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 risaltare l'uguaglianza con gli altri membri.

La comunicazione tra peer avviene in base alle 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.

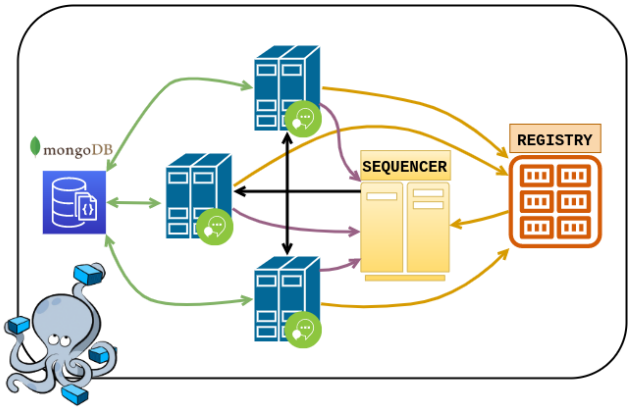

Figura 1. Schema architetturale con coordinatore centrale

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.

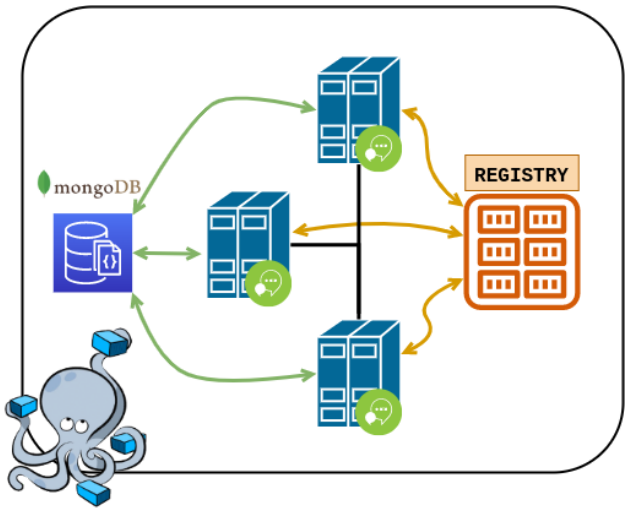

Figura 2. Schema architetturale senza coordinatore centrale

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 dover modificare 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"`. In questo modo, è stato possibile sfruttare la compatibilità *cross-services* dei tipi di dato degli argomenti passati ai metodi remoti.

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 una *slice* per restituire la definizione del gruppo di comunicazione. Questo consente, a chi riceve tale informazione, di avere anche un ordinamento globale dei partecipanti al gruppo, dato proprio dall'ordine con cui essi si sono registrati. Si tratta di un'informazione fondamentale se l'algoritmo in uso è quello di multicast totalmente ordinato implementato in modo decentralizzato.

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 questa necessità, 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

TODO

4.4 Memorizzazione dei messaggi

TODO

5 Layout del progetto

Go modules

6 Descrizione dei casi di test

TODO

7 Thin client per l'utilizzo del sistema

TODO

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