Implementazione di un Sistema di Memorizzazione Chiave-Valore Distribuito con Garanzie di Consistenza Sequenziale e Causale

Progetto SDCC 2023-2024

Alessia Cinelli
Macroarea di Ingegneria Informatica
Università degli Studi di Roma Tor Vergata
Monte San Giovanni Campano, Italia
cinellialessia@gmail.com

ABSTRACT

L'obiettivo di questo documento è descrivere l'architettura, le scelte progettuali e l'implementazione di un sistema distribuito di memorizzazione chiave-valore che offre garanzie di consistenza sequenziale e causale. Le garanzie di consistenza rivestono un ruolo cruciale nei sistemi cloud distribuiti, vengono utilizzate per mantenere i dati coerenti tra diversi datastore replicati, evitando un accesso centralizzato che potrebbe causare ritardi di rete e inefficienze, specialmente in scenari in cui le distanze fisiche tra i nodi sono elevate.

INTRODUZIONE

L'applicazione sviluppata offre all'utente la possibilità di utilizzare un sistema di memorizzazione chiave-valore con opzioni di garanzie di consistenza a scelta tra: consistenza sequenziale e consistenza causale.

La consistenza sequenziale garantisce che il risultato di una qualunque esecuzione è quello ottenuto dall'esecuzione delle operazioni (di Get, Put e Delete) in un qualunque ordine sequenziale che conservi l'ordine di programma di ciascun processo.

La consistenza causale rilassa le restrizioni sull'ordine delle operazioni, consentendo un ordine parziale basato sulla relazione causa-effetto tra le operazioni. Questo modello di consistenza offre maggiore flessibilità e prestazioni in ambienti distribuiti, consentendo una maggiore scalabilità.

Offrire diverse opzioni di garanzie di consistenza ai client consente loro di adattare il sistema alle proprie esigenze specifiche e alle caratteristiche dell'applicazione, garantendo al contempo un funzionamento ottimale e una gestione efficiente dei dati distribuiti.

DESIGN DELLA SOLUZIONE

Il sistema di memorizzazione chiave-valore è progettato su un'architettura distribuita con più server replica. Utilizzando il linguaggio di programmazione GO, il sistema è stato implementato con l'ausilio di Docker Compose e Amazon AWS per garantire scalabilità ed efficienza.

I server replica offrono ai client la possibilità di eseguire operazioni di tipo Put, Get e Delete. La comunicazione è gestita tramite il protocollo di comunicazione basato su RPC (Remote Procedure Call).
Per garantire la consistenza sequenziale, è stato implementato l'algoritmo di Multicast Totalmente Ordinato, mentre per la consistenza causale, è stato adottato l'algoritmo di Multicast Causalmente Ordinato.

Per simulare le variazioni di latenza che possono verificarsi in un ambiente distribuito reale, si è introdotto un ritardo casuale prima di ciascuna chiamata RPC.

In questa documentazione, faremo differenza tra richiesta e messaggio. Una richiesta è un'operazione che il client effettua verso il server. Un messaggio è invece generato da un server a partire dalla richiesta del client, verrà utilizzato per la sua gestione, fino al processamento a livello applicativo.

DETTAGLI DELLA SOLUZIONE

Nella realizzazione dei due algoritmi si ha bisogno di effettuare due assunzioni:

1. Comunicazione affidabile.
2. Comunicazione FIFO Ordered: i messaggi inviati da un processo ad un altro vengono ricevuti nell'ordine esatto in cui sono stati inviati.

Per l'assunzione di comunicazione affidabile, si è utilizzato il protocollo TCP nelle chiamate RPC, così che sia banalmente soddisfatta.
Per garantire la comunicazione FIFO Ordered, si è implementato un approccio che presuppone che ogni client effettui le proprie richieste verso lo stesso server replica. Questa scelta è supportata dalla pratica comune nei sistemi distribuiti, in cui i client vengono reindirizzati al server replica geograficamente più vicino, garantendo così una comunicazione efficiente e veloce.
Per implementare questa assunzione:

1. Le richieste generate dai Client avranno ad esse associato un timestamp, che riflette l'istante di creazione della richiesta.
2. I server mantengono una struttura dati per ogni client, nella quale tengono traccia delle richieste attese da quel client.

