Peer-to-Peer Systems and Distributed Hash Tables

COS 418/518: Distributed Systems
Lecture 9 & 10

Jialin Ding, Mike Freedman

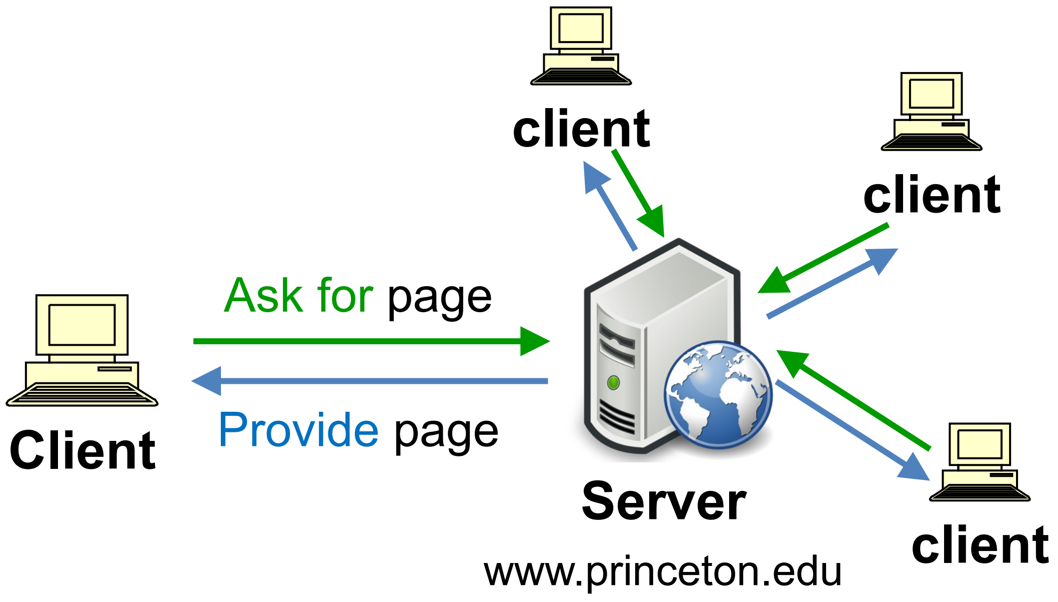
Today

1. Peer-to-Peer Systems

2. Distributed Hash Tables (DHT)

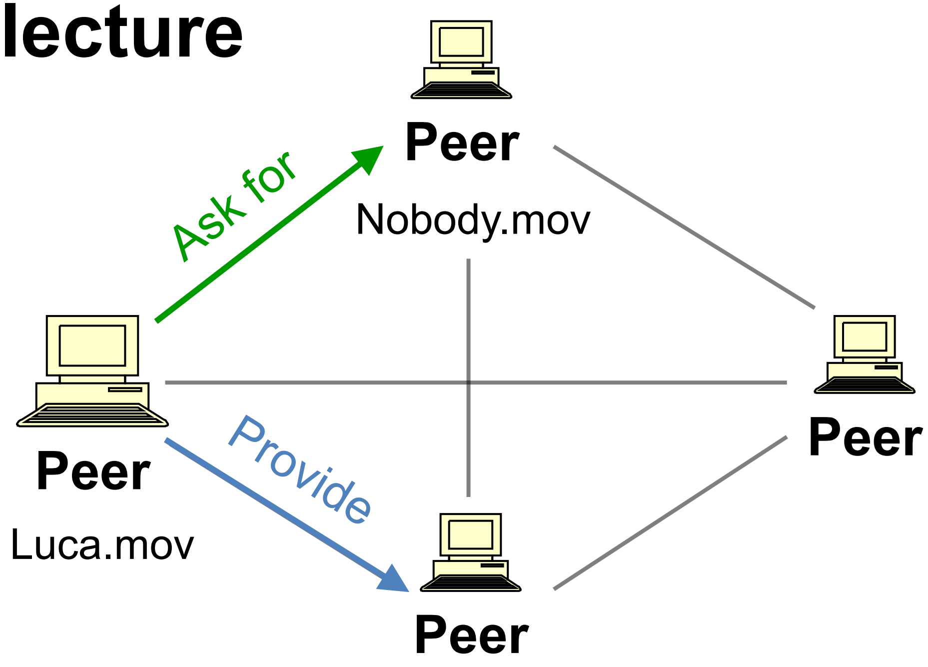
