Distributed Hash Tables: Chord

Brad Karp
(with many slides contributed by Robert Morris)
UCL Computer Science

CS M038 / GZ06

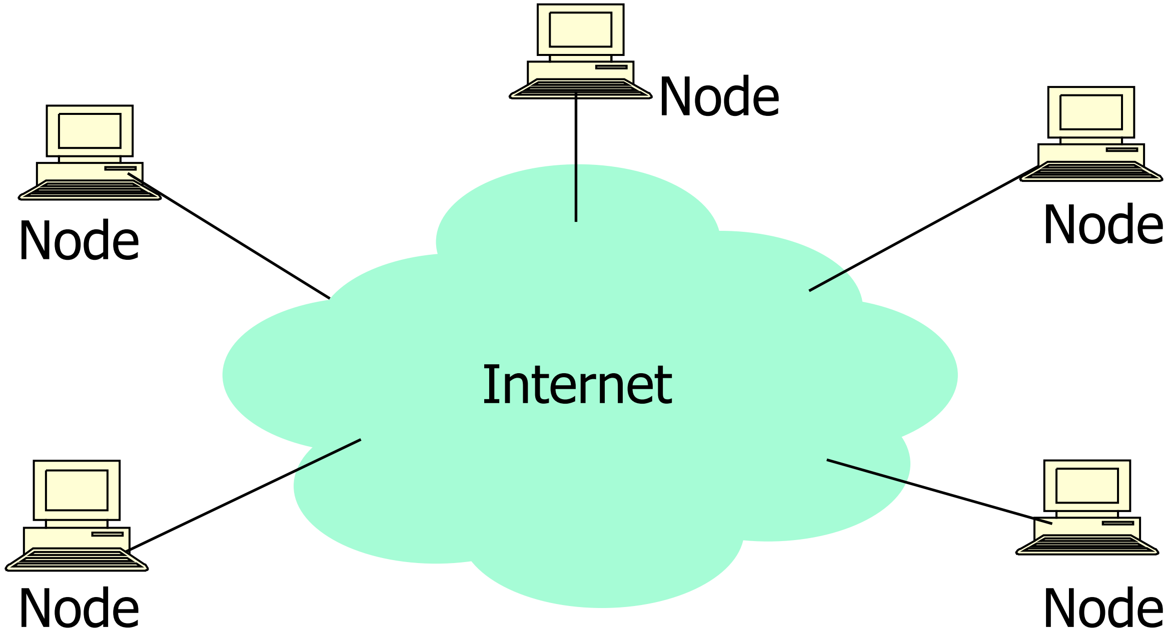
26th January, 2011

Today: DHTs, P2P

- Distributed Hash Tables: a building block
- Applications built atop them

- Your task: "Why DHTs?"
  - vs. centralized servers? (we'll return to this question at the end of lecture)
  - vs. non-DHT P2P systems?

What Is a P2P System?



- A distributed system architecture:
  - No centralized control
  - Nodes are symmetric in function
- Large number of unreliable nodes
- Enabled by technology improvements

The Promise of P2P Computing

- High capacity through parallelism:
  - Many disks
  - Many network connections
  - Many CPUs
- Reliability:
  - Many replicas
  - Geographic distribution
- Automatic configuration
- Useful in public and proprietary settings

What Is a DHT?

- Single-node hash table:

  key = Hash(name)

  put(key, value)

  get(key) -> value

  – Service: O(1) storage

- How do I do this across millions of hosts on the Internet?

  – *Distributed* Hash Table

What Is a DHT? (and why?)

Distributed Hash Table:

key = Hash(data)

lookup(key) -> IP address (Chord)

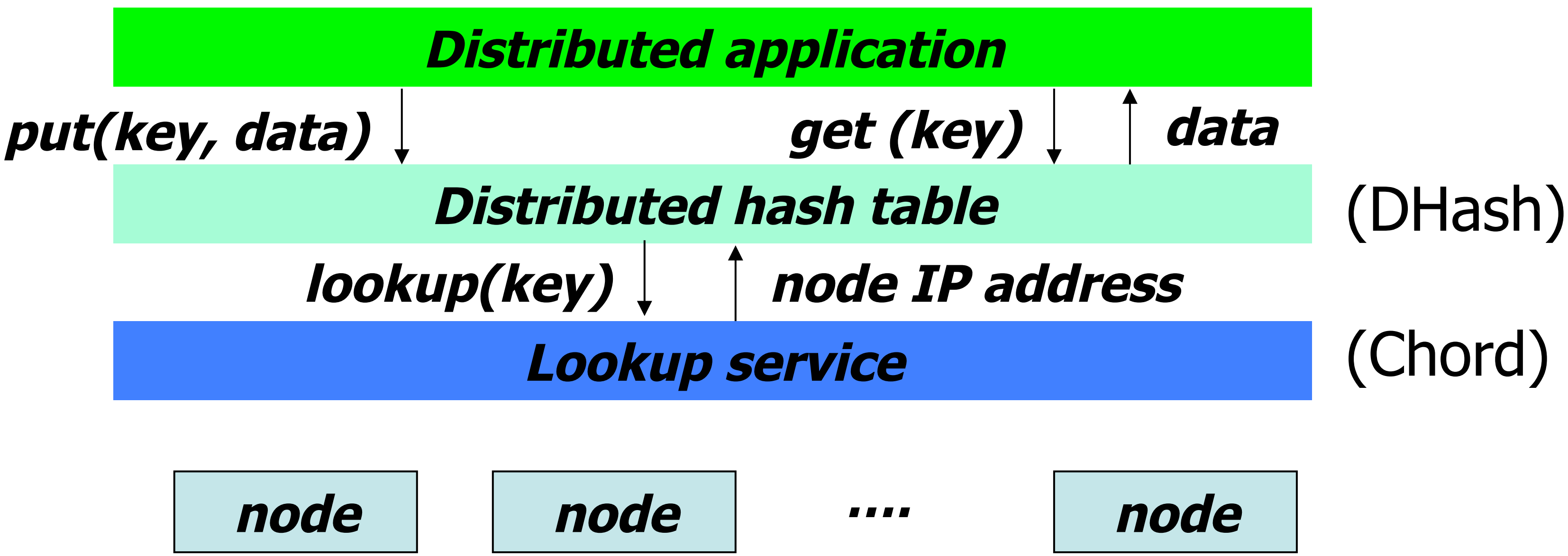
send-RPC(IP address, PUT, key, value)

send-RPC(IP address, GET, key) -> value

Possibly a first step towards truly large-scale distributed systems

- a tuple in a global database engine
- a data block in a global file system
- rare.mp3 in a P2P file-sharing system

DHT Factoring



- Application may be distributed over many nodes
- DHT distributes data storage over many nodes

