Algoritmo di Chandy-Lamport per lo snapshot in un sistema distribuito

Leonardo Pompili 0353499

*Abstract*—In questo paper presento la mia implementazione del noto algoritmo di Chandy-Lamport applicato alla mia applicazione, volto a fare uno snapshot globale del sistema. Questo torna utile in contesti distribuiti in cui dei peer potrebbero abbandonare il sistema e rientrarvi in un momento successivo, oppure potrebbero subire un crush. Si ha quindi l’esigenza di salvare lo stato del sistema in modo da mantenere un certo grado di consistenza.

Keywords—peer, Chandy-Lamport, snapshot, service registry

# Introduzione

Per il corso di Sistemi Distribuiti e Cloud Computing, tenuto dalla professoressa V. Cardellini, sono andato a realizzare una piccola applicazione su cui far girare l’algoritmo di *Chandy-Lamport* [1] per lo snapshot globale di un sistema distribuito. Come da specifica, il linguaggio di programmazione utilizzato è stato *GoLang* [2], mentre per la comunicazione tra i vari peer ho usato procedure a chiamata remota, nello specifico ho adoperato la tecnologia *gRPC* [3] per le chiamate RPC. Un vantaggio nell’uso di questo strumento sta nell’utilizzo del file proto, comodo per la leggibilità e lo sviluppo di chiamate a procedura remota. Per il salvataggio dello stato dei canali e dei peer stessi, ho scelto di mantenere le informazioni in un semplice file. Più nello specifico, per lo stato dei canali, ho utilizzato un *Interceptor* [4], che si pone l’obiettivo di catturare le chiamate RPC.

# Algoritmo di Chandy-Lamport

Il primo passo è stato quello di comprendere a fondo il funzionamento dell’algoritmo, motivo per cui sono andato a leggere il paper ufficiale di Chandy e Lamport pubblicato nel 1985, nel quale però ho fatto fatica a comprendere anche i passi principali, probabilmente a causa dell’eccessivo tecnicismo utilizzato. Quindi, ho fatto delle ulteriori ricerche online, e alla fine mi sono imbattuto nel canale delle professoressa e ricercatrice *Lindsey Kuper* [5]. Tramite le sue *videolezioni* [6], caricate ormai molti anni fa, ho compreso perfettamente gli step del noto algoritmo, anche grazie ai numerosi esempi riportati.

## Assunzioni

Innanzitutto, l’algoritmo funziona sotto determinati assunzioni:

* Ogni processo registra solamente il proprio stato e lo stato dei canali in entrata;
* Comunicazione affidabile (i messaggi non vanno persi ne danneggiati);
* Canali FIFO e unidirezionali;
* L’esecuzione dell’algoritmo è sottostante a quella dell’applicazione su cui gira, ovvero non influisce sul normale flusso di esecuzione di ciascun peer;
* Ogni nodo può avviare la registrazione, inoltre i processi che la cominciano possono essere più di uno.

## Pseudocodice

Per prima cosa, assumiamo che un solo processo inizi la registrazione dello stato, ovvero che dia inizio all’esecuzione dell’algoritmo. Allora questo effettua i seguenti passaggi:

1. Registra il proprio stato.
2. Invia un messaggio di “marker” su tutti i suoi canali in uscita.
3. Inizia a registrare i messaggi su tutti i suoi canali in entrata.

Sia ora il generico processo che riceve un messaggio di “marker”, e denotiamo con il canale tra il processo -esimo e il processo -esimo. Quando il processo riceve un messaggio di “marker” sul canale :

* Se è la prima che vede un messaggio di “marker”:

1. registra il suo stato
2. marca il canale come vuoto
3. manda un messaggio di “marker” su tutti i suoi canali in uscita
4. inizia a registrare i messaggi su tutti i suoi canali in entrata, eccetto che sul canale

* Se ha già visto un messaggio di “marker” (perché lo ha già inviato lui stesso o perché lo aveva precedentemente ricevuto):

1. smette di registra sul canale

## Esempio

Visualizziamo ora il funzionamento dell’algoritmo tramite un semplice esempio in cui compaiono 3 peer e si hanno un certo numero di eventi.