3. Ogni volta che un server riceve una richiesta, verifica se il timestamp corrisponde a quello atteso per quel client. Solo se questa condizione è soddisfatta, il server può procedere con la generazione di un messaggio per eseguire l'algoritmo di multicast, altrimenti la goroutine bufferizza il messaggio e si mette in attesa di essere sbloccata.

Con queste condizioni, garantiamo che le richieste del client vengano processate dal server nello stesso ordine di invio.

Implementazione del Multicast Totalmente Ordinato:

Ciascun server replica mantiene un clock logico, che identifica il numero di richieste di cui lui è a conoscenza, avvenute nel sistema globale.

Alla ricezione di una richiesta di scrittura o lettura inviata da un client a un server replica, se la condizione di FIFO Ordered viene soddisfatta, viene associata ad essa il suo clock logico, generando così un messaggio. Successivamente, il server ricevente si comporta in maniera differente per gli eventi interni (Get) e per gli eventi esterni (Put e Delete).

Per un evento esterno il messaggio viene inviato in multicast a tutti i server replica, incluso se stesso, mentre un evento interno non viene trasmesso in multicast, ma viene aggiunto alla coda locale del server.

La coda locale di ciascun server viene mantenuta ordinata in base al timestamp associato al messaggio, a parità di timestamp si adotta un ordinamento alfabetico su un identificativo univoco associato al messaggio.

A seguito della ricezione del messaggio da parte di un server, inizia il vero e proprio algoritmo: ogni server replica aggiunge il messaggio alla propria coda locale e inoltra in multicast un acknowledgment (ack) relativo ad esso, confermandone la corretta ricezione. Ciascun server attende che gli ack da lui inviati siano consegnati prima di continuare il processamento.
Può capitare, a causa dei ritardi di rete, di ricevere un acknowledgment (ack) relativo a un messaggio di cui il server non è a conoscenza. In questo caso, per evitare ulteriori messaggi in rete, l'acknowledgment viene considerato come il messaggio stesso e viene conteggiato l'invio dell'ack.

Successivamente, sia che la richiesta sia un evento interno o esterno, i server replica controllano ciclicamente se è possibile inviare il messaggio a livello applicativo, al fine di rispondere al client.

Il messaggio verrà inoltrato a livello applicativo se:

1. Si trova in testa alla coda, cioè, ha un clock logico minore rispetto ad altri messaggi.
2. Ha ricevuto tanti ack quante sono le repliche dei server nel sistema, per controllare se il messaggio sia stato ricevuto da tuti i server replica.
3. In coda, per ciascuna replica del sistema, c'è un messaggio con clock maggiore di quello che si sta valutando.

In aggiunta a queste condizioni specificate dall'algoritmo, a causa dei ritardi di rete e di scheduling, è stata aggiunta un ulteriore condizione, un messaggio relativo ad un client A, viene inviato a livello applicativo solo se tutte le richieste di quel client A sono state processate.

Se tutte le condizioni sono rispettate:

1. Il messaggio verrà rimosso dalla coda.
2. Verrà aggiornato il clock scalare del server, calcolando il max tra il clock associato al messaggio ed il proprio e successivamente incrementandolo di uno per conteggiare l'evento.
3. Verrà eseguito a livello applicativo, effettuando la richiesta nel datastore replica e impostando la risposta per il client.

Considerando la terza condizione d'invio del messaggio a livello applicativo, per garantire un corretto funzionamento dei test effettuati, vengono automaticamente aggiunte a seguito dei test, lato client, delle operazioni di End identificate da una chiave specifica. In questo modo la condizione sarà verificata.
Questa specifica può essere abilitata o disabilitata con una variabile d'ambiente.

