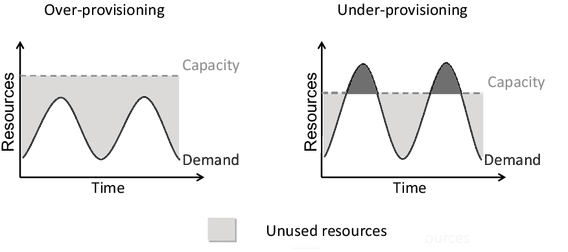
SDCC Progetto A1 - JDSys

Sistema di Storage Distribuito di tipo Chiave Valore

Danilo Dell’Orco  
 Facoltà di Ingegneria  
 Università di Roma Tor Vergata  
 Roma, Lazio, Italia  
 danilodellorcolp@gmail.com

Jacopo Fabi  
 Facoltà di Ingegneria  
 Università di Roma Tor Vergata  
 Roma, Lazio, Italia  
 jacopo.fabi.1997@gmail.com

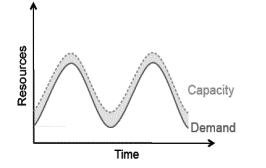


ABSTRACT

JDSys è un sistema di storage distribuito di tipo Chiave valore, in cui i nodi che gestiscono i dati sono localizzati su server differenti. L’obiettivo di JDSys è quello di fornire un servizio ad alta disponibilità ed affidabilità, in cui il numero di risorse utilizzate per il provisioning viene stabilito con un approccio elastico per massimizzare l’efficienza sia in termini di costi che in termini di lavoro. In un sistema di questo tipo, bisogna gestire correttamente il bilanciamento del carico tra i vari nodi, il monitoraggio delle risorse, e la capacità di configurare nuovi server all’occorrenza; a tale scopo, JDSys utilizza a diversi livelli i servizi Amazon Web Services, basando quindi la sua architettura su risorse cloud scalabili, affidabili ed efficienti. Per migliorare la scalabilità rispetto al numero di utenti e di dati, questi oltre ad essere distribuiti, vengono replicati su nodi differenti, e migrati sul cloud storage S3 nel caso in cui siano scarsamente acceduti.

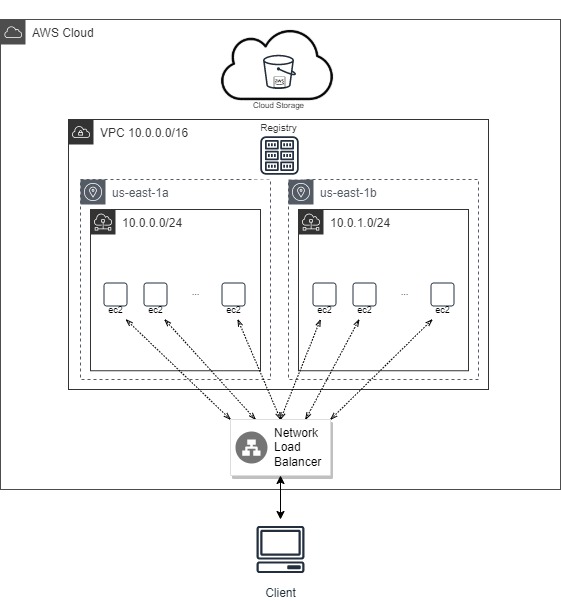
1. INTRODUZIONE

Nel corso degli anni il traffico internet è cresciuto a dismisura. Ciò significa che un qualsiasi sistema che riceve richieste tramite la rete deve gestire una quantità di dati e di operazioni sempre maggiori. In un simile scenario è importante mantenere lo stesso livello di prestazioni ed affidabilità, a prescindere dal carico di lavoro in ingresso. Gli approcci tradizionali consistono nello di scalare verticalmente l’hardware dei server. Tuttavia questo approccio ha un costo esponenziale, ed ha quindi un upper bound al livello di prestazioni raggiungibili. Un altro problema da affrontare è quello del dimensionamento della propria infrastruttura, ovvero quante risorse allocare per fornire il servizio. In un approccio classico si potranno allocare soltanto un numero fisso di risorse durante l’esecuzione del servizio. Ciò vuol dire che, a fronte di un numero di richieste variabile, si potranno avere due scenari. Nell’approccio di under provisioning si allocano un numero di risorse che non riescono a far fronte a picchi di carico. Quindi si gestisce normalmente un carico medio, ma durante I picchi il servizio non sarà disponibilile. Nell’ over provisioning invece si forniscono un numero grande di risorse, in modo da poter gestire anche i picchi di carico. Questo garantisce che il sistema sia sempre disponibile, ma ci sarà anche un numero importante di sprechi in quanto bisognerà avere un'infrastruttura robusta anche durante il traffico medio.

Sfruttando invece un’infrastruttura cloud, è possibile offrire un servizio elastico, che permette di allocare risorse in base alla quantità di traffico che bisogna gestire. Questo approccio permette di minimizzare i costi, offrendo al contempo un servizio sempre disponibile.

L’altro vantaggio offerto dall’utilizzo dell’infrastruttura cloud è quello della scalabilità, poiché si può scalare teoricamente fino ad un numero infinito di macchine sfruttando la scalabilità orizzontale anziché quella verticale.

JDSys è un servizio di storage distribuito di tipo chiave-valore. Questo può ricevere richieste concorrenti, ed un traffico variabile. Per i motivi precedentemente descritti, JDSys utilizza Amazon Web Services a diversi livelli. Le istanze EC2 hostano i singoli data server, il Load Balancer garantisce una suddivisione equa delle richieste, mentre L’Auto Scaling permette di avere sempre il corretto numero di risorse attive in base al traffico in ingresso al sistema. Per garantire una ricerca efficiente della chiave all’interno del sistema, i diversi server sono organizzati in una overlay network ad anello (chord). Inoltre, per avere una maggirore scalabilità e disponibilità, i nodi implementano delle tecniche per la replicazione e riconciliazione dei dati.

Questo documento è strutturato come segue. La sezione 2 descrive il modello di dati utilizzato. La sezione 3 presenta la panoramica delle API client. La sezione 4 presenta la struttura e la progettazione del sistema. La sezione 5 descrive nel dettaglio i principali aspetti implementativi. La sezione 6 descrive i test effettuati su JDSys. La sezione 7 conclude il documento, con le considerazioni finali sul sistema.