Immagine che contiene linea, diagramma, Parallelo, Diagramma

Descrizione generata automaticamente

Fig. 1. Esempio di scambi di messaggi in un applicazione distribuita con tre processi. Tratto dalla videolezione della professoressa Lindsey Kuper.

In questo caso il processo che avvia l’algoritmo è P1, e dopo l’evento B questo inizia a registrare il proprio stato, invia un messaggio di “marker” (segnato in rosso) su tutti i canali in uscita e inizia a registrare su tutti i canali in entrata, ovvero e . Quando P3 riceve il messaggio di “marker” da P1, inizia a registrare il proprio stato essendo il primo che vede, poi segna il canale come vuoto e manda un messaggio di marker su tutti i suoi canali in uscita:

Immagine che contiene linea, diagramma, Parallelo, Diagramma

Descrizione generata automaticamente

Fig. 2. Step 2 dell’algoritmo.

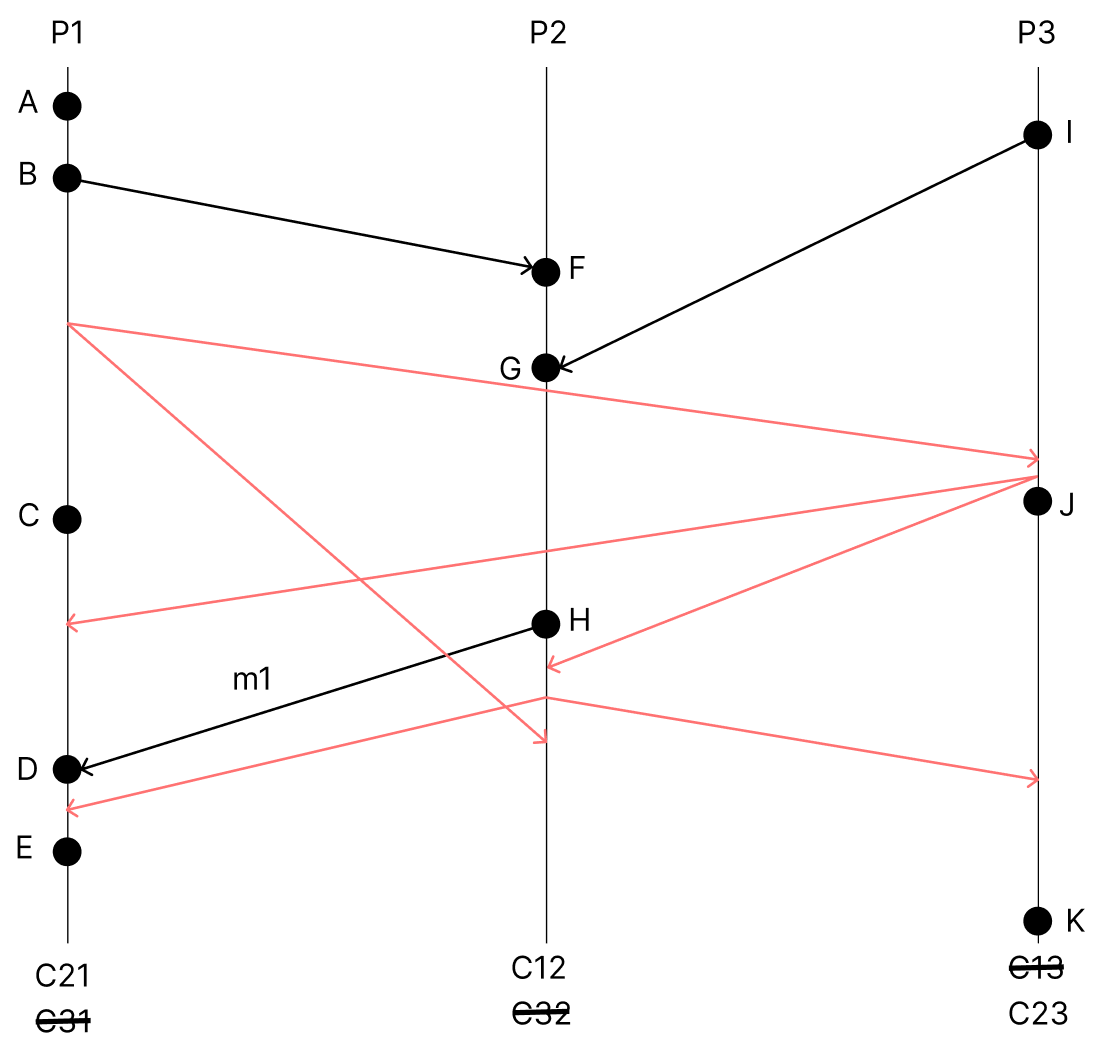
A questo punto, quando P1 riceve il messaggio di “marker” da P3 sul canale , avendo già visto il “marker” (perché lo aveva inviato lui stesso), smette di registrare su questo canale. Invece, quando P2 riceve il messaggio di “marker” sempre da P3, essendo la prima volta che lo vede, inizia a registrare il proprio stato, marca come vuoto e manda il “marker” su tutti i canali in uscita:  


