Progetto B1: Multicast totalmente e causalmente ordinato in Go

Simone Tiberi
Corso di "Sistemi Distribuiti e Cloud Computing"
Università degli Studi di Roma Tor Vergata, Facoltà di Ingegneria Informatica
simone.tiberi.98@gmail.com

***Abstract*—L'obiettivo di questo documento è quello di analizzare e descrivere la metodologia adottata nelle varie fasi dello sviluppo del progetto, ponendo particolare enfasi sull'architettura realizzata e sulle scelte implementative effettuate.**

I. Introduzione

comuniGO è un'applicazione distribuita, sviluppata principalmente in GO, che permette ad un insieme di peer connessi ad un gruppo di multicast di comunicare utilizzando diversi algoritmi.

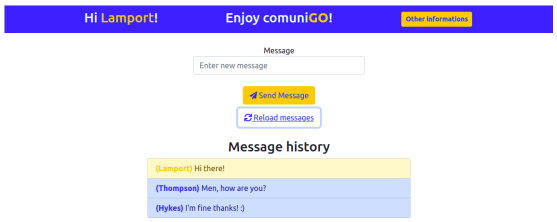

Fig. 1. Home page di comuniGO

L'applicazione:

- prevede un servizio di registrazione per i peer che vogliono partecipare al gruppo di multicast, assumendo che la membership sia statica durante l'esecuzione;
- prevede il supporto dei seguenti algoritmi:
  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.*

Inoltre, come richiesto dalla specifica, è stato testato il funzionamento degli algoritmi implementati nel caso in cui:

- vi sia un solo processo che invii il messaggio di multicast;
- molteplici processi contemporaneamente inviino un messaggio di multicast.

Per simulare condizioni di maggiore stress nel testing è stato incluso, come consigliato, un parametro *delay* configurabile, che permette di specificare un ritardo nell'invio generato in modo random in un intervallo predefinito.

Per facilitare il debugging dell'applicazione, come suggerito dalla traccia, è stato implementato un flag di tipo *verbose* che, se attivo, abilita la stampa di informazioni di logging con i dettagli dei messaggi inviati e ricevuti.

II. Tecnologie adottate

Per lo sviluppo del progetto si è fatto uso delle seguenti tecnologie:

- **Docker** per il supporto alla virtualizzazione;
- **Docker compose** per coordinare l'esecuzione dei container sul singolo nodo (e.g. startup, shutdown, interconnessione degli elementi);
- **Go** come unico linguaggio per lo sviluppo della logica applicativa;
- **gRPC** come framework RPC per lo scambio dei messaggi fra i componenti dell'applicazione;
- **Redis** come datastore in memory per la memorizzazione dei messaggi *consegnati a livello applicativo* dai vari algoritmi;
- **HTML**, **Javascript** (**jQuery**) e **CSS** per la realizzazione del frontend Web con cui interagire con l'applicazione.

III. Descrizione dell'architettura

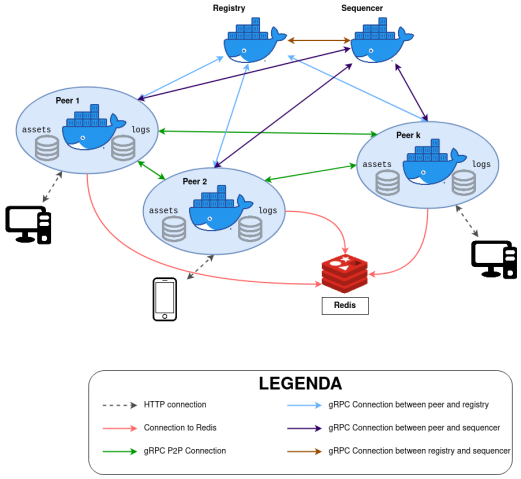

Fig. 2. Architettura realizzata.

In figura 2 è riportata una schematizzazione dell'architettura adottata per lo sviluppo dell'applicazione, nella quale si può notare che:

- peer, servizio di registrazione e sequencer siano ciascuno incapsulato all'interno di un container differente;

- a ciascun peer sono associati due volumi, uno per permettere che l'attività di logging sia persistente e l'altro per accedere agli assets web
- ogni peer è connesso ad un'istanza unica di Redis al fine di consegnare e prelevare i messaggi;
- la logica applicativa è interamente realizzata sfruttando il meccanismo RPC tra qualunque coppia di endpoint;
- l'utente finale può scambiare i propri messaggi con altri connessi al gruppo di multicast semplicemente utilizzando il proprio browser (o in alternativa sfruttando programmi come cURL).