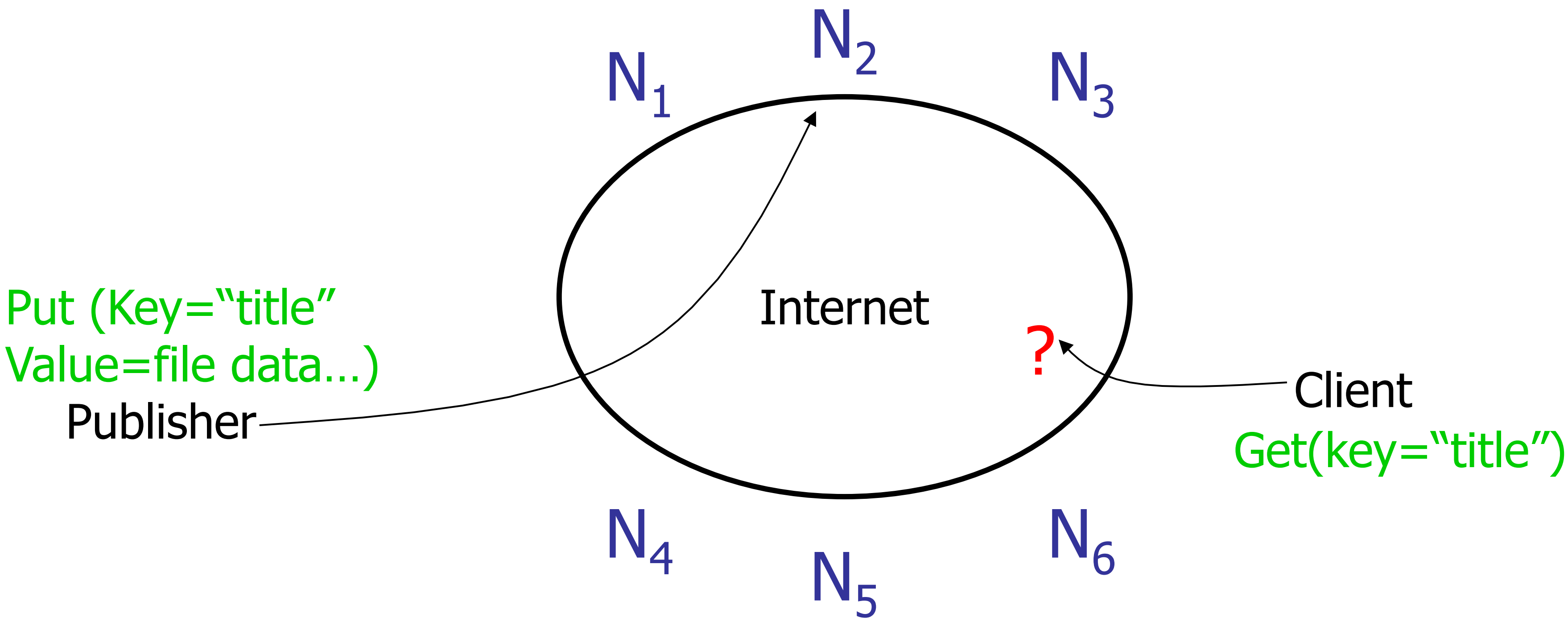
Why the put()/get() interface?

- API supports a wide range of applications
  - DHT imposes no structure/meaning on keys
- Key/value pairs are persistent and global
  - Can store keys in other DHT values
  - And thus build complex data structures

Why Might DHT Design Be Hard?

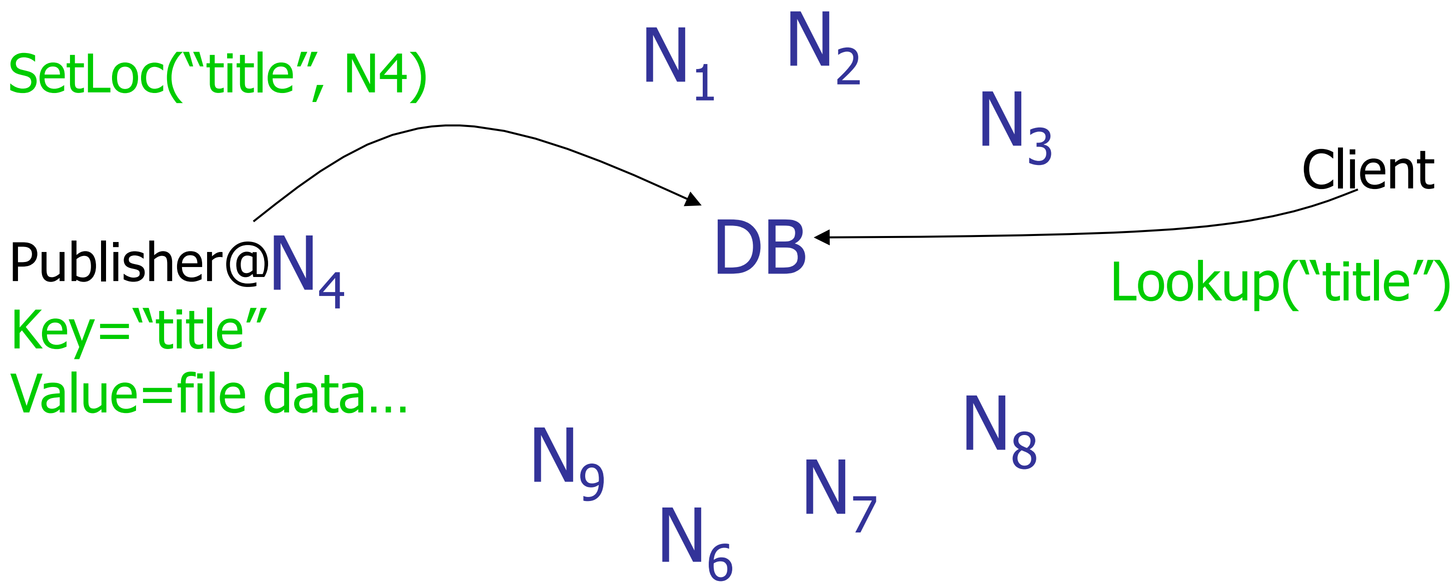
- Decentralized: no central authority
- Scalable: low network traffic overhead
- Efficient: find items quickly (latency)
- Dynamic: nodes fail, new nodes join
- General-purpose: flexible naming

The Lookup Problem



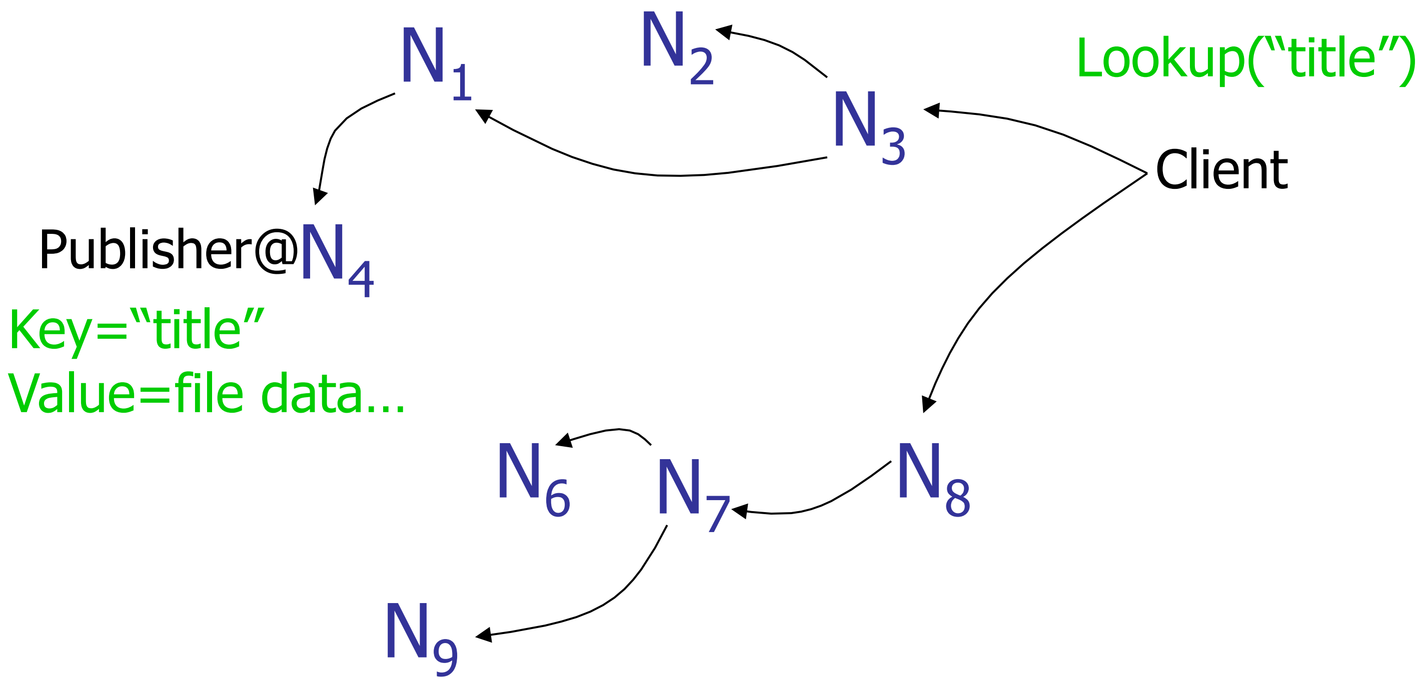
- At the heart of all DHTs

Motivation: Centralized Lookup (Napster)



