

Apache Kafka VS Apache Pulsar VS Rabbit MQ: un'analisi tecnica

Acampora Andrea
`andrea.acampora@studio.unibo.it`

Accursi Giacomo
`giacomo.accursi@studio.unibo.it`

25 gennaio 2024

Indice

1	Obbiettivi del progetto	2
2	Introduzione alle tecnologie	3
2.1	Apache Kafka	3
2.2	Apache Pulsar	5
2.3	RabbitMQ	7
3	Tecnologie a confronto	10
3.1	Componenti e protocolli utilizzati	10
3.1.1	Architettura generale	10
3.1.2	Protocolli di comunicazione	14
3.1.3	Sicurezza	15
3.2	Persistenza dei messaggi	16
3.3	Tolleranza ai guasti e disponibilità	17
3.4	Gestione del carico di lavoro	22
3.5	Modello di consegna dei messaggi	25
3.5.1	Pattern di comunicazione	25
3.5.2	Modello di consumo dei messaggi	25
3.5.3	Ordinamento e riconsegna dei messaggi	26
3.5.4	Semantica e garanzie di consegna dei messaggi	27
3.6	Schema dei messaggi	28
4	Progettazione e implementazione dei test	32
5	Analisi dei risultati	34
5.1	Facilità d'uso	34
5.2	Latenza	35
5.3	Throughput	38
5.4	Tolleranza ai guasti	39
6	Conclusioni finali	41

Capitolo 1

Obbiettivi del progetto

L'obiettivo di questo progetto è effettuare un confronto tecnico fra i più popolari message broker presenti sul mercato. In primo luogo verrà effettuata una panoramica teorica delle caratteristiche e delle funzionalità messe a disposizione da ciascun message broker. Successivamente verranno confrontate le loro architetture, il modello dei dati, il modello di consegna dei messaggi e i protocolli utilizzati. La fase pratica del progetto coinvolgerà la creazione di test per valutare e confrontare i message broker dal punto di vista delle performance, scalabilità, resilienza e facilità d'uso.

Capitolo 2

Introduzione alle tecnologie

2.1 Apache Kafka

Apache Kafka è una piattaforma distribuita per lo streaming di eventi, utilizzata per la creazione di pipeline di dati in tempo reale e applicazioni di streaming. Sebbene sia principalmente considerato come una coda di messaggi, il suo fondamento si basa sull'astrazione di un commit log distribuito.

Sin dal suo debutto come progetto open source nel 2011 da parte di LinkedIn, Kafka ha attraversato una notevole evoluzione: da una semplice coda di messaggi a una piattaforma complessa e completa per lo streaming di eventi. Lo streaming di eventi raccoglie i dati, li memorizza per l'elaborazione e li invia alla destinazione desiderata. Questo garantisce il flusso di informazioni necessario e la consegna tempestiva. Kafka implementa lo streaming di eventi utilizzando tre caratteristiche principali: memorizzazione flussi di eventi, elaborazione flussi di eventi e funzionalità di publish/subscribe per i sistemi.

Kafka è una soluzione distribuita che utilizza due tipi di nodi, così come come l'architettura generale publish/subscribe. Si tratta di server e client. I server possono essere eseguiti su uno o più dispositivi e l'architettura è separata in livelli, con il livello di archiviazione che funziona come Broker. I server eseguono Kafka Connect per elaborare i dati in entrata e in uscita. Questo rende Kafka altamente scalabile e implementa un'alta tolleranza ai guasti. Kafka Connect garantisce la scalabilità e l'affidabilità dello streaming dei dati.

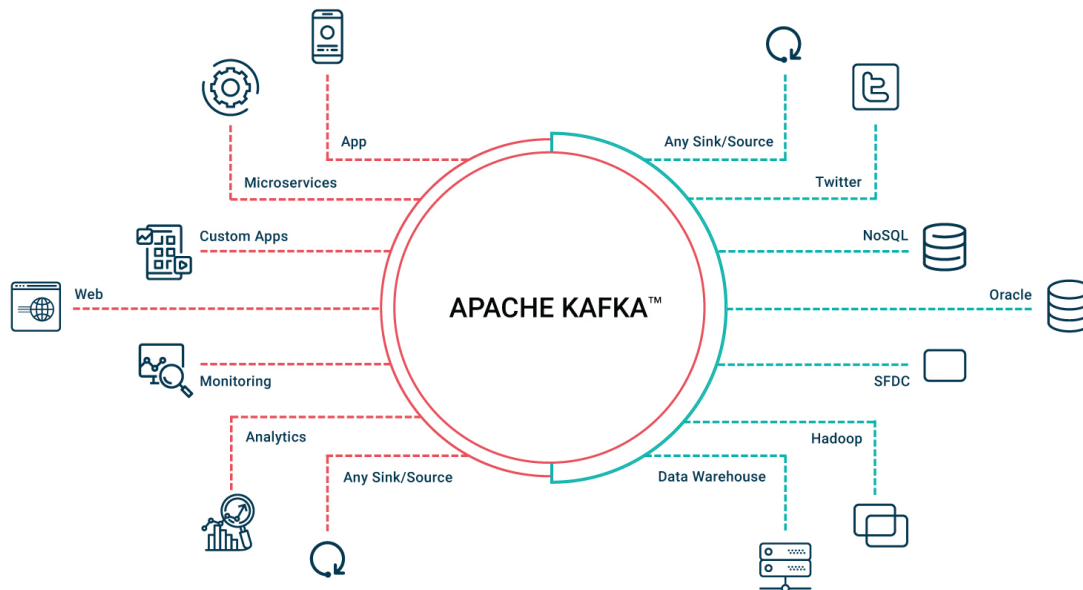


Figura 2.1: Possibili utilizzi di Apache Kafka.

I client utilizzano i server per comunicare sfruttando i topics. I principali client Kafka sono disponibili in Java e Scala. Le librerie per i client sono tuttavia disponibili in molti altri linguaggi. Kafka memorizza gli eventi o i messaggi anche dopo il consumo, a differenza dei sistemi di messaggistica tradizionali. L'archiviazione dei dati non influisce sulle prestazioni, esse rimangono pressoché costanti con dati di varie dimensioni.

I topics sono divisi in partizioni. La divisione consente operazioni di lettura e scrittura simultanee su uno o più broker. Un nuovo messaggio è sempre aggiunto alla sezione dei topic. La chiave dell'evento raggruppa gli eventi o i messaggi. Questo garantisce che i messaggi sui topics con la stessa chiave siano sempre letti nello stesso ordine con il quale sono stati inviati. Kafka permette la replica dei topic rendendo l'intero deployment ancora più tollerante ai guasti.

Kafka può essere usata per diversi scopi, tra cui:

- **Messaging:** Kafka funziona bene come sostituto di un broker di messaggi più tradizionale.
- **Tracking delle attività sui siti web:** in origine Kafka è stato progettato proprio per poter ricostruire una pipeline di monitoraggio delle attività degli

utenti come un insieme di feed publish-subscribe in tempo reale. Le attività sul sito (visualizzazioni di pagine, ricerche o altre azioni che gli utenti possono compiere) vengono pubblicate su un topic centrale, con un topic per ogni tipo di attività.

- **Metriche:** Kafka viene spesso utilizzato per la gestione dei dati per il monitoraggio operativo. Si tratta di aggregare le statistiche delle applicazioni distribuite per produrre feed centralizzati di dati operativi.
- **Aggregazione di log:** i log vengono raccolti dai server e vengono collocati in un posto centrale (un file server o HDFS) per l'elaborazione. Kafka elimina i dettagli dei file e fornisce un'astrazione più pulita dei dati dei log o degli eventi come flusso di messaggi. Ciò consente una elaborazione a bassa latenza e un supporto più semplice per più fonti di dati e per il consumo distribuito di essi.
- **Stream processing:** Molti utenti di Kafka elaborano i dati in pipeline di elaborazione composte da più fasi, in cui i dati grezzi in ingresso vengono consumati dai topics di Kafka e poi aggregati, arricchiti o altrimenti trasformati in nuovi topics per un ulteriore consumo o una successiva elaborazione.
- **Event Sourcing:** L'event sourcing è uno stile di progettazione delle applicazioni in cui i cambiamenti di stato vengono registrati come una sequenza di record ordinati nel tempo. Il supporto di Kafka per dati di log memorizzati molto grandi lo rende un backend eccellente per un'applicazione costruita in questo stile.
- **Commit Log:** Kafka può servire come una sorta di registro di commit esterno per un sistema distribuito. Il log aiuta a replicare i dati tra i nodi e agisce come un meccanismo di risincronizzazione per i nodi falliti per ripristinare i loro dati.

2.2 Apache Pulsar

Apache Pulsar è una piattaforma di messaggistica e streaming publish-subscribe sviluppata da *Yahoo* ed assorbita poi da *Apache Foundation* fino a diventare uno dei top 10 progetti dell'azienda con una comunità di utenti vasta e appassionata che spazia dalle piccole aziende alle grandi imprese. Con il crescente aumento della quantità di dati generati in tempo reale da applicazioni, sensori e dispositivi connessi, la necessità di una soluzione robusta per la gestione di flussi di dati in tempo reale è diventata cruciale. Apache Pulsar si propone di soddisfare questa esigenza,

fornendo una piattaforma che consente la trasmissione affidabile, la consegna garantita e l'elaborazione di dati in tempo reale. L'obiettivo di questa piattaforma è appunto la gestione di grandi quantità di messaggi scambiati su appositi canali fornendo una serie di funzionalità quali:

- **Affidabilità e Garanzia di Consegna:** Pulsar è progettato per garantire l'affidabilità e la consegna garantita degli eventi. Questo è cruciale in scenari in cui la perdita di dati non è accettabile e la consistenza è fondamentale.
- **Multy-tenancy:** Pulsar supporta il multitenancy, consentendo a più utenti o applicazioni di condividere la stessa istanza senza interferenze. Ciò offre una maggiore flessibilità nell'implementazione e nell'isolamento delle risorse.
- **Scalabilità orizzontale:** Pulsar scala orizzontalmente per gestire l'aumento del carico. Con la sua architettura distribuita, Pulsar consente di espandere orizzontalmente il sistema per gestire crescenti volumi di dati senza compromettere le prestazioni.
- **Geo-replicazione dei dati:** è possibile ottenere protezione contro le interruzioni in una zona grazie alla replicazione in diverse regioni geografiche. Si possono creare strategie di replica flessibili e configurabili tra cluster Pulsar distanti e inoltre viene supportato in modo unico il failover automatico dei client verso cluster sani.
- **Bassa latenza:** consente casi d'uso come code di lavoro distribuite o flussi di dati che preservano l'ordine su scale molto grandi (centinaia di nodi) e a bassa latenza (< 10 ms).
- **Supporto ufficiale multi-linguaggio:** il client di Pulsar viene attualmente mantenuto per numerosi linguaggi come Java, Go, Python, C++, Node.js e C#.
- **Serverless Functions:** è possibile scrivere e sviluppare funzioni in modo nativo utilizzando le *Pulsar Functions*. Queste funzioni si occupano di leggere e consumare i messaggi da uno o più topic, applicano una logica di elaborazione che deve essere fornita dall'utente e successivamente i risultati del calcolo vengono pubblicati su altri topics.
- **Load balancing automatico:** è possibile aggiungere o rimuovere nodi e lasciare a Pulsar la responsabilità di gestire il carico sui topics. I topics più frequentati verranno suddivisi e distribuiti in modo uniforme in diversi cluster.

Pulsar si basa sul modello *publish-subscribe*. In questo schema, illustrato in Figura 2.2 i produttori pubblicano messaggi su topic, i consumatori si iscrivono ad essi, elaborano i messaggi in arrivo e infine inviano una conferma al broker al termine dell'elaborazione.

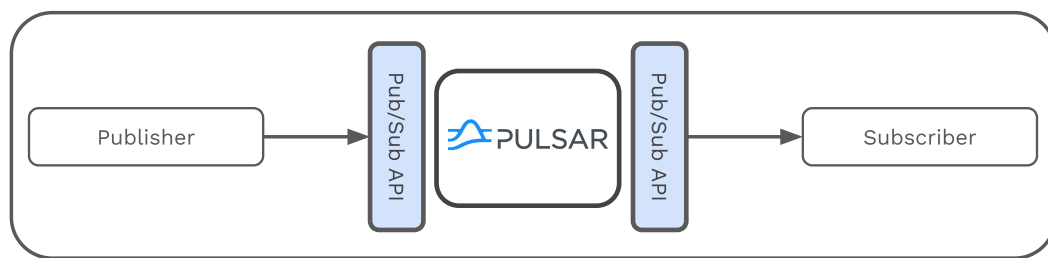


Figura 2.2: Schema del sistema di scambio di messaggi di Apache Pulsar.

