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

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

Presented by Bo Han for CMSC 818T
(based on Robert Morris’s talk on SIGCOMM 2001)
Introduction

- Core operation in peer-to-peer systems is to efficiently locate the node that stores a particular data item.

- Chord is a scalable distributed protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures.

- Only one operation: given a key, it maps the key onto a node.

- Simplicity, provable correctness, and provable performance.
The lookup problem

Publisher

Internet

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

Client
Lookup("title")
Centralized lookup (Napster)

- Publisher@\(N_4\)
  - Key="title"
  - Value=MP3 data...

- \(\text{SetLoc("title", N4)}\)

- DB

- \(N_1\) \(N_2\) \(N_3\)

- \(N_4\) \(N_6\) \(N_7\) \(N_8\) \(N_9\)

- Client

- \(\text{Lookup("title")}\)

Simple, but \(O(N)\) states and a single point of failure
Flooded queries (Gnutella)

Robust, but worst case $O(N)$ messages per lookup
Routed queries (Freenet, Chord, etc.)

Client

Lookup("title")

Publisher

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

N1 → N2 → N3

N4

N6 → N7 → N8

N9
Related Work

- Freenet (Clarke, Sandberg, Wiley, Hong)
- CAN (Ratnasamy, Francis, Handley, Karp, Shenker)
- Pastry (Rowstron, Druschel)
- Tapestry (Zhao, Kubiatowicz, Joseph)

... ... 

- Chord emphasizes simplicity
Design Objectives

- Load Balance: Distributed hash function spreads keys evenly over the nodes (Consistent hashing).
- Decentralization: Fully distributed (Robustness).
- Scalability: Lookup grows as a log of number of nodes.
- Availability: Automatically adjusts internal tables to reflect changes.
- Flexible Naming: No constraints on key structure.
Applications

- Lookup(key) algorithm that yields the IP address of the node responsible for the key.

- Notify the application of changes in the set of keys that the node is responsible for.

- Example applications:
  - Cooperative Mirroring
  - Time-shared storage
  - Distributed indexes
  - Large-Scale combinatorial search
Routing challenges

- Define a useful key nearness metric.
- Keep the hop count small.
- Keep the routing tables small.
- Stay robust despite rapid changes.
Chord properties

- Efficient: $O(\log(N))$ messages per lookup.
- Scalable: $O(\log(N))$ states per node.
- Robust: survives massive failures, join or leave. $O(\log^2(N))$ messages.
- An $N^{th}$ node joins (or leaves), only an $O(1/N)$ keys are moved to a different location.

Proofs are in paper / tech report.

(Assuming no malicious participants)
Chord overview

- Provides peer-to-peer hash lookup:
  - Lookup(key) → IP address.
  - Chord does not store the data.

- How does Chord route lookups?

- How does Chord maintain routing tables?

- How does Chord cope with changes in membership?
Chord IDs

- m-bit identifier space for both keys and nodes.
- Key identifier = SHA-1(key).
- Node identifier = SHA-1(IP address).
- Both are uniformly distributed.

- How to map key IDs to node IDs?
Consistent hashing [Karger 97]

A key is stored at its successor: node with next higher ID.
Basic lookup

“Where is key 80?”

“N90 has K80”
Simple lookup algorithm

Lookup(my-id, key-id)

n = my successor

if my-id < n < key-id

call Lookup(id) on node n  // next hop

else

return my successor  // done

Correctness depends only on successors
“Finger table” allows log(N)-time lookups

Every node knows m other nodes in the ring
Finger $i$ points to successor of $n+2^{i-1}$

Each node knows more about portion of circle close to it
Lookup with fingers

Lookup(my-id, key-id)

look in local finger table for

highest node n s.t. my-id < n < key-id

if n exists

call Lookup(id) on node n  // next hop

else

return my successor  // done
Lookups take $O(\log(N))$ hops
1. Each node’s successor is correctly maintained.
2. For every key k, node successor(k) is responsible for k.
Join (2)

