Relazione SDCC

Algoritmi di mutua esclusione

Università degli studi di Roma Tor Vergata
Facoltà di Ingegneria Informatica

Valentina Falaschi 0295947

Anno Accademico 2021/2022

1 INTRODUZIONE

Scopo del progetto è quello di realizzare un' applicazione distribuita che implementi due algoritmi di mutua esclusione, nello specifico l'algoritmo di Lamport e quello di Ricart Agrawala.

2 PROGETTAZIONE

L'applicazione è stata progettata tramite l'utilizzo di:

1. **Go**, linguaggio di programmazione;
2. **Docker Compose**, tool per definire ed eseguire applicazioni Docker multi-container.

Nella figura 1 è rappresentata l'architettura del sistema, comune ad entrambi gli algoritmi.

Come si evince dalla figura si hanno due tipologie di container Docker:

1. Register, nodo che tiene traccia dei peer nel sistema;
2. Peer, nodo che richiede l'accesso alla sezione critica.

Di seguito sono analizzati gli aspetti comuni ad entrambi gli algoritmi.

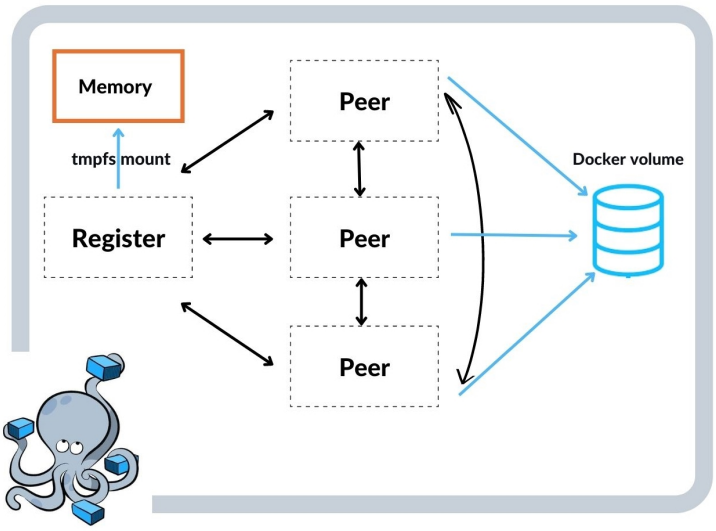

Figura 1: Architettura

Register

Il Register è il componente centralizzato le cui funzioni sono:

1. Rimanere in attesa delle connessioni di tutti i peer;
2. Salvare, su un file, per ogni peer registrato, l'indirizzo IP e numero di porta;
3. Condividere le informazioni registrate con ciascun peer, così che ognuno di essi abbia conoscenza delle rete.

Il file relativo al salvataggio delle informazioni è memorizzato in memoria, tramite l'impiego di *tmpfs*, il cui utilizzo viene approfondito nella sezione relativa all'orchestrazione di container.

Peer

I peer implementano gli algoritmi di mutua esclusione, ciascuno di essi possiede:

1. Indirizzo IP;
2. Numero di porta;
3. Id univoco;
4. Valore del clock logico scalare.

E svolge le seguenti funzioni:

1. Registrazione al nodo register;
2. Ricezione e invio di messaggi verso gli altri nodi peer della rete.

L'id univoco viene associato a ciascun peer poiché, in entrambi gli algoritmi analizzati, è fondamentale per discernere il peer, che per primo accede alla sezione critica in un contesto di parità del valore del clock logico scalare.

Orchestrazione container

I nodi descritti nell'architettura vengono implementati e gestiti tramite Docker Compose, il cui obiettivo è quello di far comunicare tra loro molteplici container in esecuzione sulla stessa macchina. Attraverso il file *.yml* viene:

1. Definita una rete virtuale tramite cui i container possono comunicare;
2. Eseguita la build di ciascun container tramite il DockerFile;
3. Associato un volume a ciascun nodo peer e un tmpfs al nodo register.