IV. Scelte progettuali

Di seguito è riportata una descrizione delle scelte effettuate nello sviluppo della soluzione proposta.

A. Uso di un frontend web per l'interazione con i peer

Per permettere l'interazione con i peer è stato scelto di realizzare un webserver implementato in Go sfruttando la libreria `labstack/echo`.

Le route servite sono di seguito analizzate brevemente:

- `[GET] /`: permette di accedere al portale di comunicazione effettivo, previa registrazione;
- `[GET] /list`: permette di ottenere la lista dei messaggi ricevuti in formato json;
- `[GET] /info`: permette di effettuare il retrieve delle informazioni necessarie a popolare l'homepage del sito;
- `[POST] /send`: permette di inviare un messaggio, specificando come parametri:
  - `message`: il corpo del messaggio,
  - `delay`: l'intervallo entro cui estrarre il ritardo con cui inviare il messaggio nel formato `start:end` **da usare solo in fase di test**;
- `[POST] /sign`: permette di inviare il proprio username al servizio di registrazione, specificandolo come parametro (`username`).

I file `*.html` e `*.js` sono stati inseriti all'interno del filesystem del container come volume *readonly* (`assets`), poiché non soggetti ad alcun processo di compilazione. Ciò permette di:

- diminuire il footprint delle immagini Docker prodotte;
- evitare di avviare la build dell'immagine del peer a seguito del cambiamento di alcuni elementi puramente grafici dell'applicazione.

B. Uso di volumi per l'attività di logging

Per quanto riguarda l'attività di logging, svolta dai vari componenti dell'applicazione, è stato scelto di renderla persistente così da non dover avere il vincolo di lanciare i vari container in modalità *attached*, tramite l'utilizzo di un volume docker dedicato.

La tipologia di volume adottato è *bind* per cui l'engine di docker non fa altro che montare la directory del filesystem locale `$COMUNIGO_PATH/logs` su `/logs` all'interno del container.

L'attività di log svolta automaticamente dal webserver è stata riversata su un file differente da quello dedicato alla logica applicativa per:

- evitare di dover sincronizzare accessi concorrenti ad un unico file di log;
- permettere facilmente di *scartare* una parte di informazioni prodotte se non di interesse (e.g. in fase di debugging della logica applicativa la presenza di entry scritte dal webserver relative alle richieste HTTP potrebbe rendere difficile la ricerca delle informazioni a cui si è interessati)

Il formato configurato per ciascuna riga del file di log autogenerato da `echo` è il seguente:

```
[<time>]: method=<method>, uri=<uri>, status=<status>
```

C. Uso di redis per la memorizzazione dei messaggi

Poiché si utilizza un frontend web per interagire con l'applicazione, è possibile che l'utenti aggiorni la pagina o semplicemente chiuda la tab del browser per poi riaprirla in un secondo momento.

In questo scenario per *consegnare i messaggi al livello applicativo* non è sufficiente unicamente restituirli (e.g. tramite una stampa su terminale) perché altrimenti non si avrebbe la possibilità di visionarli nuovamente in un secondo momento.

È necessario dunque memorizzarli e, al fine di rendere quanto più possibile i container *stateless*, è stato scelto di utilizzare un datastore separato.

La scelta è ricaduta su Redis per diversi motivi:

- si integra in maniera ottimale con Go, infatti l'API mappa praticamente 1:1 con gli effettivi comandi nativi di Redis;
- essendo la membership statica, non è previsto che i peer vengano spenti ed in seguito riaccesi per partecipare sempre allo stesso gruppo, per cui la natura *in-memory* di Redis non rappresenta un'ostacolo;
- le funzionalità `RPUSH` e `LRANGE` nativamente permettono di coprire l'intero meccanismo di salvataggio e recupero dei messaggi

