Implementazione di uno Store Chiave-Valore Distribuito con Garanzie di Consistenza Sequenziali e Causali

Filippo Muscherà
filippo.muschera@gmail.com
Università degli Studi di Roma "Tor Vergata"
Roma, Italia

Abstract

Lo scopo di questo documento è quello di descrivere l'architettura e l'implementazione del sistema di store chiave-valore distribuito che è stato realizzato. Sistemi di questo tipo sono fondamentali per la realizzazione di applicazioni su larga scala, in quanto un sistema centralizzato potrebbe sovraccaricarsi più facilmente, oltre a rappresentare un *single point of failure*. Il sistema realizzato offre garanzie sia per la consistenza sequenziale, attraverso un meccanismo di ordinamento globale delle operazioni, sia per quella causale, assicurando che tutte le dipendenze causa-effetto tra operazioni siano rispettate.

1 Introduzione

Il sistema di store chiave-valore supporta operazioni di *Put*, *Get* e *Delete*. Il numero di server è configurabile all'avvio, così come il tipo di consistenza desiderata per lo store distribuito.

La consistenza sequenziale garantisce che i processi in esecuzione concorrente vedano la stessa alternanza di operazioni, e che ogni client veda il suo ordine di programma rispettato. Quindi con la scelta della consistenza sequenziale, il sistema è realizzato in modo da trovare una sequenza per le operazioni che sia la stessa per tutte le N repliche di cui è composto, e che rispetti l'ordine in cui ogni client ha richiesto le proprie operazioni.

La consistenza causale invece rilassa questi vincoli, facendo perdere l'illusione che si abbia un'unica copia dello store chiave-valore. Per garantire questo tipo di consistenza, infatti, il sistema è realizzato in modo che tutte le operazioni con una (potenziale) relazione di causa-effetto siano viste da tutti i server replica nello stesso ordine. Operazioni di scrittura che invece sono concorrenti possono essere viste in ordine differente da processi diversi.

2 Design del Sistema

Si passa ora ad analizzare più nel dettaglio come la soluzione è stata realizzata.

2.1 Strumenti e Librerie

L'applicazione è stata realizzata con il linguaggio di programmazione Go. Per l'orchestrazione dei container si è invece utilizzato Docker Compose. Il progetto è stato poi eseguito anche all'interno di un'istanza AWS EC2.

Durante lo sviluppo si è ricorsi all'impiego della libreria esterna UUID [1] prodotta da Google. Questa libreria è stata utilizzata per poter produrre degli identificativi per i messaggi che fossero sempre unici (*UUID*: Universally Unique IDentifier), e che, essendo alfanumerici, potessero essere confrontati e ordinati.

2.2 Assunzioni

Per realizzare le garanzie di consistenza sequenziale e causale, sono stati implementati, rispettivamente, gli algoritmi del *Multicast Totalmente Ordinato* e del *Multicast Causalmente Ordinato*. Entrambi questi algoritmi si basano sulle seguenti assunzioni:

(1) Comunicazione affidabile: non si ha perdita dei messaggi.
(2) Comunicazione *FIFO ordered*: i messaggi che il generico processo p_i invia a p_j, sono ricevuti da quest'ultimo nello stesso ordine in cui p_i li ha inviati.

2.2.1 Implementazione delle Assunzioni. Queste assunzioni, non sono però necessariamente verificate in un ambiente distribuito, dove le comunicazioni tramite la rete possono subire perdite e ritardi.

Per rendere allora vere queste assunzioni sono state necessarie alcune accortezze durante lo sviluppo del sistema:

(1) Per assicurare una comunicazione affidabile si è fatto in modo che tutte le comunicazioni di rete utilizzassero come protocollo di rete per il livello di trasporto *TCP*, in grado di garantire l'affidabilità nella trasmissione dei pacchetti.
(2) Per garantire che tutti i pacchetti inviati arrivino in ordine, sia nelle comunicazioni tra client e server che tra i vari server, sono stati utilizzati numeri di sequenza. Ogni connessione, sia essa client-server o server-server, impiega numeri di sequenza per tenere traccia dei pacchetti in arrivo. In questo modo, i pacchetti possono essere ordinati correttamente, e quelli ricevuti fuori ordine vengono temporaneamente trattenuti fino a quando non è possibile processarli nella giusta sequenza. Per rendere possibile questo meccanismo si è assunto che, ogni client, durante la sua esecuzione, comunichi con un singolo server replica. Si ritiene questa assunzione realistica in quanto, in applicazioni distribuite globalmente, ogni client generalmente comunica con il server ad esso più vicino, poiché in grado di garantirgli la latenza minore.