3. The Chord Lookup Service

Distributed Application Architecture



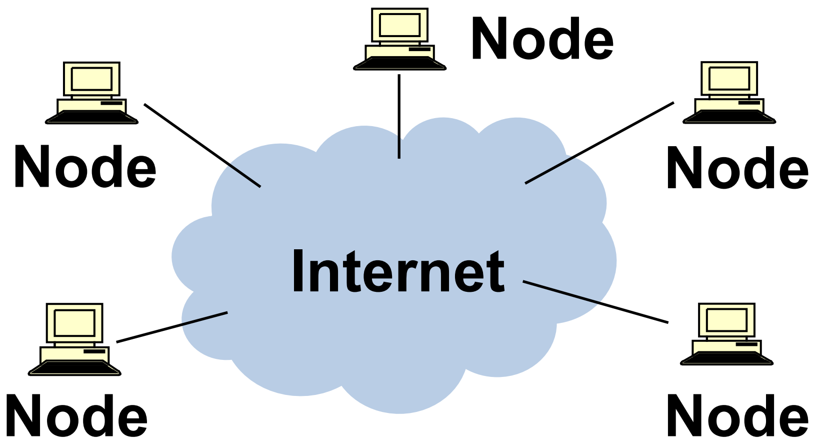
Client-Server

This lecture



Peer-to-Peer

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

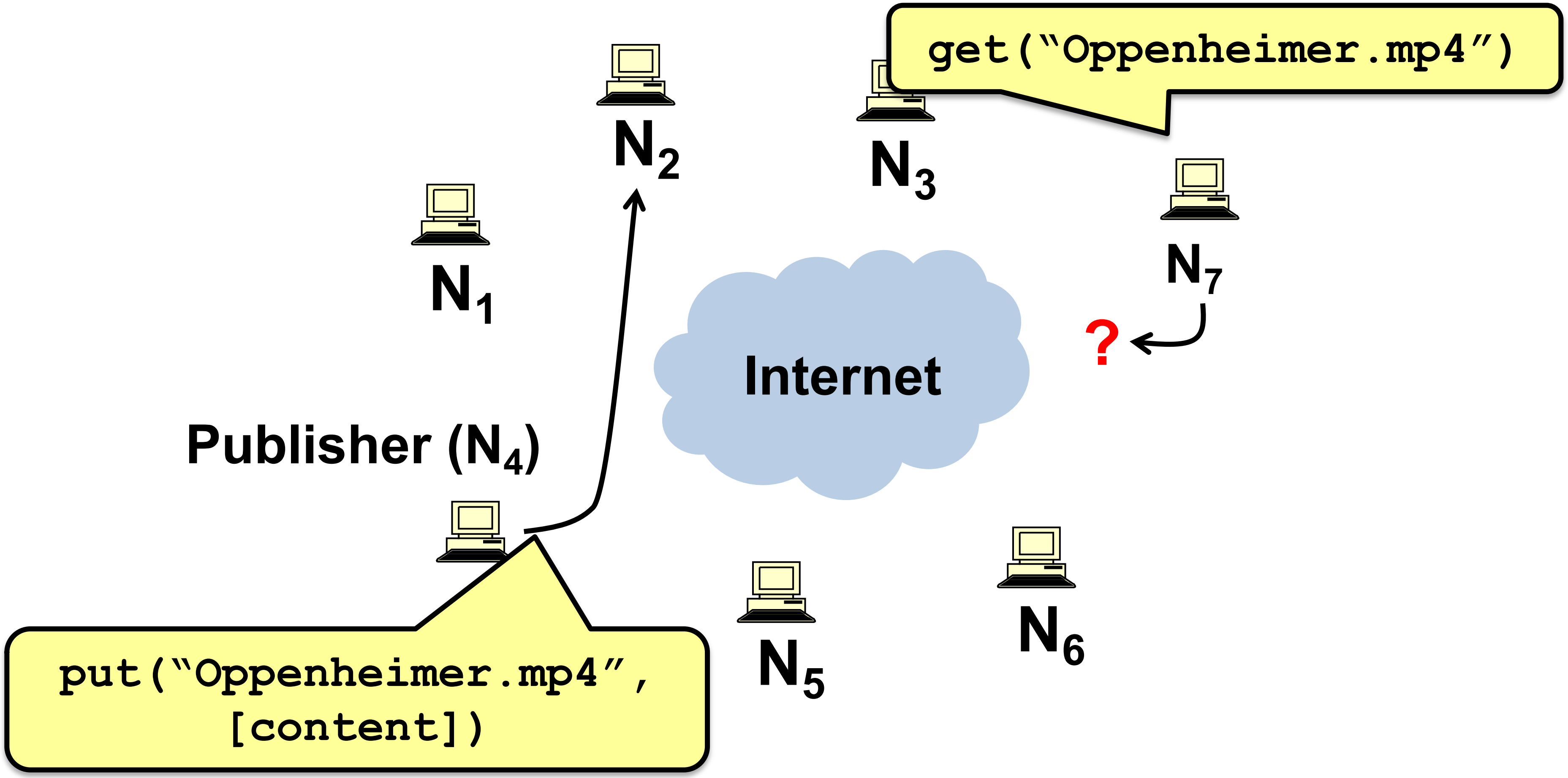
Successful adoption in **some niche areas**