2. COMPONENTI

JDSys basa lo storage sul concetto di **entry**. Una entry è coppia chiave-valore. La Chiave rappresenta un identificativo univoco per l’entry, mentre possono esserci anche stessi valori associati a chiavi diverse. La chiave è rappresentata tramite una stringa, mentre il valore è un array di stringhe in quanto può contenere un numero variabile di argomenti.

Il sistema di storage è distribuito su diversi server denominati Nodi. I nodi sono strutturati in una overlay network ad anello, chiamato JDSys Ring.

Il Service Registry è un componente centralizzato che permette la scoperta reciproca tra i nodi. Questo comunica con il load balancer, e periodicamente aggiorna la lista dei nodi healthy, che quindi faranno parte del JDSys Ring.

3. CLIENT API

JDSys permette all’utente di aggiungere, rimuovere, leggere o modificare le entry tramite quattro operazioni: get(), put(), delete() e append():

* *get(key)* ricerca la chiave all’interno del sistema e, se questa esiste, restituisce l’entry ad essa associata.
* *put(key,value)* inserisce l’ entry specificata tramite chiave-valore all’interno del sistema. Se la chiave già esiste, l’entry associata verrà aggiornata con il nuovo valore specificato.
* *delete(key)* elimina dal sistema l’ entry associata alla key specificata
* *append(key,arg1)* modifica l’entry specificata tramite la chiave key, aggiungendo al suo valore l’argomento arg1

4. ARCHITETTURA DI SISTEMA

Un sistema di storage distribuito, oltre alla persistenza dei dati, deve offrire anche soluzioni scalabili e robuste per il bilanciamento del carico, il provisioning delle risorse, la sincronizzazione delle repliche, la gestione del sovraccarico, il trasferimento dello stato e la concorrenza, l'assegnazione delle richieste e l'instradamento delle richieste, il monitoraggio e gli allarmi del sistema. JDSys, fornisce una soluzione ad ognuno di questi aspetti, ed in questa sezione andiamo quindi a descrive nel dettaglio come sono stati affrontati.

Parleremo quindi in particolare della topologia di rete utilizzata, dei meccanismi di comunicazione, dei protocolli utilizzati, e per ognuno di questi punti verranno quindi descritti i principali aspetti implementativi.

4.1 Infrastruttura di Rete

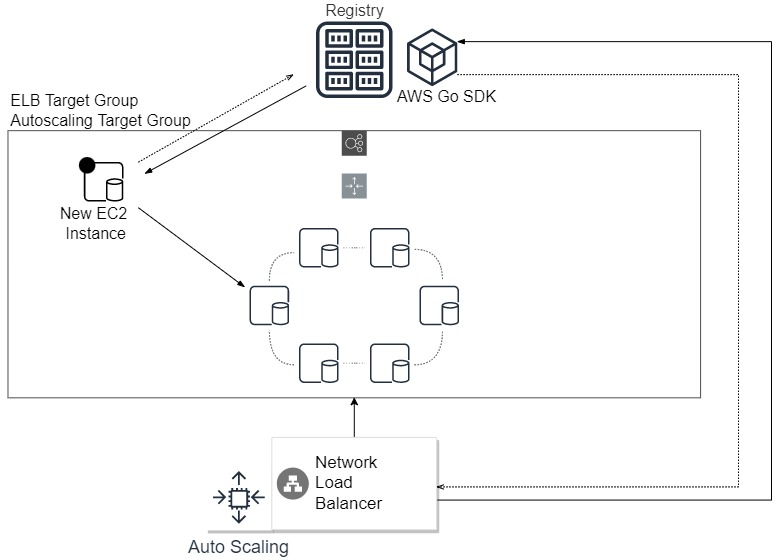
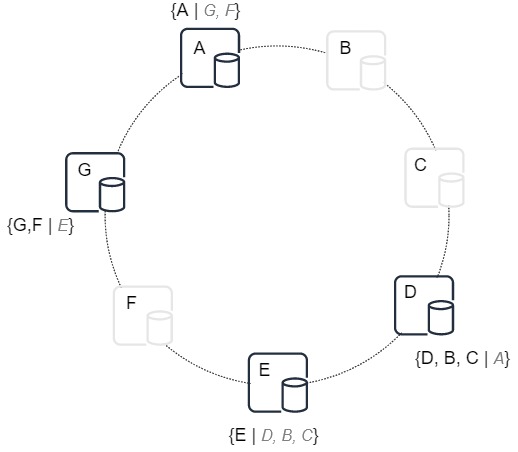
L’intera infrastruttura di rete è stata definita sfruttando Amazon Web Service. Ciò ha permesso di avere un sistema distribuito I cui nodi dell’applicazione sono hostati sul cloud.

La Figura 1 mostra l’infrastruttura di rete di JDSys. I nodi del sistema sono hostati su delle istanze Amazon EC2 micro, che forniscono delle limitate capacità computazionali e di storage.

Risulta fondamentale in questo senso distribuire equamente il carico di lavoro tra questi nodi. A tale scopo, si utilizza l’ Elastic Load Balancer di AWS, che permette di distribuire le richieste in ingresso verso le istanze registrate nel Target Group. Un client dovrà quindi contattare il load balancer tramite il suo URL DNS, e la sua richiesta verrà redirezionata in automatico su uno dei nodi del Ring JDSys. Figura 1: Infrastruttura di Rete **JDSys**

Per motivi di sicurezza ed efficienza, tutte le istanze EC2 utilizzate sono state istanziate all’interno di un Virtual Private Cloud. In questo modo i nodi possono essere contattati dal Load Balancer, e possono comunicare tra loro tramite indirizzi IP privati, ma non potranno essere contattati direttamente dall’esterno.

Per simulare in maniera più fedele una latenza di rete, i nodi vengono istanziati in due zone di disponibilità differenti, che descrivono quindi due sottoreti IP differenti (10.0.0.0/24 e 10.0.0.1/24). Il service registry si trova anch’esso all’interno della VPC. A questa istanza è stato assegnato un IP Elastico, in modo che tutti i nodi possano contattarlo con successo.

