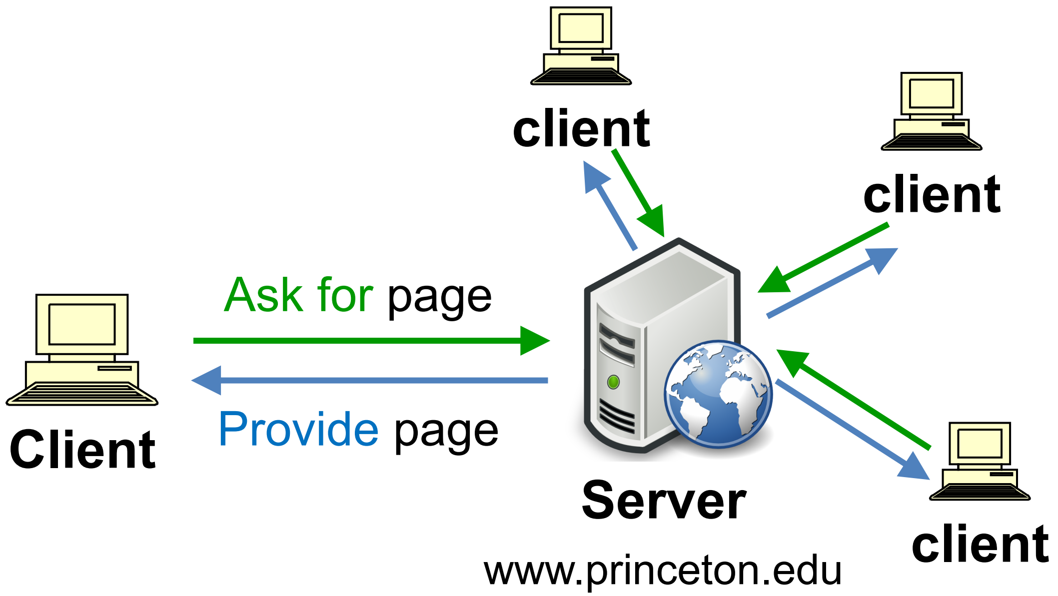
Peer-to-Peer Systems and Distributed Hash Tables

COS 418: Distributed Systems
Lecture 9

Haonan Lu

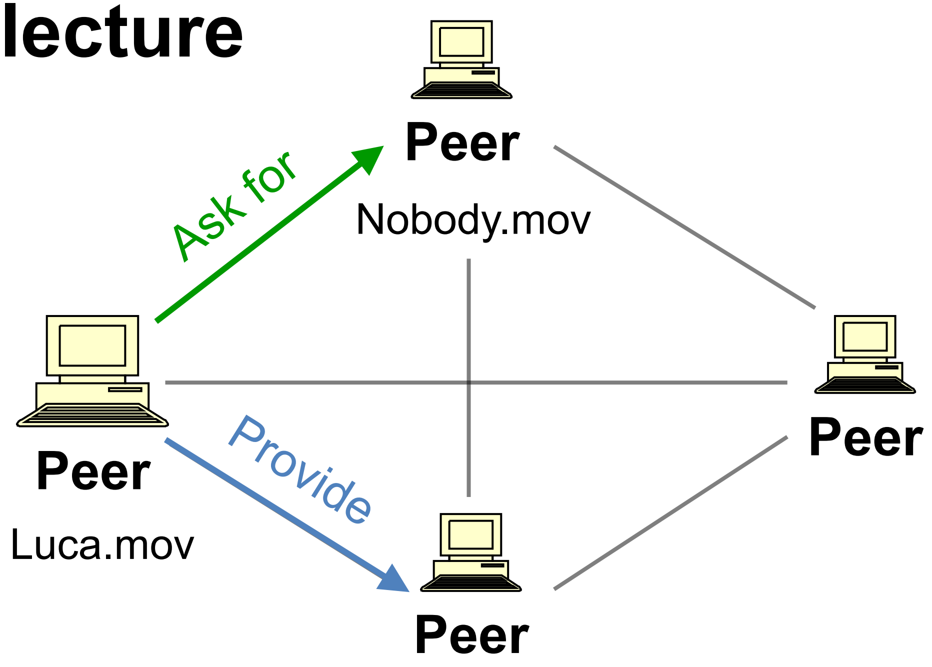
[Credit: Slides Adapted from Mike Freedman, Wyatt Lloyd, Kyle Jamieson and Daniel Suo]

Distributed Application Architecture



Client-Server

This lecture



Peer-to-Peer

Today

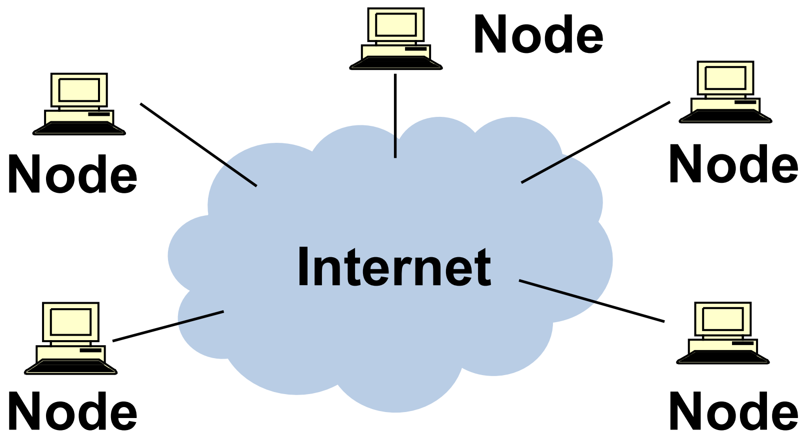
1. **Peer-to-Peer Systems**
   – **What, why, and the core challenge**

2. Distributed Hash Tables (DHT)

3. The Chord Lookup Service

4. Concluding thoughts on DHTs, P2P

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**
    1. Napster (1990s), Gnutella, BitTorrent, etc.

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

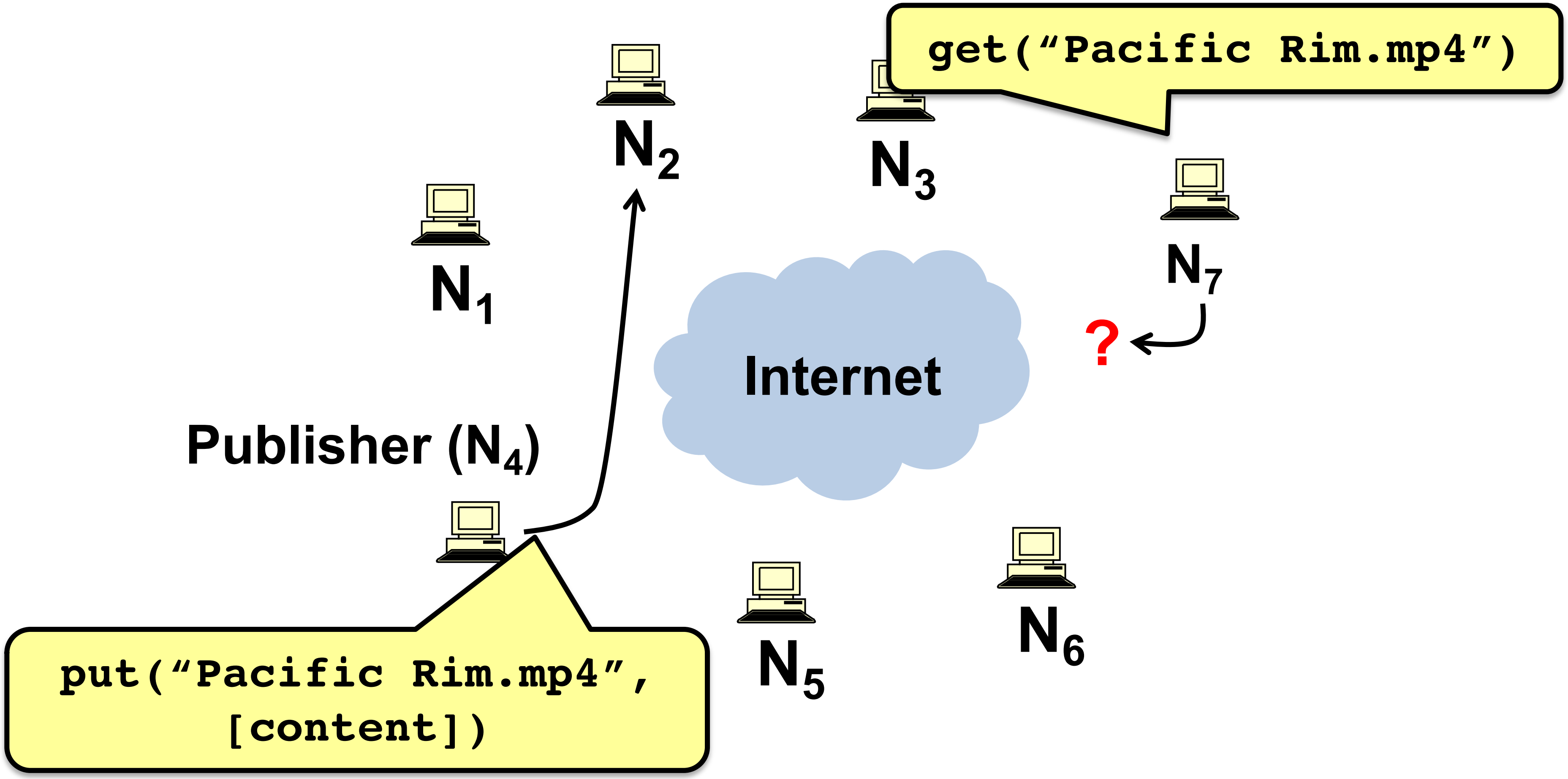
3. **Voice/video telephony:** user to user anyway (Skype in old days)
    – Issues: Privacy and control

Why might P2P be a win?

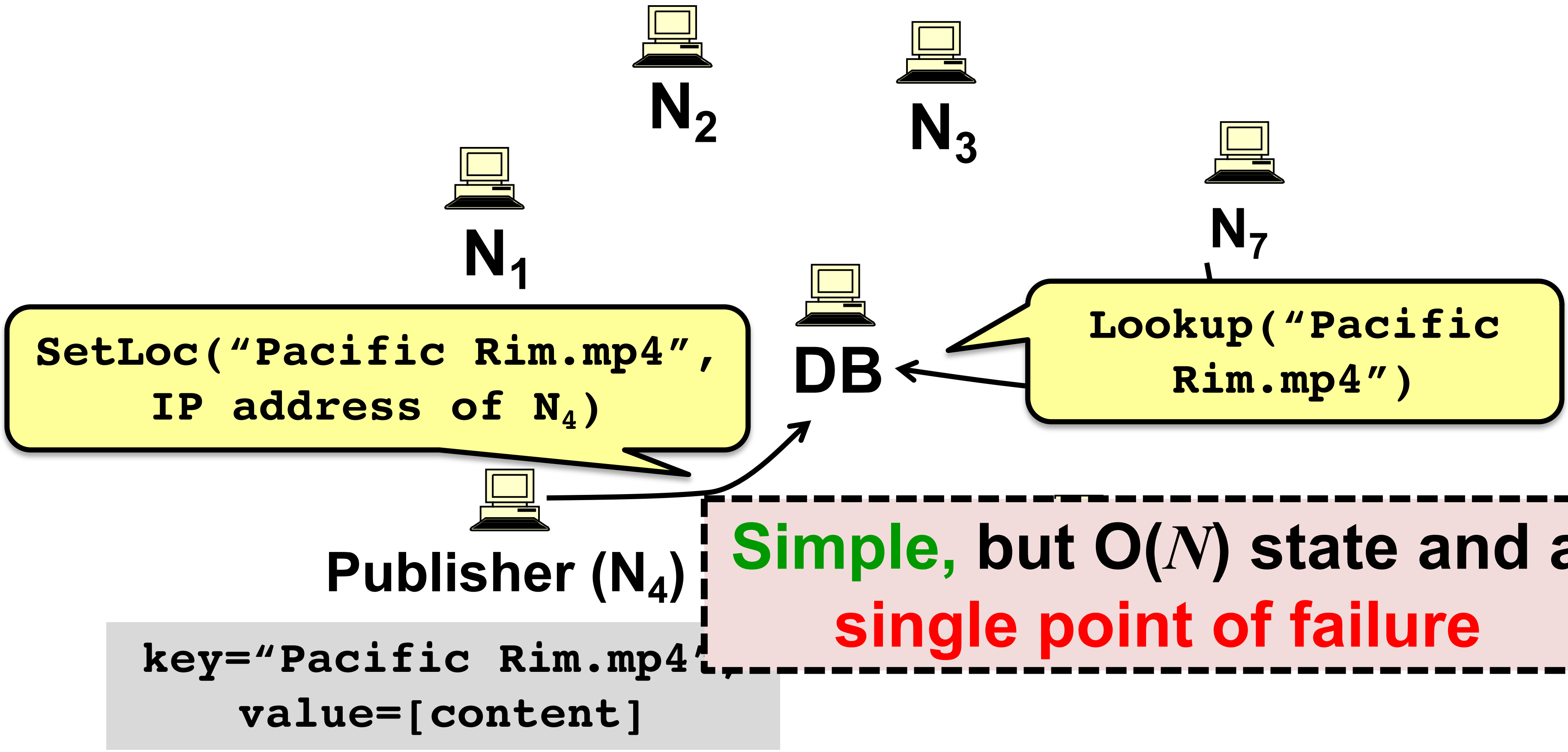
- **High capacity for services** through parallelism and scalability:
  - More disks, network connections, CPUs, etc. as peers join
  - Data are divided and duplicated, accessible from multiple peers concurrently

