Peer-to-Peer Systems and Distributed Hash Tables

COS 418: Distributed Systems
Lecture 1

Wyatt Lloyd

Slides adapted from Kyle Jamison's, which are partially adapted from Brad Karp's and Robert Morris

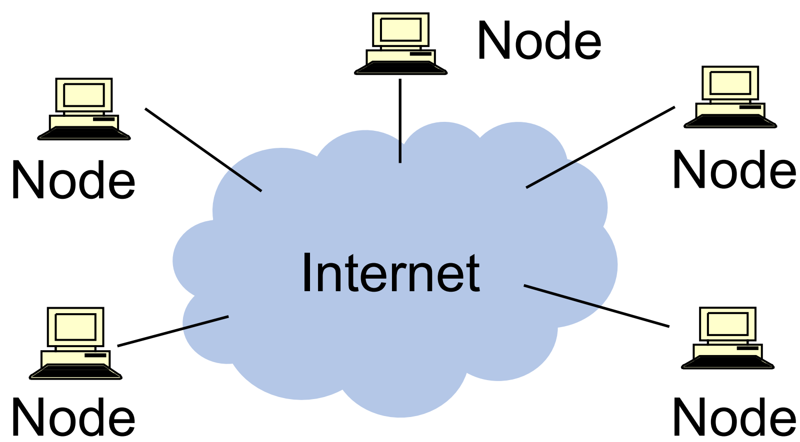
Today

1. Peer-to-Peer Systems

2. Distributed Hash Tables

3. The Chord Lookup Service

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

Advantages of P2P systems

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

- No centralized server or servers may mean:
  - Less chance of service overload as load increases
  - 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)

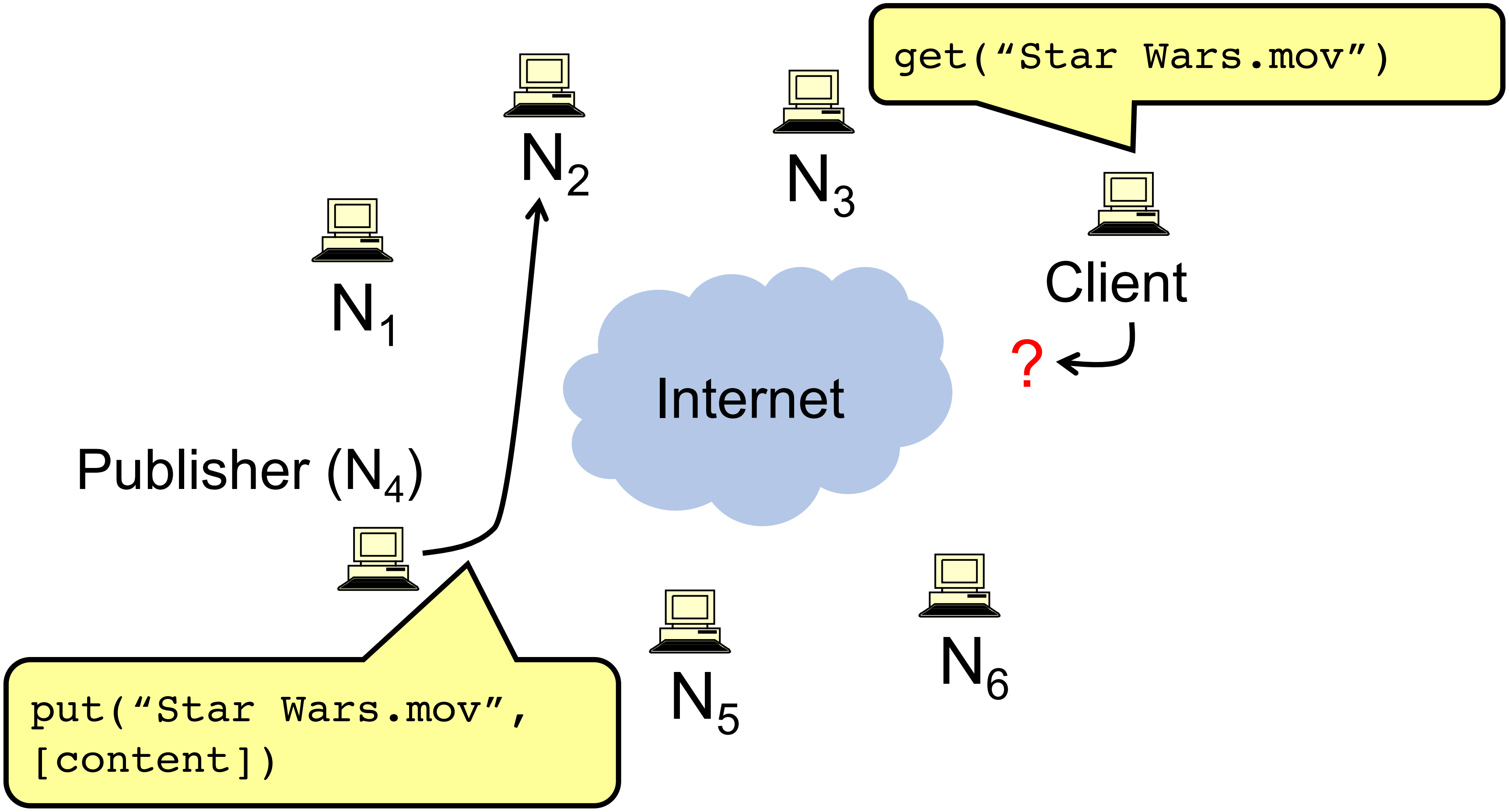
3. Voice/video telephony
   - Skype used to do this…

Example: Classic BitTorrent

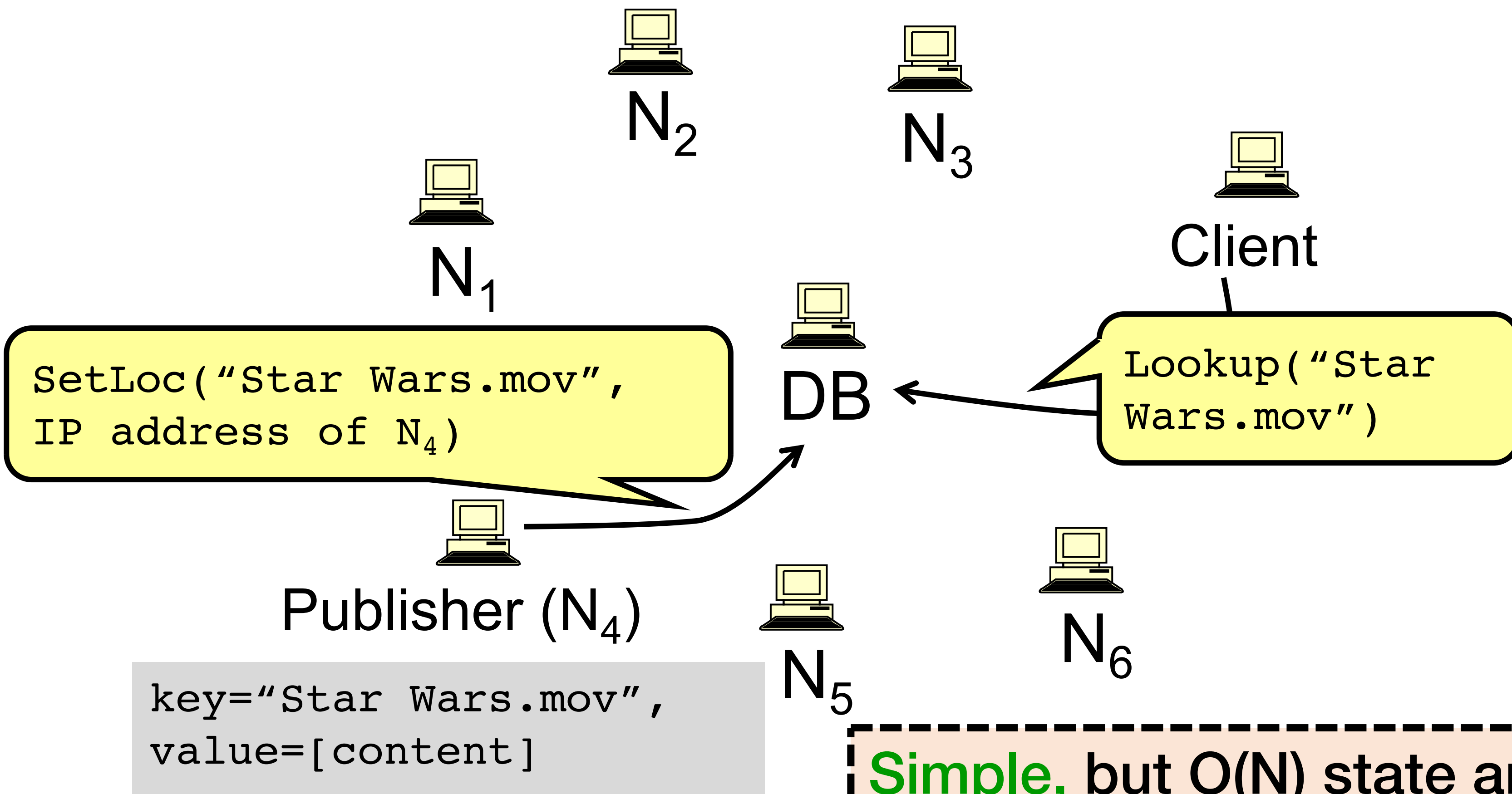
1. User clicks on download link
   - Gets torrent file with content hash, IP address of tracker
2. User's BitTorrent (BT) client talks to tracker
   - Tracker tells it list of peers who have file
3. User's BT client downloads file from one or more peers
4. User's BT client tells tracker it has a copy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

The lookup problem

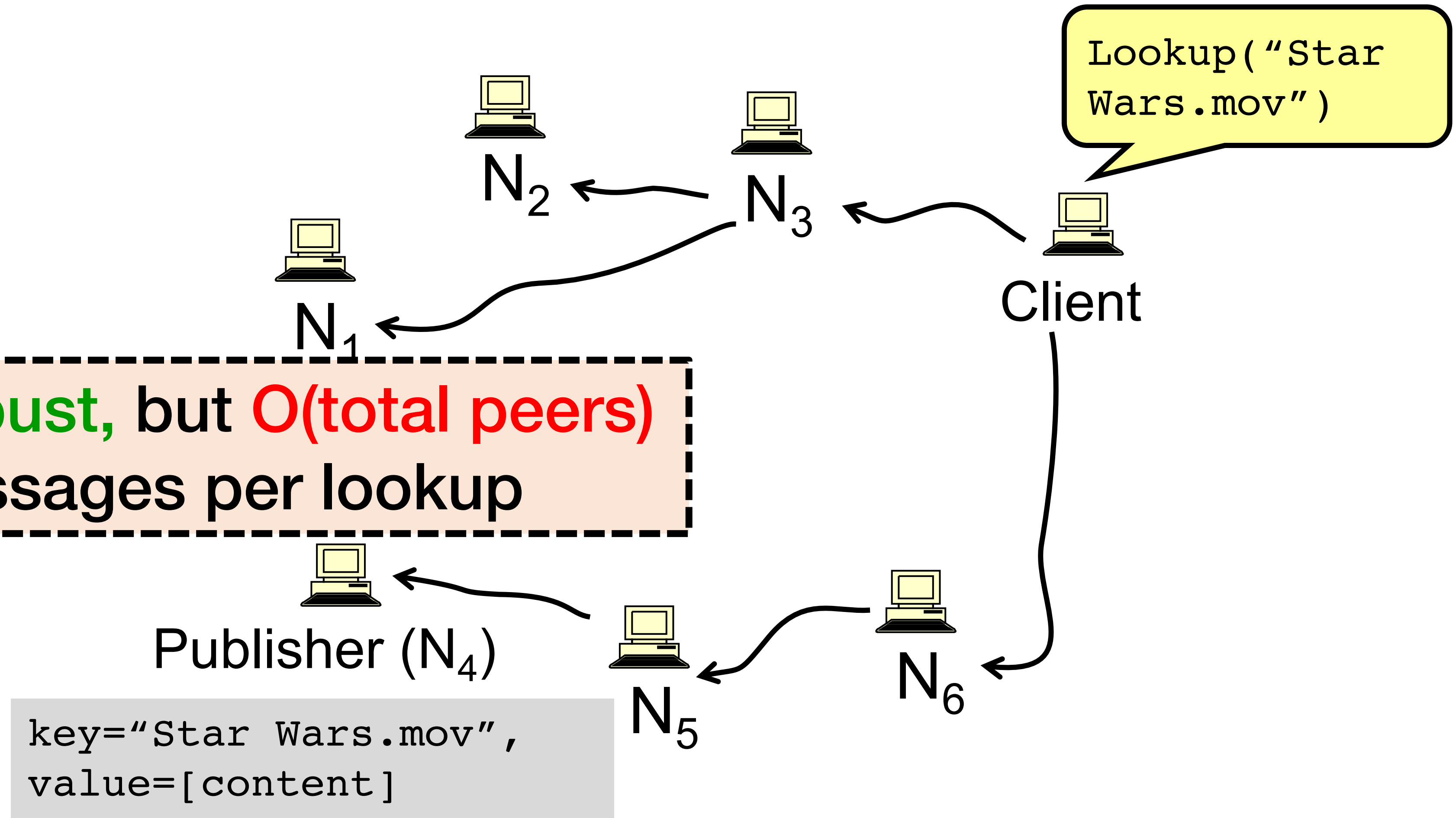


Centralized lookup (Napster)



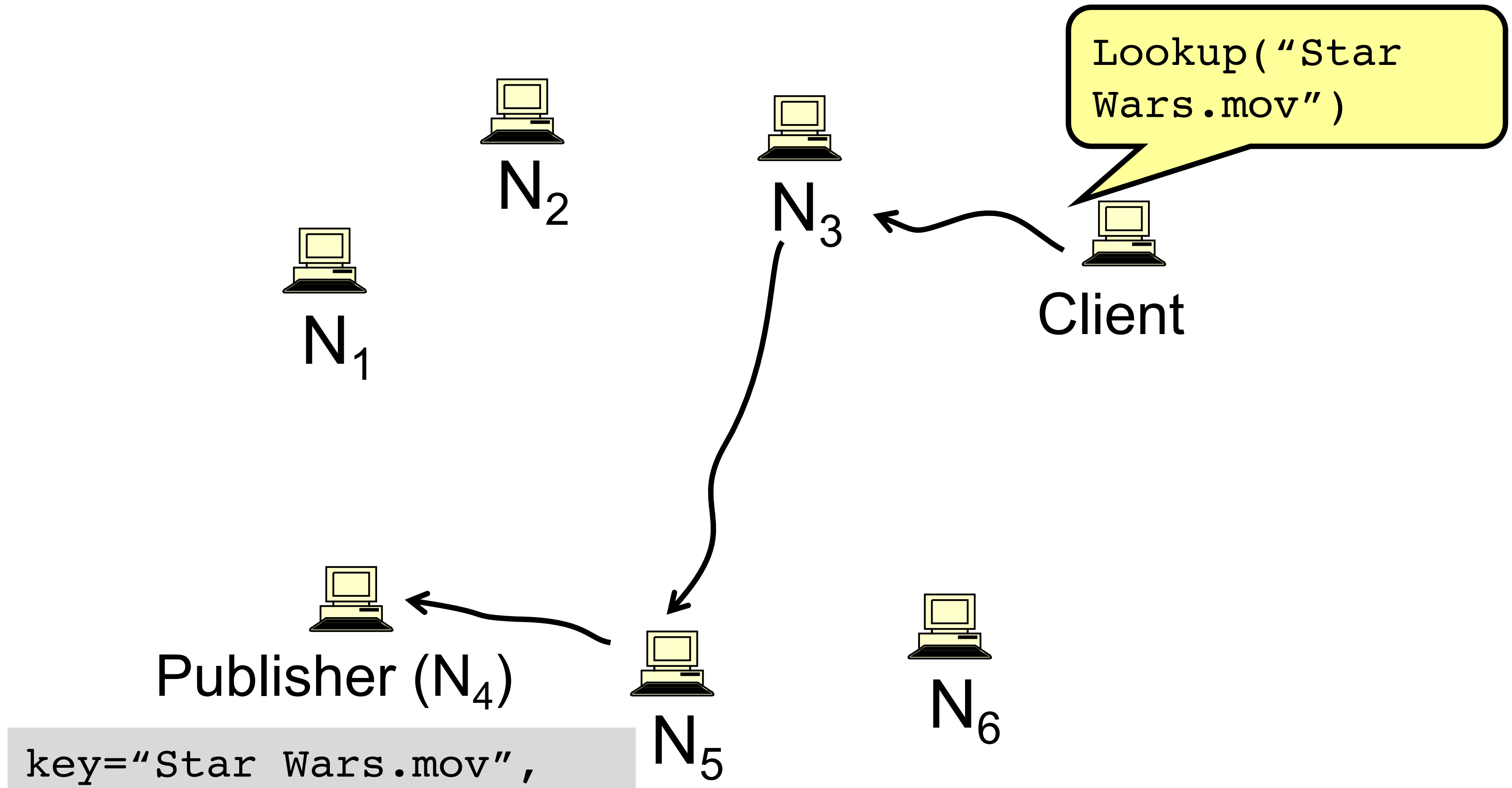
Simple, but O(N) state and a single point of failure

Flooded queries (original Gnutella)



Robust, but O(total peers) messages per lookup

Routed DHT queries (Chord)



Goal: robust, reasonable state, reasonable number of hops?

Today

1. Peer-to-Peer Systems

2. Distributed Hash Tables

3. The Chord Lookup Service

What is a DHT?

- Local hash table:

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

- Service: Constant-time insertion and lookup

Distributed Hash Table (DHT):
Do (roughly) this across millions of hosts on the Internet!

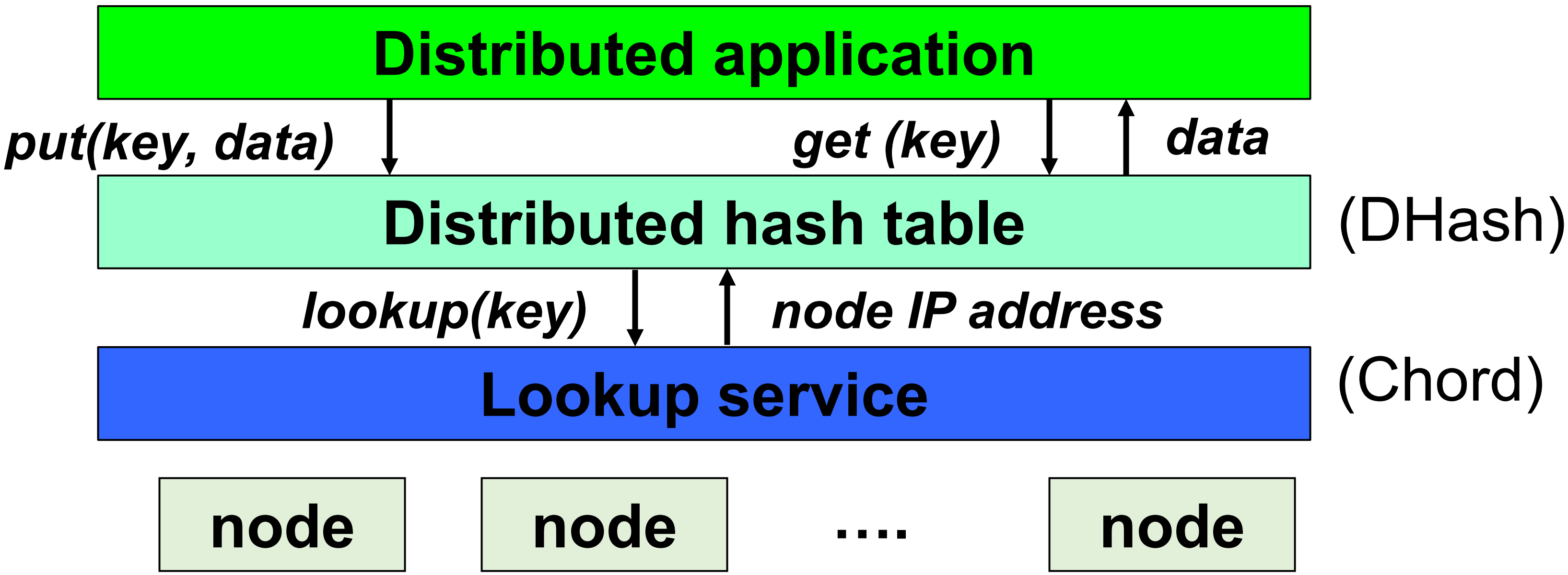
What is a DHT (and why)?

- 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
```

- Partitions data in a large-scale distributed system
  - 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 DHT for BitTorrent?

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

- Classic BitTorrent tracker is a single point of failure
  - Legal attacks
  - DoS 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**

What is hard in DHT design?

- **Decentralized:** no central authority
- **Scalable:** low network traffic overhead
- **Efficient:** find items quickly (latency)
- **Robust:** nodes fail, new nodes join

Today

1. Peer-to-Peer Systems
2. Distributed Hash Tables
3. The Chord Lookup Service

Chord lookup algorithm

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

Chord Lookup: Identifiers

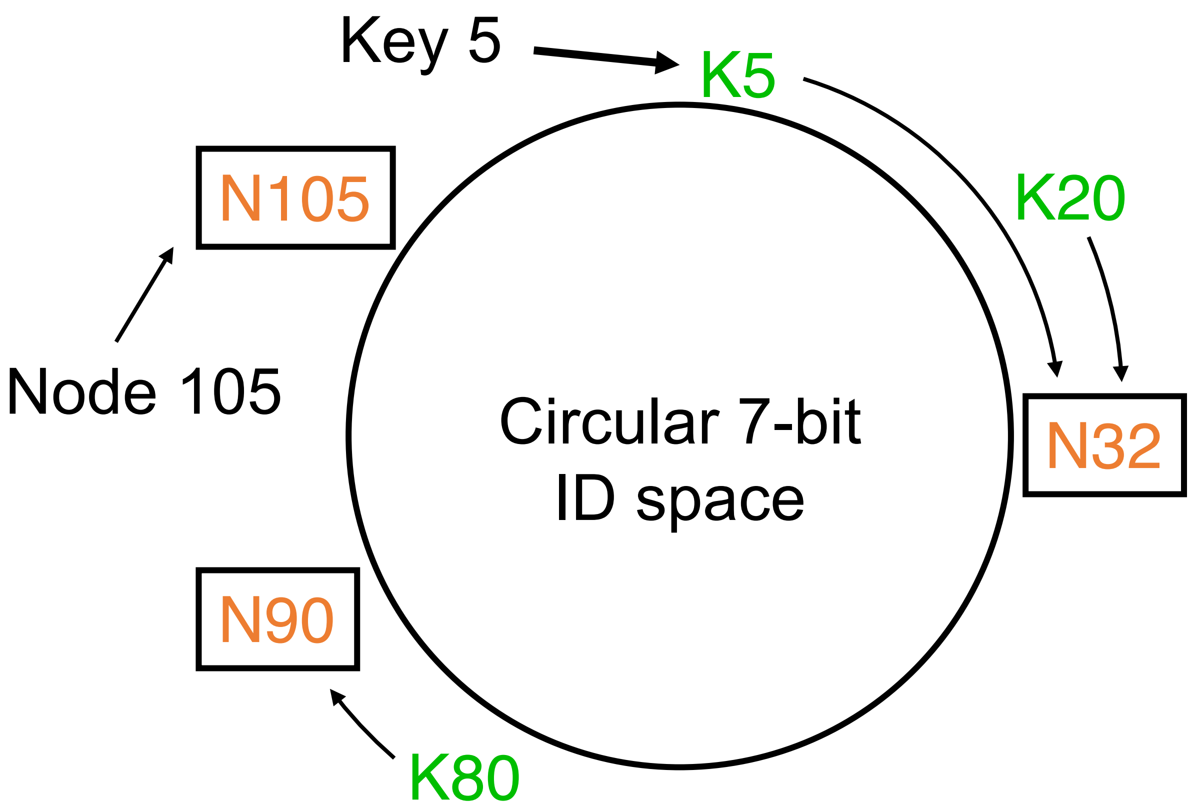
- Key identifier = SHA-1(key)

- Node identifier = SHA-1(IP address)

- SHA-1 distributes both uniformly

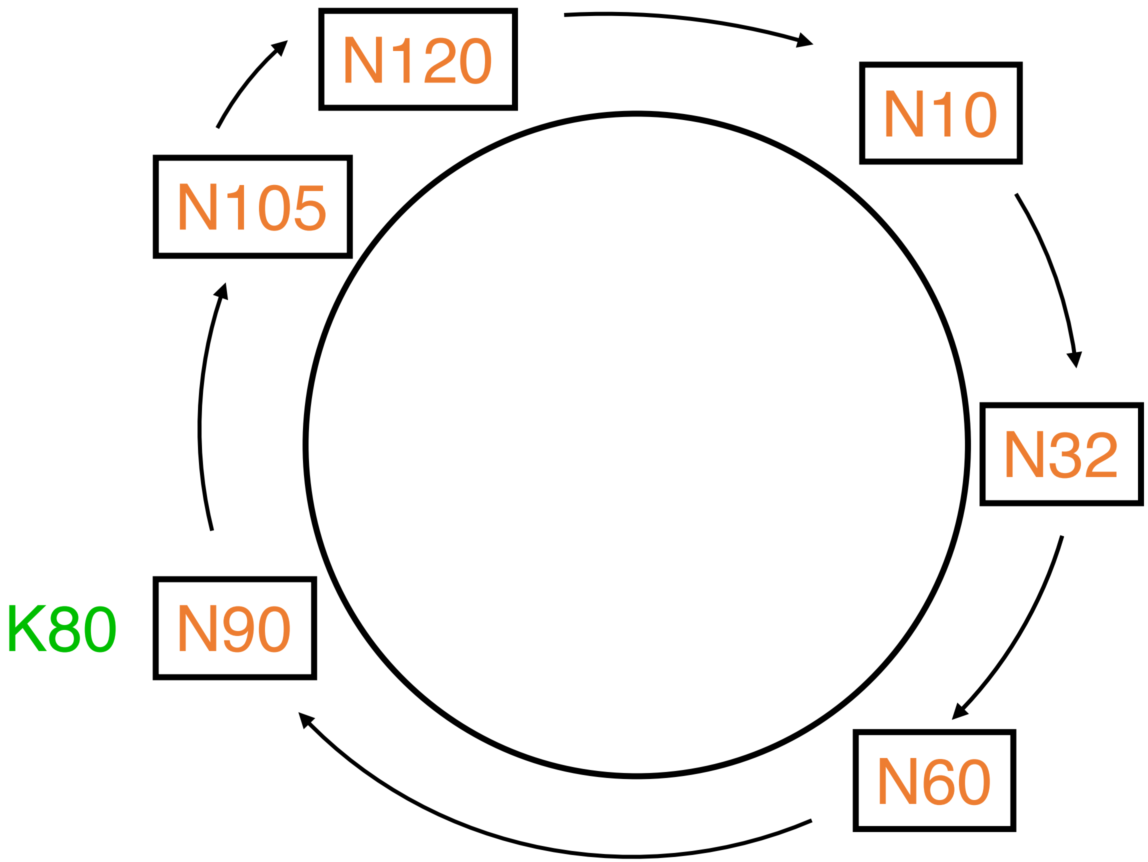
- How does Chord partition data?
    - i.e., map key IDs to node IDs

Consistent hashing

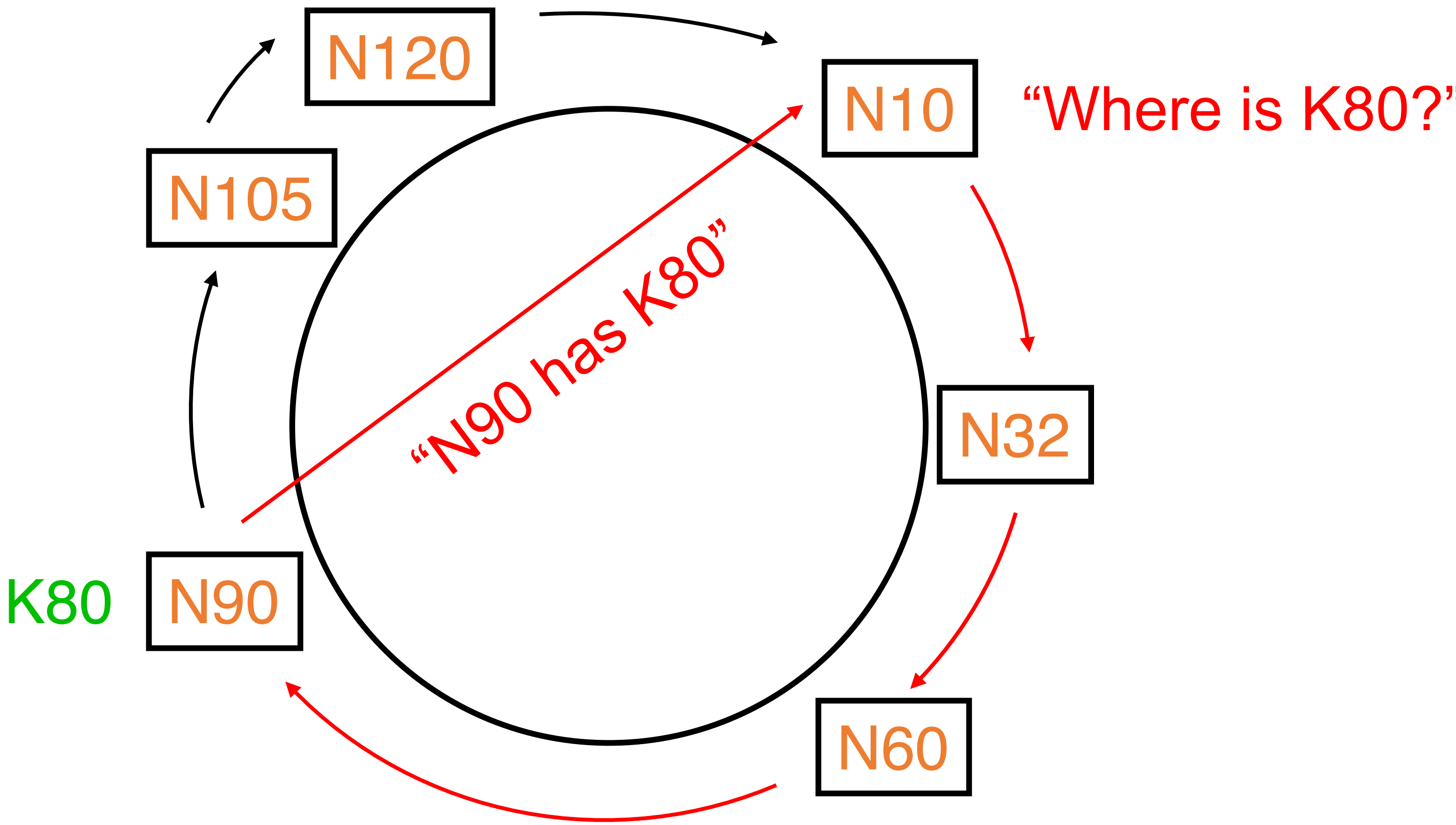


Key is stored at its successor: node with next-higher ID

Chord: Successor pointers



Basic lookup



Simple lookup algorithm

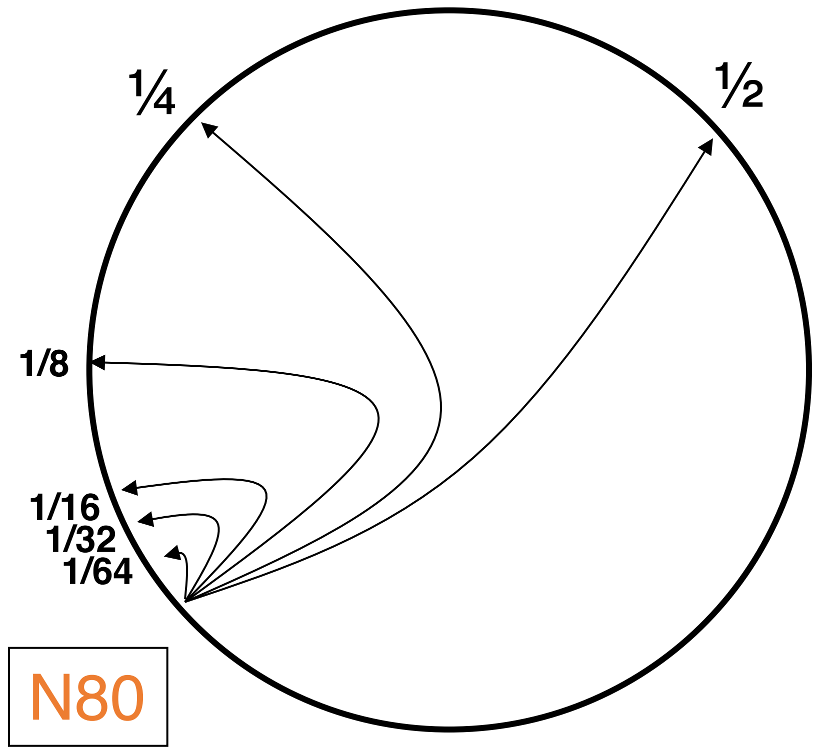
```
Lookup(key-id)
  succ ← my successor
  if my-id < succ < key-id   // next hop
      call Lookup(key-id) on succ
  else                       // done
    return succ
```

- Correctness depends only on successors

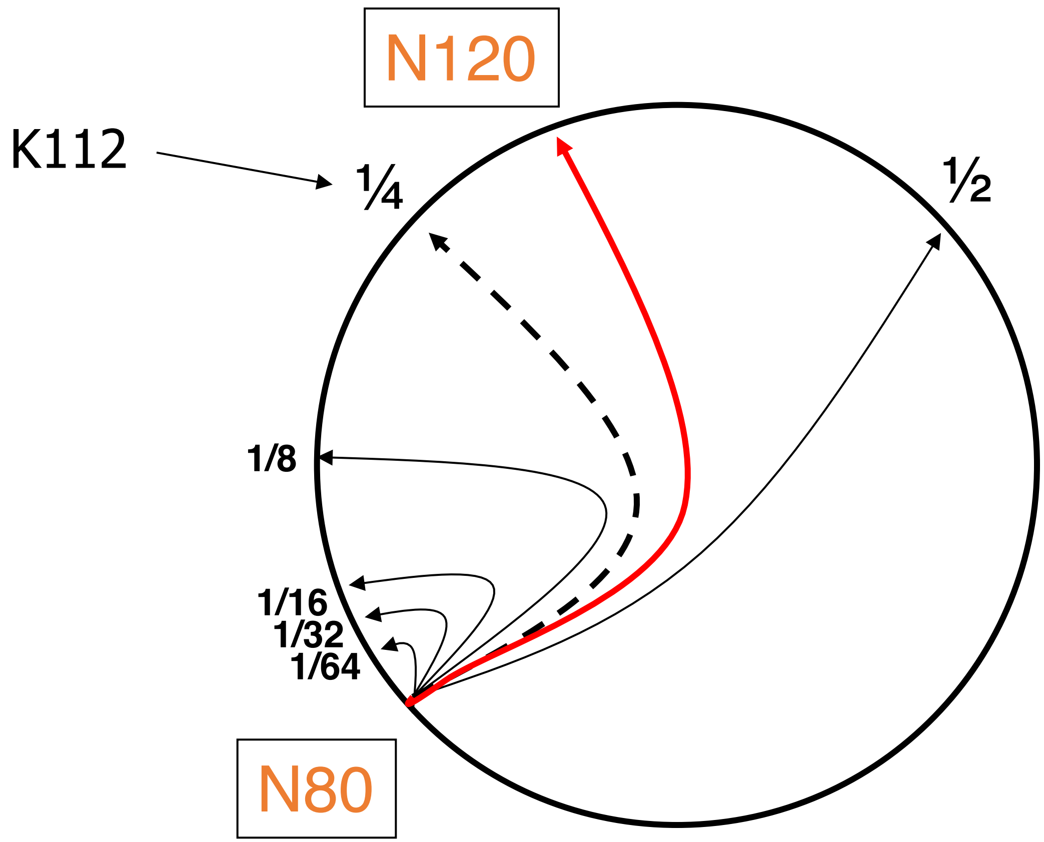
Improving performance

- Problem: Forwarding through successor is slow
- Data structure is a linked list: O(n)
- Idea: Can we make it more like a binary search?
  - Need to be able to halve distance at each step

"Finger table" for O(log *N*)-time lookups



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



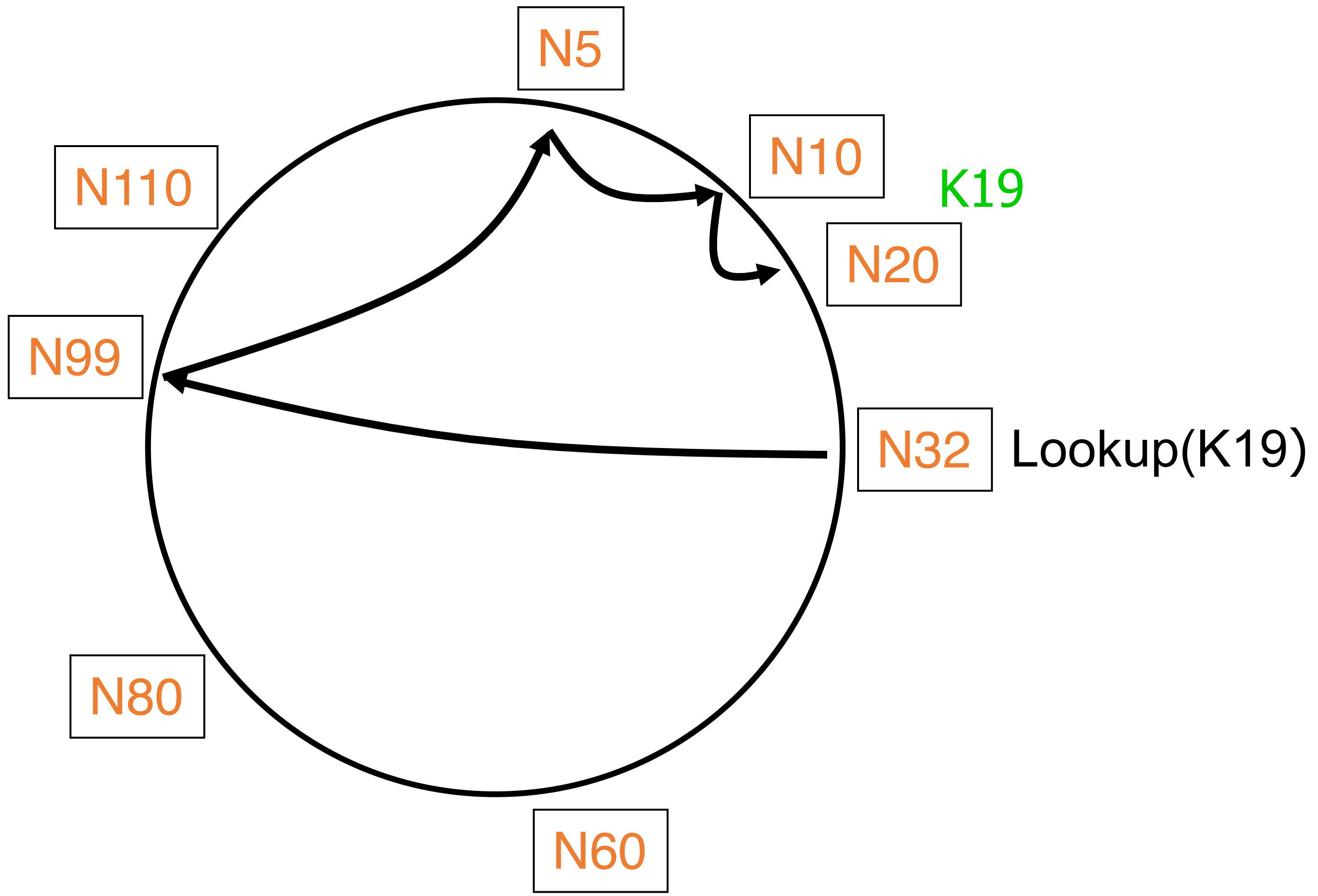
Implication of finger tables

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

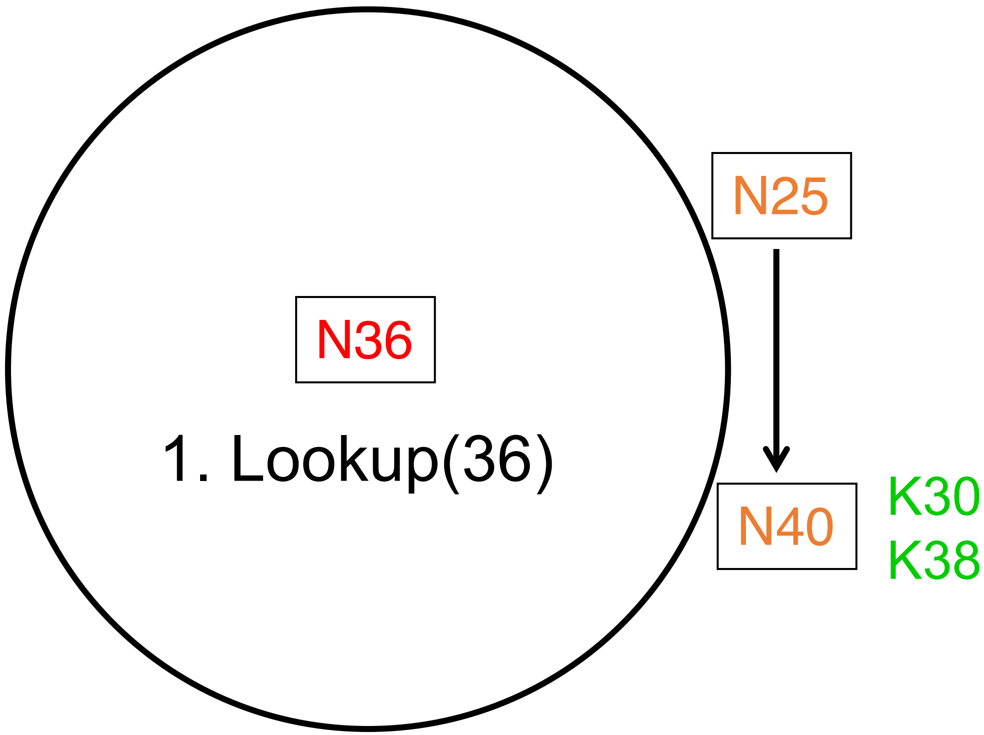
Lookups Take O(log N) Hops



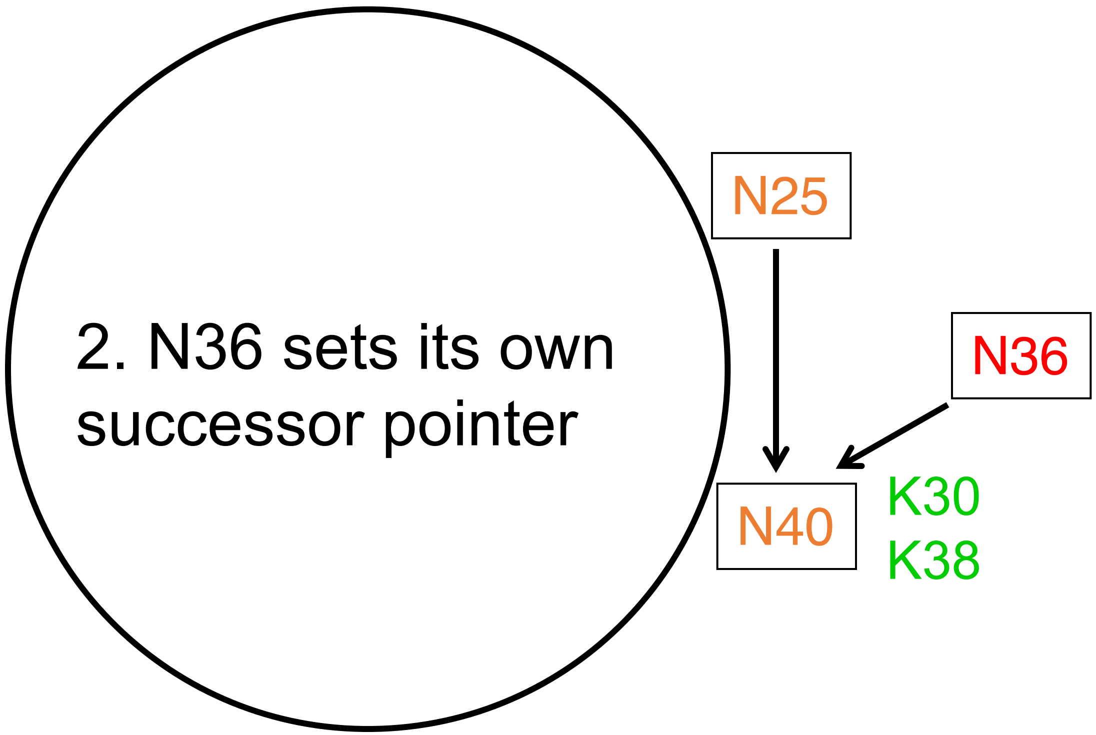
Aside: Is O(log *N*) fast or slow?

- For a million nodes, it's 20 hops
- If each hop takes 50 milliseconds, lookups take one 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

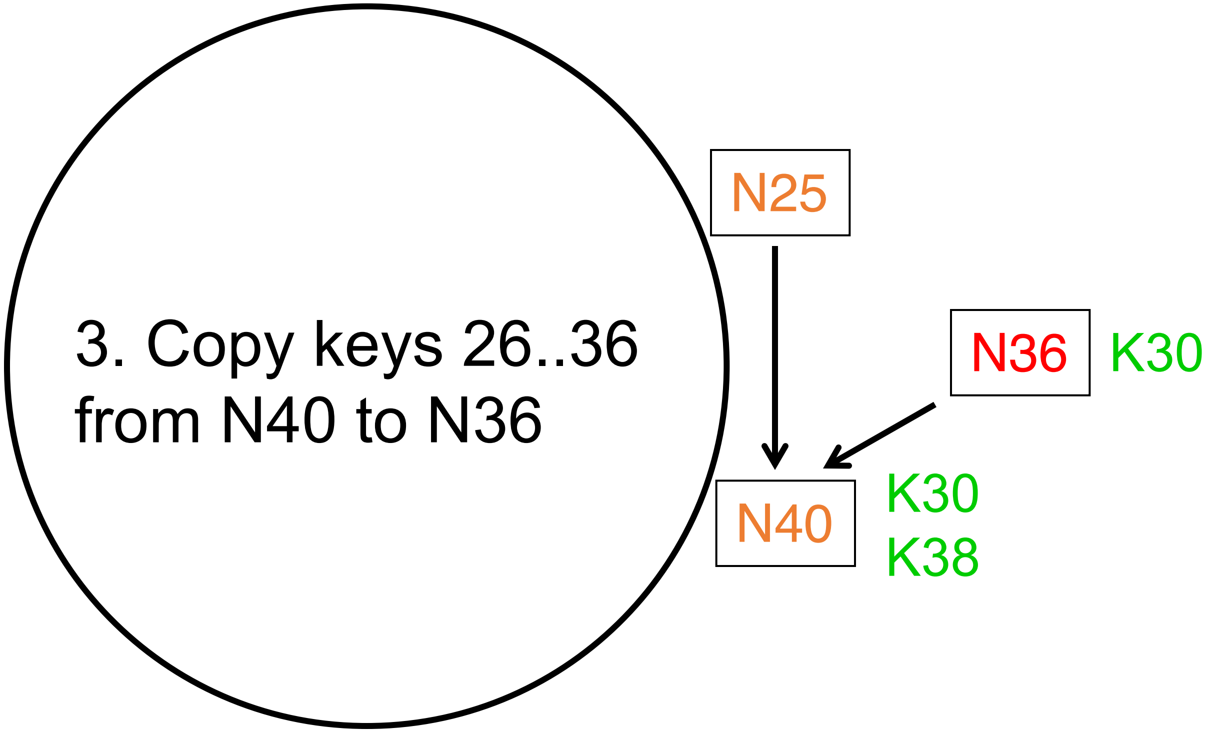
Joining: Linked list insert



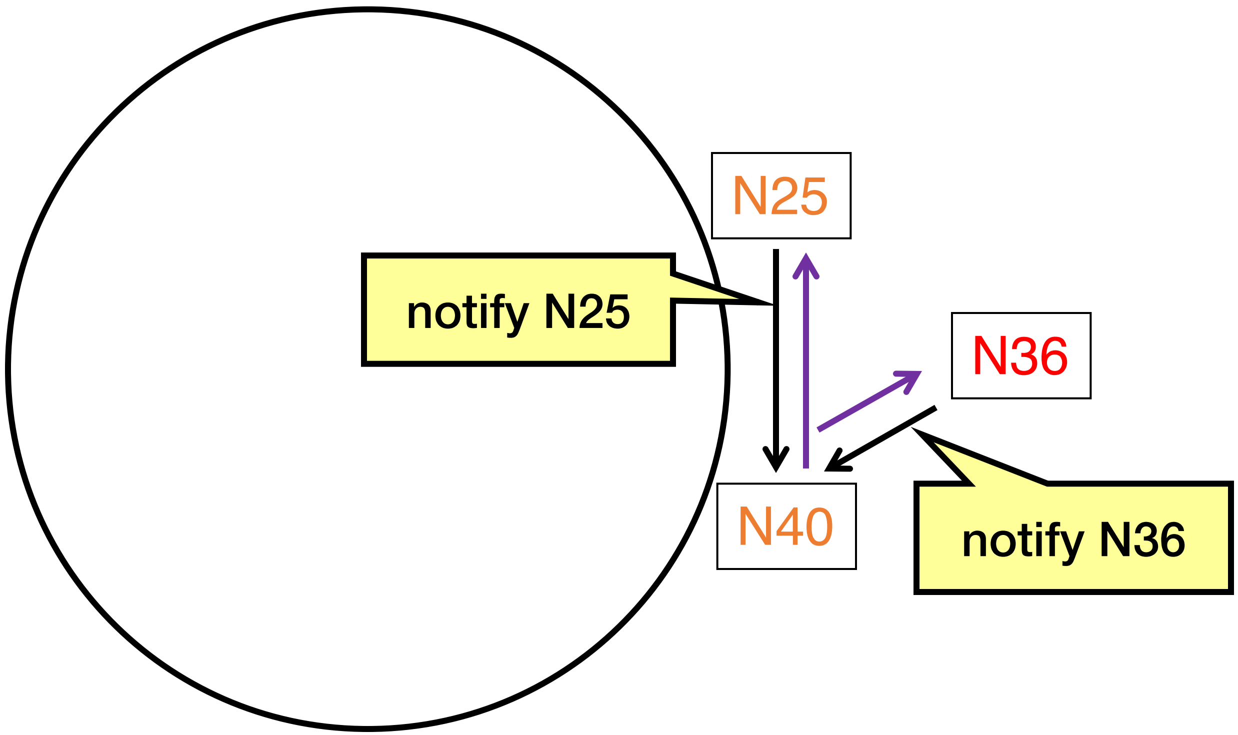
Join (2)



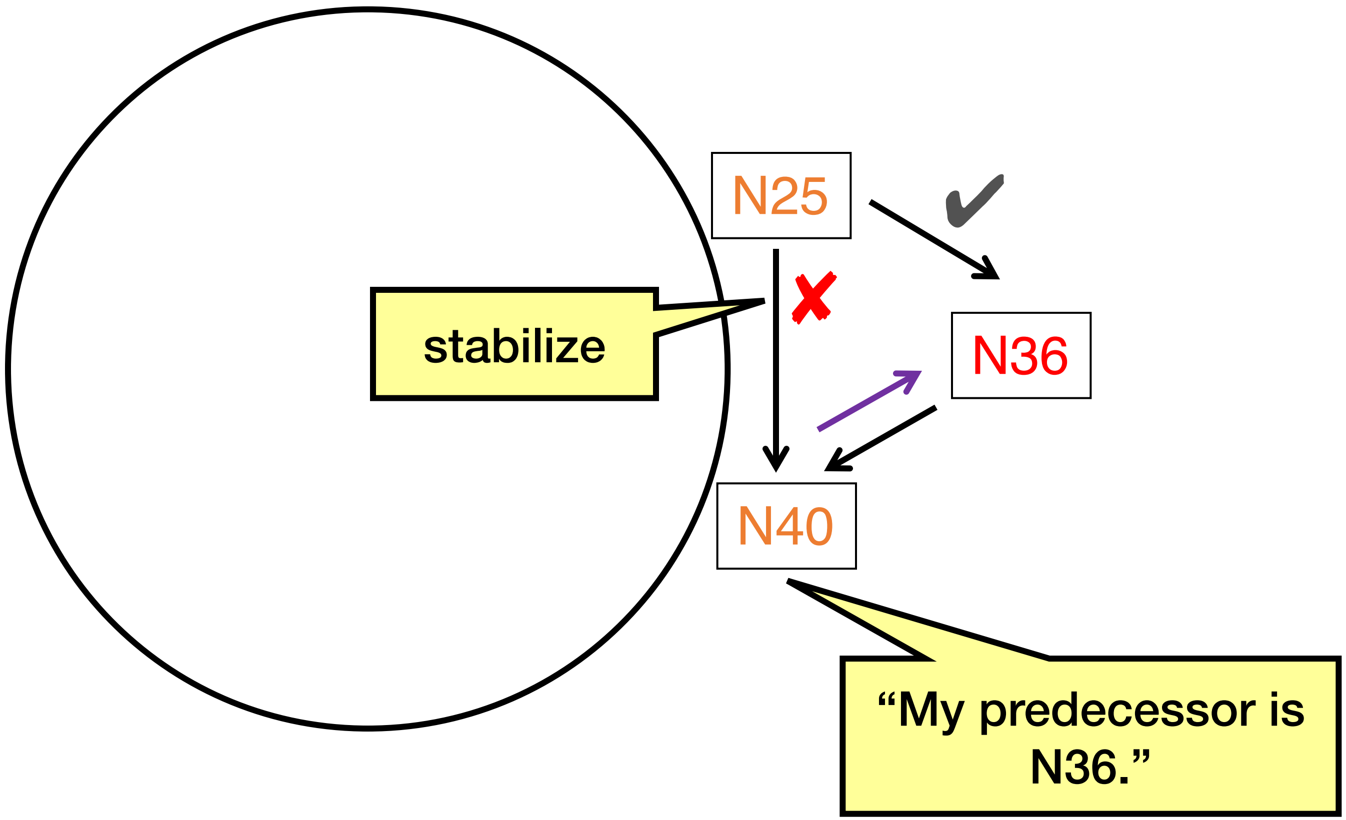
Join (3)



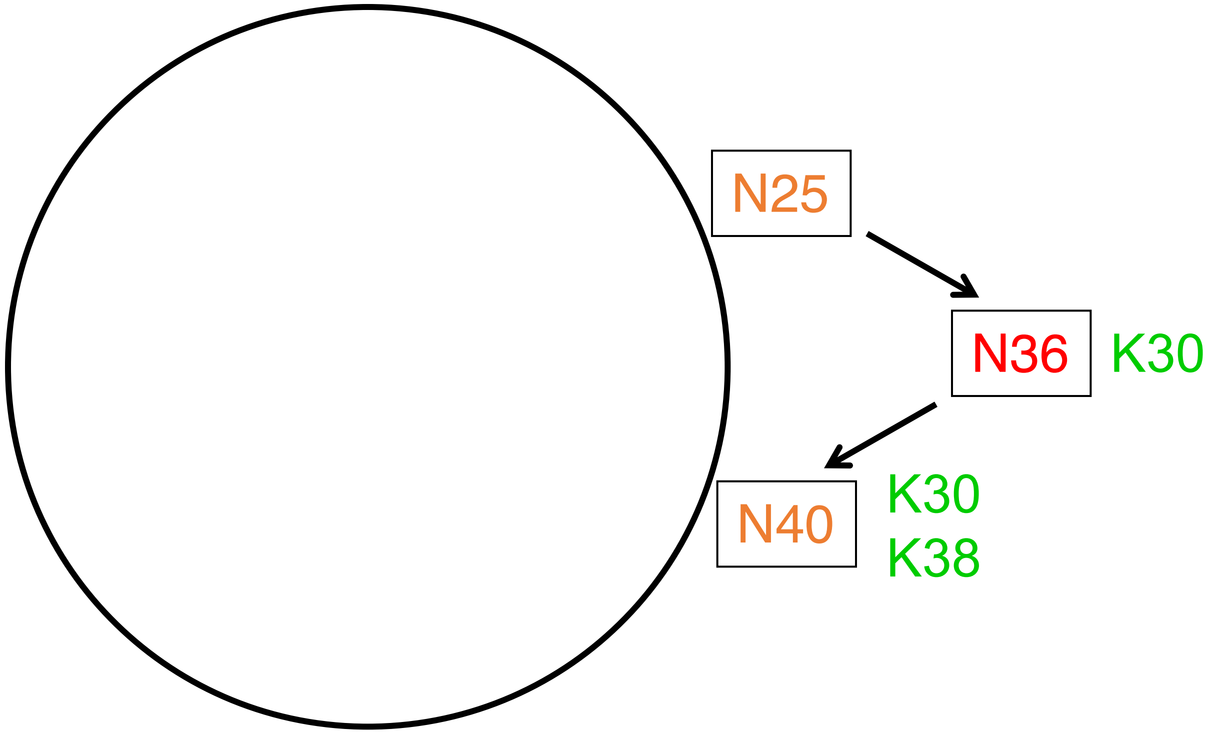
Notify messages maintain predecessors



Stabilize message fixes successor

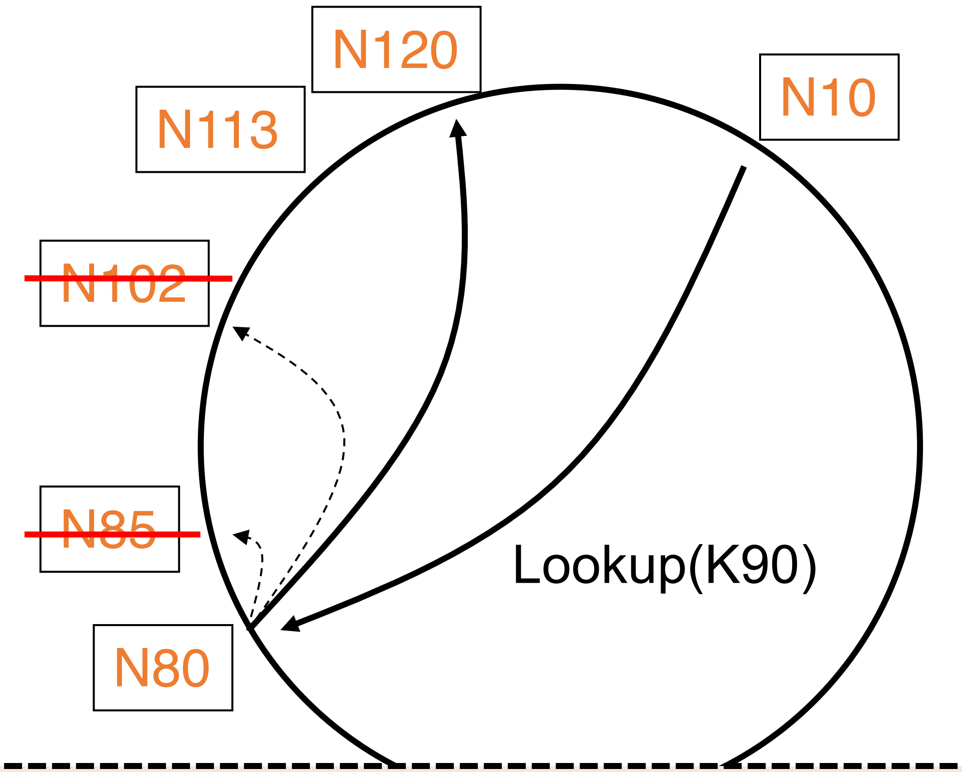


Joining: Summary



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

Failures may cause incorrect lookup



N80 does not know correct successor, so incorrect lookup

Successor lists

- Each node stores a list of its r immediate successors
    - After failure, will know first live successor
    - Correct successors generally produce correct lookups
        - Guarantee is with some probability
    - r is often log*N* too, e.g., 20 for 1 million nodes

Lookup with fault tolerance

