What’s a DHT?

- Distributed Hash Table
  - Peer-to-peer algorithm to offering put/get interface
  - Associative map for peer-to-peer applications
- More generally, provide lookup functionality
  - Map application-provided hash values to nodes
  - (Just as local hash tables map hashes to memory locs.)
  - Put/get then constructed above lookup
- Many proposed applications
  - File sharing, end-system multicast, aggregation trees

How DHTs Work

How do we ensure the put and the get find the same machine?

Step 1: Partition Key Space

- Each node in DHT will store some k,v pairs
- Given a key space K, e.g. [0, 2^160):
  - Choose an identifier for each node, id <= K, uniformly at random
  - A pair k,v is stored at the node whose identifier is closest to k

Step 2: Build Overlay Network

- Each node has two sets of neighbors
  - Immediate neighbors in the key space
    - Important for correctness
  - Long-hop neighbors
    - Allow puts/gets in O(log n) hops
Step 3: Route Puts/Gets Thru Overlay

- Route greedily, always making progress

![Diagram showing route](image)

How Does Lookup Work?

- Assign IDs to nodes
- Map hash values to node with closest ID
- Leaf set is successors and predecessors
- All that's needed for correctness
- Routing table matches successively longer prefixes
- Allows efficient lookups

![Diagram showing lookup](image)

How Bad is Churn in Real Systems?

<table>
<thead>
<tr>
<th>Authors</th>
<th>Systems Observed</th>
<th>Session Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>S6601</td>
<td>Gnutella, Napster</td>
<td>50% - 60 minutes</td>
</tr>
<tr>
<td>C1L02</td>
<td>Gnutella, Napster</td>
<td>31% - 10 minutes</td>
</tr>
<tr>
<td>SW02</td>
<td>FastTrack</td>
<td>50% - 1 minute</td>
</tr>
<tr>
<td>BSV03</td>
<td>Overnet</td>
<td>50% - 60 minutes</td>
</tr>
<tr>
<td>60:503</td>
<td>Kazaa</td>
<td>50% - 2.4 minutes</td>
</tr>
</tbody>
</table>

An hour is an incredibly short MTTF!

Note on CPS 196, Spring 2006

- We did not cover any of the following material on managing DHTs under churn.

Routing Around Failures

- Under churn, neighbors may have failed
- To detect failures, acknowledge each hop

![Diagram showing routing around failures](image)

Routing Around Failures

- If we don't receive an ACK, resend through different neighbor

![Diagram showing routing around failures](image)
Computing Good Timeouts

- Must compute timeouts carefully
  - If too long, increase put/get latency
  - If too short, get message explosion

Calculating Good Timeouts

- Use TCP-style timers
  - Keep past history of latencies
  - Use this to compute timeouts for new requests
- Works fine for recursive lookups
  - Only talk to neighbors, so history small, current

- In iterative lookups, source directs entire lookup
  - Must potentially have good timeout for any node

Recovering From Failures

- Can’t route around failures forever
  - Will eventually run out of neighbors
- Must also find new nodes as they join
  - Especially important if they’re our immediate predecessors or successors:
    - Old responsibility
    - New node
    - New responsibility
Recovering From Failures

- Obvious algorithm: reactive recovery
  - When a node stops sending acknowledgements, notify other neighbors of potential replacements
  - Similar techniques for arrival of new nodes

The Problem with Reactive Recovery

- What if B is alive, but network is congested?
- C still perceives a failure due to dropped ACKs
- C starts recovery, further congesting network
- More ACKs likely to be dropped
- Creates a positive feedback cycle

Periodic Recovery

- Every period, each node sends its neighbor list to each of its neighbors
- Breaks feedback loop
Periodic Recovery

- Every period, each node sends its neighbor list to each of its neighbors
- Breaks feedback loop
- Converges in logarithmic number of periods

my neighbors are A, B, D, and E