- **Absence of a centralized server** may mean:
  - **Less chance** of service overload as load increases
  - Easier **deployment**
  - A single failure **won't wreck** the whole system (no single point of failure)
  - System as a whole is **harder to attack**

The lookup problem: locate the data

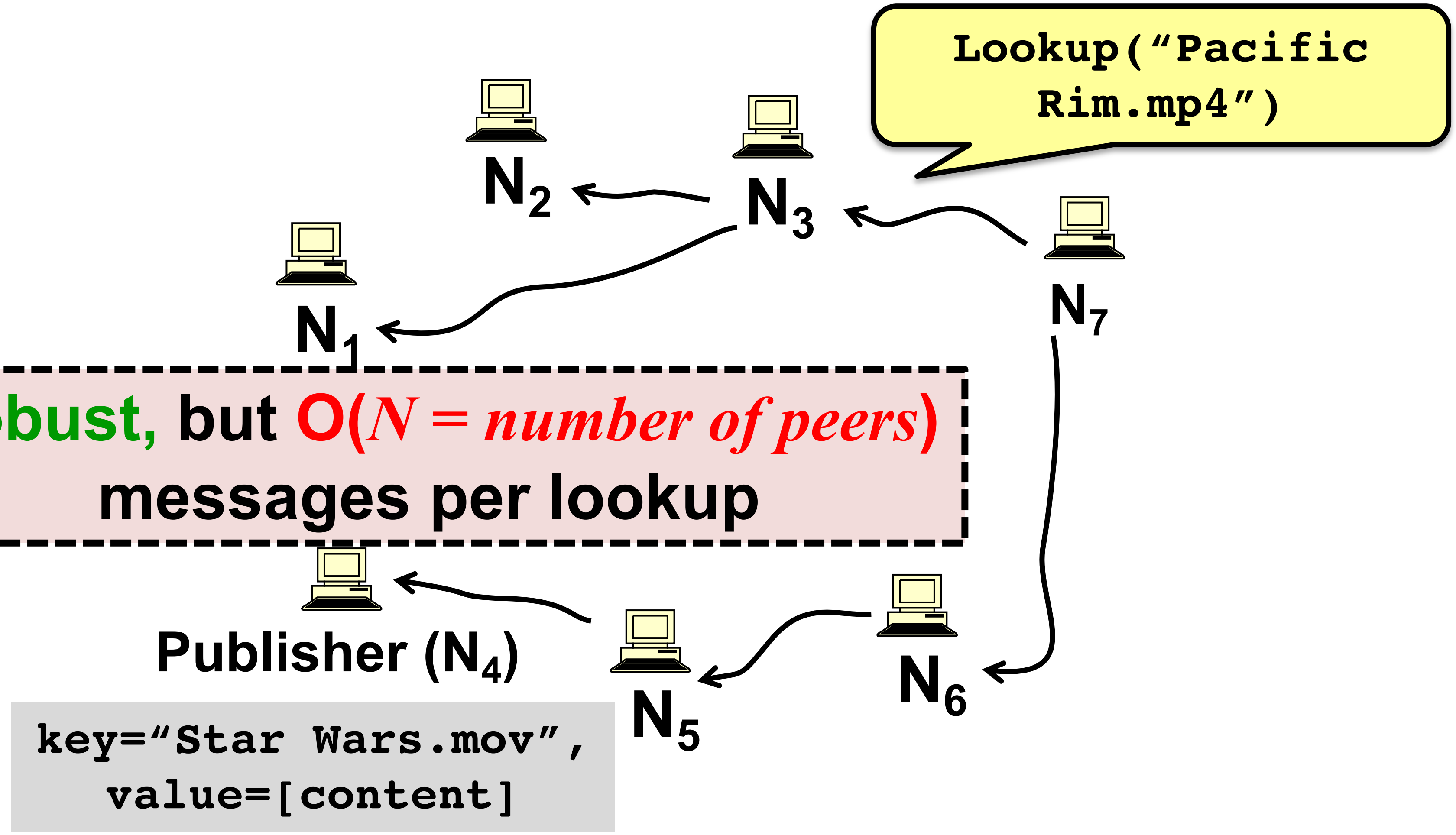


Centralized lookup (Napster)



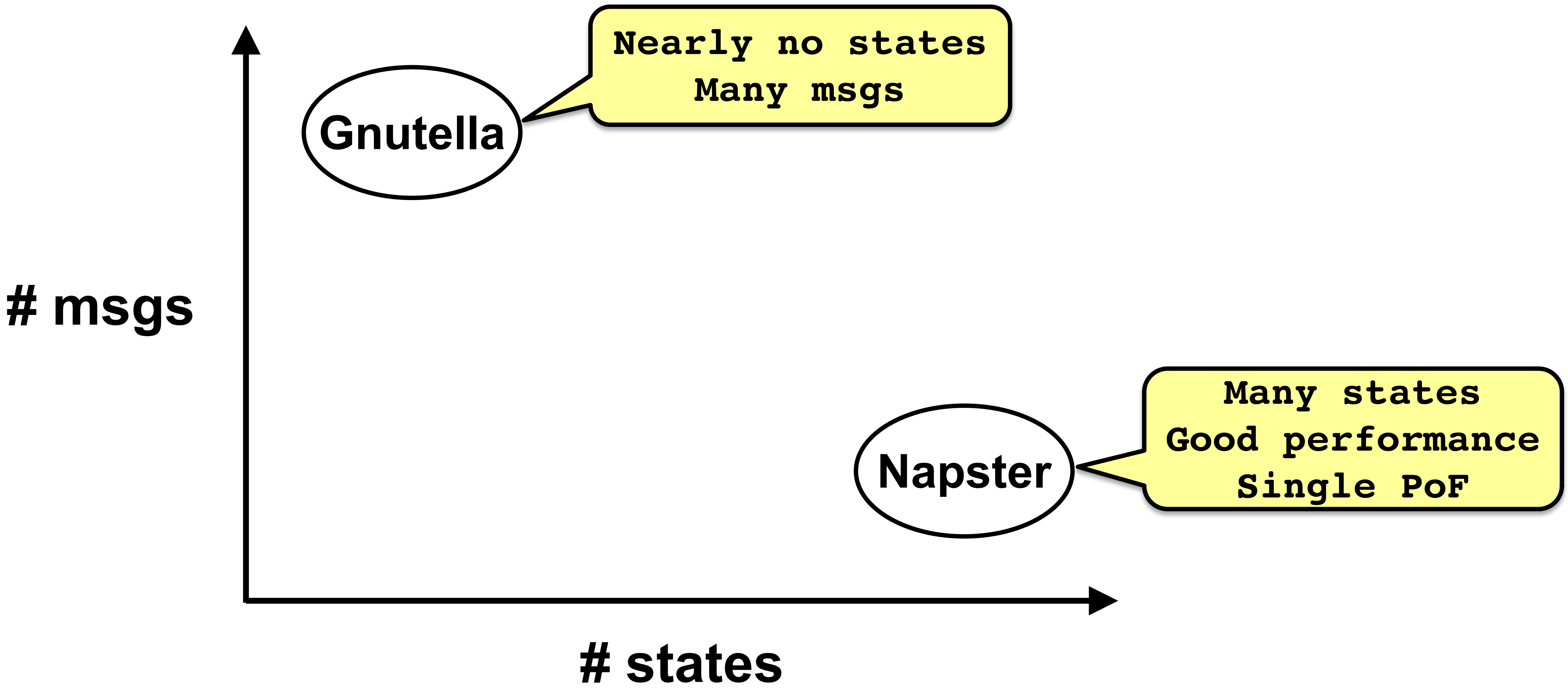
Simple, but O(N) state and a single point of failure

Flooded queries (original Gnutella)

