Datastore Chiave-Valore Distribuito con Garanzie di Coerenza

Dissan Uddin Ahmed
Università di Roma, Torvergata
Roma, Italia
dissanahmed@gmail.com

***Abstract*— In questo articolo presento la progettazione e Questo articolo descrive l'implementazione di un archivio chiave-valore distribuito (DKV) con garanzie di coerenza. Il sistema supporta operazioni fondamentali come put(key, value), get(key) e delete(key), eseguibili da qualsiasi client su una delle repliche, e per scopi di testing non è stato incluso alcun meccanismo di autorizzazione. La coerenza sequenziale è assicurata tramite orologi logici scalari, mentre la coerenza causale è garantita con l'uso di orologi vettoriali, gestendo le dipendenze tra le operazioni. Il sistema è stato sviluppato in Go, con i nodi distribuiti eseguiti all'interno di container Docker, orchestrati tramite Docker Compose. Per emulare scenari realistici, sono stati introdotti ritardi di rete, simulando condizioni di latenza variabile che possono verificarsi in ambienti distribuiti geograficamente.**

I. Introduzione

Gli archivi chiave-valore (KVS) distribuiti sono componenti fondamentali nei sistemi distribuiti moderni. Offrono un modello semplificato di gestione dei dati, dove le informazioni vengono archiviate e recuperate tramite chiavi, mentre le repliche garantiscono ridondanza e tolleranza ai guasti. Tuttavia, garantire la coerenza tra le repliche rappresenta una sfida significativa. In questo documento descrivo lo sviluppo di un KVS distribuito che offre garanzie di coerenza sequenziale e causale. Il sistema utilizza orologi logici scalare per la consistenza sequenziale e clock logici vettoriali per gestire la consistenza causale,

II. Background

I modelli di coerenza nei sistemi distribuiti definiscono come le operazioni vengono propagate tra le repliche. In questo lavoro, mi concentro su due modelli principali:

- **Coerenza Sequenziale**, che assicura che le operazioni appaiano nello stesso ordine su tutte le repliche, come percepito da tutti i client.
- **Coerenza Causale**, che garantisce che le operazioni causalmene correlate vengano visualizzate nello stesso ordine, mentre le operazioni indipendenti possono essere visualizzate in ordini diversi su repliche differenti.

Gli orologi logici forniscono un meccanismo per imporre la coerenza:

- **Orologi Scalari**, utilizzati per garantire la coerenza sequenziale assegnando un timestamp monotonicamente crescente a ogni operazione.
- **Orologi Vettoriali**, che tracciano le dipendenze causali mantenendo un vettore di contatori, uno per ogni replica.

III. Progettazione della Soluzione

Il sistema è costituito da un insieme di repliche, ovvero nodi che rappresentano la natura distribuita del DKV. Questi nodi sono organizzati in una rete overlay peer-to-peer strutturata ad anello, ordinata in base al GUID univoco di ciascun nodo (fig. 1). Sebbene questa implementazione sia principalmente focalizzata sulla coerenza, è facilmente estensibile per scenari che richiedono alta disponibilità, aggiungendo meccanismi di tolleranza ai guasti e bilanciamento del carico. In questa prima implementazione, l'obiettivo è garantire una forte consistenza, quindi ogni nodo mantiene una copia aggiornata del proprio DataStore in memoria, che viene sincronizzata in modo sequenziale o causale a seconda del modello di coerenza utilizzato, per semplicità tutti i nodi hanno informazione sugli altri. Gli aggiornamenti vengono propagati tra i nodi utilizzando chiamate RPC (Remote Procedure Call), un meccanismo nativo supportato dal linguaggio Go, che consente una comunicazione leggera ed efficiente tra le repliche. Tutti i nodi del sistema sono containerizzati utilizzando il Docker engine, il che semplifica la gestione e la distribuzione delle applicazioni. Ogni nodo in Docker Compose è rappresentato da un hostname unico, facilitando così l'identificazione e la comunicazione tra le repliche. La gestione della coerenza è coordinata attraverso un meccanismo di multicast, con i leader scelti in base alla porzione di chiavi che gestiscono, i quali si occupano di orchestrare la propagazione degli aggiornamenti alle altre repliche. I client possono interagire con il sistema eseguendo operazioni di get, put e delete su qualsiasi nodo, utilizzando chiamate HTTP RESTfull, garantendo un accesso distribuito uniforme. Di seguito sono elencati i componenti del sistema:

