Overlays and DHTs

Ashish Vulimiri and Liangliang Cao
Overlays and DHTs

- Discussion on overlay
  - “Resilient Overlay Networks” by D. Andersen, H. Balakrishnan, F. Kaashoek, R. Morris

- Review and compare different DHT algorithms
  - Pastry
  - CAN
  - Kelips
PART I
Resilient Overlay Networks

D. Andersen, H. Balakrishnan, F. Kaashoek, R. Morris
Motivation

- BGP strategy: simplify, summarize, aggregate, propagate
- Great for scalability, but causes problems
  - Loss of policy constraints
  - Performance issues
- Besides, network might not even know what to optimize for
  - Different applications have different priorities
Goals (as stated)

- Fast failure detection and recovery
  - On the order of seconds instead of minutes
- Tighter integration with applications
  - Let applications choose their performance metrics
- Expressive policy routing
  - Fine grained policy control
Basic Idea

- Establish overlay over Internet
- Nodes broadcast information, run link state protocol
- Why does this help?
  - Built-in redundancy in Internet paths
    - Not all of it exposed due to policy constraints
Basic Idea
Basic Idea

- Establish overlay over Internet
- Nodes broadcast information, run link state protocol
- Why does this help?
  - Built-in redundancy in Internet paths
    - Not all of it exposed due to policy constraints
  - BGP is not purely adaptive in nature; triangle inequality does not hold
    - There may be a node B such that the path AB-BC is actually better than the direct path AC
Basic Idea
Details

• Policy
  • Use policy tag on each packet. Policy can depend on value of any part of packet (deep packet inspection)

• Performance – use probe information as input to:
  • Loss rate optimizer
    • \( p = \text{Loss rate in last 100 samples} \)
  • Latency optimizer
    • \( \text{lat}(i) = (1 - \alpha) \text{lat}(i-1) + (\alpha) \text{sample}(i) \) [they take \( \alpha = 0.9 \)]
  • Throughput optimizer

\[
\text{score} = \frac{\sqrt{1.5}}{\text{rtt} \cdot \sqrt{p}}
\]
Experimental Results

- Two datasets
  - RON1: 12 nodes, 64 hours, Mar 2001
  - RON2: 16 nodes, 85 hours, May 2001

- Loss rate, latency, throughput, convergence time
Experimental Results: Summary

- **Loss rate**
  - RON1: Always ensured loss rate was < 30%; Internet did not. For lower loss rates, RON still generally better
  - RON2: RON better, but less dramatic

- **Latency:**
  - Better by tens/hundreds of ms. Better by >40ms on 11% of paths
  - Worse in some cases due to overhead

- **Bandwidth:**
  - 1\textsuperscript{st} %ile: 50% degradation
  - 50\textsuperscript{th} %ile: no change
  - 95\textsuperscript{th} %ile: 100% improvement
Convergence Time

More preferred path is artificially flooded at $T=5s$. RON switches to alternate path in 15s. BGP: flooding $\neq$ failure.
Discussion I

- Authors listed three goals earlier. How well did they do?
- Expressive policy routing
  - Nice solution, but overlay may not be the place for this. Lot of recent research about pushing it into network
- Fast failure detection and recovery
  - Again, ...
- Tighter integration with applications
  - This is the key take-away from this paper
- Comments?
Discussion II

- Why link state? Why not distance vector?
  - SRON [CoNEXT 2009]: $O(n\sqrt{n})$ communication cost instead of $O(n^2)$ – uses a quorum system
- RON is closer to application layer than BGP, yet no congestion control – problem?
- What happens if you have multiple parallel overlays?
  - Routing underlay
PART II: DHTs

- Overview
- Pastry
- CAN
- Kelips
- Experiments and Discussions
Overview: Distributed hash tables

- **Hash tables**
  - essential building block in software systems

- **Internet-scale distributed hash tables**
  - peer-to-peer systems
    - Napster, Gnutella, FreeNet, ...
  
  - large-scale storage management systems
    - OceanStore (on PlanetLab), PAST, CFS ...

  - mirroring on the Web
Overview: Distributed hash tables

- Idea is to support a simple index with API:
  - Insert(key, value) – saves (key,value) tuple
  - Lookup(key) – looks up key and returns value
  - The (key,value) pairs might tell us *where to look for something* but probably not *the actual thing*

- Comparison with classical hash table:
  - Implement it in a p2p network, not a server
  - Each p2p client has just part of the tuples, hence must route query to the right place
Recap

- We saw three information location mechanisms:
  - Napster, Gnutella, Chord
- Chord:
  - Hash node_id and key into same circular keyspace
  - Store value for key k in the node with node_id as small as possible while still larger than k
- We saw details about how keys are actually located, how the network is formed etc
### Memory Messages

