Peer-to-Peer Systems and Distributed Hash Tables

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency
Lecture 9

Marco Canini

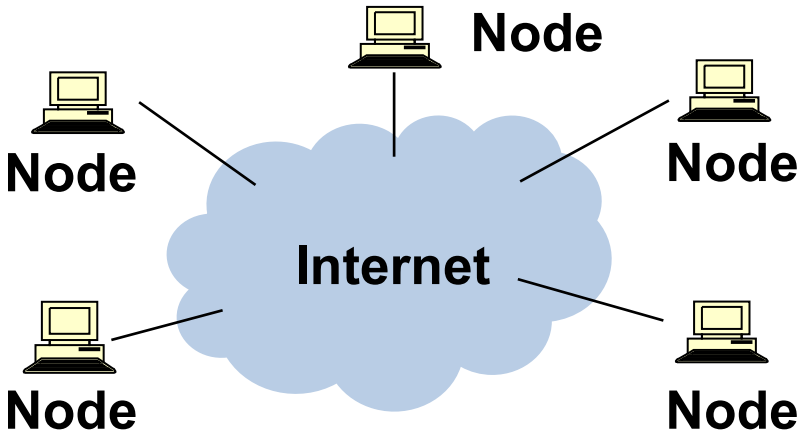
Today

1. **Peer-to-Peer Systems**
   - **Napster, Gnutella, BitTorrent, challenges**

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

P2P adoption

- Successful adoption in **some niche areas**

1. Client-to-client (legal, illegal) **file sharing**
   – Napster (1990s), Gnutella, BitTorrent, etc.

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

3. **Voice/video telephony:** user to user (old Skype)
   – Issues: Privacy and control

Why might P2P be a win?

- **High capacity for services** through resource pooling:
  - Many disks, network connections, CPUs, as peers join
  - Data are divided and duplicated, accessible from multiple peers concurrently

- **No 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**

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

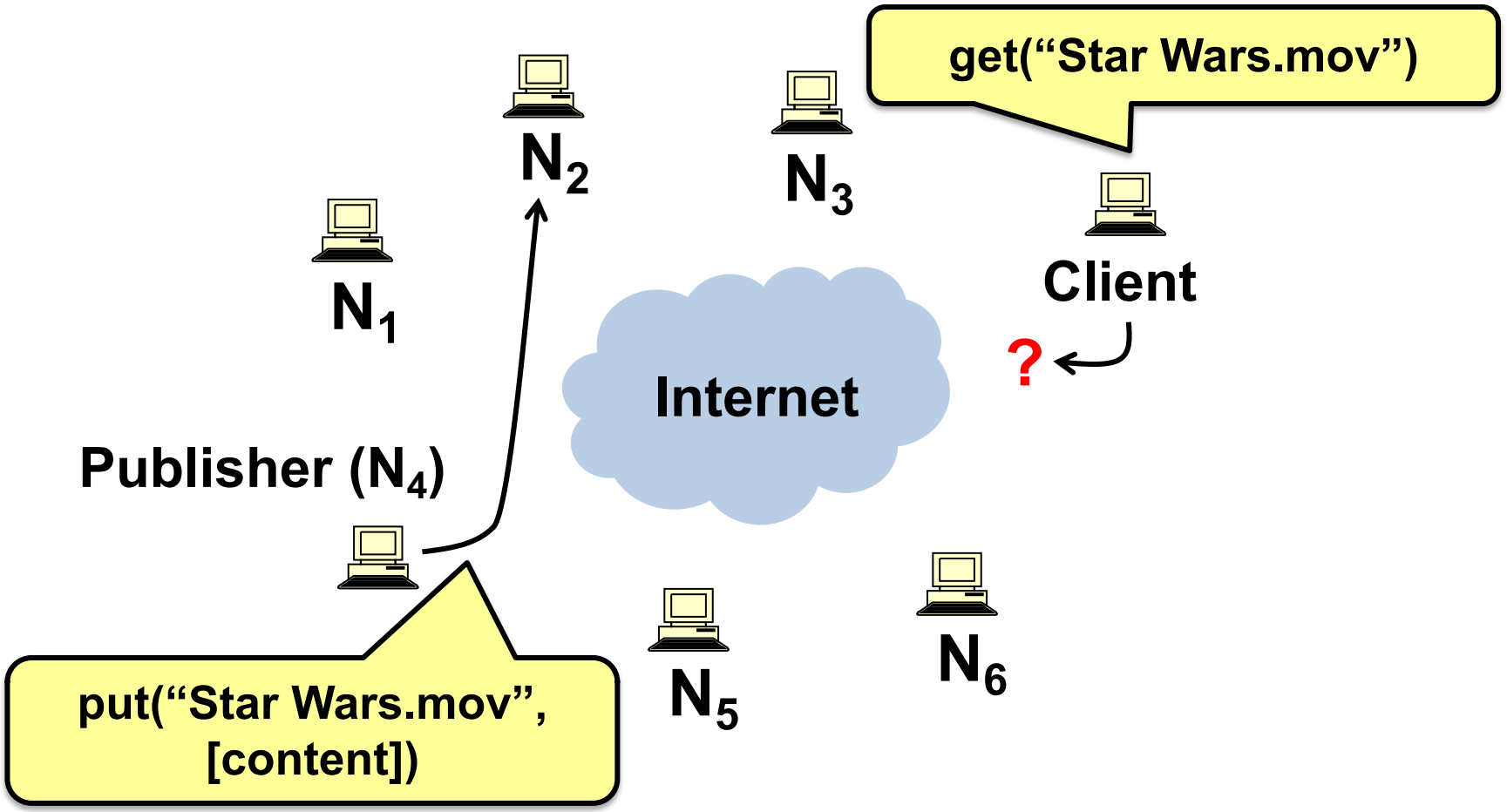
3. User's BT client downloads file from one or more peers

4. User's BT client tells tracker it has a copy now, too

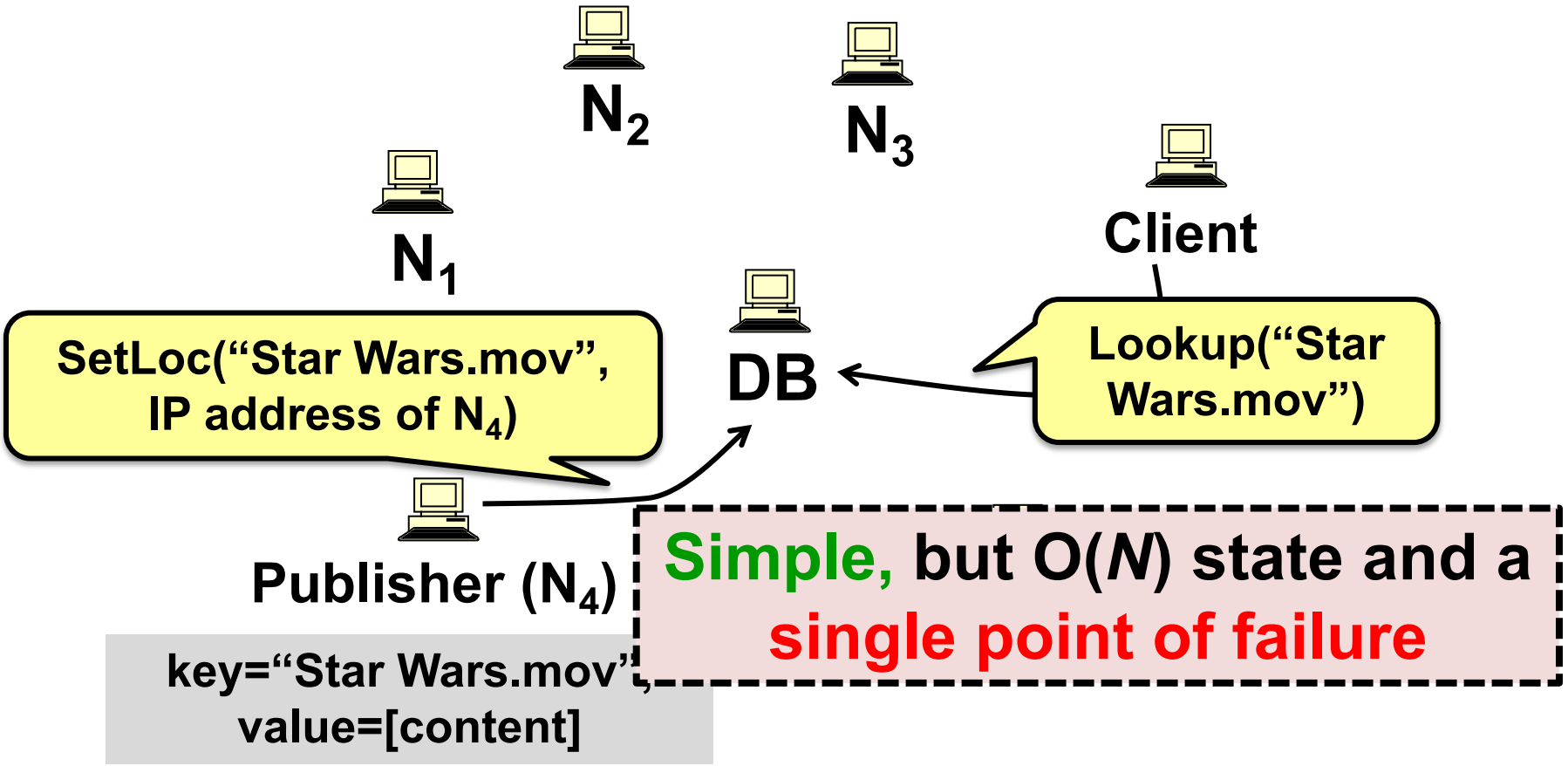
5. User's BT client serves the file to others for a while

