Distributed Hash Tables: Chord

Scott D. Stoller
References

• Slides by Christine Kiefer, with many small changes and several additions.
  http://www.mpi-inf.mpg.de/departments/d5/teaching/ws03_04/p2p-data/11-18-paper1.ppt

Overview

- **Introduction**
- **The Chord Algorithm**
  - Construction of the Chord ring
  - Localization of nodes
  - Node joins and stabilization
  - Failure of nodes
- **Applications**
- **Summary**
The lookup problem

Key="title"
Value=MP3 data...
Publisher

Client
Lookup("title")
Routed Queries (Freenet, Chord, etc.)

Publisher

Key="title"
Value=MP3 data…

Client

Lookup("title")
What is Chord?

• Distributed lookup protocol
  – lookup(key) returns the node responsible for key.
  – Chord notifies each node of changes to the set of keys for which the node is responsible, as a result of other nodes joining and leaving.

• Features:
  – Simplicity
  – Provably good expected performance
  – Proven correctness
Overview

• Introduction
• **The Chord Algorithm**
  – Construction of the Chord ring
  – Localization of nodes
  – Node joins and Stabilization
  – Failure/Departure of nodes
• Applications
• Summary
Construction of the Chord ring

• The consistent hash function assigns each node and each key an m-bit identifier. Note that all these identifiers exist in the same space (m-bit numbers).
• m should be large enough to make collisions unlikely.
• Key identifier = SHA-1(key) modulo $2^m$
• Node identifier = SHA-1(IP address) modulo $2^m$
  – To improve load-balancing, each physical node can run $v$ virtual nodes, each with its own identifier.
• SHA-1 is a cryptographic hash function published by NIST as a Federal Information Processing Standard.
Construction of the Chord ring

- Identifiers are arranged on a circle called the *Chord ring*. 
Construction of the Chord ring

- successor(id) is the first node whose identifier is \( \geq \) id. It is the first node clockwise from id. Key k is assigned to node successor(id).
Consistent hashing

// ask node $n$ to find the successor of $id$

$n$.find_successor($id$)

    if ($id \in (n; \text{successor}]$)
        return $\text{successor}$;

    else
        // forward the query around the circle
        return $\text{successor}.\text{find_successor}(id)$;

• Number of messages is linear in the number of nodes.
Simple key location

return N56
Scalable key location

• Store additional routing information at each node to accelerate lookups. For scalability, do not store info about all nodes.

• Each node contains a routing table with up to m entries, called the *finger table*.

• n.finger[i] contains the first node that succeeds node n by at least \(2^{i-1}\).

• n.finger[i] = successor(n + 2^{i-1})

• n.finger[i] is called the \(i^{th}\) *finger* of node n.

• n.successor = n.finger[1]
Scalable key location

Finger table: $\text{finger}[i] = \text{successor}(n + 2^{i-1})$
Scalable key location

**Finger table:** \( \text{finger}[i] = \text{successor}(n + 2^{i-1}) \)
Scalable key location

**Finger table:** $\text{finger}[i] = \text{successor}(n + 2^{i-1})$
Scalable key location

**Finger table**: \( \text{finger}[i] = \text{successor}(n + 2^{i-1}) \)
Scalable key location

**Finger table:** \(\text{finger}[i] = \text{successor}\ (n + 2^{i-1})\)
Scalable key location

**Finger table**: $\text{finger}[i] = \text{successor}(n + 2^{i-1})$
Scalable key location

**Finger table:** $\text{finger}[i] = \text{successor}(n + 2^{i-1})$
Scalable key location

**Finger table:** $\text{finger}[i] = \text{successor}(n + 2^{i-1})$
Scalable key location

**Finger table:** $\text{finger}[i] = \text{successor}(n + 2^{i-1})$
Scalable key location

Finger table: \( \text{finger}[i] = \text{successor}(n + 2^{i-1}) \)
Important characteristics of this scheme

• Each node stores information about only a small number of nodes
  – At most m, typically $O(\log N)$, because many low-indexed entries refer to the same node and hence do not need to be stored separately.
• Each node knows more about nodes closely following it than about nodes farer away.
• A finger table generally does not contain enough information to directly determine the successor of an arbitrary key $k$
Scalable key location

- Search in finger table for the node n’ that most immediately precedes id. Invoke find_successor on n’.
Scalable key location

• Search in finger table for the node $n'$ that most immediately precedes $id$. Invoke `find_successor` on $n'$. 
Scalable key location: Pseudocode

// ask node n to find the successor of id
n.find_successor(id)

    if (id ∈ (n, successor])
        return successor;

    else
        n' = closest_preceding_node(id);
        return n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)

    for i = m downto 1
        if (finger[i] ∈ (n, id))
            return finger[i];

    return n;
Scalable key location: Complexity

• With high probability, the number of messages used to locate a key $k$ in an $N$-node network is $O(\log N)$.
  – The hash function is assumed to distribute identifiers well (uniformly), with high probability.