2.3 Dettagli Implementativi

2.3.1 RPC e Delay. Tutte le comunicazioni tra i vari componenti del sistema sono state realizzate mediante l'utilizzo di *RPC* (*Remote Procedure Call*), offerte nativamente dalla libreria `"net/rpc"` di Go. Per simulare i ritardi di rete che si potrebbero verificare in reti congestionate o comunque per nodi lontani tra loro, si è implementata una funzione apposita. Questa funzione (`"NetworkDelay"`) è realizzata in modo da calcolare ogni volta che viene invocata un delay casuale compreso tra i 10 millisecondi e 1 secondo, e provvede ad eseguire una `Sleep` della durata calcolata. Questa funzione viene invocata ogni volta che c'è una comunicazione tra due nodi

prima dell'invocazione della `Call` vera e propria, così da simulare un ritardo di rete per ogni singolo pacchetto scambiato.

2.3.2 Richieste, Messaggi e Ack. Nel sistema realizzato si possono identificare tre diversi tipi di pacchetti:

- **Richiesta**: è un messaggio inviato dal client al server, in cui viene richiesta l'esecuzione di un'operazione di *Get*, *Put* o *Delete*.
- **Messaggio**: viene scambiato tra due server (o inviato da un server a sé stesso). Rappresenta il messaggio di update che i server si scambiano per informare le altre repliche delle operazioni che gli vengono richieste.
- **Acknowledgement**: gli *ack* vengono scambiati dai server, laddove necessario, per confermare la ricezione di un messaggio a tutti gli altri server.

2.3.3 Ricezione e Consegna. Questo sistema deve essere visto come composto da due livelli: quello applicativo, dove si può localizzare lo store key-value vero e proprio, e quello del middleware, dove vengono implementati tutti quei meccanismi volti al garantire la consistenza sequenziale o causale dello store stesso.

Per questo durante lo sviluppo del sistema è stato importante differenziare logicamente queste due fasi:

- Ricezione: si effettua a livello di middleware. Avviene quando una richiesta o un messaggio possono essere effettivamente processate rispettando l'ordinamento *FIFO* dei pacchetti, di cui si è discusso nella sezione delle assunzioni.
- Consegna: la esegue il middleware verso il livello applicativo. Quando un messaggio può essere effettivamente processato, allora avviene la consegna al livello applicativo, in cui lo store chiave-valore esegue l'operazione richiesta dal client.

3 Algoritmi

Si passa ora alla discussione degli algoritmi utilizzati per garantire i due tipi di consistenza e dei loro dettagli implementativi.

3.1 Multicast Totalmente Ordinato

Come già anticipato in precedenza, questo è stato l'algoritmo utilizzato per garantire che i messaggi inviati alle varie repliche siano ricevuti e processati in un ordine globale coerente, indipendentemente dall'ordine di arrivo sui singoli nodi. Per realizzarlo si è fatto uso di un **clock logico scalare** per ogni server.

Alla ricezione di una richiesta da parte di un server, quando questa può essere effettivamente ricevuta (va rispettato, come detto, l'ordinamento *FIFO*), viene generato un messaggio, a cui viene assegnato un *UUID* e il valore del clock logico scalare dopo il suo incremento.

A questo punto il messaggio prodotto dal server che ha ricevuto la richiesta deve essere inviato a tutti i server (compreso sé stesso). Se il messaggio è relativo a una *Get* (quindi una lettura, e dunque assimilabile a un evento interno), verrà effettivamente ricevuto solo dal server che l'ha generato, mentre gli altri lo ignoreranno. Si sarebbe potuto evitare di inviare il messaggio di *Get* in multicast, ma è stata adottata questa soluzione per favorire il riutilizzo del codice del sistema. In ogni caso, logicamente, i messaggi relativi alle letture vengono comunque gestiti da un solo server.

