Application in Go for Distributed Mutual-Exclusion Algorithms

Mary Ann Vasquez Rodriguez
Università degli Studi di Roma Tor Vergata
Matricola: 0300954
Email: annerodriguez10@yahoo.it

***Abstract*—Lo scopo del progetto consiste nel realizzare, usando il linguaggio di programmazione Go, un'applicazione distribuita che implementa tre algoritmi di mutua esclusione: uno centralizzato e due decentralizzati. Offrendo anche un servizio di registrazione ai processi che vogliono partecipare al gruppo di mutua esclusione.**

1. Introduzione

Il problema della mutua esclusione distribuita prevede che un insieme di processi distribuiti che condividono una risorsa (o una collezione di risorse) possano accedervi senza interferenze. Nella teoria dei sistemi distribuiti, il caso di interesse è quello in cui la risorsa non è gestita da una singola macchina in locale, ma un insieme di nodi deve coordinare gli accessi alle risorse tra essi condivise, utilizzando un approccio basato sullo scambio di messaggi.

L'obiettivo di un algoritmo distribuito che soddisfa la proprietà di mutua esclusione (*safety*) è garantire che, in qualsiasi momento, al massimo un nodo può trovarsi nella sua sezione critica.

L'applicazione distribuita realizzata in Go, supporta l'esecuzione di 3 algoritmi di mutua esclusione: Token-Based (centralizzato), Ricart-Agrawala [1] e Lamport distribuito [2]. Inoltre, offre un servizio di registrazione al gruppo di mutua esclusione, in modo tale da identificare i nodi interessati.

Per il deployment dell'applicazione sono stati realizzati Docker container e in particolare, è stato usato il tool di *orchestrazione* Docker-Compose per la loro gestione. Infine, è stato effettuato il deployment anche su un'istanza EC2 di AWS.

2. Architettura applicazione

La logica dell'architettura riprende il classico paradigma Client/Server, in cui però, ogni terminale può fungere simultaneamente da client e da server, a seconda che stia rispettivamente richiedendo o fornendo un servizio necessario all'algoritmo.

E come già accennato precedentemente, il sistema in esame prevede di utilizzare lo scambio di messaggi come forma di comunicazione su cui basare tutte le interazioni. Di conseguenza, otteniamo un comportamento del tipo *request/reply*.

La scelta progettuale ricade sul Middlware RPC (*Remote Procedure Call*), per quanto riguarda la comunicazione tra i nodi del sistema. *RPC* è un protocollo di comunicazione per richiedere un servizio a un programma situato su un'altra macchina. Il vantaggio è che nessuna delle parti coinvolte nella comunicazione deve comprendere i dettagli di implementazione della controparte o del meccanismo di trasporto e si preoccupa solo dell'input e dell'output.

Come anticipato, l'applicazione supporta due diverse tipologie di algoritmi distribuiti: centralizzato e decentralizzato. Pertanto l'architettura di sistema, che sarà centralizzata o decentralizzata, si adatterà a runtime in funzione dell'algoritmo di mutua esclusione scelto per l'esecuzione.

Nel primo caso, c'è un singolo nodo centrale che gestisce le richieste di accesso alla sezione critica e si occupa di garantire la proprietà di safety. Questa soluzione offre sicuramente un accesso più rapido ed efficiente in termini di numero di messaggi scambiati, ma il nodo centrale rappresenta un SPoF e un collo di bottiglia per la scalabilità.

Nel secondo, non abbiamo alcun nodo centrale che gestisce l'accesso alla sezione critica e ogni nodo si fa carico delle operazioni che, nel modello centralizzato, erano demandate al sistema centrale: ricezione e gestione delle richieste di accesso, garanzia di safety. Tuttavia, questo comporta lo scambio di un numero di messaggi maggiore.