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

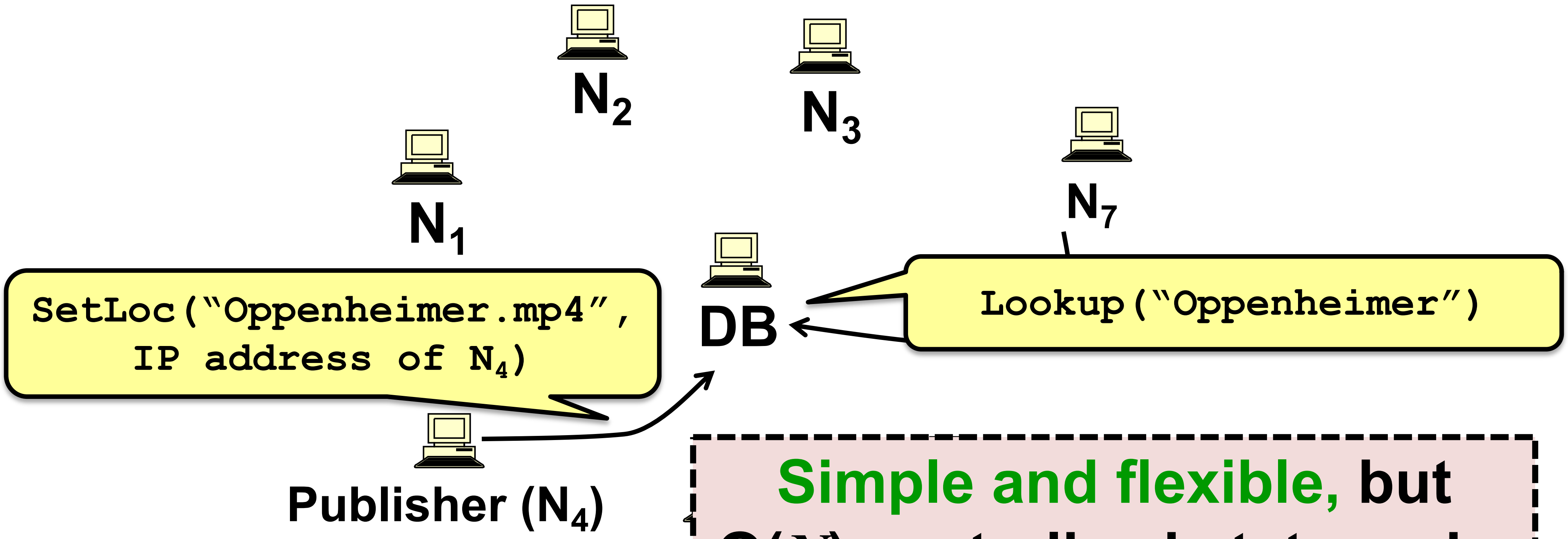
2. **Digital currency:** no natural single owner (Bitcoin)

