Sistemi Distribuiti e Cloud Computing - A.A. 2020/21
Progetto B1: Multicast totalmente e causalmente ordinato in Go

Marco Calavaro 0295233
Dipartimento di Ingegneria Civile e Ingegneria Informatica
Università degli Studi di Roma "Tor Vergata"
mar.calavaro@gmail.com

1 Traccia

Lo scopo del progetto `e realizzare nel linguaggio di programmazione Go un'applicazione distribuita che implementi gli algoritmi di multicast totalmente ordinato e causalmente ordinato. L'applicazione deve soddisfare i requisiti elencati di seguito.

- Un servizio di registrazione dei processi che partecipano al gruppo di comunicazione multicast. Si assuma che la membership sia statica durante l'esecuzione dell'applicazione; quindi, non vi sono processi che si aggiungono al gruppo od escono dal gruppo durante la comunicazione.
- Il supporto dei seguenti algoritmi di **multicast**:
    1. multicast **totalmente ordinato** implementato in modo centralizzato tramite un **sequencer**
    2. multicast **totalmente ordinato** implementato in modo decentralizzato tramite l'uso di **clock logici scalari**
    3. multicast **causalmente ordinato** implementato in modo decentralizzato tramite l'uso di **clock logici vettoriali**

Si richiede di testare il funzionamento degli algoritmi implementati nel caso in cui vi `e un solo processo che invia il messaggio di multicast e nel caso in cui molteplici processi contemporaneamente inviano un messaggio di multicast; 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 in un intervallo predefinito. Si richiede di fornire un container Docker per l'esecuzione dell'applicazione. Si progetti l'applicazione ponendo particolare cura al soddisfacimento dei requisiti sopra elencati. Si richiede inoltre che gli eventuali parametri relativi all'applicazione e al suo deployment siano configurabili.

2 Descrizione dell'architettura

In questo capitolo saranno riportati i dettagli sull'architettura sviluppata, incentrandosi sulle tecnologie e scelte adottate per poter completare l'obbiettivo dello studio.

Lo sviluppo del progetto è stato effettuato tramite il linguaggio Go e la simulazione dei vari peer necessari alla comunicazione tramite tecnologia dei container, sfruttando docker e docker-compose. (Saranno trattate nel capitolo riguardante la piattaforma software)

2.1 Configurazione statica

Come richiesto nel progetto, i vari parametri di configurazione necessari al funzionamento dell'applicativo vengono configurati staticamente mediante alcuni file. Per poter avere una completa separazione tra l'applicativo (codice Go) e la piattaforma utilizzata per il deploy (docker) si è scelto di utilizzare file separati, che devono essere mantenuti congruenti "manualmente", nello specifico i file in questione sono:

- /utility/static_config.go : file dell'applicativo utilizzato per durante l'esecuzione per recuperare i parametri "statici"
- .env mantiene i parametri del docker-compose.yml che devo essere congruenti con quelli del precedente file

Come detto in precedenza questa scelta porta a dover ripetere la configurazione ma allo stesso tempo separa la piattaforma di deploy dall'applicativo. Si è valutata anche la soluzione con il passaggio di tali valori direttamente come variabili di ambiente, ma si è preferito mantenere questa struttura che permette l'esecuzione dell'applicativo direttamente con i comandi Go.

2.2 Tipologia nodi sviluppati

Per poter soddisfare i requisiti richiesti nel progetto è stato scelto di sviluppare tre tipologie di nodi differenti. Di seguito sarà brevemente descritto lo scopo di ciascun nodo, focalizzato sulla sua architettura.

2.2.1 Nodo Register

Questo nodo implementa il servizio di registrazione (sarà trattato nel capitolo implementazione) che permette ai processi di inserirsi nel gruppo di comunicazione. Per la natura statica

dell'applicazione tale nodo è contattato solo nella fase di startup. Solo al raggiungimento del numero massimo di peer, configurato staticamente, il servizio restituisce l'elenco dei nodi nel sistema. (Tale elenco è sfruttato nell'invio di mesaggi multicast)

2.2.2 Nodo Sequencer

Nodo che implementa il ruolo del sequencer previsto dal primo algoritmo di multicast. Riceve messaggi dai peer e ne assegna un ordinamento sequenziale, restituendoli ai peer.

