Sistemi Distribuiti e Cloud Computing – AA 2020/2021

Progetto A1: Sistema di storage distribuito di tipo chiave-valore per l'edge computing

Master Degree Students: *Matteo Chiacchia, Valerio Cristofori*

I. Introduction (*Heading*)

Il sistema progettato e implementato è un'applicazione di storage distribuita per l'edge computing. Realizzata tramite il linguaggio di programmazione ***Go***, essa è stata pensata come un servizio di storage persistente, replicato e consistente di tipo chiave-valore, fornito da nodi edge che comunicano tra loro.

Si è deciso di focalizzare l'attenzione su un contesto di **Smart City**, in cui i client che interagiscono, tramite chiamate RPC, con l'edge, sono sensori di temperatura. Il tutto è stato integrato con ***DynamoDB*** (servizio di storage Cloud di tipo chiave-valore) per incrementare la scalabilità rispetto ai dati, salvando infatti in esso i valori meno acceduti. Ciò è molto utile considerando il fatto che i nodi edge hanno capacità di archiviazione limitata.

II. Ambiente di sviluppo e software utilizzati

Il software è stato scritto in *Go* (v1.17.1) utilizzando come ambiente di sviluppo ***GoLand***. Per l'accesso alla repository è stato utilizzato ***GitHub***, con *Git* come sistema di versioning. Non avendo avuto a disposizione risorse reali utilizzate dall'edge computing, si è deciso di simulare il tutto tramite container ***Docker*** (ogni edge è un container) gestiti tramite ***Docker-Compose.*** Come ambiente Cloud è stato scelto ***DynamoDB***, essendo un database distribuito NoSql di tipo chiave-valore, quindi ampiamente adatto allo scopo.

E' stato utilizzato ***Terraform*** per definire e inizializzare l'infrastruttura Cloud su AWS. Per interagire con DynamoDB sono state utilizzate funzioni ***Lambda,*** creando i .zip contententi la logica delle funzioni e caricandoli su un bucket S3 di AWS.

III. Overview

La comunicazione tra Sensori ed *Edge Layer* avviene tramite 4 tipi di chiamate RPC:

- ***Put (Key, Value)***: i client salvano un valore sull'edge associandogli una chiave
- ***Get (Key):*** i client richiedono all'edge il valore associato a una chiave
- ***Delete (Key):*** i client comunicano all'edge di voler eliminare dal datastore il valore associato a una chiave
- ***Append (Key, Value):*** i client richiedono all'edge di aggiungere un altro valore a quello già associato alla chiave in questione.

Una volta che l'*Edge Layer* ha ricevuto una richiesta *RPC* dal client ha due strade in cui muoversi:

- Soddisfare la richiesta utilizzando unicamente le proprie risorse
- Comunicare con il Cloud perché la richiesta effettuata non può essere eseguita (eg. il valore associato a una chiave non è presente nell'edge perché precedentemente trasferito nel Cloud)

Di seguito una raffigurazione ad alto livello della struttura dell'applicazione:

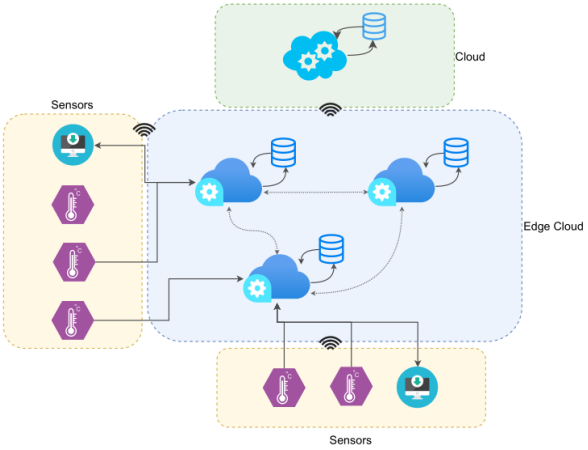

Fig. 1. Vista ad alto livello

IV. Architettura del sistema

L'architettura utilizzata per lo storage di dati chiave-valore si compone di molteplici container Docker che interagiscono tra loro per simulare in locale il comportamento di un sistema distribuito su piu' nodi.

Si distinguono tre tipologie di container Docker che corrispondono ad altrettante tipologie di immagini attraverso le quali sono stati costruiti:

- **Server**: rappresenta il nodo edge del sistema (Edge Nodes).
- **Client**: rappresenta un cluster di sensori di temperatura e siti di consumo dei valori (Client Nodes).
- **Master**: nodo responsabile della fase di registrazione nel sistema di tutti gli altri componenti.