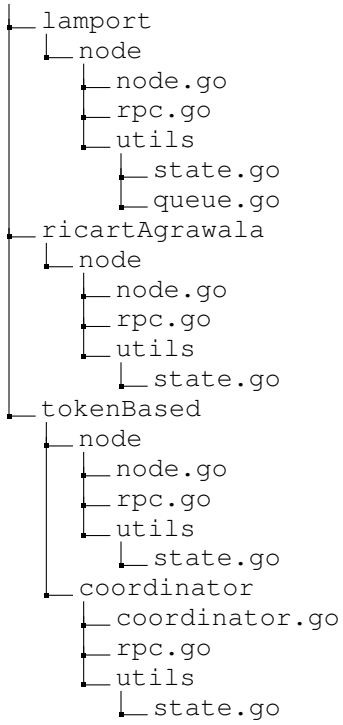
3. Implementazione

L'applicazione è stata realizzata in Go, un linguaggio di programmazione staticamente tipizzato e compilato, e che offre un supporto alla gestione della concorrenza tramite le Goroutine.

3.1. Organizzazione package in Go

Partiamo dall'analisi della struttura del progetto, facendo riferimento solo alle cartelle che contengono file sorgenti **.go*.

```
Distributed-MutualExclusion
├── main
│   └── main.go
├── go.mod
├── go.sum
└── alg
```



Nella cartella **alg/**, troviamo le logiche dell'applicazione per le funzionalità dei tre algoritmi. Ognuno di essi è stato implementato in modo indipendente rispetto agli altri, in modo da facilitarne la comprensione ed eventuali modifiche. Sono suddivisi in 3 cartelle: **lamport/**, **ricartAgrawala/**, e **tokenBased/**. E ogni cartella mantiene la stessa struttura; si ha una sottocartella **node/** (o **coordinator/**) con le logiche dei nodi per quel determinato algoritmo.

In particolare, in *node.go* (o *coordinator.go*) sono definite tutte le funzionalità relative al processo, come la join al gruppo di mutua esclusione, l'invio di una richiesta di accesso alla sezione critica o l'inizializzazione del server RPC. In *rpc.go* è riportata la definizione dei servizi offerti dal server, e quindi gli handler delle chiamate RPC, mentre in *state.go* sono mantenute le informazioni relative al processo come l'hostname, la porta del server RPC, il clock logico, ma anche quelle specifiche di un determinato all'algoritmo, insieme alle sue strutture dati: la coda delle richieste pendenti, il numero di acks ricevuti, il vettore di richieste già servite.

Infine nel file *main.go* troviamo l'entry point dell'applicazione, mentre in *go.mod* e *go.sum* sono contenute le indicazioni per la gestione delle dipendenze. Ora analizziamo singolarmente l'implementazione dei tre algoritmi e del servizio di registrazione al gruppo.

3.2. Registrazione al gruppo di mutua esclusione

Per il servizio di registrazione è stata utilizzata una specifica libreria Go, ***memberlist*** [3], che permette la gestione dell'appartenenza al cluster e anche il rilevamento dei guasti dei membri utilizzando un protocollo basato sul gossiping.

Il servizio è completamente decentralizzato, infatti ogni nodo inizializza una Memberlist e avvia i listener per consentire ad altri nodi di unirsi a questo gruppo. Per richiedere la Join, è sufficiente fornire un elenco di membri esistente che verranno contattati. Questo restituisce il numero di host contattati con successo e un errore se non è stato possibile raggiungerne nessuno. Se viene restituito un errore, il nodo non si è unito al cluster.

Una volta terminata la registrazione al cluster, ogni nodo sarà consapevole dell'esistenza degli altri membri, identificati dai loro hostnames (impostati come interi).
Per semplificare l'esecuzione dei test, un singolo nodo (*coordinatore* o *node0*) inizializzerà il cluster e le richieste di connessione saranno inviate solamente ad esso. Inoltre viene impostato un tempo di setup del gruppo, al termine del quale, potrà essere avviato l'algoritmo e ogni nodo potrà inviare la richiesta di accesso alla sezione critica.

