Progetto B2: Algoritmi di mutua esclusione distribuita in Go

Adrian Petru Baba, matricola 0320578
Università di Roma Tor Vergata
Laurea Magistrale di Ingegneria Informatica, corso Cybersecurity
email: baba.adrian13@gmail.com

***Abstract*—Il seguente documento è stato redatto con lo scopo di descrivere la realizzazione di un'applicazione distribuita che simula diversi algoritmi di mutua esclusione. Al suo interno verranno presentate la sua architettura, le scelte progettuali effettuate insieme agli algoritmi selezionati, la piattaforma utilizzata, le eventuali limitazioni riscontrate e la fase di testing effettuata.**

I. INTRODUZIONE

Gli algoritmi implementati nel progetto sono **tre**, nello specifico:

- **CENTRALIZZATI**:
    1. Token centralizzato, in cui la sezione critica è accessibile solo ai peer che ottengono il token dal coordinatore
- **DECENTRALIZZATI**:
    2. Token decentralizzato, in cui la sezione critica è accessibile solo ai peer che ottengono il token dal loro vicino all'interno dell'anello logico
    3. Ricart & Agrawala, in cui la sezione critica è accessibile solo ai peer che ottengono l'autorizzazione da parte di tutti gli altri

Le logiche degli algoritmi, spiegate meglio in seguito, sono implementate all'interno dei peer partecipanti al processo di mutua esclusione.

Per la coordinazione fra i peer, come presentato nelle specifiche, è stato implementato un servizio di registrazione a cui ciascuno di loro può rivolgersi prima di avviare il processo.

Ciascun componente del sistema è stato embeddato all'interno di un container, ed i container relativi ad uno specifico servizio - dunque, algoritmo- sono orchestrati come si vedrà in seguito. Il progetto prevede che il processo di mutua esclusione avvenga su un numero non prefissato di peers; inoltre, per l'esecuzione, sono previsti due parametri configurabili tramite flag: uno ***verbose***, che nel caso in cui venga selezionato mette a disposizione informazioni di logging sulle attività svolte dalla componente avviata, utile per debugging, e uno ***delay***, che nel caso in cui venga selezionato aggiunge un delay artificiale ad ogni messaggio trasmesso, in modo da simulare una piccola congestione di rete, utile per verificare il funzionamento del sistema anche in situazioni più delicate, monitorabili comunque tramite log. I dettagli verranno spiegati successivamente.

II. TECNOLOGIE ADOPERATE

Per la realizzazione dell'applicazione sono state utilizzate diverse tecnologie per scopi differenti, ovvero:

- **Go:** unico linguaggio di programmazione adoperato, come richiesto nelle specifiche
- **Go-RPC:** supporto per le remote procedure calls nativo del linguaggio go: semplice ed efficiente nel contesto di questa applicazione, dato che è stata realizzata con lo stesso linguaggio di programmazione
- **Docker:** supporto per la containerizzazione, utilizzato per inserire ciascun elemento del sistema in uno specifico container
- **Docker compose:** tecnologia per l'orchestrazione dei container, utilizzata per gestire in maniera semplice l'esecuzione dell'intera logica applicativa necessaria per la simulazione del processo su container differenti, con tanto di rete specifica tra i singoli container
- **EC2:** Servizio cloud di amazon utilizzato per deployare l'applicazione

La scrittura dell'applicazione, insieme ai primi test unitari, è avvenuta in ambiente **Windows** utilizzando **Visual Studio Code** come editor per tutto il codice scritto, ovvero i file Go, i file .yml, i file .sh.

III. ALGORITMI IMPLEMENTATI

Per la realizzazione del progetto sono stati selezionati tre algoritmi differenti, tenendo conto delle loro caratteristiche per essere implementati, che nello specifico sono:

TOKEN CENTRALIZZATO: Ogni peer possiede un clock vettoriale, con un elemento- clock- per ogni peer all'interno del processo di simulazione. Il peer che deve eseguire la sezione critica, per farlo, deve prima di tutto aumentare di 1 il valore della sua componente nel clock vettoriale, alla sua posizione, e successivamente inviare una richiesta ad un coordinatore contenente questo clock vettoriale. Subito dopo deve inviare i messaggi di programma ad ogni altro peer contenenti lo stesso clock vettoriale.