Provides huge download bandwidth,
without expensive server or network links

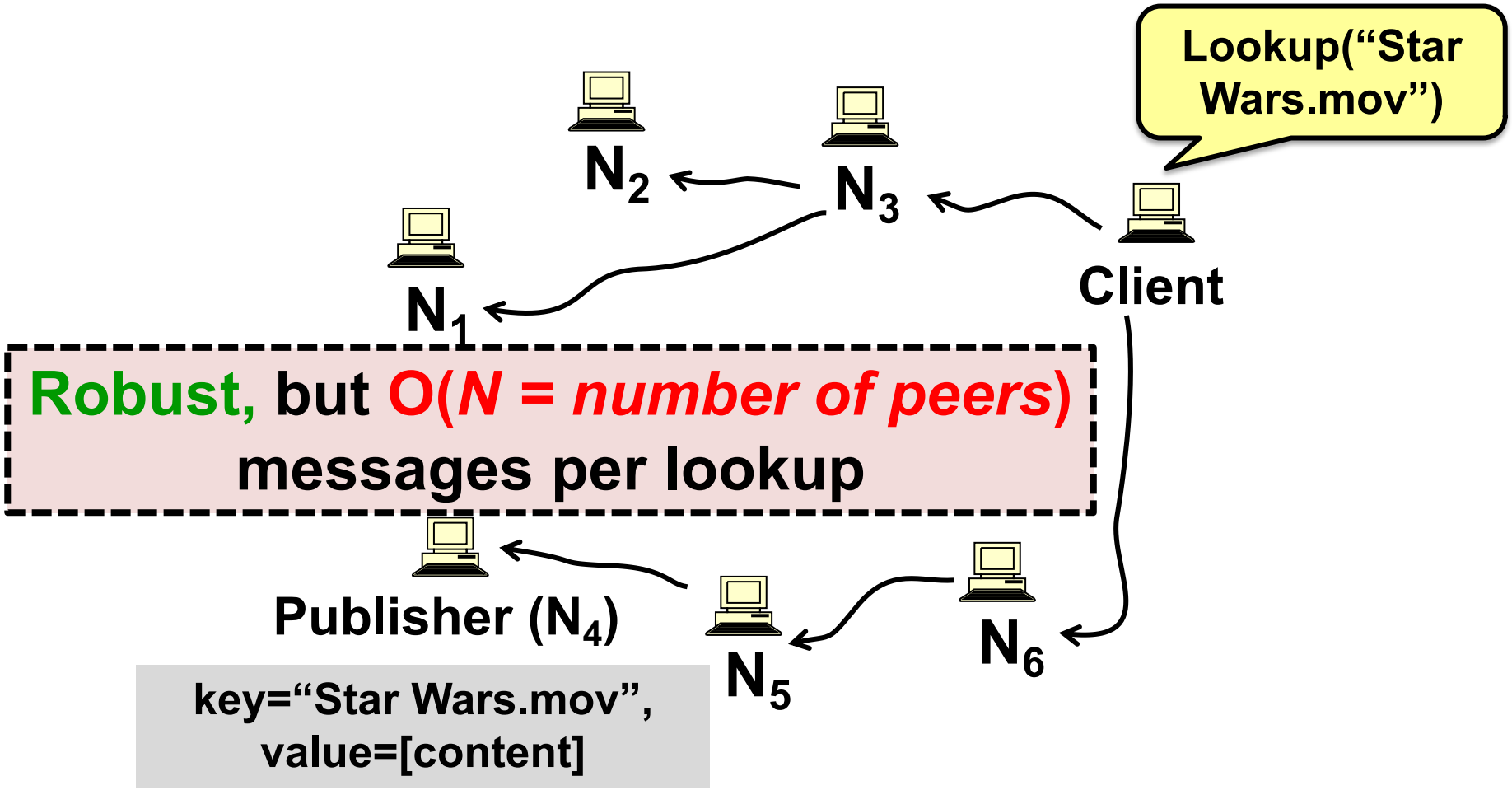
The lookup problem



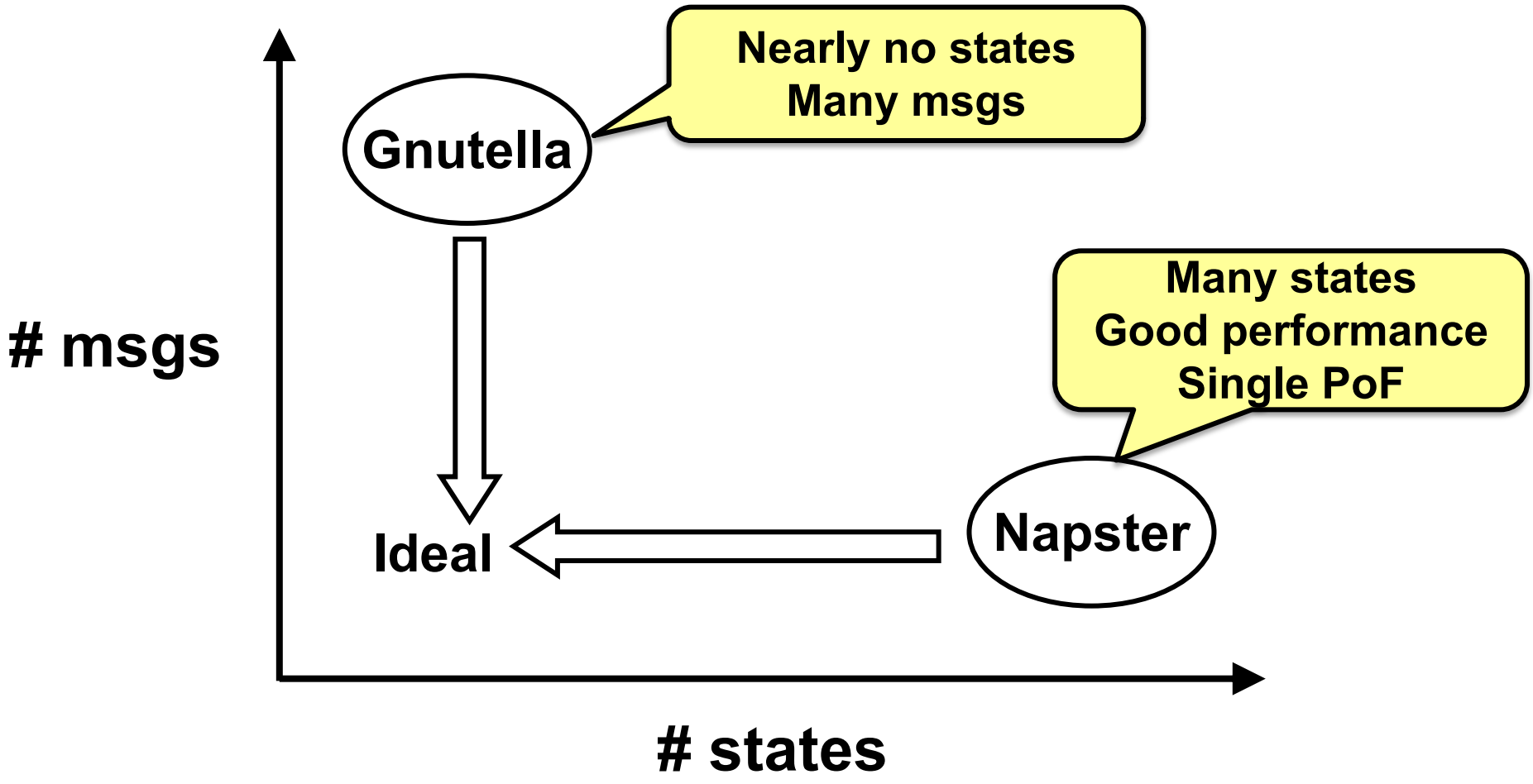
Centralized lookup (Napster)



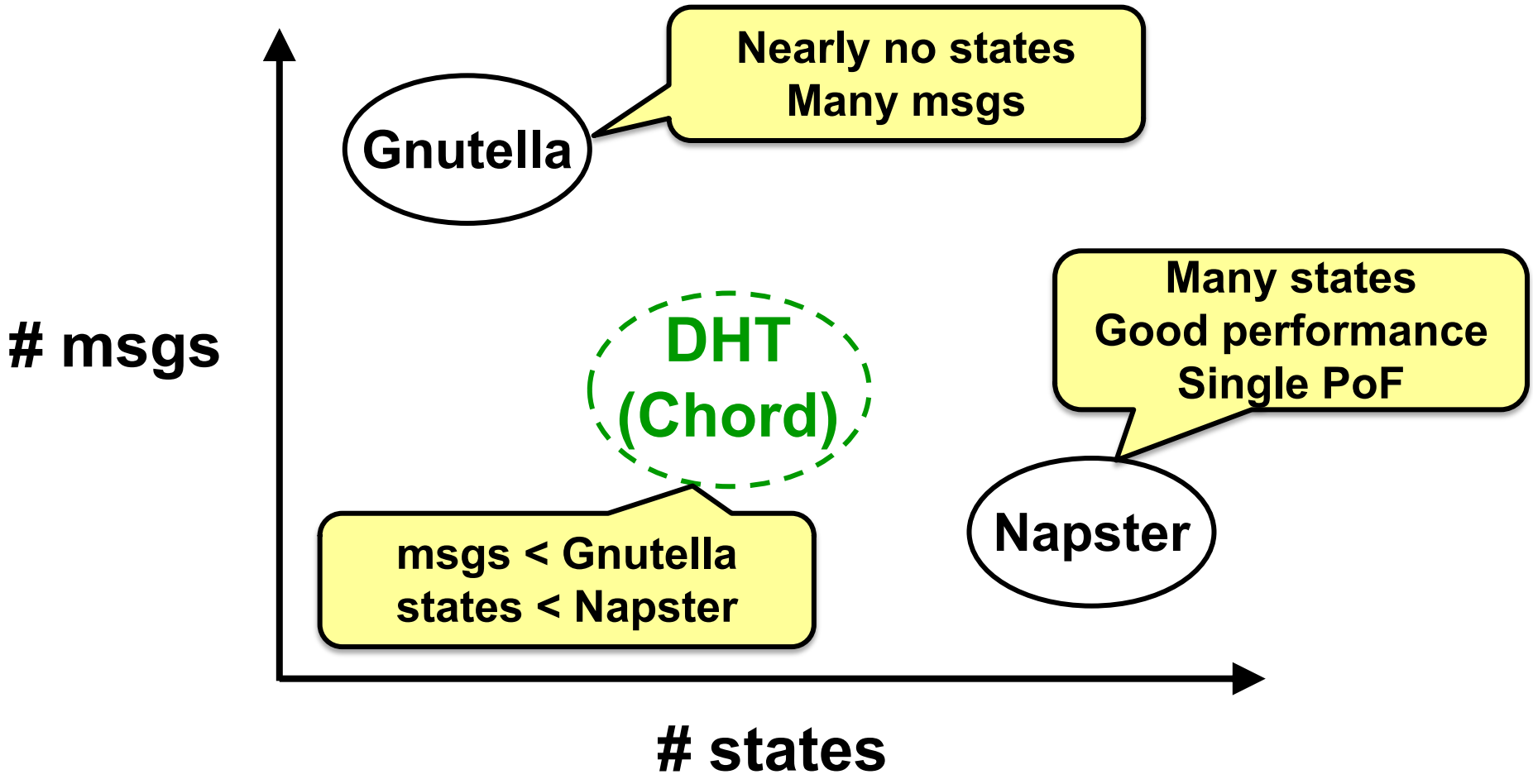
Flooded queries (original Gnutella)