2.3 RabbitMQ

RabbitMQ è un sistema di messaggistica open-source ampiamente utilizzato per la trasmissione di messaggi tra applicazioni. Creato in Erlang e basato sul modello di messaggistica "Message Queue", RabbitMQ offre una soluzione robusta per la comunicazione asincrona e la gestione di code di messaggi. RabbitMQ è spesso utilizzato in ambienti distribuiti in cui diverse parti di un'applicazione devono comunicare tra loro in modo asincrono. Questa capacità consente una maggiore flessibilità e resilienza nel design delle applicazioni. Un aspetto chiave di RabbitMQ è il concetto di "decoupling" o disaccoppiamento. Questo significa che le applicazioni possono inviare e ricevere messaggi senza la necessità di conoscere i dettagli implementativi dell'altra parte. Ciò semplifica l'integrazione e favorisce la modularità. Inoltre, RabbitMQ è progettato per essere scalabile, gestendo carichi di lavoro crescenti senza compromettere l'affidabilità e rendendolo così adatto a scenari in cui la dimensione e la complessità dell'applicazione possono variare nel tempo. Alcuni concetti chiave presenti in RabbitMQ sono:

- **Produttore:** In RabbitMQ, un produttore è un'applicazione o un modulo che invia messaggi a un exchange. I produttori sono responsabili di inoltrare informazioni o richieste al sistema, che verranno poi elaborate dai consumatori.
- **Consumatore:** Un consumatore è un'applicazione o un modulo che riceve e processa i messaggi provenienti dalle code di RabbitMQ. I consumatori sono spesso progettati per eseguire specifiche azioni o elaborazioni in risposta ai messaggi che ricevono dai produttori.
- **Code di Messaggi:** Le code di messaggi in RabbitMQ fungono da buffer temporaneo tra i produttori e i consumatori. I produttori inseriscono i messaggi in una coda, e i consumatori prelevano i messaggi da questa coda per elaborarli. Le code consentono una separazione temporale tra la produzione e il consumo di messaggi, contribuendo al disaccoppiamento delle componenti del sistema.
- **Exchange:** Gli exchange in RabbitMQ determinano come i messaggi inviati dai produttori vengono instradati alle code. Essi definiscono le regole di routing che specificano quale coda (o code) riceverà un determinato messaggio. Ciò consente una flessibilità notevole nella gestione del flusso dei messaggi, ad esempio, instradando messaggi diversi a code diverse in base a regole specifiche.

L'architettura di base di un sistema basato su "Message Queue" è semplice: ci sono applicazioni client chiamate produttori che creano messaggi e li consegnano al broker (che funge da coda di messaggi). Altre applicazioni, chiamate consumatori, si collegano alla coda e si iscrivono ai messaggi da elaborare. Il software può agire come produttore, consumatore o sia consumatore che produttore di messaggi. I messaggi inseriti nella coda vengono memorizzati finché il consumatore non li riceve ed elabora. L'utilizzo di un'architettura basata su Message Queue (MQ) in un sistema distribuito offre diversi vantaggi che contribuiscono alla scalabilità, alla resilienza e alla flessibilità dell'intero sistema, quali:

- **Disaccoppiamento:** l'utilizzo di una Message Queue consente il disaccoppiamento tra i produttori di messaggi e i consumatori. I produttori possono inviare messaggi senza dover preoccuparsi di quale applicazione o componente li consumerà. Allo stesso modo, i consumatori possono elaborare i messaggi senza dover conoscere i dettagli implementativi dei produttori. Questo disaccoppiamento favorisce la modularità e semplifica l'integrazione di nuovi componenti nel sistema.

- **Tolleranza agli Errori:** le code di messaggi possono agire come buffer temporaneo, consentendo ai messaggi di essere conservati anche in caso di guasti temporanei. In caso di malfunzionamenti temporanei di un'applicazione o di un componente del sistema, i messaggi rimangono nella coda e possono essere elaborati successivamente quando il sistema torna online. Ciò contribuisce alla resilienza e alla tolleranza agli errori del sistema distribuito.
- **Scalabilità Orizzontale:** Un'architettura basata su Message Queue facilita la scalabilità orizzontale, consentendo l'aggiunta di nuovi nodi o componenti senza impattare negativamente il resto del sistema. I messaggi vengono distribuiti tra i consumatori in modo efficiente, consentendo una gestione più agevole di carichi di lavoro crescenti.
- **Gestione Asincrona:** la messaggistica asincrona permette ai componenti del sistema di operare in modo indipendente l'uno dall'altro. I produttori inviano messaggi senza attendere una risposta immediata, consentendo ai consumatori di elaborare i messaggi quando sono pronti. Questo approccio asincrono favorisce una maggiore efficienza e responsività del sistema, specialmente in scenari di carichi di lavoro variabili.
- **Integrazione di Sistemi Eterogenei:** con un Message Queue è possibile integrare facilmente sistemi eterogenei, cioè sistemi che utilizzano tecnologie diverse o che possono essere scritti in linguaggi di programmazione diversi. I messaggi forniscono un formato standardizzato per la comunicazione tra sistemi, facilitando l'integrazione senza la necessità di modificare pesantemente i componenti esistenti.

RabbitMQ supporta diversi protocolli di messaggistica, inclusi AMQP (Advanced Message Queuing Protocol), MQTT (Message Queuing Telemetry Transport) e STOMP (Simple Text Oriented Messaging Protocol), consentendo l'integrazione con una varietà di applicazioni e dispositivi. Inoltre, per quanto riguarda il monitoraggio RabbitMQ fornisce un'interfaccia di gestione Web e un'API REST per il monitoraggio e la configurazione del broker. Ciò semplifica la gestione del sistema, la configurazione delle code e la supervisione delle prestazioni.

Capitolo 3

Tecnologie a confronto

3.1 Componenti e protocolli utilizzati

3.1.1 Architettura generale

Ad alto livello l'architettura di Kafka consiste di tre elementi principali: produttore, consumatore e broker. Il produttore scrive messaggi al broker, mentre il consumatore legge i dati dai broker, seguendo il pattern publish/subscribe. Il broker viene eseguito sul cluster Kafka, mentre produttore e consumatore sono completamente disaccoppiati dal sistema. Ogni broker salva i dati ricevuti su topics, ovvero raccolte di messaggi appartenenti allo stesso gruppo/categoria. I topics di Kafka possono essere suddivisi in più partizioni, il che comporta vantaggi come la tolleranza agli errori, scalabilità e parallelismo. Ogni broker può memorizzare partizioni selezionate di un topic, mentre le altre partizioni possono essere distribuite su altri broker. Questo approccio aiuta a suddividere e bilanciare il carico di lavoro tra i broker.

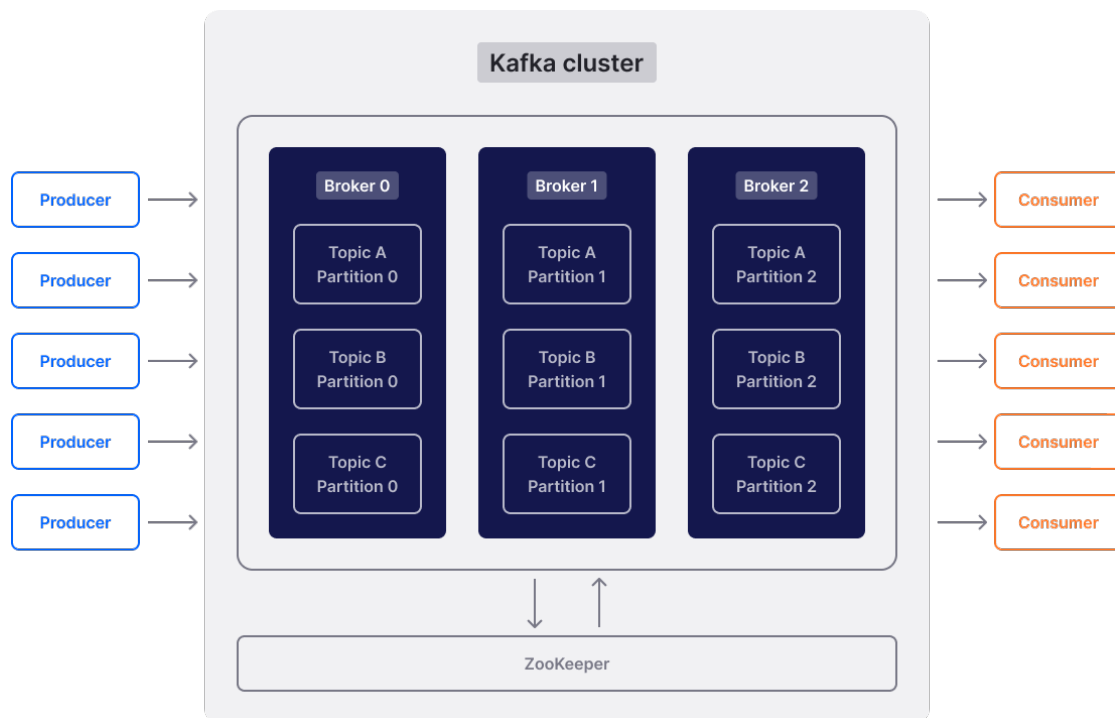


Figura 3.1: Architettura generale di Apache Kafka

In Figura 3.1 è possibile notare anche il componente Zookeeper, responsabile di:

- Memorizzare i metadati sul cluster Kafka, ad esempio le informazioni su topic, partizioni, broker e repliche.
- Gestire e coordinare i broker Kafka, compresa l'elezione del leader.
- Mantenere gli elenchi di controllo degli accessi (ACL) per motivi di sicurezza.

Con il rilascio di Apache Kafka 3.3.1 nell'ottobre 2022, un nuovo protocollo di consenso per la gestione dei metadati, chiamato KRaft, è stato dichiarato pronto per la produzione. L'esecuzione di Kafka in modalità KRaft elimina la necessità di eseguire un cluster Zookeeper accanto a ogni cluster Kafka.

Analogamente a Kafka, l'architettura di Pulsar comprende broker, produttori e consumatori. I broker operano su un cluster Pulsar, mentre produttori e consumatori sono completamente disaccoppiati dal sistema. Ogni broker gestisce i dati effettivi inviati dai produttori nei topic. Proprio come nel caso di Kafka, questi topic possono essere suddivisi in numerose partizioni, il che offre vantaggi come la tolleranza ai guasti, la scalabilità e il parallelismo. ZooKeeper è presente anche nell'architettura di Pulsar: viene utilizzato per una serie di compiti, tra cui

la gestione della configurazione, il coordinamento tra i nodi e il mantenimento dei metadati sul cluster Pulsar. Come già detto, Kafka si sta allontanando da ZooKeeper, sostituendolo con KRaft. Pulsar non sta eliminando ZooKeeper dalla sua architettura, ma offre soluzioni alternative che possono essere utilizzate al suo posto: memoria locale, *RocksDB* e *etcd* (si noti che i primi due sono applicabili solo a Pulsar standalone o a cluster Pulsar a singolo nodo).

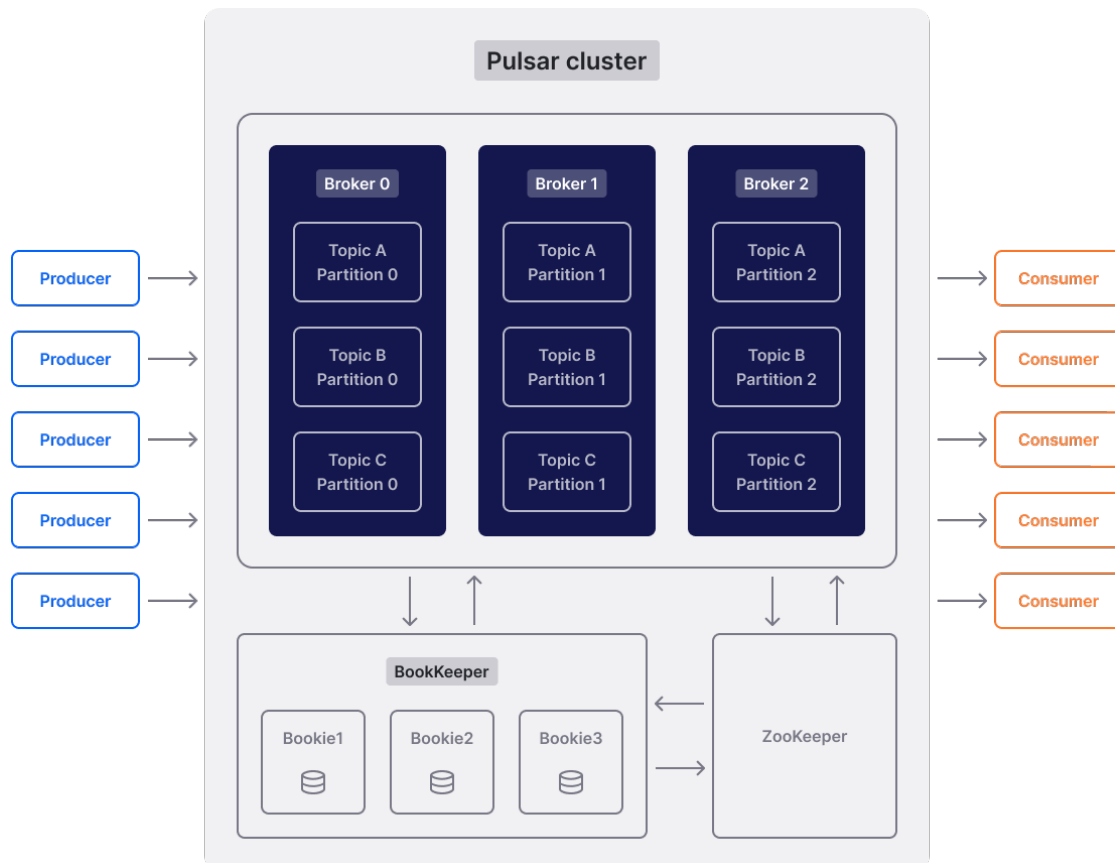


Figura 3.2: Architettura generale di Apache Pulsar.