Al ricevimento di un messaggio di programma, il peer deve aggiornare il proprio clock vettoriale andando a sostituire, per ogni elemento del clock in suo possesso, il valore maggiore tra quello attualmente presente e quello presente nel clock di chi ha inviato il messaggio di programma.
Come prima specificato, questo algoritmo prevede la presenza di un coordinatore esterno al processo di mutua esclusione, possessore del token, che lo invia in base ad un ordine di eleggibilità decretata in base al valore corrente del clock del coordinatore stesso. Una richiesta è eleggibile quando tutte le componenti del clock del richiedente hanno al massimo un valore pari a quello dell'elemento corrispondente nel clock del coordinatore, eccetto che per la componente che identifica il richiedente stesso.

TOKEN DECENTRALIZZATO: i peer non utilizzano nessun tipo di clock, ma si organizzano in una struttura condivisa di anello- un anello logico, che nulla ha a che vedere con la collocazione fisica dei peer stessi.
Per entrare in sezione critica un peer deve ricevere il token dal suo predecessore all'interno dell'anello. La tipologia di algoritmo però è in qualche modo definibile "asincrona", caratteristica data dal fatto che la ricezione del token è indipendente dall'effettiva necessità del peer di entrare in sezione critica. Questa cosa si traduce nel fatto che un peer potrebbe ottenere un token senza che abbia bisogno di entrare in sezione critica, e quindi inviarlo subito al peer vicino all'interno dell'anello, cosa che significa far girare a vuoto il token.

RICART & AGRAWALA: Ciascun peer possiede un clock scalare e uno stato corrente che può essere: "**CS**" quando è correntemente in sezione critica, "**NCS**" quando non è in sezione critica e non ha bisogno di entrarci, e "**Requesting**" quando non è in sezione critica ma ha bisogno di entrarci e quindi ha effettuato le richieste agli altri peer.
Per entrare in sezione critica il peer deve ottenere il permesso da parte di tutti gli altri peer.
Un peer che ha bisogno di entrare in sezione critica, dunque, la richiede a tutti gli altri inviando il suo identificatore e il suo clock. Un peer che riceve una richiesta la appende tra le ricevute a cui dovrà rispondere in futuro se si trova già in sezione critica o se ha già richiesto la sezione critica e il suo clock è più piccolo di quello del richiedente- oppure di pari valore ma con un identificatore (univoco) inferiore a quello del richiedente-. Indipendentemente dall'esito, il clock verrà aggiornato al valore più grande tra il corrente e quello del peer richiedente.
Il peer che termina l'esecuzione della sezione critica, oltre a cambiare il proprio stato, invia il permesso di entrata in sezione critica a tutti quelli che l'avevano richiesta fino a quel momento.

IV. ASSUNZIONI E PREMESSE

Nella realizzazione dell'applicazione sono state fatte differenti assunzioni, alcune ricavate dalle specifiche, altre imposte artificialmente per semplificare il lavoro- che di fatto è finalizzato all'implementazione degli algoritmi descritti in seguito, senza badare a dettagli di contorno- che sono le seguenti:

- I vari componenti, in particolar modo i peer, non possono **crashare**: non crasheranno prima della simulazione del processo e non crasheranno durante il processo. Meccanismi di hearthbeat per il monitoraggio non sono stati implementati. Per via di questa assunzione le comunicazioni tra peer sono state implementate in modo diretto, senza meccanismi di controllo: in caso di errore, scenario escluso da questa premessa, l'applicazione verrebbe terminata e dismessa
- I peer all'interno del sistema vengono identificati dall'indirizzo IP all'interno della rete, dalla porta su cui si metteranno in ascolto e da un'identificativo numerico. Queste informazioni vengono memorizzate e trasmesse a tutti dal registratore. L'identificativo, di fatto, è l'indice di quello specifico peer all'interno della struttura che contiene questi dati
- I peer potrebbero non essere avviati contemporaneamente: il primo tentativo di contattare un peer prevede l'eventuale presenza di un errore nella connessione, arginato da una finestra di **10** tentativi di connessione intervallati da ***150ms***; lo stesso ragionamento è stato fatto per quanto riguarda il primo tentativo di connessione al registratore da parte del peer, con le stesse iterazioni e tempistiche
- Le specifiche non prevedono un tempo di simulazione prefissato: per rendere le cose semplici la simulazione avviene con un numero hardcoded di iterazioni, dove ciascuna iterazione rappresenta una eventuale-come verrà spiegato in seguito- entrata in sezione critica da parte del peer
- Per lo shutdown dei vari componenti si è scelto un meccanismo di polling-descritto meglio in seguito- in modo da arginare il problema dei peer che terminano prima di altri e vengono spenti, il che ovviamente causerebbe un errore nel tentativo di contattarli
- La sezione critica è solo una simulazione artificiale che prevede esclusivamente l'utilizzo di una sleep per un tempo variabile tra ***1s*** e ***10s***.
  Similmente è stato fatto per il delay, implementato con una sleep per un tempo variabile fra ***500ms*** e ***1500ms***; similmente è stato fatto anche per il delay che simula l'esecuzione di un peer tra il completamento di una sezione critica e la successiva richiesta di entrata in sezione critica, con tempi che vanno da **1s** a **5s**- ovviamente tempi uguali a 0 sarebbero poco rappresentativi di una situazione reale.
  In ogni caso i valori sopra menzionati sono stati selezionati arbitrariamente in quanto apparentemente consoni, solo per rendere la simulazione più realistica ma senza l'eventuale studio di una situazione reale.
- I componenti che gestiscono coordinamento e registrazione ascoltano su porte fisse- hardcoded-, rispettivamente **8100** e **8200**
- Un peer che ottiene il token, nell' esecuzione dell'algoritmo del token decentralizzato, ha una probabilità del **30%** di non necessitare del token e quindi trasmetterlo subito al vicino senza usufruirne

V. ARCHITETTURA

L'applicativo è stato suddiviso in package, uno per ciascuna funzionalità prevista- e quindi componente eseguibile stand alone-, ovvero registratore, coordinatore e tre tipologie di peer, una per ciascuna tipologia di algoritmo richiesto.
Addizionalmente è stato introdotto un packege "utils" che racchiude funzionalità di utilità comuni a tutti: in questo modo si è ottenuta una certa modularità che ha permesso di semplificare il lavoro andando a lavorare su ogni servizio in modo separato.
Il main, esterno a questi sub-package, in una prima versione dell'applicativo- realizzata per il testing in locale- faceva da rilocatore per l'esecuzione della specifica funzione richiesta in base a un paramentro in input, prendendo dai paramentri anche le altre variabili di configurazione -ovvero i flag-, cosa modificata in corso d'opera per supportare la funzionalità con l'inclusione della containerizzazione, come spiegato successivamente.
Ciascun package prevede due file. Uno contiene dati e funzioni di avvio della logica della componente richiesta, di pubblicazione dei servizi della componente richiesta ed esecuzione della logica dell'algoritmo, nel caso in cui si tratti di un peer. Un secondo file invece include tutte le procedure esposte dalla componente- ed eventuali procedure locali a supporto di quelle esposte-.
Il package utils contiene due file, uno che comprende una serie di dati e procedure esposte comuni a tutti gli altri e un file contenente dati e procedure correlate alla funzionalità di logging. Una volta implementata l'applicazione e verificato il suo corretto funzionamento (il testing verrà visto in dettaglio successivamente) si è passati alla containerizzazione delle singole funzionalità. A questo punto dell'implementazione i parametri di configurazione non potevano più essere passati con facilità a docker compose- non è satto possibile identificare un modo immediato per utilizzare variabili in modo dinamico durante la messa in atto del file compose.yml: per sopperire a questo problema è stato utilizzato il supporto menzionato anche dalla documentazione, ovvero le variabili d'ambiente settate all'interno di un file .env nell'ambito della directory del progetto. Questo è, ovviamente, facilmente modificabile con uno script o staticamente per effettuare le prove.
Ciascun container è stato progettato per eseguire una delle funzionalità previste dal sistema, e per avviarlo è stato utilizzato un solo **Dockerfile**- indipendenetemente dalla funzionalità- che avvia l'applicazione con un servizio basato sul parametro da settare nell'argomento del Dockerfile "service"- i dettagli verranno visti in seguito.
Per la realizzare l'intero servizio di mutua esclusione, sono stati scirtti tre file docker-compose differenti che contengono la combinazione di servizi richiesti per l'esecuzione del processo. Per runnare il giusto numero di peer è stato utilizzato il parametro di configurazione "**--scale** <nome_servizio> = <numero_repliche>".
Successivamente, per terminare l'implementazione, è stata inizializzata una macchina amazon linux con EC2, opportunamente configurata, per far eseguire l'applicativo in questione.