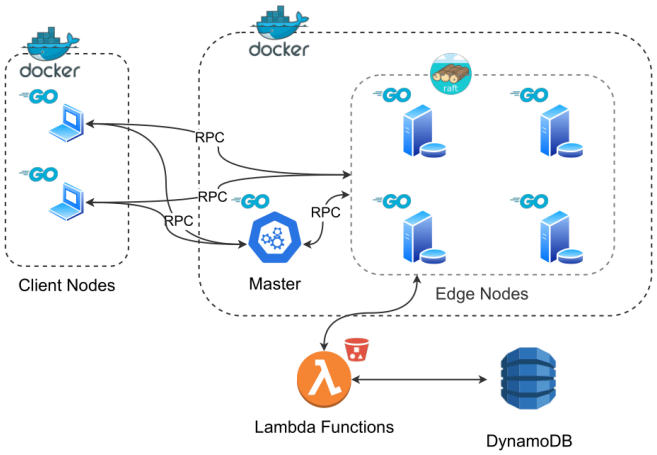

Fig. 2. Architettura del sistema

4.1 Server

Nodo responsabile dello storage persistente su una struttura dati di tipo *sync.Map(key,value)*. Il nodo edge offre 4 funzioni che vengono chiamate dai Client per interagire con lo storage (sia locale al server che remoto nell'istanza di DynamoDB) attraverso RPC.

Altro aspetto importante di questo nodo e' la continua gestione della memoria che alloca, inviando i valori meno acceduti recentemente, nel caso di sovraccarico, a DynamoDB. Questa interazione con il Cloud viene resa possibile tramite chiamate a delle *Lambda Functions* che risiedono su un Bucket S3 su AWS.

I nodi Server del cluster sono in continua interazione tra loro. E' stato infatti implementato l'algoritmo per il consenso distribuito *RAFT* con lo scopo di replicare (in maniera consistente: consistenza *sequenziale*) il log delle varie operazioni sullo storage locale.

4.2 Client

Il nodo che interagisce con i nodi edge attraverso chiamate RPC, facendo eseguire ai Server le 4 funzioni di interazione con lo storage. Questo nodo e' stato inserito nel cluster dei container Docker per simulare l'interazione tra sensori e nodi edge (nell'ottica di un contesto di Smart City).

4.3 Master

Il nodo e' responsabile della fase di registrazione al sistema da tutti i nodi che lo compongono. Tiene traccia, infatti, di ogni nodo Server che appartiene al sistema e restituisce la lista dei Server running ai nodi Client che si registrano.

4.4 Lambda Functions

La logica dei 4 tipi di chiamate RPC viene riportata anche come interazione con il Cloud. Sono state implementate 4 differenti funzioni *Lambda* in AWS per operare sull'istanza di DynamoDB in esecuzione. Sono state scelte le funzioni *Lambda* per la loro scalabilita' rispetto all'aumentare di richieste: inizialmente si e' pensato di interagire con *DynamoDB* senza utilizzare le *Lambda* ma, confrontando le performance, si e' arrivati alla conclusione che utilizzando queste ultime il sistema scalava meglio.

4.5 DynamoDB

Responsabile di fare storage persistente di dati che vengono acceduti scarsamente. Questo viene reso possibile da un thread in ogni nodo Server che si occupa della pulizia dello storage locale, e invia a DynamoDB i valori se la memoria locale utilizzabile scarseggia.

Questa opzione e' attivabile tramite il settaggio a *'true'* del campo *'optionClean'* nel file di configurazione *conf.json* nella directory *./server.*

V. SCELTE PROGETTUALI E IMPLEMENTATIVE

Sono state effettuate scelte di progettazione guidate dai requisiti che il sistema deve soddisfare:

5.1 Consistenza

Essendo un sistema di storage persistente per valori relativi a cluster di sensori di temperature la consistenza tra i nodi distribuiti e' stata considerata importante. Si e' riusciti ad arrivare ad un livello di consistenza *sequenziale:* le operazioni effettuate nelle repliche sono viste nello stesso ordine da tutti (anche se non seguono un ordinamento temporale). Questo e' dovuto all'uso di RAFT per il consenso tra le repliche: il nodo Leader coordina l'insieme delle operazioni di write nello storage distribuito. Tuttavia questa scelta limita le prestazioni e la disponibilita' per quanto riguarda le operazioni di *Delete, Put* e *Append.* Alta disponibilita' e alte prestazioni per le operazioni di *Get* che vengono chiamate in ogni nodo del sistema.

