Beyond Theory: DHTs in Practice
CS 194 - Distributed Systems
Sean C. Rhea
April 18, 2005
In collaboration with: Dennis Geels, Brighten Godfrey, Brad Karp, John Kubiatchowycz, Sylvia Ratnasamy, Timothy Roscoe, Scott Shenker, Ion Stoica, and Harlan Yu

Talk Outline
• Bamboo: a churn-resilient DHT
  – Churn resilience at the lookup layer [USENIX’04]
  – Churn resilience at the storage layer
    [Cates’03], [Unpublished]
• OpenDHT: the DHT as a service
  – Finding the right interface [IPTPS’04]
  – Protecting against overuse [Under Submission]
• Future work

Making DHTs Robust:
The Problem of Membership Churn
• In a system with 1,000s of machines, some machines failing / recovering at all times
• This process is called *churn*
• Without repair, quality of overlay network degrades over time
• A significant problem deployed peer-to-peer systems

How Bad is Churn in Real Systems?

<table>
<thead>
<tr>
<th>Author(s)</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!

Refresher: DHT Lookup/Routing

Put and get *must* find the same machine

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

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

Bamboo Basics: 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 $k$

Bamboo Basics: 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

Bamboo Basics: Route Puts/Gets Thru Overlay

- Route greedily, always making progress

get($k$)
Routing Around Failures

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

Computing Good Timeouts

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

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

Timeout Estimation Performance

- Fixed 5s Timeouts
- Smart Timeouts
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:

```
  0          1          2          3          4
          ↑          ↑          ↑          ↑
  old      new      old      new
  responsibility
```

```
  0          1          2          3          4
          ↑          ↑          ↑          ↑
  new      old      new      old          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

```
  0          1          2          3          4
          ↑          ↑          ↑          ↑
  B failed, use D  B failed, use A
```

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

```
  0          1          2          3          4
          ↑          ↑          ↑          ↑
  B failed, use D  B failed, use A
```

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

```
  0          1          2          3          4
          ↑          ↑          ↑          ↑
  B failed, use D  B failed, use A
```

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

