Handling Churn in a DHT

Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz
UC Berkeley and Intel Research Berkeley

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

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

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_i \in 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
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

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>SGG02</td>
<td>Gnutella, Napster</td>
<td>50% &lt; 60 minutes</td>
</tr>
<tr>
<td>CLL02</td>
<td>Gnutella, Napster</td>
<td>31% &lt; 10 minutes</td>
</tr>
<tr>
<td>SW02</td>
<td>FastTrack</td>
<td>50% &lt; 1 minute</td>
</tr>
<tr>
<td>BSV03</td>
<td>Overnet</td>
<td>50% &lt; 60 minutes</td>
</tr>
<tr>
<td>GDS03</td>
<td>Kazaa</td>
<td>50% &lt; 2.4 minutes</td>
</tr>
</tbody>
</table>

An hour is an incredibly short MTTF!

Can DHTs Handle Churn?

A Simple Test

- Start 1,000 DHT processes on a 80-CPU cluster
  - Real DHT code, emulated wide-area network
  - Models cross traffic and packet loss
- Churn nodes at some rate
- Every 10 seconds, each machine asks:
  “Which machine is responsible for key k?”
  - Use several machines per key to check consistency
  - Log results, process them after test

Test Results

- In Tapestry (the OceanStore DHT), overlay partitions
  - Leads to very high level of inconsistencies
  - Worked great in simulations, but not on more realistic network
- And the problem isn’t limited to Tapestry:
  - FreePastry
  - MIT Chord

The Bamboo DHT

- Forget about comparing Chord-Pastry-Tapestry
  - Too many differing factors
  - Hard to isolate effects of any one feature
- Instead, implement a new DHT called Bamboo
  - Same overlay structure as Pastry
  - Implements many of the features of other DHTs
  - Allows testing of individual features independently

How Bamboo Handles Churn (Overview)

1. Chooses neighbors for network proximity
   - Minimizes routing latency in non-failure case
2. Routes around suspected failures quickly
   - Abnormal latencies indicate failure or congestion
   - Route around them before we can tell difference
3. Recovers failed neighbors periodically
   - Keeps network load independent of churn rate
   - Prevents overlay-induced positive feedback cycles
Routing Around Failures

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

Routing Around Failures

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

Computing Good Timeouts

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

Computing Good Timeouts

- Chord errs on the side of caution
  - Very stable, but gives long lookup latencies

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

Computing Good Timeouts

- Keep past history of latencies
  - Exponentially weighted mean, variance
- Use to compute timeouts for new requests
  - timeout = mean + 4 × variance
- When a timeout occurs
  - Mark node “possibly down”: don’t use for now
  - Re-route through alternate neighbor
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

  old responsibility

  new node

  new responsibility

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
The Problem with Reactive Recovery

• What if B is alive, but network is congested?
• This was the problem with Pastry
  – Combined with poor congestion control, causes network to partition under heavy churn

Periodic Recovery

• Every period, each node sends its neighbor list to each of its neighbors

Periodic Recovery Performance

• Reactive recovery expensive under churn
• Excess bandwidth use leads to long latencies
Virtual Coordinates

- Machine learning algorithm to estimate latencies
  - Distance between coords. proportional to latency
  - Called Vivaldi; used by MIT Chord implementation
- Compare with TCP-style under recursive routing
  - Insight into cost of iterative routing due to timeouts

Proximity Neighbor Selection (PNS)

- For each neighbor, may be many candidates
  - Choosing closest with right prefix called PNS
  - One of the most researched areas in DHTs
  - Can we achieve good PNS under churn?
- Remember:
  - leaf set for correctness
  - routing table for efficiency?
- Insight: extend this philosophy
  - Any routing table gives O(log N) lookup hops
  - Treat PNS as an optimization only
  - Find close neighbors by simple random sampling

PNS Results

(very abbreviated—see paper for more)

- Random sampling almost as good as everything else
  - 24% latency improvement free
  - 42% improvement for 40% more b.w.
  - Compare to 68%-84% improvement by using good timeouts
- Other algorithms more complicated, not much better

Conclusions/Recommendations

- Avoid positive feedback cycles in recovery
  - Beware of “false suspicions of failure”
  - Recover periodically rather than reactively
- Route around potential failures early
  - Don’t wait to conclude definite failure
  - TCP-style timeouts quickest for recursive routing
  - Virtual-coordinate-based timeouts not prohibitive
- PNS can be cheap and effective
  - Only need simple random sampling

For code and more information:
  bamboo-dht.org