D. Altre scelte

- Nella stesura del codice è stato adottato un approccio object oriented, seppure Go non è un linguaggio puramente OO come ad esempio lo è Java.
  Tuttavia questo approccio ha l'obiettivo di rendere la stesura del codice modulare, facilmente manutenibile ed estendibile.
- Per l'invio dei messaggi è stato scelto di adottare connessioni TCP persistenti così da avere:
  - un overhead inferiore in termini di messaggi di handshaking inviati;
  - la garanzia di consegna FIFO ordered dei messaggi.
- Nel codice si è fatto significativamente uso di:
  - *goroutines* per parallelizzare l'esecuzione guadagnando in termini prestazionali;
  - *canali* al fine di:

    - permettere lo scambio di informazioni fra le varie goroutines;
    - sincronizzare le goroutines limitando l'uso esplicito di lock.
- Per realizzare una chiusura *pulita* dell'applicazione è stato utilizzato il meccanismo nativo di Go dei *contesti con cancellazione* in combinazione con la cattura dei segnali di `SIGINT` e `SIGTERM` che, stando alla documentazione Docker, vengono invati da docker-compose nel momento in cui si richiede lo stop dei container.
- Per minimizzare il footprint delle immagini Docker si è fatto uso delle *multi-stage build* che permettono di:
    1) compilare gli applicativi occupando eventualmente *tanta* memoria per ospitare i sorgenti necessari all'interno di un container *worker*;
    2) copiare dal *worker* unicamente l'eseguibile prodotto su un nuovo container;
    3) restituire al richiedente l'immagine a footprint ridotto e contestualmente permettere l'eliminazione del worker.

V. Descrizione dell'implementazione

A. Registrazione

Lato frontend, la registrazione prevede che:

1) l'utente si colleghi ad uno dei peer del gruppo, i quali possono essere individuati tramite lo script di discovery oppure un programma Go apposito, descritti all'interno del README file;
2) inserisca un nome utente unico all'interno del gruppo;
3) attenda che tutti quanti i membri siano realmente connessi prima di entrare effettivamente nel portale di comunicazione.

Lato backend:

1) il peer inoltra al servizio di registrazione l'username ricevuto dal frontend;
2) le procedure remote server-side inviano quanto ricevuto dai peer ad un'apposita goroutine che mantiene la lista non ancora completa degli utenti connessi;
3) nel momento in cui si raggiunge la dimensione del gruppo prevista, la routine in questione *sblocca* le procedure in attesa di poter inviare la composizione del gruppo al peer richiedente
4) una volta ricevuto l'elenco dei peer connessi, esso viene inoltrato via HTTP al client web affichè:
    - si effettui il redirect verso il reale portale per la comunicazione;
    - si mostri un messaggio d'errore, ove necessario.

B. Comunicazione tramite sequencer

Nel caso in cui, la modalità di comunicazione selezionata è quella basata sull'utilizzo di un sequencer, il nodo di registrazione comunica la composizione del gruppo di multicast anche a quest'ultimo prima di terminare la sua esecuzione.

Una volta completata la fase di registrazione, la comunicazione tramite sequencer segue l'iter di seguito riportato:

- Nella comunicazione *dal peer al sequencer*
    1) il peer inoltra al sequencer il messaggio ricevuto dal frontend;
    2) le procedure remote server-side inviano quanto ricevuto dai peer ad un'apposita goroutine, responsabile di ordinare in modo univoco i messaggi ricevuti e marcarli con un apposito timestamp (*sequence number*).
- Nella comunicazione *dal sequencer al peer*, ogni qual volta la goroutine dedicata all'ordinamento marca un nuovo messaggio con un determinato timestamp, questo viene inoltrato sulle varie connessioni aperte in precedenza verso tutti i peer.

C. Comunicazione basata su clock logici scalari

Nel caso in cui la modalità di comunicazione selezionata è quella basata sull'utilizzo del clock logico scalare, una volta completata la fase di registrazione, la comunicazione si basa sull'approccio di seguito descritto.

- Per l'*invio degli update*, una goroutine ad hoc:
    1) preleva quanto inserito dall'utente tramite il frontend come corpo del messaggio;
    2) appone al messaggio il proprio timestamp incrementato di una unità;
    3) inoltra il messaggio sulle varie connessioni in precedenza aperte verso i restanti partecipanti al gruppo di multicast.
- Alla *ricezione degli update*, il peer:
    1) aggiorna il proprio clock, ponendolo pari a:
    $$L = \max\{L, t\} \tag{1}$$
    dove L è il clock logico scalare locale del peer e t quello contenuto nel messaggio ricevuto;
    2) riscontra la ricezione inviando un ack a tutti i partecipanti del gruppo;
    3) inserisce quanto ricevuto all'interno della coda dei messaggi pendenti, in modo ordinato rispetto al clock logico scalare e l'identificativo del membro (`username`);
    4) consegna i messaggi per cui valgono le seguenti condizioni:
        - sono i primi nella coda totalmente ordinata;
        - sono stati riscontrati da tutti gli altri membri;
        - per ogni altro membro sono presenti messaggi in coda.
- Alla *ricezione degli ack*:
    1) si incrementa il contatore degli ack associato al messaggio riscontrato;
    2) si consegnano i messaggi per cui vale la condizione sopra proposta.

