Realizzazione di uno Store Key-Value con Garanzie di Consistenza Sequenziale o Causale

Applicazione del Multicast Totalmente Ordinato o Causalmente Ordinato

Matteo Pallagrosi
Università degli Studi di Roma
"Tor Vergata"
Roma, Italia
matteo.pallagrosi18@gmail.com

ABSTRACT

In questo documento è descritta l'architettura e l'implementazione di un data-store di coppie chiave-valore, realizzato fornendo garanzie di consistenza sequenziale oppure causale, ottenute rispettivamente attraverso l'applicazione di un algoritmo di multicast totalmente ordinato e causalmente ordinato. Tali garanzie si rivelano fondamentali nel contesto della replicazione dei dati su molteplici nodi, in applicazioni distribuite su larga scala geografica, dove all'utente è tipicamente fornito l'accesso alla replica più vicina, così da ridurre la latenza delle operazioni e migliorare il sistema dal punto di vista della availability, della scalabilità orizzontale e della tolleranza ai guasti, ma introducendo la necessità di gestione della consistenza tra le diverse repliche.

KEYWORDS

Multicast totalmente ordinato, Multicast causalmente ordinato, Consistenza sequenziale, Consistenza causale, Replicazione, Sistemi distribuiti, Clock logico scalare, Clock logico vettoriale

1 Introduzione

Il sistema realizzato fornisce la possibilità di interagire con lo store key-value attraverso le operazioni di:

- **PUT**: inserimento di una coppia chiave-valore. Se la chiave è già presente il valore associato viene aggiornato.
- **GET**: lettura del valore associato ad una data chiave.
- **DELETE**: rimozione di una coppia chiave-valore associata ad una data chiave.

È inoltre possibile scegliere se ottenere garanzia di consistenza sequenziale o causale e il numero di repliche distribuite dello store chiave-valore.

La consistenza sequenziale garantisce che il risultato di una qualunque esecuzione concorrente delle operazioni sulle diverse repliche dello store sia uguale a quello ottenuto se le operazioni fossero eseguite secondo un ordine sequenziale, e in modo tale che le operazioni richieste ad ogni replica compaiano nella sequenza realizzata secondo l'ordine di programma, ossia l'ordine con cui ogni client inoltra le operazioni alla replica a cui è connesso. Da ciò consegue che, quando le repliche eseguono operazioni di update dello store in concorrenza, una qualunque alternanza di queste operazioni è accettabile, sempre nel rispetto dell'ordine di programma, purché tutte le repliche vedano la medesima alternanza delle operazioni. Poiché ogni operazione di update, ossia PUT e DELETE, richiesta ad una specifica replica si propaga verso le altre repliche attraverso un opportuno scambio di messaggi in multicast, tale vincolo sull'ordinamento delle operazioni si traduce nella necessità di garantire che tutti i messaggi propagati vengano consegnati ad ogni destinatario nello stesso ordine, e ciò può essere ottenuto mediante l'applicazione del multicast totalmente ordinato.

La consistenza causale rilassa invece tali vincoli, imponendo che solo le operazioni di update in potenziale relazione di causa/effetto siano realizzate da tutte le repliche nello stesso ordine, permettendo invece loro di realizzare le operazioni di update concorrenti in un ordine qualunque. Tale comportamento si realizza tramite la gestione di uno scambio di messaggi mediante multicast causalmente ordinato. Si perde quindi l'illusione della singola copia dello store che invece la consistenza sequenziale garantisce, ma a favore di un miglioramento della latenza nell'accesso ai dati.

2 Progettazione della soluzione

La soluzione distribuita fornita si offre come alternativa a soluzioni di tipo centralizzato, dove è presente un coordinatore, che riceve dalle diverse repliche le operazioni di update, e assegnando ad ogni operazione un numero di sequenza univoco, invia messaggi in multicast alle diverse repliche dello store, che eseguono le operazioni associate a tali messaggi secondo l'ordine definito dal coordinatore. La soluzione proposta non presenta invece alcun elemento centralizzato, risolvendo in tal modo problemi di scalabilità e single point of failure.

2.1 Assunzioni

La soluzione proposta è realizzata basandosi sugli algoritmi di multicast totalmente e causalmente ordinato. Entrambi tali algoritmi, al fine di perseguire l'ordinamento dei messaggi desiderato, prevedono le seguenti assunzioni, rispettate anche nella realizzazione del sistema qui presentato:

1) Comunicazione affidabile: il sistema non prevede la perdita dei messaggi.
2) Comunicazione *FIFO ordered*: i messaggi inviati dalla replica p_i a p_j sono ricevuti da p_j nello stesso ordine in cui sono stati inviati da p_i.

