Sistemi Distribuiti e Cloud Computing - A.A. 2020/21

Progetto B1: Multicast totalmente e causalmente ordinato in Go

Cristiano Cuffaro (matricola: 0299838)

cristiano.cuffaro@outlook.com

Sommario

L'obiettivo di questo documento è quello di descrivere il lavoro, i ragionamenti e le scelte che hanno portato alla realizzazione del progetto.

1 Descrizione del progetto

Lo scopo del progetto è quello di realizzare, nel linguaggio di programmazione *Go*, un'applicazione distribuita che implementi gli algoritmi di multicast totalmente ordinato e causalmente ordinato.

Tale applicazione deve prevedere:

- un servizio di registrazione dei processi che partecipano al gruppo di comunicazione multicast;
- 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. 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.

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.

Si richiede di fornire un container Docker per l'esecuzione dell'applicazione.

Si richiede inoltre che gli eventuali parametri relativi all'applicazione e al suo deployment siano configurabili.

1.1 Terminologia e assunzioni di base

Un partecipante alla comunicazione, ovvero un membro del gruppo di comunicazione multicast, è indicato anche come *peer*, per esaltare l'uguaglianza con gli altri membri.

La comunicazione tra peer avviene sulla base delle seguenti assunzioni:

- almeno due peer partecipano al gruppo di comunicazione multicast;
- la membership è statica durante l'esecuzione dell'applicazione, quindi non vi sono peer che si aggiungono od escono dal gruppo durante la comunicazione;
- il ritardo di trasmissione di un messaggio è impredicibile, ma finito;
- la comunicazione è affidabile, ovvero:
  - i messaggi non vengono persi e/o duplicati;
  - non vi è la presenza di messaggi spuri[1];
- la comunicazione è FIFO ordered[2].

2 Architettura

Di seguito sono descritti i microservizi che compongono l'applicazione distribuita.

- `peer_service` è il servizio che partecipa alla comunicazione multicast ed è quindi in grado di ricevere ed inviare messaggi aderendo ad uno dei tre algoritmi. Il numero di istanze di questo microservizio è configurabile.
- `registration_service` è il servizio che offre ai peer la possibilità di registrarsi al gruppo e offre anche un servizio di *service discovery*, in modo tale che solo a runtime sia effettivamente nota la specifica composizione del gruppo di comunicazione. Questo permette di avere un binding dinamico anziché statico tra i servizi, ma introduce anche un *single point of failure* all'interno del sistema.
- `sequencer_service` è il servizio che si occupa di coordinare la comunicazione dei peer nel caso in cui l'algoritmo adottato sia quello di multicast totalmente ordinato implementato in modo centralizzato. Questo permette ai peer di non conoscersi necessariamente tra loro, ma introduce nel sistema un *single point of failure* ed un collo di bottiglia per le prestazioni.
- `MongoDB` è un datastore (database NoSQL) orientato ai documenti, gratuito e open-source, che è stato utilizzato per la memorizzazione dei messaggi a livello applicativo per ciascun peer.

[1] I messaggi *spuri* sono messaggi che qualcuno riceve, ma che nessuno all'interno del sistema ha inviato. La loro assenza implica che nel sistema non vi sia un processo corrotto che segnala falsamente la ricezione di messaggi.

[2] I messaggi inviati dal peer p_i al peer p_j sono ricevuti da p_j nello stesso ordine in cui p_i li ha inviati.

Quando l'applicazione distribuita è in esecuzione, ciascun microservizio attivo è incapsulato in un container Docker, godendo di tutti i vantaggi che ne derivano.

Il deployment dell'applicazione è eseguito su singolo host utilizzando Docker Compose, che permette la configurazione dei servizi dell'applicazione e una loro semplice istaziazione, rimozione e gestione.

2.1 Architettura centralizzata

Come mostrato in figura 1, nel caso di algoritmo di multicast totalmente ordinato implementato in modo centralizzato, i servizi interagiscono tra loro secondo il seguente schema:

- i peer contattano il servizio di registrazione per aderire al gruppo di comunicazione;
- il sequencer reperisce la definizione completa del gruppo di comunicazione dal servizio di registrazione;
- i peer inviano i propri messaggi al sequencer, che si occupa di inoltrarli a tutti i partecipanti al gruppo di comunicazione;
- i messaggi che i peer consegnano al lato applicativo, vengono memorizzati sfruttando il datastore.