Fuori dal VPC troviamo invece un Bucket S3, che fornisce lo storage Cloud necessario per una maggiore scalabilità ai dati. I nodi interagiscono con S3 utilizzando le SDK di AWS, sfruttando quindi la rete internet pubblica in questo caso.

4.2 Autoscaling & Cloudwatch

Un aspetto fondamentale di JDSys è quello della scalabilità automatica. E’ stato utilizzato a tale scopo il sistema di Autoscaling fornito da AWS, che permette di definire un numero minimo e massimo di istanze da istanziare in modalità on-premise. In JDSys sono state utilizzate un minimo di 3 istanze attive, fino ad un massimo di 20. Si noti come il parametro massimo è impostato ad un valore “basso” in virtù delle limitazioni fornite da AWS Educate.

Per monitorare il traffico di richieste verso il sistema, si utilizza il servizio Cloudwatch. La metrica utilizzata è il numero di pacchetti in ingresso al gruppo di autoscaling, in quanto fornisce un indice indicativo sul numero di richieste e quindi anche sul numero di operazioni che ogni nodo dovrà gestire complessivamente.

Analizzando i dati forniti da questo servizio, sono stati ricavati i valori critici in termini di traffico. Si è osservato infatti che per un traffico medio di 30000 pacchetti, i nodi possono andare in saturazione (out of memory) non riuscendo più a gestire un numero elevato di richieste concorrenti. Analogamente si è osservato che per traffico medio sotto le 8000 richieste, il sistema garantisce sempre la normale operatività.

Sono stati quindi definiti due allarmi, uno per lo Scale In ed uno per lo Scale Out, proprio sfruttando i dati ricavati in precedenza. Lo Scale Out consiste nella rimozione di un’istanza dal gruppo di autoscaling, e viene attivato quando il traffico medio in ingresso risulta superiore a 10000 pacchetti per più di 5 minuti. Si è deciso di mantenere questa soglia pari ad 1/3 di quella critica in modo da fornire al servizio di autoscaling il tempo necessario ad istanziare la macchina e a rendere UP & Running il JDSys Node.

4.3 Overlay Network & Partitioning

Uno dei requisiti di progettazione chiave per JDSys è che deve scalare proporzionalmente al numero di richieste in ingresso al sistema. Ciò richiede un meccanismo per partizionare dinamicamente i dati sull'insieme di nodi, ed avere al contempo un metodo efficace di lookup su queste risorse distribuite.

A tale scopo, JDSys utilizza l’algoritmo Chord. Ai nodi e alle chiavi viene assegnato un identificatore utilizzando il Consistent Hashing. Utilizzando il protocollo di lookup di Chord, i nodi e le chiavi sono disposti in una struttura ad anello, in cui ogni nodo ha un successore e un predecessore. Il successore di un nodo è il nodo successivo nel cerchio identificativo in senso orario. Il predecessore è in senso antiorario. Il concetto di successore viene utilizzato anche per le identificare il nodo che deve gestire una determinata entry. Il nodo successore di una chiave k è il primo nodo il cui ID è uguale a k o segue k all’interno dell’anello. Ogni chiave è assegnata al suo nodo successore, per cui la ricerca di una chiave consiste nel fare una richiesta verso succ(k).

Per evitare la ricerca lineare sopra, Chord implementa un metodo di ricerca più veloce richiedendo a ciascun nodo di mantenere una finger table. Per il nodo n, l'entry i-esima di questa tabella mantiene l'indirizzo IP del nodo n+2^{i-1})mod2^m. Ogni volta che un nodo vuole cercare una chiave k, passerà la query al successore o predecessore più vicino alla chiave k, basandosi sulla sua Finger Table. L'idea è quindi quella per cui un nodo conosca bene i nodi nelle sue vicinanze, ed abbia solo una conoscenza approssimativa dei nodi più lontani. Questo algoritmo di lookup, rende la ricerca estremamente più efficiente, in quanto il numero di nodi che devono essere contattati per trovare un successore in una rete di N nodi è O(log(N))

Un altro vantaggio offerto da questo algoritmo, è che l’ingresso (leave) o l'arrivo (join) di un nodo, modifica solo le finger table dai suoi vicini immediati mentre gli altri nodi rimangono inalterati. Questo risulta fondamentale nell’ architettura elastica di JDSys in cui le risorse vengono allocate e deallocate dinamicamentesdf

Un nodo, all’ingresso nel sistema, contatterà il service registry che gli fornirà una lista di istanze attive. Il nodo contatterà quindi un’istanza casuale per inserirsi nell’anello ed aggiorna le sue informazioni su successore e predecessore. Per inizializzare il suo storage locale, il nuovo nodo chiederà al predecessore di inviargli le sue entry, diventando quindi il gestore delle chiavi secondo il Consistent Hashing.

Al leave, invece il nodo contatterà il suo successore informandolo della terminazione. A questo invierà tutte le sue entry e conseguentemente verranno aggiornate tutte le informazioni delle finger table

4.4 Replicazione

Per ottenere **una maggiore** disponibilità e **scalabilità**, **JDSys** replica i propri dati su più **nodi**. Ogni nodo JDSys sarà responsabile di un certo sottoinsieme di entry, ed allo stesso tempo si occuperà di replicare tali entry sul suo nodo successore.

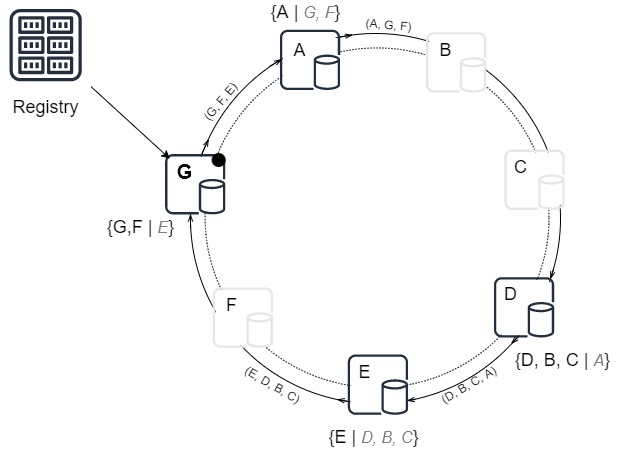
Il successore secondo chord di una chiave k svolge il ruolo di gestore di tale chiave. Il gestore è responsabile quindi, oltre che della memorizzazione di tali dati, anche della loro replicazione sul nodo successivo. Nella Figura ??, il nodo A replica la chiave A di cui è gestore sul nodo D. Il nodo D replica le chiavi B,C,D sul nodo E. Il nodo E replica la chiave E di cui è gestore sul nodo G. Il nodo G replica le chiavi G ed F sul nodo A.