Per garantire tali assunzioni sono stati adottati i seguenti meccanismi:

1) La comunicazione tra client e la replica con cui interagisce, e lo scambio di messaggi tra le diverse repliche al fine di garantire la consistenza richiesta, sono realizzate utilizzando TCP come protocollo di comunicazione a livello di trasporto, il quale fornisce comunicazione affidabile attraverso meccanismi di ritrasmissione, acknowledgment ed eliminazione dei duplicati.
2) L'ordine FIFO dei messaggi scambiati tra coppie di server, essendo il sistema caratterizzato da possibili ritardi nello scambio dei messaggi tra le repliche, è garantito assegnando a ciascun messaggio inviato da una replica un numero di sequenza, come proposto in [2]. Ogni replica mantiene quindi l'informazione sul numero di sequenza del successivo messaggio atteso da ogni altra replica nel sistema. Qualora un messaggio venisse quindi ricevuto con un numero di sequenza diverso da quello atteso, a causa di un ritardo di rete sul messaggio corretto, tale messaggio viene collocato in una coda che la replica ricevente mantiene per quel mittente, in attesa della ricezione del messaggio con il numero di sequenza corretto. In un sistema con N repliche, ognuna di esse mantiene quindi N-1 code, una per ogni altra replica nel sistema, al fine di gestire il riordinamento FIFO dei messaggi provenienti da quella replica. Una volta ricevuto il messaggio nell'ordine corretto, questo viene consegnato a tutti gli effetti al livello applicativo, che realizza la ricezione secondo la semantica specifica del multicast considerato. La realizzazione di tale meccanismo si è resa necessaria a causa del fatto che i ritardi di rete sono simulati a livello applicativo, e quindi la consegna fuori ordine dei messaggi non può essere risolta da TCP.

2.2 Descrizione degli algoritmi

2.2.1 Multicast totalmente ordinato. Al fine di realizzare un unico ordinamento sequenziale delle operazioni realizzato da ogni replica dello store, questo algoritmo di multicast utilizza il meccanismo del Clock logico scalare, proposto da Lamport. Ogni replica mantiene infatti un contatore locale C_i. Questi contatori sono aggiornati seguendo i passi qui descritti:

1) Prima di eseguire un evento qualunque, come inviare o ricevere un messaggio o processare un evento interno, la replica incrementa il suo contatore $Ci : Ci \leftarrow Ci + 1$.
2) Quando la replica p_i invia un messaggio m ad una qualunque replica p_j, inserisce nel messaggio un timestamp $ts(m)$ pari al valore del suo clock corrente Ci.
3) Alla ricezione di un messaggio m, la replica p_j imposta il suo clock locale al valore $Cj \leftarrow max\{Cj , ts(m)\}$.

È necessario introdurre però un ulteriore elemento. Poiché due eventi possono essere associati al medesimo valore del clock scalare, essendo questo locale alla singola replica, al fine di ottenere un ordinamento totale tra gli eventi, il timestamp da associare ai messaggi contiene oltre al valore del clock corrente, anche l'indice della replica che propaga il messaggio. Quindi se due messaggi m e m' sono associati rispettivamente ai timestamp $<Ci,i>$ *e* $<Cj,j>$, m precede m' se:

1) $Ci < Cj$, oppure
2) Ci = Cj e i < j

Il multicast totalmente ordinato è quindi un algoritmo completamente distribuito, che utilizza il clock logico scalare, aggiornato da ogni replica secondo i passi descritti. Ogni messaggio è quindi etichettato con il clock scalare locale alla replica che invia il messaggio e con l'indice che identifica la replica. Ogni qualvolta un client richiede un'operazione di update (PUT o DELETE) alla replica p_i a cui è connesso, la replica deve propagare tale operazione alle altre per mantenere la consistenza dello store, secondo le modalità qui descritte:

1) p_i invia in multicast agli altri processi, incluso a sé stesso, il messaggio di update m.
2) Ogni processo ricevente p_j mette il messaggio m ricevuto in una coda locale, ordinata in base al valore del timestamp.
3) p_j invia in multicast agli altri processi un messaggio di ack, per confermare l'avvenuta ricezione di m.
4) p_j consegna m all'applicazione, ossia processa l'operazione corrispondente, se sono verificate le seguenti condizioni:
    a. m è in testa alla coda di p_j.
    b. Per ogni processo p_k (k diverso da j) c'è un messaggio (update o ack) nella coda di p_j con un timestamp maggiore di m.
5) Dopo aver processato il messaggio, p_j rimuove dalla coda locale il messaggio e tutti gli ack associati.

