Algoritmo di Chandy-Lamport: Progettazione e Realizzazione in Ambiente Distribuito

Federica Cantelmi
federicacantelmii@gmail.com
Università degli Studi di Roma "Tor Vergata" Roma, Italia

Abstract

Nel seguente documento viene descritta l'implementazione dell'algoritmo di Chandy-Lamport. Il progetto è stato implementato usando il linguaggio di programmazione Go e testato su un sistema distribuito di processi usando Docker Compose su un'istanza di EC2.

1 Introduzione

Nei sistemi moderni, la componente distribuita gioca un ruolo fondamentale per garantire migliori prestazioni, disponibilità e affidabilità dei dati, portando però con sé criticità relative a diversi aspetti, tra cui la determinazione di uno stato globale coerente. Questo avviene poiché in sistemi distribuiti spesso i nodi eseguono in maniera asincrona, perciò non vi è alcun clock globale con cui etichettare gli eventi, e non vi è alcuna memoria condivisa tra i nodi. Registrare uno stato globale consistente del sistema diventa necessario per debugging e gestione dei guasti e l'algoritmo di Chandy-Lamport fornisce un metodo per catturare questo stato.

Nel report viene presentata un'implementazione dell'algoritmo di Chandy-Lamport usando il linguaggio di programmazione Go, al fine di sfruttare la gestione concorrente supportata dalle *goroutine* e dai canali Go per la comunicazione asincrona tra I nodi e le parti interne dei nodi. Ciascun nodo è stato eseguito come container Docker e successivamente il sistema è stato distribuito su un'istanza AWS EC2 al fine di garantire la scalabilità e l'affidabilità offerta da un'infrastruttura *cloud* come quella di AWS.

La struttura del documento è la seguente: verrà fornita una spiegazione teorica dell'algoritmo di Chandy-Lamport, per poi passare ad una descrizione dell'architettura del sistema. Infine, verranno discussi i controlli effettuati per validare il sistema e i risultati ottenuti.

2 Background

L'algoritmo di Chandy-Lamport è stato introdotto da K. Mani Chandy e Leslie Lamport come soluzione per ottenere uno *snapshot* globale consistente in sistemi distribuiti asincroni.

2.1 Assunzioni

L'algoritmo assume che:

- La comunicazione sia affidabile, ossia né i canali né i processi possono fallire, così che ogni messaggio inviato sia eventualmente ricevuto intatto esattamente una volta.
- I canali di comunicazione tra un processo e l'altro siano unidirezionali e la consegna dei messaggi avvenga in ordine FIFO.
- Il grafo dei processi e dei canali sia fortemente connesso, ovvero deve essere sempre previsto un percorso tra due processi del sistema.
- Ogni processo deve essere in grado di continuare nella sua esecuzione ed inviare o ricevere messaggi durante l'esecuzione di uno *snapshot*.

2.2 Funzionamento

Definisco con *canali in entrata* ad un processo pi i canali da cui pi riceve messaggi da parte degli altri processi del sistema; mentre con *canali in uscita* i canali tramite cui pi invia messaggi ad altri canali.

L'algoritmo prevede l'utilizzo di messaggio speciali detti *marker* e i seguenti passi:

1. Quando un processo del sistema decide di iniziare l'esecuzione di uno *snapshot*, invia un messaggio di *marker* a tutti i processi del sistema sui *canali in uscita*.
2. Quando un processo riceve un messaggio di *marker* su un *canale in entrata*, dovrà comportarsi in modo diverso a seconda che il *marker* ricevuto sia il primo *marker* o no:
    a. Nel caso in cui questo fosse il primo *marker*, il nodo deve procedere registrando il suo stato interno,

etichettando il *canale in entrata* da cui ha ricevuto il *marker* come "vuoto" ed inviando a sua volta dei messaggi *marker* a tutti i processi del sistema.

b. Nel caso in cui questo non fosse il primo *marker* ricevuto, il processo deve procedere etichettando il canale da cui ha ricevuto il *marker* come "chiuso", perciò i successivi messaggi che riceverà su quel canale non apparterranno più allo *snapshot* che si sta realizzando.

3. Non appena ogni processo avrà ricevuto i messaggi di *marker* da ogni processo del sistema, l'algoritmo potrà terminare.