Tradeoffs in distributed systems



Tradeoffs in distributed systems



Today

1. Peer-to-Peer Systems

2. **Distributed Hash Tables**

3. The Chord Lookup Service

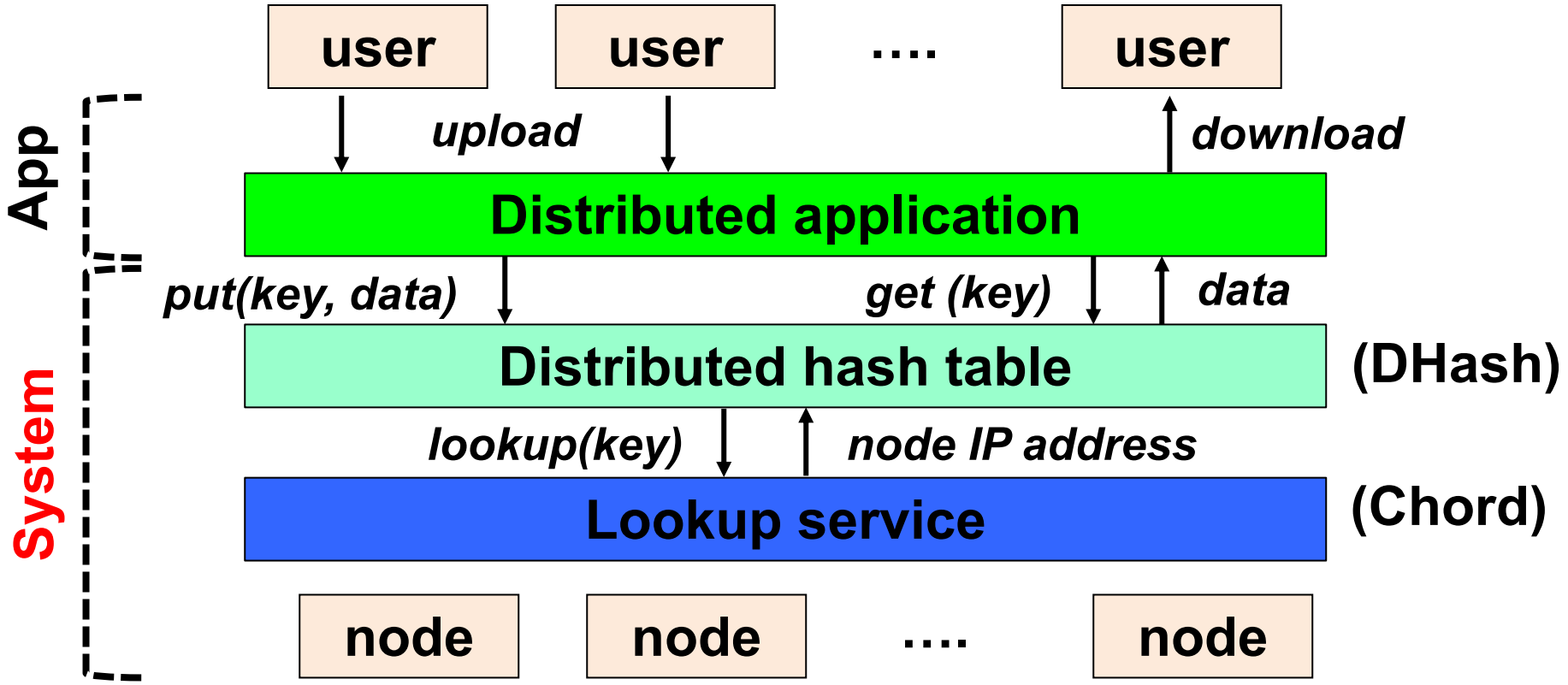
What is a DHT (and why)?

- 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 **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



DHT is expected to be

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

Today

1. Peer-to-Peer Systems

2. Distributed Hash Tables

3. **The Chord Lookup Service**

Chord identifiers

- **Hashed values (int) using the same hash function**
  - **Key identifier** = SHA-1(key)
  - **Node identifier** = SHA-1(IP address)

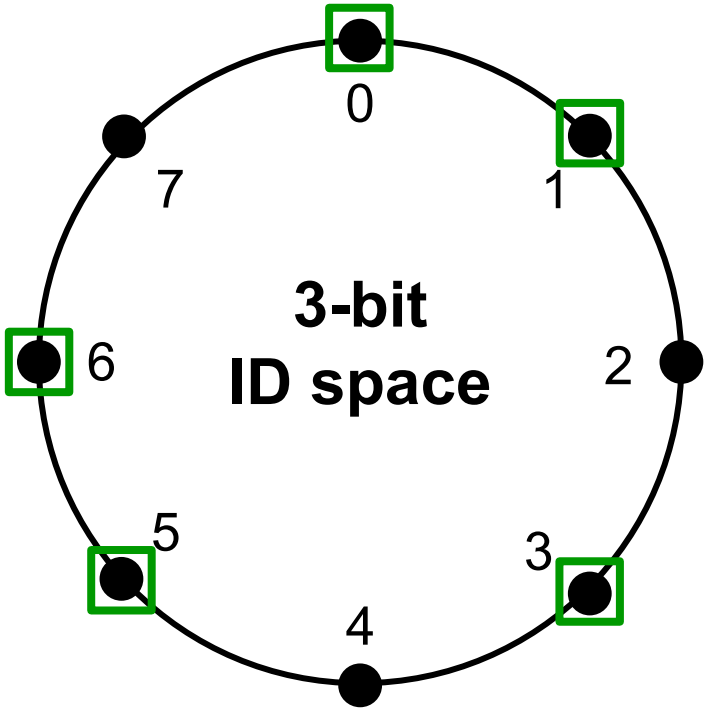
- ***How does Chord partition data?***
  - *i.e.*, map key IDs to node IDs

- **Why hash key and address?**
  - Uniformly distributed in the ID space
  - Hashed key → load balancing
  - Hashed IP address → independent failure

Consistent hashing: data partition

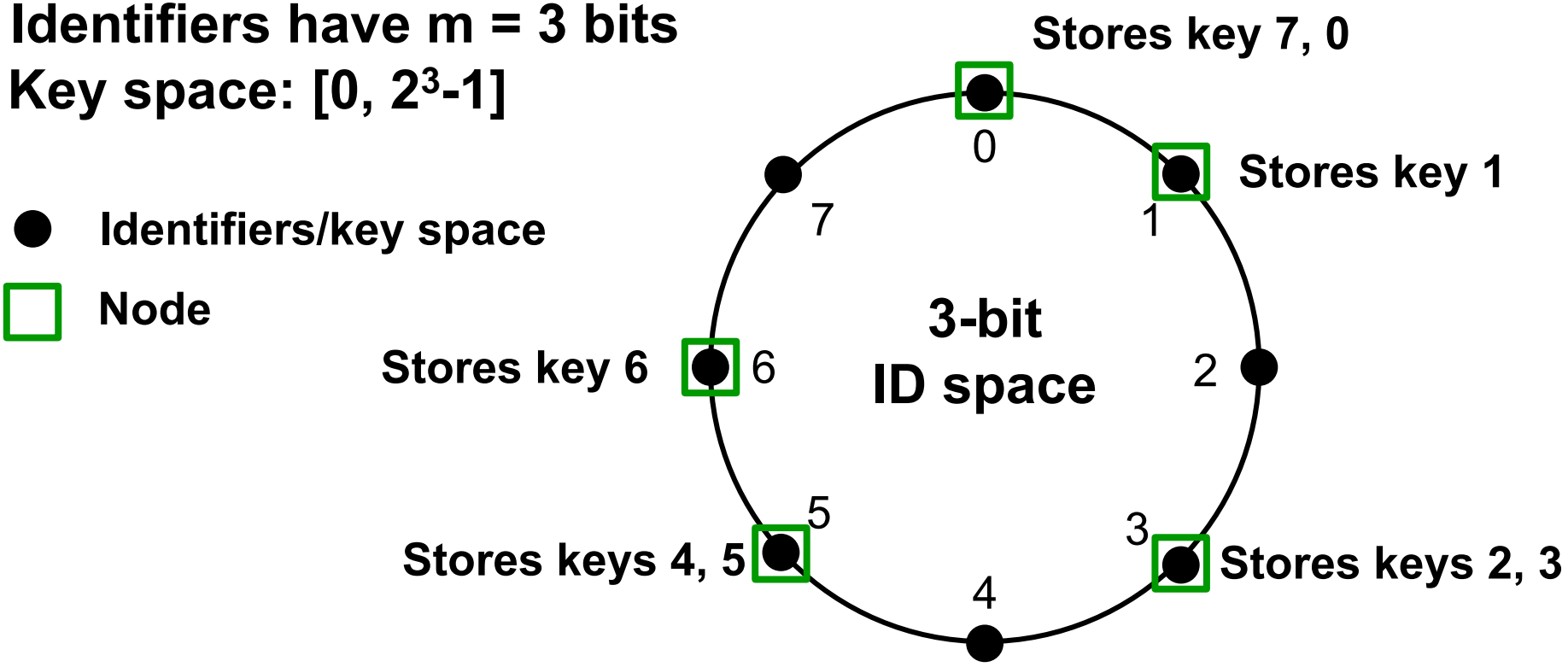
Identifiers have m = 3 bits
Key space: [0, 2^3-1]