Fig. 3. Step 3 dell’algoritmo.

Infine, quando P3 riceve il messaggio di “marker” da P2, questo smette di registrare su , mentre P1 riceve prima il messaggio m1, il quale andrà registrato visto che P1 stava ancora registrando i messaggi in entrata su . Solo quando riceverà il messaggio di “marker” successivamente da P2 smetterà di registrare su :

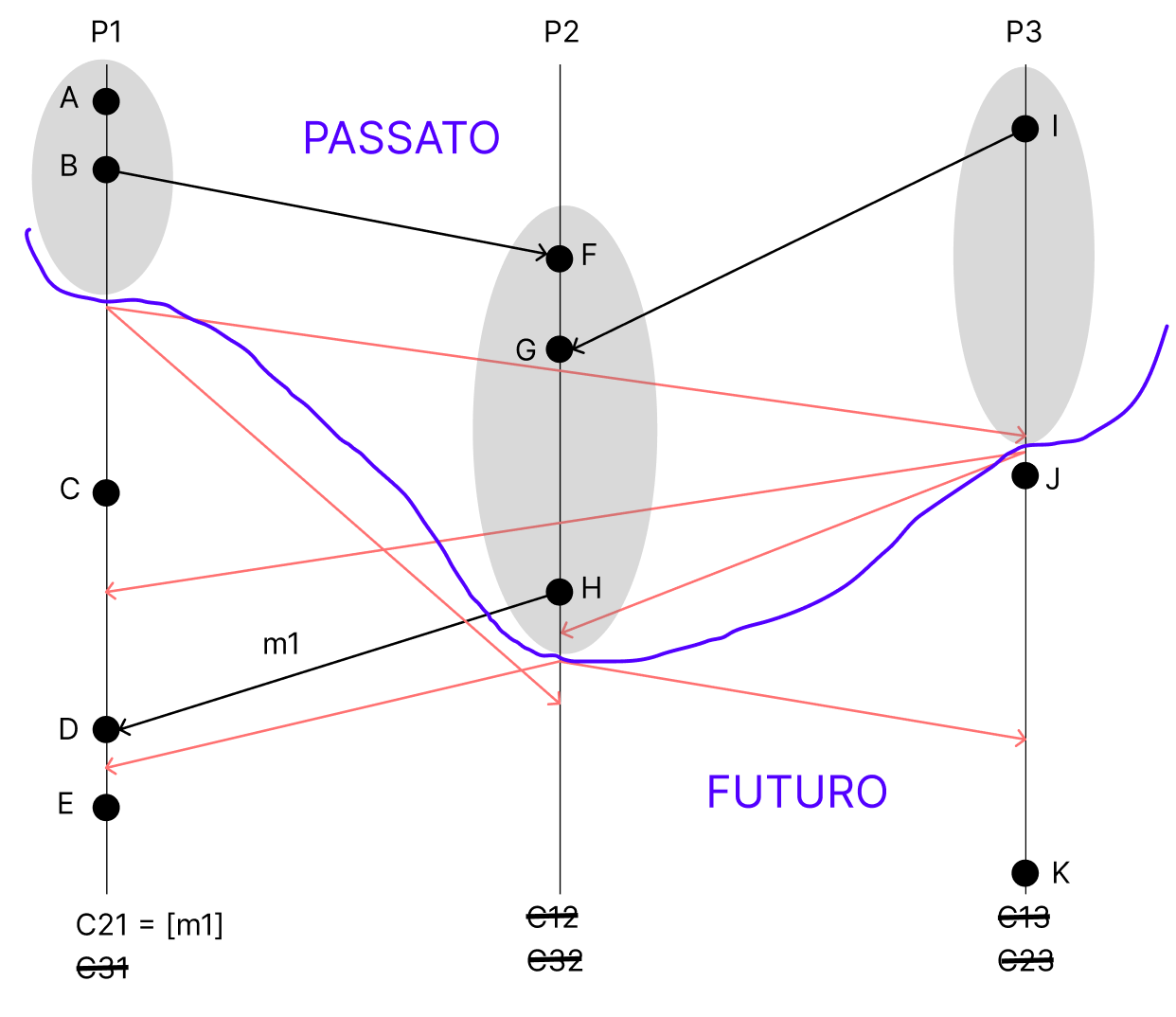


Fig. 4. Stato finale dell’algoritmo. È quindi possibile tracciare una linea tra “passato” e “futuro”, ovvero tra prima e dopo che ogni peer inizi a registrare il proprio stato.

# Gestione dell’algoritmo

Per gestire l’algoritmo, ho creato una serie di funzioni utili alla registrazione, che sono all’interno del package Chandy\_lamport, in chandy\_lamport.go. Quando si inizializza un nuovo nodo tramite la funzione StartNodeServer, viene creata un’istanza di ChandyLamportServer: questa struttura mantiene tutte le informazioni utili all’algoritmo e funzionali alla registrazione dello stato di un peer. Analizziamo ora tutti i campi più importanti che sono coinvolti nell’algoritmo.

* Version: indica la versione dello stato; questa viene incrementata ogni volta che un peer registra un nuovo stato, in modo da conoscere l’ultimo stato salvato e poter ripartire da quello;
