

Sistemi Distribuiti e Cloud Computing - A.A. 2021/22

Progetto B2: Algoritmi di mutua esclusione distribuita in Go

Martina De Maio

Corso di laurea magistrale in Ingegneria Informatica
Università degli studi di Roma "Tor Vergata"
martina.demaio@alumni.uniroma2.eu

Abstract—La seguente trattazione descrive la progettazione di un'applicazione distribuita che implementa tre algoritmi di mutua esclusione distribuita. Si riporta una descrizione degli algoritmi sviluppati, dell'architettura del sistema implementato e le relative scelte progettuali.

1. INTRODUZIONE

Lo scopo del progetto è realizzare nel linguaggio di programmazione Go un'applicazione distribuita che implementi 3 algoritmi di mutua esclusione distribuita. L'applicazione deve soddisfare i requisiti elencati di seguito:

- Offrire un servizio di registrazione dei processi che partecipano al gruppo di mutua esclusione. Si assume che la membership al gruppo di processi sia statica durante l'esecuzione dell'applicazione, quindi che non vi siano processi che si aggiungano al gruppo od escano dal gruppo durante la comunicazione.
- Supportare l'esecuzione di:
 - 1) un algoritmo di mutua esclusione **centralizzato** tramite un **coordinatore**;
 - 2) due algoritmi di mutua esclusione **distribuita** a scelta.

Per il deployment dell'applicazione si richiede di utilizzare container Docker, di fornire i relativi file per la creazione delle immagini e di effettuare il deployment dell'applicazione su una istanza EC2. Inoltre, come previsto dai requisiti progettuali, viene effettuata un'attività di testing sugli algoritmi implementati nel caso in cui vi sia un solo processo che richiede l'accesso alla risorsa condivisa e nel caso in cui molteplici processi richiedano contemporaneamente l'accesso alla risorsa condivisa; tali test devono essere forniti nella consegna del progetto. Per il debugging, si raccomanda di implementare un flag di tipo `verbose` che permette di stampare informazioni di logging con i dettagli dei messaggi inviati e ricevuti. Inoltre, per effettuare il testing in condizioni di maggiore stress, si consiglia di includere nell'invio dei messaggi un parametro `delay` che permette di specificare un ritardo, generato in modo random all'interno di un intervallo temporale predefinito.

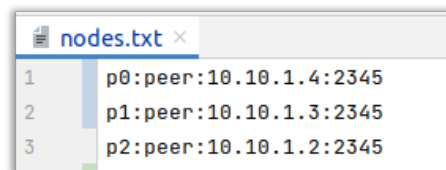
Gli algoritmi implementati sono:

- 1) algoritmi di **Lamport distribuito** e **Ricart-Agrawala**: algoritmi basati su **autorizzazioni** in cui un processo che vuole accedere alla sezione critica (CS) chiede l'autorizzazione, in questo caso gestita in modo completamente distribuito
- 2) algoritmo **token-asking**: algoritmo centralizzato basato su token, secondo il quale tra i processi circola un messaggio speciale, il token, unico in ogni istante di tempo, che viene gestito dal coordinatore. Solo chi possiede il token può accedere in CS.

2. ARCHITETTURA

Per il deployment dell'applicazione sono stati utilizzati i container Docker, configurati tramite **Docker Compose**, utile a definire ed eseguire applicazioni multi-container di tipo Docker. Docker Compose infatti consente di definire e condividere applicazioni Docker multi-container tramite un file yaml denominato `docker-compose.yml`, nel quale si specifica la configurazione dei servizi della propria applicazione, e con un solo comando si creano e avviano tutti i servizi a partire dalla configurazione specificata. Per implementare correttamente l'applicazione sono state sviluppate 3 tipologie differenti di nodi:

- 1) Nodo **register**: implementa il servizio di registrazione dei processi che partecipano al gruppo di mutua esclusione. Tale nodo viene contattato da tutti i peer nella fase di startup, rimane in attesa delle connessioni di tutti e, al raggiungimento del numero massimo di peer, restituisce ad ognuno di essi l'elenco dei partecipanti al gruppo e salva sul file `nodes.txt`, contenuto all'interno di un volume Docker, gli indirizzi e gli ID dei peer registrati, come mostrato nella figura successiva.



```
nodes.txt
1 p0:peer:10.10.1.4:2345
2 p1:peer:10.10.1.3:2345
3 p2:peer:10.10.1.2:2345
```

Figure 1. Esempio di file relativo alla registrazione dei peer.

