MCMD L22 : distrib | Distributed Hash Tables

distributed nodes

Many nodes in graph
- each node knows only small number of neighbors
- need to communicate to calculate

key bottleneck is communication

-------------------------

Distributed Hash Tables
store massive data
- quick look-up (routing)
- robust to (many) node failures
- no node stores too much data
- small degree

History:

Napster (1999):
- central index
- data stored distributed
- all routing through central node.
  (not scalable, vulnerable to attack & lawsuit)

Gnutella (2000):
- query sends request to all nodes (no central index)
- data stored distributed
- slow queries, but safe(r) from attacks & lawsuits

Freenet (2000):
- distributed storage
- heuristic routing, not guarantee to find data

2001 (very exciting times):
  CHORD (Oct 01), Pastry (Nov 01), Tapestry (TR), CAN (TR)
  - decentralized storage and routing
  - fault tolerant (many nodes come, go)
  - scalable (degree small, routing fast)

-------------------------

KEY SPACE
hash (SHA-1) h : data → key (with 128 or 160 bits)

K = key-space, circular so largest value (111...11) next to smallest
  (000...00)
each node has ID_i in K and responsible for data such that
ID_i <= h(data) < ID_{i+1}
(and usually a bit more for limited redundancy)

ROUTING

key-based routing: greedy algorithm.
- needs notion of distance between keys d(k1, k2)

On query get(key, ID_i) at node i either:
- returns object (since it stores it)
- or calls get(key, ID_j) at node j such that
d(key,ID_i) > d(key,ID_j)
(must converge)

Routing degree tradeoff (on n nodes)
<table>
<thead>
<tr>
<th>degree</th>
<th>routing</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(1)</td>
<td>O(log n) (tree, or expander)</td>
</tr>
<tr>
<td></td>
<td>either low tolerance, or hard to maintain</td>
</tr>
<tr>
<td>O(log n)</td>
<td>O(log n) most common, flexible for other properties</td>
</tr>
<tr>
<td>O(sqrt n)</td>
<td>O(1) degree too costly</td>
</tr>
<tr>
<td>O(log n)</td>
<td>O(log n / log log n) theoretically optimally, too restrictive</td>
</tr>
</tbody>
</table>

Example: Pastry

- node ID_i assigned randomly when entering network
  (recall by Chernoff bound, they are well-distributed - no more than double gap)

- key-space K is 128 bit integer

- node has degree deg = 128/b * (2^b-1) + L + M + "slack"
  (choose some b >= 1)
  + For each j in [1,2,...,128/b] link to node with first same (j-1)b
    bits,
  different jth set of b bits (2^b) links for each j
  + L other leaf nodes (closest L/2 in either direction by
    d(ID_i, .))
  + M closest peers in latency
  typically b = 4, L = 2^b, M = 2^b
  deg =~ 34 * 16 ~ 500
  (large enough that on many random failures all nodes still connected)

- ROUTING:
  match prefix of key, and send to key in neighborhood with largest
  aligned prefix
  - if failure, route to other node with same length prefix of size
j in $[128/b]$, but next $b$ bits numerically closer – still converges.

- Data Entry/Storage: (PAST)
  key = $h(data)$
  find $ID_i = \text{argmin} |ID_i - \text{key}|$.
  Add data to $ID_i$ and closest $L$ nodes (usually in neighborhood list)

  (note, since IDs are random, data is automatically distributed
  - geographically
  - by latency)

  On build neighbors, choose node with same j-prefix with smallest latency
  - then on look-up, tend to find data with smallest latency
    (bit more potential for attacks)

- Publish/Subscribe: (SCRIBE)
  each node can publish categories
  (of data it will send out, like blog RSS, twitter)
  each node can subscribe to categories

  + to announce: compute key = $h(\text{category})$, and route towards key:
    using hierarchy
    + on subscribe, send "subscribe to key" up hierarchy,
      nodes register direction where "subscribe" came from
    + on publish: route towards key, and if node sees route to key,
      and has subscribe, sends towards subscriber.
      By DFS principals, sends messages with low over-head and
      efficiently.