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

The lookup problem: locate the data



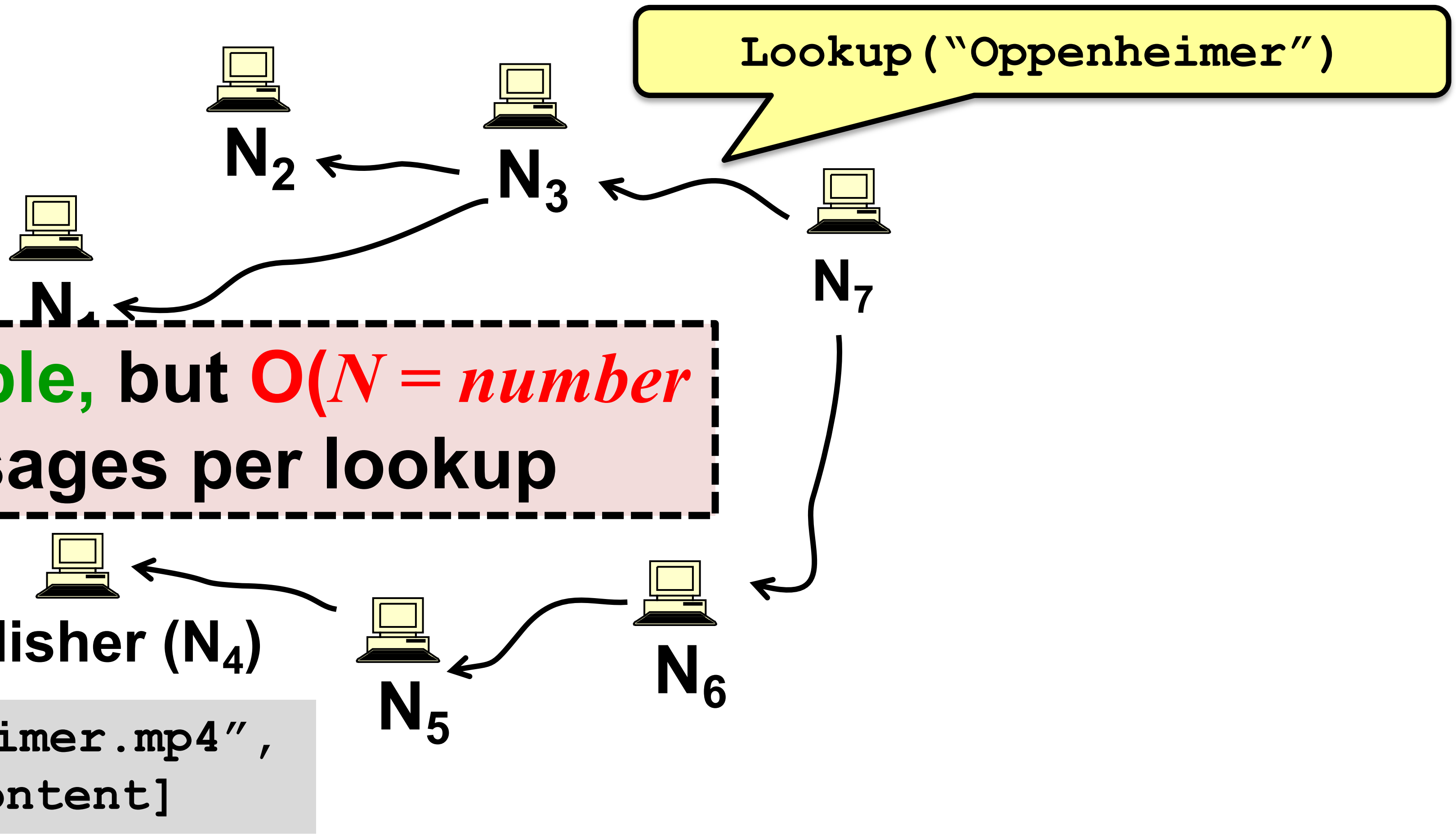
Centralized lookup (Napster)



