the Chord routing Protocol

A bipolar presentation by Chris Mantas

...in 2017

Siri, play me “Baby one more time” by Britney Spears

... back in 2001

How do i look for
“baby_one_more_time-Britney_Spears_1999_128kbps.mp3”
?

2001, SIGCOMM
>13K citations

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

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

MIT and Berkeley

From P2P to NoSQL

Kademlia is BitTorrent's DHT lookup service

Akamai,one of the largest CDNs, also uses a DHT architecture

- Dynamo
- Cassandra
- Riak
- Voldemord
- Memcached (?)

A peer-to-peer storage problem

- 1000 scattered music enthusiasts
- Willing to store and serve replicas
- How do you find the data?

Consistent Hashing Primer: keys and nodes

- values of keys are stored in nodes
  - In our example, keys might be "unique" filenames
- nodes IDs (N1 … Nn) (eg. IPs)
  - each stores a subset of key/values

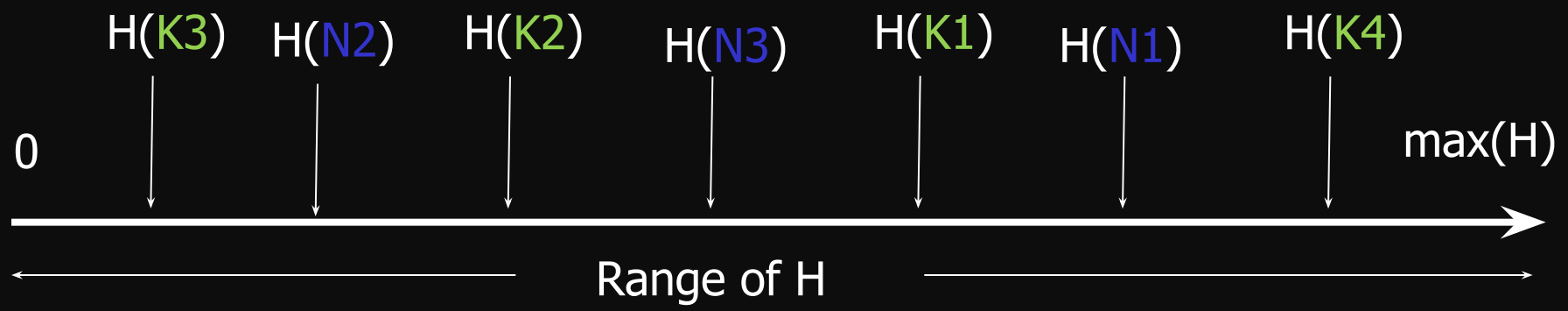
Consistent Hashing Primer: hash everything

H(x) : hash function

- Use for both keys and Node IDs
  - H("baby_...mp3") → number in H's range
  - H("12.34.5.6") → number in H's range
- Both keys and node IDs map in H's range
- Used as "ordering"

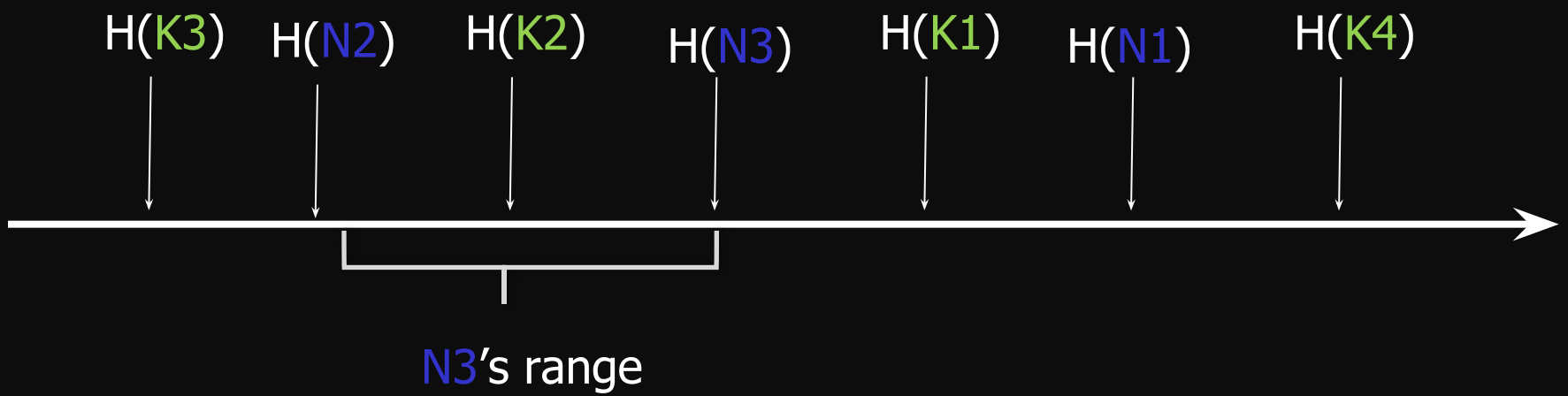
Consistent Hashing (Flat)

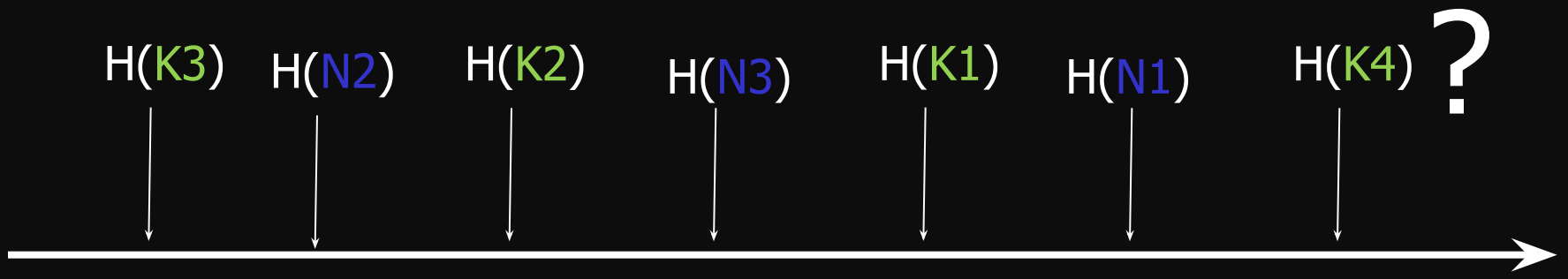
- Keys: K1, K2, K3, K4
- Nodes: N1, N2, N3



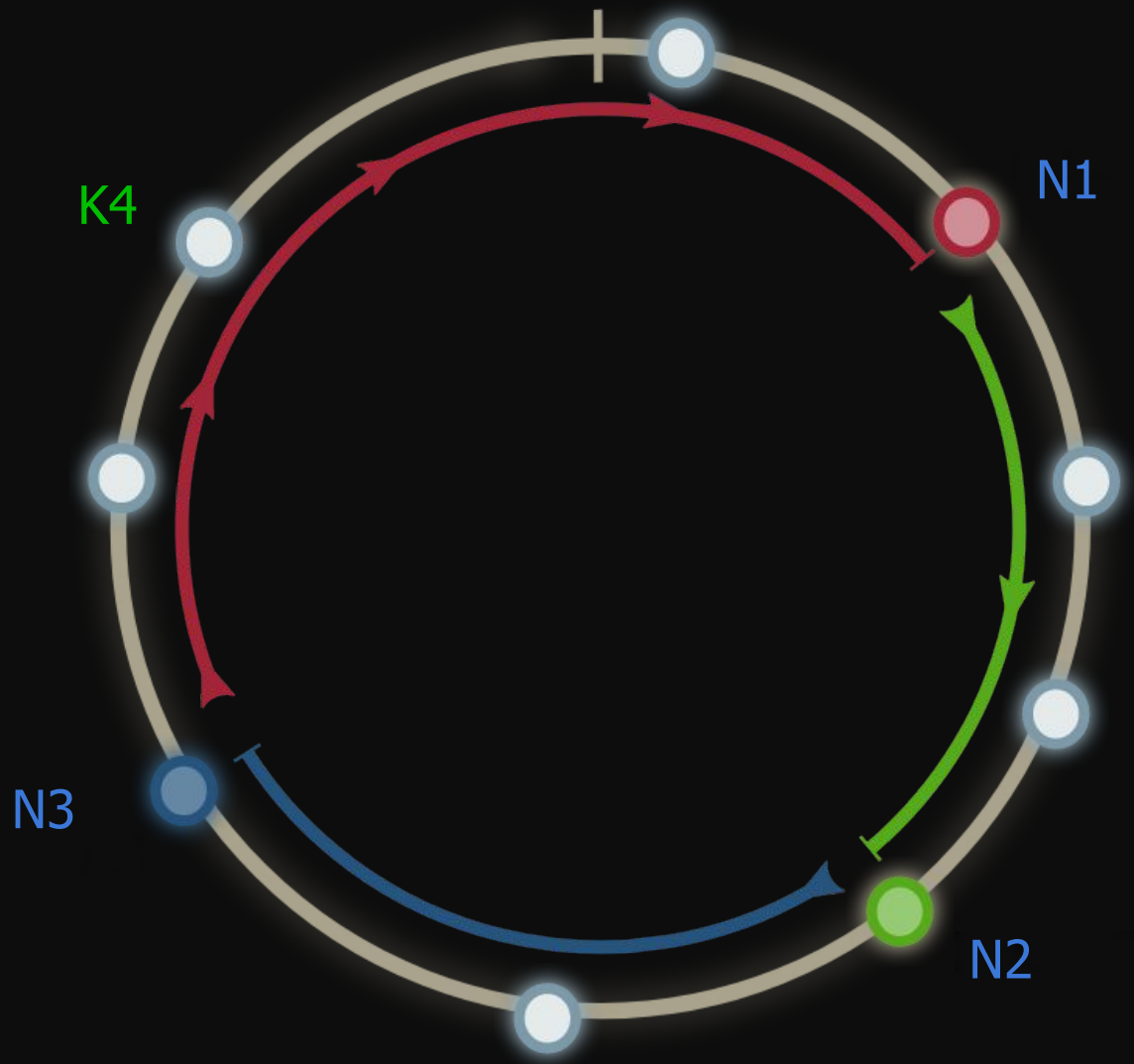
Consistent Hashing: Successors

- N3 responsible for range (H(N2), H(N3)]
- Stores the "values" - in this case an IP
  - (not the actual MP3)
- N3 is the "successor" of K2




H(K3)
H(N2)
H(K2)
H(N3)
H(K1)
H(N1)
H(K4)
?

Consistent Hashing (Ring)

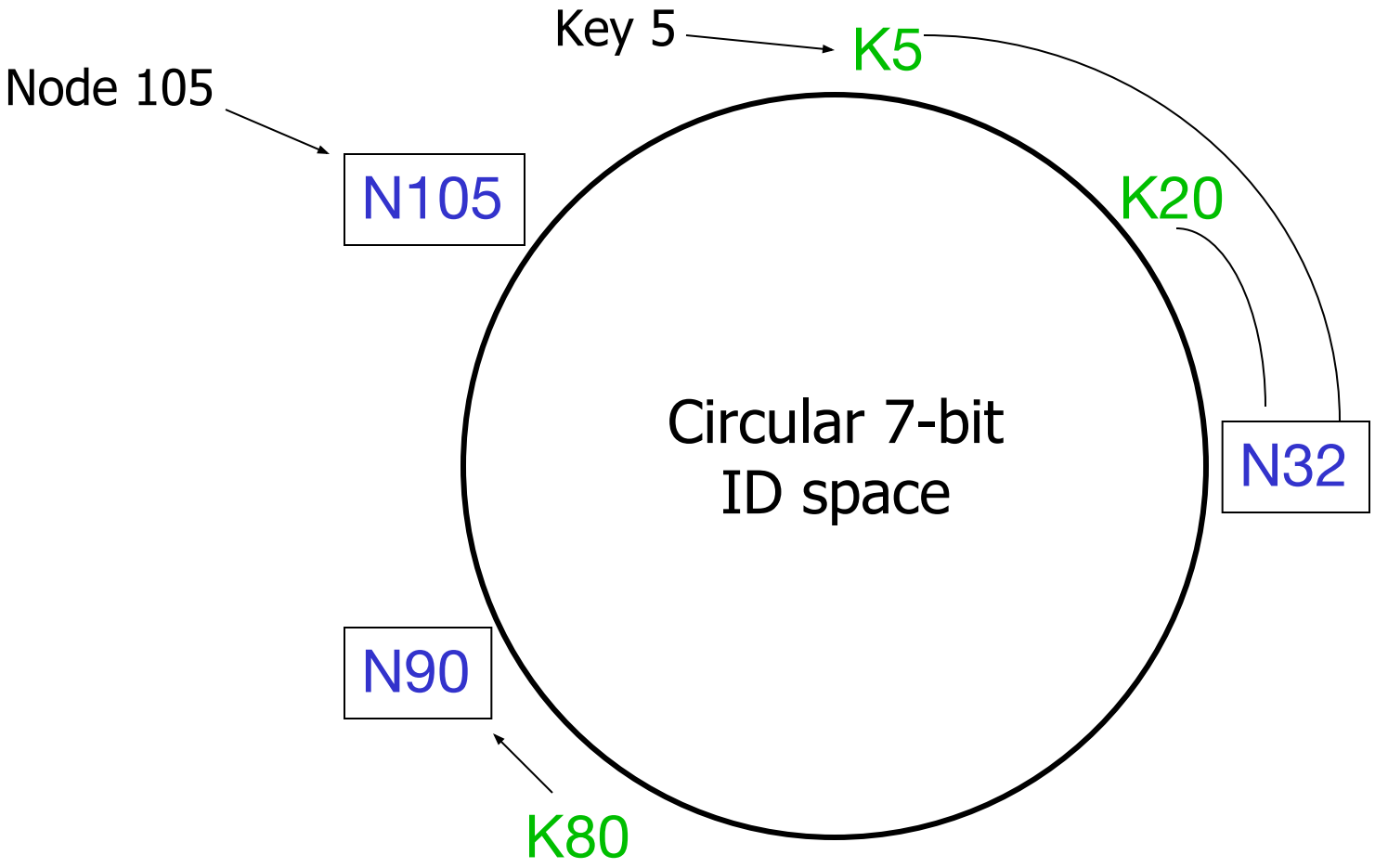


←Simpler notation

If H("baby...mp3") = K4, then you must ask N1 for the IP where the mp3 is stored

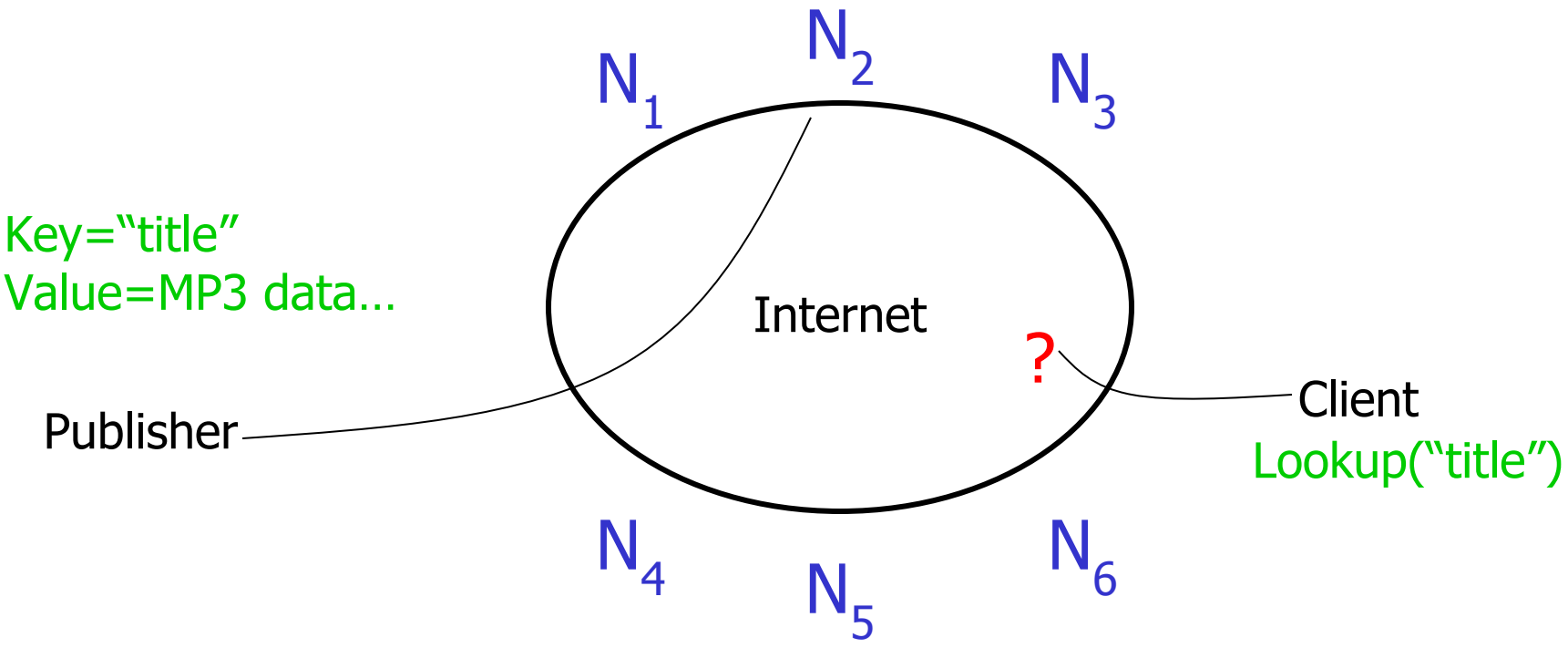
(means you need to know of N1)

[Spoiler]

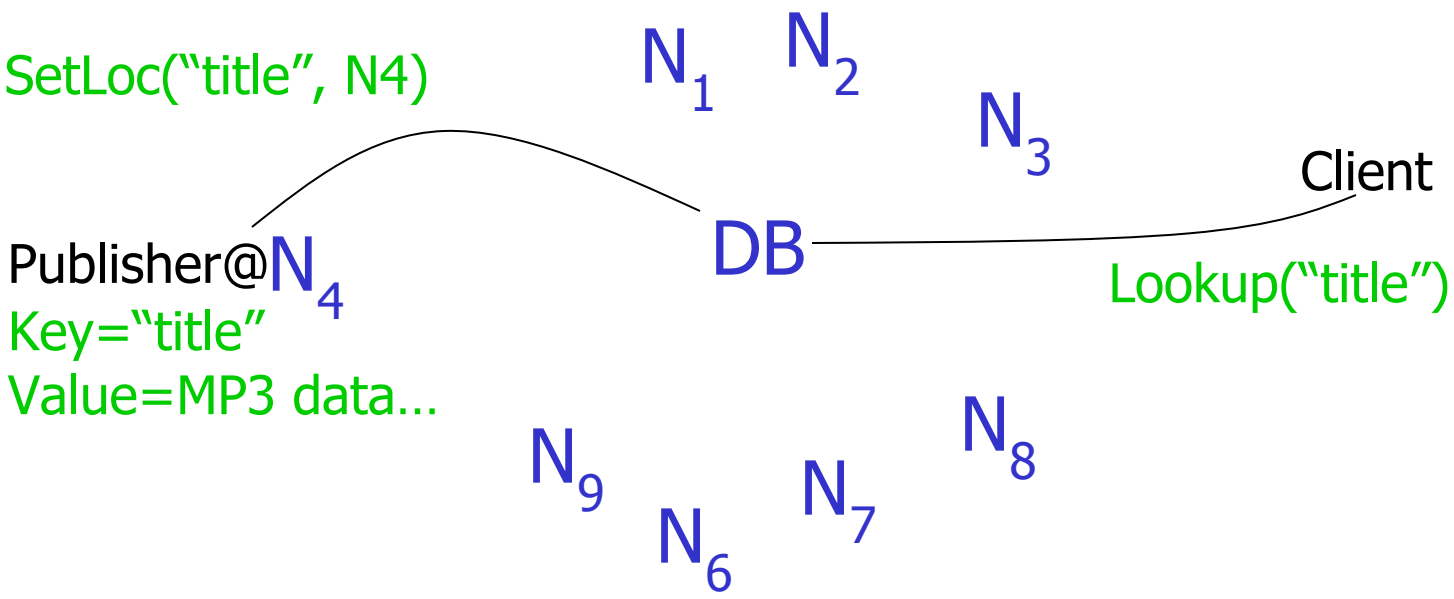


A key is stored at its successor: node with next higher ID

The lookup problem

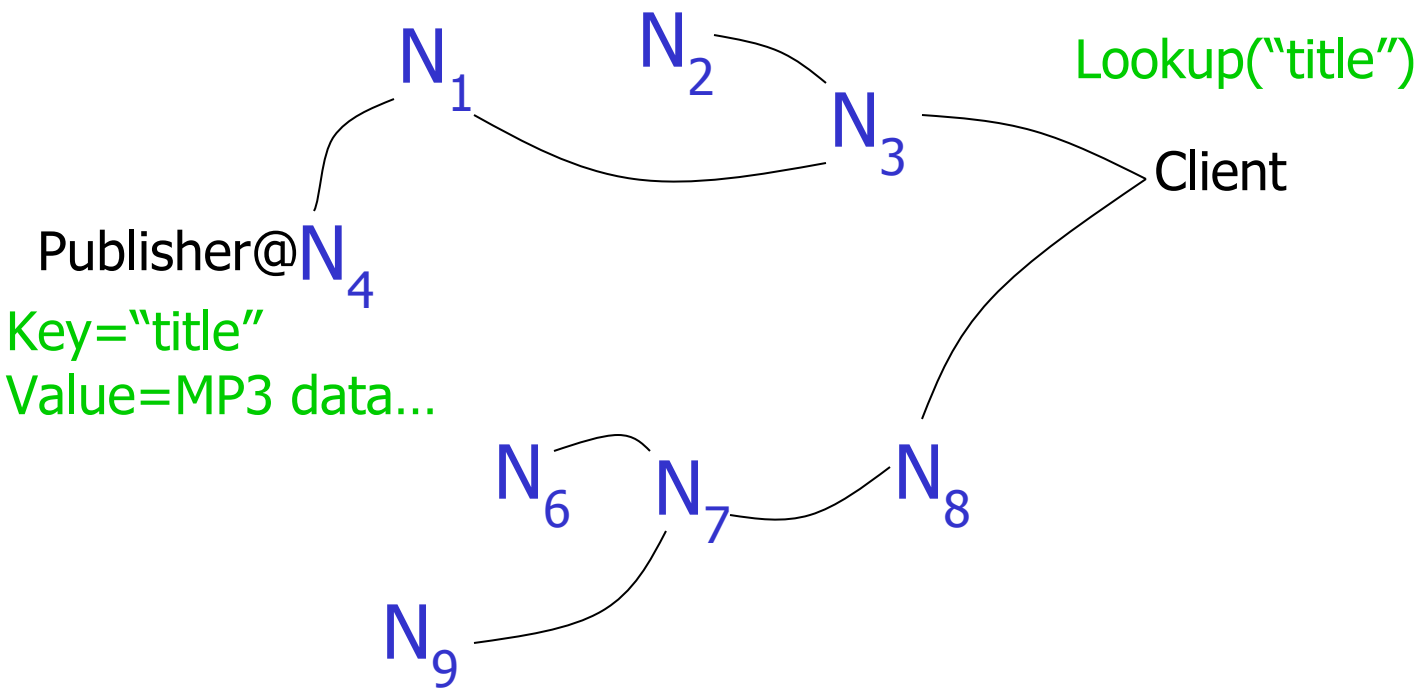


Centralized lookup (Napster)



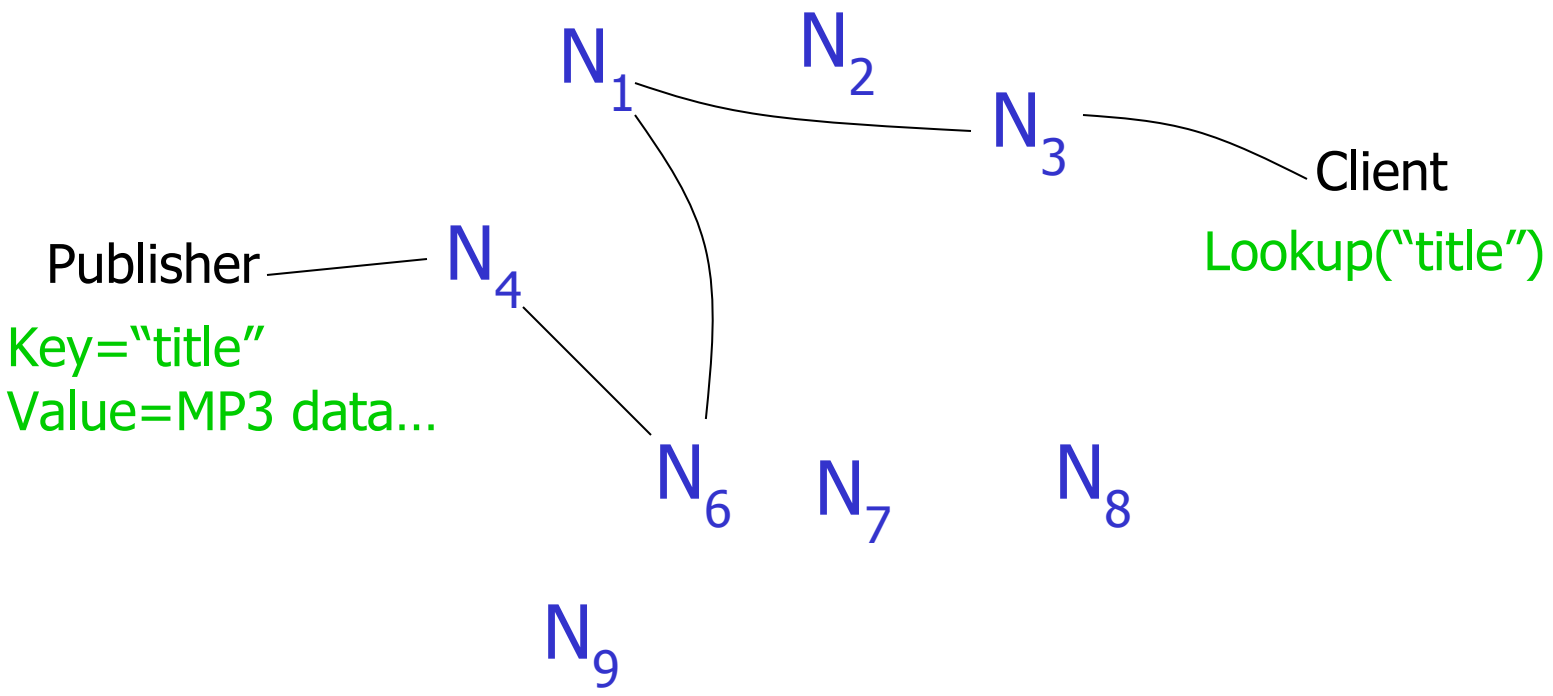
Simple, but O(N) state and a single point of failure

Flooded queries (Gnutella)



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

Routed queries (Freenet, Chord, etc.)



Routing challenges

- Define a useful key nearness metric
- Keep the hop count small
- Keep the tables small
- Stay robust despite rapid change

- Freenet: emphasizes anonymity
- Chord: emphasizes efficiency and simplicity

Chord properties

- Efficient: O($log(N)$) messages per lookup
  - N: # of servers
- Scalable: O($log(N)$) state per node
- Robust: survives massive failures

- Proofs are in paper / tech report
  - Assuming no malicious participants

Datacenter friendly

- Generic + Simple
- “natural” indexing
  - metadata location != data location
- Master-Less
- Linearly scalable
- Elastic partitioning
  - + Replication easily added

Chord overview

- P2P hash lookup:
  - Lookup(key) → IP address
  - Chord **does not store the data**
- How does Chord route lookups?
- How does Chord maintain routing tables?

Chord API

lookup(key)

Chord API

key = H("baby_one_more_time-Britney_Spears_1999_128kbps.mp3")

publisher = lookup(key)

go fetch your MP3 from "publisher" (if any)

Goal & Non-Goals

Chord is a “distributed lookup protocol”.

“given a key, it maps the key onto a node”.

Does NOT take care of the following:

- Actual data storage (eg MP3s)
- Availability
- Replication

Basically it only does 1 thing:

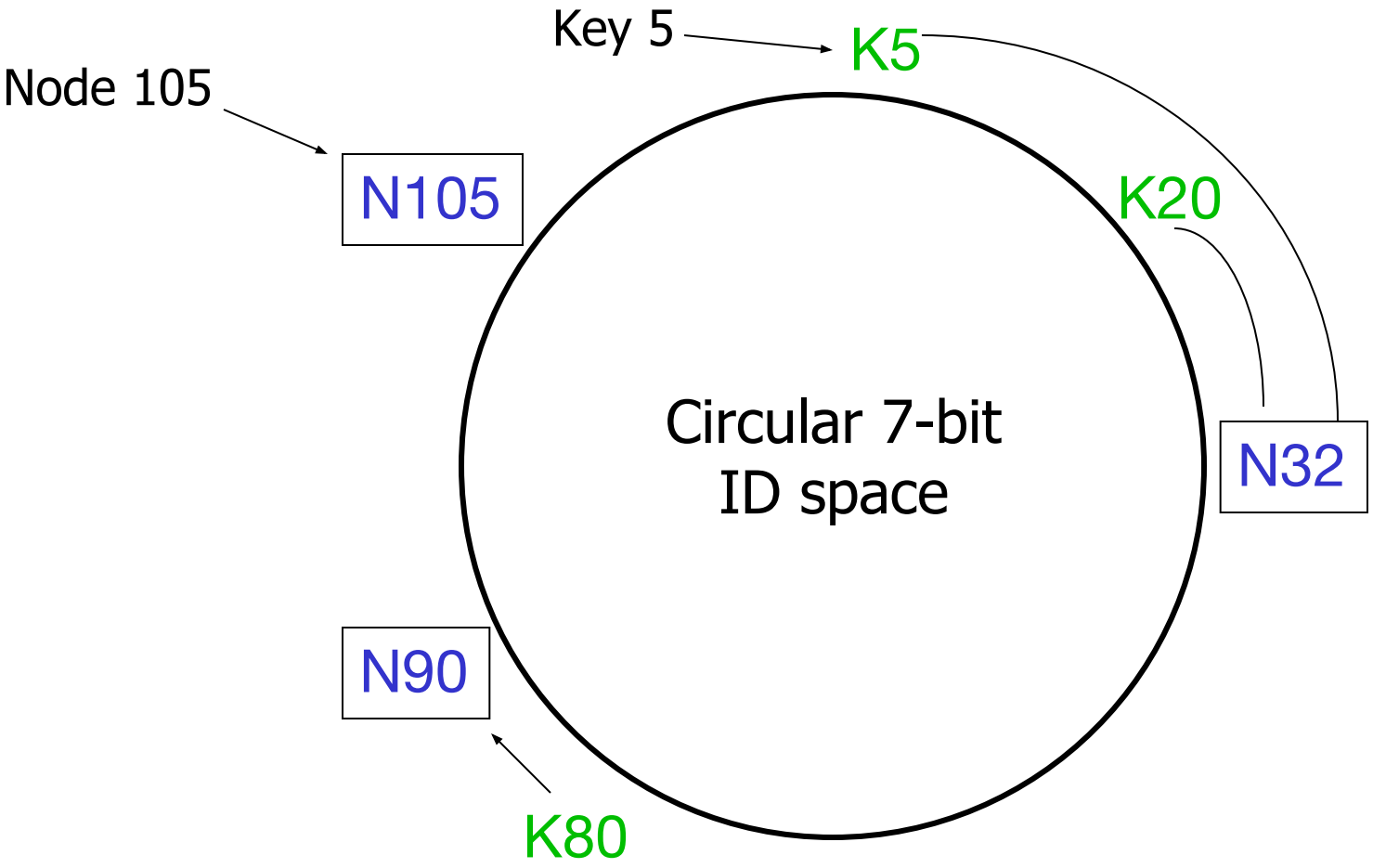
Routing of the lookup process

Chord IDs

- Key identifier = SHA-1(key)
- Node identifier = SHA-1(IP address)
- Both are uniformly distributed
- Both exist in the same ID space

- How to map key IDs to node IDs?

Consistent hashing [Karger 97]



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

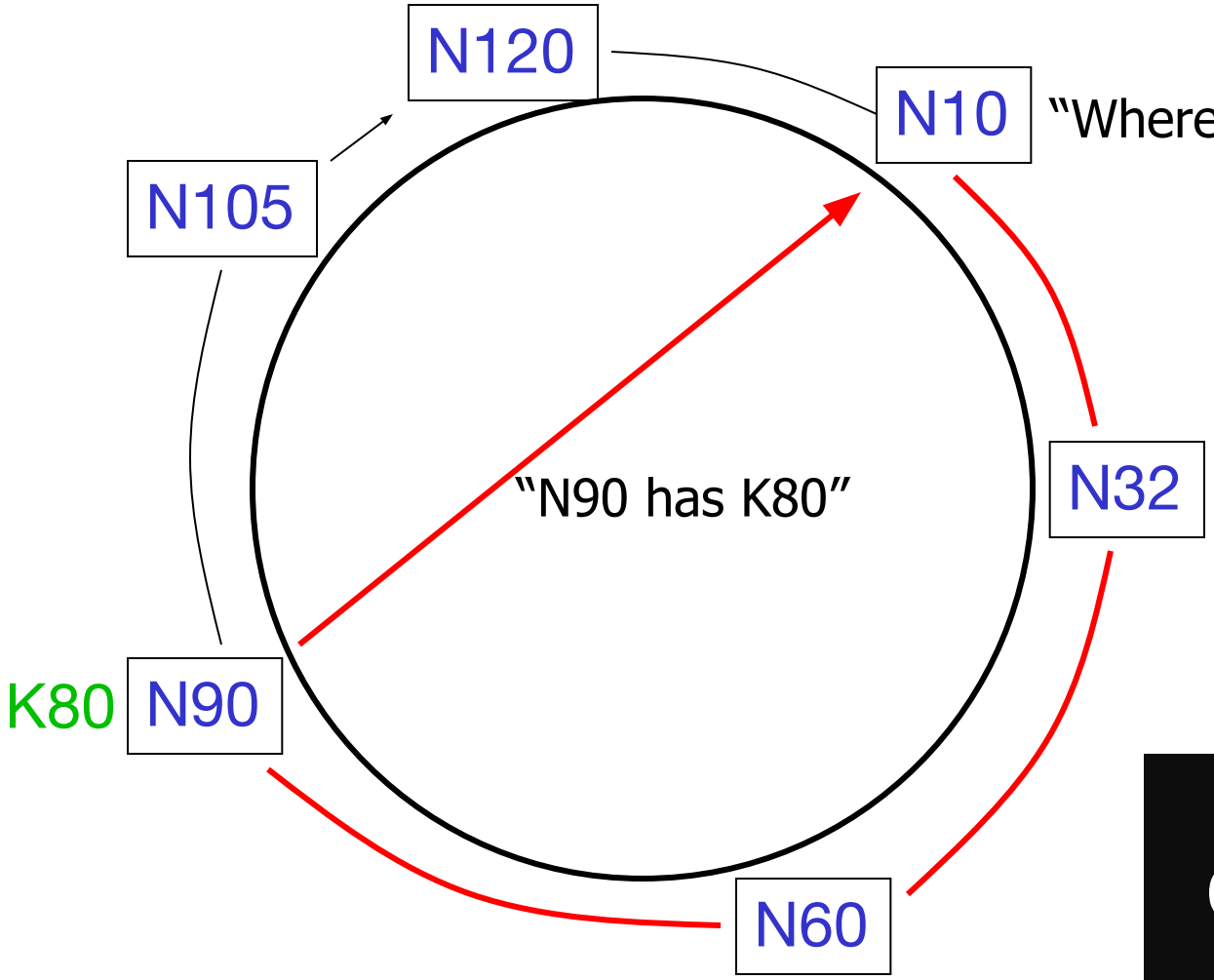
Naive Consistent Hashing

- Every node knows all nodes
  - does not scale well

In (basic) Chord,
a node only knows about its successor

That's the only thing needed to guarantee valid lookups

Basic lookup



O(N)

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

Fingering other nodes

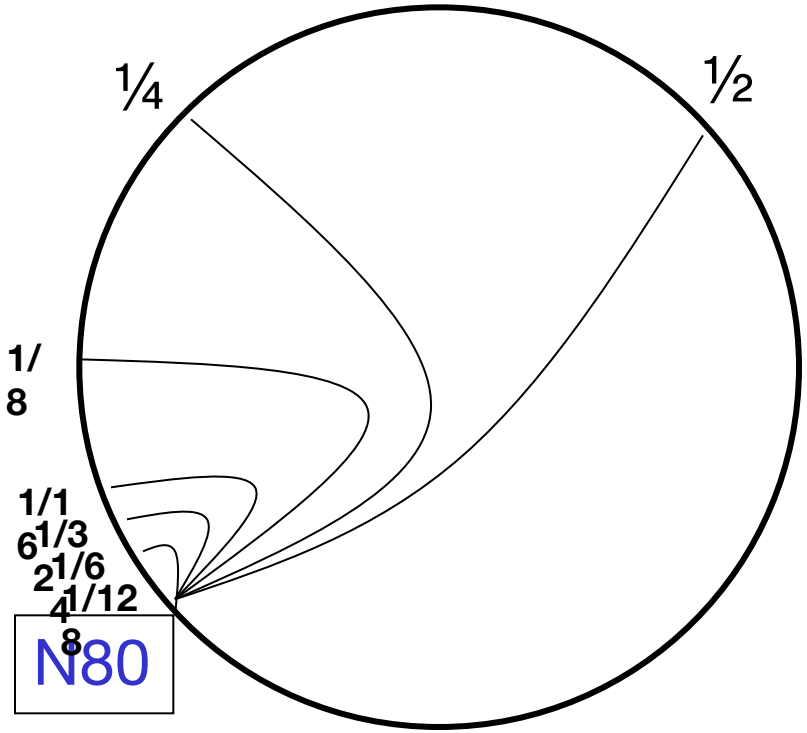
For hash range of 2^h :
If your node's id is n

keep "fingers" for the successor nodes of keys:
~~$n+1$~~, $n+2^2$, $n+2^3$ … $n + 2^{h-1}$

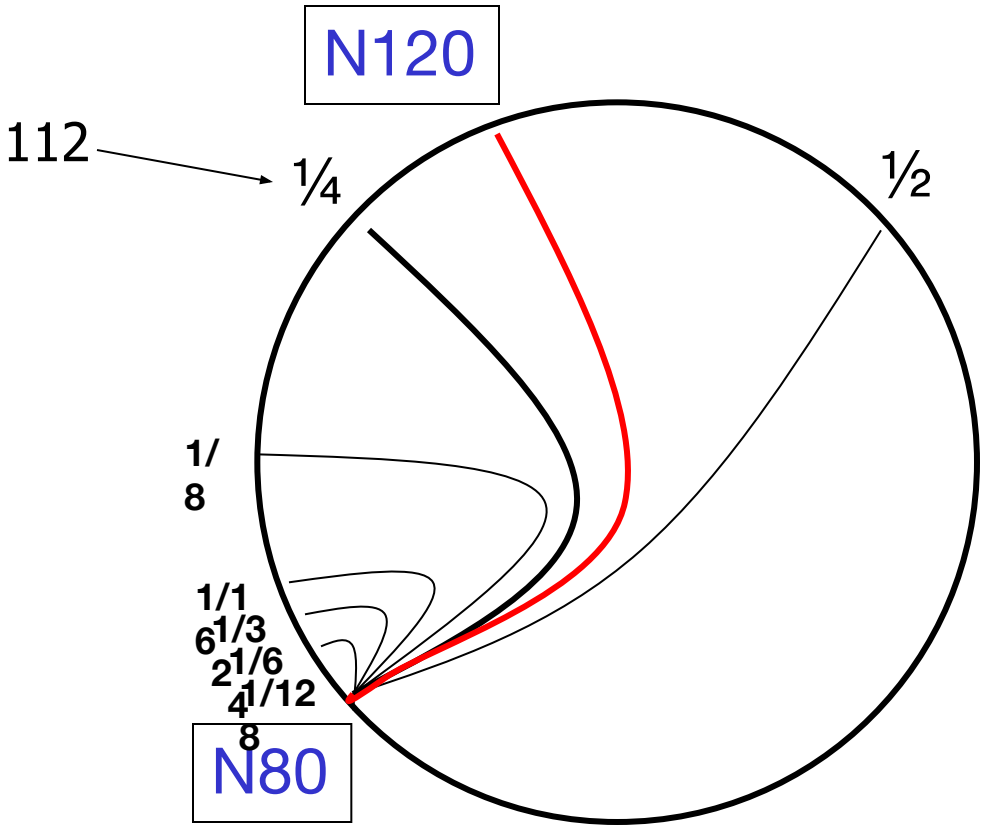
for speeding up the lookup process

(computed lazily, not required for correctness

“Finger table” allows log(N)-time lookups



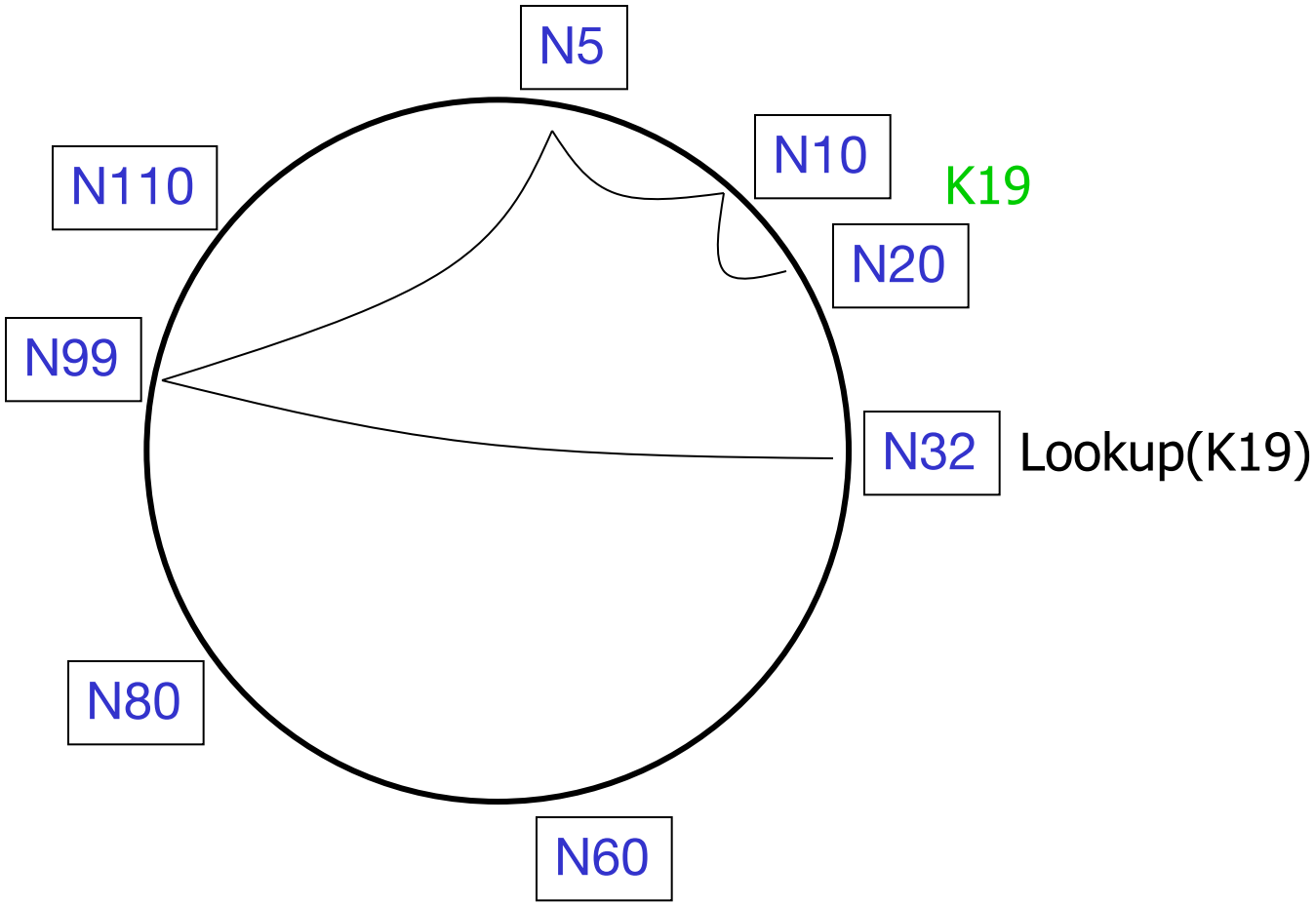
Finger i points to successor of $n+2^i$



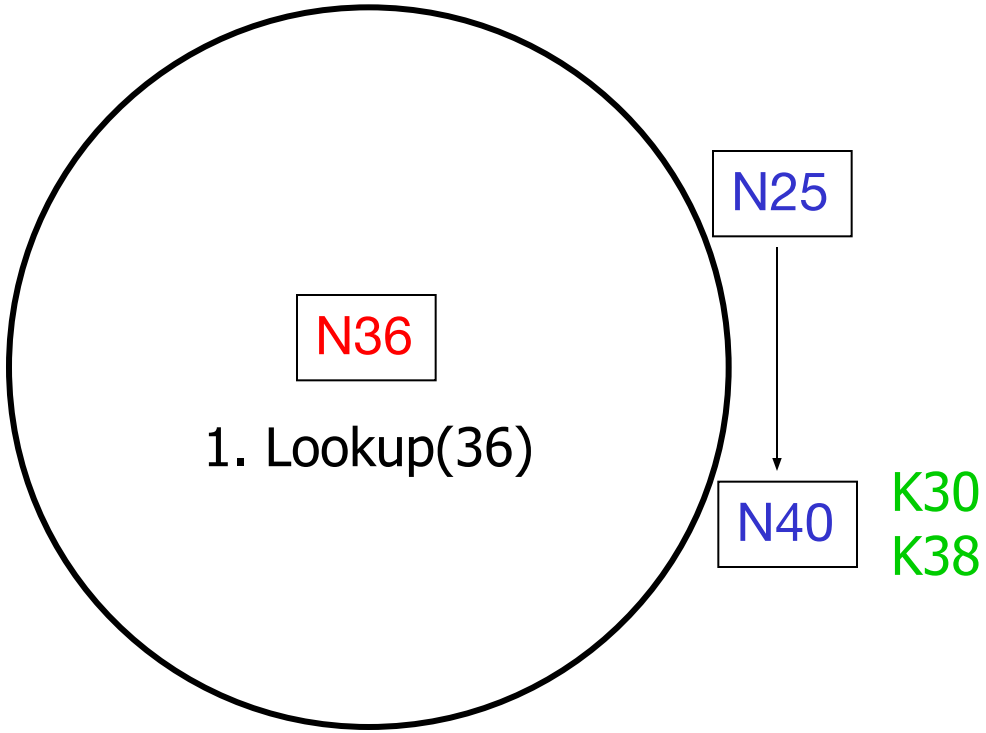
Lookup with fingers

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

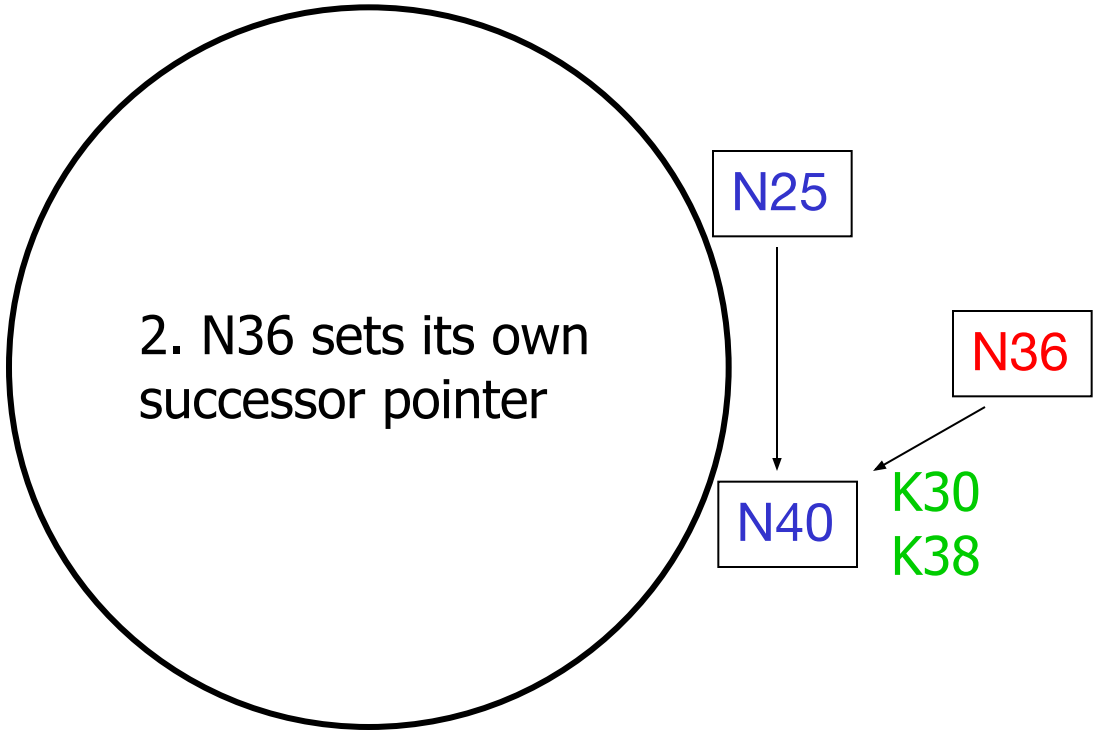
Lookups take O($log(N)$) hops



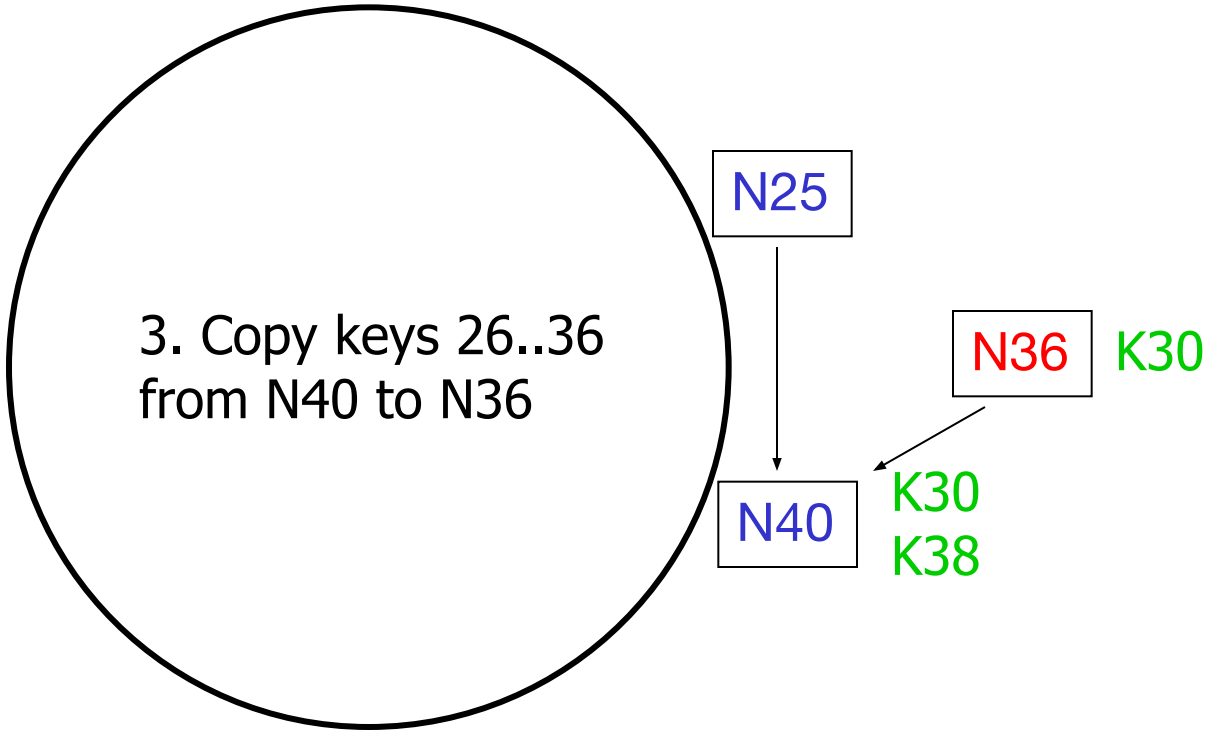
Joining: linked list insert



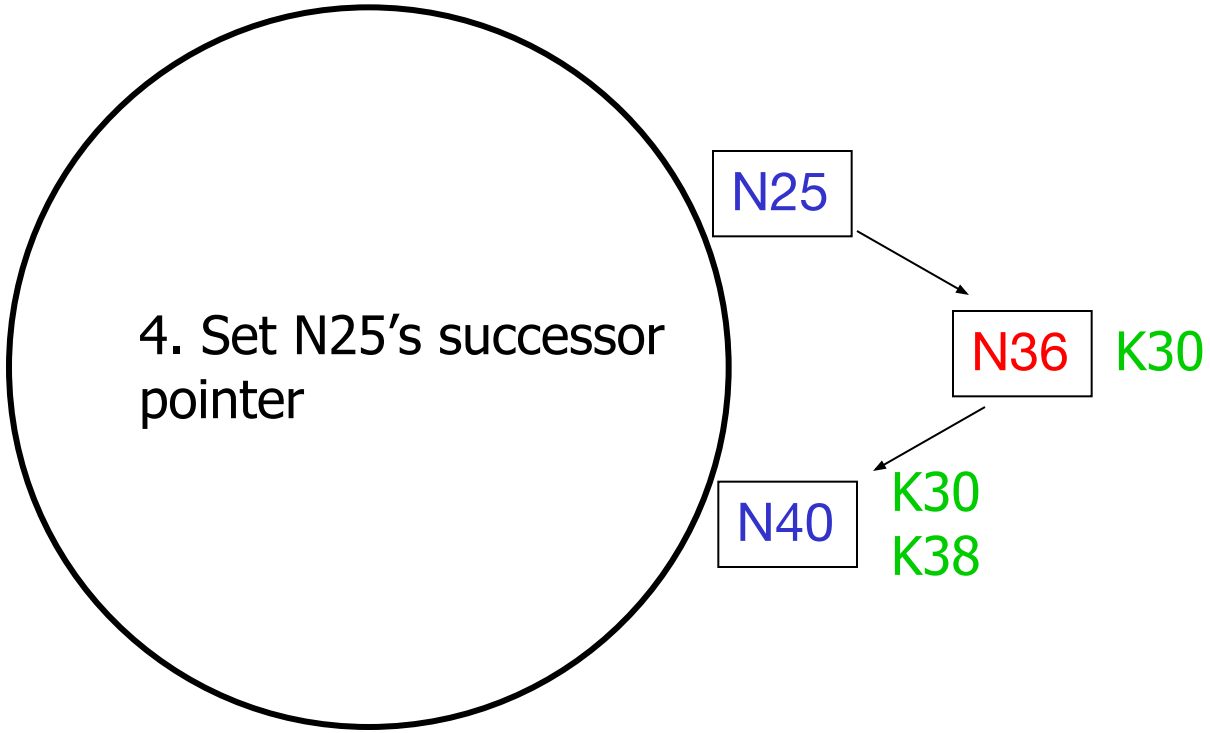
Join (2)



Join (3)



Join (4)



Update finger pointers in the background
Correct successors produce correct lookups

Stabilization

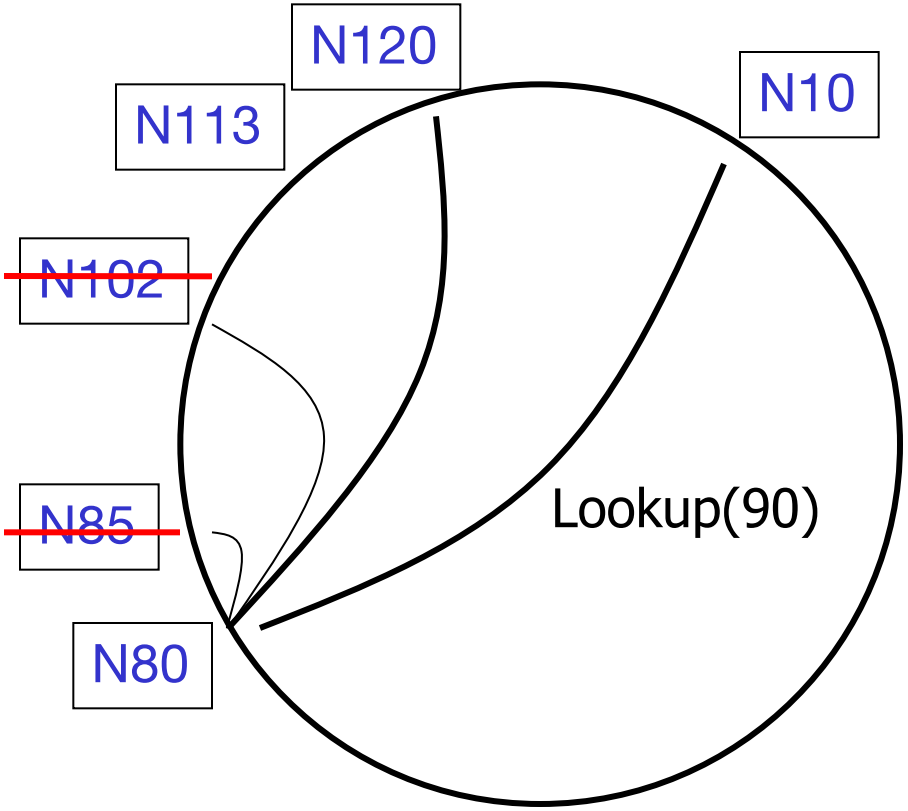
Every node runs `*stabilize*` periodically

