare la partizione del topic di destinazione. Se non viene specificata, la scelta della partizione verrà effettuata seguendo una politica round-robin, alternando in modo regolare le partizioni a cui i nuovi messaggi verranno inviati.

Per conservare l'ordine temporale di una sequenza di messaggi, è importante fare in modo che questi vengano inseriti nella stessa partizione.

I consumer si inseriscono autonomamente in un *consumer group*, e ogni record pubblicato in un topic viene spedito a un solo consumer per ogni gruppo sottoscritto. Se tutte le istanze di consumer appartengono allo stesso gruppo, i record verranno distribuiti tra i consumer in modo bilanciato; al contrario, nel caso tutti i consumer appartengano a gruppi diversi, a ciascuno di loro verranno trasmessi tutti i record. Per permettere il consumo di uno stesso messaggio a consumer diversi, in modo da permettere alle relative applicazioni di processare i dati in modo indipendente tra loro, è necessario che questi non appartengano allo stesso consumer group.

In Kafka, il consumo dei messaggi viene implementato ripartendo le partizioni del topic tra le istanze di consumer, in modo che ciascuno sia il consumer esclusivo di una percentuale equa di partizioni. All'inserimento di nuovi consumer in un gruppo, verranno riassegnate alcune partizioni dagli altri membri; analogamente, se un'istanza non risulta più essere disponibile, le partizioni a essa assegnate verranno distribuite agli altri membri del consumer group[6].

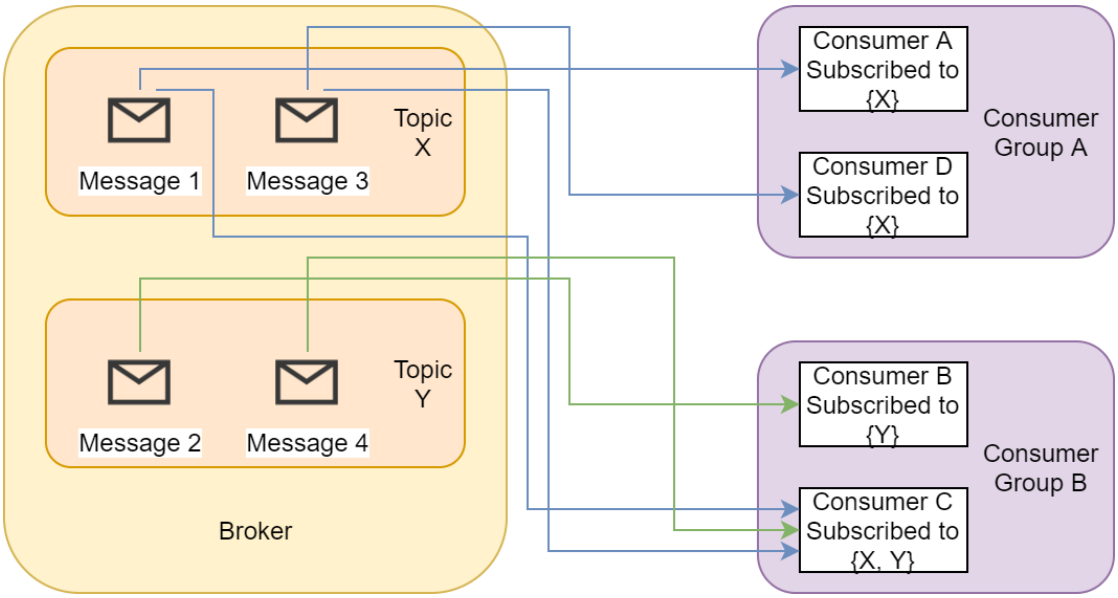

Figura 4: Rappresentazione della distribuzione di messaggi in consumer group

In un cluster di broker, uno di essi ha il ruolo di *cluster controller*, responsabile dell'assegnamento di partizioni ai broker e del monitoraggio di eventuali guasti.

Una partizione può essere posseduta solamente da un broker, il quale è il *leader* della partizione, e replicata negli altri componenti del cluster, detti *follower*. Il leader gestisce tutte le richieste di lettura e scrittura della partizione, mentre i follower replicano passivamente le operazioni da lui effettuate. Nel caso il leader si imbatta in un guasto, il ruolo di leader verrà assegnato a uno dei follower. In un cluster bilanciato, ciascun broker è il leader di una percentuale equa delle partizioni totali[7].

Una delle caratteristiche chiavi di Kafka è la *ritenzione*, che permette la conservazione durevole di messaggi per un periodo di tempo configurabile, pari a 7 giorni di default. I messaggi più vecchi verranno eliminati anche al raggiungimento di una determinata dimensione di dati raccolti per liberare spazio su disco. Ciascun topic può adottare politiche diverse di ritenzione in base alla tipologia di messaggi a esso associati.

L'utilizzo di una politica di ritenzione distingue Kafka da altri sistemi di messaggistica, che per gestire l'eliminazione di messaggi non più necessari conservano le informazioni relative nei broker, portando a maggiore complessità e overhead.

I consumer, ripristinando l'offset del messaggio atteso a valori precedenti, hanno quindi la possibilità di consumare più di una volta gli stessi messaggi, permettendo la gestione di errori nelle applicazioni[8].

L'affidabilità è spesso espressa in termini di garanzie, determinati comportamenti di un sistema che devono essere preservati in diverse circostanze, la cui comprensione è critica per chiunque voglia costruire applicazioni affidabili utilizzando Kafka. Le più comuni sono:

- in una partizione, Kafka garantisce il mantenimento dell'ordine dei messaggi inviati dallo stesso producer. Un consumer leggerà i messaggi di una partizione da quello inviato più recentemente, con offset maggiore, a quello più vecchio, con offset minore;
- una volta che i messaggi vengono scritti sia nel leader che replicati in altri broker, non verranno persi finché almeno una replica è disponibile e la politica di ritenzione è valida;
- la semantica per la ricezione di messaggi utilizzata da Kafka di default è at-least-once, attraverso la quale il producer può inviare il messaggio più di una volta se non riceve un acknowledgement di ricezione da parte del consumer, oppure il consumer può richiedere più volte lo stesso messaggio che non è stato spedito.

Un cluster Kafka ha la possibilità di applicare dei limiti (*quota*) sulle richieste dei client per controllare le risorse che ha a disposizione, relativi alla larghezza di banda di rete e alle percentuali di utilizzo della CPU. In questo modo è possibile prevenire la produzione e il consumo dei dati a un tasso eccessivo, capace di portare alla monopolizzazione delle risorse del broker, saturazione della rete e in generale danneggiare altri client e i broker stessi. I limiti sono applicabili a singoli utenti o gruppi di client; tutte le connessioni di un gruppo condividono i limiti configurati per quest'ultimo.

Un'altra caratteristica di Kafka è la compattazione dei registri (*log compaction*), che assicura la conservazione di almeno l'ultimo valore noto per ciascuna chiave dei messaggi inseriti in una partizione di un topic. È usata in scenari dove vengono effettuati caricamenti della cache dopo il riavvio dell'applicazione o dove è necessario effettuare ripristini a stati precedenti a seguito di errori. Un topic compattato può essere utilizzato dai consumer per ripristinare il loro stato.

Un registro compattato è composto da una testa, dove tutti i messaggi vengono conservati in ordine sequenziale, e una coda, dove vengono mantenuti solo i messaggi più recenti per ciascuna chiave. Questi mantengono il loro offset originale assegnato al momento della scrittura, e letture sugli offset corrispondenti a messaggi compattati restituiranno un insieme di messaggi correlati al messaggio con l'offset successivo presente nel registro.

I messaggi con una chiave e un valore nullo corrispondono a una cancellazione dal registro. Tutti i messaggi precedenti con la stessa chiave e offset minore verranno eliminati al raggiungimento di un punto di ritenzione.[7]

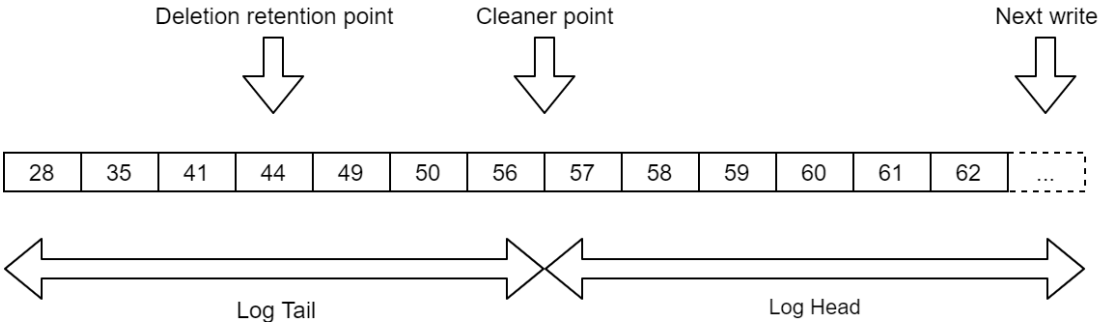

Figura 5: Esempio di partizione compattata

2.2 Configurazione

Kafka è configurato mediante coppie chiave-valore dichiarate in formato `.properties` da file o a livello di codice[7].

I principali parametri relativi ai broker sono:

- `zookeeper.connect`: specifica la stringa di connessione al servizio ZooKeeper nel formato `hostname:port`, o `hostname1:port1, hostname2:port2...` per permettere la connessione ad altri nodi in caso di indisponibilità;
- `broker.id`: numero identificativo del broker;
- `log.dirs`: l'elenco di directory usate per la memorizzazione dei registri;
- `auto.create.topics.enable`: valore booleano relativo alla creazione automatica dei topic sul server. Di default è `true`;

- `min.insync.replicas`: se il producer ha il parametro `acks=all`, questo valore specifica il numero minimo di broker con repliche che devono inviare un messaggio di acknowledgement al producer per la scrittura del record al fine di considerare riuscita la produzione;
- `zookeeper.session.timeout.ms`: definisce il tempo di attesa di una risposta da ZooKeeper dopo un'operazione di lettura o scrittura prima della chiusura di una sessione. Di default è pari a `6000` millisecondi;
- `zookeeper.connection.timeout.ms`: definisce il tempo massimo di attesa per stabilire una connessione con ZooKeeper. Di default è pari a `6000` millisecondi;
- `zookeeper.sync.time.ms`: definisce il tempo di sincronizzazione tra un follower ZooKeeper e il suo leader. Di default è pari a `2000` millisecondi.

I principali parametri relativi ai producer sono:

- `key.serializer` e `value.serializer`: classi di serializzazione della coppia chiave-valore che implementano l'interfaccia Kafka `Serializer`;
- `acks`: la durabilità dei record è in parte controllata dal numero di acknowledgement che il producer riceve dal leader per considerare completa una richiesta di invio. Questo parametro può assumere i seguenti valori[7]:
  - `acks=0` (semantica at-most-once): dopo l'invio del messaggio, il producer non si aspetta di ricevere acknowledgement, portando a perdite di dati nel caso il leader non sia disponibile;
  - `acks=1` (semantica at-least-once): il broker leader invierà un messaggio di acknowledgement al producer dopo avere scritto il record sul suo registro locale, ma senza verificare che sia stato scritto anche nei follower. Se il leader fallisce subito dopo avere inviato l'ack, il record potrebbe essere andato perduto se non è stato ancora replicato in altri broker;
  - `acks=all` (semantica exactly-once): il messaggio di acknowledgement verrà inviato solamente dopo avere verificato che il record sia stato ricevuto sia dal leader che da tutti i follower, garantendo maggiore durabilità rispetto alle configurazioni precedenti;
- `bootstrap.servers`: un elenco di coppie `host:port` usate per stabilire la connessione iniziale al cluster Kafka. Non rappresenta l'elenco completo dei server utilizzati dal client;
- `buffer.memory`: i byte di memoria utilizzabili dal producer per mantenere temporaneamente nel buffer i record da inviare al cluster Kafka. Se i record vengono inviati più velocemente di quanto possono essere consegnati al server, il producer si bloccherà per un tempo pari a `max.block.ms`.

I principali parametri relativi ai consumer sono:

- `key.deserializer` e `value.deserializer`: classi di deserializzazione della coppia chiave-valore che implementano l'interfaccia Kafka `Deserializer`;
- `bootstrap.servers`: un elenco di coppie `host:port` usate per stabilire la connessione iniziale al cluster Kafka. Non rappresenta l'elenco completo dei server utilizzati dal client;
- `group.id`: identifica il consumer group di appartenenza.

2.3 Operazioni

Le operazioni più comuni eseguibili sui cluster Kafka sono le seguenti:[7]

- *creazione di topic*: è possibile creare topic manualmente o automaticamente alla produzione di un messaggio verso un topic ancora non esistente. Kafka mette a disposizione lo script `kafka-topics.sh`, da utilizzare con la seguente sintassi:

```
1 bin/kafka-topics.sh --bootstrap-server broker_host:port --create --
      topic my_topic_name --partitions 20 --replication-factor 3 --
      config [name]=[value]
```

  Il parametro `replication-factor` specifica il numero di broker che replicheranno ciascun messaggio che verrà scritto; se è pari a 3 e 2 broker su 3 falliscono, sarà possibile accedere ai dati dal broker restante. Ciascun topic verrà suddiviso nel numero di partizioni indicate da `partitions`;

- *rimozione di topic*: sempre utilizzando `kafka-topics.sh`, è possibile rimuovere manualmente un topic specificando il suo nome nel parametro `delete`;

```
1 bin/kafka-topics.sh --bootstrap-server broker_host:port --delete --
      topic my_topic_name
```

- *modifica di topic*: è possibile aggiungere il numero di partizioni di un topic, lasciando però inalterato il partizionamento dei dati presenti:

```
1 bin/kafka-topics.sh --bootstrap-server broker_host:port --alter --
      topic my_topic_name --partitions 40
```

  È possibile aggiungere e rimuovere configurazioni di un topic attraverso lo script `kafka-configs.sh`:

```
1 bin/kafka-configs.sh --bootstrap-server broker_host:port --entity-
      type topics --entity-name my_topic_name --alter --add-config [
      name]=[value]
2
3 bin/kafka-configs.sh --bootstrap-server broker_host:port --entity-
      type topics --entity-name my_topic_name --alter --delete-config [
      name]
```

- *chiusura dei broker*: il cluster Kafka individua automaticamente qualsiasi chiusura o guasto dei broker, procedendo eventualmente all'elezione di nuovi leader. Quando un server viene chiuso manualmente, salva preventivamente i dati del registro su disco per evitare la fase di ripristino al suo riavvio e, nel caso sia impostato il parametro `controlled.shutdown.enable=true`, trasferisce i dati delle partizioni di cui è leader agli altri broker prima di arrestarsi;

- *bilanciamento dei leader*: quando un broker si arresta, su comando o a seguito di un errore, trasferirà il ruolo di leader su altri broker aventi le partizioni relative come repliche. Ciò significa che, al suo riavvio, di default il broker sarà follower delle sue partizioni originarie.

  Kafka permette di specificare una lista di preferenze del leader per ciascuna partizione, assegnando il ruolo al primo broker attivo seguendo l'ordine della lista. Il riassegnamento dei ruoli viene effettuato col comando:

```
1 bin/kafka-preferred-replica-election.sh --zookeeper zk_host:port/
      chroot
```

  È possibile configurare Kafka per compiere la stessa operazione automaticamente impostando il parametro `auto.leader.rebalance.enable=true`;

- *bilanciamento delle repliche tra rack*: per rack si intende un insieme di macchine fisicamente vicine e connesse allo stesso commutatore di rete. È possibile associare ai broker il rack di appartenza utilizzando il parametro `broker.rack` per distribuire equamente tra diversi rack repliche della stessa partizione e prevenire il fallimento di tutti i broker appartenenti a uno stesso rack;

- *mirroring dati tra cluster*: attraverso lo script `mirror-maker.sh`, Kafka permette di replicare dati tra cluster Kafka diversi e indipendenti leggendo i dati da un cluster sorgente e scrivendoli in un cluster destinazione utilizzando topic con lo stesso nome. Il mirroring non è pensato come meccanismo di fault-tolerance, in quanto la indipendenza tra i cluster porta alla differenza della posizione del consumer, ma unicamente come operazione di clonazione;

```
1 bin/kafka-mirror-maker.sh --consumer.config consumer.properties --
      producer.config producer.properties --whitelist my-topic
```

- *controllo della posizione del consumer*: lo script `kafka-consumer-groups.sh` permette di monitorare gli offset successivi che i consumer di un consumer group si aspettano e la loro distanza dall'ultimo offset memorizzato nel registro. Nello specifico, stampa i seguenti dati: topic, partizione, offset corrente, offset a fine registro, distanza tra i due offset, ID del consumer, host, ID del client;

```
1 bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --
      describe --group my-group
```

- *gestione dei consumer group*: lo script `kafka-consumer-groups.sh` può essere usato per visualizzare informazioni o eliminare consumer group. Un consumer group può essere eliminato sia automaticamente, quando l'ultimo messaggio consegnato a esso associato cessa di esistere, che manualmente, nel caso non ci siano membri attivi;

```
1 bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --
      list
2
3 bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --
      delete --group my-group --group my-other-group
```

- *espansione del cluster*: per assegnare partizioni esistenti a server appena creati, è necessario intervenire manualmente attraverso lo script `kafka-reassign-partitions.sh`. Questo dispone di tre modalità: `generate`, per la creazione di un piano di riassegnamento delle partizioni date una lista di topic e una lista di broker, `execute`, per l'applicazione di un piano passato come file JSON o generato dal tool, e `verify`, che verifica lo stato di riassegnamento per tutte le partizioni utilizzate nell'ultimo `execute`;

```
1 bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --topics
      -to-move-json-file topics-to-move.json --broker-list "5,6" --
      generate
2
3 bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --
      reassignment-json-file expand-cluster-reassignment.json --execute
4
5 bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --
      reassignment-json-file expand-cluster-reassignment.json --verify
```

- *aumento del replication factor*: è possibile aumentare il replication factor di una partizione esistente utilizzando lo script `kafka-reassign-partitions.sh` e passando come argomento un file JSON contenente le informazioni relative ai nuovi assegnamenti, con il flag `execute`.

```
1 bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --
      reassignment-json-file increase-replication-factor.json --execute
```

  Per controllare lo stato del riassegnamento delle partizioni senza alterarlo è sufficiente sostituire il flag `execute` con `verify`;

```
1 bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --
      reassignment-json-file increase-replication-factor.json --verify
```

- *limitazione dell'uso di banda durante la migrazione dei dati*: Kafka permette la definizione di un punto di strozzatura sul traffico di trasferimento delle repliche tra broker, riducendo l'impatto che operazioni con grandi moli di dati possono avere sugli utenti. Il parametro è impostato come attributo (`throttle`) dello script `kafka-reassign-partitions.sh` in MB/s:

```
bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --
    execute --reassignment-json-file bigger-cluster.json -throttle
    50000000
```

2.4 Kafka Streams

Kafka Streams è una libreria per l'elaborazione e l'analisi dei dati conservati nei cluster Kafka, utilizzabile in applicazioni Java e Scala. È costruita sulle primitive principali fornite da Kafka, sfruttando sia le API dei producer per prendere in ingresso i dati nei cluster che quelle dei consumer per inviare messaggi. Può essere usata per costruire applicazioni e microservizi distribuiti altamente scalabili e fault-tolerant. Dal rilascio 0.11.0.0, viene supportata una semantica exactly-once, per garantire che ciascun record venga processato solamente una volta anche a fronte di guasti incontrati nel corso dell'elaborazione, sia da parte dei client che dai broker.

La più importante astrazione della libreria è lo stream, un flusso continuo di record in tempo reale e senza limiti, dove un record corrisponde a una coppia chiave-valore. L'API Kafka Streams è un sistema atto a trasformare ed arricchire questi stream, in cui ciascun record viene processato in modo indipendente senza ricorrere a tecniche di batching, con latenza in millisecondi. Le operazioni di stream processing avvengono completamente all'interno dell'applicazione senza ricorrere a cluster di elaborazione esterni, rendendo quindi possibile il deploy in ambienti di sviluppo e sistemi operativi diversi senza complicazioni. Per aumentare la potenza computazionale, è possibile distribuire il lavoro tra più applicazioni.

