Trasparenza alla replicazione per le chiamate RPC in go

Gentili Emanuele*
emanuele.gentili@students.uniroma2.eu
Università di Roma Tor Vergata
Roma, Italia, Europa

ABSTRACT

Negli ultimi anni i dispositivi, e quindi gli utenti, connessi ad internet sono aumentati repentinamente. Questo ha causato un aumento del traffico al quale i server sono sottoposti. Se si pensa ad organizzazioni le cui risorse sono richieste da milioni di persone ogni giorno, esse devono fare in modo che ciascun utente goda della migliore esperienza possibile utilizzando i servizi offerti dall'organizzazione. Ovviamente non tutte le richieste per una risorsa possono essere gestite da un unico server. Quindi, si devono creare delle repliche dei server che devono essere posizionate in maniera strategica. Quando la distribuzione delle repliche avviene su larga scala si ha una content delivery network. L'aspetto, interessante per il contesto di questo progetto non è dove le repliche vengono collocate, ma il fatto che replicando i server che gestiscono le risorse si riesce a ridurre la latenza percepita dall'utente finale [5]. Affinché questo sia possibile, si devono distribuire le richieste in maniera uniforme tra le repliche. Lo strumento che si occupa della distrbuzione delle richieste, migliorando anche scalabilità e prestazioni [2] si chiama load balancer. Inoltre, l'utilizzo del load balancer permette di raggiungere la trasparenza alla replicazione in quanto esso nasconde agli utenti finali la presenza delle repliche e dei loro guasti.

1 INTRODUCTION

L'obiettivo di questo progetto è quello di raggiungere la **trasparenza alla replicazione** per le chiamate RPC in Go. Affinché questo sia possibile, il client non deve essere a conoscenza che per una risorsa esistono delle repliche[4]. Nel caso particolare di RPC in Go la trasparenza alla replicazione non è supportata. Infatti, il client per poter inviare richieste ad un server deve conoscere l'indirizzo Ip e il numero di porta a cui inviare la richiesta. Quindi se ci fossero più repliche, il client per invargli richieste ne deve conoscere gli indirizzi. Nei prossimi paragrafi verrà spiegata una possibile soluzione a questo problema. L'idea generale è quella di utilizzare un load balancer che permette di:

- distribuire le richieste
- gestire eventuali guasti

Una volta implementata la soluzione in Go, ciascuna componente verrà containerizzata e distribuita su un'instanza EC2 [1].

2 BACKGROUND

Possiamo dunque dire che il load balancer si comporta come un server dal punto di vista del client e come un client dal punto di vista delle repliche. Infatti, esso attende le richieste dei client e gestisce ciascuna richiesta inviandola ad una o più repliche. Una volta che il load balancer ha inviato una richiesta, si mette in attesa di una risposta da parte delle repliche selezionate. Gli scenari in cui esso si può trovare sono i seguenti:

- riceve il risultato senza errori: allora restituisce il risultato al client.
- riceve un errore: per soddisfare la richiesta potrebbe essere necessario scegliere nuove repliche a cui inviare la richiesta. In questo caso per soddisfare le richieste del progetto si deve fare in modo che la richiesta venga servita da altre repliche.

Dunque, il client è a conoscenza solo del load balancer e deve sfruttare questa conoscenza per poter comunicare con le repliche. Tuttavia, le repliche potrebbero comunque rispondere direttamente al client. Infatti esistono due tipi di load balancer:

- **one-way**: la risposta viene inviata direttamente al client dalla replica che ha servito la richiesta.
- **two-way**: la replica invia la richiesta al load balancer che farà da tramite con il client anche in fase di risposta.

Entrambe le soluzioni sono valide per il raggiungimento della trasparenza all'applicazione. Un altro aspetto interessante è la politica che il load balancer utilizza per instradare le richieste ricevute. In particolare, si può distinguere tra politiche:

- **State aware**: I server a cui viene richiesto di processare la richiesta vegnono scelti sulla base del loro stato.
- **State less**: I server a cui inoltrare la richiesta vegnono scelti senza guardare lo stato. Ad esempio si potrebbe utilizzare una politica di scelta random oppure round robin.

