Multicast ordinato in linguaggio Go

Benedetti Simone matr. 0295385
Università di Roma Tor Vergata
Ingegneria Informatica
Roma, Lazio, Italia
simone.benedetti8@gmail.com

***Abstract*— Descrizione e implementazione di algoritmi di multicast quali multicast totalmente ordinato con clock scalare (SCMulticast), multicast causalmente ordinato con clock vettoriale (VCMulticast) e totalmente ordinato centralizzato (SQMulticast) con relativa interfaccia applicativa.**

Keywords— SCMulticast, SQMulticast, VCMulticast, BMulticast

I. *Introduzione*

Il Progetto consiste nella realizzazione di una applicazione distribuita che implementi una serie di algoritmi di multicast quali multicast totalmente ordinato con clock scalare (SCMulticast), multicast causalmente ordinato con clock vettoriale (VCMulticast) e totalmente ordinato centralizzato (SQMulticast). Inoltre, per la realizzazione dei seguenti algoritmi è stato introdotto anche il multicast basilare (BMulticast).

II. Implementazione

A. *BMulticast*

Questo algoritmo può garantire che un processo effettuerà la deliver del messaggio fin tanto che non avvenga un crash nella comunicazione. La sendPacket, contenuta nel package endToEnd, è una chiamata a un servizio implementato attraverso gRPC che permette l'invio di un messaggio ad un altro nodo della rete. Questa funzione è usata per inviare in modo iterativo a tutti i nodi della rete lo stesso messaggio, fintanto che il numero di connessioni (quindi di nodi) non sia coperto. Inoltre, come forma di affidabilità ulteriore, contiene anche l'uso di un canale per l'invio di un ack al nodo mittente del messaggio per avere la certezza che la chiamata alla gRpc sia andata a buon fine.

B. *SCMulticast*

Algoritmo: Un processo p(i) invia in multicast, incluso a se stesso, il messaggio di update msg(i). Il messaggio viene posto da ogni processo ricevente p(j) in una cosa locale queue(j), ordinata in base al valore di timestamp. Il processo p(j) invia in multicast un messaggio di ack della ricezione di msg(i). 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 e se per ogni processo p(k) c'è un messaggio msg(k) in queue(j) con timestamp maggiore di quello di msg(i).

Scelte implementative: Per l'implementazione dell'algoritmo è stata necessaria la costruzione di una struttura nodo che, tra le variabili, è presente una coda che contiene i messaggi ricevuti con il timestamp associato, una coda contenente tutti gli ack ricevuti del messaggio inviato e infine una coda contenente i messaggi che andranno consegnati a livello applicativo.

Il clock scalare è associato alla struttura nodo descritta sopra e viene aggiornato ad ogni invio e ricezione dei messaggi nella rete.

Nella fase di invio degli Ack è stata usata implementazione del BMulticast, che garantisce i livelli minimi di ricezione dei messaggi. Nella fase di invio degli ack, i processi inseriscono nella struttura del messaggio il timestamp del primo elemento in coda ordinata dei messaggi processati che hanno in quel momento.

La funzione di ricezione dei messaggi e della deliver sono state implementate attraverso l'utilizzo di go Routine che legge continuamente il contenuto della coda dei messaggi. Queste routine effettuano l'ordinamento dei messaggi in coda in base al loro timestamp, cioè messaggi con timestamp minore hanno maggiore priorità.

C. *SQMulticast*

Questa versione è detta centralizzata e prevede l'implementazione di un sequencer, ovvero un membro che si occupa di assegnare il numero di sequenza ai messaggi ricevuti dai membri del gruppo.

Algoritmo: Ogni processo invia il proprio messaggio di update al sequencer. Esso assegna ad ogni messaggio un numero di sequenza univoco e poi invia in multicast il messaggio a tutti i processi, 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 aggiornano i propri clock ad ogni ricezione di messaggio dal sequencer.