2.2.2 Nodo Peer

È il nodo che implementa la maggior parte della logica dell'applicativo. In esso è stato codificato l'invio dei messaggi e la ricezione, inoltre per avere una interazione più semplice con l'utente è stata implementata una interfaccia grafica simile alla shell Linux che permette di inserire un set limitato di comandi. (La sua implementazione è trattata nel capitolo)

2.3 Organizzazione package

L'applicazione è stata organizzata in package, sfruttando le funzionalità native del Go. Ne segue un breve elenco e descrizione. Il target node tra parentesi rappresenta su quale nodo il package viene "installato", se non presente vuol dire che il target node corrisponde al nome del package.

2.3.1 Menu (target node peer)

Package che implementa il codice per il frontend del peer. Il codice è stato sviluppato basandosi su un progetto github (https://github.com/turret-io/go-menu) opportunamente modificato per l'utilizzo nell'applicazione.

2.3.2 Register

Package per il codice del register node, internamente il package Go è main.

2.3.2 Sequencer

Strutturato come il precedente package, contiene il codice del sequencer.

2.3.3 Peer

Package che contiene il codice per il peer node, anche qui internamente il package Go è main, ma alcune considerazioni nate durante lo sviluppo dei test, quindi nelle fasi finali del progetto, hanno evidenziato che si sarebbe potuto creare una maggiore modularità creando un subpackage Go che contenesse la logica del peer e il main per il solo avvio del peer. In questo modo si sarebbe potuta sviluppare un'architettura di test che prevedesse un apposito package.

2.3.4 Utility (taget node all)

Questo package viene montato su ogni tipologia di nodo, esso mantiene del codice di utility organizzato in file il cui nome cerca di descriverne il contenuto. Alcune di queste utility non sono necessariamente utilizzate da ogni nodo, ma per evitare duplicazione di codice si è optato per un unico package utility piuttosto che averne diversi per ciascun nodo.

2.3 Organizzazione docker

Il progetto è organizzato in un solo contesto che corrisponde alla cartella root, nella quale si trova il file di configirazione per docker-compose (nominato docker-compose.yml), all'interno di questo file vi è il link per ogni nodo al DOKERFILE da utilizzare per il boot di esso.

2.3.1 Networks

Per poter far comunicare i peer è stata creata una rete locale, sfruttando i network di docker, l'assegnazione degli indirizzi segue le seguenti regole:

- Nodi di tipo register e sequencer hanno assegnazione statica
- Nodi di tipo peer assegnazione dinamica a runtime

2.3.2 Pesistenza

Per la persistenza sono state utilizzate due diverse tecnologie docker, soprattutto a scopo didattico, per poter avere memorizzazione di alcuni file necessari al funzionamento dell'applicativo.

In particolare, è stato utilizzato il **tmpfs,** che consiste in un file sistem temporaneo installato in memoria, esso permette di avere un isolamento tra container (a differenza per esempio dei volumi), simulando in questo modo macchine "separate" per ciascun nodo. Inoltre, poiché non vi era la necessità di mantenere i file prodotti durante l'esecuzione dell'applicazione in modo persistente al termine dell'applicazione, tale scelta è risultata vantaggiosa perché la memoria occupata viene liberata automaticamente alla chiusura (eliminando così i file prodotti).

La seconda tecnologia usata è stata quella dei **volumes**, utilizzata solo in contesto di testing e nei nodi peer, permettono di creare uno spazio di memoria condiviso tra container e poiché vi era la necessità di far condividere i risultati dell'esecuzione di un test tra i vari peer per effettuare un confronto tra i risultati, tale tecnologia ha semplificato tale lavoro.

2.4 Architettura per i test

I test sono stati sviluppati direttamente nel package main del peer, organizzati nei file peer_test_file.go che contiene il codice dei test e peer_test_utils.go che contiene il codice di alcune funzioni di secondarie per il run dei test.

I test vengono eseguiti in automatico configurando l'ambiente di test, tramite un parametro statico nel file di configurazione.

Alcuni test richiedono un clean completo delle strutture dati, per motivi di tempo si è scelto di eseguire i test in due run separate.

(Maggiori dettagli implementativi nel prossimo capitolo)

3 Descrizione dell'implementazione

In questo capitolo saranno trattate le scelte implementative e il funzionamento dell'applicativo realizzato.

3.1 Peer modi operandi

Come detto in precedenza il peer racchiude la maggior parte del cuore dell'applicativo, ed ha la possibilità di essere eseguito in due modalità, a seconda di com'è impostato il valore del parametro Launch_Test nel file di configurazione statica.