VI. IMPLEMENTAZIONE

Molte delle costanti utilizzate e riprese nel resto del seguente capitolo fanno riferimento a quelle presentate nel capitolo IV.
La prima implementazione prevedeva, come è stato detto, l'intera configurazione delle funzionalità tramite parametri del main. Inizilamente sono stati previsti per l'utilizzo fino a un massimo di 3 parametri: il primo un intero rappresentante il servizio richiesto- 0 per il registratore, 1 per il coordinatore, 2 per il il token centralizzato, 3 per il peer da token decentralizzato e 5 per il peer da ricart & agrawala-, parametro mantenuto anche nell'implementazione finale, mentre gli altri due contenenti gli eventuali flag -v, verbose, se si intende avere un servizio di logging delle attività della simulazione, e -d, se si intende avere la simulazione di un sistema congestionato. Successivamente, in base al servizio richiesto, si va ad eseguire un ramo di computazione specifico.
Il registratore non fa altro che mettersi in ascolto su una porta predefinita, specificata in precedenza, e attende richieste di adesione da parte dei peer. Per ciascuna richiesta di adesione, inserisce l'indirizzo completo- ip e porta- del peer in una slice, ovviamente se non è già presente, ed invia al richiedente il suo identificativo che come è stato specificato in precedenza rappresenta il suo indice all'interno di questa slice. Quando riceve abbastanza richieste di adesione- come da specifiche, il numero dei peer partecipanti è costante durante il processo di mutua esclusione- il registratore invia questa struttura dati contenente tutti i dati necessari per la comunicazione ai peer, per poi effettuare lo shutdown.
Il coordinatore si mette in ascolto sulla porta specificata prima, acquisisce il token e si mette in attesa di richieste da parte dei peer che utilizzano l'algoritmo di token centralizzato. Ovviamente l'implementazione tiene conto delle caratteristiche teoriche previste per il coordinatore nel capitolo III, e dunque accoda le richieste, verifica quali sono eliggibili inviando i token di conseguenza, e aggiorna man mano il suo timestamp.
Il peer del token centralizzato inizialmente non possiede il token. Per prima cosa recupera il suo indirizzo ip all'interno della rete e genera un numero di porta casuale tra **10000** e **11000**, sulla quale si mette in ascolto, esportando così i suoi servizi, e poi invia la richiesta di adesione al registratore. Una volta che il registratore gli invia la lista di tutti i peer, inizia il processo per il quale per un numero prefissato di iterazioni il peer simula un certo tempo di esecuzione prima di aggiornare il proprio clock ed inviare la richiesta per il token al coordinatore, e poi agisce come visto nel capitolo III, inviando i messaggi di programma, attendendo il suo token, eseguendo la sezione critica e riconsegnando il token.
Il peer per il token decentralizzato genera la propria porta su cui si mette in ascolto ed invia la richiesta di adesione. Il peer con identificativo 0 è stato scelto come inizializzatore del processo. Una volta iniziato, si svolge come specificato nel capitolo III: un peer simula l'esecuzione fino all'ottenimento di un token da parte del suo peer vicino; in quel momento se ha bisogno di entrare in sezione critica entra- in base alla regola descritta in precedenza-, altrimenti reinvia il token al prossimo peer. L'anello logico formato per l'esecuzione di questo processo di mutua esclusione è uno semplice banalmente basato

