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

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

Presented by Veranika Liaukevich

Jacobs University Bremen

May 14, 2014
**Problem:** efficient location of data items in peer-to-peer systems

**Chord** builds a distributed hash table and provides only one operation: given a key, it maps it onto a node, responsible for this key.

The communication costs scale logarithmically with the number of Chord nodes.
Chord features

- Load balance
- Decentralization
- Scalability
  - each node stores information about $O(\log N)$ nodes
  - lookups are resolved in $O(\log N)$ messages
  - nodes joins and leaves result in $O(\log^2 N)$ messages in the network
- Availability
- Flexible naming
- Provable correctness and performance
Consistent Hashing

There are nodes and keys in a system.

Each node and each key are mapped into $m$-bit identifiers.

The identifiers universe is represented as a circle with $2^m$ points.

So each node and each key are represented as points on the circle.

Identifiers circle with $m = 3$. Nodes are marked as circles, keys are marked as squares.
Consistent Hashing

The key $k$ is assigned to the first node whose point it encounters moving clockwise from the key’s point, denoted by $\text{successor}(k)$.

When a node joins or leaves the network, only part of keys are reassigned.
Consistent Hashing: Properties

For mapping nodes and keys into identifiers $k$-universal hash function must be used to ensure uniform distribution of identifiers.

For any set of $N$ nodes and $K$ keys, with high probability:

- Each node is responsible for at most $(1 + \epsilon)K/N$ keys, where $\epsilon = O(\log N)$.
- When an $(N + 1)$-st node joins/leaves, $O(K/N)$ keys are reassigned to/from the joining/leaving node.
The Base Chord Protocol

Chord provides fast distributed computation of a consistent hash function mapping keys to nodes responsible for them.

SHA-1 is used for mapping keys and nodes to identifiers, thus making the protocol deterministic.

The base protocol specifies
- how to locate a key
- how node joins the network

The base protocol doesn’t support concurrent joins or failures.
### Chord: denotations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>number of bits in identifiers</td>
</tr>
<tr>
<td>successor</td>
<td>the next node on the identifier circle</td>
</tr>
<tr>
<td>predecessor</td>
<td>the previous node on the circle</td>
</tr>
<tr>
<td>finger table</td>
<td>routing table with at most $m$ records</td>
</tr>
<tr>
<td>$\text{finger}[k].\text{start}$</td>
<td>$(n + 2^{k-1}) \mod 2^m$</td>
</tr>
<tr>
<td>$\text{finger}[k].\text{node}$</td>
<td>first node $\geq n.\text{finger}[k].\text{start}$</td>
</tr>
<tr>
<td>$\text{finger}[k].\text{interval}$</td>
<td>$[\text{finger}[k].\text{start}, \text{finger}[k+1].\text{start})$</td>
</tr>
</tbody>
</table>
Finger tables, finger intervals and key locations in a network.
To resolve a lookup query for an identifier \( id \) we inspect at most \( O(\log N) \) nodes on the circle approaching the resulting node clockwise.

- \( n.\text{closest}_\text{preceding}_\text{finger}(id) \)
  Finds the closest node preceding \( id \) among fingers in the \( n \)'s node table.

- \( n.\text{find}_\text{predecessor}(id) \)
  Finds \( id \)'s predecessor by series of calls to \( \text{closest}_\text{preceding}_\text{finger} \).

- \( n.\text{find}_\text{successor}(id) \)
  \( \text{successor}(id) = \text{successor}(\text{predecessor}(id)) \)
Invariants

Invariants, preserved by Chord:

- Each node’s successor is correctly maintained.
- For every key $k$, node $successor(k)$ is responsible for $k$.

With such invariants we can perform lookups correctly. To perform lookups fast, the correct finger tables are needed.

For simplification each node also stores a $predecessor$ pointer.
Node Joins

New node $n$ learns an existing Chord node $n'$ by some external mechanism, and uses it for adding itself to the network.

When a node $n$ joins the network...

- Initialize its predecessor and fingers table in $O(\log N)$ messages.
- Update the fingers and predecessors of existing nodes in $O(\log^2 N)$ messages.
- Transfer data associated with keys.
Lookups in an unstable network

Possible situations during lookups:

- If all involved finger tables are correct, the successor is found in $O(\log N)$ steps.
- If successor pointers are correct, the successor could be found in $O(N)$ steps.
- If successor pointers are incorrect or data haven’t migrated yet, the lookup cannot be performed.
Idea: separate correctness and performance goals. Basic stabilization algorithm updates only successor pointers.

- Every node periodically runs stabilize to check correctness of its successor pointer.
- Every node periodically runs fix_fingers to refresh its finger table.
- At some time after the last join all successor pointers will be correct.
- As long as time needed to adjust fingers is less than the time it takes the network to double in size, lookups continue to take $O(\log N)$. 
For faster recovery after some node’s failure, each node maintains a list of its $r$ nearest successors.

This list could be used to update the higher layer application when successors come and go.

The list also helps in data replication, as typically each key $k$ is replicated at its next $p$ successors.
Conclusions

Chord is a decentralized lookup protocol. Given a key, it efficiently determines the node responsible for this key. Chord scales well with the number of nodes, recovers from large numbers of simultaneous node failures and joins, and answers most lookups correctly even during recovery.

The correctness and performance of the protocol are provable.
Questions?
Chord: A scalable peer-to-peer lookup service for internet applications

Web caching with consistent hashing