3.1.1 Normal mode

Corrisponde al normale utilizzo del peer, il valore di Launch_Test è impostato a false. Quando viene avviato in questa modalità, completata la registrazione, lancia il frontend per poter utilizzare "manualmente" gli algoritmi di multicast

3.1.2 Test mode

Corrisponde alla modalità che lancia i test, il valore di Launch_Test è impostato a true. Questa modalità non prevede interazione con l'utente ma una volta lanciata, completata la registrazione, va ad eseguire i test specificati nella funzione startTests() in peer_test_file.go.

3.2 Comunicazione tra nodi

La comunicazione rappresenta uno degli elementi centrali del progetto. All'avvio del peer in normal mode è possibile eseguire nella stessa run qualsiasi tipo di algoritmo di Multicast, ogni messaggio scambiato avrà un apposito campo per capire come deve essere processato.

I messaggi di registrazione sono trattati in modo differente rispetto a quelli di Multicast e anche l'approccio per l'invio non è uguale, di seguito sono riportate le due modalità sviluppate

3.2.1 Remote Procedure Call

È stato scelto di utilizzare questo meccanismo nella sola fase di registrazione, nella quale i nodi devono appunto effettuare una procedura remota situata nel nodo di registrazione.

Tale procedura prevede che si rimanga in attesa finché non si raggiunge il numero di nodi previsto dalla configurazione statica, l'attesa è implementata tramite l'uso di **sync.WaitGroup.** Ogni peer che si registra viene salvato su file e al completamento della procedura l'elenco viene restituito a ciascun nodo in attesa.

Si è usato direttamente il package go net/rpc per lo sviluppo sopra descritto.

3.2.2 Comunicazione diretta

Per quanto riguarda l'invio dei messaggi di multicast si è scelto di adottare un approccio diverso rispetto a quello descritto nel precedente capitolo. Questo perché la visione adottata per tali messaggi è stata quella di un servizio di basso livello che potrebbe far parte di un'applicazione più ampia, ad esempio per l'invio di messaggi di aggiornamento. Non vi era quindi la necessità di eseguire una vera e propria procedura remota, ma si è sviluppato un protocollo applicativo che si basa direttamente sull'utilizzo di TCP (tramite package net di GO).

Ogni messaggio viene inviato su una nuova connessione, anche se il frontend prevede l'invio di più messaggi tramite comando send, questa scelta lascia libertà su come sviluppare la funzione di send, separandola dall'implementazione del protocollo. Valorizzando quindi la visione descritta a inizio capitolo, si è previsto che dei peer inviassero un singolo messaggio di aggiornamento per volta. Inoltre, in questo modo non è stato necessario prevedere un messaggio che comunicasse l'effettiva chiusura della socket al termine dei messaggi generati dall'utente

L'invio inoltre come richiesto nella traccia prevede un delay casuale che viene applicato prima dell'operazione di send, per simulare ritardi nella rete, come richiesto nella traccia

3.2.3 Struct dei messaggi per il multicast

```
type Message struct {
        Type        MsgType
        SendID      int
        Date        string
        Text        string
        SeqNum      []uint64
}
```

Questa è la struct dei messaggi di multicast che viene creata a prescindere del tipo di algoritmo scelto, il compito di distinzione del tipo è dato al campo type. Avendo 4 tipi differenti e un'unica struct i campi su citati sono utilizzati a seconda di casi in modo differente.
L'invio e ricezione dei messaggi avviene tramite le funzionalità presenti nel package encoding/gob (Encoder, Decoder).
I messaggi possono avere i seguenti tipi:

- SeqMsg: messaggi generati nell'esecuzione del primo algoritmo, non vi sono note particolari sui campi se non che il SeqNum è uno slice di dimensione 1.
- ScalarClockMsg: messaggi generati nell'esecuzione del secondo algoritmo, utilizzano il SeqNum in modo similare ai SeqMsg per passare il valore del clock scalare.
- VectorClockMsg: messaggi generati nell'esecuzione del terzo algoritmo, utilizzano il SeqNum per l'invio del clock vettoriale.
- ScalarAck: sono i messaggi generati in risposta alla ricezione nel secondo algoritmo, in questo caso sono utilizzati i soli campi Type, SendId e Text, in particolare in text sarà presente la chiave che permette di identificare l'ack. La chiave è una stringa così composta: sendID + "-" + SeqNum.

3.2 Ricezione e consegna dei messaggi

Poiché l'applicazione prevede la possibilità di cambiare il tipo di comunicazione, tutte le strutture e funzioni necessarie alla ricezione vengono istanziate alla startup per ogni peer.

Le goroutine lanciate saranno trattate nei capitoli appositi per gli algoritmi.