* LocalState: rappresenta letteralmente lo stato locale del peer ed è quindi ciò che verrà salvato;
* SeenMarkers: questo campo tiene traccia se un peer ha già visto un messaggio di “marker” in generale (inviato o ricevuto);
* Recording: indica se attualmente un peer sta registrando il suo stato e quello dei canali in entrata;
* SeenMarkerForTheFirstTime: apparentemente uguale a SeenMarker, questo campo è stato introdotto per comodità e viene utilizzato per controllare se un peer deve iniziare a registrare. La differenza è che questo indica che il peer ha appena ricevuto un messaggio di “marker”, e che quindi dovrebbe iniziare a registrare, mentre SeenMarkers ci dice solamente che il nodo ha già visto messaggi di “marker” (quindi potenzialmente anche più di una volta) e viene usato per il controllo nella funzione ReceiveMarker;
* SeenMarkerFrom: questo campo tiene traccia, per ogni peer, da quale peer si è ricevuto un messaggio di “marker”, ed è necessario in quanto, tranne per il processo che inizia l’algoritmo, i peer dovrebbero registrare dai canali in entrata escluso il canale dal quale proveniva il messaggio di “marker”;
* ChannelState: allo stesso modo di LocalState, qui viene immagazzinato lo stato che dovrà essere salvato, ma per i canali. Ovvero, qui verranno scritti i messaggi (chiamate RPC) intercettate dall’Interceptor.

