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

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

Presented by Jibin Yuan
ION STOICA

- Professor of CS at UC Berkeley
- Leading RISELab
- Known for Chord
- Author of Spark, Mesos
- Cofounder of Databricks, Conviva Networks
OUTLINE

• Motivation
• Chord
• Experiment
• Discussion
Motivation

• Peer-to-peer (P2P)
• Data are stored in the nodes
• Hard to scale the location/routing in pure P2P

How to locate the node that stores a particular data item efficiently?
CHORD

Base
• Consistent hashing
• Key search
• Node join

Concurrent
• Stabilization
• Failures and Replication

Model
• Load balance
• Decentralization
• Scalability
• Availability
• Flexible naming
Consistent Hashing

- $m$ bit identifier space for both keys and nodes

- key identifier = $SHA-1(key)$
  
  $key = ‘oathkeeper’ \xrightarrow{SHA-1} ID = 60$

- node identifier = $SHA-1(IP\ address)$
  
  $IP = ‘168.10.10.15’ \xrightarrow{SHA-1} ID = 127$

- Both are uniformly distributed
CHORD – BASE PROTOCOL

Consistent Hashing

\[ 2^m - 1 \]

K60

N127
**CHORD – BASE PROTOCOL**

Consistent Hashing

\[ m = 8 \]
CHORD – BASE PROTOCOL

Key Search
• Successor: the next node (clockwise) on the identifier circle

I am the successor of N1
**CHORD – BASE PROTOCOL**

Key Search
- Predecessor:
  - the previous node (counter-clockwise)
  - on the identifier circle

My predecessor is N1

N1

N127
CHORD – BASE PROTOCOL

Key Search

Successor: N1

Time: $\Theta(n)$
Space: $\Theta(1)$

Successor: N127

Successor: N180
**CHORD – base protocol**

Key Search
- Each node has a finger table

For node $n$, using $m$-bit identifiers:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{finger}[k].\text{start}$</td>
<td>$(n + 2^{k-1}) \mod 2^m$, $1 \leq k \leq m$</td>
</tr>
<tr>
<td>$\text{.interval}$</td>
<td>$[\text{finger}[k].\text{start}, \text{finger}[k+1].\text{start}]$</td>
</tr>
<tr>
<td>$\text{.node}$</td>
<td>$\text{first node } \geq n.\text{finger}[k].\text{start}$</td>
</tr>
<tr>
<td>$\text{successor}$</td>
<td>$[\text{finger}[1].\text{node}]$</td>
</tr>
</tbody>
</table>
CHORD – BASE PROTOCOL

Key Search

Finger table:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>((n + 2^{k-1}) \mod 2^m, 1 \leq k \leq m)</td>
</tr>
<tr>
<td>interval</td>
<td>([\text{finger}[k].\text{start}, \text{finger}[k+1].\text{start}])</td>
</tr>
<tr>
<td>node</td>
<td>(\text{first node} \geq n.\text{finger}[k].\text{start})</td>
</tr>
<tr>
<td>successor</td>
<td>The next node on the identifier circle; ([\text{finger}[1].\text{node}]</td>
</tr>
</tbody>
</table>

// ask node \(n\) to find id’s successor
\(n.\text{find.successor}(id)\)
\(n' = \text{find.predecessor}(id)\);
return \(n'.\text{successor}\);

// ask node \(n\) to find id’s predecessor
\(n.\text{find.predecessor}(id)\)
\(n' = n\);
while \((id \not\in (n', n'.\text{successor})\) \(n' = n'.\text{closest.preceding.finger}(id)\);
return \(n'\);

// return closest finger preceding id
\(n.\text{closest.preceding.finger}(id)\)
for \(i = m\) downto 1
if \((\text{finger}[i].\text{node} \in (n, id))\)
    return \(\text{finger}[i].\text{node}\);
return \(n\);

Time: \(\Theta(\log n)\) Space: \(\Theta(\log n)\)
CHORD – BASE PROTOCOL

Node Join

When the node $n$ joins the network:
1. Initialize the predecessor and fingers of node
2. Update the existing nodes
3. Transfer state from successor to new node
CHORD – base protocol

Node Join – step 1
• Locate any node \( p \) in the ring
• Ask node \( p \) to lookup fingers of new node N79
• Return results to new node

<table>
<thead>
<tr>
<th>start</th>
<th>Interval</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>[80,81]</td>
<td>127</td>
</tr>
<tr>
<td>81</td>
<td>[81,83]</td>
<td>127</td>
</tr>
<tr>
<td>83</td>
<td>[83,87]</td>
<td>127</td>
</tr>
<tr>
<td>87</td>
<td>[87,95]</td>
<td>127</td>
</tr>
<tr>
<td>95</td>
<td>[95,111]</td>
<td>127</td>
</tr>
<tr>
<td>111</td>
<td>[111,143]</td>
<td>127</td>
</tr>
<tr>
<td>143</td>
<td>[143,207]</td>
<td>180</td>
</tr>
<tr>
<td>207</td>
<td>[207, 80]</td>
<td>1</td>
</tr>
</tbody>
</table>
CHORD – base protocol

Node Join – step 2

• New node calls update function on existing nodes
• Existing nodes can recursively update fingers of other nodes