La differenza principale con Kafka è che Pulsar separa i livelli di memorizzazione e di servizio. Nell'architettura di Pulsar, i broker gestiscono l'instradamento e la consegna dei messaggi, mentre *Apache BookKeeper* gestisce l'archiviazione a lungo termine. In particolare, ogni messaggio inviato da un produttore viene scritto su BookKeeper per essere conservato. Questo approccio di archiviazione su più livelli porta ad una maggiore complessità architetturale. La separazione consente tuttavia di scalare i livelli di storage e di servizio in modo indipendente.

L'architettura di RabbitMQ ruota sempre attorno ai concetti di produttore, consumatore e broker. I produttori generano messaggi che verranno poi letti dai

consumatori. Nello specifico, i produttori inviano messaggi ad entità interne del broker chiamate **Exchange**. Gli exchange instradano i messaggi verso **code** specifiche, le quali sono in **binding** con l'exchange tramite una **routing key**. La routing key (possiamo vederla come un header) viene posta sul messaggio per specificare la queue di destinazione, come una sorta di indirizzo. Infine, i broker RabbitMQ consegnano i messaggi ai consumatori iscritti alle code.

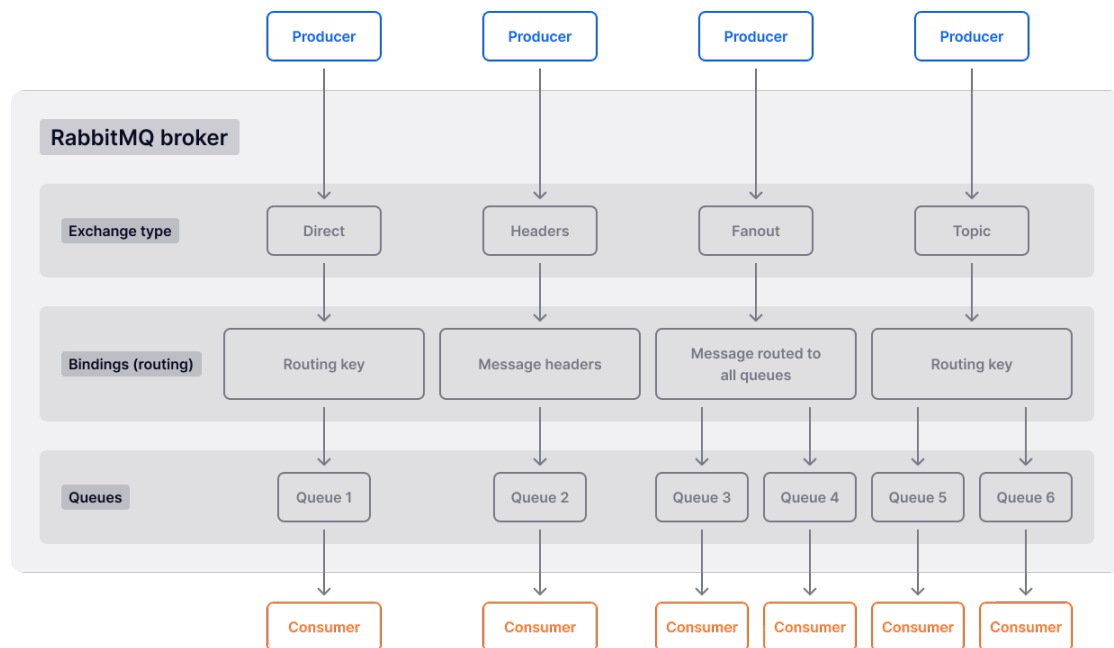


Figura 3.3: Architettura generale di RabbitMQ.

Le code sono buffer per memorizzare messaggi. Le applicazioni possono decidere i nomi delle code o chiedere al broker di generarli. I nomi possono contenere fino a 255 byte di carattere UTF-8 e non possono iniziare con “amq”, poiché tali nomi sono riservati dal broker per uso interno. La coda può sopravvivere a un riavvio del broker e si ottiene persistendo la coda su disco. Questo non implica che anche i messaggi siano durevoli. Una coda è utilizzata da una sola connessione e cancellata una volta chiusa la connessione. Se qualsiasi altra connessione tenta di accedere a una coda esclusiva, viene lanciata un’eccezione a livello di canale.

Ci sono diverse tipologie di exchange:

- *Direct Exchange*: invia i messaggi che riceve dai producer direttamente ai consumer, questa azione avviene lavorando sulla routing key. Il messaggio viene inviato con una routing key, ad esso associata, che permette all’exchange di inoltrarlo alla queue in ascolto su quella specifica chiave. Possiamo avere

anche una queue in ascolto su più routing key o più queues in ascolto sulla stessa, in questo caso l'exchange inoltra il messaggio ad entrambe le queues.

- *Topic exchange*: amplia il concetto alla base del direct exchange. Il matching tra la queue e la sua routing key avviene tramite un pattern matching. Al messaggio viene associata una routing key strutturata usando dei punti "." come carattere di separazione per separare i topic, mentre la queue è in ascolto su una routing key strutturata usando caratteri diversi: oltre al punto può contenere l'asterisco "*" (per matchare una parola in una posizione specifica della routing key) o il cancelletto "#" (per matchare zero o più parole).
- *Headers Exchange*: progettato per l'instradamento dei messaggi alle code in base a più attributi espressi come intestazioni dei messaggi piuttosto che come routing keys.
- *Fanout exchange*: i messaggi vengono instradati verso tutte le code collegate all'exchange. Questo tipo di exchange è la scelta migliore per casi d'uso di tipo broadcast. Non viene utilizzata alcuna routing key.

3.1.2 Protocolli di comunicazione

Per quanto riguarda i protocolli di comunicazione utilizzati, i tre sistemi di messaggistica hanno alcune differenze chiave:

- RabbitMQ: utilizza il protocollo di comunicazione *AMQP* (*Advanced Message Queuing Protocol*). È un protocollo aperto, standardizzato e indipendente dal fornitore, progettato per la messaggistica avanzata. AMQP supporta messaggi persistenti, code di messaggi, scambio di messaggi, filtri di messaggi e altre funzionalità avanzate. Come anticipato, RabbitMQ supporta anche altri protocolli come MQTT e STOMP.
- Apache Kafka: utilizza un protocollo binario su TCP. Il protocollo definisce tutte le API come coppie di messaggi richiesta/risposta. Tutti i messaggi hanno una dimensione limitata. Il client avvia una connection socket, quindi scrive una sequenza di messaggi di richiesta e legge il messaggio di risposta corrispondente. Non sono richiesti handshake al momento della connessione e della disconnessione. Il server garantisce che su una singola connessione TCP le richieste vengano elaborate nell'ordine in cui vengono inviate e che anche le risposte vengano restituite nello stesso ordine.
- Apache Pulsar: anch'esso utilizza un protocollo binario personalizzato per le comunicazioni tra produttori/consumatori e broker. Questo protocollo è

stato progettato per supportare le caratteristiche richieste, come i riconoscimenti e il controllo di flusso, garantendo al contempo la massima efficienza di trasporto e implementazione.

3.1.3 Sicurezza

Autenticazione

Kafka utilizza *SASL* (*Simple Authentication and Security Layer*) per eseguire l'autenticazione. Attualmente supporta molti meccanismi, tra cui *SCRAM*, *OAUTHBEARER* e *GSSAPI* e consente all'amministratore di inserire implementazioni personalizzate. L'autenticazione può essere abilitata tra broker, tra client e broker e tra broker e ZooKeeper. Consente di limitare l'accesso solo alle parti che dispongono dei segreti richiesti.

Per implementare l'autenticazione in un ambiente Pulsar, è comune utilizzare le configurazioni di sicurezza fornite da Pulsar stessa e, se necessario, integrare un sistema di gestione dell'identità esterno. Le configurazioni di sicurezza di Pulsar includono opzioni per abilitare l'autenticazione tramite Token *JWT* (JSON Web Token) o *TLS* (Transport Layer Security).

RabbitMQ offre diverse opzioni che possono essere configurate per soddisfare i requisiti di sicurezza. Le principali modalità di autenticazione sono: *plain authentication*, *LDAP authentication*, *SSL/TLS authentication*.

Autorizzazione

Kafka gestisce l'autorizzazione tramite *ACL* (Access Control Lists), attraverso un autorizzatore e fornisce un'implementazione predefinita dell'autorizzatore (*AclAuthorize*) che memorizza le ACL in Zookeeper. Se non esiste tale configurazione, tutti sono autorizzati ad accedere a qualsiasi risorsa. All'avvio, il broker Kafka avvia un ACL. La cache ACL popolata viene mantenuta e utilizzata per l'autorizzazione ogni volta che arriva una richiesta.

Pulsar utilizza un modello di controllo degli accessi basato su ruoli (RBAC). Gli utenti e le applicazioni vengono assegnati a ruoli specifici, e i permessi vengono associati a tali ruoli. Questo consente di gestire in modo efficiente l'autorizzazione in base alle funzionalità o ai compiti specifici degli utenti. Pulsar consente la definizione di regole di autorizzazione personalizzate, consentendo di specificare i dettagli delle autorizzazioni in modo granulare. Le regole possono essere definite a livello di namespace, di topic o di subscription.

Anche RabbitMQ utilizza un sistema di controllo basato su ruoli. Vengono utilizzati i ruoli per definire i privilegi di accesso. Un ruolo è associato a uno o più utenti e specifica quali operazioni sono consentite su quali risorse. Le risorse

in RabbitMQ includono code, exchange, e altri elementi del sistema. Le regole di autorizzazione possono essere configurate specificamente per ciascun tipo di risorsa. Ad esempio, è possibile concedere a un ruolo il permesso di pubblicare messaggi su un exchange o consumare messaggi da una coda.

Cifratura dei messaggi

In tutti e tre i sistemi, *SSL/TLS* è comunemente utilizzato per stabilire connessioni sicure, sfruttando certificati X.509 per autenticare il broker e i partecipanti alle comunicazioni, garantendo così riservatezza e integrità durante il trasferimento dei dati. In aggiunta a ciò, sia Pulsar che RabbitMQ offrono la possibilità di implementare la crittografia dei messaggi a livello di payload. Questa caratteristica consente di cifrare il contenuto effettivo dei messaggi durante la produzione e di decriptarlo durante il consumo, aggiungendo un ulteriore strato di sicurezza per proteggere i dati scambiati.

3.2 Persistenza dei messaggi

In Apache Kafka, trattandosi di un sistema di commit-log, i record vengono aggiunti alla fine di ogni partizione e ogni partizione è divisa in segmenti. I segmenti aiutano a eliminare i record più vecchi attraverso la compattazione e a migliorare le prestazioni. Kafka permette di ottimizzare le configurazioni relative ai log: possiamo controllare il rolling dei segmenti, la conservazione dei log ecc... Queste configurazioni determinano la durata, la conservazione del record e impattano anche sulle performance del broker. Come detto precedentemente, i topic Kafka sono divisi in partizioni. Una partizione è un'unità logica di lavoro in Kafka in cui vengono aggiunti i record, ma non è l'unità di memorizzazione. Le partizioni sono ulteriormente suddivise in segmenti che sono i file effettivi del file system. Per migliorare le prestazioni e la manutenzione, vengono creati più segmenti e, invece di leggere da un'unica grande partizione, i consumatori possono leggere da un file di segmento più piccolo.

