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

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan
MIT Laboratory for Computer Science

Presenters: Van Quach & Dale Pedzinski

Topics

- Introduction
- Related Work
- System Model
- Chord Protocol
- Concurrent Operations and Failures
- Simulation and Experimental Results
- Future Work & Conclusion
- Q & A
- References

Introduction

- What is Chord Protocol?
- Consistent Hashing
    - Node maintains information about only O(log N) nodes
    - Resolves all lookups via O(log N) messages
    - Node joins/leave result in no more than $O(\log^2 N)$ messages
- Main Features of Chord
    - Simplicity
    - Provable Correctness
    - Provable Performance
- Funded by DARPA and the Space and Naval Warfare Systems Center (SPAWAR), San Diego, under contract N66001-00-1-893

Related Work

- Key/Value Mappings
  - DNS hostname to IP address mapping
  - CAN
- Lookup Operations
  - Freenet peer-to-peer storage system
- Consistent Hashing
  - Ohaha system & Freenet-style query routing
  - Globe System
- Distributed data location protocol
  - Oceanstore
- Lookup Service
  - Napster

System Model

- Core Features
  - Load Balancing
  - Decentralization
  - Scalability
  - Availability
  - Flexible naming

- Applicable Applications
  - Cooperative Mirroring
  - Time-Shared Storage
  - Distributed Indexes
  - Large-Scale Combinatorial Search

Chord Protocol

“Chord Protocol defines how to find the location keys, how new nodes join the system, and how to recover from failure of existing nodes.”

What is included?

- Consistent Hashing
- Scalable Key Locations
- Node Joins

Consistent Hashing

“THEOREM 1. For any set of N nodes and K keys, with high probability:

1. Each Node is Responsible for at most (1 + e) K/N keys

2. When an (N + 1)st node joins or leaves the network, responsibility for O(K/N) keys changes hands (and only to or from the joining or leaving node). “

- hash function assigns each node and key an m-bit identifier
    - Like SHA-1
- node identifier is node’s hashed IP address
- key’s identifier is produced by hashing the key □
- Identifiers are ordered in an identifier circle modulo 2^m -1 □
- First node is successor(k) on the circle

Consistent Hashing Terms

- Key k is assigned to the first node with an identifier equal to or in the identifier space
  - This node = successor node of key
- Node Joins the Network
  - Keys on n’s successor move
- Node Leaves the Network
  - Keys move to successor

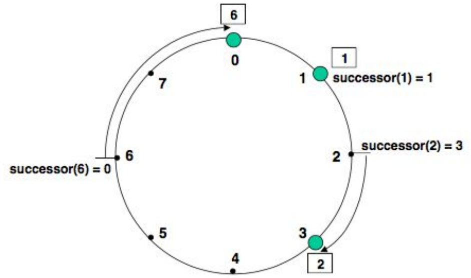


Figure 2: An identifier circle consisting of the three nodes 0, 1, and 3. In this example, key 1 is located at node 1, key 2 at node 3, and key 6 at node 0.

Scalable Key Locations

“THEOREM 2. With high probability (or under standard hardness assumptions), the number of nodes that must be contacted to find a successor in an N-node network is O(log N)”

- A node only needs to know its successor node on the circle □
- Queries utilize these successor nodes by finding the first node that succeeds the requested identifier □
- It is possible to traverse all N nodes