L'uso di questa libreria non ha permesso di testare l'applicazione in *locale*, perchè la libreria utilizza l'hostname ed il suo IP per identificare un nodo.

3.3. Token-Based Centralizzato

Per l'algoritmo centralizzato Token-Based, sono state distinte due entità: il coordinatore e un nodo generico. Il coordinatore inizializza il gruppo di mutua esclusione e gestisce le connessioni per l'accesso al cluster. Al termine del tempo di setup, inizializza il suo *State* con la coda di richieste pendenti e una mappa *v* che tiene conto del numero di richieste servite per ogni nodo e infine inizializza il server RPC.

Come anticipato, per la comunicazione tra i processi è stato scelto il middleware RPC e Go offre un supporto ufficiale a RPC mediante la libreria *net/rpc*. Gli handler per le Call RPC inviate al coordinatore sono *AccessCS* e *ResendToken*.

La prima gestisce le richieste di accesso alla sezione critica e il payload della richiesta include l'id del sender (hostname) e il timestamp della richiesta; la funzione handler verifica se la richiesta è eleggibile e se il token è *free*, in caso positivo invia subito il Token come risposta, altrimenti la inserisce in coda nella lista delle richiesta pendenti *queue*.

La seconda gestisce il reinvio del token da parte dei nodi che sono usciti dalla loro sezione critica. Il coordinatore incrementa il contatore in *v* delle richieste servite, cerca la prossima richiesta eleggibile nella coda gestita tramite politica FIFO e, se presente, invia il token al sender della richiesta in questione.
In questa implementazione, il coordinatore non è interessato ad accedere alla sezione critica, pertanto non invia nè richieste di accesso alla CS, nè messaggi di programma.

La logica dei nodi interessati ad accedere nella CS è definita nella cartella **node/**. In questo caso si richiede la partecipazione al cluster inserendo come parametro nella funzione di join, il coordinatore. Una volta terminato il setup del gruppo, inizializza il server RPC e il proprio stato contenente il clock logico vettoriale.

Dopo un certo intervallo di tempo randomico, ogni nodo incrementa il proprio clock e invia una richiesta per accedere alla CS. Per mantenere i clock vettoriali sincronizzati, ogni nodo invia anche un *messaggio di programma* contenente il clock vettoriale a tutti gli altri membri.
Gli handler per le Call RPC, in questo caso sono: *Receive-Token* e *ProgramMessage*. Il primo serve per la ricezione del Token per indicare che la sua richiesta è diventata eleggibile; segue l'entrata nella sezione critica e il reinvio del Token. Il secondo processa i messaggio di programma, aggiornando eventualmente, il proprio clock vettoriale.

3.4. Lamport

In questo caso, l'algoritmo è completamente decentralizzato e ogni nodo assumerà lo stesso comportamento. Solo nella fase di setup del gruppo, le richieste di join al cluster verranno inviate al processo *node0*. Segue l'inizializzazione dello stato del processo che include il *clock logico scalare*, il *numero di acks* ricevuti dagli altri processi e una coda di priorità *queue*, ordinata in ordine crescente secondo la relazione di ordine totale delle richieste ricevute. La coda è gestita in modo tale che durante l'inserimento di una nuova richiesta nella lista, sia mantenuto l'ordinamento delle richieste. Questo facilita la ricerca della prossima richiesta che dovrà essere servita.

Dopo un certo intervallo di tempo randomico, ogni nodo incrementa il proprio clock logico, invia una richiesta per accedere alla CS a tutti i processi del gruppo e la inserisce nella coda. Attende la terminazione di ogni call RPC per ricevere i messaggi di Ack e incrementare il numero di acks ricevuti. Se la richiesta in testa alla coda è la sua e ha ricevuto tutti gli ack, il nodo entra nella sezione critica e alla sua uscita invia un messaggio di *Release* agli altri processi per informarli.