Se le condizioni di invio non sono rispettate, il messaggio viene bufferizzato in un'ulteriore coda locale, e la goroutine entra in uno stato di wait condizionato. Questa attesa terminerà solo quando la variabile di riferimento relativa al messaggio bufferizzato verrà impostata a true.
Ad ogni ack ricevuto e ad ogni messaggio che viene eseguito a livello applicativo, viene chiamata un ulteriore goroutine che scorre i messaggi bufferizzati per controllare se è possibile eseguire anch'essi a livello applicativo, e quindi sbloccare l'attesa.

Implementazione del Multicast Causalmente Ordinato:

Ciascun server replica mantiene un vettore di clock logici, di cui ciascun indice conteggia il numero di eventi avvenuti su una singola replica, di cui lui ne è a conoscenza.

Quando un server replica riceve una richiesta da parte del client, indipendentemente se è un evento interno o esterno, incrementa il proprio indice nel vettore di clock logici e lo associa alla richiesta del client, generando un messaggio per l'invio in multicast.
Successivamente, i server replica inseriscono il messaggio ricevuto nella propria coda locale e verificano periodicamente se è possibile inoltrare la richiesta a livello applicativo.

Il controllo per determinare se una richiesta può essere eseguita a livello applicativo avviene attraverso due condizioni:

1. Se il messaggio è esattamente il successivo atteso del processo che l'ha inviato, ovvero se il timestamp del

messaggio t(m) è uguale al vettore di clock logici del mittente Vj incrementato di uno t(m)[i] = Vj[i] + 1

2. Se, per ogni altro valore del clock logico, il processo che desidera inviare la richiesta a livello applicativo ha già visto tutti i messaggi inviati da altri processi. In altre parole, verifica se il vettore di clock logici del mittente (Vj è maggiore o uguale ai vettori di clock logici di tutti gli altri processi (Vk) per ogni processo k != i (ovvero Pj ha visto almeno gli stessi messaggi di Pk visti da Pi).

Al verificarsi di tutte le condizioni:

1. Il messaggio viene rimosso dalla coda locale.
2. Viene aggiornato l'indice del valore del clock vettoriale locale relativo all'Id del server che ha ricevuto la richiesta dal client.
3. Viene eseguito a livello applicativo, effettuando la richiesta nel datastore replica e impostando la risposta per il client.

Similmente all'implementazione del Multicast Totalmente Ordinato, se almeno una delle condizioni non è soddisfatta, il messaggio verrà bufferizzato in un ulteriore coda locale e la goroutine entrerà in attesa. Si verifica se ci sono messaggi bufferizzati che possono essere eseguiti ogni volta che viene processato un messaggio, se il controllo va a buon fine, il messaggio viene rimosso dalla coda, la variabile condizionale viene impostata a true e la goroutine riprende la sua esecuzione.

DISCUSSIONE

i. Una problematica affrontata riguardante la consistenza sequenziale evidenziava il rischio che un server potesse ricevere un acknowledgment (ack) di un messaggio di cui non era ancora a conoscenza (non era nella coda locale) a causa dei ritardi nella rete. Per risolvere tale situazione, è stata adottata una soluzione che sfrutta il valore di ritorno della chiamata RPC: se un server riceve un ack relativo a un messaggio che non ha ancora ricevuto, risponde con un valore booleano false, ed il mittente avrebbe re-inviato l'ack. Tuttavia, questa soluzione ha subito un'ulteriore evoluzione per migliorare l'algoritmo, in particolare, è stata implementata una procedura aggiuntiva in cui, se il server riceve un ack di un messaggio che non è ancora presente nella sua coda locale, lo aggiunge alla coda e incrementa il contatore degli ack. Questo approccio ha permesso di evitare ulteriori messaggi in rete, ottimizzando l'efficienza del sistema e mitigando i problemi legati ai ritardi di rete.