Le condizioni indicate assicurano che il messaggio m venga processato solo quando p_j è certo di non poter ricevere da nessun'altra replica un messaggio con timestamp minore o uguale.

Si sottolinea come oltre ai messaggi di update, anche i messaggi di ack presentino un timestamp associato (sempre secondo le regole di aggiornamento del clock e di assegnamento ai messaggi indicate in precedenza). Tali messaggi di ack, per costruzione stessa dell'algoritmo di aggiornamento del clock scalare, presentano necessariamente un timestamp maggiore del timestamp del messaggio di cui realizzano l'acknowledgment, poiché l'invio degli ack è sicuramente successivo alla ricezione del messaggio di update da parte delle repliche. Poiché il messaggio di update è idealmente inviato da ogni replica anche a sé stessa, anche la replica che avvia la propagazione del messaggio invia ad ogni altra replica l'ack relativo a quel messaggio. L'utilizzo dei timestamp anche nei messaggi di ack garantisce che le condizioni a. e b. producano l'ordinamento sequenziale dei messaggi come indicato in precedenza, rendendo non necessaria la ricezione di tutti gli ack relativi ad un certo messaggio. È sufficiente che la replica abbia ricevuto da ogni altro processo un messaggio (ack o update) con timestamp maggiore del messaggio in testa alla coda. Ogni replica, infatti, per mostrare che è in accordo sull'ordinamento sequenziale delle operazioni che ogni processo svolge, deve semplicemente "reagire" ad un messaggio ricevuto con un messaggio di ack o update inviato in multicast. Questo permette di mitigare il problema dei ritardi sui messaggi di ack, permettendo di processare il messaggio anche quando non sono stati ancora ricevuti tutti.

L'operazione di GET è considerata come un evento interno, poiché viene eseguita solo localmente alla replica a cui è richiesta, e non deve essere quindi propagata alle altre repliche, poiché non modifica lo stato dello store. Per rispettare l'ordine di programma, la replica deve comunque incrementare il clock, prima di inoltrare idealmente il messaggio di GET a sé stessa, ossia collocarlo nella propria coda locale, per poi estrarlo quando si trova in testa a tale coda.

2.2.2 Multicast causalmente ordinato. L'obiettivo di questo algoritmo di multicast è quello di consegnare un messaggio all'applicazione, ossia processare l'operazione corrispondente, solo se tutti i messaggi che lo precedono causalmente sono già stati consegnati. Un modo per poter identificare i messaggi in potenziale relazione di causa-effetto è quello di utilizzare i Clock logici vettoriali, che a differenza dei clock scalari, riescono a catturare la causalità nello scambio di messaggi. Ogni replica mantiene quindi localmente un clock vettoriale *Vi*, caratterizzato come un vettore di N elementi, dove N è il numero di repliche dello store. Tale vettore presenta le seguenti caratteristiche:

- $V_i[i]$ rappresenta il numero di eventi avvenuti presso il processo p_i.
- Se $V_i[j] = k$, allora p_i sa che sono avvenuti k eventi presso p_j.

L'aggiornamento del clock vettoriale realizzato nel multicast causalmente ordinato differisce da quello utilizzato in altri contesti. Inerentemente a questa tipologia di multicast, il clock è infatti aggiornato nel seguente modo:

- Prima di inviare un messaggio, la replica incrementa il suo clock $V_i[i]$: $V_i[i] \leftarrow V_i[i] + 1$. Tale clock sarà assegnato al messaggio inviato.

- Quando una replica realizza la *delivery* di un messaggio *m* con timestamp *ts(m),* aggiorna il valore del suo clock in modo tale che $V_i[k] \leftarrow max\{V_i[k], ts(m)[k]\}$ per ogni k.

Si sottolinea la differenza tra la *receive* di un messaggio, momento in cui tale messaggio raggiunge la replica, e l'effettiva *delivery* del messaggio, ossia la consegna allo strato applicativo, che comporta la realizzazione dell'operazione associata al messaggio stesso. Per il corretto funzionamento dell'algoritmo di multicast l'aggiornamento deve essere realizzato sulla *send* e sulla *delivery*, ma non sulla *receive.* Le condizioni necessarie per realizzare la delivery sono indicate nel proseguimento della trattazione.