Il livello di persistenza in Pulsar è affidato ad Apache BookKeeper, il quale presenta un'architettura innovativa che combina efficienza, scalabilità e tolleranza ai guasti attraverso concetti chiave come ledger, segmentazione, e un approccio a livelli multipli. In Pulsar, i dati sono organizzati in "ledger". Un ledger è essenzialmente una sequenza di write-ahead log (WAL) associata a un topic specifico. La segmentazione è un elemento distintivo di Pulsar: all'interno di un ledger, i dati vengono suddivisi in "segmenti". Questo approccio consente una gestione più granulare dei dati e semplifica l'archiviazione e il recupero dei segmenti specifici. La segmentazione non solo migliora l'efficienza dello storage ma contribuisce anche

alla scalabilità orizzontale, consentendo l'aggiunta di nodi senza impatto significativo sulla coerenza dei dati. Utilizzando un approccio a livelli multipli per lo storage, i dati attivi o recenti, che richiedono un accesso veloce, vengono mantenuti in memoria, garantendo prestazioni elevate per i consumatori che leggono i dati più recenti. D'altra parte, i dati persistenti sono conservati su disco per garantire la persistenza a lungo termine.

A differenza di sistemi di streaming come Apache Kafka o Apache Pulsar, RabbitMQ è più orientato alla messaggistica punto a punto e alla messaggistica di tipo publish/subscribe. In RabbitMQ la gestione dello storage è principalmente orientata alla conservazione temporanea dei messaggi fino a quando non vengono elaborati o consegnati ai consumatori. Come descritto in precedenza, i messaggi sono inviati a una coda specifica, ogni coda utilizza una struttura di storage che mantiene i messaggi in ordine di arrivo. Le code possono essere configurate come “durable” o “non durable”. Le code “durable” garantiscono la persistenza dei messaggi anche a seguito del riavvio del server, scrivendo i dati su disco. Le code “non durable” invece non mettono a disposizione questo servizio.

3.3 Tolleranza ai guasti e disponibilità

Nel contesto della messaggistica, la fault tolerance e la disponibilità sono aspetti cruciali per garantire che i sistemi siano resilienti e in grado di gestire guasti imprevisti. Indipendentemente dal settore, infatti, quando si verifica un evento imprevisto che interrompe le operazioni quotidiane, un'organizzazione ha bisogno di un piano di disaster recovery ben preparato per ripristinare rapidamente il servizio ai clienti. Tuttavia, un piano di disaster recovery richiede solitamente un'implementazione multi-datacenter con data center geograficamente dispersi. Tale distribuzione multi-datacenter richiede un meccanismo di georeplicazione per fornire una ridondanza aggiuntiva in caso di guasto di un data center.

Le strategie di failover in Apache Kafka si concentrano sulla gestione di situazioni in cui un nodo o un'intera regione diventa inaccessibile o fallisce. L'insieme delle strategie e dei meccanismi di failover in Apache Kafka appartengono ad un modulo chiamato **Control Plane**, che si distingue dal **Data Plane** il quale si occupa del flusso effettivo dei dati. Storicamente, il control plane di Kafka è stato gestito attraverso il servizio esterno ZooKeeper. Un broker è designato come controllore ed esso è responsabile della comunicazione con ZooKeeper e con gli altri broker del cluster. I metadati del cluster sono conservati all'interno di ZooKeeper. Come anticipato, a partire da Ottobre 2022 l'esecuzione di Kafka in modalità KRaft elimina la necessità di eseguire un cluster Zookeeper accanto a ogni cluster Kafka. In KRaft, un sottoinsieme di broker viene designato come controllers, i quali forniscono i servizi di consenso che prima erano forniti da ZooKeeper. Tutti i metadati

del cluster sono ora memorizzati nei topic di Kafka e gestiti internamente. Alcuni dei vantaggi di KRaft sono:

- Distribuzione e amministrazione più semplici: avendo una sola applicazione da installare e gestire, Kafka ha ora un'impronta operativa molto più ridotta. In questo modo è anche più facile sfruttare Kafka in edge-device meno performanti.
- Scalabilità migliorata: il tempo di ripristino è un ordine di grandezza più veloce con KRaft
- Propagazione dei metadati più efficiente: La propagazione dei metadati basata su log e guidata dagli eventi si traduce in un miglioramento delle prestazioni per molte delle funzioni principali di Kafka.

Per quanto riguarda la georeplicazione dei dati, essa viene gestita nel seguente modo:

- Replicazione tra Data Center: consente di implementare la replicazione tra data center. Ciò significa che i dati possono essere replicati su cluster Kafka in posizioni geografiche diverse, garantendo la disponibilità dei dati anche in caso di guasti localizzati.
- MirrorMaker: Kafka offre uno strumento chiamato MirrorMaker che facilita la replicazione dei dati tra cluster Kafka in diverse posizioni. Questo è particolarmente utile per la georeplicazione in scenari distribuiti su vasta scala.

Un altro meccanismo fondamentale per garantire durabilità e disponibilità è sicuramente la replicazione dei dati. Essa in Apache Kafka è abilitata a livello di topic. Quando viene creato un nuovo topic, è necessario specificare, in modo esplicito quante repliche vogliamo o lasciare il valore di default. Il numero di repliche specificato è chiamato fattore di replica. Ogni partizione di un topic sarà replicata quel numero di volte. Con un fattore di replica pari a N , in generale possiamo tollerare $N - 1$ guasti, senza perdita di dati e mantenendo l'availability. Una volta che le repliche sono state create, una replica per ogni partizione sarà eletta replica leader e il broker che detiene tale replica sarà il leader per quella partizione. Le repliche rimanenti invece saranno follower. I produttori scriveranno sulla replica leader e i follower recupereranno i dati per rimanere sincronizzati con la replica leader. Anche i consumatori, in genere, prelevano dalla replica leader, ma possono essere configurati anche per prelevare da altre repliche.

Il leader della partizione, insieme a tutti i follower, fanno parte dell'insieme di repliche in-sync (ISR). In una situazione ideale tutte le repliche ne fanno parte. Se un leader fallisce, o se per qualche altro motivo è necessario scegliere un

nuovo leader, uno dei broker nell'ISR sarà scelto come nuovo leader. Il processo di elezione del leader e la notifica ai follower interessati sono gestiti dal piano di controllo. L'aspetto importante per il piano dati è che nessun dato venga perso nel processo. Per questo motivo un nuovo leader può essere selezionato solo tra quelli appartenenti all'ISR, a meno che il topic non sia stato specificamente configurato per consentire la selezione di repliche non sincronizzate.

Il broker che contiene la replica leader svolge un lavoro maggiore rispetto alle repliche follower. Per questo motivo è meglio non avere un numero sproporzionato di repliche leader su un singolo broker. Per evitare questo, Kafka utilizza il concetto di replica preferita. Quando viene creato un nuovo topic, la prima replica per ogni partizione viene designata come replica preferita.

Poiché Kafka si occupa già di distribuire uniformemente le partizioni tra i broker disponibili, di solito si ottiene un buon equilibrio di leader. Inoltre, poiché le elezioni dei leader avvengono per vari motivi, i leader potrebbero finire su repliche non preferite e questo potrebbe portare a uno squilibrio. Di conseguenza Kafka controlla periodicamente se sono presenti squilibri nelle repliche dei leader, utilizzando una soglia configurabile. Se trova uno squilibrio, esegue un ribilanciamento dei leader per riportare essi sulle loro repliche preferite.

Per quanto riguarda Apache Pulsar, uno dei principali meccanismi utilizzati per garantire tolleranza ai guasti è sicuramente la georeplicazione dei dati. Questa viene solitamente utilizzata per il disaster recovery, consentendo la replicazione dei dati dei messaggi memorizzati in modo persistente su più data center. Ad esempio, l'applicazione pubblica i dati in una regione e desidera elaborarli per il consumo in altre regioni. Con il meccanismo di georeplicazione di Pulsar, i messaggi possono essere prodotti e consumati in diverse località geografiche.

Il diagramma presente in Figura 3.4 illustra il processo di georeplicazione. Ogni volta che tre produttori (P1, P2 e P3) pubblicano rispettivamente messaggi nel topic T1 in tre cluster, questi messaggi vengono immediatamente replicati tra i cluster. Una volta replicati i messaggi, due consumatori (C1 e C2) possono consumarli dai loro cluster.

Il meccanismo di georeplicazione può essere classificato in strategie di georeplicazione sincrona e asincrona. Pulsar supporta entrambi i meccanismi di replica.

- Georeplicazione sincrona: nella georeplicazione sincrona, i dati vengono replicati in modo sincrono su più data center e il client deve attendere una conferma dagli altri data center. Quando il client invia una richiesta di scrittura a un cluster, i dati scritti vengono replicati agli altri due data center. La richiesta di scrittura viene confermata al client solo quando la maggior parte dei data center ha confermato che la scrittura è avvenuta. La georeplicazione sincrona in Pulsar è realizzata da BookKeeper. Un cluster georeplicato

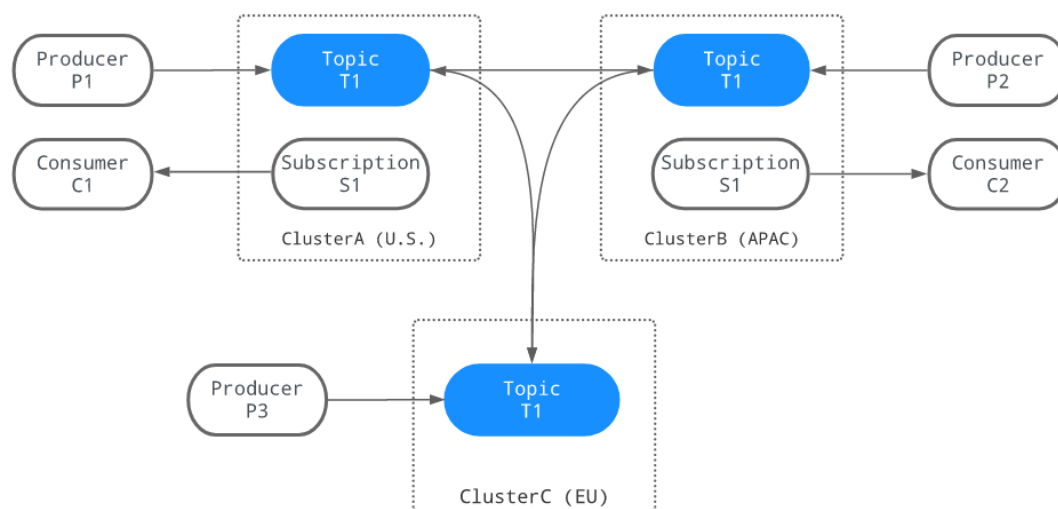


Figura 3.4: Geo-replicazione dei dati in Apache Pulsar.

sincrono è costituito da un cluster di bookies e da un cluster di broker che vengono eseguiti in più data center e da un'installazione globale di Zookeeper. La georeplicazione sincrona offre la massima disponibilità e garantisce anche una maggiore coerenza dei dati tra i diversi data center. Tuttavia, nelle applicazioni viene aggiunta un'ulteriore latenza.

- **Georeplicazione asincrona:** un cluster georeplicato in modo asincrono è composto da più cluster fisici situati in diversi data center. I messaggi prodotti su un topic vengono prima salvati nel cluster locale e poi replicati in modo asincrono ai cluster remoti dai broker. In casi normali, quando non ci sono problemi di connettività, i messaggi vengono replicati immediatamente, nello stesso momento in cui vengono spediti ai consumatori locali. In genere, la latenza di consegna end-to-end è definita dal tempo di andata e ritorno della rete (RTT) tra i data center. Le applicazioni possono creare produttori e consumatori in qualsiasi cluster, anche quando i cluster remoti non sono raggiungibili. La georeplicazione asincrona offre una latenza inferiore, ma può comportare garanzie di consistenza più deboli a causa del potenziale ritardo di replica di alcuni dati non replicati.

Inoltre, è possibile scegliere il pattern di replicazione dei dati tra diversi data center:

- *Full-mesh replication:* utilizzando una replicazione full-mesh e applicando la replica selettiva dei messaggi, è possibile personalizzare le strategie e le topologie di replica tra un numero qualsiasi di data center.

- *Active-active replication*: è una variante della replicazione full-mesh, con due soli data center. I produttori possono essere eseguiti in qualsiasi data center per produrre messaggi e i consumatori possono consumare tutti i messaggi da tutti i data center.
- *Aggregation replication*: viene utilizzata in genere quando si replicano i messaggi dall'edge al cloud. Ad esempio, si supponga di avere 3 cluster in 3 data center vicini e un cluster aggregato in un data center centrale e di voler replicare i messaggi da più data center al data center centrale a scopo di aggregazione. È quindi possibile creare un namespace individuale per i topic utilizzati da ciascun data center e assegnare il data center aggregato a tali namespace.

