Proseminar Peer-To-Peer Informationsystems

Topic: *Chord: A scalable peer-to-peer look-up protocol for internet applications*

Write-Up by Cathrin Weiß

WS 2004/2005, Prof. Weikum

Saarland University, November 30, 2004

Contents

Chapter 1

Introduction

Peer-to-Peer Systems become more and more popular in our modern network world. Peer-to-Peer networks have no central unit, a server, to organize the clients. Clients have to organize themselves within the network.
Without a server it is difficult for one client that looks for specific information within this network to localize (as efficiently as possible) a peer that could provide the desired information.
This is one of the main problems to be solved in Peer-to-Peer networks.

There exist systems, like the file-sharing application Gnutella for example, that use a flooding mechanism to retrieve information about clients that host desired data. The problem there is that this method causes to much traffic on the network (in this case on the BIG network, the Internet, which is quite often overloaded).

to avoid such disadvantages the protocols peer-to-peer software is based on must fullfill requirements like scalibilty, reliability, performance and so on... and there are several protocols. One of them is Chord.

Chapter 2

The Chord Protocol

2.1 What is Chord?

Chord is an efficient, simple and scalable lookup protocol for Peer-to-Peer networks. Its correctness and performance are provable. Chord supports only one operation: It maps a given key onto a node in the network. Thus, it is possible to implement data localization by associating a key with a data item.

2.2 Chord's Topology and Consistent Hashing

Chord has its own network topology, called Chord Ring. The Chord protocol uses a consistent hash function (SHA-1, secure hash standard) to assign a m-bit identifier to each node and each key. The integer m is chosen big enough to minimize the probability that two elements on the ring are assigned the same identifier and provides an ID space from 0 to $2^m - 1$. Nodes are arranged on the ring by hashing their IP addresses, keys by hashing the key itself. A Key's ID always is assigned to that node whose identifier is greater or equal than the key's. (To simplify up to now "'key"' is used for both, the key itself and its identifier; even so it will be done with "'node"'.) This node (the first node clockwise from the key) is called the key's $successor$ or $successor(key)$.

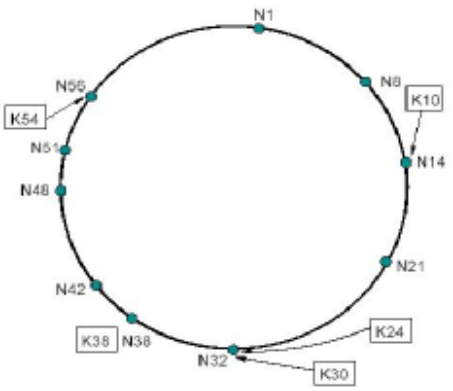

Figure 2.1: Illustration shows how identifiers are arranged on the chord ring and assignment keys to successors

2.3 Simple key location

Let's develop a first simple algorithm for key location on a Chord ring. The idea is to forward a node's request over any succeeding node to, finally, reach a node that is able respond. Our first protocol consists of one function, the `find_successor()` function. Let's define $n.successor$ as n's pointer to its succeeding node and $n.predecessor$ as n's pointer to its preceding node.

//ask node n to find the successor of id

```
n.find_successor(id)
 if (id in (n, successor])
  return n.successor;
 else
  return successor.find_successor(id);
```

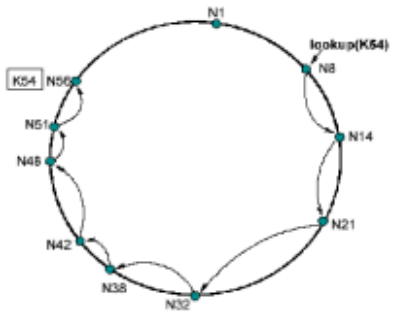

Figure 2.2: Illustration shows the lookup of node 8 for key 54's successor over the network

The algorithm works but it is not very efficient. Suppose the case that n wants to lookup its predecessor's predecessor. Then there the number of required hops is alsmost equal to the number of nodes on the Chord ring. Thus, the number of messages to be forwarded increases linearly to the number of nodes in the network. This fact is not acceptable for huge networks. Chord uses an improved algoritm.

2.4 Scalable key location

In Chord, each node has a routing table, called finger table. The entries there are determined by steps of 2^i and i can be an integer between 0 and m (m is the identifier length). The i-th entry is the node that succeeds the actual node clockwise by at least 2^{i-1}. Thus, a routing table could contain up to m entries. A routing table contains not enough information to reach a node directly but enough to forward a query to its nearest neighbourhood.

