Distributed Systems

Day 5: Distributed Hash Tables [Part 2]
Outline: Distributed Hash Tables Continued

• Consistent hashing (Chash) Recap

• Chord

• Tapestry
Distributed Hash Table (DHT)
Consistent Hash Summary (Recap.)

• Challenge: Node servers fail frequently
  • During server Add/remove – Reduce # of keys moved

• Consistent Hash
  • Both KeyID and ServerID are in same space
    • Need a good hash function to ensure even distribution
  • Adding/Removing a server
    • Keys moved from at most one server

• Open Questions
  • How do you recover keys on failed server(s)?
  • How do you detect failed servers(s)?
Replication: ReSalt V. Multiple Chash

Key1 = Hash \((\text{Salt0}+\text{Name})\) Mod N
Key2 = Hash \((\text{Salt1}+\text{Name})\) Mod N
Key3 = Hash \((\text{Salt2}+\text{Name})\) Mod N

Key1 = Hash1 \((\text{Name})\) Mod N
Key2 = Hash2 \((\text{Name})\) Mod N
Key3 = Hash3 \((\text{Name})\) Mod N
Replication: ReSalt V. Multiple Chash

• **Lookup:**
  - Pick a random salt
  - Calculate key
  - Lookup key

One hash function: slightly different names keys

Multiple hash functions

• **Lookup:**
  - Pick a random Chash
  - Chash—determines Hash function
  - Calculate key
  - Lookup key
Lookup Mechanisms

- Tradeoff: memory versus look-up speed
  - Choose based on hardware requirements

Web-scale DHTs
E.g., DynamoDB, Cassandra

Each node maintains a list of all other nodes
Lookup is easy $O(1)$

Traditional DHTs
E.g., Chord, Tapestry

Each node maintains a list of subset of nodes
Lookup is complex $O(\log N)$
Implications of memory trade-offs

Web-scale DHTs
E.g., DynamoDB, Cassandra

Traditional DHTs
e.g., Chord, Tapestry

\text{Chash}( \text{Key}, \quad )

\text{Chash}( \text{Key}, \quad )

\text{Lookup is easy } O(1)

\text{Lookup is complex } O(\log N)
Why do you need efficient Lookup Mechanisms?

• Tradeoff: memory versus look-up speed
  • Choose based on hardware requirements

• Hardware specification

16GB, Google servers 10 years ago

256MB, Google Servers 20 years ago

8MB, IoT Device

https://www.memoryxsun.com/a14ujc19s512aq.html?viewfullsite=1
https://towardsdatascience.com/iot-machine-learning-is-going-to-change-the-world-7c4e0cd7ac32
Memory Requirements

- **IoT**
  - 8MB -> 800000
- **Server 1998**
  - 256MB -> 256000000
- **Server 2009**
  - 16GB → 1600000000

The graph shows the memory requirements for different servers and devices. The server from 1998 requires 256MB of memory, which is substantial but within the range of the available data. However, the server from 2009 requires 16GB, which is too large for IoT applications. The graph also highlights that 7% of the memory requirements of the 1998 server is 800000, indicating that the 16GB is significantly more than what is needed for most IoT applications.
Google servers 10 years ago

Google Servers 20 years ago

IoT Device 8MB

Significant Memory

Little Memory

Lookup is complex $O(\log N)$

Chash( Key, )

Chash( Key, )

Chash( Key, )
Chord
Chord: DHT

- **Terminology:**
  - Successor Node: server in charge of key
  - Lookup: find the server which has the key
  - Routing Table: list of nodes stored at server

\[
\text{Chash}(\text{Key}, \quad )
\]
Chord: LookUp Procedure

finger table for node at id $i$

<table>
<thead>
<tr>
<th>finger</th>
<th>node id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>succ($i$)</td>
</tr>
<tr>
<td>2</td>
<td>succ($i + 2$)</td>
</tr>
<tr>
<td>$j$</td>
<td>succ($i + 2^{j-1}$)</td>
</tr>
</tbody>
</table>
Chord - Overview

Identifier ring

0 = node
= key

N = 16
Table size = \log_2(N) = \log_2(16) = 4

<table>
<thead>
<tr>
<th>finger</th>
<th>node id</th>
<th>node id</th>
<th>node id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>succ(i+1)</td>
<td>Succ(5)</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>succ(i + 2)</td>
<td>Succ(6)</td>
<td>8</td>
</tr>
<tr>
<td>j</td>
<td>succ(i + 2^j - 1)</td>
<td>Succ(8)</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>succ(4 + 2^3)</td>
<td>Succ(12)</td>
<td>1</td>
</tr>
</tbody>
</table>
Chord: Lookup

Identifier ring

lookup(10)
follow finger 3 to node id 8
Tapestry
Tapestry Terminology

• Root
  • Server in charge of a key

• Publish (put)
  • add a key into the “network” of servers

• Unpublish (delete)
  • delete a key into the “network of servers

• Node ID/KeyID
Publishing (a Key)

• Nodes periodically **add keys** to Tapestry
  • Every ``N'' seconds add your keys
  • If keys are not re-added they are deleted.

• **Interesting properties:**
  • If ``Root’’ crashes --- object is not lost
  • ``Root’’ can cleanup local <k,V> if object is not republished
Tapestry Node

• **Object Store**
  - Local key-value
  - I.e., keys where this node is “Root”

• **Route—Table**
  - A table of neighbors

• **Backpointers**
  - A list of nodes (who have this node as in their route Table)