SDCC Progetto B1: Multicast totalmente e causalmente ordinato in Go

ALESSANDRO CHILLOTTI

Additional Key Words and Phrases: Go, Docker, Multicast, Peer

ACM Reference format:
Alessandro Chillotti. 2021. SDCC Progetto B1: Multicast totalmente e causalmente ordinato in Go. *ACM Trans. Graph.* 37, 4, Article 1 (October 2021), 4 pages.
https://doi.org/10.1145/1122445.1122456

1 TRACCIA DEL PROGETTO

Lo scopo del progetto è 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 è 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 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 ASSUNZIONI

Le assunzioni che sono state fatte per realizzare questo applicativo sono:

- La comunicazione è affidabile, ovvero non è possibile che ci sia la presenza di messaggi persi;
- La comunicazione è *FIFO* ordered, ovvero se un processo p_i invia molteplici messaggi al processo p_j, quest'ultimo riceve i messaggi nello stesso identico ordine in cui p_i li ha inviati;
- È stato assunto un ritardo massimo nell'invio del messaggio pari a 3 secondi perché ritenuto realistico ed adatto allo scopo in questione. Questo ritardo non è altro che uno `sleep` del processo mittente, quindi viene generato un numero pseudo-randomico fra 0 e 3 al momento dell'inoltro del messaggio.

3 SCELTE PROGETTUALI

In questa sezione sono presentate e motivate le scelte progettuali effettuate in fase di sviluppo dell'applicativo.

3.1 Consegna di un messaggio di multicast

Tipicamente gli algoritmi di multicast vengono eseguiti al livello del middleware e quindi un importante aspetto da considerare nell'implementazione degli algoritmi è sicuramente la consegna a livello applicativo. Infatti, si ha una sostanziale differenza fra i termini *ricezione* di un messaggio e *consegna* di un messaggio. In particolare:

- La *ricezione* riguarda la fase in cui il messaggio arriva al nodo desiderato.
- La *consegna* riguarda la fase in cui il messaggio viene consegnato all'applicazione al livello sovrastante.

Poiché gli algoritmi sono già eseguiti a livello applicativo, è stato scelto di simulare la consegna di un messaggio attraverso il salvataggio su un file.

3.1.1 Struttura del file di consegna. Il file di consegna è visto come una tabella con i seguenti campi:

- *Id*: questo è un campo speciale perché rappresenta l'oggetto chiave con cui l'algoritmo lavora per mantenere l'ordinamento desiderato. In particolare, varia in base all'algoritmo selezionato:
  - Algoritmo 1: questo campo corrisponde all'identificativo assegnato dal *sequencer*.
  - Algoritmo 2: questo campo corrisponde al valore del clock logico scalare del peer mittente, nel momento in cui invia il messaggio.
  - Algoritmo 3: questo campo corrisponde al valore del clock logico vettoriale del peer mittente, nel momento in cui invia il messaggio.
- *Timestamp*: questo campo rappresenta l'istante temporale in cui quel messaggio è stato inviato dal peer mittente.
- *Username*: questo campo rappresenta l'username con cui un peer si identifica[1].
- *Messaggio*: questo campo rappresenta il contenuto del pacchetto effettivo.

Con lo scopo di facilitare la comprensione della struttura del file di log, viene mostrato un esempio che rappresenta lo snapshot, ad uno specifico istante temporale, del file di log nel caso in cui si adottando l'algoritmo 1.

[1] L'username è specificato in fase di registrazione.

Table 1. Esempio del file di consegna (Algoritmo 1)

Id	*Timestamp*	*Username*	*Messaggio*
1	12:59:32	peer_1	messaggio 1
2	13:02:17	peer_2	messaggio 2
3	13:05:27	peer_3	messaggio 3
4	13:08:17	peer_2	messaggio 4

3.2 Architetture

Per lo sviluppo degli algoritmi si è scelto di adottare due architetture differenti, in modo tale da poter scegliere di averne una più adatta in base all'algoritmo richiesto.

3.2.1 Scelte generali. Come descritto in precedenza, la consegna al livello applicativo è stata simulata con il salvataggio del contenuto del messaggio, più relativi metadati, in un file.

Il file risiede all'interno di volume *Docker* per ogni peer appartenente al gruppo multicast.