La logica computazionale di Kafka Streams è definita da una *topologia* di processori, nodi che rappresentano le iterazioni di trasformazione dei dati degli stream ricevuti dai processori upstream e della produzione di record di output indirizzati a processori downstream. I processori che producono e inviano nuovi stream di input consumando record da uno o più topic Kafka, senza trasformare stream ereditati da nodi upstream, sono detti *source processor*. I processori che inviano i record ricevuti in topic Kafka senza avere nodi downstream sono invece detti *sink processor*.

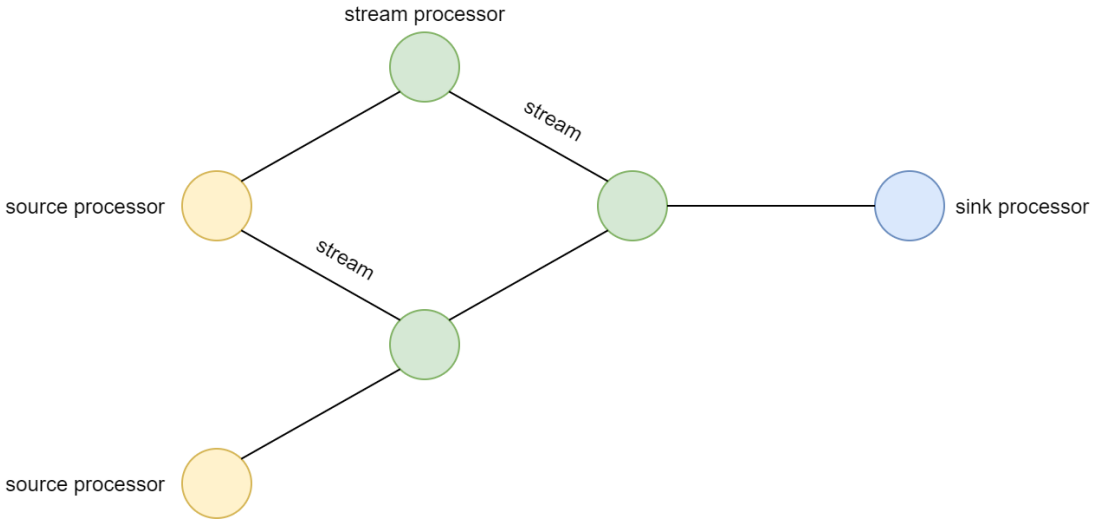

Figura 6: Rappresentazione di una topologia di processori in Kafka Streams

Sono supportate operazioni di trasformazione stateless, come `filter` e `map`, e operazioni stateful, come `join` e `aggregation`; queste ultime fanno uso di operazioni di windowing, atte a raggruppare record con la stessa chiave nelle cosiddette *finestre*.

È possibile specificare un periodo di ritenzione per la finestra, volto a controllare il tempo di attesa per i record arrivati in ritardo o fuori sequenza, e allo scadere del quale il record verrà scartato. La semantica del tempo di ritardo può essere relativa al momento in cui il record viene creato (*event time*) o al momento in cui viene memorizzato in una partizione di un topic (*ingestion time*); non può essere espresso in termini di tempo di elaborazione da parte dell'applicazione di stream processing (*processing time*), che quindi sarebbe nullo.

Dalla versione di Kafka 0.10.x in poi, vengono generati e allegati ai messaggi dei timestamp, che in base alla configurazione di Kafka possono essere relativi all'event time o all'ingestion time. La rispettiva impostazione di configurazione Kafka può essere specificata al livello del broker o per topic, e la semantica del tempo utilizzata dall'applicazione dipende da quella dei timestamp raccolti.

Per quanto riguarda i record arrivati fuori sequenza, vi sono due possibili cause:

- all'interno di un topic, l'incremento del timestamp di un record potrebbe essere superiore a quello dell'offset. In quanto Kafka Streams segue l'ordine degli offset, si potrebbe verificare il caso in cui un record con timestamp maggiore viene elaborato prima di un record con offset maggiore ma timestamp minore, proveniente dalla stessa partizione;

- a un'applicazione che prende record da partizioni diverse, e configurata a scegliere la partizione con il timestamp minore senza aspettare che contegano tutte già dei dati, potrebbe essere restituito un messaggio con timestamp minore di almeno uno già ricevuto e proveniente da un'altra partizione.

Nelle operazioni stateful, i dati fuori sequenza possono portare all'incorrettezza della logica di elaborazione. Per gestirli correttamente, è necessario fare in modo che le applicazioni attendano un tempo ulteriore e, durante il tempo di attesa, consultino i loro *state store* per prendere decisioni sul da farsi. Per state store di un'applicazione Kafka Streams si intende un'unità di memoria in cui è possibile immagazzinare e leggere dati, che possono consistere in coppie chiave-valore, hashmap o altre strutture dati[9].

2.4.1 Architettura

Kafka Streams fa appoggio sulle librerie relative ai consumer e ai producer per potenziare le capacità di Kafka, offrendo parallelismo dei dati, coordinazione distribuita e fault-tolerance.

Gli elementi chiave del modello di Kafka Streams sono:

- *partizioni di stream*, sequenze ordinate di record associate univocamente a partizioni di topic Kafka;
- *record di dati*, corrispondenti a messaggi dei topic Kafka:
- *chiavi* dei record, che determinano il partizionamento dei dati.

La topologia di un'applicazione è scalata mediante la suddivisione in *stream task*. Per ciascuno stream, viene creato un numero fissato di task proporzionale al numero di partizioni da cui l'applicazione può prendere i dati dei record. I task possono instanziare le loro topologie di processori, mantenendo un buffer per ciascuna delle loro partizioni assegnate e processando da essi i dati in input un record alla volta. I task possono quindi essere processati in modo autonomo e in parallelo senza necessità di interventi manuali.

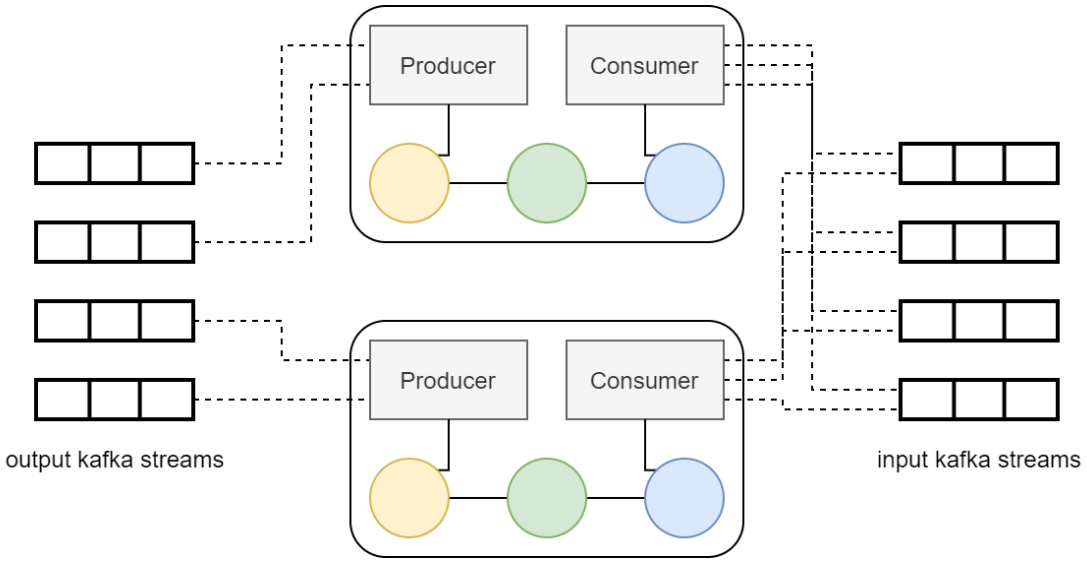

Figura 7: Architettura di un'applicazione che fa uso di Kafka Streams

Il livello di parallelismo massimo che un'applicazione può ottenere è quindi proporzionale al numero massimo dei stream task, che a sua volta è determinato dal numero massimo di partizioni del topic di input da cui l'applicazione sta prendendo dati.

Nel caso vengano eseguite più istanze della stessa applicazione da parte di diverse macchine, i task possono essere distribuiti automaticamente dalla libreria dalle applicazioni stesse. L'assegnamento delle partizioni ai task rimane invariato; se un'istanza fallisce, tutti i suoi task verranno riavviati su altre istanze e continueranno a consumare dalle stesse partizioni di stream.

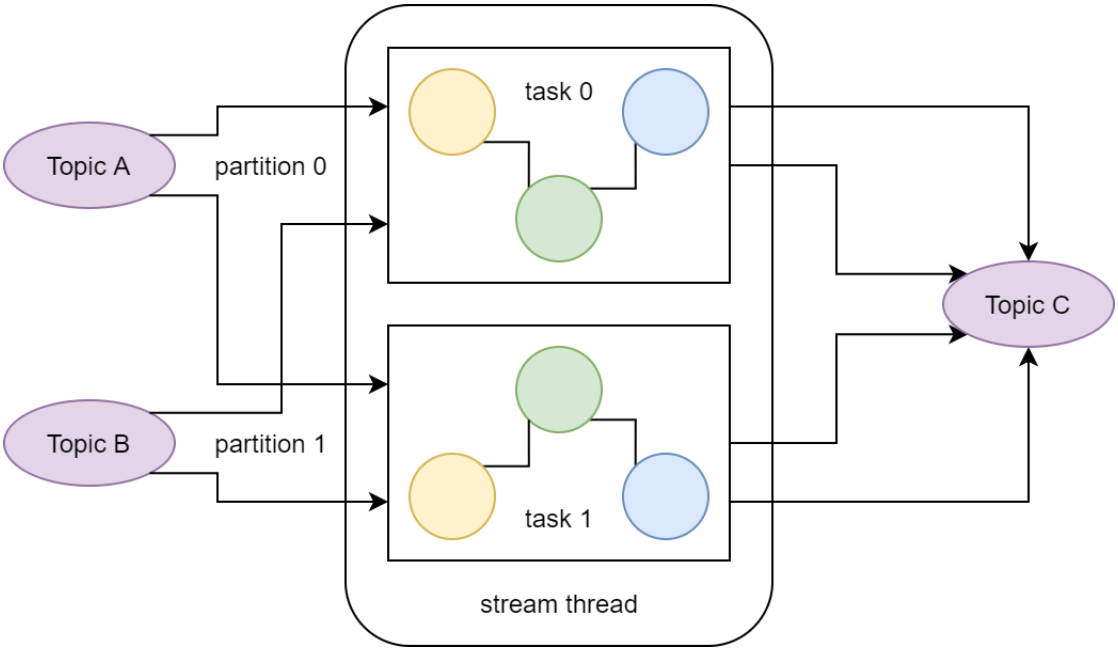

Figura 8: Esempio di distribuzione di partizioni tra task

Kafka Streams permette all'utente di configurare il numero di thread che la libreria può usare per parallelizzare lo stream processing in un'applicazione. Ciascun thread può eseguire uno o più task con le proprie topologie di processori in modo indipendente.

Kafka Streams è costruito sulle capacità di fault-tolerance integrate nativamente da Kafka. Grazie alla scalabilità orizzontale di Kafka, i dati degli stream possono essere accessibili anche nel caso l'applicazione debba riavviare la sua attività di elaborazione a seguito di imprevisti.

Per fare in modo che anche gli state store delle applicazioni siano resistenti a guasti, viene usato un topic Kafka per memorizzare in partizioni diverse i cambiamenti relativi a ciascuno di loro[10].

È possibile sviluppare un'applicazione Streams utilizzando tre diverse API:

- **Kafka Processor**: ha una logica di programmazione a basso livello, fornendo allo sviluppatore alta flessibilità;

- **Kafka Streams DSL** (Domain Specific Language): è costruito sulla base di Kafka Processor e permette di eseguire operazioni di elaborazione di dati in poche righe di codice con uno stile di programmazione dichiarativo e funzionale;
- **KSQL**: fornisce un'interfaccia SQL per lo stream processing su Kafka senza necessità di scrivere codice. Permette di ottimizzare le operazioni di trasformazione senza nessuno sforzo da parte dello sviluppatore.

2.4.2 Kafka Processor

Processor permette agli sviluppatori di definire e connettere tra loro processori personalizzati, che elaborano un record alla volta e fanno uso di state store per formare una topologia di processori.

È possibile definire un processore personalizzato implementando l'interfaccia `Processor` con il relativo metodo `process()`, chiamato per ogni ricezione di record.

Durante la fase di configurazione, il processore invoca il metodo `init()`, che fa uso di un'istanza `ProcessorContext` per accedere ai metadati del record attualmente processato, tra cui il relativo offset e il topic e la partizione di origine. Altre funzioni fornite da questa istanza sono `schedule()`, per programmare una funzione di interruzione, `forward()`, per inoltrare un nuovo record a processori downstream, e `commit()`, per registrare i progressi.

Quando un record viene inoltrato a processori downstream, gli viene assegnato un timestamp. Se il `forward()` viene invocato all'interno di `process()` il timestamp verrà ereditato dal record di input, mentre se viene invocato all'interno di `terminate()` il timestamp corrisponderà all'istante dell'interruzione.

Un processore può essere stateful o stateless; per essere stateful, deve fare uso di uno o più state store, usati per sfruttare le informazioni relative ai record ricevuti recentemente in input, effettuare operazioni di aggregazione, eliminare dati duplicati e altro.

Uno state store può essere persistente o volatile. Di default, è fault-tolerant: ciò significa che il suo contenuto viene aggiornato in un topic Kafka di riserva, detto *changelog topic*, il quale deve essere compatto per prevenire la sua crescita indefinita, ridurre i dati consumati nel cluster Kafka associato e minimizzare i tempi di ripristino dello state store.

L'implementazione di una topologia con Processor prevede la definizione di un'istanza `Topology`, alla quale vengono associati attraverso metodi appositi il source processor, processori intermedi, i relativi eventuali state store e il sink processor. Di seguito viene riportato un esempio[11]:

```
1 Topology builder = new Topology();
2
3 builder.addSource("Source", "source-topic")
4     .addProcessor("Process", () -> new WordCountProcessor(),
5         "Source")
6     .addStateStore(countStoreBuilder, "Process")
7     .addSink("Sink", "sink-topic", "Process");
```

2.4.3 Kafka Streams DSL

Per convertire i byte dei record in tipi specifici e viceversa, Kafka Streams DSL fa uso di serializzatori e deserializzatori (*serde*); in un record, i serde della chiave sono indipendenti da quelli del valore. Permette la costruzione di serde per tipi personalizzati, oltre a mettere già a disposizione quelli relativi a tipi primitivi come String, Long, ecc. e formati composti come JSON e Avro.

La libreria fornisce un'astrazione sugli stream chiamata `KStream`, che consente di generare continuamente record dal topic di input, trasformare i loro contenuti e scriverli su un topic di output. Per collezionare informazioni correlate tra loro, ovvero dei cambiamenti relativi a una determinata informazione che possono sovrascriversi tra di loro, è usata l'astrazione `KTable`, che può essere paragonata a una tabella di un database dove se il dato è già presente viene effettuato un aggiornamento del valore presente, altrimenti un inserimento.
Vi è inoltre l'astrazione `GlobalKTable` che, a differenza di `KTable`, frammenta i dati in più applicazioni Kafka Streams attive e mantiene una copia completa dei dati in ciascuna applicazione.

Le operazioni supportate dalle interfacce `KStream` e `KTable` possono essere interpretate come uno o più processori connessi in una topologia di processori.
Le trasformazioni stateless non richiedono uno state store per l'elaborazione. Le principali sono:

- `filter`: esamina una funzione booleana per ogni elemento dell'oggetto `KStream` o `KTable` e mantiene quelli che per cui la funzione è vera;
- `map`: produce dei nuovi record a partire da quelli contenuti nel `KStream` in ingresso. Quando non è necessario modificare le chiavi dei record, è preferibile usare `mapValues`, compatibile sia con `KStream` che con `KTable`, e che non porta al ripartizionamento dei dati;
- `groupBy`: raggruppa i record di un oggetto `KStream` o `KTable` per una nuova chiave, che potrebbe essere di un tipo diverso, restituendo rispettivamente un `KGroupedStream` o un `KGroupedTable`. Il raggruppamento è un prerequisito necessario per l'aggregazione di stream e tabelle, e assicura che il dato sia stato partizionato propriamente per le operazioni successive. Quando non è necessario modificare le chiavi dei record, è preferibile usare `groupByKey`, che non porta al ripartizionamento dei dati;
- `branch`: divide un `KStream` in una o più istanze `KStream` in base al predicato fornito. È possibile usare più predicati, esaminati in ordine. Può essere utile per indirizzare record verso topic diversi in downstream;

- `merge`: unisce due `KStream` in un unico oggetto, senza garantire l'ordinamento tra record provenienti da diversi stream;
- `toStream`: crea un `KStream` con tutti i record del `KTable` dato in input.

Le trasformazioni stateful dipendono dallo stato dell'applicazione per essere eseguite. Possono essere suddivise in quattro categorie principali:

- *aggregazione*: si occupano di aggregare tra loro i record raggruppati per chiave tramite `groupByKey` o `groupBy`. Le principali, eseguibili sia su dati raggruppati per finestre che non, sono:
  - `reduce`: combina i valori di record con la stessa chiave. Il valore del record corrente viene combinato con l'ultimo valore ridotto, restituendo un nuovo valore a ogni iterazione. Il tipo del valore restituito non può essere cambiato;
  - `aggregate`: è una generalizzazione di `reduce`, permettendo quindi di restituire valori di tipi diversi da quelli di input;
  - `count`: restituisce il numero di record per chiave;
- *join*: in maniera similare ai database relazionali, in Kafka Streams è possibile effettuare operazioni di inner join, left join e outer join con stream e tabelle. Diverse applicazioni che hanno necessità di effettuare operazioni di join su dati contenuti in un database remoto le effettuano su stream trasferiti su cluster Kafka tramite l'API Kafka Connect per migliorare la velocità e l'efficienza, evitando di interrogare il database per ciascun record.

  Per garantire che i record con la stessa chiave vengano consegnati allo stesso stream task, l'utente deve assicurarsi che i dati in input siano co-partizionati durante la fase di joining, ovvero che siano associati allo stesso numero di partizione seppur di topic diversi.
- *windowing*: controllano il raggruppamento dei record con la stessa chiave per le operazioni stateful in *finestre*. Una volta raggruppati i record, il windowing permette di suddividere ulteriormente in sottogruppi i record di una chiave.

