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

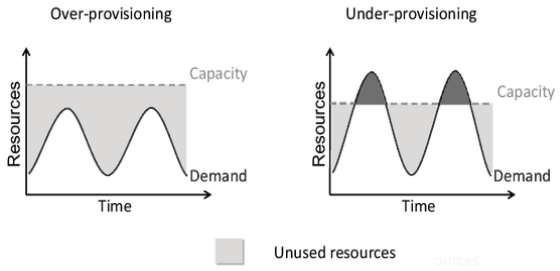
Jacopo Fabi
Facoltà di Ingegneria
Università di Roma Tor Vergata
Roma, Lazio, Italia
jacopo.fabi1997@gmail.com

ABSTRACT

JDSys è un sistema di storage distribuito di tipo Chiave valore, in cui i nodi che gestiscono i dati sono localizzati su differenti server. 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 in termini sia di costi che di carico su ogni nodo. In un sistema di questo tipo, bisogna gestire correttamente il bilanciamento del carico tra i vari server, monitorare l'utilizzo delle risorse, e configurare nuove macchine 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. I dati, oltre ad essere distribuiti su diversi nodi, vengono anche replicati e migrati sul cloud storage S3, per migliorare la scalabilità rispetto al numero di utenti e di dati.

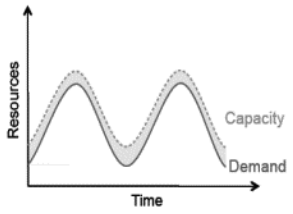
1. INTRODUZIONE

Con l'evoluzione delle tecnologie e l'aumento del traffico internet, 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 si basano su un *provisioning* fisso delle risorse, e su uno *scaling* di tipo verticale. Lo scaling verticale ha costi che crescono esponenzialmente, per cui si avrà un limite superiore alla capacità computazionale raggiungibile. Con un provisioning fisso delle risorse, a fronte di un numero di richieste che può variare nel tempo, si possono seguire due possibili approcci. Il primo approccio è quello di *under provisioning*, in cui si alloca un'infrastruttura che riesce a gestire il carico medio, ma che non riesce a smaltire i picchi di carico. Nell' approccio di over provisioning invece si fornisce un'infrastruttura sovradimensionata, capace di gestire qualsiasi picco di carico. Questo garantisce che il sistema sia sempre disponibile, ma durante periodi di traffico medio ci sarà un sottoutilizzo delle



risorse, e quindi costi superflui.

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



Anche in termini di scalabilità, sfruttando i servizi cloud, è possibile sfruttare la scalabilità orizzontale, non migliorando il singolo server ma aggiungendo più istanze di capacità simile tra di loro. In questo modo si mantiene l'illusione di avere a disposizione una quantità illimitata di risorse.

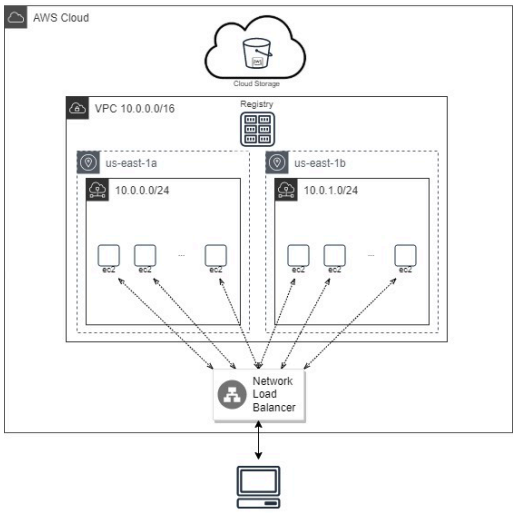


Figura 1: Infrastruttura di rete in JDSys

A fronte di queste considerazioni, l'architettura di JDSys è stata sviluppata sulla base degli **Amazon Web Services**, sfruttando in particolare i componenti Cloudwatch, Elastic Load Balancing ed Autoscaling.

