Integrazione Scritta

Gianluca Bresolin

1 RPCs: sincrono o asincrono?

In *Go*, il package "*net/rpc*" fornisce un sistema RPC che permette di invocare, attraverso la rete, i metodi esportabili di una struttura che seguono un preciso schema.
In particolare, l'invocazione di una RPC può avvenire mediante due modalità:

- **Call**: l'invocazione di una RPC avviene in modalità sincrona, ovvero il chiamante invia una richiesta al chiamato e attende una risposta. In questo caso il chiamante non può eseguire altre operazioni fino a quando non riceve la risposta dal chiamato;
- **Go**: l'invocazione di una RPC avviene in modalità asincrona, ovvero il chiamante invia una richiesta al chiamato e può continuare a eseguire senza attendere la risposta dal chiamato, risposta che può essere ricevuta in un secondo momento.

Sebbene le RPC siano intrinsecamente sincrone, nella distribuzione questo tipo di comunicazione non è ideale: la rete infatti non ci dà garanzie, con il rischio di incorrere in fallimenti e latenze, imponendo così dipendenze esterne al chiamante e al chiamato. Per limitare queste problematiche, nella nostra implementazione abbiamo optato per l'utilizzo di RPC invocate in modalità asincrona (ovvero mediante il metodo *Go*), in modo da permettere al chiamante di poter individuare, mediante un opportuno timeout inizializzato immediatamente prima dell'invocazione dell'RPC, eventuali problemi di rete o di latenza. In questo modo, nel caso in cui il timeout venga consumato prima di ricevere una risposta, il chiamante può effettuare un *retry* dell'invocazione della RPC mediante l'utilizzo di un protocollo *request-reply* di tipo *at least once*, ideale data l'idempotenza delle RPC in Raft.
Se questo approccio da un lato permette di far fronte ad eventuali problemi di rete (come ad esempio la perdita di pacchetti), non si può dire lo stesso nel caso in cui il chiamato sia in una situazione di sovraccarico: il chiamante, invocando la RPC in modo eccessivo, potrebbe infatti aumentare il carico di lavoro del chiamato, degradando ulteriormente le sue prestazioni. Una mitigazione a questo problema è individuata nell'utilizzo di un *backoff* incrementale, il quale permette di aumentare progressivamente il tempo di attesa tra un tentativo di invocazione e l'altro.

2 Cardinalità del cluster

Il cluster in Raft, inteso come l'insieme di nodi interconnessi tra loro, può contenere al suo interno sia nodi che partecipano attivamente all'elezione e alla replicazione delle entries del log, che nodi non votanti (i *non-voting nodes* appunto). Quest'ultimi nodi, in quanto tali, non vengono considerati per il raggiungimento del quorum a maggioranza assoluta.
A fronte delle precedenti precisazioni, quando si parla di cardinalità del cluster, specialmente in relazione alla maggioranza, ci riferiamo al numero di nodi che partecipano attivamente all'interno del cluster.
In Raft, la cardinalità del cluster può essere sia pari che dispari: l'algoritmo non impone infatti particolare restrizioni in merito al numero di nodi che partecipano attivamente all'interno del cluster. Un esempio di controllo per verificare il raggiungimento del quorum all'interno della nostra implementazione (in particolare per determinare il successo di elezione di un *candidate*), è infatti il seguente:

```
// rn.Config = map dei nodi che partecipano attivamente
if rn.votes >= int(len(rn.Config)/2) + 1 {
    // quorum raggiunto
    rn.winElection()
}
```