3.2.2 Architettura per l'algoritmo 1. L'architettura ideata per la realizzazione dell'algoritmo 1 è la seguente[2].

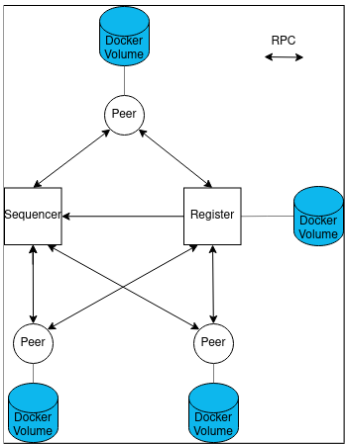

Fig. 1. Architettura algoritmo 1

Si può notare dalla figura che ad ogni peer è stato montato un volume Docker, nel quale è presente il file utilizzato per simulare la consegna del messaggio a livello applicativo.

Gli attori presenti nell'architettura sono:

- Il *peer* rappresenta un partecipante al gruppo di multicast. Ovviamente, per poter partecipare alla comunicazione si ha la necessità di eseguire una registrazione ad esso.
- Il *register* rappresenta il nodo che permette di:
  - Accettare le registrazione dei peer finché non si raggiunge il numero di partecipanti stabilito al gruppo multicast.
  - Inviare la lista dei peer partecipanti al gruppo ad ogni peer registrato.
- Il *sequencer* è il nodo che implementa l'algoritmo, assegnando un identificativo ad ogni pacchetto che permette di avere un ordinamento totale.

3.2.3 Architettura per gli algoritmi 2 e 3. L'architettura ideata per la realizzazione degli algoritmi 2 e 3 è la seguente[2].

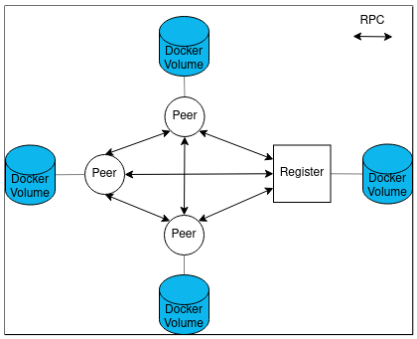

Fig. 2. Architettura algoritmi 2 e 3

Si può notare, come rispetto all'architettura precedente, non sia presente il *sequencer* poiché gli algoritmi 2 e 3 sono algoritmi realizzati in modo distribuito.

4 DETTAGLI IMPLEMENTATIVI GENERICI

In questa sezione saranno descritti i dettagli implementativi riguardante la realizzazione degli algoritmi.

4.1 Orchestrazione dei container

Per l'orchestrazione dei container è stato utilizzato *Docker Compose*. In particolare, sono state fatte le seguenti scelte:

- È stata creata una rete virtuale per effettuare la comunicazione fra i diversi container up and running.
- È stato creato un profilo `sequencer` che permette di adattare l'istanziazione dell'infrastruttura in base alla scelta dell'algoritmo. In particolare, se viene scelto l'algoritmo numero 1, il `sequencer` verrà istanziato, altrimenti esso non farà parte della struttura dell'applicazione.

  Questo permette di avere una flessibilità dell'infrastruttura e di non sprecare risorse in caso in cui si utilizzino algoritmi diversi dall'1.
- Sono state create delle variabili d'ambiente che permettono di avere dei parametri configurabili. In particolare, le variabili d'ambiente sono utilizzate per la scelta dell'algoritmo e per il numero di nodi partecipanti alla comunicazione multicast.

4.2 Istanziazione dei container

Per la gestione dell'istanziazione dei container è stato utilizzato *Docker* e si è l'immagine di base selezionata è `golang:1.16-alpine` poiché ha una footprint minore rispetto, ad esempio, Ubuntu e conteniamo tutto ciò che è necessario per l'applicazione corrente.

[2]In figura è mostrata un'architettura con solamente 3 peer, ma questo è stato fatto solamente per semplificare l'architettura. Quindi, è possibile scegliere quale numero di peer far partecipare al gruppo multicast in fase di startup dell'applicazione.

Vengono sfruttate le variabili d'ambiente specificate nella sottosezione precedente per essere importate all'interno dell'applicazione *Go*.

4.3 Comunicazione fra i container

I nodi, descritti nella sottosezione 3.2, comunicano fra loro attraverso il meccanismo di *Remote Procedure Call*. Per l'implementazione di RPC non è stato utilizzato `gRPC`, ma è stato usato il package `"net/rpc"` poiché l'applicativo è totalmente scritto in *Go* e quindi non si necessitava di una versatilità per quanto riguarda il linguaggio di programmazione.

4.3.1 *Struttura di un pacchetto.* RICORDA STRUTTURA PACKET

4.4 Struttura di un peer

Il peer è la figura più complessa che è stata gestita per la realizzazione di questo applicativo poiché, in base all'algoritmo da eseguire, esso ha dei compiti differenti.