```
  0          1          2          3          4
          ↑          ↑          ↑          ↑
  B failed, use D  B failed, use A
```
Periodic Recovery

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

Periodic Recovery Performance

- Reactive recovery expensive under churn
- Excess bandwidth use leads to long latencies

Proximity Neighbor Selection (PNS)

- For each neighbor, may be many candidates
  - Choosing closest with right prefix called PNS
How Important is PNS?

• Only need leaf set for correctness
  – Must know predecessor and successor to determine what keys a node is responsible for
• Any filled routing table gives efficient lookups
  – Need one neighbor that shares no prefix, one that shares one bit, etc., but that’s all
• Insight: treat PNS as an optimization only
  – Find initial neighbor set using lookup

PNS by Random Sampling

• Already looking for new neighbors periodically
  – Because doing periodic recovery
• Can use results for random sampling
  – Every period, find potential replacement with lookup
  – Compare latency with existing neighbor
  – If better, swap

Talk Outline

• Bamboo: a churn-resilient DHT
  – Churn resilience at the lookup layer
  – Churn resilience at the storage layer
• OpenDHT: the DHT as a service
  – Finding the right interface
  – Protecting against overuse
• Future work
A Reliable Storage Layer

- Don’t just want to do lookup, also want to do
  - DHash’s put/get: store (key, value) pairs
  - Tapestry’s publish/route: store (key, pointer) pairs
- Problem statement:
  *Keep data/pointers available despite churn*

Why Storage Is Tricky

- The claim: DHT replicates within leaf set
  - A pair \((k,v)\) is stored by the node closest to \(k\) and replicated on its successor and predecessor
- Why is this hard?

DHTs Meet Epidemics

- Candidate Algorithm
  - For each \((k,v)\) stored locally, compute SHA\((k,v)\)
  - Every period, pick a random leaf set neighbor
  - Ask neighbor for all its hashes
  - For each unrecognized hash, ask for key and value
- This is an epidemic algorithm
  - All \(m\) members will have all \((k,v)\) in \(\log m\) periods
  - But as is, the cost is \(O(C)\), where \(C = \text{disk size} \) (although the constant is pretty small)

Merkle Trees

- Merkle trees are an efficient summary technique
  - Interior nodes are the secure hashes of their children
  - E.g., \(I = \text{SHA}(A,B)\), \(N = \text{SHA}(K,L)\), etc.

Using Merkle Trees as Summaries

- Improvement: use Merkle tree to summarize keys
  - B gets tree root from A, if same as local root, done
  - Otherwise, recurse down tree to find difference
Using Merkle Trees as Summaries

• Improvement: use Merkle tree to summarize keys
  – B gets tree root from A, if same as local root, done
  – Otherwise, recurse down tree to find difference

• New cost is $O(d \log C)$
  – $d =$ number of differences, $C =$ size of disk

• Still too costly:
  – If A is down for an hour, then comes back, changes will be randomly scattered throughout tree

• Solution: order values by time instead of hash
  – Localizes values to one side of tree

PlanetLab Deployment

• Been running Bamboo / OpenDHT on PlanetLab since April 2004
• Constantly run a put/get test
  – Every second, put a value (with a TTL)
  – DHT stores 8 replicas of each value
  – Every second, get some previously put value
    (that hasn’t expired)
• Tests both routing correctness and replication algorithms (latter not discussed here)

Excellent Availability

• Only 28 of 7 million values lost in 3 months
  – Where “lost” means unavailable for a full hour
• On Feb. 7, 2005, lost 60/190 nodes in 15 minutes to PL kernel bug, only lost one value

Talk Outline

• Bamboo: a churn-resilient DHT
  – Churn resilience at the lookup layer
  – Churn resilience at the storage layer
• OpenDHT: the DHT as a service
  – Finding the right interface
  – Protecting against overuse
• Future work
A Small Sample of DHT Applications

- Distributed Storage Systems
  - CFS, HiveCache, PAST, Pastiche, OceanStore, Pond
- Content Distribution Networks / Web Caches
  - Bslash, Coral, Squirrel
- Indexing / Naming Systems
  - Chord-DNS, CoDoNS, DOA, SFR
- Internet Query Processors
  - Catalogs, PIER
- Communication Systems
  - Bayeux, i3, MCAN, SplitStream

Questions:

- How many DHTs will there be?
- Can all applications share one DHT?

Benefits of Sharing a DHT

- Amortizes costs across applications
  - Maintenance bandwidth, connection state, etc.
- Facilitates “bootstrapping” of new applications
  - Working infrastructure already in place
- Allows for statistical multiplexing of resources
  - Takes advantage of spare storage and bandwidth
- Facilitates upgrading existing applications
  - “Share” DHT between application versions

Challenges in Sharing a DHT

- Robustness
  - Must be available 24/7
- Shared Interface Design
  - Should be general, yet easy to use
- Resource Allocation
  - Must protect against malicious/over-eager users
- Economics
  - What incentives are there to provide resources?
The DHT as a Service

OpenDHT Clients

What is this interface?

What does this node do with it?

Challenges:
1. Distribution
2. Security

How are DHTs Used?

1. Storage
   - CFS, UsenetDHT, PKI, etc.
2. Rendezvous
   - Simple: Chat, Instant Messenger
   - Load balanced: i3
   - Multicast: RSS Aggregation, White Board
   - Anycast: Tapestry, Coral

What about put/get?

- Works easily for storage applications
- Easy to share
  - No upcalls, so no code distribution or security complications
- But does it work for rendezvous?
  - Chat? Sure: put(my-name, my-IP)
  - What about the others?
Recursive Distributed Rendezvous

- Idea: prove an equivalence between lookup and put/get
  - We know we can implement put/get on lookup
  - Can we implement lookup on put/get?
- It turns out we can
  - Algorithm is called Recursive Distributed Rendezvous (ReDiR)

ReDiR

- Goal: Implement two functions using put/get:
  - join(namespace, node)
  - node = lookup(namespace, identifier)

ReDiR

- Goal: Implement two functions using put/get:
  - join(namespace, node)
  - node = lookup(namespace, identifier)
ReDiR

• Join cost:
  – Worst case: $O(\log n)$ puts and gets
  – Average case: $O(1)$ puts and gets

• Goal: Implement two functions using put/get:
  – $\text{join}(\text{namespace}, \text{node})$
  – $\text{node} = \text{lookup}(\text{namespace}, \text{identifier})$

ReDiR Performance
(On PlanetLab)

• Lookup cost:
  – Worst case: $O(\log n)$ gets
  – Average case: $O(1)$ gets
OpenDHT Service Model

- **Storage Applications:**
  - Just use put/get

- **Rendezvous Applications:**
  - You provide the nodes
  - We provide cheap, scalable rendezvous

Talk Outline

- **Bamboo: a churn-resilient DHT**
  - Churn resilience at the lookup layer
  - Churn resilience at the storage layer

- **OpenDHT: the DHT as a service**
  - Finding the right interface
  - Protecting against overuse

- Future work

Protecting Against Overuse

- Must protect system resources against overuse
  - Resources include network, CPU, and disk
  - Network and CPU straightforward
  - Disk harder: usage persists long after requests

- Hard to distinguish malice from eager usage
  - Don’t want to hurt eager users if utilization low

- Number of active users changes over time
  - Quotas are inappropriate

Fair Storage Allocation

- Our solution: give each client a fair share
  - Will define “fairness” in a few slides

- Limits strength of malicious clients
  - Only as powerful as they are numerous

- Protect storage on each DHT node separately
  - Must protect each subrange of the key space
  - Rewards clients that balance their key choices

The Problem of Starvation

- Fair shares change over time
  - Decrease as system load increases