ii. La problematica riscontrata riguardante la consistenza causale evidenzia la sfida nell'identificare correttamente la relazione causa-effetto tra due tipi di richieste: una di lettura (read) e una di scrittura (write) su uno stesso key. In particolare, si è verificato che la prima richiesta ricevuta da un server era una richiesta di lettura e la prima ricevuta da un altro server era una richiesta di scrittura, si dovrebbe prevedere una relazione di causa effetto, di cui la scrittura sia la causa della lettura. Le condizioni per l'inoltro a livello applicativo sembravano essere soddisfatte ma non veniva correttamente identificata la relazione causale. Ciò poteva portare al processamento della richiesta di lettura prima della richiesta di scrittura, compromettendo la consistenza dei dati. Per risolvere questa problematica, è stata implementata una soluzione che verifica se la lettura nel datastore va a buon fine o meno. In particolare, la richiesta di lettura viene inviata a livello applicativo solo se la chiave (key) associata alla richiesta è presente nel datastore. Questo approccio garantisce che la richiesta di lettura venga eseguita solo dopo che la richiesta di scrittura corrispondente ha avuto successo e che quindi sia stata effettivamente introdotta la modifica nel datastore. In questo modo, si assicura una corretta sequenzialità delle operazioni e si mantiene la coerenza dei dati all'interno del sistema distribuito.

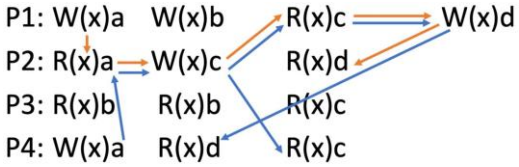


Figura 1: Scenario descritto

RISULTATI

I risultati ottenuti dimostrano che il sistema di memorizzazione chiave-valore, implementato per garantire sia la consistenza sequenziale che causale, riesce a mantenere un ordine coerente delle operazioni. Tuttavia, durante la realizzazione del Multicast Totalmente Ordinato, è emerso un alto utilizzo dei mutex per garantire la corretta sincronizzazione dell'accesso alle risorse e un alto numero di messaggi scambiati, a differenza del Multicast Causalmente Ordinato, poiché esso non utilizza un sistema di acknowledgment.

Entrambe queste condizioni possono influenzare le prestazioni del sistema, poiché sia l'acquisizione e il rilascio dei mutex che l'attesa della ricezione di ciascun acknowledgment comportano un certo overhead.

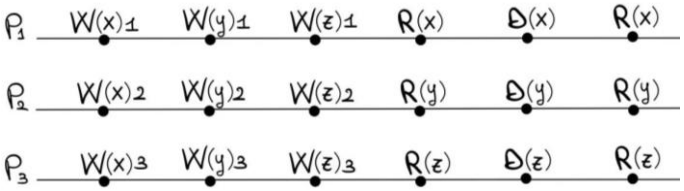
In sintesi, utilizzare una consistenza sequenziale rappresenta un costo maggiore in termini di prestazioni, al contrario della consistenza causale; tuttavia, con la consistenza causale si perde l'idea di avere una sola replica, mentre per la consistenza sequenziale questa idea è soddisfatta.

TEST EFFETTUATI

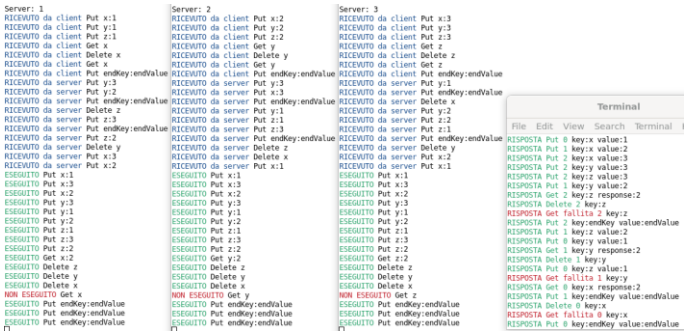
Qui di seguito mostriamo alcuni dei test effettuati per mostrare il comportamento delle rispettive garanzie di consistenza.