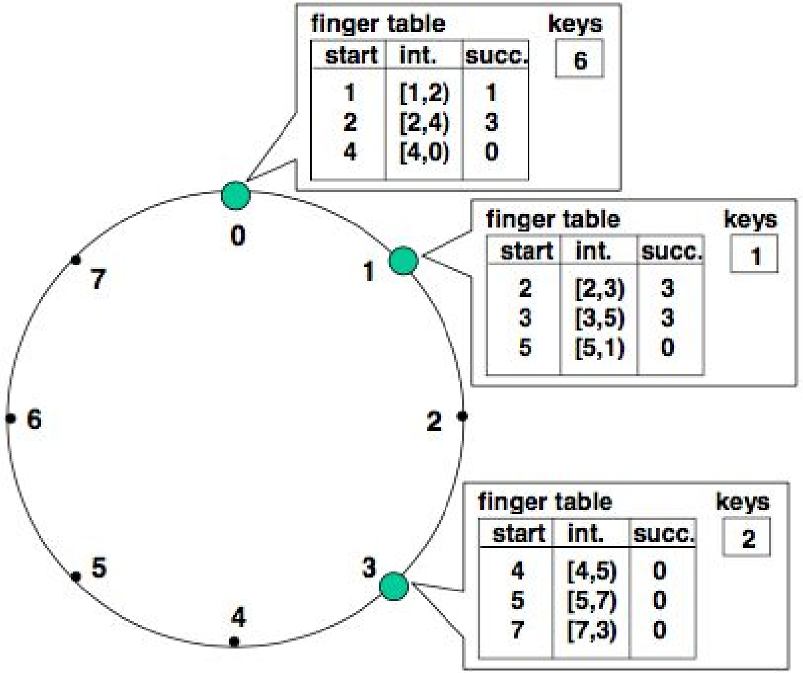
Scalable Key Locations Terms

- **m**=> number of bits in the key/node identifiers □
- **finger table**=> each node maintains a routing table with, at most, m entries (includes the Chord identifier and the IP address of the relevant node)
  - A node’s finger table generally does not contain enough information to determine the successor of a key
- **successor** => first entry of node n’s finger table, or the next node on the identifier circle
- **predecessor** => previous node on the identifier circle
- **ith finger** =>□ The ith entry of node n’s finger table contains the identity of the first node, s, that succeeds n by at least 2^{i-1} on the identifier circle, i.e., s=successor(n + 2i-1), where 1 ≤ i ≤m and all arithmetic is modulo 2 m

Search Process

- find _successor: finds immediate predecessor node of identifier
- Find_predecessor: moves forward around the Chord circle towards id

- Number of forwards necessary for search: O(log N)
- Average lookup time is 1/2 log N



finger table (node 0) — keys: 6

start	int.	succ.
1	[1,2)	1
2	[2,4)	3
4	[4,0)	0

finger table (node 1) — keys: 1

start	int.	succ.
2	[2,3)	3
3	[3,5)	3
5	[5,1)	0

finger table (node 3) — keys: 2

start	int.	succ.
4	[4,5)	0
5	[5,7)	0
7	[7,3)	0

(b)

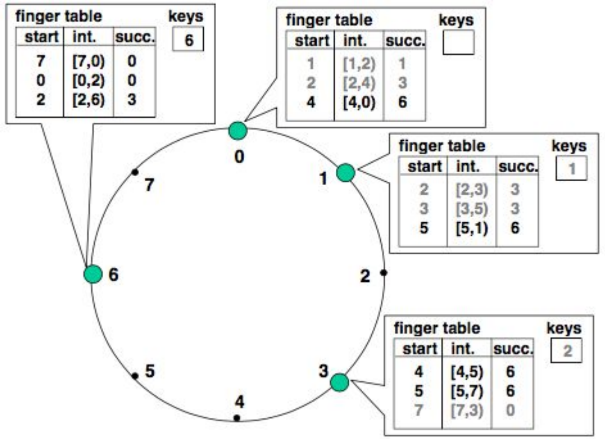
Node Joins

“THEOREM 3. 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 preserves two invariants:
  - node’s successor is correctly maintained
  - For every key k, node successor(k) is responsible for k
  - (In order for lookups to be fast, it is also desirable for the finger tables to be correct)
- Each node in Chord maintains a predecessor pointer
  - □contains the Chord identifier and IP address of the immediate predecessor of that node

Node Joins

- Chord must perform three tasks when a node joins the network:
    - Initialize the fingers and predecessor of node n
    - Update the fingers and predecessors of existing nodes to reflect the addition of n
    - Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node n is now responsible for



finger table (node 6) — keys: 6

start	int.	succ.
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

finger table (node 0) — keys:

start	int.	succ.
1	[1,2)	1
2	[2,4)	3
4	[4,0)	6

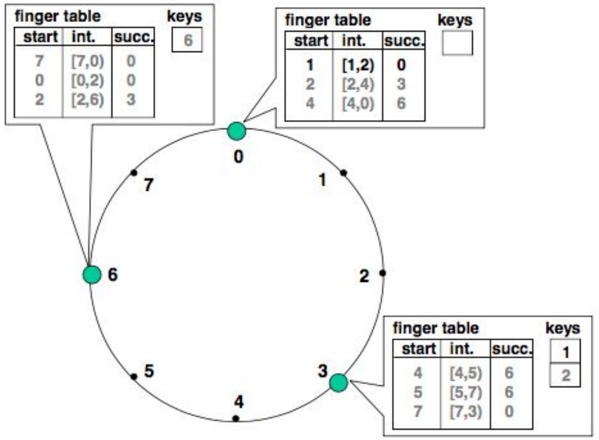
finger table (node 1) — keys: 1

start	int.	succ.
2	[2,3)	3
3	[3,5)	3
5	[5,1)	6

finger table (node 3) — keys: 2

start	int.	succ.
4	[4,5)	6
5	[5,7)	6
7	[7,3)	0

(a)



finger table (node 6) — keys: 6

start	int.	succ.
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

finger table (node 0) — keys:

start	int.	succ.
1	[1,2)	0
2	[2,4)	3
4	[4,0)	6

finger table (node 3) — keys: 1, 2

start	int.	succ.
4	[4,5)	6
5	[5,7)	6
7	[7,3)	0

(b)

Initialize the fingers and predecessor of node n

- Node n learns its predecessors by asking n’ to look them up
  - O(m log N)
  - Number of finger entries looked up via utilizing a check on ith finger entry: O(log N)
  - Overall time: O($\log^2$ N)

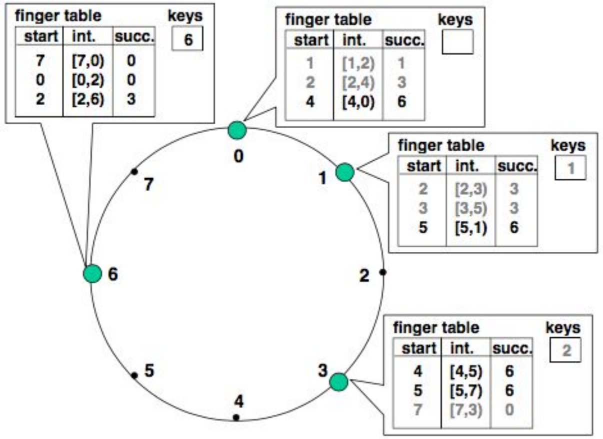
- Optimization: a newly joined node n can ask immediate neighbor for a copy of its finger table and it’s predecessor
  - Reduces overall time to O(log N)

Update the fingers and predecessors

- Node n will need to be entered into the finger tables of some existing nodes □
- Node n will become the ith finger of node p iff
    - p precedes n by at least 2^{i-1} □
    - The ith finger on node p succeeds n

- Find predecessor p of node n
    - Check if table needs updates
    - Repeat for predecessor of i

- Number of nodes that need an update: O(log n)
- Overall time it takes: O($\log^2$ N)



finger table (node 6) — keys: 6

start	int.	succ.
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

finger table (node 0) — keys:

start	int.	succ.
1	[1,2)	1
2	[2,4)	3
4	[4,0)	6

finger table (node 1) — keys: 1

start	int.	succ.
2	[2,3)	3
3	[3,5)	3
5	[5,1)	6

finger table (node 3) — keys: 2

start	int.	succ.
4	[4,5)	6
5	[5,7)	6
7	[7,3)	0

(a)

Transfer State

- Move data, associated with each key, over to new node
- Node n only needs to contact one node to transfer keys to it

Dynamic Operations and Failures

- Practical issues:
    a. Nodes join system concurrently
    b. Nodes fail or leave voluntarily

Stabilization protocol

Main Goal: Keep nodes’ successor pointers up to date

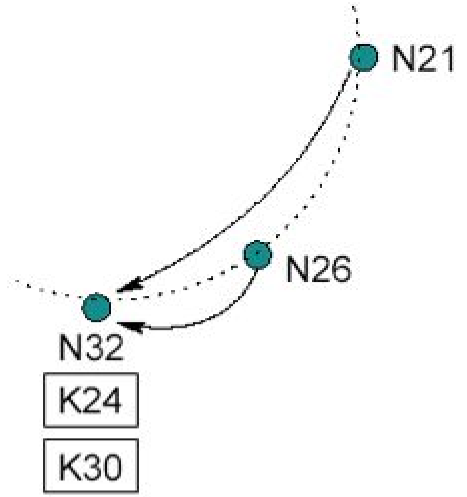
Stabilization scheme:

1. join()
2. stabilize()
3. notify()
4. fix_fingers()

join()

- Asks m to find the immediate successor of n.
- Doesn't make rest of the network aware of n.

