Implementazione di Chord

Arianna Quinci
Corso L.M. Ingegneria informatica
Università degli studi di Roma
Tor Vergata
arianna.quinci99@gmail.com

***Abstract*—Questo documento vuole descrivere il lavoro svolto per implementare l'algoritmo distribuito Chord per il corso di sistemi distribuiti e cloud computing.**

I. Introduzione

Il progetto descritto in questo documento consiste in una possibile implementazione del protocollo distribuito Chord.

In questo articolo si esamineranno: le tecnologie utilizzate per questa implementazione, i meccanismi utilizzati del protocollo Chord, le scelte implementative adottate ed i problemi da esse derivanti.

II. Tecnologie

Il linguaggio di programmazione utilizzato è Go, per quanto riguarda l'implementazione dell'algoritmo, mentre ho usato il linguaggio bash per avviare un nodo dopo che il sistema è già in esecuzione.
Il deploy è avvenuto utilizzando i servizi offerti da Amazon Web Services. Su un'instanza EC2 sono stati installati Docker e Docker Compose, strumenti necessari per utilizzare la containerizzazione.
Ogni nodo dell'anello viene messo in esecuzione su un container Docker differente, lo stesso vale per il service registry. Il client, in esecuzione sull'istanza EC2 interagisce con il service registry utilizzando il suo hostname ed il numero di porta su cui espone i servizi. Docker Compose si occupa invece dell'orchestrazione dei containers: crea una rete su cui i containers possono comunicare tra loro, si occupa di avviare i containers in un determinato ordine e di mappare le porte del container su specifiche porte della macchina host.
Per gestire la comunicazione tra attori su containers diversi ho utilizzato le chiamate a procedura remota offerte dal linguaggio Go.
Per la configurazione statica delle informazioni dei nodi ho utilizzato dei files json, sia per quanto riguarda i nodi da immettere nella rete all'avvio, sia per quanto riguarda il nodo da aggiungere quando il sistema è già avviato.
Anche per la configurazione del service registry e della grandezza della rete ho utilizzato un file json.

III. Descrizione del protocollo

Il protocollo Chord è stato progettato per affrontare uno dei problemi principali nelle reti P2P: la localizzazione efficiente di risorse all'interno della rete, senza la necessità di una struttura centralizzata o gerarchica. Questo è particolarmente importante nelle reti P2P, dove i nodi possono unirsi e lasciare la rete dinamicamente.

Le reti P2P sono overlay networks, ovvero reti logiche che interconnettono i peers, ci sono due tipologie di overlay networks: overlay networks strutturate e non strutturate. Chord appartiene alla prima tipologia e presenta una struttura ad anello. Questo implica dire che ci sono dei vincoli su come le risorse vadano posizionate nella rete rispetto ai nodi. Il vantaggio di avere reti strutturate è che il costo di lookup diminuisce, d'altra parte aumenta il costo per la join e la leave dei nodi; in particolare emerge che il costo computazionale per il lookup in una rete strutturata basata sul protocollo Chord è pari a O(log(N)), con N numero di nodi nell'anello; mentre per quanto riguarda la join e la leave il costo è O($\log^2 (N)$).

A. Routing

Il routing in Chord è basato su DHT, che sono un'astrazione distribuita delle hash tables, in cui le risorse sono mappate in una tabella utilizzando come identificativo l'hash della risorsa.

Chord utilizza consistent hashing, ciò implica dire che:

- nodi e risorse sono mappati sullo stesso spazio degli identificatori
- ogni nodo gestisce una porzione contigua di risorse

Il vantaggio principale del consistent hashing risiede nel fatto che in caso di ridimensionamento della rete il numero di chiavi da riassegnare risulta minimo.
Le risorse gestite da un nodo sono quelle che hanno id compreso tra lui stesso ed il suo predecessore in senso antiorario; ne consegue che la risorsa k risulta gestita dal nodo con identificativo p>=k più piccolo possibile.
L'anello Chord ha un numero di nodi n esprimibili sempre come potenza di 2, indichiamo la taglia dell'anello come 2^m; ogni nodo possiede una finger table, una struttura dati che mantiene un numero di entries pari al numero di bits necessari per il GUID, dunque essa conterrà m entries. L'utilizzo delle finger tables permette ad ogni nodo di avere una conoscenza parziale del sistema, il suo dimensionamento è frutto di un tradeoff tra la conoscenza che un nodo può avere del sistema ed il quantitativo di informazioni da memorizzare in ciascun nodo. Se ogni nodo conoscesse ogni altro nodo del sistema il costo di lookup diventerebbe *O(1)*, ma sarebbe grande la mole di informazioni che il sistema dovrebbe mantenere, quindi le operazioni di aggiornamento della finger table stessa diventerebbero più onerose.
Presa la finger table del nodo p l'i-esima entry contiene $succ(p + 2^{i-1} \% 2^m)$.
L'algoritmo di routing funziona come segue.