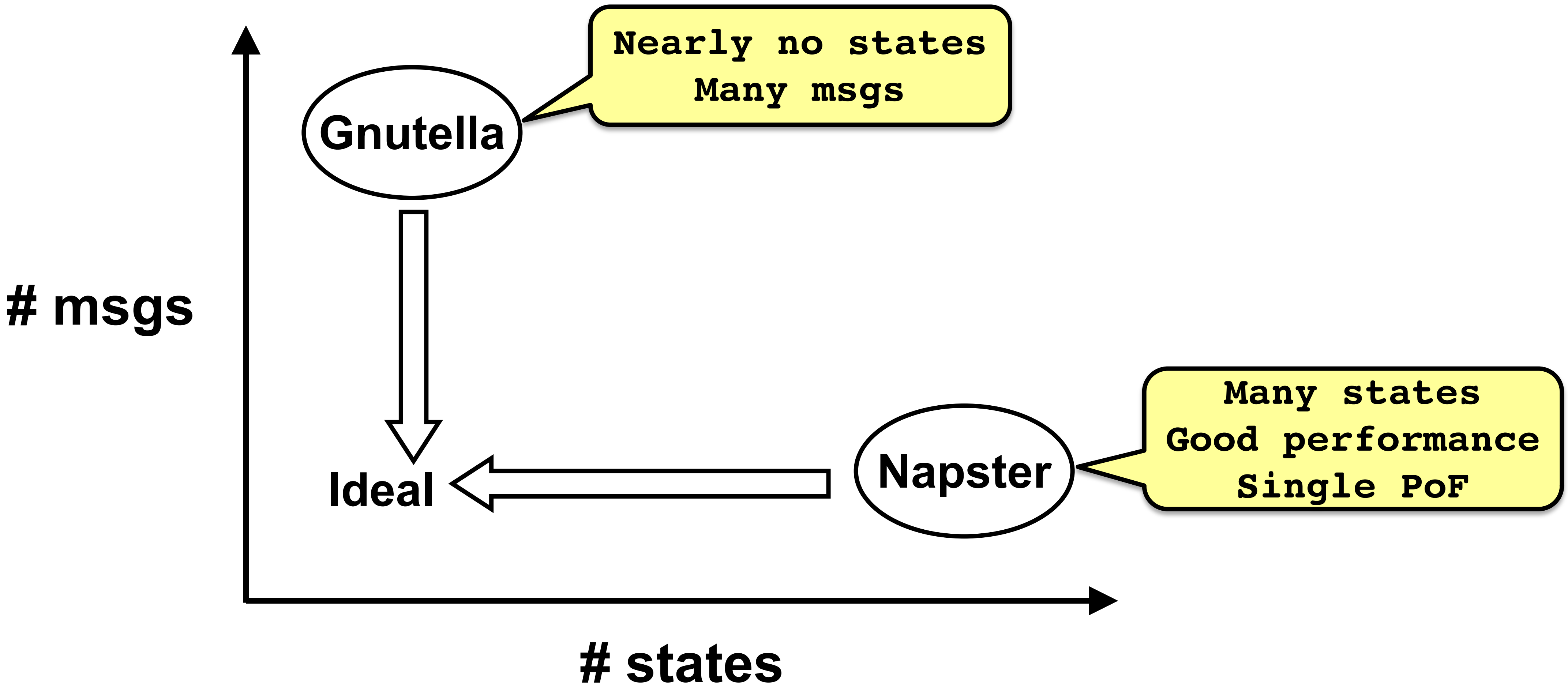


Robust, but **O(N = *number of peers*)** messages per lookup

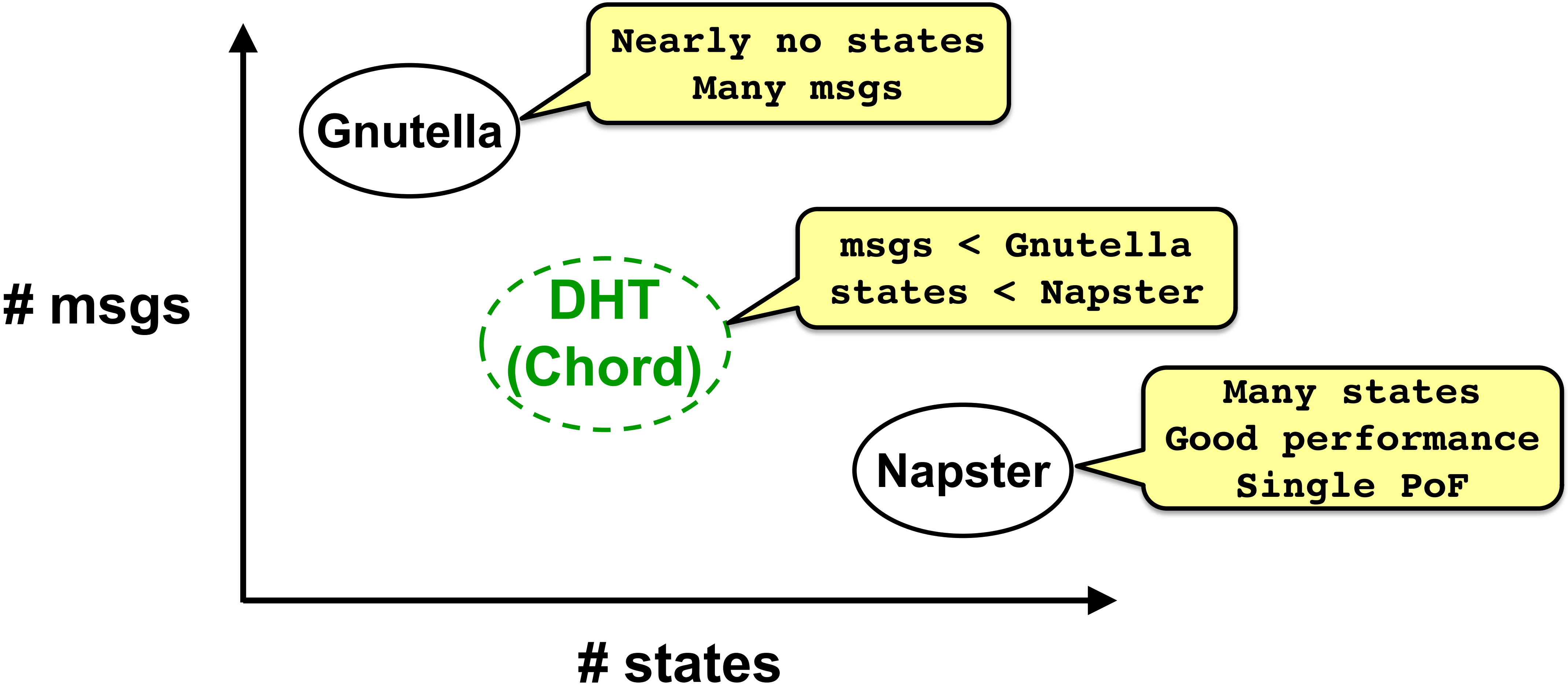
Tradeoffs in distributed systems



Tradeoffs in distributed systems



Tradeoffs in distributed systems



Today

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

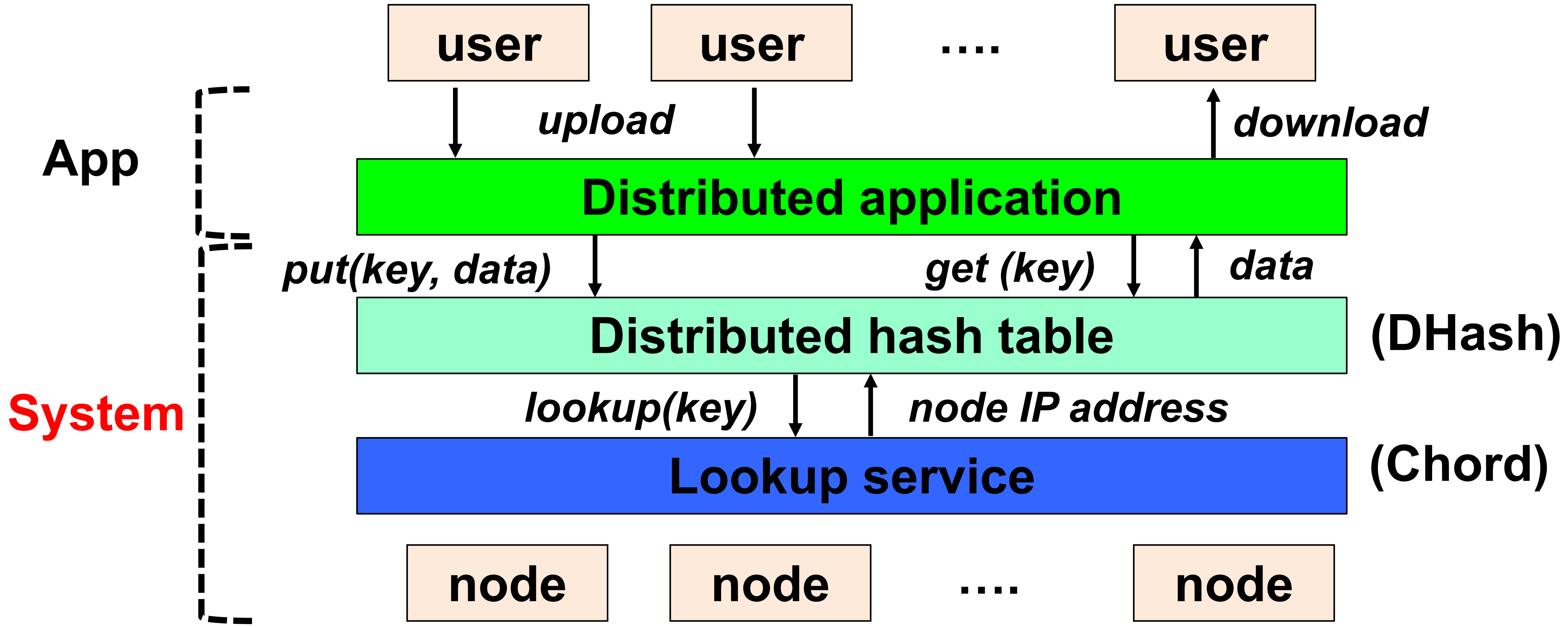
What is a DHT (and why)?