Questo documento ha lo scopo di descrivere nel dettaglio l'architettura di JDSys e la sua implementazione, motivando le varia scelte progettuali effettuate. Il documento è strutturato come segue. La sezione 2 descrive i componenti del sistema ed il modello di dati utilizzato. La sezione 3 presenta la struttura e la progettazione del sistema. La sezione 4 descrive nel dettaglio i principali aspetti implementativi. La sezione 5 descrive i test effettuati su JDSys. La sezione 6 conclude il documento, con le considerazioni finali sul sistema.

2. COMPONENTI

JDSys basa lo storage sul concetto di **Entry**, ovvero una struttura dati composta da una *chiave* ed un *valore*. La chiave rappresenta un identificativo univoco per l'entry, mentre possono esserci anche valori uguali associati a chiavi diverse.

Il sistema di storage è distribuito su diversi server denominati **Nodi**. I nodi sono strutturati in una overlay network ad anello; l'insieme di nodi facenti parte dell'anello è denominato **JDS Ring**.

Il **Service Registry** è un componente centralizzato che permette la scoperta reciproca tra i nodi per formare la rete ad anello. Questo comunica con il *Load Balancer* per ottenere la lista di Istanze *EC2 Healthy*, e mantiene quindi la lista di nodi che fanno parte del *JDS Ring*.

3. 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 della concorrenza, l'interazione con lo storage cloud, il monitoraggio dello stato e degli allarmi del sistema. JDSys, fornisce una soluzione ad ognuno per ognuno questi aspetti, ed in questa sezione andremo a discutere le scelte progettuali effettuate a riguardo.

3.1 Infrastruttura di Rete

L'intera infrastruttura di rete è stata definita sfruttando gli Amazon Web Services a diversi livelli. La *Figura 1* mostra l'architettura di rete in JDSys. I nodi del sistema sono hostati su delle istanze Amazon EC2 t2.micro, che forniscono delle limitate capacità computazionali e di storage (*8GB Disk, 1GB RAM*), che permettono di ridurre i costi durante lo *scaling orizzontale*.

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 inoltrare le richieste in ingresso verso le diverse istanze registrate nel *Target Group*. Un client dovrà quindi solamente contattare il load balancer tramite il suo indirizzo DNS, e la sua richiesta verrà ridirezionata in automatico su uno dei nodi del Ring 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 sia Load Balancer, che da altri nodi tramite un indirizzo IP Privato, e non potranno essere contattati direttamente dall'esterno da altri terminali.

Per simulare in maniera più fedele la latenza di rete tra i nodi del sistema, questi sono stati istanziati in due differenti zone di disponibilità, che descrivono quindi due sottoreti IP differenti (*10.0.0.0/24* e *10.0.0.1/24*). Il service registry si trova all'interno della stessa VPC, ed ha assegnato un IP Privato statico, noto a tutti i nodi del sistema. In questo modo ogni nodo può contattare il registry per scoprire gli altri nodi ed inserirsi nella rete.

Fuori dalla rete privata, troviamo invece un Bucket S3, che fornisce lo storage Cloud necessario per migrare i dati scarsamente acceduti. I nodi interagiscono con S3 utilizzando le SDK di AWS, sfruttando quindi in questo caso non il VPC, ma la rete internet pubblica.

3.2 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-demand. In JDSys sono state utilizzate un minimo di 3 istanze attive, fino ad un massimo di 20.

Figure 2 e 3: Utilizzo di memoria con un traffico medio di 2000 pacchetti

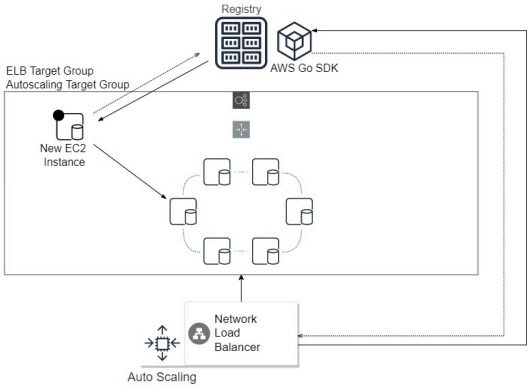


Figura 4: Join di un Nodo nell'anello Chord

Per monitorare il traffico di richieste verso il sistema, si utilizza il servizio **Cloudwatch**. La metrica utilizzata per effettuare *scale-in/scale-out* è una metrica user-defined, non presente tra quelle offerte da Cloudwatch. Questa scelta deriva dal fatto che su AWS non è possibile monitorare l'utilizzo della RAM, informazione tuttavia cruciale in quanto la memoria delle istanze EC2, essendo limitata, tende a saturarsi per un traffico elevato (vd. Sezione 7: Testing)

