Chandy-Lamport Distributed Algorithm per il corso di SDCC

Massimo Buniy, 0350022

***Abstract*—In questo paper vado ad esporre la mia implementazione dell'algoritmo di Chandy-Lamport per il progetto del corso di Sistemi Distribuiti e Cloud Computing tenuto dalla professoressa V. Cardellini. Ho realizzato una libreria scritta in GoLang e che utilizza gRPC per effettuare le chiamate a procedura remota.**

***Index Terms*—Chandy-Lamport, peer, service registry, interceptor, reflection.**

I. Introduzione

Per il progetto del corso di Sistemi Distribuiti e Cloud Computing, tenuto dalla professoressa V. Cardellini, sono andato a realizzare una libreria scritta in GoLang [1] che implementasse l'algoritmo di Chandy-Lamport [2] per fare lo snapshot globale e consistente del sistema distribuito.

Come tecnologie si è andati ad utilizzare il linguaggio *GoLang* [1] come da specifica di progetto. I vantaggi nell'utilizzare questo linguaggio sono state la facile utilizzazione di chiamate a procedura remota, l'avere a disposizione strumenti come *gRPC* [3] ed in fine il garbage collector il quale evitava di dover stare a gestire anche la memoria.

Per quanto riguarda la comunicazione tra i vari peer sono state utilizzate le chiamate a procedura remota come se queste fossero dei messaggi scambiati tra un client ed un server. In particolare, come strumento per le chiamate RPC si è scelto di adoperare *gRPC* [3] sia per l'utilizzo dei file proto, che semplificano la lettura e lo sviluppo di una procedura remota, sia perché mette a disposizione in maniera semplice l'*interceptor* [4], strumento utile per riuscire ad intercettare le chiamate a procedura remota in entrata.

Per il salvataggio dello snapshot all'interno di un file si è fatto ricorso al *package gob* [5]. Questo package semplifica il salvataggio e il recupero di strutture complesse da file in memoria salvando il tipo di dato per come questo è, semplificando la fase di serializzazione.

Proseguendo, è doveroso andare a citare la repository su GitHub del progetto *grpCurl* [6]. Da questa repository ho tratto spunto sul come riuscire a fare chiamate a procedura remota possedendo i dati da passare come parametri alla procedura remota e la firma della procedura remota ma senza ricevere fisicamente una richiesta da un altro peer. Inoltre mi ha permesso di venire a conoscenza della *reflection* [7] in GoLang, ossia uno strumento che permette di ispezionare i tipi dei dati, delle strutture e dei valori a tempo di esecuzione, consentendo operazioni dinamiche fondamentali per l'implementazione di questo algoritmo come libreria.

II. Algoritmo di Chandy-Lamport

Come primo passo, sono andato a studiarmi l'algoritmo pensato da Chandy-Lamport per la realizzazione di uno snapshot globale e coerente di un sistema distribuito. Per fare questo ho per prima cosa fatto riferimento al paper ufficiale [2] pubblicato nel 1 febbraio 1985, nel quale ho trovato alcune difficoltà negli aspetti più tecnici e delicati dell'algoritmo.

A causa di ciò, andando alla ricerca di altre fonti, mi sono imbattuto nel blog della professoressa *Lindsey Kuper* [8], ricercatrice nel campo dell'informatica ed attualmente affiliata all'Università della California, Santa Cruz. Grazie alla sua spiegazione passo passo dell'algoritmo e alla *videoregistrazione* [9] di una delle sue lezione, nella quale è scesa nel dettaglio di questo algoritmo, sono riuscito ad avere un'idea chiara sul come funzionasse l'algoritmo, ritrovandomi nella spiegazione disponibile su *Wikipedia* [10].

A. Assunzioni

Le assunzioni dell'algoritmo sono le seguenti:

- Non ci sono guasti e tutti i messaggi arrivano intatti e solo una volta (comunicazione affidabile).
- I canali di comunicazione sono unidirezionali e ordinati FIFO.
- Esiste un percorso di comunicazione tra due processi qualsiasi del sistema.
- Qualsiasi processo può avviare l'algoritmo di snapshot.
- L'algoritmo di snapshot non interferisce con la normale esecuzione dei processi.
- Ogni processo del sistema registra il proprio stato locale e lo stato dei propri canali in ingresso.