- Distributed Hash Table: an abstraction of hash table in a distributed setting

```
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 (DHT)

3. **The Chord Lookup Service**

Chord identifiers

- **Hashed values (integers) 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

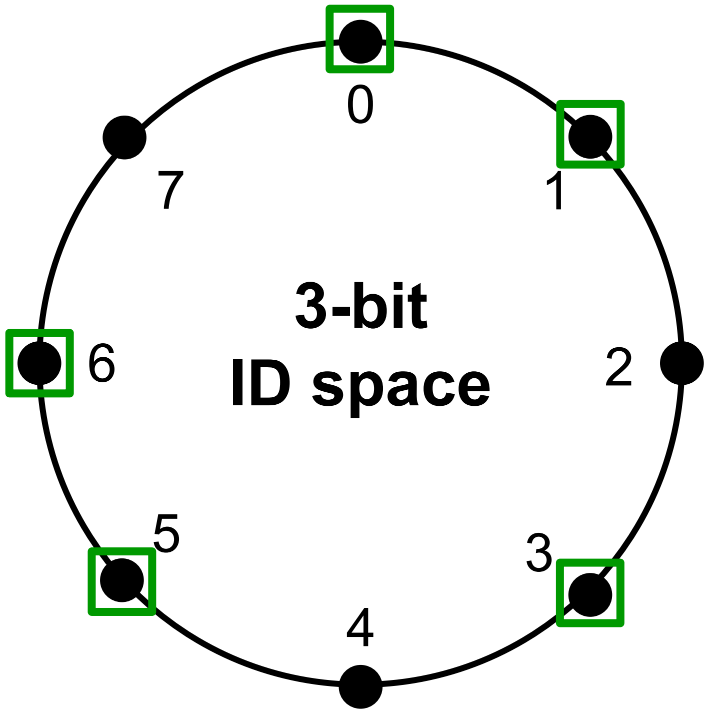
- **Why hash key and address?**
  - Uniformly distributed in the ID space
  - Hashed key → load balancing; hashed address → independent failure

Consistent hashing [Karger ‘97] – data partition

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

● **Identifiers/key space**

□ **Node**



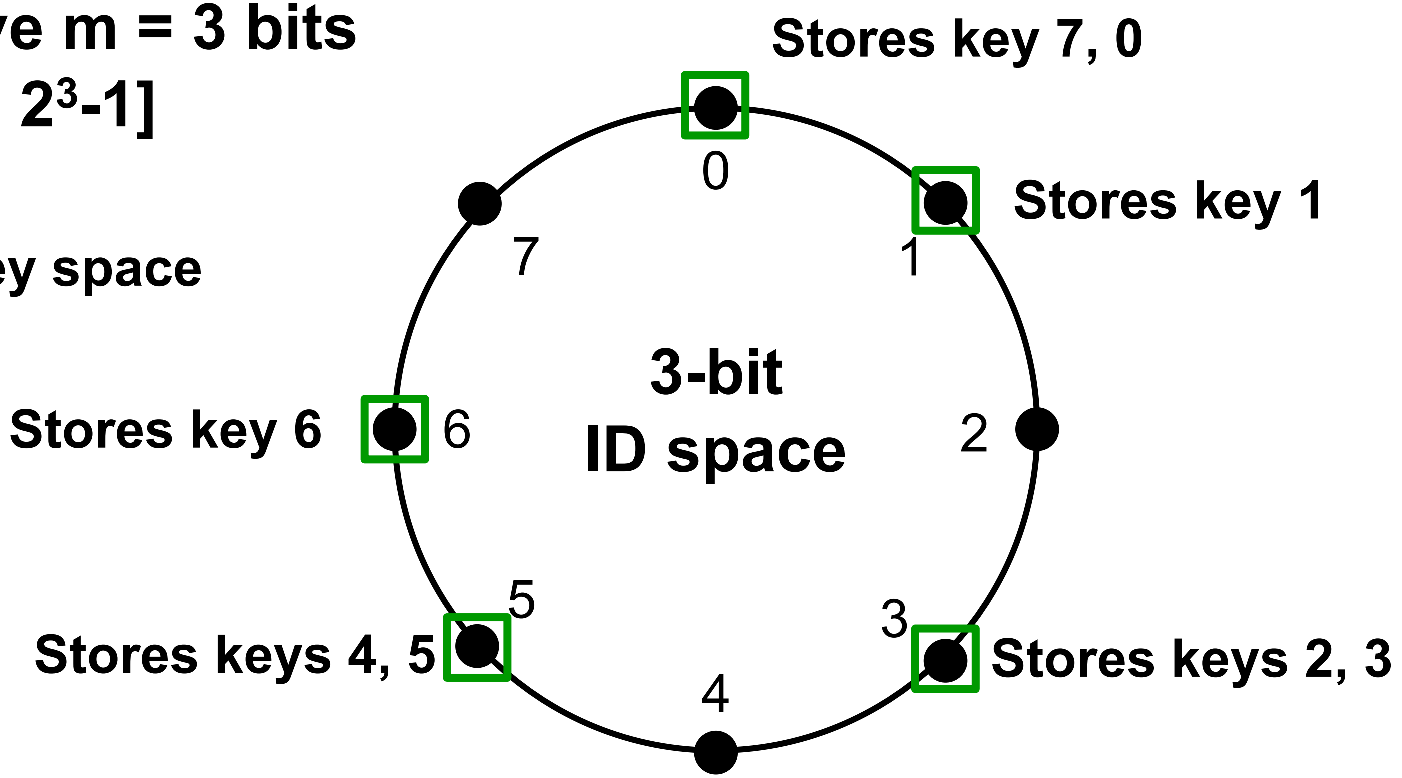
Consistent hashing [Karger ‘97] – data partition

Identifiers have m = 3 bits

Key space: [0, 2^3-1]

● **Identifiers/key space**

□ **Node**



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

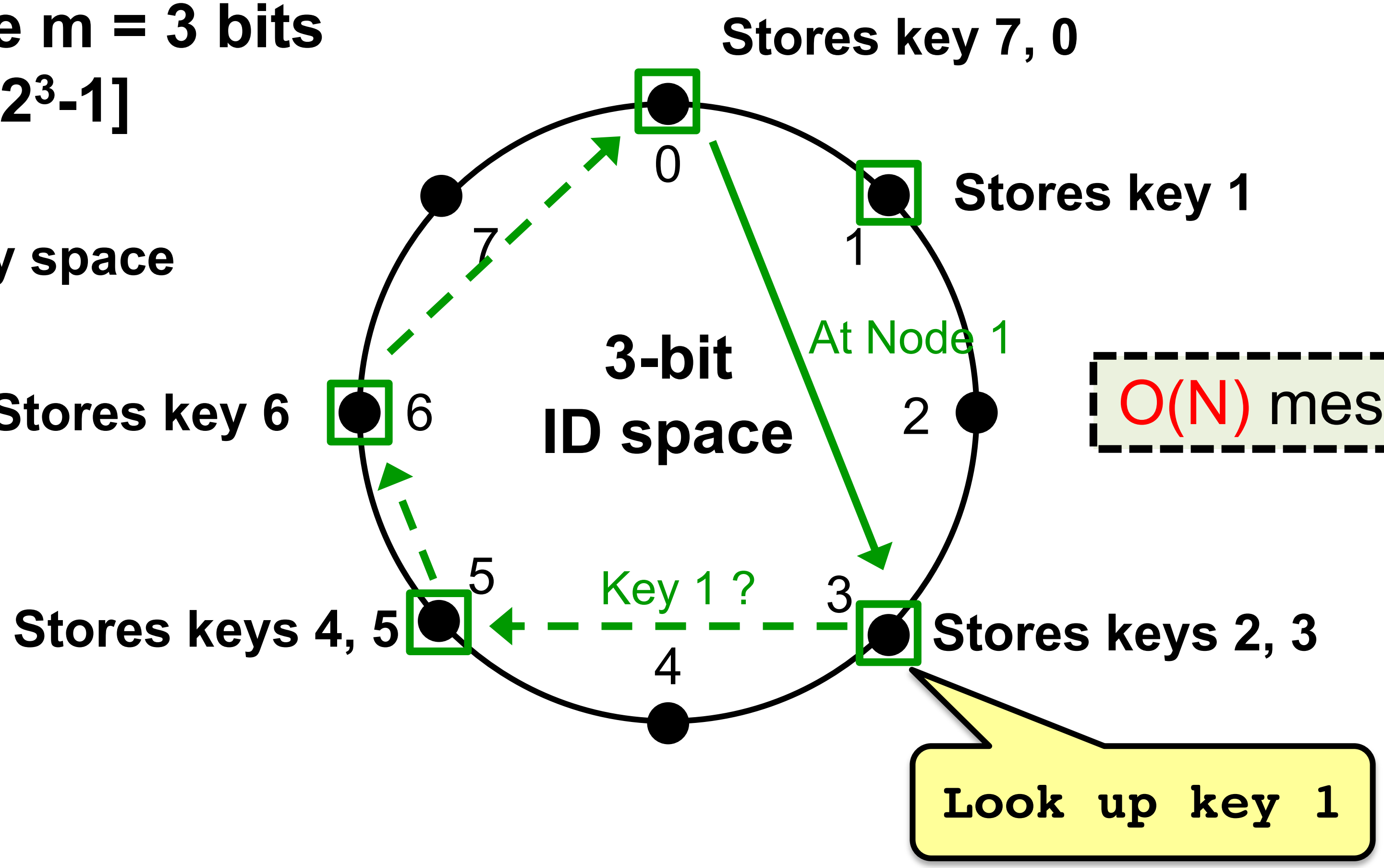
Consistent hashing [Karger '97] – basic lookup

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

● Identifiers/key space

□ Node

- - ➜ Successor pointer

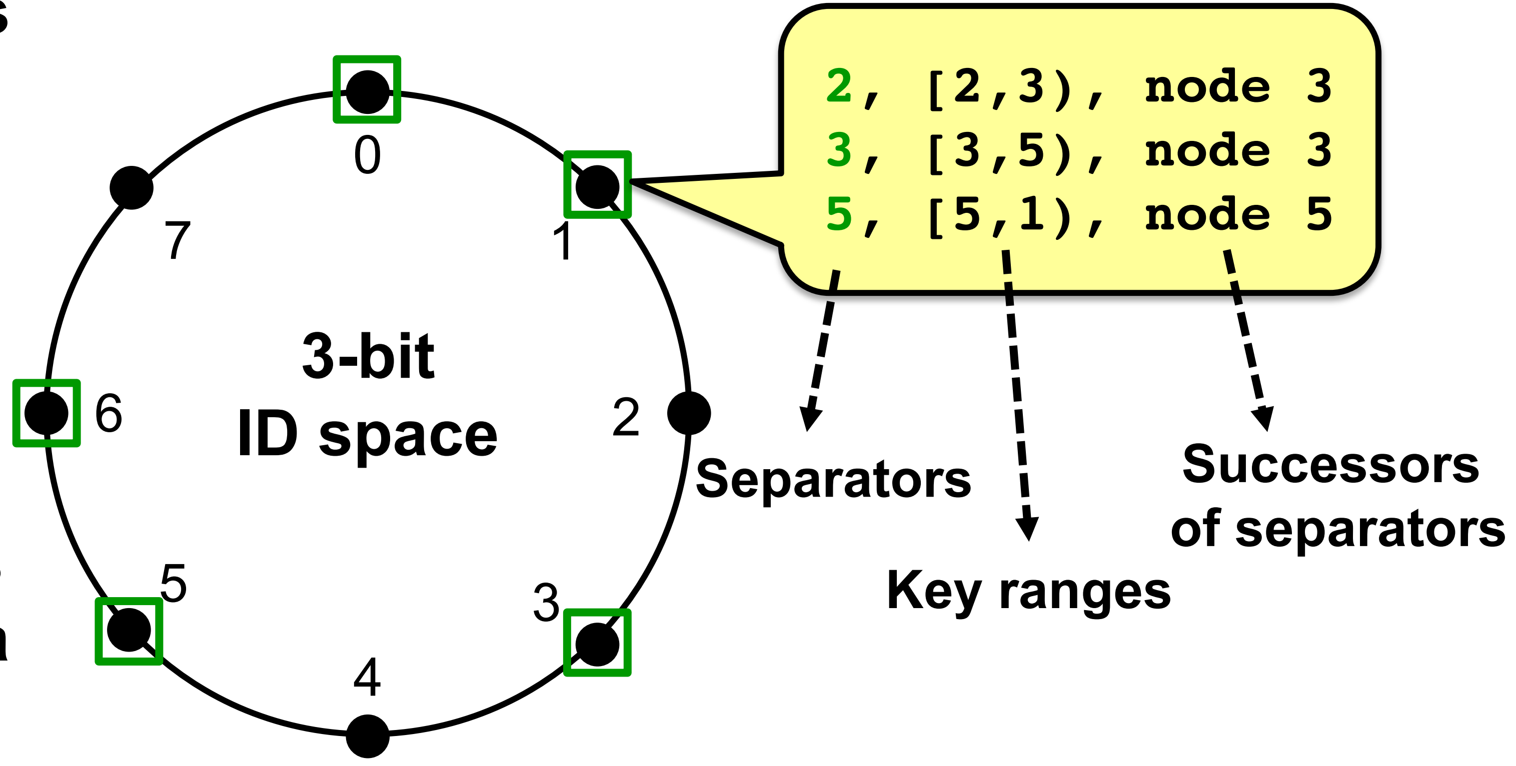


O(N) messages and hops!

Chord – finger tables

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

- ● Identifiers/key space
- □ Node



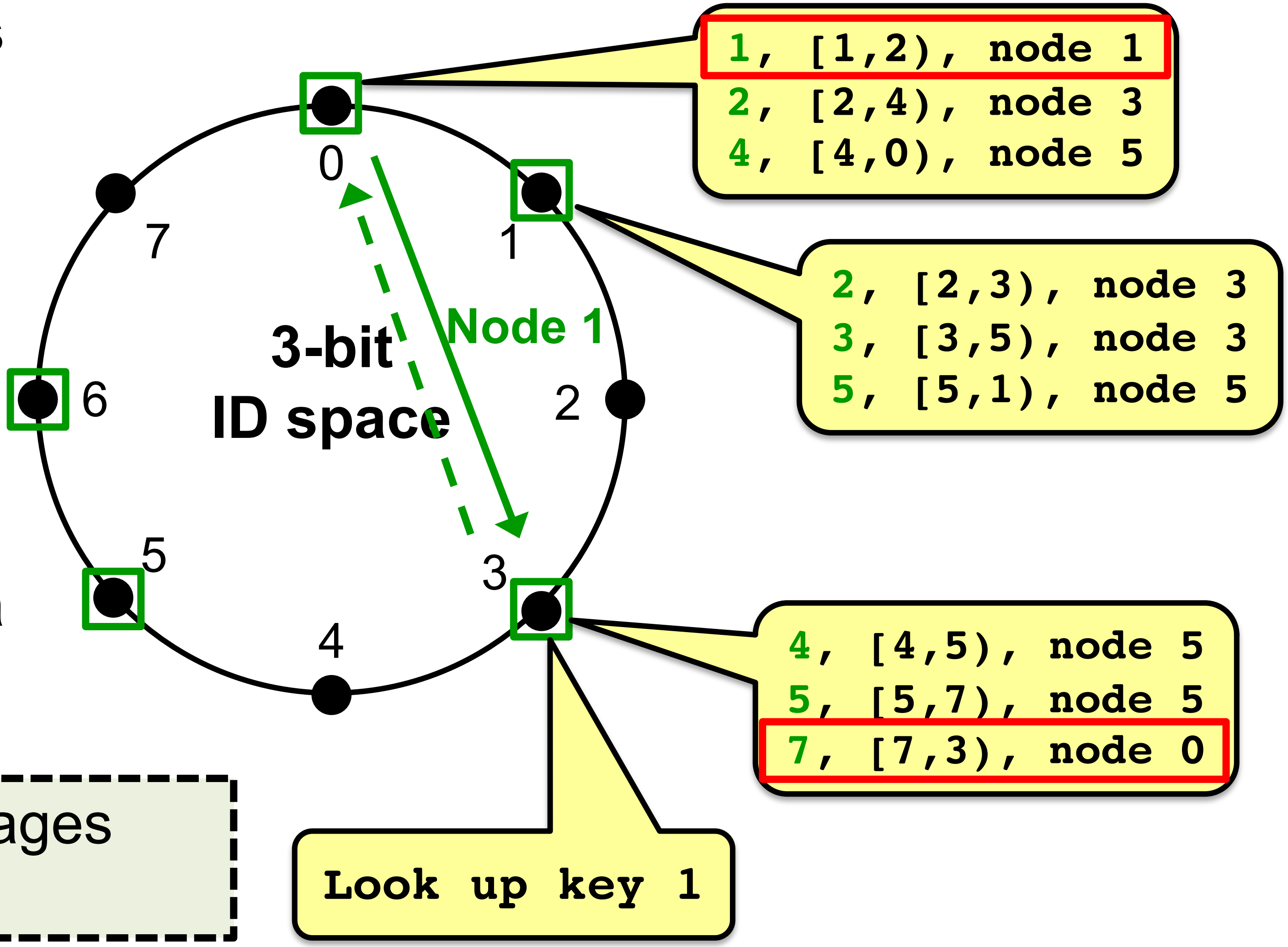
Each node keeps **m** states
Key space → **m** ranges via
$(N+2^{k-1}) \bmod 2^m$, $1 \le k \le m$

Chord – finger tables

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

● **Identifiers/key space**
□ **Node**

Each node keeps m states
Key space → m ranges via
$(N+2^{k-1}) \bmod 2^m$, 1<=k<=m



O(logN) messages
and hops!

Implication of finger tables

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

- Better than arranging 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

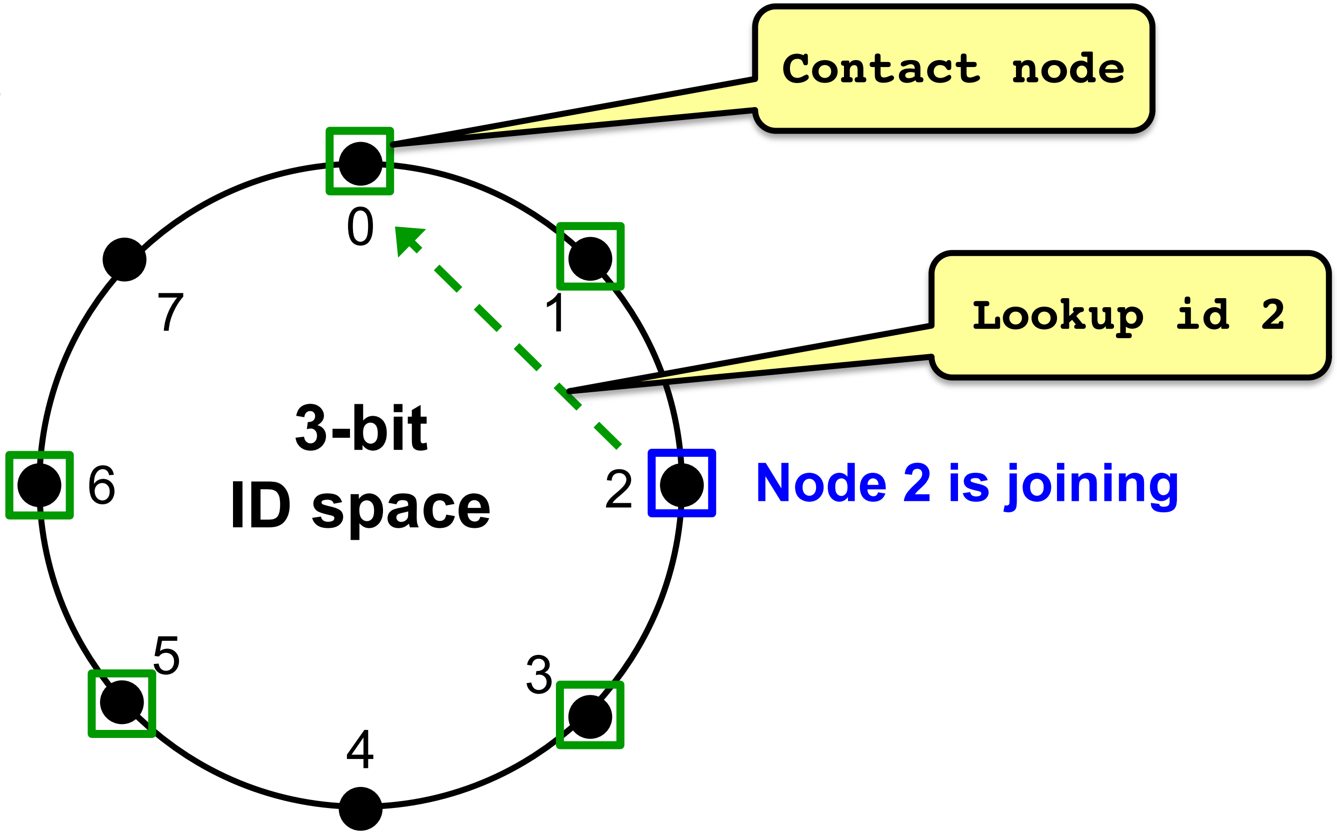
Chord lookup algorithm properties

- **Interface:** lookup(key) $\rightarrow$ IP address
- **Efficient:** O(log N) messages per lookup
  - N is the total number of nodes (peers)
- **Scalable:** O(log N) state per node
- **Robust:** survives massive failures

Chord – node joining

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

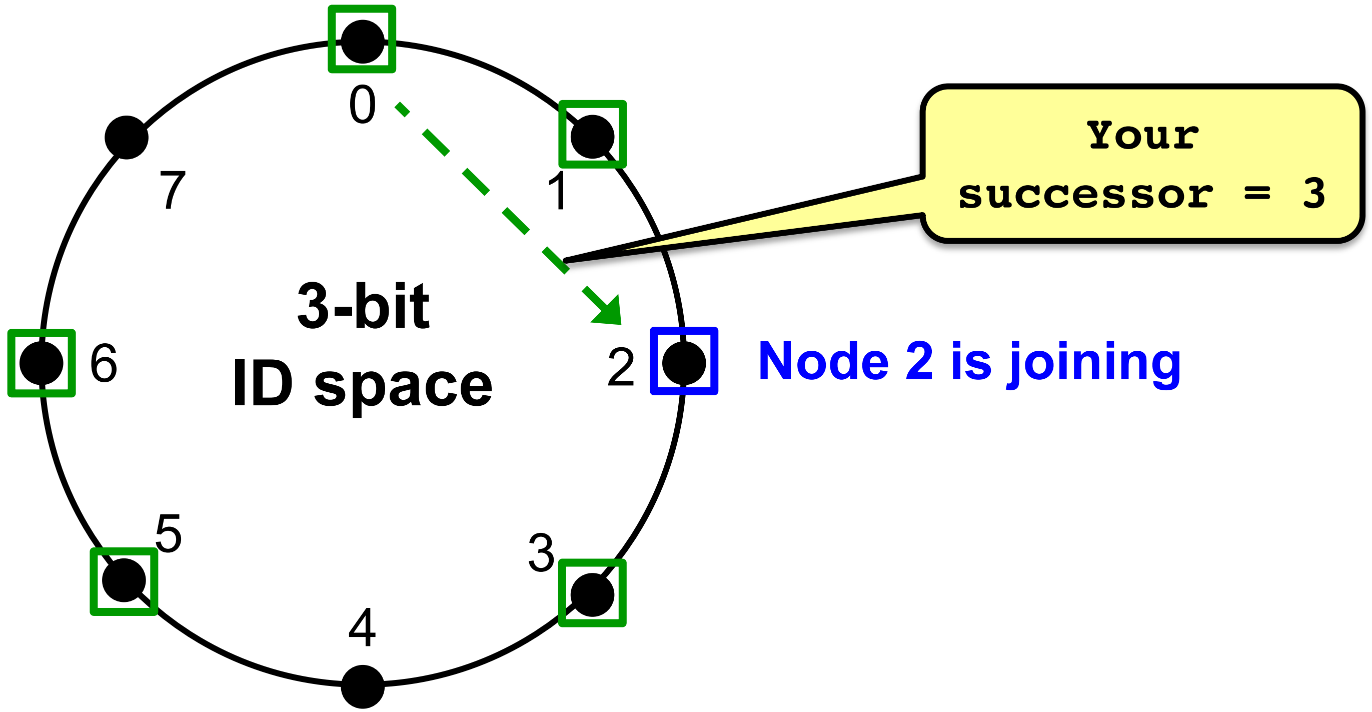