Tranne che nel caso di LocalState, per tutti questi altri campi ho pensato di usare la struttura dati map, che associa chiavi a valori. In questo modo, risulta facile associare ad un peer un certo valore, come ad esempio SeenMarkers[peer1] = true.

In chandy\_lamport.go sono state poi implementate le funzioni di salvataggio dello stato del peer e del canale, denominate rispettivamente SaveStateToFile e RecordChannelState. Per quanto riguarda l’esecuzione locale, ho deciso di salvare lo stato dei peer e dei canali in un semplice file di testo, denominato distributed\_snapshot.txt. Inoltre, è presente anche la funzione StopRecordingOnChannel, che serve ad indicare ad un peer che non deve più registrare e viene chiamata quando quest’ultimo riceve il secondo messaggio di “marker”.

Infine, vi è la funzione UnaryServerInterceptor, che ha lo scopo di intercettare le chiamate RPC di un nodo prima che vengano eseguite. Successivamente controlla se quel peer sta registrando tramite il campo Recording e, in caso affermativo, salva lo stato di quel canale tramite la funzione RecordChannelState. Una volta fatto questo, si ritorna al gestore originale della chiamata RPC, per proseguire col normale flusso di esecuzione. All’interno di questa funzione viene chiamata la funzione di supporto extractOperationName, che banalmente identifica il tipo di chiamata RPC e torna la stringa corrispondente.

# Applicazione distribuita

L’applicazione pipeline che ho pensato di sviluppare per eseguire l’algoritmo, simula una sorta di controllo distribuito per delle transazioni bancarie in contesti untrusted. Nello specifico, ho previsto tre componenti o ruoli: due peer interpretano ovviamente il ruolo di sender e receiver della transazione, mentre gli altri peer nella rete (uno o più) faranno da intermediari e verificheranno la transazione. Quest’ultima, infatti, invece di andare direttamente al receiver, passa per tutti gli altri nodi della rete, noti grazie al service registry, che fornisce la lista aggiornata dei peer attualmente presenti nella rete. Questi peer intermediari hanno lo scopo di verificare se conoscono il destinatario; in caso positivo andranno a incrementare un valore noto come confidence score. Prima che l’ultimo peer che precede il destinatario lasci “passare” la transazione verso quest’ultimo, viene tornato il valore del confidence score aggiornato al mittente, il quale, tramite una politica opportunamente scelta, confermerà se procedere col il trasferimento del denaro.

Immagine che contiene testo, computer, diagramma, schermata

Descrizione generata automaticamente

Fig. 5. Esempio con icone dei componenti dell’applicazione distribuita.

Le principali funzioni che implementano la logica dell’applicazione sono le seguenti:

* TransferMoney: è la funzione che viene sempre eseguita dal processo che inizia l’algoritmo. Si occupa di inviare la richiesta ti trasferimento tra i peer successivi per il controllo. Se tutto va a buon fine, alla fine verrà addebitato dal bilancio del peer un ammontare pari a quello trasferito, ammesso che il bilancio fosse maggiore all’importo da trasferire;
* VerifyTransaction: eseguita dal peer successivo al sender, si occupa di controllare la transazione verificando se conosce il receiver. In caso positivo andrà ad incrementare il confidence score. Successivamente, questa verifica se ci sono altri peer che non sono il destinatario: se esistono, allora inoltra la richiesta di controllo anche a loro chiamando la AnotherVerification, altrimenti torna il confidence score al sender chiedendo l’approvazione da parte sua per confermare la transazione;
* RequestApproval: questa ha lo scopo di approvare la richiesta di trasferimento successivamente ai controlli da parte degli altri peer nella rete. La politica banalmente scelta per l’approvazione, è che il confidence score sia pari al numero di nodi nella rete (esclusi sender e receiver), ma può facilmente essere regolata;
* UpdateBalance: aggiorna il bilancio del receiver a seguito della conferma della transazione.

