Chord

Motroi Valeriu B5

December 23, 2016

1 Introducere

În sistemele descentralizate, o problemă majoră este găsirea eficientă a unui calculator din aceeași rețea. Pentru a soluționa această problemă, de multe ori sunt folosite tabelele hash distribuite. Sunt mai multe protocoale de implementare a tabelelor hash distribuite: Pastry, CAN, Tapestry, Chord. Eu o să descriu protocolul Chord.

Chord este un protocol și un algoritm pentru sistemele descentralizate care păstrează o tabelă hash distribuită în care se păstrează o structură pentru a mapa perechie cheie-valoare prin asignarea a unei chei la diferite computere (numite noduri). Un nod v-a păstra niște valori de care este responsabil. Diferența față de tabelele hash simple, este posibilitatea de a mapa informația într-un sistem descentralizat.

Scopul este de a avea o structură generală balansată, pentru a păstra sistemul robust și rapid, indiferent dacă se alătură sau pleacă nodurile. Cu o mare probabilitate, în sistemul Chord cu N noduri, toate operațiile se întâmplă în $O(log^2 N)$ mesaje.

2 Tehnologii utilizate

Pentru comunicare se utilizează protocolul TCP/IP, pentru că este foarte important să primim toate mesajele corecte și în ordinea corectă. În cazul în care am pierde vreun mesaj, structura noastră nu mai este balansată sau/și nu conține informații valide.

Pentru fluxul de date input/output, se folosește API-ul Socket BSD. Pentru că este cel mai stabil și ușor utilizabil.

Pentru a avea un sistem în care clienții comunică concurent, fiecare nod o să trateze fiecare mesaj folosind multithreading.

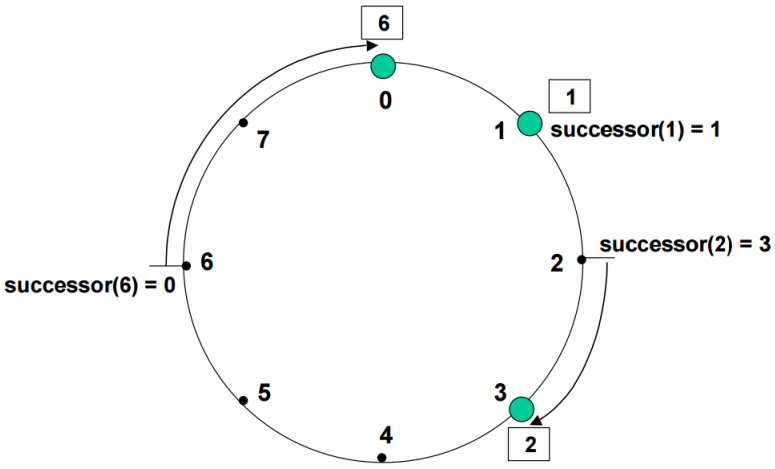
3 Descriere Chord

Chord suportă operațiile: introduce în nou nod în rețea, elimină un nod din rețea, schimbă informația dintr-o cheie și returnează informația dintr-o cheie.

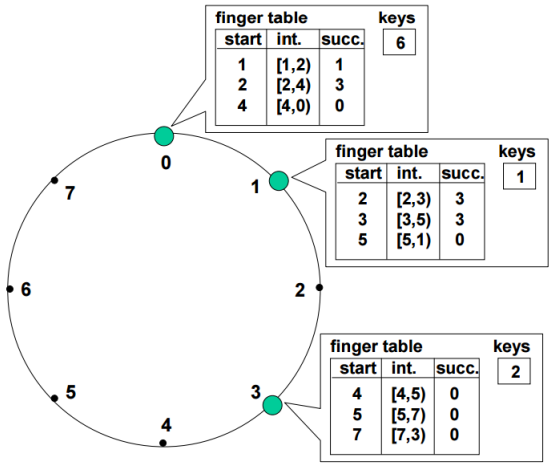
Tabela hash este una circulara, care conține noduri de la 0 la $2^M - 1$ noduri. Fiecare client care vrea să se alăture la rețea primește unul din aceste 2^M noduri. Valoare acestui nod se calculează folosind o funcție hash ($SHA - 1$) din care se păstrează doar primii M biți. Aceasta are mai multe avantaje:

1. Probabilitate mare ca fiecare nod din rețea, să păstreze aproximativ același număr de chei
2. Probabilitatea mare ca doar $O(\frac{1}{N})$ din chei sa fie mutate pe poziții noi

Noul client care s-a alăturat rețelei, este responsabil de toate cheile care se află între valoarea asignată lui și primul client care se află în sens trigonometric. Putem spune că fiecare informația mapată cu o cheie, se păstrează în primul calculator care se întâlnește pe drumul circular în sens anti-trigonometric începând cu valoarea cheii. În imaginea de mai jos, avem 3 clienți (0, 1 și 3).
Putem observa, că cheia 1 se păstrează în nodul 1, cheia 2 în nodul 3 și cheia 6 în nodul 0.