- this is how newly joined nodes are noticed by the network

When node n runs stabilize:

- asks my successor, p, for its predecessor: o
- if o != n
  - set n.successor = o

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 the 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

Bonus Performance

```
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
```

Chord status

- Working implementation as part of CFS
- Chord library: 3,000 lines of C++
- Deployed in small Internet testbed
- Includes:
  - Correct concurrent join/fail
  - Proximity-based routing for low delay
  - Load control for heterogeneous nodes
  - Resistance to spoofed node IDs

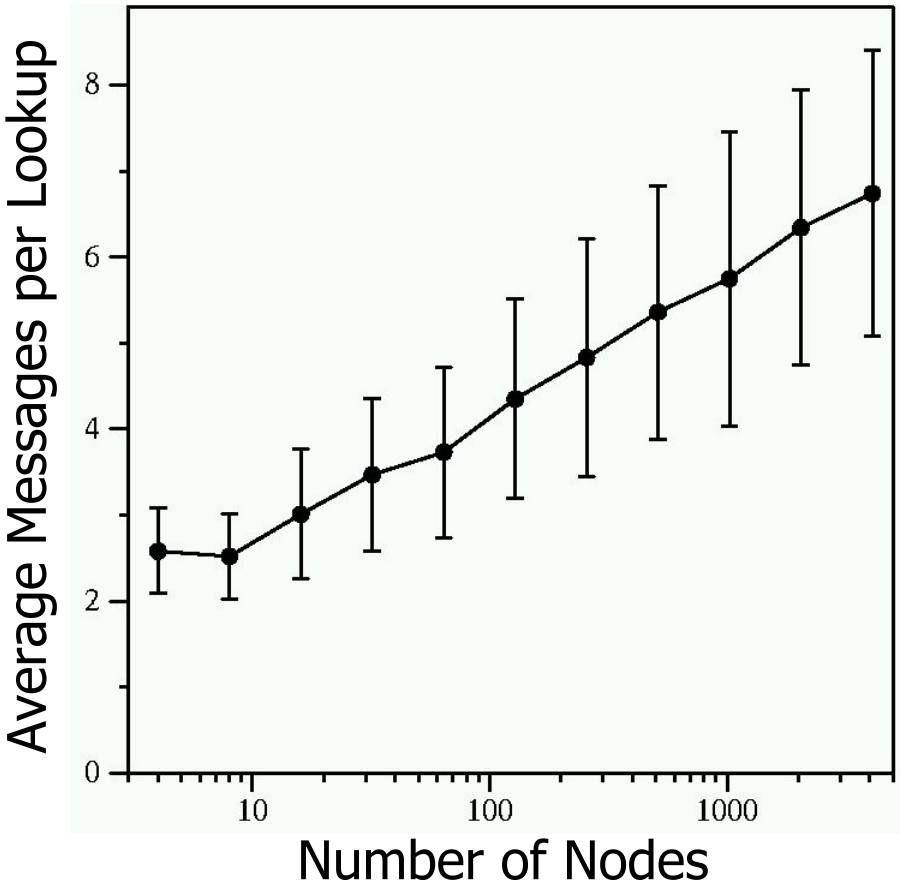
Experimental overview

(simulation)

- Quick lookup in large systems
- Low variation in lookup costs
- Robust despite massive failure
- See paper for more results

Experiments confirm theoretical results

Chord lookup cost is O(log N)

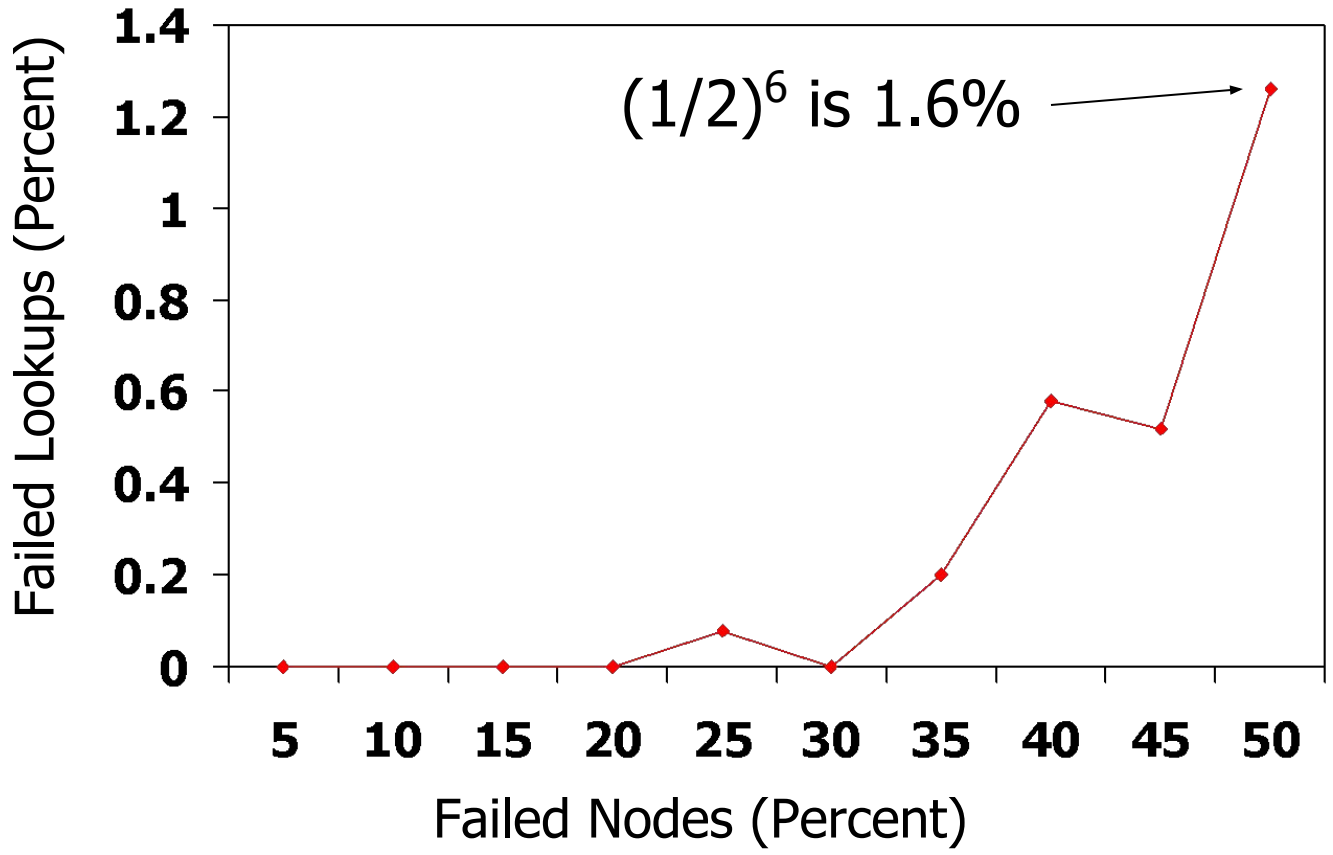


Constant is 1/2

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

Massive failures have little impact



10^4 Nodes, 10^6 Keys

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