Attraverso questo controllo, in quanto *int()* restituisce il numero intero inferiore del valore passato come argomento, la maggioranza viene rilevata come la metà arrotondata per difetto del numero di nodi che partecipano attivamente nel cluster più uno, indipendentemente dal fatto che la sua cardinalità fosse pari o dispari.
Nonostante ciò, eseguendo una analisi in termini di tolleranza ai guasti, è possibile notare che un cluster con un numero dispari di nodi offre la stessa tolleranza di un cluster con un numero pari di nodi immediatamente successivo, al netto di un overhead maggiore. Più precisamente, il cluster di cardinalità pari richiede un quorum uguale ad un cluster avente cardinalità uguale al numero dispari immediatamente successivo (il quale a sua volta è però in grado di tollerare un guasto in più).
Un esempio di questa affermazione è il seguente:

- Un cluster di cardinalità 3 tollera 1 guasto e richiede un quorum di 2 nodi;
- Un cluster di cardinalità 4 tollera 1 guasto e richiede un quorum di 3 nodi;
- Un cluster di cardinalità 5 tollera 2 guasti e richiede un quorum di 3 nodi.

In conclusione, è consigliabile utilizzare un numero dispari di nodi per il cluster, in quanto questo permette di avere una tolleranza uguale a quella di un cluster di cardinalità pari immediatamente successiva, ma con un overhead minore.

3 Elezione con il Bully Algorithm

Come visto a lezione, un possibile algoritmo per l'elezione di un leader all'interno di un cluster è il *Bully Algorithm*, proposto da Hector Garcia-Molina nella sua pubblicazione *Distributed Computing System*[1].
A differenza della elezione proposta nel *Raft Paper*[2], il *Bully Algorithm* non richiede al *candidate* (o meglio, in un nodo che si trova nello status *election*, facendo riferimento alla terminologia utilizzata in *Distributed Computing System*[1]) di raggiungere un quorum a maggioranza assoluta per essere eletto leader (*coordinator* nella pubblicazione di riferimento).
L'idea alla base del *Bully Algorithm* è infatti che un nodo, a seguito di un timeout scaduto senza ricevere heartbeat dal *coordinator*, avvii una elezione contattando tutti i nodi del cluster aventi un identificativo maggiore del proprio: è importante notare come questo algoritmo richieda che a ciascun nodo sia associato un identificativo numerico univoco e che ogni nodo conosca tali identificativi di tutti i nodi del cluster. Una volta contattati i nodi con identificativo maggiore, ci sono due possibili esiti:

- Se nessuno dei nodi contattati risponde entro un certo timeout, il nodo che ha avviato l'elezione si autoproclama *coordinator* e comunica la sua elezione a tutti i nodi del cluster;
- Se uno dei nodi contattati risponde prima dello scadere del timeout, questo nodo si prende la responsabilità di portare a termine il tentativo di elezione, silenziando il nodo che lo aveva avviato (da qui il nome *Bully Algorithm*).

Nel caso in cui la richiesta raggiunga il nodo con identificativo maggiore all'interno del cluster, questo nodo diventa immediatamente il nuovo *coordinator* e comunica la sua elezione a tutti i nodi del cluster.
Il *Bully Algorithm* permette dunque di effettuare una elezione che non necessita una richiesta di voto a tutti i nodi del cluster (con conseguente attesa di un raggiungimento del quorum), come invece avviene nella nostra implementazione di Raft, fedele alla pubblicazione *Raft Paper*[2].
Come però evidenziato dallo stesso H. Garcia-Molina in *Distributed Computing System*[1], il *Bully Algorithm* garantisce l'elezione di un unico *coordinator* solamente sotto due assunzioni fondamentali: l'assenza di fallimenti nel sub-sistema di comunicazione e il fatto che i nodi del cluster non si fermino mai e rispondano sempre ai messaggi in arrivo senza ritardi. In particolare, l'assunzione in merito dell'assenza di fallimenti nel sub-sistema di comunicazione non è realista in un contesto pratico. Quando tale assunzione viene meno, possono verificarsi partizionamenti del cluster: in questa condizione, il *Bully Algorithm* non è in grado di garantire l'elezione di un unico *coordinator*. Una situazione del genere (che non è invece possibile mediante una elezione tramite quorum a maggioranza assoluta, in quanto garantisce sempre al più l'elezione di un solo leader) non è tollerabile in Raft, in quanto la presenza di più *coordinator* all'interno del cluster porterebbe a situazioni di inconsistenza del log.
A fronte di queste considerazioni, oltre che per la presenza di una documentazione più vasta e di un numero maggiore di implementazioni su cui poter

far riferimento, abbiamo deciso di implementare l'elezione del leader mediante l'algoritmo descritto nel *Raft Paper*[2].