Scelte implementative: Per l'implementazione si è scelto che il sequencer fosse un nodo scelto randomicamente all'interno dei nodi della rete. I messaggi scambiati durante la comunicazione vengono distinti attraverso una coppia chiave-valore inserita nei metadati inviati insieme al messaggio. Questo permetteva di vedere se il destinatario fosse un nodo o il sequencer e quindi di distinguere le operazioni da compiere per entrambi i casi.

La deliver è stata implementata attraverso l'uso di una go Routine che legge continuamente il contenuto di una coda dei messaggi ricevuti. Questa routine effettua la deliver a livello applicativo se il valore del clock allegato al messaggio è almeno pari al proprio clock scalare.

D. *VCMulticast*

Il multicast causalmente ordinato ha l'obiettivo di garantire l'ordine dei messaggi secondo una relazione causa-effetto. I messaggi concorrenti, cioè non aventi un rapporto di causa-effetto tra di loro, possono essere consegnati a livello applicativo in tempi differenti.

Algoritmo: un processo p(i) invia un messaggio m con timestamp t(m) basato su clock logico vettoriale V(i). Vj[i] conta il numero di messaggi da p(i) a p(j).

Il processo p(j) riceve m da p(i) e ne ritarda la consegna a livello applicativo finché il messaggio m è il messaggio successivo che p(j) si aspetta da p(i) e se per ogni processo p(k), p(j) ha visto almeno gli stessi messaggi visti da p(i). Non

accade che p(j) non abbia ancora ricevuto alcuni messaggi ricevuti dal processo i-esimo.

Scelte implementative: Alla base di questo algoritmo c'è implementazione del clock logico vettoriale. Esso è un vettore di lunghezza pari al numero di nodi e viene associato alla struttura nodo per tenerne traccia insieme, insieme alla coda dei messaggi ricevuti e alla coda di deliver dei messaggi.

La trasmissione del vettore avviene insieme al messaggio in una struttura apposita, di cui viene fatto il marshaling e i byte inviati attraverso l'uso di gRPC. Alla ricezione del messaggio viene effettuato l'unmashaling e ottenuto il vettore che era insieme al messaggio. Il marshaling e l'unmarshaling della struttura è stato implementato con l'uso della libreria JSON offerta da Go.

Anche in questo caso l'implementazione della deliver è attraverso una go Routine che legge continuamente il contenuto della coda dei messaggi ricevuti ed effettua la consegna quando entrambe le condizioni dell'algoritmo sono state effettuate.

III. TEST

Per la fase di testing è stata utilizzata la libreria "Testing" di Go che contiene una sottolibreria denominate "BufConn" che permette di creare una connessione fittizia tramite un canale bufferizzato tra nodi di testing. É servita a simulare la connessione al server gRPC originale implementato nel package EndToEnd.

A. *Testing SCMulticast*

Il test consiste in un invio di 3 messaggi one-to-many e many to many e viene controllato che ogni nodo riceva tutti e tre i messaggi (cioè che essi siano contenuti nella loro Deliver Queue) e che tutti i nodi abbiano la stessa sequenza di messaggi nella coda.

B. *Testing SQMulticast*

Il test consiste in un invio di 3 messaggi one-to-many e many to many e viene controllato che ogni nodo riceva tutti e tre i messaggi (cioè che essi siano contenuti nella loro Deliver Queue) e che tutti i nodi abbiano la stessa sequenza di messaggi nella coda.

C. *Testing VCMulticast*

Il test consiste in un invio di 3 messaggi one-to-many e many to many e viene controllato che ogni nodo riceva tutti e tre i messaggi (cioè che essi siano contenuti nella loro Deliver Queue) e che tutti i nodi abbiano la stessa sequenza di messaggi nella coda. In questo caso si controlla che valga il rapporto causa-effetto dei messaggi. Infatti il nodo 1 invia tre messaggi. Alla ricezione del primo messaggio, il nodo 2 invia un messaggio di risposta in multicast e il nodo 3 fa lo stesso. Al termine, viene verificato che la sequenza contenuta nella coda di Deliver rispetti questo rapporto di causa-effetto.