Garanzie di consistenza sequenziale

Questo test sulla consistenza sequenziale punta nel mostrare il funzionamento di tutte le possibili richieste effettuabili da un client ad un server, e di come viene garantita la consistenza sequenziale.

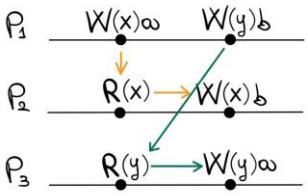


Dall'esecuzione del programma, è evidente che tutte le richieste sono state elaborate da tutti i server replica in un ordine coerente, rispettando così la consistenza sequenziale. Tuttavia, per interpretare correttamente l'output del terminale, è necessario considerare che ogni client invia le proprie richieste a uno specifico server replica. Ad esempio, le risposte visualizzate per il client denominato 0 sono indicate dalla dicitura 'RISPOSTA operation 0', mentre le risposte per gli altri client sono etichettate di conseguenza. Questa convenzione aiuta a seguire l'ordine delle operazioni e a comprendere come vengono processate le richieste da parte dei diversi client e server replica.



Garanzie di consistenza causale

Questo test basico sulla consistenza sequenziale mostra, concentrandoci sul Server 2 o 3, come nonostante avessero ricevuto due richieste di Get e successivamente una di Put, sia stata riconosciuta la relazione di causalità ed eseguita prima la richiesta di Put.



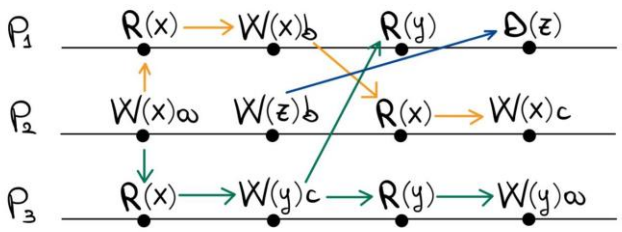

```
Inserisci il numero dell'operazione desiderata: 1

Il test basico sulla consistenza causale invia in goroutine:
- una richiesta di put x:a e put y:b al server1
- una richiesta di get x e put x:b al server2
- una richiesta di get y e put y:a al server3
RUN Get x
RUN Get y
RUN Put x:a
RISPOSTA Put 0 key:x value:a
RUN Put y:b
RISPOSTA Get 1 key:x response:a
RUN Put x:b
RISPOSTA Put 0 key:y value:b
RISPOSTA Get 2 key:y response:b
RUN Put y:a
RISPOSTA Put 1 key:x value:b
RISPOSTA Put 2 key:y value:a
```

```
Terminal
File Edit View Search Term
Server: 1
RICEVUTO da client Put x:a
ESEGUITO Put x:a
RICEVUTO da server Get y
RICEVUTO da server Get x
ESEGUITO Get x:a
RICEVUTO da client Put y:b
ESEGUITO Put y:b
ESEGUITO Get y:b
RICEVUTO da server Put x:b
ESEGUITO Put x:b
RICEVUTO da server Put y:a
ESEGUITO Put y:a
```

```
Terminal
File Edit View Search Term
Server: 2
RICEVUTO da client Get x
RICEVUTO da server Get y
RICEVUTO da server Put x:a
ESEGUITO Put x:a
ESEGUITO Get x:a
RICEVUTO da client Put x:b
RICEVUTO da server Put y:b
ESEGUITO Put y:b
ESEGUITO Get y:b
ESEGUITO Put x:b
RICEVUTO da server Put y:a
ESEGUITO Put y:a
```

```
Terminal
File Edit View Search Term
Server: 3
RICEVUTO da client Get y
RICEVUTO da server Get x
RICEVUTO da server Put x:a
ESEGUITO Put x:a
ESEGUITO Get x:a
RICEVUTO da server Put y:b
ESEGUITO Put y:b
ESEGUITO Get y:b
RICEVUTO da client Put y:a
RICEVUTO da server Put x:b
ESEGUITO Put x:b
ESEGUITO Put y:a
```