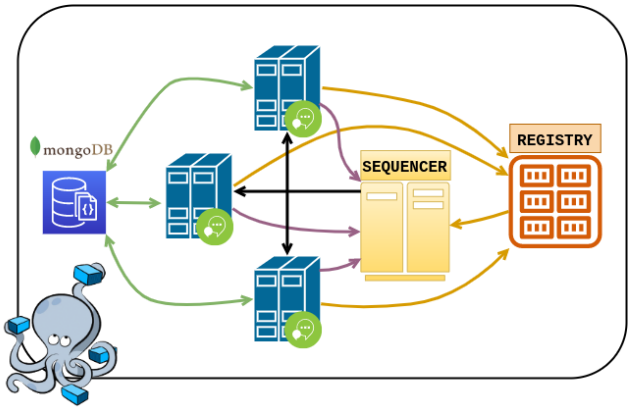

Figura 1. Schema architetturale con coordinatore centrale

2.2 Architettura decentralizzata

Come mostrato in figura 2, nel caso di multicast totalmente o causalmente ordinato implementato in modo decentralizzato, i servizi interagiscono tra loro secondo il seguente schema:

- i peer contattano il servizio di registrazione per aderire al gruppo di comunicazione ed ottenerne la definizione completa;
- i peer si scambiano messaggi direttamente tra loro;
- i messaggi che i peer consegnano al lato applicativo, vengono memorizzati sfruttando il datastore.

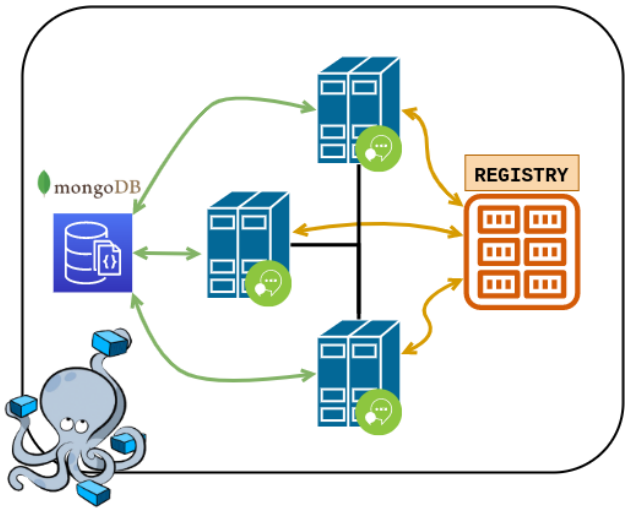

Figura 2. Schema architetturale senza coordinatore centrale

3 Deployment e configurazione del sistema

All'interno del file `yml` su cui si basa Compose, è stato possibile definire alcune importanti caratteristiche della composizione, descritte di seguito.

- È stato definito un *service profile* per il servizio relativo al sequencer, che consente di attivare tale servizio solo nello specifico scenario in cui l'algoritmo di comunicazione in uso è quello di multicast totalmente ordinato implementato in modo centralizzato.
- È stato creato un file `.env` contenente tutte le variabili necessarie a configurare l'ambiente in cui i servizi vengono posti in esercizio. Il vantaggio nel suo utilizzo è che permette di modificare facilmente i valori assegnati alle variabili d'ambiente, senza alterare il file `yml`, poiché definito in modo parametrico.
- È stata utilizzata l'opzione `scale` per il servizio relativo al peer, in modo tale che Compose effettui il deployment del numero di container specificato per tale servizio. Questo permette anche di testare il sistema in scenari diversificati dalla presenza di più o meno peer all'interno del gruppo di comunicazione multicast.

Configurando le variabili d'ambiente è possibile specificare:

- il numero di partecipanti al gruppo di comunicazione;
- i numeri di porta associati ai servizi specificati (in questo modo è possibile effettuare delle scelte che non vadano in conflitto con le configurazioni di default di eventuali altri servizi in uso);
- l'algoritmo di comunicazione multicast da utilizzare;
- il valore del parametro *delay*, espresso in millisecondi, che può essere sfruttato soprattutto per effettuare il testing in condizioni di maggiore stress.

3.1 Dipendenze tra servizi

Come primo task, i peer a startup necessitano di registrarsi presso l'apposito servizio di registrazione, pertanto quest'ultimo deve essere portato a steady state prima che gli altri servizi lo contattino. È possibile controllare l'ordine in cui avviene lo startup dei servizi sfruttando l'opzione `depends_on`, che permette a Compose di avviare i container seguendo l'ordine di dipendenza. Questo però non garantisce che quando un container viene avviato, i precedenti siano già in stato di "*ready*", ma solo che siano "*running*".