● **Identifiers/key space**
□ **Node**



Chord – node joining

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

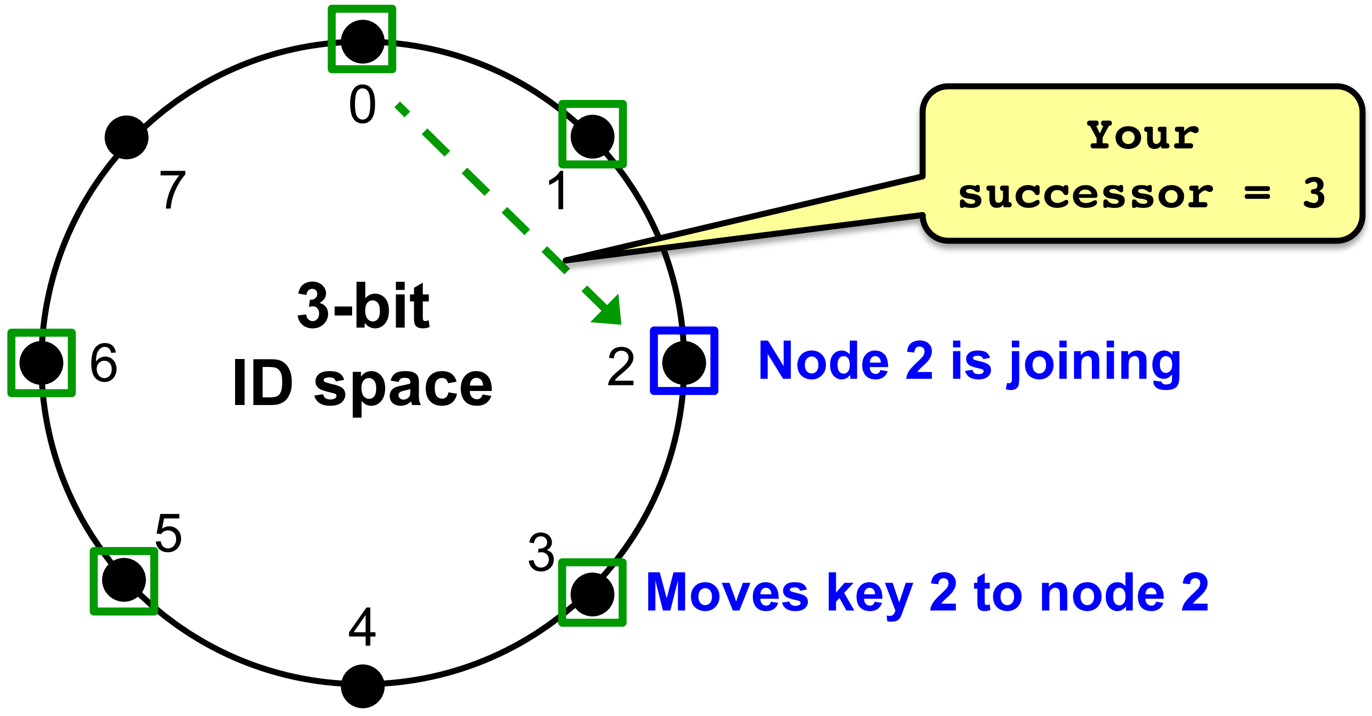
- ● **Identifiers/key space**
- □ **Node**



Chord – node joining

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

- ● **Identifiers/key space**
- ☐ **Node**



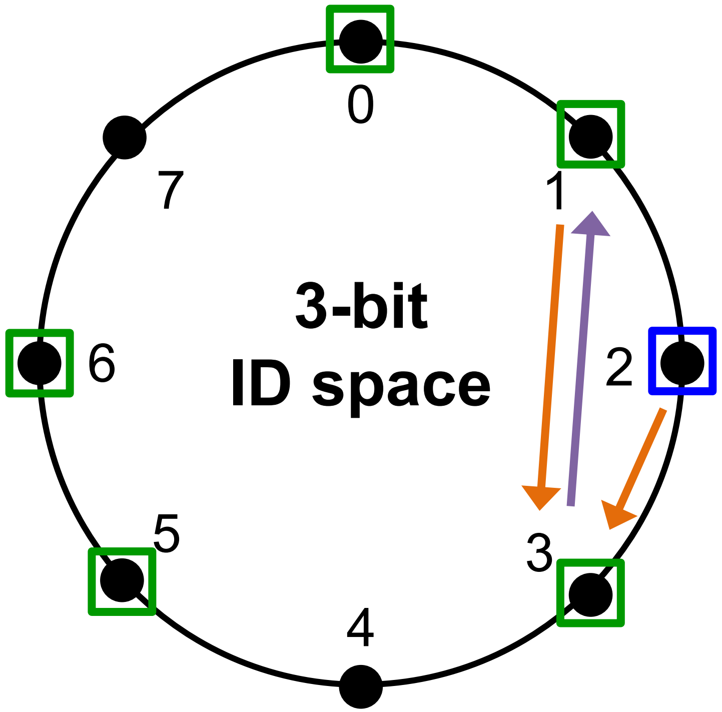
Chord – node joining

Identifiers have m = 3 bits

Key space: [0, 2^3-1]

● **Identifiers/key space**

□ **Node**



Points to successor

Points to predecessor

Node 2 is joining

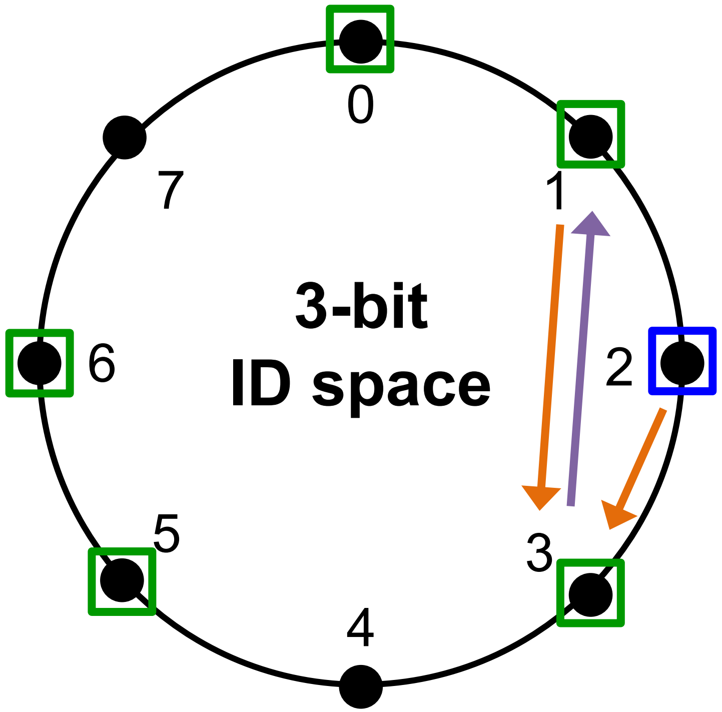
Periodic stabalization messages from each node to its successor maintain node positions

Chord – node joining

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

● **Identifiers/key space**

□ **Node**



→ **Points to successor** (orange)

→ **Points to predecessor** (purple)

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

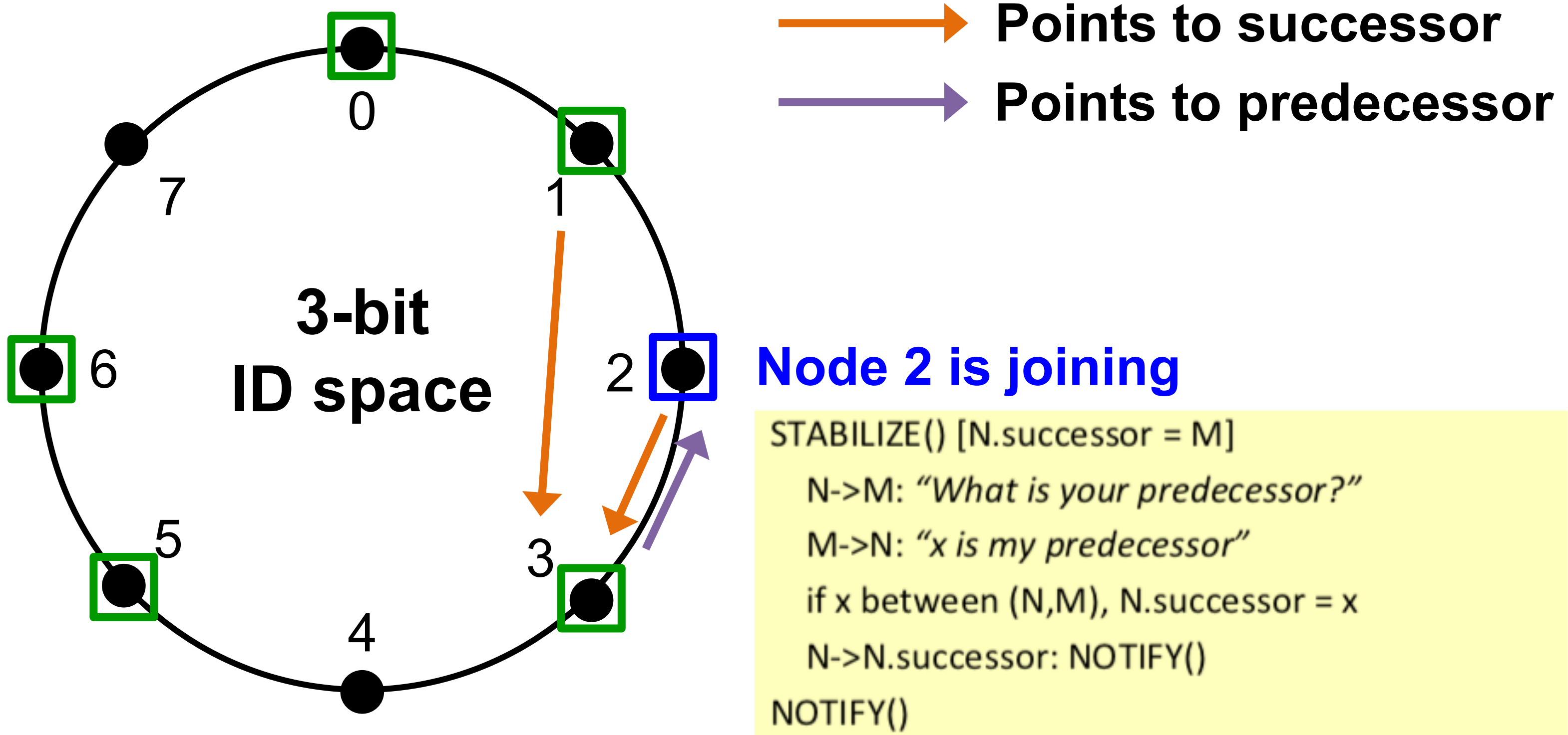
*The pseudocode comes from Rodrigo Fonseca's lecture notes

Chord – node joining

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

● **Identifiers/key space**

□ **Node**



Node 2 is joining

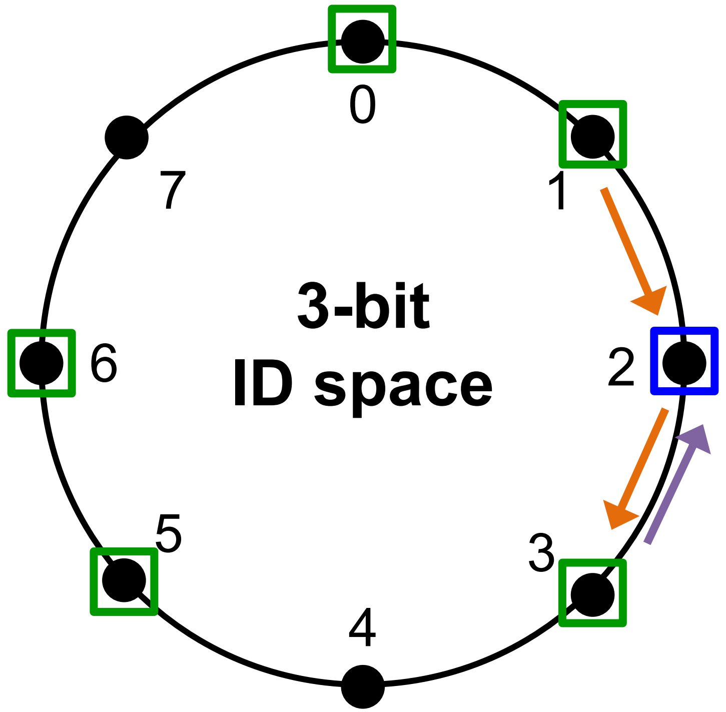
STABILIZE() [N.successor = M]
N->M: *"What is your predecessor?"*
M->N: *"x is my predecessor"*
if x between (N,M), N.successor = x
N->N.successor: NOTIFY()
NOTIFY()
N->N.successor: *"I think you are my successor"*
M: upon receiving NOTIFY from N:
If (N between (M.predecessor, M))
M.predecessor = N

*The pseudocode comes from Rodrigo Fonseca's lecture notes

Chord – node joining

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

● **Identifiers/key space**
□ **Node**



Points to successor (orange arrow)
Points to predecessor (purple arrow)

Node 2 is joining

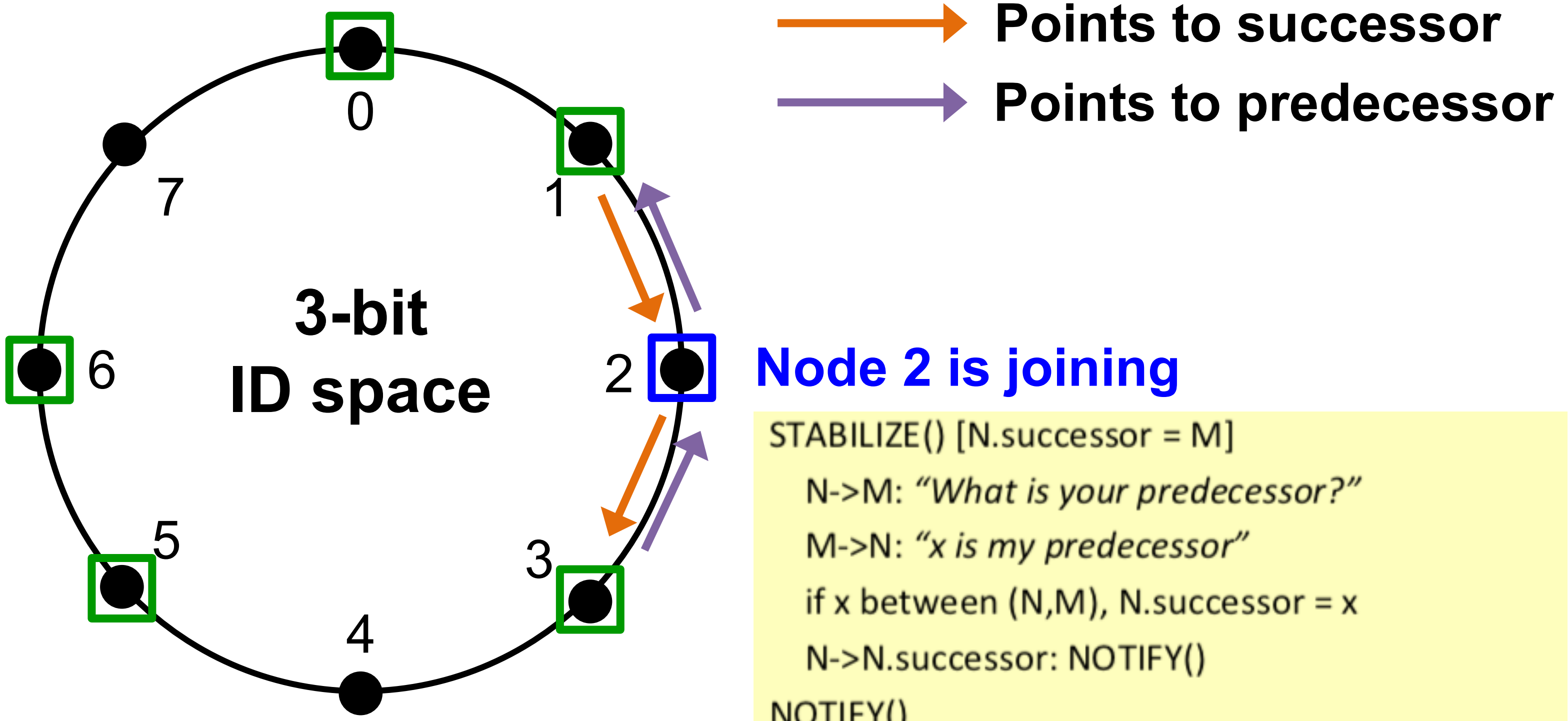
```
STABILIZE() [N.successor = M]
   N->M: "What is your predecessor?"
   M->N: "x is my predecessor"
   if x between (N,M), N.successor = x
   N->N.successor: NOTIFY()
NOTIFY()
    N->N.successor: "I think you are my successor"
M: upon receiving NOTIFY from N:
    If (N between (M.predecessor, M))
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