Il sistema di georeplicazione dei dati di Pulsar appena descritto consente di ottenere un meccanismo di failover automatico a livello di cluster. Esso infatti consente di ottenere il passaggio dei client Pulsar da un cluster primario a uno o più cluster di backup in modo automatico e senza interruzioni quando rileva un evento di failover. Una volta che il cluster primario funziona di nuovo, i client Pulsar possono passare al cluster primario e nella maggior parte dei casi gli utenti non si accorgono di nulla. Questo meccanismo di failover automatico offre allo stesso tempo tolleranza agli errori, disponibilità continua oltre ad una serie di vantaggi, tra cui:

- Riduzione dei costi: i servizi possono essere commutati e ripristinati automaticamente senza perdita di dati.
- Gestione semplificata: le aziende possono operare in modalità "always-on", poiché non è richiesto l'intervento immediato dell'utente.
- Maggiore stabilità e robustezza: garantisce prestazioni continue e riduce al minimo i tempi di inattività del servizio.

RabbitMQ non offre una soluzione nativa per la georeplicazione dei dati come parte integrante del sistema. Tuttavia, esistono approcci per implementare la georeplicazione:

- Plugin di replicazione: RabbitMQ supporta plugin di replicazione che consentono di replicare i dati tra nodi di RabbitMQ. Tuttavia, questa replicazione è di solito configurata all'interno di un singolo cluster e può comportare alcune sfide nella gestione della georeplicazione su vasta scala.
- Approcci di terze parti: per implementare la georeplicazione su vasta scala, spesso è necessario fare affidamento su soluzioni di terze parti o personalizzate che possono complicare la gestione.

In sintesi, per quanto riguarda la georeplicazione, Apache Kafka e Apache Pulsar forniscono soluzioni integrate e native su più posizioni geografiche, mentre RabbitMQ richiede spesso approcci aggiuntivi o soluzioni di terze parti per affrontare questa sfida in scenari distribuiti su vasta scala.

Per quanto riguarda invece la tolleranza ai guasti, RabbitMQ offre la possibilità di creare cluster, consentendo la distribuzione del carico e fornendo ridondanza per garantire l'affidabilità del sistema. Nei cluster RabbitMQ, i nodi possono condividere code e messaggi, offrendo una maggiore resilienza. In caso di guasto di un nodo, altri nodi del cluster possono continuare a gestire le richieste senza interruzioni significative. RabbitMQ consente di configurare il mirroring delle code su più nodi del cluster. Questo implica la replicazione delle code su nodi diversi, consentendo la disponibilità dei messaggi anche in caso di guasto di un nodo. Il mirroring delle code contribuisce significativamente alla ridondanza e alla distribuzione del carico nel sistema. Con l'introduzione delle *"Quorum Queues"* in versioni più recenti di RabbitMQ, è stata implementata una soluzione avanzata per garantire l'affidabilità. Le Quorum Queues sono progettate per garantire la coerenza e la persistenza dei dati attraverso la replicazione su un numero dispari di nodi del cluster. Ciò migliora ulteriormente la disponibilità e la fault tolerance del sistema. Inoltre RabbitMQ è progettato per rilevare automaticamente i guasti e reagire di conseguenza. Il sistema è in grado di riconoscere la perdita di connettività con un nodo o altre anomalie e adottare misure per garantire la continuità del servizio.

3.4 Gestione del carico di lavoro

Il bilanciamento del carico è importante nei sistemi di messaggistica e streaming distribuiti. Senza di esso, si possono verificare anomalie nei broker, come degrado delle prestazioni, indisponibilità dei cluster e spreco di risorse. A causa del volume imprevedibile dei messaggi nei topic e della distanza fisica tra i broker distribuiti, non è facile distribuire dinamicamente i carichi di messaggi tra i broker. Il sistema deve monitorare e instradare continuamente i carichi di messaggi in base a condizioni mutevoli, senza compromettere le prestazioni del sistema. Ad esempio:

- Quando i topic ricevono un traffico elevato, che esaurisce le risorse di CPU o di memoria di determinati broker, il cluster scarica i broker sovraccarichi e ridistribuisce il carico ad altri broker.
- Quando i broker hanno un traffico ridotto, diventano inattivi o vengono aggiunti o rimossi. In questo caso il cluster riequilibra il carico per evitare di sprecare risorse.
- Quando i topic vengono ridistribuiti ad altri broker, il cluster assicura che essi siano immediatamente disponibili per i client.

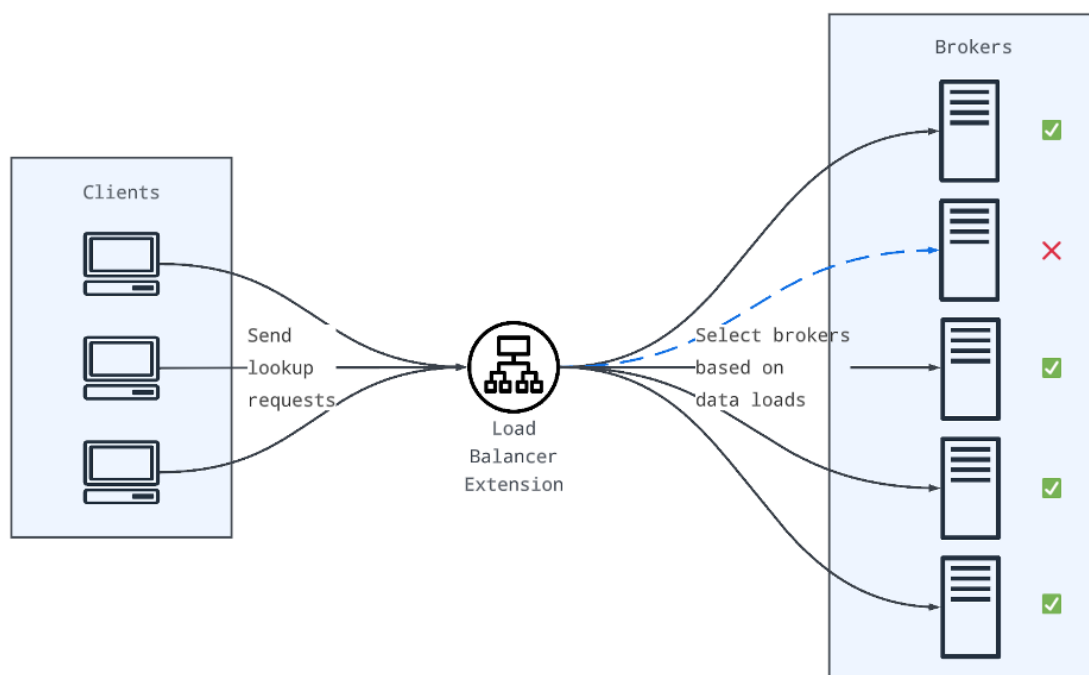


Figura 3.5: Schema di funzionamento del Load Balancer di Apache Pulsar.

Poiché Pulsar separa il livello di servizio e di memorizzazione dei messaggi, esso è stato progettato per favorire il bilanciamento del carico. A livello di persistenza (BookKeeper), i segmenti di messaggi nei topic sono bilanciati tra tutti i bookie del cluster. Quando un singolo bookie esaurisce la capacità di memorizzazione, i segmenti rimanenti vengono caricati nei bookies disponibili. Al livello di servizio (broker), la riorganizzazione dei topic avviene in maniera continua senza interrompere il servizio. Infatti, i broker non hanno bisogno di copiare i messaggi da un broker all'altro quando riequilibrano i topic: il broker carico di messaggi chiude temporaneamente il topic e le sessioni dei client e trasferisce la proprietà ad un altro broker selezionato. In questo modo, il broker selezionato assume la proprietà del topic e apre nuovamente le sessioni ai client. Pulsar utilizza il bilanciamento automatico del carico dei broker per monitorare internamente il carico dei broker e quindi bilancia dinamicamente i topic in base al carico tra tutti i broker disponibili nel modo più uniforme, dinamico e flessibile possibile.

In Figura 3.5 è possibile notare uno schema del funzionamento del load balancer di Pulsar.

In Apache Kafka, il load balancing è ottenuto principalmente attraverso la distribuzione dei dati su diverse partizioni, parallelizzando così il processo di produzione e consumo dei messaggi. Come anticipato in precedenza, i topic in Kafka

sono suddivisi in partizioni e ciascuna di esse è un'unità di parallelismo e consente la distribuzione dei dati. Ogni partizione è un registro di commit log ordinato e immutabile. I messaggi all'interno di un topic vengono assegnati a una specifica partizione.

All'interno della partizione, come si può notare in Figura 3.6 a ogni evento viene assegnato un identificatore univoco, chiamato offset. L'offset per una data partizione continuerà a crescere con l'aggiunta di eventi e gli offset non vengono mai riutilizzati. L'offset ha molti usi in Kafka, è usato per esempio dai consumatori per tenere traccia di quali eventi sono stati elaborati.

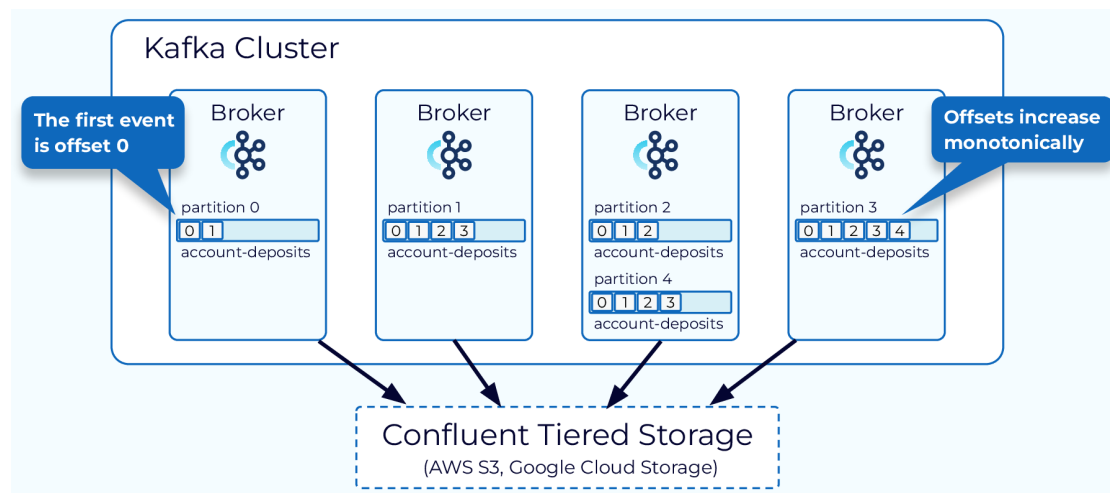


Figura 3.6: Schema di funzionamento delle partizioni in Apache Kafka.

I consumatori in Kafka leggono i messaggi dalle partizioni. Ogni consumatore è associato a una o più partizioni da cui legge i messaggi. La combinazione di più consumatori che lavorano su diverse partizioni fornisce il parallelismo necessario per gestire grandi volumi di dati. Anche Kafka fornisce un meccanismo automatico di bilanciamento delle partizioni tra i consumatori. Infatti, quando nuovi consumatori si uniscono o consumatori esistenti vengono rimossi, il coordinatore del gruppo si occupa di riassegnare le partizioni in modo equo tra i consumatori attivi. Inoltre, l'aggiunta di nuovi nodi al cluster Kafka consente di distribuire ulteriormente il carico. Il sistema automaticamente bilancia il numero di partizioni e repliche tra i nodi, garantendo una distribuzione uniforme del lavoro. Attraverso questo approccio basato su partizioni e repliche, Apache Kafka consente una gestione efficace del carico, garantendo al contempo la ridondanza e la fault tolerance necessarie per un sistema distribuito affidabile.

RabbitMQ utilizza una coda round-robin per ripetere i messaggi. Per aumentare il throughput e bilanciare il carico, i messaggi vengono suddivisi tra le code.

Inoltre, consente a numerosi consumatori di leggere i messaggi da diverse code contemporaneamente.

3.5 Modello di consegna dei messaggi

3.5.1 Pattern di comunicazione

Apache Kafka si basa su un modello di comunicazione **Publish-Subscribe** nel quale i produttori inviano messaggi a "topic" ed i consumatori, organizzati in gruppi di consumatori, si sottoscrivono a questi topic per ricevere i messaggi. Questo modello è particolarmente adatto per scenari in cui la durabilità, l'ordine dei messaggi e l'analisi dei dati sono priorità.

Come anticipato in precedenza, RabbitMQ segue il modello classico di **Message Queue**, implementando il protocollo *AMQP (Advanced Message Queuing Protocol)*. In questo modello, i produttori inviano messaggi a code specifiche, e i consumatori estraggono e elaborano i messaggi da queste code. RabbitMQ consente anche la modalità Publish-Subscribe attraverso lo scambio (exchange) dei messaggi. Questo modello è particolarmente adatto per scenari in cui è necessario gestire le code e garantire la consegna affidabile dei messaggi.

