Sistema di storage distribuito, basato sul paradigma dell'edge computing

Progetto per il corso di Sistemi distribuiti e Cloud computing

Gian Marco Falcone
matr. 0300251

Pierciro Caliandro
matr. 0299815

***Sommario*—Il seguente documento riporta l'analisi e le scelte progettuali effettuate per svolgere la traccia di progetto 1A assegnata durante l'AA 2020/2021 per il corso di Sistemi Distribuiti e Cloud Computing, presso l'università di Roma Tor Vergata. Il template utilizzato per la realizzazione del documento è scaricabile al link [1]**

I. Introduzione

A. Specifiche di progetto

Il progetto svolto riguarda l'implementazione di un sistema di storage distribuito, di tipo chiave-valore, che si basi sul paradigma dell'edge computing: un'architettura distribuita di tale tipo prevede la presenza di nodi ai bordi della rete, quindi vicini agli utenti che utilizzano l'applicazione e caratterizzati da una bassa latenza di rete. Tali nodi possono essere di tipo eterogeneo, come

- sensori
- dispositivi IoT
- micro data-center

e quindi caratterizzati da una scarsa capacità di memorizzazione. Per questo motivo, vengono affiancati a servizi Cloud più moderni.
Le API offerte, mediante RPC, dai nodi edge sono le seguenti:

- `Put(key, value)`: permette di salvare un valore associato ad una chiave su uno dei nodi edge;
- `Get(key)`: permette di recuperare il valore associato ad una chiave;
- `Append(key, value)`: consente di appendere un valore ad una chiave già esistente;
- `Del(key)`: permette di cancellare una chiave da un nodo, inclusi i valori associati ad essa.

I client hanno quindi la possibilità di effettuare tali operazioni sul nodo edge al quale sono collegati. A tal proposito sarebbe stato possibile sviluppare il front-end dell'applicazione con diversi tipi di tecnologie. Nel caso di questo progetto, considerando l'eventuale limitata capacità computazionale dei possibili client del sistema, si è scelto di sviluppare un front-end basato su Command Line Interface.
Maggiore attenzione è invece stata posta sullo sviluppo dei nodi edge e sulla loro integrazione con i servizi Cloud. Il sistema è stato sviluppato mediante l'utilizzo del linguaggio Golang.

B. Requisiti non funzionali dell'applicazione

L'applicazione ha diversi requisiti non funzionali, che vengono riassunti in seguito:

- **Essere scalabile rispetto al numero di utenti connessi**: a tal scopo è stato introdotto un meccanismo di replicazione dello stesso dato su molteplici nodi edge all'interno del sistema. Inoltre, il sistema deve garantire una **semantica degli errori almeno di tipo at least once** ed avere **consistenza almeno di tipo finale**;
- **essere distribuito su molteplici nodi**, i quali vengono scoperti durante l'avvio del sistema di storage attraverso un apposito **servizio di registrazione** e possono essere soggetti a guasti durante l'operatività del sistema;
- **supportare l'integrazione con un servizio di storage Cloud**, con lo scopo di aumentare la scalabilità rispetto ai dati andando a memorizzare i valori di grande dimensione o scarsamente acceduti.

II. Architettura del sistema

A. Componenti principali del sistema

Una rappresentazione grafica dell'architettura del sistema di storage viene mostrata nella seguente figura ([23]):

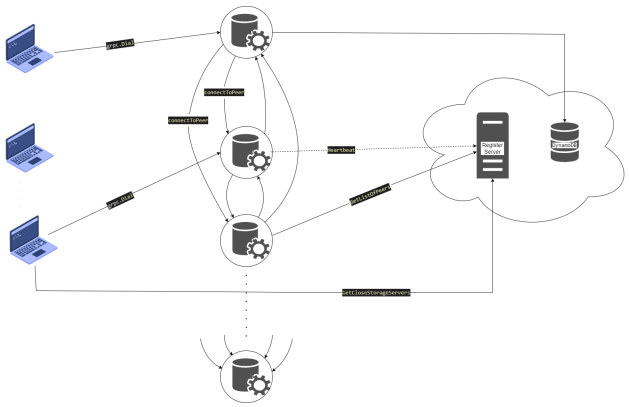

Figura 1. Architettura del sistema distribuito.

Nell'architettura in questione viene messo in evidenza come sia i client che i nodi edge sfruttino il servizio di registrazione:

- i nodi edge si registrano allo startup del sistema e, periodicamente, inviano al register un messaggio di "heart-

beat", così che quest'ultimo possa rendersi conto di eventuali guasti che essi possano subire;
- i client utilizzano tale nodo per poter ottenere la lista dei nodi edge attivi, in modo da connettersi a quello con la latenza di rete minore.

Inoltre, vi è l'integrazione col sistema di storage Cloud **DynamoDB**, al fine di poter garantire maggiore scalabilità rispetto ai dati, facendo sì che i nodi edge salvino dati pesanti in termini di occupazione di memoria oppure dati scarsamente acceduti dai client.
Si noti che ogni nodo edge ha la possibilità di comunicare con qualunque altro nodo nel sistema.