2.3 **Proprietà**

L'algoritmo consente la creazione di un'immagine globale dello stato del sistema senza la necessità di dover interrompere il sistema stesso.
Inoltre preserva le relazioni causali tra gli eventi, infatti lo stato globale catturato con lo *snapshot* riflette l'ordine causale tra gli eventi nel sistema distribuito.
Altro aspetto rilevante è il fatto che l'algoritmo termina in un tempo finito.

3 **Architettura del sistema**

Il sistema implementato prevede la presenza di due entità: un nodo, che si occuperà di tutte le operazioni interne al sistema e delle operazioni inerenti alla realizzazione dell'algoritmo, e un client.

Ogni nodo è stato progettato seguendo un'architettura modulare, pensata come una triade composta da tre componenti principali: un server RPC, responsabile della gestione delle comunicazioni tra client e nodo, un processo principale, che rappresenta la logica computazionale del nodo, e un processo di *snapshot*, incaricato di gestire la logica dell'algoritmo di Chandy-Lamport.

Le tre componenti che costituiscono ogni nodo comunicano tra di loro per mezzo di canali Go.

Il sistema realizzato è basato sul mantenimento di un bilancio interno ad ogni processo e, successivamente, la consistenza verrà verificata proprio controllando questo aspetto.

3.1 **Server RPC**

Il server RPC si occupa di fornire un'interfaccia unica a ciascun nodo per la ricezione delle richieste di invio di messaggi o di avvio di *snapshot*. A questo scopo vengono esposti due metodi:

- *SendMessage*. Metodo che si occupa di ricevere come argomenti l'identificativo del processo destinatario e il contenuto del messaggio, ovvero la quantità da inviare al processo.
  Il metodo si occupa poi di inviare, tramite un canale di comunicazione precedentemente istanziato, un messaggio di tipo *Command* alla componente *Process* in cui specifica come nome del comando "*SendMessage*", come *payload* il messaggio costruito a partire dagli argomenti ricevuti e un canale di comunicazione appena creato su cui il server si porrà in ascolto in attesa dell'esito dell'operazione di invio.
  Una volta ricevuta la risposta sul canale di risposta, il server restituirà al client RPC una stringa con l'esito dell'operazione.
- *StartSnapshot*. Metodo che si occupa di inviare un messaggio di "*StartSnapshot*" sul canale in comune con il processo di *snapshot*, per poi restituire l'esito al client RPC.

3.2 **Processo**

La componente *Process*, appena istanziata, avvia due *goroutines*:

- *listenOnChannel*, la quale avvia la *goroutine processMessageQueue* e si occupa di rimanere in attesa su una porta definita tramite il file *.env* per connessioni TCP da parte di altri processi. Una volta ricevuto un pacchetto su quella porta, ne passerà la gestione alla *goroutine handleConnection*.
- *RunCommand*, la quale si occupa di rimanere in attesa sul canale *ProcessChannel* sul quale riceve comandi da parte delle altre due entità del sistema, per poi gestirli avviando le *goroutines* responsabili. I possibili comandi sono:
  - *SendMessage*, che riceve dal server RPC ed avvia la *goroutine sendMessage*.
  - *SendMarkers*, che riceve da *SnapshotProcess* quando è stata eseguita la logica di creazione di uno snapshot e bisogna procedere all'invio dei *marker*. Lancia la *goroutine sendMarkers*.
  - *MutexRelease*, per cui si scenderà successivamente nei dettagli.
  - *StartingSnapshot*, per cui si scenderà successivamente nei dettagli.
  - *ForcedTermination*, che riceve dal *main* nel caso in cui il nodo riceva un segnale di tipo *SIGINT* o *SIGTERM* e che viene gestito semplicemente stampando a schermo il bilancio corrente del processo.

Le altre funzioni invocate sono:
- *handleConnection*: funzione che si occupa di gestire il messaggio ricevuto da parte di un altro processo. Questo messaggio verrà poi aggiunto ad una coda di messaggi unica per processo. La presenza della coda è necessaria poiché occasionalmente il processo deve bloccare temporaneamente l'elaborazione dei `messaggi` quando riceve un `marker.`
- *ProcessMessageQueue*: funzione che si occupa di processare il primo messaggio presente nella coda di messaggi. Il messaggio potrà essere di due tipi: *MESSAGE* oppure *MARKER*. Nel primo caso verrà invocata la funzione *handleMessage*, che si occupa di aggiornare il bilancio corrente del processo con la quantità ricevuta, e verrà inviato un comando di tipo *"MessageReceived"* sul canale *SnapshotProcessChannel* per la gestione del messaggio nel caso in cui sia attivo uno snapshot; mentre nel secondo caso viene inviato un comando di tipo "*MarkerReceived"* sul canale SnapshotProcessChannel per la gestione del marker.
- *sendMessage*: funzione che si occupa di inviare un messaggio al processo destinatario.
- *sendMarkers*: funzione che si occupa di iterare sul numero di processi attivi per inviare a ciascuno di loro un *marker* con l'id dello *snapshot* relativo.