- Identifiers/key space
- Node



Consistent hashing: data partition

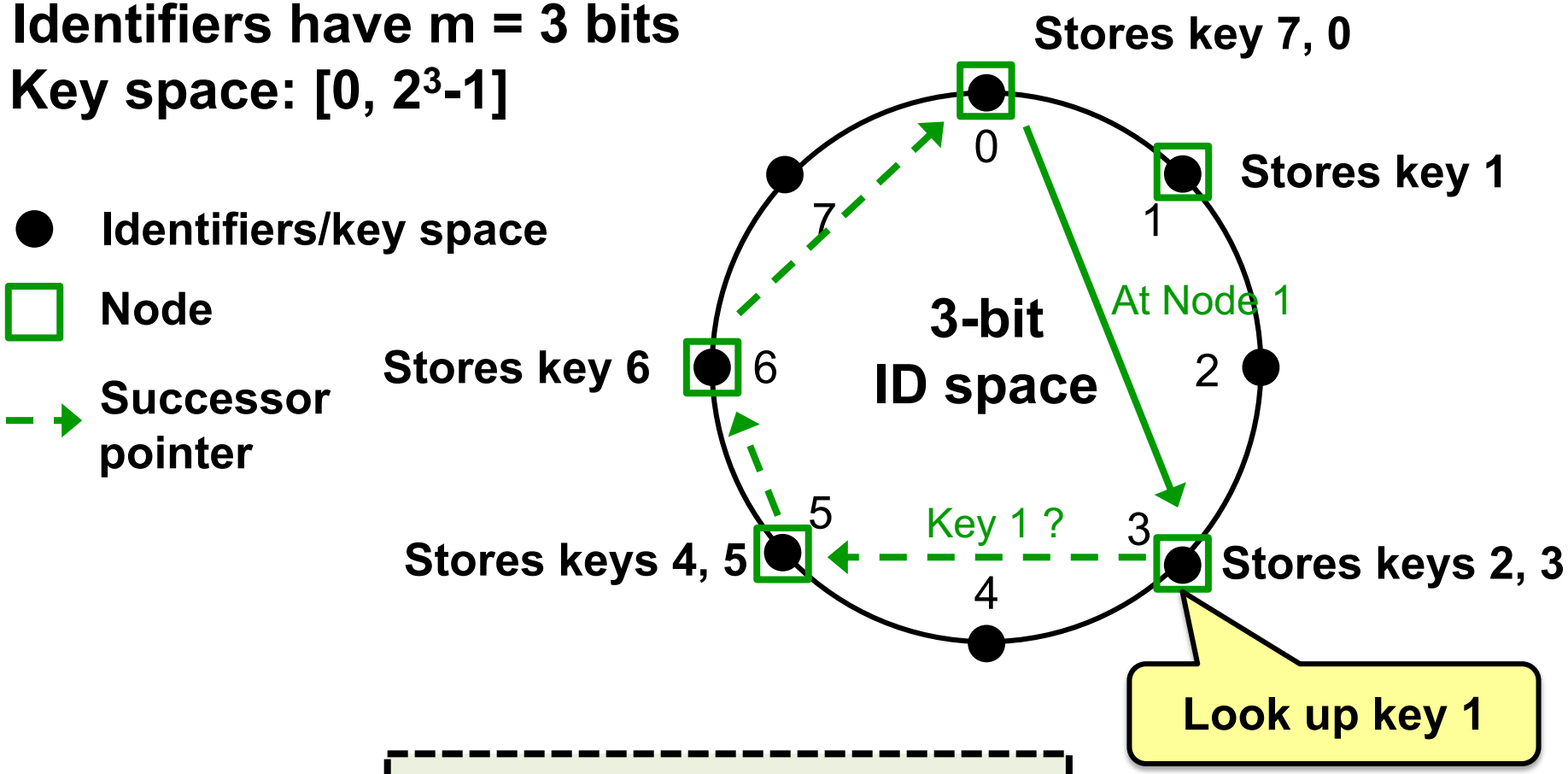
Identifiers have m = 3 bits
Key space: [0, 2^3-1]



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

Consistent hashing: basic lookup

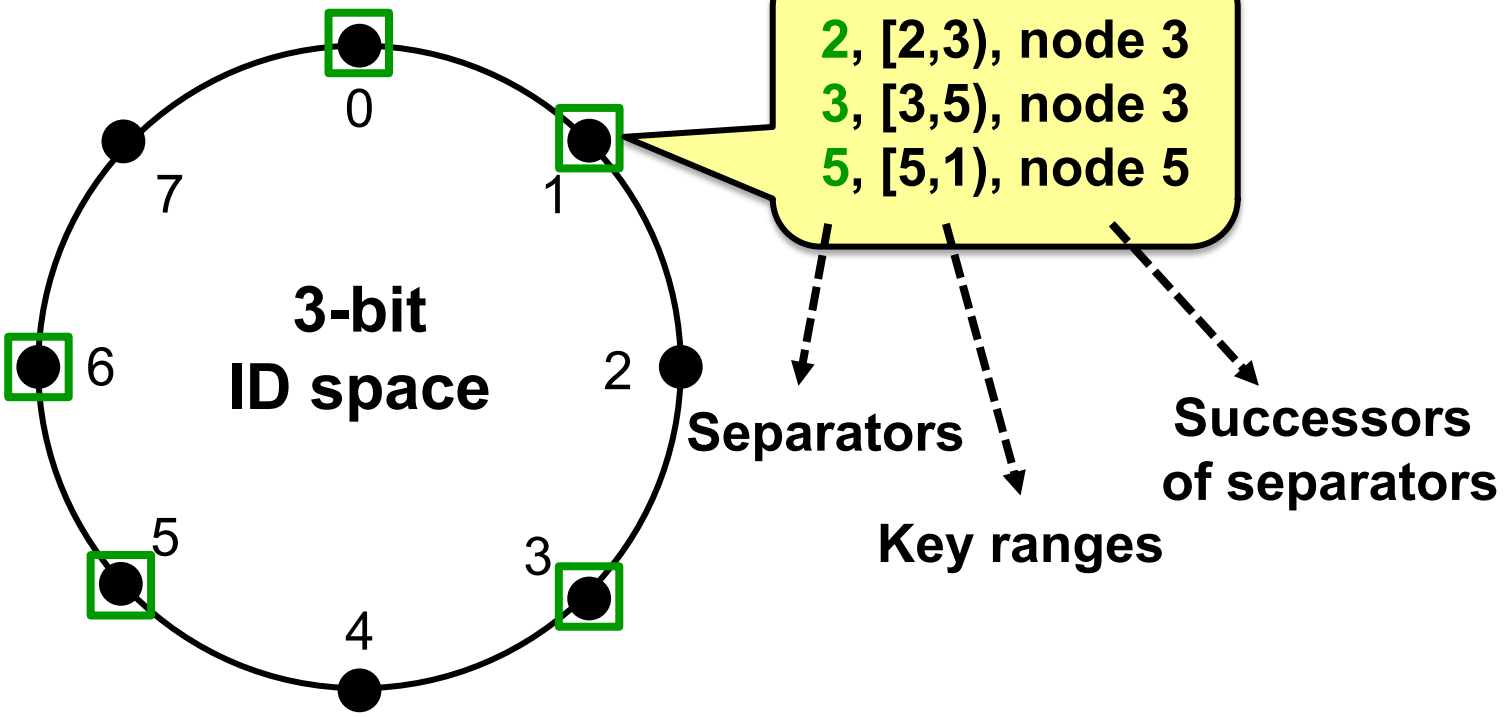
Identifiers have m = 3 bits
Key space: [0, 2^3-1]



O(N) messages and hops!

Chord: finger tables

Identifiers have m = 3 bits



Each node keeps m states

Key space → m ranges via

$(N+2^{k-1}) \bmod 2^m$, $1<=k<=m$

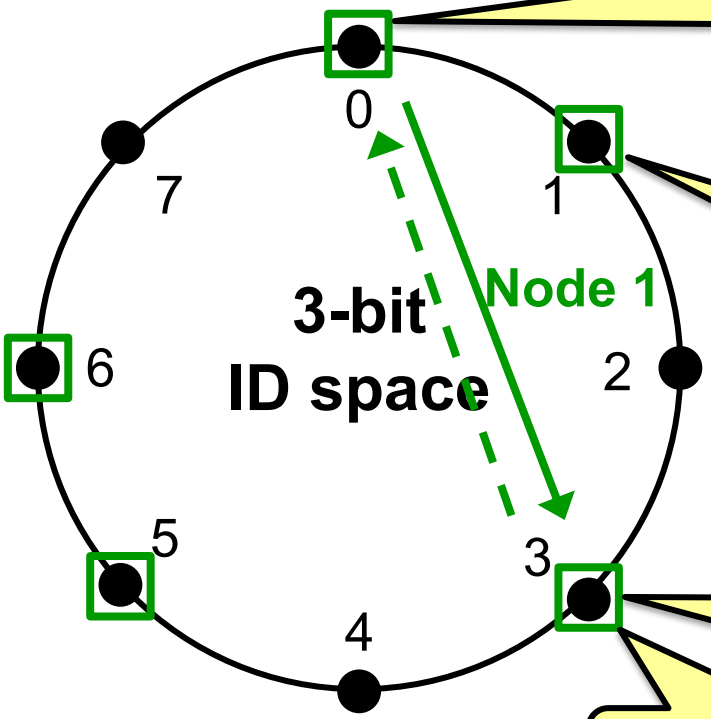
k=1 → range size 1

k=2 → range size 2

k=3 → range size 4

Chord: finger tables

Identifiers have m = 3 bits



1, [1,2), node 1
2, [2,4), node 3
4, [4,0), node 5

2, [2,3), node 3
3, [3,5), node 3
5, [5,1), node 5

4, [4,5), node 5
5, [5,7), node 5
7, [7,3), node 0

Look up key 1

Each node keeps m states

Key space → m ranges via

$(N+2^{k-1}) \bmod 2^m$, $1<=k<=m$

O(log N) messages and hops!

