LPN: Log Partition Network

Daniele Ferrarelli*, Marco Ferri†, Lorenzo Valeriani‡
Dipartimento di Ingegneria Civile e Ingegneria Informatica
Università degli studi di Roma "Tor Vergata"
Roma, Italia
*daniele.ferrarelli@alumni.uniroma2.eu
†marco.ferri.98@alumni.uniroma2.eu
‡lorenzo.valeriani.459326@alumni.uniroma2.eu

***Abstract*—In questo documento si illustra una soluzione implementativa di uno storage chiave-valore distribuito per edge computing scritta in Go.**

***Index Terms*—sistemi distribuiti, peer-to-peer, edge computing, cloud computing, distributed system, key-value storage**

I. Introduzione

Le reti peer-to-peer rappresentano una soluzione comune per il problema dello storage distribuito. Tra queste, quelle strutturate si prestano particolarmente alle tipologie di storage chiave-valore. Attraverso tecniche come il consistent hashing, si supportano le operazioni di ricerca delle informazioni in modo efficiente L'edge computing è un paradigma di computazione che estende il cloud tradizionale con le funzionalità di computazione e storage offerte da nodi localizzati ai bordi della rete e quindi vicino agli utenti. La soluzione proposta sfrutta una rete peer-to-peer strutturata per indicizzare le risorse distribuite sui vari nodi edge, cercando di trarre beneficio da entrambe le realtà. Il protocollo utilizzato dalla rete peer-to-peer è Kademlia[1], nell'implementazione in Go fornita da IPFS. I dati sono replicati su più nodi affinché il sistema sia tollerante ai guasti. La consistenza tra le repliche dei dati è affidata all'algoritmo di consenso distribuito Raft[2][3]. Il progetto fa largo uso dei servizi cloud forniti da Amazon Web Services, quali RDS, Lambda e DynamoDB. La comunicano tra i nodi e con i client è attuata tramite chiamate gRPC.

II. Topologia

I nodi del sistema sono connessi per formare una rete peer-to-peer strutturata, su questa è presente un'ulteriore serie di collegamenti che costituisce la rete di replicazione dei dati. Quando si inserisce un nuovo dato nel sistema, questo sarà memorizzato almeno nel nodo che ha ricevuto l'operazione di inserimento, per far sì che la latenza di rete tra l'utente ed il dato sia minima. Quest'ultimo nodo sarà il responsabile di quel dato, identificato da una chiave. Oltre a questo, il dato sarà replicato su un insieme di nodi che possono essere contattati in caso di fallimento del responsabile. I vari nodi vengono identificati tramite il loro indirizzo IP.

A. Rete di indexing

Il sistema sfrutta la DHT di Kademlia per associare ad ogni chiave memorizzata l'indirizzo IP del nodo responsabile. Così facendo si mantiene l'efficienza del lookup di una rete strutturata (O(log(N))), permettendo la possibilità di posizionare il dato in un nodo che minimizzi la latenza con il client che vi accede. La rete peer-to-peer memorizza inoltre per ogni nodo il cluster di replicazione a cui appartiene, in caso di crash del nodo responsabile è necessario eseguire un'altra richiesta sulla DHT per cercare una replica attiva del dato. La complessità totale dell'operazione resta O(log(N)). Al fine di supportare questo comportamento, si è realizzata una struttura per incapsulare la libreria fornita da IPFS, estendendo il suo comportamento in modo trasparente alle richieste.
Le operazioni aggiunte sono:

- PutValue e GetValue
- PutCluster e GetCluster

La distinzione tra le operazioni di ricerca chiave-responsabile e responsabile-cluster è eseguita impostando correttamente il primo byte della chiave. Allo stesso modo, in caso di ricerca chiave-responsabile, osservando il primo byte del campo corrispondente al responsabile, è possibile capire se la chiave è presente sulla DHT oppure ha subito offloading sul cloud. I dettagli di quest'ultima operazione sono affrontati nella corrispondente sezione.

B. Rete di replicazione

L'insieme dei nodi del sistema è partizionato in un insieme di cluster di replicazione. I nodi che appartengono allo stesso cluster si replicano vicendevolmente e i dati sono mantenuti consistenti tramite Raft. La formazione dei cluster avviene quando i nodi richiedono l'ingresso nel sistema. Nel caso in cui non ci siano cluster già formati, i nodi aspettano l'ingresso di altri, per essere certi di poter assicurare un determinato livello di resistenza ai guasti, che può essere configurato dall'amministratore. Quando la quantintà di nodi è sufficiente, questi formano un primo cluster. Gli ingressi successivi andranno a far parte di questo cluster, finché la dimensione di quest'ultimo non sarà due volte la dimensione minima di un cluster: in questo caso, metà dei nodi si staccheranno per formare un nuovo cluster.