3 SOLUTION DESIGN

Per raggiungere la trasparenza alla replicazione in Go, si è scelto di utilizzare un load balancer two-way che utilizza una politica state-aware per l'instradamento delle richieste. Inizialmente, il load balancer non conosce quali repliche sono disponibili per servire la richiesta. Ciascuna replica, in fase di set-up, invierà al service registry una richiesta di iscrizione. Il service registry, per ciascuna replica memorizza:

- indirizzo IP
- numero di porta

Ogni volta che il service registry riceve una richiesta da una replica invia i dati che ha memorizzato al load balancer. Quest'ultimo, manterrà per ciascuna replica che gli viene comunicata uno stato. Lo stato che si è scelto di far mantenere al load balancer è il numero di richieste pendenti. Quando si parla di richieste pendenti, si intende il numero di richieste che il load balancer ha assegnato ad una replica ma per le quali non è stata ancora ricevuta una riposta. Ogni volta che il load balancer riceve una richiesta, seleziona casualmente una replica a cui inviare la richiesta tra quelle che hanno il minor numero di richieste pendenti. Per cercare di ridurre la latenza, per ciascuna richiesta il load balancer sceglie due repliche a cui inviare la richiesta, quando riceve la risposta da una delle due blocca la

*Studente magistrale di Ingegneria Informatica

computazione sull'altro server. Riassuamendo, le componenti che costituiscono il sistema sono:

- uno o più **client**
- una o più **repliche**
- un **load balancer**
- un **service registry**

Come detto nella precedente sezione, ciascuna componente deve essere containerizzata. Ovvero, deve essere creato un container. I container [3] sono un'astrazione a livello delle applicazioni che impacchetta insieme codice e dipendenze. Più container possono essere eseguiti sulla stessa macchina e condividere il kernel del sistema operativo. Ciascuno container viene eseguito come un processo isolato nello spazio utente. Si dovranno poi connettere le compoenti rispettando le connessioni che possiamo osservare in figura 1.

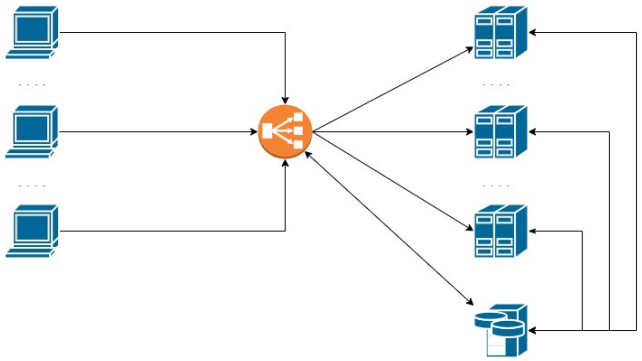

Figure 1: Collegamenti tra le varie componenti

4 SOLUTION DETAILS

In questo paragrafo, viene spiegato nel dettaglio come si è raggiunta la trasperenza alla replicazione in Go, garantendo:

- gestione della scoperta dei servizi
- gestione dei crash delle repliche
- una politica di load balancing state-aware
- un meccanismo per ridurre la latenza

4.1 Gestione della scoperta dei servizi

Ciascun server in fase di set-up, deve sapere il numero di porta sul quale deve esporre i servizi. Questa informazione viene fornita al server tramite una varibile d'ambiente. Una volta che questa informazione è stata recuperata, il server comunicherà al service registry:

- Il proprio indirizzo IP
- Il numero di porta su cui è esposta l'applicazione.

Per ogni nuova replica, il service registry memorizza le suddette informazioni e comunica al load balancer che un nuovo server è attivo. Ogni volta che il load balancer riceve la comunicazione che c'è un nuovo nodo attivo, invia al service registry la propria lista di nodi. In questo modo, il service registry può aggiornare la propria lista sulla base dei server che hanno fallito e che il load balancer ha identificato come falliti. Nel progetto questo aggiornamentro potrebbe sembrare superfluo. Tuttavia, in un contesto in cui il service registry viene utilizzato anche da altri load balancer o in generale da qualche altra componente, la lista del service registry si deve tenere aggiornata.