*The pseudocode comes from Rodrigo Fonseca's lecture notes

Chord – node joining

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

- Identifiers/key space
- Node



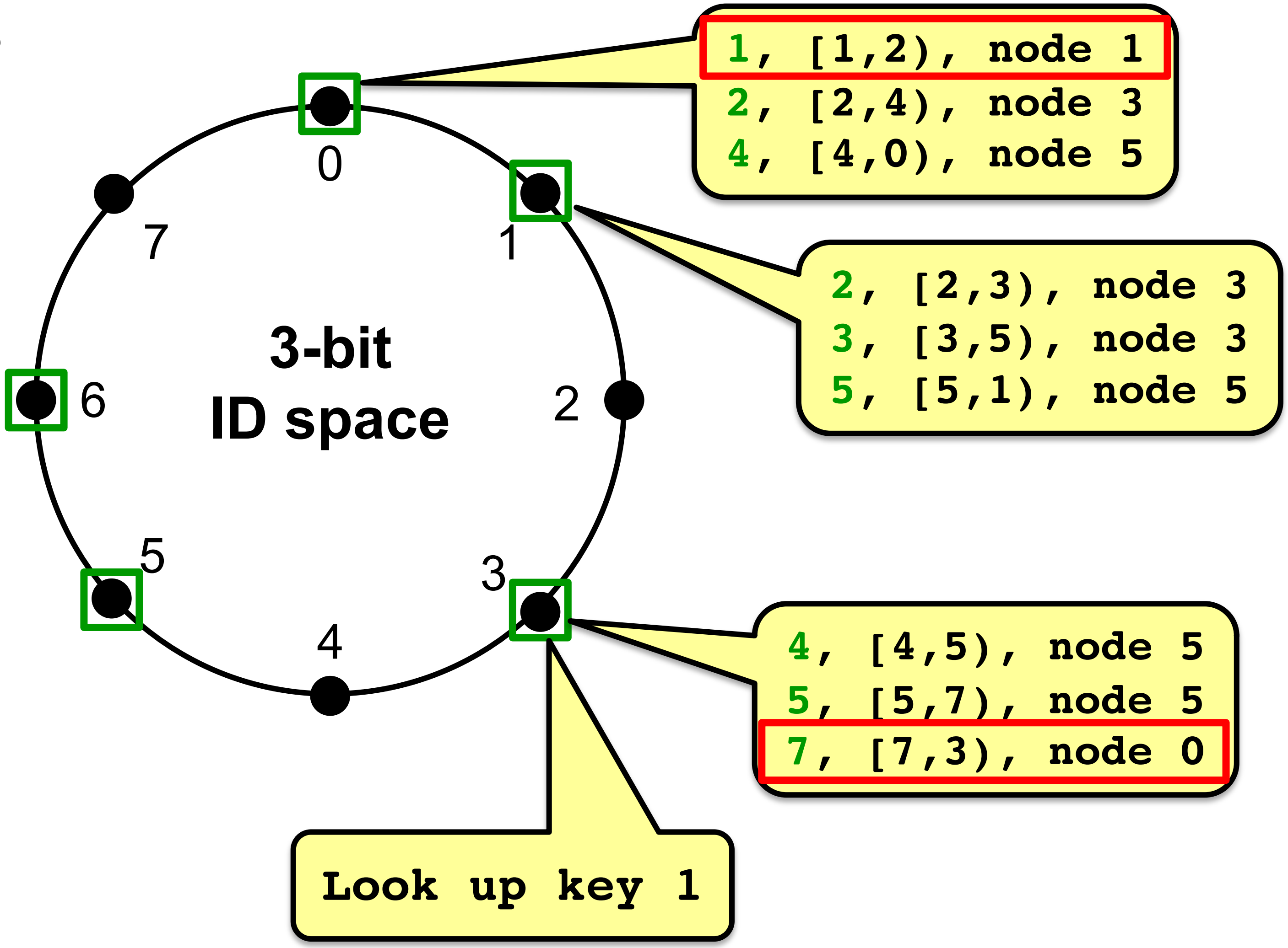
STABILIZE() [N.successor = M]
N->M: *"What is your predecessor?"*
M->N: *"x is my predecessor"*
if x between (N,M), N.successor = x
N->N.successor: NOTIFY()
NOTIFY()
N->N.successor: *"I think you are my successor"*
M: upon receiving NOTIFY from N:
If (N between (M.predecessor, M))
M.predecessor = N