- 2) Nodo **coordinatore**: nodo che implementa il ruolo di coordinator previsto dall'algoritmo token-asking. Gestisce il token che circola all'interno del sistema, riceve i messaggi di REQUEST dai peer e dà loro il token, tramite un TOKEN message, quando i messaggi diventano eleggibili.
- 3) Nodo **peer**: nodo partecipante al gruppo di mutua esclusione, che si scambia messaggi con gli altri partecipanti. In base all'algoritmo scelto, si crea un'entità *Peer* tramite una struct avente campi necessari per implementare i vari algoritmi.

Per ognuno di questi nodi è stato istanziato un container Docker con il relativo volume, all'interno del quale vengono salvati i file che riportano i messaggi scambiati e gli aggiornamenti dei clock logici. Nel docker-compose.yml sono presenti le immagini dei container da avviare, e, in particolare, per i peer si imposta manualmente lo scaling desiderato andando a modificare il file *yml*.

L'architettura utilizzata per la realizzazione dei vari algoritmi è mostrata nelle figure 2 e 3 in seguito riportate.

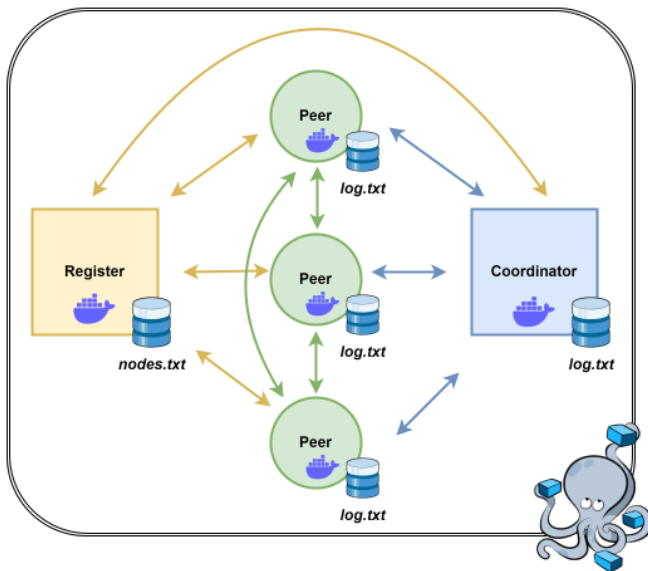


Figure 2. Architettura algoritmo token-asking.

Per il deployment dell'applicazione è stata utilizzata un'istanza Linux di **Amazon EC2** (Elastic Computing Cloud) di tipo **t2.large**, tipo di istanza per uso generico e a basso costo che offre un livello base di prestazioni della CPU, con la possibilità di espanderne la potenza in base alle proprie esigenze. Le caratteristiche di un'istanza t2.large sono le seguenti:

- 2 vCPU,
- 8 GiB di RAM,
- processore ad alta frequenza *Intel Xeon* scalabile fino a 3,0 GHz (*Haswell E5-2676 v3* o *Broadwell E5-2686 v4*),

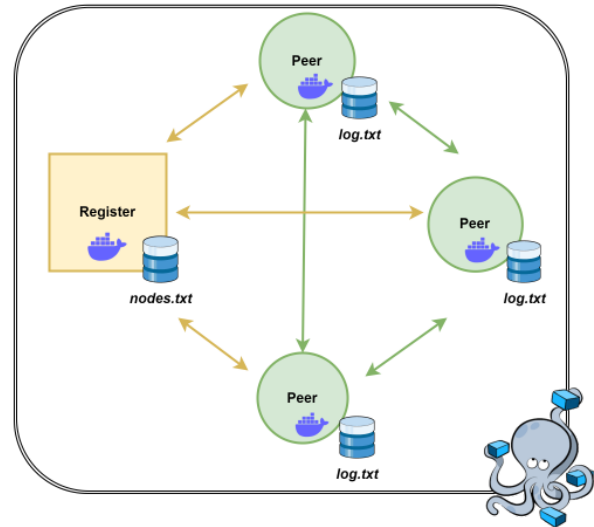


Figure 3. Architettura algoritmi Lamport e Ricart-Agrawala.