  Nelle operazioni di join, le finestre vengono utilizzate per memorizzare i record ricevuti fino a un determinato momento nei limiti della dimensione della finestra. Nelle operazioni di aggregazione, vengono memorizzati gli ultimi risultati di aggregazione per ciascuna finestra. Le finestre e i rispettivi contenuti vengono eliminati allo scadere del periodo di ritenzione configurato da Kafka Streams.
- applicazione di processi e trasformatori personalizzati

[12]

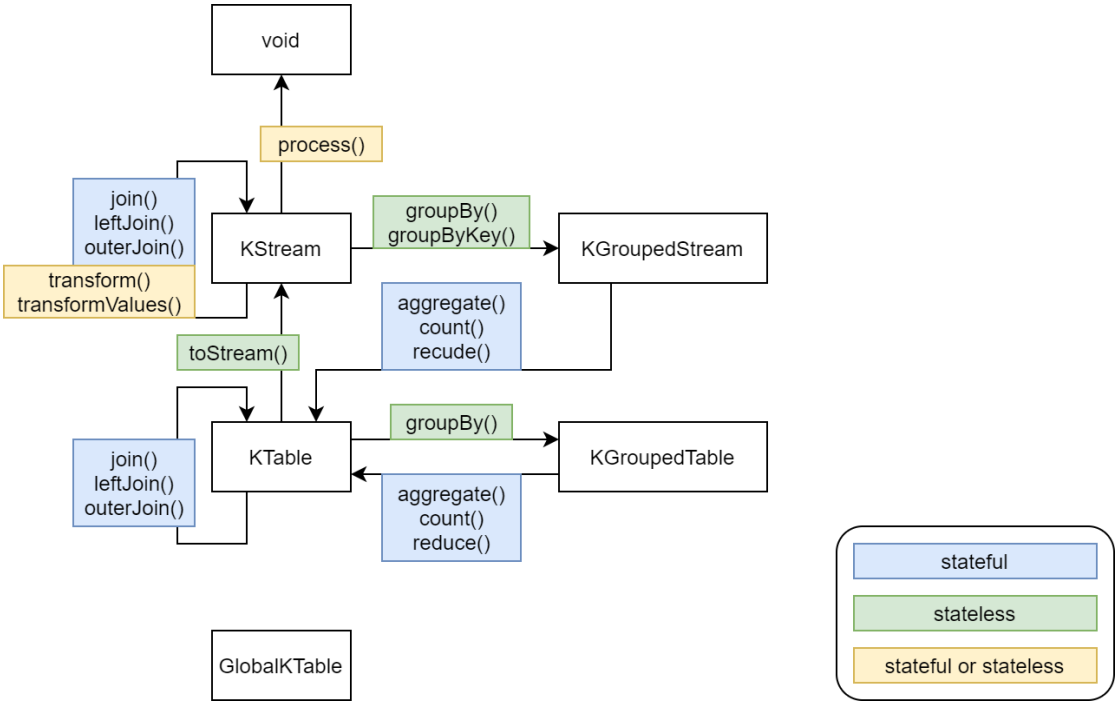

Figura 9: Relazioni tra trasformazioni stateful in Kafka Streams DSL

2.4.4 KSQL

KSQL permette lo sviluppo di applicazioni di streaming utilizzando dichiarazioni e query SQL. È composto da tre componenti principali:

- Motore KSQL, attraverso il quale viene definita la logica dell'applicazione ed eseguite le operazioni relative sui server KSQL disponibili. Un motore KSQL interpreta le dichiarazioni KSQL e costruisce le topologie corrispondenti;
- CLI KSQL, mette a disposizione una console con interfaccia a linea di comando per interagire con il motore KSQL;
- Interfaccia REST, permette la comunicazione tra CLI e motore.

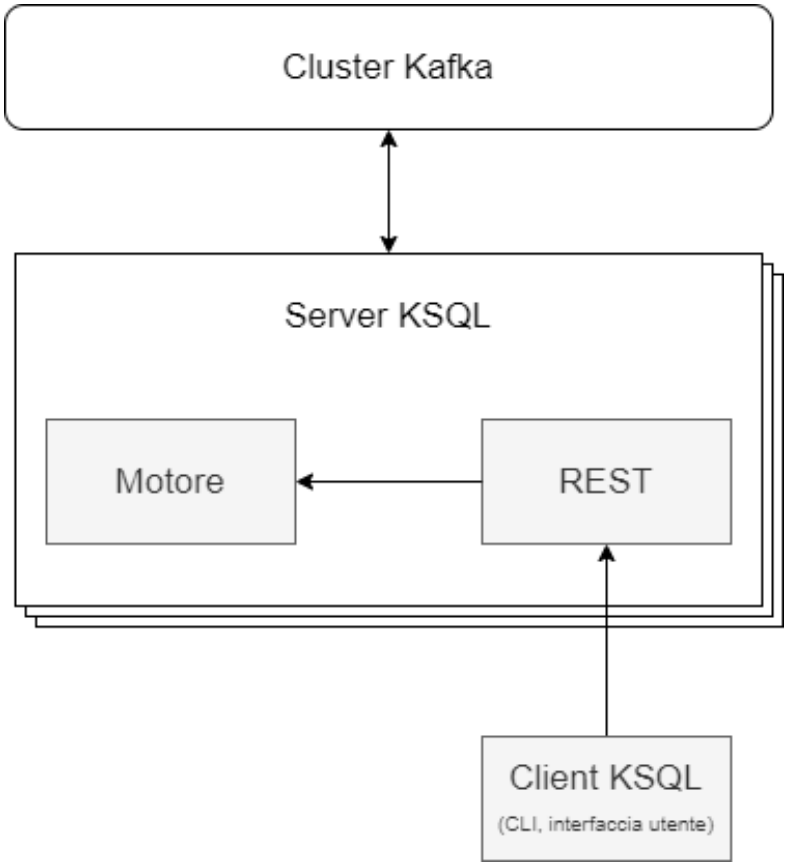

Figura 10: Architettura KSQL

KSQL supporta due categorie di dichiarazioni: Data Definition Language (DDL), per la creazione e l'eliminazione di stream e tabelle, e Data Manipulation Language (DML), per la loro lettura e modifica.

2.5 Kafka Connect

Kafka Connect è un framework per connettere i broker Kafka con sistemi esterni utilizzando *connettori* (*connector*), componenti che dispongono di molteplici implementazioni per permettere l'esportazione e l'importazione dei dati relativi a sorgenti e punti di scarico (sink) diversi. È possibile creare implementazioni personalizzate per tipologie di sistemi non coperte da quelle esistenti.

Un *source connector* colleziona dati da database e tabelle di stream e colleziona le metriche dai server dell'applicazione nei topic Kafka, rendendo i dati disponibili per uno stream processing con bassa latenza.

Un *sink connector* consegna dati dai topic Kafka a indici secondari come Elasticsearch o sistemi batch come Hadoop per analisi offline.

Kafka Connect può essere eseguito sia come un processo autonomo per l'esecuzione di compiti su una singola macchina che come un servizio distribuito, scalabile e fault-

tolerant. Ciò gli permette di essere ridimensionato in base alla quantità del carico operativo utilizzato[13].

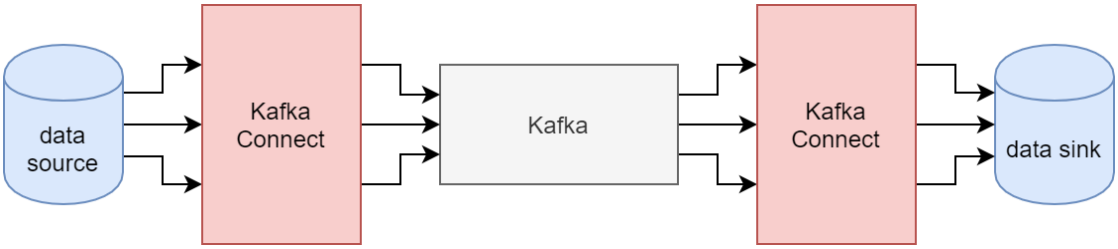

Figura 11: Logica di Kafka Connect

L'esecuzione in modalità standalone, e quindi da parte di un singolo processo (*worker*), avviene mediante lo script `connect-standalone.sh`, il quale riceve come primo parametro il file `.properties` di configurazione contenente i seguenti parametri:

- `bootstrap.servers`: elenco dei server Kafka usati per stabilire la connessione al cluster Kafka;
- `key.converter` e `value.converter`: classi implementanti l'interfaccia `Converter` per convertire nel formato di Kafka Connect i dati serializzati contenuti nei topic Kafka, e viceversa;
- `offset.storage.file.filename`: nome del file usato dai source connector per memorizzare gli offset dei dati in input e quindi ricordare fino a dove è stata eseguita la lettura.

Questi parametri hanno lo scopo di permettere ai producer e consumer usati da Kafka Connect di accedere a informazioni relative a configurazioni, offset e topic di stato. Per la configurazione di task source e sink è possibile usare gli stessi parametri con, rispettivamente, i prefissi `consumer` e `producer`.
I parametri successivi passati all'esecuzione dello script sono relativi ai file di configurazione dei connettori.

L'esecuzione in modalità distribuita avviene mediante lo script `connect-distributed.sh`. Informazioni su configurazioni, offset e stati sono conservate in topic Kafka, che possono essere creati manualmente con il numero di partizioni e il replication factor desiderati o creati automaticamente all'avvio di Kafka Connect con i parametri di default. Il file di configurazione passato nel primo parametro dello script permette l'avvio di più worker distribuiti, usando oltre ai parametri precedentemente descritti per la modalità standalone:

- `group.id`: tutti i worker con lo stesso id fanno parte dello stesso cluster Connect. Non deve entrare in conflitto con id di consumer group;

- `config.storage.topic`: topic usato per memorizzare configurazioni di connettori e task. Deve essere un topic a singola partizione, altamente replicato e compattato;
- `offset.storage.topic` e `status.storage.topic`: topic usati per memorizzare rispettivamente offset e stati. Devono avere tante partizioni, essere replicati e configurati per ottenere compattazione.

La configurazione dei connettori non è gestita passando all'avvio dello script i nomi dei file relativi, ma mediante un'interfaccia API REST usata per creare, modificare ed eliminare connettori. È interfacciabile mediante gli indirizzi specificati nel campo `listeners` nel file di configurazione Kafka; di default, viene utilizzata la porta `8083` con protocollo HTTP.

In modalità standalone i parametri di configurazione dei connettori sono definiti in file `.properties` all'avvio del processo Connect; in modalità distribuita vengono inclusi nel payload JSON nella richiesta relativa alla creazione del connettore. I parametri di configurazione più comuni sono:

- `name`: nome univoco del connettore;
- `connector.class`: la classe Java del connettore;
- `tasks.max`: il numero massimo di task creabili per il connettore;
- `key.converter` e `value.converter`: sono opzionali e sovrascrivono i valori di conversione impostati dal worker;
- `topics`: usato dai sink connector, l'elenco dei topic di input;
- `topics.regex`: usato dai sink connector, espressione regolare che definisce i possibili topic di input.

I connettori possono essere configurati con *trasformazioni*, funzioni che effettuano modifiche semplici e leggere sui singoli messaggi ricevuti in input.

Le catene di trasformazioni possono essere specificate nella configurazione del connettore:

- `transforms`: elenco di alias relativi alle trasformazioni da eseguire sull'input, eseguite in ordine;
- `transforms.$alias.type`: nome della classe associata alla trasformazione;
- `transforms.$alias.$transformationSpecificConfig`: proprietà di configurazione utilizzata dalla trasformazione.

Kafka Connect include un insieme di trasformazioni di utilità generale[7]:

- `InsertField`: aggiunge un campo usando attributi dai metadati contenuti nel record o un valore statico configurato;

- `ReplaceField`: filtra o rinomina un campo;
- `MaskField`: sostituisce un campo con un valore nullo valido;
- `ValueToKey`: sostituisce la chiave del record con una nuova chiave formata da un sottoinsieme di campi del valore del record;
- `HoistField`: racchiude dati usando il nome del campo specificato in un oggetto Struct se vi è uno schema o Map altrimenti;
- `ExtractField`: estrae il campo specificato da uno Struct o un Map;
- `SetSchemaMetadata`: modifica il nome o la versione dello schema dato;
- `TimestampRouter`: modifica il topic di un record utilizzando il nome del topic originale e il timestamp corrente;
- `RegexRouter`: modifica il topic di un record utilizzando il nome del topic originale, una stringa sostitutiva e un'espressione regolare.

3 Tecnologie ausiliarie

3.1 Apache ZooKeeper

Apache ZooKeeper è un servizio atto alla coordinazione e alla configurazione dei dati di sistemi distribuiti. Un insieme di server ZooKeeper tiene traccia di diverse informazioni condivise tra i consumer e i broker Kafka.

I dati in ZooKeeper sono distribuiti tra molteplici collezioni di nodi, raggiungendo alta disponibilità e consistenza. Nel caso un nodo fallisca, ZooKeeper può effettuare una migrazione istantanea di failover, selezionando in tempo reale un altro nodo per continuare l'esecuzione dell'operazione. Nel caso un client in fase di connessione al server non riceva una risposta, può procedere all'interrogazione di un altro nodo[14].

Kafka utilizza ZooKeeper per individuare l'aggiunta e la rimozione di broker e consumer, per poi ribilanciare la distribuzione delle partizioni a seguito dei cambiamenti avvenuti; fino alla versione 0.9, la traccia degli offset dei messaggi attesi dai consumer era tenuta da ZooKeeper, ed è stata passata ai broker Kafka nelle versioni successive.

Quando ciascun broker o consumer viene avviato, memorizza le sue informazioni in un registro ZooKeeper. I registri dei broker contengono i relativi nome dell'host, porta e riferimenti ai topic e alle partizioni memorizzati in essi. I registri dei consumer memorizzano il consumer group di appartenenza e il set di topic a cui sono sottoscritti[8].

È necessario che sia presente un numero dispari di istanze di ZooKeeper per mantenere un *quorum*, ovvero un numero valido per mantenere un cluster con un solo leader attivo in caso di guasti.

Il quorum è definito dalla formula `Q=2N+1`, dove `Q` corrisponde al numero di nodi richiesti alla formazione di un insieme sicuro capace di tollerare guasti in `N` nodi; seguendo la formula, è possibile osservare che numeri pari di nodi coprono lo stesso numero di guasti di numeri dispari precedenti, e non sono quindi consigliabili.

Il numero di istanze ZooKeeper utilizzate è solitamente 3, 5 o 7, in quanto un numero maggiore porta a eccessiva latenza nelle prestazioni[15].

Per eseguire un servizio ZooKeeper, è necessario configurarlo mediante un apposito file che può contenere i seguenti principali parametri:

- `tickTime`, l'unità temporale standard in millisecondi utilizzata da ZooKeeper per misurare il tempo di vita del sistema. Il timeout di sessione è due volte questo valore;
- `dataDir`, la posizione per memorizzare gli snapshot del database in memoria e i relativi aggiornamenti;
- `clientPort`, porta utilizzata per le comunicazioni coi client. È convenzione usare la porta `2181`;
- `server.X` (`server.1`, `server.2`...) per ciascun server del cluster, viene specificato il relativo indirizzo con 2 porte TCP. La prima viene usata dai follower per comunicare con il leader, mentre l'altra è usata per l'elezione di un nuovo leader. Convenzionalmente, sono impostate rispettivamente a `2888` e `3888`.

3.2 Apache Storm

Apache Storm è un sistema open source, distribuito, affidabile e fault-tolerant per l'elaborazione di stream di dati distribuiti in tempo reale.

I suoi componenti principali sono *spout*, sorgenti di stream, e *bolt*, che dopo avere ricevuto i dati dagli spout si occupano di elaborarli ed emettere nuovi stream. Un cluster Storm può essere visto come una catena di bolt, che eseguono una serie di trasformazioni dei dati generati da uno spout. I flussi di lavoro tra spout e bolt sono detti *topologie*.

Per ottimizzare lo streaming real-time di grandi quantità di dati, è possibile integrare cluster Kafka e Storm. È disponibile un'implementazione spout per la lettura da un cluster Kafka (`KafkaSpout`), che richiede i seguenti parametri:

- elenco dei broker Kafka;
- numero di partizioni per host;
- nome del topic di riferimento per l'estrazione dei messaggi;
- percorso di ZooKeeper per la memorizzazione degli offset dei consumer;
- ID del consumer richiesto per la memorizzazione degli offset.

Il codice seguente mostra un esempio di istanziazione di un oggetto `KafkaSpout`:

```
1 KafkaSpout kafkaSpout = new KafkaSpout(new SpoutConfig(
2  ImmutableList.of("localhost:9092", "localhost:9093"),
3  2,
4  " othertopic",
5  "/kafkastorm",
6  "consumID"));
```

Di default, nel caso non venga specificato il percorso del cluster ZooKeeper, Storm ne utilizza uno proprio[16].

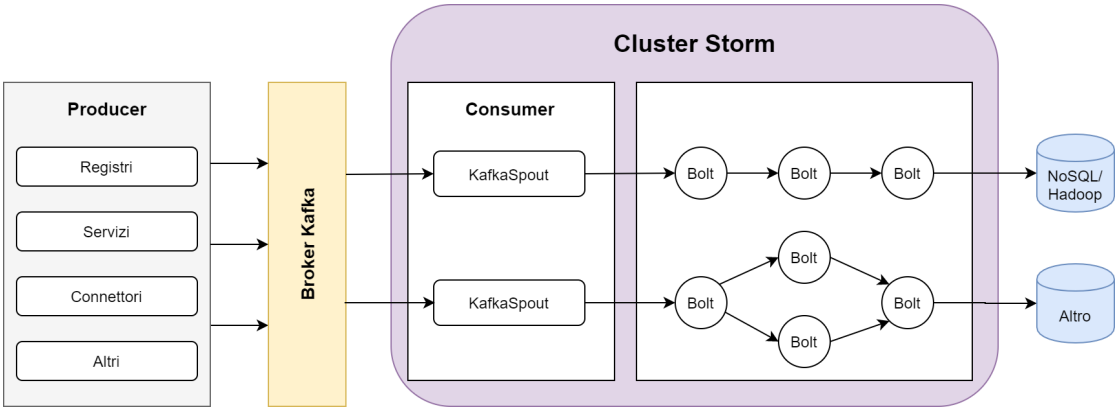

Figura 12: Grafico raffigurante l'integrazione tra Storm e Kafka

3.3 Apache Hadoop

Apache Hadoop è un framework per il calcolo distribuito su larga scala, in grado di gestire l'elaborazione di dati in termini di petabyte e utilizzando potenzialmente migliaia di nodi.

I moduli principali che compongono Hadoop sono:

- Hadoop Common: comprende un set di funzionalità comuni (operazioni di input/output, error detection, ecc.) insieme ad alcuni tool di configurazione necessari agli altri moduli di Hadoop;
- HDFS (Hadoop Distributed File System): file system distribuito che memorizza i dati nei cosiddetti *blocchi* replicandoli su più nodi, garantendo fault-tolerance e alta disponibilità. Si basa su un'architettura Master Slave: mentre i DataNode si occupano di mantenere e recuperare i blocchi di dati, i NameNode gestiscono le richieste del client avendo una visione completa dell'organizzazione dei dati all'interno del cluster;
- Hadoop YARN: responsabile della gestione delle risorse del cluster, permette una separazione tra l'infrastruttura e il modello di programmazione. Quando occorre eseguire un'applicazione, viene fatta richiesta al ResourceManager, il quale è incaricato di trovare un NodeManager libero. Quest'ultimo si occuperà di avviare l'applicazione all'interno di un contenitore;
- Hadoop MapReduce: software per il processamento di grandi quantità di dati distribuiti su un cluster di nodi. Il dataset di ingresso viene diviso in chunk di dati (tipicamente di 128 MB), i quali vengono elaborati in parallelo dalla funzione di Map che si occupa di organizzare i dati in coppie chiave-valore. Il risultato

viene mandato in input alla funzione di Reduce che produrrà il risultato finale in output[17].

I componenti principali dell'HDFS sono:

- Name Node, è l'applicazione che gira sul server principale. Memorizza le informazioni relative ai blocchi;
- Secondary Name Node, memorizza le modifiche sui registri, al fine di ripristinare lo stato più recente dell'HDFS in caso di guasti;
- Data Node, nodi che memorizzano i dati distribuiti in blocchi, oltre a repliche di dati di altri nodi;
- Job Tracker, responsabile della suddivisione dei compiti MapReduce in task più piccoli;
- Task Tracker, responsabile dell'esecuzione dei task predisposti dal Job Tracker.

Hadoop può essere integrato anche con Kafka sia come producer che come consumer.

Un producer Hadoop può interfacciarsi ai topic Kafka mediante degli URI:
`kafka://<kafka-broker>/<kafka-topic>`
Per leggere i dati da Hadoop, vengono adottati due metodi:

- utilizzare uno script Pig per scrivere messaggi in formato Avro binario, dove ogni riga corrisponde a un messaggio: per trasmettere i dati nel cluster Kafka, la classe `AvroKafkaStorage` prende lo schema Avro come primo argomento e effettua la connessione tramite URI;
- utilizzare la classe Kafka `OutputFormat`: questo approccio prevede l'invio di messaggi come byte e fornisce controllo sull'output mediante metodi di publishing di basso livello. La classe `OutputFormat` utilizza a sua volta la classe `KafkaRecordWriter` per scrivere un record sul Hadoop Kafka.