Se invece il messaggio è relativo a un'operazione di *Put* o *Delete* (quindi un'operazione di scrittura), il messaggio viene realmente ricevuto da tutti i server. Se il messaggio in questione rispetta l'ordinamento *FIFO*, viene allora inserito in una coda di messaggi che viene mantenuta ordinata secondo un ordinamento totale. Vengono prima confrontati i valori del clock logico scalare associati a ogni messaggio. Se due messaggi presentano lo stesso time-stamp, questi vengono ordinati secondo il loro *UUID* (che, per definizione, non potrà mai ripetersi, e dunque sarà sempre possibile ordinare due messaggi).

Effettuata la ricezione del messaggio, il valore del clock logico scalare ad esso associato viene confrontato con quello del clock logico scalare del server. Il valore del clock del server viene impostato al massimo valore tra i due.

Effettuata la ricezione del messaggio, il server invia in multicast i relativi *ack*.

A questo punto il messaggio entra in una fase di attesa. Infatti vengono periodicamente controllate le condizioni che, quando verificate, consento al messaggio di essere consegnato al livello applicativo. Nello specifico, prima di poter essere consegnato:

- Il messaggio deve aver ricevuto un numero di *ack* pari al numero di repliche del sistema.
- Per ogni processo deve essere presente, nella coda di messaggi, un messaggio con clock logico scalare maggiore di quello del messaggio corrente.
- Il messaggio deve essere il primo nella coda.

Queste condizioni garantiscono che il messaggio sia stato ricevuto da tutti i server, che sia il primo nella coda e che non possa esserci un messaggio che lo precederebbe in arrivo da un altro server.

A questo punto, allora, il messaggio può essere consegnato al livello applicativo, e poi rimosso dalla coda.

Se è un messaggio relativo a un'operazione di *Get*, il client che ha richiesto l'operazione riceverà la risposta relativa. Se invece si tratta di un'operazione di scrittura (*Put* o *Delete*) riceverà solo un valore di ritorno relativo all'errore (`nil` nel caso in cui tutto sia andato a buon fine).
Dovendo rispettare sempre tutte queste condizioni, ci si assicura che tutte le repliche del sistema processino i messaggi nello stesso ordine.

3.1.1 Criticità nell'Implementazione dell'Algoritmo.

Ack per messaggi non ancora ricevuti. Si è dovuto considerare il caso in cui una replica ricevesse un *ack* per un messaggio che ancora non era stato ricevuto. Questo scenario è possibile in quanto ogni pacchetto sperimenta un *delay* casuale.

In questo caso la soluzione adottata è stata la seguente: in fase di ricezione dell'*ack* si scorre la coda di messaggi. Se si trova un messaggio il cui *UUID* corrisponde a quello che si trova nell'*ack*, allora si procede a incrementare il contatore degli *ack* ricevuti per quel messaggio. Se invece nessun messaggio presente nella coda corrisponde, si genera un nuovo messaggio e lo si inserisce nella coda. Questo messaggio ovviamente avrà il contatore degli *ack* settato a `1`, e non a `0`.

Quando questo messaggio verrà effettivamente ricevuto dal server, la funzione per l'inserimento dei messaggi in coda, andrà dunque a controllare se il messaggio è già presente oppure no, per

evitare che si generino dei duplicati.

End Operations. A causa della natura di questo algoritmo, che richiede di attendere un messaggio con un clock logico scalare maggiore da ciascun processo prima di procedere alla consegna a livello applicativo, è stato necessario introdurre operazioni specifiche, denominate *End Operations* (o `EndOps` nel codice).

Queste particolari operazioni sono state inserite nei test effettuati per la consistenza sequenziale, per fare in modo che tutte le operazioni richieste dai client venissero effettivamente eseguite. Senza di queste, infatti, le ultime operazioni di ogni processo non potrebbero essere consegnate a livello applicativo, proprio per la natura stessa dell'algoritmo del Multicast Totalmente Ordinato. Nei test sequenziali, quindi, è stata inserita una `EndOp` in coda alle operazioni richieste da ogni client al relativo server. Queste operazioni vengono inserite nella coda dei messaggi delle repliche, ma non richiedono l'esecuzione di una operazione di lettura o scrittura, e sono infatti identificate da una speciale coppia chiave-valore. Si è poi realizzata un'apposita *goroutine* che periodicamente (ogni 3 secondi, in modo da non pesare troppo sulla *CPU*) controlla se nella coda di messaggi sono rimaste solo ed esclusivamente `EndOps`. In questo caso allora provvede a svuotare la coda del server e a stampare sulla console del server stesso il contenuto dello store chiave-valore, dato che questa condizione rappresenta la fine del test.