Analizzando le diverse metriche offerte da Cloudwatch, ci si è resi conto che nessuna di queste fornisce un dato significativo rispetto all'utilizzo di memoria; ad esempio, l'utilizzazione di CPU si avvicina solo al 20% quando un'istanza arriva a saturare la propria memoria (*out of memory error*)

Per i nostri scopi di monitoraggio, è stata quindi definita la metrica **MeanProcessedPackets**, che corrisponde al rapporto tra il numero di pacchetti processati dal LB ed il numero di istanze EC2 correntemente attive. In questo modo, il sistema di autoscaling lavora in base al numero medio di pacchetti processati dalle istanze, avendo dunque a disposizione una metrica che risulta comunque proporzionale rispetto al consumo di memoria della singola istanza.

Al fine di identificare la soglia critica di *Mean Processed Packets* per effettuare le attività di scaling, si è effettuato uno "stress-test" del sistema, individuando quindi il carico medio di pacchetti che un'istanza riesce a gestire prima di saturare la RAM a disposizione. La Figura 2 mostra l'andamento dei pacchetti medi processati, mentre la Figura 3 mostra lo stato della memoria sulle 3 istanze EC2. Basandoci su questi dati sono stati definiti i seguenti allarmi per effettuare lo scaling.

- *AlarmScaleOUT* quando si raggiungono i 1800 pacchetti medi processati per istanza.
- *AlarmScaleIN* quando si hanno meno di 500 pacchetti medi processati per istanza.

La scelta della soglia di scale-out è impostata su un valore largamente inferiore al limite di circa 2200 pacchetti medi per cui un'istanza inizia a saturare la memoria. In questo modo si fornisce al sistema di autoscaling il tempo necessario per istanziare una nuova macchina e rendere UP & Running il Nodo JDSys.

3.3 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 per individuare queste risorse distribuite.

A tale scopo, JDSys utilizza il protocollo **Chord** per realizzare il routing, basato su DHT, in una rete P2P strutturata. I nodi e le chiavi delle risorse sono mappati in un anello tramite la funzione hash detta *consistent hashing*, ed ogni nodo è responsabile delle chiavi poste tra sé stesso ed il nodo precedente dell'anello. Ciò significa che la risorsa con chiave K è sarà gestita dal nodo con il più piccolo ID>=K, detto *successore di K*.

Per evitare la ricerca lineare, Chord implementa un metodo di ricerca basato sulle *finger table*. Ogni nodo mantiene informazioni soltanto su un sottoinsieme dei suoi vicini, per cui l'idea è 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 (*join*) o l'uscita (*leave*) di un nodo dall'anello modifica solo le finger table dei suoi vicini immediati mentre gli altri nodi rimangono inalterati, garantendo robustezza ed elasticità all'architettura di JDSys.

Un nodo, all'ingresso nel sistema (*Figura 4*), contatterà il service registry, che gli fornirà una lista di istanze attive. Il

nodo contatterà quindi un'istanza casuale per inserirsi nell'anello ed aggiornerà 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

