Documento di supporto al progetto di Chord

Comprensione del contesto di lavoro

See: Home · sit/dht Wiki (github.com) See: 3-DS_Architecture.pdf

Chord è un algoritmo lookup di ricerca delle risorse in un contesto P2P. I *nodi/peers* e le *risorse* sono mappate in un anello usando *hashing consistente*.

> Voglio evitare che, nel caso dovessi ridefinire la dimensione della Distribuited Hash Table, ciò comportasse anche il remapping di *tutte le chiavi*. Cioè voglio evitare di riassegnare tutte le chiavi, vorrei aver meno impatto possibile. La soluzione è il *consistent hashing*. Il tutto prevede un contesto *exact-match*, ovvero cerchiamo la singola risorsa per nome e troviamo quella risorsa ad esempio, ma non possiamo cercare "tutte le risorse in certo range".
>
> Manca da definire la Distribuited Hash Table, cioè la versione distribuita di una Hash Table, che mappa chiavi in valori. In pratica, la risorsa che cerchiamo ha una certa *chiave*, quello che facciamo è mappare questa chiave in un certo *slot* che la contiene, e cercare in questo slot (detto anche Bucket). E' distribuita perchè questi bucket sono distribuiti tra nodi diversi.

Soffermiamoci sul capire come funziona una Distribuited Hash Table. Nella DHT abbiamo coppie <key K, value V>. La chiave K identifica la *risorsa* **contenuta in V**, e K corrisponde al *resource GUID* (composto da m bits, ad esempio usando SHA-1), cioè l'identificatore. Quali operazioni possiamo compiere per relazionarci ad una DHT?

- **V = get(K)**: Ottengo V associato alla chiave K dal nodo che memorizza V.
- **put(K,V)**: Memorizzo la risorsa V nel nodo responsabile della risorsa identificata da K.
- **remove(K)**: Rimozione del riferimento di K associato a V.

E' la nostra applicazione distribuita che si interfaccia con la DHT.

Ogni nodo gestisce una porzione **contigua** della DHT. L'ID del nodo e delle risorse sono mappate sullo stesso spazio di indirizzamento. L'idea di routing è: "**Data K, la mappo nel GUID del nodo più vicino a K.**"

Un protocollo basato su questi concetti è CHORD.

Chord nello specifico

I nodi e le risorse sono mappate in un ring usando **consistent hashing**. Ogni nodo è responsabile delle chiavi tra sè stesso e il nodo *precedente*, assumendo un senso orario. La risorsa con chiave *k* è gestita da un nodo avente per identificativo il più piccolo identificativo tale che $id \geq k$. Tale nodo è chiamato il ***successore della chiave k, o succ(k)***. Vediamo un veloce esempio.

 1.png

Chi gestisce la risorsa 1? Il più piccolo nodo avente $id \geq 1$, quindi partendo da 0 scorro finchè non eccedo 1, e prendo il precedente. In questo caso **succ(1)=1**, perchè il nodo 1 è il primo che è ≥ 1. Il successore di 10? scorro finchè non trovo il primo nodo che supera/è uguale la risorsa. In questo caso **succ(10)=12**.

Consistent Hashing in CHORD

E' rilevante perchè, in questa applicazione, la maggior parte delle chiavi non vengono rimappate se un nodo arriva o esce dalla rete. Inoltre, ogni nodo mediamente gestirà lo stesso numero di risorse, realizzando quindi *load balancing.*

Lookup in CHORD

Chord utilizza le **Finger Table** per realizzare il lookup. E' un buon compromesso tra il conoscere tutte la rete (molto veloce, ma troppe info da gestire), o conoscere solo il nodo successivo (meno informazioni, ma più lento, opera in O(N)). La finger table è una lista parziale di nodi progressiva, più vado "lontano" nella tablella, più le informazioni sono vaghe.

Realizzazione di una FINGER TABLE

Essa ha *m righe*, dove *m* sono i bit dedicati ai vari GUID. Ad esempio, se m=3, allora posso definire fino ad $2^3 = 8$ nodi.

