CS551
Distributed Hash Tables
Structured Systems
Bill Cheng

http://merlot.usc.edu/cs551-f12
CS551
Chord
[Stoica01a]

Bill Cheng

http://merlot.usc.edu/cs551-f12
Chord

- A structured peer-to-peer system
- Map key to value
- Emphasis on good algorithmic performance
  - uses consistent hashing
  - $O(\log N)$ route storage, $O(\log N)$ lookup cost, $O(\log^2 N)$ cost to join/leave
  - vs. FreeNet w/emphasis on anonymity
- Easy if static, but must deal with node arrivals and departures
Compare Search in Several Peer-to-Peer Systems

- Napster: central search engine
- Freenet: search towards keys, but no guarantees
- Chord:
  - map keys to linear search space
  - keep pointers (fingers) into exponential places around space
  - probabilistic (depends on hashing)
Hashing Nodes and Data

Nodes hash IP addresses to key space
- because this hashing is random, can expect nodes to be evenly distributed in key space

Store data in the successor of the data item’s key

Property:
- If each node maintains successor,
- ... can find any data item
Hashing Nodes and Data

- Nodes hash IP addresses to key space
  - because this hashing is random, can expect nodes to be evenly distributed in key space

- Store data in the successor of the data item’s key

- Property:
  - If each node maintains successor,
  - ... can find any data item

- Nodes have a successor pointer
  - but O(n) performance
Improving Search Performance with Finger Tables

- Finger tables enable logarithmic lookup
- i-th finger of node x is successor of $x + 2^{i-1}$
- at each step, we halve the remaining distance (in key space) to the target

Challenge: maintaining finger tables!
Finger tables enable logarithmic lookup

- i-th finger of node x is successor of $x + 2^{i-1}$
- at each step, we halve the remaining distance (in key space) to the target

Ex: look for key y
Improving Search Performance with Finger Tables (Cont...)

Finger tables enable logarithmic lookup

- i-th finger of node x is successor of \( x + 2^{i-1} \)
- at each step, we halve the remaining distance (in key space) to the target

Ex: look for key \( y \)

- case 1: \( y \) is just beyond \( x + 2^{159} \)
- forward to successor(\( x + 2^{159} \))
- way more than half the distance to \( y \)
Improving Search Performance with Finger Tables (Cont...)

> Finger tables enable logarithmic lookup
> 1. i-th finger of node x is successor of $x + 2^{i-1}$
> 2. at each step, we halve the remaining distance (in key space) to the target

- Ex: look for key y
  - case 2: y is just inside $x + 2^{159}$
  - forward to successor($x + 2^{158}$)
  - a little over half the distance to y
Improving Search Performance with Finger Tables (Cont...)

- Finger tables enable logarithmic lookup
  - i-th finger of node \( x \) is successor of \( x + 2^{i-1} \)
  - at each step, we halve the remaining distance (in key space) to the target

- Ex: look for key \( y \)
  - case 3: \( y \) is just beyond \( x + 2^{158} \)
  - forward to successor(\( x + 2^{158} \))
  - way more than half the distance to \( y \)
  - and so on...
Finger Tables Example

- The i-th finger of node x is successor of x + 2^{i-1}

**Example:**

- **finger[1]**: start = 2, interval = [2, 3)
  - keys: | start | interval | succ.
  - 1 | [1,2) | 1
  - 2 | [2,4) | 3
  - 4 | [4,0) | 0

- **finger[2]**: start = 3, interval = [3, 5)
  - keys: | start | interval | succ.
  - 2 | [2,3) | 3
  - 3 | [3,5) | 3
  - 5 | [5,1) | 0

- **finger[3]**: start = 5, interval = [5, 0)
  - keys: | start | interval | succ.
  - 4 | [4,5) | 0
  - 5 | [5,7) | 0
  - 7 | [7,3) | 0
Node Joins

- Must keep successors and finger table current
  - can always fall back on them to find a key

- Use successors for *correctness*

- Use finger table for *performance*
  - must update it, but can tolerate temporary errors

- Keep successor and *predecessor* so we can update our neighbors

- Key observation: can find successors and fingers by doing a lookup on the existing Chord ring!
Finding Predecessor and Successor

node.\texttt{find\_successor}(key)
\begin{verbatim}
    n = \texttt{find\_predecessor}(key);
    return n.successor;
\end{verbatim}

node.\texttt{find\_predecessor}(key)
\begin{verbatim}
    n = node;
    while (key \notin (n,n.successor])
        n = n.\texttt{closest\_preceding\_finger}(key);
    return n;
\end{verbatim}

node.\texttt{closest\_preceding\_finger}(key)
\begin{verbatim}
    for (i=m; i > 0; i--)
        if (finger[i].node \in (node,key))
            return finger[i].node;
    return node;
\end{verbatim}
Join Example

when new node enters, it establishes its successor and predecessor and then builds its finger table, and moves any keys it now "owns"
Robustness

**Stabilization** algorithm to confirm ring is correct
- every 30s, ask successor for its predecessor
  - fix your own successor based on this
  - successor fixes its predecessor if necessary
- also, pick and verify a random finger table entry
  - rebuild finger table entries this way
  - important observation: finger tables can be incorrect for some time (between network sizes of N and 2N)

Dealing with unexpected failures:
- keep successor list of r successors
- can use these to replicate data
Applications

File Systems

Multicast and Anycast (using rendezvous)
Chord Performance

- Performance dominated by *lookup* cost
  - how long does it take to get to the node that stores a key?

- Chord promises few $O(\log N)$ hops on the overlay
  - but, on the physical network, this can be quite far
    - this is often the problem with *overlay networks*