La replicazione avviene non appena si riceve una nuova entry o un aggiornamento su di essa. Viene immediatamente contattato il nodo successore inviandogli la replica.

Sfruttando questo meccanismo, migliora inoltre la consistenza ai guasti, in quanto al crash di un nodo, le sue informazioni non andranno perse ma saranno presenti su almeno un altro nodo. Viene quindi demandata a Chord la ricostruzione dell’anello e quindi la riorganizzazione dei gestori delle varie chiavi.

4.5 Consistenza

JDSys offre un modello di consistenza finale; ciò vuol dire che per alcuni lassi di tempo le repliche su diversi nodi possono essere inconsistenti tra loro, e la consistenza verrà raggiunta soltanto in un secondo momento. Quando un nodo riceve una richiesta di scrittura o aggiornamento, risponderà immediatamente al chiamante, e parallelamente procederà ad inviare la replica (o aggiornarla) presso il suo successore. Una chiamata put() può quindi fornire la risposta al suo chiamante prima ancora che l'aggiornamento sia stato effettuato ed applicato a tutte le repliche; ciò vuol dire che possono verificarsi scenari in cui un'operazione di get() può restituire un oggetto che non dispone degli aggiornamenti più recenti, in quanto l’aggiornamento deve ancora essere propagato.

*4.4.1 Riconciliazione*

Per risolvere i conflitti tra tutte le repliche, giungere così alla consistenza finale, si utilizza un algoritmo di riconciliazione di tipo Asyncrhonous Repair. Periodicamente il service registry avvia il servizio di riconciliazione, contattando un nodo a caso tra quelli dell’anello. Questo nodo sarà il gestore della riconciliazione, ed invierà le proprie entry al nodo successivo. Il nodo che riceve questo messaggio, utilizzerà l’aggiornamento ricevuto per riconciliare le sue repliche, e successivamente invierà a propria volta le sue entry al suo successore.

Per risolvere un conflitto tra due entry con la stessa chiave, si utilizza la tecnica Last Write Wins, mantenendo quindi la copia più recentemente aggiornata. In tal senso, ad ogni scrittura su una entry si aggiorna il suo timestamp, sfruttando il Network Time Protocol che garantisce un’accuratezza di 100ms.

Questo processo di riconciliazione, per avere la certezza di risolvere tutte le copie conflittuali, necessita al più di due giri dell’anello. Questo perché può verificarsi lo scenario in cui, per una chiave k, la copia aggiornata sia in possesso del predecessore del gestore della replicazione, e quindi con un solo giro nessun nodo riconcilierà la sua copia di k.

Per quanto riguarda invece le operazioni di delete(), si è utilizzata una semantica un po' più forte. Questo perché con il meccanismo di replicazione, può verificarsi lo scenario in cui delle copie eliminate “tornino in vita”. Per evitare questo, al momento dell’eliminazione di un’entry, viene immediatamente propagata la richiesta su tutto l’anello e non solo verso il successore. Questo perché a seguito del join o del leave, i nodi possono avere delle repliche fuori dalla portata del gestore. Se queste verranno comunque aggiornate grazie alla riconciliazione, ciò non sarà vero in caso di cancellazione.

4.6 Cloud Storage

Per migliorare la scalabilità ai dati, JDSys integra i propriservizi con un Bucket S3. Questo bucket è utilizzato globalmente da tutti i nodi JDSys per migrare le entry che sono scarsamente accedute. Tutte le entry non accedute da almeno 30 minuti (sia in lettura che in scrittura) verranno rimosse dallo storage locale e verranno caricate sullo storage S3. Ogni nodo mantiene quindi traccia delle entry presenti sul cloud, e alla prima richiesta ricevuta per una di quelle entry, procederà a scaricarla nuovamente da S3 ed inserirla nel suo storage locale.

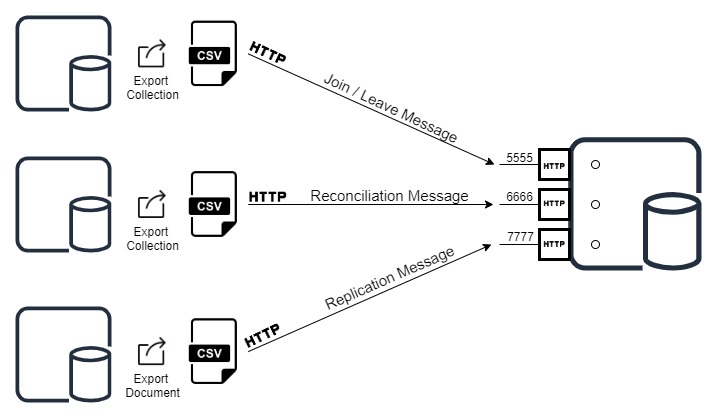
4.7 Semantica di Errore

La richiesta prima di giungere al nodo corretto, deve passare per il load balancer e per un certo numero di nodi. In questo scenario possono verificarsi errori nella comunicazione di rete, che possono portare quindi alla perdita di una richiesta. Alcuni di questi problemi possono essere tuttavia solo temporanei rispetto che a veri e propri guasti, ad esempio durante la riconfigurazione della struttura dei nodi. In questo scenario è fondamentale non perdere queste richieste, e a tale scopo JDSys utilizza la semantica at least once per le richieste dal client al server. Questa utilizza il meccanismo RR1, basato su un semplice timer che parte insieme all’invio della richiesta; allo scadere del timer, il messaggio si considera perso e si procede con una ritrasmissione. Tuttavia, è possibile che la comunicazione di rete sia effettivamente interrotta, e per questo sono previsti un numero massimo di tentativi al termine del quale verrà mostrato un messaggio di errore e si considererà la chiamata effettivamente fallita.