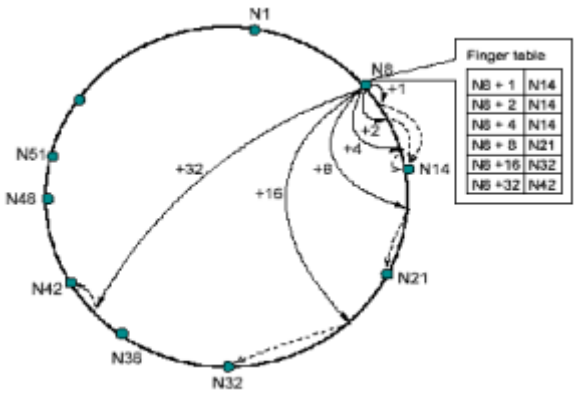

Figure 2.3: Illustration shows which nodes could be reached via a finger table

The lookup procedures are:

//search for the highest preceding node of id

```
n.closest_preceding_node(id)
 FOR i = m DOWNTO 1 DO
  IF (finger[i] in (n ; id))
   RETURN finger[i];
  RETURN n;
```

//ask node n to find the successor of id

```
n.find_successor(id)
  IF (id in (n ; successor])
   RETURN successor;
  ELSE n' = n.closest_preceding_node(id);
   RETURN n'.find_successor(id);
```

Now we have a logarithmic lookup time. That means in a N node network the number of required hops to find a specific node is in $O(\log N)$.

2.5 Joining nodes and stabilization process

If you perform a lookup you expect it to be correct. Therefor there are needed some procedures to stabilize the Chord ring after node joins.

```
n.create()
 predecessor = nil;
 successor = n;
```

```
m.join(n)
 predecessor = nil;
 successor = n.find_successor(m);
```

`n.create()` is the creation of a chord ring; one node builds a chord ring with itself as successor. Now nodes are able to join.

`m.join(n)`: A node m joins the system containing node n already; m sets its precessor nil and asks n for its successor.

```
n.notify(m)
 IF (predecessor = nil
     OR m in (predecessor;n))
  predecessor = m;
```

```
n.stabilize()
 x = successor.predecessor;
 IF (x in (n; successor))
  successor = x;
  successor.notify(n);
```

The stabilzation process: During `stabilize()` a node checks whether its successor's predecessor is itself. If not it changes its successor to its old successors predecessor and notifies its new successor about its existence via `notify()`. During `notify()` the notified node checks whether its predecessor equals nil or if the notifying node is between its old predecessor and itself. If so, it changes its predecessor to the node which sent the notify.
There are two more procedures; `fix_fingers()`, which takes care that the finger tables are up to date and `check_predecessor()` which checks if the actual node's predecessor has failed. If so, the predecessor is set to nil. That makes the node free for accepting a new predecessor during `notify()`. Those four procedures are running periodically in the background to keep the system state stable.

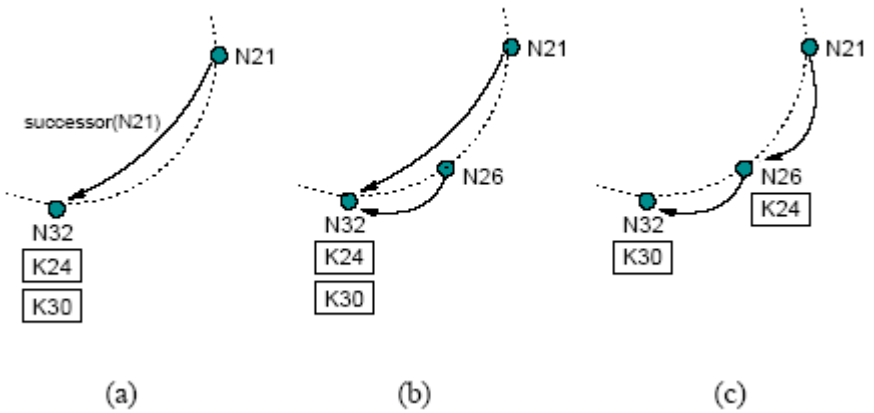

Figure 2.4: a) Initial state b) Node 26 joins the system;accepts Node 32 as successor c) Node 21 accepts Node 26 during stabilize() as its new successor

Do node joins influence lookup correctness or performance?
The answer is that if successor pointers and routing tables are already corrected they do not have any influence. Lookup time still is in $O(\log N)$ and correct lookups are received. If the successor pointers are correct but routing tables are not it might be that there are a few linear hops between the lookup what means that lookup time increases negligibly; lookups are still executed correctly
If neither routing tables nor successors are updated lookup will fail. That means that no incorrect lookup is returned. In this case it is upper layer's business to set a timeout and retry the lookup afterwards.