Per ovviare a questo problema, come suggerisce la documentazione ufficiale di Docker[3], è stato incluso nell'immagine dei servizi relativi a peer e sequencer lo script `wait-for-it.sh`, che non fa altro che tentare di stabilire una connessione con il server target fintantoché l'operazione non ha successo.

3.2 Ottimizzazione dei Dockerfile

Per ottimizzare i Dockerfile, ovvero renderne minimale la dimensione ed aumentarne la leggibilità e la manutenibilità, sono stati utilizzati i *multi-stage build*[4].

Adottando questo approccio, l'idea quella di definire più `FROM` *statement*, ove ciascuno di essi identifica uno stage della build, e copiare artefatti da uno stage all'altro, facendo sì che nell'immagine finale sia presente solo ciò che si desidera includere.

4 Implementazione

Come richiesto, tutti i servizi sono stati implementati utilizzando il linguaggio di programmazione *Go*.

Nelle seguenti sottosezioni sono indicate le scelte implementative principali su cui si basa la realizzazione dell'applicazione distribuita.

4.1 Comunicazione tra i servizi

Per la comunicazione è stato sfruttato il supporto di *Go* definito per le chiamate a procedura remota, fornito all'interno del package "`net/rpc`" della libreria standard di *Go*. In questo modo, è stato possibile sfruttare la compatibilità *cross-services* dei tipi di dato degli argomenti passati ai metodi remoti.

Indipendentemente dall'algoritmo utilizzato, le connessioni con i peer sono stabilite un'unica volta durante lo startup dei servizi e poi mantenute aperte dalla funzione `RpcHandler` (package "`utils`"), la quale sfrutta i canali di *Go* per ricevere i messaggi da passare come argomento ai metodi remoti.

[3] https://docs.docker.com/compose/startup-order/
[4] https://docs.docker.com/develop/develop-images/multistage-build/

Di seguito è riportata una descrizione sintetica dei metodi accessibili da remoto con cui i servizi dell'applicazione interagiscono.

- `RegisterMember` è il metodo remoto, esposto dal servizio di registrazione, che i peer possono invocare per potersi registrare al gruppo di comunicazione.
- `RetrieveMembership` è il metodo remoto, esposto dal servizio di registrazione, che il sequencer può invocare per ottenere la definizione completa del gruppo di comunicazione multicast.
- `SendInMulticast` è il metodo remoto, esposto dal sequencer, che i peer possono invocare per inviare in multicast un messaggio di update. Tale metodo è disponibile solo se l'algoritmo in uso è quello di multicast totalmente ordinato implementato in modo centralizzato.
- `ReceiveMessage` è il metodo remoto, esposto dai peer, che il sequencer o gli altri peer possono utilizzare per far sì che colui che espone il metodo invocato riceva un messaggio.

4.2 Servizio di registrazione

Per eseguire la registrazione, le informazioni relative a ciascun peer vengono scritte su una nuova linea di un file, il cui contenuto viene poi riversato in un array per restituire la definizione completa del gruppo di comunicazione.

Avendo a che fare con una membership statica, non ha senso che il servizio di registrazione rimanga attivo dopo aver completato il suo lavoro. Per far fronte a questo problema, è stata realizzata la funzione `acceptn`, che non è altro che una reimplementazione del metodo `Accept` per `*rpc.Server`, che permette di accettare esattamente fino a n connessioni in ingresso, consentendo così la chiusura del servizio.

4.3 Messaggi di multicast

Nello snippet 1 è mostrata la definizione della struttura che rappresenta un messaggio di multicast.

```
1 type Message struct {
2     ID        uint64
3     Host      string
4     Content   string
5     Timestamp []uint64
6     Type      string
7 }
```

Snippet 1. Definizione struttura `Messaggio`

- `ID`, rappresenta un codice numerico progressivo che localmente a ciascun peer viene attribuito al messaggio prima di consegnarlo al livello applicativo e stabilisce l'ordinamento dei messaggi anche all'interno del datastore.