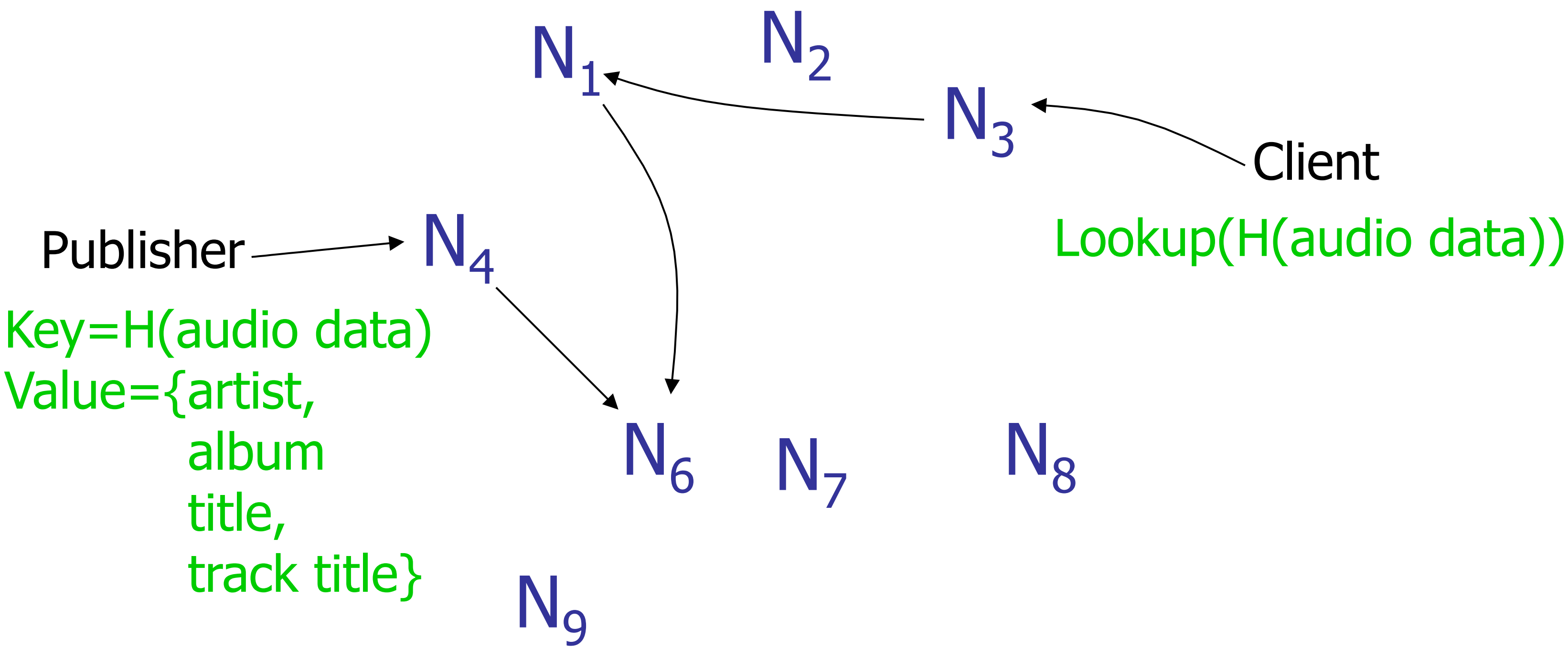
Simple, but O(N) state and a single point of failure

Motivation: Flooded Queries (Gnutella)



Robust, but worst case O(N) messages per lookup

Motivation: FreeDB, Routed DHT Queries (Chord, *&c.*)



DHT Applications

They're not just for stealing music anymore…

- global file systems [OceanStore, CFS, PAST, Pastiche, UsenetDHT]
- naming services [Chord-DNS, Twine, SFR]
- DB query processing [PIER, Wisc]
- Internet-scale data structures [PHT, Cone, SkipGraphs]
- communication services [i3, MCAN, Bayeux]
- event notification [Scribe, Herald]
- File sharing [OverNet]

Chord Lookup Algorithm Properties

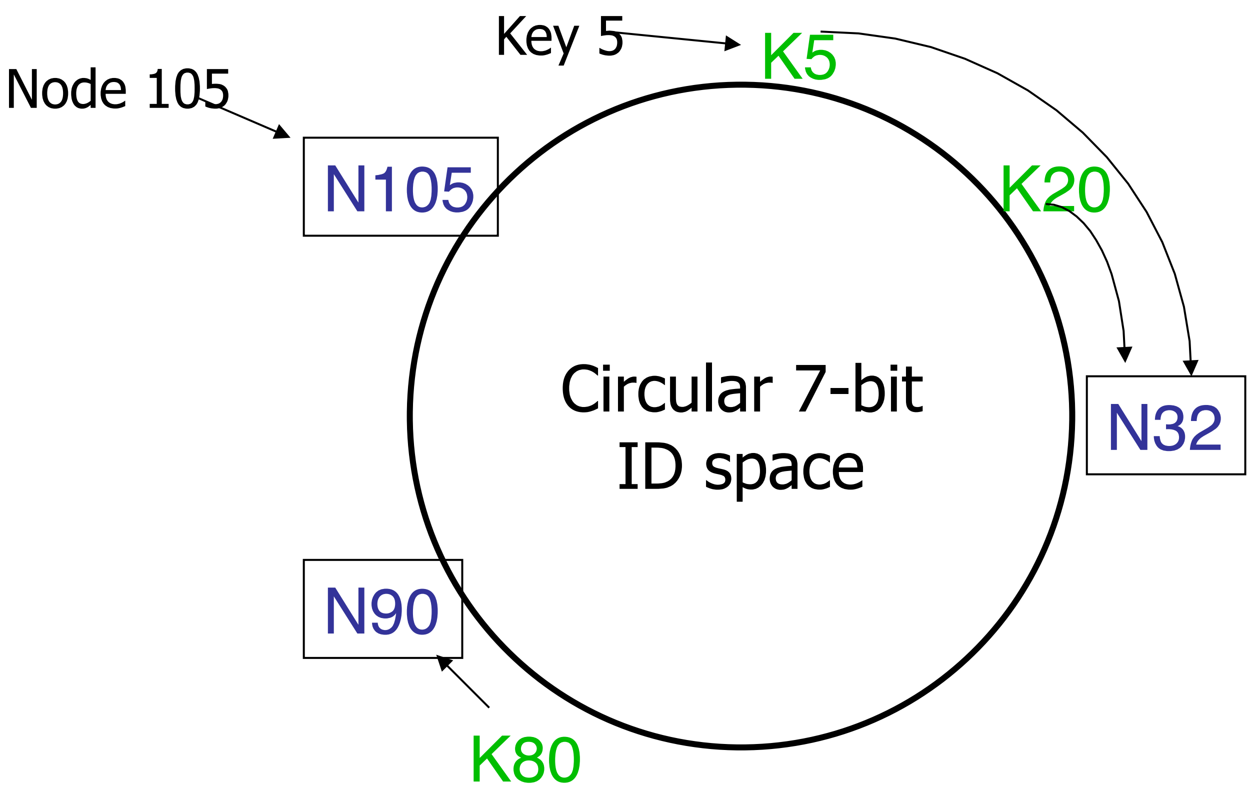
- Interface: lookup(key) → IP address
- Efficient: O(log N) messages per lookup
  - N is the total number of servers
- Scalable: O(log N) state per node
- Robust: survives massive failures
- Simple to analyze