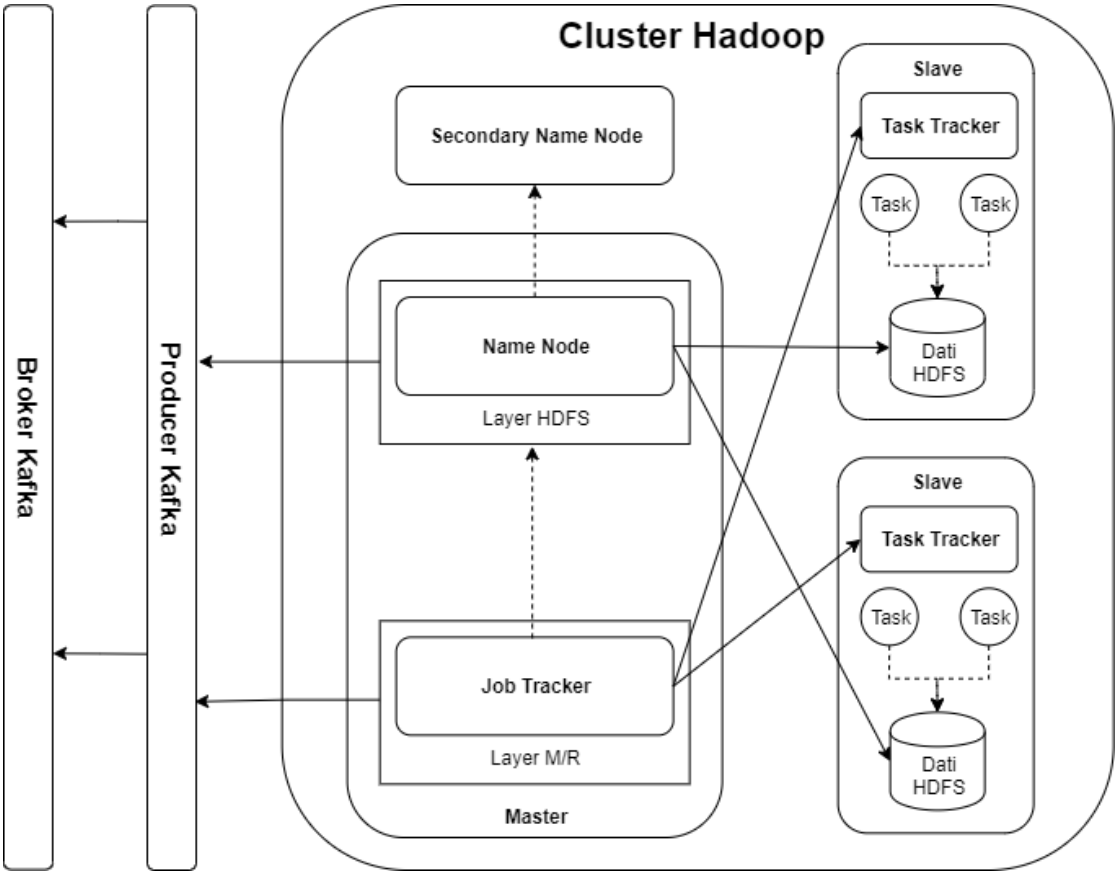

Figura 13: Producer Hadoop

In qualità di consumer, un'unità di lavoro Hadoop effettua caricamento parallelo da Kafka all'HDFS, con un numero di mapper per il caricamento dei dati dipendente dalla quantità dei file nella directory di input. La directory di output contiene dati provenienti da Kafka e gli offset dei messaggi relativi. I singoli mapper scrivono l'offset dell'ultimo messaggio consumato sull'HDFS al termine del task di mapping. Nel caso l'operazione fallisca e sia necessario un riavvio, ciascun mapper riprende l'elaborazione a partire dagli offset memorizzati nell'HDFS[16].

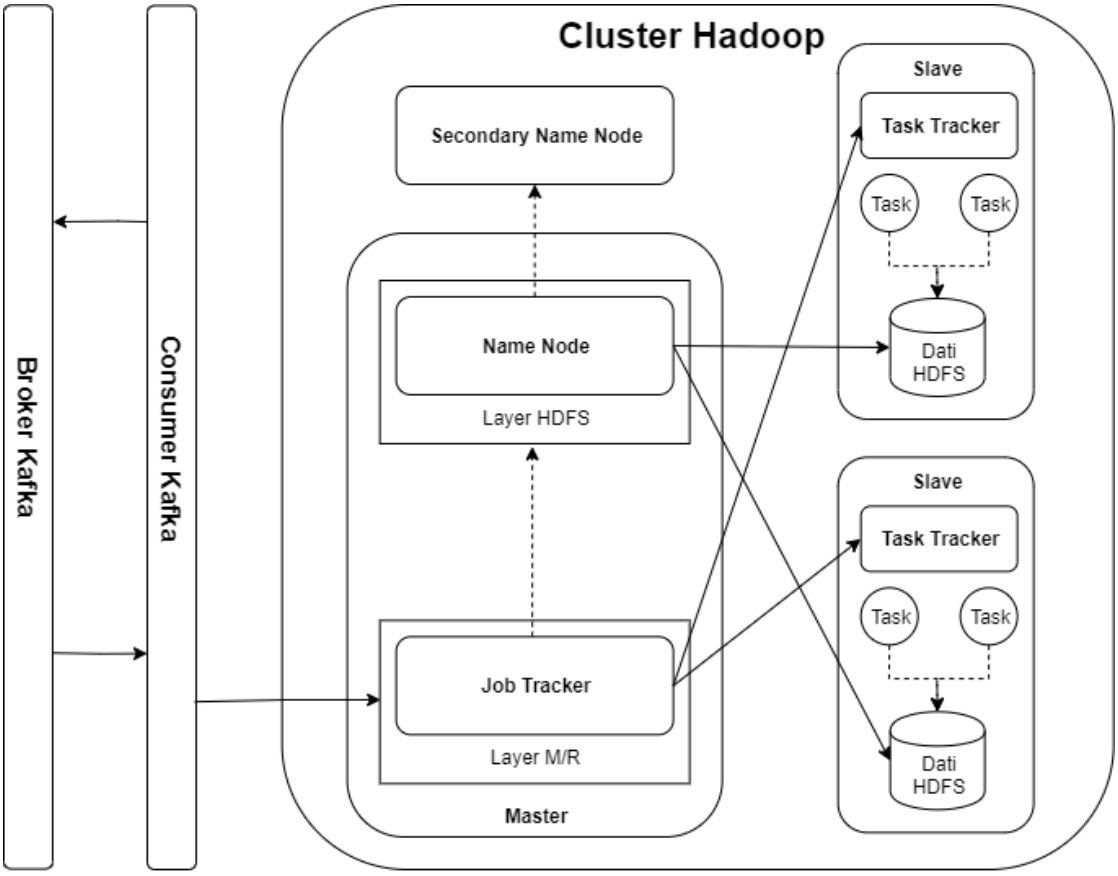

Figura 14: Consumer Hadoop

3.4 Apache Avro

Apache Avro è un framework per la serializzazione di dati, sviluppato all'interno del progetto Apache Hadoop. Utilizza JSON per la definizione di tipi di dati e protocolli, serializza dati in un formato compatto binario.

Il suo impiego primario è di supporto ad Apache Hadoop, per fornire un formato di serializzazione per i dati persisenti, un formato per la comunicazione tra i nodi e per programmi client che interagiscono con i servizi di Hadoop. Avro usa uno schema per la struttura dei dati che devono essere codificati.

Un esempio di schema Avro può essere:

```
1 {
2    "namespace": "example.avro",
3    "type": "record",
4    "name": "User",
5    "fields": [
6       {"name": "name", "type": "string"},
7       {"name": "favorite_number",  "type": ["null", "int"]},
8       {"name": "favorite_color", "type": ["null", "string"]}
```

```
    ]
}
```

4 Test

Sono state sviluppate diverse applicazioni di test utilizzando le API Consumer e Producer di Kafka per evidenziarne le principali potenzialità. È stato utilizzato Docker per la creazione di un cluster Kafka e un cluster ZooKeeper con cui interfacciare i diversi client Java. I singoli server sono associati a dei *container*, processi isolati dal resto del sistema che si interfacciano tra di loro mediante indirizzi IP, costruiti a partire da *immagini* `bitnami/kafka` e `bitnami/zookeeper`, file reperibili sul Docker Hub e utilizzati per la creazione e l'inizializzazione dei server. L'utilizzo dei comandi Docker viene gestito a livello di codice.

Prima dell'esecuzione di funzioni di test specifiche, viene chiamato il comando `docker-compose up` passando come argomento un file di configurazione YAML contenente informazioni necessarie alla creazione di un network con i container ZooKeeper e Kafka utilizzati. Ciascun container memorizza i dati generati in appositi `volumi`, aree di memoria completamente gestite da Docker e non accessibili dagli altri container in quanto usati localmente. A ciascun server ZooKeeper è stata mappata tra host e container una porta a partire da `2181`, e a ciascun broker Kafka a partire da `9092`.

Per i container ZooKeeper sono state utilizzate le seguenti variabili d'ambiente:

- `ZOO_SERVER_ID`: identifica il singolo server nel cluster ZooKeeper, di default è pari a `1`;
- `ZOO_PORT_NUMBER`: la porta di accesso al client ZooKeeper, di default è pari a `2181`;
- `ZOO_SERVERS`: elenco di server ZooKeeper facenti parte del cluster, in formato `<indirizzo>:2888:3888`;
- `ALLOW_ANONYMOUS_LOGIN`: impostato a `yes`, abilita la connessione a utenti non autentificati.

Per i container Kafka, le principali variabili d'ambiente utilizzate sono:

- `KAFKA_CFG_ZOOKEEPER_CONNECT`: elenco dei server ZooKeeper con cui il broker può connettersi;
- `KAFKA_CFG_BROKER_ID`: identifica il singolo server nel cluster Kafka;
- `KAFKA_CFG_LOG_DIRS`: percorsi in cui il broker salva i messaggi;
- `ALLOW_PLAINTEXT_LISTENER`: impostato a `yes`, permette l'utilizzo di un canale di comunicazione non-criptato e non autenticato;

- `KAFKA_CFG_LISTENERS`: elenco di indirizzi IP utilizzati per accedere al broker all'interno dell'ambiente Docker;
- `KAFKA_CFG_ADVERTISED_LISTENERS`: elenco di indirizzi IP utilizzati per accedere al broker al di fuori dell'ambiente Docker.

I test sviluppati per primi verificano la funzionalità dei sistemi prodotti. Successivamente, sono stati sviluppati test per verificare le prestazioni e le feature più particolari della tecnologia Kafka.

Le specifiche tecniche della macchina su cui sono stati eseguiti i test sono le seguenti:

- **Processore**: Intel Core i7-7700HQ 2.8 GHZ
- **RAM**: 8 GB
- **Sistema Operativo**: Windows 10 Home 64 bit

Fare riferimento al README.md presente nel repository per ulteriori informazioni.

4.1 Automazione dei test

Data la complessità e numerosità di configurazioni necessarie ai test, si è reso necessario automatizzare il più possibile l'esecuzione di script e singoli comandi; l'ambiente di lavoro dei client Kafka viene resettato ad ogni esecuzione di un test. A questo fine è stata implementata la classe `DockerTerminal`, che aggrega più comandi secondo una determinata funzione e non richiede all'utente alcuna interazione con il sistema. Questo permette anche nelle classi di test di ordinare temporalmente eventi riguardanti i client e l'ambiente, così da fare risaltare l'aspetto da analizzare in questione. Il test `TestDockerTerminal` crea, aggiunge un topic e distrugge tre broker in modo automatico (senza che l'utente debba lanciare comandi da terminale):

```
1    assertTrue(DockerTerminal.composeUp(COMPOSER));
2    assertTrue(DockerTerminal.createTopic("testTopic"));
3    assertTrue(DockerTerminal.composeDown(COMPOSER));
```

L'automazione avviene con l'ausilio della shell *bash* (indipendentemente dal sistema operativo).

4.2 Funzionalità: Comunicazione di base

Test atto alla verifica del funzionamento di un sistema basato su 3 broker, un producer e un consumer. Verifica che il consumer riceva correttamente i 10 messaggi inviati dal producer. Eventualmente, il consumer può ricevere in maniera non ordinata i messaggi, poiché l'implementazione data al producer non garantisce l'ordine di ricezione, e siccome il topic è ripartito su 3 partizioni l'ordine è garantito solo all'interno di ciascuna di esse, ma non nel loro complesso. Pertanto il producer invia sempre in ordine:

```
...
[P] Sent message: (1, Message_1)
[P] Sent message: (2, Message_2)
[P] Sent message: (3, Message_3)
[P] Sent message: (4, Message_4)
[P] Sent message: (5, Message_5)
[P] Sent message: (6, Message_6)
[P] Sent message: (7, Message_7)
[P] Sent message: (8, Message_8)
[P] Sent message: (9, Message_9)
[P] Sent message: (10, Message_10)
...
```

Mentre un esempio tipico di ricezione da parte del consumer può essere:

```
...
[C] Received message: (7, Message_7) at offset 0
[C] Received message: (9, Message_9) at offset 1
[C] Received message: (1, Message_1) at offset 0
[C] Received message: (2, Message_2) at offset 1
[C] Received message: (4, Message_4) at offset 2
[C] Received message: (3, Message_3) at offset 0
[C] Received message: (5, Message_5) at offset 1
[C] Received message: (6, Message_6) at offset 2
[C] Received message: (8, Message_8) at offset 3
[C] Received message: (10, Message_10) at offset 4
...
```

Dagli offset dei messaggi si può facilmente dedurre che i messaggi 7 e 9 prodotti sono finiti nella prima partizione e pertanto processati per primi, I messaggi 1, 2 e 4 nella seconda partizione e similmente i rimanenti nella terza partizione. Si può anche notare come all'interno di ogni partizione i messaggi siano ordinati.

Un test molto simile `TestHeavyCommunication` è stato sviluppato per testare la possibilità di inviare file al posto di semplici messaggi (in questo caso file dummy di 16MB). Il funzionamento richiede l'override delle proprietà dei broker come `message.max.bytes` e `replica.fetch.max.bytes`, che indicano rispettivamente la dimensione massima del batch di un record e la dimensione massima di un record su una partizione per cui fare fetch.

4.3 Funzionalità: Comunicazione Assign&Seek

Questo scenario verifica il funzionamento di un sistema basato su 3 broker, un producer e un consumer che usa un diverso approccio per il fetch dei messaggi detto Assign&Seek. Questo approccio può essere brevemente descritto come una singola lettura di più messaggi alla volta. L'implementazione consiste in due step:

- Setup:
  - Assign: assegnazione al consumer del topic a cui appartengono i messaggi da ricevere e della partizione su cui saranno depositati;

- Seek: override dell'offset su cui fare il fetch dei messaggi;

- Fetch: fetch unico del numero di messaggi necessario.

Output prodotto dal test:

```
...
[P] Sent message: (1, Message_1)
[C] Key: 1, Value: Message_1 Partition: 0, Offset: 0
[C] Key: 2, Value: Message_2 Partition: 0, Offset: 1
[P] Sent message: (2, Message_2)
[P] Sent message: (3, Message_3)
[C] Key: 4, Value: Message_4 Partition: 0, Offset: 2
[P] Sent message: (4, Message_4)
[P] Sent message: (5, Message_5)
[P] Sent message: (6, Message_6)
[P] Sent message: (7, Message_7)
[P] Sent message: (8, Message_8)
[P] Sent message: (9, Message_9)
[P] Sent message: (10, Message_10)
...
```

Sono stati ricevuti solo i messaggi appartenenti alla partizione 0.

4.4 Funzionalità: Stream

È stata sviluppata una piccola applicazione di test usando Kafka Streams DSL che sfrutta la classe `TopologyTestDriver` per simulare un'applicazione che raccoglie i dati da un topic di input e li invia a un topic di output rielaborando il loro contenuto. La topologia consuma i messaggi nel topic `TextLinesTopic`, suddivide i valori in parole e le raggruppa, producendo nel topic `WordsWithCountsTopic` nuove coppie chiave-valore dove le chiavi corrispondono alle parole e i valori alle rispettive quantità.

```
final StreamsBuilder builder = new StreamsBuilder();
    final KStream<String, String> textLines =
    builder.stream("TextLinesTopic");
    final KTable<String, Long> wordCounts = textLines
            .flatMapValues(textLine ->
                Arrays.asList(textLine.toLowerCase()
                      .split("\\W+")))
            .groupBy((key, word) -> word)
            .count(Materialized.<String, Long,
                KeyValueStore<Bytes, byte[]>>
                as("counts-store"));
    wordCounts.toStream().to("WordsWithCountsTopic",
    Produced.with(Serdes.String(), Serdes.Long()));
    return builder.build();
```

Il conteggio fa uso di eventuali informazioni contenute nello state store: se ad esempio lo state store contiene la coppia chiave-valore `test 3`, la produzione di un messaggio con

valore `test` nel topic `TextLinesTopic` porterà alla produzione del messaggio `test 4` nel topic `WordsWithCountsTopic`.

Lo state store inizialmente avrà valori nulli per parole mai prodotte prima.

4.5 Performance: Integrazione tra diverse implementazioni

Viene testata la comunicazione fra due client scritti con diversi linguaggi di programmazione su due diverse macchine fisiche. Il sistema è composto da broker e consumer Java su una macchina e un producer C su un'altra macchina. L'applicazione C chiede di specificare l'indirizzo IP a cui inviare il messaggio da inserire, usando la porta 9092. Il consumer a quel punto terminerà una volta ricevuto il messaggio. È necessario che il parametro `listener` abbia valore `0.0.0.0` in modo da permettere l'ascolto da qualunque mittente (l'indirizzo IP non è noto a priori e può avere un assegnamento dinamico all'interno della rete). Di default il file compose YAML utilizzato espone la porta (per il primo broker 9092) alla macchina host mappandola con lo stesso valore.

4.6 Performance: Latenza di comunicazione

Kafka è in grado di gestire la consegna di messaggi a latenze molto basse: per dimostrarlo, è stato costruito un test che misuri per ciascun messaggio l'intervallo di tempo tra la sua produzione in un topic da parte di un producer e il relativo consumo da parte di un consumer. Viene prodotto un messaggio al secondo per una durata di 8 ore.

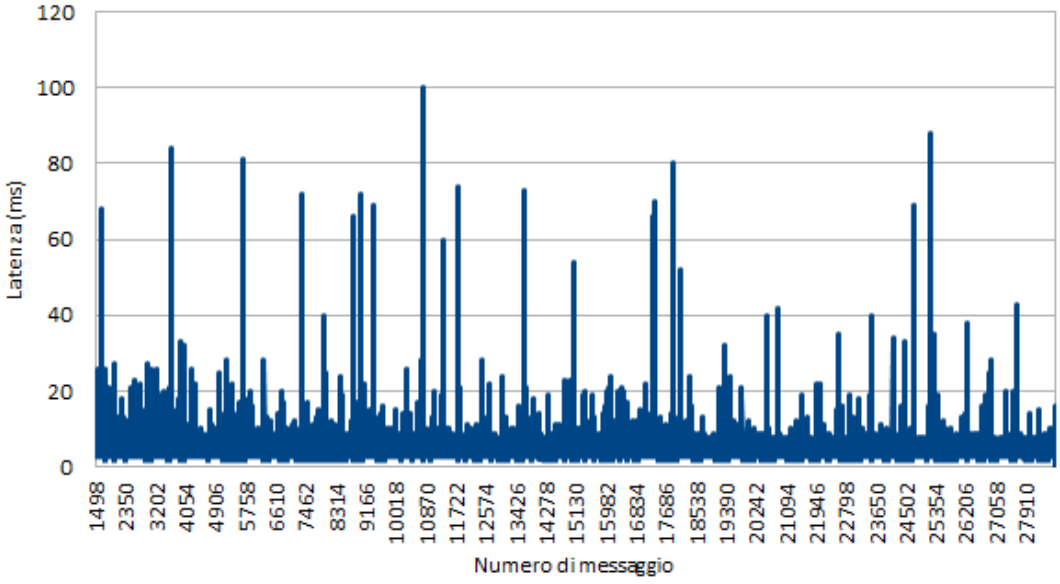

Figura 15: Tempi di latenza di trasmissione dei messaggi

Salvo durante la fase di avvio dei client Kafka e durante uno ristabilimento della connessione dei client ai broker, i tempi di latenza della trasmissione dei messaggi dal producer al consumer non hanno superato i 106 millisecondi. La latenza media complessiva è pari a 5.73 millisecondi.