Quest'ulteriore test sulla consistenza causale è utile nel mostrare come anche in presenza di molteplici relazioni di causa-effetto, l'algoritmo proposto funziona correttamente.




```
In questo complexTestCE vengono inviate in goroutine:
- una richiesta di get x, put x:b, get x, del z al server1
- una richiesta di put x:a, put z:b, get x, put x:c al server2
- una richiesta di get x, put y:c, get y, put y:c al server3
RUN Get x
RUN Get x
RUN Put x:a
RISPOSTA Put 1 key:x value:a
RISPOSTA Get 0 key:x response:a
RUN Put z:b
RISPOSTA Get 2 key:x response:a
RUN Put x:b
RUN Put y:c
RISPOSTA Put 0 key:x value:b
RISPOSTA Put 1 key:z value:b
RUN Get y
RISPOSTA Put 2 key:y value:c
RUN Get x
RUN Get y
RISPOSTA Get 0 key:y response:c
RISPOSTA Get 2 key:y response:c
RUN Delete z
RISPOSTA Get 1 key:x response:b
RUN Put y:a
RUN Put x:c
RISPOSTA Put 1 key:x value:c
RISPOSTA Put 2 key:y value:a
RISPOSTA Delete 0 key:z
Consistenza causale, scegli il t
1. Basic Test
2. Medium Test
3. Complex Test
4. Esci
```

```
Terminal
File Edit View Search Term
Server: 1
RICEVUTO da client Get x:
RICEVUTO da server Put x:a
ESEGUITO Put x:a
ESEGUITO Get x:a
RICEVUTO da server Get x:
ESEGUITO Get x:a
RICEVUTO da client Put z:b
RICEVUTO da server Put y:c
ESEGUITO Put y:c
RICEVUTO da server Put x:b
ESEGUITO Put x:b
ESEGUITO Put z:b
RICEVUTO da client Get y:
RICEVUTO da server Get x:
ESEGUITO Get y:c
RICEVUTO da server Get y:
ESEGUITO Get y:c
RICEVUTO da client Delete z
ESEGUITO Delete z
RICEVUTO da server Put y:a
ESEGUITO Put y:a
RICEVUTO da server Put x:c
ESEGUITO Put x:c
```

```
Terminal
File Edit View Search Term
Server: 2
RICEVUTO da client Put x:a
RICEVUTO da server Get x:
ESEGUITO Put x:a
ESEGUITO Get x:a
RICEVUTO da server Get x:
ESEGUITO Get x:a
RICEVUTO da client Put z:b
RICEVUTO da server Put y:c
ESEGUITO Put y:c
RICEVUTO da server Put x:b
ESEGUITO Put x:b
ESEGUITO Put z:b
RICEVUTO da client Get x:
RICEVUTO da server Get y:
ESEGUITO Get y:c
RICEVUTO da server Get y:
ESEGUITO Get y:c
RICEVUTO da client Put x:c
RICEVUTO da server Delete z
ESEGUITO Delete z
ESEGUITO Put x:c
RICEVUTO da server Put y:a
ESEGUITO Put y:a
```

```
Terminal
File Edit View Search Termi
Server: 3
RICEVUTO da client Get x:
RICEVUTO da server Get x:
RICEVUTO da server Put x:a
ESEGUITO Put x:a
ESEGUITO Get x:a
ESEGUITO Put y:c
RICEVUTO da server Put z:b
ESEGUITO Put z:b
ESEGUITO Put x:b
ESEGUITO Put y:c
RICEVUTO da client Get y:
RICEVUTO da server Get y:
ESEGUITO Get y:c
RICEVUTO da server Get x:
ESEGUITO Get x:b
RICEVUTO da client Put y:a
RICEVUTO da server Put x:c
ESEGUITO Put x:c
ESEGUITO Put y:a
RICEVUTO da server Delete z
ESEGUITO Delete z
```