Considerati i meccanismi di aggiornamento del clock vettoriale presentati, ogni qualvolta un client richiede un'operazione di update (PUT o DELETE) alla replica p_i a cui è connesso, la replica realizza tale operazione e la propaga alle altre per mantenere la consistenza causale dello store, secondo le modalità qui descritte:

1) p_i inoltra il messaggio *m* di update alle altre repliche assegnandogli come timestamp il suo clock vettoriale *Vi* corrente.
2) p_j riceve il messaggio *m* (*receive*) da p_i e ne ritarda la consegna (*delivery*), ossia lo colloca in una coda di attesa, finché non son verificate le seguenti condizioni:
    a. $ts(m)[i] = V_j[i] + 1$
    b. $ts(m)[k] \leq Vj[k]$ per ogni $k \neq i$

La condizione a. garantisce che il messaggio di cui stiamo realizzando la consegna allo strato applicativo sia il messaggio successivo che p_j si aspetta da p_i. La condizione b. garantisce che p_j ha visto (e quindi conosce gli eventi associati) almeno lo stesso numero di messaggi visti da p_i, provenienti da ogni altra replica *k*.
Queste condizioni permettono ad una replica di scoprire se il messaggio ricevuto rappresenta un effetto, di cui non ha ancora ricevuto la causa, ed eventualmente ritardarne la consegna finché non è stata ricostruita la causalità.
Si evidenzia che, quando una replica riceve una richiesta da un client, non deve ritardare la consegna della richiesta a sé stessa, poiché la causalità in questo caso è data dall'ordine con cui le richieste da quel client sono ricevute, assumendo la garanzia di ricezione in ordine delle richieste assicurata da TCP nell'interazione tra client e replica.

Anche in questo caso la GET è considerata come un evento interno, e quindi non deve essere propagata alle altre repliche.

3 Dettagli implementativi della soluzione

Il sistema è stato realizzato utilizzando Go come linguaggio di programmazione. Si analizzano quindi le scelte implementative adottate.

3.1 Comunicazione

3.1.1 Client-server. L'interazione tra i client e le repliche è ottenuta utilizzando la libreria nativa di Go *net/rpc*, che realizza la comunicazione attraverso il meccanismo delle *Remote Procedure Call*. Le repliche, che costituiscono i server in questa interazione client-server, espongono un'interfaccia che permette l'interazione con lo store key-value tramite le operazioni di *Get*, *Put* e *Delete*.

3.1.2 Server-server. L'interazione tra le repliche, necessario per propagare i messaggi di update con l'obiettivo di mantenere la consistenza desiderata dello store, è realizzata sempre tramite la libreria *net* nativa di Go, ma utilizzando uno scambio di messaggi codificati in formato json tramite le funzioni della libreria *encoding/json*, quindi senza utilizzare i meccanismi rpc. In entrambi i casi tutte le comunicazioni utilizzano TCP a livello di trasporto.

Ogni replica, quindi, utilizza una specifica porta per l'interazione con i client tramite *rpc*, e una diversa porta, non nota ai client, per realizzare la propagazione dei messaggi tra le repliche.

3.1.3 Network delay. Il ritardo di rete nello scambio di messaggi tra le diverse repliche è simulato durante l'invio di ogni messaggio, che viene realizzato dopo un quantitativo casuale di millisecondi interrompendo la routine di invio con una *Sleep*. Questo rende necessario gestire in ricezione il riordinamento FIFO dei messaggi tra ogni coppia di server, secondo il meccanismo presentato nella sezione delle Assunzioni.

3.2 Accesso alle sezioni critiche

Poiché le repliche realizzano uno scambio di messaggi in multicast, e invio e ricezione dei messaggi sono gestiti tramite delle goroutine in modo da permettere ad ogni replica di realizzare tali operazioni in parallelo, possono verificarsi simultanee richieste di accesso alle strutture mantenute da ogni replica, come il clock locale, oppure la copia locale dello store key-value. La manipolazione di tali strutture rappresenta quindi una sezione critica, gestita tramite l'utilizzo di mutex forniti dalla libreria *sync* di Go. Questo garantisce che un'unica routine per volta abbia accesso esclusivo alle strutture dati, garantendo la corretta esecuzione degli algoritmi di mutlicast e aggiornamento dei clock.

3.3 Criticità nel multicast totalmente ordinato

3.3.1 Acknowledgment. Ogni messaggio è identificato tramite un ID univoco localmente alla singola replica. Per confermare l'avvenuta ricezione di un messaggio, ogni replica invia un messaggio di ack, che deve essere associato al messaggio di cui realizza l'acknowledgment. Poiché gli ID dei messaggi sono locali ad ogni replica, per realizzare questa corrispondenza messaggio-ack, viene inserito nell'ack l'ID del messaggio associato e l'ID del server che ha avviato la propagazione di quel messaggio. Tale coppia di informazioni è infatti univoca in tutto il sistema distribuito e permette di realizzare la corrispondenza 1-1 tra messaggio e ack.