4.7 Performance: Tweaking dei messaggi

I messaggi conservati nei topic Kafka non possono essere modificati direttamente, ma è necessario consumarli e produrre un nuovo messaggio utilizzando i dati dei messaggi consumati modificati. Per dimostrare il funzionamento di questa logica, è stata costruita la classe `TweakerClient` che fa uso di un consumer Kafka per ricevere i messaggi di un topic e di un producer per inoltrarli con valore modificato a un secondo topic.

Un test verifica che ciascun messaggio prodotto verso il primo topic sia stato consumato correttamente con valore modificato dal secondo topic, oltre a registrare i rispettivi tempi di latenza. Il confronto tra i messaggi originali e quelli modificati è stato possibile utilizzando un'unica partizione per topic, garantendo l'ordine temporale di invio.

La latenza media complessiva è pari a 11.40 millisecondi; si può osservare che corrisponde all'incirca al doppio del risultato del test precedente.

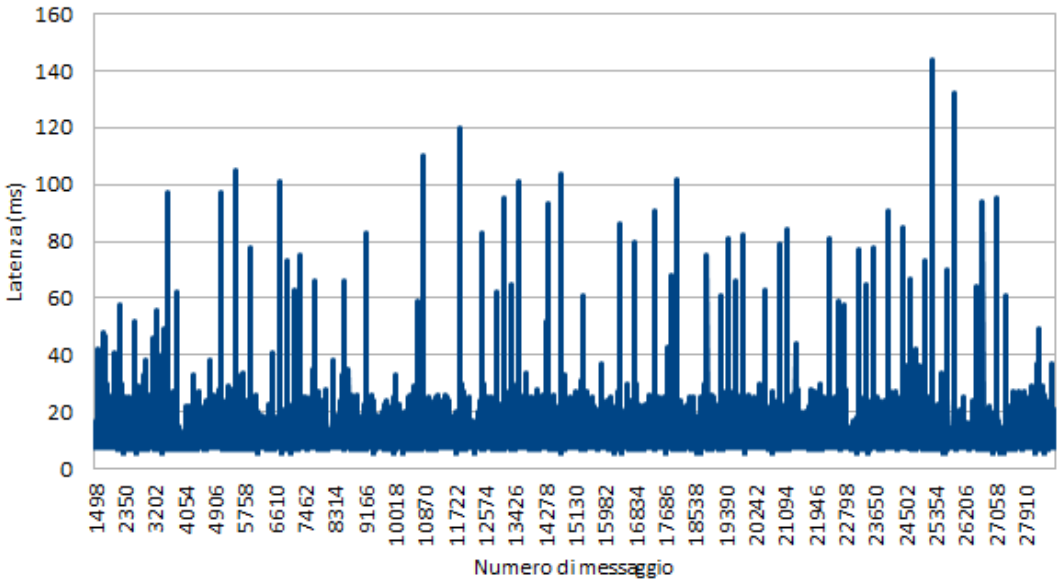

Figura 16: Tempi di latenza di trasmissione dei messaggi

4.8 Performance: Throughput di comunicazione

È stato costruito un test che, dato un producer che produce un messaggio dopo l'altro per una durata di 8 ore e un consumer, registri ogni 6 secondi la quantità e le dimensioni in byte dei messaggi prodotti e consumati.

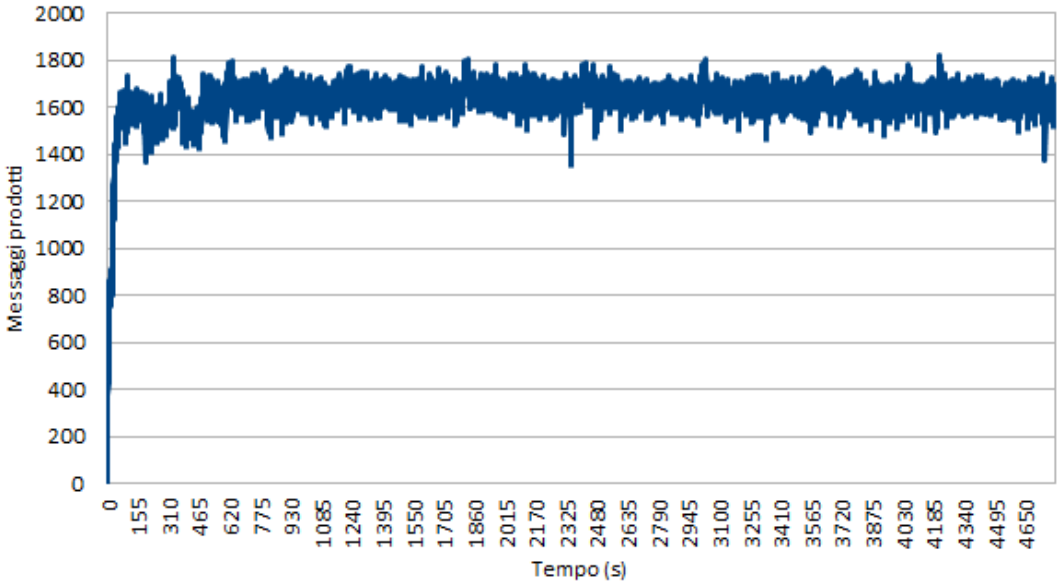

Figura 17: Grafico rappresentante la variazione della quantità di messaggi prodotti

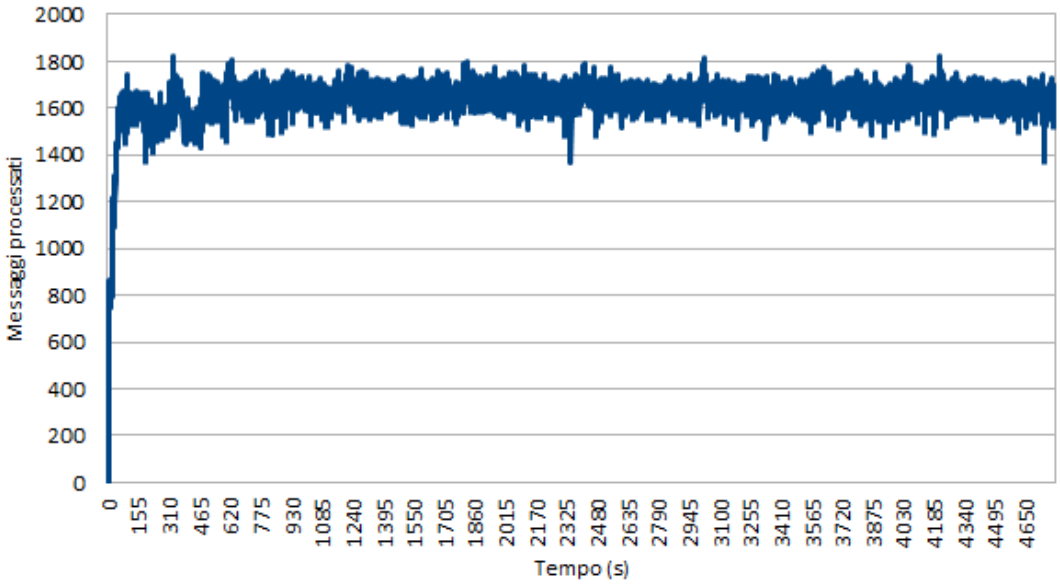

Figura 18: Grafico rappresentante la variazione della quantità di messaggi ricevuti

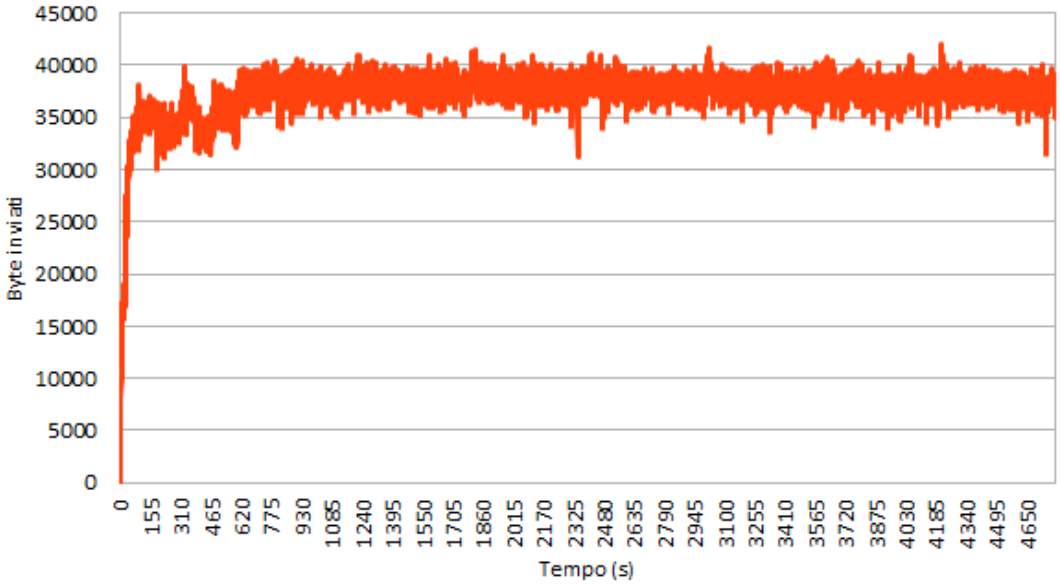

Figura 19: Grafico rappresentante la variazione della quantità di byte inviati

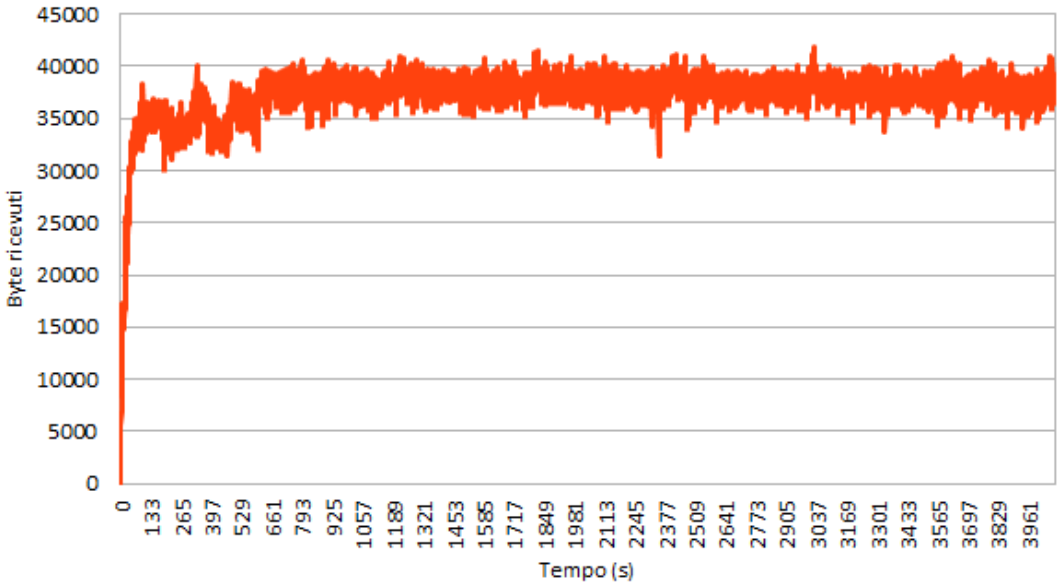

Figura 20: Grafico rappresentante la variazione della quantità di byte ricevuti

4.9 Performance: Fault-tolerance

Grazie alla replica dei messaggi su più broker, Kafka è in grado di garantire un corretto funzionamento delle operazioni di produzione e consumo anche in caso di guasti del sistema. È stata simulata la produzione di un messaggio al secondo e il relativo consumo per 8 ore alternando ogni 30 minuti l'attivazione e l'arresto di due dei tre broker utilizzati inizialmente, dimostrando che le operazioni Kafka non cessano.

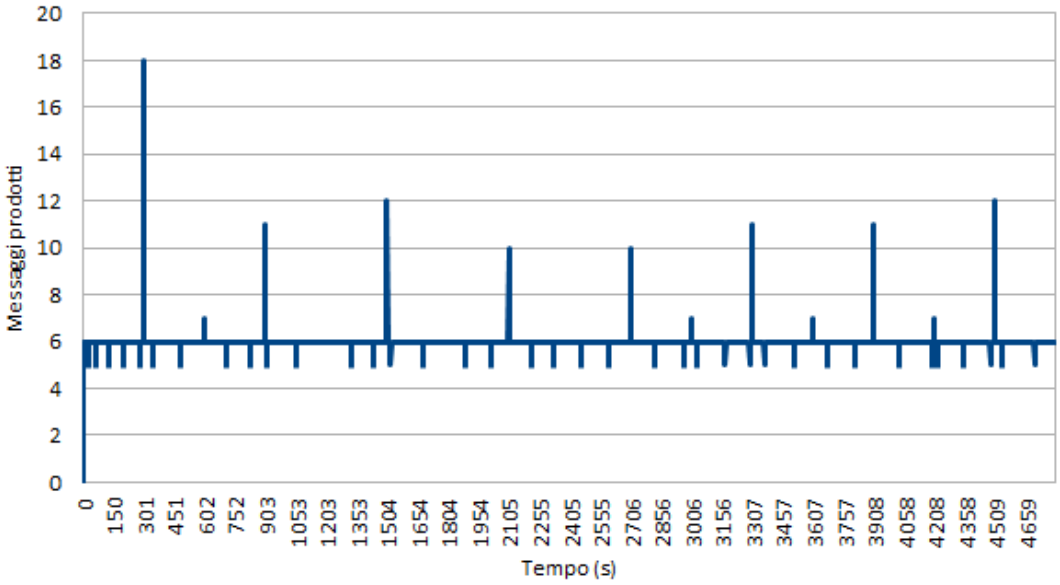

Figura 21: Grafico rappresentante la variazione della quantità di messaggi prodotti

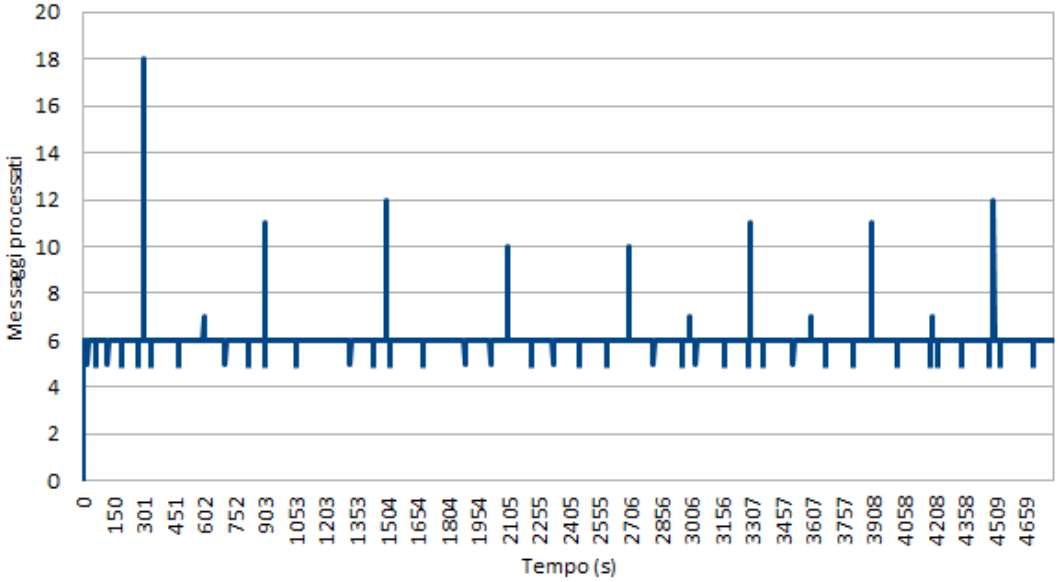

Figura 22: Grafico rappresentante la variazione della quantità di messaggi ricevuti

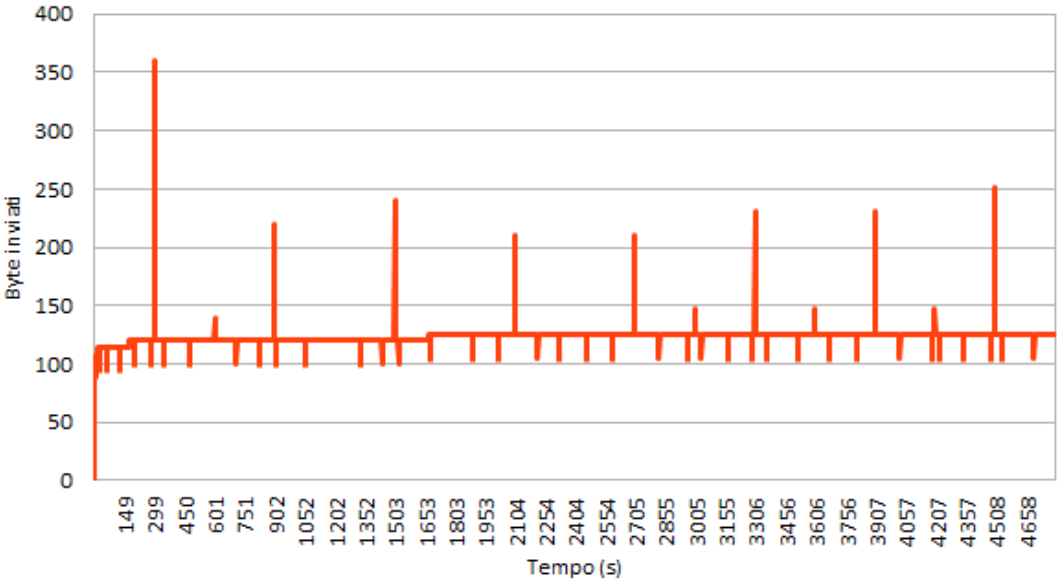

Figura 23: Grafico rappresentante la variazione della quantità di byte inviati

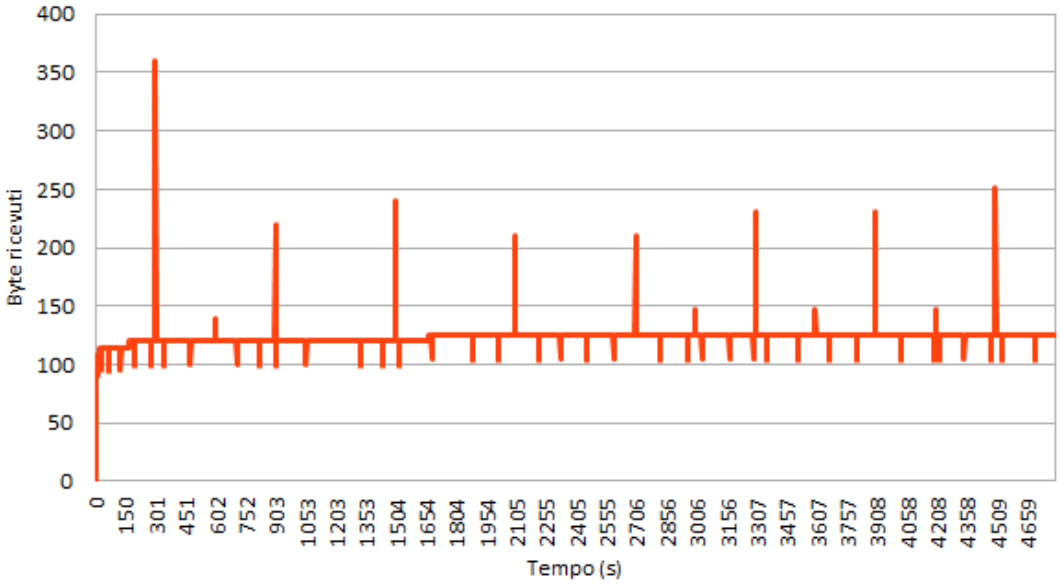

Figura 24: Grafico rappresentante la variazione della quantità di byte ricevuti

Vi è la possibilità che i messaggi prodotti non vengano replicati su tutti i broker prima della chiusura dei broker leader, portando il consumer a restare in attesa del messaggio memorizzato in almeno uno dei due broker arrestati fino alla relativa riattivazione. Finché il messaggio non può essere consumato, i nuovi messaggi verranno prodotti verso l'unico broker rimasto attivo al momento.