B. Algoritmo

Definite le assunzioni necessarie, precediamo vedendo nell'astratto il funzionamento dell'algoritmo. Possiamo andare a vedere questo algoritmo da due punti di vita, processo *inizializzatore*, ossia il processo che da il via all'algoritmo, e generico processo P_i che *riceve un messaggio "marker"* sul canale C_{ki} (messaggio inviato dal peer k tramite il canale che collega il peer k al peer i).

Per quel che riguarda il processo inizializzatore:

1) il processo registra il suo stato
2) manda un messaggio di tipo "marker" a tutti i suoi canali uscenti
3) inizia a registrare i messaggi che riceve, esclusi i messaggi marker, da tutti quanti i canali

Per quel che riguarda il processo generico P_i che riceve un messaggio di tipo "marker" sul canale C_{ki}:

- Se è la prima volta che P_i riceve un messaggio "marker" (sia inviato che ricevuto):
  1) P_i registra il suo stato
  2) marca il canale C_{ki} come *vuoto*
  3) invia un messaggio "marker" a tutti i suoi canali in uscita
  4) inizia a registrare i messaggi provenienti da tutti i canali non ancora marcati
- Altrimenti:
  1) P_i smette di registrare il canale C_{ki}

L'algoritmo termina una volta che il processo P_i ha ricevuto almeno una volta il messaggio "marker" da tutti i canali a cui ha inviato un messaggio di marker a sua volta.

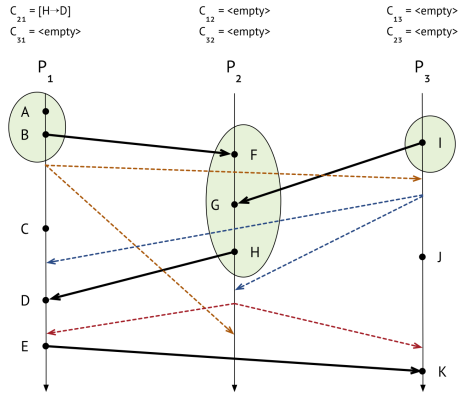

Fig. 1. Esempio applicazione distribuita. Sopra riusciamo a vedere i messaggi salvati dal singolo peer rispetto ai vari canali. Gli ovali verdi invece rappresentano lo snapshot del sistema, quindi cosa si è andati a salvare e dove termina lo snapshot.
Fonte: Lindsey Kuper, URL: chandy-lamport-8.png

Quello che si ottiene alla fine dell'algoritmo è il seguente stato del sistema:

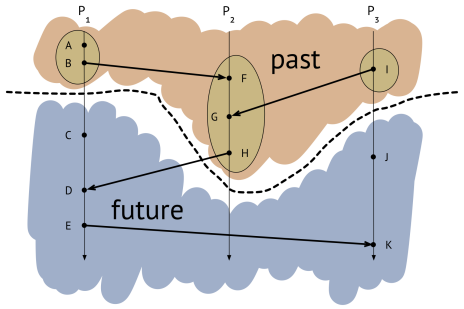

Fig. 2. Risultato finale dell'esecuzione dell'algoritmo. Possiamo notare come vi sia una separazione consistente tra quello chiamato "past" e quello chiamato "future", senza alcun messaggio che violi il taglio andando dal futuro al passato.
Fonte: Lindsey Kuper, URL: chandy-lamport-8-final-snapshot.png

III. Libreria "ChLamLib"

Il nome del package GoLang da me sviluppato è `ChLamLib`. Nella repository GitHub [11] lo si trova nell'omonima cartella.

All'interno sono presenti i seguenti file:

- `chLam.proto` - contiene il servizio gRPC utilizzato dalla libreria per riuscire a fare lo snapshot distribuito.
- `chLam.go` - contiene
  - l'implementazione della procedura remota `ChLamSnapshot` dichiarata nel file proto.
  - la funzione `NewServer`, invocata dall'utilizzatore della libreria per riuscire ad ottenere un server gRPC che abbia registrato la procedura remota `ChLamSnapshot`.
  - la funzione `RetrieveDataSnapshot` da invocare quando si vuole riprendere l'esecuzione a partire da uno snapshot.
  - la funzione `RestoreMethodsSnapshot` che va invocata successivamente a `RetrieveDataSnapshot` per riuscire a ripristinare anche i messaggi presenti nello stesso file di snapshot. La scelta di implementare due funzioni separate per il ripristino dello snapshot è dovuta al fatto che quando si viene invocata la prima, questa restituisce una `interface{}`, data dall'utilizzatore della libreria, contenente tutti i dati dello snapshot da ripristinare. Per evitare restrizioni sui parametri di una possibile funzione che ripristini lo stato del processo, si è scelto di lasciare all'utilizzatore il compito di invocare questa funzione di libreria.
- `interceptor.go` contiene l'implementazione dell'`interceptor` invocato da gRPC all'arrivo di chiamate a procedura remote. In particolare questo viene utilizzato per implementare la parte dell'algoritmo che si occupa di memorizzare tutte le richieste in entrata mentre si sta realizzando lo snapshot e di salvare in un file lo snapshot una volta raccolti i messaggi "marker" da tutti quanti i peer.
  Inoltre in questo file si trova la funzione `callGRPCMethod` se serve ad invocare i le procedure remote a partire dal loro nome. Per funzionare correttamente occorre passare alla funzione non solo i parametri da dover passare poi all'implementazione della procedura remota, ma anche un client, ossia un `grpc.ClientConnInterface`, necessario per l'invocazione della corretta procedura remota. Per ottenere questo si chiede all'utilizzatore della libreria di implementare una funzione con una specifica firma che dato il nome del metodo restituisca, tramite `interface{}`, la connessione al client corretta.
- `boundaryChLam.go` - contiene tutte quante le funzioni di libreria che un suo utilizzatore deve richiamare prima di poter utilizzare correttamente la libreria. Si occupano infatti di impostare correttamente tutte quante le variabili globali di libreria utilizzate dalle varie funzioni.

Oltre ai file citati è presente la cartella utils, la quale racchiude il file `gobFileManagement.go` e che contiene i due metodi per andare a scrivere sul file gob e per andare a riprendere il contenuto dal file gob.

A. Utilizzazione della libreria

Per quando riguarda il setup per al corretta utilizzazione di questa libreria occorre:

- ottenere il `grpc.Server` su cui registrare tutti i servizi dell'utilizzatore tramite la funzione di libreria
  `chLam.NewServer()`
- registrare l'indirizzo sul quale sarà attivo il server per la ricezione di richieste gRPC
  `chLam.RegisterServerAddr(peerServiceAddr)`
- registrare il nome del file che si vuole utilizzare per salvare/riprendere lo snapshot
  `chLam.RegisterSnapFileName(snapFileName)`
- per ogni nuovo peer di cui si viene a conoscenza, occorre registrare l'indirizzo sul quale questo è in ascolto anche nella libreria tramite la funzione di libreria
  `chLam.RegisterNewPeer(peerToRegister.Addr)`
- registrare il puntatore alla funzione per fare lo snapshot del sistema. La funzione deve avere la seguente firma:
  `func() (interface{}, error)`
  e restituire nell'interface un puntatore ad una singola struttura nella quale è contento l'intero snapshot di sistema. La struttura è a discrezione dell'utilizzatore della libreria
  `chLam.RegisterFunctionForSnapshot(peerSnapshot)`
- registrare il puntatore alla funzione che abbia come firma
  `func(string) (interface{}, error)`
  e che restituisca nell'interface un puntatore ad un oggetto `grpc.ClientConnInterface`
  `chLam.RegisterFunctionRetrieveChLamClient(retrieveChLamClient)`
- tramite la funzione di libreria
  `chLam.RegisterType(ptr interface{})`
  occorre registrare tutte le strutture utilizzate all'interno della libreria. Questo passaggio è fondamentale per riuscire a riprendere dal file gob un puntatore correttamente costruito. Nel codice fornito questo avviene o così
  `chLam.RegisterType((*pb.Peer)(nil))`
  oppure
  `chLam.RegisterType((*CountingSnapshot)(nil))`
  In entrambi i casi occorre passare alla funzione di libreria un puntatore alla struttura che si vuole andare a registrare.

Una volta fatti questi passi la libreria risulta completamente operativa ed è possibile:

- andare a riprendere lo snapshot utilizzando la funzione di libreria
  `chLam.RetrieveDataSnapshot(snapFileName)`
  che restituisce una `interface{}`. Per riuscire ad utilizzarla correttamente, questa va castata
  `var snapStruct *CountingSnapshot = data.(*CountingSnapshot)`
- iniziare lo snapshot, la libreria mette a disposizione la seguente funzione
  `chLam.StartSnapshot()`
  Invocandola si da inizio allo snapshot seguendo l'algoritmo di Chandy-Lamport.