Se FT_p è la tabella del nodo *p*, allora la riga i-esima sarà:$FT_p[i] = succ(p + 2^{i-1}) \ mod \ 2^m$, con $1 \leq i \leq m$

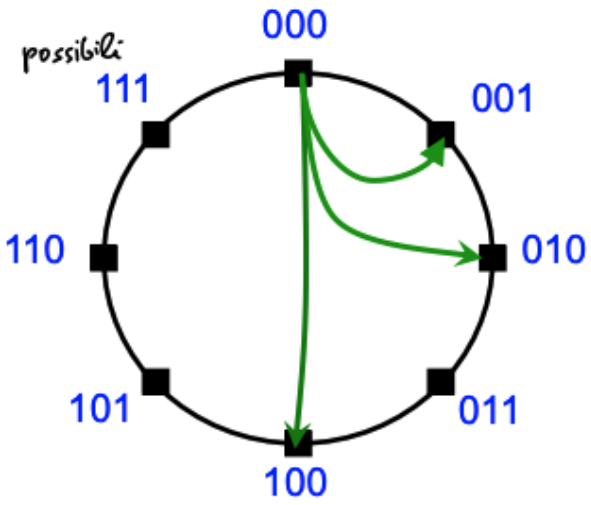
- Basic idea
  - Each node knows “well” closer nodes and has a rough knowledge of farther nodes

In the example m=3 (2^3 ID possibili)

$FT_0[1]=0+2^0=1$

$FT_0[2]=0+2^1=2$

$FT_0[3]=0+2^2=4$



a Cardellini - SDCC 2021/22

Algoritmo di routing di CHORD

Vogliamo mappare la chiave *k* in *succ(k)*, partendo da un nodo *p.*

- Se $k \in p$, l'algoritmo termina.
- Se $p < k \leq FT_p[1]$, p inoltra la richiesta al suo successore. In pratica la prima riga della FT di p identifica il suo successore. Se vediamo che quella risorsa è di sua competenza, inoltriamo a lui la richiesta.
- Altrimenti, cerco il nodo più lontano *q* di indice *j* che non ecceda la risorsa, ovvero: $FT_p[j] \leq k < FT_p[j+1]$. Se ad esempio abbiamo nella FT il nodo 2 nella riga "j", e il nodo 4 nella riga "j+1" e la risorsa da cercare ha k=2 oppure k=3, allora tale risorsa sarà gestita dal nodo "2". Altrimenti vado avanti nelle righe.

Gestione JOIN & LEAVE

Per facilitare il tutto, ogni nodo mantiene un puntatore anche al suo predecessore, per semplificare le operazioni. Come lo individuo? In senso **antiorario**, il predecessore del nodo p è il primo nodo incontrato partendo da p.

JOIN nel dettaglio

Quando un nodo *p* entra, deve trovare il suo posto nell'anello. Il nodo *p* chiede ad un altro nodo di trovare il successore *succ(p+1)* nell'anello. Il nodo *p* si lega al suo successore *p+1*

informandolo della sua presenza. Il nodo *p* inizializza la sua FT, computando quindi i suoi successori, (il primo lo abbiamo trovato nei punti precedenti!) $succ(p + 2^{i-1}),\ 2 \leq i \leq m$.

Il nodo *p* informa il suo predecessore di aggiornare la sua FT.

Il nodo *p* trasferisce, dal suo successore a sè stesso, le chiavi di cui deve diventare responsabile.

LEAVE nel dettaglio

Nel caso di addio volontario, il nodo *p* avverte la rete. Il nodo *p* trasferisce le key per cui è responsabile al suo successore. Il nodo *p* comunica al suo successore *p+1* che il suo predecessore non è più *p*, bensì *p-1*. Il nodo *p* comunica al suo predecessore *p-1* che il suo successore non è più *p*, bensì *p+1*. Per mantenere la finger table aggiornata, periodicamente ogni nodo una procedura di stabilizzazione dell'anello (come?).

FAULT TOLERANCE nel dettaglio (facoltativo)

Supponiamo che un nodo crashi, c'è necessità di replicare i dati <K,V>. Si creano R repliche, ciascuna delle quali viene memorizzata negli R-1 nodi successori nell'anello. R configurabile. Tuttavia richiede di conoscere successori multipli.