Distributed Hash Tables

Jasleen Kaur

Fall 2016

Why Distributed Object Discovery?

- Distributed storage and lookup
  - Store (and look up) objects in a distributed manner
  - No centralized component
  - Objects could be files, names, music, video, servers, …

- “Middleware” for building scalable distributed systems
  - DNS
  - File Systems
  - I3
  - P2P Content Sharing

- How to quickly find (and retrieve) objects?
  - Data is important (location is not)
# Structured vs. Unstructured

<table>
<thead>
<tr>
<th>Unstructured Systems</th>
<th>Structured Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trivial overlay construction and maintenance</td>
<td>Conform to a particular graph structure</td>
</tr>
<tr>
<td>Unreliable and random search</td>
<td>Complex construction and maintenance</td>
</tr>
<tr>
<td>e.g., gnutella</td>
<td>Allow reliable and efficient object location</td>
</tr>
<tr>
<td></td>
<td>e.g., chord, pastry</td>
</tr>
</tbody>
</table>

How do we map objects onto nodes (not always an option in unstructured systems)?

How do we route requests to node responsible for a given object?

---

# Distributed Hash Table (DHTs)

- Hash table: data structure that maps “keys” to “values”
  - essential building block in software systems
- Distributed Hash Table: similar, but spread across the Internet
  - Each node stores (key, value) pairs
    - Node N stores keys, such that hash(key) = N
  - Interface:
    - `insert(key, value)`
    - `lookup(key)`
    - `join/leave`
  - Each DHT node in the overlay supports single operation:
    - given input key, route messages toward node holding key
Operation: take key as input; route messages to node holding key
DHT In Action: lookup()

Operation: take key as input; route messages to node holding key

DHT Design Goals

- An “overlay” network with:
  - flexible mapping of keys to physical nodes
  - small network diameter
  - small degree
  - local routing decisions

- A “storage” or “memory” mechanism with
  - best-effort persistence (soft state)

- We’ll look at two designs:
  - Chord
  - Pastry
**Consistent Hashing**

- **Suitability of hashing:**
  - Requires spreading objects evenly across set of nodes
  - Requires knowledge of number of nodes (buckets)
    - If smaller than actual, some nodes don’t store anything
    - If larger than actual, some buckets will not correspond to actual nodes!

- **Consistent Hashing:**
  - Hash set of objects across large ID space
  - Hash nodes too across same ID space
  - Each object maintained at nodeID “closest” to objectID
  - Like hashing, distributes objects fairly across nodes
    - Unlike hashing, only a small number of objects need to move when nodes join/leave

---

**Chord**

- Based on logical $m$-bit identifiers
  - 0 to $2^m - 1$ ordered in an identifier “circle” (modulo $2^m$)

- (Key, Value) pairs are stored/located by using a consistent hash function ($CH$) to map keys ($K$) onto a point ($F$) on the circle
  - $F = CH(K)$

- System nodes are also mapped onto points, $N_i$, on the same identifier circle
  - # Key values may be greater than # Nodes

- Node $N_i$ stores all $(K,V)$ pairs where $K$ maps to a point $F$ such that $N_i$ is the first node where $F \leq N_i$ ($N_i$ is the successor of $F$)
Hash IP address to Node ID (e.g., m=6)

<table>
<thead>
<tr>
<th>IP Address</th>
<th>Hashed ID</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>152.2.137.47</td>
<td>21</td>
<td>(USA)</td>
</tr>
<tr>
<td>203.199.213.5</td>
<td>8</td>
<td>(India)</td>
</tr>
<tr>
<td>166.111.4.37</td>
<td>42</td>
<td>(China)</td>
</tr>
<tr>
<td>169.229.60.105</td>
<td>14</td>
<td>(USA)</td>
</tr>
<tr>
<td>64.236.24.12</td>
<td>1</td>
<td>(USA)</td>
</tr>
<tr>
<td>207.46.150.20</td>
<td>38</td>
<td>(USA)</td>
</tr>
<tr>
<td>198.175.96.33</td>
<td>32</td>
<td>(USA)</td>
</tr>
<tr>
<td>144.82.100.130</td>
<td>63</td>
<td>(UK)</td>
</tr>
<tr>
<td>138.96.146.2</td>
<td>51</td>
<td>(France)</td>
</tr>
<tr>
<td>129.126.11.23</td>
<td>48</td>
<td>(Germany)</td>
</tr>
</tbody>
</table>

Nodes Maintain Successor Pointer (S')

- S' = {N1, 64.236.24.12}
- S' = {N63, 144.82.100.130}
- S' = {N51, 138.96.146.2}
- S' = {N48, 129.126.11.23}
- S' = {N42, 166.111.4.37}
- S' = {N38, 207.46.150.20}
- S' = {N32, 198.175.96.33}
- S' = {N21, 152.2.137.47}
- S' = {N14, 169.229.60.105}
Key locations in example