5. IMPLEMENTAZIONE

JDSys è sviluppato utilizzato il linguaggio Go. Il vantaggio principale è dovuto alla gestione efficiente della concorrenza grazie alle goroutine ed ai canali. Inoltre permette facilmente di utilizzare le Remote Procedure Call, che semplificano di molto l’interazione client-server o nodo-nodo.

JDSys, è composto da quattro componenti fondamentali

* Engine di Storage Locale
* Gestore dell’anello Chord
* Listener dei messaggi HTTP
* Listener delle chiamate RPC

5.1 Storage Locale

JDSys utilizza MongoDB per lo storage sul singolo nodo. La scelta è ricaduta su questo engine in quanto fornisce prestazioni ottimali, una buona scalabilità, ed una gestione automatica della concorrenza (che discuteremo più avanti). Inoltre, MongoDB offre lo strumento mongoexport, particolarmente utile per l’invio delle entry tra i diversi nodi. Per l’interazione tra mongo ed il software del nodo, è stata utilizzata la libreria mongo-driver.

Per ogni nodo è stato utilizzato utilizzato un singolo database (sdcc-local-sys) ed una singola collezione (sdcc-local-storage). Ogni entry è associata ad un documento MongoDB, salvato in formato BSON. Prima di inserire, modificare o leggere un documento, questo viene serializzato in formato binario sfruttando bson.D.

In JDSys è necessario che un nodo possa inviare e ricevere delle copie di alcune entry. Per questo si utilizza la funzione mongoexport che permette di esportare una collezione o un sottoinsieme di documenti in formato csv. Un nodo che riceve un aggiornamento potrà procedere con due tipi di operazioni: merge o reconciliate. Il merge consiste nella risoluzione dei conflitti in modalità last write wins, e nel’inserimento nello storage locale di tutte le nuove entry ricevute. Un caso d’uso tipico è quando un nodo si inserisce o esce dall’anello, ed il controllo delle entry dovrà passare ad un altro nodo. La riconciliazione invece si concentra solo sulla risoluzione dei conflitti; quindi il nodo locale che riceve un documento non presente nel suo db, semplicemente ignorerà tale entry.

Per risolvere i conflitti, si utilizza un confronto sull’ultima scrittura effettuata. A tale scopo, ogni documento include oltre ai campi key e value, anche un campo timest. Dati quindi due documenti aventi stessa chiave, si confrontano i timestamp, mantenendo soltanto l’entry più recente

Questo valore viene assegnato o aggiornato ad ogni operazione di scrittura tramite il Network Time Protocol. A tale scopo si è utilizzata la libreria beevik-ntp, ottenendo il tempo corrente tramite il server remoto NTP *0.beevik-ntp.pool.ntp.org*. La scelta di utilizzare questa soluzione è dovuta ad una maggiore semplicità di gestione ed implementazione rispetto ad un degrado di prestazioni minimale.

Le operazioni di effettuate dal client si tradurranno sempre in query eseguite sullo storage mongo locale. L’operazione di get viene effettuata tramite FindOne(). Questa scrive il risultato all’interno di un oggetto bson, Successivamente accedendo ai campi di questo oggetto si istanzia una struttura di tipo MongoEntry, che verrà restituita al chiamante in caso di successo. Se l’entry invece non è stata trovata, viene restituito un messaggio di errorre. Le operazioni di Put si effettuano tramite InsertOne(). Nel caso in cui l’entry già esiste, verrà ritornato l’errore E11000, ed in tal caso questa verrà aggiornata eseguendo UpdateOne(). L’operazione di append corrisponde ad eseguire una UpdateOne(). L’operazione di delete viene effettuata tramite DeleteOne(). Ogni operazione restituisce l’errore specifico riscsontrato

5.2 Gestione dell’Anello

La gestione dell’anello è fatta da diversi componenti. Le principali librerie utilizzate sono AWS-SDK e Chord.

*5.2.2 Service Registry*

Il service registry svolge un ruolo fondamentale in quanto permette di mantenere la lista di istanze attive nel target group e permette di intercettare tutte le attività di scale-in e scale-out dell’autoscaling, particolarmente utili quando bisogna identificare le istanze in fase di terminazione. Per fare ciò, il service registry utilizza le sdk aws per go. In particolare per ottenere la lista di istanze attive contatta il load balancer tramite il suo codice ARN, ed effettua il parsing del messaggio di risposta per ottenere la lista di healthy instances. Per ognuna di queste viene istanziato un oggetto InstanceEC2 che tiene traccia dell’ID dell’istanza e del suo indirizzo IP Privato.

Per ottenere invece la lista di istanze in fase di terminazione viene contattato tramite ARN l’autoscaling e si ottiene la lista delle attività di scaling. Da queste si ricava la lista di istanze in terminazione filtrando il messaggio “terminating” o “WaitingForELBDraining”. Per ognuno di questi nodi viene istanziata una struttura InstanceEC2.

*5.2.2 Chord*

[JACOPO] Per mantenere la struttura ad anello, si utilizza il protocollo chord. La sua implementazione è stata effettuata tramite la libreria XXX Chord.

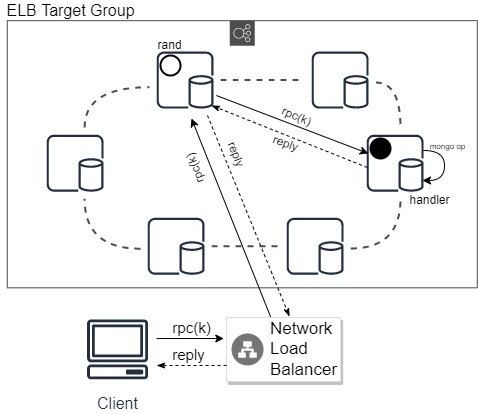
5.3 Messaggi

Per implementare i servizi di Riconciliazione, Replicazione, Join e Leave, i nodi utilizzano dei particolari messaggi di aggiornamento in formato csv. L’invio di questi file csv avviene sfruttando HTTP. Ogni nodo è in ascolto per questi messaggi su porte differenti, in modo da poter discriminare un comportamento diverso su porte differenti.