Gli handler per le Call RPC sono: *AccessCS* e *Release*. Il primo per gestire le richieste di accesso alla sezione critica, inserendo la richiesta nella coda locale e inviando un messaggio di Ack, e il secondo per processare i messaggi di Release. Alla ricezione di questi ultimi, ogni nodo ricerca nella propria coda, la prossima richiesta che dovrà essere servita, e se questa è la sua, verifica il numero di acks ricevuti ed eventualmente, entra in sezione critica.

3.5. Ricart-Agrawala

L'algoritmo di Ricart-Agrawala rappresenta un'estensione e un'ottimizzazione dell'algoritmo di Lamport distribuito, ed è sempre basato sul clock logico scalare e sulla relazione di ordine totale. L'implementazione è simile a quella precedente, ma si eliminano i messaggi di *Release* e non viene utilizzata una coda di priorità. Le richieste ricevute vengono inserite direttamente in una lista non ordinata. Una volta terminata la creazione del gruppo, che avviene in modo analogo all'algoritmo precedente, avviene l'inizializzazione dello stato che comprende il clock, il *numero di reply* ricevuti, il timestamp della richiesta inviata (*LastReq*), la lista di richieste ricevute e lo stato (NCS, CS, Requesting).

Gli handler per le Call RPC sono: *AccessCS* e *Reply*. Il primo per gestire le richieste di accesso alla sezione critica: verifica se $\{\text{LastReq, id}\} < \{\text{req.Timestamp, req.SenderId}\}$, in caso positivo la richiesta viene inserita nella lista, altrimenti si invia un messaggio di REPLY.

La seconda gestisce la ricezione di messaggi di Reply inviati dopo l'uscita dalla sezione critica. Si incrementa il numero di reply e se sono stati ricevuti tutti, il processo entra in CS e alla sua uscita, invia Reply a tutti i processi nella sua coda locale.

3.6. Main

Nell'entrypoint dell'applicazione si definiscono dei flag di input: *–alg, –mode, –node-id*. Il primo si aspetta tre stringhe che indicano gli algoritmi supportati, ovvero *token-centr, lamport e ricartagrawala*; il secondo è usato solo per l'algoritmo centralizzato per identificare il coordinatore e i nodi; il terzo indica l'id del nodo che deve essere un intero.

Inoltre si definiscono due parametri di configurazione: *portRPC* che indica la porta su cui sono in ascolto i server RPC, che per semplicità sono impostate tutte sulla 9090 e *time_setup*, che indica il tempo di attesa di ogni nodo prima che termini la creazione del gruppo di mutua di esclusione (impostato a 10 secondi).

3.7. Call RPC asincrone

Per la comunicazione tra i processi è stata preferita una comunicazione asincrona, per permettere ad ogni nodo di eseguire altre funzioni in attesa dei messaggi di risposta. La libreria *net/rpc* offre un supporto alle chiamate rpc asincrone, mediante la funzione *Client.Go*.

Tuttavia per una scelta implementativa, nei casi di Lamport e Ricart-Agrawala, per ogni messaggio di richiesta inviato, la goroutine incaricata di inviare le richieste, si pone in attesa delle risposte inviate dagli altri processi.

3.8. Clock logici

Per la sincronizzazione dei processi è stato sfruttato il concetto di *clock logico*; in particolare, quello *scalare* per gli algoritmi di Lamport e Ricart-Agrawala e quello *vettoriale* per quello Token-Based.

Per la gestione del clock logico scalare è stato utilizzato l'algoritmo di Lamport che prevede l'aggiornamento (clock = $\max\{\text{timestamp, clock}\}$) e l'incremento del clock alla ricezione e all'invio di ogni messaggio. A tal fine, ad ogni messaggio inviato è inserito un timestamp corrispondente al clock logico del processo sender.

Mentre per quello centralizzato, l'aggiornamento del clock vettoriale avviene solo all'occorrenza di due eventi: l'invio della richiesta e la ricezione del messaggio di programma. Il clock non viene aggiornato alla ricezione e al reinvio del token.

