Multicasting totalmente e casualmente ordinato in Go

Progetto del corso Sistemi Distribuiti e Cloud Computing – A.A 2020/2021

Martina Salvati
mtr. 0292307
Università degli studi di Roma Tor Vergata
Roma, Lazio, Italia
salvatimartina97@gmail.com

Introduzione

Il Progetto ha come obiettivo quello di realizzare una applicazione distribuita che implementi gli algoritmi di multicast totalmente e casualmente ordinato.

In particolare *TOC* (multicasting totalmente ordinato centralizzato), *TOD* (multicasting totalmente ordinato distribuito) e *CO* (multicasting casualmente ordinato). In aggiunta è stato implementato l'algoritmo *B-multicast* in quanto viene utilizzato dagli altri algoritmi per alcune fasi secondarie (ad esempio, lo scambio di ACK).

Descrizione dell'architettura

L'applicazione è basata su una architettura multi-container orchestrate attraverso ***docker-compose***. Docker-compose permette la comunicazione tra molteplici container sullo stesso host, senza effettuare alcuna configurazione manuale di rete. Per l'applicazione, si è assunto che la membership sia statica, quindi il numero di membri facenti parte del gruppo multicast rimane invariata. I container ospitano servizi diversi :

a. Un container (registry) è dedicato al service registry per la registrazione dei processi al gruppo multicast.

b. Gli altri container ospitano l'applicazione RESTful e rappresentano i membri del gruppo multicast .

Building e Variabili d'ambiente

Attraverso le variabili d'ambiente settate nel docker-compose è possibile gestire i parametri principali dell'applicazione. Ad ogni membro del gruppo multicast è assegnato un id specifico tramite variabile d'ambiente, utilizzato nei vari algoritmi per la loro identificazione.

In particolare, è possibile effettuare la build dell'applicazione con il parametro (verbose=true) per consentire il logging.
Il progetto prevede di settare un ritardo massimo (delay sec) che ogni nodo effettua nella comunicazione End To End.
Inoltre, è possibile settare anche il numero di thread che popoleranno la pool di processamento dei messaggi.

Implementazione

L'applicazione è stata implementata utilizzando il linguaggio ad alto livello open-source ***Go***. Go ci permette di concentrarci sulla logica dell'applicazione o del sistema distribuito perché ci offre un buon supporto per la concorrenza e per le chiamate a procedura remota. Infatti, per i servizi dell'applicazione è stato utilizzato il supporto di ***Grpc***. Grpc offre un layer per l'astrazione delle chiamate a procedura remota e permette la comunicazione tra servizi. In dettaglio, per i servizi Grpc sono stati utilizzati i ***protocol buffers***, che permettono uno scambio efficiente e binario dei dati. E' stata aggiunta all'implementazione l'utilizzo di una pool di go-routines che permette al mittente di inviare più messaggi senza che esso si blocchi. La pool di go-routines effettua il processamento del messaggio da parte del client che ne richiede la trasmissione. La pool permette al client di inviare più messaggi senza dover attendere che siano ricevuti oppure che siano processati.

I servizi di base implementati attraverso Grpc sono due :

1. *Servizio base di comunicazione Client/Server per lo scambio di messaggi.*

   Il servizio viene utilizzato da tutti i membri del gruppo multicast. La procedura offerta dal servizio EndToEnd è la SendMessage. La SendMessage al suo interno implementa la logica per ritardare i messaggi di un delay randomico, utilizzato soprattutto per stressare i test degli algoritmi.

   In particolare il messaggio è visto come un pacchetto contenente *Header* e *Payload*.

   L'Header è una mappa di stringhe che svolge un ruolo fondamentale nell'applicazione:

   - Il campo **'type'** può assumere i valori : B,TOC,TOC2,TOD,CO,ACK e CloseGroup. 'Type' distingue la tipologia del gruppo multicast che viene utilizzata dal server per distinguere le azioni necessarie da effettuare in base al messaggio ricevuto.

- Il campo '**ProcessId**' assume l'identificativo del processo che richiede il processamento del messaggio,
- Il campo '**i**' assume il codice UID del messaggio inviato. Ogni messaggio ha il proprio codice univoco, utilizzato per essere distinto nelle varie fasi di comunicazione.
- Altri campi dell'header dipendono dalla tipologia di comunicazione.