2.6 Node departures/failures

When a node n in the system fails, it could happen that that nodes predecessor m is not able to realize its new successor because it does not appear in m's finger table. If, for example, any node in m's routing table would fail, there is no chance for m to accept ANY successor. To avoid this unpleasant situation, each node maintains a successor list if size r with its r succeeding nodes (now really succeeding and not in steps of 2^i). If we have this successor list and every node fails with probability p, the probability that every node in the successor list fails is p^r. Thus, increasing r makes the system more robust but also requires more storage.
It is provable, that even if half of the nodes in the system fail the `find_successor()` procedure still runs in $O(\log N)$ and returns correct lookups.
In simulations was shown that even high failure rates have few influence on lookup performance.
Even during failures, the periodically executed procedures take care of lookups returned correctly. Thus correctness still is guaranteed though lookups may fail which again is upper layer's business.
Another point of view would be if nodes leave volutarily. That might be treated as a failure as well. But it is in fact quite more efficient if the node informs its predecessor and its successor about its aim so that they could update their pointers immetiately.

Chapter 3

Disadvantages and Improvements

Now we got to know chord as a simple, performant and reliable lookup protocol. But there still are some disadvantages and some issues that could be improved.

3.1 Chord's Disadvantages

Chord's lookup mechanism is assymmetric which means that a lookup from a node n to a node p could take a different number of hops than vice versa. Chord lookups only move clockwise. That is a disadvantage in huge Chord rings if a node n wants to lookup its predecessors predecessor. Then the query is routed around almost the complete ring whereas it would have taken one or two hops if the query had been routed to the other direction. Another disadvantage in this assymmetric aspect is that updating finger tables requires some procedure steps because there is no symmetry in the finger tables.

Some possibilities to solve those problems are offered by the protocol **S-Chord**. S-Chord has some symmetries like routing entry symmetry, which means that if node n has an entry to node p then p has an entry to n. This property makes it possible to delete routing entries during node leaves without correction by our stabilization procedures. This is called "in-place notification". It also means that lookups from node n to node p take very likely equal number of hops. That is called routing cost symmetry because the paths are not supposed to be identical (this would be routing symmetry).

3.2 Chord and Search engines

Now one could suppose chord being suitable for a search engine. It might be a disappointment but Chord itself unfortunately could not be a search engine. First, Chord alone is a protocol which has to be implemented in an application to obtain its beautiful properties. Another point is, that Chord supports the only operation to map a given key onto a node. Thus, Chord only supports "exact match" and cannot handle queries similar to one or more keys. It is not able to return a ranking, just one specific node that is determined by the key. It is important to have very specific information about what should be found. Due to that it becomes obvious that Chord does not really fit as a search engine protocol because generally we search with similarities and hope to retrieve more than one hit.

Well, there exist proposals to extend Chord in applications such, that it is able to build meta data search engines, like in the proposal of Marcelo M. Vanzin.
The idea is that there are meta data categories whose hash values are mapped onto nodes. Nodes maintain tables that map the meta data values to document identifiers which leads to a lookup result consisting of the identifier of the matching file which itself could be looked up in Chord's standard mechanism.

That leads to the conclusion that Chord has considerable extension possibilities.

Chapter 4

Applications using Chord

Though Chord is not directly suitable for search engines there are several applications that could use Chords mechanism.

DNS
DNS is used to lookup IP addresses with information about host names. That is a good job for Chord: hash values of host names can be keys and the hashed IP addresses can be values.
Generally, servers are used for DNS. Disadvatageous is that everything depends on this single server and if this server failed, no lookups would be possible. Chord, as a decentralized system, does not depend on a single server, which means that it is more robust. Another advantage is, that routing tables are updated automatically in Chord whereas those have to be updated manually on a server.

Cooperative Mirroring
If there are several providers of the same content, Chord could be used to spread load evenly over all machines by mapping specific data blocks onto hosts.

Time-shared Storage
If some clients in a network have important data that should be always available but the cannot be online all the time, the data could be offered to other peers in the system while the machines are online in order to have the data available on the other machine(s) during own downtime. In return they can offer other machines data while they are online and other machines are offline and so on. Chord can map the data onto a living node to a given time.

Chapter 5

Summary

- Chord provides:
  - Simplicity
  - Scalability
  - Provable correctness
  - Provable performance
- Any lookup requires O(log N) messages in a N node network
- Even in unstable systems lookups are correct
- Different possibilities for developers to implement the Chord protocol
  $\Rightarrow$ extendable in applications

Bibliography

[1] http://www.pdos.lcs.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf
(by Ian Stoica et al., last visited: 11/15/2004)

[2] http://www.info.ucl.ac.be/~valentin/s_chord/pdpta03.ps
(by V.A. Mesaros, B. Carton, P. van Roy, last visited: 11/15/2004)

[3] http://www.cs.utexas.edu/users/plaxton/c/395t/projects/writeups/vanzin.pdf
(by Marcelo N. Vancin, last visited: 11/15/2004)