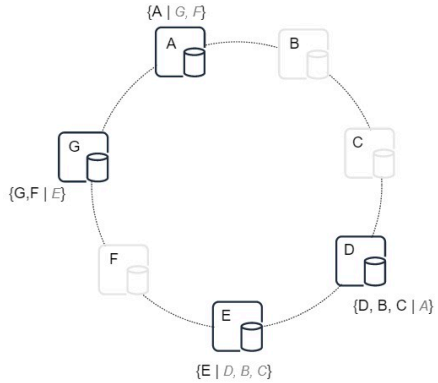


Figura 5: Meccanismo di Replicazione in JDSys

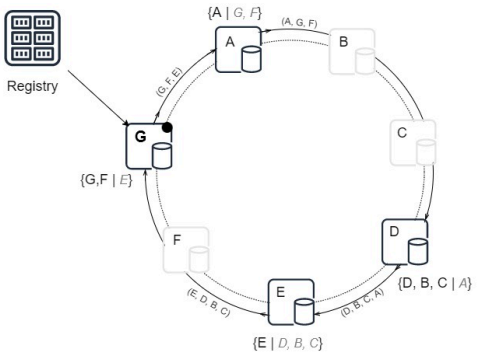


Figura 6: Meccanismo di Riconciliazione in JDSys

entry in modo che questo possa diventarne il gestore senza perdere nessuna informazione.

3.4 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 5*, il nodo A replica la chiave A di cui è gestore sul nodo D. Il nodo D replica le chiavi B,C e 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 verso di essa; subito dopo aver risposto al chiamante, viene contattato il nodo successore inviandogli la replica.

3.5 Consistenza

In un contesto in cui si hanno dati replicati, risulta fondamentale fornire un meccanismo per raggiungere la consistenza, per non avere copie di dati discordanti. 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 tra le diverse copie 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.

3.3.1 Riconciliazione

Per risolvere i conflitti tra tutte le repliche, e giungere così alla consistenza finale, si utilizza un algoritmo di riconciliazione di tipo **Asyncrhonous Repair**. Come mostrato in *Figura 6*, periodicamente il service registry avvia il servizio di riconciliazione, contattando un nodo casuale 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 soltanto la copia più recentemente aggiornata. In tal senso, ad ogni scrittura su una entry si aggiorna il suo *timestamp*, sfruttando il *Network Time Protocol*. La scelta di utilizzare questa soluzione per la sincronizzazione delle repliche è dovuta ad una maggiore semplicità di gestione ed implementazione rispetto ad un degrado di prestazioni comunque minimale.

Questo processo di **riconciliazione**, per avere la certezza di risolvere tutte le copie conflittuali, necessita al più di due giri completi 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.

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 subito propagata la richiesta di delete 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 *stale*, fuori dalla portata del gestore.

3.6 Cloud Storage

Per migliorare la scalabilità ai dati, JDSys integra i propri servizi 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.

3.7 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, ad esempio dovuti alla riconfigurazione della rete overlay. In questo scenario è fondamentale non perdere queste richieste, e a tale scopo JDSys utilizza la **semantica at least once** per le richieste effettuate dal client verso il sistema. Questa semantica utilizza il solo 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.

4. IMPLEMENTAZIONE

JDSys è sviluppato utilizzato il linguaggio Go. Il vantaggio principale è dovuto alla gestione leggera ed efficiente della concorrenza grazie alle *goroutine* ed ai *canali*. Inoltre, Go offre supporto nativo alle Remote Procedure Call, che permettono di simulare una chiamata di procedura locale durante l'interazione *client-server* o *nodo-nodo*.

In JDSys, ogni nodo del sistema è composto da quattro componenti fondamentali

- Engine di Storage Locale
- Gestore dell'anello Chord
- Listener dei messaggi HTTP
- Listener delle chiamate RPC

4.1 Storage Locale

JDSys utilizza MongoDB come engine di storage del singolo nodo. La scelta è ricaduta su questo DBMS in quanto fornisce prestazioni ottimali, una buona scalabilità, ed una gestione automatica della concorrenza *(vd. Sezione 4.7: Concorrenza)*. Per l'interazione tra mongo ed il software del Nodo, è stata utilizzata la libreria mongo-driver[1].

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 la funzione *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 nell'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; 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 e aggiornato ad ogni operazione di scrittura tramite il *Network Time Protocol*. A tale scopo si è utilizzata la libreria beevik-ntp[2], ottenendo il tempo corrente tramite il server remoto NTP *0.beevik-ntp.pool.ntp.org*.