3. ORGANIZZAZIONE DEL PROGETTO IN PACKAGE

Il progetto è stato organizzato in package, costituiti da diversi file sorgenti *.go*:

3.1. Package coordinator

Contiene il codice per eseguire il peer che svolge il ruolo di coordinatore nell'algoritmo token-asking. In questo package è presente un solo file, che contiene il main per tale nodo.

3.2. Package peer

Contiene il codice per iniziare l'esecuzione di ogni peer, che avviene all'interno del main contenuto all'interno del file *main.go*. In tale package è presente il file *menu.go*, che contiene il codice per il frontend di un peer. Una volta che il peer si è registrato, infatti, l'utente può interagire con il programma e specificare quale algoritmo si vuole eseguire, tramite un menu il cui codice è stato prelevato dalla repository github <https://github.com/manifoldco/promptui> e opportunamente modificato per gli scopi del progetto. *Promptui* è una libreria che fornisce una semplice interfaccia per creare prompt interattivo della riga di comando per go. Il package corrente è suddiviso in 3 ulteriori subpackage, uno per ogni algoritmo implementato.

3.3. Package register

Package per il codice del nodo register, in cui è presente un unico file nel quale è presente il *main*, da cui inizia l'esecuzione del nodo register.

3.4. Package utilities

Package che contiene strutture dati e funzioni di ausilio per l'esecuzione dell'applicazione. Di notevole importanza è il file *configurations.go*, all'interno del quale si possono impostare i vari parametri necessari per l'esecuzione dell'applicativo.

4. DESCRIZIONE DELL'IMPLEMENTAZIONE

Di seguito si andranno ad indicare le varie scelte implementative e il funzionamento del progetto realizzato.

4.1. Scelta dell'algoritmo e dei test

Una volta avviati i container, tramite riga di comando l'utente può decidere lo username per ogni peer, e successivamente si avvia la registrazione tramite il nodo register. Ogni peer dovrà attendere che tutti gli altri peer si siano registrati. Una volta registrati, si apre un menù tramite il quale l'utente può decidere quale algoritmo avviare (è importante che si scelga lo stesso algoritmo per ogni peer). Mentre il register effettua sempre le stesse operazioni, i peer possono essere eseguiti in modalità test o meno a seconda di come è impostato il parametro `Test` nel file `configurations.go` all'interno del package `utilities`. Quando non viene avviato in modalità test, una volta terminata la registrazione, si può scegliere quale algoritmo far implementare ai peer tramite il menu. Una volta scelto l'algoritmo che si vuole avviare, viene aperto in un loop un altro menu tramite il quale l'utente può scegliere se far inviare un messaggio di REQUEST al processo:



Figure 4. Menu iniziale.

Anche la modalità test prevede l'interazione con l'utente, il quale anche questa volta può decidere che tipi di test effettuare tramite il menu, scegliendo l'algoritmo e il numero di processi che inviano contemporaneamente il messaggio di REQUEST:

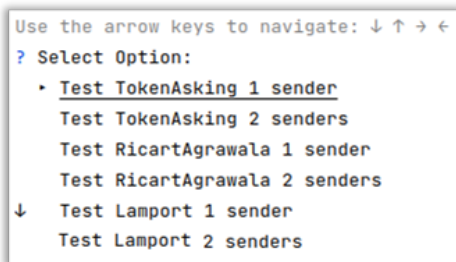


Figure 5. Menu test.