sull'identificativo dei singoli peer, in ordine crescente, dove ovviamente il peer con identificativo maggiore avrà come vicino quello con identificativo minore, ovvero 0.
Il peer per ricart&agrawala genera la propria porta, si mette in ascolto e invia la richiesta di adesione, attendendo la lista. Successivamente inizia il processo di mutua esclusione rispettando ciò che è stato detto nel capitolo III, inviando richieste quando si vuole entrare in sezione critica e attendendo le autorizzazioni necessarie prima di entrare in sezione critica.
Come menzionato in precedenza, un problema possibile è dato dal fatto che, al termine dell'esecuzione di un peer, questo subiva subito lo shutdown, rendendolo impossibile da contattare in caso di necessità, per esempio da parte di un peer che sta cercando di inviare messaggi di programma nell'esecuzione dell'algoritmo del token centralizzato.
Per ovviare a questo problema è stato applciato un meccanismo di polling: attraverso una goroutine la componente- peer o coordinatore- si mette in moto ogni tot secondi controllando una certa variabile di stato booleana che vale true non appena c'è un'interazione con la rispettiva componente- dunque una rpc call- e vale false all'inizio e non appena questa goroutine termina l'esecuzione del controllo: se la goroutine trova la variabile a false significa che per un arco di tempo significativo nessuno ha interagito con la componente e quindi il processo di mutua esclusione viene considerato terminato e la goroutine termina la sua esecuzione. Ovviamente, dopo che un peer o il coordinatore ha eseguito le sue operazioni di simulazione, si mette in attesa della terminazione della goroutine sopracitata, e quindi lo shutdown avviene solo alla sua terminazione. Il meccanismo di polling si mette in moto ogni **15s** per un tutti i componenti tranne per i peer dell'algoritmo token decentralizzato: dato che l'unica comunicazione che l'algoritmo prevede è la ricezione di un token e questa avviene una volta che il token ha percorso tutto l'anello, è stato ritenuto più opportuno assegnare un valore temporale dipendente dal numero del peer nel processo di mutua esclusione, in particolare di **(15 * numero peer)s**.
Per la simulazione della congestione della rete, come specificato precedentemente, prima di inviare uno qualsiasi dei messaggi, e quindi per ciascuna rpc, è stato introdotto un delay artificiale fino a un secondo e mezzo di attesa. Per il logging è stato scelto un meccanismo predefinito dal linguaggio che permette la creazione di un logger su un file dato in input con estensione .log su cui si può scrivere facilmente una stringa a piacere, e su cui sarà il logger ad aggiungere le informazioni temporali-data e ora-. La struttura adoperata prevede che i file di logging vadano a finire tutti in una directory "logs" all'interno della directory del progetto. Nello specifico, poi, al suo interno i file di log finiranno all'interno di una sottodirectory denominata in base al servizio richiedente. I singoli file hanno nomi univoci identificati dai timestamp, della forma **log+timestamp.log** .
La configurazione di questo servizio, quindi l'eventuale creazione delle directory sopra citate e la creazione del file corrente, correlato al servizio richiedente, avviene come fase iniziale nell'esecuzione del servizio, in base al valore del parametro -v.
Nella fase successiva è satto realizzato il dockerfile e i docker-compose.
Come specificato in precedenza, per far funzionare correttamente il tutto, i parametri di configurazione sono stati inseriti all'interno di un file .env. i parametri in questione sono "**NPEERS**", ovvero il numero di peer che si vogliono per la simulazione corrente, "**VERBOSE**", che se vale true indica che si vuole il logging nella simulazione corrente, e "**DELAY**" che, se vale true, sta ad indicare la volontà di simulare la congestione di rete nella simulazione. Per far funzionare questo approccio il main dell'applicativo è stato modificato affinchè prenda i parametri di configurazione dalle variabili d'ambiente del sistema e non dai parametri in input all'applicativo. Per automatizzare il sistema è stato scritto uno script shell .sh che permettesse in modo semplice di raccogliere i parametri, scriverli nel file di configurazione e avviare la simulazione. In particolare questo script prevede 5 flag: 'h' per visualizzare le indicazioni di utilizzo per questo script; 'v' e 'd' per indicare rispettivamente che si vuole il logging ed il delay; 'n' seguito da un numero per specificare il numero di peers voluti e 'a' seguito da 'c','d' oppure 'r' per specificare di voler utilizzare rispettivamente algoritmo token centralizzato, token decentralizzato e ricart&agrawala. i valori previsti di default, nel caso non venissero specificati con l'esecuzione di questo script, sono 4 peer, log e delay non previsti e token centralizzato come algoritmo da utilizzare. Lo script, una volta acquisiti i parametri in input, elimina- se presente- il vecchio file .env ricreandolo con la nuova configurazione, droppa- se presenti- i vecchi cluster di container creati con compose per ciascuno dei 3 file docker-compose e poi avvia- selezionando opportunamente il file .yml da utilizzare e il numero di peer- il cluster con il comando compose e il numero previsto di peer.