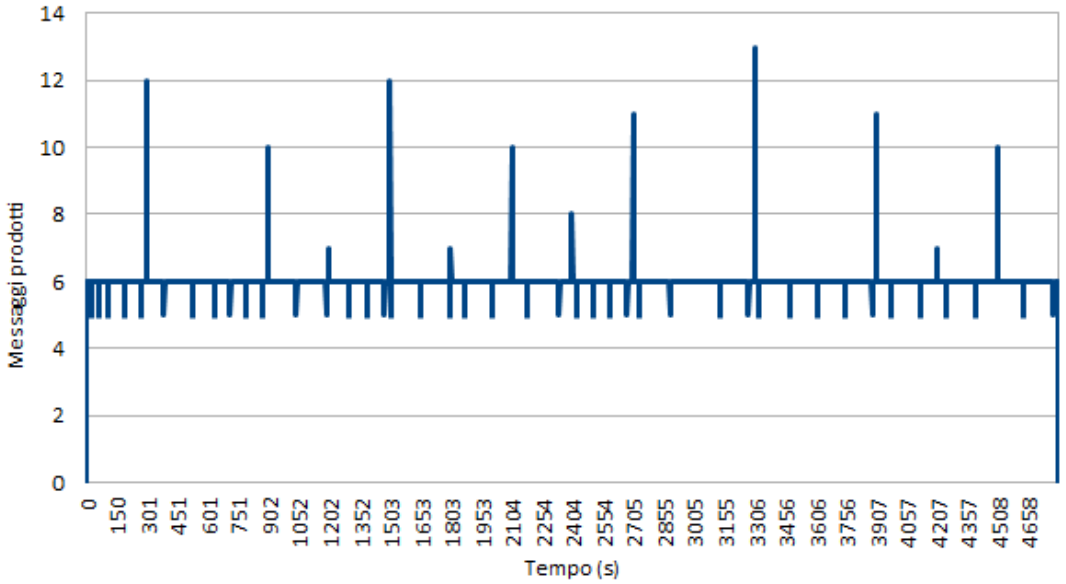

Figura 25: Grafico rappresentante la variazione della quantità di messaggi prodotti con messaggi da consumare in broker arrestati

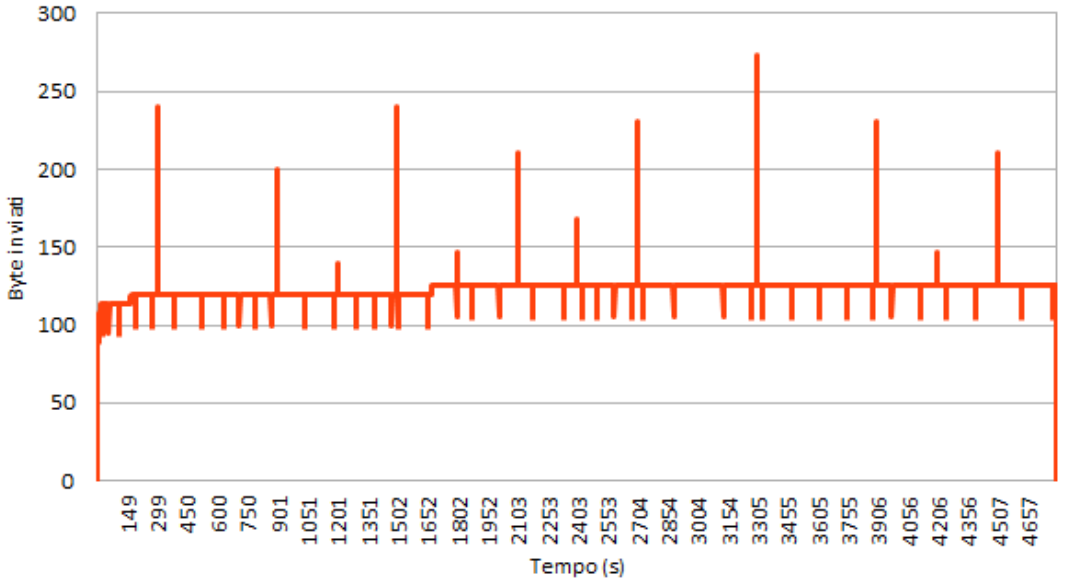

Figura 26: Grafico rappresentante la variazione della quantità di byte inviati con messaggi da consumare in broker arrestati

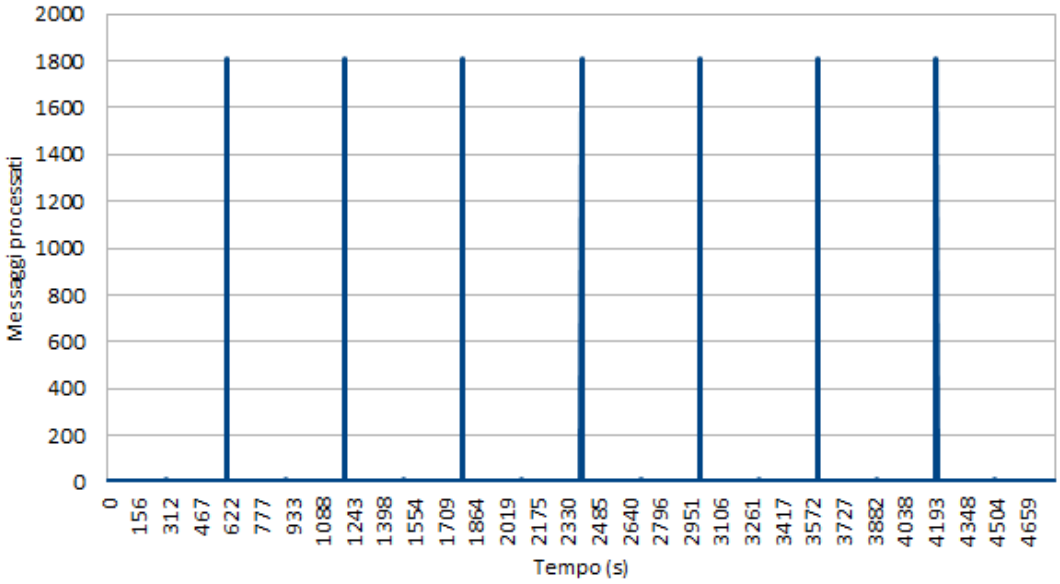

Figura 27: Grafico rappresentante la variazione della quantità di messaggi ricevuti con messaggi da consumare in broker arrestati

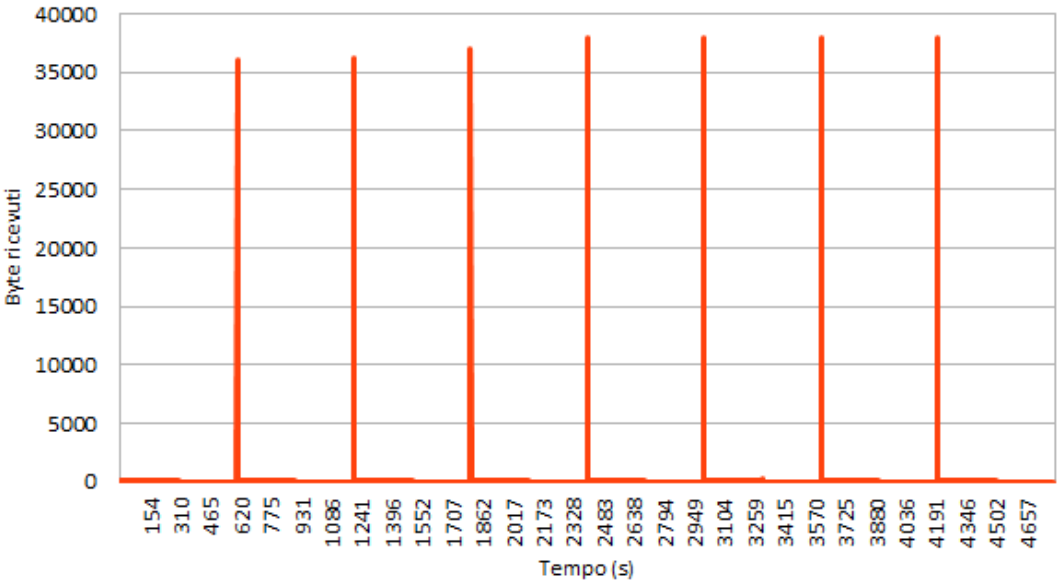

Figura 28: Grafico rappresentante la variazione della quantità di byte ricevuti con messaggi da consumare in broker arrestati

4.10 Performance: Dinamicità dell'ambiente

Si vuole testare la dinamicità dell'ambiente Kafka in termini di configurabilità. L'obiettivo è vedere quanto è possibile modificare l'ambiente senza interrompere e/o ostacolare il meno possibile le operazioni dei client e dei broker. Le proprietà di ambiente corrispondono alle proprietà dei broker, che possono essere configurate tramite file di configurazione apposito o programmaticamente (ad esempio dal terminale di Docker). Le proprietà possono essere divise per modalità di aggiornamento (Update Mode nella documentazione ufficiale) a seconda di quando e come possono essere modificate:

- read-only, proprietà che richiede il riavvio del broker per avere efficacia;
- per-broker, proprietà che viene settata singolarmente per ciascun broker in modo dinamico;
- cluster-wide, proprietà che viene settata a livello di cluster in modo dinamico, pertanto ha effetto su tutti i broker presenti.

Poiché solo il secondo e il terzo tipo di proprietà permettono di modificare l'ambiente a run-time, è necessario che le proprietà read-only siano settate correttamente a priori, queste proprietà possono anche determinare il grado di libertà di modifica dopo l'avvio dei broker.

Dato l'insufficiente supporto a sistemi di sicurezza dato dall'immagine di default Bitnami, per questo test si è utilizzata un'immagine Confluent sia per ZooKeeper che per Kafka. Le operazioni di generazione delle certificazioni o qualsiasi altro file, il cui riferimento posizionale è necessario fra le proprietà di una entità, avvengono sul file system della Virtual Machine poiché non è possibile fare riferimento ad un file system esterno (ad esempio la root del progetto).

4.10.1 Crittografia e autenticazione con SSL

A broker già avviati viene deciso di estendere il tipo di connessioni supportate del broker numero 1 aggiungendo SSL. Vengono generate tramite un apposito script la chiave SSL e certificato all'interno della macchina Docker, configurato l'hostname a cui fare riferimento, creata la Certificate Authority e firmato il certificato[1]:

```
1  keytool -keystore server.keystore.jks -alias localhost -validity 365
   -keyalg RSA -genkey -storepass password123 -keypass password123 -dname
    'CN=alessandro.oliva@mail.it, OU=Scaldasedie, O=DisoccupatoLtd, L=
   Imola, S=Bologna, C=IT'
2  openssl req -new -x509 -keyout ca-key -out ca-cert -days 365 -passout
    pass:password123 -subj '/C=IT/ST=Bologna/L=Imola/O=DisoccupatoLtd/OU=
   Scaldasedie/CN=alessandro.oliva@mail.it'
3  keytool -noprompt -keystore server.truststore.jks -alias CARoot -
   import -file ca-cert -storepass password123
4  keytool -noprompt -keystore client.truststore.jks -alias CARoot -
   import -file ca-cert -storepass password123
```

[1]Per semplicità viene tutto gestito in chiaro, password comprese

```
5     keytool -noprompt -keystore server.keystore.jks -alias localhost -
      certreq -file cert-file -storepass password123
6     openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-
      signed -days 365 -CAcreateserial -passin pass:password123
7     keytool -noprompt -keystore server.keystore.jks -alias CARoot -import
       -file ca-cert -storepass password123
8     keytool -noprompt -keystore server.keystore.jks -alias localhost -
      import -file cert-signed -storepass password123
```

Per implementare la verifica dell'hostname viene aggiunta la proprietà `ssl.endpoint.identification.algorithm=HTTPS` nel broker. Inoltre, deve essere definita una porta specifica per SSL nella proprietà `listeners`. Percorsi e password di *keystore* e *truststore* sono definiti con le apposite proprietà.

A intervalli costanti di tempo questo certificato viene sostituito con un altro, senza compromettere l'operatività dei broker che continuano a funzionare correttamente. Viene mantenuta comunque la connessione di tipo `PLAINTEXT` in modo da permettere ai client precedenti di continuare a funzionare per motivi di semplicità e dimostrativi. Tramite `ssl.secure.random.implementation` è possibile configurare l'implementazione PRNG (pseudo-random number generator) nel caso non si voglia utilizzare quella impiegata dalla JRE/JDK di default.

4.10.2 Autenticazione con SASL

Kafka usa JAAS (Java Authentication and Authorization Service) per la configurazione SASL. Password e username possono essere configurati attraverso `listener.name.[nomeListener].[meccanismoSASL].sasl.jaas.config` tra le proprietà del broker altrimenti specificando un file di configurazione per `sasl.jaas.config`. Il meccanismo utilizzato nel test è *PLAIN*:

```
1     KafkaServer {
2   org.apache.kafka.common.security.plain.PlainLoginModule required
3   username="admin"
4   password="password123"
5   user_admin="password123"
6   user_someone="password123"
7 }
```

Attraverso `KAFKA_OPT="-Djava.security.auth.login.config=[pathToConf]"` viene impostato il path al file di configurazione JAAS tramite un parametro della JVM, specificato come proprietà del broker. Meccanismo e protocollo vanno comunque esplicitati nella configurazione del broker.

A intervalli costanti di tempo il meccanismo `PLAIN` viene sostituito con il meccanismo `SCRAM` e viceversa, senza compromettere l'operatività dei broker che continuano a funzionare correttamente. `SCRAM` viene impostato similmente a `PLAIN`, aggiungendo le credenziali utenti e amministratore per Docker[2]:

[2]Per semplicità viene tutto gestito in chiaro, password comprese

```
1     docker exec -i composers_kafka2_1 kafka-configs.sh --zookeeper
      zookeeper2:2182 --entity-type users --entity-name admin --alter --add-
      config 'SCRAM-SHA-256=[password=password123],SCRAM-SHA-512=[password=
      password123]'
2     docker exec -i composers_kafka2_1 kafka-configs.sh --zookeeper
      zookeeper2:2182 --entity-type users --entity-name user --alter --add-
      config 'SCRAM-SHA-256=[password=password123],SCRAM-SHA-512=[password=
      password123]'
3 }
```

4.10.3 Configurazione del broker

Le proprietà dei topic possono essere alterate in fase di runtime. Alcune proprietà dei broker di livello cluster-wide fanno override delle proprietà a livello di topic (in quel caso tutti i topic registrati) qualora questi non dispongano di una propria configurazione specifica. Queste proprietà riguardano quasi esclusivamente i log dei topic e sono relative a ritenzione, memoria, sincronizzazione, compressione e tempistiche.

4.10.4 Configurazione del topic

È stato fatto uso di proprietà di configurazione a livello di topic. Le proprietà non definite vengono ereditate da quelle specificate a livello di broker. Nel test sono state applicate le seguenti modifiche:

```
1     DockerTerminal.setFileDeleteDelayMs(TOPIC_A, 120000);
2   DockerTerminal.setFollowerReplicationThrottledReplicas(TOPIC_A,
      Collections.emptyList());
3   DockerTerminal.setIndexIntervalBytes(TOPIC_B, 8192);
4   DockerTerminal.setMessageDownConversionEnable(TOPIC_C, false);
```

4.10.5 Pulizia dei log

Riguarda la modalità di pulizia dei messaggi rimasti nei broker. È possibile specificare numero di thread da impiegare, quanti byte processare al secondo, memoria da impiegare, dimensione dei buffer e tempo massimo di sleep prima di pulire.

4.10.6 Configurazione thread

Riguarda la quantità di thread da allocare per determinate operazioni, come rete, I/O, fetcher delle repliche dei messaggi, recovery e operazioni di background.

4.10.7 Configurazione numero di connessioni

Riguarda la quantità di connessioni ammesse per indirizzo IP, sia a livello generale che per specifico indirizzo IP. Avviene attraverso `max.connections.per.ip` e `max.connections.per.ip.overrides`.

4.10.8 Configurazione dei listener

Riguarda l'aggiunta, modifica o rimozione dei listener, e l'aggiunta o rimozione di listener tra quelli che ZooKeeper può mettere a disposizione per i diversi client. È possibile anche ridefinire i protocolli di sicurezza che ciascuna connessione deve seguire; nel test vengono aggiunti SSL e SASL mantenendo tuttavia il supporto tramite PLAINTEXT.

5 Architettura dell'infrastruttura di test

I casi di test si appoggiano su una infrastruttura comune. Si è scelto di privilegiare la comodità in termini di facilità di implementazione per questa architettura di base.

5.1 Composizione di un test

Il test inizializza tutti i client coinvolti e tramite il metodo `getInfo()` ne riceve lo stato per registrare l'andamento del test. Può inoltre modificare l'ambiente a livello di container, inizializzando i broker, creando topic, applicando proprietà alle entità che lo popolano, ecc.

