A Framework for Peer-to-Peer Lookup Services Based on k-ary Search

Sameh El-Ansary, Luc Onana Alima, Per Brand, Seif Haridi

Michalis Polychronakis

October, 2004
What is the Problem?

- How to efficiently locate the node that stores a particular item

- Fundamental operation of P2P applications
Roadmap

- Introduction
- Chord basics
- Scalable lookup in Chord
- k-ary search
- Discussion
Lookup Services

Main issues: **Scalability** and **Hit Guarantee**

- **Napster**
  - Centralized directory
  - Scalability: ✗
  - Hit Guarantee: ✓

- **Gnutella**
  - Flooding
  - Scalability: ✗
  - Hit Guarantee: ✗
Distributed Hash Table

- A hash table partitioned among a number of nodes
- No central control

Issues
- How to distribute the table
- How to locate items
Chord

- Distributed P2P Lookup Protocol
- Given a key, it maps a key onto a node
- Distributed routing table
  - Queries require communication with a few other nodes
- Key features
  - Simplicity
  - Provable correctness
  - Provable performance
Consistent Hashing

- *Like* normal hashing, assigns items to buckets so that each bucket receives roughly the same number of items
  - All Chord nodes receive roughly the same number of keys
- *Unlike* normal hashing, a small change in the bucket set does not induce a total remapping of items to buckets
  - Little movement of keys when nodes join or leave the system
Consistent Hashing in Chord

- A hash function assigns an \( m \)-bit identifier to each node and to each key
  - \( \text{hash}(139.91.70.63) \rightarrow \text{node identifier} \)
  - \( \text{hash(terlengas_strofes.mp3)} \rightarrow \text{key identifier} \)
- Identifiers are ordered in an identifier circle
- A key is stored at its successor: the first node with an identifier equal or greater than the identifier of the key
m = 3 bits

nodes 0, 1, and 3

successor(k) is the first node clockwise from k
successor(0) = 0
successor(1) = 1
successor(2) = 3
successor(3) = 3
successor(4) = 0
successor(5) = 0
successor(6) = 0
successor(7) = 0
Simple Key Location

- Each node is aware only of its successor node on the circle
- Queries are passed around the circle until they encounter a node that succeeds the given identifier
- Hit guarantee
- Inefficient
  - $O(N)$ steps
Scalable Key Location

- Chord maintains these successor pointers
  - Ensures that lookups are resolved correctly

- Additional routing information at each node
  - Not essential for correctness, but accelerates lookup

- **Finger Table**
  - A small routing table at each node
  - Stores information about only a few other nodes
  - Each node knows more about nodes following it closely on the identifier circle, than about nodes far away
Finger Table

- $O(\log N)$ entries
- Finger $i$ of node $n$ points to the successor of identifier $n+2^i$
- If a node does not know the successor of a key $k$, it passes the query to the node whose identifier in the finger table precedes $k$ most immediately
Chord Lookup

\[ \text{Where is key 53?} \]

- \( O(\log N) \) hops

<table>
<thead>
<tr>
<th>idntf.</th>
<th>intrvl.</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>50+1</td>
<td>51..51</td>
<td>55</td>
</tr>
<tr>
<td>50+2</td>
<td>52..53</td>
<td>55</td>
</tr>
<tr>
<td>50+4</td>
<td>54..57</td>
<td>55</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>41+8</td>
<td>49..56</td>
<td>50</td>
</tr>
<tr>
<td>41+16</td>
<td>57.8</td>
<td>0</td>
</tr>
<tr>
<td>41+32</td>
<td>9.41</td>
<td>13</td>
</tr>
</tbody>
</table>
m=4
2^4 nodes

The finger pointers at repeatedly doubling distances cause each iteration to halve the distance to the target identifier.
Lookup Services as k-ary Search

$k=4$

- Lookup: $O(\log_k N)$
- Finger table size: $(k-1)\log_k N$
  - $(4-1)\log_4 16 = 6$
**k-ary Search for Improving Chord**

- We can choose a suitable $k$ to improve the lookup length of Chord to a desired value $H$
  - $H = \log_k N$
  - Routing table entries $R = (k-1)\log_k N$

- Chord authors suggested the Chord(d) generalization
  - place fingers at intervals that are powers of $1+1/d$ instead of 2
  - $H = \log_{1+d} N$
  - $R = \log N / \log(1+1/d)$

- Comparison of the two methods
  - Let $x = 1 + d$ and take $k = x = 4$ (now both have the same $H$)
At last! Some Numbers

- $k = x = 4$

- 38% smaller number of routing table entries
Discussion 1: Chord

Pros

- Hit Guarantee
- $O(\log N)$ lookup length
- $O(\log N)$ per node state
- Minimum change-hands of keys during node joins and leaves
- Retains correctness with high probability under very dynamic joins and failures

Open issues

- Topological sensitivity/congestion-awareness
- Authentication
- Malicious Chord table information
- Anonymity
- Support for non-exact match search is difficult
Discussion2: k-ary Search

We have learned that there is a tradeoff in Chord between the number of hops for lookups and the finger table size.

How much to turn the knob?

- For N=1,000,000?
- For N=1,000,000,000?
- For N=1,000,000,000,000?

...