Le operazioni di effettuate dal client si tradurranno sempre in query eseguite sullo storage mongo locale, sfruttando le operazioni *FindOne()*, *InsertOne()*, *UpdateOne()* e *DeleteOne()*

4.2 Gestione dell'Anello

In JDSys la gestione dell'overlay network è effettuata in modo decentralizzato sfruttando il protocollo chord. Il service registry rappresenta l'unico elemento centralizzato per permettere la registrazione ed il discovery dei nodi.

4.2.2 Service Registry

Il service registry permette di mantenere la lista di istanze attive nel target group, e di intercettare tutte le attività di scale-in e scale-out schedulate dal servizio di Autoscaling. La libreria utilizzata per l'interazione con i servizi AWS è aws-go-sdk[3]. In particolare, il registry, 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, mantenendo per ognuna di queste ID ed indirizzo IP Privato.

Per ottenere invece la lista di istanze in fase di terminazione, si contatta l'Autoscaling tramite ARN, richiedendo la lista di attività di scaling. Partendo da questa lista si ricavano le sole istanze in fase di terminazione, filtrando il messaggio di stato come "WaitingForELBDraining".

4.2.2 Chord

Per effettuare il routing delle risorse, come già detto si utilizza il protocollo Chord. Riguardo l'implementazione, si è fatto uso della libreria cbocovic/chord[4].

Anzichè importare tale libreria sfruttando il meccanismo nativo di go, si è deciso di utilizzare direttamente il codice sorgente presente su Github, in modo tale da adattarne l'implementazione alle esigenze dirette di JDSys. In particolare, sono stati modificati alcuni parametri di configurazione non direttamente gestibili tramite import. La modifica principale effettuata riguarda la riduzione del tempo di attesa prima che un nodo contatti i suoi vicini per aggiornare la sua Finger Table, in modo da rendere la ricostruzione dell'anello più efficiente.

4.3 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 per ogni servizio, anche mantenendo lo stesso formato dei messaggi.

4.3.1 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 presenti in locale.

4.3.2 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.

4.3.3 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.

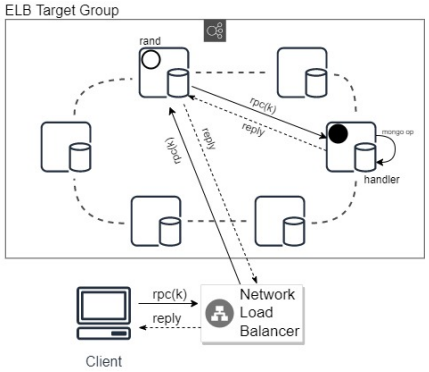


Figura 7: Propagazione delle RPC in JDSys

4.4 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 nodo espone diverse RPC, che si differenziano in base al loro utilizzo e al loro scopo. In particolare, alcuni metodi verranno invocati dal client, altri verranno invocati da altri nodi, ed altri ancora dal service registry.

4.4.1 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 verso il Load Balancer. Il Load Balancer seguendo la sua politica di bilanciamento, si occupa 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.

4.4.1.1 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 grazie alla replicazione potrebbe avere una copia di quella risorsa anche se non è il suo gestore secondo chord. Se la chiave non viene trovata in locale su mongodb, viene effettuato il lookup e la RPC viene inoltrata al nodo che deve gestire quella chiave.

4.4.1.2 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.

4.4.1.3 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.

4.4.2 Altre RPC

4.4.2.1 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.

4.4.2.2 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 ogni 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.

4.4.2.3 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, ed incrementerà la variabile *round* che tiene traccia del giro corrente del servizio. Successivamente invierà al successore un *messaggio di riconciliazione*, terminando la procedura quando verrà completato il secondo giro.

4.5 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 *(Figura 8)*.

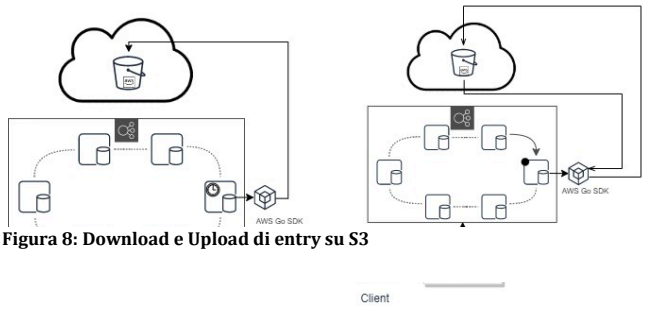


Figura 8: Download e Upload di entry su S3

4.3.3 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.