Implication of finger tables

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

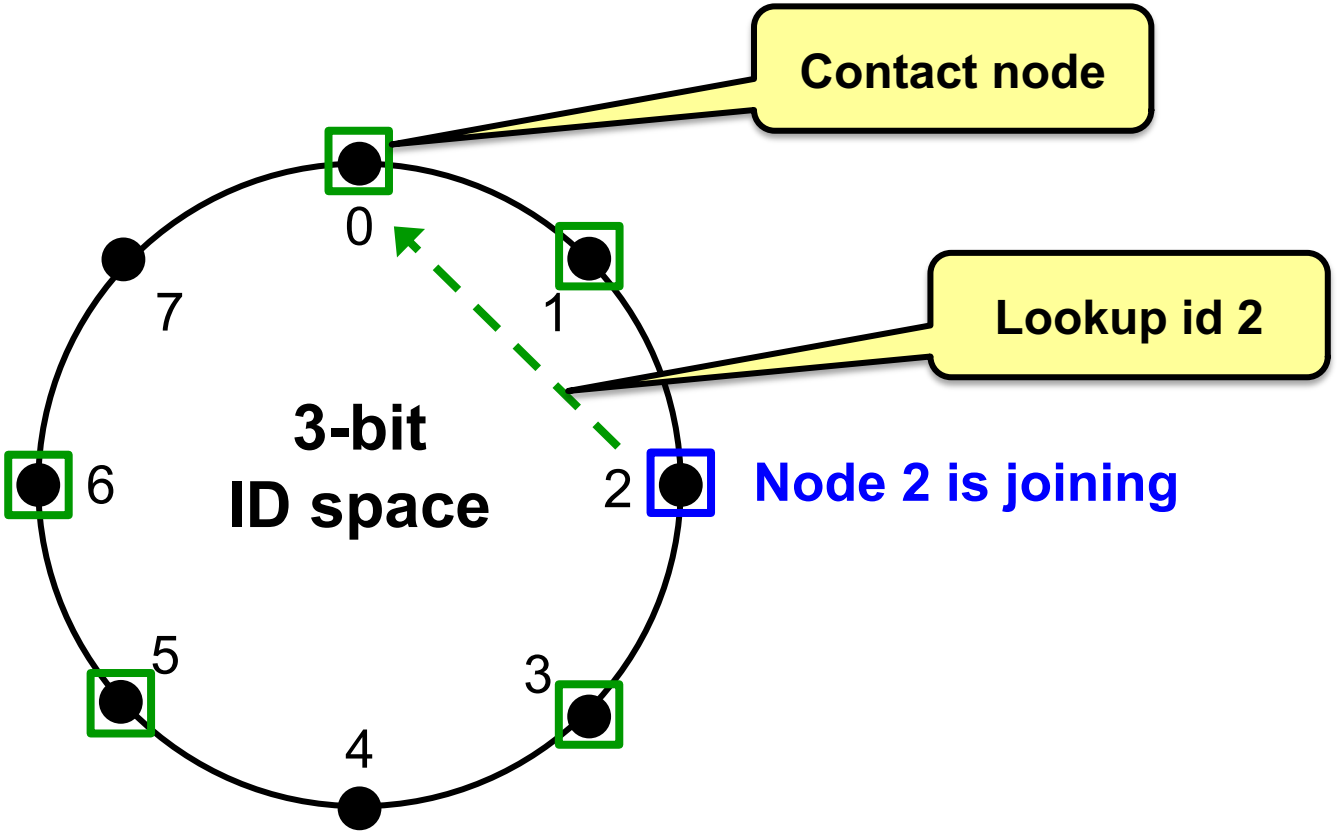
- This is **better** than simply arranging the nodes in a single tree
  - Every node acts as a root
    - So there's **no root hotspot**
    - **No single point** of failure
    - But a **lot more state** in total

Chord lookup algorithm properties

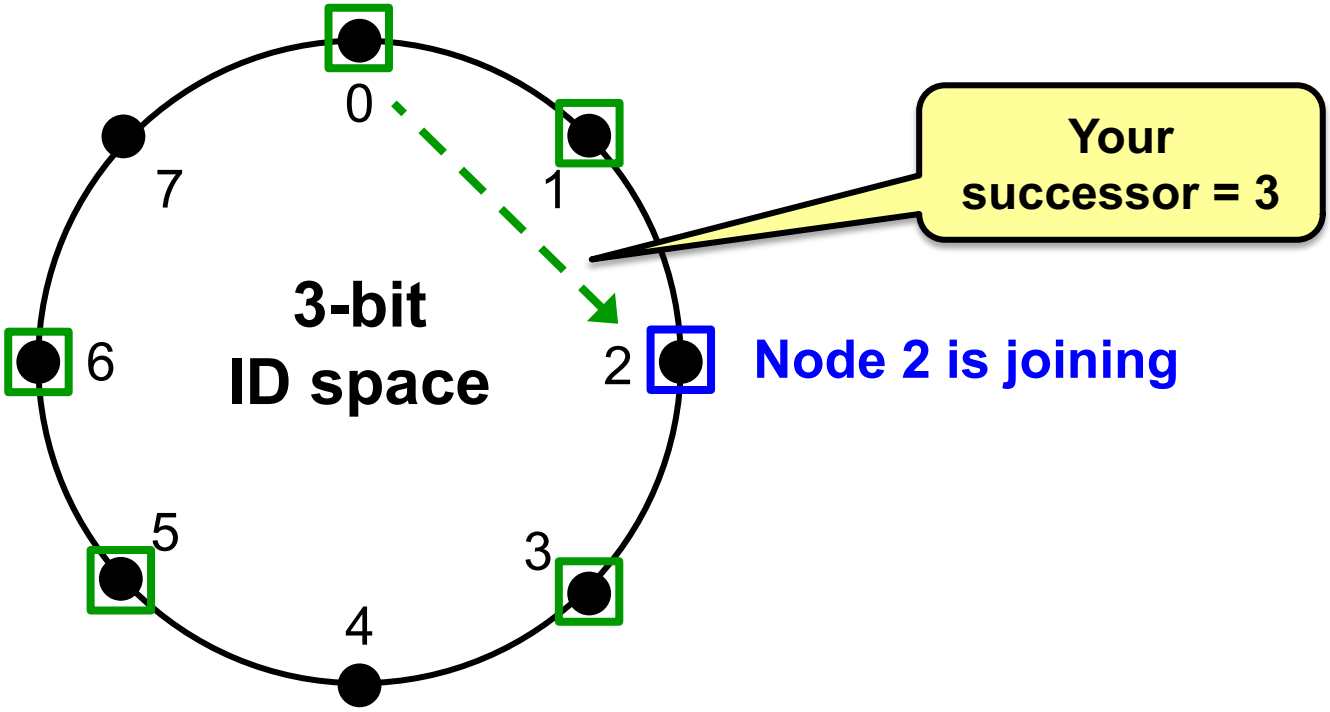
Interface: lookup(key) → IP address

- **Efficient:** O(log *N*) messages per lookup
  - *N* is the total number of nodes
- **Scalable:** O(log *N*) state per node
- **Robust:** survives massive failures

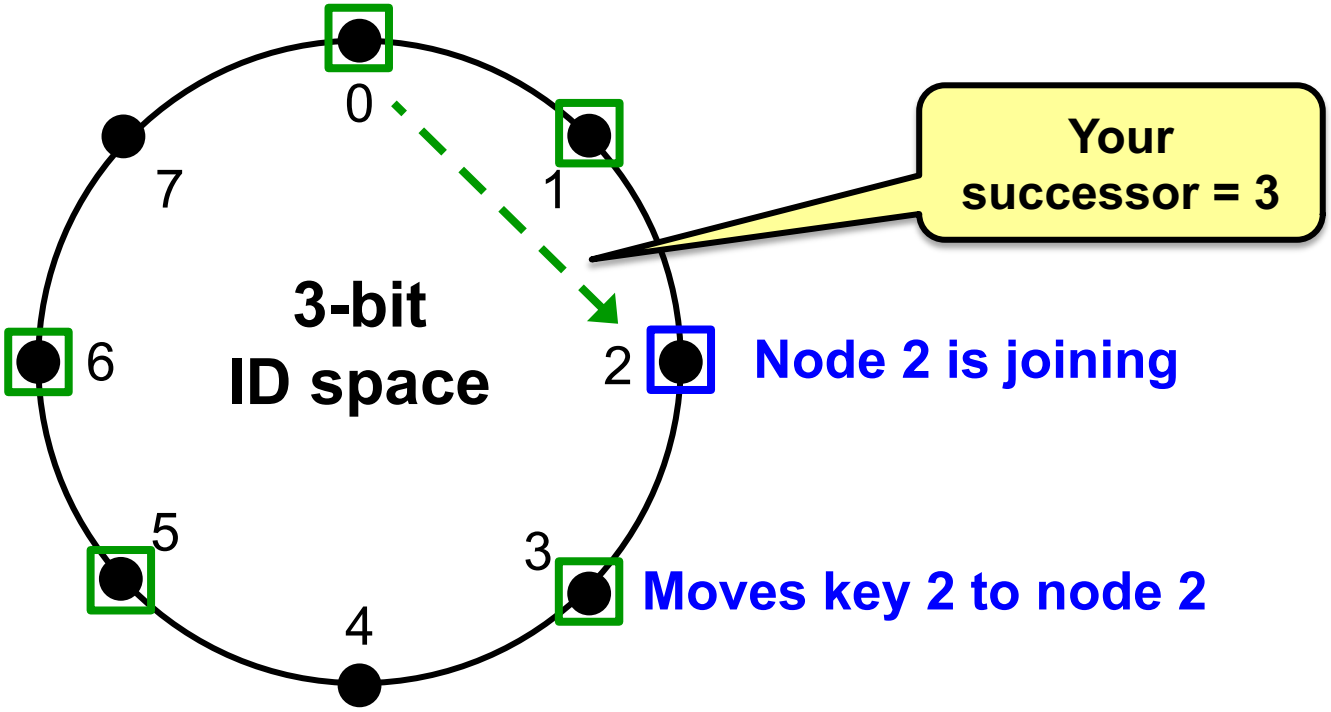
Chord – node joining



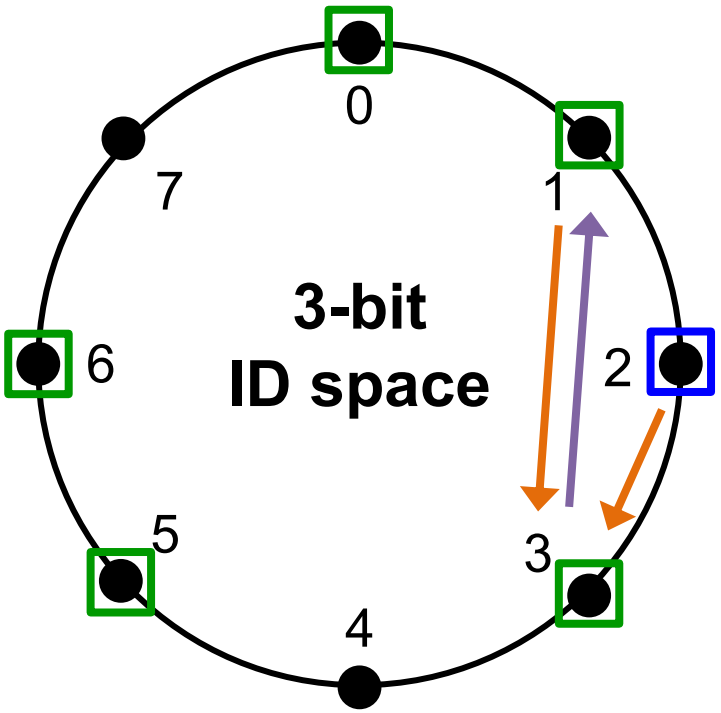
Chord – node joining



Chord – node joining



Chord – node joining



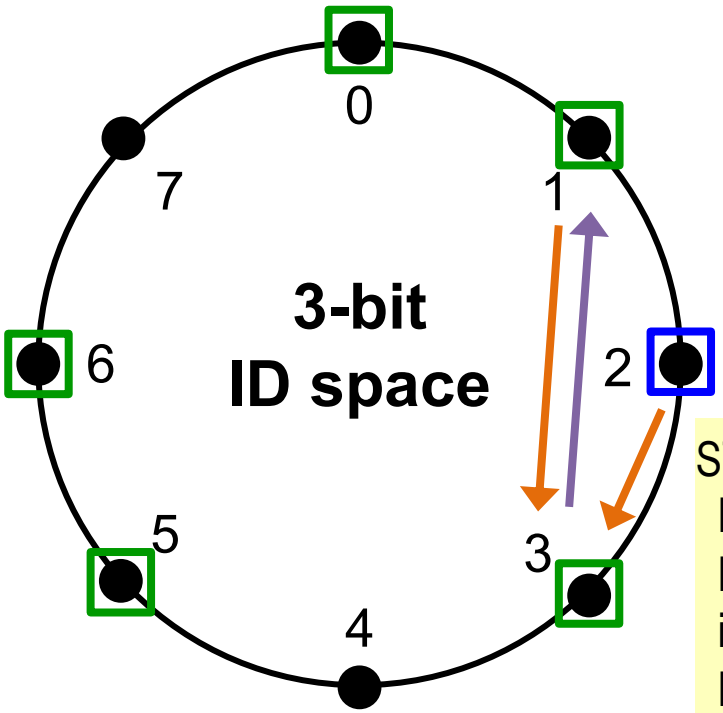
Points to successor

Points to predecessor

Node 2 is joining

Periodic stabilization messages from each node to its successor maintain node positions