Chord IDs

- Key identifier = SHA-1(key)
- Node identifier = SHA-1(IP address)
- SHA-1 distributes both uniformly

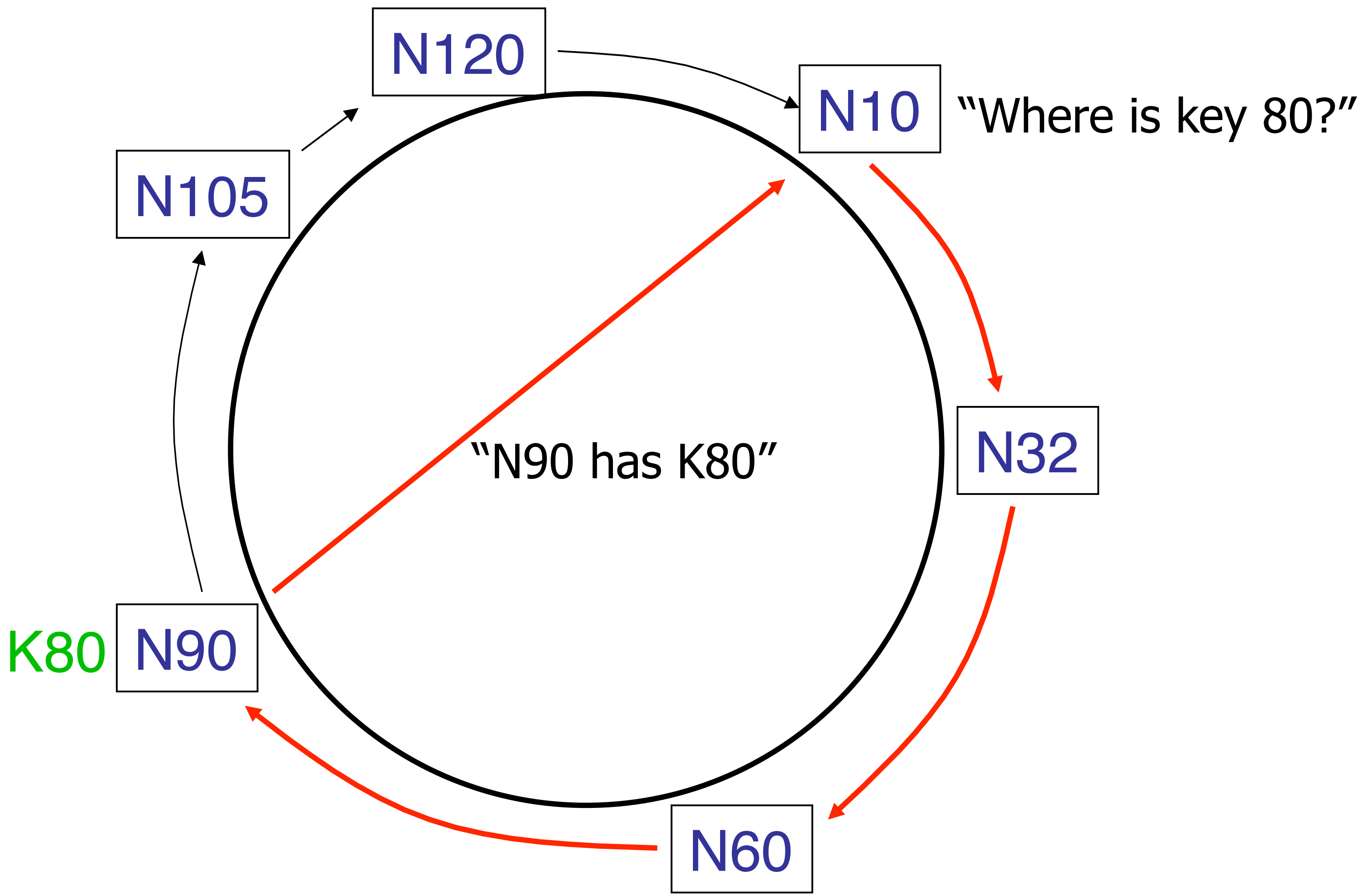
- How to map key IDs to node IDs?

Consistent Hashing [Karger 97]