- `Host`, rappresenta l'hostname del peer che invia il messaggio di update.
- `Content`, rappresenta il contenuto del messaggio.
- `Timestamp`, serve per marcare il messaggio e il suo significato varia al variare dell'algoritmo in uso:
  - multicast totalmente ordinato implementato in modo centralizzato → rappresenta il numero di sequenza assegnato dal coordinatore centrale;
  - multicast totalmente ordinato implementato in modo decentralizzato → rappresenta il valore del clock logico scalare del peer che invia il messaggio di update;
  - multicast causalmente ordinato implementato in modo decentralizzato → rappresenta il valore del clock logico vettoriale del peer che invia il messaggio di update.
- `Type`, rappresenta il tipo di messaggio e il valore attribuito può essere quello di una delle seguenti macro definite nel package `"utils"`:
  - `UPDATE`, per indicare che si tratta di un aggiornamento da consegnare all'applicazione;
  - `ACK`, per indicare che si tratta di un riscontro per un messaggio di update ricevuto.
    Tale macro è utilizzata solo nell'algoritmo di multicast totalmente ordinato decentralizzato.

4.4 Algoritmo di multicast totalmente ordinato implementato in modo centralizzato

L'implementazione di questo algoritmo prevede semplicemente l'invio dei messaggi al sequencer, che si occuperà poi di marcarli ed inviarli a tutti i partecipanti al gruppo di comunicazione multicast. Per farlo, quando riceve un messaggio `m` genera in modo atomico il numero di sequenza progressivo e lo piazza nella zeresima entry dell'array `m.Timestamp`, dopodiché manda il messaggio su tutti i canali verso gli handler delle connessioni RPC.

Quando un peer riceve un messaggio dal sequencer, può effettuare subito la consegna al lato applicativo, pertanto, tale messaggio viene stampato e anche inserito nella collezione di documenti presso il datastore.

4.5 Algoritmo di multicast totalmente ordinato implementato in modo decentralizzato

Per gestire l'avanzare del tempo logico all'interno del sistema, ciascun peer mantiene una struttura come quella definita nello snippet 2, ove il clock logico scalare corrisponde alla zeresima entry dell'array `Clock`, che viene incrementato in maniera atomica sfruttando il mutex `Lock`.

```
type Time struct {
    Clock []uint64
    Lock  sync.Mutex
}
```

Snippet 2. Definizione struttura `Time`

Di seguito sono descritte le strutture dati utilizzate per l'implementazione di questo algoritmo:

- `queue`, è un array in cui sono mantenuti i messaggi che non possono ancora essere consegnati al livello applicativo;
- `ackForMessages`, è una mappa che permette di associare ad ogni messaggio `m` in coda, il numero di ack ricevuti per tale messaggio.
  La stringa che identifica il messaggio è costruita secondo la regola: `"hostname:timestamp"`.
- `messagesPerPeer`, è una mappa che permette di conoscere il numero di messaggi presenti in coda che sono stati inviati da ciascun peer.
  Si tratta di un'ottimizzazione per evitare la scansione della coda dei messaggi pendenti, riducendo l'overhead dei controlli che si effettuano per valutare la *deliverability* dei messaggi.

4.6 Algoritmo di multicast causalmente ordinato implementato in modo decentralizzato

Anche per questo algoritmo è stata sfruttata la struttura mostrata in 2 per la gestione del tempo logico, con la differenza che qui l'array `Clock` rappresenta il clock logico vettoriale, pertanto, il numero delle sue entry corrisponde alla dimensione del gruppo di comunicazione multicast.

L'unica struttura di supporto per la gestione dei messaggi è l'array `queue`, in cui non avviene più un inserimento ordinato, ma un semplice accodamento dei messaggi.

4.7 Utilizzo del datastore

`MongoDB` memorizza i documenti in una rappresentazione binaria chiamata BSON (Binary JSON) ed è possibile effettuare il marshaling/unmarshaling direttamente di una `struct` definita in *Go*, utilizzando l'apposito driver[5].

Ciascun peer esegue la connessione con il datastore una sola volta durante lo startup, memorizzando le informazioni in una struttura `datastoreHandler`, che permette di interagre con esso in ogni momento.

Per default, i dati memorizzati vengono scritti sul disco del sistema host usando un volume *Docker-managed* e ciò consente all'utente di effettuarne una gestione semplice e trasparente.

5 Layout di progetto

Il layout di progetto segue uno standard[6] non ufficiale, ma che è molto presente tra i progetti che emergono all'interno dell'ecosistema *Go*.

[5] https://docs.mongodb.com/drivers/go/
[6] https://github.com/golang-standards/project-layout

L'obiettivo di dare al progetto una struttura ben definita è quello di renderlo facilmente espandibile ed introdurre una gestione dei package che permetta di aumentarne la riusabilità del codice definito al loro interno.