<table>
<thead>
<tr>
<th>start</th>
<th>Interval</th>
<th>node</th>
<th>start</th>
<th>Interval</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>[2,3)</td>
<td>127</td>
<td>2</td>
<td>[2,3)</td>
<td>79</td>
</tr>
<tr>
<td>3</td>
<td>[3,5)</td>
<td>127</td>
<td>3</td>
<td>[3,5)</td>
<td>79</td>
</tr>
<tr>
<td>5</td>
<td>[5,9)</td>
<td>127</td>
<td>5</td>
<td>[5,9)</td>
<td>79</td>
</tr>
<tr>
<td>9</td>
<td>[9,17)</td>
<td>127</td>
<td>9</td>
<td>[9,17)</td>
<td>79</td>
</tr>
<tr>
<td>17</td>
<td>[17,33)</td>
<td>127</td>
<td>17</td>
<td>[17,33)</td>
<td>79</td>
</tr>
<tr>
<td>33</td>
<td>[33,65)</td>
<td>127</td>
<td>33</td>
<td>[33,65)</td>
<td>79</td>
</tr>
<tr>
<td>65</td>
<td>[65,129)</td>
<td>127</td>
<td>65</td>
<td>[65,129)</td>
<td>79</td>
</tr>
<tr>
<td>129</td>
<td>[129, 1)</td>
<td>180</td>
<td>129</td>
<td>[129, 1)</td>
<td>180</td>
</tr>
</tbody>
</table>

K220

K35

K60

N1 (Succ: N79)

N127 (Pred: N79)

N180
CHORD – base protocol

Node Join – step 3

- Transfer state (e.g. key) from successor node to the new node

Time: $\Theta(\log^2 n)$

After optimization:
Time: $\Theta(\log n)$
CHORD – CONCURRENT OPERATIONS AND FAILURES

Stabilization
• Handle concurrent join
• Updating successor pointers is sufficient to guarantee correctness

Three behaviors:
• Everything is up to date → ok
• Successor is up to date → ok
• Nothing is up to date → retry after a pause
CHORD – CONCURRENT OPERATIONS AND FAILURES

Stabilization
Node join (concurrently, lazy update)
1. Initialize only the finger to successor node
2. Periodically verify immediate successor, predecessor
3. Periodically refresh finger table entries

Node join (base protocol, aggressive update)
1. Initialize the predecessor and fingers of node
2. Update the existing nodes
3. Transfer state from successor to new node

Lazy is a very strong word. I like to call it “Selective participation.”
Failures and Replication
• Failure of nodes might cause incorrect lookup
• N127 doesn’t know correct successor so lookup fails

• Successor fingers are enough for correctness
Failures and Replication
Use successor list
• Each node knows $r$ immediate successors
• After failure, will know first live successor
• Correct successors guarantee correct lookups

Guarantee is with some probability
• Can choose $r$ to make probability of lookup failure arbitrarily small
CHORD – model

Addressing these problems:
• Load balance
• Decentralization
• Scalability
• Availability
• Flexible naming
Experiment

Simulator
• Iterative (vs recursive)

Experiments
• Load balance
• Path length
• Simultaneous node failures
• Lookups during stabilization
• Latency measurements
Experiment – LOAD BALANCE

Reason of variance: node identifiers do not uniformly cover the entire identifier space.
Experiment – LOAD BALANCE

Solution for variance:
• Associate the key with virtual nodes
• Map multiple virtual nodes to real node (consistent hash paper)

Figure 9: The 1st and the 99th percentiles of the number of keys per node as a function of virtual nodes mapped to a real node. The network has $10^4$ real nodes and stores $10^6$ keys.
Experiment – PATH LENGTH

Figure 10: (a) The path length as a function of network size. (b) The PDF of the path length in the case of a $2^{12}$ node network.

- $N = 2^k$ nodes, $100 \times 2^k$ keys, $3 \leq k \leq 14$
- Path length about $\frac{1}{2} \log_2 N$
Definition: correct lookup of a key: finds the node that was originally responsible for the key, before the failures

Worry: Network partition (Did not happen)

Figure 11: The fraction of lookups that fail as a function of the fraction of nodes that fail.

Lookup failure rate = Node failure rate
Experiment – LOOKUPS DURING STABILIZATION

Setup
- 500 nodes initially
- stabilize() called per 30s in average
- one lookup per sec (Poisson)
- x fail/join per sec (Poisson)

Figure 12: The fraction of lookups that fail as a function of the rate (over time) at which nodes fail and join. Only failures caused by Chord state inconsistency are included, not failures due to lost keys.
Experiment – LOOKUPS DURING STABILIZATION

Definition: correct lookup of a key: find current successor of the desired key

From Probability: If $p$ nodes fail btw stabilization, the failure rate would be $p\%$

Why it is worse:
May $\geq 1$ stabilization to clear out a failed node

Figure 12: The fraction of lookups that fail as a function of the rate (over time) at which nodes fail and join. Only failures caused by Chord state inconsistency are included, not failures due to lost keys.
Experiment – LATENCY MEASUREMENTS

Setup
• 10 sites
• SHA - 1
• TCP
• Iterative

Lookup latency grows slowly with the total number of nodes.

Figure 13: Lookup latency on the Internet prototype, as a function of the total number of nodes. Each of the ten physical sites runs multiple independent copies of the Chord node software.
Future Work

- No mechanism to heal partitioned rings
- Incorrect view of ring from malicious participants
- Decrease lookup latency
DISCUSSION

Strengths
• Load balance, Decentralization, Scalability, Availability…
• Based on theoretical work (consistent hashing)
• Proven performance in many different aspects
• Simplicity, proven correctness

Limitations
• Load balance on hot spot
• No mechanism to handle node leave
• Transfer state between nodes due to join/fail
Q/A