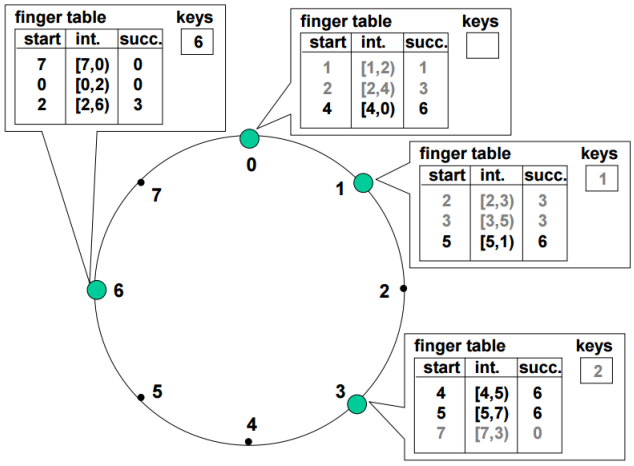


În acest moment, putem observa, că dacă am ști succesorul fiecărui nod, atunci am puteam să găsim ce calculator ar trebui să păstreze o anumită cheie. Totuși, o astfel de abordare ar duce la complexitate $O(N)$, care nu este tocmai ce ne-am dorit. Pentru o căutare mai eficientă, o să memorăm pentru fiecare nod M vecini într-un tabel numit $Finger[]$, unde $Finger[i]$ păstrează informații despre primul nod care se află la distanță cel puțin 2^i.



În imaginea de mai sus, putem observa cum se păstrează informația în tabela $Finger[]$.

Când se alătură un nou client la rețea, tabela $Finger[]$ suferă niște schimbări și anume, unele noduri își modifică unele câmpuri din tabela $Finger[]$, completând unde este necesar cu noul nod adaugat. În imaginea de mai jos putem observa ce schimbari au avut loc după înserarea unui nou nod.



Operația de adăugarea a unui nou nod, este una dintre cele mai costisitoare și este așteptat să lucreze în complexitate $O(log^2N)$, deoarece fiecare nod păstrează informații despre $O(logN)$ noduri și fiecare operație de comunicare necesită $O(logN)$ mesaje, în final obținem $O(log^2N)$.

La plecarea unui nod, toate informațiile care le deține acel nod, se transferă primului succesor în ordine anti-trigonometrică. În așa fel, păstrăm proprietățile tabelei $Finger[]$.

4 Detalii de implementare

În sistemul P2P, fiecare nod este și client și server, pentru asta, fiecare proces, o să conțină mai multe threaduri. Fiecare client, o să comunice cu alt client, folosind un client nou, pentru a putea comunica concurent. Clientul v-a putea efectua comenzile:

- Find File_Name - această comandă caută un fișier și afișează drumul de la nodul curent la nodul unde a găsit fișierul sau -1 în caz că fișierul nu a fost găsit.
- Insert File_Name - Insereaza un fișier nou în tabela hash distribuită. Pentur a găsi cheia fișierului o să aplicăm $SHA-1$ pe denumirea fișierului și o să folosim primii M biți a hash-ului obținut ca fiind cheia fișierului.

Pseudocodul la funcțiile necesare:

```
// Cere de nodul N sa gaseasca succesorul lui X
find_succ(N, X)
   N# = find_pred(N, X)
   return N#.succ

// Cere de la nodul N sa gaseasca predecesorul lui X
find_pred(N, X)
   N# = N
   while(X not in (N#, N#.succ))
      N# = closest_precending_finger(N#, X)
   return N#

// Returneaza cel mai mare nod, mai mic ca X
closest_precending_finger(N, X)
   for i = M downto 1
      if (N.finger[i].node in (N, X))
         return N.finger[i].node
   return N

// Nodul N vrea sa se alature la retea
// N# este un nod arbitrar in retea
join(N, N#)
   if (N#)
      init_finger_table(N, N#)
      update_others(N)
   else // N e primul nod in retea
      for i = 1 to m
         finger[i].node = n
      n.pred = n
      n.succ = n
```

```
// Initializeaza tabela finger a unui nod
// N# este un nod arbitrar deja in retea
init_finger_table(N, N#)
   N.finger[1].node = find_succ(N#, N.finger[1].start)
   N.pred = (N.finger[1].node).pred
   (N.finger[1].node).pred = N
   for i = 1 to m - 1
      if(N.finger[i + 1].start in [N, N.finger[i].node))
         N.finger[i + 1].node = N.finger[i].node
      else
         N.finger[i + 1].node = find_succ(N#, N.finger[i + 1].start)

// Schimba toate tabelele finger
// ale nodurilor care trebuie unite cu N
update_others(N)
   for i = 1 to m
      // gaseste ultimul nod p care trebuie schimbat la distanta 2^i
      p = find_pred(N, N - 2^(i - 1))
      update_finger_table(p, N, i)

// daca Z este al i-lea finger a lui N
// modifica N.finger[i] cu s
update_finger_table(N, s, i)
   if (s in [N, N.finger[i].node))
      N.finger[i].node = s
      p = N.pred
      update_finger_table(p, s, i)

// Stabilizarea periodica, in caz de plecari din retea neanuntate
// si anunta pe succesor despre N
stabilize(N)
   x = (N.finger[1].node).pred
   if (x in (N, N.finger[1].node))
      N.finger[1].node = x
   notify(N.finger[1].node, N)

// N# credea ca poate fi predecesor lui N
notify(N, N#)
   if(N.pred == NULL or N# in (N.pred, N))
      N.pred = N#

// reinoirea periodica la tabela finger
fix_finger(N)
   i = random_index > 1
   N.finger[i].nod = find_succ(N, N.finger[i].start)
```

5 Concluzii

Protocolul Chord este o alegere bună când vine vorba de sisteme descentralizate. Este ușor implementabil și un punct forte este creșterea logaritmică a costului operațiilor, la alăturarea a noi clienți în rețea. Cu o foarte mare probabilitate, rămâne robust și scalabil. Un lucru important este faptul că putem demonstra complexitatea fiecărei operații și corectitudinea lor.

6 Bibliografie

- https://en.wikipedia.org/wiki/Chord_(peer-to-peer)
- https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf
- https://en.wikipedia.org/wiki/Distributed_hash_table
- https://en.wikipedia.org/wiki/SHA-1