B. Progettazione e sviluppo dei nodi edge

Le RPC rese disponibili dal server ed i messaggi scambiati mediante esse sono stati definiti all'interno del file `pb_storage/Storage.proto`.
I messaggi supportati dall'applicazione sono due

- `StorageRecordMessage`: questo messaggio viene usato per impacchettare tutte le informazioni relative ad un record di tipo chiave:valore ed è usato sia in input che in output;
- `OpStatus`: messaggio usato per il ritorno dalla RPC, contenente un codice numerico che identifica se l'operazione è andata a buon fine o meno. Nel caso in cui ci sia stato un fallimento, l'operazione viene ritrasmessa per garantire semantica di tipo at least once.

1) Consistenza sul singolo nodo edge: Il singolo nodo edge mantiene le coppie chiave-valore all'interno di un'array. Poiché diversi client possono chiamare concorrentemente le operazioni di RPC, ossia è possibile che diversi thread andranno ad insistere sulla stessa struttura di dati, è stato necessario introdurre delle primitive di sincronizzazione per la lettura e scrittura dell'array.
La scelta è stata quella di usare i `RWMutex`, ovvero dei mutex per la lettura e scrittura esclusiva: tali mutex, permettono di avere, in uno stesso istante di tempo, diversi lettori su una struttura dati, ma un singolo scrittore. In questo modo, quindi, vengono favorite le operazioni di `Get`, in quanto al singolo thread occorre prendere un lock in lettura dal mutex per poter effettuare l'operazione, senza dover aspettare che altri eventuali thread completino la loro operazione di lettura. Per l'operazione di scrittura, invece, è necessario, per il thread che intende prendere il corrispettivo lock, aspettare che tutti gli altri abbiano terminato l'operazione che stavano eseguendo e rilasciato il token, in quanto egli sarà l'unico a poter operare sulla struttura dati in quel determinato istante di tempo.
Viene in seguito mostrato un esempio di utilizzo dei mutex, per prendere un lock in lettura oppure in scrittura:

```
s.lock.RLock()
for _, elem = range s.savedValues {
        if structs.GetKey(&elem) =
        = putRecord.GetKey() {
                key_present = true
                break
        }
}
s.lock.RUnlock()
...

s.lock.Lock() // lock before the insertion
s.savedValues[indexElem] = newRecord
s.lock.Unlock() // unlock after the insertion
```

In particolare, vengono utilizzate due API

- `RLock/RUnlock` per ottenere/rilasciare un token in lettura;
- `Lock/Unlock` per ottenere/rilasciare token in scrittura. Un solo processo per volta può scrivere sulla struttura dati.

C. Progettazione e sviluppo dei nodi client

Per i nodi client è stata sviluppata sia la parte di front-end che la parte di back-end.
Per quanto riguarda il back-end, è stata realizzata una apposita API, fornita dal package `client_utils`, nel file `client_utils.go` che contiene due funzioni fondamentali:

- l'API `AtLestOnceRetx`, che implementa una consegna di messaggi mediante RPC per cui sia garantita una semantica degli errori di tipo **at least once**. Se l'operazione ha un valore di ritorno che indica che l'RPC non è stata completata con successo, allora si ritrasmette il messaggio affinché l'operazione abbia successo almeno una volta. In particolare, poiché l'operazione potrebbe fallire a seguito di un sovraccarico di richieste sul sistema, è stato implementato un meccanismo di backoff esponenziale per permettere al sistema di smaltire eventuali richieste pendenti, prima di riprovare la trasmissione della stessa. Tale API è un wrapper per la funzione `ServeDecision`;
- la funzione `ServeDecision` è il cuore del back-end lato client, in quanto si occupa dell'effettiva chiamata alle RPC esposte dal server.
  Tale API viene utilizzata sia lato client che lato server, andando a discriminare quale dei due nodi l'ha invocata mediante controlli sul valore della chiave passata in input ed su un parametro booleano `auto`.
  I client utilizzano tale funzione per effettuare le operazioni desiderate sul sistema, mentre i server ne fanno uso al fine di comunicare eventuali aggiornamenti sul valore delle chiavi presenti, in modo da garantire la consistenza sui dati e sullo stato globale dell'architettura.

III. Primo requisito non funzionale

A. Replicazione dei dati

Per ottenere scalabilità rispetto al numero di utenti è stato necessario replicare i dati su più di un server, in modo che diversi client che si collegassero su diversi server, avessero la possibilità di trovare una particolare coppia chiave:valore

