Relazione Progetto SDCC

Realizzazione di un Datastore Distribuito (Progetto A1)

Matteo Ferretti (0300049)
Dipartimento di Ingegneria
Università di Tor Vergata
Roma Lazio Italia
matteo.ferretti.97@alumni.uniroma2.eu

INTRODUZIONE

Il sistema (sviluppato in go) ha l'obiettivo di fornire un servizio di storage replicato di tipo chiave-valore in uno scenario di edge computing. Più nello specifico, le chiavi ed i valori sono interpretati dal sistema come nome e contenuto di veri e propri file salvati sui nodi. Questa scelta è stata effettuata per semplicità implementativa e gestionale.

Le funzionalità operative che il sistema prevede sono:

1. **put(key, value):** consente di memorizzare il valore key a cui è associata una chiave value, eventualmente sovrascrivendo il valore già presente.
2. **get(key):** consente di leggere il valore a cui è associata la chiave key specificata
3. **del(key):** consente di rimuovere il valore a cui è associata la chiave key

ARCHITETTURA DEL SISTEMA

Il sistema realizzato è composto da tre tipi di nodi differenti, più il nodo client che si connetterà per la fruizione del servizio.

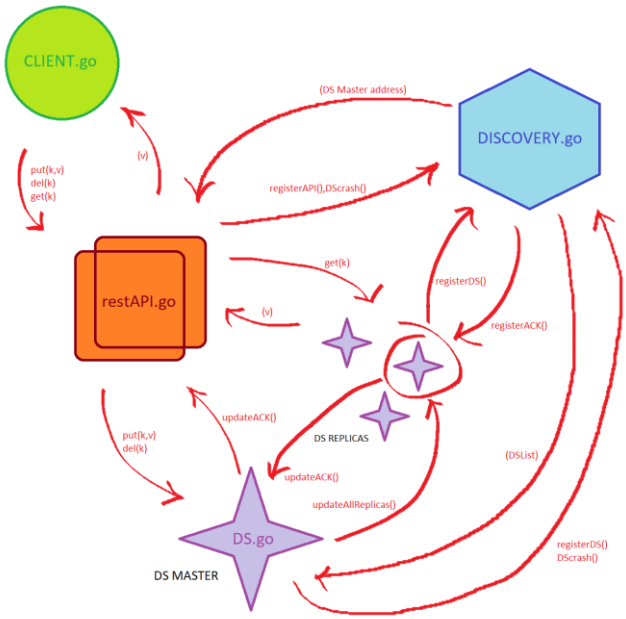
Il nodo **Discovery** consiste in un servizio di registrazione, senza la quale nessun altro nodo può connettersi entrando a far parte del sistema. Il nodo di discovery è l'unico che è stato realizzato in maniera centralizzata, e qualora quest'ultimo subisca un crash, il sistema può continuare a lavorare esclusivamente con i nodi già presenti; se dei nodi esterni, di qualsiasi tipo, cercassero di unirsi al sistema, dovranno attendere che il nodo discovery torni operativo prima di riuscirci.

Il nodo **Datastore** (abbreviato con DS) è il nodo responsabile dell'effettivo salvataggio delle chiavi e dei valori, intesi come file. I nodi di tipo datastore possono essere molteplici, e potranno unirsi al sistema continuamente.

Il nodo **restAPI** è, come dice il nome, una API di tipo rest. Essa ha il compito di interfacciarsi con i client che si connettono al sistema per andare a chiamare delle funzionalità sui datastore. Anche queste possono essere molteplici, ed ogni client comunicherà con una di esse.

Il nodo **Client** è il nodo che si connette al sistema per richiederne le funzionalità. Ovviamente anche i client possono essere molteplici.

Il funzionamento di base del sistema prevede quindi la connessione di un client, al quale verrà assegnata una delle API connesse in quel momento. Il client sottoporrà la richiesta all'API, la quale procederà con il comunicarla ad uno dei datastore. Quest'ultimo procederà con l'effettuare l'operazione richiesta, per rispondere infine all'API, la quale a sua volta la riporterà al Client.