Ogni peer utilizza un'unica porta per ricevere i messaggi e utilizza il campo Type per distinguerli ed eseguire la giusta risposta per il tipo riscontrato.

Si parla di ricezione da non confondere con la consegna effettiva all'utente. La consegna infatti dipende da quale algoritmo si sta utilizzando e segue le regole da esso imposte.

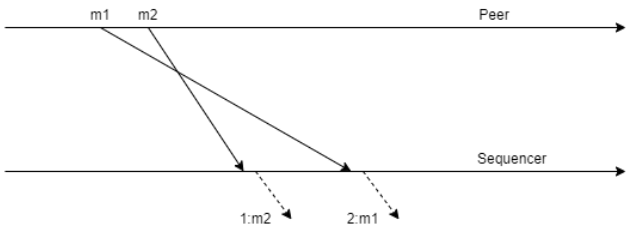
Quando un messaggio soddisfa le condizioni di consegna per il suo tipo viene lanciata la funzione save_msg, essa prevede un funzionamento leggermente diverso in caso i peer siano nello stato di test, è inoltre l'unica funzione con tale comportamento. Il salvataggio, che altro non è che la consegna del messaggio è stato realizzato tramite scrittura su specifico file, suddivisi per tipologia di algoritmo, dei dati relativi al messaggio.

La funzione prevede nel caso di test di consegnare il messaggio anche ad un apposito canale di testing chiamato testMsgChan.

3.3 Algoritmo 1: Sequencer

Il peer che vuole inviare un messaggio non crea una particolare struttura, ma invia direttamente la stringa (che rappresenta il Text del messaggio) al sequencer, il quale alla sua ricezione creerà la struct Message specifica da inviare ai vari peer.

La centralità di questo algoritmo e l'utilizzo di TCP garantisce che i messaggi una volta processati dal sequencer vengano ricevuti e consegnati nello stesso ordine in ciascun peer. Questo algoritmo non garantisce che ordinamento tra l'invio multiplo di messaggi da parte di un singolo peer.



Alla ricezione del messaggio dal sequencer, ciascun peer lo pone in una coda, implementata tramite buffer channel, per poterne gestire la consegna. Tale lavoro è delegato alla goroutine seqMsg_reordering che si occupa di leggere da tale canale e di consegnare il solo messaggio aspettato, confrontato il SeqNum con il valore locale. In questo modo è garantito che venga mantenuto l'ordinamento generato dal sequencer.

3.4 Algoritmo 2: Scalar clock

Il peer che vuole inviare il messaggio, crea la struct Message riempiendo i vari campi in modo adeguato ed effettua poi il send verso tutti i peer (anche sé stesso).

Alla ricezione del messaggio esso viene inserito in una coda in modo ordinato, l'ordinamento è fondamentale per garantisce la giusta consegna al livello applicativo. In seguito, viene lanciata una goroutine per messaggio che ha il ruolo di verificare ciclicamente che le due condizioni previste per la consegna dall'algoritmo vengano rispettate. In caso affermativo viene lanciata save_msg. Oltre a quest'ultima goroutine ne viene lanciata un'altra con il compito di invio degli ack.

La lista è implementata tramite il package List di Go, per garantire l'inserimento in ordine è stata sviluppata la funzione utility.InsertInOrder che pone il messaggio nel giusto "slot" basandosi sul SeqNum.

3.4 Algoritmo 3: Vector clock

Similmente al precedente algoritmo, viene creata la struct Message durante l'invio, riempiendo in modo corretto gli appositi campi.

Quando viene ricevuto un messaggio vi è la discriminazione del sendId, questo perché se coincide con l'id del peer (cioè è un messaggio che ritorna a sé stessi) non possono essere applicate le regole previste dall'algoritmo per la consegna. Se invece il valore è diverso il messaggio viene inserito in un buffered channel. (Non in modo ordinato)"

La gestione del buffer è delegata alla goroutine vectMsg_reordering che ciclicamente controlla i messaggi in coda e ne verifica le condizioni di consegna previste dall'algoritmo

3.5 Testing

In questo capitolo saranno brevemente trattate le varie funzioni e i test sviluppati.

I test sono stati creati seguendo un approccio modulare:

- La funzione di test deve essere del seguente tipo

```
func testName (testId int) bool
```

- La sua esecuzione avviene tramite

```
func executeTest ((testId int, testFunc func(testId int) bool)
```

3.5.1 Utils

Oltre alla funzione executeTest che esegue il test e ne salva i risultati, sono state sviluppate altre utility che permettono:

- L'invio di messaggi con delay.
- Attendere la consegna dei messaggi, anche con un timeout di attesa, la consegna salva i messaggi su un file per peer e per test.
- Confrontare i risultati ottenuti.