3.3.2 Rimozione degli ack. Quando un messaggio viene estratto dalla coda, tutti gli ack relativi a quel messaggio correntemente presenti nella coda vengono rimossi. Come i messaggi di update, anche i messaggi di ack possono subire ritardi, e la condizione b. del multicast totalmente ordinato permette di processare un messaggio anche quando non sono stati ancora ricevuti tutti gli ack. A questo consegue che, quando un messaggio di ack si trova in testa alla coda, questo necessariamente risulta essere un ack arrivato in ritardo per un messaggio già estratto dalla coda e processato in precedenza, poiché ogni ack ha un clock maggiore del messaggio di cui realizza l'acknowledgment, e quindi lo succede nella coda ordinata per timestamp. Per tale motivo ogni qualvolta si incontra un messaggio di ack in testa alla coda, questo può essere scartato, proseguendo nel processamento dei messaggi che lo seguono in coda.
Si sottolinea come il multicast causalmente ordinato non presenti tali criticità, poiché non utilizza messaggi di ack.

3.3.3 Invio del messaggio di update a sé stessi. Quando una replica deve propagare un'operazione di update alle altre, idealmente effettua questa propagazione anche verso sé stessa. A livello implementativo ciò si realizza collocando il messaggio nella propria coda locale, sempre a seguito di un opportuno aggiornamento del clock, senza eseguire una effettiva *send*, e quindi senza subire un ritardo comunicativo che non si ritiene sia ragionevole avere quando l'invio da una replica è diretto alla replica stessa. Affinché l'algoritmo di multicast operi correttamente, è però necessario che la replica, dopo aver collocato il messaggio nella propria coda ordinata secondo il timestamp, inoltri comunque un ack di avvenuta ricezione a tutti gli altri, perché ciò permette che la condizione b. per il processamento dei messaggi possa verificarsi, anche in assenza di altri messaggi di update propagati.

3.3.4 Recupero del valore ottenuto tramite Get. Quando il client richiede l'esecuzione di una operazione di Get, un messaggio che incapsula tale operazione è collocato nella coda ordinata per timestamp, per poi essere estratto successivamente dalla coda locale quando tale messaggio risulta essere in testa alla coda. Il messaggio di Get richiede che sia verificata solo la condizione a. per il processamento, in modo da rispettare l'ordine di programma richiesto dal client alla replica. La condizione b. non risulta

necessaria poiché la Get non comporta alcuna propagazione verso le altre repliche, ma rappresenta solo un evento interno alla replica stessa. Poiché la ricezione della richiesta da parte del client e l'effettiva estrazione del messaggio dalla coda, con conseguente esecuzione dell'operazione di Get operano su goroutine differenti, il passaggio del valore letto è effettuato utilizzando un canale tra le due routine, mediante il meccanismo nativo in Go dei *channel*. Ciò comporta che il client rimanga in attesa finché non viene letto il valore richiesto. L'update e le delete non risultano invece bloccanti, poiché possono essere eseguite in maniera asincrona rispetto alla richiesta da parte del client. Il multicast totalmente ordinato garantisce però che vengano effettuate secondo l'ordine richiesto, su tutte le repliche.

3.4 Criticità nel multicast causalmente ordinato

3.4.1 Gestione della delivery dei messaggi. Quando una replica riceve un messaggio di update, verifica che le condizioni di delivery siano rispettate. In caso positivo l'operazione può essere processata, altrimenti inserisce il messaggio in una coda di attesa.
La delivery di un messaggio comporta l'aggiornamento del clock vettoriale, secondo la procedura indicata nella sezione 2.2.2. Tale aggiornamento potrebbe permettere la delivery dei messaggi attualmente in coda. Per tale motivo dopo aver effettuato una delivery, l'implementazione realizzata controlla che possano essere estratti ulteriori messaggi, reiterando ciclicamente questo controllo. L'estrazione termina nel momento in cui in coda di attesa non è più presente alcun messaggio che rispetta le condizioni di delivery.

3.4.2 Gestione del clock nella Get. Quando un client richiede una operazione di Get, considerata come evento interno alla replica, tale operazione non richiede l'incremento del clock vettoriale. Tale clock, infatti, mantiene l'informazione sul numero di eventi avvenuti localmente, e gli eventi noti provenienti dalle altre repliche. Poiché l'operazione di Get non è propagata alle altre repliche, non rappresenta un evento di cui quest'ultime possono venire a conoscenza, e quindi non richiede l'incremento del clock.