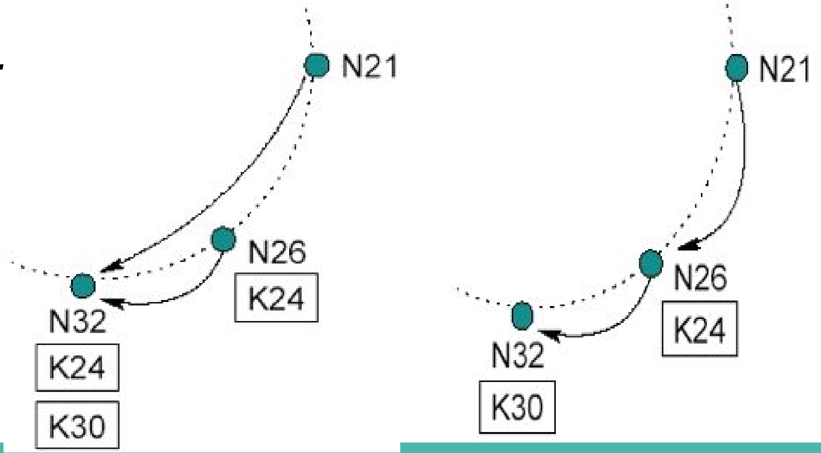
```
n.join(m)
  predecessor = nil;
  successor = m.find_successor(n);
```



stabilize()

- Called periodically to learn about new nodes
- Asks n's immediate successor about successor's predecessor p
  - Checks whether p should be n's successor instead
  - Also notifies n's successor about n's existence, so that successor may change its predecessor to n, if necessary

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



notify()

- m thinks it might be n’s predecessor

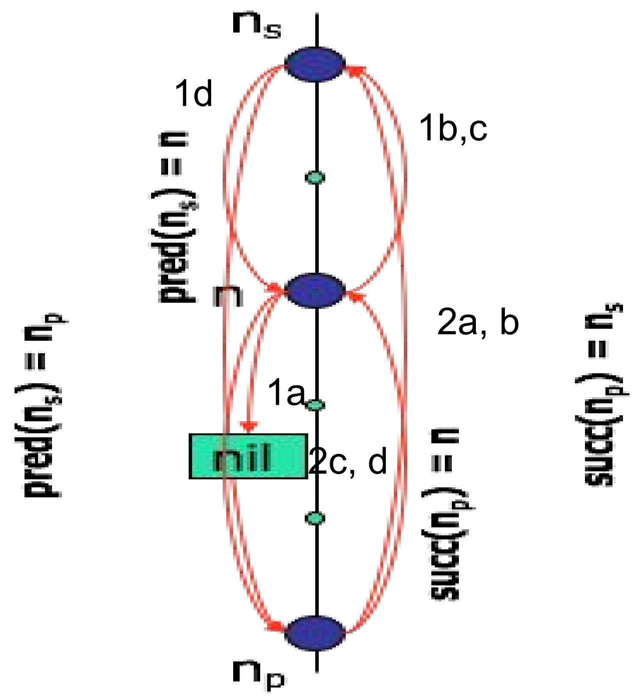
```
n.notify(m)
  if (predecessor is nil or m ∈ (predecessor, n))
      predecessor = m;
```

fix_fingers()

- Periodically called to make sure that finger table entries are correct
  - New nodes initialize their finger tables
  - Existing nodes incorporate new nodes into their finger tables

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

Stabilization after join



1. **n joins**
   a. predecessor = nil
   b. n acquires n_s as successor
   c. n notifies n_s being the new predecessor
   d. n_s acquires n as its predecessor
2. **n_p runs stabilize**
   a. n_p asks n_s for its predecessor (now n)
   b. n_p acquires n as its successor
   c. n_p notifies n
   d. n will acquire n_p as its predecessor
3. **all predecessor and successor pointers are now correct**
4. **fingers still need to be fixed, but old fingers will still work**