```
Client 1 arrives  Client 2 arrives  Client 3 arrives
fills 50% of disk  fills 40% of disk  max share = 10%
time
```

Preventing Starvation

- Simple fix: add time-to-live (TTL) to puts
  - put (key, value) → put (key, value, ttl)
  - (A different approach is used by Palimpsest.)

- Prevents long-term starvation
  - Eventually all puts will expire
**Preventing Starvation**

- Simple fix: add time-to-live (TTL) to puts
  - put(key, value) → put(key, value, ttl)
  - (A different approach is used by Palimpsest.)
- Prevents long-term starvation
  - Eventually all puts will expire
- Can still get short term starvation

**Fair Storage Allocation**

Queue full: reject put
Per-client put queues
Not full: enqueue put
Select most under-represented
Wait until can accept without violating \( r_{\text{min}} \)
Store and send accept message to client

---

**Defining “Most Under-Represented”**

- Not just sharing disk, but disk over time
  - 1 byte put for 100s same as 100 byte put for 1s
  - So units are bytes \( \times \) seconds, call them commitments
- Equalize total commitments granted?
  - No: leads to starvation
  - A fills disk, B starts putting, A starves up to max TTL

---

**Preventing Starvation**

- Stronger condition: Be able to accept \( r_{\text{min}} \) bytes/sec new data at all times
- This is non-trivial to arrange!

\[
f(\tau) = B(t_{\text{new}}) - D(t_{\text{new}} + \tau) + r_{\text{min}} \times \tau
\]

To accept put of size \( x \) and TTL \( l \):

\[
f(\tau) + x < C \quad \text{for all } 0 \leq \tau < l
\]

- Can track the value of \( f \) efficiently with a tree
  - Leaves represent inflection points of \( f \)
  - Add put, shift time are \( O(\log n) \), \( n = \# \) of puts
Defining “Most Under-Represented”

• Instead, equalize rate of commitments granted
  – Service granted to one client depends only on others putting “at same time”

  ![Diagram](image)

  - Client A arrives
  - Client B arrives
  - B catches up with A

  A & B share available rate

  Time

Defining “Most Under-Represented”

• Instead, equalize rate of commitments granted
  – Service granted to one client depends only on others putting “at same time”

  • Mechanism inspired by Start-time Fair Queuing
    – Have virtual time, v(t)
    – Each put gets a start time $S(p_i) = S(p_i') + \text{size}(p_i) \times \text{ttl}(p_i')$
    – FST Performance
    – S(t) = \max (\text{max}(A(p_i') - \epsilon, F(p_{i-1})))$
    – $v(t) =$ maximum start time of all accepted puts

Talk Outline

• Bamboo: a churn-resilient DHT
  – Chum resilience at the lookup layer
  – Chum resilience at the storage layer

• OpenDHT: the DHT as a service
  – Finding the right interface
  – Protecting against overuse

• Future work

Future Work: Throughput

• High DHT throughput remains a challenge
  – Each put/get can be to a different destination node

• Only one existing solution (STP)
  – Assumes client’s access link is bottleneck

• Future Work: Throughput

• High DHT throughput remains a challenge
  – Each put/get can be to a different destination node

• Only one existing solution (STP)
  – Assumes client’s access link is bottleneck

• Have complete control of DHT routers
  – Can do fancy congestion control: maybe ECN?

• Have many available paths
  – Take advantage for higher throughput: mTCP?
Future Work: Upcalls

- OpenDHT makes a great common substrate for:
  - Soft-state storage
  - Naming and rendezvous
- Many P2P applications also need to:
  - Traverse NATs
  - Redirect packets within the infrastructure (as in i3)
  - Refresh puts while intermittently connected
- All of these can be implemented with upcalls
  - Who provides the machines that run the upcalls?

Future Work: Upcalls

- We don’t want to add upcalls to the core DHT
  - Keep the main service simple, fast, and robust
- Can we build a separate upcall service?
  - Some other set of machines organized with ReDiR
  - Security: can only accept incoming connections, can’t write to local storage, etc.
- This should be enough to implement
  - NAT traversal, reput service
  - Some (most?) packet redirection
- What about more expressive security policies?

For more information, see
http://bamboo-dht.org/
http://opendht.org/