4 Deployment

L'istanziazione del sistema avviene tramite Docker, in particolare utilizzando *Docker Compose* per la gestione dei container che costituiscono i diversi nodi del sistema distribuito presentato.

Ogni replica è istanziata su un container, descritto mediante un apposito *Dockerfile*, che determina le operazioni da eseguire per la corretta creazione del container, qui di seguito riportate:

```
FROM golang:latest

WORKDIR /app

# Copia i file di modulo e scarica le dipendenze
COPY go.mod go.sum ./
RUN go mod download

# Copia l'applicazione
COPY . .

# Compila il server
RUN go build -o server-replica ./server
```

Listing 1. *Dockerfile* di una replica dello store.

Come immagine di base è utilizzata *golang*, che contiene tutti gli strumenti necessari a realizzare la build di applicazioni scritte in Go e la loro esecuzione.
I client invece sono descritti mediante un secondo Dockerfile, simile a quello utilizzato per le repliche, con la differenza che realizzano la build del sorgente in Go per il client.
Docker Compose si occupa di istanziare i diversi container delle repliche e dei client, descrivendo i vari servizi da lanciare in un file *docker-compose.yml*, a partire dai *Dockerfile* descritti in precedenza. Ai fini della simulazione del sistema, per avere pieno controllo sulle operazioni che i diversi client di test realizzano verso le repliche a cui sono connessi, tutti i client sono lanciati all'interno di uno stesso container. Di fatto tali client sono realizzati a livello implementativo come *goroutine* che in parallelo richiedono operazioni alle repliche. Nella sezione successiva sono analizzati i risultati di tale simulazione. Si sottolinea come nel caso del container che rappresenta i client, sia esplicita la dipendenza di quest'ultimo dalle repliche, tramite la keyword *depends_on*, che garantisce l'istanziazione del container client solo a seguito di tutte le repliche.

```
services:

  server-0:
    build:
      context: .
      dockerfile: DockerfileServer
    command: ./server-replica 0
    container_name: server-0
```

Listing 2. Porzione di *docker-compose.yml* per istanziare una replica dello store.

```
client:
  build:
    context: .
    dockerfile: DockerfileClient
  command: ./client-test
  depends_on:
    - server-0
    - server-1
    - server-2
    - server-3
  container_name: client
```

Listing 3. Porzione di *docker-compose.yml* che istanzia il container per i client.

Istruzioni per lanciare il sistema localmente oppure su un'istanza AWS EC2 sono presenti nel repository su Github indicato in [3].

5 Test e risultati

Per verificare il corretto perseguimento della consistenza sequenziale o causale, sono eseguiti dei test che simulano diversi client, i quali in parallelo richiedono operazioni alle repliche dello store. Per ogni tipologia di consistenza da rispettare vengono eseguiti un test *simple* e un test *complex*, che testano il corretto funzionamento del multicast implementato manipolando in misura più o meno complessa lo store, dal punto di vista delle chiavi accedute, delle relazioni di causa-effetto generate o del numero di operazioni realizzate.

Ogni client esegue una sequenza di operazioni, in completa concorrenza rispetto agli altri. Per ottenere questa concorrenza, ogni client interagisce con una replica differente dello store. Inoltre, al fine di simulare un comportamento reale messo in atto dai client, l'intervallo tra un'operazione e la successiva è generato randomicamente, così come i ritardi nella propagazione dei messaggi in rete, come indicato in precedenza. In tal modo l'*interleaving* delle richieste da parte dei diversi client è completamente casuale, così da poter valutare se il risultato conseguito rispetta i vincoli della consistenza desiderata, senza avere alcun controllo sull'istante di esecuzione delle richieste, ma solo sull'odine con cui queste vengono propagate da ogni client alla replica con cui è associato. Poiché ogni test simula un'*interleaving* tra le richieste sempre diverso, data la casualità dei ritardi, tali test sono stati eseguiti molteplici volte per verificare l'effettiva robustezza del sistema realizzato. In questo documento sono riportati per ogni tipologia di test condotta un esempio di risultati ottenuti, con indicazioni delle operazioni richieste dai diversi client.