<table>
<thead>
<tr>
<th>Network</th>
<th>Memory</th>
<th>Messages exchanged</th>
</tr>
</thead>
<tbody>
<tr>
<td>Napster</td>
<td>$O(1)$ $(O(N)$ at server)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Gnutella</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Chord</td>
<td>$O(\log(N))$</td>
<td>$O(\log(N))$</td>
</tr>
<tr>
<td>Pastry</td>
<td>$O(\log(N) / \log(2^b) \times [2^b - 1])$</td>
<td>$O(\log(N) / \log(2^b))$</td>
</tr>
<tr>
<td>CAN</td>
<td>$O(d)$</td>
<td>$O(dn^{1/d})$</td>
</tr>
<tr>
<td>Kelips</td>
<td>$O(\sqrt{N})$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems

A. Rowstron and P. Druschel
Pastry

- Somewhat similar to Chord – same 128-bit circular keyspace
  - Same hash algorithm (SHA-1) is recommended in both papers

- Differences:
  - Algorithm: Chord uses binary search, Pastry uses radix search
  - Pastry tries to take advantage of locality information
Routing in Pastry

- All ids are treated as base-2^b numbers (e.g. if b = 4, all ids are treated as hexadecimal numbers)
- Assume network has N nodes
- In each routing step, goal is to forward message to a node whose id is one digit closer to destination’s than current node’s
- Each node maintains three data structures to help decide where to forward
Routing in Pastry

- Let D be nodeId of current node
- Two data structures mostly related to routing:
  - **Routing Table (R):** Each row has \((2^b - 1)\) entries. Each entry in nth row has first n digits same as in D, but (n+1)th digit different from D’s.
  - **Leaf Set (L):** List of nodes with ids closest to D. Half the nodes have id larger than D, other half have smaller id
- Another data structure
  - **Neighbourhood set (M):** List of nodes closest (most proximate) to D. Mostly used for maintaining locality
## NodeID 10233102

### Leaf Set

<table>
<thead>
<tr>
<th>Leaf Set</th>
<th>SMALLER</th>
<th>LARGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>10233033</td>
<td>10233021</td>
<td>10233120</td>
</tr>
<tr>
<td>10233001</td>
<td>10233000</td>
<td>10233230</td>
</tr>
</tbody>
</table>

### Routing Table

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>-2-2301203</th>
<th>-3-1203203</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0-2212102</td>
<td>0</td>
<td>1-1-301233</td>
<td>1-2-230203</td>
</tr>
<tr>
<td>0</td>
<td>10-0-31203</td>
<td>2</td>
<td>10-3-23302</td>
</tr>
<tr>
<td>10-0-31203</td>
<td>0</td>
<td>10-1-32102</td>
<td>1</td>
</tr>
<tr>
<td>102-0-0230</td>
<td>1023-0-322</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1023-0-322</td>
<td>10233-0-01</td>
<td>10233-2-32</td>
<td>102331-2-0</td>
</tr>
<tr>
<td>10233-0-01</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

### Neighborhood Set

<table>
<thead>
<tr>
<th></th>
<th>10231022</th>
<th>10200230</th>
<th>11301233</th>
<th>31301233</th>
</tr>
</thead>
<tbody>
<tr>
<td>02212102</td>
<td>22301203</td>
<td>31203203</td>
<td>33213321</td>
<td></td>
</tr>
</tbody>
</table>
Routing Algorithm

- Suppose node A receives query with key D
- Case I:
  - If D lies in the range of L, deliver D to the node in L with closest nodeId
- Otherwise: Check R for the entry that is one digit closer to D than A
- Case II:
  - If the entry is not null, forward query to that node
- Case III:
  - Otherwise, find the nodeId in L, R, or M that is closest to D, and forward to that node
Analysis

- Case I leads to query termination
- Case II moves the search one digit forward
- Case III is the only problematic one
- Case III is infrequent: probability is 0.6% when $|L| = (2 \cdot 2^b)$
- Query time is still $O(\log(N))$ when no failures
  - Authors experimentally demonstrate graceful degradation in presence of failures
Node Arrival

- Assume any node X wanting to join the DHT knows a proximate neighbour A already in the DHT
- X sends a query to A with its own nodeId
- Query returns Z, the node already in the DHT whose id is closest to that of X
- Suppose query path is X-A-X₁-X₂-...-Xₘ-Z
Node Arrival

- X sets:
  - M from M of A (because of proximity)
  - L from L of Z (because nodeIds are close)
  - R values from rows in the Xi nodes (because search process proceeds digit-by-digit, as we saw earlier)
- After constructing all the sets, X sends update notifications to all the nodes in the sets
- Can be shown that this method of joining does not impact locality properties
Node Departure

- Neighbourhood set is actively kept current
- Node failures are detected in other two sets only when attempt is made to contact them
- Updating the sets in event of a failure:
  - M: Request neighbour lists from the nodes still in M
  - L: Ask node in L with highest (or lowest, as necessary) remaining nodeId for a successor (or predecessor)
  - R: Iterate through the table R, starting from the row containing the failure, and ask the nodes for a replacement until one is found
Summary