Sono state utilizzate connessioni TCP in quanto nelle assunzioni dell'algoritmo è prevista comunicazione affidabile. Inoltre l'atomicità delle azioni previste quando si riceve un *marker* relativo ad uno *snapshot* che non è stato ancora avviato nel sistema è stata garantita nel seguente modo:
- Quando viene gestito un messaggio di *marker* in *handleConnection* o quando viene avviato uno *snapshot* (vedere il *case* "*StartingSnapshot*" in *runCommand*), viene effettuato un *lock* sul *mutex sendingMarkersMutex*.
- Viene effettuato l'*unlocking* di questo *mutex* dopo l'invio dei *marker* oppure, se il *marker* non è il primo *marker* dello *snapshot*, in *runCommand* quando viene ricevuto il comando "*MutexRelease"*.

La coda *msgQueue* è stata inserita al fine di garantire la consegna in ordine FIFO dei messaggi come previsto dalle assunzioni dell'algoritmo.

3.2 Processo di snapshot

Questa componente è quella che si occupa di gestire la logica di realizzazione dell'algoritmo. Appena istanziata, avvia la *goroutine listen*, la quale si occupa di rimanere in attesa sul canale *SnapshotProcessChannel* per eventuali comandi da parte del server RPC e da *Process*. I possibili comandi sono:
- *StartSnapshot*, per cui viene lanciata la *goroutine startSnapshot*.
- *MessageReceived*, per cui viene invocata la *goroutine handleMessageReceived*.
- *MarkerReceived*, per cui viene invocata la *gorouine handleMarker*.

Le funzioni invocate sono:
- *startSnapshot*: funzione che, come prima cosa, invia un comando di "*StartingSnapshot*" sul canale *ProcessChannel* per avvertire che va eseguito atomicamente l'avvio di uno *snapshot* con i relativi invii dei *marker*. Successivamente crea un'istanza di *Snapshot* per poi aggiungerla alla lista di *snapshot* correntemente attivi nel sistema e invia il comando "*SendMarkers"* nel canale *ProcessChannel*.
- *handleMarker*: funzione che scandisce la lista di *snapshot* attivi nel sistema per verificare se quello relativo al *marker* appena ricevuto è stato già attivato nel processo corrente. Se ancora non esiste, viene creato e aggiunto alla lista. Successivamente, in entrambi i casi, viene impostato il canale da cui è stato ricevuto il *marker* come *"empty"* (se non esiste un'istanza dello stato del canale relativo a questo canale) oppure *"done"* (se esiste un'istanza dello stato del canale contenente dei messaggi ricevuti dall'inizio dello *snapshot*). Infine viene impostato a *true* il campo della mappa *receivedMarkers* relativo al canale appena ricevuto e, se sono stati ricevuti tutti i *marker*, viene invocata *terminateLocalSnapshot*.
- *terminateLocalSNapshot*: funzione che si occupa di invocare la funzione *saveSnapshot* e di rimuovere l'istanza di *snapshot* dalla lista degli *snapshot* attivi nel sistema.
- *saveSnapshot*: funzione che salva lo *snapshot* in un file.

Poiché, nel caso in cui alla ricezione di un *marker* relativo ad uno *snapshot* già avviato, non si richiede l'invio dei *markers* agli altri processi, viene richiesto il rilascio del mutex per consentire al *Process* di continuare con l'invio dei messaggi tramite l'invio del comando "*MutexRelease*" sul canale *ProcessChannel* all'interno di *handleMarkers*.

3.3 Ambiente di sviluppo

Il *setup* del sistema è stato realizzato usando un'istanza EC2 di Amazon, configurata per eseguire container Docker che rappresentano singoli nodi del sistema distribuito ed il client.
L'utilizzo di Docker ha facilitato la distribuzione e la gestione dei nodi, semplificando la replica nell'ambiente di test.