Nello specifico, per quanto riguarda la gestione della persistenza, vengono usati i volumi al fine di permettere la condivisione del file *log.txt* tra i diversi nodi peer. Al termine dell'esecuzione dei container è possibile visionare il contenuto di questo file e conseguentemente verificare il corretto accesso dei nodi alla risorsa condivisa. Tale verifica viene approfondita successivamente.
D'altro canto, il file relativo alla registrazione dei nodi peer è un file che non necessita di essere visualizzato al termine dell'esecuzione, per questo motivo si è scelta l'opzione del tmpfs, poiché permette di eliminare automaticamente il file nel momento in cui i container vengono stoppati.

Comunicazione tra nodi

Si hanno due tipi di comunicazione:

1. Chiamata a procedura remota per la registrazione dei peer verso il register;
2. Comunicazione diretta tra peer tramite package *net* di Go.

Il linguaggio Go offre un supporto nativo per le chiamate di procedura remota offerto nel package *net/rpc*. Nello specifico il Register invoca la funzione *RegistryName*, la quale consente di registrare come remoto il metodo che permette di salvare l'indirizzo del peer sul file temporaneo precedentemente citato. Successivamente il register, che in questo contesto può essere visto come server, si mette in ascolto di chiamate provenienti da client remoti (peer) usando i comandi di *listen* e *accept*. Per quanto riguarda la comunicazione tra peer, questi si scambiano messaggi la cui struttura si compone dei seguenti campi:

1. Type, tipo del messaggio;
2. Clock, valore del clock logico scalare associato al messaggio al momento dell'invio;
3. Text, testo del messaggio;
4. SendID, id del nodo che invia il messaggio.

Ogni volta che un nodo peer riceve un messaggio viene eseguita una specifica azione in base al campo *type*.

Sezione Critica

La sezione critica è implementata tramite lo *sharedFile.txt*, un file, come descritto precedentemente, comune a tutti i nodi peer. Ogni componente della rete può accedere alla risorsa condivisa in mutua esclusione e, una volta acceduto, scrivere sul file:

1. un messaggio (una qualsiasi stringa che può essere scelta dall'utilizzatore dell'applicazione);
2. Id del peer;
3. valore del clock logico scalare.

In particolare, gli ultimi due valori permettono di verificare che effettivamente i messaggi vengano scritti in ordine rispetto al valore del clock logico scalare e, in caso di parità, in ordine rispetto all'id del nodo. Per ogni peer viene

lanciata una go routine che ha il compito di verificare, ciclicamente, che le condizioni necessarie per accedere alla sezione critica siano soddisfatte. Nel momento in cui si verificano le condizioni viene consentito l'accesso al file condiviso.

3 ALGORITMI

Di seguito è descritta l'implementazione specifica per ciascun algoritmo.

Lamport distribuito

L'implementazione di questo algoritmo prevede l'uso di clock logici scalari e di una coda locale per ciascun peer, utilizzata per mantenere le richieste di accesso alla sezione critica ricevute dagli altri processi.
Di seguito sono descritte le funzioni implementate dai nodi peer:

1. Invio di messaggi, si distinguono in:
   1.1. Messaggio di *reques*t per l'accesso alla sezione critica, inviato a tutti i nodi peer della rete;
   1.2. Messaggio di *reply*, inviato al peer che ha fatto richiesta per l'accesso alla sezione critica;
   1.3. Messaggio di *release*, inviato a tutti i peer della rete in seguito all'uscita di un peer dalla sezione critica.

2. Ricezione di messaggi, si distinguono in:
   2.1. Messaggio di request, comporta l'inserimento del messaggio nella coda e l'invio del messaggio di reply al peer che aveva inviato la richiesta.
   2.2. Messaggio di reply, comporta l'incremento del numero di ack associati al messaggio di richiesta;
   2.3. Messaggio di release, comporta l'eliminazione del messaggio di richiesta dalla coda locale del peer.