Per i peer partecipanti al gruppo multicast sono state create 3 *struct*, a seconda del tipo di algoritmo che si sta utilizzando, per raccogliere in una singola entità tutti i campi di interesse per lo specifico peer che servono per l'implementazione dei vari algoritmi. Le analogie tra queste *struct* sono i campi `Username`, `ID`, `Address`, `Port` (per identificare il peer), un oggetto `sync.Mutex` e la lista dei peer coinvolti nel gruppo di mutua esclusione; le differenze, invece,

riguardano prevalentemente il tipo di timestamp utilizzato per etichettare i messaggi, in quanto per il token-asking si utilizza un **clock logico vettoriale**, mentre per gli altri 2 si usa un **clock logico scalare**, e nelle strutture dati utilizzate per memorizzare i vari messaggi scambiati.

4.2. Comunicazione tra i nodi

4.2.1. Fase di registrazione: Remote Procedure Call.

Nella fase di registrazione si utilizza il meccanismo delle RPC, tramite cui i nodi effettuano una chiamata a una procedura remota situata nel nodo register. Il peer che vuole effettuare la registrazione, una volta specificato il proprio username (che l'utente può scegliere tramite riga di comando), rimane in attesa che venga completata la registrazione per tutti i peer partecipanti, il cui numero viene impostato manualmente all'interno del file `configuration.go`. Per implementare l'attesa è stato utilizzato un contatore particolare, ossia `sync.WaitGroup`, che può essere manipolato in modo sicuro da più **goroutine** e permette di rimanere in attesa finché non diventa zero (tramite `Wg.Wait()`). Il contatore viene incrementato ogni volta che un peer effettua la registrazione, e, quando la registrazione di tutti i peer è terminata, il nodo register chiama `Wg.Add(-PEER_NUMBER)`, per settare a 0 il valore del contatore e quindi sbloccare l'attesa dei peer. Terminata la registrazione, il peer con il relativo username e indirizzo viene salvato sul file `nodes.txt` nel volume del nodo register, e viene fornita a tutti i peer la lista dei partecipanti. A questo scopo si è utilizzato il package `net/rpc` [1]

4.2.2. Comunicazione tra i peer. Per la comunicazione tra i peer, quindi per lo scambio di messaggi tra essi, si è utilizzato il package `net` di Go [2], che fornisce un'interfaccia per l'I/O di rete inclusi TPC/IP, UDP. È stata utilizzata l'interfaccia di base fornita dalle funzioni `Dial`, `Listener`, `Accept` e le interfacce associate `Conn` e `Listener`, tramite le quali è stato possibile l'invio e la ricezione dei messaggi. Prima dell'invio di un messaggio, viene applicato un **delay** casuale per simulare ritardi nella rete, implementato tramite una `Sleep` per un tempo random che va dai 300ms ai 2000ms. Per i messaggi vengono create due *struct*, una per l'algoritmo del token-asking e una per gli altri due algoritmi, che differiscono per il tipo di timestamp con il quale vengono etichettati i messaggi e per il tipo di messaggi che vengono scambiati.

Per gestire la ricezione e l'invio dei messaggi è stato utilizzato il package `gob` [3], tramite il quale si gestiscono i flussi di gob - valori binari scambiati tra un `Encoder` (trasmettitore) e un `Decoder` (ricevitore). Lato sender si crea un `Encoder` e si codifica il messaggio tramite `enc := gob.NewEncoder(conn)` e `enc.Encode(msg)`, lato receiver si utilizza un `Decoder` per decodificarlo, tramite `dec := gob.NewDecoder(conn)` e `dec.Decode(msg)`, per recuperare i valori dallo stream codificato e decomprimerli in variabili locali. Ogni peer utilizza un'unica porta per ricevere i messaggi, e tutti

i messaggi, sia ricevuti che inviati, a seconda che il flag `verbose` sia attivo o meno, vengono salvati all'interno di un file `"peer_id.log"`, uno per ogni peer, all'interno di volumi Docker. Esempi di tali file si trovano nella repository github nella cartella `src/peer/volumes/logs`.