4.3.3 Download da S3

Quando un nodo riceve la richiesta locale di Get, Put, Delete o Append, prima ancora di cercare la chiave su *mongodb*, verifica se questa è presente nel cloud, scorrendo la slice globale *cloudkeys*. Se la chiave è stata migrata sul cloud, 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 sempre tramite RPC.

4.6 Semantica At-Least-Once

La semantica at-least-once garantisce al client che il servizio, se eseguito, sia stato eseguito almeno una volta dal server. Questo implica che il server può eseguire il servizio anche molteplici volte a causa della duplicazione delle richieste dovuta alle ritrasmissioni. La semantica in uso prevede solamente il meccanismo RR1 lato client, motivo per cui la semantica è adatta a servizi idempotenti. Quindi per quanto riguarda l'operazione di append, in caso di ritrasmissioni si avrà un'alterazione del risultato atteso dal client aggiungendo più volte all'entry lo stesso valore.

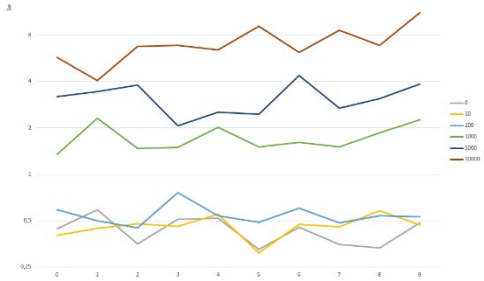


Figura 9: Test workload1, Tempi di Risposta nel transiente

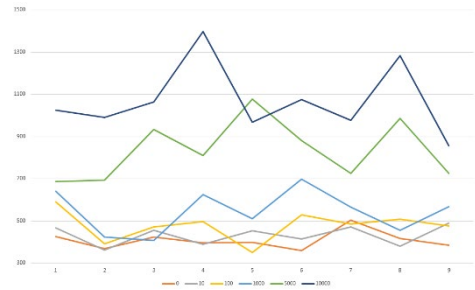


Figura 10: Test workload1, Tempi di Risposta in stato stazionario

Per la realizzazione della semantica di ritrasmissione, non avendo Go dei meccanismi nativi di timeout per le chiamate RPC, è stato implementato un timeout che viene avviato in una goroutine parallelamente ad ogni RPC effettuata dal client. Per la gestione del timeout si fa uso di un ciclo per realizzare un massimo numero di ritrasmissioni, e al suo interno ci si pone in attesa bloccante su due canali, il primo utilizzato dal timeout e il secondo utilizzato dalla RPC per scrivere la risposta o l'errore. Se il primo canale a ricevere il messaggio è quello relativo al timeout, vuol dire che il timer è scaduto; si effettua quindi una nuova chiamata RPC e si avvia un nuovo timer. Se l'RPC risponde sul canale prima dello scadere del timeout, si stoppano le ritrasmissioni, interrompendo l'apposito ciclo *for.*

4.6 Gestione della Concorrenza

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

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 crea una nuova connessione per client che viene accettato[5].

RPC parallele si traducono in accessi concorrenti verso l'engine locale MongoDB. Mongo, gestisce la concorrenza con un meccanismo nativo; vengono permesse le read concorrenti, mentre per i conflitti in scrittura vengono risolti grazie a WiredTiger Storage Engine[6].

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 sul singolo nodo il corretto isolamento tra le richieste relative ad uno stesso servizio, per evitare scenari in cui un file csv venga sovrascritto o eliminato prima ancora di effettuare il merge. A tale scopo vengono utilizzati 3 mutex differenti per l'invio di un csv, la ricezione di un csv, e la gestione di join/leave. In questo modo il singolo nodo anche su richieste concorrenti di questo tipo, prenderà un lock all'inizio del servizio, e lo rilascerà solo dopo aver effettuato l'import/export e aver pulito l'ambiente di esecuzione.