`key="Oppenheimer.mp4", value=[content]`

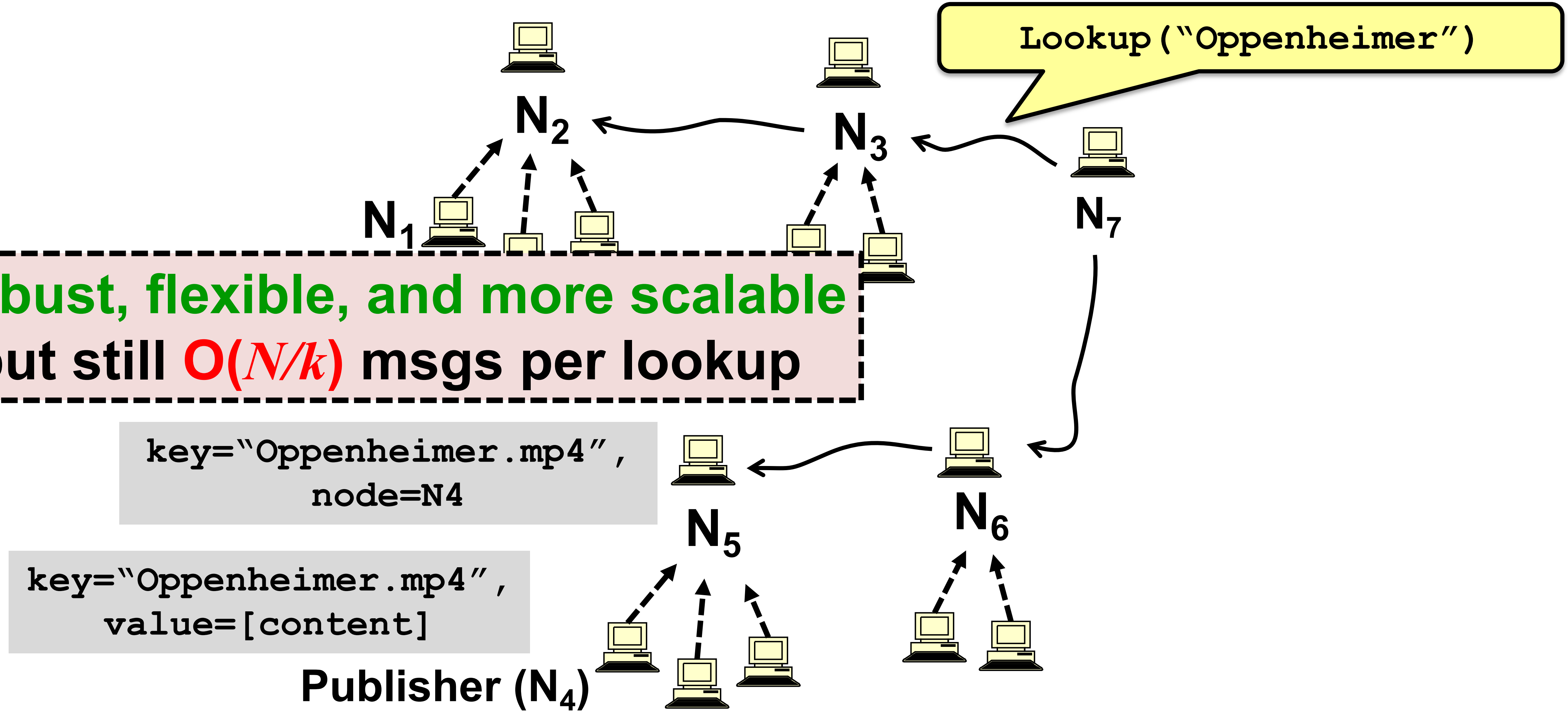
Simple and flexible, but $O(N)$ centralized state and a **single point of failure**