*The pseudocode comes from Rodrigo Fonseca's lecture notes

Chord – failures and successor list

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

● **Identifiers/key space**
□ **Node**

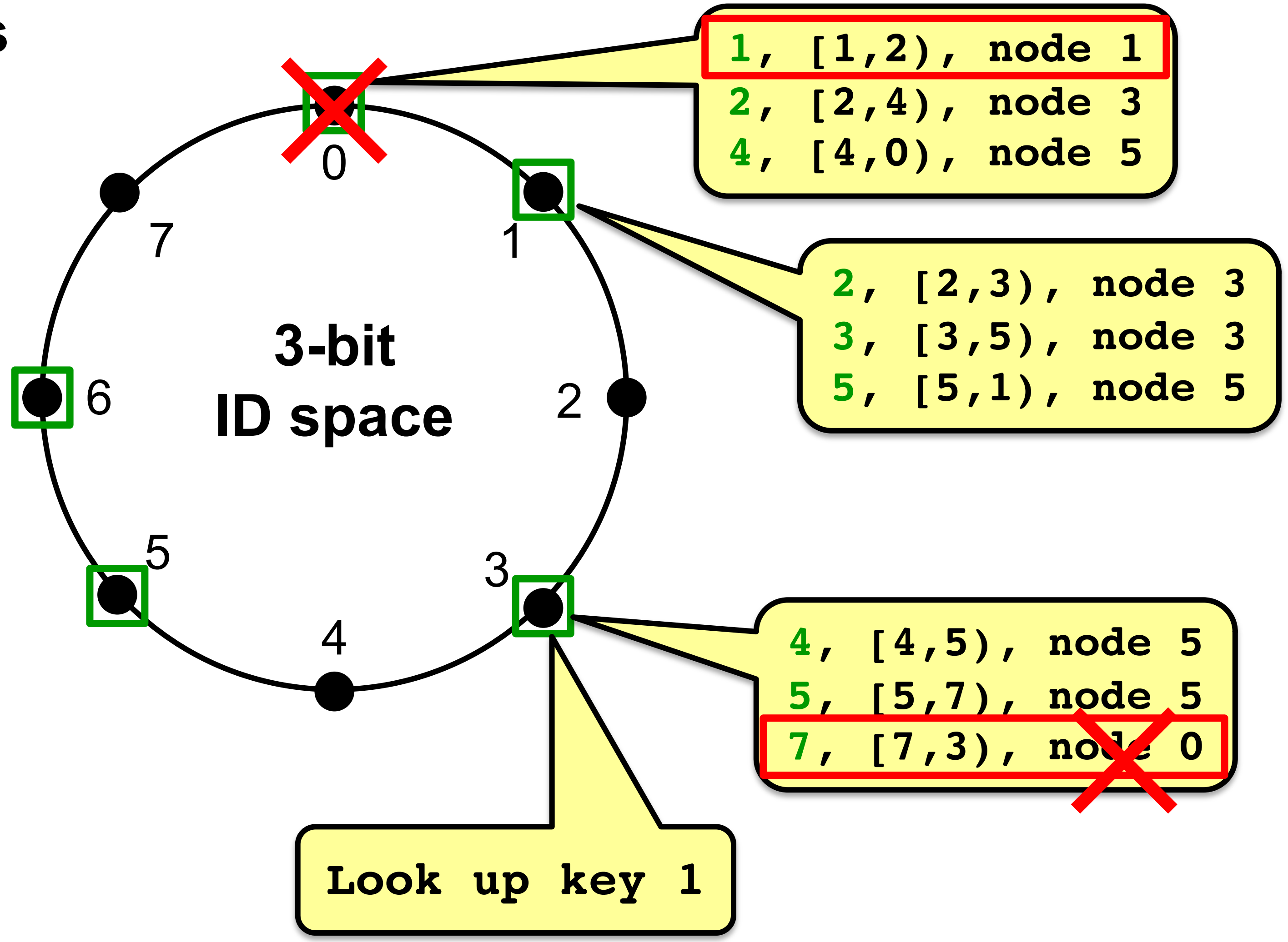


Chord – failures and successor list

Identifiers have m = 3 bits

Key space: [0, 2^3-1]

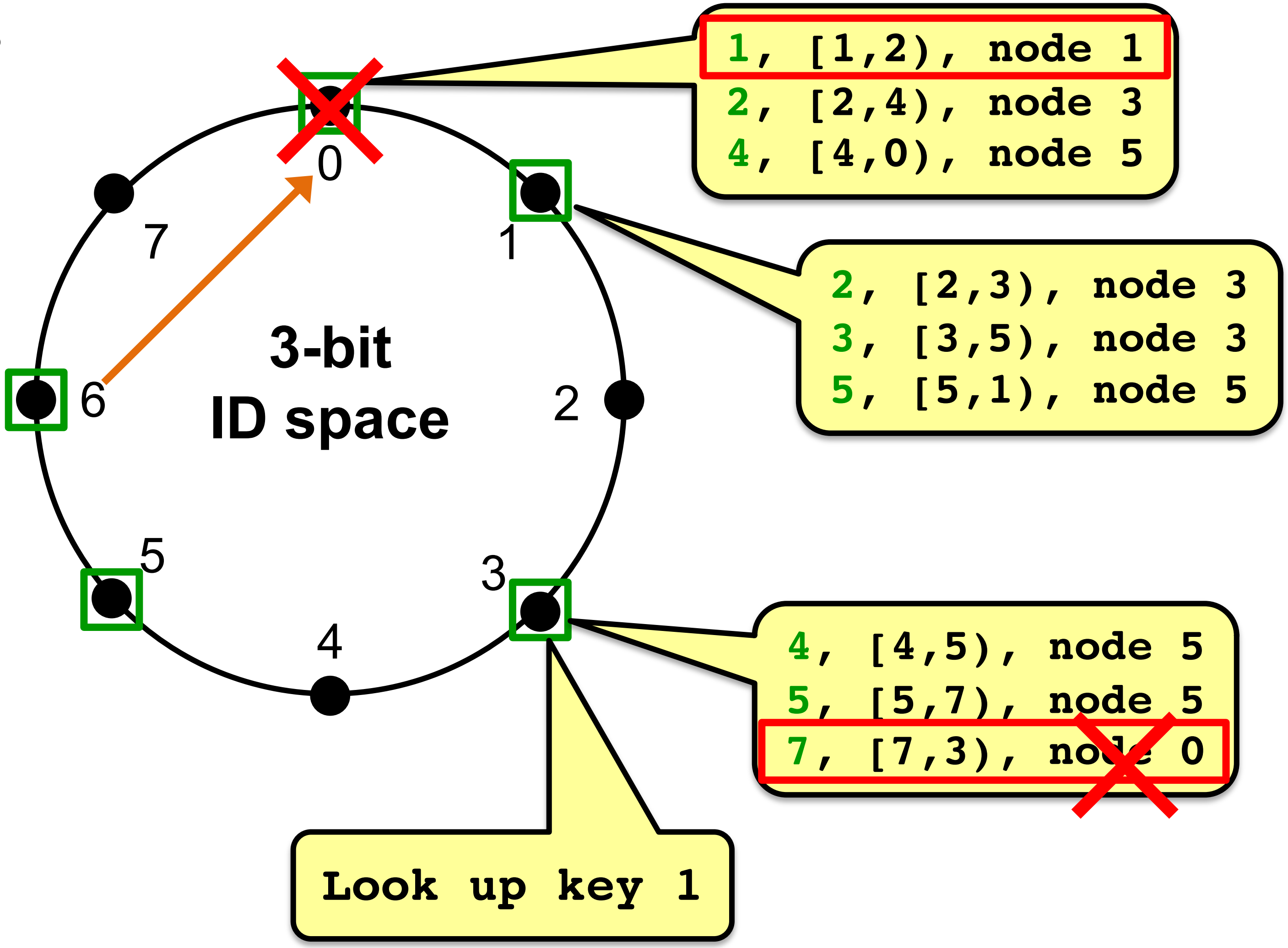
- Identifiers/key space
- Node



Chord – failures and successor list

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

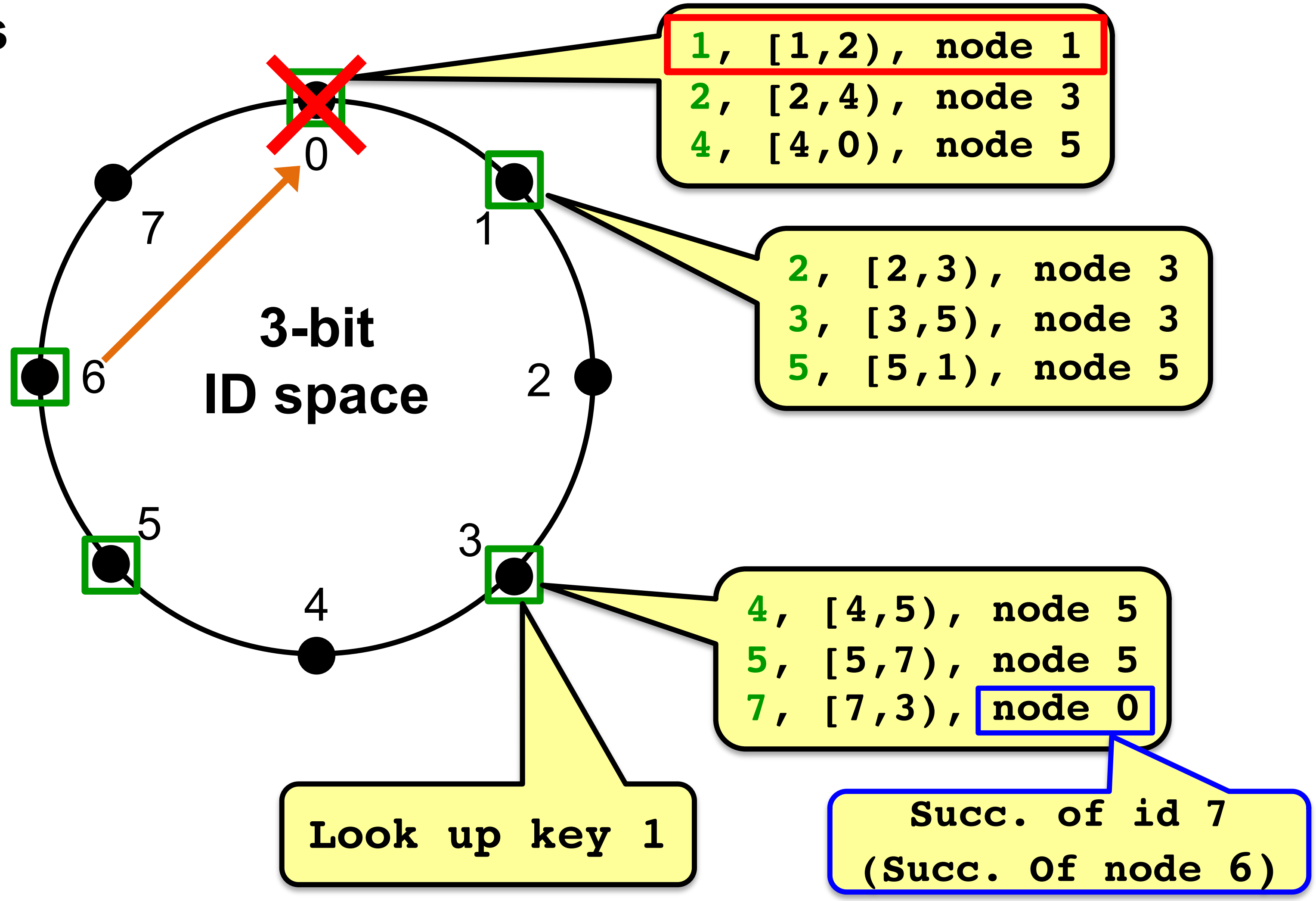
- ● **Identifiers/key space**
- □ **Node**
- → **Points to successor**