A. Node

Il componente *Node* rappresenta un singolo nodo all'interno della rete distribuita. Contiene informazioni chiave come il nome del nodo, un identificatore univoco globale (GUID) e i dettagli delle porte utilizzate per tre tipologie di comunicazione:

- Overlay port: per entrare o uscire dalla rete.
- Multicast port: per effettuare le sincronizzazioni.
- Data port per le interazioni con i client.

Questo permette a ogni nodo di integrarsi nella rete e di gestire le comunicazioni necessarie per il coordinamento e l'aggiornamento dei dati distribuiti.

B. Replica

Il componente *Replica* estende le funzionalità del *Node*, aggiungendo informazioni su tutti gli altri nodi del sistema distribuito. Gestisce anche le politiche relative a ritardi di rete simulati, decidendo come comportarsi in caso di latenza elevata. Inoltre, questo componente definisce quale funzione di hash utilizzare per la distribuzione delle chiavi tra le

repliche. La *Replica* espone le chiamate RPC necessarie per supportare la rete overlay.

C. *RpcSequentialMulticast*

Questo componente è responsabile della gestione del multicast sequenziale, garantendo che i dati siano distribuiti in modo consistente in tutti i nodi. Usa chiamate RPC per propagare gli aggiornamenti e garantire la coerenza sequenziale. I suoi principali sottocomponenti includono:

- Coda di messaggi: Utilizzata per ordinare i messaggi in base al loro timestamp, assicurando che vengano applicati nell'ordine corretto.
- Orologio logico scalare: Tiene traccia dell'ordine totale nel sistema, consentendo a tutti i nodi di applicare le operazioni nello stesso ordine globale.

D. *RpcCausalMulticast*

Simile a *RpcSequentialMulticast*, questo componente gestisce la coerenza causale. Utilizza una coda per posticipare l'esecuzione di messaggi quando le relazioni di causalità non vengono rispettate, e impiega un orologio logico vettoriale per monitorare e ordinare le operazioni nel sistema. In questo modo, garantisce che le operazioni causalmente correlate siano eseguite nell'ordine corretto, senza imporre un ordine totale tra eventi indipendenti.

E. *Handler*

Il *Handler* funge da punto di accesso per i client. Esso espone le API per le operazioni get, put e delete, consentendo ai client di interagire con il sistema distribuito. Attraverso chiamate HTTP RESTful, i client possono eseguire queste operazioni su qualsiasi nodo, il quale si occuperà di coordinare l'aggiornamento o la lettura dei dati all'interno della rete distribuita, garantendo le corrette proprietà di consistenza.

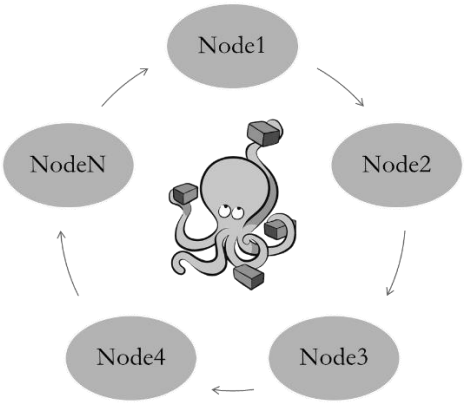

Fig. 1. Architettura del sistema

IV. Implementazione della Consistenza

A. *RpcSequentialMulticast*

La coerenza sequenziale viene garantita utilizzando un orologio logico scalare. Quando viene eseguita una richiesta relativa a una chiave, il nodo leader responsabile della gestione di quella chiave viene contattato tramite una chiamata RPC di multicast (fig. 2). Il leader ha il compito di coordinare l'intera operazione, coinvolgendo tutti i nodi del sistema, incluso sé stesso (fig. 3). Per ogni richiesta ricevuta, il leader invia una chiamata al metodo Receive su ciascun nodo. Ogni nodo, a sua volta, registra la richiesta in una coda di priorità, ordinandola in base al timestamp associato al messaggio e, in caso di parità, utilizzando il GUID del nodo. Successivamente, il nodo invia una notifica di avvenuta ricezione (acknowledgment) a tutti gli altri nodi presenti nel sistema. Una volta che il leader riceve tutti gli acknowledgment dai nodi, verifica due condizioni:

- La richiesta deve essere il primo elemento nella coda di priorità del nodo.
- Tutti i nodi devono aver confermato la ricezione della richiesta.

Quando entrambe le condizioni sono soddisfatte, l'operazione è completata, e il leader comunica al client il successo dell'operazione (fig. 4). Questo processo garantisce che tutte le repliche applichino le operazioni nello stesso ordine, preservando la coerenza sequenziale all'interno del sistema.

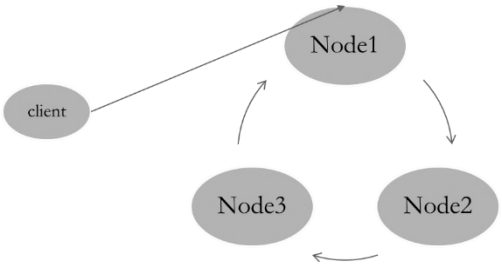

Fig. 2. Richiesta di aggiornamento da parte del client

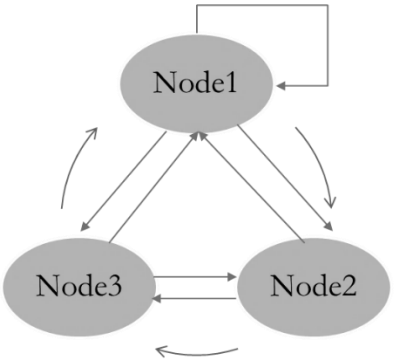

Fig. 3. Invio di Ack, per il multicasting

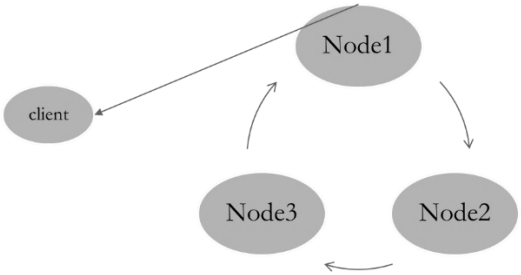

Fig. 4. Risposta del server al client

B. *RpcCausalMulticast*

Il componente RpcCausalMulticast gestisce la coerenza causale nel sistema, garantendo che gli aggiornamenti siano applicati in un ordine che rispetti le relazioni di causalità tra le operazioni. Quando un client invia una richiesta di aggiornamento (fig. 5), il nodo responsabile risponde immediatamente al client se il proprio stato è causalmente conforme (fig. 6), il che significa che l'aggiornamento può essere applicato senza violare la coerenza causale del sistema. Questo approccio consente di migliorare la latenza percepita dal client, poiché non deve attendere il completamento dell'aggiornamento su tutte le repliche. In caso di concorrenza, il nodo leader che gestisce la chiave aggiunge il messaggio in un buffer per motivi di tolleranza ai guasti e avvia l'invio dell'aggiornamento a tutti i nodi nel sistema. Ogni nodo ricevente verifica se il messaggio è causalmente corretto, confrontando il vector clock del messaggio con il proprio. Il vector clock è una struttura dati che tiene traccia delle versioni locali e delle dipendenze causali di ogni nodo all'interno del sistema. Ogni volta che un nodo riceve un messaggio, aggiorna il proprio vector clock in base all'ID del nodo mittente e il timestamp dell'operazione. Questo permette ai nodi di mantenere una visione coerente dello stato del sistema e di determinare se possono applicare un messaggio o se devono ritardarne l'applicazione. Se il messaggio è considerato causalmente corretto, i nodi rispondono affermativamente al mittente, consentendo la rimozione del messaggio dal buffer. In caso contrario, il nodo accoda la richiesta per un'applicazione futura, poiché la sua applicazione deve essere ritardata fino a quando non si verifica la condizione di causalità. È importante notare che, a causa della natura distribuita del sistema e dei ritardi nella propagazione degli aggiornamenti, un altro client che invia una richiesta di get a un nodo differente potrebbe ancora non vedere l'aggiornamento recente non avendo più l'impressione che il server sia unico. Questa architettura permette di gestire efficacemente gli aggiornamenti in ambienti distribuiti, garantendo che le operazioni siano applicate in modo tale da preservare la coerenza causale, anche in presenza di ritardi nella comunicazione tra i nodi.

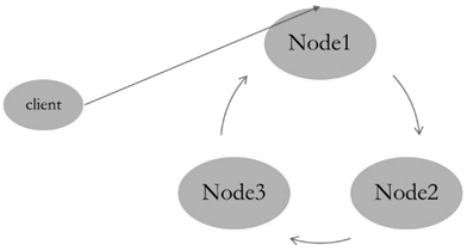

Fig. 5. Richiesta del client di un aggiornamento

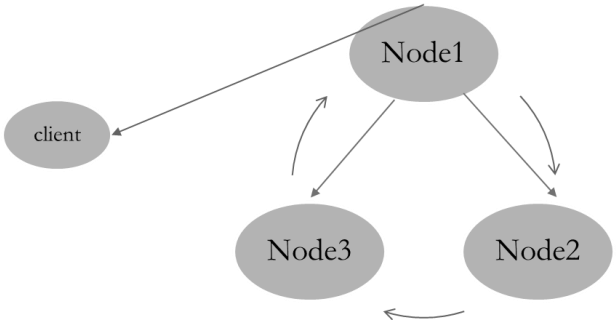

Fig. 6. Risposta del server al client e invio del multicast

V. TEST

Per valutare le prestazioni del sistema, sono stati eseguiti test di latenza utilizzando diverse configurazioni di thread concorrenti (20, 50 e 100). I test sono stati condotti su un'istanza EC2 t4g.large situata nel datacenter eu-south-1 (Milano), con tre server simulati in località differenti: Rome_Center, New_York_TimeSquare e Madrid_PlazaMajor. A ciascun server è stata associata una fake latency per simulare le condizioni di rete in scenari distribuiti: Rome_Center con 1 ms, New_York_TimeSquare con 64 ms e Madrid_PlazaMajor con 12 ms.

I test sono stati eseguiti per due modalità operative: sequential e causal. I risultati mostrano un confronto tra i tempi medi di risposta per ogni server in ciascuna configurazione tab1 e tab2.

TABLE I.

Latenza sequenziale			
Threads	Rome_center	New-york	Madrid
20	0.329s	0.300s	0.266s
50	0.231s	0.337s	0.296s
100	0.264s	0.335s	0.287s

TABLE II.

Latenza causale			
Threads	Rome_center	New-york	Madrid
20	0.104s	0.191s	0.108s
50	0.095s	0.127s	0.055s
100	0.090s	0.120s	0.058s

Dai risultati ottenuti, emerge chiaramente che la modalità causal presenta una latenza media inferiore rispetto alla modalità sequential. Questo è dovuto al fatto che la modalità causal, grazie al meccanismo di ordinamento causale, gestisce in modo più efficiente le operazioni concorrenti, riducendo i tempi di attesa complessivi. In particolare:
In modalità sequential, i tempi di risposta tendono ad aumentare con il crescere del numero di thread, evidenziando un impatto più significativo della latenza fittizia applicata ai server (soprattutto New_York_TimeSquare con 64 ms di ritardo). In modalità causal, si osserva una gestione migliore delle risorse e una distribuzione più bilanciata del carico, con latenze decisamente inferiori, anche per server con latenza elevata come New_York_TimeSquare. Questi test dimostrano l'efficacia dell'approccio causal nel ridurre la latenza media e migliorare le prestazioni complessive del

sistema in contesti di concorrenza elevata. È importante sottolineare che i risultati dipendono anche dalla connessione del client, poiché le prestazioni del sistema possono essere influenzate da fattori esterni come la velocità della rete del client e la sua posizione geografica rispetto ai server. Una connessione più lenta o instabile può amplificare le latenze complessive.

