The Chord P2P Network

Some slides have been borrowed from the original presentation by the authors
Chord vs. Tapestry

- The topology of the Chord network, as defined by the successor pointers, must satisfy a well-defined structure.

- Tapestry (uses Plaxton routing) requires the root of the object to be placed in a designated node, but the object can be placed locally. In contrast, Chord requires the object to be placed at a designated node.
Main features of Chord

- Load balancing via Consistent Hashing
- Small routing tables: $\log n$
- Small routing delay: $\log n$ hops
- Fast join/leave protocol (polylog time)
Consistent Hashing

Assigns both nodes and objects from an \( m \)-bit key.

Order these nodes around an identifier circle (what does a circle mean here?) according to the order of their keys (0 .. \( 2^m-1 \)). This ring is known as the Chord Ring.

Object with key \( k \) is assigned to the first node whose key is \( \geq k \) (called the successor node of key \( k \))
Consistent Hashing

Example: Node 90 is the “successor” of document 80.
Consistent Hashing [Karger 97]

Property 1
If there are $N$ nodes and $K$ keys, then with high probability, each node is responsible for $(1 + \epsilon)K/N$ keys.

Property 2
When a node joins or leaves the network, the responsibility of at most $O(K/N)$ keys changes hands (only to or from the node that is joining or leaving.

When $K$ is large, the impact on individual nodes is quite small.
Consistent hashing

An object with key $k$ is stored at its successor (node with key $\geq k$)
The log N Fingers

Each node knows of only $\log N$ other nodes.
Finger $i$ points to successor of $n + 2^i$
Node n’s i-th entry: **first node** ≥ n + 2^{i-1}
Node 32, lookup(82): 32 $\rightarrow$ 70 $\rightarrow$ 80 $\rightarrow$ 85.
New Node Joins

N20’s Finger Table

<table>
<thead>
<tr>
<th>1</th>
<th>21..21</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>22..23</td>
</tr>
<tr>
<td>3</td>
<td>24..27</td>
</tr>
<tr>
<td>4</td>
<td>28..35</td>
</tr>
<tr>
<td>5</td>
<td>36..51</td>
</tr>
<tr>
<td>6</td>
<td>52..83</td>
</tr>
<tr>
<td>7</td>
<td>84..19</td>
</tr>
</tbody>
</table>

Assume N20 knows one of the existing nodes.
Node 20 asks *that* node for successor to 21, 22, …, 52, 84.
The Join procedure

The new node id asks a gateway node n to find the successor of id

\[ n.(\text{find\_successor}(id)) \]

if id = (n, successor) *n<id<successor of n*

then return successor

else forward the query \textit{around the circle}

fi


Needs O(n) messages. This is slow.
Steps in join

Successor(n)  

Finally

But the transition does not happen immediately
A More Efficient Join

// ask n to find the successor of id
if id = (n, successor]
    then return successor
else n’= closest_preceding_node (id)
    return n’.find_successor(id)
fi

// search for the highest predecessor of id
n. closest_preceding_node(id)
for i = log N downto 1
if (finger[i] is between (n,id)
    return finger[i]
Example

N20 wants to find out the successor of key 65
Node 20 moves documents from node 32.

N20’s Finger Table

- 21..21  N32
- 22..23  N32
- 24..27  N32
- 28..35  N32
- 36..51  N40
- 52..83  N52
- 84..19  N102

Notify nodes that must include N20 in their table. N113[1]=N20, not N32.
Three steps in join

Step 1. Initialize predecessor and fingers of the new node.

(Knowledge of predecessor is useful in stabilization)

Step 2. Update the predecessor and the fingers of the existing nodes. (Thus notify nodes that must include N20 in their table. N113[1] = N20, not N32.

Step 3. Transfer objects to the new node as appropriate.
Concurrent Join

[Before]

[After]
Stabilization

Periodic stabilization is needed to integrate the new node into the network and restore the invariant.

Predecessor.successor(n1) ≠ n1, so n1 adopts predecessor.successor(n1) = n as its new successor.
The complexity of join

With high probability, any node joining or leaving an N-node Chord network will use $O(\log^2 N)$ messages to re-establish the Chord routing invariants and finger tables.
Chord Summary

• Log(n) lookup messages and table space.
• Well-defined location for each ID.
  • No search required.
• Natural load balance.
• No name structure imposed.
• Minimal join/leave disruption.