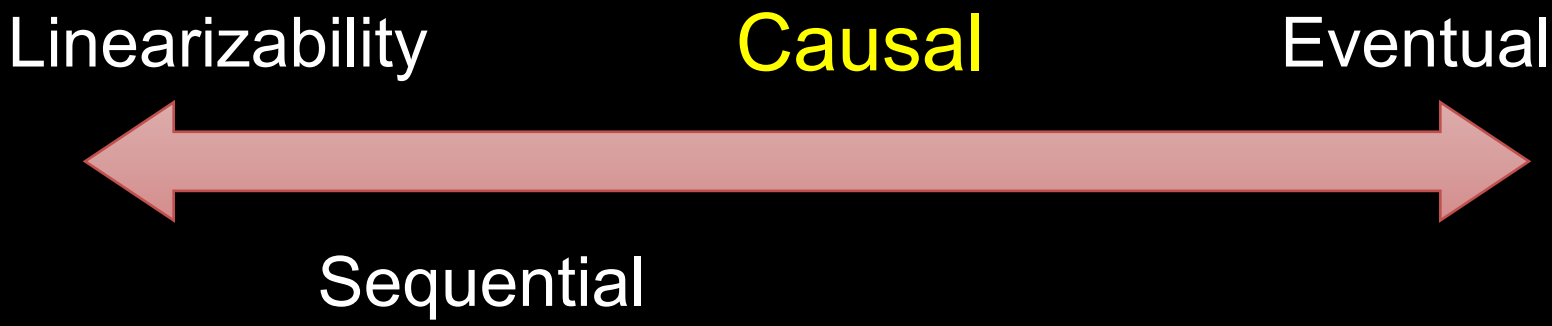
Distributed Hash Tables

CS6450: Distributed Systems

Lecture 12

Ryan Stutsman

Consistency models



Recall use of logical clocks

- Lamport clocks: C(a) < C(z) Conclusion: None
- Vector clocks: V(a) < V(z) Conclusion: a → … → z

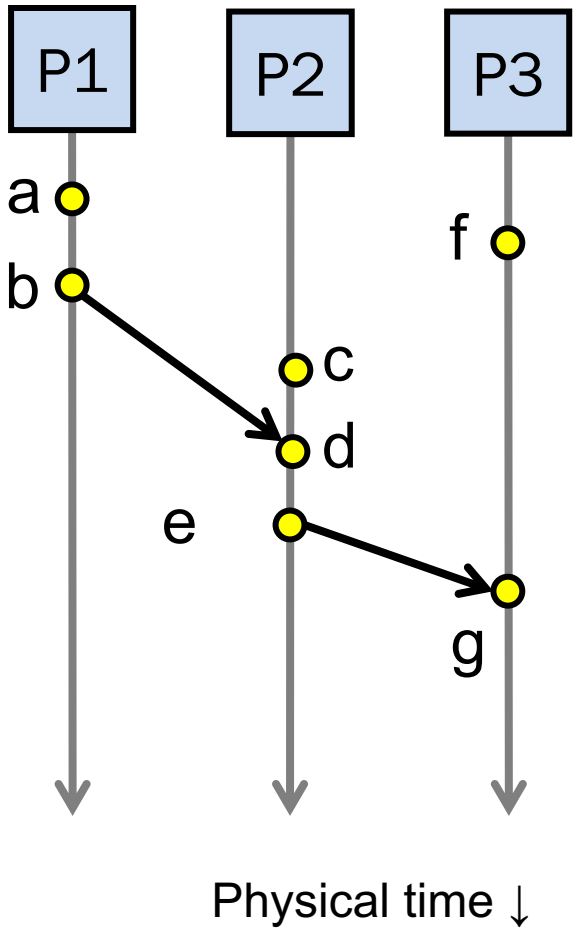
- Distributed bulletin board application
  - Each post gets sent to all other users
  - Consistency goal: No user to see reply before the corresponding original message post
  - Conclusion: Deliver message only **after** all messages that **causally precede** it have been delivered

Causal Consistency

1. Writes that are ***potentially*** causally related must be seen by all machines in same order.

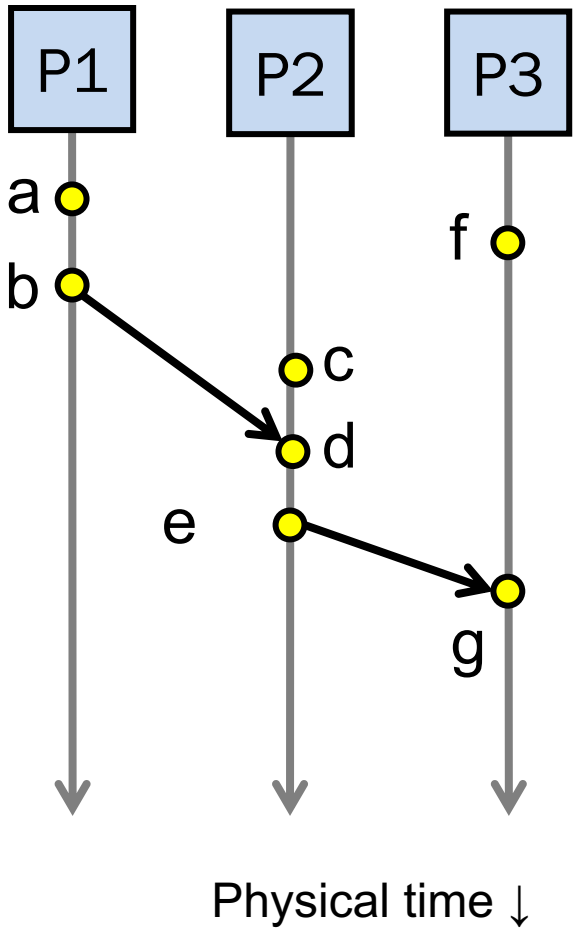
2. Concurrent writes may be seen in a different order on different machines.

- Concurrent: Ops not causally related



Causal Consistency

Operations	Concurrent?
a, b	N
b, f	Y
c, f	Y
e, f	Y
e, g	N
a, c	Y
a, e	N



Causal Consistency: Quiz

P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

Causal Consistency: Quiz

P1:	W(x)a		W(x)c		
P2:		R(x)a	W(x)b		
P3:		R(x)a		R(x)c	R(x)b
P4:		R(x)a		R(x)b	R(x)c

- Valid under causal consistency
- **Why?** *W(x)b* and *W(x)c* are concurrent
  - So all processes don't (need to) see them in same order
- P3 and P4 read the values 'a' and 'b' in order as potentially causally related. No 'causality' for 'c'.

Sequential Consistency: Quiz

P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

Sequential Consistency: Quiz

P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

- Invalid under sequential consistency
- **Why?** P3 and P4 see b and c in different order
- But fine for causal consistency
  - B and C are not causually dependent
  - Write after write has no dep's, write after read does

Causal Consistency

P1: W(x)a				
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(a)

P1: W(x)a				
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(b)

Causal Consistency

P1: W(x)a				
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(a)

P1: W(x)a				
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(b)

A: Violation: W(x)b is potentially dep on W(x)a

B: Correct. P2 doesn't read value of a before W

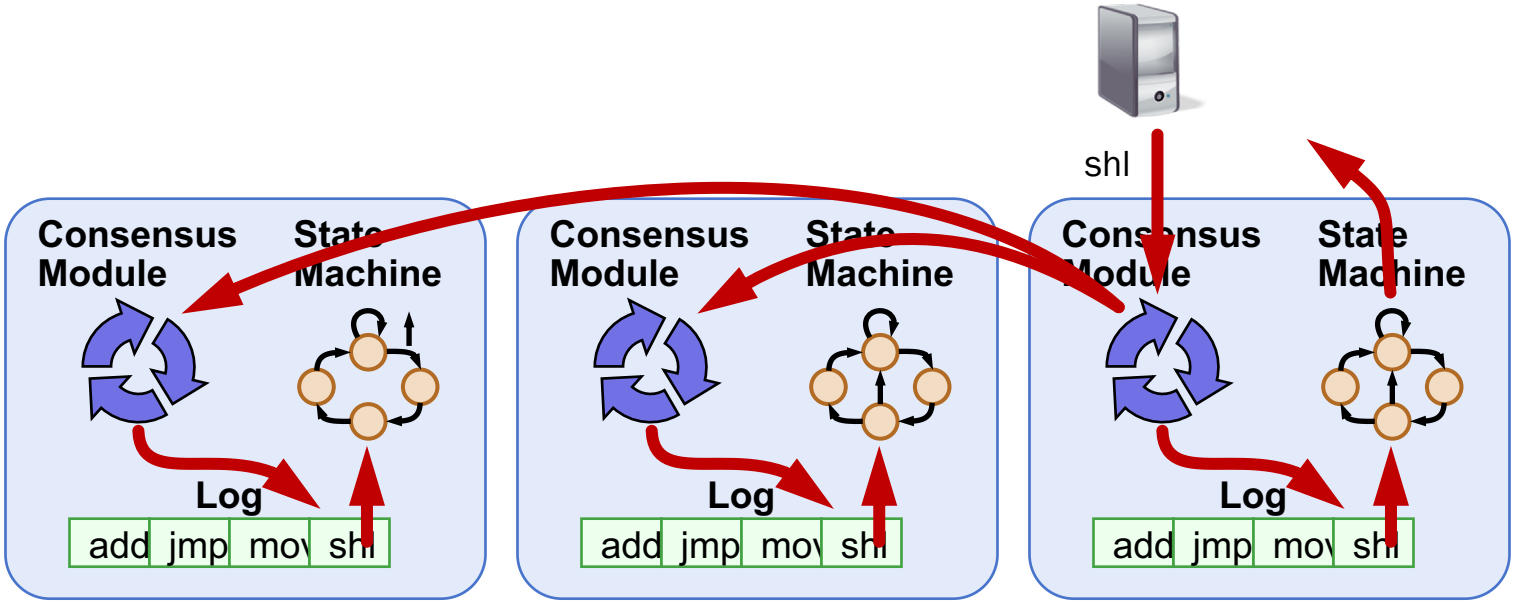
Why is SC Unavailable?

P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

- Invalid under sequential consistency
- What if we just totally order (using e.g. Lamport clocks) and force P3 and P4 to see W(x)b and W(x)c in the same order?
- Consider P4 issuing R(x)b; it must be sure that no other op with a lower timestamp *has or could* affect x, which forces it to communicate with everyone

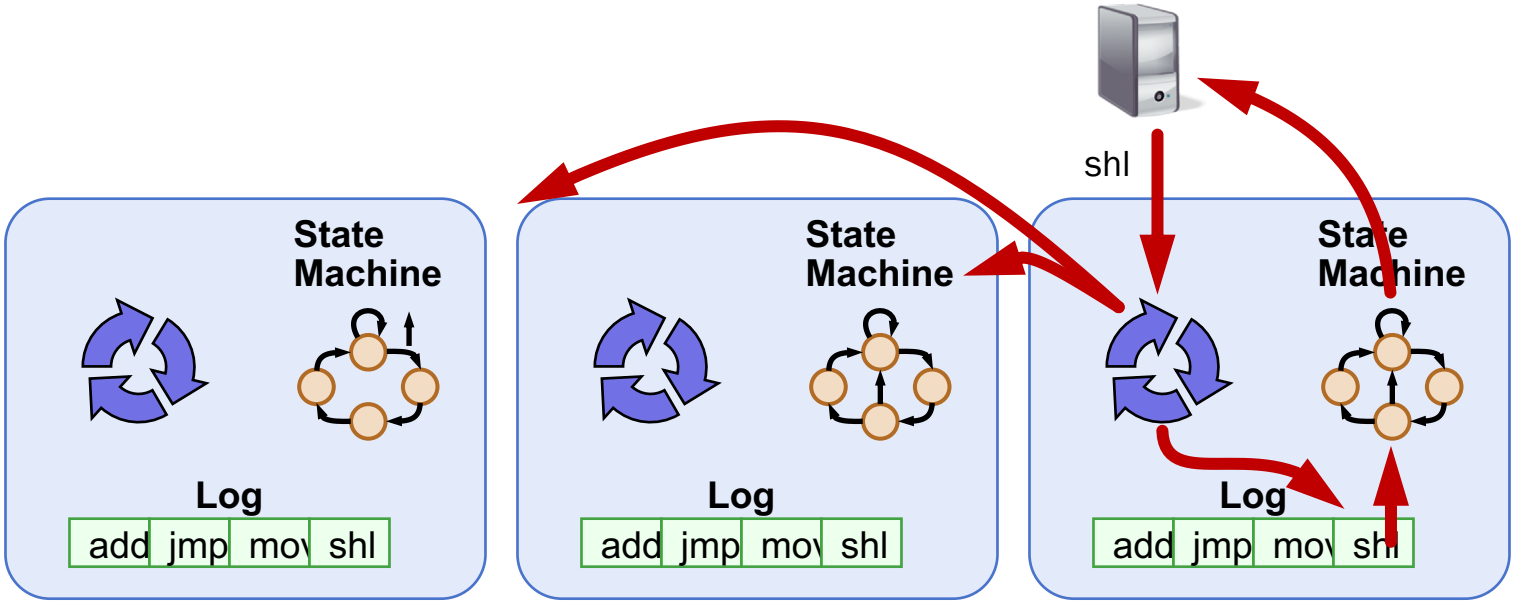
Causal consistency within replication systems

Implications of laziness on consistency



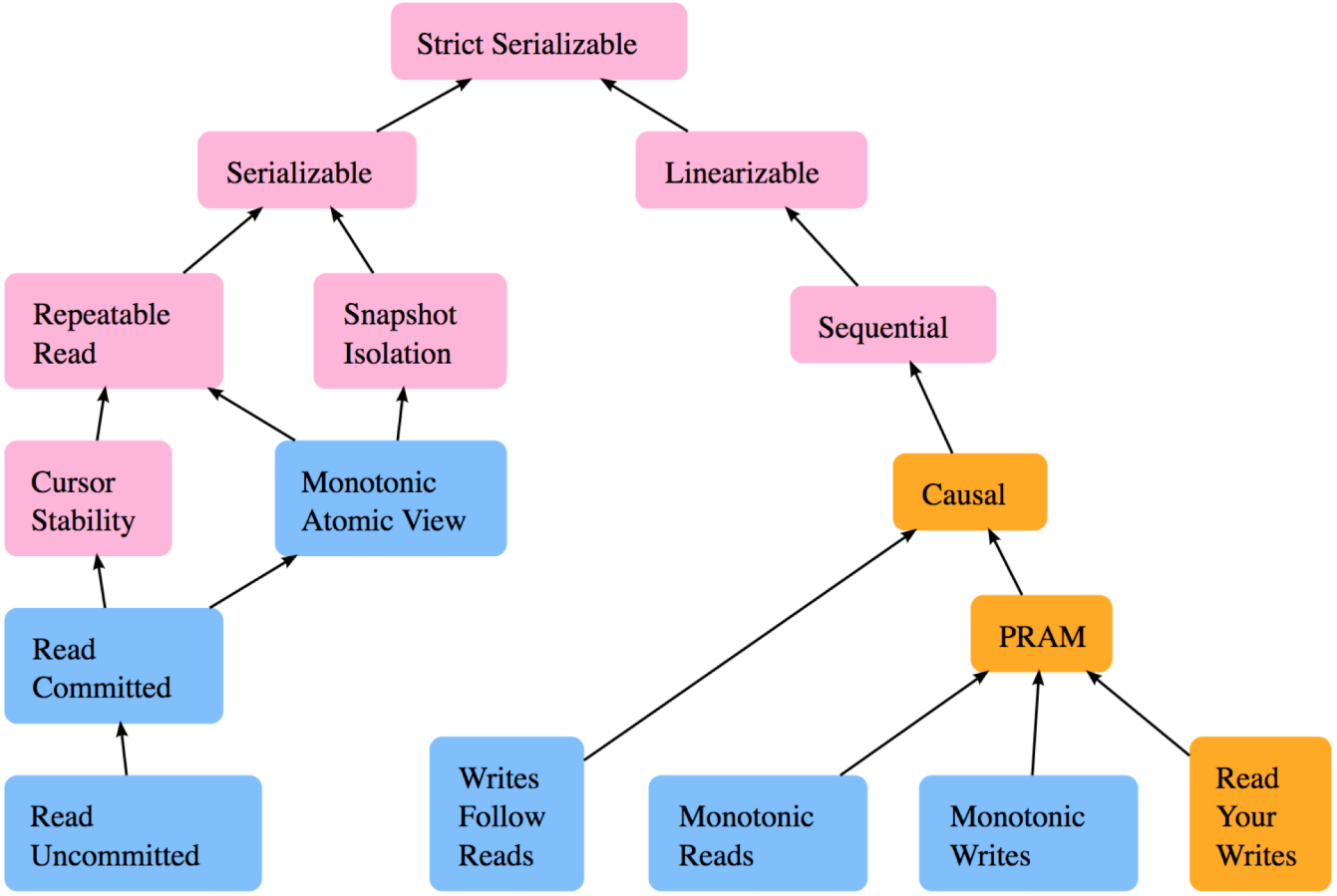
- Linearizability / sequential: Eager replication
- Trades off low-latency for consistency

Implications of laziness on consistency



- Causal consistency: Lazy replication
- Trades off consistency for low-latency
- Maintain local ordering when replicating
- Operations may be lost if failure before replication

Consistency Models



jepsen.io/consistency

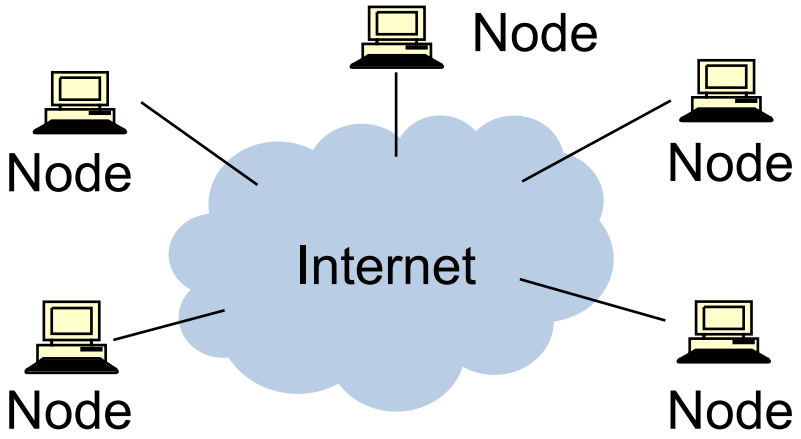
Today

1. Peer-to-Peer Systems
   - Napster, Gnutella, BitTorrent, challenges
2. Distributed Hash Tables
3. The Chord Lookup Service
4. Concluding thoughts on DHTs, P2P

Takeaways

- P2P systems aggregate lots of resources by avoiding centralization
  - Scale well, avoid single point of failure (both technically and legally)
- Naming/routing is challenging
  - Centralized: state doesn't scale
  - Unstructured: messaging costs don't scale
- Hybrid approach: DHTs
  - Usually consistent hashing with routing on opaque content-based identifiers
  - Individual nodes don't need total state, but avoids need to flood for lookups
  - Tolerates high churn (node add/remove impacts a handful of nodes)

What is a Peer-to-Peer (P2P) system?



- A **distributed** system architecture:
  - **No centralized control**
  - Nodes are **roughly symmetric** in function

- **Large** number of **unreliable** nodes

Why might P2P be a win?

- **High capacity for services** through parallelism:
  - Many disks
  - Many network connections
  - Many CPUs

- **Absence of a centralized server** or servers may mean:
  - Less chance of service overload as load increases
  - Easier deployment
  - A single failure won't wreck the whole system
  - System as a whole is harder to attack

P2P adoption

- Successful adoption in **some niche areas** –

1. Client-to-client (legal, illegal) **file sharing**
   - Popular data but owning organization has no money

2. **Digital currency:** no natural single owner (Bitcoin)

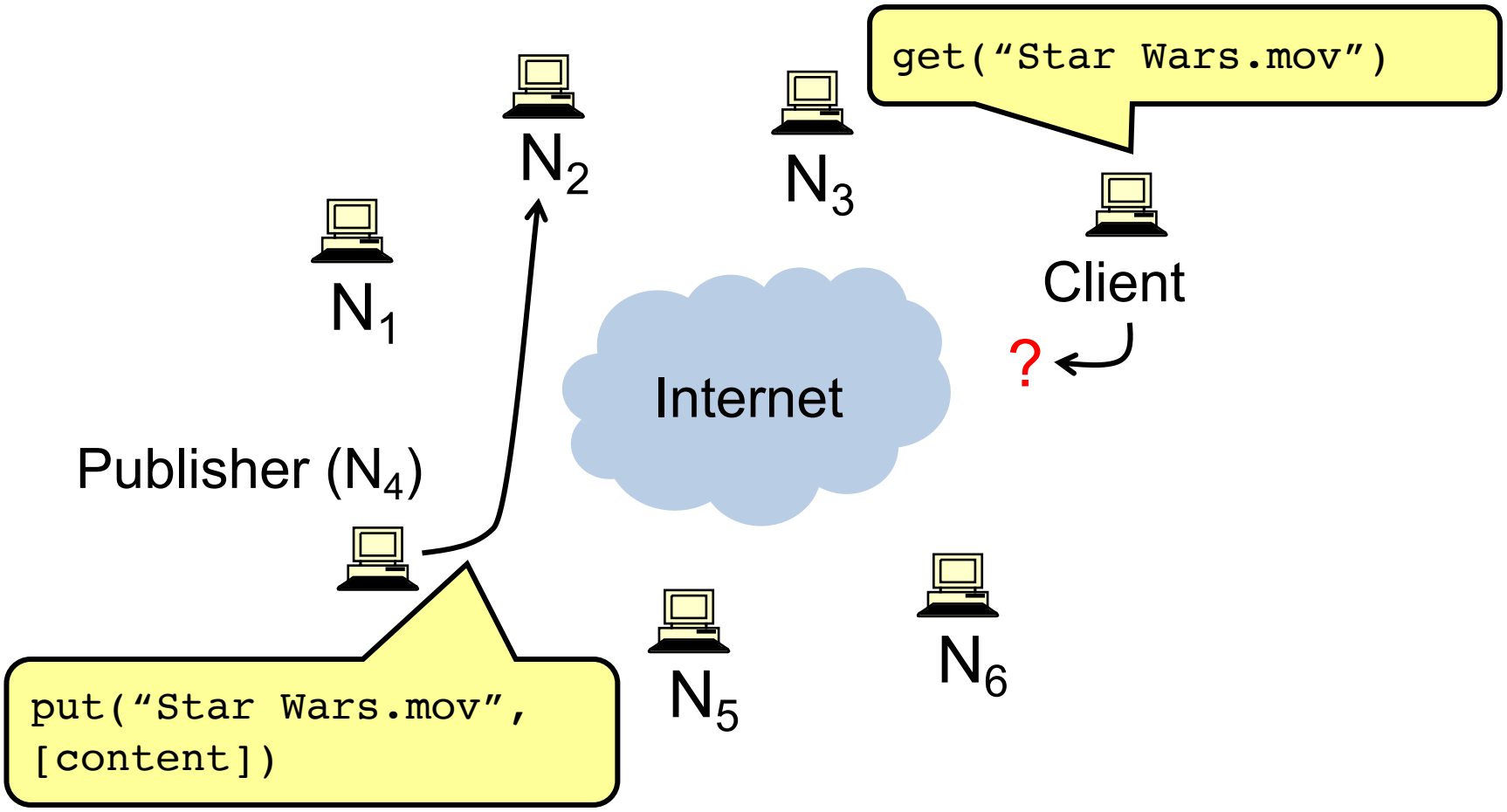
3. **Voice/video telephony:** user to user anyway
   - Issues: Privacy and control

Example: Classic BitTorrent

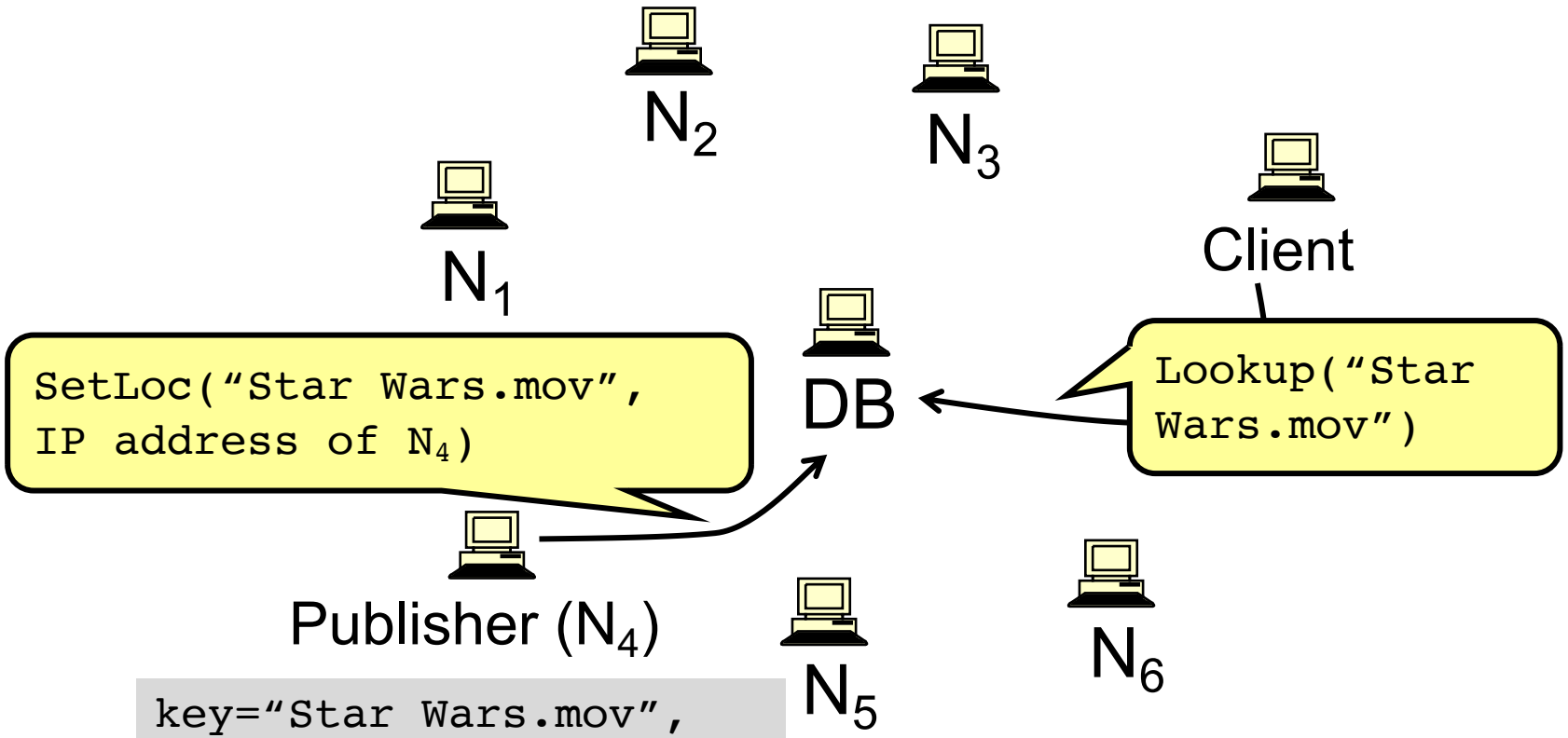
1. User clicks on download link
   - Gets *torrent* file with content hash, IP addr of *tracker*
2. User's BitTorrent (BT) client talks to tracker
   - Tracker tells it **list of peers** who have file
3. User's BT client downloads file from one or more peers
4. User's BT client tells tracker it has a copy now, too
5. User's BT client serves the file to others for a while

Provides huge download bandwidth, without expensive server or network links;
still tracker is point of failure/attack

The lookup problem

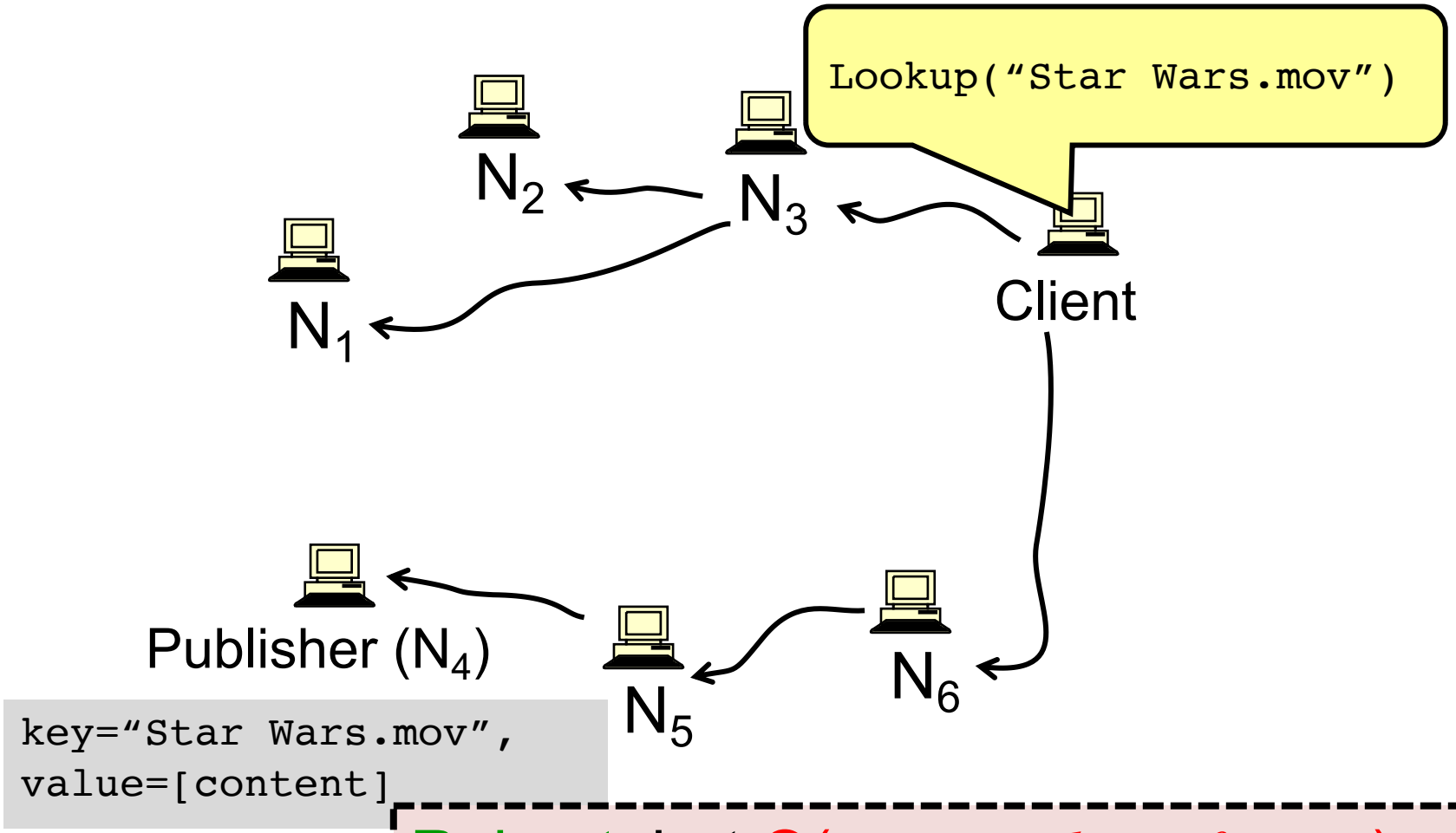


Centralized lookup (Napster)



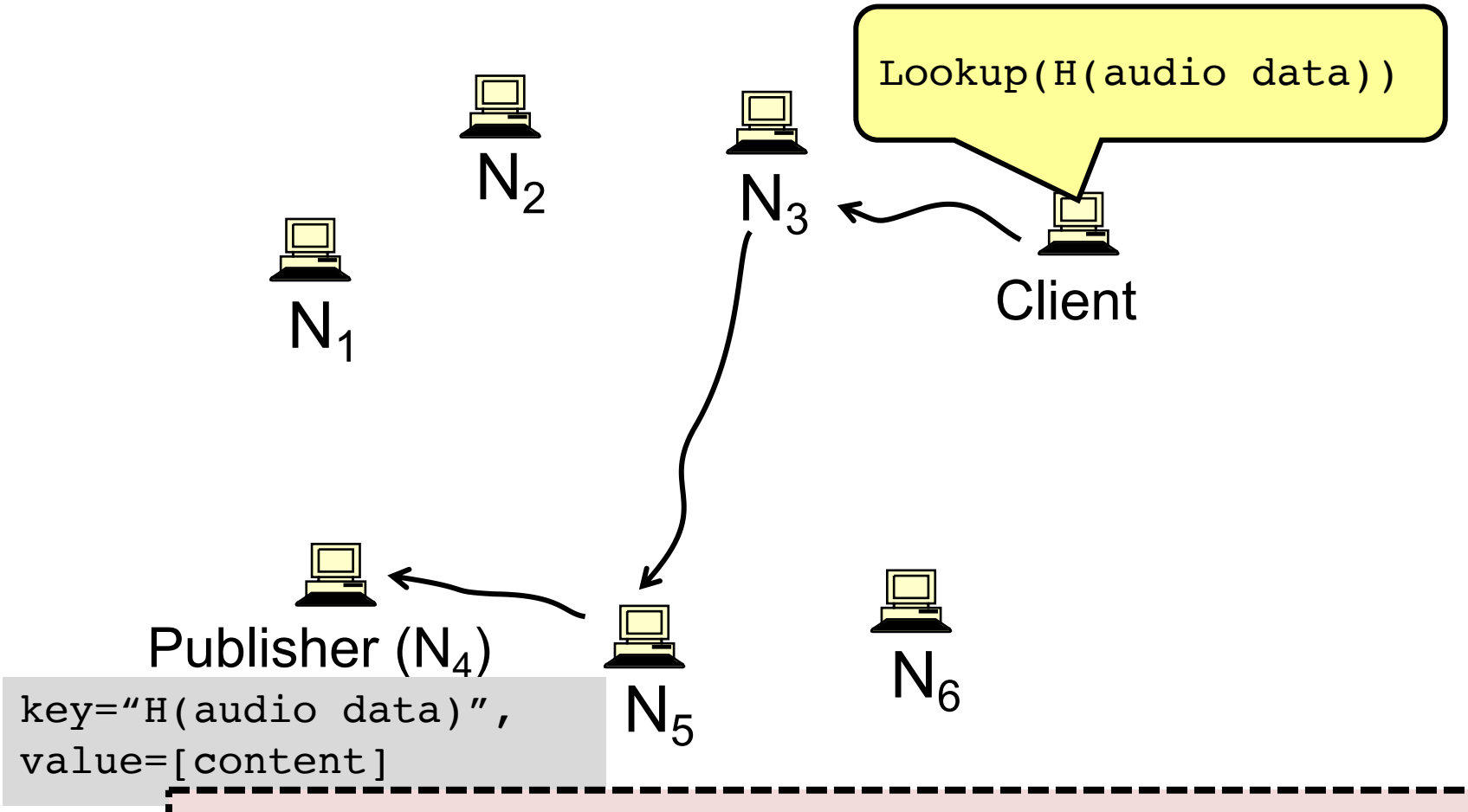
Simple, but O(N) state and a single point of failure

Flooded queries (original Gnutella)



Robust, but O(*N = number of peers*) messages per lookup

Routed DHT queries (Chord)



Can we make it robust, reasonable state, reasonable number of hops?

Today

1. Peer-to-Peer Systems
2. **Distributed Hash Tables**
3. The Chord Lookup Service
4. Concluding thoughts on DHTs, P2P

What is a DHT (and why)?

- Local hash table:

```
key = Hash(name)
put(key, value)
get(key) → value
```

- **Service:** Constant-time insertion and lookup

How can I do (roughly) this across millions of hosts on the Internet?

Distributed Hash Table (DHT)

What is a DHT (and why)?

- Distributed Hash Table:

  `key = hash(data)`

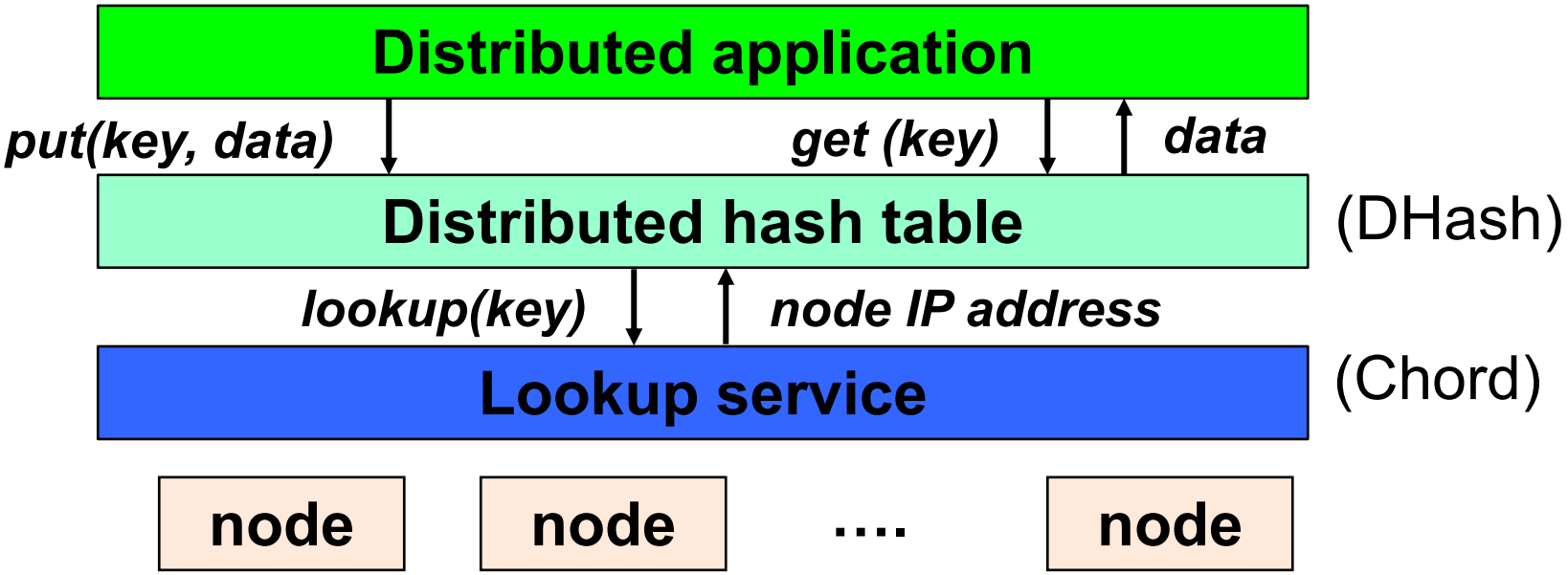
  `lookup(key) → IP addr` (Chord lookup service)

  `send-RPC(IP address, put, key, data)`

  `send-RPC(IP address, get, key) → data`

- **Partitioning data** in truly large-scale distributed systems
  - Tuples in a global database engine
  - Data blocks in a global file system
  - Files in a P2P file-sharing system

Cooperative storage with a DHT



- App may be **distributed** over many nodes
- DHT **distributes data storage** over many nodes

BitTorrent over DHT

- BitTorrent can use DHT instead of (or with) a tracker

- BT clients use DHT:
  - Key = **file content hash** ("infohash")
  - Value = **IP address of peer** willing to serve file
    - Can store multiple values (*i.e.* IP addresses) for a key

- Client does:
  - `get(infohash)` to find other clients willing to serve
  - `put(infohash, my-ipaddr)` to identify itself as willing

Why might DHT be a win for BitTorrent?

- The DHT comprises a single giant tracker, less fragmented than many trackers
  - So peers more likely to find each other

- Maybe a classic tracker too exposed to **legal *&* *c.* attacks**

Why the put/get DHT 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 (churn)

Today

1. Peer-to-Peer Systems

2. Distributed Hash Tables

3. **The Chord Lookup Service**
   - **Basic design**
   - Integration with *DHash* DHT, performance

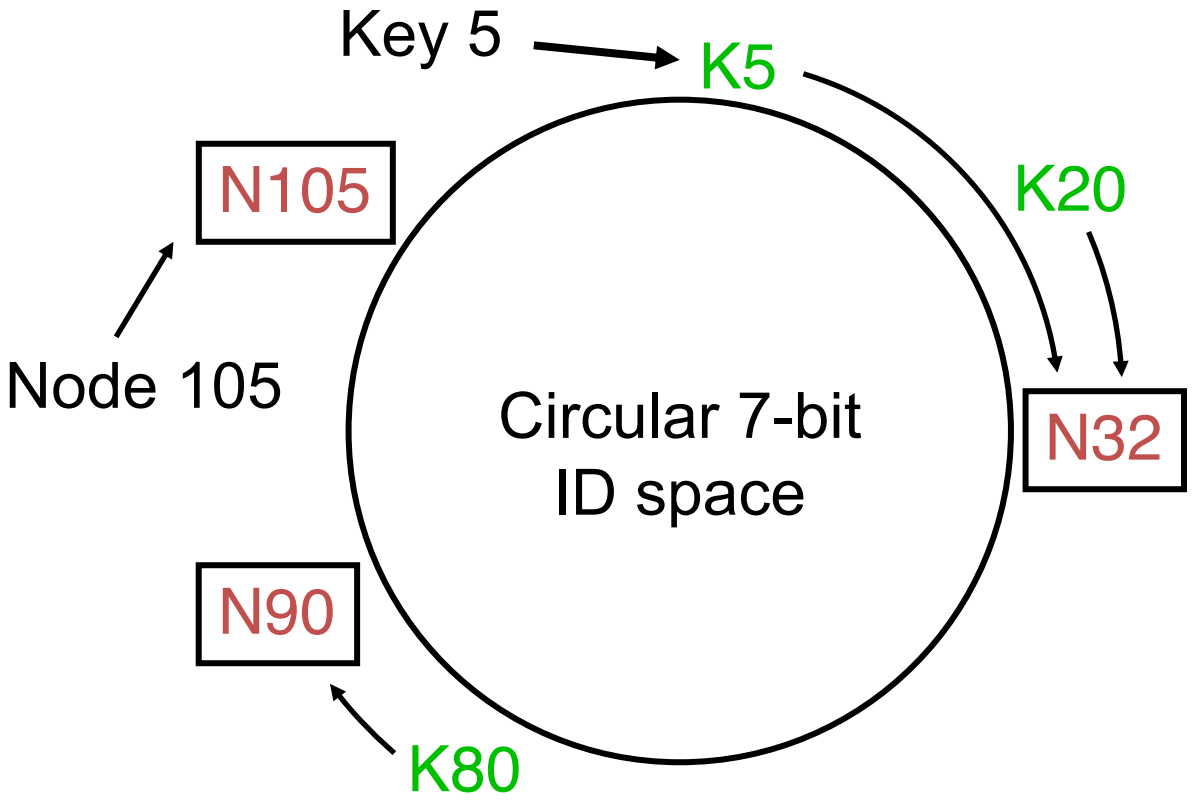
Chord lookup algorithm properties

- **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 identifiers

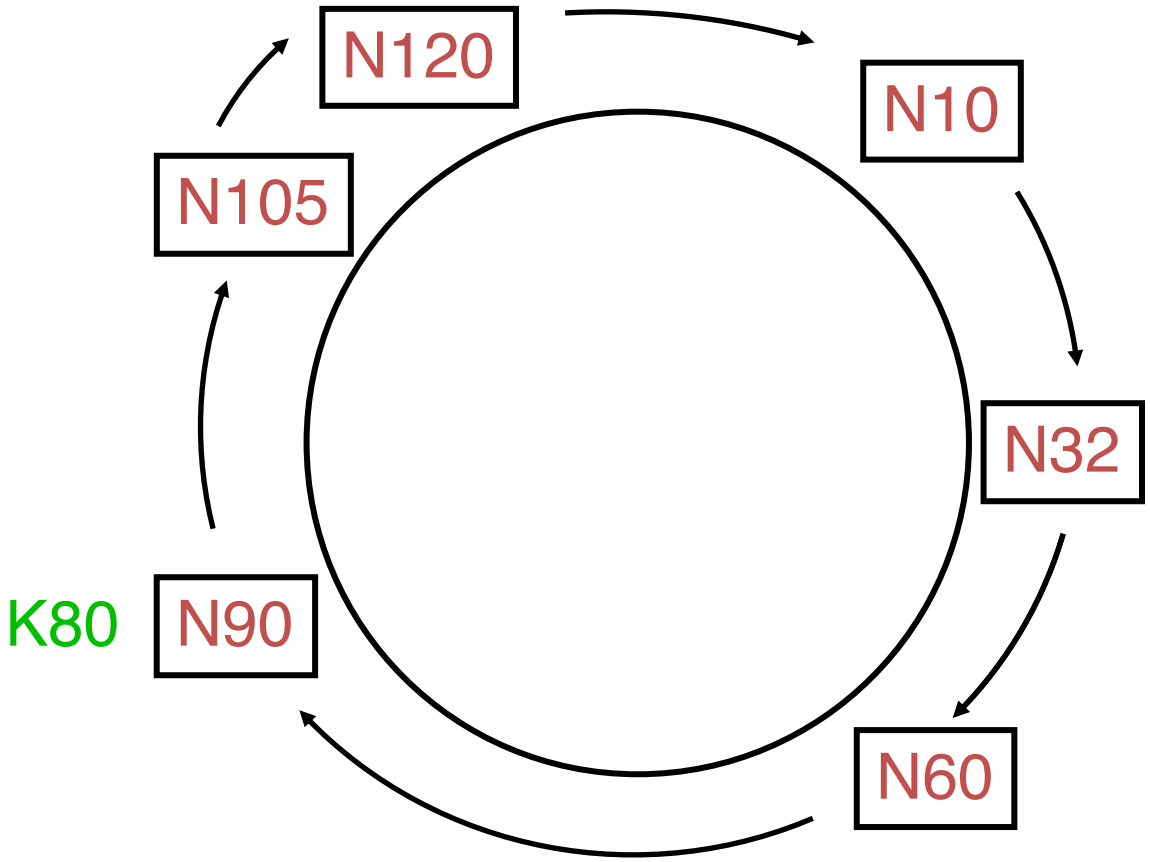
- **Key identifier** = SHA-1(key)
- **Node identifier** = SHA-1(IP address)
- SHA-1 distributes both uniformly
- *How does Chord partition data?*
  - *i.e.*, map key IDs to node IDs

Consistent hashing [Karger '97]

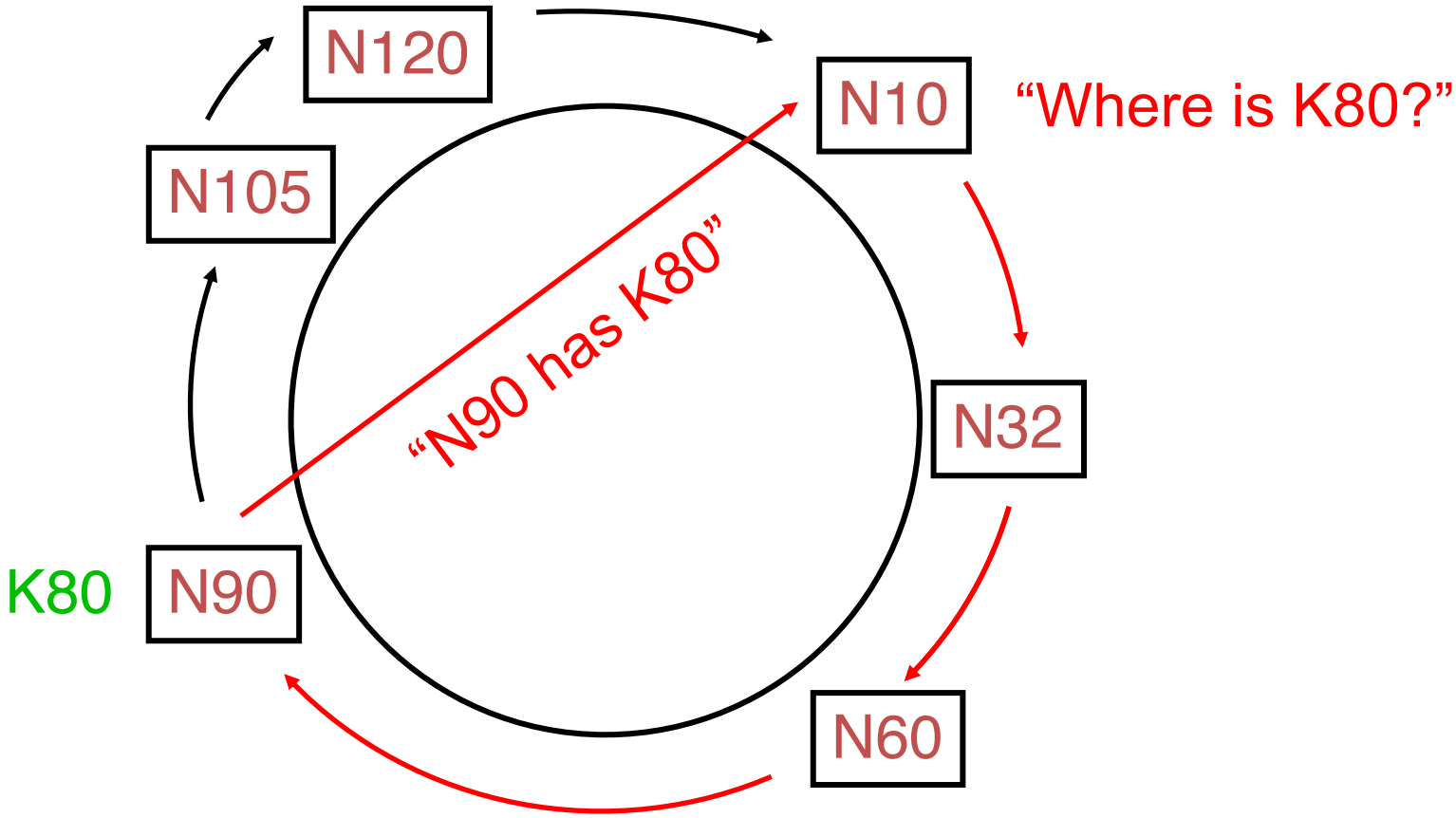


Key is stored at its successor: node with next-higher ID

Chord: Successor pointers



Basic lookup



Simple lookup algorithm

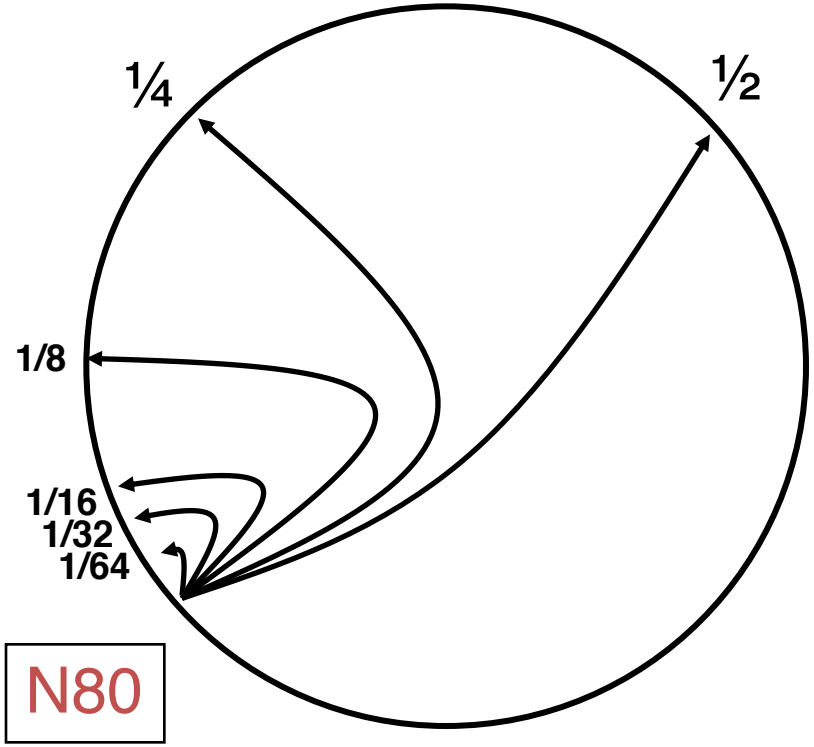
```
Lookup(key-id)
  succ ← my successor
  if succ between my-id & key-id // next
    return succ.Lookup(key-id)
  else                                  // done
    return succ
```

- Correctness depends only on successors

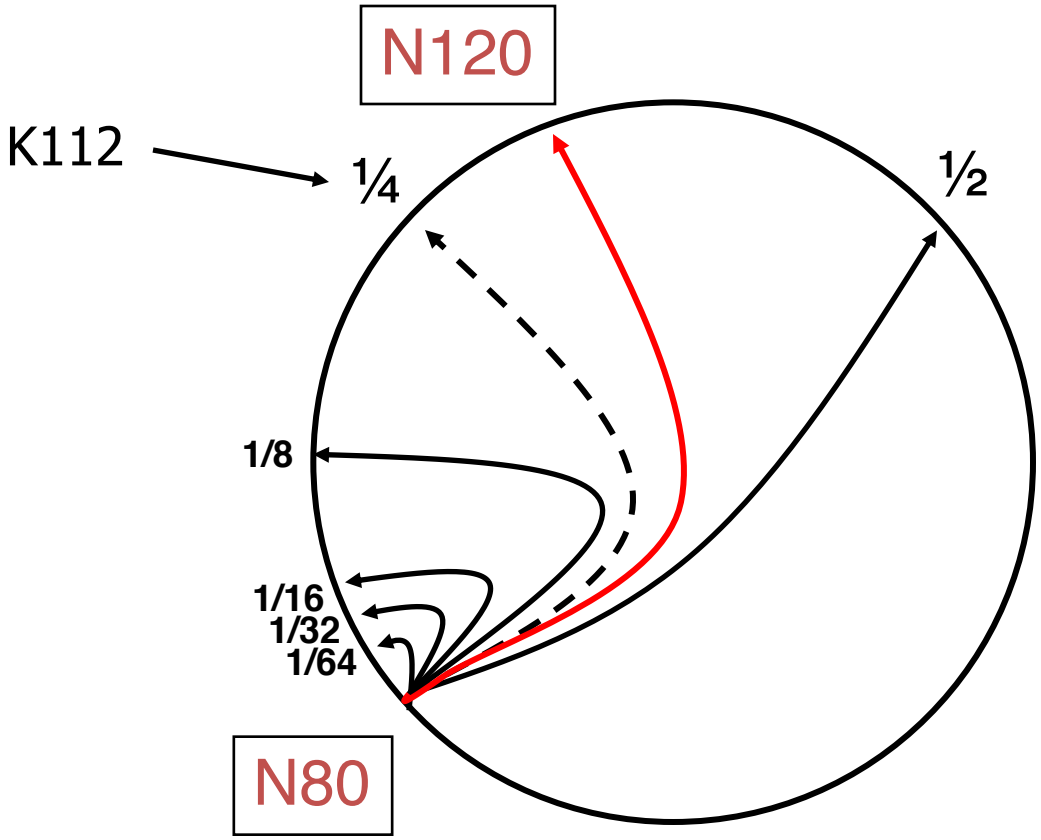
Improving performance

- **Problem:** Forwarding through successor is slow
- Data structure is a linked list: O(n)
- **Idea:** Can we make it more like a binary search?
  - Cut distance by half at each step

“Finger table” allows log N-time lookups



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



Implication of finger tables

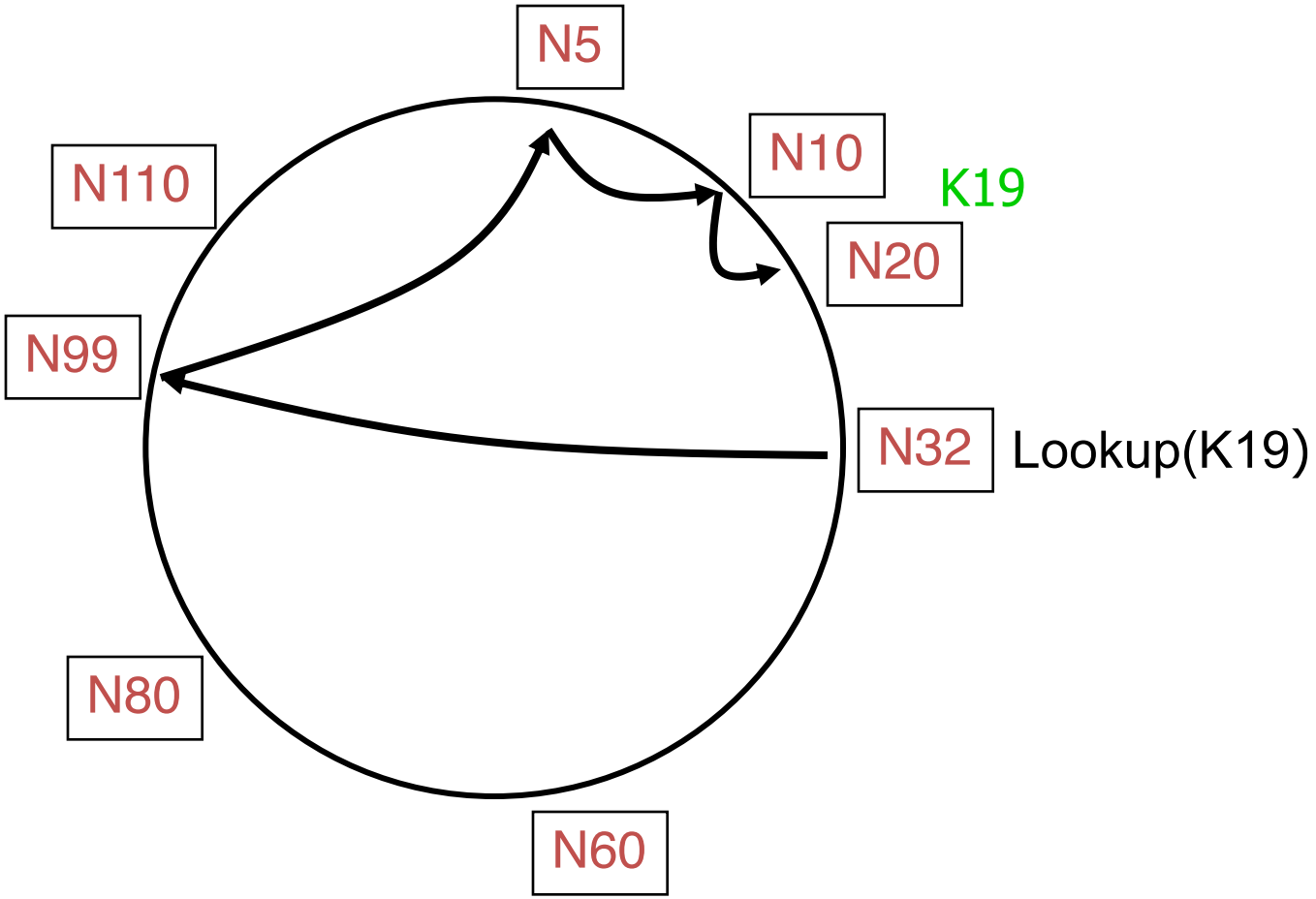
- A **binary lookup tree** rooted at every node
  - Threaded through other nodes' finger tables

- This is better than simply arranging the nodes in a single tree
  - Every node acts as a root
    - So there's no root hotspot
    - No single point of failure
    - But a lot more state in total

Lookup with finger table

```
Lookup(key-id)
  look in local finger table for
     highest n: n between my-id & key-id
  if n exists
     return n.Lookup(key-id)          // next
  else
     return my successor              // done
```

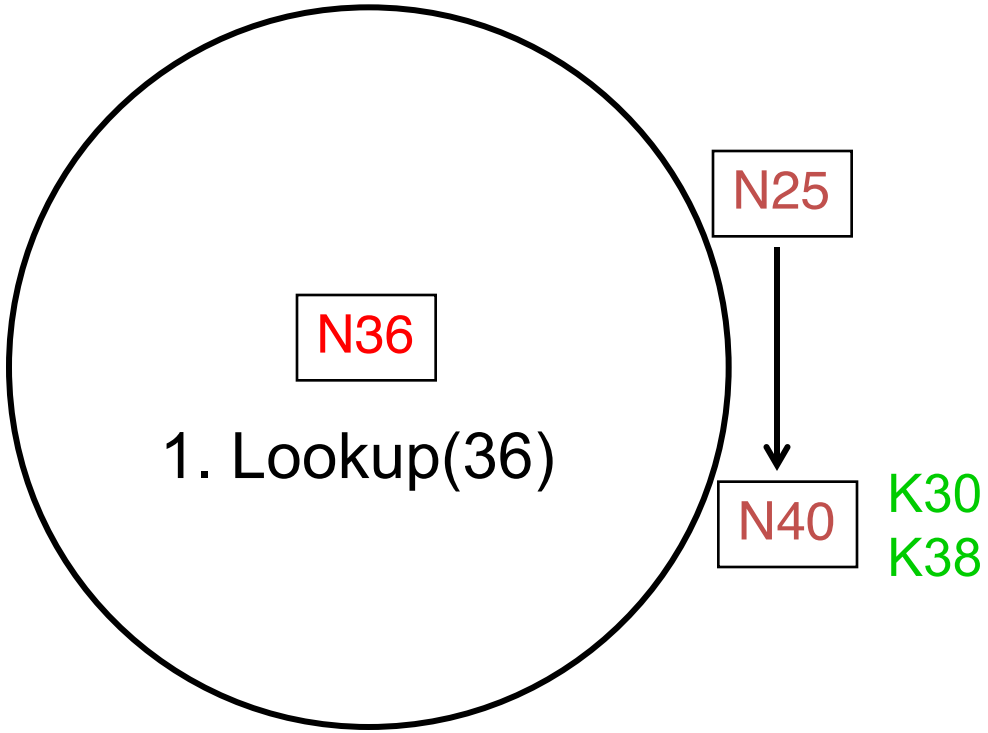
Lookups Take O(log *N*) Hops



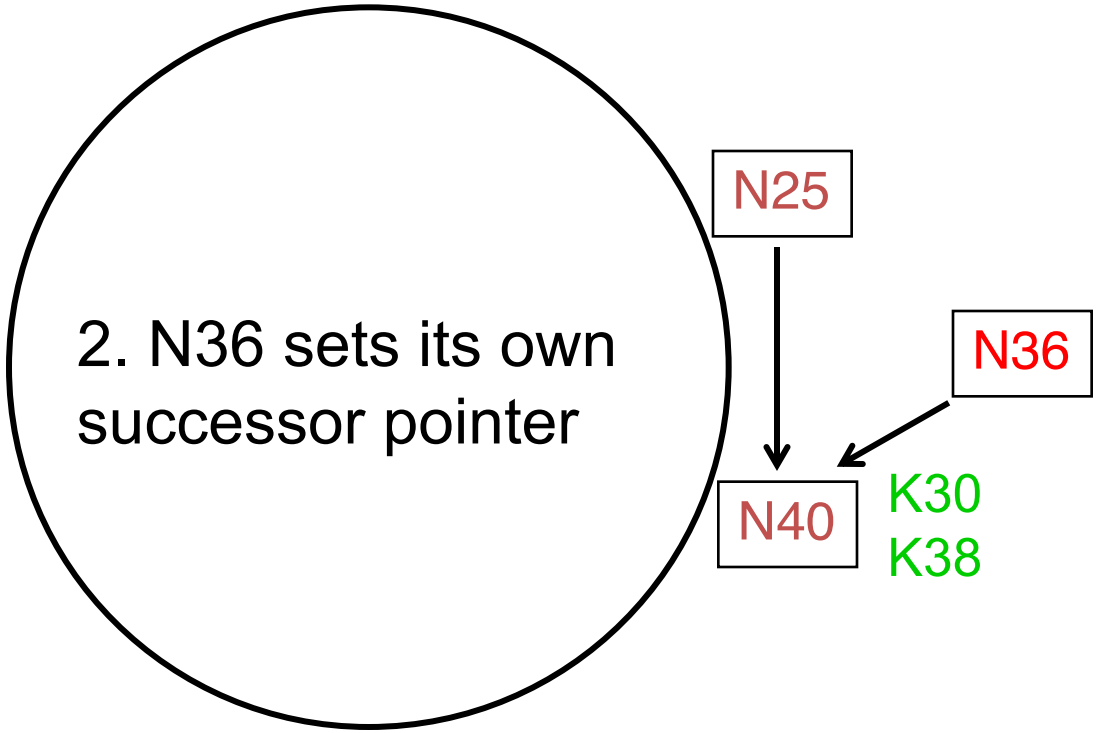
An aside: Is log(n) fast or slow?

- For a million nodes, it's 20 hops
- If each hop takes 50 milliseconds, lookups take a second
- If each hop has 10% chance of failure, it's a couple of timeouts
- So in practice log(n) is better than O(n) but not great

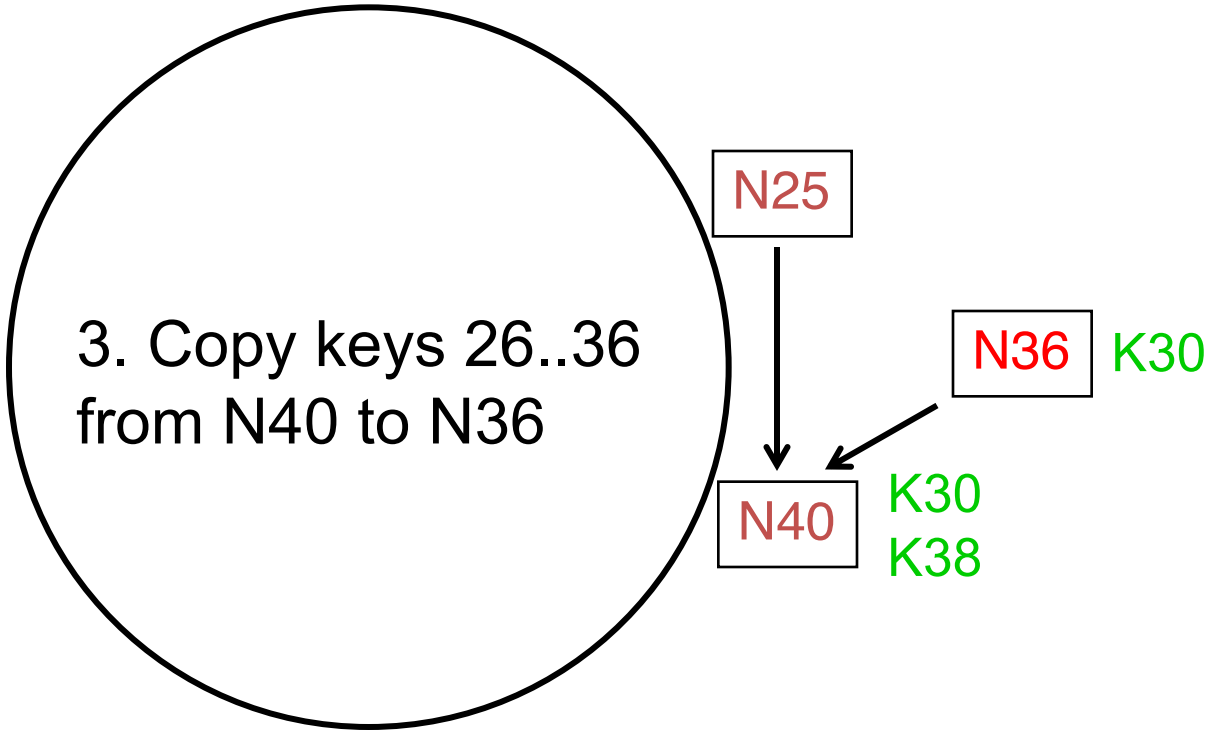
Joining: Linked list insert



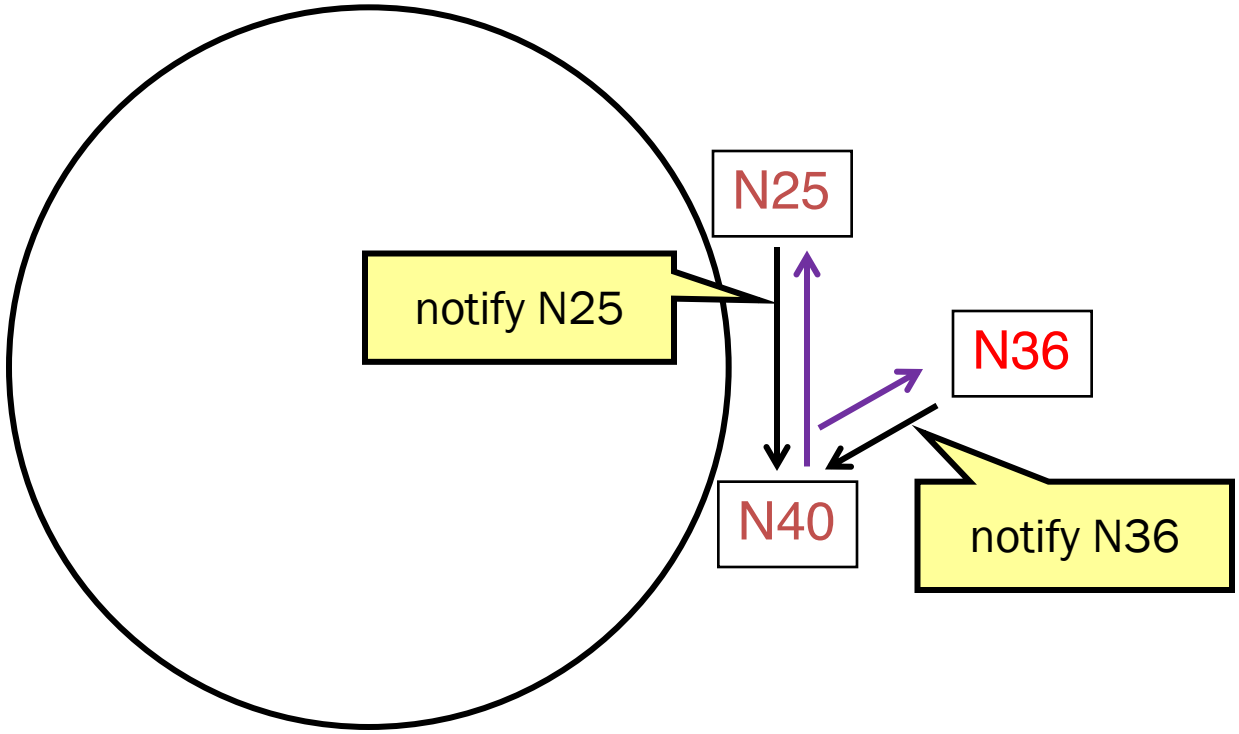
Join (2)



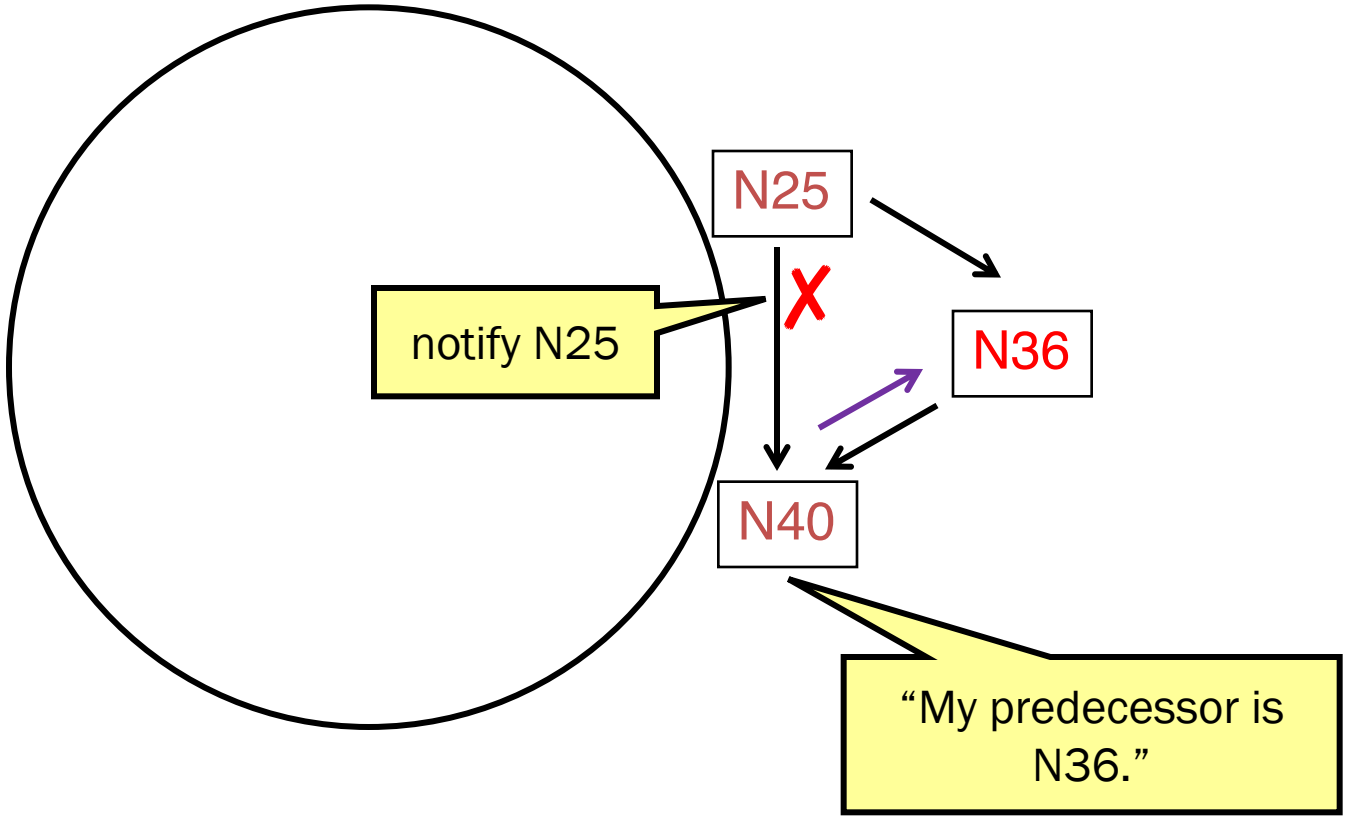
Join (3)



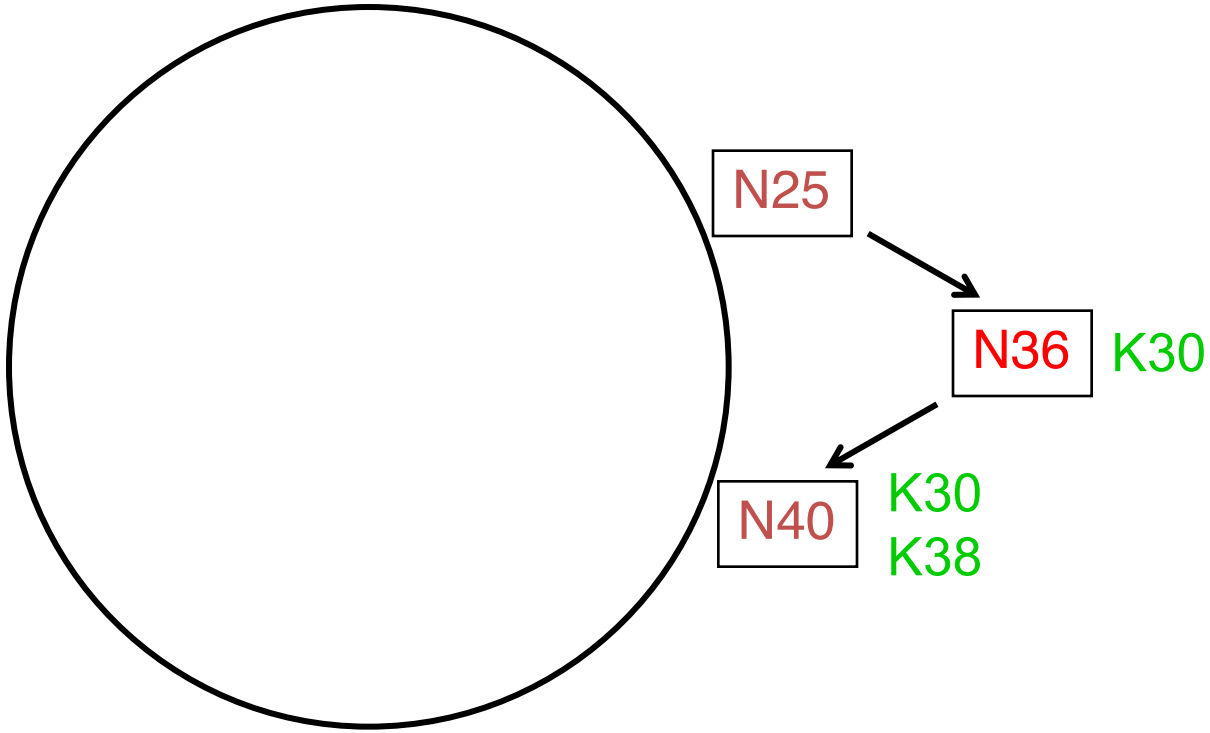
Notify messages update predecessors



Successors fixed when inconsistent

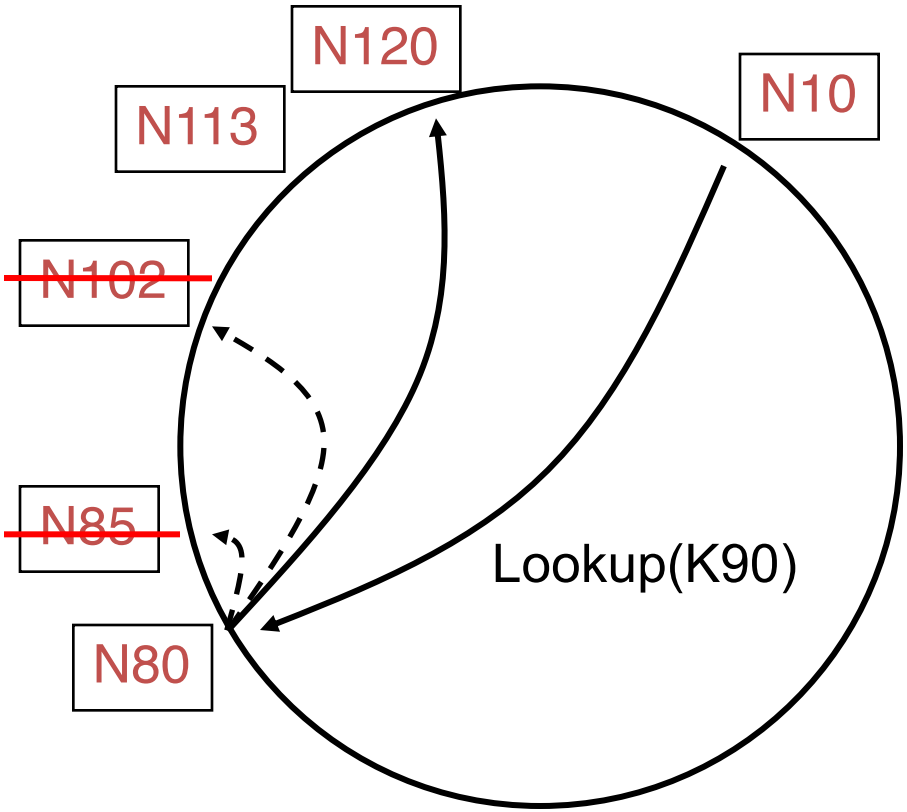


Joining: Summary



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

Failures may cause incorrect lookup



N80 does not know the closest living successor!
Breaks having replicas on k immediate successors

Successor lists

- Each node stores a **list** of its ***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 **one half** of the nodes **fail**
- P(successor list all dead) = $(\frac{1}{2})^r$
  - *i.e.*, P(this node breaks the Chord ring)
  - Depends on independent failure
- Successor list of **size r = O(log N)** makes this probability $1/N$: low for large N

Lookup with fault tolerance

```
Lookup(key-id)
  look in local finger table and successor-list
        for highest n: n between my-id & key-id
  if n exists
        return n.Lookup(key-id)                    // next
        if call failed,
              remove n from finger table and/or
                    successor list
              return Lookup(key-id)
  else
      return my successor                          // done
```

Today

1. Peer-to-Peer Systems

2. Distributed Hash Tables

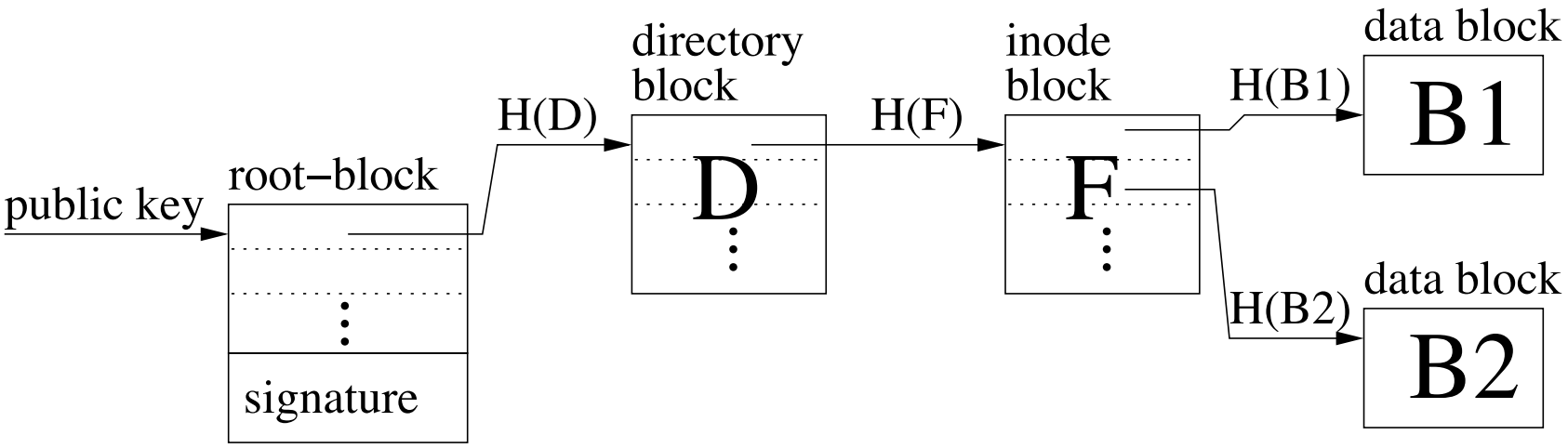
3. **The Chord Lookup Service**
   - Basic design
   - **Integration with *DHash* DHT, performance**

The DHash DHT

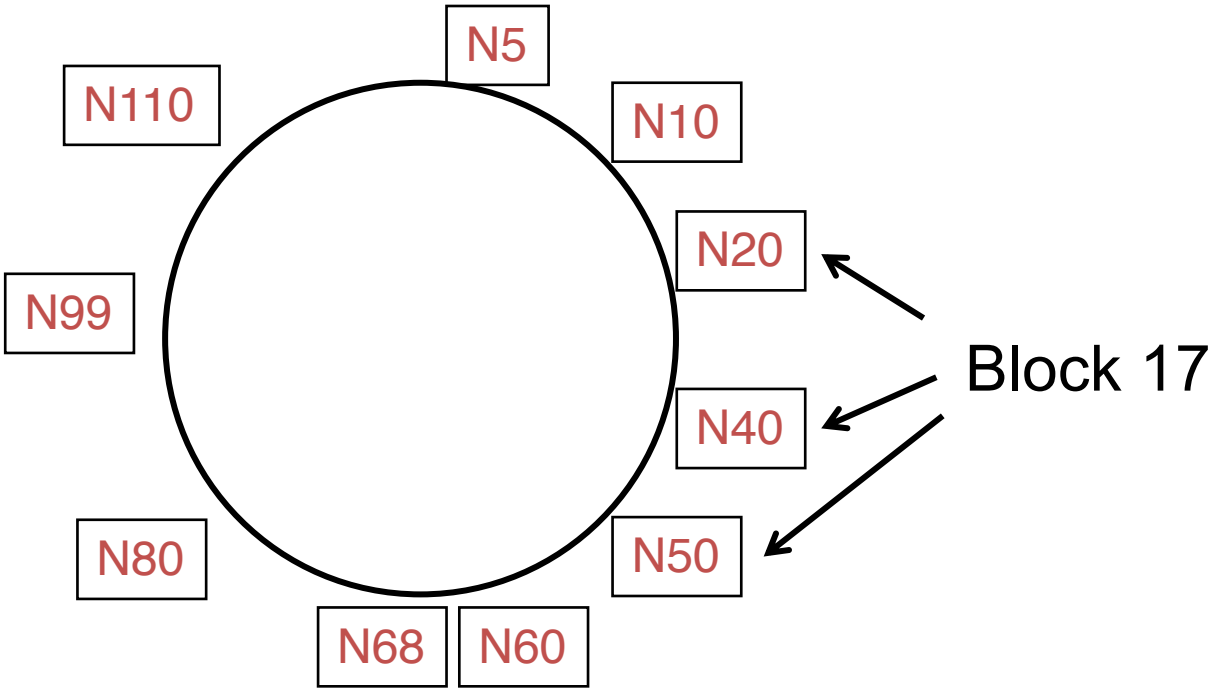
- Builds key/value storage on Chord
- **Replicates** blocks for availability
  - Stores k **replicas** at the k **successors** after the block on the Chord ring
- **Caches** blocks for load balancing
  - **Client** sends **copy of block** to each of the servers it contacted along the **lookup path**
- **Authenticates** block contents

DHash data authentication

- Two types of DHash blocks:
  - Content-hash: key = SHA-1(data)
  - Public-key: key is a cryptographic public key, data are signed by corresponding private key
- Chord File System example:



DHash replicates blocks at *r* successors



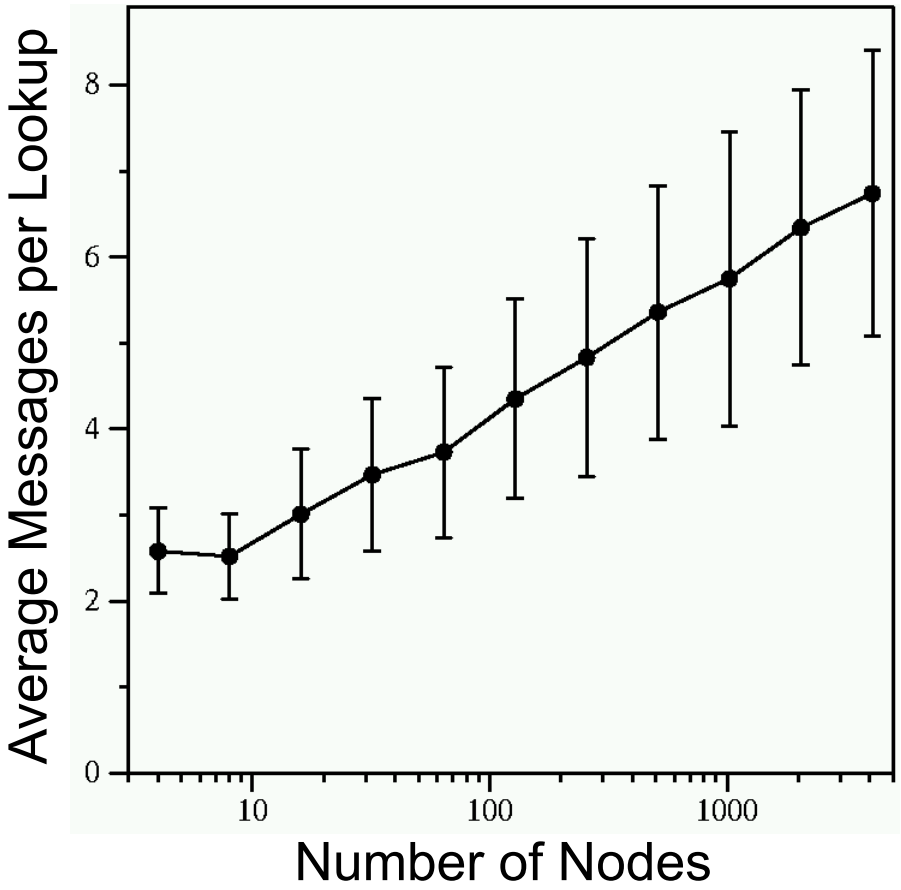
- **Replicas** are **easy to find** if successor fails
- Hashed node IDs ensure **independent failure**

Experimental overview

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

Goal: Experimentally confirm theoretical results

Chord lookup cost is O(log N)



Constant is 1/2

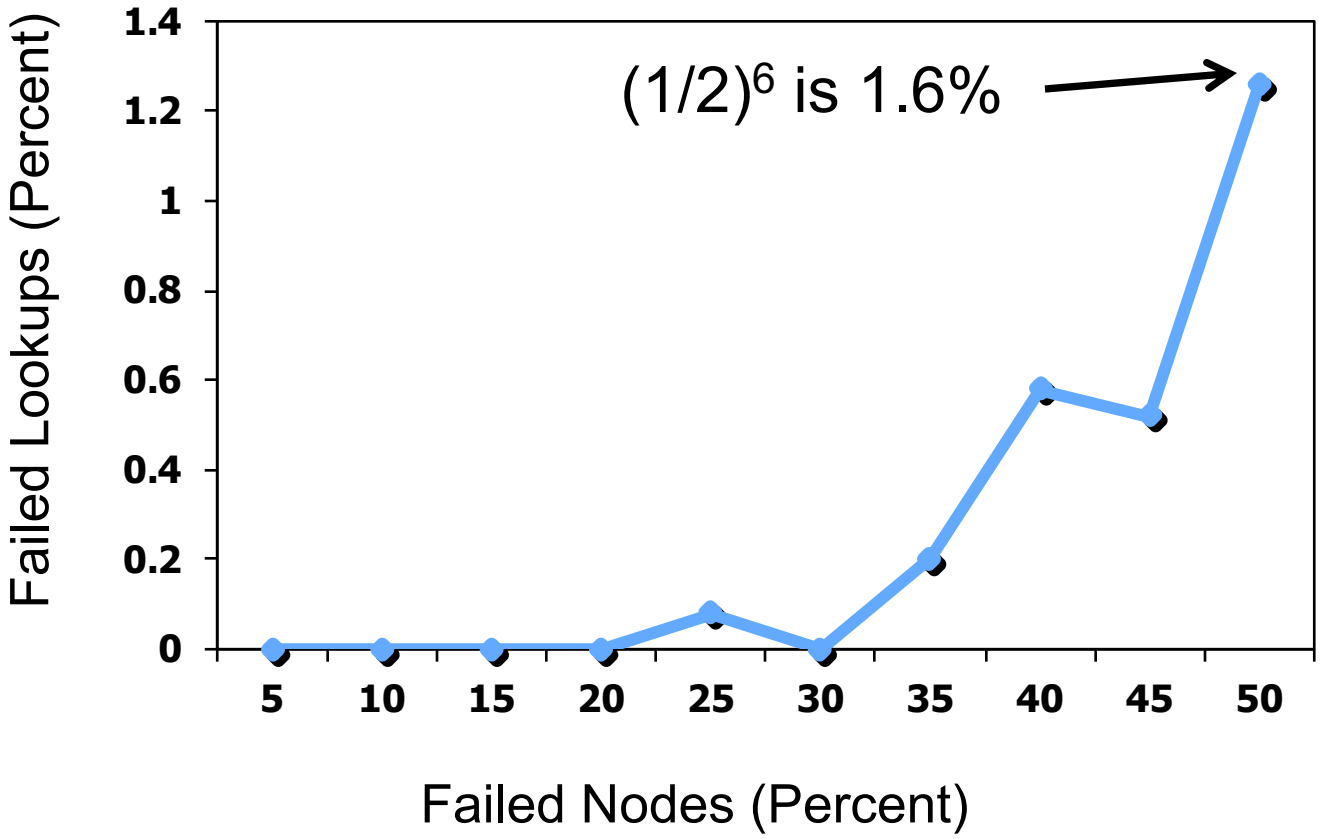
Failure experiment setup

- Start **1,000 Chord servers**
  - Each server's **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 make 1,000 lookups

Massive failures have little impact



Today

1. Peer-to-Peer Systems

2. Distributed Hash Tables

3. The Chord Lookup Service
   - Basic design
   - Integration with *DHash* DHT, performance

4. **Concluding thoughts on DHT, P2P**

DHTs: Impact

- 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, OceanStore, Pond, PAST)
  - Naming systems (e.g., SFR, Beehive)
  - DB query processing [PIER, Wisc]
  - Content distribution systems (e.g., Coral)
  - distributed databases (e.g., PIER)

Why don't all services use P2P?

1. High latency and limited bandwidth between peers (*cf.* between server cluster in datacenter)

2. User computers are less reliable than managed servers

3. Lack of trust in peers' correct behavior
   - Securing DHT routing hard, unsolved in practice

DHTs in retrospective

- Seem promising for finding data in large P2P systems
- Decentralization seems good for load, fault tolerance

- **But:** the security problems are difficult
- **But:** churn is a problem, particularly if log(n) is big

- So DHTs have not had the impact that many hoped for

What DHTs got right

- Consistent hashing
  - Elegant way to divide a workload across machines
  - Very useful in clusters: actively used today in Amazon Dynamo and other systems
- Replication for high availability, efficient recovery after node failure
- Incremental scalability: “add nodes, capacity increases”
- Self-management: minimal configuration
- **Unique trait:** no single server to shut down/monitor