VII. TESTING

La fase di testing è stata effettuata andando ad analizzare, come da specifiche, due situazioni particolari: un caso in cui ci sono diversi peer concorrenti che tentano l'accesso in sezione critica, anche contemporaneamente, ed un caso in cui all'interno del processo di mutua esclusione c'è un solo peer, questo per ciascuno dei tre algoritmi.
Per la simulazione sono state tenute un numero di iterazioni pari a cinque ed il delay attivato. Il numero di peer scelto è di quattro. Per testare il corretto funzionamento degli algoritmi ovviamente nel caso di diversi peer si è controllato che i tempi di accesso in sezione critica, indipendentemente dal momento della richiesta, non siano sovrapposti. Nel caso del peer singolo si è visto che l'esecuzione non venisse interrotta per un qualsiasi motivo, e che tendenzialmente l'accesso in sezione critica avvenisse in un intervallo ragionevole dalla richiesta. Ovviamente in nessun caso non deve verificarsi la stavation.
I peer sono stati nominati in base all'identificativo. I risultati sono sintetizzati nelle tabelle seguenti:

Iterazione	Peer #0	Peer #1	Peer #2	Peer #3
1 request	17:07:02	17:07:01	17:07:03	17:07:02
1 start CS	17:07:06	17:07:02	17:07:28	17:07:17
1 end CS	17:07:14	17:07:05	17:07:34	17:07:26
2 request	17:07:21	17:07:09	17:07:39	17:07:29
2 start CS	17:07:39	17:07:36	17:07:54	17:07:47
2 end CS	17:07:45	17:07:38	17:08:03	17:07:52
3 request	17:07:51	17:07:40	17:08:09	17:07:55
3 start CS	17:08:12	17:08:06	17:08:27	17:08:21
3 end CS	17:08:20	17:08:10	17:08:34	17:08:24

Tabella 1: token centralizzato, 4 peer, tentativi di accessi concorrenti

Fase	Peer #0	Peer #1	Peer #2	Peer #3
1 acquisizione tk	18:37:44	18:37:51	18:38:01	18:38:02
1 rilascio tk	18:37:50	18:38:00	18:38:01	18:38:05
2 acquisizione tk	18:38:05	18:38:15	18:38:24	18:38:28
2 rilascio tk	18:38:14	18:38:22	18:38:27	18:38:35
3 acquisizione tk	18:38:36	18:38:41	18:38:47	18:38:53
3 rilascio tk	18:38:39	18:38:46	18:38:52	18:38:53