3.2 Multicast Causalmente Ordinato

Il multicast causalmente ordinato rappresenta un rilassamento delle condizioni rispetto a quanto richiesto dall'algoritmo del multicast totalmente ordinato. In questo caso, infatti, un messaggio viene consegnato a livello applicativo solo se tutti i messaggi che lo precedono **causalmente** sono già stati consegnati. Questo significa che si perde l'illusione di avere un'unica replica del sistema (come invece succedeva con il precedente algoritmo), perché repliche diverse possono vedere operazioni concorrenti in ordine differente, senza che questo rappresenti una violazione della consistenza causale.

Per implementare questo algoritmo si è fatto uso di un **clock logico vettoriale** V_j per ogni processo p_j. La componente $i-sima$ del clock logico vettoriale $V_j[i]$ conta il numero di messaggi inviati dal processo p_i al processo p_j che p_j ha già osservato.

Avendo ora un clock non più scalare ma vettoriale, cambiano anche le modalità con cui questo viene aggiornato:

- In fase di invio di un messaggio da parte del processo p_j la componente $V_j[j]$ viene incrementata.
- In fase di ricezione, quando si riceve un messaggio dal processo p_i (con $i \neq j$), il processo p_j incrementa la componente $V_j[i]$ del suo clock logico vettoriale.

Nota: nel caso di un messaggio inviato da p_j e ricevuto da p_j stesso, il clock viene incrementato in fase di invio, ma non in fase di ricezione (da qui la condizione $i \neq j$), poiché altrimenti si finirebbe per incrementare due volte il clock per uno stesso messaggio.

Con questo secondo algoritmo, quindi, le condizioni per cui è possibile consegnare un messaggio a livello applicativo cambiano. Confrontando il time-stamp del messaggio m (ts_m) con il clock logico vettoriale del processo j (V_j), sarà possibile eseguire la consegna solo quando:

(1) $ts_m[i] = V_j[i] + 1$, ovvero quando il messaggio ricevuto da p_i è il successivo che p_j si aspetta.
(2) $ts_m[k] \leq V_j[k], \forall k \neq i$, ovvero se per ogni altro processo p_k, p_j ha visto almeno gli stessi messaggi visti da p_i.

Nota: la seconda condizione ha il vincolo $k \neq i$ perché altrimenti non sarebbe mai verificata: infatti essendo questo un nuovo messaggio da p_i a p_j, è ovvio che p_j non possa ancora aver visto un numero di messaggi maggiore o uguale a p_i per quanto riguarda la componente $V_j[i]$.

3.2.1 Criticità nell'Implementazione dell'Algoritmo.

Relazione Cause-Effetto Read-Write. Uno dei principali aspetti critici nell'implementazione dell'algoritmo per garantire la consistenza causale è stato assicurarsi che i messaggi fossero sempre consegnati al livello applicativo nel giusto ordine: prima la causa, poi l'effetto. Per questo motivo, sono stati sviluppati meccanismi aggiuntivi che garantissero che, se un processo scrive una variabile e un altro la legge, l'operazione di scrittura fosse sempre eseguita prima di quella di lettura a livello applicativo. Questo accorgimento si basa sul presupposto che la lettura rappresenti potenzialmente l'effetto della scrittura, che ne costituisce quindi la causa.

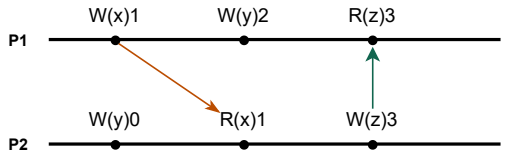

Figura 1: Esempi di relazioni causa-effetto tra letture e scritture in processi diversi.