Lo schema appena riportato corrisponde ad un'illustrazione semplificata dei fondamenti operativi del sistema.

SCELTE PROGETTUALI

Si è scelto di considerare le coppie chiavi valore come nomi e contenuto di file salvati localmente sui nodi di tipo datastore. Questa scelta è stata effettuata al fine di non appoggiarsi ad alcun tipo di servizio di datastore esistente, riuscendo lo stesso ad avere una semplice gestione della persistenza. Non coinvolgere servizi esterni inoltre riduce le dipendenze, e questo si traduce in un vantaggio prestazionale e gestionale non indifferente dal momento in cui si è anche scelto di eseguire l'applicativo su un supporto di containerizzazione, ovvero Docker.

Altra scelta sostanziale effettuata è stata quella di gestire le restful API come gruppo di nodi, e non come nodo centralizzato. Il motivo di questa scelta risiede nel fatto che in questo modo il sistema può godere di una maggiore tolleranza ai guasti, in quanto qualora l'API contattata dal Client subisca un crash, a quest'ultimo verrà semplicemente comunicata un'altra API da usare.

Discorso analogo vale per i datastore: per garantire tolleranza ai guasti e mantenere semplice la gestione della concorrenza, i datastore sono organizzati secondo un'architettura di tipo

Master/Slave. Nello specifico, il datastore master è colui che viene contattato al fine di effettuare operazioni di put() e del() nel sistema, così da semplificare gli aspetti di concorrenza. Le operazioni di get() risultano invece distribuite, e potranno andare a contattare qualsiasi datastore nel sistema (a patto ovviamente che non si trovi in stato di aggiornamento: in tal caso dovrà attenderne il completamento).
Infine, come accennato prima, si è scelto di containerizzare l'applicazione utilizzando Docker, così da garantire una facile gestione degli aspetti di scalabilità. Per l'aggiunta in qualsiasi momento di qualsiasi tipo di nodo sarà sufficiente mandare in esecuzione un altro container.

ASPETTI IMPLEMENTATIVI

Si è scelto di realizzare il sistema sfruttando chiamate HTTP tra i nodi. Ogni nodo è in grado di mettersi in ascolto su una certa porta o di inviare richieste di un certo tipo ad altri nodi.
All'inizio, qualsiasi nodo che vuole entrare nel sistema è costretto a registrarsi al Discovery, il quale salverà in persistenza l'indirizzo e la tipologia di ogni componente. Questa scelta aumenta la tolleranza ai guasti, poiché qualora il nodo Discovery andasse in crash e necessitasse di un reset, al riavvio ritroverà tutti i nodi già presenti e li riacquisirà,
Quando è un datastore a registrarsi, esso verrà eventualmente eletto come master qualora fosse il primo. Successivamente, per ogni datastore che arriverà nel sistema, verrà comunicato al master l'indirizzo della nuova replica che dovrà aggiornare. Ogni volta che è una replica ad effettuare la join al sistema, essa verrà riallineata dal master immediatamente, non terminando di fatto la registrazione fino a quando non risulta aggiornata (non potendo essere nel frattempo contattata da operazioni di get() quindi).
In sostanza il master è quello che viene contattato per le put() e le del(), e che viene informato dal discovery di ogni nuova replica che si connette.
Quando è una API a registrarsi, oltre il fatto che il discovery ne salva l'indirizzo in persistenza, provvederà con il consegnarle la lista di tutti i datastore presenti fino a quel momento, compreso il master. Anche le API vengono aggiornate ogni volta che si unisce una replica, così da conoscerla qualora bisogni effettuare un'operazione di get.
L'unico nodo di cui il Discovery non tiene traccia non salvandolo in persistenza è il nodo Client. Quando questo nodo si registra, il discovery non fa altro che comunicargli una API da utilizzare.
Quando si verifica un crash, di solito il sistema se ne accorge appena qualcuno prova a comunicare con il nodo coinvolto. Il comportamento iniziale in talune circostanze è quello di avvisare immediatamente il discovery, il quale conoscendo tutti i nodi nel sistema procederà con l'effettuare diverse azioni. Qualora si verifichi il crash di un datastore replica, alla prossima operazione di aggiornamento (put() o del()) il master se ne accorgerà e lo comunicherà al discovery, che a sua volta avviserà le API cosi che possano rimuoverlo dalla loro lista locale escludendolo dalle successive operazioni di get. Potrebbe anche verificarsi che sia proprio una API ad accorgersi del crash di una replica, proprio durante una get, a patto che sia la prima a contattarla. Anche in questo caso verrà avvisato il discovery che procederà con il comunicarlo al master e alle altre api. Qualora fosse il master a crashare, sarà una API ad accorgersene la prossima volta che proverà a contattarlo per eseguire un aggiornamento richiesto dal client, oppure potrebbe anche essere una replica che risponde dopo un aggiornamento. Stesso schema, lo comunicherà al discovery che a sua volta lo comunicherà alle altre API. Ciò che avviene di ulteriore in questo caso è l'elezione di un nuovo master da parte del Discovery nella lista dei datastore, per poi comunicare la decisione all'eletto. Il datastore eletto imposterà a true la sua variabile booleana di master, così da accedere da quel momento in poi alle sezioni di codice ulteriori caratterizzanti del comportamento da master.
Il crash di una API è scoperto dal client nel momento in cui prova ad effettuare una richiesta, o da una replica che sta inviando indietro il risultato di un'operazione richiesta. Anche qui viene contattato il discovery dal nodo che ha scoperto il crash, così che il discovery possa rimuoverlo dalla lista delle api comunicate ai client, e fornire ai client nuove api da usare.
Il crash del discovery viene scoperto quando qualsiasi componente esterno prova a connettersi al sistema, o quando qualsiasi nodo già presente prova a comunicare un crash. In queste situazioni il nodo viene messo in attesa che il discovery torni operativo, il quale riacquisirà localmente i nodi già presenti e procederà col servire le richieste arrivate nel frattempo.