4.2 Gestione dei crash delle repliche

Per ciascuna replica, oltre al numero di richieste pendenti, si mantiene anche il numero di volte che essa è stata scelta ma in fase di connessione o durante la computazione ha restituito al load balancer un errore. Inoltre, si definisce una soglia di tolleranza che rappresenta il numero di volte che si permette ad un server di restituire un errore senza che esso venga considerato guasto. Quando il contatore supera la tolleranza, il server viene rimosso dalla lista dei nodi selezionabili per servire le richieste. Per quanto detto, il server in fase di set up comunica la propria attività al service registry. Quando il service registry propaga al load balancer questa informazione, quest'ultimo andrà ad azzerare il contatore per la replica in questione. Oppure, nel caso in cui il contatore per la replica non era presente viene creato. Quando un server viene decretato come fallito, la richiesta che esso avrebbe dovuto gestire, deve essere inviata ad un altro server. Tuttavia, ci si potrebbe trovare nel caso in cui tutte le repliche sono state decretate come fallite. In tal caso, si attende un tempo T prima di comunicare al client che non è stato possibile servire la richiesta. In questo modo evitiamo che il load balancer rimanga per un tempo indefinito ad attendere che qualche server torni attivo. Ci si aspetta che il client proverà nuovamente a fare la richiesta dopo un certo tempo.

4.3 Progettare una politica di balancing state-aware

Avere un bilanciamento equo delle richieste permette di ridurre il tempo di risposta percepito dal client. Infatti se tutte le richieste fossero inviate verso un unico server, esso rappresenterebbe un collo di bottiglia. Ad ogni richiesta ricevuta, il load balancer cerca i server che hanno il minor numero di richieste pendenti. Per individuare i suddetti server, si utilizza l'algoritmo 1. Una volta che si è ottenuta la sotto-lista che contiene i server meno carichi casualmente se ne sceglie uno.

Algorithm 1 State-Aware Balancing Policy

```
Require: servers[]
Ensure: minServer        ▷ Server meno carico scelto casualmente
  minLoad ← min(servers.numberOfPending)
  for s in servers do
      if s.numberOfPending == minLoad AND s != toIgnore then
          minServers.append(s)
      end if
  end for
  minServer ← minServers[rand]
```

4.4 Progettare un meccanismo per ridurre la latenza

La politica utilizzata è sensibile alla variabilità del carico. Ovvero, le richieste fatte dal client, possono richiedere tutte dei tempi di computazione diversi e la politica non tiene conto di questo fatto.

Infatti, quando una richiesta viene indirizzata verso il server che ha meno richieste pendenti non si tiene conto del tempo di servizio rimanente del server scelto. Ovvero, non ci si domanda quanto tempo rimane prima che il server termini la propria computazione. Per diminuire il tempo di risposta e rendere il load balancer più robusto ai guasti si utilizza un meccanismo che prevede di inviare la richiesta a due server, scelti entrambi con l'algoritmo 1. Quando il load balancer riceve la risposta da uno dei due server invoca la terminazione della computazione sul server che sta impiegando più tempo a rispondere. L'algoritmo che si deve implementare per fare quanto detto è l'algoritmo 2.

Algorithm 2 Reduce Latency Method (RLM)

Require: $server_1, server_2, arg$ ▷ Server a cui inviare la richiesta
Ensure: $result$ ▷ Risposta alla richiesta sottoposta dal client

```
handler1 ← sendRequests(server1, arg)
handler2 ← sendRequests(server2, arg)
while true do
    if handler1.Done == true then
        result ← handler1.Result
        invokeTermination(server2)
    end if
    if handler2.Done == true then
        result ← handler2.Result
        invokeTermination(server1)
    end if
end while
```

5 RESULT

Arrivati a questo punto si vuole valutare quanto fatto fino ad ora. In particolare ci si domanda:

(1) La politca implementata bilancia il carico in maniera adeguata?
(2) Cosa accade se durante la computazione di una richiesta uno o entrambi i server che stanno servendo la richiesta fallisce?
(3) La trasparenza alla replicazione è stata raggiunta?