5.1 Test del multicast totalmente ordinato

5.1.1 Simple Test. Questo test verifica il corretto conseguimento della consistenza sequenziale nel caso di manipolazione di una singola chiave in parallelo da parte di molteplici client. Le operazioni realizzate sono mostrate in figura:

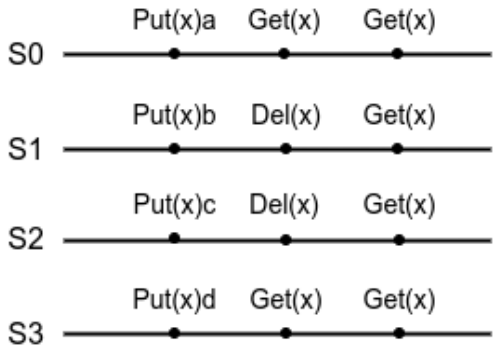

Figura 1. Test sequenziale semplice.

Di seguito sono mostrati i risultati ottenuti, ed in particolare l'ordine con cui le operazioni sono realizzate su ciascuna delle repliche. Per brevità vengono riportati gli output di due delle repliche, essendo i risultati delle altre analoghi:

```
PUT key x value b
PUT key x value d
PUT key x value a
PUT key x value c
DELETE key x
GET key x value NOT FOUND
DELETE key x
GET key x value NOT FOUND
Timeout reached, shutting down server...
The store is empty.
```

Listing 4. Output della Replica 0.

```
PUT key x value b
PUT key x value d
PUT key x value a
PUT key x value c
DELETE key x
DELETE key x
GET key x value NOT FOUND
Timeout reached, shutting down server...
The store is empty.
```

Listing 5. Output della Replica 1.

Come si evince analizzando gli output presentati, tutte le repliche realizzano la medesima sequenza delle operazioni di update, e tali operazioni sono realizzate su ciascuna replica rispettando gli ordini di programma indicati in Figura 1. Inoltre, essendo la rimozione di x l'ultima operazione realizzata, lo stato finale dello store è vuoto su tutte le repliche. Le operazioni di Get, essendo eventi interni, compaiono solo localmente alla replica verso cui sono dirette.

5.1.2 Complex Test. Questo test verifica il corretto conseguimento della consistenza sequenziale nel caso di client che, in concorrenza, manipolano molteplici chiavi. Le operazioni realizzate sono mostrate in figura:

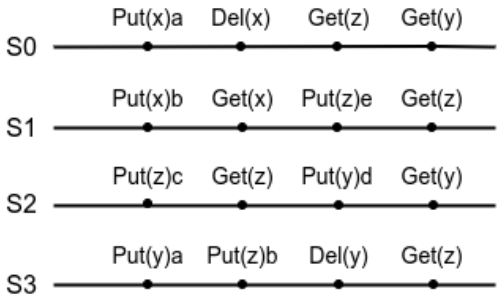

Figura 2. Test sequenziale complesso.

Sono riportati di seguito gli output ottenuti su due della quattro repliche considerate, per le restanti i risultati sono analoghi.

```
PUT key x value a
PUT key z value c
PUT key x value b
PUT key y value a
DELETE key x
PUT key z value b
PUT key z value e
PUT key y value d
GET key z value e
DELETE key y
GET key y value NOT FOUND
Timeout reached, shutting down server...
+-----+-------+
| Key | Value |
+-----+-------+
| z   | e     |
+-----+-------+
```

Listing 6. Output della Replica 0.

```
PUT key x value a
PUT key z value c
PUT key x value b
PUT key y value a
DELETE key x
GET key x value NOT FOUND
PUT key z value b
PUT key z value e
PUT key y value d
DELETE key y
GET key z value e
Timeout reached, shutting down server...
+-----+-------+
| Key | Value |
+-----+-------+
| z   | e     |
+-----+-------+
```

Listing 7. Output della Replica 1.

Si evidenzia come anche in questo caso le operazioni di update siano eseguite su tutte le repliche secondo un ordinamento sequenziale unico, che rispetta ciascun ordine di programma. Lo stato di ogni replica dello store risulta al termine della simulazione consistente, poiché la chiave z presenta i medesimi valori in ogni replica, e le chiavi x e y risultano non presenti in quanto rimosse.

5.2 Test del multicast causalmente ordinato

5.2.1 Simple Test. Questo test verifica il corretto conseguimento della consistenza causale, mediante l'esecuzione di una coppia di scritture in relazione di causa-effetto (mostrate con le frecce in figura e sottolineate), e diverse scritture concorrenti (cerchiate in figura).