Durante lo sviluppo, allora, è stata definita un'ulteriore condizione che una *Get* deve rispettare prima di poter essere consegnata al livello applicativo, ovvero che la relativa chiave fosse presente all'interno dello store chiave-valore.

Relazione Causa-Effetto Write-Delete. In maniera analoga, configurando opportunamente la variabile d'ambiente `DELETE_CAUSAL` come definito nella documentazione [2], verranno considerate in relazione di causa effetto anche le operazioni di scrittura ed eliminazione effettuate da processi diversi sulla stessa variabile. In questo caso, allora, una *Delete* si comporterà in maniera simile a una *Get*: prima di poter essere consegnata a livello applicativo, bisognerà assicurarsi che esista la chiave richiesta nello store.

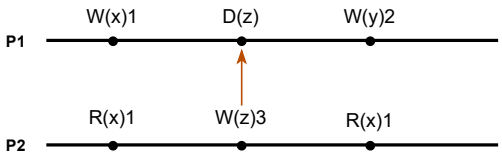

Figura 2: Esempio di potenziale relazione causa-effetto tra scrittura ed eliminazione della stessa chiave in processi diversi.

Configurando il sistema in questo modo, si presuppone che un'operazione di *Delete* venga eseguita in seguito a una precedente operazione di *Put*, analogamente a quanto avviene per le operazioni di lettura. In questa prospettiva, la scrittura della variabile diventa la causa, mentre la cancellazione rappresenta l'effetto. Il sistema, pertanto, si occupa di mantenere l'ordine causa-effetto tra queste operazioni su tutte le repliche.

4 Confronto

È interessante confrontare, almeno ad alto livello, i costi dei due algoritmi.

Il Multicast Totalmente Ordinato, che garantisce una consistenza sequenziale su tutte le repliche dello store chiave-valore, richiede un uso molto più intensivo di *mutex* e un maggior numero di messaggi scambiati.

In entrambe le soluzioni, si è cercato di sfruttare al massimo il meccanismo delle *goroutine* offerto da *Go*, gestendo ciascuna richiesta, messaggio o *ack* con una goroutine differente su ogni server. Questa scelta è stata fatta anche sulla base di quanto riportato nella documentazione di Go [3], dove si afferma che *"It is practical to create hundreds of thousands of goroutines in the same address space"*.

La presenza di molteplici flussi di esecuzione concorrenti ha reso necessario un rigoroso controllo sull'accesso a tutte le strutture dati condivise, per prevenire *race conditions* o aggiornamenti concorrenti dei dati. Di conseguenza, è stato necessario utilizzare un numero elevato di *mutex*, il che potrebbe portare a un degrado delle prestazioni quando un gran numero di *goroutine* si contende la stessa risorsa.

Inoltre, per garantire la consistenza sequenziale, è necessario l'impiego di *ack* inviati in multicast. Questo significa che, per ogni messaggio, si generano un numero di *ack* pari al numero di repliche dello store chiave-valore. Questo può sia provocare congestionamenti della rete, sia rallentare la consegna di un messaggio a livello applicativo, qualora un *ack* subisse un forte rallentamento di rete.

L'utilizzo del Multicast Causalmente Ordinato invece offre prestazioni potenzialmente migliori: utilizza meno strutture dati condivise, di conseguenza meno *mutex*, e inoltre non fa uso di *ack*, riducendo drasticamente il numero di pacchetti scambiati tra i vari server replica. È chiaro, però, che questo miglioramento di prestazioni si ottiene a discapito del livello di garanzie di consistenza offerte dal sistema. Rilassando i vincoli e passando dalla consistenza sequenziale a quella causale, infatti, si perde l'illusione di avere un unica copia dello store chiave-valore.

Si tratta dunque di un *trade-off* tra prestazioni e livello delle garanzie di consistenza offerte.

5 Deployment

Per quanto riguarda la fase di deployment del sistema, si sono andati a creare due *Dockerfile* distinti (uno per il server e uno per il client), e un file `.yml` per l'utilizzo con *Docker Compose*.

5.1 Dockerfile

I due *Dockerfile* sono estremamente simili dal punto di vista logico, e possono essere suddivisi in due fasi: quella di *build* e quella di *run*.

