Chord

A scalable peer-to-peer look-up protocol for internet applications

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

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

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

Internet

N₁ N₂ N₃

N₄ N₅ N₆

Client

Lookup("title")
Routed queries (Freenet, Chord, etc.)

Publisher

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

Client

Lookup("title")
What is Chord?

- Problem addressed: efficient node localization
- Distributed lookup protocol
- Simplicity, provable performance, proven correctness
- Support of just one operation: given a key, Chord maps the key onto a node
Chord software

- 3000 lines of C++ code
- Library to be linked with the application
- provides a lookup(key) – function: yields the IP address of the node responsible for the key
- Notifies the node of changes in the set of keys the node is responsible for
Overview

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

- use Consistent Hash Function assigns each node and each key an m-bit identifier using SHA 1 (Secure Hash Standard).
  
m = any number big enough to make collisions improbable
  
  Key identifier = SHA-1(key)
  Node identifier = SHA-1(IP address)

- Both are uniformly distributed
- Both exist in the same ID space
The Chord algorithm – Construction of the Chord ring

- Identifiers are arranged on a identifier circle modulo $2^m$ => Chord ring
The Chord algorithm – Construction of the Chord ring

- a key $k$ is assigned to the node whose identifier is equal to or greater than the key’s identifier
- this node is called successor($k$) and is the first node clockwise from $k$. 
The Chord algorithm – Simple node localization

// ask node n to find the successor of id
n.find_successor(id)
    if (id ∈ (n; successor])
        return successor;
else
    // forward the query around the circle
        return successor.find_successor(id);

=> Number of messages linear in the number of nodes!
The Chord algorithm – Scalable node localization

- Additional routing information to accelerate lookups
- Each node $n$ contains a routing table with up to $m$ entries ($m$: number of bits of the identifiers) => finger table
  - $i^{th}$ entry in the table at node $n$ contains the first node $s$ that succeeds $n$ by at least $2^{i-1}$
  - $s = \text{successor } (n + 2^{i-1})$
  - $s$ is called the $i^{th}$ finger of node $n$
The Chord algorithm – Scalable node localization

Finger table:

\[ \text{finger}[i] = \text{successor} \left( n + 2^{i-1} \right) \]
The Chord algorithm – Scalable node localization

Finger table:

\[ \text{finger}[i] = \text{successor}(n + 2^{i-1}) \]
The Chord algorithm –
Scalable node localization

Finger table:
$\text{finger}[i] = \text{successor}(n + 2^{i-1})$
The Chord algorithm – Scalable node localization

Finger table:

\[ \text{finger}[i] = \text{successor} \ (n + 2^{i-1}) \]
The Chord algorithm – Scalable node localization

Finger table:
\[ \text{finger}[i] = \text{successor} \left( n + 2^{i-1} \right) \]
The Chord algorithm – Scalable node localization

Finger table:

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

\[ \text{finger[i]} = \text{successor}(n + 2^{i-1}) \]
The Chord algorithm – Scalable node localization

Finger table:
\[ \text{finger}[i] = \text{successor}(n + 2^{i-1}) \]
The Chord algorithm – Scalable node localization

Finger table:
\[ \text{finger}[i] = \text{successor} \left( n + 2^{i-1} \right) \]
The Chord algorithm –
Scalable node localization

Finger table:
\[ \text{finger}[i] = \text{successor} \ (n + 2^{i-1}) \]
The Chord algorithm – Scalable node localization

Important characteristics of this scheme:

- Each node stores information about only a small number of nodes (m)
- Each node knows more about nodes closely following it than about nodes further away
- A finger table generally does not contain enough information to directly determine the successor of an arbitrary key k
The Chord algorithm – Scalable node localization

- Search in finger table for the nodes which most immediately precedes id
- Invoke find_successor from that node

=> Number of messages $O(\log N)$!
The Chord algorithm – Scalable node localization

- Search in finger table for the nodes which most immediately precedes id
- Invoke find_successor from that node

=> Number of messages $O(\log N)$!
The Chord algorithm – Node joins and stabilization
The Chord algorithm – Node joins and stabilization
The Chord algorithm – Node joins and stabilization
The Chord algorithm – Node joins and stabilization

- To ensure correct lookups, all successor pointers must be up to date
- => stabilization protocol running periodically in the background
- Updates finger tables and successor pointers
The Chord algorithm – Node joins and stabilization

Stabilization protocol:

- **Stabilize()**: n asks its successor for its predecessor p and decides whether p should be n‘s successor instead (this is the case if p recently joined the system).
- **Notify()**: notifies n‘s successor of its existence, so it can change its predecessor to n
- **Fix_fingers()**: updates finger tables
The Chord algorithm – Node joins and stabilization
The Chord algorithm –
Node joins and stabilization

- N26 joins the system
- N26 acquires N32 as its successor
- N26 notifies N32
- N32 acquires N26 as its predecessor
The Chord algorithm – Node joins and stabilization

- N26 copies keys
- N21 runs stabilize() and asks its successor N32 for its predecessor which is N26.
The Chord algorithm – Node joins and stabilization

- N21 acquires N26 as its successor
- N21 notifies N26 of its existence
- N26 acquires N21 as predecessor
The Chord algorithm – Impact of node joins on lookups

- All finger table entries are correct => $O(\log N)$ lookups
- Successor pointers correct, but fingers inaccurate => correct but slower lookups
The Chord algorithm – Impact of node joins on lookups

- Incorrect successor pointers => lookup might fail, retry after a pause
- But still correctness!
The Chord algorithm – Impact of node joins on lookups

- Stabilization completed => no influence on performance
- Only for the negligible case that a large number of nodes joins between the target's predecessor and the target, the lookup is slightly slower
- No influence on performance as long as fingers are adjusted faster than the network doubles in size
The Chord algorithm – Failure of nodes

- Correctness relies on correct successor pointers
- What happens, if N14, N21, N32 fail simultaneously?
- How can N8 acquire N38 as successor?
The Chord algorithm – Failure of nodes

- Correctness relies on correct successor pointers
- What happens, if N14, N21, N32 fail simultaneously?
- How can N8 acquire N38 as successor?
The Chord algorithm – Failure of nodes

- Each node maintains a successor list of size \( r \)
- If the network is initially stable, and every node fails with probability \( \frac{1}{2} \), `find_successor` still finds the closest living successor to the query key and the expected time to execute `find_successor` is \( O(\log N) \)
- Proofs are in the paper
Massive failures have little impact

$(1/2)^6$ is 1.6%
Overview

- Introduction
- The Chord Algorithm
  - Construction of the Chord ring
  - Localization of nodes
  - Node joins and stabilization
  - Failure/Departure of nodes
- Applications
- Summary
- Questions
Applications: 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, powerful protocol
- Only operation: map a key to the responsible node
- Each node maintains information about $O(\log N)$ other nodes
- Lookups via $O(\log N)$ messages
- Scales well with number of nodes
- Continues to function correctly despite even major changes of the system
Questions?
Thanks!