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

Basic Lookup

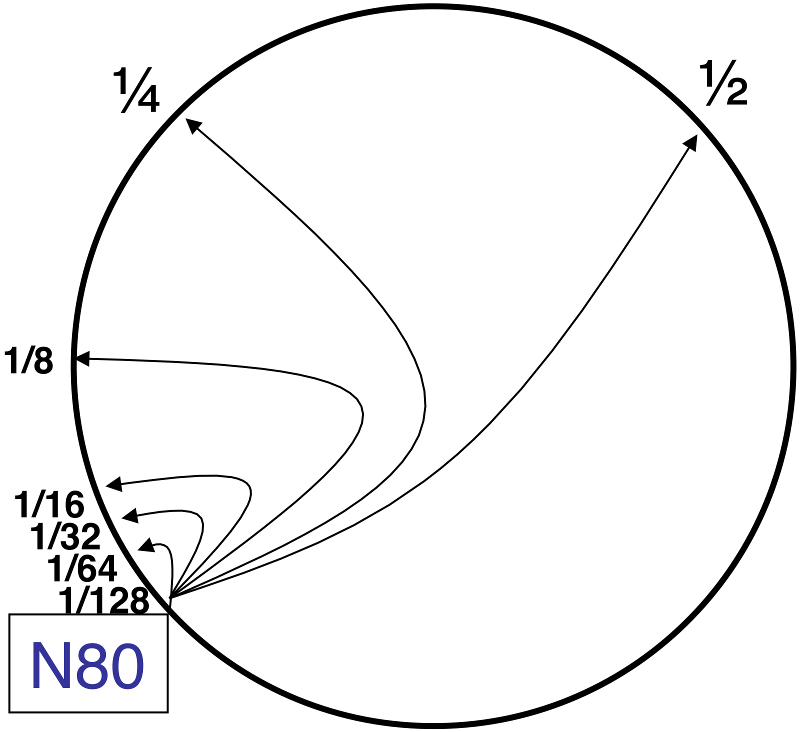


Simple lookup algorithm

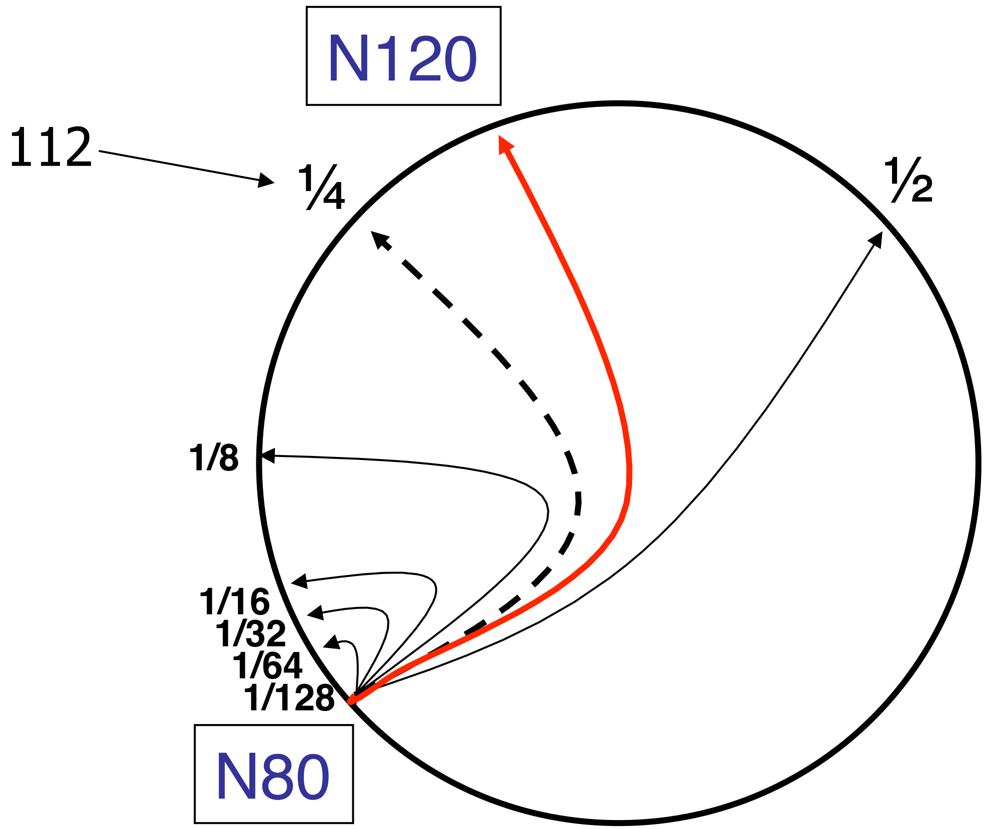
```
Lookup(my-id, key-id)
   n = my successor
   if my-id < n < key-id
      call Lookup(key-id) on node n   // next hop
   else
      return my successor             // done
```

- Correctness depends only on successors

"Finger Table" Allows log(N)-time Lookups



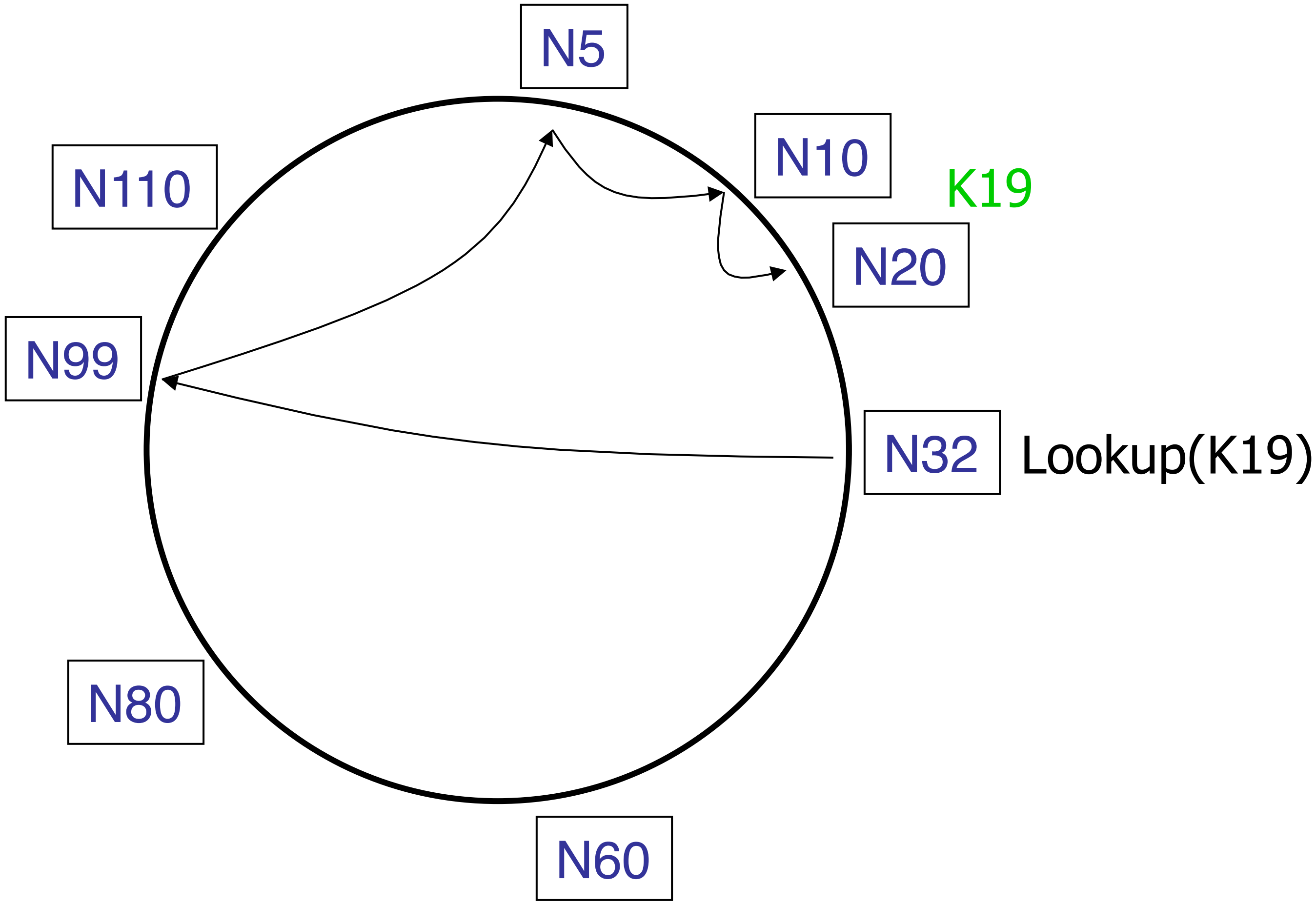
Finger *i* Points to Successor of $n+2^i$



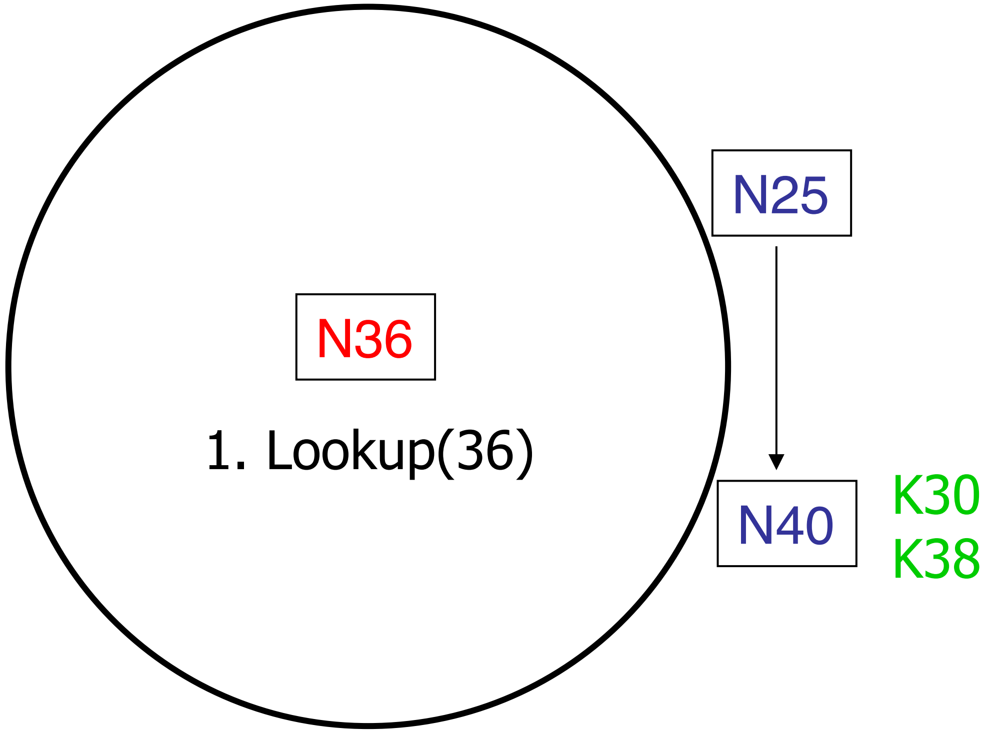
Lookup with Fingers

```
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(key-id) on node n        // next hop
  else
      return my successor                  // done
```

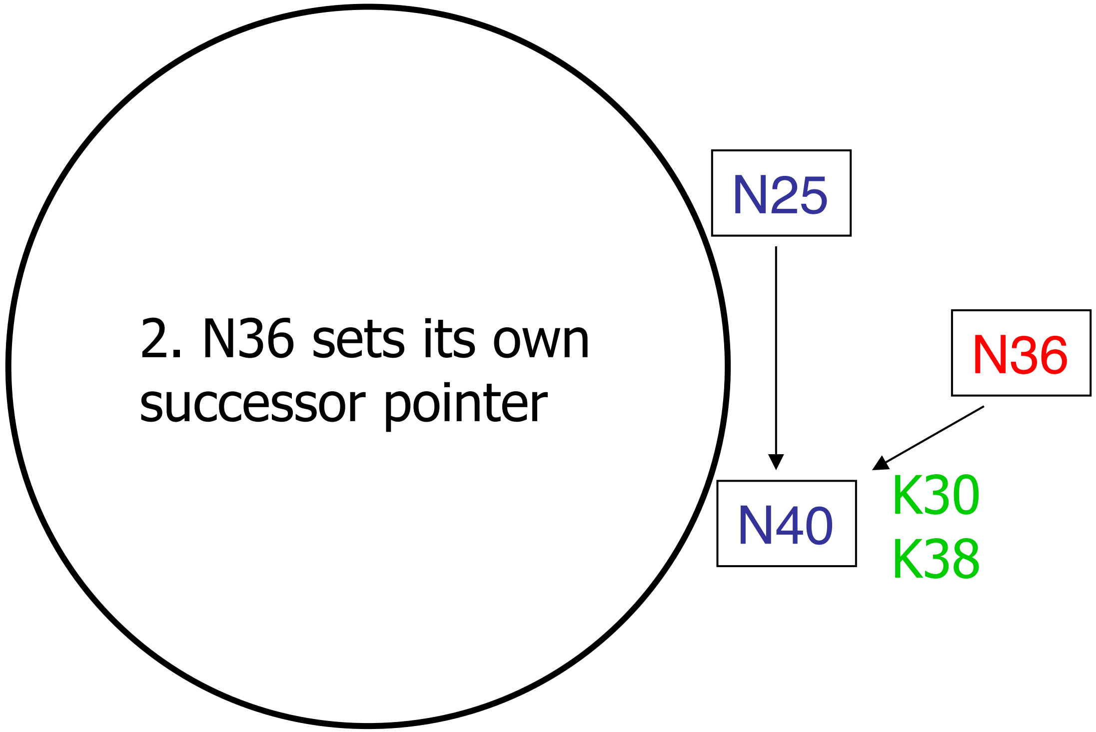
Lookups Take O(*log(N)*) Hops



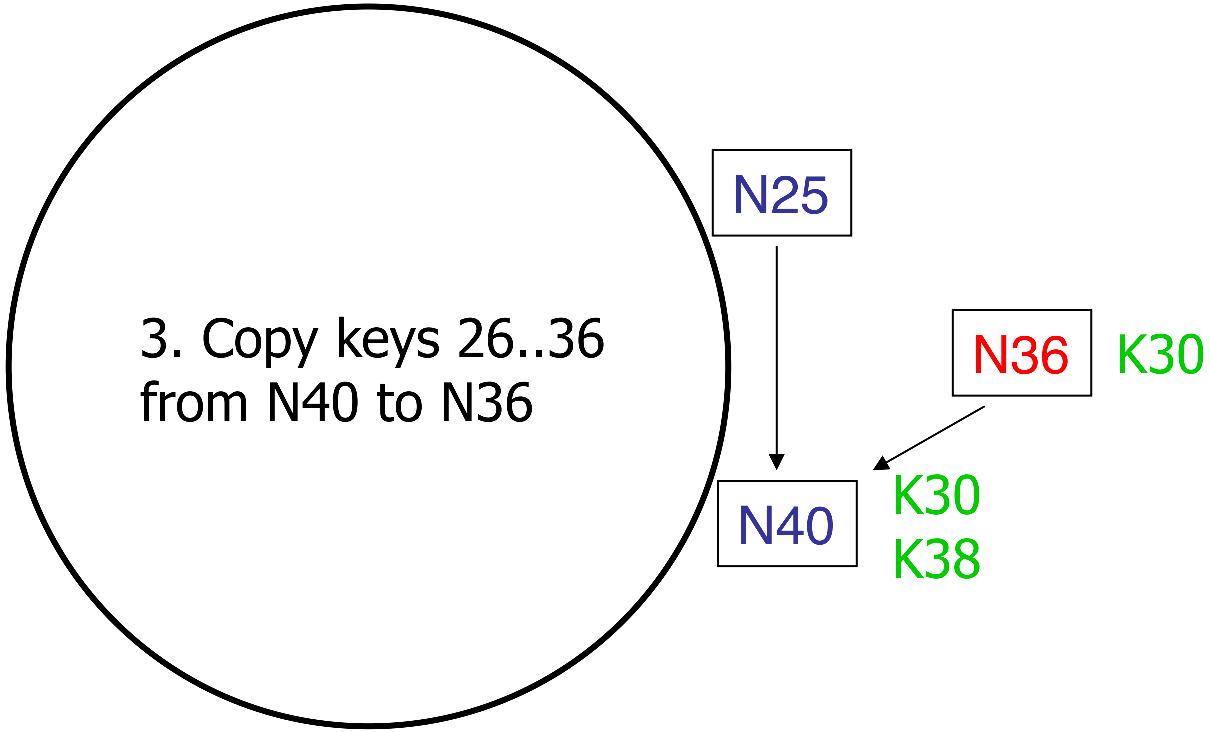
Joining: Linked List Insert



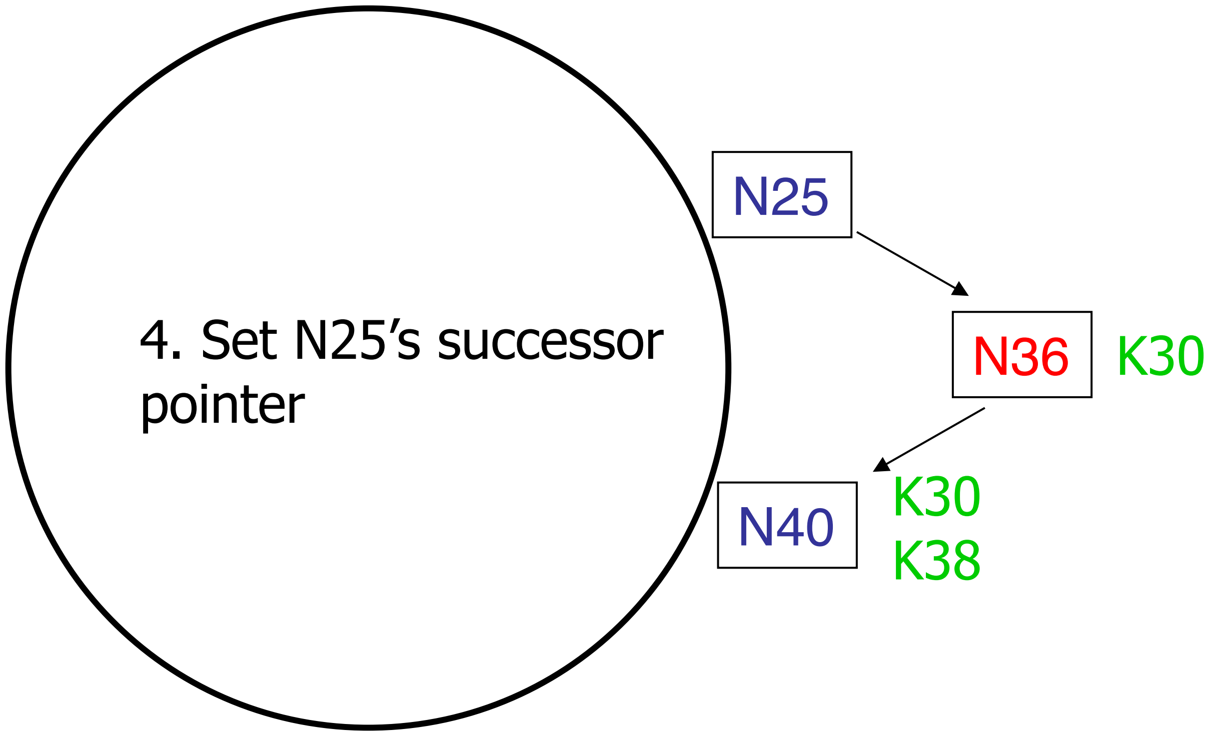
Join (2)