Il nodo p riceve la richiesta di lookup per una certa risorsa j:

1. Controlla se è lui stesso a gestire la risorsa, in tal caso la ritorna direttamente, altrimenti:
2. Se $p<j<FT[0]$ inoltra la richiesta al suo successore
3. Se $p>FT[0]$ scorre la propria finger table ed inoltra la richiesta al nodo i tale per cui $FT[i-1]<j<=FT[i]$

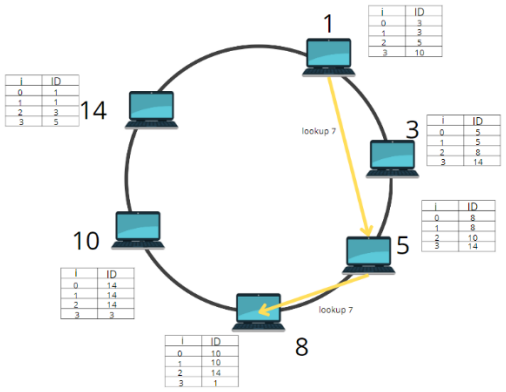

Fig.1 Esempio di ricerca di risorsa con id pari a 7

Come possiamo notare dall'algoritmo la ricerca è totalmente decentralizzata, sono i nodi stessi che comunicando tra loro riescono a trovare la risorsa, senza bisogno di un'entità centralizzata.

B. Join

Quando un nodo p vuole entrare nella rete deve trovare il proprio posto nell'anello, quindi:

1. Avvia una richiesta di ricerca del successore di p+1
2. Si lega al proprio successore e lo informa di aggiornare il proprio predecessore
3. Chiede al successore di comunicargli il proprio predecessore
4. Informa il predecessore di essere entrato nella rete
5. Inizializza la propria finger table
6. Trasferisce le risorse di cui deve divenire responsabile dal proprio successore a lui stesso.

C. Leave

Quando un nodo p vuole uscire dall'anello:

1. Trasferisce le proprie risorse al suo successore
2. Aggiorna il predecessore del proprio successore, facendolo puntare al suo predecessore
3. Aggiorna il successore del proprio predecessore, facendolo puntare al suo successore

IV. DETTAGLI IMPLEMENTATIVI

Nel sistema realizzato ci sono tre tipologie di attori:

1. Il service registry
2. I nodi
3. Il client

A. Service registry:

Il service registry mantiene il mapping tra gli identificativi dei nodi nella rete con il loro indirizzo IP ed il loro port number. Questo mapping è realizzato con il campo "ServiceMapping" della struct "ServiceRegsitry". I nodi per comunicare tra loro contattano il service registry per ottenere hostaname e port number del nodo con cui vogliono comunicare, lo stesso avviene da parte del client, quando vuole interagire con il sistema.

Le funzioni esposte dal service registry sono:

1. JoinRequest: si connette al nodo di boostrap ed invoca la funzione Join
2. LeaveRequest: chiama il nodo che deve lasciare l'anello ed invoca la funzione Leave
3. Get: si connette al nodo di bootstrap ed invoca la Get
4. Put: si connette al nodo di bootstrap ed invoca la Put
5. Delete: si connette al nodo di bootstrap ed invoca la Delete
6. Retrievenodes: restituisce un array contenente gli identificatori di tutti i nodi attualmente presenti nel sistema.

Le altre funzioni che può eseguire servono per implementare queste funzionalità.

Tra queste abbiamo "generateChordNodeId" e "generateResourceId", che calcolano la funzione SHA-1 per calcolare id dei nodi e delle risorse. In particolare nel calcolare l'id del nodo se un certo id è già in uso va via via ad incrementare di 1 in valore del nuovo id finchè non ne trova uno libero, dopo un numero di iterazioni pari alla taglia dell'anello si ferma, in quanto significa che non c'è alcun id disponibile.