Pulsar offre un modello di comunicazione flessibile, combinando gli aspetti di Publish-Subscribe e Message Queue. I produttori inviano messaggi a "topic", e i consumatori possono scegliere di consumare i messaggi come sottoscrittori di tipo Publish-Subscribe o come consumatori di tipo Message Queue. Questo approccio consente una maggiore flessibilità, adattandosi alle esigenze specifiche degli scenari di utilizzo.

La scelta del modello di comunicazione dipende dalle esigenze specifiche del progetto. Kafka eccelle in scenari in cui la durabilità e l'analisi dei dati sono prioritari. Pulsar offre una flessibilità significativa, adattandosi alle esigenze di Publish-Subscribe o Message Queue. RabbitMQ è una scelta solida per applicazioni basate su code, garantendo la consegna affidabile dei messaggi. La decisione finale dovrebbe basarsi sulla natura dei dati, sulla necessità di durabilità, sull'ordine dei messaggi e sulla flessibilità richiesta per soddisfare le esigenze specifiche del sistema distribuito.

3.5.2 Modello di consumo dei messaggi

Kafka utilizza un'architettura basata sul **pull**, in cui i consumatori prelevano i messaggi dal server e viene utilizzato il long-polling per garantire che i nuovi messaggi siano resi disponibili istantaneamente. RabbitMQ utilizza un approccio **push**, tipico dei sistemi di messaggistica tradizionali. Anche Pulsar utilizza un

approccio push, ma con un'API che simula il pull dei consumatori. Le architetture basate su pull sono spesso preferibili per carichi di lavoro ad alta velocità, in quanto consentono ai consumatori di gestire il proprio controllo di flusso, recuperando essenzialmente solo ciò di cui hanno bisogno. Le architetture basate su push richiedono che il controllo del flusso e la backpressure siano integrati nel broker.

3.5.3 Ordinamento e riconsegna dei messaggi

Kafka fornisce forti garanzie di ordinamento a livello di partizione anche quando si consuma un topic in parallelo tra molti consumatori. RabbitMQ fornisce un ordinamento forte per i topic publish-subscribe e per i singoli consumatori, ma le code sottoscritte da più consumatori possono infrangere le garanzie di ordinamento. Le violazioni dell'ordine sono possibili anche nelle modalità "shared" di Pulsar e "key_shared" simile a Kafka.

Poiché sia Kafka che Pulsar possono memorizzare i dati in modo permanente, supportano casi d'uso non presenti invece in broker di messaggistica tradizionali. RabbitMQ, infatti, può riprodurre i messaggi che non sono stati riconosciuti, ma il numero di messaggi che possono essere conservati e riprodotti in modo efficiente è inferiore rispetto a quello di Kafka e Pulsar.

In particolare Apache Pulsar supporta la gestione dei guasti e garantisce che i dati non vadano persi. Il software è sempre soggetto a condizioni impreviste e a volte i messaggi non vengono consegnati correttamente. Pertanto, è importante disporre di un meccanismo integrato che gestisca i guasti, in particolare nella messaggistica asincrona, come evidenziato negli esempi seguenti.

- I consumatori si disconnettono dal database o dal server HTTP. Quando ciò accade, il database è temporaneamente offline mentre il consumatore sta scrivendo i dati su di esso e il server HTTP esterno che il consumatore chiama non è momentaneamente disponibile.
- I consumatori vengono disconnessi da un broker a causa di arresti anomali, connessioni interrotte, ecc. Di conseguenza, i messaggi non riconosciuti vengono consegnati ad altri consumatori disponibili.

Il meccanismo di riconoscimento negativo consente di inviare una notifica al broker che indica che il consumatore non ha elaborato un messaggio. Quando un consumatore non riesce a consumare un messaggio e ha bisogno di consumarlo di nuovo, esso invia un riconoscimento negativo (ack) al broker, attivando il broker per riconsegnare il messaggio al consumatore. I messaggi vengono riconosciuti negativamente singolarmente o cumulativamente, a seconda del tipo di sottoscrizione. In alternativa è possibile impostare un meccanismo di timeout il quale consente di impostare un intervallo di tempo durante il quale il client tiene traccia

dei messaggi non riconosciuti. Dopo questo periodo di timeout, il client invia al broker una richiesta di riconsegna dei messaggi non riconosciuti e il broker reinvia i messaggi non riconosciuti al consumatore.

Un altro meccanismo per la riconsegna dei messaggi in Apache Pulsar sono i *retry letter topic*, i quali consentono di memorizzare i messaggi che non sono stati consumati e di riprovare a consumarli in un secondo momento. Con questo metodo, è possibile personalizzare l'intervallo con cui i messaggi vengono riconsegnati. I consumatori del topic originale sono automaticamente iscritti anche al *retry letter topic*. Una volta raggiunto il numero massimo di tentativi, i messaggi non consumati vengono spostati in *dead letter topic* per l'elaborazione manuale. L'intento di utilizzare un *retry letter topic* è diverso da quello di utilizzare la consegna ritardata dei messaggi, anche se entrambi mirano a consumare un messaggio in un secondo momento. Il *retry letter topic* serve a gestire i guasti attraverso la riconsegna dei messaggi per garantire che i dati critici non vadano persi, mentre la consegna ritardata dei messaggi ha lo scopo di consegnare un messaggio con un determinato ritardo.

3.5.4 Semantica e garanzie di consegna dei messaggi

Sia RabbitMQ che Kafka e Pulsar offrono garanzie di messaggistica. Tutti offrono garanzie *at-most-once* e *at-least-once*, ma Kafka offre anche garanzie *exactly-once* in alcuni scenari molto limitati.

Le garanzie possono essere quindi classificate nel seguente modo:

- *at-most-once*: significa che un messaggio non sarà mai consegnato più di una volta, ma i messaggi potrebbero essere persi.
- *at-least-once*: significa che non si perderà mai un messaggio, ma che un messaggio potrebbe essere consegnato a un consumatore più di una volta.
- *exactly-once*: tutti i messaggi saranno consegnati esattamente una volta.

Pulsar supporta le scritture idempotenti, il che significa che i messaggi duplicati non vengono memorizzati nel livello di memorizzazione, ma non supporta le letture transazionali necessarie per l'elaborazione *exactly-once*. Ad esempio, Pulsar non può garantire la semantica *exactly-once* quando i messaggi vengono letti e poi scritti su un topic secondario, un requisito comune per molti casi d'uso pratici. Kafka, invece, supporta le semantiche *at-most-once*, *at-least-once* e anche *high-throughput exactly-once*, specificamente rivolte ai casi d'uso di elaborazione di flussi end-to-end grazie ai Kafka Streams. Ciò si ottiene con una combinazione di scritture idempotenti su Kafka e con la sua API per le transazioni. RabbitMQ non include transazioni o idempotenza nel broker, il che significa che supporta solo la semantica *at-least-once* e *at-most-once*.

3.6 Schema dei messaggi

La gestione degli schemi dei messaggi è una componente critica che può influenzare la flessibilità, la sicurezza e l'efficienza delle applicazioni distribuite. I messaggi Pulsar sono memorizzati come array di byte non strutturati e la struttura dati (nota come schema) viene applicata a questi dati solo quando vengono letti. Pertanto, sia il produttore che il consumatore devono concordare la struttura dati dei messaggi, compresi i campi e i tipi associati. Pulsar schema è il metadato che definisce come tradurre i byte grezzi dei messaggi in un tipo di struttura dati più formale, fungendo da protocollo tra le applicazioni che generano i messaggi e quelle che li consumano. Lo schema serializza i dati in byte grezzi prima che vengano pubblicati su un topic e deserializza i byte grezzi prima che vengano consegnati ai consumatori.

Pulsar utilizza uno *schema registry* come archivio centrale per memorizzare le informazioni di schema e consente ai produttori/consumatori di coordinare lo schema dei messaggi di un topic attraverso i broker.

La "type safety" è estremamente importante in qualsiasi applicazione costruita attorno a un sistema di messaggistica e streaming. I byte grezzi sono flessibili per il trasferimento dei dati, ma la flessibilità e la neutralità hanno un costo: bisogna sovrapporre il controllo del tipo di dati e la serializzazione/deserializzazione per garantire che i byte immessi nel sistema possano essere letti e consumati con successo. In altre parole, è necessario assicurarsi che i dati siano leggibili e utilizzabili dalle applicazioni.

Pulsar schema risolve questi problemi con le seguenti funzionalità:

- applica la "type safety" quando un topic ha uno schema definito. Di conseguenza, i produttori/consumatori possono connettersi solo se utilizzano uno schema "compatibile".
- fornisce una posizione centrale per la memorizzazione delle informazioni sugli schemi utilizzati all'interno dell'organizzazione, semplificando notevolmente la condivisione di queste informazioni tra i team applicativi.
- funge da unica fonte di verità per tutti gli schemi dei messaggi utilizzati da tutti i servizi e i team di sviluppo, facilitando la loro collaborazione.
- mantiene la compatibilità dei dati tra le versioni degli schemi. Quando vengono caricati nuovi schemi, le nuove versioni possono essere lette dai vecchi consumatori.

Gli schemi Pulsar vengono applicati e fatti rispettare a livello di topic. I produttori e i consumatori caricano gli schemi sui broker e di conseguenza essi funzionano su entrambi i lati.

Caso d'uso lato produttore

1. L'applicazione utilizza un'istanza di un Pulsar schema per costruire un'istanza di produttore. L'istanza di schema definisce lo schema dei dati che vengono prodotti utilizzando l'istanza del produttore. Prendendo come esempio Avro, Pulsar estrae la definizione dello schema dalla classe POJO e costruisce lo SchemaInfo.
2. Il produttore chiede di connettersi al broker con lo SchemaInfo estratto dall'istanza di schema passata.
3. Il broker consulta il registro degli schemi per verificare se si tratta di uno schema registrato. Se lo schema è registrato, il broker restituisce la versione dello schema al produttore. Altrimenti, si passa al punto 4.
4. Il broker controlla se lo schema può essere aggiornato automaticamente. Se non è consentito l'aggiornamento automatico, lo schema non può essere registrato e il broker rifiuta il produttore. Altrimenti, si passa al punto 5.
5. Il broker esegue il controllo di compatibilità dello schema definito per il topic. Se lo schema supera il controllo di compatibilità, il broker lo memorizza nel registro degli schemi e restituisce la versione dello schema al produttore. Tutti i messaggi prodotti da questo produttore sono etichettati con la versione dello schema. In caso contrario, il broker rifiuta il produttore.

Caso d'uso lato consumatore

1. L'applicazione utilizza un'istanza dello schema per costruire un'istanza del consumatore.
2. Il consumatore si connette al broker con lo SchemaInfo estratto dall'istanza di schema passata.
3. Il broker controlla se il topic è in uso (ha almeno uno degli oggetti: schema, dati, produttore attivo o consumatore). Se un topic ha almeno uno dei suddetti oggetti, si va al passo 5. Altrimenti, si va al passo 4.
4. Il broker verifica se lo schema può essere aggiornato automaticamente. Se possibile, il broker registra lo schema e connette il consumatore. In caso contrario, il broker rifiuta il consumatore.
5. Il broker esegue la verifica di compatibilità dello schema. Se lo schema supera la verifica di compatibilità, il broker connette il consumatore. In caso contrario, il broker rifiuta il consumatore.

Kafka non impone un formato specifico per i dati nei messaggi. I dati possono essere inviati come array di byte e quindi interpretati in base alle esigenze dell'applicazione. Kafka utilizza *Serializer* e *Deserializer* per convertire i dati da oggetti Java in byte array e viceversa prima di inviarli o dopo averli ricevuti. Molti produttori e consumatori predefiniti di Kafka utilizzano il *ByteArraySerializer*, che consente di inviare dati in forma di array di byte grezzi senza alcuna struttura o interpretazione specifica. È comune utilizzare formati di serializzazione più strutturati per semplificare la gestione dei dati. Ad esempio:

- **Avro**: un formato di serializzazione binario con uno schema, consente l'evoluzione dei dati.
- **JSON**: i dati in formato JSON sono più leggibili ma generalmente meno efficienti in termini di spazio rispetto ai formati binari.
- **Protobuf**: Protocol Buffers, un formato binario sviluppato da Google per la serializzazione strutturata dei dati.