5.1 La politica implementata bilancia in carico in maniera adeguata?

Per rispondere a questa domanda si consideri un server che fornisce due servizi:

- Calcolo del numero di Fibonacci di un numero proposto dal client
- Calcolo della potenza di un numero proposto dal client.

Consideriamo un esempio in cui ci sono 10 server pronti a servire le richieste. Il client invia in parallelo n richieste, metà per il calcolo del numero di Fibonacci e metà per il calcolo della potenza. Consideriamo due casi: $n = 100$ e $n = 200$. Nel caso $n = 100$ si può osservare una varianza più bassa rispetto al caso con 200 richieste. Questo dipende dal fatto che le richieste utilizzate per calcolare questo risultato, sono state scelte dal client in maniera casuale. D'altra parte, il load balancer guarda solamente il numero di richieste pendenti, ma non la computazione richiesta per terminare la richiesta pendente. Quindi, un server potrebbe ricevere una maggioranza di richeste leggere terminando prima la computazione rispetto ad un server che ha ricevuto delle richieste più pesanti. Dunque il server che riceve richieste più leggere avrà maggiore probabilità di essere scelto più volte. I risultati ottenuti dopo un'esecuzione si possono osservare nella tabella 1.

indirizzo	con 100 richieste	con 200 richieste
172.23.0.6:1234	22	37
172.23.0.3:1234	20	40
172.23.0.7:1234	20	45
172.23.0.13:1234	22	44
172.23.0.4:1234	25	42
172.23.0.11:1234	19	46
172.23.0.9:1234	18	32
172.23.0.5:1234	16	37
172.23.0.8:1234	17	41
172.23.0.12:1234	21	36
varianza	7	20
media	20	40

Table 1: Mostra il numero di volte che il load balancer selezione ciascuna replica al variare del numero di richieste

5.2 Cosa accade se durante la computazione di una richiesta uno o entrambi i server che stanno servendo la richiesta fallisce?

Prendiamo il caso in cui il load balancer assegna la richiesta a due server replica, siano A e B. Gli scenari possibili sono 2:

- Uno solo tra A e B fallisce: in questo caso la computazione continua sull'altro server.
- A e B falliscono in questo caso si deve trovare almeno un altro server a cui inviare la richiesta di nuovo.

Chiaramente nel caso in cui ci si trova nel primo scenario, non dovendo inviare una nuova richiesta permette di risparmiare del tempo perché magari il crash è avvenuto quando entrambi i server avevano già computato in parte il risultato.

5.3 La trasparenza alla replicazione è stata raggiunta?

La risposta è si. Infatti, il client invia la richiesta e riceve la risposta senza sapere che:

- la richiesta è stata servita da due server.
- durante la computazione uno o entrambi i server che stavano servendo la richiesta hanno subito un fallimento.

L'unico caso in cui il client riceve un errore è quando nessun server è disponibile per servire la richiesta e l'attesa supera un tempo stabilito. In ogni altro caso, il server riceverà la risposta alla richiesta proposta.

6 DISCUSSION

In questa sezione si discute di possibili problemi, di possibili soluzioni e di vantaggi dovuti all'utilizzo della soluzione presentata. Un primo

problema è il load balancer, che rappresenta un single point of failure in quanto non è distribuito. Questo significa che se il load balancer fallisce, le repliche non sono più raggiungibili dai client. Una possibile soluzione è quella di avere più load balancer. Replicando il load balancer otteniamo diversi vantaggi:

- aumenta la disponibilità
- aumenta la tolleranza ai guasti
- migliorano le prestazione attraverso la scalabilità