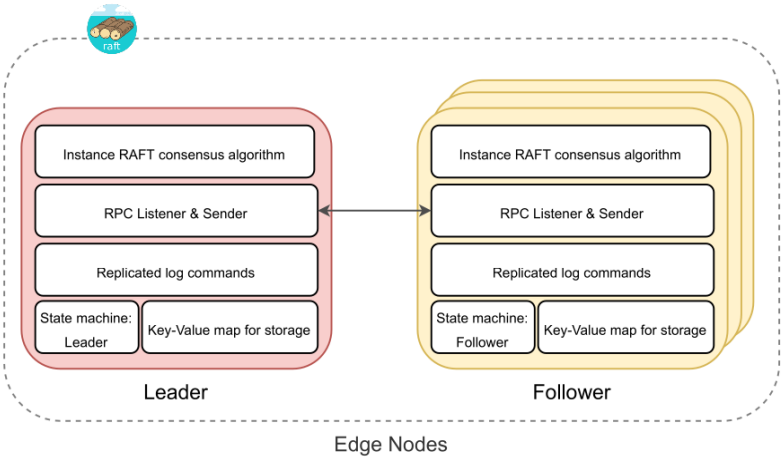

Fig. 3. RAFT: struttura per storage replicato nei nodi edge.

La comunicazione tra i nodi del cluster avviene sempre tramite chiamate RPC e si garantisce semantica degli errori di tipo *at-least-once.*

5.2 Scalabilita'

La scalabilita' e' data dalla replicazione dei dati in ogni nodo del sistema. Scelte progettuali per la consistenza abbassano il grado di scalabilita' per le operazioni di *Put* e *Append*. Viene incrementata anche dall' uso di uno storage Cloud come *DynamoDB* per memorizzare valori scarsamente acceduti, questo garantisce una maggiore disponibilita' dei nodi Server, sempre in grado di memorizzare i valori ricevuti.

Il sistema mantiene nei nodi edge i valori piu' acceduti e cerca di minimizzare le chiamate al Cloud, minimizzando la latenza delle chiamate.

5.3 Tolleranza a guasti

Il sistema e' tollerante a crash dei nodi Server che lo compongono. Il crash di un nodo Server e' preceduto dal salvataggio, in un volume Docker, dello stato del nodo in quel momento. In piu' RAFT e' nativamente tollerante a guasti,

quindi, se il nodo soggetto a guasto e' il nodo Leader, si avvia il meccanismo di elezione implementato in ogni nodo Server.

5.4 Utilizzo del Cloud

Ogni richiesta ricevuta dal Server deve essere anche gestita a livello Cloud.

• ***Get*** : Se un valore richiesto è presente nell'Edge node allora il Cloud non viene contattato. Nel caso in cui l'oggetto è presente nel *cloud* ma non in locale, viene effettuata la *put* del valore sul nodo *Edge*.

• ***Put*** : I valori vengono inseriti sempre nell'*Edge Layer*. Quando la memoria è piena, si eliminano valori locali (di grandi dimensioni o scarsamente acceduti) per trasferirli nel Cloud e liberare spazio d'archiviazione

• ***Append*** : Se un valore è presente sia nell'*Edge* che nel *Cloud* viene aggiornato in entrambi i livelli.

• ***Delete*** : Ogni chiamata di questo tipo elimina valori sia nell'*Edge* che nel *Cloud.*

VI. TEST E RISULTATI

TABLE I. TEST 1

TEST 1	***NUMERO DI RPC TOTALI CON CHIAMATE CLOUD ASINCRONE***				
	1.000	5.000	10.000	25.000	50.000
Latenza **(ms)**	17271	17562	18010	32141	63695

TEST 1	***NUMERO DI RPC TOTALI CON CHIAMATE CLOUD SINCRONE***			
	1.000	5.000	10.000	25.000
Latenza **(ms)**	43367	118786	238942	512235

Fig. 4. TEST1: 85% GET & 15% PUT

TABLE II. TEST 2

TEST 2	***NUMERO DI RPC TOTALI CON CHIAMATE CLOUD ASINCRONE***				
	1.000	5.000	10.000	25.000	50.000
Latenza **(ms)**	18271	23652	42717	72766	120760

TEST 2	***NUMERO DI RPC TOTALI CON CHIAMATE CLOUD SINCRONE***			
	1.000	5.000	10.000	25.000
Latenza **(ms)**	59637	329153	828122	2513395