2. N36 sets its own successor pointer

Initialize the new node finger table
3. Set N25's successor pointer

Join (3)

Update finger pointers of existing nodes
Join (4)

4. Copy keys 26..36 from N40 to N36

Transferring keys
Stabilization Protocol

- To handle concurrent node joins/fails/leaves.
- Keep successor pointers up to date, then verify and correct finger table entries.
- Incorrect finger pointers may only increase latency, but incorrect successor pointers may cause lookup failure.
- Nodes periodically run stabilization protocol.
- Won’t correct a Chord system that has split into multiple disjoint cycles, or a single cycle that loops multiple times around the identifier space.
Failures might cause incorrect lookup

N80 doesn’t know correct successor, so incorrect lookup
Solution: successor lists

- 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.
Choosing the successor list length

- Assume 1/2 of nodes fail.

- \( P(\text{successor list all dead}) = (1/2)^r \)
  - I.e. \( P(\text{this node breaks the Chord ring}) \)
  - Depends on independent failure

- \( P(\text{no broken nodes}) = (1- (1/2)^r)^N \)
  - \( r = 2\log(N) \) makes prob. = \( 1 - 1/N \)
Lookup with fault tolerance

Lookup(my-id, key-id)

look in local finger table and successor-list

for highest node n s.t. my-id < n < key-id

if n exists

call Lookup(id) on node n // next hop

if call failed,

remove n from finger table

return Lookup(my-id, key-id)

else return my successor // done
Simulation overview

- Quick lookup in large systems.
- Low variation in lookup costs.
- Robust despite massive failure.
- Iterative implementation.

- 10,000 nodes, No. of keys range from $10^5$ to $10^6$.

Experiments confirm theoretical results
No. of Keys per Node
Chord lookup cost is $O(\log N)$

Constant is $1/2$
Chord Summary

- Chord provides peer-to-peer hash lookup.
- Efficient: $O(\log(n))$ messages per lookup.
- Robust as nodes fail and join.
- Good primitive for peer-to-peer systems.

http://www.pdos.lcs.mit.edu/chord
Misc.

- Sound theoretical work (about 1173 citations).
- Has been used in: CFS (SOSP 2001) and Ivy (OSDI 2002)
- Ring Partitions might pose a problem.
- Scalability of Stabilization protocol.
  - How often does the stabilization procedure need to run?
  - How to balance consistence and network overhead?
- Virtualized ID space lacks locality characteristics.
  - Physical topology of the underlying IP network.
Wide-area Cooperative Storage with CFS

Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica

Presented by Randy Baden
CFS

- Cooperative File System
- Basic Idea: Build “read-only” block-level storage on top of Chord
  - Scales indefinitely...
  - Decentralized
- Add extra features to make Chord work better for a file system
Layers

<table>
<thead>
<tr>
<th>Layer</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS</td>
<td>Interprets blocks as files; presents a file system interface to applications.</td>
</tr>
<tr>
<td>DHash</td>
<td>Stores unstructured data blocks reliably.</td>
</tr>
<tr>
<td>Chord</td>
<td>Maintains routing tables used to find blocks.</td>
</tr>
</tbody>
</table>

- **Three layers:**
  - FS : Root blocks and data blocks, standard stuff
  - Chord : Works mostly like normal Chord, manages finger tables
  - DHash : Major contributions, extra functionality, overrides some Chord operations
FS - Block Storage

- Recall Chord's 160-bit ID space using SHA-1
- Signed blocks
  - Root blocks
  - Chord ID = H(publisher's public key)
- Unsigned blocks
  - Directory blocks, inode blocks, data blocks
  - Chord ID = H(block contents)
FS Layer: Read-only

- System is read-only...
FS Layer: Read-only

- System is read-only, but publishers can replace their directory structure, since only they can sign it.
FS Layer: Read-only