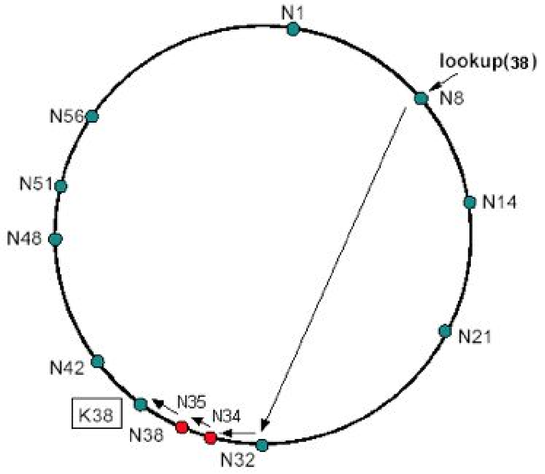
Lookups before stabilization finish

Three behaviors can occur before Chord ring stable:

1. All finger table entries are reasonably current ⇒ take O(logN) steps
2. Successor pointers are correct, but fingers inaccurate ⇒ correct lookups, but slower
3. Incorrect successor, or keys not yet migrated to newly joined nodes ⇒ lookups may fail, can pause and retry.

Failure Recovery

- Main goal: maintain correct successor pointers
- How:
  - Each node maintains a successor list of r nearest successors on the ring
  - If node n notices its successor has failed, it replaces the failed node with the 1st live entry
  - *stabilize* will correct finger table entries and successor-list entries pointing to failed node



Voluntary Node Departures

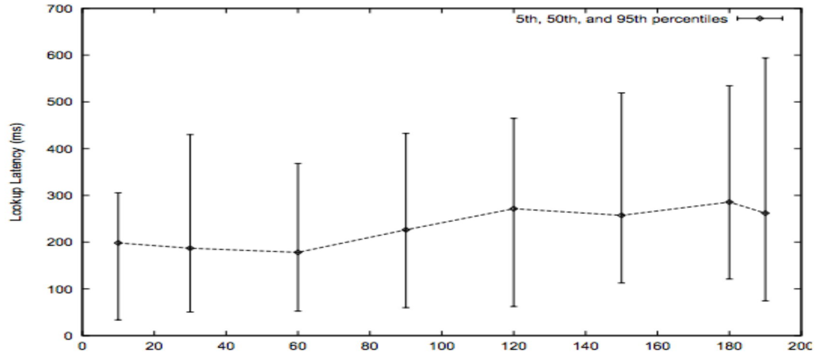
- Can be treated as node failures with enhancements
  - Transfer all keys to its successor
  - Notify its predecessor and successor

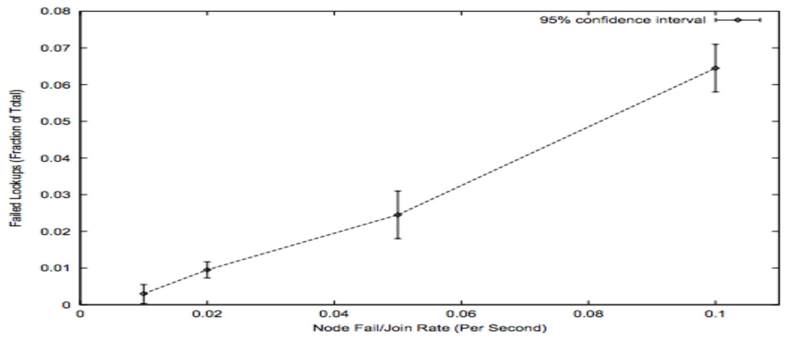
Chord - Fact

- Every node is responsible for about K/N keys
- When a node joins or leaves an N-node network, only O(K/N) keys change hands (and only to and from joining or leaving node)
- Lookups need O(logN) messages
- Stabilization for node leaving / joining need $O(\log^2 N)$ message

Experimental Results

1. Latency grows slowly with total number of nodes
2. Chord is robust for multiple node failures





Q & A

References

https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf