

Kelips & Chord P2P DHT Systems

Rishi Bhardwaj

CS523

April 7, 2006

Overview

- Chord
 - Description of the Chord Algorithm
 - $O(\log N)$ lookup cost
 - $O(\log N)$ memory requirements
- Kelips
 - Description of the Kelips system
 - $O(1)$ lookup cost
 - $O(\sqrt{N})$ memory requirements
- Comparison

What are P2P Systems?

- Large self organizing Distributed Systems
- Without any centralized control or hierarchy
- Software running at each node is equivalent in functionality

P2P Systems

Efficient Location of Data Items?

- The core problem for P2P Systems
 - How do we know which node stores the data we are looking for?
- An obvious (inefficient) solution:
 - Store “data : node” translation entry for all data in all the nodes. Grossly inefficient
- Divide the routing information among the nodes, Distributed Hash Tables (DHT).

Chord Protocol

- Provides only one operation: given a key, it maps the key onto a node identifier.
- Node identifier = SHA-1 hash(IP address)
- Key = SHA - 1 hash(Data Identifier)
- Keys and Node identifiers both lie between 0 ... $(2^m)-1$

Who 'stores' Data with key k ?

- Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space.
- This node is known as $\text{succ}(k)$

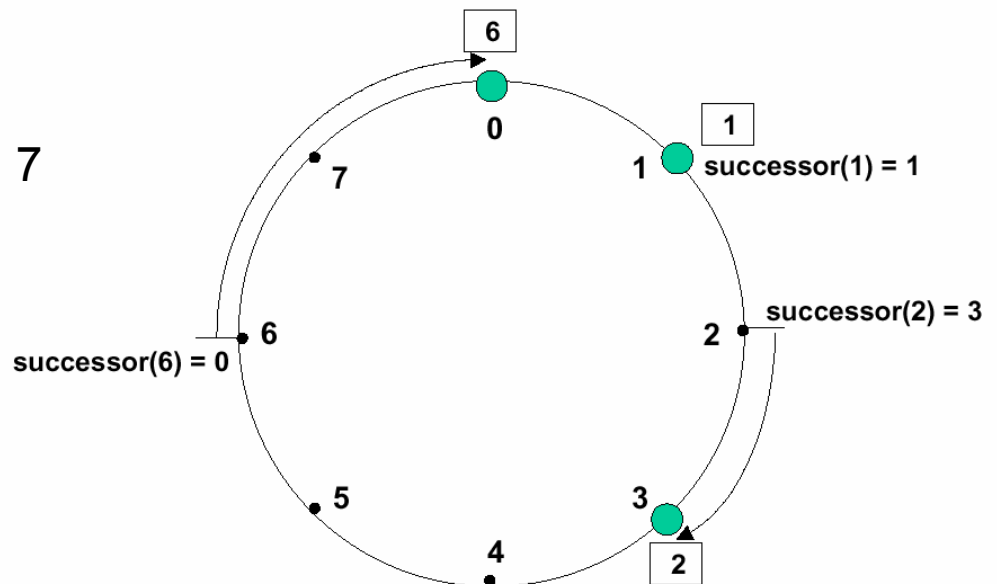
$m = 3$, possible node identifier $0 \dots 7$

Three active nodes, $0, 1, 3$

Node 0 is responsible for key 6

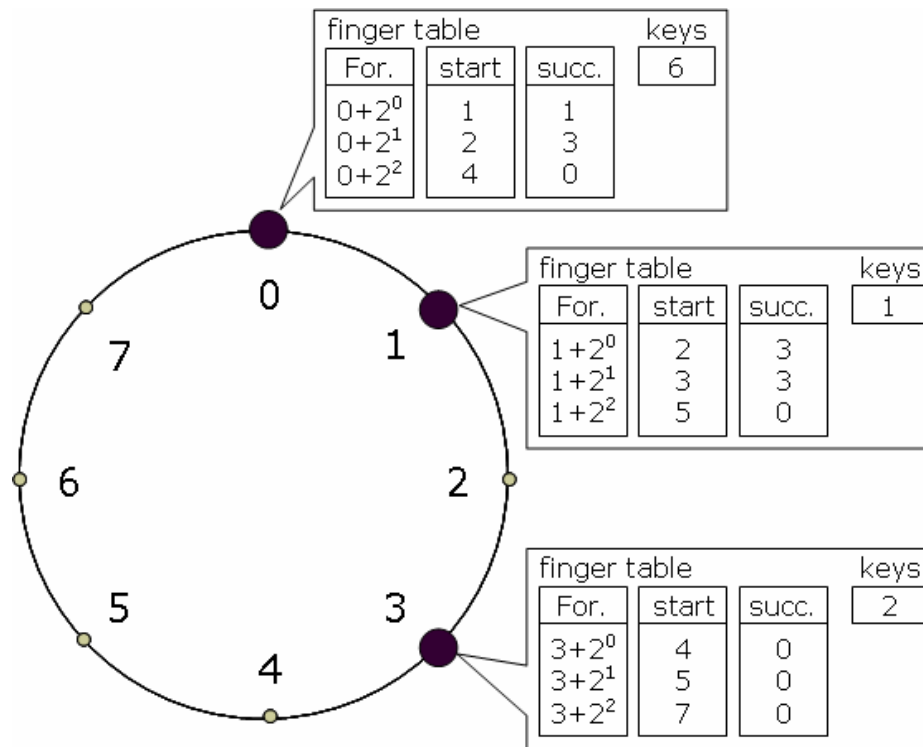
Node 1 is responsible for key 1

Node 3 is responsible for key 2

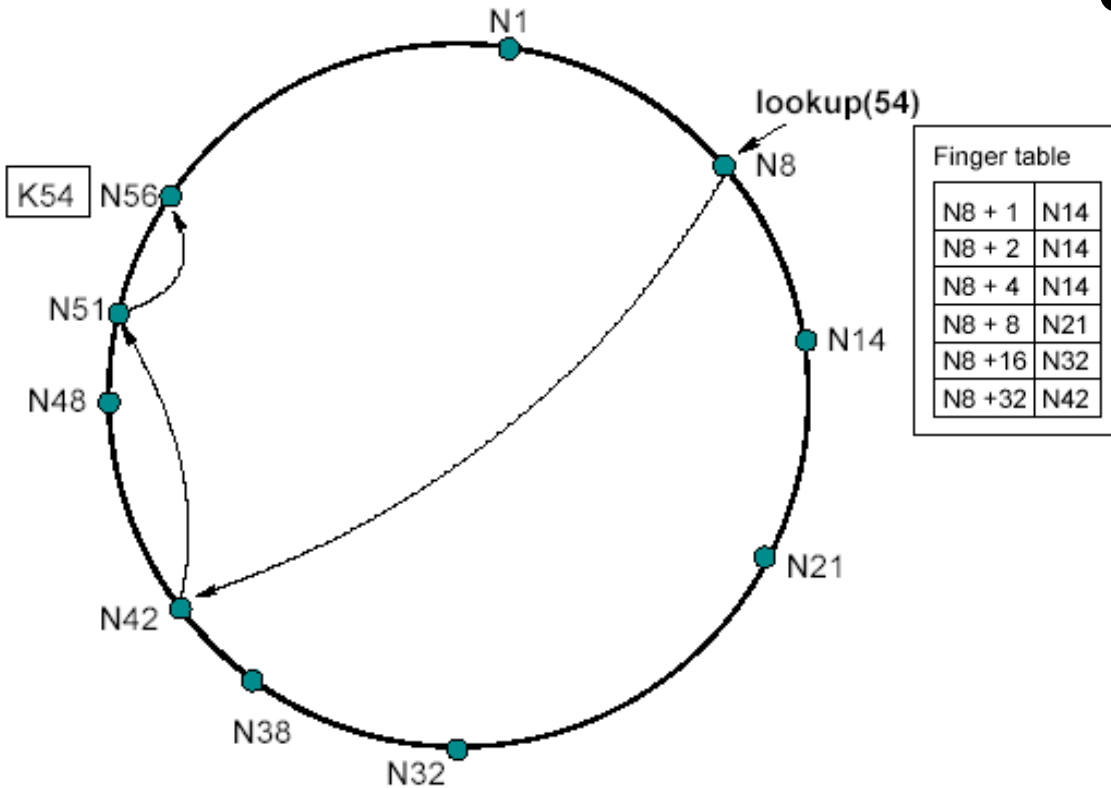


Distributed Routing Table

- Finger table contains (at most) m entries
- The i th entry at node n contains the identity of $\text{succ}(n + 2^{i-1})$, $1 \leq i \leq m$



Routing



N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 + 16	N32
N8 + 32	N42

N42 + 1	N48
N42 + 2	N48
N42 + 4	N48
N42 + 8	N51
N42 + 16	N1
N42 + 32	N14

```
// ask node n to find id's successor
n.find_successor(id)
  n' = find_predecessor(id);
  return n'.successor;
```

```
// ask node n to find id's predecessor
n.find_predecessor(id)
  n' = n;
  while (id ∉ (n', n'.successor])
    n' = n'.closest_preceding_finger(id);
  return n';
```

```
// return closest finger preceding id
n.closest_preceding_finger(id)
  for i = m downto 1
    if (finger[i].node ∈ (n, id))
      return finger[i].node;
  return n;
```

Summarizing Chord

- Each node maintains $O(\log N)$ routing table entries
- Lookup takes $O(\log N)$ messages (Avg: $\frac{1}{2}(\log N)$)
- On Average each node is responsible for K/N keys
- Joining of new node takes at most $O(\log N * \log N)$ messages to re-establish routing information

Kelips

- Provides $O(1)$ lookup mechanism to find the Node responsible for a File.
- Consists of k virtual *affinity groups* ($0 \dots k-1$)
- Hash of node's IP (consistent hashing) determines its group $[0, k-1]$
- Hash of File Name gives the *affinity group* where the file is stored.

Kelips: Core Design

- A Node stores three tables:
 - *Affinity Group View*: Info about its own grp members: IP, RTT, heartbeat count
 - *Contacts*: For each of the other grps, info of some small set of nodes in the foreign affinity group: IP, RTT, heartbeat count
 - *Filetuples*: For each file belonging to its own affinity group, info about the nodes storing the file

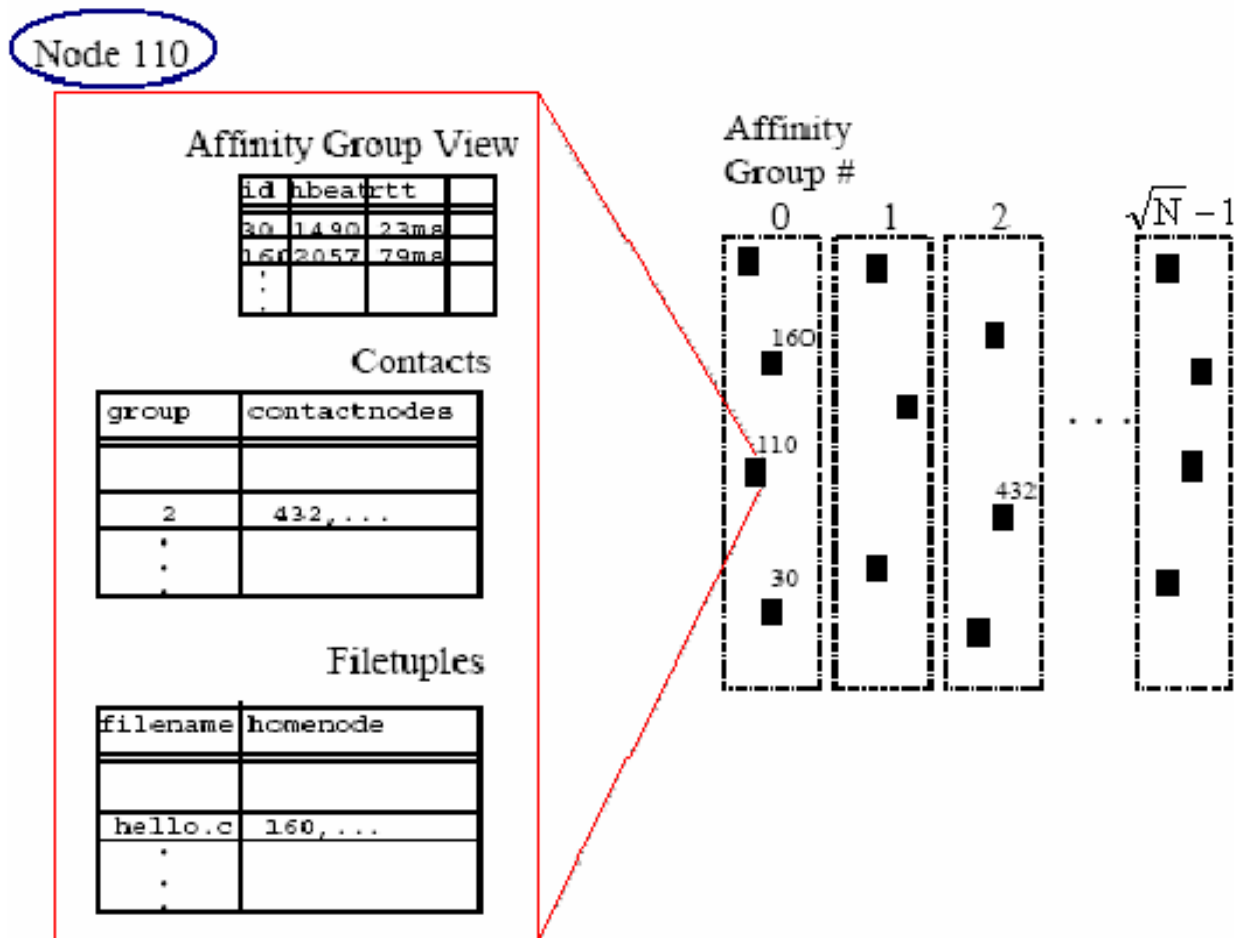
Kelips: Core Design

- Hash function: **SHA-1**
- Memory Usage at a node:

$$(n/k) + c(k-1) + (F/k)$$

- Lookup queries return the location of the file within **$O(1)$ time** and message complexity

- Memory utilization is minimized at $k = O(\sqrt{N})$
- $F = O(N)$, Util = $O(\sqrt{N})$



The above slide is borrowed from Professor I. Gupta's CS598 class.

File Look Up

- Querying node maps file name to the appropriate affinity group
- Sends lookup request to the topologically closest '*Contact*' node from that group.
- Recvd lookup request resolved by searching among *filetuple* table.
- $O(1)$ time and message complexity

Background Overhead

- Existing *View*, *Contacts*, *Filetuple* entries refreshed periodically.
- Hearbeat updates originate at the responsible node.
- Disseminated through P2P gossip style protocol. Multicasting to group of nodes which further multicast to group of nodes...
- Latency in single group ($\log N'$) and across groups $(\log N) * (\log N')$

Comparison: Chord & Kelips

	Chord	Kelips
Lookup	$O(\log N)$	$O(1)$
Memory	$O(\log N)$	$O(\sqrt{N})$
Topological Routing	Not Possible	Possible
Node Join Msgs Sent	$O(\log N * \log N)$	Done with the regular refresh messages.
Updation, Refresh	Low overheads for updation, refresh	Higher latency & overheads.

Comparison

- Kelips, good for medium size networks.
 - 100,000 nodes over 317 affinity groups, 10 million files entails 1.93 MB of soft state.
 - 100,000 probably is an upper limit.
- Chord can handle massive P2P networks 10 to 100(maybe even more) times bigger than Kelips networks.

References

- I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. *Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications*. In Proc. of ACM SIGCOMM '01, pages 149--160, San Diego, CA, August 2001.
- I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. *Kelips: Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead*. In Proc. of the 2nd International Workshop on Peer-toPeer Systems (IPTPS '03), 2003.

Questions?

Thank You