Per gestire la logica delle richieste di trasferimento di denaro, ho implementato un client in client.go, situato all’interno della directory client. Questo, per quanto riguarda l’esecuzione locale, accetta (oltre al flag -localhost) solo tre argomenti, ovvero il sender, il receiver e la quantità di saldo da trasferire. In questo modo è possibile configurare di volta in volta l’operazione di trasferimento che si desidera effettuare. Inoltre, come verifica, sono andato anche a creare una funzione chiamata printAllNodesBalances che recupera il bilancio attuale di tutti i peer e lo stampa, così da poterla chiamare prima e dopo l’esecuzione del client e osservare il bilancio aggiornato.

## Interazione con l’algoritmo

Sottostante al flusso di esecuzione dell’applicazione, le funzioni principali eseguite dai vari peer tengono conto di tutti gli steps dell’algoritmo di Chandy-Lamport. Questi controlli sono stati inseriti solo una volta che ho terminato lo sviluppo dell’applicazione stessa, costringendomi talvolta ad apportare delle modifiche. Ogni volta che su un peer viene eseguita una chiamata RPC, questo verifica se ha già visto il messaggio di “marker” accedendo al campo SeenMarkerForTheFirstTime, e in caso positivo aggiornerà il proprio stato con l’operazione attuale, incrementerà il numero della versione del proprio stato e scriverà tutto all’interno del file distribuited\_snapshot.txt. Successivamente, dopo aver inviato eventuali messaggi dell’applicazione, manda un messaggio di “marker” su tutti i suoi canali in uscita, tramite la funzione SendMarkerToOutgoingChannels, la quale invocherà poi la funzione di ReceiveMarker su tutti i peer a cui il messaggio di “marker” era destinato.

La funzione ReceiveMarker altro non fa che verificare se un peer ha già visto il messaggio di “marker” e, in tal caso, chiamerà la StopRecordingOnChannel in modo che il peer non registri più né il suo stato, né i messaggi sui suoi canali in entrata. Nel caso invece fosse la prima volta che quel peer vede il messaggio di “marker”, la funzione si limiterà a segnarlo come “visto” impostando al valore true SeenMarkers, tenendo anche traccia del nodo da cui proviene in SeenMarkerFrom. Infine imposterà anche SeenMarkerForTheFirstTime al valore true. In questo modo, appena quel nodo eseguirà una certa funzione, saprà che ha ricevuto un messaggio di “marker” per la prima volta e che quindi dovrà registrare.

## Limitazioni e miglioramenti

Per quanto riguarda l’algoritmo implementato, non è stata inserita una funzionalità che permetta ad un peer che si disconnette improvvisamente, di rientrare nella rete recuperando il proprio stato. Inoltre, per semplicità, tutto lo stato è stato banalmente salvato in locale in un file di testo, ma si potrebbe tenere in considerazione l’idea di farne varie copie e mantenerle in maniera distribuita sui vari peer. Infine, per lo sviluppo dell’applicazione pipeline, ho previsto che un peer non può comunicare direttamente con tutti gli altri presenti nella reta, ma solamente con quei peers che ho impropriamente chiamato “successore” e “predecessore”.

Dal punto di vista dell’applicazione, invece, mi sono limitato alle funzionalità di trasferimento di denaro e verifica delle transazioni, ma si potrebbe pensare di ampiarla in modo che ci siano più compiti da svolgere. Un altro aspetto riguarda la politica scelta per l’approvazione del confidence score: io ho tenuto conto solo dell’indirizzo del receiver come parametro di controllo, ma si potrebbe pensare di aggiungere ulteriori parametri. Inoltre, in un sistema distribuito più ampio, si potrebbe voler decidere che è sufficiente l’approvazione di un numero limitato di altri peer.