Per la prima fase si fa uso di un'immagine builder che contiene tutti gli strumenti necessari per compilare il progetto tramite i comandi offerti da *Go*:

```
# Fase di build
FROM golang:latest AS builder

WORKDIR /app

# Copia dei file di modulo e download delle dipendenze
COPY go.mod go.sum ./
RUN go mod download

# Copia del resto del codice sorgente
COPY . .

# Compilare il client
RUN go build -o client ./main/client
```

Listing 1: Frammento della fase di build del dockerfile relativo al client

Per la fase di *run* non si vuole utilizzare la stessa immagine, dato che sarebbe appesantita dalla presenza di tutti i tools utilizzati per la compilazione. A tale scopo si sceglie una nuova immagine per eseguire il sistema, vi si copia il file compilato e si definisce il comando standard per l'esecuzione:

```
# Fase di runtime
FROM fedora:latest

WORKDIR /app

# Copia del binario compilato dalla fase di build
COPY --from=builder /app/server /app/

# Comando predefinito
CMD ["./server"]
```

Listing 2: Frammento della fase di run del dockerfile relativo al server float

Nota: analizzando le dimensioni delle due immagini una volta estratte, tramite i comandi
`docker inspect -f "{{ .Size }}" golang:latest` *e*
`docker inspect -f "{{ .Size }}" fedora:latest`
si può vedere come la prima abbia un peso di circa 837 MB, contro i 222 MB della seconda, che risulta quindi molto più leggera.

5.2 Docker Compose

Per quanto riguarda invece il deployment tramite *Docker Compose*, si provvedere a lanciare un container per ogni server e (per semplicità) un unico container in cui eseguiranno i client. Tutti i container fanno uso del file `.env` per la definizione delle variabili

del loro ambiente. In questo modo una modifica a questo file applica le modifiche a tutti i container.

Tutti i container sono collegati a una rete definita con il nome di `app-network`:

```
networks:
  app-network:
    driver: bridge
```

Listing 3: Definizione della rete per i container

Inoltre, per i client, è definito anche il parametro `depends_on`, per fare in modo che quel container venga avviato solo dopo che tutti i server sono già attivi.

5.3 EC2

Il progetto può essere eseguito con *Docker Compose* all'interno di un'istanza *AWS EC2* eseguendo il *clone* della repository, installando *Docker* e *Docker Compose* e avviando il sistema con il comando `docker-compose up --build`.

Le istruzioni dettagliate sono disponibili nella documentazione presente nella repository GitHub [2] del progetto.

Il progetto è stato testato su varie taglie di istanze *EC2*, e anche quella con hardware più limitato (`t2.nano`, con 1 vCPU e 0.5 GiB di memoria) è in grado di eseguire il progetto tramite *Docker Compose*. Chiaramente, con istanze più potenti, la fase di build richiede meno tempo.

6 Testing

Infine, vengono analizzati i test effettuati per verificare il corretto funzionamento del sistema con entrambe le garanzie di consistenza: sequenziale e causale. In tutti i test vengono considerate $N = 3$ repliche, e per semplicità si fa in modo che ogni client parli con il server corrispettivo.

6.1 Testing Consistenza Sequenziale

Per avere uno svolgimento dei test sequenziali in cui tutte le operazioni svolte dai client vengano effettivamente eseguite, sono state inserite anche le *End Operations* discusse in precedenza.

Test Base. Si comincia con un semplice test mostrato in figura:

```
+-----------+-----------+-----------+-----------+-----------+
| Operazione|     1     |     2     |     3     |     4     |
+-----------+-----------+-----------+-----------+-----------+
| Processo 0| put x:1   | get x     | del x     | get x     |
+-----------+-----------+-----------+-----------+-----------+
| Processo 1| put x:2   | get x     | del x     | get x     |
+-----------+-----------+-----------+-----------+-----------+
| Processo 2| put x:3   | get x     | del x     | get x     |
+-----------+-----------+-----------+-----------+-----------+
[CLIENT 0] Connecting to server localhost:8080
[CLIENT 2] Connecting to server localhost:8082
[CLIENT 1] Connecting to server localhost:8081
[CLIENT 0] Answer from server: GET of Key 'x' Value = 1
[CLIENT 1] Answer from server: GET of Key 'x' Value = 1
[CLIENT 2] Answer from server: GET of Key 'x' Value = 1
[CLIENT 0] Answer from server: GET of Key 'x' Value = KeyNotFound
[CLIENT 1] Answer from server: GET of Key 'x' Value = KeyNotFound
[CLIENT 0] Done
[CLIENT 1] Done
[CLIENT 2] Answer from server: GET of Key 'x' Value = KeyNotFound
[CLIENT 2] Done
All operations have completed.
```

Figura 3: Test Sequenziale di Base

Si può notare come tutte le repliche vedano come ultima scrittura di x il valore 1, e come poi x venga eliminata correttamente da tutte le repliche.