4.3. Implementazione degli algoritmi

4.3.1. Token-asking. Per la gestione del token la struct rappresentativa di ogni peer, compreso il coordinatore, possiede un campo booleano `HasToken`, pari a `true` se il peer possiede il token, `false` altrimenti. Si ricorda che il token è unico all'interno del sistema quindi solo un peer alla volta possiederà il token. Allo startup, il token è posseduto dal coordinatore, che lo gestisce e lo dà ai peer una volta che le loro richieste risultano eleggibili. Il processo che vuole inviare un messaggio di REQUEST incrementa di 1 la propria componente del clock logico vettoriale e invoca il metodo `NewRequest(sender string, date string, vc VectorClock)` per creare un messaggio di richiesta (in cui allega il proprio clock vettoriale) da inviare al coordinatore. Il peer crea anche un messaggio di PROGRAMMA tramite `NewProgramMessage(sender string, date string, vc VectorClock)`, che viene inviato agli altri peer partecipanti per far sì che essi abbiano il valore del clock vettoriale aggiornato. Alla ricezione del messaggio di programma, il peer aggiorna il proprio clock logico vettoriale ponendolo al massimo, componente per componente, tra il proprio clock logico vettoriale e quello del messaggio ricevuto $VC = \max(VC, msg.VC)$. Una volta ricevuto il messaggio di REQUEST il coordinatore controlla se esso è eleggibile: detto V il clock vettoriale del coordinatore, una richiesta inviata da p_i è eleggibile se il suo timestamp è al più pari a V per le componenti $j \neq i$. Quando una richiesta è eleggibile, il coordinatore aggiorna il proprio clock vettoriale incrementando di 1 la componente relativa al processo che ha ricevuto il token, e tramite il metodo `NewTokenMessage(date string, sender string, receiver string, vc VectorClock)` crea un TOKEN message che verrà inviato al peer che ha mandato la richiesta, per informargli che la sua richiesta è eleggibile e dargli l'accesso al token. Una volta ricevuto il token, il peer entra in sezione critica e all'uscita invia un TOKEN message al coordinatore per restituirgli il token. Il coordinatore possiede una `ReqList`, un tipo di dato `*list.List` nel quale salva richieste pendenti, ossia quelle richieste ricevute ma non ancora servite. Per far sì che il sistema sia asincrono, dopo l'invio del messaggio di REQUEST viene lanciata una goroutine per messaggio, che verifica ciclicamente che le condizioni previste per entrare in sezione critica vengano rispettate. Il codice sorgente relativo a tale algoritmo si trova nella cartella `src/peer/tokenAsking` della repository GitHub.

4.3.2. Algoritmo di Lamport distribuito. Ogni processo mantiene un clock logico scalare, inizialmente inizializzato a 0, ed una coda locale nella quale memorizza le richieste di accesso alla CS. Si applica anche la relazione di

ordine totale \Rightarrow , per far sì che se ci sono 2 richieste di accesso in CS che hanno lo stesso valore del clock logico scalare vince quella con id del processo più piccolo. Per implementare la coda locale è stata utilizzata una `map[ScalarClock][]Message`, ossia una mappa avente come chiave il clock logico scalare, che è un tipo di dato `int`, e come valore una lista di messaggi di REQUEST aventi come timestamp quella chiave. L'inserimento all'interno di tale mappa avviene in modo ordinato, ossia per ogni chiave viene inserito all'inizio della lista relativa il messaggio avente id del sender minore, in questo modo si rispetta \Rightarrow . In questo modo, prendendo il primo elemento della mappa, a parità di timestamp, si assicura che venga preso il messaggio avente timestamp e id del sender minore. La struct relativa al peer possiede anche un campo `replySet`, una `*list.List` che contiene i messaggi di REPLY ricevuti dal peer. Il peer che vuole inviare un messaggio di REQUEST incrementa di 1 il proprio clock logico scalare, crea il messaggio tramite `NewRequest(sender string, date string, timestamp ScalarClock)`, lo invia agli altri peer partecipanti e lo aggiunge alla sua coda locale. Il peer che riceve il messaggio di richiesta lo memorizza all'interno della sua coda e invia al mittente del messaggio un messaggio di REPLY, creato tramite `NewReply(sender string, receiver string, date string, timestamp ScalarClock)`, e aggiorna il proprio valore del clock logico scalare ponendolo al massimo tra il proprio clock e il timestamp del messaggio ricevuto. Tramite una goroutine il mittente del messaggio di richiesta controlla ciclicamente se le condizioni per entrare in CS sono verificate, ossia se:

- la sua richiesta con timestamp t precede tutti gli altri messaggi di richiesta in coda (ossia presenti all'interno della coda (ossia t è il minimo applicando \Rightarrow))
- il processo ha ricevuto da ogni altro processo un messaggio (di ack o nuova richiesta) con timestamp maggiore di t

Grazie alle strutture dati utilizzate il peer dovrà semplicemente controllare se ha ricevuto tutti i messaggi di REPLY e se il primo messaggio nella sua coda locale è il suo. In caso affermativo, entra in CS, e all'uscita invia un messaggio di RELEASE, creato tramite `NewRelease(sender string, date string, timestamp ScalarClock)`, a tutti gli altri processi, i quali eliminano la richiesta corrispondente dalla loro coda. Il codice relativo a tale algoritmo si trova nella cartella `/src/peer/lamport` nella repository Github.

4.3.3. Algoritmo di Ricart e Agrawala. È un'estensione ed ottimizzazione dell'algoritmo di Lamport distribuito, basato anch'esso su clock logico scalare e relazione d'ordine totale. Rispetto all'algoritmo precedente si elimina il messaggio di RELEASE, ed il messaggio di REPLY è differito all'uscita della sezione critica. In questo algoritmo viene utilizzata la stessa struct per il messaggio che viene usata in Lamport; i

metodi per la creazione dei messaggi sono quindi gli stessi e non vengono riportati. Per quanto riguarda la struct usata per il peer, a differenza di Lamport qui il peer possiede un campo *State*, che può essere *Requesting*, *CS* o *NCS*, che descrive se il processo sta rispettivamente chiedendo l'accesso alla CS, si trova già in CS o non in CS. Per controllare se il peer può accedere in CS viene mantenuto un contatore *replies* che tiene conto del numero di messaggi di *REPLY* ricevuti. Il peer possiede anche una lista *DeferSet*, in cui memorizza le richieste pendenti, un campo *num* che rappresenta il clock logico scalare e un campo *LastReq* che corrisponde al timestamp del messaggio di richiesta. Il peer che vuole accedere in CS pone il suo stato a *REQUESTING*, incrementa di 1 il valore di *num*, pone *LastReq=num* e invia il messaggio di richiesta agli altri peer, e tramite una goroutine controlla ciclicamente che il numero di *REPLY* ricevute è pari al numero degli altri peer presenti nel sistema. Una volta entrato in CS pone il suo stato pari a *CS*, e all'uscita invia un messaggio di *REPLY* a tutti i peer nella coda locale. Il peer che riceve un messaggio di richiesta, aggiorna il proprio clock logico scalare ponendolo al massimo tra il proprio e quello ricevuto, e se il suo stato è *CS* o se lo stato è *REQUESTING* e il suo timestamp è inferiore di quello ricevuto (applicando \Rightarrow), mette il messaggio in coda, altrimenti invia *REPLY*. Il peer che riceve *REPLY* incrementa di 1 la variabile *replies*. Il codice relativo a tale algoritmo si trova nella cartella */src/peer/ricartAgrawala* nella repository Github.

5. SEZIONE CRITICA

Per implementare la sezione critica si è deciso di utilizzare una *Sleep* di 10 secondi, che corrisponde al tempo in cui il peer si trova in sezione critica.