• Intuition: Since each node has finger entries at power-of-two intervals around the Chord ring, each node can forward a query at least halfway along the remaining distance between itself and $k$. 
Node joins and stabilization
Node joins and stabilization

Node N26 wants to join.
Node joins and stabilization

Need to update N21.successor, and move key K24 from N32 to N26. Also need to update finger tables.
Node joins and stabilization

• To ensure correct lookups, all successor pointers must be up to date.

• *Stabilization protocol* runs periodically in the background, to update successor pointers and finger tables.
Node joins and stabilization: Stabilization protocol

- `n.stabilize()`: n asks its successor s for s’s predecessor p and decides whether p should be n’s successor instead (this is the case if p recently joined the system).

- `n.notify()`: notifies n’s successor s of n’s existence, so s can set its predecessor to n, if appropriate.

- `n.fix_fingers()`: updates n’s finger table
Node joins and stabilization

// create a new Chord ring.
\texttt{n.create()}

\texttt{predecessor = \texttt{nil};}
\texttt{successor = n;}

// join a Chord ring containing node \texttt{n'}.  
\texttt{n.join(n')}

\texttt{predecessor = \texttt{nil};}
\texttt{successor = n'.find\_successor(n);}
Node joins and stabilization

// called periodically. verifies n’s immediate
// successor, and tells the successor about n.

n.stabilize()

x = successor.predecessor;

if (x ∈ (n, successor))
    successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.

n.notify(n')

if (predecessor is nil or n' ∈ (predecessor, n))
    predecessor = n';
Node joins and stabilization

// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.

n.fix_fingers()
next = next + 1;
if (next > m)
    next = 1;
finger[next] = find_successor(n + 2^{next-1});

// called periodically. checks whether predecessor has failed.

n.check_predecessor()
if (predecessor has failed)
    predecessor = nil;
Node joins and stabilization

![Diagram showing node joins and stabilization](image-url)
Node joins and stabilization

- N26 runs join(N21), calls N21.find_successor(N26), and acquires N32 as its successor.
Node joins and stabilization

- N26 copies keys from its successor N32.
Node joins and stabilization

- N21 runs stabilize() and asks its successor N32 for its predecessor which is N26.
- N21 acquires N26 as its successor
- N21 notifies N26 of its existence
- N26 acquires N21 as its predecessor
- N26 runs stabilize() and calls N32.notify(N26), causing N32 to acquire N26 as its predecessor.
Impact of node joins on lookups

• If all finger table entries have been updated, $O(\log N)$ lookups are needed.

• If successor pointers are correct, but some fingers have not been updated, lookups are correct but may be slower (part of circle might be traversed linearly). Fingers that do not reflect recent joins can only cause the algorithm to undershoot, not overshoot, the target, because nodes forward queries to their estimate of the key’s predecessor (not successor).
Impact of node joins on lookups

• If some successor pointers are incorrect, lookup might fail (key not found). Application can pause and then retry to get the answer.
Failure of nodes

- Successful lookup depends on correct successor pointers. If N14, N21, and N32 fail simultaneously, how can N8 acquire N38 as successor?
Making Chord more fault-tolerant

• Each node maintains a *successor list* of size \( r \).
• If \( n \)‘s immediate successor does not respond, \( n \) contacts the second entry in its successor list, and so on.
  – This requires small changes to the pseudo-code for finding successors and stabilization.
• The Chord ring is disrupted only if all nodes in a node’s successor list fail almost simultaneously. This is unlikely, even for modest values of \( r \).
Overview

• Introduction
• The Chord Algorithm
  – Construction of the Chord ring
  – Localization of nodes
  – Node joins and stabilization
  – Failure/Departure of nodes
• Applications
• Summary
Application: Time-shared storage service

• Useful for storage servers with intermittent availability.

• Each server stores some data for other servers and serves it for them when the owning server is unavailable.

• Chord can be used to distribute the data among the servers (content-based routing).
Application: Chord-based DNS

• DNS provides a lookup service
  keys: host names; values: IP addresses
  Chord could hash each host name to a key
• Chord-based DNS:
  – no special root servers
  – no manual management of routing information
  – no naming structure
  – can find objects not tied to particular machines
Summary

• Simple, scalable protocol with one operation: map a key to the responsible node.

• With high probability:
  – Each node maintains information about $O(\log N)$ other nodes.
  – Lookups use $O(\log N)$ messages.
  – Continues to function correctly even when nodes continually join or fail.
Related Work

• Several other DHTs have been developed.
• An important distinction is whether the routing algorithm takes network topology into account to reduce the routing latency, i.e., to reduce the total network distance traveled by the messages used to answer a query.
• Chord does not.
• Pastry [Rowstron and Druschel, 2001] does.
1. Consider the Chord ring in Fig. 2 of the Chord paper (also on slide 25). Which nodes participate in the computation of N1.lookup(45)?

2. Write pseudo-code for n.stabilize() for the version of Chord that uses successor lists. It should update n’s successor list, based on n.successor’s successor list. It should make progress even if some of n’s successors failed.