Test Avanzato. Con questo test si vuole dimostrare che tutte le repliche dello store *key-value* concordino su un'unica sequenza per l'ordine di esecuzione di tutte le operazioni, andando ad aumentare il numero di variabili scritte e lette:

```
+-----------+-----------+-----------+-----------+-----------+
| Operazione|     1     |     2     |     3     |     4     |
+-----------+-----------+-----------+-----------+-----------+
| Processo 0| put x:1   | get y     | get x     | get z     |
+-----------+-----------+-----------+-----------+-----------+
| Processo 1| put y:2   | get x     | get z     | put z:5   |
+-----------+-----------+-----------+-----------+-----------+
| Processo 2| get x     | put x:3   | put z:4   | del x     |
+-----------+-----------+-----------+-----------+-----------+
[CLIENT 2] Connecting to server localhost:8082
[CLIENT 0] Connecting to server localhost:8080
[CLIENT 1] Connecting to server localhost:8081
[CLIENT 0] Answer from server: GET of Key 'y' Value = 2
[CLIENT 0] Answer from server: GET of Key 'x' Value = 3
[CLIENT 0] Answer from server: GET of Key 'z' Value = 5
[CLIENT 1] Answer from server: GET of Key 'x' Value = 1
[CLIENT 2] Answer from server: GET of Key 'x' Value = KeyNotFound
[CLIENT 0] Done
[CLIENT 1] Answer from server: GET of Key 'z' Value = KeyNotFound
[CLIENT 2] Done
[CLIENT 1] Done
All operations have completed.
```

Figura 4: Test Sequenziale Avanzato

Si può notare come in questa esecuzione del test, tutte le repliche concordino sulla sequenza di operazioni:

$$R_2(x)\varnothing,\ W_1(y)2,\ W_0(x)1,\ R_0(y)2,\ R_1(x)1,\ W_2(x)3,\ R_0(x)3,\ R_1(z)\varnothing,\ W_2(z)4,\ D_2(x),\ W_1(z)5,\ R_0(z)5$$

confermando il fatto che il sistema stia garantendo consistenza sequenziale.

Analizzando poi lo store chiave-valore dei server dopo l'esecuzione del test, tutti riportano la tabella:

```
---------------
| Key | Value |
---------------
| y   | 2     |
| z   | 5     |
---------------
```

6.2 Testing Consistenza Causale

Durante l'esecuzione dei test per la consistenza causale sono state considerate in potenziale relazione causa-effetto anche le operazioni di write e delete in processi diversi della stessa variabile (ponendo la variabile d'ambiente `DELETE_CAUSAL=1`).

Test Base. Il primo test effettuato è quello riassunto nella figura sottostante:

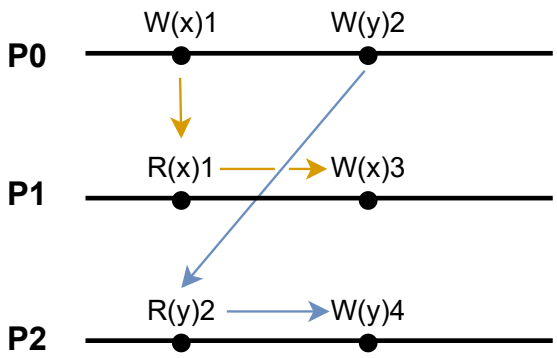

Figura 5: Schema delle Operazioni del Test Causale di Base

Andando ad analizzare l'output dei client vediamo come tutte le operazioni in relazione di causa-effetto vengano effettivamente eseguite nell'ordine corretto:

```
[CLIENT 2] Connecting to server localhost:8082
[CLIENT 0] Connecting to server localhost:8080
[CLIENT 1] Connecting to server localhost:8081
[CLIENT 1] Answer from server: GET of Key 'x' Value = 1
[CLIENT 1] Done
[CLIENT 0] Done
[CLIENT 2] Answer from server: GET of Key 'y' Value = 2
[CLIENT 2] Done
All operations have completed.
```