*5.3.1 Messaggi di Join e Leave*

Un nodo per effettuare il Join nell’anello deve richiedere la lista delle entry al suo predecessore, mentre per effettuare il Leave deve inviare la lista delle sue entry al successore. Un messaggio di Join/Leave consiste quindi nell’export csv di una intera collezione, che verrà poi utilizzata dal nodo ricevente per aggiornare la sua lista di entry. Ogni nodo è quindi in ascolto per questo servizio sulla porta 5555. Quando si riceve un messaggio su questa porta, viene effettuato il merge del csv ricevuto con il database locale, diventando così il gestore delle nuove entry e risolvendo eventuali conflitti.

*5.3.2 Messaggi di Riconciliazione*

Il nodo è in ascolto sulla porta 6666 per i messaggi di riconciliazione. Quando si riceve un csv da questa porta, vengono riconciliate tutte le repliche locali, mantenendo solo le copie più aggiornate per ogni entry. Successivamente, se il nodo non è il gestore e se non sono stati già due giri dell’anello, si effettua l’export csv dello storage locale, e lo si invia come messaggio di riconciliazione al nodo successore.

*5.3.3 Messaggi di Replicazione*

Quando un nodo effettua il Put o Append, oltre a inserire/aggiornare l’entry locale, è anche responsabile della replicazione di questa entry presso il successore. Per fare ciò si esporta in formato csv il documento appena modificato, e lo si invia sulla porta 7777 al successore. Ogni nodo è quindi in ascolto sulla porta 7777 per i messaggi di replicazione. Quando si riceve un csv da questa porta, vengono aggiornate tutte le repliche locali, mantenendo solo le copie con timestamp più recente per ogni entry.

5.4 Remote Procedure Call

Per permettere al client di interagire col sistema, e per permettere ai nodi l’esecuzione di particolari servizi, si utilizza il meccanismo delle Remote Procedure Call. Queste sono utilizzate tramite la libreria standard di go net/rpc. Ogni espone diverse RPC, che si differenziano in base al loro utilizzo. In particolare alcuni metodi verranno invocati dal client, altri verranno invocati da altri nodi, ed altri ancora dal service registry.

*5.4.1 Client RPC*

I metodi remoti esposti da un nodo verso il client sono GetRPC, PutRPC, AppendRPC e DeleteRPC. Ogni operazione del client, consiste quindi in una chiamata RPC, che viene inviata sempre al Load Balancer. Il Load Balancer seguendo la sua politica di bilanciamento, si occuperà poi di propagare la chiamata verso il nodo da lui scelto. Il nodo riceverà quindi una chiamata relativa ad una certa chiave k, e dovrà verificare innanzitutto chi è il gestore di quella chiave. La prima operazione eseguita è quindi il Lookup di chord, cercando il successore della chiave k. Se il successore è il nodo stesso, questo esegue l’operazione richiesta in locale, e risponde al chiamante (quindi il load balancer). Se invece il successore è un nodo remoto, verrà inoltrata l’RPC verso quel nodo che sarà sicuramente il gestore di tale chiave. Il nodo gestore esegue quindi in locale l’operazione richiesta, e risponde al nodo chiamante. Sarà quindi sempre il nodo scelto dal load balancer che risponderà in ultima istanza al client.

*5.4.1.1 GetRPC*

Un nodo che riceve una richiesta di get dal load balancer, verifica innanzitutto se quella entry è presente nel suo storage locale. Questo perché in virtù della replicazione potrebbe avere una copia di quella risorsa anche se non è il suo gestore. Se la chiave non viene trovata su mongodb, viene effettuato il lookup e la RPC viene inoltrata al nodo che deve gestire quella chiave.

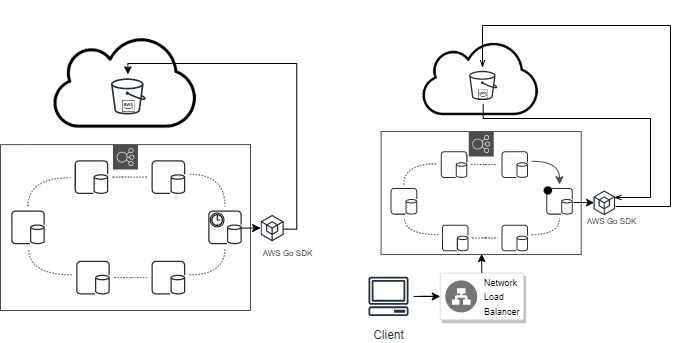
*5.4.1.2 PutRPC e AppendRPC*

Un nodo che riceve la richiesta di Put o Append, effettua il lookup per vedere chi deve gestire quella chiave. La richiesta viene subito inoltrata al nodo gestore, e una volta eseguito l’inserimento/ modifica si invia la risposta al chiamante. In parallelo, viene inviato un messaggio di replicazione al successore, contenente l’entry appena aggiornata.

*5.4.1.3 DeleteRPC*

Un nodo che riceve la richiesta di Delete, effettua il lookup per vedere chi deve gestire quella chiave. La richiesta viene subito inoltrata al nodo gestore, e una volta eseguita l’eliminazione si invia la risposta al chiamante. In parallelo, viene lanciata tramite goroutine una RPC verso il nodo successore in modo tale da eliminare anche la replica che quel nodo mantiene. Come descritto in precedenza nell’architettura di sistema, la richiesta di delete viene inoltrata su tutto il Ring JDSys in modo da cancellare definitivamente ogni replica di quella risorsa ed evitare che questa torni “viva”

*5.4.2 Scalability Service RPC*

Ogni JDSys Node espone anche una serie di RPC che vengono utilizzate internamente dal sistema per implementare i servizi per la gestione di Join, Leave, Replicazione e Riconciliazione.

*5.4.2.1 JoinRPC*

Un nodo all’ingresso nell’anello Chord, dovrà chiedere al suo predecessore di inviargli tutte le sue entry. Questa richiesta è implementata tramite una chiamata RPC. Al momento del Join quindi un nodo effettuerà una chiamatata JoinRPC verso il suo predecessore. Il nodo che riceve la chiamata invierà un messaggio di Join al suo successore, in modo che il nuovo nodo possa inizializzare correttamente il suo storage locale.