Stores any key in (52...63) N63
Stores any key in (49...51) N51
Stores any key in (43...48) N48
Stores any key in (39...42) N42
Stores any key in (33...38) N38
Stores any key in (22...32) N32
Stores any key in (9...14) N14
Stores any key in (5...8) N8
Stores any key in (2...8) N1
Stores any key in (49...51) N51
Stores any key in (43...48) N48
Stores any key in (39...42) N42
Stores any key in (33...38) N38
Stores any key in (22...32) N32
Stores any key in (9...14) N14
Stores any key in (5...8) N8
Stores any key in (2...8) N1
Stores any key in (49...51) N51
Stores any key in (43...48) N48
Stores any key in (39...42) N42
Stores any key in (33...38) N38
Stores any key in (22...32) N32
Stores any key in (9...14) N14
Stores any key in (5...8) N8
Stores any key in (2...8) N1

Chord

- **DHT API:**
  - Each node stores (key, value) pairs
  - **Interface:**
    - `insert(key, value)`
    - `lookup(key)`
    - `Join/leave`

- Each DHT node in the overlay supports single operation:
  - given input key, route messages toward node holding key
Simple Lookup -- recursive mode
(part one: find successor of key)

Simple Lookup -- recursive mode
(part two: return successor & send query)

Memory: O(1)
Mean lookup is O(n/2)
=> Not Scalable!
Scalable Lookup With Small Node State
(part one: use local “finger table”)

Finger table at node $j$:
for $1 \leq k \leq m$
$\text{finger}[k] = \text{SuccessorNode}((j + 2^{k-1}) \mod 2^m)$

Scalable Lookup With Small Node State
(part two: use remote finger table data)
Scalable Lookup With Small Node State
(part three: locate successor node)

Finger tables help halve the ID-space distance in each step

Mean lookup is: $O((\log_2 n)/2)$
With $m$ table entries
=> Scalable!

Chord

- DHT API:
  - Each node stores (key, value) pairs
  - Interface:
    - lookup(key)
    - insert(key, value)
    - Join/leave
  - Each DHT node in the overlay supports single operation:
    - given input key, route messages toward node holding key
Node Join (e.g., Hash(128.250.6.182) = 26)

- Nodes also maintain a predecessor link (not used for search)
- (1) Joining node contacts any existing node to find successor
- (2) Successor link created from returned value.

(3) Successor Notified and data for keys ≤ 26 moved and predecessor link made.

(4) Periodic Stabilize protocol run by all nodes updates successor link in predecessor node (N21) and predecessor link in new node; Fix_Fingers also run to fix finger tables (uses FindSuccessor() search)

Broken ring -- what if a node fails?
### Replication & Robustness:
Each node maintains list of \( r \) successors

\[
S' = \{N63, 144.82.100.130, N1, 64.236.24.12, N8, 203.199.213.5, N14, 169.229.60.105\}
\]

\[
S = \{N32, 198.175.96.33, N38, 207.46.150.20, N42, 166.111.4.37, N48, 129.126.11.23\}
\]

Protects against simultaneous node failures that could result in loss of correct successor links

Applications can replicate data at \( k \) of the \( r \) successors to provide high availability in event of node failures

### The Chord Theorems

**Per-node storage:**

*Theorem IV.1:* For any set of \( N \) nodes and \( K \) keys, with high probability, the following is true.

1. Each node is responsible for at most \((1 + \epsilon)K/N\) keys.
2. When an \((N + 1)\)th node joins or leaves the network, the responsibility for \( O(K/N) \) keys changes hands (and only to or from the joining or leaving node).

**Lookup:**

*Theorem IV.2:* With high probability, the number of nodes that must be contacted to find a successor in an \( N \)-node network is \( O(\log N) \).

**Ring consistency:**

*Theorem IV.3:* If any sequence of join operations is executed interleaved with stabilizations, then at some time after the last join the successor pointers will form a cycle on all the nodes in the network.
The Chord Theorems (cont.)

Lookups with sudden churn:

**Theorem IV.4:** If we take a stable network with \( N \) nodes with correct finger pointers, and another set of up to \( N \) nodes joins the network, and all successor pointers (but perhaps not all finger pointers) are correct, then lookups will still take \( O(\log N) \) time with high probability.

Correctness with node failures:

**Theorem IV.5:** If we use a successor list of length \( r = \Omega(\log N) \) in a network that is initially stable, and then every node fails with probability 1/2, then with high probability \( \text{find\_successor} \) returns the closest living successor to the query key.

Lookup with node failures

**Theorem IV.6:** In a network that is initially stable, if every node then fails with probability 1/2, then the expected time to execute \( \text{find\_successor} \) is \( O(\log N) \).