comunque disponibile. La chiave potrebbe essere stata precedentemente caricata da un altro client e potenzialmente su un altro nodo server rispetto a quello su cui avviene la richiesta di `Get`.
Per ottenere questo obiettivo, il singolo nodo edge è stato progettato in modo da riceve, all'atto della connessione col nodo di registrazione, una lista contenente indirizzo IP e porta di ascolto degli altri nodi edge attivi nel sistema.
In questo modo ogni server può, nel momento della ricezione di una chiamata RPC per le operazioni di `Put`, `Append` o `Del` propagare il messaggio ricevuto agli altri edge: il server si comporta quindi come un client e, mediante l'API `AtLeastOnceRetx` (alla quale vengono passati determinati parametri per indicare che l'operazione viene svolta da un nodo server, attraverso la funzione `connectToPeer`), invoca una RPC verso i nodi edge a cui è connesso. In questo modo, è possibile riutilizzare al meglio il codice, evitando di dover re-implementare una logica che sarebbe stata molto simile a quella sviluppata per il client.
In particolare, al fine di ottenere un **modello di consistenza finale** per il sistema, ogni chiave ha associati, tra i suoi metadati, l'indirizzo IP e la porta del server che per tale chiave svolge il ruolo di `owner`.
Il compito dell'`owner` è quello di propagare eventuali aggiornamenti per i valori di una chiave a gli altri nodi edge attivi nel sistema.
Ogni server può essere `owner` di più chiavi ma ogni chiave ha il suo unico `owner`. Tale metadato viene impostato al valore del server che per primo ha ricevuto, da un client a lui collegato, un'operazione di `Put` per quella chiave ed è importante che il suo valore sia coerente tra le varie repliche del dato stesso.
Nel momento in cui un'operazione di aggiornamento per una determinata chiave dovesse avvenire su un server che non è l'`owner` di quella chiave, il compito di tale nodo è quello di informare l'`owner` dell'aggiornamento ricevuto e lasciare ad esso il compito di propagare tale informazione agli altri nodi nel sistema.
Si era inizialmente pensato di comunicare ogni aggiornamento solo ad un sottoinsieme dei nodi totali presenti nel sistema ma ciò avrebbe comportato un'inconsistenza di informazioni tra le varie repliche, rendendo indispensabile implementare un ulteriore meccanismo di aggiornamento anche a seguito della richiesta di operazioni di `Get` da parte di un client. Oltre ad aumentare la complessità totale del sistema ciò avrebbe provocato anche una latenza non di poco conto anche per le operazioni di sola lettura.
Si è quindi deciso che il nodo `owner` abbia il compito di informare tutti i nodi nel sistema dell'aggiornamento ricevuto, ma tale aggiornamento è delegato ad appositi thread così che il server possa comunque mantenere la propria operatività in tale periodo. Tale scelta fa si che il lavoro di mantenere consistenti i dati venga svolto all'atto della ricezione di una RPC di scrittura, favorendo così le letture dei dati.
Per questo motivo le operazioni di `Get`, presentano un overhead pressoché nullo, avendo ogni server la possibilità di restituire la sua copia locale del dato richiesto, se esistente nel sistema, con evidenti vantaggi in termini prestazionali per carichi di lavoro costituiti da un gran numero di operazioni di lettura rispetto a quelle di scrittura.
Naturalmente potrebbe occorrere un breve intervallo di tempo affinché il valore di una certa chiave sia propagato dal suo `owner` a tutti gli altri nodi attivi nel sistema, tempo in cui un certo dato potrebbe non essere consistente tra le varie repliche. Poichè potrebbe avvenire che più server chiedano di essere `owner` per una stessa chiave, anche se tale situazione risulta essere poco probabile in un caso reale di utilizzo del sistema, si sono sviluppati due meccanismi per evitare che tali conflitti abbiano luogo:

- un meccanismo per ridurre la finestra temporale nel quale tale conflitto posso avvenire, implementato anticipando la richiesta di un server agli altri nodi di diventare l'`owner` per una chiave, in modo che questi possano essere informati dell'esistenza di tale chiave nel più breve tempo possibile;
- un meccanismo per risolvere eventuali conflitti rimanenti, basati su un algoritmo di riconciliazione in cui, tra i potenziali pretendenti, venga scelto come `owner` il server che presenti il numero più basso della porta su cui è in ascolto o, a parità di porte, l'indirizzo IP più basso. Tale scelta è naturalmente univoca, e tale deve essere, per tutti i server presenti nel sistema.

IV. Secondo requisito non funzionale

A. Progettazione e sviluppo del servizio di registrazione

I nodi server hanno bisogno di conoscere gli altri peers presenti nel sistema e i client che vogliono effettuare le operazioni offerte dalle API hanno bisogno di un server a cui collegarsi. A tal proposito è stato implementato un server register, caricato su un'istanza EC2 e raggiungibile pubblicamente, che permette ai diversi nodi del sistema di ottenere le informazioni di cui hanno bisogno. Si assume che il register sia in ascolto su una porta ben precisa e abbia un indirizzo IP conosciuto.
Il register viene contatto tramite l'uso di specifiche funzioni RPC, definite nel file `pb_register/Register.proto`. In particolare esso espone i seguenti servizi:

- `GetCloseStorageServers()`: permette al client che la invoca di ottenere indirizzo IP e porta dei server attualmente attivi, in modo che esso possa collegarsi al più vicino in termini di latenza di rete;
- `GetListOfPeers()`: permette al server che la invoca di ottenere indirizzo IP e porta degli altri server attualmente attivi;
- `Heartbeat()`: invocata periodicamente da ogni server per comunicare al register che esso è ancora attivo e funzionante e la quale restituisce ai server la lista degli altri nodi ancora presenti nel sistema, in modo che essi possano venire a conoscenza di eventuali nodi che siano andati in crash.