# Deployment dell’applicazione

Come da specifica, successivamente all’implementazione dell’algoritmo di Chandy-Lamport, ho eseguito il deployment della mia applicazione, prima su Docker e poi attraverso il servizio Amazon Elastic Compute Cloud (EC2) offerto da Amazon Web Service (AWS).

## Docker

Per quanto riguarda la virtualizzazione in container tramite Docker, ho creato un Docker file per i peers (DockerFileNode), uno per il service registry (DockerFileSR) e uno per il client (DockerFileClient), nei quali ho specificato tutte le istruzioni per la costruzione dell’immagine Docker. Poi ho creato un file compose.yaml nel quale ho specificato i servizi e la configurazione da adottare per orchestrare i vari container. Qui ho anche aggiunto delle variabili d’ambiente da utilizzare per la configurazione dei peers e del client in ambiente Docker, quali nome e bilancio per ogni peer (ne ho previsti tre già configurati ma è possible aumentare il numero di repliche per ognuno), e sender, receiver e quantità di denaro da trasferire per il client. Nello eseguire correttamente la mia applicazione su Docker, ho riscontrato non pochi problemi, principalemente legati al client. Il motivo era che il container di quest’ultimo veniva eseguito prima che il service registry avesse registrato tutti i peer, nonostante lo abbia specificato tramite l’opzione depends\_on nel file compose.yaml. Non riuscendo a risolvere questo problema, ho adottato un’altra strategia, ovvero ho creato una funzione in client.go denominata getAllNodesWithRetry, nella quale il client richiede a intervalli regolari la lista dei peer al service registry, fino a quando non ce ne sono a sufficienza da poter eseguire la transazione (in questo caso almeno tre: un sender, un receiver e un altro che effettua la verifica). Un altro aspetto, legato invece all’algoritmo di snapshot globale, riguarda il salvataggio dello stato dei nodi e dei canali. Qui, infatti, ogni peer salva il suo stato in un file chiamato distributed\_snapshot.txt, che però resta solamente all’interno di quel container. Una volta interrotto il container, è possibile comunque recuperare il file, con la differenza che ogni peer avrà il suo file distributed\_snapshot.txt nella memoria del proprio container che conterrà soltanto le cose da lui registrate. Di conseguenza, per comporre lo stato globale è necessario recuperare tutti questi file.

## EC2

Come ultima cosa, ho creato un’istanza EC2 passando da Learner Lab, per il quale ci è stato messo a disposizione un credito di 100$ da spendere su AWS. Per la configurazione dell’istanza, ho lasciato tutte le impostazioni di default suggerite, fatta eccezione per la chive: anzichè lasciare quella di default, ne ho generata una personalizzata salvandomi il file .pem associato, che ho riutilizzato successivamente per connettermi all’istanza tramite SSH da terminale. Una volta connesso, ho istallato Github e Docker, clonato la mia repository Github contenente il codice, ed eseguito l’applicazione tramite il comando sudo docker-compose -f compose.yaml up. In quest’ultima fase, ho riscontrato principalmente due problemi. Il primo riguardava la mancata connessione all’istanza EC2 causata da alcuni permessi relativi al file .pem, che ho risolto accedendo direttamente alle proprietà del file stesso. Il secondo, consisteva nel fatto che i container venivano eseguiti ma l’applicazione no, e l’ho risolto semplicemente avviando un’altra istanza EC2 più potente: sono passato dalla t2.micro ad una t3.medium.

##### References

1. “Distributed Snapshots: Determining Global States of Distributed System”, <https://lamport.azurewebsites.net/pubs/chandy.pdf>.
2. <https://go.dev/doc/>.
3. <https://grpc.io/>.
4. <https://grpc.io/docs/guides/interceptors/>.
5. <https://users.soe.ucsc.edu/~lkuper/>.
6. <https://www.youtube.com/watch?v=WK3FuD7f9g8&t=3946s>.