- System is read-only, but data expires unless publisher periodically refreshes it.
Desired Properties

- **Fulfilled by Chord**
  - Decentralized
  - Scalability
  - Availability – replicas in different network locations
  - Persistence – like availability, for an agreed-on time

- **Fulfilled by DHash**
  - Load Balance – block storage, caching, virtual servers
  - Quotas – protect against malicious use
  - Efficiency – delay comparable to FTP...
Chord

• **One modification: Server Selection**
  
  – Preferentially contact nodes based on latency
  – Cost for a hop is the delay for that hop plus the average delay times the expected remaining hops
    
    • Delay for a hop based on observed latency
    • Average delay based on average observed latency for all
    • Expected remaining hops based on arc length covered by successor list, and distance between hop and destination

  – No extra measurements!
  – Latency measurements don't generalize
DHash – Replication

- Replicate on k successors (same as Chord)
  - Suggest using coded pieces of blocks to conserve storage, but don't
  - Preferential replica downloads
    - Download from the successor replica with the lowest latency measurement from the predecessor for the ID
    - “works best when proximity in the underlying network is transitive”
**DHash – Caching**

- **Override Chord lookups**
  - Return the predecessor for the ID, and the successor list should be a list of servers with a replica of it.
  - Also, when doing a lookup, *cache the results of the lookup on all of the nodes visited on the lookup path*.

- **Cache items are evicted in LRU fashion**
  - Nodes near end of lookup path are in most lookup paths.
    - Nodes close to the target continue to cache the data longer.
    - Nodes farther away cache only popular data.
DHash – Caching (more)

- **Freshness**
  - Data blocks can't become stale; any change will cause a change in the content hash
  - Root blocks could become stale, but point to internally consistent data, and clients can check freshness

- **Caching vs. Replication**
  - Both store data in more places
  - # cached copies of a block can fall to 0
  - Replicas stored in predictable places
DHash – Load Balance

- Use virtual servers to manage storage and network capacity
  - Chord ID = H(IP + virtual server number)
  - Virtual server can increase number of hops in lookup, so allow virtual servers to share information
  - Server can use estimate of total nodes in the network to decide how many virtual servers to run
  - Automatic scaling – if whole system is overloaded, could lead to a cascade of deleted virtual servers
DHash – Quotas

- Propose a strategy for applying quotas
  - Decentralized approximation of reliable identification using IP addresses as identifiers
  - IPs can be forged, but can be limited by challenge-response
  - Can limit quota to linear in number of servers, or to a constant total amount
  - Didn’t implement
Implementation

- C++, using RPCs and SFS toolkit
- Chord maintains finger tables, uses SHA-1
- DHash
  - Each instance is a virtual server – they communicate with each other with RPCs
  - Implementation of Chord lookup algorithm using Chord's finger tables
- Client interface acts like NFS server
- Use UDP instead of TCP
Experiments

- Test each new feature individually
  - Server Selection has modest improvement, but degrades with high prefetch windows due to network congestion
  - Better than FTP...?
  - Lookups take $O(\log(N))$ RPCs... using standalone Chord
  - Virtual servers improve load balance
  - Caching reduces average # hops after just a few lookups
  - Lookup failures only occur when all of a block's replicas fail simultaneously
Caching Results

Figure 11: Impact of caching on successive client fetches of the same block. Each point is the average number of RPCs for 10 successive fetches from randomly chosen servers; the error bars indicate one standard deviation. The system has 1,000 servers.
Failure Results

Figure 13: Fraction of block request failures as a function of the fraction of 1,000 CFS servers that fail. Each data point is the average of 5 experiments involving 1,000 block lookups; the error bars indicate the minimum and maximum results.
Conclusion

- Basic idea of putting a file system on a DHT is kind of cool
- Some good ideas, including caching for popular files, and server selection
- Some rehashing of previous stuff – Chord performance and virtual servers in particular
- Some stuff glossed over – quotas, security