Fig. 5. TEST1: 40% GET & 40% PUT & 20% APPEND

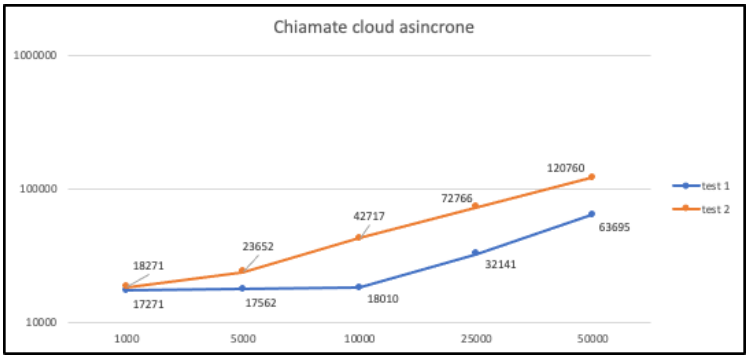

Fig. 6. CHIAMATE CLOUD ASINCRONE: CONFRONTO LATENZA TRA TEST 1 E TEST 2

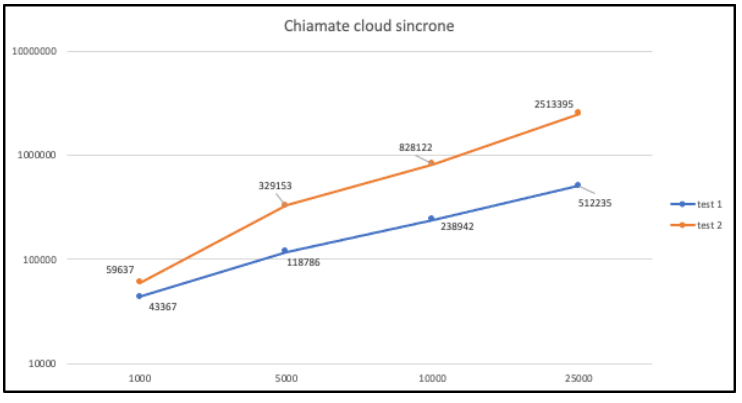

Fig. 7. CHIAMATE CLOUD SINCRONE: CONFRONTO LATENZA TRA TEST 1 E TEST 2

6.1 Considerazioni

Dai test si nota chiaramente che le prestazioni dell'applicazione migliorano notevolmente nel caso in cui le chiamate *Cloud* sono ***asincrone***. A livello di implementazione, infatti, il *main thread* non deve occuparsi della gestione della comunicazione con il *Cloud*, potendosi quindi dedicare alle richieste dei *client*. Nel caso ***sincrono***, invece, il *main thread* rimane in attesa del completamento dell'operazione *Cloud*, lasciando in coda le richieste dei *Client*. Inoltre si vede che l'applicazione scala molto bene in entrambi i casi (non si notano, infatti, significativi aumenti di latenza con l'aumentare della mole di richieste). In generale le uniche operazioni che non riescono a scalare completamente sono quelle di *Append.* Questo si nota specialmente nel test 2 del caso sincrono.

Essendo un'applicazione di storage distribuito orientata su un contesto di Smart City si e' preferito focalizzare l'attenzione sulla consistenza dello storage replicato a discapito di una alta disponibilita' del sistema.

6.2 Miglioramenti

Si e' visto, facendo test, che l'applicazione non riesce a gestire una grande mole di richieste sia per il sovraccarico lato edge sia per le troppe chiamate alle *Lambda Functions.*

Un possibile miglioramento e' inserire uno strato tra i nodi edge implementando un servizio di accodamento di messaggi distribuito (attraverso, ad esempio, SQS di AWS).

Infine, se il contesto dell'applicazione diventa elevato, con molti sensori da gestire, si potrebbe inserire un nuovo livello di storage persistente attraverso S3 per immagazzinare dati di grandi dimensioni e non acceduti.

VII. Architettura per il test

I test sono stati effettuati su una macchina con le seguenti specifiche:

• **SO:** Manjaro Linux (kernel: 5.13.19-2-MANJARO)

• **CPU**: AMD Ryzen 7 3700X (16) @ 3.6GHz

• **RAM**: 16GB DDR4 Synchronous 3200 MHz (0.3 ns)

• **GPU**: NVIDIA GeForce GTX 1660 ti

VIII. Riferimenti

[1]: https://aws.amazon.com/it/dynamodb/

[2]: https://raft.github.io/

[3]: http://www.ce.uniroma2.it/courses/sdcc2021/