È stata implementata una classe `Peer` che astrae e cattura le caratteristiche e le funzioni di base di un semplice peer, in modo tale da specializzare questi aspetti in base allo scenario in cui essi si trovano.

Le caratteristiche di base di un peer individuate sono

- L'indice del processo[3];
- L'indirizzo IP;
- Il numero di porta;
- L'username.

Le funzioni base di un peer sono:

- La registrazione all'interno del gruppo di multicast;
- La ricezione della lista dei nodi registrati e la ricezione, da parte del frontend, del messaggio da inoltrare nella rete.

Per gli altri "tipi" di peer sono state realizzate altre classi (i.e. strutture in *Go*) che vanno ad estendere la classe `Peer` appena descritte (i.e. la struttura `Peer` viene inglobata nelle nuove strutture). Ogni classe avrà i propri metodi, in modo tale che essi si adattino il miglior modo possibile all'algoritmo da realizzare.

Questo approccio ha permesso di avere un codice modulare che si adatta in base all'algoritmo richiesto poiché, nella fase di startup, è presente uno `switch` che ha il compito di individuare l'algoritmo da eseguire e di creare le corrette istanze dei peer.

5 ALGORITMO 1

In fase di invio di un messaggio, il peer mittente prepara il `Packet` ed invia il messaggio al nodo *Sequencer*. A sua volta, il *Sequencer* targa il messaggio con un Id. Quindi, esso incapsula il `Packet` ricevuto all'interno di una struttura ad-hoc per mantenere il metadato relativo all'Id associato dal *Sequencer*.

L'idea addottata, per quanto riguarda un nodo ricevente, è la seguente:

(1) Alla ricezione, il messaggio viene posto all'interno di un buffer (`channel`[4]).
(2) In fase di consegna, viene scannerizzato ciclicamente il buffer finché non è presente il messaggio da inviare al livello applicativo.

6 ALGORITMO 2

In questo algoritmo, il peer invia un messaggio di update per comunicare con gli altri peer. Quindi, è stata creata una struttura `Update` che associa ad ogni `Packet` il relativo `timestamp` (i.e. `int`).

L'idea adottata per la ricezione e la consegna è similare a quanto descritto per l'algoritmo precedente.

6.1 Lista ordinati di messaggi di update

Uno dei punti chiavi dell'implementazione dell'algoritmo 2 è sicuramente la realizzazione di una lista d'attesa dei messaggi di update ordinati in base al timestamp.

Per far fronte a questo aspetto, si è scelto di realizzare una lista collegata. Il punto chiave di questa implementazione è che l'inserimento in questa lista avviene in maniera ordinata, ovvero si scandisce la lista finché non si trova un nodo con timestamp maggiore del nodo candidato ad entrare e, dopo averlo trovato, si effettua l'inserimento nella posizione corretta.

È stato scelto questo approccio in modo tale da evitare di inserire randomicamente all'interno della lista un nodo e, a valle dell'inserimento, ordinare i componenti.

6.1.1 *Struttura di un nodo della lista.* Il nodo appartenente alla lista è la seguente struttura.

```
type Node struct {
    Update Update
    Next   *Node
    Ack    int
}
```

Si può notare come l'`Ack` sia stato inserito come metadato del nodo, in modo tale da facilitare la gestione e l'implementazione della consegna dei pacchetti.

7 ALGORITMO 3

In questo algoritmo si è nuovamente fatto uso di un buffer per la memorizzazione dei pacchetti ricevuti ma non ancora consegnati, in quanto non si necessitava più di avere un ordinamento dei messaggi.

È stata creata la classe `Vector Clock` per astrarre il concetto di clock vettoriale e definire alcuni metodi base per la sua gestione. Quindi, ad ogni invio di un messaggio da parte di un peer, il messaggio di `Update` viene marcato con uno snapshot in quell'istante di tempo del clock logico vettoriale relativo al peer mittente.

L'idea adottata per la ricezione e la consegna è similare a quanto descritto per gli algoritmi precedenti.

[3]Gli algoritmi si basano anche sull'indice numerico del processo nel momento in cui si verificano situazioni particolari, quindi questa è stata pensata come una caratteristica chiave di un peer.

[4]Questo canale è costituito da `100` componenti poiché sembra essere, considerando il seguente scenario, un numero infinitamente grande rispetto ai pacchetti che si possono bufferizzare.