Le operazioni che effettivamente il client richiede, a livello di sistema sono viste delle scritture o letture di file locali di un nodo datastore. Ogni volta che bisogna effettuare una delle suddette operazioni, il datastore acquisisce un mutex, così da garantire la mutua esclusione per la sezione critica tra le diverse goroutine che potrebbero essere state chiamate.
Come accennato in precedenza, il nodo datastore può essere di tipo master o di tipo slave. Non è stato implementato un codice differente per le due tipologie di questo nodo, piuttosto è stata realizzata una variabile booleana che ne tiene conto, così da far capire al master di dover effettuare delle operazioni aggiuntive in alcune circostanze. Quando viene richiesta una scrittura (put() o del()), viene sempre contattato il master, il quale procederà con l'aggiornare prima se stesso e poi tutti gli altri datastore in maniera parallela.
Sono stati adottati dei mutex ogni volta che un nodo ha bisogno di scrivere o leggere risorse condivise. Ad esempio, ogni datastore prende mutex per leggere o scrivere file in persistenza. Inoltre, le API, il discovery e i datastore prendono e rilasciano mutex differenti ogni volta che devono aggiornare le proprie liste locali di conoscenza degli altri nodi del sistema. Questa scelta è fondamentale per garantire la consistenza in un linguaggio come go, dato che ogni volta che viene chiamata una funzione su un nodo, essa parte come goroutine. Se infatti si verifica un crash di un datastore mentre un altro si unisce al sistema, mentre il master aggiorna la propria lista da solo per via del crash di una replica che stava aggiornando, potrebbe ritrovarsi ad appenderne invece uno nuovo nello stesso momento, comunicato dal discovery, di

fatto scrivendo concorrentemente sulla sua lista locale. Per evitare questi problemi sono stati adottati dei mutex.