5. TESTING

Per realizzare il testing delle prestazioni del sistema, si è andato a stimare il tempo di risposta all'aumentare del numero di richieste degli utenti. In particolare, per i due workload richiesti, si è effettuata la misurazione dei tempi di risposta con cinque diversi carichi di lavoro: 10, 100, 1000, 5000, 10000 thread concorrenti. I test sono stati realizzati lanciando la precisa percentuale di operazioni *Get*/*Put*/*Append* in base al carico di lavoro selezionato. Ogni volta che un thread in esecuzione termina, viene spawnato immediatamente un nuovo thread, in modo da mantenere un carico di lavoro più o meno costante per il sistema. Le misurazioni sono state realizzate sia nel transiente (durante lo scale out) che a steady-state (scale out completato).

I tempi di risposta, nei test, vengono calcolati come la media di diverse misurazioni. Una misurazione consiste nel tempo medio di risposta per effettuare l'aggiunta, la modifica, l'append, la lettura e la cancellazione di una entry. Vengono effettuate dieci misurazioni, e si calcola il tempo medio tra queste. Questo permette di avere una maggiore accuratezza ed effettuare un'analisi più dettagliata dal momento che le richieste possono avere tempi diversi di gestione in base al nodo su cui vengono ridirezionate.

5.1 Workload 1

Il workload1 è composto dall'85% di Get ed il 15% di Put. Analizzando i risultati dei test effettuati nel **transiente**, possiamo vedere come, per un carico di lavoro di 0, 10 e100 richieste, i tempi di risposta si mantengono più o meno sullo stesso livello. Questo è dovuto al fatto che i nodi non vengono sovraccaricati e sono in grado di gestire in maniera ottimale il carico senza effettuare autoscaling. Notiamo che in questo caso si misurano tempi di risposta compresi tra i 250ms e 1.5s.

Di contro, per quanto riguarda carichi di lavoro di 1000, 5000 e 10000 richieste, i tempi di risposta subiscono un notevole incremento. Questo è conseguenza del fatto che è necessario l'intervento dell'autoscaling, ma lo scale-out non avviene in maniera istantanea ed è necessario un periodo di circa 4-5 minuti prima che il nuovo nodo diventi attivo. Per questo motivo carichi elevati causano tempi di risposta elevati in questa fase di transizione, in quanto dovranno gestire un

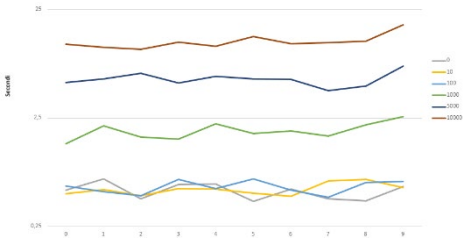


Figura 11: Test workload2, Tempi di Risposta nel transiente

numero medio maggiore di richieste. I tempi di risposta che si misurano in questo caso sono compresi tra i 2s ed i 16s.

Analizzando invece risultati dei test effettuati nello **stato stazionario**, possiamo osservare che raggiunta la stabilità i tempi di risposta diminuiscono notevolmente, anche per i carichi che tendevano nel transiente quasi a saturare le istanze EC2. Possiamo quindi vedere come, per carichi di lavoro di 1000, 5000 e 10000 richieste, i tempi di risposta migliorano nettamente rispetto all'analisi nel transiente e si mantengono comunque molto vicini a quelli misurati per carichi di lavoro bassi. Il leggero incremento dei tempi che si ha al crescere del workload è dovuto al fatto che aumentando il numero di nodi, crescerà anche il tempo necessario per effettuare il lookup di chord.

5.1 Workload 2

Il workload2 è composto dal 40% di Get, 40% di Put ed il 20% di Append. Analizzando i risultati dei test, possiamo vedere come ci sia un significativo aumento dei tempi di risposta, sia nel transiente che in stato stazionario. Questo incremento è dovuto al fatto che si effettuano per il 60% operazioni di scrittura, e di conseguenza MongoDB dovrà gestire molte più richieste concorrenti sul database. Inoltre, risulta molto più probabile che le richieste di scrittura vengano ridirezionate verso i nodi gestori rispetto alle Get, in cui nodi che ricevono la richiesta possono avere una replica e rispondere immediatamente.