Per la gestione delle dipendenze è stata utilizzata la tecnologia dei *Go modules*[7], che si contrappone all'*old style* in cui era previsto che il posizionamento del progetto avvenisse all'interno della directory `$GOPATH/src`. Con il nuovo sistema di gestione delle dipendenze un modulo è visto come un insieme di package memorizzati in un *file tree*, ove il file `go.mod` è collocato alla radice. In particolare, questo sistema sfrutta principalmente due file:

- `go.mod`, che definisce il *module path* che coincide anche con l'*import path* utilizzato per la root directory e i suoi requisiti di dipendenza, ovvero gli altri moduli necessari per la build;
- `go.sum`, che mantiene i *cryptographic hashes* del contenuto di specifiche versioni dei moduli.
  Tale file è utilizzato anche per motivi di sicurezza, per assicurare che i futuri download dei moduli effettuino il retrieve degli stessi bit ottenuti nel primo download.

6 Descrizione dei casi di test

Per testare gli algoritmi implementati sono stati ricreati degli scenari che hanno l'obiettivo di mostrare:

- nel caso di multicast totalmente ordinato, che i messaggi ricevuti da ciascun peer seguono lo stesso ordinamento per tutti;
- nel caso di multicast causalmente ordinato, che i messaggi ricevuti da ciascun peer sono sempre ordinati in accordo alla relazione di causa-effetto.

In tutti i test il parametro *delay* gioca un ruolo fondamentale, perché consente di posticipare, di un certo tempo massimo configurabile, l'invio di ciascun messaggio verso ogni peer. Di default, tale parametro è posto a 5 millisecondi, per simulare una latenza dovuta ad una eventuale distanza geografica tra nodi su cui è effettuato il deployment dell'applicazione distribuita.

6.1 Test con sorgente singola

Nello scenario in cui il peer che invia i messaggi è soltanto uno, il test consiste nel far inviare a quel peer dieci messaggi corrispondenti ai numeri da `"0"` a `"9"`.

Il test ha successo se e solo se tutti i partecipanti al gruppo di comunicazione ricevono i messaggi nello stesso ordine.

6.2 Test con sorgente multipla - multicast totalmente ordinato

Anche in questo caso i messaggi da inviare corrispondono ai numeri da `"0"` a `"9"`, ma stavolta ciascun messaggio viene fatto inviare ad un peer diverso. Se i peer sono meno di dieci, in maniera modulare si ricomincia dal primo peer per inviare il messaggio successivo.

Il test ha successo se e solo se tutti i partecipanti al gruppo di comunicazione ricevono i messaggi nello stesso ordine.

6.3 Test con sorgente multipla - multicast causalmente ordinato

Per questo caso di test è stato scelto di riprodurre un piccolo dialogo tra due peer, in cui i messaggi coinvolti sono necessariamente in relazione di causa-effetto tra loro:

1. `catOwner:` "Oh no! My cat just jumped out the window."
2. `catOwner:` "Whew, the catnip plant broke her fall."
3. `friend:` "I love when that happens to cats!"

Per realizzarlo, il primo peer invia i due messaggi relativi a `catOwner` e il secondo peer, nel frattempo, attende di riceverli per inviare la risposta relativa a `friend`. Tutti gli altri peer inviano dei *dummy messages* non causalmente correlati tra loro.

Il test ha successo se e solo se tutti i partecipanti al gruppo di comunicazione ricevono i messaggi relativi al dialogo in maniera ordinata.

7 Thin client per l'utilizzo del sistema

Allo scopo di illustrare il funzionamento dell'applicazione distribuita, è stato realizzato un *thin client*, che permette di fare le seguenti cose:

- associare uno username a ciascun partecipante al gruppo di comunicazione multicast;
- comunicare, attraverso la rete, dei messaggi ai peer per far sì che essi li inviino in multicast a tutti gli altri;
- visualizzare i messaggi ricevuti da parte dei peer, con l'indicazione dei seguenti dettagli:
  - timestamp associato al messaggio;
  - username del peer che ha inviato il messaggio;
  - contenuto del messaggio.
    Questa opzione è presente solo se si esegue il programma in modalità *verbose*.

Per l'interazione con i container, all'interno del client è stato utilizzato l'SDK[8] di Docker per il linguaggio *Go*.

[7] https://go.dev/blog/using-go-modules

[8] https://docs.docker.com/engine/api/sdk/#go-sdk