3. Controllo ciclico sulle condizioni necessarie per accedere alla sezione critica.

Per ogni invio del messaggio di *request*, il clock logico scalare del peer viene incrementato di una unità in maniera atomica, tramite il *mutex.Lock*. Mentre ad ogni ricezione del messaggio di *request* il peer incrementa di una unità il massimo tra il suo valore del clock e il valore del clock associato al messaggio di richiesta. La gestione relativa agli ack prevede una mappa che ha come chiave il messaggio di richiesta di accesso alla sezione critica e come valore il numero di messaggi di *reply* ricevuti per quella richiesta.

Questo algoritmo soddisfa le proprietà di mutua esclusione, in particolare soddisfa la *safety* perché garantisce che un solo processo alla volta entri in sezione critica, la *liveness* perché non ci sta starvation ed è anche *fair* poiché le richieste vengono servite secondo un ordine temporale determinato dal clock logico. Tuttavia non garantisce ordering, perché tra due richieste arrivate in tempi diversi, che hanno stesso timestamp, vince la richiesta inviata dal peer con id più piccolo.

Ricart Agrawala

Questo algoritmo ha molte caratteristiche in comune con l'algoritmo di Lamport, tuttavia si evidenziano le seguenti differenze:

1. Ogni peer ha uno parametro aggiuntivo che indica lo stato, il quale può essere:
   1.1. *cs*, il peer *i* è in sezione critica. In questo caso qualsiasi altra richiesta di accesso alla sezione critica viene salvata nel coda locale del peer i-esimo;
   1.2. *ncs*, il peer non è in sezione critica e non è interessato ad accedere alla risorsa. In questo caso viene mandato un messaggio di *reply* per qualsiasi richiesta di accesso alla sezione critica proveniente dagli altri peer e tali richieste non vengono salvate nella coda locale;
   1.3. *request*, il peer è interessato ad accedere alla sezione critica ed attende che siano verificate le condizioni per accedervi.

2. Il messaggio di reply viene inviato dal peer *i* al peer *j* soltanto nel caso in cui

il peer i si trova nello stato *ncs* oppure si trova nello stato di *request* ma vale la relazione di ordine totale per cui il valore del clock del messaggio i è maggiore del valore del clock del messaggio j, oppure, in caso di parità, $i > j$;

3. Il peer uscito dalla sezione critica invia il messaggio di release, non a tutti i peer della rete, ma soltanto a quei peer che si trovano nella sua coda locale;
4. Il valore del clock viene incrementato solo in corrispondenza dell' invio di messaggi e non in corrispondenza delle ricezioni;
5. Un peer accede alla sezione critica solo nel momento in cui ha ricevuto un messaggio di reply da tutti gli altri peer della rete;

Ricart Agrawala rispetto all'algoritmo di Lamport permette di risparmiare il numero di messaggi scambiati. Questo aspetto viene discusso nella sezione relativa ai Test.

4 TEST

I test svolti hanno permesso di verificare sia il numero di messaggi scambiati in ogni algoritmo, sia la correttezza dell'ordine con cui i peer accedono alla sezione critica. In particolare si è simulato uno scenario in cui ogni peer invia una richiesta di accesso alla sezione critica, e si è tenuto traccia del numero di messaggi scambiati da ogni nodo. In questo modo è stato possibile provare che per Lamport vengono scambiati esattamente 3(N-1) messaggi, dove N è il numero di nodi peer, mentre per Ricart Agrawala vengono scambiati 2(N-1) messaggi. Inoltre tramite il file condiviso *log.txt*, come anticipato precedentemente, è stato possibile verificare anche che l'accesso alla sezione critica venisse eseguito secondo il giusto ordine. Ogni peer che accede al file infatti scriverà su di esso, oltre ad un messaggio testuale, anche il suo id e il valore del clock logico scalare associato al messaggio. Durante la fase di test i messaggi di richiesta vengono inviati con un ritardo, generato ogni volta in maniera randomica, al fine di simulare ritardi nella rete.

Indice