Lo Schema Registry in Kafka è un componente che facilita la gestione degli schemi utilizzati per la serializzazione dei dati. È particolarmente utile quando si utilizzano formati di serializzazione come Avro, JSON Schema, o altri formati che richiedono la definizione di uno schema per i dati. Lo schema registry memorizza gli schemi utilizzati per serializzare i dati nei messaggi. Gli schemi sono versionati e possono essere registrati, recuperati e gestiti centralmente. Lo schema registry supporta regole di compatibilità degli schemi. Ciò significa che gli schemi possono evolvere nel tempo mantenendo la compatibilità con le versioni precedenti. La compatibilità può essere configurata in modo da garantire che nuovi schemi siano compatibili con i consumatori esistenti. I produttori e i consumatori possono integrarsi con lo schema registry per registrare e recuperare gli schemi utilizzati. Durante la scrittura, i produttori registrano gli schemi dei dati inviati. Durante la lettura, i consumatori recuperano gli schemi associati ai dati ricevuti per interpretarli correttamente. L'utilizzo dello Schema Registry semplifica la gestione degli schemi, consentendo una maggiore flessibilità nell'evoluzione dei dati nel tempo. In particolare, è utile quando si lavora con dati strutturati, come Avro, in cui la definizione dello schema è fondamentale per garantire una corretta interpretazione dei dati da parte dei consumatori.

RabbitMQ, con la sua filosofia di massima flessibilità, adotta un approccio più aperto nella gestione degli schemi dei messaggi. La piattaforma tratta i messaggi come semplici byte array, consentendo agli sviluppatori di definire la struttura dei dati secondo le esigenze specifiche dell'applicazione. Questa flessibilità è un tratto distintivo di RabbitMQ, in quanto non impone vincoli sul formato o sulla struttura dei messaggi. Tuttavia, questa libertà comporta una responsabilità aggiuntiva

per gli sviluppatori. A differenza di Pulsar e Kafka, RabbitMQ non fornisce un supporto nativo per la type safety o per la definizione formale degli schemi. Gli sviluppatori devono assumere un ruolo più attivo nella gestione della struttura dei messaggi, inclusa la deserializzazione e l'interpretazione corretta dei dati ricevuti. Dato che RabbitMQ non offre una soluzione integrata per gli schemi, consente una flessibilità totale per gli sviluppatori che possono adattare la struttura dei messaggi dinamicamente. Questo può essere un vantaggio in scenari in cui la struttura dei dati è soggetta a frequenti cambiamenti o quando è richiesta un'interoperabilità con sistemi che utilizzano formati diversi. È importante notare che, sebbene RabbitMQ non fornisca una soluzione nativa per la gestione degli schemi, è possibile implementare strategie personalizzate a livello applicativo o utilizzare formati di dati standard, come JSON, per introdurre un certo livello di strutturazione.

In conclusione, Pulsar offre un vantaggio distintivo tramite i *Pulsar Schema*, fornendo type safety e gestione degli schemi senza complicazioni aggiuntive. Kafka, con la sua flessibilità, richiede strumenti esterni per raggiungere funzionalità simili. RabbitMQ offre la massima flessibilità, ma richiede agli sviluppatori di gestire più attivamente la struttura dei messaggi. La scelta tra queste piattaforme dipenderà dalla priorità assegnata alla gestione degli schemi e alla type safety nel contesto specifico del progetto.

Capitolo 4

Progettazione e implementazione dei test

In seguito all'analisi svolta è stato creato un progetto con l'obiettivo di valutare e confrontare i message broker in questione. In questo capitolo verrà descritto l'approccio utilizzato e il processo che ha portato alla progettazione ed esecuzione dei vari test effettuati. Per ciascuno dei message broker si intende valutare le seguenti caratteristiche:

1. **Facilità d'uso:** questo test permette di comprendere la facilità e l'esperienza richiesta per utilizzare il Message Broker. Viene considerato quindi sia l'esecuzione del broker stesso che la produzione ed il consumo dei messaggi tramite una libreria lato client.
2. **Latenza:** in questo test viene valutato il ritardo che intercorre tra la produzione di un messaggio ed il suo effettivo consumo.
3. **Throughput:** in questo test viene valutata la quantità di dati massima che il servizio riesce a produrre in un determinato lasso di tempo. I dati vengono espressi in messaggi prodotti sul broker e l'unità di misura è quindi *MB/s*.
4. **Tolleranza ai guasti:** in questo test viene simulato il fallimento di un nodo del cluster al fine di valutare la capacità di esso di continuare a funzionare e mantenere la continuità del servizio.

Il progetto è hostato in una repository *GitLab* e come linguaggio di programmazione è stato scelto *Kotlin* per l'interoperabilità con Java, in quanto viene fornito il supporto da tutti e tre i Message Broker.

Il progetto in questione comprende quattro sotto-progetti:

- **Kafka-Driver:** utilizza la libreria client di Kafka e mette a disposizione un insieme di classi per connettersi, produrre e consumare messaggi su un broker Kafka.
- **Pulsar-Driver:** utilizza la libreria client di Pulsar e mette a disposizione un insieme di classi per connettersi, produrre e consumare messaggi su un broker Pulsar.
- **RabbitMQ-Driver:** utilizza la libreria client di RabbitMQ e mette a disposizione un insieme di classi per connettersi, produrre e consumare messaggi su un broker RabbitMQ.
- **Benchmark:** questo progetto contiene la logica dei test evidenziati. In questo modo i test risultano indipendenti dalla tecnologia. Inoltre, mette a disposizione le interfacce **BenchmarkProducer** e **BenchmarkConsumer** le quali devono essere implementate da tutti i client al fine di fornire un contratto comune per la conduzione dei test.

Al fine di rendere agile la fase di configurazione dei message broker, per ciascuno di essi viene creato un file espresso in formato *YAML* contenente tutti i parametri e le proprietà necessarie. Ciascun sotto-progetto contenente i driver dei message broker comprende anche il test per valutare la facilità d'uso in una classe ad-hoc. Il progetto principale, invece, contiene le classi necessarie per eseguire gli ulteriori test e analizzare i risultati ottenuti.

Al fine di testare i driver creati sono stati eseguiti alcuni *Unit Tests* utilizzando la suite **TestContainer**, necessaria per l'esecuzione dei message broker all'interno di container **Docker**, creando così un ambiente isolato e riproducibile. Per agevolare la conduzione dei test i message broker sono stati eseguiti all'interno di container **Docker** tramite l'utilizzo del tool **Docker Compose**. Il file contenente tutti i servizi necessari è presente all'interno del repository. Inoltre, per ciascun message broker è presente anche un ulteriore file **docker-compose** che consente l'esecuzione di cluster con più di un broker utilizzato per il test inerente alla tolleranza ai guasti. Ciascuno dei test effettuati è eseguibile mediante un apposito task di *Gradle*.

Capitolo 5

Analisi dei risultati

In questo capitolo vengono mostrati i risultati ottenuti in seguito all'esecuzione dei test descritti nel capitolo precedente. Tutti i test sono stati svolti su un computer portatile con le seguenti caratteristiche:

- Processore: Intel Core i5-8350u
- RAM: 16 GB DDR4
- Memoria: SSD 256 GB
- OS: Linux Ubuntu 22.04

5.1 Facilità d'uso

Per quanto riguarda la complessità relativa all'utilizzo delle librerie client messe a disposizione, non sono state riscontrate differenze significative tra le varie tecnologie. Ciascuna di esse, infatti, richiede solamente l'indirizzo del message broker a cui connettersi.

Sono state riscontrate invece alcune differenze nella facilità e velocità necessarie per l'avvio dei broker. In questo contesto, RabbitMQ risulta il message broker che richiede meno configurazioni per l'avvio in quanto è sufficiente mandare in esecuzione solo il servizio del broker stesso. D'altro canto, Pulsar e Kafka richiedono la presenza di ulteriori componenti quali ad esempio *Apache Bookkeeper*, per la gestione della persistenza e *Apache Zookeeper* per la gestione dei metadati del cluster.

5.2 Latenza

Come anticipato in precedenza, in questo test viene valutata la latenza di ciascun message broker ossia il ritardo che intercorre tra la produzione ed il consumo di un messaggio. Il test ha previsto l'invio di 1000 messaggi, aggiungendo una pausa tra due messaggi consecutivi, necessaria per valutare solo il tempo effettivo tra l'invio ed il consumo del messaggio e non la velocità di invio. Di seguito vengono mostrati, per ciascun message broker, i grafici con i risultati ottenuti.

Apache Kafka

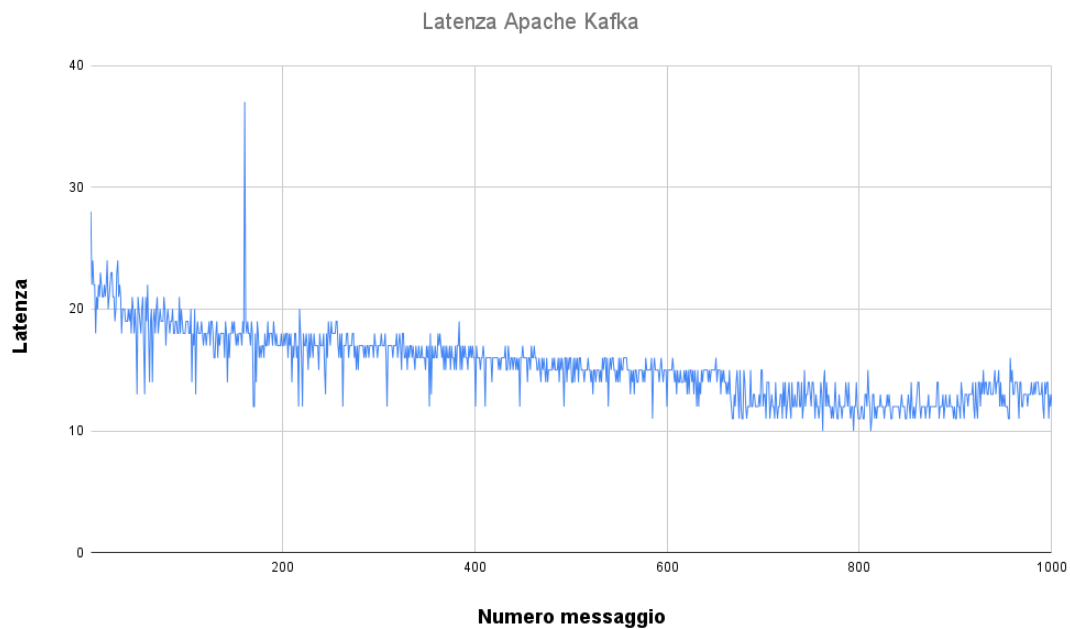


Figura 5.1: Grafico della latenza di Apache Kafka.

Apache Pulsar

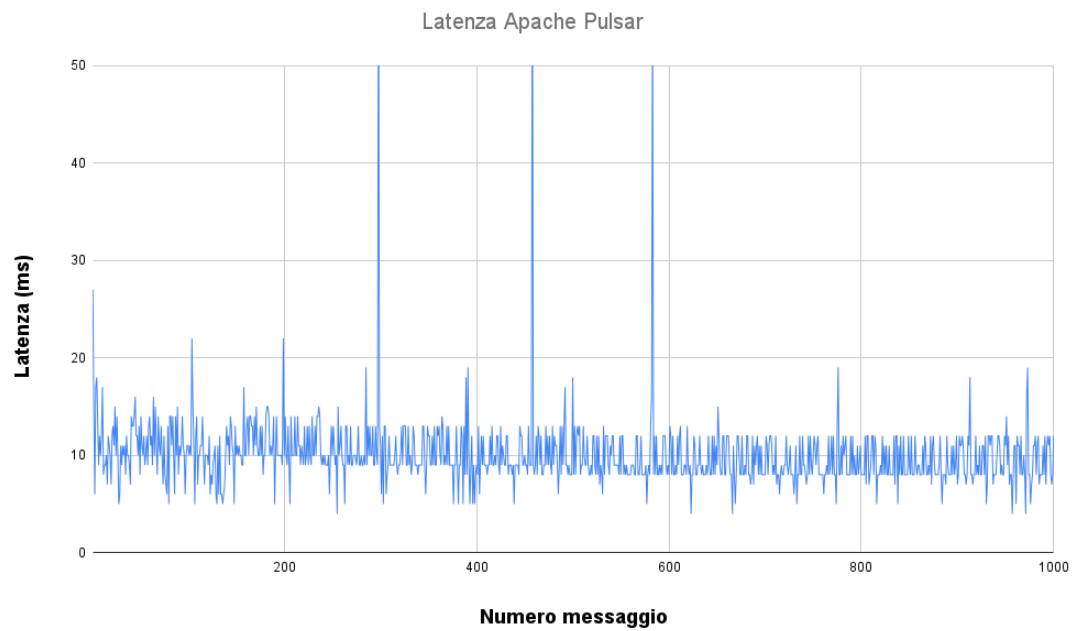


Figura 5.2: Grafico della latenza di Apache Pulsar.