B. Dettagli dell'implementazione

Adesso andiamo a vedere il funzionamento della libreria nel dettaglio. Per prima cosa andiamo a vedere le strutture definite e le variabili globali dichiarate. Per quanto riguarda le strutture definite troviamo:

```
type interfaceSnap struct {
    Bytes []byte
    Type  string
}
```

Struttura 1. Questa struttura viene utilizzata di default per riuscire a convertire i tipi di dato in un vettore binario e per mantenere in `Type` la stringa del tipo salvato. Il tipo viene mappato tramite una mappa globale che vedremo successivamente.

```
type methodSnap struct {
    Req         interfaceSnap
    ServiceName string
    MethodName  string
}
```

Struttura 2. Con questa struttura si salva in `ServiceName` il nome del servizio remoto che si vuole chiamare, in `MethodName` si salva invece il nome della specifica procedura remota che il servizio `ServiceName` mette a disposizione ed in `Req` si vanno a salvare i parametri da passare alla procedura remota invocata. In particolare `Req` è di tipo interfaceSnap, definito sopra, per rendere la `interface{}` serializzabile

```
type snapWrap struct {
        Data interfaceSnap
        MethodList []methodSnap
}
```

Struttura 3. Questa struttura è quella che poi verrà effettivamente salvata nel file gob per essere poi recuperata. È composta da due campi: il primo `Data` serve a memorizzare lo snapshot che viene prodotto dalla funzione passata dall'utilizzatore della libreria tramite la funzione `chLam.RetrieveDataSnapshot`, mentre il campo `MethodList` serve a memorizzare le chiamate a procedura remota ricevute durante la fase di snapshot, memorizzandole in una lista di `methodSnap`

Per le variabili globali invece:

```
peerServerAddr = chLamProto.ChLamPeer{Addr: ""}
```

Variabile 1. Nella variabile `peerServerAddr` viene memorizzato l'indirizzo sul quale il peer starà in ascolto per riceve chiamate a procedura remota. Viene passato come parametro alla procedura remota ChLamSnapshot che la utilizza per mappare quali peer gli hanno invito il messaggio di marker e quali no.

```
peerAddrList []string
```

Variabile 2. Questa variabile mantiene una lista di stringhe, in cui ogni stringa identifica l'indirizzo del server sul quale un altro peer si è messo in ascolto per ricevere richieste di chiamata a procedura remota. Questa lista viene popolata tramite la funzione di libreria `chLam.RegisterNewPeer()` ed utilizzata per controllare se tutti quanti i peer abbiano inviato il messaggio "marker". Se durante l'invio di un messaggio di "marker" a un peer si riceve un errore, si procede rimuovendo quel peer dalla lista.

```
peerMap = make(map[string]bool)
```

Variabile 3. Questa variabile serve a registrare tutti i peer da cui è arrivato il messaggio di "marker". Inoltre, viene utilizzato per stabilire se il processo di snapshot sia terminato o meno, confrontando il numero di marker ricevuto con il numero di stringhe nella lista `peerAddrList`.

```
snapFileName string
```

Variabile 4. Serve solamente a memorizzare il nome del file su cui andare a scrivere lo snapshot una volta completato. Questo viene impostato tramite la funzione di libreria `chLam.RegisterSnapFileName`.

```
var takePeerSnapshot func()
 ↪  (interface{}, error)
```

Variabile 5. Questa variabile ha il compito di memorizzare un puntatore a funzione con la firma lì specificata. In particolare, questa variabile viene impostata dalla funzione di libreria `chLam.RegisterFunctionForSnapshot` e la funzione passata deve essere definita dall'utilizzatore della libreria. Tremite questa funzione si va a recuperare lo snapshot del peer, restituito tramite una singola struttura.

```
var retrievePeerClient
 ↪  func(string) (interface{},
 ↪  error)
```

Variabile 6. Come prima, sempre un puntatore a funzione che memorizza la funzione per prendere il `grpc.ClientConnInterface` data la stringa che identifica la procedura remota che si vuole invocare. Questa viene impostata tramite la sua apposita funzione di libreria che è `chLam.RegisterFunctionRetrieveChLamClient`.

IV. Applicazione distribuita in pipeline

L'applicazione distribuita su cui è stata testata la libreria si compone di tre componenti: un primo peer si occupa di andare a leggere un file riga per riga e di inviare questa ad un secondo peer; il secondo peer si occupa di andare a contare le parole che occorrono nella frase ed inviare il conteggio alla fine della riga ad un terzo peer; il terzo peer si occupa di andare a fare la somma totale delle parole che erano presenti nel file e preoccupandosi di mantenere il conteggio totale.