III. Storage locale

Lo storage locale di ogni nodo è stato implementato come un'interfaccia in grado di esporre le operazioni:

- `Get(key []byte) ([][]byte, error)`

- `Put(key []byte, value [][]byte) error`
- `Append(key []byte, value [][]byte) ([][]byte, error)`
- `Del(key []byte) error`
- `DeleteExcept([]string) error`
- `GetAllKeys() ([]string, []string)`

L'uso di un'interfaccia permette un'astrazione a livello di codice ed una migliore semplicità di espansione del progetto al fine di supportare diverse tecnologie di storage. Attualmente, sono forinte due implementazioni della stessa interfaccia: una che fa uso di BadgerDB, persistente, e l'altra di Redis, in-memory. L'interfaccia che sfrutta Redis, in particolare, si collega ad un indirizzo IP, che può essere esposto da un container. Alcune operazioni, come l'append, richiedono un'esecuzione transazionale dal momento che sfruttano una get seguita da una put. In Redis, sono state sfruttate le condizioni di lock ottimistico secondo le quali una transazione è mandata in abort e fatta ripartire in caso di race condition nei confronti della stessa chiave. Interfaccia DB, Badger, Redis e transazioni

IV. Consistenza e tolleranza ai guasti

Per tolleranza ai guasti, tutti i dati sono replicati all'interno di un cluster. Le varie repliche sono tenute aggiornate tramite Raft. L'algoritmo, prevedendo la presenza di un leader, garantisce l'ordinamento sequenziale delle operazioni secondo l'ordine di programma di questo. Si è tuttavia scelto di gestire le operazioni di get esternamente a Raft: nel caso in cui il dato sia presente sulla replica contattata, quest'ultima lo ritorna immediatamente, anche se potrebbe non essere la versione più aggiornata. Gli errori supportati sono solamente errori temporanei: se un nodo subisce crash, si assume che questo tornerà ad essere disponibile dopo un intervallo finito di tempo. La quantità massima di crash gestitibili in un signolo cluster in contemporanea dipende dalla dimensione minima del cluster configurata: per gestire k errori la dimensione deve essere $2k + 1$. È opportuno notare che, essendo la dimensione di un cluster compresa tra $2k + 1$ e $4k + 1$, i cluster più grandi potrebbero essere in grado di tollerare più di k guasti. I nodi comunicano tra loro e con i client tramite chiamate gRPC. La semantica di errore nelle chiamate a procedura remota è stata resa at-least-once impostando una policy di retry nella configurazione di gRPC.

V. Migrazione

Il sistema supporta la migrazione delle chiavi da un nodo ad un altro. Questa operazione viene richiesta automaticamente in base alla quantità di operazioni richiese da un client ad un nodo nei confronti di una singola chiave. È possibile configurare un peso per le oprerazioni in scrittura, ovvero put e append, e per le operazioni in lettura, ovvero get. Si mantengono in memoria per ogni nodo le operazioni richieste recentemente dai client e verso quali chiavi. Si può configurare per quanto tempo tenere traccia delle operazioni ricevute. Tutte le operazioni ricevute e le chiavi alle quali queste si riferiscono sono memorizzate in una struttura dati ordinata nel tempo, supportata da una slice. Quando si accede a questa struttura, si tiene conto del tempo attuale e si eliminano tutte le operazioni ritenute obsolete. La ricerca dell'indice dell'ultima operazione da rimuovere, essendo la struttura dati ordinata, avviene tramite ricerca binaria. Una volta trovata questa si rimuovono tutte le operazioni precedenti. Una singola goroutine gestisce l'accesso a questa struttura per evitare la necessità di sincronizzazione e, allo scadere di un intervallo configurabile, la scorre interamente, valutando per ogni chiave la quantità di operazioni che la riguardano ed il costo di queste. Nel caso in cui la somma dei costi per ogni operazione di una chiave superi una soglia configurabile, il nodo richiede la mirazione al responsabile della chiave. Il responsabile, dopo aver ricevuto la richiesta di migrazione, valuta i costi delle operazioni da lui ricevute per la stessa chiave ed in caso questi siano inferiori migra la chiave verso il nuovo responsabile. Sliding window, single thread che valuta e ricerca binaria

VI. Integrazione con il cloud

La soluzione presentata fa largo uso dei servizi Cloud offerti da AWS.
Le funzionalità implementate con il supporto Cloud sono:

- Servizio di registrazione
- Offloading dei dati di grandi dimensioni
- Servizio client per ottenere gli ip dei nodi edge

In generale si è preferito un approccio Serverless utilizzando Lambda piuttosto che istanze EC2. Questa scelta è stata dettata dalla volontà di garantire tutti i vantaggi del Serverless. In particolare data la necessità di utilizzare questo servizio solo in fase di registrazione si è preferito un servizio pay-per-use piuttosto che mantenere accesa un'istanza EC2 che potrebbe ricevere poche richieste.
Al livello teorico inoltre EC2 potrebbe rappresentare un Single Point of Failure nonchè collo di bottiglia in caso di più richieste concorrenti. Per ovviare a ciò sarebbe stato necessario generare più instanze con un load balancer. Le lambda permettono invece di bypassare questi problemi con l'unico svantaggio il Cold Start, ma diventa trascurabile dal momento in cui ci si aspettano tante invocazioni.