RabbitMQ

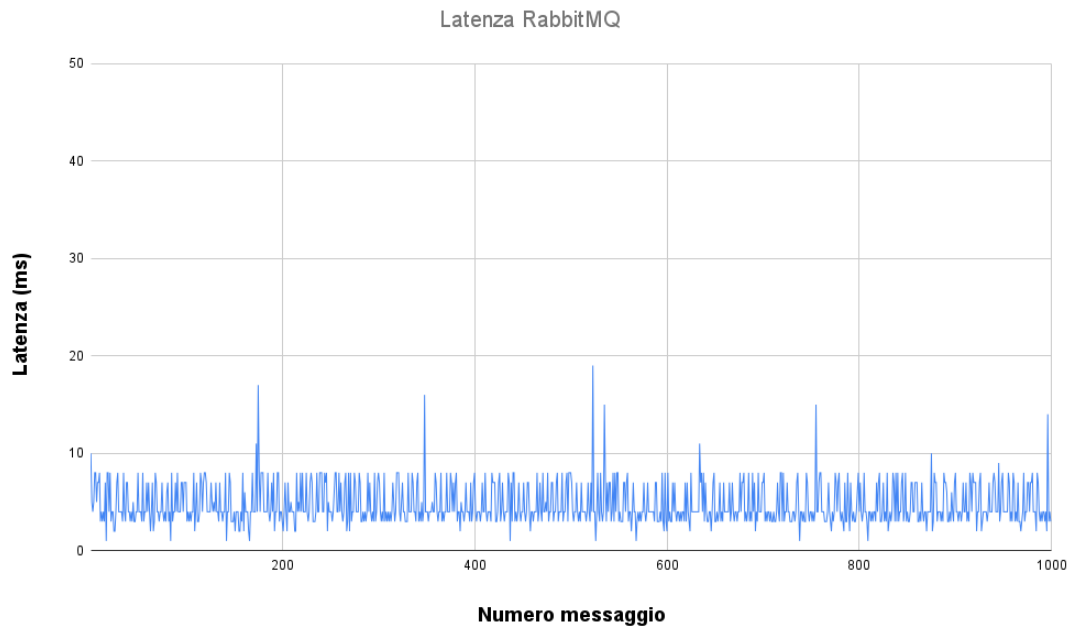


Figura 5.3: Grafico della latenza di RabbitMQ.

Latenza media

Come si può notare in Figura 5.4 il message broker con latenza minore è risultato RabbitMQ con 4,77 *ms*. Di seguito Apache Pulsar con 10,08 *ms* e infine Apache Kafka con 15,25 *ms*.

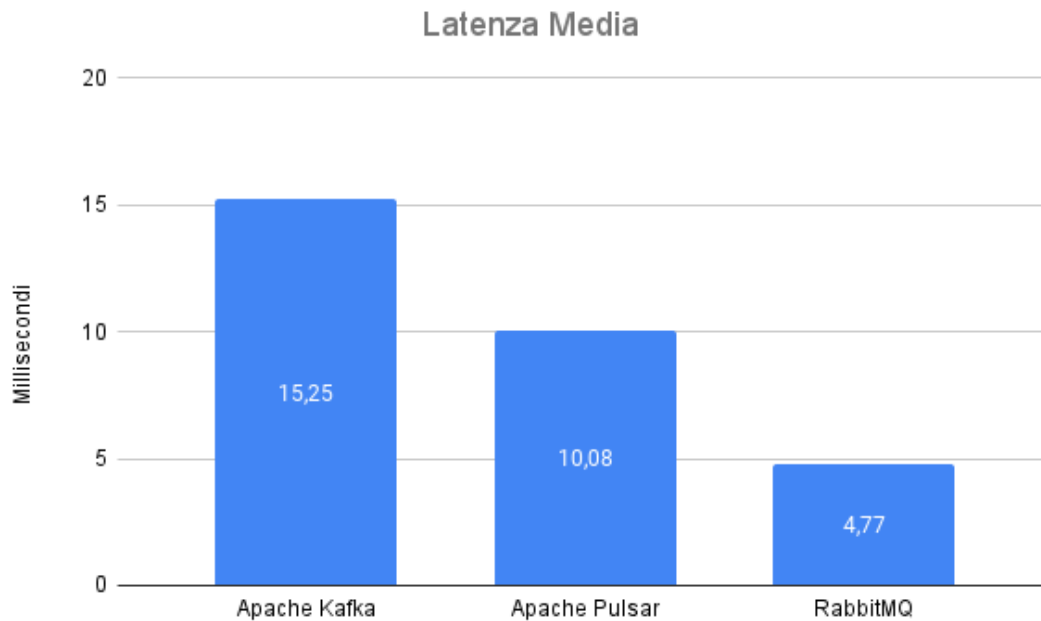


Figura 5.4: Grafico della latenza media dei tre message broker.

5.3 Throughput

Come descritto in precedenza, in questo test sono stati valutati i message broker nella produttività relativa all'invio dei messaggi in un determinato arco di tempo. Il test ha avuto quindi una durata di 1 minuto nella quale i broker hanno inviato in modo continuo messaggi della grandezza di 1 KiloByte.

Come si può notare in Figura 5.5, il message broker con throughput maggiore è Apache Kafka con 150,58 *ms*, di seguito Apache Pulsar con 62,25 *ms* ed infine RabbitMQ con 29,35 *ms*.

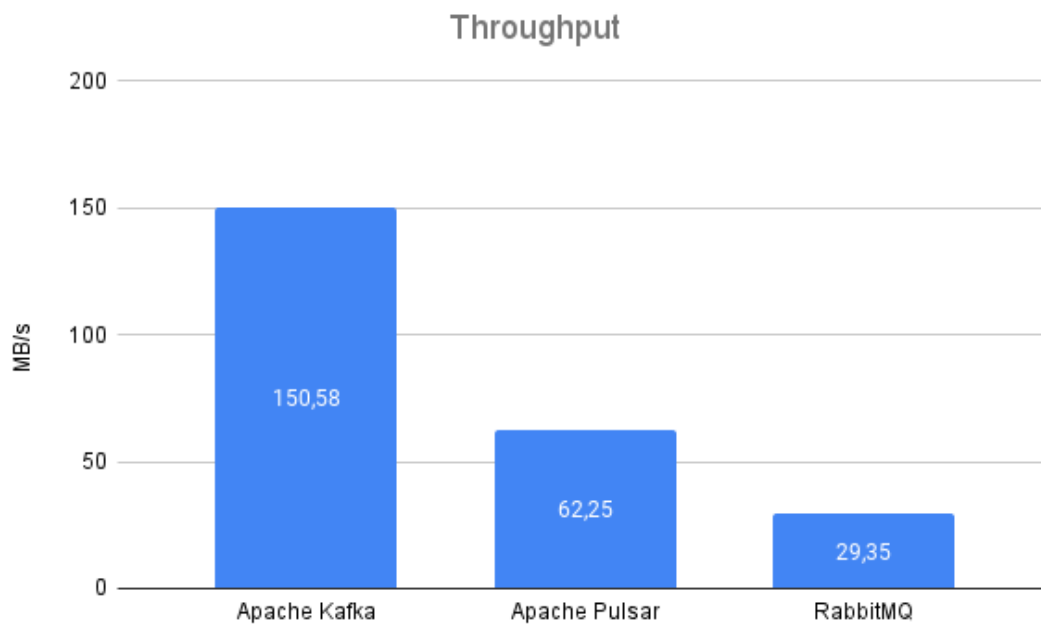


Figura 5.5: Grafico del throughput dei tre message broker.

5.4 Tolleranza ai guasti

In questo ultimo test è stato simulato il fallimento di un nodo del cluster al fine di valutare la capacità di esso di continuare a funzionare e mantenere la continuità del servizio. Durante lo svolgimento del test è stato controllato se, a seguito del fallimento di uno o più broker del cluster, il Message Broker gestisse correttamente l'elezione del nuovo leader, non ci fossero perdite di messaggi e l'availability fosse assicurata. La simulazione del fallimento di un nodo è stata effettuata stoppando, in modo alternato, tutti i container Docker contenenti i broker del cluster.

Apache Kafka

Come anticipato in precedenza, in Kafka quando viene creato un nuovo topic è necessario specificare in modo esplicito quante repliche vogliamo. Il numero di repliche specificato è chiamato fattore di replica ed ogni partizione di un topic sarà replicata quel numero di volte. Con un fattore di replica pari a N , in generale possiamo tollerare $N - 1$ guasti, senza perdita di dati e mantenendo l'availability. Una volta che le repliche sono state create, una replica per ogni partizione sarà eletta replica leader e il broker che detiene tale replica sarà il leader per quella par-

tizione. Il nostro test ha previsto la creazione di un cluster contenente due broker Kafka ed un controller, responsabile del coordinamento di attività quali elezione del leader della partizione, gestione broker del cluster, allocazione di partizioni, etc. Successivamente è stato creato un topic ad-hoc per il test con un fattore di replica pari a 2 ed è stato settato il parametro `acks=all` attraverso il quale il messaggio viene considerato inviato con successo solo quando è stato replicato su tutte le partizioni.

Apache Pulsar

Per quanto riguarda Apache Pulsar, il test è stato svolto in modo simile a quanto descritto nella sezione precedente relativa a Kafka. Infatti, è stato creato un cluster contenente due broker Pulsar oltre a Apache Bookkeeper ed Apache Zookeeper. A differenza di Kafka, però, Pulsar non utilizza un sistema di repliche e partizioni dei dati ma si affida a Apache Bookkeeper per la persistenza dei messaggi in registri chiamati *bookie*. I messaggi devono essere replicati tra molteplici bookie per garantire la continuità del servizio. Infatti, in presenza di N broker all'interno del cluster viene garantita la disponibilità in caso di fallimento di N-1 broker, tuttavia per evitare la perdita di messaggi a fronte del fallimento di Apache Bookkeeper è necessario replicare anche quest'ultimo.

RabbitMQ

In RabbitMQ uno dei meccanismi principali per ottenere fault tolerance è l'utilizzo delle *Quorum Queues*. Quest'ultime, infatti, hanno di recente rimpiazzato le *Mirroring Queues*. Infatti, invece della classica replicazione a mirror, che prevede una copia primaria e una o più copie di backup, le code quorum usano una replicazione distribuita attraverso nodi chiamati "*Quorum peers*". I Quorum peers eleggono un leader dinamicamente e se esso fallisce viene avviato un processo di elezione per selezionare il nuovo leader. In questo caso è stato creato un cluster contenente tre broker RabbitMQ, numero minimo necessario per garantire fault tolerance in caso di utilizzo delle Quorum Queues.

In conclusione, al termine dei test svolti possiamo affermare che tutti e tre i message broker sono stati in grado di gestire correttamente il fallimento di uno o più nodi. Grazie alle tecniche di replicazione e gestione della disponibilità, essi sono riusciti a garantire la continuità del servizio anche in presenza di guasti hardware o software.

Capitolo 6

Conclusioni finali

In questo progetto, abbiamo inizialmente esplorato le differenze architetturali e in seguito condotto un'analisi sulle prestazioni, la facilità d'uso e la tolleranza ai guasti di tre importanti message broker: Apache Kafka, Apache Pulsar e RabbitMQ. L'obiettivo principale era comprendere le caratteristiche distintive di ciascuna piattaforma, fornendo una base informativa per le scelte architetturali nelle implementazioni di sistemi distribuiti. Abbiamo approfondito le architetture di Kafka, Pulsar e RabbitMQ, rivelando le differenze chiave nei modelli di consegna dei messaggi, nella gestione dei topic e nei principali meccanismi utilizzati per il bilanciamento del carico e scalabilità. Possiamo affermare che Kafka si distingue per il suo modello di log distribuito, Pulsar per l'architettura multi-layer, e RabbitMQ per la velocità e la flessibilità nel supportare vari protocolli di messaggistica. L'analisi della facilità d'uso ha rivelato aspetti cruciali relativi all'installazione, la configurazione e la gestione quotidiana dei message broker. Mentre Kafka e Pulsar offrono strumenti e configurazioni avanzate, RabbitMQ si distingue per la sua semplicità e facilità d'uso. Nella fase di test delle prestazioni, abbiamo esaminato il throughput e la latenza di ciascuna piattaforma. I risultati hanno evidenziato come Apache Kafka risulti la tecnologia con throughput maggiore mentre RabbitMQ quella con latenza minore. Questi dati sono essenziali per la selezione di una piattaforma in base alle esigenze specifiche del sistema. Inoltre, un aspetto cruciale per i sistemi distribuiti è la capacità di tollerare i guasti senza interruzioni del servizio. I nostri test, in questo caso, hanno dimostrato che tutte e tre le piattaforme sono state in grado di gestire con successo il fallimento di uno o più nodi del cluster. In conclusione, la scelta tra Kafka, Pulsar e RabbitMQ dipende fortemente dai requisiti specifici del sistema, dalle preferenze di implementazione e dalle priorità degli sviluppatori.