3.5.1 Test prodotti

In generale sono stati prodotti due test per algoritmo, uno con un solo sender e un secondo con tutti i peer che inviano i messaggi. Il codice è abbastanza simile per ogni test, ma vi sono alcune precisazioni da fare sul funzionamento di ognuno, segue quindi una breve descrizione per test.

Send singola algoritmo 1

Vengono inviati 10 messaggi al sequencer dal solo peer con ID 1, completata la ricezione si vanno a controllare i file generati dal test e si confronta che ogni peer veda l'ordine corretto dei messaggi.

Multi send algoritmo 1

Funzionamento simile al precedente algoritmo ma ogni peer invia i messaggi, 3 per peer, l'attesa è quindi di 3* MAXPEER

Send singola algoritmo 2

Vengono inviati 10 messaggi dal peer con ID 1 in questo caso ci si aspetta che non avvenga nessuna consegna dall'algoritmo perché non sono stati ricevuti messaggi da altri peer. È quindi necessario impostare un timeout sulla ricezione e il contenuto dei file prodotti deve essere nullo.

Multi send algoritmo 2

Ogni peer invia 3 messaggi in questo caso non ci si aspetta di ricevere tutti i messaggi ma che solo 2 riescano a essere consegnati.

Send singola e multipla algoritmo 3

Non vi sono particolari accorgimenti da riportare per tali test, il funzionamento è molto simile a quelli per il primo algoritmo sia per send singola che multipla.

4 Collegamento GUI e command

Quando viene lanciata l'applicazione con docker-compose vengono anche lanciati i vari peer, per potersi connettere ad essi basta utilizzare il comando docker attach su una nuova shell. Poiché il container target sarà già running dopo aver effettuato l'attach non sarà visualizzato il menu (si può utilizzare l'apposito comando per mostralo).

Per semplificare ulteriormente questo collegamento è stato sviluppato un semplice script bash (connect_peer.sh), esso si aspettta un argomento che corrisponde al numero del peer con il quale effettuare l'attach.

La GUI prevede l'utilizzo di comandi per interagire con l'applicativo, inizialmente viene lanciato il menu principale che permette di eseguire comandi legati al passaggio ad un menu secondario, uno per ogni tipo di comunicazione multicast.

3.2 Comandi generali

- menu: permette di visualizzare il menu corrente
- quit or exit: permette di uscire dal menu corrente

3.2 Comandi specifici

- Menu principale:
    - seq: accedere al sottomenu per comunicazione con sequencer
    - sc: accedere al sottomenu per comunicazione con clock scalare
    - vc: accedere al sottomenu per comunicazione con clock vettoriale

- Sottomenu:
    - send arg…: permette l'invio degli argomenti, il separatore di stringhe fa iniziare un nuovo argomento
    - show: mostra i messaggi che sono stati salvati su file

5 Limitazioni riscontrate

L'unica limitazione riscontrata riguarda la scelta di sviluppare gli algoritmi mediante comunicazione diretta senza implementare dei servizi "remoti". Per questo motivo sia il frontend che i test devono essere eseguiti direttamente sul peer.

Se invece si fosse scelto di implementare anche gli algoritmi come procedure remote (rpc) si sarebbe potuta creare un'architettura separata per eseguire sia il frontend che i test.

La scelta adottata non va ad impattare sulle funzionalità del programma e non risulta quindi essere una vera e propria limitazione.

Una eventuale soluzione che non vada ad impattare il funzionamento sviluppato per i tre algoritmi è quella di esportare la funzionalità di send come procedura remota, in questo modo si poteva sviluppare un frontend separato che richiedesse l'invio di un messaggio da remoto. Per motivi di tempo tale soluzione non è stata sviluppata.

6 Piattaforma software

Di seguito è riportata la piattaforma software utilizzata per lo sviluppo del codice.

- Sistema operativo Ubuntu 20.04 LTS
- Linguaggio programmazione Go
- Docker per la creazione di container
- Docker-compose per orchestrazione
- Github per versioning del progetto

I punti rossi rappresentano le piattaforme necessarie al funzionamento dell'applicazione.

È stato preso spunto dalla libreria riportata al seguente link per lo sviluppo del menù.

7 La guida all'istallazione

La guida all'istallazione è stata riportata nel readme file del progetto.