Chord – failures and successor list

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

● Identifiers/key space
□ Node
→ Points to successor



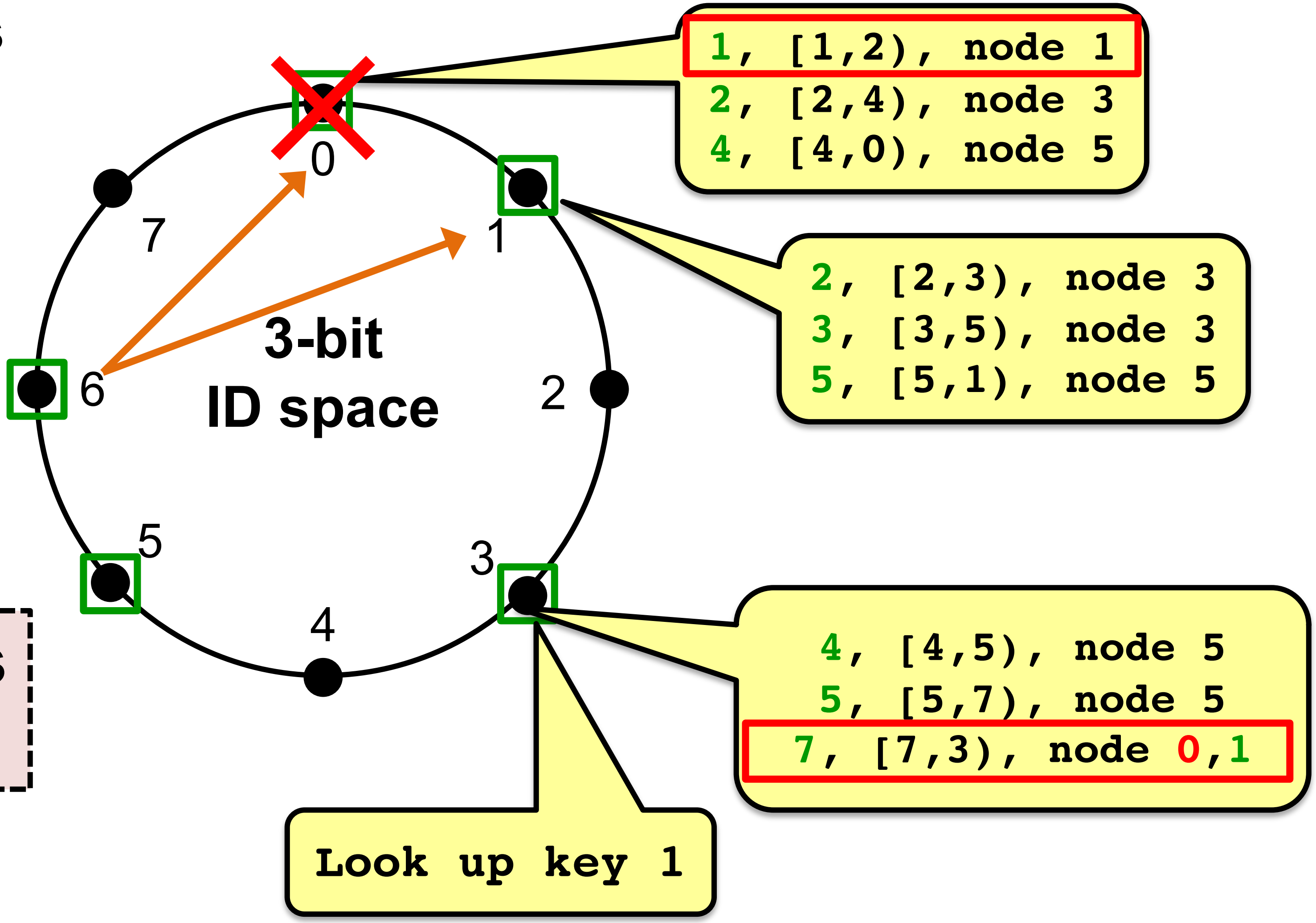
Chord – failures and successor list

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

● **Identifiers/key space**
□ **Node**

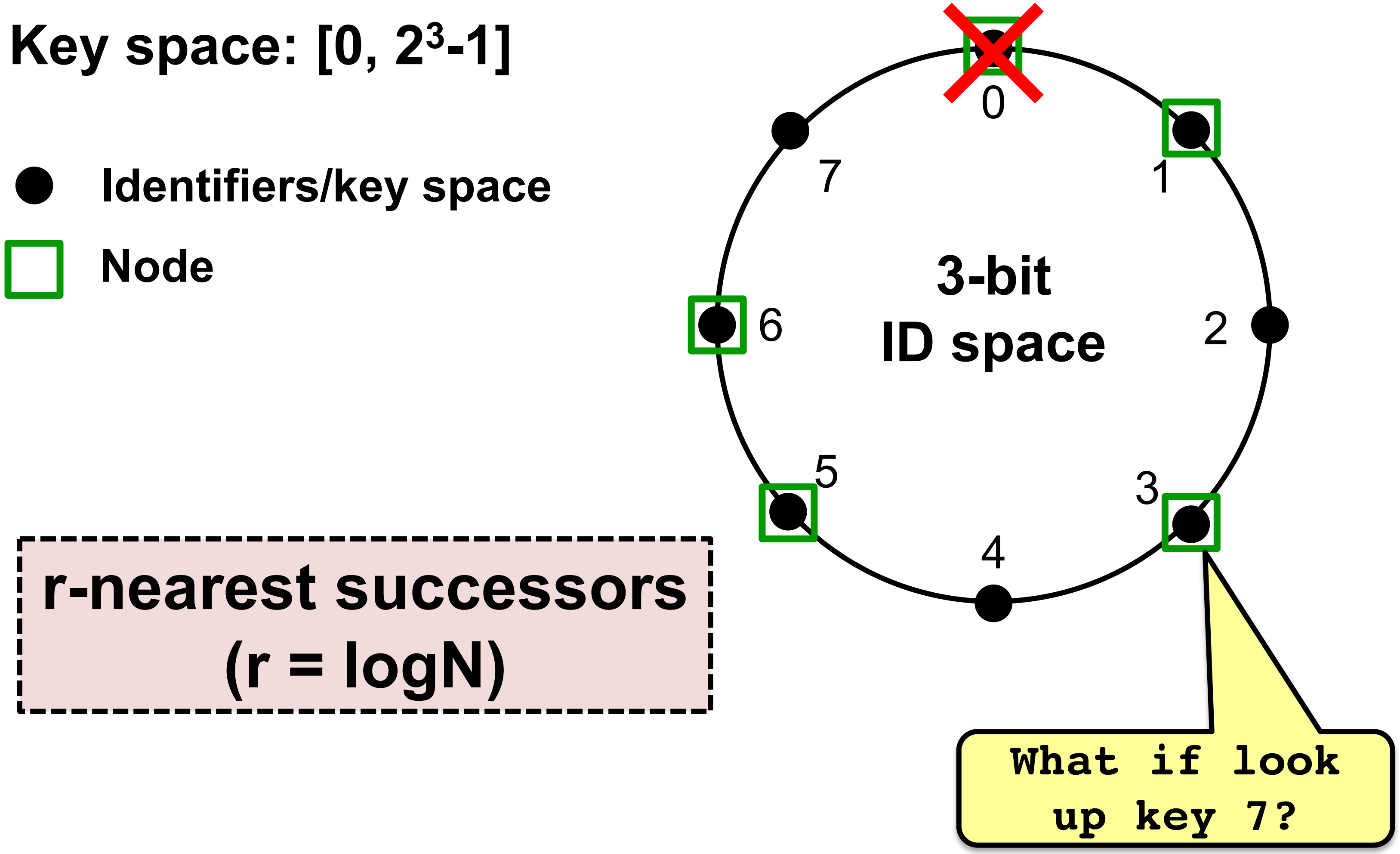
→ **Points to successor**

r-nearest successors (r = logN)



Chord – failures and successor list

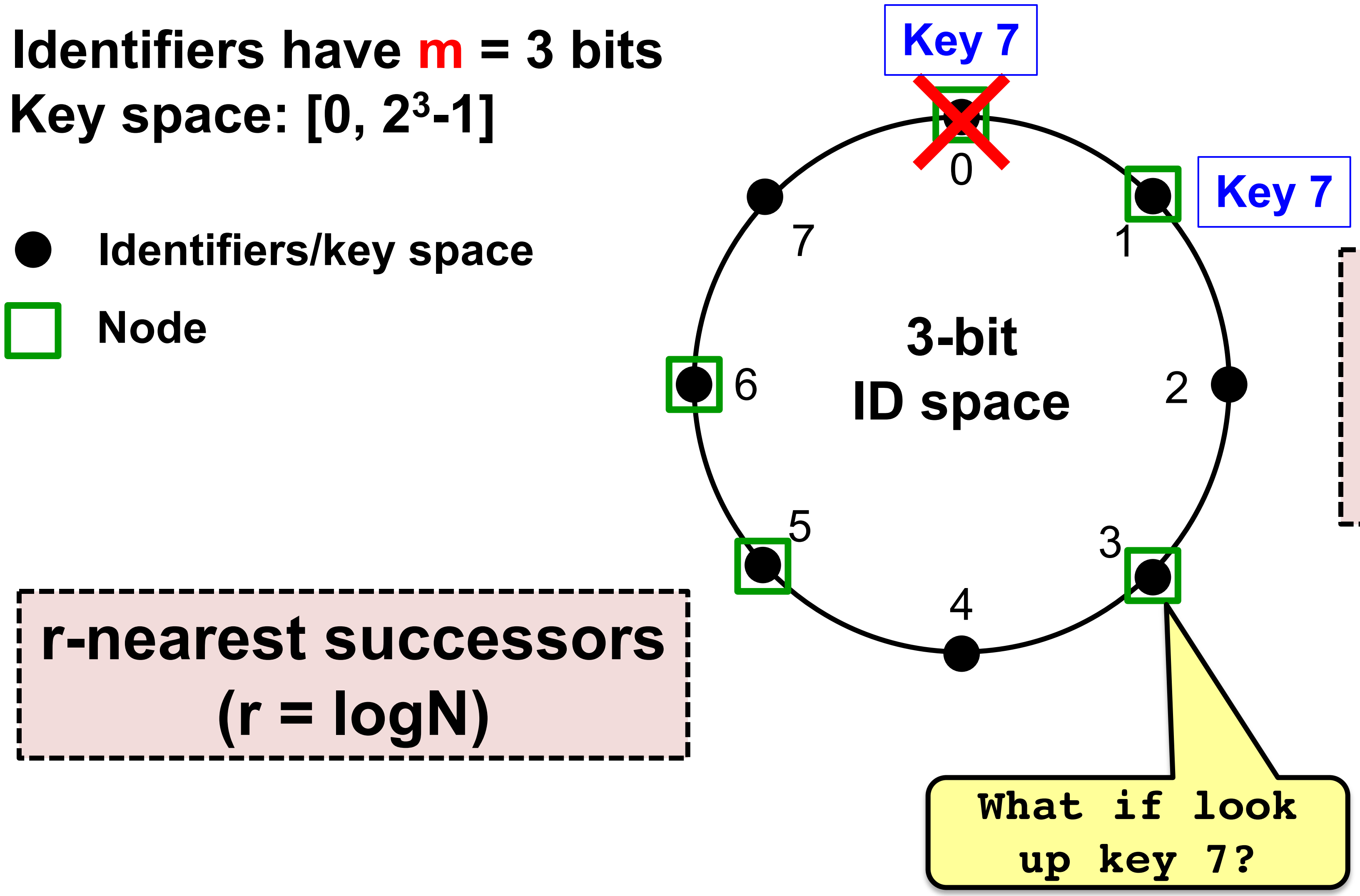
Identifiers have m = 3 bits
Key space: [0, 2^3-1]



r-nearest successors (r = logN)

DHash replicates blocks at *r* successors

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



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

r-nearest successors (r = logN)

Today

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

Why don't all services use P2P?

1. **High latency and limited bandwidth** between peers (vs. intra/inter-datacenter, client-server model)
   1. 1 M nodes = 20 hops; 50 ms / hop gives 1 sec 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

- 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 and other systems
- **Replication** for high availability, efficient recovery
- **Incremental scalability**
  - Peers join with capacity, CPU, network, etc.
- **Self-management:** minimal configuration