*5.4.2.2 LeaveRPC*

Quando un nodo viene schedulato per la terminazione, prima di lasciare l’anello chord dovrà inviare tutte le sue entry al nodo successore, in modo che questo possa diventarne il gestore. La lista di nodi in terminazione viene mantenuta dal registry. Questo notificherà l’istanza in terminazione tramite una chiamata LeaveRPC. Il nodo che riceve questa chiamata, dovrà quindi inviare il suo database al nodo successore, tramite un messaggio di Leave.

*5.4.2.3 StartReconciliationRPC*

Il service registry periodicamente inizializza il servizio di riconciliazione, e per fare ciò deve eleggere un nodo che diventerà il gestore di questo servizio. Dopo aver scelto un nodo randomico tra quelli dell’anello, il Registry effettua una chiamata StartReconciliationRPC verso di esso. Il nodo che riceve tale chiamata, dovrà quindi registrarsi come gestore della riconciliazione, settando a true l’apposita variabile globale. Verrà quindi incrementato di 1 una variabile round che tiene traccia del giro corrente del servizio. Successivamente invierà al successore un messaggio di riconciliazione, terminando il servizio quando sarà completato il secondo giro.

5.5 Cloud Storage

Come già descritto in precedenza, JDSys utilizza la migrazione dei dati poco acceduti sul cloud per migliorare la scalabilità ai dati. Per effettuare il download e l’upload di entry da/verso il bucket S3, si sfruttano le SDK di AWS disponibili per Go.

*5.3.3 Upload su S3*

Per verificare le entry scarsamente accedute, ogni nodo mantiene edegue una goroutine, che periodicamente verifica l’ultimo accesso effettuato su tutte le sue entry tramite l’apposito campo lastAcc. Questo campo viene aggiornato ad ogni accesso in lettura e scrittura, al contrario di timest che viene modificato soltanto con la scrittura dell’entry. Durante il controllo periodico, tutte le entry non accedute da oltre un’ora vengono esportate in formato CSV, e caricate su S3 tramite l’SDK. Tutte le entry migrate vengono aggiunte nella slice del nodo CloudKeys, che mantiene tutte le chiavi presenti nel cloud per non dover ogni volta interrogare S3.

*5.3.3 Download da S3*

Quando un nodo riceve la richiesta locale su mongodb, verifica innanzitutto se questa è presente nel cloud, scorrendo la slice cloudkeys. Se la chiave è stata migrata, si procede a riportarla nello storage locale, eseguendo il download da S3 del file csv ed eseguendo un Mongo Merge. Si esegue a questo punto la query richiesta in locale, e si fornisce la risposta tramite RPC.

5.6 Semantica at Least Once

La semantica at least once garantisce che una richiesta da parte del client arrivi almeno una volta su un nodo. Go non prevede dei meccanismi nativi di timeout per le chiamate RPC, per cui è stato implementato un timeout che parte in maniera sincrona ad ogni RPC effettuata dal client. Allo scadere del timeout si effettua una nuova chiamata RPC. Per implementare questo meccanismo si effettua la chiamata RPC in maniera asincrona, si fa partire il timer. Questo timer usa due canali… [JACOPO]

5.6 Gestione della Concorrenza

JDSys offre accessi concorrenti ai client connessi. La concorrenza viene gestita su 3 livelli: RPC, Messaggi HTTP e MongoDB.

Per quanto riguarda le RPC, diversi client possono effettuare richieste concorrenti verso lo stesso nodo di rete. Un nodo per registrare il servizio RPC e mettersi in ascolto delle chiamate RPC, utilizza la funzione http.ListenAndServe() sulla porta 80, che permette di servire le richieste in ingresso. Utilizzando questa funzione viene garantita automaticamente la concorrenza, in quanto ListenAndServe internamente invoca Serve, che chiama Accept in un ciclo. Accept crea una nuova connessione ogni volta che viene accettato un nuovo client, quindi le connessioni non verranno accodate ma servite in parallelo.

RPC parallele si traducono in accessi concorrenti all’engine locale MongoDB. Mongo, gestisce la concorrenza con un meccanismo interno; sono permesse le read concorrenti, mentre per i conflitti in scrittura si utilizza WiredTiger Storage Engine. Un WriteConflict indica quando più utenti stanno tentando contemporaneamente di aggiornare lo stesso documento. Ogni volta che l'API WiredTiger rileva un WriteConflict a causa di una violazione della concorrenza, MongoDB ritenta internamente gli aggiornamenti finché non viene completato senza incontrare conflitti.

https://docs.mongodb.com/manual/faq/concurrency/

La gestione della concorrenza nello scambio dei messaggi HTTP in JDSys richiede invece una gestione più a grana fine.

Messaggi diversi avviano servizi diversi, ognuno dei quali opera su una specifica porta e su una specifica cartella nel file system. Questo permette di gestire in totale parallelismo i servizi di Replicazione, Riconciliazione e Join/Leave.

Tuttavia, è necessario gestire il corretto isolamento tra le richieste di uno stesso servizio. Infatti, per ogni csv ricevuto, bisognerà esportare anche il csv locale per poi effettuare il merge su MongoDB. Successivamente verrà effettuato il clean delle cartelle utilizzate in modo da non avere interferenze nei messaggi successivi. In questo scenario risulta fondamentale garantire che la gestione di un messaggio per un servizio specifico, inizi solo dopo che la precedente richiesta è stata completamente evasa, per evitare scenari in cui ad esempio viene sovrascritto un file csv ricevuto/esportato, o viene effettuato un clean concorrente della cartella da un altro thread prima che quello corrente abbia effettuato il merge.

6. TESTING

[TXT JACOPO] IDEA FARE ANALISI SIA DEL TRANSIENTE (quindi durante lo scale out) CHE A STEADYS TATE (quindi quando ha scalato) In questo modo diciamo tipo il sistema offre sempre gli stessi tempi di risposta (dopo che ha scalato DEVE essere cosi) però nel transiente abbiamo un aumento dei tempi perche bisogna appicciare il nodo inizializzarlo aspettare che le richieste in sovraccarico vengono smaltite eccetera

7. PUNTI DI FORZA E LIMITAZIONI

Punti di Forza: Sistema molto veloce sia in locale che globale grazie a mongo che salva i dati in RAM, scala automaticamente, non ha presentato crash anche sotto carichi di lavoro considerevoli, completamente decentralizzato, funziona anche usando nodi molto scarsi, completamente configurabile tramite il file Configuration.go, sistema completamente concorrente.