Per garantire una semantica degli errori di tipo at least once, ogni volta che un nodo si ritrova a non ricevere risposta da un altro nodo durante la sottomissione di una richiesta, entrerà in un ciclo al fine di riprovare la comunicazione ogni tre secondi, comunicando al Discovery il crash, ed eventualmente ricevendo un nuovo indirizzo da contattare che lo porti fuori dall'attesa.
Per garantire una consistenza che sia almeno di tipo finale, si ha la certezza che prima o poi tutte le repliche vengano aggiornate. Questo perché il master si aspetta ogni volta una risposta dalle repliche (non bloccante per il client), le quali se non rispondono inizialmente vengono rimosse dalle repliche online, considerate come faulty. Quando le repliche torneranno operative, non potranno terminare la registrazione fino a quando non saranno riallineate dal master. Di fatto quindi, o riesce subito l'aggiornamento, o se crashano vengono aggiornate al riavvio. Questo garantisce che quindi prima o poi vengano aggiornate.
La simulazione delle latenze nel sistema è effettuata tramite degli sleep messi all'interno delle funzioni che si occupano di eseguire le operazioni richieste dall'utente, solo sul datastore (proprio per rappresentarne il tempo di connessione).

SOFTWARE UTILIZZATO

Per sviluppare il sistema è stato utilizzato Visual Studio Code, con le estensioni di golang per programmare in go, quelle di Docker per la gestione dei container e quelle di git per la sincronizzazione delle modifiche su github. I container di Docker utilizzano come immagine base del Dockerfile Ubuntu 20.04, con installati sopra solamente git, golang e la libreria aggiuntiva utilizzata, ovvero github.com/gorilla/mux. Questa libreria ha semplificato notevolmente le capability di ascolto dei diversi nodi e dell'invio delle richieste, permettendo tramite poche ed intuitive righe di codice, di effettuare le suddette operazioni.
Per quanto riguarda il testing, esso è stato effettuato quasi totalmente tramite le richieste inviate dal Client e dallo strumento di debug di Visual Studio Code. Si è comunque provveduto poi a scrivere i casi di test per ogni funzione del sistema, utilizzando la libreria testing di go. In ogni test è stata utilizzata la libreria net/http per la realizzazione di richieste http mock, oltre i normali oggetti mock passati alle funzioni.

TESTING PRESTAZIONALE

Per effettuare il testing delle prestazioni del sistema realizzato, è stato scritto all'interno della cartella Performance Tester, uno script in go (performance_tester.go). All'interno del codice è possibile vedere come vengano creati inizialmente dei nomi "mock" da leggere o scrivere all'interno dei nodi datastore. Il test realizzato sottopone il sistema a un numero arbitrario di richieste divise in un 15% di put ed un 85% di get. Nello specifico il numero scelto per realizzare il testing è stato 100 richieste.
Ciò che accade è che all'inizio vengono preparati dei file già presenti nei datastore che potrebbero essere richiesti o meno (è possibile anche che vengano richiesti file non presenti, proprio per garantire uno scenario più reale possibile di richieste che non per forza vadano a buon fine). Si procede poi con un ciclo che invia le richieste una dopo l'altra.
Sono stati realizzati due test, uno mandando le richieste tutte insieme, ed un altro separandole invece un secondo l'un l'altra.
Riportando in un grafico quindi gli andamenti per richieste a latenza zero secondi l'una dall'altra e un secondo, si notano interessanti risultati.
La linea rossa rappresenta ciò che accade ai tempi di attesa quando tutte le richieste arrivano insieme, e la crescita di quota è indicativa del fatto che le richieste vanno ad aspettare sempre di più prima di essere servite, proprio perché avranno in coda davanti a loro le richieste precedenti che tengono i mutex occupati. Le prime vengono servite velocemente, ma poi i tempi vanno a crescere.
Diverso è l'effetto invece del separare di un secondo le richieste tra loro, poiché si nota come i tempi restano stabili senza salire.
Separare le richieste riesce a dare il tempo giusto di completare ognuna prima di iniziare a lavorare la successiva, garantendo tempi di risposta ottimali.
Lo studio di entrambe le statistiche è riportato nel foglio di calcolo all'interno della cartella performance tester. Per realizzarle inoltre è stato preso l'andamento medio di 3 run per ciascuna, modificando il seed randomico di estrazione ogni volta.
Le statistiche osservate quindi, non sono il risultato di una run singola, ma di diverse run di cui è stata effettuata la media dei valori.
È possibile modificare il seed e raccogliere altri dati sfruttando i parametri lasciati come modificabili all'interno del performance tester.