Fig. 3. Stilizzazione dell'applicazione distribuita in pipeline

A. Integrazione della libreria nell'applicazione

Per simulare l'inserimento della libreria all'interno di un'applicazione già sviluppata ho deciso di sviluppare in un progetto prima l'applicazione pipeline e dopo aver terminato lo sviluppo della libreria, tentare di riportare questa all'interno dell'applicazione. Questo passaggio però non è stato banale. Ho dovuto modificare buona parte del codice per riuscire ad adattare l'applicazione a se stante con la libreria ChLamLib.

B. Problemi dell'applicazione sviluppata

Nella fase di test è emerso un problema con l'applicazione. Infatti, dovendo gestire la concorrenza su determinate strutture e variabili, ho fatto un utilizzo massiccio di mutex, cercando di ridurre al minimo il loro utilizzo. Tuttavia, in maniera totalmente casuale, mi è capitato di riscontrare un crash a causa di una modifica concorrente su una mappa. Non sono stato in grado di identificare quele fosse la mappa nello specifico, essendosi il problema verificato solamente un paio di volte in maniera totalmente casuale.

V. Conclusione

Avendo prima dell'implementazione della libreria sviluppato un progetto nel quale l'algoritmo di Chandy-Lamport fosse completamente in simbiosi con l'applicazione, posso andare ad esprimere alcune osservazioni sulle due fasi di sviluppo.

Per quel che riguarda l'applicazione completamente legata all'algoritmo, la sua fase di sviluppo è stata molto più lineare. Infatti, avendo piena conoscenza dei costrutti utilizzati dall'applicazione sovrastante, era possibile andare ad implementare in maniera abbastanza semplice una logica che permettesse di eseguire uno snapshot distribuito. Al contrario però richiedeva il fatto di dover ben conoscere le fasi dell'algoritmo e implementare un modo per riuscire a distinguere tra messaggi "normali" e messaggi "marker". Nella soluzione che avevo sviluppato non vi era un altro servizio on un metodo apposito dichiarato nel file proto, ma bensì si andava a specificare direttamente nei parametri che si passavano alla procedura remota il tipo di messaggio che si era deciso di inviare.

Per l'implementazione tramite la libreria, questa ha richiesto lo sviluppo di funzioni apposite ed in uno specifico formato, per riuscire a gestire l'interazione tra il sistema sviluppato e la libreria. Il problema principale era infatti che la libreria non poteva fare nessuna assunzione sul come il sistema venisse sviluppato, per cui tutta una serie di parametri prima facilmente ottenibili dal normale flusso dell'applicazione, ora dovevano esplicitamente venir passati alla libreria (ad esempio l'indirizzo dei peer). L'aspetto che sicuramente favorisce l'utilizzo della libreria è la sua riutilizzabilità, ma per riuscire ad aver questo si deve andare a fornire funzioni anche non banali.

References

[1] A. A. A. Donovan and B. W. Kernighan, *The Go Programming Language*, Addison-Wesley, 2015.

[2] Leslie Lamport and K. Mani Chandy, "Distributed snapshots: Determining global states of a distributed system," https://lamport.azurewebsites.net/pubs/chandy.pdf, 1985.
[3] "grpc," https://grpc.io/.
[4] "Interceptors — grpc," https://grpc.io/docs/guides/interceptors/.
[5] The Go Authors, "gob package documentation," https://pkg.go.dev/encoding/gob.
[6] F. Developers, "grpcurl," https://github.com/fullstorydev/grpcurl.
[7] "The laws of reflection - the go programming language," https://go.dev/blog/laws-of-reflection.
[8] Lindsey Kuper, "An example run of the chandy-lamport snapshot algorithm," https://decomposition.al/blog/2019/04/26/an-example-run-of-the-chandy-lamport-snapshot-algorithm/, 2019.
[9] L. Kuper, "Cse138 (distributed systems) l6: Chandy-lamport snapshot algorithm," https://youtu.be/WK3FuD7f9g8.
[10] Wikipedia, "Chandy-lamport algorithm," https://en.wikipedia.org/wiki/Chandy%E2%80%93Lamport_algorithm.
[11] Massimo Buniy, aka Ralisin, "Chandy-Lamport Library Project," https://github.com/Ralisin/Chandy-Lamport_GoLibrary.