Limitazioni: Presente un componente centralizzato che è il registry. Questo si è reso necessario perché AWS Educate non permette l’utilizzo di eventi e quindi non sarebbe stato possibile ottenere notificare un’istanza in terminazione. Infatti avremmo potuto usare il LB come registry, i nodi chiamavano direttamente il registry per ottenere la lista di nodi attivi. Anche per la riconciliazione avremmo potuto usare un meccanismo di elezione del leader invece che far chiamare un nodo random dal registry.

Mongo salva i dati in RAM quindi si ha un tradeoff tra prestazioni e carico sulla memoria. Per questo è necessario scalare relativamente presto per evitare problemi out of memory e non si puo usare una istanza piu piccola della nano che usiamo. Tempi di risposta leggermente maggiori nel transiente comunque accettabili visto il meccanismo automatico che abbiamo. Sistema non completamente tollerante ai guasti. Si ha un certo grado di resistenza grazie alla replicazione, migrazione al leave tutti i nodi hanno un po tutto. Con lo scaling inoltre un nodo non andrà out of memory ed il crash della singola istanza EC2 è da escludere perche ci si affida al 99.99% di reliability assicurata da amazon. Al crash di un nodo inoltre ci pensa chord a riorganizzare l’anello visto tramite fix() e stabilize(). Tuttavia l’applicazione anche se è stata testata può essere soggetta ad errori improvvisi, ed in tal caso non si ha la sicurezza di non perdere nessuna entry visto che non è stato implementato un vero algoritmo di fault tolerance.

Non avendo una metrica chiave precisa per la misurazione cloudwatch abbiamo si un elastic provisioning ma comunque abbastanza largo tendente all’over

Forse altre, non mi vengono in mente però.

https://www.enterpriseai.news/2015/01/06/aws-rates-highest-cloud-reliability/

8. CONCLUSIONI

In sintesi il nostro sistema risulta affidabile, fornisce tempi di risposta ottimali sotto diversi carichi di lavoro per tipologia e dimensione. Teoricamente dovrebbe scalare all’infinito, l’unica limitazione è data dai costi di amazon.

ACM Reference format:

FirstName Surname, FirstName Surname and FirstName Surname. 2018. Insert Your Title Here: Insert Subtitle Here. In *Proceedings of ACM Woodstock conference (WOODSTOCK’18). ACM, New York, NY, USA, 2 pages.* https://doi.org/10.1145/1234567890

1 Insert Heading Level 1

The updated template, user manuals, samples, and required fonts, all are available at the URL <https://www.acm.org/publications/proceedings-template>. It contains said information for all three versions of MS Word (Windows and 2 versions of Mac). There are also separate links to the user guide, which can be referred to by the user. This URL also contains some useful video links, which describe how to add the template, structure the paper, and generate the layout, in different clips. **Display Formula with Number**

 (1)

**Continuation part of Paragraph Text** The user must style this paragraph in **ParaContinue** style, which follows immediately after the **DisplayFormula** (numbered equation). The **DisplayFormula** style is applied only in case of a numbered equation. A numbered equation always has a number to its right. Insert paragraph text here. **Display Formula without Number**



The **DisplayFormulaUnnum** style is applied only in case of an unnumbered equation. An unnumbered display equation never contains an equation number to its right, and this unique property distinguishes it from a numbered equation.



Figure 1: Figure Caption and Image above the caption [In draft mode, Image will not appear on the screen]

**Theorem/Proof/Lemma.** Insert text here for the enunciation or Math statement. Insert text here for the enunciation or Math statement. Insert text here for the enunciation or Math statement. Insert text here for the enunciation or Math statement. Insert text here for the enunciation or Math statement.

....Insert text here for the Quotation or Extract, Insert text here for the Quotation or Extract, Insert text here for the Quotation or Extract, Insert text here for the Quotation or Extract, Insert text here for the Quotation or Extract, Insert text here for the Quotation or Extract.

1.1 Heading Level 2

In the below paragraph, it is explained how alt-txt value is placed in **MS Word 2010**. To add alternative text to a picture in Word 2010, follow these steps:

1. In a Word 2010 document, insert a picture.
2. Right click on the inserted picture and select the **Format Picture** option.
3. Select the **Alt Txt** option from the left-side panel options.
4. In the "Title:" and "Description:" text boxes, type the text you want to represent the picture, and then click "Close".

Below are steps to place alt-txt value in **MS Word 2013/2016**. To add alternative text to a picture in Word 2013/2016, follow these steps:

1. In a Word 2013/2016 document, insert a picture.
2. Right click on the inserted picture and select the **Format Picture** option.
3. In the settings at the right side of the window, click on the "Layout & Properties" icon (3rd option).
4. Expand **Alt Txt** option.
5. In the "Title:" and "Description:" text boxes, type the text you want to represent the picture, and then click "Close".

*1.1.1 Heading Level 3.* Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here.

*1.1.1.1 Heading Level 4.*Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here.

ACKNOWLEDGMENTS

Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here.

REFERENCES

[1] Patricia S. Abril and Robert Plant, 2007. The patent holder's dilemma: Buy, sell, or troll? *Commun. ACM* 50, 1 (Jan, 2007), 36-44. DOI: <https://doi.org/>10.1145/1188913.1188915.

[2] Sten Andler. 1979. Predicate path expressions. In *Proceedings of the 6th. ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '79)*. ACM Press, New York, NY, 226-236. DOI:https://doi.org/10.1145/567752.567774

[3] Ian Editor (Ed.). 2007. *The title of book one* (1st. ed.). The name of the series one, Vol. 9. University of Chicago Press, Chicago. DOI:https://doi.org/10.1007/3-540-09237-4.

[4] David Kosiur. 2001. *Understanding Policy-Based Networking* (2nd. ed.). Wiley, New York, NY..

Conference Name:ACM Woodstock conference

Conference Short Name:WOODSTOCK’18

Conference Location:El Paso, Texas USA

ISBN:978-1-4503-0000-0/18/06

Year:2018

Date:June

Copyright Year:2018

Copyright Statement:rightsretained

DOI:10.1145/1234567890

RRH: F. Surname et al.

Price:$15.00