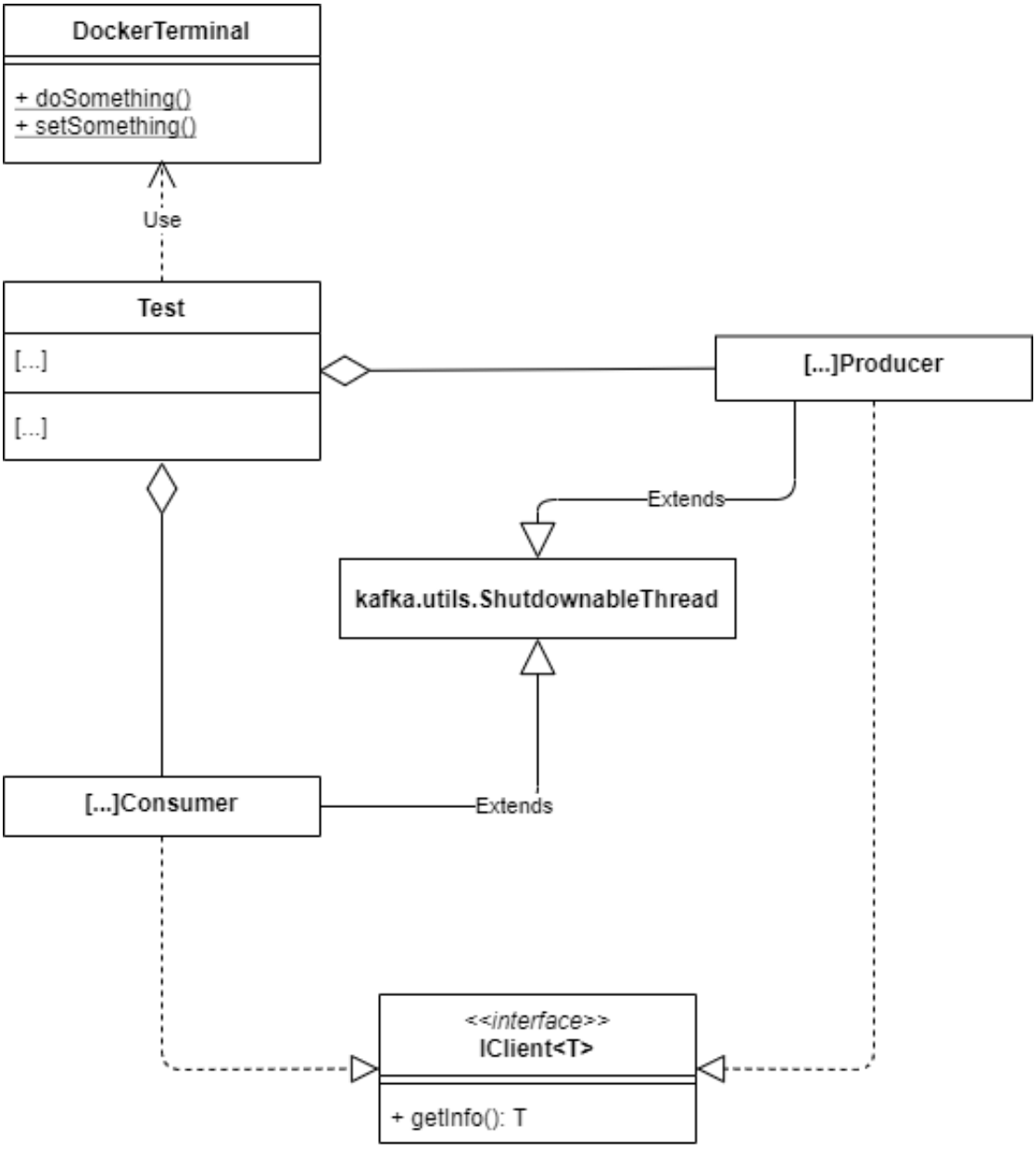

Figura 29: Composizione base di un qualsiasi caso di test

5.2 Composizione dei client

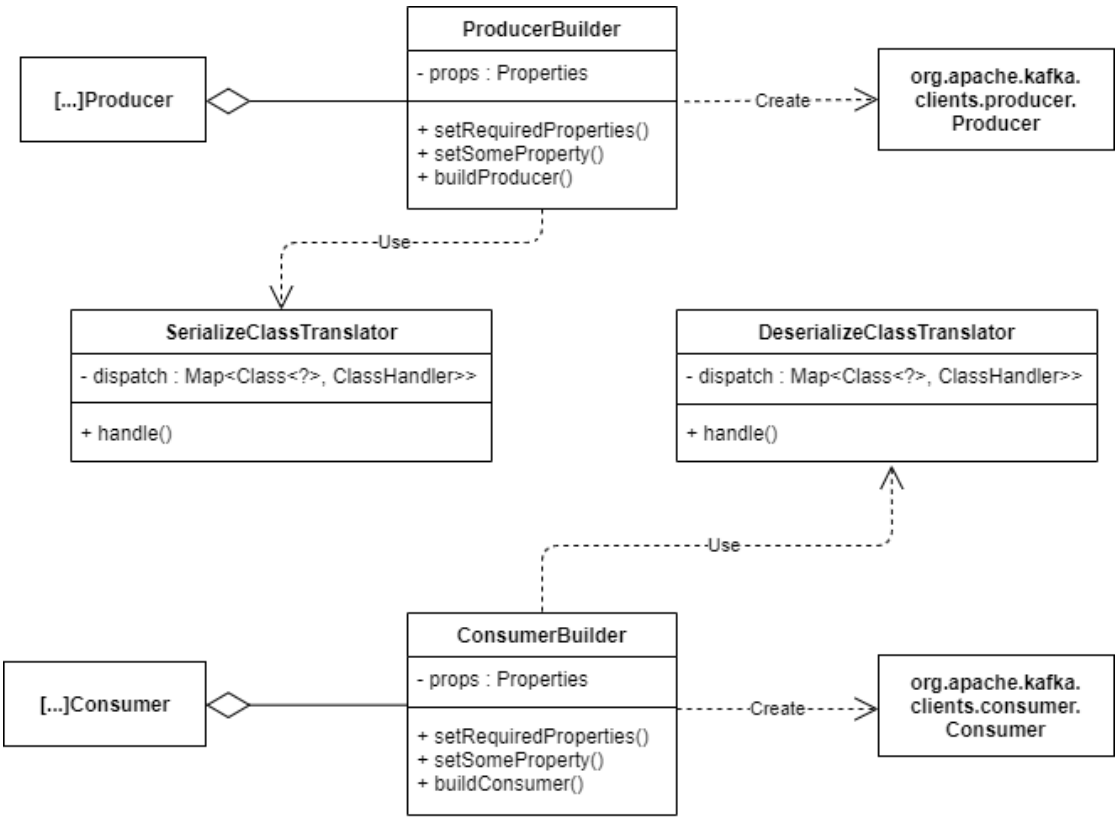

Figura 30: Composizione base dei client

I client presenti nell'omonimo package sono delle astrazioni più generali delle classi `Producer` e `Consumer` fornite da Kafka, per permettere di avere client che possono produrre diversi tipi di messaggi. Il tipo di messaggio e proprietà del client sono infatti definiti nelle classi di Kafka e diverse proprietà sono *read-only*, motivo per cui la gestione di messaggi di tipo diverso richiede più oggetti `Producer` e `Consumer` Kafka dentro lo stesso client. Le classi builder permettono di specificare le proprietà desiderate, fra cui il tipo di chiave e valore, e mediante una classe di traduzione deriverà le classi per serializzare o deserializzare i messaggi settandole a sua volta nelle proprietà del client.

5.3 Automazione da terminale

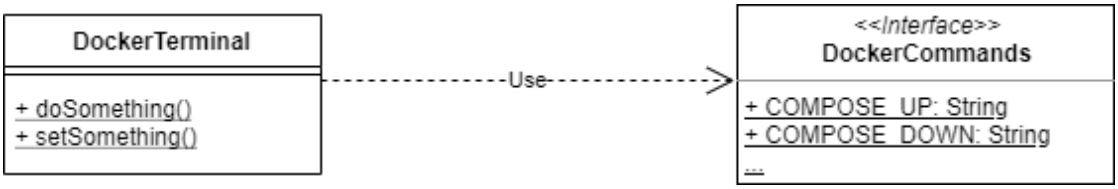

Figura 31: Classi per l'automazione da terminale

Le operazioni eseguibili da questa classe possono avvenire sia a livello locale (da macchina host verso il container) oppure a livello di Virtual Machine. Il primo tipo di operazioni è tipico per l'aggiornamento di proprietà in maniera dinamica, mentre i broker sono operativi. Il secondo tipo di operazioni riguarda ad esempio la creazione di certificati di sicurezza sul file system del container Docker. La classe `DockerCommands` contiene delle macro per i metodi usati in `DockerTerminal`.

6 Conclusioni

Apache Kafka è una tecnologia solida, però non applicabile in qualunque contesto.

6.1 Pro

- **Integrazione**: producer e consumer possono essere scritti in diversi linguaggi di programmazione, grazie alle molteplici librerie disponibili fra cui progetti open-source. La scelta di un certo linguaggio può essere dettata da diverse esigenze, che portano il client ad avere uno specifico comportamento. Si possono quindi avere modi particolari di gestire i messaggi da e/o verso Kafka, come avviene in 4.5.
- **Throughput elevato**: Kafka può gestire grandi volumi di dati ad alta velocità come dimostrato in 4.2 e 4.8. Viene sfruttata la cache del sistema operativo e l'ordine sequenziale di scrittura dei dati. Per aumentare ulteriormente le prestazioni, Kafka memorizza i record ricevuti in una cache in memoria del sistema operativo detta *page cache*, che deciderà quando scrivere il record su disco. I dati inviati ai consumer verranno prelevati direttamente dalla cache utilizzando il principio della memoria a copia zero[18];
- **Bassa latenza**: la consegna dei messaggi può avvenire in termini di millisecondi in quanto, con le configurazioni adeguate, è possibile adottare una semantica di consegna at-most-once e non avere latenza dovuta all'invio dei messaggi di acknowledgement al producer da parte del broker, come dimostrato in 4.6. Questa è una valida ottimizzazione per le operazioni di log aggregation, in quanto i dati devono essere consegnati in modo asincrono per evitare di congestionare il traffico; in questo caso, è preferita la perdita di un messaggio alla sua consegna ripetuta[8];

- **Real-time**: data la bassa latenza (anche nonostante una diversa distribuzione fisica degli host come avvenuto in 4.5) si possono avere ottimi risultati anche per applicazioni time-sensitive;
- **Persistenza**: Kafka ammette di base la persistenza dei messaggi, grazie al meccanismo della ritenzione;
- **Tolleranza ai guasti (fault-tolerant)**: per fronteggiare eventuali guasti in una o più parti del sistema, Kafka replica i dati in più partizioni di broker diversi, come avviene in 4.9. Il numero di copie è determinato dal cosiddetto *replication factor*;
- **Distribuito**: la natura intrinseca di sistema distribuito permette a Kafka di essere scalabile attraverso il partizionamento e la replica dei dati, ed è ciò che permette anche la già citata tolleranza ai guasti;
- **Durabilità**: la persistenza su disco per un determinato periodo di ritenzione e la replica su diversi broker assicurano che, in caso di guasti, un messaggio possa essere comunque accessibile senza perdite di dati. Quando un broker è guasto (offline), i messaggi ricevuti sono comunque mantenuti, e saranno reperibili quando il broker tornerà operativo (come avviene nel test descritto in 4.9);
- **Elevata scalabilità**: un sistema Kafka può essere ridimensionato senza interruzioni aggiungendo broker a cluster già esistenti e operativi;
- **Altamente concorrente**: permette di gestire centinaia di messaggi al secondo in condizioni di bassa latenza e alta produttività;
- **Gestione di batch**: data la persistenza dei messaggi, Kafka può essere utilizzato per funzionalità ETL;
- **Broker basilari**: sopperisce a qualsiasi bisogno soddisfatto dai broker classici, pertanto l'impiego è conveniente anche per applicazioni con sistemi di messaggistica basilari, non rendendo necessario lo sviluppo di sistemi di broker da zero;
- **Adatto per i data lake**: operazioni su data lake come log aggregation e tracking di attività web sono nativamente supportate ad un livello molto esteso.

6.2 Contro

- **Problemi con il tweaking dei messaggi**: l'unica modalità ammessa di tweaking dei messaggi è l'impiego di un client intermedio che rielabori il messaggio e lo invii a sua volta a destinazione. L'implementazione di questo pattern tuttavia implica una riduzione di performance, a partire dalla latenza (vedi 4.7);
- **Messaggi pesanti difficilmente gestibili**: messaggi di dimensioni troppo elevate portano alla riduzione della produttività e hanno un impatto significativo sulle operazioni. Questo avviene perché Kafka cercherà di comprimere il messaggio a

livello di producer e consumer, rallentando qualsiasi altra operazione. Dividere un messaggio pesante in messaggi più leggeri è preferibile[19];

- **Rischi di produzione**: l'utilizzo di molteplici consumer può rallentare le performance di produzione e aumentare il rischio di cache miss;
- **Spreco di risorse di computazione**: i consumer Kafka devono necessariamente leggere tutto il record anche solo per accedere a informazioni specifiche[20];
- **Performance incostanti**: spesso in termini di throughput e latenza si possono avere picchi di performance molto più elevati della media o variazioni in negativo. Nel caso sia necessaria una comunicazione più precisa e uniforme sono necessari interventi aggiuntivi di normalizzazione delle prestazioni;
- **Paradigmi implementabili limitati**: paradigmi base come *request/reply* o code *point-to-point* non sono nativamente supportati. Questi paradigmi possono comunque essere implementati utilizzando in modo particolari i client Kafka. Ciò implica che per certe applicazioni Kafka potrebbe non essere la tecnologia adatta;
- **Tool di monitoring insufficienti**: un sistema già in funzione è piuttosto opaco, ed è quindi difficile esaminarne le proprietà senza avere conoscenze pregresse del sistema. Spesso in fase di test ricreare da zero un broker riscrivendo tutte le proprietà è stata una soluzione più appetibile rispetto alla ricerca e correzione dell'errore, che ha quasi sempre richiesto un approccio per tentativi;
- **Implementazioni non a pari passo**: le API sono mantenute da diversi individui o aziende.

7 Appendice

Di seguito vengono mostrati ulteriori grafici relativi ad alcuni dei test descritti nella sezione apposita.