Join (3)

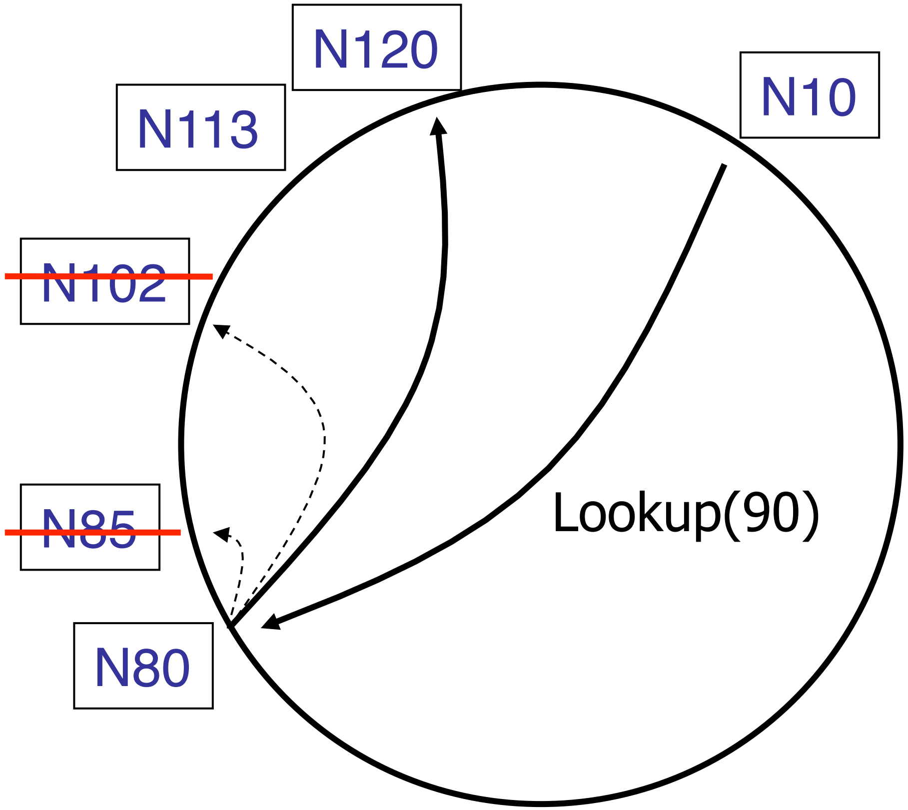


Join (4)



Predecessor pointer allows link to new host
Update finger pointers in the background
Correct successors produce correct lookups

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

Choosing Successor List Length

- Assume 1/2 of nodes fail
- P(successor list all dead) = $(1/2)^r$
  - i.e., P(this node breaks the Chord ring)
  - Depends on independent failure
- P(no broken nodes) = $(1 - (1/2)^r)^N$
  - $r = 2log(N)$ makes prob. = $1 - 1/N$

Lookup with Fault Tolerance

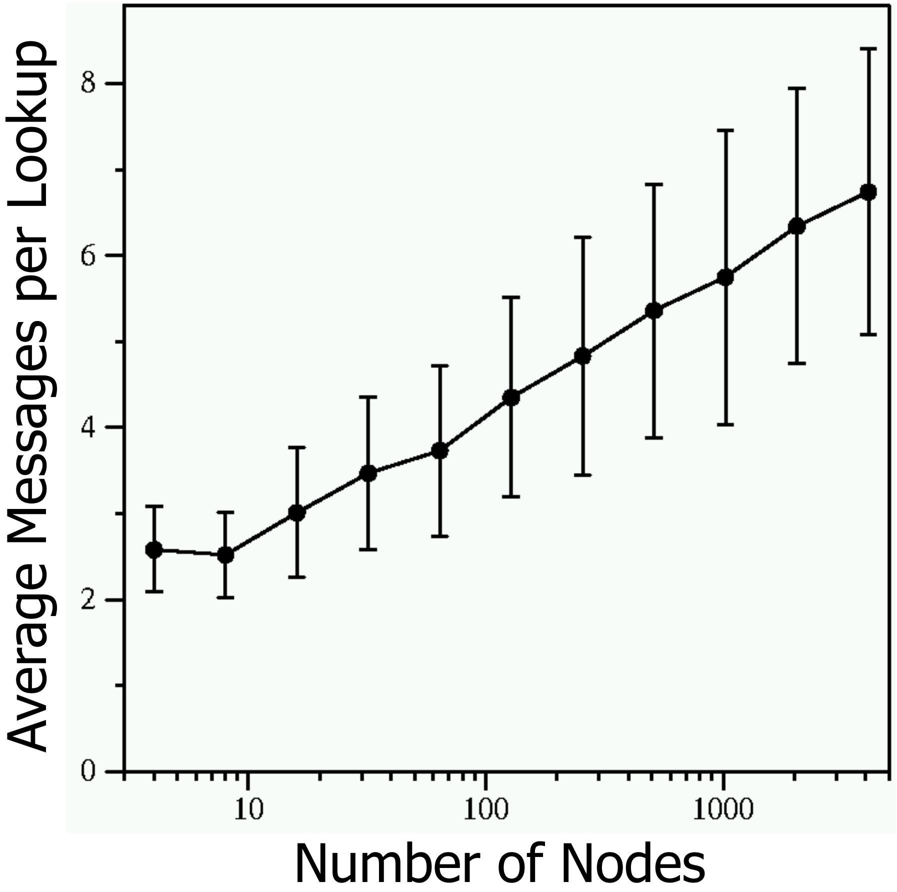
```
Lookup(my-id, key-id)
  look in local finger table and successor-list
      for highest node n s.t. my-id < n < key-id
  if n exists
      call Lookup(key-id) on node n        // next hop
      if call failed,
          remove n from finger table
          return Lookup(my-id, key-id)
  else return my successor                 // done
```

Experimental Overview

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

Experiments confirm theoretical results

Chord Lookup Cost Is O(log N)