Si è scelto di implementare un servizio di registrazione centralizzato in quanto esso rappresenta la soluzione più semplice

da realizzare e non necessita di overhead ulteriore per sincronizzare le eventuali molteplici istanze altrimenti presenti.
Inoltre, nonostante il register sia sottoposto ad un carico di lavoro maggiore in base al numero di server presenti che deve gestire, l'interazione che egli ha con ogni server non è particolarmente onerosa in termini di risorse computazionali. Esso viene contattato solo nel momento in cui un nuovo server o un nuovo client chiede di partecipare al sistema distribuito di storage o riceve, ad intervalli regolari, l'heartbeat dai server ancora attivi.
Le informazioni che il register mantiene riguardano la lista dei server attualmente presenti nel sistema:

- memorizza indirizzo IP e porta di ogni nuovo server in una lista, associandogli un contatore che indica il tempo trascorso rispetto all'ultimo messaggio ricevuto da tale server;
- mantiene aggiornata tale lista, reimpostando il valore del contatore ad ogni nuovo messaggio. Se il register non riceve alcun messaggio da un server entro un certo periodo di tempo, assume che tale server sia stato soggetto a crash e lo elimina dalla lista.

Da notare che, se fosse il register ad andare in crash, non sarebbe più possibile per nuovi server o client ottenere la lista dei server attivi nel sistema e quindi di collegarsi ad essi ma sarebbe comunque possibile, per i server già attivi e i client collegati, di continuare ad operare in maniera corretta e funzionante.
Nel momento in cui un nuovo server entra nel sistema, esso si collega agli altri nodi edge attivi in modo da recuperare le informazioni sulle coppie chiavi:valori già presenti e di cui ha bisogno affinché la replicazione dei dati sia consistente. A tal proposito esso utilizza l'API `Update` grazie alla quale:

- può comunicare la propria presenza agli altri peers, i quali possono quindi aggiornare la lista che essi mantengono contenente i server attualmente attivi, prima che il timeout per l'invio dell'heartbeat al register sia scaduto (ad ogni heartbeat il register risponde restituendo la lista dei server presenti in quel momento nel sistema);
- chiede ad ogni altro server le informazioni sulle chiavi che esso mantiene e delle quali quel server risulta essere l'`owner`.

Tale funzionalità permette quindi ad eventuali nodi server di partecipare al sistema di storage condiviso anche dopo che questo è già stato avviato, senza che ci siano difformità nelle informazioni memorizzate dai diversi nodi. Il numero di nodi server può quindi incrementare dinamicamente durante il periodo di funzionamento, risultando perciò scalabile rispetto all'eventuale aumento dei client ad esso collegati.

V. Terzo requisito non funzionale

A. Integrazione con il cloud

Il servizio di storage Cloud utilizzato nel sistema distribuito è DynamoDB, del provider AWS. Tale servizio di storage offre una database di tipo noSQL, in cui è possibile creare delle tabelle per salvare record di tipo chiave:valore ed a cui è possibile accedere mediante una apposita SDK.
All'interno del sistema, l'interfacciamento col Cloud avviene in diverse situazioni, a seconda di qual è l'RPC invocata da parte del client. Nei casi di `Put`, `Append` e `Del` è solo l'`owner` della chiave ad occuparsi di aggiornare i valori, mentre l'operazione di `Get` è effettuata allo stesso modo da tutti i nodi edge. In particolare:

- RPC `Put`: il nodo server controlla se il valore appena ricevuto abbia una taglia che eccede il limite che può essere memorizzato in locale. In tal caso, effettua la `Put` del record chiave:valore sul Cloud e per tenere traccia locale dell'esistenza della chiave, viene salvato un record che ha come valore una lista vuota. Se invece il valore abbia una taglia inferiore ma sia già presente sul Cloud un valore associato a quella chiave (perché il precedente valore era maggiore del limite memorizzabile o semplicemente perché quel dato era stato caricato in quanto non acceduto da diverso tempo), il server procede ad eliminare il vecchio dato presente sul Cloud;
- RPC `Append`: si verifica se il valore da appendere fa si che la taglia complessiva associata alla chiave in questione ecceda il massimo memorizzabile in locale e, se ciò accade, l'intero record viene salvato in Cloud. Anche in questo caso, si tiene traccia dell'esistenza della chiave come avviene nel caso della `Put`;
- RPC `Del`: il record chiave:valore viene cancellato in locale e, se il valore risulta essere una lista vuota, si va ad effettuare una cancellazione anche sul Cloud;
- RPC `Get`: il nodo edge effettua una scansione delle chiavi salvate in locale e nel caso in cui trovi la chiave richiesta dall'utente ma associata ad un valore pari ad una lista vuota, effettua una query verso il Cloud per recuperare tale valore ed inviarlo al client. Al termine dell'operazione, se il valore ha una taglia inferiore al limite che può essere memorizzato sul nodo, allora il dato chiave:valore viene salvato anche in locale, così che successive operazioni di `Get` sulla stessa chiave abbiano latenze inferiori.

Per poter interagire agevolmente col servizio DynamoDB è stata realizzata una apposita API, che a sua volta si basa sulla SDK per il linguaggio Golang: tale API è fornita nel package `dynamo_db_utils`, e permette di realizzare la corrispettiva operazione di ognuna delle RPC offerte dal nodo edge.
Per poter interagire col servizio Cloud vengono usati due file, entrambi con percorso sulla macchina `$HOME/.aws`:

- *credentials*: questo file contiene le credenziali per poter utilizzare la sessione AWS ed accedere al servizio Cloud;
- *config*: questo file contiene invece la regione Cloud in cui viene offerto il servizio, così come anche l'indicazione sul formato dell'output ricevuto a seguito della chiamata ad una funzione della SDK.