VI. Sviluppi futuri

Questo progetto prevede l'implementazione di un semplice datastore chiave-valore. Per migliorarne le funzionalità e renderlo più adatto a scenari reali, si potrebbero introdurre le seguenti caratteristiche:

- Meccanismi di Tolleranza ai Guasti: Attualmente, il sistema si basa su repliche consistenti, ma non prevede un meccanismo completo di tolleranza ai guasti. Potrebbe essere implementato un sistema di replication factor, in combinazione con protocolli di consenso distribuito come Raft o Paxos, per garantire la continuità operativa anche in caso di guasti di rete o fallimenti di nodi. Ciò consentirebbe al datastore di continuare a funzionare correttamente in condizioni di fault-tolerance.

- Bilanciamento del Carico: Si potrebbe migliorare il sistema introducendo un load balancer per distribuire uniformemente le richieste tra i nodi, evitando sovraccarichi durante periodi di traffico elevato. Questo permetterebbe di ottimizzare le prestazioni del datastore, garantendo una gestione efficiente delle risorse e migliorando la scalabilità del sistema.

- Persistenza su Disco: Attualmente, i dati vengono memorizzati in memoria, il che comporta una potenziale perdita di dati in caso di riavvio o crash del sistema. Si potrebbe introdurre un sistema di persistenza su disco utilizzando un meccanismo di log write-ahead, che registra le operazioni prima di applicarle in memoria. In questo modo, sarebbe possibile ripristinare lo stato del nodo dopo un crash, aumentando l'affidabilità e la durabilità del sistema.

- Algoritmi di Gossip: Per migliorare la propagazione delle informazioni tra i nodi, si potrebbe integrare un protocollo di gossip. Questo permetterebbe aggiornamenti rapidi e affidabili all'interno di reti distribuite, anche in presenza di latenza significativa, migliorando così la scalabilità e la robustezza del sistema.

- Supporto per Partizionamento e Scalabilità Orizzontale: Un'estensione del sistema potrebbe prevedere il supporto per il partizionamento delle chiavi o sharding, distribuendo i dati tra più nodi. Questa soluzione consentirebbe di gestire volumi di dati più grandi e di migliorare la scalabilità orizzontale del sistema. Con un sistema di partizionamento efficace, i carichi di lavoro potrebbero essere distribuiti tra più nodi, aumentando così la capacità complessiva.

- Monitoraggio e Logging Avanzato: Si potrebbe integrare il sistema con strumenti di monitoraggio e logging distribuito avanzato, come Prometheus o Elasticsearch, per facilitare il debugging, il controllo delle prestazioni e la diagnosi di eventuali problemi di rete o di sincronizzazione tra repliche. Questi strumenti permetterebbero un miglior controllo sullo stato del sistema e sulle sue prestazioni.

- Modelli di Coerenza Flessibili: Si potrebbe offrire supporto per diversi modelli di coerenza, come la coerenza eventuale o la coerenza forte, lasciando che il client scelga il modello più adatto al proprio caso d'uso. Questo approccio garantirebbe una maggiore flessibilità, consentendo agli utenti di bilanciare la consistenza dei dati con le prestazioni in base alle proprie esigenze applicative.

- Supporto Multi-datacenter: Infine, il sistema potrebbe essere esteso per supportare operazioni su più datacenter distribuiti geograficamente, affrontando le problematiche di geo-replicazione e latenza globale. Tecniche come il Geo-Replication Delay Compensation potrebbero essere utilizzate per mantenere la coerenza dei dati tra i datacenter, migliorando la resilienza del sistema su scala globale e offrendo un servizio più robusto e distribuito.

Questi miglioramenti porterebbero il sistema chiave-valore a un livello più avanzato, rendendolo più scalabile, affidabile e adatto ad ambienti reali e distribuiti.