4 Testing

Il *testing* viene realizzato facendo avviare al *mainClient* tante *goroutines* quanti sono i nodi del sistema, per simulare l'esecuzione concorrente delle richieste ai nodi.
Il codice di *testing* è stato definito nella funzione *RunTest* e prevede l'invio di messaggi e la richiesta di avvio di *snapshot*.
Nel file *bash* utilizzato per l'avvio del sistema è prevista la stampa su terminale delle seguenti informazioni:

- La somma dei bilanci iniziali di ciascun processo:

```
Bilancio iniziale totale: 6000
```

- Il contenuto di ogni file di *snapshot* creato al termine di ciascuna operazione di registrazione dello stato:

```
Analizzando il file: snapshots/process1/snapshot_1_process_1.json con snapshot ID: 1
{
  "timestamp": "2024-10-10T20:13:28.460572679Z",
  "snapshot_id": 1,
  "process_id": 1,
  "intern_state": 1000,
  "channel_state": {
    "2": [
      600
    ],
    "3": "empty"
  }
}
```

```
Analizzando il file: snapshots/process1/snapshot_2_process_1.json con snapshot ID: 2
{
  "timestamp": "2024-10-10T20:13:28.501888331Z",
  "snapshot_id": 2,
  "process_id": 1,
  "intern_state": 1600,
  "channel_state": {
    "2": "empty",
    "3": "empty"
  }
}
```

```
Analizzando il file: snapshots/process1/snapshot_3_process_1.json con snapshot ID: 3
{
  "timestamp": "2024-10-10T20:13:29.502316516Z",
  "snapshot_id": 3,
  "process_id": 1,
  "intern_state": 1600,
  "channel_state": {
    "2": "empty",
    "3": "empty"
  }
}
```

```
Analizzando il file: snapshots/process2/snapshot_1_process_2.json con snapshot ID: 1
{
  "timestamp": "2024-10-10T20:13:28.728438311Z",
  "snapshot_id": 1,
  "process_id": 2,
  "intern_state": 1410,
  "channel_state": {
    "1": "empty",
    "3": "empty"
  }
}
```

```
Analizzando il file: snapshots/process2/snapshot_2_process_2.json con snapshot ID: 2
{
  "timestamp": "2024-10-10T20:13:28.462042325Z",
  "snapshot_id": 2,
  "process_id": 2,
  "intern_state": 1400,
  "channel_state": {
    "1": "empty",
    "3": [
      10
    ]
  }
}
```

```
Analizzando il file: snapshots/process2/snapshot_3_process_2.json con snapshot ID: 3
{
  "timestamp": "2024-10-10T20:13:28.731968938Z",
  "snapshot_id": 3,
  "process_id": 2,
  "intern_state": 1410,
  "channel_state": {
    "1": "empty",
    "3": "empty"
  }
}
```

```
Analizzando il file: snapshots/process3/snapshot_1_process_3.json con snapshot ID: 1
{
  "timestamp": "2024-10-10T20:13:29.053336716Z",
  "snapshot_id": 1,
  "process_id": 3,
  "intern_state": 2990,
  "channel_state": {
    "1": "empty",
    "2": "empty"
  }
}
```