VI. Limitazioni riscontrate

Vengono in seguito riportate alcune delle limitazioni riscontrate durante lo svolgimento del progetto

- la taglia massima per un item in DynamoDB è pari a 400 KB. Quindi, se è necessario salvare dei record chiave:valore che eccedono questa taglia, occorrerebbe salvare all'interno della tabella non il record in sé, bensì un link ad un bucket del servizio AWS S3, al cui interno andrebbe salvato il reale record chiave:valore sotto forma di file all'interno del bucket stesso;
- non essendo l'applicazione sviluppata tollerante ai guasti, eventuali crash di nodi server nel sistema risulta in un comportamento non atteso e una conseguente perdita nella consistenza tra le varie replicazioni di quelle chiavi per cui il nodo mal funzionante risulti essere l'`owner`.
- essendo i nodi edge del sistema sono scoperti all'avvio del sistema stesso, non si è in grado di gestire correttamente eventuali abbandoni volontari o crash dei nodi server. In particolare sarebbe stato necessario implementare un algoritmo che permetta di definire nuovi `owner` per le chiavi gestite dal server che ha abbandonato il sistema.

VII. Testing del sistema

A. Test delle funzionalità

Il funzionamento del sistema sviluppato è stato testato al variare di diversi scenari di utilizzo. Nelle diverse situazioni sono state alternate le operazioni di `Put`, `Append`, `Get` e `Del` verificando che, trascorso un certo tempo, tutti i server presentassero i valori aggiornati per le diverse chiavi.
In particolare, i diversi scenari di utilizzo pensati sono stati i seguenti:

- Un singolo client, collegato sempre allo stesso server, che effettua operazioni su un un'unica chiave la cui taglia è tale per cui questa rimanga sempre salvate in locale. Non vi è quindi, in questo caso, interazione con DynamoDB.
- Un singolo client, collegato sempre allo stesso server, che effettua operazioni su un'unica chiave la cui taglia è tale per cui è necessario che questa sia salvata su DynamoDB e correttamente da lì recuperata.
- Molteplici client, collegati a diversi server, che effettuano operazioni su un'unica chiave la cui taglia è tale per cui questa rimanga sempre salvate in locale.
- Molteplici client, collegati a diversi server, che effettuano operazioni su un'unica chiave la cui taglia è tale per cui è necessario che questa sia salvata su DynamoDB e correttamente da lì recuperata.
- Molteplici client, collegati a diversi server, che effettuano operazioni su diverse chiavi la cui taglia è tale per cui queste rimangano sempre salvate in locale.
- Molteplici client, collegati a diversi server, che effettuano operazioni su diverse chiavi la cui taglia è tale per cui è necessario che queste siano salvate su DynamoDB e correttamente da lì recuperate.
- Un client che tenta di effettuare operazioni di `Append`, `Get` e `Del` su chiavi non esistenti nel sistema, che devono correttamente tornare e gestire dei messaggi di errore.

Non è stato possibile riportare un test che dimostri il corretto caricamento dei record chiave:valore su DynamoDB al termine del timeout, embeddato nel codice, che categorizza una chiave come scarsamente acceduta. Ciò poiché non è possibile configurare, per scelta implementativa, tale parametro allo startup del sistema e, inoltre, il client è trasparente rispetto a dove il server a cui è collegato recupera la chiave richiesta.
L'unico modo che il client ha per riconoscere se la chiave richiesta era presente in locale o sul Cloud è calcolare il tempo impiegato dal server per ritornarne i valori. Purtroppo tale meccanismo, pur essendo un buon indicatore, non è affidabile in quanto potrebbe essere influenzato da fattori quali la velocità di connessione, il carico sul sistema e la presenza dei lock sulle struttura di memorizzazione dei server.

B. Test delle performance

La seconda tipologia di test effettuata si focalizza sul misurare le prestazioni del sistema distribuito, in particolare monitorando come esso reagisca in base alle diverse tipologie di workload a cui è sottoposto. I due workload utilizzati hanno le seguenti caratteristiche:

- Workload 1:
  - 85% operazioni di `Get`;
  - 15% operazioni di `Put`.
- Workload 2:
  - 40% operazioni di `Get`;
  - 40% operazioni di `Put`;
  - 20% operazioni di `Append`.

Per ognuno dei workload inoltre sono state valutate le prestazioni, in termini del tempo di risposta sperimentato dai client, al variare sia del numero di client che del numero di server, utilizzando i seguenti valori:

- Numero di server:
  - 5;
  - 10;
  - 15;
- Numero di client:
  - 10;
  - 20;
  - 30;

È stato sviluppato un apposito package per condurre tali test, `auto_client`, che contiene il file `auto_client.go` il quale, utilizzando le API offerte per i nodi client, può effettuare in automatico RPC verso il server.
All'interno del file, sono state definite il numero di operazioni totali da fare e la divisione di tali operazioni, per ciascuno dei due workload. La scelta è stata quella, come detto, di far aumentare il numero di client piuttosto che il numero di operazioni totali, che è stato fissato a 50 per ogni singolo client. Ciò perché non è importante il numero di operazioni totali che vengono effettuate quanto più il numero di operazioni concorrenti che il sistema deve gestire. Naturalmente aumentare il numero di client comporta un conseguente aumento nel numero delle operazioni simultanee effettuate in ogni istante temporale.
Vi è inoltre uno script Python (`script.py`) utilizzato per creare in sequenza diversi thread che lanciano l'eseguibile

`auto_client`: ognuno dei thread effettua le operazioni di uno dei due workload ed, all'interno del file, è inoltre possibile cambiare i parametri passati al singolo thread ed usati per definire la configurazione del client, tra questi:

- quale chiave utilizzare per le operazioni. Per far si che i client vadano in concorrenza, sono state utilizzate in ogni test un sotto-insieme delle chiavi totali presenti all'interno del file;
- quale valore associato alla chiave utilizzare. Anche in questo caso, vi sono valori di taglia diversa, in modo da stimolare sia operazioni che interagiscano col Cloud sia quelle locali;
- quale tipo di workload eseguire.

Per simulare un caso di utilizzo il più realistico possibile, ogni client dopo aver eseguito un'operazione genera un certo timeout casuale su cui viene eseguita una `Sleep`, per permettere a client differenti di effettuare operazioni a tempi differenti: tale timeout è distribuito secondo un'esponenziale con media di 2.5s, valore che consente di caricare il sistema con un grande numero di operazioni ravvicinate da gestire.
Inoltre, all'inizio di ciascuna funzione che implementa uno dei due workload, viene impostato il seme da cui verranno generati i valori random pari al valore del PID del thread corrente.
Di seguito, vengono riportati ed analizzati alcuni risultati grafici per i valori ottenuti con i test delle performance.

C. Risultati per il workload 1

Consultando i grafici da [2] a [10], è possibile notare che, a parità del numero di client, maggiore è il numero di server e migliore è la scalabilità del sistema: tale scalabilità diventa particolarmente evidente nei casi di 30 client.
Infatti, confrontando le 2, 3 e 4 (corrispondenti alle [4], [7] e [10]) si evince chiaramente una diminuzione nel tempo di risposta, sia medio che massimo, nel momento in cui si aumenta il numero di server a disposizione.
Nonostante questo, essendo le operazioni in lettura le operazioni meno onerose e con i tempi di risposta più brevi, la differenza non è eccessiva e questo è perfettamente in linea con le scelte progettuali effettuate.
Invece, per un numero basso di client, le prestazioni migliori si hanno con un numero di server minori: prendendo ad esempio i casi delle 5, 6, 7 (corrispondenti alle [2], [5], [8]) è possibile notare che il caso migliore è quello in cui vengono impiegati 5 server.
Ciò avviene poiché, avendo un numero maggiore di server, si fa entrare il sistema in un caso di over-provisining che ha come conseguenza uno scambio di un numero maggiore di messaggi fra i server nel gestire le operazioni in scrittura, necessario a mantenere consistenti tra loro le diverse repliche di ogni record. In tal modo si introduce un certo overhead che, se pur minimale, è comunque osservabile e causa un leggero aumento nei tempi di risposta.
È interessante notare come in queste situazioni, essendo il sistema sottoposto ad un carico non particolarmente eccessivo, è facilmente riconoscibile ogni tipo di operazione effettuata. In particolare i punti più bassi, nell'ordine del microsecondo sono relativi a richieste di `Get` riferite a chiavi salvate localmente.
I punti con tempi di risposta di un centinaio di millisecondi corrispondono a richieste di `Get` per le quali il recupero dei valori richiede l'interazione con DynamoDB.
I punti con tempi di risposta superiori corrispondo a richieste di`Put`.
È possibile notare una differenza, seppur meno accentuata, tra il caso in cui la chiave venga salvata in locale e il caso in cui essa debba essere caricata su DynamoDB.

D. Risultati per il workload 2

Il workload 2, al contrario di quello precedente, prevede che vi sia una maggiore presenza di operazioni di scrittura piuttosto che di lettura.
I risultati grafici prodotti sono riassunti nei [11]- [19].
Anche in questo caso, a parità del numero di client, si riesce ad ottenere scalabilità aumentando il numero dei server, seppur minore di quella ottenibile nel primo caso. Questo è però nuovamente in linea con la scelta progettuale di delegare la consistenza dei dati a valle delle operazioni di scrittura, quindi, in questo caso, delle `Put` e delle `Append`.
La scalabilità risulta evidente nel momento in cui si passa da 10 client a 20: infatti, consultando i grafici, è possibile notare un sovraccarico eccessivo nel caso in cui si adottino solo 5 server ([11], 8, [12]), che viene invece gestito in maniera molto più efficace passando a 10 server ([14], 9, [15]), fino ad un ulteriore evidente miglioramento ottenibile adottando 15 server ([17], 10, [18]).
Prendendo invece in esame i casi a 30 client, le diverse configurazioni portano il sistema ad essere molto carico, a causa della presenza massiva di operazioni di aggiornamento fra i server. Ciò si traduce in tempi di latenza minori nei primi momenti di utilizzo del sistema per un numero di server maggiori, che riesce in questa fase a smaltire meglio le richieste ricevute ([20], [21], [22]).
Si arriva però poi ad una situazione in cui l'aumento delle richieste pendenti di scrittura e la necessità di propagare tali aggiornamenti agli altri server porta il sistema in una situazione di stallo che lo rende, per un certo intervallo di tempo, incapace di accettare nuove richieste dai client finché le precedenti operazioni non siano state completate ([13], [16], [19]). Tale situazione è anche accentuata dal fatto che server e client risultano essere in esecuzione sulla stessa macchina, se pur in container diversi, portando quindi ad una condivisione delle risorse fisiche della stessa. Avendo la macchina in questione, come naturale che sia, un numero massimo di processori logici (8), essa consente ad un numero limitato di thread di poter eseguire contemporaneamente, numero non sufficiente a smaltire tutti i thread che vengono creati dal sistema.
Inoltre, avendo realizzato i test in locale, in quanto non era possibile disporre di una piattaforma di edge computing, e avendo implementato una logica per cui un client si connetta al server che risulta avere una latenza di rete minore, non è assicurato che i client vengano distribuiti in maniera equa

sui server a disposizione. È quindi possibile che nodi diversi presentino un carico di lavoro diverso.

VIII. Piattaforma di sviluppo e testing

A. Piattaforma hardware e software

Sono state utilizzate le seguenti piattaforme per lo sviluppo ed il testing del sistema
Piattaforma software:

- Golang v1.17;
- Visual Studio Code IDE;
- ProtocolBuffer versione 1.26;
- Docker versione 20.10.7;
- docker-compose versione 1.27.4;

Piattaforme hardware:

- Piattaforma 1:
  - CPU: intel i5-10501G (4cores, 1.0Ghz);
  - 12 RAM DDR4;
- Piattaforma 2:
  - CPU: AMD Ryzen 7 4700U (8cores, 2.0Ghz);
  - 16 RAM DDR4;
- Sistemi Operativi:
  - Ubuntu 20.04 LTS;
  - Windows 10;

B. Librerie esterne

Sono state utilizzate le seguenti librerie esterne a supporto dello sviluppo software del sistema

- Libreria per gRPC: ”google.golang.org/grpc”
- Amazon AWS SDK:
  - ”github.com/aws/aws-sdk-go/aws”
  - ”github.com/aws/aws-sdk-go/aws/session”
- DynamoDB SDK:
  - ”github.com/aws/aws-sdk-go/service/dynamodb”
  - ”github.com/aws/aws-sdk-go/service/dynamodb/dynamodbattribute”
  - ”github.com/aws/aws-sdk-go/service/dynamodb/expression”

  Tutte le dipendenze sono riassunte all’interno del file `go.mod`

Riferimenti bibliografici

[1] IEEE LaTeX template
[2] Workload 1, 10 client e 5 server
[3] Workload 1, 20 client e 5 server
[4] Workload 1, 30 client e 5 server
[5] Workload 1, 10 client e 10 server
[6] Workload 1, 20 client e 10 server
[7] Workload 1, 30 client e 10 server
[8] Workload 1, 10 client e 15 server
[9] Workload 1, 20 client e 15 server
[10] Workload 1, 30 client e 15 server
[11] Workload 2, 10 client e 5 server
[12] Workload 2, 20 client e 5 server
[13] Workload 2, 30 client e 5 server
[14] Workload 2, 10 client e 10 server
[15] Workload 2, 20 client e 10 server
[16] Workload 2, 30 client e 10 server
[17] Workload 2, 10 client e 15 server
[18] Workload 2, 20 client e 15 server
[19] Workload 2, 30 client e 15 server
[20] Workload 2, 30 client e 5 server, primi 60 secondi di esecuzione
[21] Workload 2, 30 client e 10 server, primi 60 secondi di esecuzione
[22] Workload 2, 30 client e 15 server, primi 60 secondi di esecuzione
[23] Architettura del sistema distribuito ad alto livello

Grafici

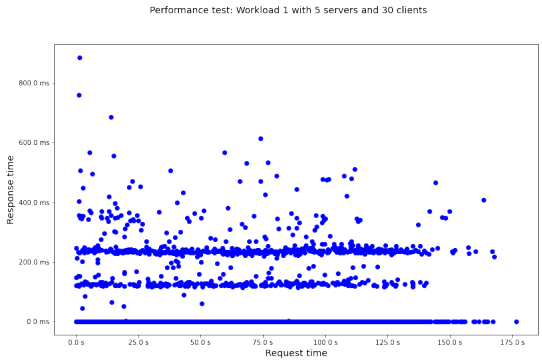

Figura 2. Caso 30 client, 5 server, workload 1.

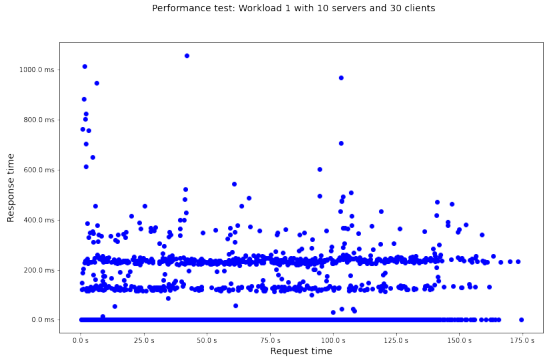

Figura 3. Caso 30 client e 10 server, workload 1.