6. TEST

All'interno del file *configurations.go* è presente un flag *TEST* che può essere impostato in caso si vogliano eseguire i test. Si è deciso di testare il funzionamento degli algoritmi implementati nel caso in cui vi sia un solo processo che richiede l'accesso alla risorsa condivisa e nel caso in cui due processi richiedano contemporaneamente l'accesso alla CS. Si può scegliere quale test effettuare tramite il menu che appare appena si avviano i vari container (vedere paragrafo 4.1). I test devono essere eseguiti necessariamente con il flag *verbose* attivato, in quanto andranno a leggere il contenuto dei vari file di log che sono stati scritti durante l'esecuzione degli algoritmi e che riportano i messaggi scambiati tra le varie entità del sistema e gli accessi alla CS con i relativi tempi di accesso. Per tutti e 3 gli algoritmi sono state verificate le due proprietà di mutua esclusione:

- **Safety:** al più un solo processo alla volta può eseguire la CS
- **No starvation:** le richieste di ingresso nella CS vengono prima o poi soddisfatte. Questa è una proprietà di liveness, ossia che non ci sono ritardi indefiniti, e una condizione di imparzialità (fairness) nei confronti dei processi.

Inoltre, è stato verificato il numero atteso di messaggi scambiati tra i vari peer per accedere alla CS, pari a :

- **3(N-1)** per Lamport distribuito
- **2(N-1)** per Ricart-Agrawala
- **3** (1 REQUEST + 2 TOKEN) per token-asking (si contano solo i messaggi scambiati con il coordinatore, senza contare i messaggi di programma)

dove N è il numero di peer che partecipano al gruppo di mutua esclusione.

6.1. Test safety

Per verificare che gli algoritmi soddisfano la proprietà di safety, su ogni file di log è riportato, per ogni peer, l'orario di accesso e uscita dalla CS. In questo modo si controllano che gli orari in cui un peer è in CS non si sovrappongano con quelli di un altro peer, garantendo così che ci sia un solo processo alla volta in CS.

```
[20:16:11.979] : p1 enters the critical section at 20:16:11.979.
[20:16:42.017] : p1 exits the critical section at 20:16:42.017.
```

Figure 6. Esempio di file di log in cui è riportato l'orario in cui un peer si trova in CS.

6.2. Test no starvation

Per controllare che ogni peer che ha effettuato un messaggio di *REQUEST* entri effettivamente in CS, senza rimanere indefinitamente nella sua trying section (sequenza di istruzioni che precede la CS), si è controllato semplicemente che il file di log relativo al peer contenga effettivamente le frasi "*enters the critical section*" e "*exits the critical section*".

6.3. Test numero di messaggi scambiati

All'interno dei file di log dei vari peer, come precedentemente affermato, vengono riportati anche i vari messaggi scambiati, come mostra ad esempio la seguente figura relativa ai messaggi scambiati tra i peer che implementano l'algoritmo di Ricart-Agrawala.

```
peer_0.log
1 Initial logical scalar clock of p0 is 0
2 [23:12:51.948] : p0 updates its local logical timeStamp to 1
3 [23:12:51.953] : p0 sends Request message: {Request p1 23:12:51.948 [1]} to p1.
4 [23:12:51.957] : p0 sends Request message: {Request p2 23:12:51.948 [1]} to p2.
5 [23:12:51.963] : p0 waits all peer reply messages.
6 [23:12:51.963] : p0 receives Reply message: {Reply 23:12:51.956 [1]} from p1
7 [23:12:51.967] : p0 receives Reply message: {Reply 23:12:51.963 [1]} from p2
8 [23:12:51.968] : p0 receives all peer reply messages successfully.
9 [23:12:51.972] : p0 enters the critical section at 23:12:51.972.
10 [23:13:21.998] : p0 exits the critical section at 23:13:21.998.
```

Figure 7. Esempio di file di log in cui sono riportati i messaggi scambiati tra i vari peer che implementano l'algoritmo di Ricart-Agrawala. In tutti e 6 i casi di test sono state soddisfatte le 3 proprietà.

References

- [1] <https://pkg.go.dev/net/rpc>
- [2] <https://pkg.go.dev/net>
- [3] <https://pkg.go.dev/encoding/gob>