4. Limitazioni

Una limitiazione è stata riscontrata nell'uso della libreria *net/rpc*, che non consente agli handler delle call RPC, di accedere ai metadati delle richieste (ip, hostname..). Questo ha forzato l'inserimento all'interno dei messaggi, degli id (hostname) dei sender, per poter identificare i processi che hanno inviato i messaggi.

Un'altra limitazione è relativa all'algoritmo di Lamport: la sua dipendenza dalla comunicazione di tipo First-In-First-Out (FIFO). In fase di implementazione, è stato osservato che la differenza dei ritardi di trasmissione tra i processi può causare problemi alla mutua esclusione.

Ad esempio, supponiamo di avere un sistema distribuito con soli due processi. Se un processo con una coda vuota invia una richiesta e riceve una risposta, dovrebbe entrare nella sua sezione critica. Ma è possibile che il processo che ha inviato la risposta abbia inviato una richiesta in precedenza e a causa di un ritardo, la risposta ha preceduto la richiesta. Di conseguenza, entrambi i processi potrebbero entrare nella loro sezione critica.

Al contrario, per la natura dell'algoritmo di Ricart-Agrawala, non c'è alcuna restrizione su una comunicazione di tipo FIFO. Questo perché ogni processo è personalmente responsabile della concessione dei permessi e anche in caso di messaggi che subiscono ritardi di trasmissione, il messaggio di REPLY non verrà inviato grazie ai clock logici.

5. Deployment applicazione

Per il deployment dell'applicazione è stato richiesto di realizzare container Docker. Nella cartella **docker/**, della root del progetto, sono riportati tre Dockerfile per la creazione delle immagini dei nodi per ogni singolo algoritmo e tre file docker-compose.*.yml utilizzati per testare singolarmente gli algoritmi. Ciascuno inizializza 4 processi (container) con id: *0 (coordinator), 1, 2, 3* e i vari test sono stati eseguiti tra questi nodi. Ovviamente se si vuole ampliare l'enviroment, basta aggiungere ulteriori istanze nei diversi file *yml*.

Per la creazione delle immagini, è stato scelto l'approccio *multi-step build*: nel primo step si genera il binario eseguibile dell'applicazione, mentre nel secondo step si crea l'immagine contenente solo l'eseguibile. Questo ha permesso di realizzare images di dimensioni molto piccole.

Nella cartella **test/** invece sono salvati i file di log dei processi per ogni tipologia di test eseguito:

- test su richieste concorrenti:
- test con un singolo nodo che richiede l'accesso alla CS;
- test generico;

5.1. AWS EC2

L'applicazione è stata caricata su un'istanza EC2, dove sono stati installati *git*, *docker* e *docker-compose*. Una volta clonato il repository su git, è stato possibile eseguire l'applicazione con i diversi file docker-compose. Tuttavia l'applicazione non è accessibile da internet, ma è necessario collegarsi in ssh all'istanza EC2 e poi eseguire i comandi riportati nel file README per avviare l'applicazione multi-container.

6. Piattaforma, tool e librerie di supporto

Per la realizzazione del progetto è stata usato l'editor Visual Studio Code con l'estensione per il linguaggio Go. Mentre per la creazione delle immagini e il deployment è stata usata la piattaforma software Docker. Le librerie di supporto utilizzate sono:

- **net/rpc** [4] come supporto nella comunicazione tra i processi;
- **memberlist** [3] per il servizio di registrazione al gruppo;
- **vclock** [5] per la gestione dei clock logici vettoriali nell'algoritmo basato su token. Ogni clock è inizializzato come *map[string]uint64*, ovvero una mappa dove le chiavi sono gli id dei processi e i valori sono i rispettivi clock.

References

[1] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Communications of the ACM*, vol. 24, no. 1, pp. 9–17, 1981.

[2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[3] Memberlist package. https://pkg.go.dev/github.com/hashicorp/memberlist.

[4] net/rpc package. https://pkg.go.dev/net/rpc.

[5] Vclock package. https://pkg.go.dev/github.com/DistributedClocks/GoVector/govec/vclock.