Tuttavia, il load balancer, ha uno stato che deve essere mantenuto consistente con quello degli altri load balancer. Dunque si dovrà anche scegliere un livello di consistenza che deve essere rispettato. Dato che, l'obiettivo del progetto era quello di raggiungere la trasparenza alla replicazione in Go, replicando il load balancer, si deve fare in modo che agli occhi del client la richiesta venga sempre servita da un'unica componente. Quindi si deve evitare che il client debba conoscere tutti gli indirizzi dei load balancer. Inoltre, un singolo load balancer rappresenta anche un collo di bottiglia quando i server aumentano. Un altro single point of failure e collo di bottiglia è il service registry che potrebbe essere anch'esso replicato e seguirebbero tutte le considerazioni appena fatte per il load balancer. Dunque, un'idea per gli sviluppi futuri di questo progetto potrebbe essere quella di distribuire il service registry tenendo conto di tutte le considerazioni fatte per il load balancer. Come detto nei paragrafi precedenti, si è scelto di utilizzare come stato il numero di richieste pendenti. Questa scelta, ci permette di tenere il conto del numero di richieste che un server sta gestendo. Il concetto di **richiesta pendente** è generale. Dunque, se al posto del client e del server che sono stati utilizzati nella soluzione si utilizzassero client e server differenti la politca presentata funzionerebbe lo stesso. Per quanto riguarda la dimensione della richiesta quanto appena detto non vale. Ad esempio, il tempo che si impiega a calcolare il numero di Fibonacci dipende dalla grandezza del parametro fornito al servizio. Quindi si poteva pensare di scegliere di utilizzare come stato i parametri delle richieste. Tuttavia, per un'altra applicazione che esegue un task completamente diverso si sarebbe dovuto cambiare il concetto di dimensione della richiesta. Quindi si è scelto di utilizzare la politica presentata in percedenza che tiene conto solamente del numero di richieste pendenti. Per decretare un server come fallito si è detto che si deve scegliere una tolleranza, ovvero, il numero di volte per cui un server che restituisce un errore viene comunque mantenuto nella lista dei nodi utilizzabili per servire la richiesta. Se un nodo restituisce un errore per un motivo diverso dal fallimento, esso è comunque attivo e quindi non invierà mai una richiesta al service registry per registrarsi. Quindi se il load balancer non gli da il giusto numero di possibilità, il server viene dimenticato e non viene più utilizzato per servire le richieste. Un aspetto simile al precedente è il tempo massimo per cui si attende che almeno una replica torni disponibile. Nel caso specifico si è scelto di utilizzare una tolleranza pari a 6 e un tempo massimo di attesa pari a un minuto. Ci si aspetta che un server non impieghi molto tempo a rispondere in quanto tutte le repliche sono sullo stesso nodo. Infine, un possibile sviluppo futuro è quello di distribuire su più istanza EC2 i conteiner in modo da creare un contesto ancora più realistico.

REFERENCES

[1] Amazon. [n. d.]. Amazon EC2. https://aws.amazon.com/it/pm/ec2/?gclid=CjwKCAjw3NyxBhBmEiwAyofDYQ1xTaGmyV2APhxTTHkvbp3h0XzV9_0XdypeYarZkJ07zEKG9JxayxoCxPkQAvD_BwE&trk=be8ffa73-d198-48a9-8728-2fdd4b8a06a0&sc_channel=ps&ef_id=CjwKCAjw3NyxBhBmEiwAyofDYQ1xTaGmyV2APhxTTHkvbp3h0XzV9_0XdypeYarZkJ07zEKG9JxayxoCxPkQAvD_BwE:G:s&s_kwcid=AL!4422!3!494972673739!e!!g!!ec2!12196406589!115425120925

[2] Gregory Detal, Christoph Paasch, Simon van der Linden, Pascal Mérindol, Gildas Avoine, and Olivier Bonaventure. 2013. Revisiting flow-based load balancing: Stateless path selection in data center networks. *Computer Networks* 57, 5 (2013), 1204–1216. https://doi.org/10.1016/j.comnet.2012.12.011

[3] Docker Inc. [n. d.]. What is a container. https://www.docker.com/resources/what-container/#:~:text=A%20Docker%20container%20image%20is,tools%2C%20system%20libraries%20and%20settings.

[4] Sudheer R Mantena. [n. d.]. Transparency in Distributed Systems.

[5] Jagruti Sahoo, Mohammad A. Salahuddin, Roch Glitho, Halima Elbiaze, and Wessam Ajib. 2017. A Survey on Replica Server Placement Algorithms for Content Delivery Networks. *IEEE Communications Surveys Tutorials* 19, 2 (2017), 1002–1026. https://doi.org/10.1109/COMST.2016.2626384