IV. APPLICATION LEVEL : REST API

L'applicazione è basata su un'architettura multi-container orchestrata con l'utilizzo di Docker e nello specifico del tool Docker compose. Esso permette la comunicazione tra diversi container. Per l'applicazione, si è assunto che la membership sia statica, quindi il numero scelto di nodi della rete è 3. C'è un quarto container che esercita le veci del service registry, che si occupa di registrare i nodi al gruppo multicast. I restanti nodi ospitano l'applicazione vera e propria.

A. *Fase di Building*

I vari parametri da settare attraverso l'uso delle variabili di ambiente sono i seguenti: delay (specificato in millisecondi), verbose (tipo booleano per consentire logging di tipo debug), numero di thread che popoleranno la pool del processamento dei messaggi.

B. *Implementazione*

L'applicazione è stata implementata utilizzando il linguaggio di alto livello open-source Go. Esso permette di concentrarci sulla logica dell'applicazione perché ci offre un buon supporto per la concorrenza e per le chiamate a procedura remota (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 buffer che permettono uno scambio efficiente dei dati. Inoltre è stata aggiunta l'implementazione di una pool di thread che permette al mittente di inviare più messaggi garantendo l'invio dei messaggi in modo FIFO.

I servizi implementati tramite gRPC sono i seguenti:

1) Servizio base di comunicazione per scambio messaggi:
Il servizio viene utilizzato da tutti I membri del Gruppo. La procedura offerta è la SendPacket che implementa l'invio del messaggio e la logica per ritardare I messaggi con un delay scelto in modo random. La struttura Packet contiene l'Header per l'invio di informazioni riguardanti il messaggio come il suo mittente e il Payload che contiene il contenuto del messaggio stesso.

2) Servizio di Registry per la registrazione dei processi che partecipano al gruppo di comunicazione:
Permette la registrazione statica dei membri del Gruppo. Le funzioni che implementa sono le seguenti:

a) Register: procedura di registrazione dei processi ad un gruppo multicast

b) StartGroup: procedura che permette l'inizializzazione delle strutture necessarie per l'inizio della comunicazione tra nodi.

c) CloseGroup: procedura che permette la chiusura di un gruppo multicast

d) GetStatus: procedura che permette la lettura dello stato di un gruppo multicast.

e) Ready: procedura che permette di rendere un nodo della rete in stato Pronto per la comunicazione.

C. *Descrizione API*

É stato utilizzato il framework Gin-Gonic per l'implementazione dell'api. Esso permette di fare la route dei vari path esposti dall'api ed è il framework per le applicazioni Web più diffuso e utilizzato in Go.

TABLE I. API – BASEPATH : MULTICAST/V1

Path	*Method*	*Description*
/groups	***POST***	*Permette la creazione di un nuovo Gruppo multicast, specificando il nome e il tipo. Se il Gruppo è già esistente viene richiesto al service registry di aggiornare le strutture dati del Gruppo aggiungendo le informazioni del membro.*
/groups/{mId}	***GET***	*Permette di recuperare le informazioni di uno specifico Gruppo multicast.*
/groups/{mId}	***DELETE***	*Permette la chiusura di un Gruppo specifico.*
/groups/{mId}	***PUT***	*Permette l'avvio del Gruppo di multicast.*
/messages/{mId}	***GET***	*Permette di recuperare I messaggi scambiati nel Gruppo multicast.*
/messages/{mId}	***POST***	*Permette di inviare un messaggio al Gruppo multicast.*
/deliverQueue/{mId}	***GET***	*Permette di recuperare tutti I messaggi che sono stati processati.*

Per la creazione e avvio di un gruppo multicast bisogna seguire le seguenti fasi: Ogni membro del gruppo deve effettuare una richiesta di aggiunta al gruppo, inserendo il tipo di multicast da effettuare. Successivamente un membro effettua la start del gruppo attraverso la medesima API e infine, con la POST, si può iniziare a inviare messaggi in multicast.

REFERENCES

[1] https://github.com/SimoBenny8/MulticastSDCCProject