2. *Servizio di Registry per la registrazione dei processi che partecipano al gruppo di comunicazione multicast.*
   Le procedure implementate dal servizio registry:
   - **Register** : procedura di registrazione dei processi ad un gruppo multicast
   - ***StartGroup*** : procedura che avvia le strutture necessarie per l'inizio della comunicazioni tra nodi
   - **Ready** : procedura che permette di rendere un nodo della rete pronto alla comunicazione
   - **CloseGroup** : procedura che permette la chiusura di un gruppo multicast
   - **GetStatus** : procedura che permette la lettura dello stato di un gruppo multicast

Entrambi i servizi sono stati implementati attraverso Protocol Buffers e Grpc.

API e OpenAPI

E' stato utilizzato il framework **Gin-Gonic** per effettuare la build dell'API REST-full. E' stato scelto Gin-Gonic in quanto il framework per le web applications scritte in Go più diffuso e utilizzato.

Framework	GitHub Repository	Stars	First Release
Gin	https://github.com/gin-gonic/gin	48.2k+	2014
Macaron	https://github.com/go-macaron/macaron	3.1k+	2016
Martini	https://github.com/go-martini/martini	11.2k+	2013
Echo	https://github.com/labstack/echo	18.9k+	2016
Mux	https://github.com/gorilla/mux	13.4k+	2016

Figure 1: Gin gonic , il framework più utilizzato dalla community di Go.

Descrizione dell'API

PATH	METHOD	SUMMARY
/deliver/{mId}	GET	Permette di recuperare i messaggi che sono stati processati e inviati a livello applicativo di un dato gruppo con id.
/deliver/	GET	Permette di recuperare tutti i messaggi che sono stati processati e inviati dall'host.
/groups	GET	Permette di ottenere le informazioni di tutti i gruppi esistenti.
/groups	POST	Permette la creazione di un nuovo gruppo multicast, specificando tipo e nome del gruppo. Se il gruppo è già esistente viene richiesto al registry di aggiornare le strutture dati del gruppo aggiunge le informazioni del membro.
/groups/{mId)	PUT	Permette l'avvio di un gruppo multicast.
/groups/{mId}	GET	Permette di recuperare le informazioni di uno specifico gruppo multicast.
/groups/{mId}	DELETE	Permette la chiusura di uno specifico gruppo.
/messaging/{mId}	GET	Permette di recuperare i messaggi scambiati dal gruppo.
/messaging/{mId}	POST	Permette di effettuare il multicasting di un messaggio al gruppo specificato

Per la descrizione dell'API implementata è stato utilizzato il linguaggio OpenAPI. OpenAPI, noto come Swagger, è DSL (linguaggio specifico del dominio) per descrivere le API REST.

E' possibile accedere allo swagger tramite il path specifico : *localhost:808x/multicast/v1/swagger/index.html.*

Per la creazione e avvio di un gruppo multicast bisogna eseguire tre fasi:

i. Ogni membro del gruppo deve effettuare una richiesta di creazione (o join) del gruppo. Questo permette al registry di aggiornare tutte le strutture del multicast group.
ii. Un membro del gruppo effettua l'avvio del gruppo multicast attraverso la PUT.
iii. Inizio del multicasting dei messaggi.

Algoritmi di multicasting

BMULTICAST

L'algoritmo Basic-Multicast può garantire che un processo effettuerà la deliver del messaggio, fin tanto che non avvenga un crash di comunicazione.

La send è una semplice chiamata al servizio EndToEnd che permette la comunicazione tra due EndPoint.
La B-delivery è una operazione che permette ad ogni processo di inserire in una slice *Delivery* i messaggi che devono essere consegnati a livello applicativo, specificando il proprio indirizzo e se il messaggio è consegnabile. L'algoritmo Bmulticast è alla base di tutti gli altri algoritmi, in effetti viene utilizzato dagli stessi per dei passaggi intermedi (come l'invio dell'ack nell'algoritmo tod).

Multicast totalmente ordinato

Il Multicast totalmente ordinato permette che i messaggi siano consegnati a livello applicativo nello stesso ordine ad ogni destinatario.

TOCMULTICAST – versione centralizzata

La versione centralizzata dell'algoritmo prevede l'implementazione di un sequencer, ovvero il membro che si occupa di assegnare il numero di sequenza ai messaggi ricevuti dai membri del gruppo.

Descrizione dell'algoritmo

Ogni processo invia il proprio messaggio di update al sequencer. Il sequencer assegna ad ogni messaggio di update un numero di sequenza univoco (*fase 1*) e poi invia in multicast il messaggio a tutti i processi (*fase 2*), che eseguono gli aggiornamenti in ordine in base al numero di sequenza. Il sequencer aggiorna il proprio clock ad ogni richiesta di multicast, mentre gli altri processi aggiornato i propri clock ad ogni ricezione di messaggio dal sequencer. Lo scambio dei valori di clock avviene tramite l'aggiornamento dei campi dell'header, in particolare il campo 's' (sequence).