A. Servizio di registrazione

Questo servizio gestisce i cluster di replicazione. Tiene conto dell'eventuale generazione di nuovi cluster, dell'eventuale crash temporaneo dei nodi edge per mantenere i cluster originali ed il caso in cui non ci sono abbastanza nodi per generare il primo cluster. Tutto ciò viene fatto in maniera completamente trasparente al nodo edge che riceve semplicemente la lista degli ip del suo cluster.
Tutti i dati vengono mantenuti in un'instanza RDS, le query vengono effettuate tramite invocazioni a funzioni Lambda implementate in python . Per ragioni di sicurezza è stata generata una VPC in cui vive l'instanza RDS.

B. Offloading

Data l'ipotesi di una rete formata da nodi edge, ossia nodi con limitata capacità computazionale e di memoria ed il contesto di uno storage chiave valore si è reso necessario

mantenere valori con dimensioni particolarmente eleveate.
E' stato utilizzato quindi il servizio di DynamoDB che è uno storage chiave valore per mantenerle nel Cloud. La principale limitazione di questa soluzione è che Dynamo ha una dimensione limite per i valori da salvare. Essa può essere bypassata utilizzando il servizio di storage S3 oppure utilizzare la combinazione di entrambe memorizzando in Dynamo i riferimenti a oggetti S3.

C. Servizio Client

Il client per contattare il nodo edge con latenza minore ha necessità di conoscere gli ip di tali nodi per applicare un metodo di verifica della latenza. Questa necessità è stata ovviata tramite la realizzazione di un supporto Serverless. In particolare i servizi utilizzati sono:

- RDS: instanza di Database relazionale che mantiene tutti i metadati dei nodi edge e dei loro cluster.
- Lambda: servizio serverless per fare query all'istanza RDS
- Gateway: non si ritiene che il client abbia necessità delle credenziali AWS per poter invocare servizi. Per poter invocare la Lambda è stato realizzato un Gateway aperto che ovvia a questo problema.
  Un possibile upgrade

RDS, Lambda, Gateway, DynamoDB con offloading e registrazione

VII. Limitazioni

Attualmente, l'unico servizio di storage cloud supportato è DynamoDB, pertanto la grandezza massima supportata per un valore è 400KB[**dynamo**].
L'operazione di leave Max size dynamo, leave non ancora supportata, join e leave costose lineare per il numero di chiavi e per il numero di nodi(rete strutturata), delete sulla dht non implementata. Non è stato gestito il caso in cui un nodo abbia storage pieno

VIII. Sviluppi futuri

Eventualmente far passare i nodi edge sul Gateway con autentiazione Cognito SSS, possibile implementazione della leave, fantascientificamente network proximity e vivaldi. Offloading che tenga conto anche di richieste

IX. Testing

I test sono stati eseguiti su una rete locale(?) utilizzando 10 nodi. Per misurare le performance per le varie operazioni eseguite si è utilizzato una serie di thread che fanno operazioni sul sistema. All'aumentare del numero di thread concorrenti si misura la latenza per le varie operazioni supportate. Su 50 chiavi in concorrenza. Configurazione 1 Offloading a 4096 test generati con 15% put e 85% get 0-5000 Configurazione 2 // 0-4096 Configurazione 3 Offloading a 4096 test generati con 40% put 20% append 40% get 0-5000 Configurazione 4 all'aumentare del numero di chiavi I test sono stati eseguiti generando traffico e misurando con nodi in rete locale su un subset di chiavi comuni Thread connessi ognuno ad un singolo nodo per simulare lo use case effettivo Come abbiamo fatto i test ed i risultati

A. Risultato delle fasi di test

Tempo medio(?) di get locale è di circa 7 ms, in remoto 110 ms
Tempo medio(?) di put locale è di circa 30 ms, in remoto 130 ms

X. Manuale d'uso

Dettagli su docker, porta utilizzata 50051,42424

A. Installazione

Utilizzo dei docker

B. Configurazione

File di configurazione

C. Esecuzione

Eseguire come docker che come non docker, all'inizio aspetta di avere una numero di nodi sufficiente

References

[1] David Mazières Petar Maymounkov. *Kademlia: A Peer-to-peer Information System Based on the XOR Metric*. URL: https : / / pdos . csail . mit . edu / ~petar / papers / maymounkov-kademlia-lncs.pdf.

[2] ThetaRangers. *The Raft Consensus Algorithm*. URL: https://raft.github.io.

[3] Hashicorp. *Raft*. URL: https://github.com/hashicorp/raft.