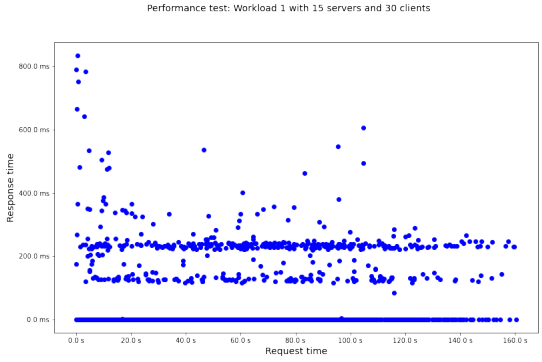

Figura 4. Caso 30 client e 15 server, workload 1.

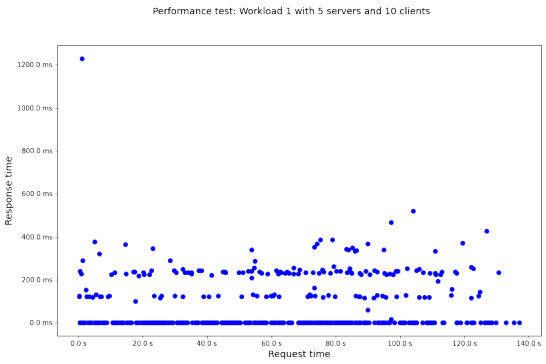

Figura 5. Caso 10 client, 5 server, workload 1.

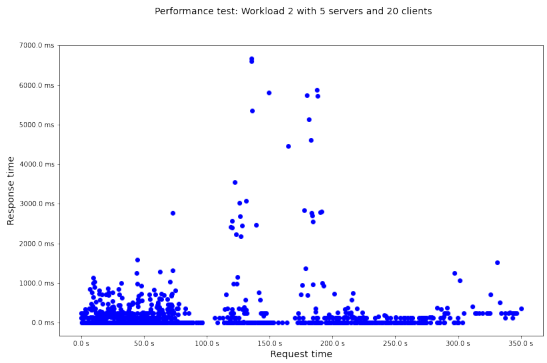

Figura 8. Caso 20 client, 5 server, workload 2.

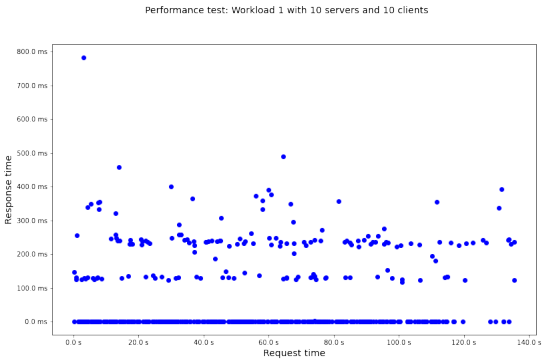

Figura 6. Caso 10 client e 10 server, workload 1.

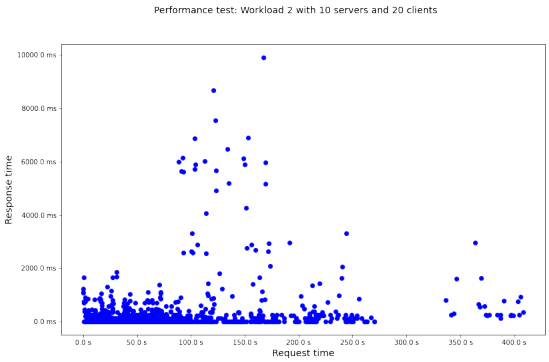

Figura 9. Caso 20 client, 10 server, workload 2.

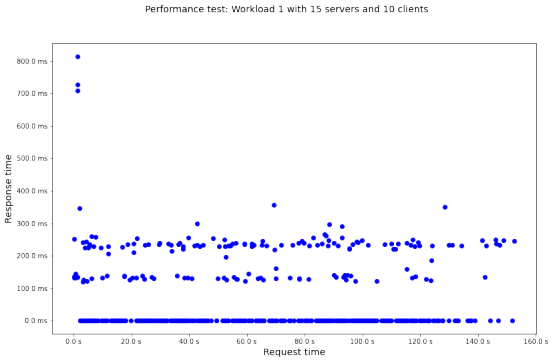

Figura 7. Caso 10 client e 15 server, workload 1.

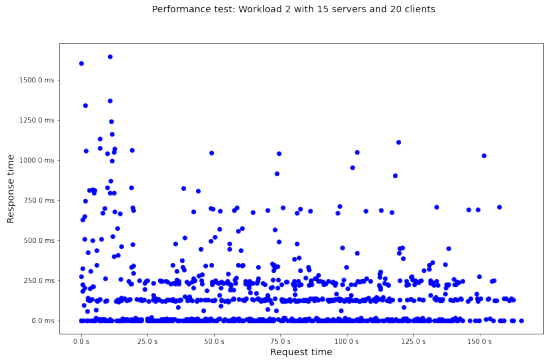

Figura 10. Caso 20 client, 15 server, workload 2.