Distributed Hash Tables

- Chord is a generic example of a distributed hash table
- Processors organized into an interconnection topology with diameter $O(\log n)$ and degree $O(\log n)$
- Strengths:
  - Probabilistic load-balance algorithm
  - Implicit routing: location of object can be determined by using the hash function on the object's key
- Weaknesses:
  - Destroys integrity of key-space
  - Related objects will be mapped to arbitrary locations
  - "mozilla-3.5.rpm" might be stored in an arbitrarily different place from "mozilla-3.6.rpm"
  - Cannot query for "mozilla*"

Content Addressable Networks (CAN)

- Chord (and other DHT’s like Tapestry, Pastry) organize the nodes into hypercube-like interconnection networks
- CAN organizes nodes into a multi-dimensional mesh
- CAN supports a similar abstraction as Chord
  - Insert(key, value)
  - Lookup(key) $\Rightarrow$ value
- Components of CAN:
  - Assigning regions of a multidimensional space to nodes
  - Handle node joins and leaves
  - Routing and routing optimizations

Division of Space

- Cartesian coordinate space is divided into zones through a randomized process

Can

- 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

Insert Operation

node $I$: insert($K,V$)

1. $a = h_n(K)$
   $b = h_y(K)$
2. route($K,V$) $\Rightarrow$ ($a,b$)
3. ($a,b$) stores ($K,V$)
**Lookup Operation**

node J::retrieve(K)

1. \( a = h_x(K) \)
2. \( b = h_y(K) \)
3. Route "retrieve(K)" to \((a,b)\)

\( J \)::\( retrieve(K) \)

**CAN (Node Join)**

1. Discover some node "I" already in CAN
2. Pick random point in space

**CAN (Node Join cont.)**

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

4. Split J's zone in half... new owns one half

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

**Scalability & Performance**

- For a uniformly partitioned space with \( n \) nodes and \( d \) dimensions
  - Per node, number of neighbors is \( 2d \)
  - Average routing path is \( (dn^{3/4})/4 \) hops
  - Simulations show that the above results hold in practice

- Can scale the network without increasing per-node state

- Chord/Plaxton/Tapestry/Buzz
  - \( \log(n) \) nbrs with \( \log(n) \) hops

**Node Failures**

- Need to repair the space
  - Recover database
    - Use replication, rebuild database from replicas
  - Repair routing
    - Takeover algorithm

- CAN: takeover algorithm
  - 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.
Routing Resilience

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

- Dimensions

Distributed Binning

- **Goal:** Bin nodes such that co-located nodes land in same bin
- **Approach:**
  - Well known set of landmark machines
  - Each CAN node measures its RTT to each landmark
  - Orders the landmarks in order of increasing RTTs
- **CAN construction:**
  - Place nodes from the same bin close together on the CAN

Peer-to-Peer Information Retrieval

- **Distributed Hash Table**
  - Scalable, fault tolerant, self-organizing
  - Only support exact key match
  - $K_w$ = hash ("books on computer networks")
  - $K_q$ = hash ("computer network")
- **pSearch:**
  - Extend DHTs with content-based search
  - Full-text search, music/image retrieval
  - Build large-scale search engines using P2P technology

pSearch Key Idea

- Background: Vector Space Model

<table>
<thead>
<tr>
<th>Vocabulary</th>
<th>$V_a$</th>
<th>$V_q$</th>
<th>$V_b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>book</td>
<td>0.5</td>
<td>0.89</td>
<td>0</td>
</tr>
<tr>
<td>computer</td>
<td>0.5</td>
<td>0.5</td>
<td>0.72</td>
</tr>
<tr>
<td>network</td>
<td>0.8</td>
<td>0.8</td>
<td>0.9</td>
</tr>
<tr>
<td>routing</td>
<td>0</td>
<td>0</td>
<td>0.6</td>
</tr>
</tbody>
</table>

A: “books on computer networks”
B: “network routing in P2P networks”
Q: “computer network”
Background: Latent Semantic Indexing

documents

V_a  V_b

terms

SVD

V_a'  V_b'

semantic vectors

SVD: singular value decomposition
- Reduce dimensionality
- Suppress noise
- Discover word semantics
- Car <-> Automobile

pLSI Basic Idea

- Use a CAN to organize nodes into an overlay
- Use semantic vectors generated by LSI as object key to store doc indices in the CAN
  - Index locality: indices stored close in the overlay are also close in semantics
- Two types of operations
  - Publish document indices
  - Process queries

Content-directed Search

- Search the node whose zone contains the query semantic vector. (query center node)
- Search direct (1-hop) neighbors of query center
- Selectively search some 2-hop neighbors
  - Focusing on "promising" regions suggested by samples

Content-Aware Node Bootstrapping

- On node join, CAN picks a random point and splits the zone that contains the point
- pSearch randomly picks the semantic vector of an existing document for node bootstrapping