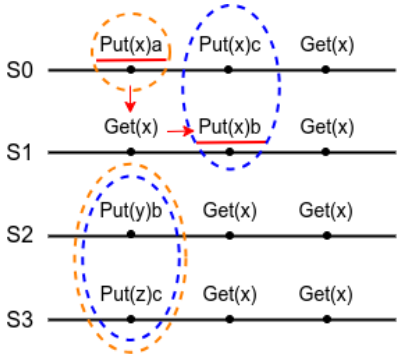

Figura 3. Test causale semplice.

Sono in relazione di causa-effetto le Get seguite da Put o Delete sulla stessa replica, e le Put o Delete di una chiave, seguite da Get della stessa chiave su repliche diverse (e ciò che consegue per proprietà transitiva). Si mostra quindi l'output relativo a due delle quattro repliche simulate, le restanti si comportano in maniera analoga.

```
PUT key x value a
PUT key z value c
PUT key y value b
PUT key x value c
PUT key x value b
GET key x value b
Timeout reached, shutting down server...
+-----+-------+
| Key | Value |
+-----+-------+
| x   | b     |
+-----+-------+
| z   | c     |
+-----+-------+
| y   | b     |
+-----+-------+
```

Listing 8. Output della Replica 0.

```
PUT key y value b
PUT key z value c
PUT key x value a
PUT key x value b
PUT key x value c
GET key x value c
GET key x value c
Timeout reached, shutting down server...
+-----+-------+
| Key | Value |
+-----+-------+
| y   | b     |
+-----+-------+
| z   | c     |
+-----+-------+
| x   | c     |
+-----+-------+
```

Listing 9. Output della Replica 2.

Le scritture in relazione causa-effetto sono eseguite nello stesso ordine da ogni replica. Le scritture concorrenti, come quelle evidenziate in blu in figura 3, o quelle evidenziate in arancione, sono invece eseguite in ordine diverso nelle diverse repliche. La consistenza causale risulta quindi rispettata.

5.2.2 Complex Test. In questo test vengono realizzate maggiori scritture in relazione di causa effetto, sia Put che Delete, e aumenta il livello di concorrenza tra i client.

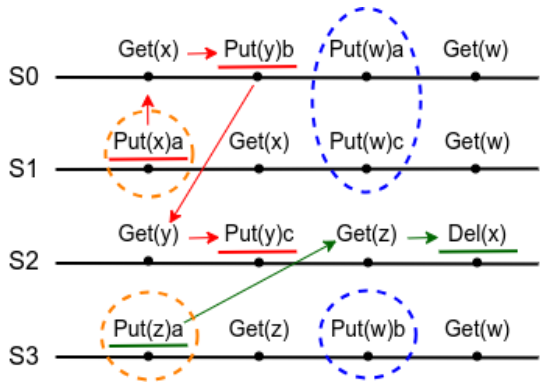

Figura 4. Test causale complesso.

Sono riportati gli output relativi all'esecuzione delle operazioni indicate, per due delle quattro repliche dello store.

```
PUT key x value a
PUT key z value a
GET key x value a
PUT key y value b
PUT key w value b
PUT key w value c
PUT key w value a
PUT key y value c
GET key w value a
DELETE key x
Timeout reached, shutting down server...
+-----+-------+
| Key | Value |
+-----+-------+
| z   | a     |
+-----+-------+
| y   | c     |
+-----+-------+
| w   | a     |
+-----+-------+
```

Listing 10. Output della Replica 1.

```
PUT key z value a
PUT key x value a
GET key z value a
PUT key y value b
PUT key w value b
GET key w value b
PUT key w value a
PUT key w value c
PUT key y value c
DELETE key x
Timeout reached, shutting down server...
+-----+-------+
| Key | Value |
+-----+-------+
| z   | a     |
+-----+-------+
| y   | c     |
+-----+-------+
| w   | c     |
+-----+-------+
```

Listing 11. Output della Replica 3.

Gli update in relazione causa-effetto risultano eseguiti su tutte le repliche nello stesso ordine, mentre le operazioni concorrenti, come quelle sulla chiave w, portano ad avere diverso stato dello store inerente a tale chiave, poiché sono eseguite in ordine diverso su ogni replica.

5.3 Conclusioni

I test dimostrano la corretta realizzazione dei meccanismi di multicast totalmente e causalmente ordinato, avendo ottenuto esecuzioni delle operazioni che rispettano di volta in volta la consistenza desiderata.

REFERENCES

[1] Marteen Van Sten and Andrew S. Tanenbaum, 2023. *Distributed Systems* (4th ed.).
[2] G. Coulouris, J. Dollimore, T. Kindberg, G. Blair, 2012. *Distributed Systems, Concepts and Design* (5th ed.).
[3] https://github.com/matteopallagrosi/dbService.