B. Nodo:

I nodi sono i server nel sistema, essi gestiscono risorse e possiedono un loro indirizzo IP, un loro hostname ed un loro port number. Ogni nodo ha associato un id, calcolato mediante la funzione di hash crittografico SHA-1 applicata al proprio IP ed al proprio port number, concatenati tra loro.

Un nodo per comunicare con un altro nodo avvia una richiesta di lookup, da cui ottiene l'id del nodo con cui comunicare, a questo punto contatta il service registry per conoscere IP e PN relativi a tale id.

Le funzionalità esposte dal nodo sono:

1. Join: implementa l'algoritmo di join di un nodo
2. Leave: implementa l'algoritmo di leave di un nodo
3. Get: permette di ottenere una risorsa nel sistema
4. Put: permette di inserire una risorsa nel sistema
5. Delete: permette di rimuovere una risorsa dal sistema.

Le altre funzioni presenti sono invocate internamente e cooperano per offrire le funzionalità di cui sopra.

Tra queste la funzione checkId permette di verificare all'interno di un nodo se l'id per una risorsa è già in uso ed in tal caso di trovarne un altro disponibile, eventualmente passando la risorsa al proprio successore.

Nella funzione put, dalla quale viene invocata checkId, per limitare le iterazioni vado a passare tra gli argomenti un contatore, che va a contare quante volte checkId viene invocata, nel caso in cui venga invocata un numero di volte pari alla taglia dell'anello significa che non è stato trovato un id libero, dunque l'anello è pieno e non può più accettare risorse.

C. Client:

Il client interagisce con il sistema Chord per richiedere una serie di servizi. Le funzioni che egli può invocare sono:

1. Put: inserimento di una risorsa nell'anello. La put prende in input il valore della risorsa e va ad invocare il service registry in modo tale che vada a contattare il nodo di bootstrap per dare inizio ad una procedura di lookup della risorsa. Il valore di ritorno della rpc al service registry consiste nell'id assegnato alla risorsa, esso verrà poi stampato a schermo.
2. Get: richiesta di una risorsa dall'anello. Prende in input l'identificatore di una risorsa e invoca tramite rpc il service registry, che darà inizio ad una procedura di lookup della risorsa, andando a contattare il nodo di bootstrap. Il valore di ritorno della rpc sarà il valore della risorsa corrispondente all'id e verrà stampato a schermo.

3. Delete: cancellazione di una risorsa dall'anello. Il funzionamento della funzione delete è analogo a quello della "get", con la differenza che una volta che nell'anello la risorsa viene trovata, viene anche rimossa dal sistema.
4. Leave: richiesta di far uscire un nodo dall'anello. Per far uscire un nodo dall'anello il client prima ottiene la lista dei nodi presenti, invocando tramite rpc la funzione "RetrieveNodes" sul service registry, dopodichè inserisce il valore dell'id del nodo che vuole far uscire dall'anello ed invoca, sempre mediante rpc, la "LeaveRequest" sul service registry.

Nel client non è disponibile alcuna funzione che permetta di aggiungere un nuovo nodo nel sistema, in quanto per realizzare questa operazione ho utilizzato uno script bash che legge da un file di configurazione

V. Conclusioni

In questa implementazione dell'algoritmo non è previsto alcun meccanismo di tolleranza ai guasti, si cerca semplicemente di limitare i danni in caso di crash di un nodo: qualora il service registry si accorga che un nodo non risponde, mette in atto una procedura di recupero, per cui riesce a ricostruire la struttura dell'anello, andando a collegare tra loro il predecessore del nodo andato il crash con il suo successore, fa questi consultando la propria "mappa dei servizi".
Affinché il sistema risulti fault tolerant si dovrebbe inserire un meccanismo di replicazione, per cui ogni risorsa dovrebbe essere gestita da un sottoinsieme dei nodi del sistema.
Altro problema relativo a questa implementazione sta nel fatto che il service registry risulta essere un collo di bottiglia e sigle point of failure per il sistema, anche qui per risolvere il problema si potrebbe usare un meccanismo di replicazione.