8 PIATTAFORMA SOFTWARE

La piattaforma software utilizzata per la realizzazione dell'applicativo è la seguente:

- Il sistema operativo è Ubuntu 20.04.3 LTS;
- Il linguaggio di programmazione utilizzato è *Go*;
- Per l'istanziazione di ogni nodo è stato utilizzato *Docker*;
- Per l'orchestrazione dei container è stato utilizzato *Docker Compose*.

Per eseguire l'applicativo si ha bisogno solamente di aver istallato, all'interno della propria piattaforma software, *Docker* perché permetterà di istanziare l'applicazione.

9 TESTING DELL'APPLICAZIONE

Per verificare il corretto comportamento degli algoritmi implementati sono stati ideati dei test. In particolare, per ogni algoritmo:

- Un test riguarda l'invio del messaggio di multicast da parte di un solo peer.
- Un test riguarda l'invio del messaggio di multicast, anche in modo concorrente, da parte di più peer.

9.1 Test per l'algoritmo 1

9.1.1 Un solo sender. In questa tipologia di test un peer invia sei messaggi, uno dopo l'altro in modo tale da rispettare l'assunzione *FIFO* ordered.

Il risultato aspettato è che ogni peer consegni, nello stesso identico ordine, i messaggi ricevuti al livello applicativo.

9.1.2 Più sender. In questa tipologia di test ogni peer, in modo concorrente, invia un messaggio al *sequencer* e dopo aver inviato il primo messaggio, inoltra anche un secondo messaggio.

Il risultato aspettato è che ogni peer consegni, nello stesso identico ordine, i messaggi ricevuti al livello applicativo.

9.2 Test per l'algoritmo 2

9.2.1 Un solo sender. In questa tipologia di test un peer invia sei messaggi, uno dopo l'altro in modo tale da rispettare l'assunzione *FIFO* ordered.

Il risultato aspettato è che nessun peer consegni messaggi a livello applicativo, in quanto non viene mai rispettata la condizione di consegna poiché è solamente un peer ad effettuare l'inoltro del messaggio in multicast.

9.2.2 Più sender. In questa tipologia di test ogni peer, in modo concorrente, invia un messaggio al *sequencer* e dopo aver inviato il primo messaggio, inoltra anche un secondo messaggio.

Il risultato aspettato è che i primi N messaggi consegnati al livello applicativo da ciascun peer siano i medesimi, dove N è il numero minimo di messaggi consegnati dai peer.

9.3 Test per l'algoritmo 3

9.3.1 Un solo sender. In questa tipologia di test un peer invia sei messaggi, uno dopo l'altro in modo tale da rispettare l'assunzione *FIFO* ordered.

Il risultato aspettato è che ogni peer consegni, nello stesso identico ordine, i messaggi ricevuti al livello applicativo poiché, essendo un solo sender, non c'è relazione di causa-effetto fra i messaggi.

9.3.2 Più sender. In questa tipologia di test si è seguito uno schema ben preciso per avere un test valido dal punto di vista della verifica. Lo schema è rappresentato nella seguente figura.

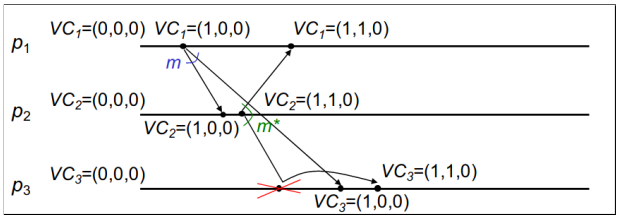

Fig. 3. Test per l'algoritmo 3

In particolare, lo scenario è il seguente:

- Il peer p_1 invia un messaggio con timestamp $(1, 0, 0)$ in multicast. Questo messaggio viene ricevuto dal peer p_2, ma a causa di un ritardo non viene ricevuto subito dal peer p_3.
- Il peer p_2, a causa del messaggio ricevuto, inoltra un messaggio in multicast con timestamp $(1, 1, 0)$. Questo messaggio viene ricevuto da entrambi i peer senza ritardi.
- Il peer p_3, al momento della ricezione del messaggio inviato da p_2, deve posticipare la consegna del messaggio con timestamp $(1, 1, 0)$. Una volta ricevuto e consegnato il messaggio con timestamp $(1, 0, 0)$, può procedere con la consegna del messaggio inviato da p_2.

Il risultato aspettato è che tutti i peer consegnino i messaggi rispettando la relazione di causa-effetto. Quindi, in questo caso significa che ogni peer consegni, nello stesso ordine, i due messaggi al livello applicativo.