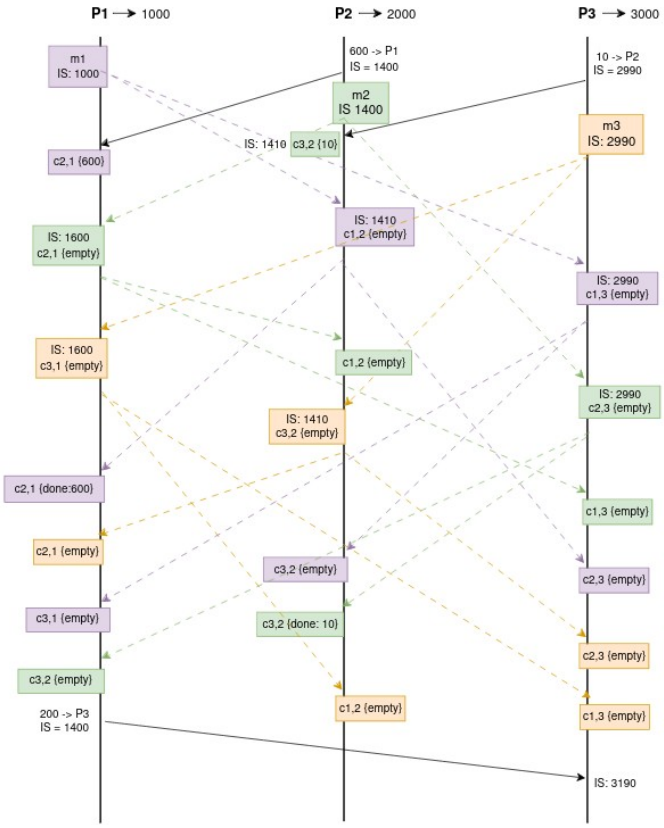
```
Analizzando il file: snapshots/process3/snapshot_2_process_3.json con snapshot ID: 2
{
  "timestamp": "2024-10-10T20:13:29.054270907Z",
  "snapshot_id": 2,
  "process_id": 3,
  "intern_state": 2990,
  "channel_state": {
    "1": "empty",
    "2": "empty"
  }
}
```

```
Analizzando il file: snapshots/process3/snapshot_3_process_3.json con snapshot ID: 3
{
  "timestamp": "2024-10-10T20:13:28.512576479Z",
  "snapshot_id": 3,
  "process_id": 3,
  "intern_state": 2990,
  "channel_state": {
    "1": "empty",
    "2": "empty"
  }
}
```

- L'*output* del confronto tra la somma dei bilanci iniziali e la somma dei bilanci ottenuti, per ciascuno *snapshot*, sommando i valori dello stato interno e dello stato dei canali.

```
Bilancio totale per Snapshot ID 3: 6000
Successo: Il bilancio totale dello Snapshot con ID 3 corrisponde al bilancio iniziale.
Bilancio totale per Snapshot ID 2: 6000
Successo: Il bilancio totale dello Snapshot con ID 2 corrisponde al bilancio iniziale.
Bilancio totale per Snapshot ID 1: 6000
Successo: Il bilancio totale dello Snapshot con ID 1 corrisponde al bilancio iniziale.
```

La sequenza di operazioni eseguite è consultabile nel seguente grafico:



Resta comunque possibile consultare la sequenza di operazioni eseguite da ogni processo eseguendo il comando *"sudo docker logs <container_name>"*.
Dai risultati risulta evidente che il sistema è riuscito a catturare correttamente uno stato globale consistente. Gli *snapshot* ottenuti dai vari processi sono stati verificati e i bilanci interni dei processi con i messaggi nei canali corrispondono al bilancio iniziale, confermando che l'algoritmo conserva il bilancio senza perdite.
Inoltre risulta verificato che i messaggi sono stati inviati e ricevuti in maniera ordinata e corretta.
Il sistema ha mantenuto un carico di rete accettabile durante l'invio dei *marker* e dei messaggi. Tuttavia durante il test si è notato un leggero aumento della latenza in alcuni processi a causa della gestione simultanea dell'invio di *marker* e messaggi di dati, in particolare nei nodi con più connessioni attive.

5 Possibili miglioramenti

Durante il test si è riscontrato un leggero *overhead* dovuto all'uso del *lock* per la gestione dei *marker*. Si potrebbe esplorare l'uso di meccanismi di sincronizzazione più efficienti al fine di minimizzare il tempo di attesa dei processi.
Inoltre, i i test attuali sono stati realizzati usando un numero limitato di nodi e messaggi, sarebbe utile eseguire ulteriori test con un carico di lavoro maggiore.

6 Conclusioni

Nel complesso il test eseguito indica che il sistema implementato è in grado di catturare più *snapshot* consistenti e di mantenere la coerenza dello stato globale, come richiesto dall'algoritmo di Chandy-Lamport.
Con alcuni miglioramenti nella gestione della concorrenza e della latenza, il sistema può essere ulteriormente ottimizzato per garantire prestazioni migliori in ambienti più complessi.

References

[1] George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair: *Distributed Systems: Concept and Design*. 5Th ed. Addison-Wesley, 2011.

[2] Federica Cantelmi. "Progetto SDCC: Implementazione dell'algoritmo di Chandy-Lamport per la creazione di uno snapshot globale." GitHub repository, 2024. Disponibile online: https://github.com/federicacantelmi/chandy-lamport-SDCC.git