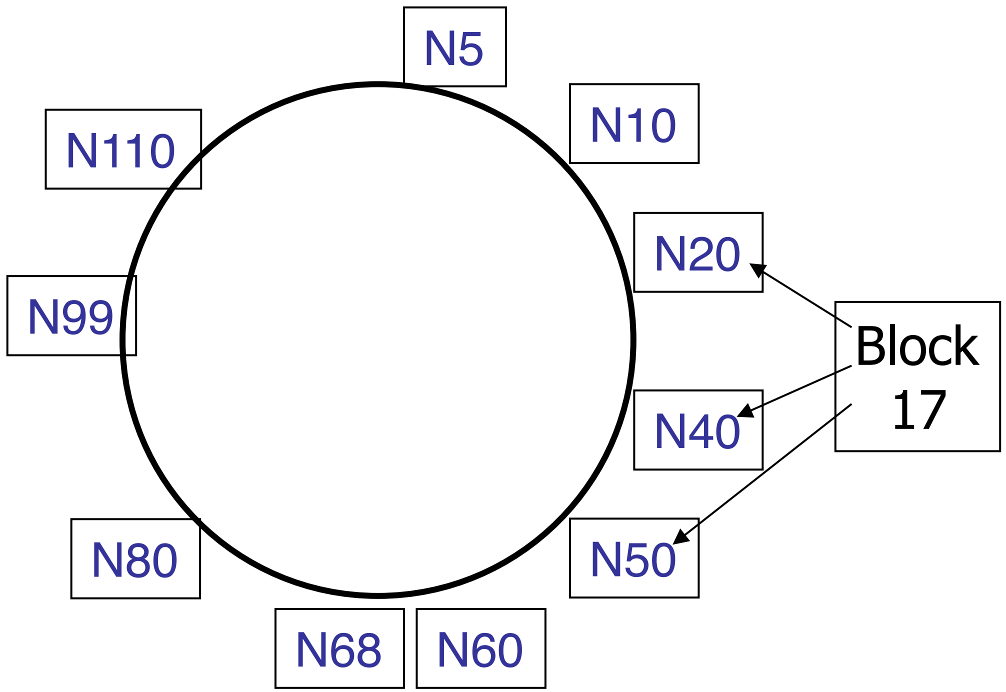


Constant is 1/2

Failure Experimental Setup

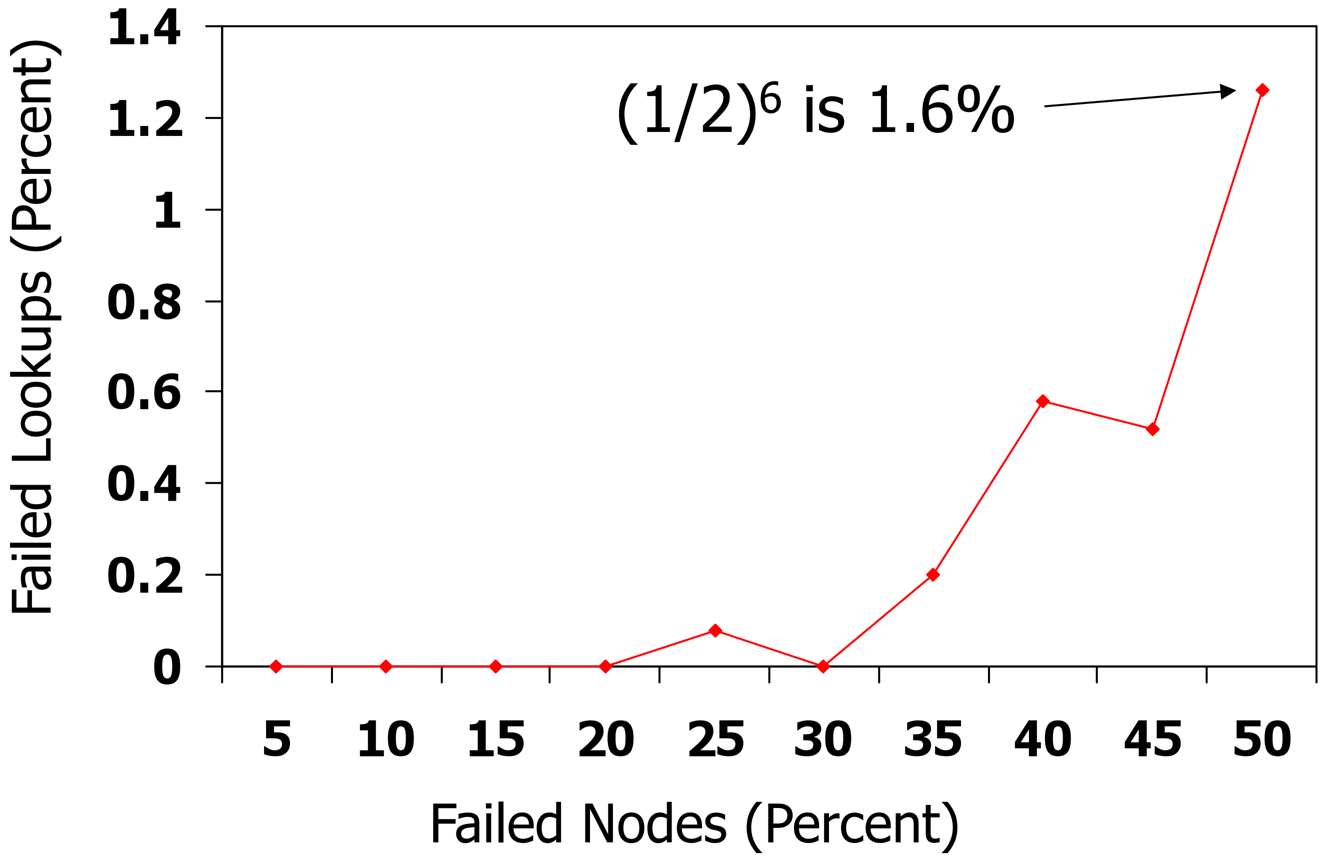
- Start 1,000 CFS/Chord servers
  - Successor list has 20 entries
- Wait until they stabilize
- Insert 1,000 key/value pairs
  - Five replicas of each
- Stop X% of the servers
- Immediately perform 1,000 lookups

DHash Replicates Blocks at *r* Successors



- Replicas are easy to find if successor fails
- Hashed node IDs ensure independent failure

Massive Failures Have Little Impact



DHash Properties

- Builds key/value storage on Chord
- Replicates blocks for availability
  - What happens when DHT partitions, then heals? Which (k, v) pairs do I need?
- Caches blocks for load balance
- Authenticates block contents

DHash Data Authentication

- Two types of DHash blocks:
  - Content-hash: key = SHA-1(data)
    immutable
  - Public-key: key is a public key, data are signed by that key
    read/write, but authenticated
- DHash servers verify before accepting
- Clients verify result of get(key)

DHTs: A Retrospective

- Original DHTs (CAN, Chord, Kademlia, Pastry, Tapestry) proposed in 2001-02
- Following 5-6 years saw proliferation of DHT-based applications:
  - filesystems (e.g., CFS, Ivy, Pond, PAST)
  - naming systems (e.g., SFR, Beehive)
  - indirection/interposition systems (e.g., i3, DOA)
  - content distribution systems (e.g., Coral)
  - distributed databases (e.g., PIER)
  - &c.…

DHTs: A Retrospective

Have these applications succeeded—are we all using them today?

Have DHTs succeeded as a substrate for applications?

- filesystems (e.g., CFS, Ivy, Pond, PAST)
- naming systems (e.g., SFR, Beehive)
- indirection/interposition systems (e.g., i3, DOA)
- content distribution systems (e.g., Coral)
- distributed databases (e.g., PIER)
- &c.…

What DHTs Got Right

- Consistent Hashing
  - simple, elegant way to divide a workload across machines
  - very useful in clusters: actively used today in Dynamo, FAWN-KV, ROAR, …
- Replication for high availability, efficient recovery after node failure
- Incremental scalability: “add nodes, capacity increases”
- Self-management: minimal configuration

What DHTs Got Right

- Consistent Hashing
  - simple, elegant way to divide a workload across machines

Unique trait: no single central server to shut down, control, or monitor

…well suited to "illegal" applications, be they sharing music or resisting censorship

- Incremental scalability: "add nodes, capacity increases"
- Self-management: minimal configuration

DHTs’ Limitations

- High latency between peers
- Limited bandwidth between peers (as compared to within a cluster)
- Lack of centralized control: another sort of simplicity of management
- Lack of trust in peers’ correct behavior
  - securing DHT routing hard, unsolved in practice