Chord – node joining



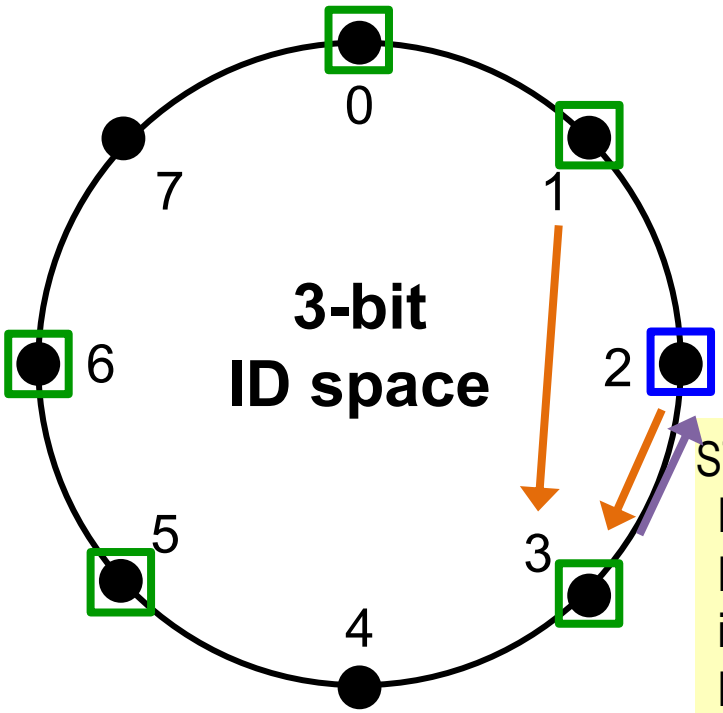
Points to successor

Points to predecessor

Node 2 is joining

STABILIZE() [N.successor = M]
N → M: "What is your predecessor?"
M → N: "X is my predecessor"
if X between (N, M): N.successor = X
N → N.successor: NOTIFY()
NOTIFY()
N → N.successor: "I think you are my successor"
M: upon receiving NOTIFY from N:
if (N between (M.predecessor, M)
M.predecessor = N

Chord – node joining



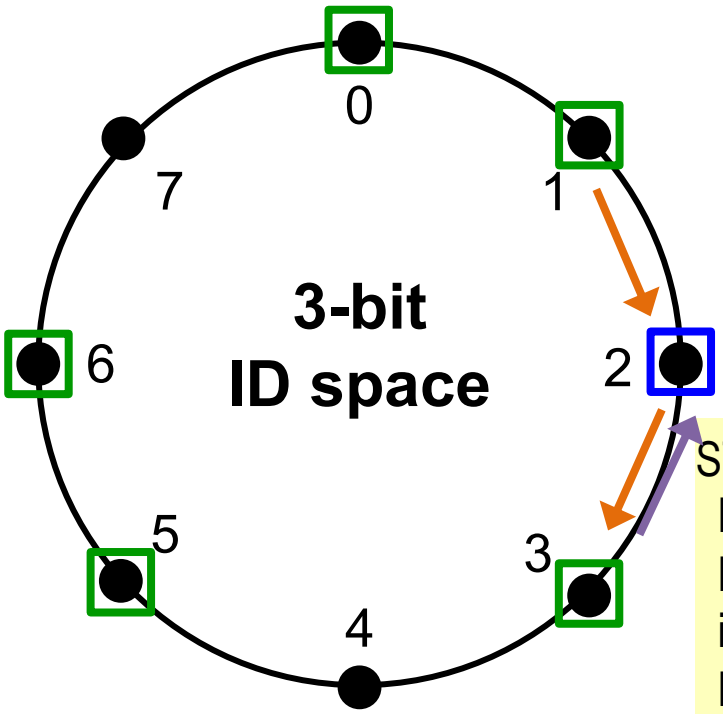
Points to successor

Points to predecessor

Node 2 is joining

STABILIZE() [N.successor = M]
N → M: "What is your predecessor?"
M → N: "X is my predecessor"
if X between (N, M): N.successor = X
N → N.successor: NOTIFY()
NOTIFY()
N → N.successor: "I think you are my successor"
M: upon receiving NOTIFY from N:
if (N between (M.predecessor, M)
M.predecessor = N

Chord – node joining



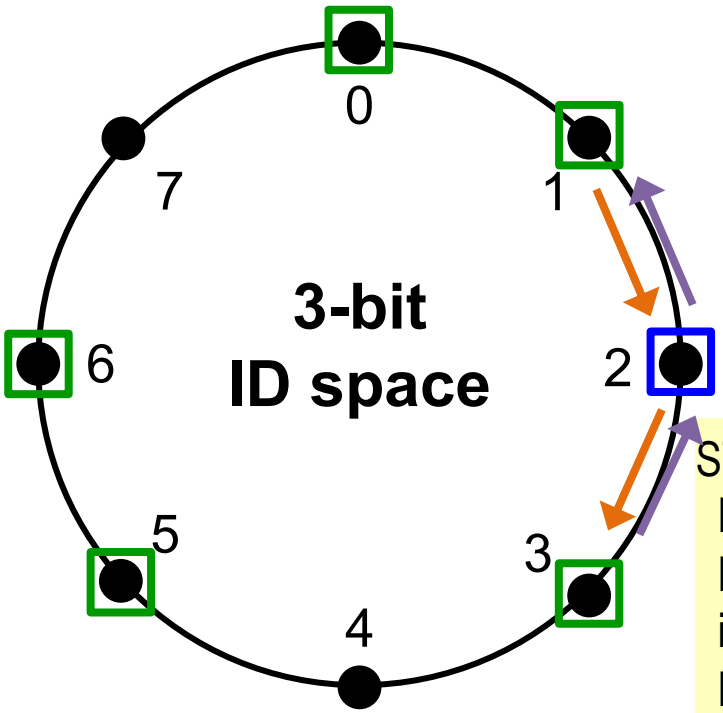
Points to successor

Points to predecessor

Node 2 is joining

```
STABILIZE() [N.successor = M]
  N → M: "What is your predecessor?"
  M → N: "X is my predecessor"
  if X between (N, M): N.successor = X
  N → N.successor: NOTIFY()
NOTIFY()
  N → N.successor: "I think you are my successor"
M: upon receiving NOTIFY from N:
  if (N between (M.predecessor, M)
    M.predecessor = N
```