```
Lookup(key-id)
  look in local finger table and successor-list
        for highest n: 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 and/or
                       successor list
              return Lookup(key-id)
  else
     return my successor                  // done
```

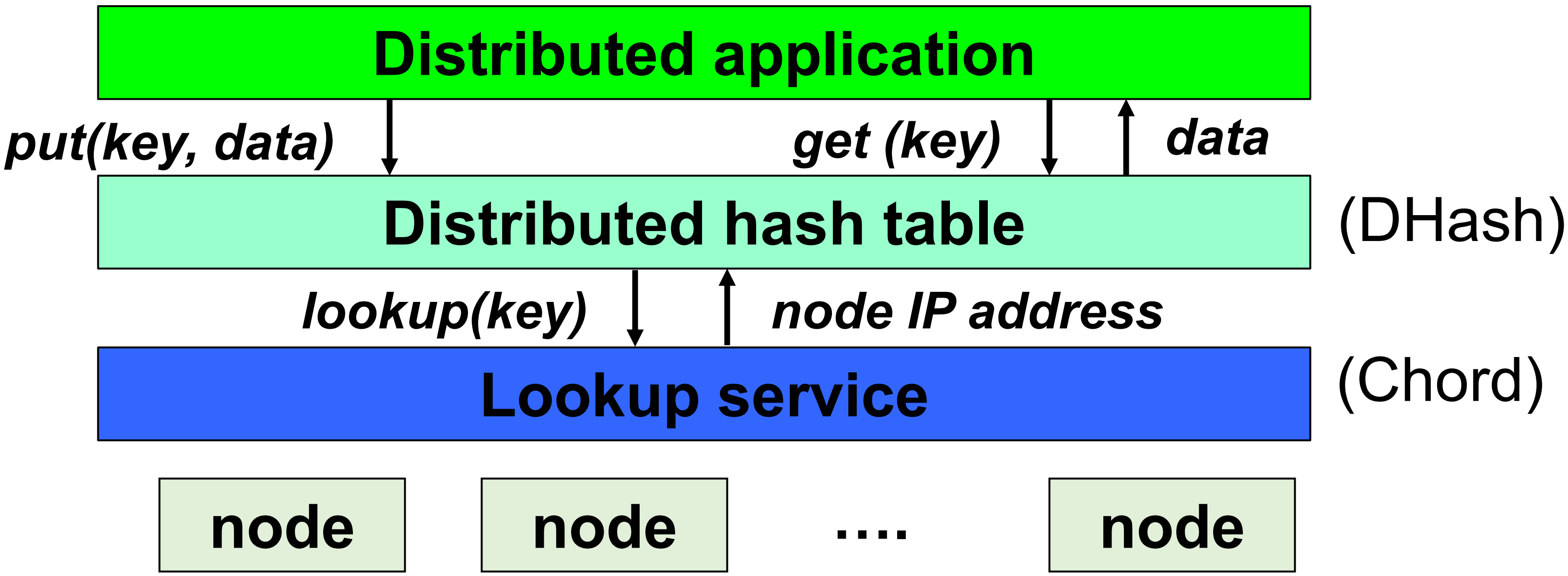
Today

1. Peer-to-Peer Systems

2. Distributed Hash Tables

3. The Chord Lookup Service

Cooperative storage with a DHT

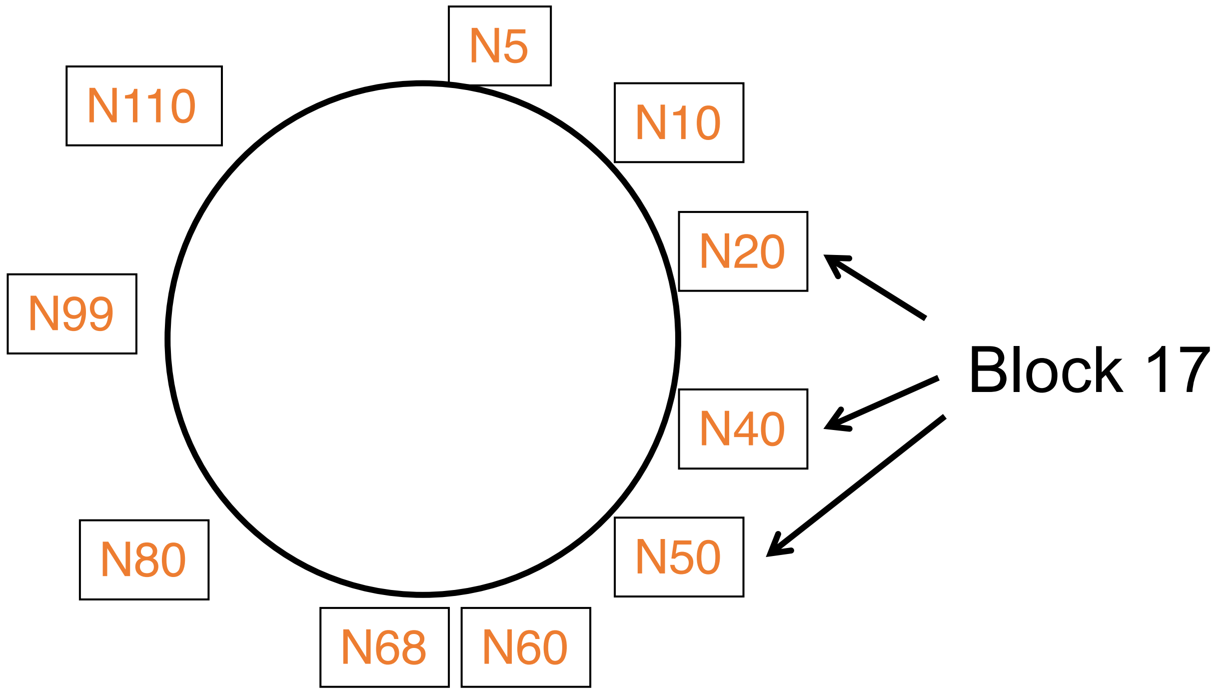


- App may be distributed over many nodes
- DHT distributes data storage over many nodes

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

DHash replicates blocks at *r* successors



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

Today

1. Peer-to-Peer Systems

2. Distributed Hash Tables

3. The Chord Lookup Service

- Concluding thoughts on DHTs, P2P

Why don’t all services use P2P?

1. High latency and limited bandwidth between peers (vs servers 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: used 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