Essendo questo algoritmo indubbiamente il più elaborato fra i tre, di seguito è riportata un analisi sintetica delle strutture dati utilizzate per renderlo quanto più possibile efficiente.

- È stato utilizzato uno slice nativo di Go per la memorizzazione dei messaggi pendenti, il che implica un costo

$O(n)$ ad ogni nuovo inserimento, per via della necessità di mantenere l'ordinamento.

- Viene utilizzata una mappa stringa $\rightarrow$ intero per contare gli ack ricevuti per ciascun messaggio pendente affinché il costo dell'incremento sia $O(1)$.
  È opportuno osservare che, mantenere il contatore degli ack come metadato presente all'interno di ogni entry della lista, avrebbe causato un costo per l'aggioramento, nel caso peggiore, pari a $O(n)$ per via della ricerca sequenziale, dove n è pari alla taglia della coda.
- Viene utilizzata una mappa stringa $\rightarrow$ intero per tener traccia del numero di messaggi in coda presenti per ciascun peer connesso al gruppo di multicast.
  Questa soluzione permette di realizzare la condizione *"per ogni altro membro sono presenti messaggi in coda"* con un costo $O(k)$, dove k è pari al numero di peer connessi ($k-1$ accessi alla mappa ciascuno dei quali a costo $O(1)$).
  È opportuno osservare che, nel caso in cui non fosse stata utilizzata questa struttura dati, il costo della verifica sarebbe stato pari a $O(n)$, dove n è pari al numero di messaggi in coda.
  Ovviamente in scenari reali d'utilizzo $k << n$.

D. Comunicazione basata su clock logici vettoriali

Nel caso in cui la modalità di comunicazione selezionata è quella basata sull'utilizzo del clock logico vettoriale, una volta completata la fase di registrazione, la comunicazione si basa sull'approccio di seguito descritto.

- Per l'*invio dei messaggi* da parte del peer i-esimo, una goroutine ad hoc:
  - preleva quanto inserito dall'utente come corpo del messaggio;
  - appone ad esso il proprio timestamp dopo averlo modificato come segue:
    $$V_i[i] = V_i[i] + 1 \tag{2}$$
  - inoltra il messaggio sulle varie connessioni in precedenza aperte verso i restanti partecipanti al gruppo di multicast;
  - consegna direttamente il messaggio poiché certamente rispetta la causalità.
- Alla *ricezione di un messaggio* proveniente dal peer i-esimo da parte del peer j-esimo :
  1) si inserisce il messaggio all'interno della coda dei messaggi pendenti;
  2) si consegnano tutti i messaggi per cui valgono le seguenti condizioni:
     - $t[i] = V_j[i] + 1$;
     - $t[k] \leq V_j[k]$ per ogni $k \neq i$.

VI. Testing & debugging

Per lo sviluppo dei casi di test si è fatto uso del supporto *built-in* offerto da Go, ovvero `go test`.

Nel caso di *invio singolo adottando multicast totalmente ordinato* è stato simulato l'evento di send per ciascun processo connesso, attendendo tra un invio ed il successivo un tempo pari al massimo delay sperimentabile, così da avere la garanzia che tutti i peer ricevano la medesima sequenza ordinata di messaggi.

Nel caso di *invio multiplo adottando multicast totalmente ordinato* è stato realizzato uno scenario analogo, ove anziché inviare l'uno dopo l'altro, i processi mandano i propri messaggi in modo simultaneo mediante l'ausilio di goroutines.