Chord – node joining



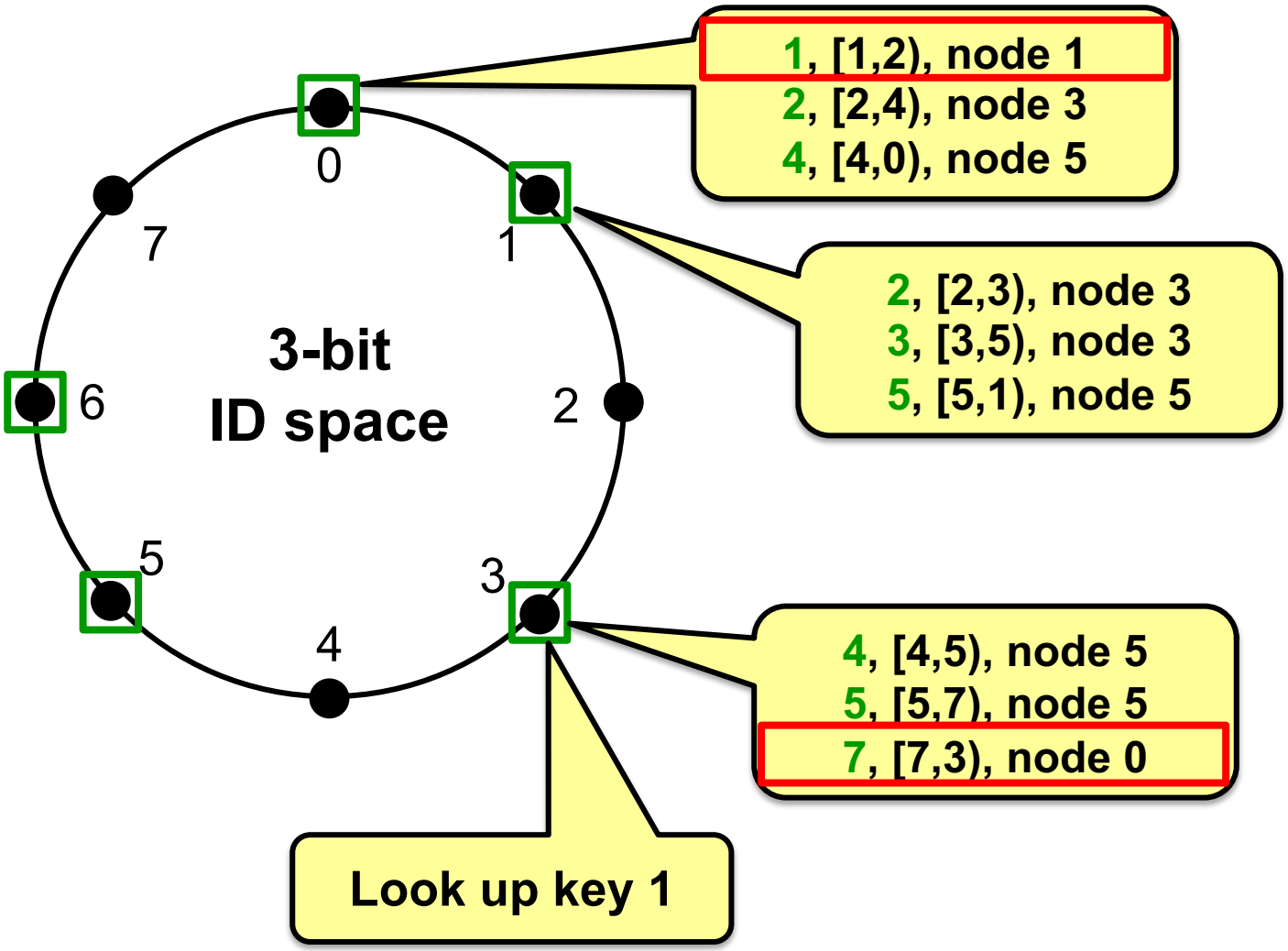
Points to successor

Points to predecessor

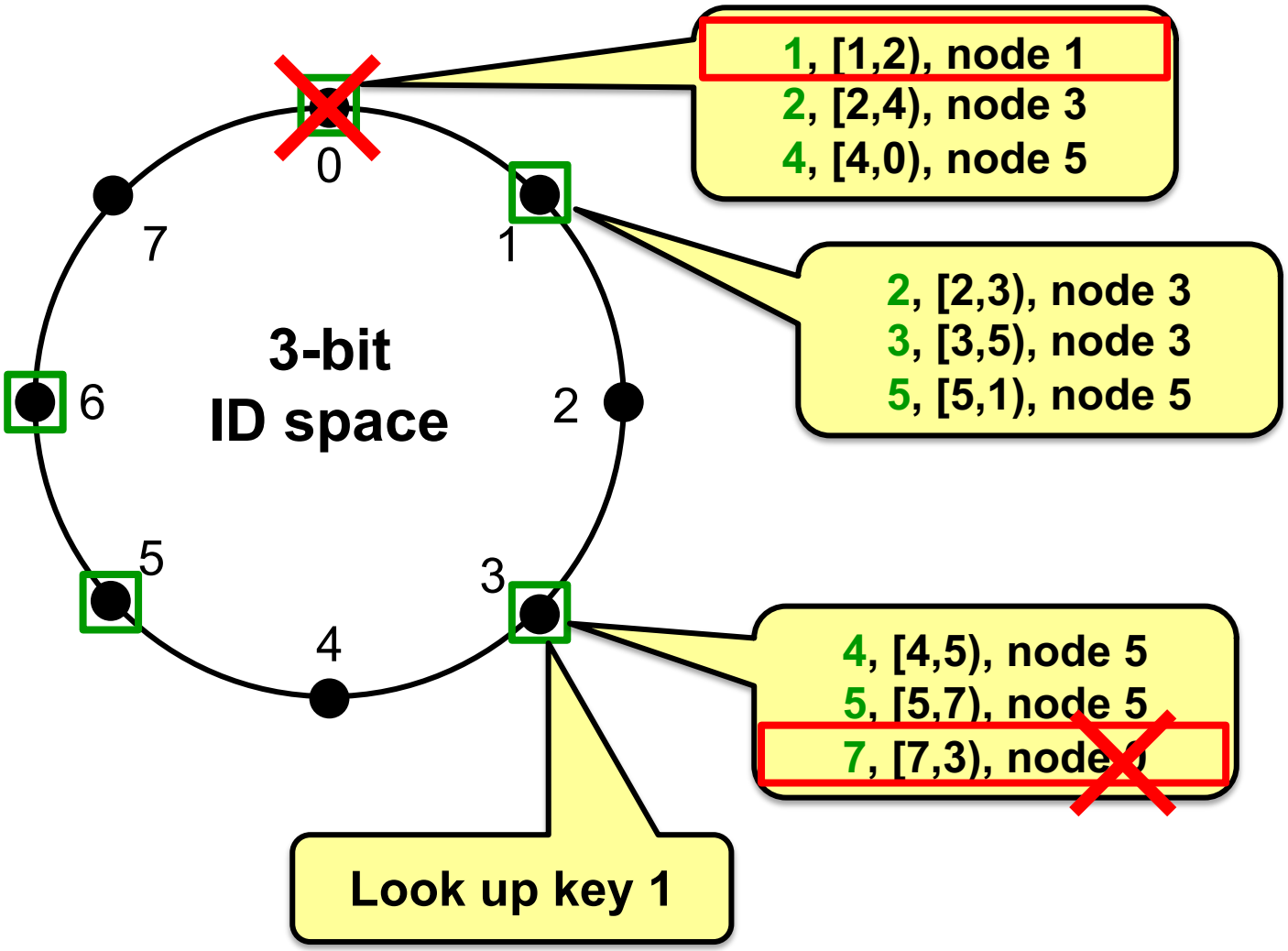
Node 2 is joining

STABILIZE() [N.successor = M]
N → M: "What is your predecessor?"
M → N: "X is my predecessor"
if X between (N, M): N.successor = X
N → N.successor: NOTIFY()
NOTIFY()
N → N.successor: "I think you are my successor"
M: upon receiving NOTIFY from N:
if (N between (M.predecessor, M)
M.predecessor = N