Figura 6: Output Test Causale di Base

Ad ulteriore conferma abbiamo il fatto che al termine del test, tutte le repliche concordano sullo stato dello store:

```
---------------
| Key | Value |
---------------
| x   | 3     |
| y   | 4     |
---------------
```

vedendo dunque nell'ordine corretto le operazioni.

Test Avanzato. Le operazioni che si vanno a svolgere in questo test più avanzato sono quelle riportate nel seguente schema:

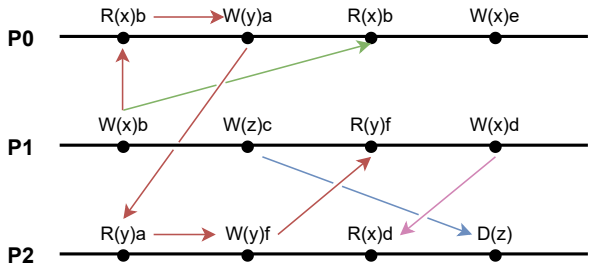

Figura 7: Schema delle Operazioni del Test Causale Avanzato

L'output di questo test è il seguente:

```
[CLIENT 2] Connecting to server localhost:8082
[CLIENT 0] Connecting to server localhost:8080
[CLIENT 1] Connecting to server localhost:8081
[CLIENT 0] Answer from server: GET of Key 'x' Value = b
[CLIENT 0] Answer from server: GET of Key 'x' Value = b
[CLIENT 2] Answer from server: GET of Key 'y' Value = a
[CLIENT 1] Answer from server: GET of Key 'y' Value = f
[CLIENT 0] Done
[CLIENT 1] Done
[CLIENT 2] Answer from server: GET of Key 'x' Value = d
[CLIENT 2] Done
All operations have completed.
```

Figura 8: Output Test Causale Avanzato

Andandolo ad analizzare notiamo come lo schema di dipendenze illustrato nella Figura 7 sia effettivamente rispettato, garantendo che tutte le operazioni in relazione causa-effetto siano viste nello stesso ordine da tutte le repliche. In questo caso è più interessante andare ad analizzare lo stato dello store chiave-valore al termine dell'esecuzione:

```
server0_1  | ---------------
server0_1  | | Key | Value |
server0_1  | ---------------
server0_1  | | x   | d     |
server0_1  | | y   | f     |
server0_1  | ---------------
server2_1  | ---------------
server2_1  | | Key | Value |
server2_1  | ---------------
server2_1  | | x   | d     |
server2_1  | | y   | f     |
server2_1  | ---------------
server1_1  | ---------------
server1_1  | | Key | Value |
server1_1  | ---------------
server1_1  | | x   | e     |
server1_1  | | y   | f     |
server1_1  | ---------------
```

Figura 9: Stato dello store chiave-valore sulle repliche a seguito del Test Causale Avanzato

si può notare infatti come la replica numero 1 presenti un valore di x diverso da quanto riportato dalle repliche 0 e 2. Questo è uno scenario possibile in alcune esecuzioni proprio perché, come si evince osservando la Figura 7, $W_0(x)e$ e $W_1(x)d$ sono operazioni di *write* **concorrenti**. Proprio per come è definita la consistenza causale, è allora possibile che vengano viste in ordine diverso da repliche diverse.

6.3 Conclusioni

L'analisi di questi test dimostra come gli algoritmi implementati consentano effettivamente di offrire un sistema di archiviazione chiave-valore distribuito su N repliche. L'applicazione si è mostrata in grado di offrire garanzie di consistenza sia sequenziale che causale, così come richiesto dalle specifiche.

Riferimenti

[1] Google. Uuid package. https://pkg.go.dev/github.com/google/uuid.
[2] Filippo Muscherà. Repository github. https://github.com/FilippoMuschera/SDCC.
[3] Go Doc. Faq. https://go.dev/doc/faq#goroutines.