4 Split vote

Come affermato nella risposta precedente, una elezione tramite il raggiungimento di un quorum a maggioranza assoluta garantisce sempre al più un'elezione di un solo leader per termine: ciò non significa però che ogni termine di elezione porti all'elezione di un leader. In quanto ogni nodo votante può votare al massimo per un solo candidato per termine, nel caso in cui due o più nodi inizino un'elezione in un momento simile, è possibile che si verifichi una situazione di *split vote*: in questo caso, nessun nodo candidato è in grado di raggiungere il quorum a maggioranza assoluta, portando così ad una situazione di stallo che necessita di un nuovo tentativo di elezione. Al verificarsi di una situazione di *split vote*, il cluster, non avendo un leader, non è in grado di elaborare nuove richieste di scrittura nel log, portando ad un periodo di interruzione del servizio. In una elezione gestita mediante il *Bully Algorithm*, una situazione di *split vote* non è possibile. In quanto tale algoritmo non richiede il raggiungimento di un quorum di voti, ogni tentativo di elezione viene infatti risolto con successo o dal nodo stesso o da un nodo con identificativo maggiore. Il *Bully Algorithm* garantisce dunque l'individuazione di almeno un nodo leader (*coordinator*) per ogni elezione avviata, limitando l'interruzione del servizio del cluster al solo periodo necessario per completare l'elezione.

5 Idempotenza delle RPC in Raft

Nel caso di arresti anomali di follower o candidati, le future *RequestVoteRPC* e *AppendEntriesRPC* inviate falliranno: in Raft, queste situazioni vengono gestite mediante nuovi tentativi indefiniti. Se un nodo dovesse subire un arresto dopo aver completato una RPC, ma prima di riuscire a restituire la risposta al chiamante, al riavvio riceverà una nuova richiesta della stessa RPC: da qui l'esigenza di idempotenza delle RPC, al fine di evitare situazioni di inconsistenza.
Per idempotenza si intende che l'applicazione di una RPC più volte produce lo stesso effetto di una sola applicazione della stessa RPC. Di conseguenza, una richiesta di voto effettuata più volte da un nodo porterà alla stessa risposta, ovvero il successo nel caso in cui il voto sia stato concesso e il fallimento nel caso in cui il voto non sia stato garantito. In merito ad *AppendEntriesRPC* contenenti log entries già presenti nel log del follower chiamato, queste verranno semplicemente ignorate nell'elaborazione della nuova richiesta ricevuta.
L'idempotenza ricopre un ruolo fondamentale per garantire la consistenza in Raft e viene raggiunta mediante meta-dati associati ad ogni RPC e alle entries nel log, quali il termine dell'elezione corrente in cui è stata effettuata l'RPC (nel caso di richieste di voto) o in cui è stata generata l'entry (nel caso di *AppendEntriesRPC*), l'identificativo del nodo chiamante e la posizione dell'entry nel

log.

