Distributed Hash Tables (DHTs)
Chord & CAN

CS 699/IT 818
Sanjeev Setia

Acknowledgements

The following slides are borrowed or adapted from talks by Robert Morris (MIT) and Sylvia Ratnasamy (ICSI)

Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Robert Morris
Ion Stoica, David Karger,
M. Frans Kaashoek, Hari Balakrishnan

MIT and Berkeley
SIGCOMM Proceedings, 2001

Chord Simplicity

- Resolution entails participation by $O(\log(N))$ nodes
- Resolution is efficient when each node enjoys accurate information about $O(\log(N))$ other nodes
- Resolution is possible when each node enjoys accurate information about 1 other node

"Degrades gracefully"
Chord Algorithms

- Basic Lookup
- Node Joins
- Stabilization
- Failures and Replication

Chord Properties

- Efficient: $O(\log(N))$ messages per lookup
  - $N$ is the total number of servers
- Scalable: $O(\log(N))$ state per node
- Robust: survives massive failures

Proofs are in paper / tech report
- Assuming no malicious participants

Chord IDs

- Key identifier = SHA-1(key)
- Node identifier = SHA-1(IP address)
- Both are uniformly distributed
- Both exist in the same ID space
- How to map key IDs to node IDs?

Consistent Hashing [Karger 97]

- Target: web page caching
- Like normal hashing, assigns items to buckets so that each bucket receives roughly the same number of items
- Unlike normal hashing, a small change in the bucket set does not induce a total remapping of items to buckets
Consistent Hashing [Karger 97]

Circular 7-bit ID space

Key 5 is stored at node N105

A key is stored at its successor: node with next higher ID

Basic lookup

“Where is key 80?”

“N90 has K80”

Simple lookup algorithm

Lookup(my-id, key-id)
    n = my successor
    if my-id < n < key-id
        call Lookup(id) on node n  // next hop
    else
        return my successor  // done

Correctness depends only on successors

“Finger table” allows log(N)-time lookups
Finger $i$ points to successor of $n + 2^i$.

Lookup with fingers

Lookup(my-id, key-id)
look in local finger table for highest node $n$ s.t. $\text{my-id} < n < \text{key-id}$
if $n$ exists
call Lookup(id) on node $n$  // next hop
else
return my successor  // done

Lookups take $O(\log(N))$ hops

Node Join - Linked List Insert
2. N36 sets its own successor pointer

3. Copy keys 26..36 from N40 to N36

4. Set N25’s successor pointer

Update finger pointers in the background. Correct successors produce correct lookups.

- Case 1: finger tables are reasonably fresh
- Case 2: successor pointers are correct; fingers are inaccurate
- Case 3: successor pointers are inaccurate or key migration is incomplete

Stabilization algorithm periodically verifies and refreshes node knowledge:
- Successor pointers
- Predecessor pointers
- Finger tables
Failures and Replication

Solution: successor lists
- Each node knows its immediate successors
- After failure, will know first live successor
- Correct successors guarantee correct lookups
- Guarantee is with some probability

Choosing the successor list length
- Assume 1/2 of nodes fail
- \( P(\text{successor list all dead}) = (1/2)^r \)
  - I.e. \( P(\text{this node breaks the Chord ring}) \)
  - Depends on independent failure
- \( P(\text{no broken nodes}) = (1 - (1/2)^r)^N \)
  - \( r = 2 \log(N) \) makes prob. \( 1 - 1/N \)

Chord status
- Working implementation as part of CFS
- Chord library: 3,000 lines of C++
- Deployed in small Internet testbed
- Includes:
  - Correct concurrent join/fail
  - Proximity-based routing for low delay
  - Load control for heterogeneous nodes
  - Resistance to spoofed node IDs
Experimental overview

- Quick lookup in large systems
- Low variation in lookup costs
- Robust despite massive failure
- See paper for more results

Experiments confirm theoretical results

Chord lookup cost is $O(\log N)$

Failure experimental setup

- Start 1,000 CFS/Chord servers
  - Successor list has 20 entries
- Wait until they stabilize
- Insert 1,000 key/value pairs
  - Five replicas of each
- Stop X% of the servers
- Immediately perform 1,000 lookups

Massive failures have little impact

(1/2)$^x$ is 1.6%
Latency Measurements

- 180 Nodes at 10 sites in US testbed
- 16 queries per physical site (sic) for random keys
- Maximum < 600 ms
- Minimum > 40 ms
- Median = 285 ms
- Expected value = 300 ms (5 round trips)