Tabella 2: token decentralizzato, 4 peer

Stato	Peer #0	Peer #1	Peer #2	Peer #3
1 REQUESTING	19:51:16	19:51:19	19:51:21	19:51:20
1 CS	19:51:22	19:51:37	19:51:44	19:51:28
1 NCS	19:51:26	19:51:43	19:51:45	19:51:36
2 REQUESTING	19:51:33	19:51:47	19:51:48	19:51:40
2 CS	19:51:46	19:51:57	19:52:02	19:51:52
2 NCS	19:51:51	19:52:00	19:52:09	19:51:55
3 REQUESTING	19:51:57	19:52:05	19:52:14	19:52:00
3 CS	19:52:09	19:52:24	19:52:28	19:52:18
3 NCS	19:52:17	19:52:27	19:52:29	19:52:23

Tabella 3: ricart&agrawala, 4 peer, tentativi di accessi concorrenti

Iterazione	Peer #0
1 request	20:02:36
1 start CS	20:02:38
1 end CS	20:02:43
2 request	20:02:46
2 start CS	20:02:47
2 end CS	20:02:48
3 request	20:02:53
3 start CS	20:02:54
3 end CS	20:02:57

Tabella 4: token centralizzato con un solo peer

Fase	Peer #0
1 acquisizione tk	20:14:33
1 rilascio tk	20:14:33
2 acquisizione tk	20:14:34
2 rilascio tk	20:14:37
3 acquisizione tk	20:14:38
3 rilascio tk	20:14:44

Tabella 5: token decentralizzato con un solo peer

Stato	Peer #0
1 REQUESTING	20:24:21
1 CS	20:24:21
1 NCS	20:24:30
2 REQUESTING	20:24:34
2 CS	20:24:34
2 NCS	20:24:41
3 REQUESTING	20:24:44
3 CS	20:24:44
3 NCS	20:24:47

Tabella 6: ricart&agrawala con un solo peer

Nota: nel caso dell'algoritmo token decentralizzato, tempi di acquisizione e rilascio token equivalenti nella stessa iterazione equivalgono al caso in cui il peer non necessita di entrare in sezione critica, come evidenziato anche all'interno del file di log.

La fase di testing ha permesso di evidenziare due bug abbastanza importanti. Uno di loro riguardava l'algoritmo di ricart&agrawala: il confronto per definire se una richiesta in arrivo avesse timestamp maggiore rispetto alla richiesta effettuata dal peer ricevente avveniva sul clock corrente- valore che può essere aggiornato in vari momenti da parte delle altre richieste- e non con il valore del clock dell'ultima richiesta- valore fisso ad ogni richiesta, appunto-, errore che può causare una situazione in cui i peer inviano reply message anche quando non dovrebbero con conseguente entrata in sezione critica contemporanea fra più peer.
Il secondo bug, più subdolo, si manifestava con molta più probabilità nella simulazione con delay- e quindi in una situazione più realistica- e riguardava l'algoritmo del token centralizzato. Inizialmente il token del coordinatore veniva tolto solo una volta inviato al peer corretto. La sequanza di operazioni però è critica: dato che l'update del timestamp avviene come prima operazione al raggiungimento di una richiesta, due richieste conseguenti e contemporanee ed entrambe eleggibili quando il clock viene aggiornato con il giusto tempismo potrebbero ottenere entrambe il token, dato che l'operazione di invio del token richiede del tempo prima di essere completata- e fino a quel momento il coordinatore continua a mantenere il token.
La soluzione ovvia è quella di rimuovere il token come prima operazione, e solo dopo inviarlo al peer.
Le varie tabelle ci mostrano come indipendentemente dall'ordine delle richieste, i peer entrano in servizio in ordini diversi e con tempi diversi, cosa naturale data la presenza di delay, tranne nel caso del token decentralizzato dove l'ordine dell'anello logico viene, e deve essere, rispettato indipendentemente dalla congestione di rete.
In ogni caso, il comportamento del sistema è quello atteso in tutti e sei i casi di test.