- Pastry: similar to Chord
- Asymptotically similar memory and message complexity
  - But with a potentially smaller constant, depending on b
- Also tries to take advantage of locality information
A Scalable, Content-Addressable Network

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker
Berkeley, AT&T

Some figures are courtesy to Ratnasamy
CAN: basic idea

insert \((K_1, V_1)\)

retrieve \((K_1)\)
CAN: basic ideas

- virtual Cartesian coordinate space

- entire space is partitioned amongst all the nodes
  - every node “owns” a zone in the overall space

- nodes only maintain state for their immediate neighbors
CAN: simple example
CAN: simple example
CAN: simple example
CAN: simple example
CAN: simple example
CAN: simple example

node I::insert(K,V)
CAN: simple example

node $I::\text{insert}(K,V)$

(1) $a = h_x(K)$
CAN: simple example

node I::insert(K,V)

(1) \( a = h_x(K) \)
    \( b = h_y(K) \)
CAN: simple example

node I::insert(K,V)

(1) \( a = h_x(K) \)
    \( b = h_y(K) \)

(2) route(K,V) -> (a,b)
CAN: simple example

node I::insert(K,V)

(1) a = \( h_x(K) \)
    b = \( h_y(K) \)

(2) route(K,V) -> (a,b)

(3) (a,b) stores (K,V)
CAN: simple example

node J::retrieve(K)

(1) \( a = h_\chi(K) \)
    \( b = h_\gamma(K) \)

(2) route “retrieve(K)” to (a,b)
CAN: routing table
A node only maintains state for its immediate neighboring nodes.
CAN: node insertion

1) Discover some node “I” already in CAN
1) discover some node “I” already in CAN
CAN: node insertion

2) pick random point in space

new node
CAN: node insertion

3) I routes to (p,q), discovers node J
CAN: node insertion

4) split J’s zone in half... new owns one half
CAN: node insertion

- Inserting a new node affects only a single other node and its immediate neighbors

- Need to repair the space
  - recover database
  - repair routing by takeover algorithm
CAN: takeover algorithm

- Simple failures
  - know your neighbor’s neighbors
  - when a node fails, one of its neighbors takes over its zone

- More complex failure modes
  - simultaneous failure of multiple adjacent nodes
  - scoped flooding to discover neighbors
  - hopefully, a rare event
Summary

- **CAN**
  - an Internet-scale hash table
  - potential building block in Internet applications
- **Scalability**
  - $O(d)$ per-node state
- **Low-latency routing**
  - simple heuristics help a lot
- **Robust**
  - decentralized, can route around trouble
Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead

Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, Robbert van Renesse

IPTPS 2003
Kelips: basic idea

- Two-level hashing with virtual affinity group
  - Hashing of node IP: map to affinity group
  - Hashing of FileName: map to where it is stored
- Kelips: faster lookup, consume more memory
  - $O(1)$ look up cost, $O(\sqrt{N})$ memory
  - Contrast: Chord: $O(\log(N))$ lookup, $O(\log(N))$ memory
Kelips: Core design

- Three tables stored in each node
  1. Affinity group view:
     - info. of group member: IP, RTT, etc
  2. Contract
     - Info of other groups (small set): IP, RTT, heartbeat count
  3. Filetuples
     - Info of the nodes storing a given file in the affinity group
Kelips: Core design

Hash function: \textbf{SHA-1}

Memory Usage at a node $(n/k) + c(k-1) + (F/k)$

Lookup queries return the location of the file within \textbf{O(1)} time and message complexity

Memory utilization is minimized at $k = O(\sqrt{N})$

$F = O(N)$, $\text{Util} = O(\sqrt{N})$
File Look up

- Querying node maps file name to the appropriate affinity group
- Sends lookup request to the topologically closest ‘Contact’ node from that group.
- Receive lookup request resolved by searching among filetupletable.
- $O(1)$ time and message complexity
Discussions: DHT for data center

Several classes of data stores:
- SQL/XML (object-relational) databases
- Distributed hash table (DHT),
- the Hadoop Distributed File System (HDFS)

Disadvantages of DHT:
- No full guarantee for data consistency and integrity
- Limited to simple query: <key, value> mapping
- No authority, not designed for events/triggers
Discussions: DHT for data center

- Disadvantages of DHT:
  - No full guarantee for data consistency and integrity
  - Limited to simple query: <key, value> mapping
  - No authority, not designed for events/triggers

- Advantages of DHT:
  - Resilient to network, handle node changes/removal
  - Data is automatically distributed
  - Data is replicated across nodes
Discussions: DHT for data center

- Conclusion: DHT might not be as competitive as other data center options, however, it is powerful tool to be combined with others.
- Examples: add a DTH layer in Hadoop file system
  - Hadoop has different ways for large file replication (blocks of X MB) across a huge cluster.
  - Using a DHT layer to store the locations of each replicated block.
- Benefits: remove the need for the authoritative and centralized HDFS file directory server and make the network overall more robust and resilient (the HDFS central server is a single point of failure).