Sequencer

La selezione del sequencer avviene tramite una pseudo-random function in cui il seed è pari alla lunghezza del gruppo multicast.
I messaggi scambiati durante la comunicazione vengono distinti attraverso i campi dell'header del messaggio. In particolare, il messaggio inviato al sequencer ha nell'header due valori specifici assegnati : seq=true e type=TOC. Mentre il messaggio inviato dal sequencer ai membri del gruppo ha segnato nell'header : order=true e type=TOC2 (fase due dell'algoritmo).

Implementazione della deliver

La deliver è implementata attraverso un thread (go-routine) che legge continuamente il contenuto della coda dei messaggi ricevuti. Il thread effettua la verifica del clock scalare. Se il valore del clock allegato al messaggio è pari al proprio clock scalare, allora il messaggio viene consegnato all'applicazione ed eliminato dalla coda.

Implementazioni alternative (TOC2)

E' stata implementata una variante dell'algoritmo. La variazione dell'algoritmo prevede che il messaggio venga inizialmente inviato a tutti i componenti del gruppo in una coda a bassa priorità, il componente eletto come sequencer in una seconda fase, invia il numero di sequenza da assegnare allo specifico messaggio. In una terza fase, tutti i processi spostano il messaggio ad una coda prioritaria assegnando il numero di sequenza ricevuto dal sequencer. Questa soluzione è chiaramente più distribuita ma più complessa da implementare per le diverse strutture da gestire.

TODMULTICAST

Descrizione algoritmo

(1)Un processo p_i invia in multicast (incluso se stesso) il **messaggio di update** msg_i
(2) Msg_i viene posto da ogni processo ricevente p_j in una coda locale $queue_j$, **ordinata in base al valore del timestamp.**
(2a)Il processo p_j invia in multicast un messaggio di **ack** della ricezione di msg_i
(3) Il processo p_j consegna msg_i all'applicazione se:

- msg_i è in testa a $queue_j$ e tutti gli ack relativi a msg_i sono stati ricevuti da p_j (*prima condizione*)
- per ogni processo p_k c'è un messaggio msg_k in $queue_j$ con timestamp maggiore di quello di msg_i (*seconda condizione*).

Implementazione dell'algoritmo

Per l'implementazione dell'algoritmo è stato necessaria la costruzione di due code locali : TODQueue e AckQueue. La prima coda contiene i messaggi ricevuti con il relativo numero di sequenza associato. La seconda coda contiene tutti gli ack ricevuti della ricezione del messaggio inviato. Attraverso queste due code è possibile controllare le condizioni necessarie per effettuare la consegna a livello applicativo.

Scalar Clock

Il clock scalare è associato ad ogni processo ed effettua le operazioni di **Tick** (incremento del clock), **Tock** (Recupero del valore di clock) e **Leap**(salto del clock ad un valore specifico). Queste operazioni permettono la gestione del clock scalare nelle varie operazioni del multicasting.

Implementazione fase ACK

Per l'invio dell'ack è stata utilizzata la B-Multicast, che garantisce i livelli minimi di ricezione dei messaggi. Un processo non effettua la consegna del messaggio a livello applicativo fin tanto che non riceve tutti gli ack relativi ad esso. Nella fase di invio degli ack, i processi inseriscono nell'header del messaggio il loro timestamp corrente. In questo modo gli altri processi vengono aggiornati riguardo al minimo timestamp contenuto in coda e quindi possono verificare la *seconda condizione* dell'algoritmo.

Implementazione della Deliver

La deliver è implementata attraverso un thread (go-routine) che legge continuamente il contenuto della coda dei messaggi ricevuti. Il thread effettua l'ordinamento dei messaggi in coda in base al loro timestampt ; i messaggi che hanno nell'header un timestampt con valore minore sono prioritari. Inoltre, viene effettuata una lettura della coda contenente gli ack per valutare se tutti gli ack relativi al messaggio sono stati ricevuti.

Costo dell'algoritmo

L'algoritmo prevede la presenza di n^2 *ack* all'interno dell'applicazione, questo infatti è problematico nei sistemi a larga scala e rende meno scalabile il sistema.

Multicast casualmente ordinato

Con il multicast casualmente ordinato si ha l'obiettivo di garantire l'ordine dei messaggi secondo una relazione causa-effetto. I messaggi concorrenti – non correlati da nessuna nozione di casualità – possono essere consegnati a livello applicativo in maniera diversa dai processi.

COMULTICAST

Descrizione dell'algoritmo

Un processo pi invia il messaggio m con timestamp t(m) basato sul clock logico vettoriale Vi.
Vj [i] conta il numero di messaggi da pi a pj.

Il processo pj riceve m da pi e ne ritarda la consegna al livello applicativo (ponendo m in una coda di attesa) finché non si verificano entrambe le condizioni :

1. t(m)[i] = Vj [i] + 1
   m è il messaggio successivo che pj si aspetta da pi,
2. t(m)[k] ≤ Vj [k] per ogni k≠i per ogni processo pk, pj ha visto almeno gli stessi messaggi visti da pi. Non accade che pj non abbia ancora ricevuto alcuni messaggi ricevuti dal processo iesimo.

Algorithm for group member p_i $(i = 1, 2\ldots, N)$

On initialization

$V_i^g[j] := 0\ (j = 1, 2\ldots, N)$;

To CO-multicast message m to group g

$V_i^g[i] := V_i^g[i] + 1$;

B-multicast$(g, <V_i^g, m>)$;

On B-deliver$(<V_j^g, m>)$ *from* p_j $(j \neq i)$, *with* $g = group(m)$

place $<V_j^g, m>$ in hold-back queue;

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ $(k \neq j)$;

CO-deliver m; // after removing it from the hold-back queue

$V_i^g[j] := V_i^g[j] + 1$;

Figure 2: CO-Multicast Algorithm.

Implementazione del *VectorClock*

Il clock vettoriale è alla base dell'implementazione dell'algoritmo casualmente ordinato. Il VectorClock viene inizializzato prima che avvenga ogni comunicazione ed in base al numero di nodi presenti nel gruppo multicast. Ogni elemento i-esimo del vector clock è un clock scalare e può effettuare operazioni di : ***LeapI(i,n)*** (salto del valore i-esimo del vettore di un valore n),***TickV(i)*** (incremento del valore i-esimo del vettore) e ***TockV(i)***(retrieve del valore i-esimo del vettore).

Implementazione della COQueue

E' stato necessaria l'implementazione di una coda locale FIFO che permettesse la memorizzazione dei messaggi ricevuti. In particolare, ogni messaggio inviato da un processo contiene i valori del suo VectorClock.
Questi valori sono campi dell'header del messaggio, utilizzati poi dai processi riceventi per ricostruire il vettore del processo inviante. Queste strutture, permettono l'individuazione delle condizioni necessarie per la consegna a livello applicativo del messaggio.

Implementazione della Deliver

La consegna a livello applicativo dei messaggi è implementata attraverso un thread (go-routine) che legge continuamente il contenuto della coda dei messaggi ricevuti. Il thread effettua la consegna solo se le due condizioni dell'algoritmo CO sono state soddisfatte (figura 5).

Testing in GO

Per la fase di testing è stato necessario l'utilizzo della libreria *"google.golang.org/grpc/test/bufconn"*.
Tramite questa libreria è stato possibile implementare un server mock che si comporta similmente al server gRPC originale implementato nel servizio EndToEnd.
La '*bufconn'* è una libreria ampiamente usata per testare gRPC questo in quanto permette di effettuare una connessione tramite un canale bufferizzato.

Per l'utilizzo viene implementato un particolare *Dialer* che setta la dial in modo tale da effettuare la comunicazione bufferizzata tra client e server.

Testing multicast totalmente ordinato

Per il testing degli algoritmi totalmente ordinati è stato necessario implementare una funzione che permettesse di verificare :

- Numero di messaggi ricevuto
- Ordine dei messaggi inviati a livello applicativo : viene verificata che la slice contenente i messaggi inviati a livello applicativo ('deliverati') sia uguale per tutti i processi, in ordine e contenuto.

Testing multicast casualmente ordinato

Per il testing degli algoritmi casualmente ordinati è stato necessario settare un ordine di casualità. Per la prima tipologia di test, ovvero verificare l'invio di messaggio da uno a molti, è stato sufficiente verificare che l'ordine dei messaggi deliverati sia uguale per tutti. E' stato assunto che i messaggi inviati da uno stesso processo siano in relazione di causa-effetto tra di loro. Per la tipologia di test con maggiore stress, dove tutti inviano a tutti contemporaneamente, è stato necessario settare le relazioni di casualità. L'algoritmo di test verifica l'ordine dei messaggi con relazione causa-effetto e ignora l'ordine dei messaggi concorrenti.

REFERENCES

[1] https://github.com/msalvati1997/b1multicasting
[2]https://hub.docker.com/repository/docker/martinasalvati/b1_multicasting
[3]G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. Distributed Systems: Concepts and Design.
Addison-Wesley, 5th edition, 2011.