Una volta effettuato l'invio, in entrambi i casi i processi effettuano il retrieve della lista dei messaggi consegnati per poi:

- nel caso di implmentazione tramite *sequencer* verificare che le liste ricevute siano tutte uguali;
- nel caso di implementazione tramite *clock logico scalare* verificare che la parte comune a tutte le liste sia uguale. Questo perché, per via del meccanismo di consegna è possibile, ad esempio, si verifichi che:
  - il processo A riceva k messaggi;
  - il processo B riceva h messaggi;
  - il processo C riceva l messaggi

  con $l \leq h \leq k$. In questo scenario il test verifica che i primi l messaggi di ciascuna lista siano equivalenti.

Per il testing dell'algoritmo basato sull'utilizzo dei clock logici vettoriali è stato scelto di verificare l'effettivo rispetto della causalità nel modo seguente:

1) ciascun peer invia un messaggio avente come body il proprio username;
2) attende un certo Δt;
3) effettua il retrieve dei messaggi ricevuti;
4) costruisce un messaggio *riassuntivo* contenente l'elenco dei body ricevuti separati da `":"` (e.g `"A:B"`);
5) invia il messaggio riassuntivo,
6) si verifica, per ciascun peer, che all'interno della lista dei messaggi quelli *riassuntivi* siano successivi a quelli che riscontrano.
   Ad esempio se il messaggio riassuntivo è `"A:B"` si verifica che i messaggi `"A"` e `"B"` lo precedano in ogni lista.

Ovviamente ciò che differenzia il caso di test di invio singolo da quello multiplo, è la modalità con cui ciascun peer manda i messaggi:

- nel caso *singolo* se ne invia uno alla volta;
- nel caso *multiplo* si fa uso di goroutines per realizzare il parallelismo.

Per quanto riguarda il debugging, poiché l'attività di logging è stata resa persistente e quindi è possibile ispezionare nel dettaglio come agiscono i peer, è stato scelto di implementare la logica collegata al flag `verbose` semplicemente mostrando su console javascript il dettaglio dei messaggi ricevuti, comprensivo di timestamp associato ad essi, come mostrato in figura 3

VII. Modifiche necessarie per il deployment su cluster reali

Nel momento in cui si volesse deployare questa applicazione distribuita su un cluster di nodi (e.g. mediante l'ausilio di

```
*** VERBOSE ENABLED ***
Current peer...................: Lamport
Current type of service offered: VECTORIAL
List of other peers connected..:
        Hykes@172.21.0.6
        Thompson@172.21.0.5
New message list retrieved from server:
        [ID: [0,1,0], From: Lamport, Body: Hi there!]
        [ID: [0,1,1], From: Thompson, Body: Hi guys]
        [ID: [1,1,1], From: Hykes, Body: How are you?]
```

Fig. 3. Implementazione del flag `verbose`

Kubernetes o di istanze EC2) ovviamente non sarebbe idoneo utilizzare un'unica istanza attiva di Redis, poiché si perderebbe il vantaggio della decentralizzazione degli algoritmi tornando ad avere un SPOF.

Lo scenario ideale di deployment sarebbe, dunque, quello in cui su ciascun nodo fosse presente un'istanza di peer e una di Redis, avendo così un'architettura come quella riportata in figura 4, che rispetta inoltre il pattern *database-per-service*.

Avere la configurazione dei peer non hardcoded nel sorgente, permette di realizzare il porting a quest'architettura più realistica, in modo semplice senza che sia necessario cambiare massivamente il codice.

Per modificare la configurazione, in definitiva è possibile:

- editare il file `comunigo.cfg`, all'interno della directory di `build`;
- utilizzare i parametri passati da linea di comando allo script di startup;

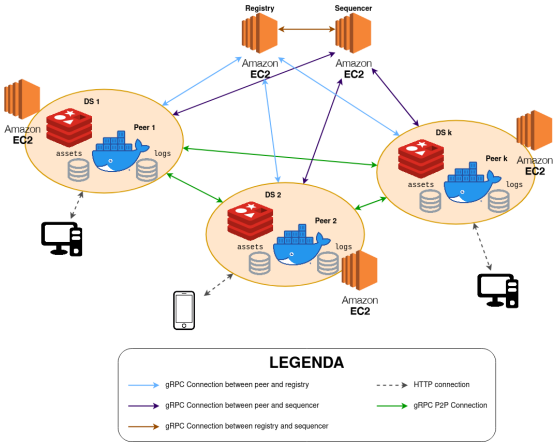

Fig. 4. Possibile architettura su deployment reale