7.1 Performance: Throughput di comunicazione

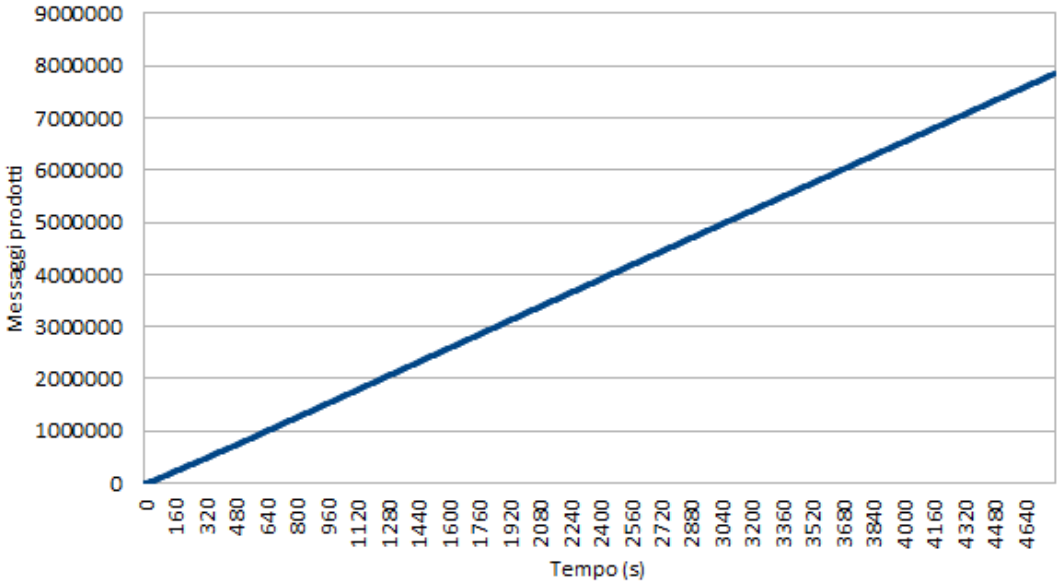

Figura 32: Grafico rappresentante l'andamento dei messaggi prodotti

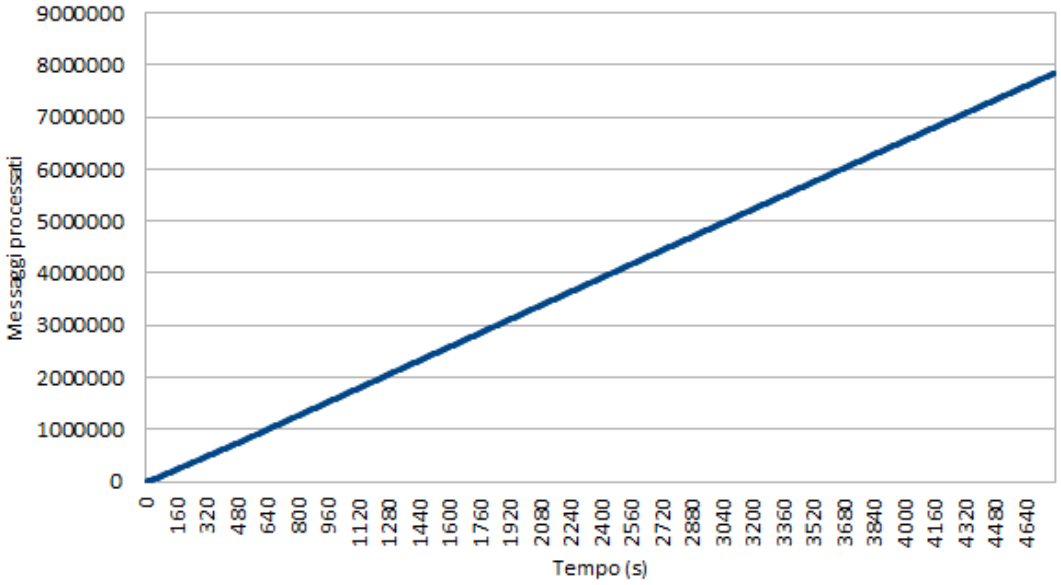

Figura 33: Grafico rappresentante l'andamento dei messaggi ricevuti

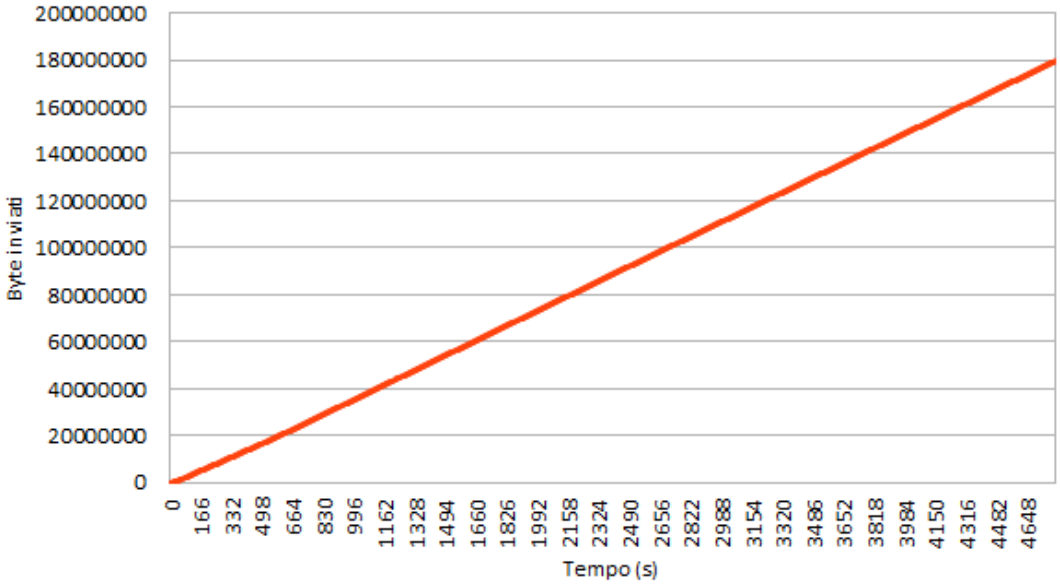

Figura 34: Grafico rappresentante l'andamento dei byte inviati

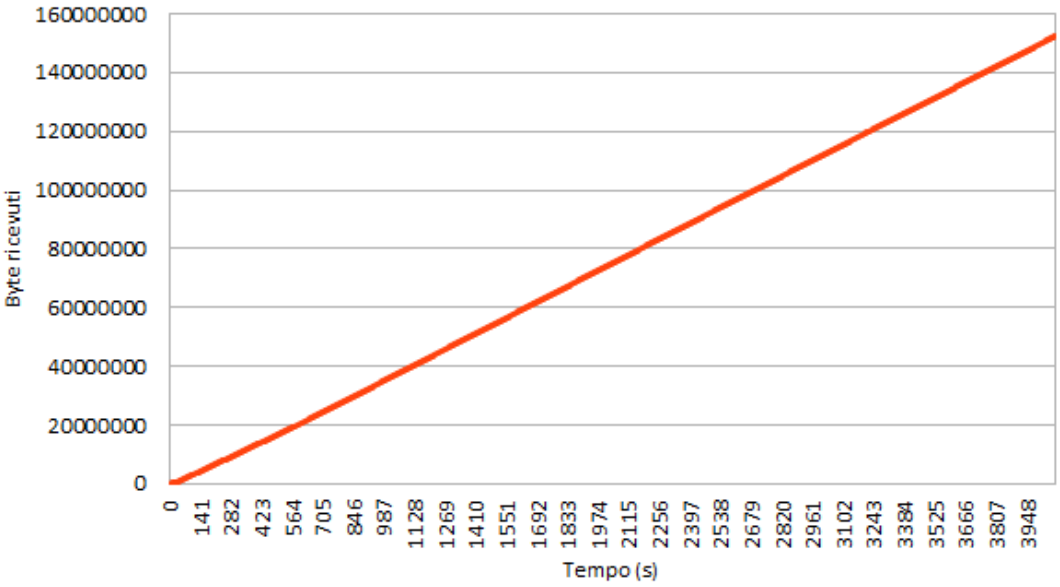

Figura 35: Grafico rappresentante l'andamento dei byte ricevuti

7.2 Performance: Fault-tolerance

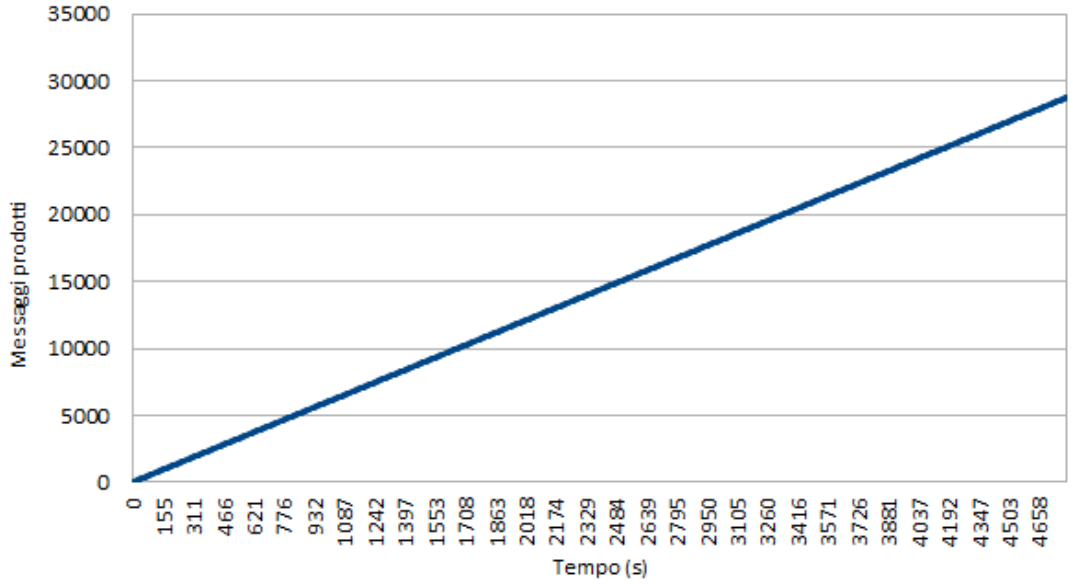

Figura 36: Grafico rappresentante l'andamento dei messaggi prodotti

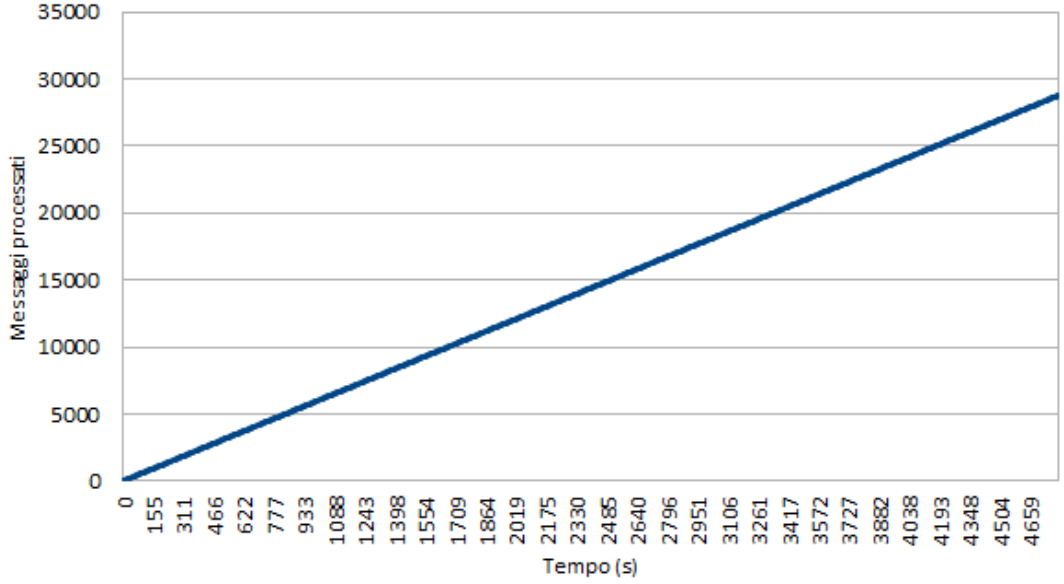

Figura 37: Grafico rappresentante l'andamento dei messaggi ricevuti

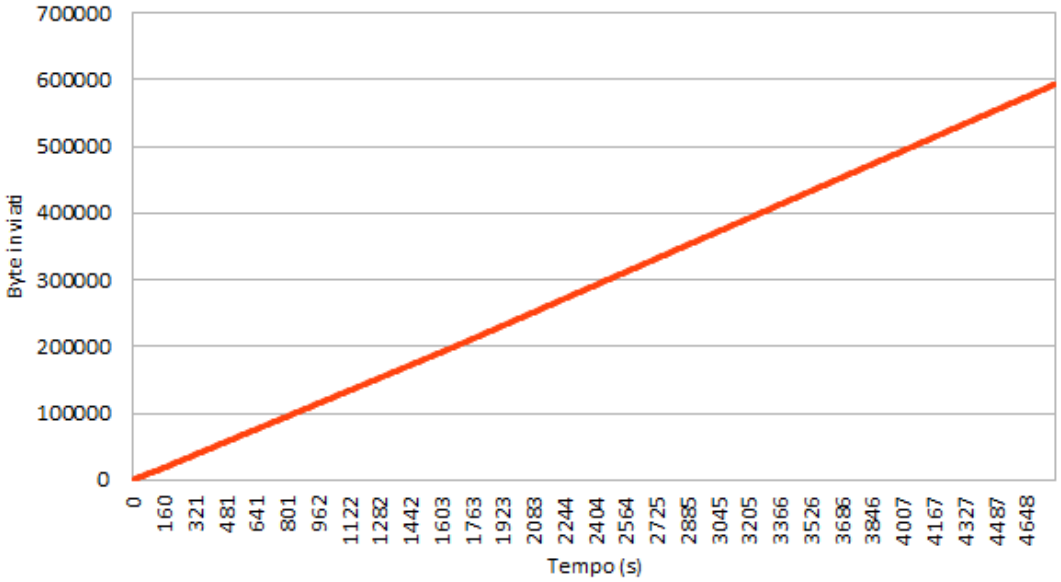

Figura 38: Grafico rappresentante l'andamento dei byte inviati

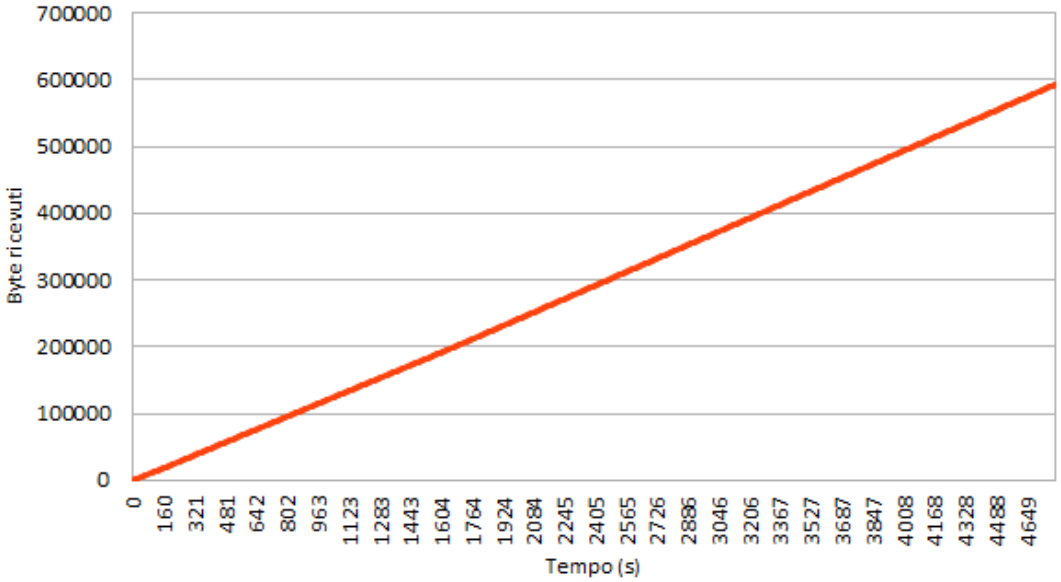

Figura 39: Grafico rappresentante l'andamento dei byte ricevuti

7.3 Performance: Fault-tolerance (caso con messaggi in broker arrestati)

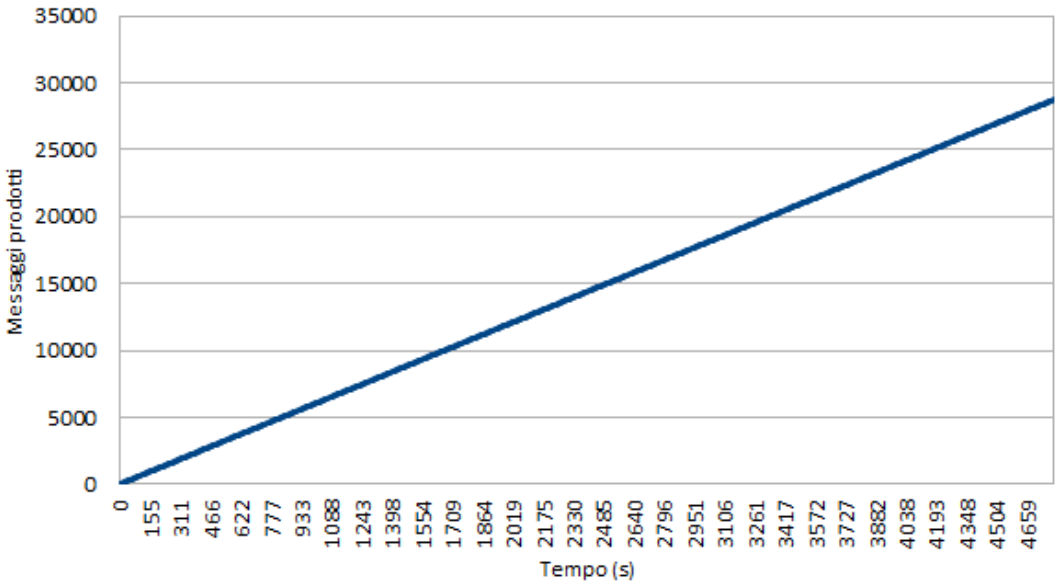

Figura 40: Grafico rappresentante l'andamento dei messaggi prodotti con messaggi da consumare in broker arrestati

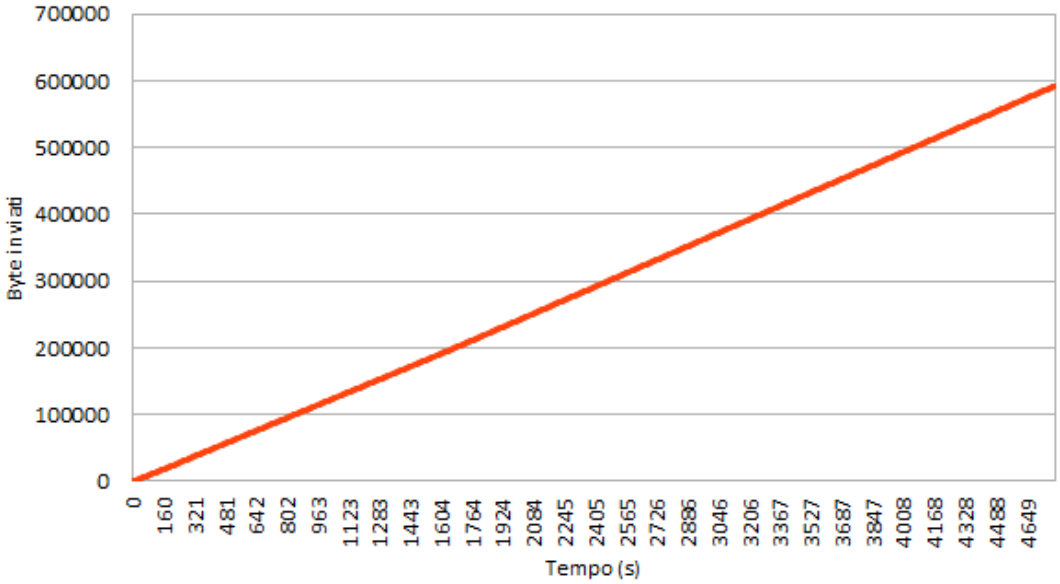

Figura 41: Grafico rappresentante l'andamento dei byte inviati con messaggi da consumare in broker arrestati

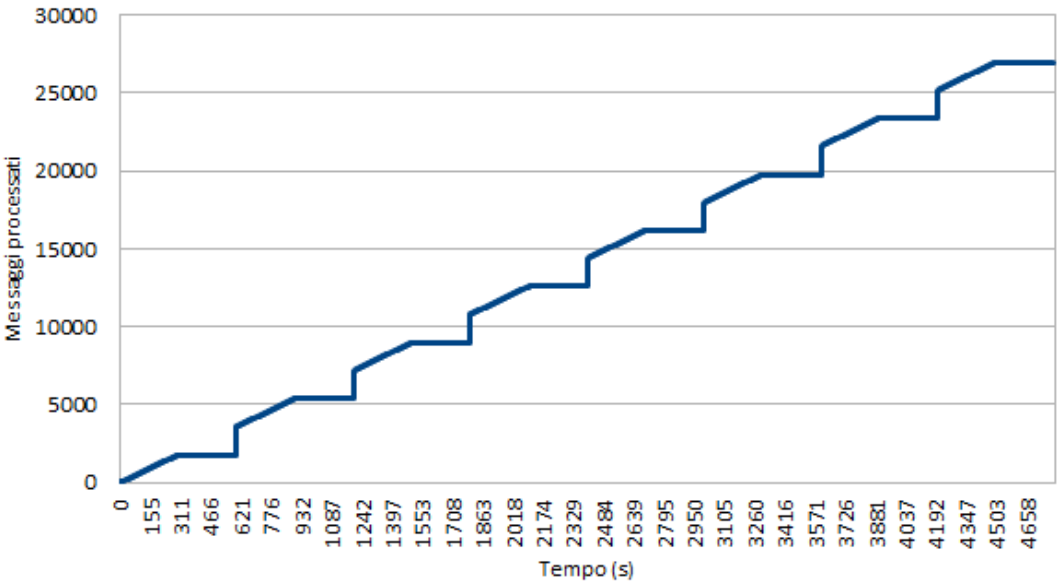

Figura 42: Grafico rappresentante l'andamento dei messaggi ricevuti con messaggi da consumare in broker arrestati

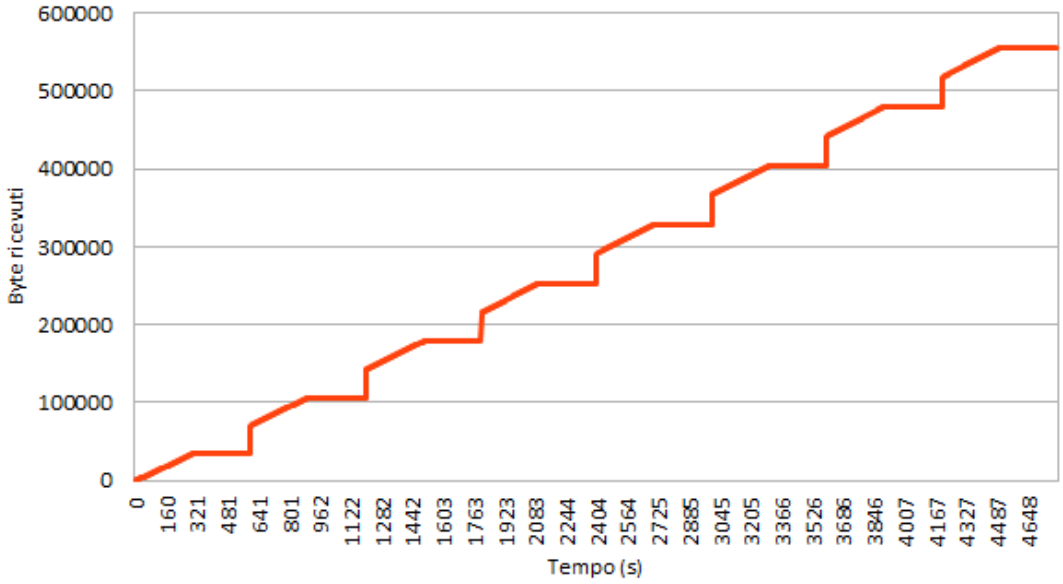

Figura 43: Grafico rappresentante l'andamento dei byte ricevuti con messaggi da consumare in broker arrestati

Riferimenti bibliografici

[1] S. Perera, "What is stream processing? a gentle introduction." https://dzone.com/articles/what-is-stream-processing-a-gentle-introduction, aprile 2018.

[2] "Log aggregation 101: A complete guide, from how it works to the tools you must know about." https://sematext.com/blog/log-aggregation/#toc-what-is-log-aggregation-a-simpler-definition–0.

[3] J. Rao, "Open-sourcing kafka, linkedin's distributed message queue." https://blog.linkedin.com/2011/01/11/open-source-linkedin-kafka, gennaio 2011.

[4] "What is apache kafka?." http://cloudurable.com/blog/what-is-kafka/index.html, maggio 2017.

[5] "Schema management." https://docs.confluent.io/current/schema-registry/index.html, dicembre 2019.

[6] "Introduction." https://kafka.apache.org/intro.

[7] "Documentation." https://kafka.apache.org/documentation/.

[8] J. Kreps, N. Narkhede, and J. Rao, "Kafka: a distributed messaging system for log processing," tech. rep., LinkedIn Corp., 2011.

[9] "Core concepts." https://kafka.apache.org/23/documentation/streams/core-concepts.

[10] "Architecture." https://kafka.apache.org/23/documentation/streams/architecture.html.

[11] "Processor api." https://kafka.apache.org/24/documentation/streams/developer-guide/processor-api.html.

[12] "Streams dsl." https://kafka.apache.org/23/documentation/streams/developer-guide/dsl-api.

[13] "Kafka connect." https://docs.confluent.io/current/connect/index.html.

[14] E. Vinka, "What is zookeeper and why is it needed for apache kafka?." https://www.cloudkarafka.com/blog/2018-07-04-cloudkarafka_what_is_zookeeper.html, luglio 2018.

[15] "difference between ensemble and quorum in zookeeper." https://stackoverflow.com/questions/25174622/difference-between-ensemble-and-quorum-in-zookeeper/45212088.

[16] N. Garg, *Apache Kafka*. Packt Publishing, ottobre 2013.

[17] G. Maggio, "Progettazione di un sistema per l'analisi di dati real-time," Master's thesis, Politecnico di Torino - Ingegneria Informatica, 2018.

[18] K. Pi, "How kafka achieves high throughput low latency." https://www.pixelstech.net/article/1552056773-How-Kafka-achieves-high-throughput-low-latency, marzo 2018.

[19] T. John and P. Misra, *Data Lake for Enterprises*, pp. 237–240. Packt Publishing, maggio 2017.

[20] "Apache kafka with and without a data lake." https://www.upsolver.com/blog/blog-apache-kafka-and-data-lake.