6 RPC vs gRPC

In Go, per invocare metodi remoti, abbiamo due alternative principali: le RPC, fornite dal package standard *net/rpc*, e le *gRPC*, messe a disposizione dal package *google.golang.org/grpc*.
Mentre le RPC offerte da *net/rpc* utilizzando il package *encoding/gob* per la serializzazione e de-serializzazione dei dati, le *gRPC* si basano sul protocollo *Protocol Buffers* (o *protobuf*). Entrambi sono formati binari che presentano però differenze significative in termini di efficienza: *Gob*, nativo di Go, tende a produrre payload meno compressi in relazione a *protobuf*. Quest'ultimo è stato infatti progettato per essere altamente compatto, riducendo notevolmente la dimensione del payload e velocizzando i tempi di elaborazione. Un payload più leggero implica, inoltre, un minor consumo di banda nelle comunicazioni di rete e una ridotta necessità di spazio per l'archiviazione dei dati. Un confronto tra i due formati di serializzazione si può consultare nell'articolo di R. Shermeta *Benchmarking Gob vs Protobuf* [3], il quale afferma quanto affermato in precedenza.
Essendo inoltre *Protocol Buffers* un *interface definition languagge* (IDL), ovvero un linguaggio che consente di definire le strutture dati e i servizi in modo indipendente dal linguaggio di programmazione, le *gRPC* garantiscono un'elevata interoperabilità tra sistemi eterogenei, agevolando la comunicazione tra essi tramite la generazione automatica di codice client e server in diversi linguaggi. Questo è particolarmente utile per architetture distribuite e microservizi, dove la comunicazione tra componenti scritti in linguaggi di programmazione differenti è spesso necessaria.
In conclusione, rispetto alle RPC di *net/rpc*, le *gRPC* di *google.golang.org/grpc* garantiscono prestazioni superiori, unite ad una maggiore interoperabilità tra sistemi sviluppati in linguaggi differenti.

7 Condivisione di memoria

Il paradigma di concorrenza proposto da Go è basato sull'ideologia *share memory by communicating* e non *communicate by sharing memory*: a supporto di questa filosofia, il runtime di Go fornisce un sistema di comunicazione tra goroutine basato su canali (o *channels*), i quali permettono di inviare e ricevere dati in modo sincrono e thread-safe.
I vantaggi derivanti da una comunicazione basata su canali sincroni e thread-safe rispetto ad una condivisione di memoria si manifestano principalmente nello spostamento della responsabilità dal programmatore al runtime di Go. Questo approccio permette infatti di prevenire potenziali data race e di garantire accessi controllati alle sezioni critiche, riducendo così la necessità di utilizzare meccanismi espliciti di sincronizzazione come i mutex. L'uso di quest'ultimi, infatti,

espone il programmatore a complessità maggiori nella gestione della concorrenza, aumentando il rischio di deadlock e livelock, difficili sia da prevenire che da diagnosticare.
Sebbene l'idea alla base della concorrenza proposta da Go sia teoricamente valida e auspicabile, nella pratica ciò che emerge è un approccio in cui la complessità viene trasferita dalla gestione della memoria condivisa ad una progettazione che adatti lo scenario specifico all'uso dei canali, una visione progettuale che potrebbe non essere immediatamente naturale in tutti i contesti.
Le implicazioni di questo paradigma sono evidenti, con una centralizzazione logica per la gestione di risorse (che altrimenti sarebbero condivise) in goroutine dedicate, in quello che diventa un vero e proprio *actor model*. Nonostante quest'ultimo modello permetta una chiara separazione delle responsabilità, la complessità di progettazione che ne deriva può risultare superiore rispetto ad una più semplice ed immediata condivisione di memoria, in particolare nei casi in cui lo scenario di utilizzo non si adatti facilmente ad un *actor model*.
In conclusione, in quanto questo progetto consistesse in proof of concept, si è optato, qualora risultasse necessario, per l'utilizzo della condivisione della memoria mediante mutex, a discapito dell'uso di canali, al fine di mantenere una logica di sistema più semplice e dalla immediata comprensione.

References

[1] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 1982.

[2] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. *USENIX Annual Technical Conference*, 2014.

[3] Roman Shermuta. Benchmarking gob vs protobuf. https://rsheremeta.medium.com/benchmarking-gob-vs-protobuf-9dc36ea56ba4, 2023. Ultimo accesso: 5 Aprile 2025.