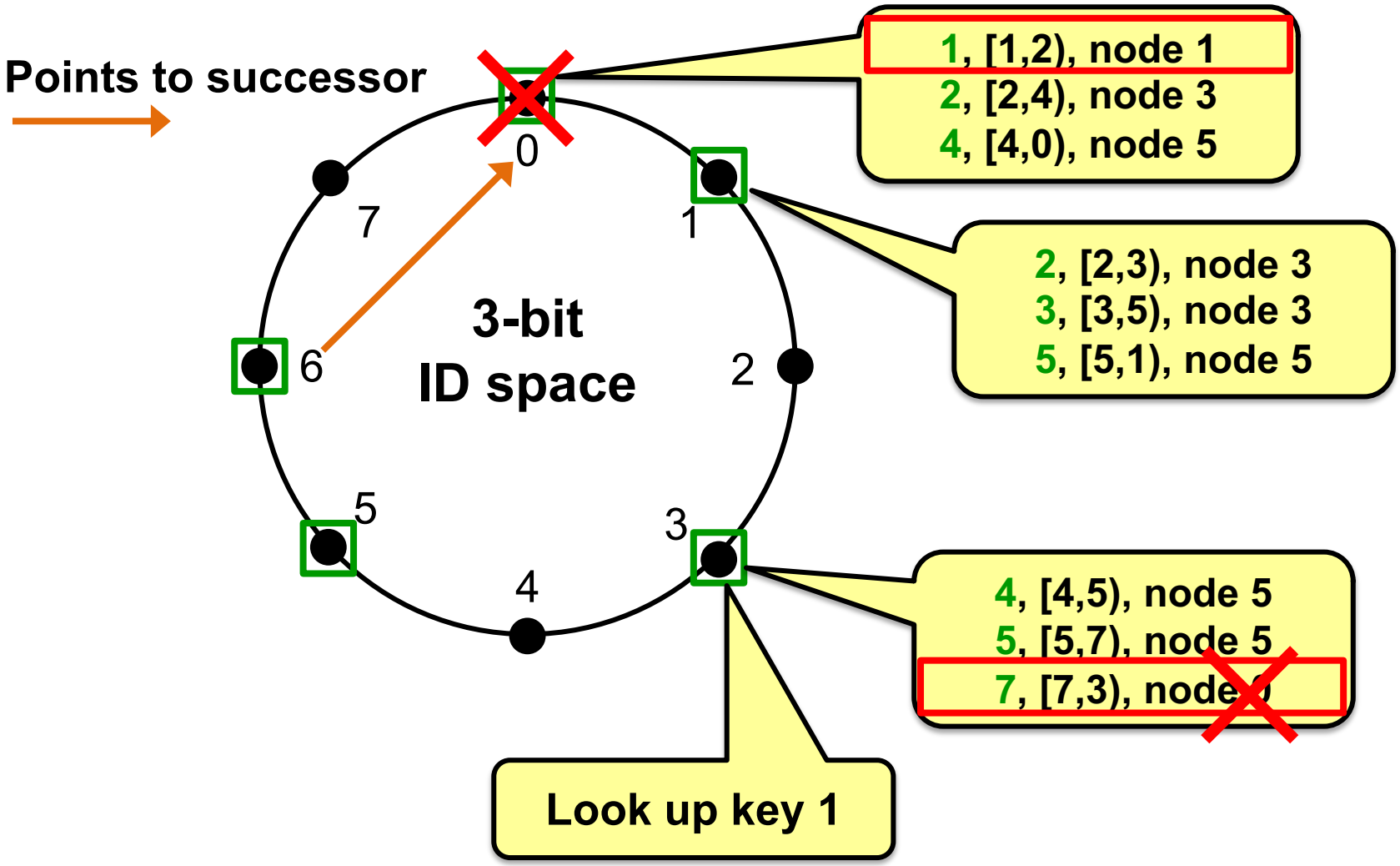
Chord – failures and successor list



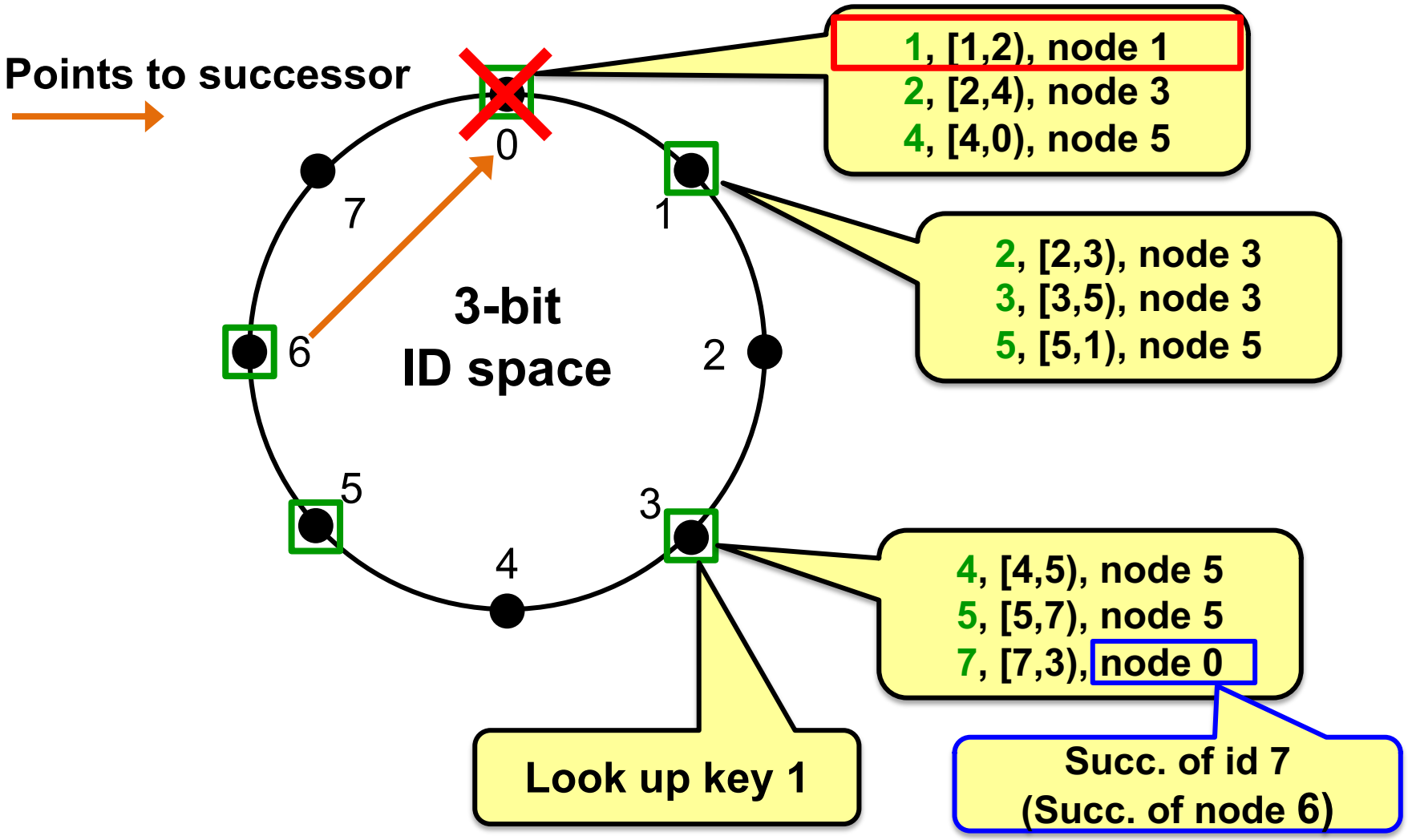
Chord – failures and successor list



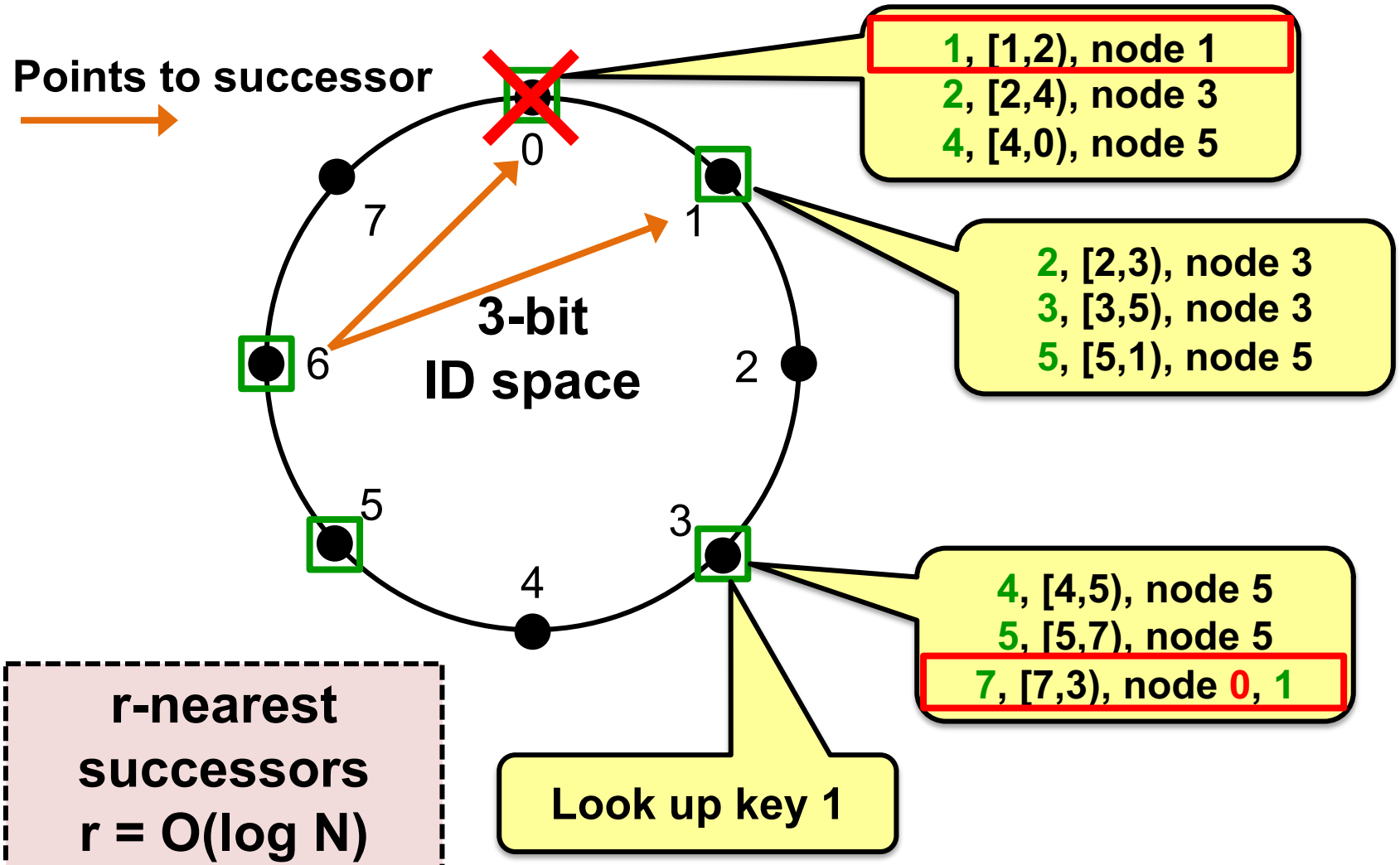
Chord – failures and successor list



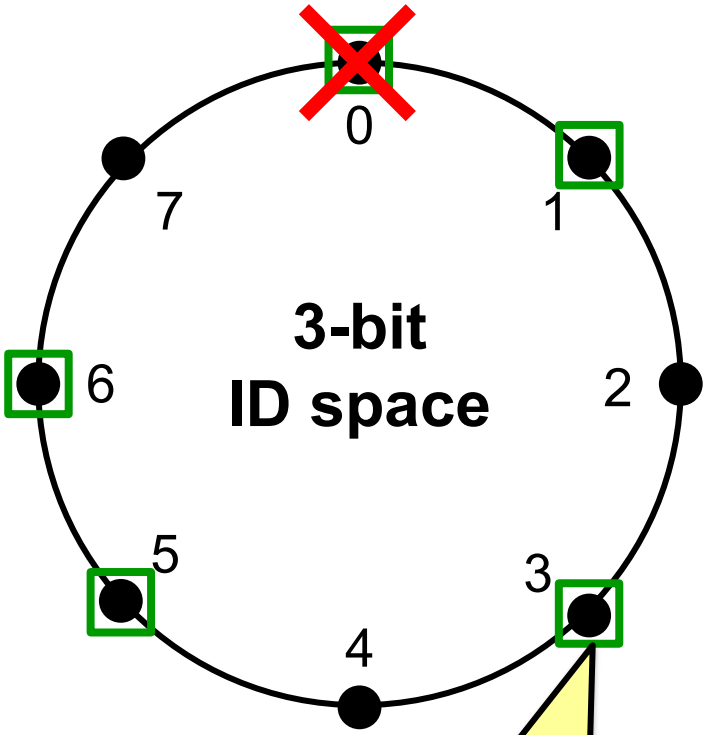
Chord – failures and successor list



Chord – failures and successor list



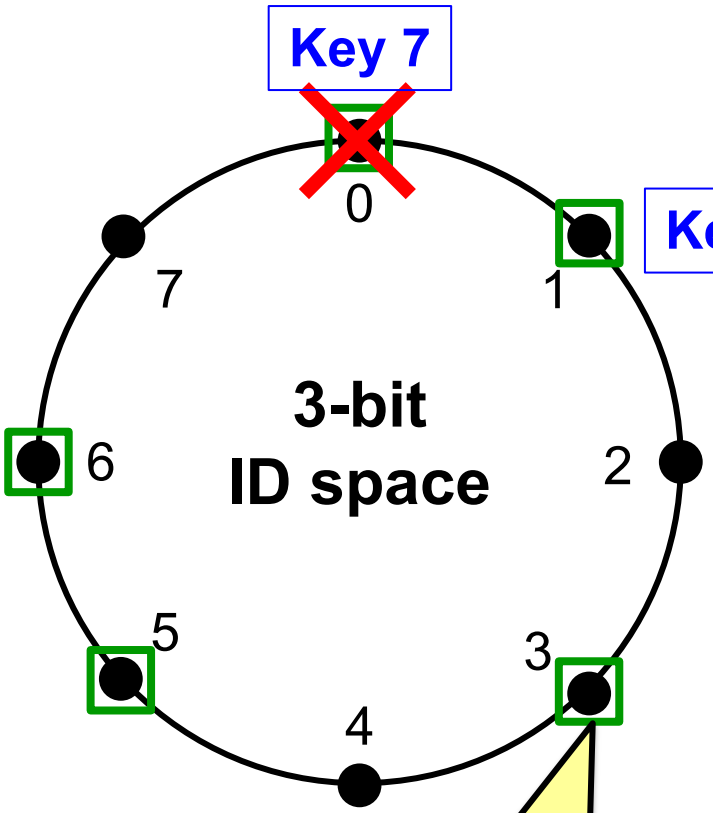
Chord – failures and successor list



r-nearest successors
r = O(log N)

What if look up key 7?

DHash replicates blocks at *r* successors



"Adjacent" nodes in the ring may be far away in the network
→ Independent failures

r-nearest successors
r = O(log N)

What if look up key 7?

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.* intra/inter-datacenter, client-server model)
   - 1M nodes = 20 hops; 50ms/hop → 1s lookup latency

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 hoped-for impact

What DHTs got right

- **Consistent hashing**
  - Elegant way to divide a workload across machines
  - Very useful in clusters: actively used today in Amazon Dynamo, Apache Cassandra 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