Chord Summary

- Chord provides peer-to-peer hash lookup
- Efficient: $O(\log(n))$ messages per lookup
- Robust as nodes fail and join
- Good primitive for peer-to-peer systems

http://www.pdos.lcs.mit.edu/chord

A Scalable, Content-Addressable Network

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker

Content-Addressable Network (CAN)

- CAN: Internet-scale hash table
- Interface
  - insert(key, value)
  - value = retrieve(key)
- Properties
  - scalable
  - operationally simple
  - good performance
- Related systems: Chord/Pastry/Tapestry/Buzz/Plaxton...
Problem Scope

- Design a system that provides the interface
  - scalability
  - robustness
  - performance
  - security

- Application-specific, higher level primitives
  - keyword searching
  - mutable content
  - anonymity

CAN: basic idea

- Insert \((K, V)\)
**CAN: basic idea**

- $(K, V)$
- $K$ and $V$

**CAN: solution**

- virtual Cartesian coordinate space
- entire space is partitioned amongst all the nodes
  - every node "owns" a zone in the overall space
- abstraction
  - can store data at "points" in the space
  - can route from one "point" to another
- point = node that owns the enclosing zone

**CAN: simple example**
node \texttt{I::insert(K,V)}

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

\(x = a\)
**CAN: simple example**

node I::insert(K,V)
(1) \( a = h_x(K) \)
    \( b = h_y(K) \)
(2) route(K,V) \( \rightarrow \) (a,b)

**CAN: simple example**

node J::retrieve(K)
(1) \( a = h_x(K) \)
    \( b = h_y(K) \)
(2) route "retrieve(K)" to (a,b)

**CAN**

Data stored in the CAN is addressed by name (i.e. key), not location (i.e. IP address)
A node only maintains state for its immediate neighboring nodes.
1) discover some node "I" already in CAN

2) pick random point in space (p,q)

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

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.

**CAN: node failures**

- Need to repair the space
  - recover database
    - soft-state updates
    - use replication, rebuild database from replicas
  - repair routing
    - 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

**CAN: node failures**

Only the failed node's immediate neighbors are required for recovery.
Design recap

- Basic CAN
  - completely distributed
  - self-organizing
  - nodes only maintain state for their immediate neighbors
- Additional design features
  - multiple, independent spaces (realities)
  - background load balancing algorithm
  - simple heuristics to improve performance

Evaluation

- Scalability
- Low-latency
- Load balancing
- Robustness

CAN: scalability

- For a uniformly partitioned space with $n$ nodes and $d$ dimensions
  - per node, number of neighbors is $2d$
  - average routing path is $\frac{(dn^{\log d})/4}{n}$ hops
  - simulations show that the above results hold in practice
- Can scale the network without increasing per-node state
- Chord/Plaxton/Tapestry/Buzz
  - $\log(n)$ nbs with $\log(n)$ hops

CAN: low-latency

- Problem
  - latency stretch = $\frac{\text{(CAN routing delay)}}{\text{(IP routing delay)}}$
  - application-level routing may lead to high stretch
- Solution
  - increase dimensions
  - heuristics
    - RTT-weighted routing
    - multiple nodes per zone (peer nodes)
    - deterministically replicate entries
**CAN: load balancing**

- Two pieces
  - Dealing with hot-spots
    - popular (key, value) pairs
    - nodes cache recently requested entries
    - overloaded node replicates popular entries at neighbors
  - Uniform coordinate space partitioning
    - uniformly spread (key, value) entries
    - uniformly spread out routing load

**Uniform Partitioning**

- Added check
  - at join time, pick a zone
  - check neighboring zones
  - pick the largest zone and split that one
Uniform Partitioning

65,000 nodes, 3 dimensions

<table>
<thead>
<tr>
<th>Volume</th>
<th>Percentage</th>
<th>w/o check</th>
<th>w/ check</th>
</tr>
</thead>
<tbody>
<tr>
<td>V/8</td>
<td>16</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>V/4</td>
<td>8</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>V/2</td>
<td>4</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>V</td>
<td>2</td>
<td></td>
<td>65</td>
</tr>
</tbody>
</table>

V = total volume

CAN: Robustness

- Completely distributed
  - no single point of failure
- Not exploring database recovery
- Resilience of routing
  - can route around trouble

Routing resilience

Routing resilience
Routing resilience

Node X::route(D)
If (X cannot make progress to D)
- check if any neighbor of X can make progress
- if yes, forward message to one such nbr
Routing resilience

CAN size = 16K nodes
Pr(node failure) = 0.25

Routing resilience

CAN size = 16K nodes
#dimensions = 10

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