6. CONCLUSIONI

6.1 Punti di Forza

Il sistema risulta essere molto veloce in termini di tempi di risposta. Questo è dovuto in parte all'efficienza di MongoDB, che mantiene i dati in RAM per fornirli in maniera più veloce.

Grazie all'autoscaling, si riesce a mantenere un livello di performance accettabile con qualsiasi carico di lavoro fornito in input al sistema, avendo solamente un leggero degrado delle prestazioni nell'ordine dei millisecondi.

Il sistema è completamente decentralizzato e il costo di mantenimento è ridotto al minimo utilizzando una delle tipologie di istanze più piccole offerte da Amazon.

Il sistema è completamente configurabile apportando le opportune modifiche nel file *utils/Configuration.go*

Il sistema è completamente concorrente, riuscendo a gestire un numero di richieste pari al massimo numero possibile di goroutine che un'istanza EC2 è in grado di spawnare.

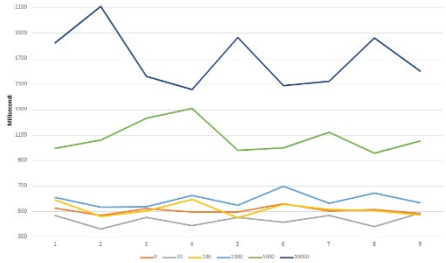


Figura 12: Test workload2, Tempi di Risposta in stato stazionario

6.1 Limitazioni

Il sistema presente un componente centralizzato che è il service registry. Questa scelta è stata necessaria a causa dell'account AWS Educate, che non permettendo l'utilizzo di eventi non ha reso possibile implementare un meccanismo di notifica diretta verso un'istanza in terminazione. Infatti, si sarebbe potuto utilizzare il LB come registry, i nodi lo avrebbero contattato direttamente per ottenere la lista di istanze attive nel momento in cui era richiesta la Join nell'anello. Dovendo avere questo service registry all'interno dell'architettura del sistema, si è scelto di attivare la riconciliazione tramite notifica del registry ad un nodo random anzichè utilizzare un meccanismo di elezione del leader tra i vari nodi.

Ogni nodo del sistema utilizza MongoDB che salva una quantità significativa di dati in RAM, per questo motivo è stato necessario individuare un trade-off tra prestazioni e utilizzazione di memoria non avendo quest'ultima a disposizione come metrica. Di conseguenza, il sistema scala relativamente presto per evitare eventuali crash dei nodi per l'errore "out of memory", avendo quindi un *elastic provisioning* che potrebbe essere ulteriormente raffinato.

Si misurano tempi di risposta elevati nel transiente per carichi di lavoro significativi, tempi che comunque migliorano in breve tempo non appena terminano le attività dell'autoscaling.

Il sistema non è completamente tollerante ai guasti, ma si ha comunque un certo grado di resistenza grazie alla *replicazione* e alla *migrazione* effettuata al leave dei nodi dall'anello. Inoltre, grazie all'autoscaling, un nodo non andrà "out of memory" ed il crash della singola istanza EC2 è da escludere perchè ci si affida al SLA di Amazon che assicura il 99.99% di reliability. Tuttavia l'applicazione, anche se è stata testata, può essere soggetta ad errori improvvisi, non avendo in questo caso la certezza di mantenere correttamente tutte le entry. Questo è dovuto al fatto che non è stato implementato nessun algoritmo specifico per la fault-tolerance.

RIFERIMENTI

[1] Mongo Driver: https://pkg.go.dev/go.mongodb.org/mongo-driver@v1.8.0/mongo
[2] Beevik NTP: https://github.com/beevik/ntp
[3] AWS Go SDK: https://github.com/aws/aws-sdk-go
[4] Go Chord: https://pkg.go.dev/github.com/cbocovic/chord
[5] Concorrenza Http: https://eli.thegreenplace.net/2019/on-concurrency-in-go-http-servers/
[6] Concorrenza Mongo:https://docs.mongodb.com/manual/faq/concurrency/