Flooded queries (original Gnutella)



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

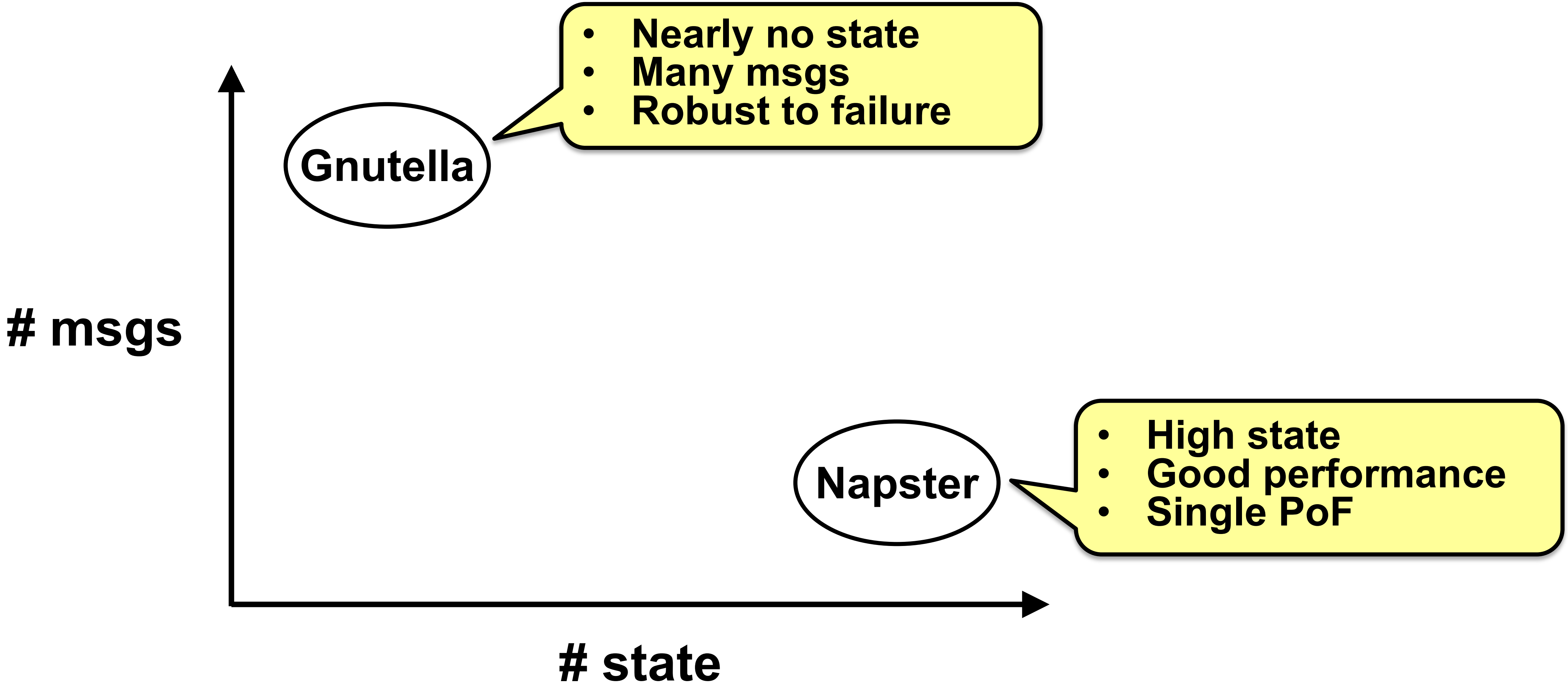
Flooded queries pt 2 (Gnutella w/ SuperPeers)



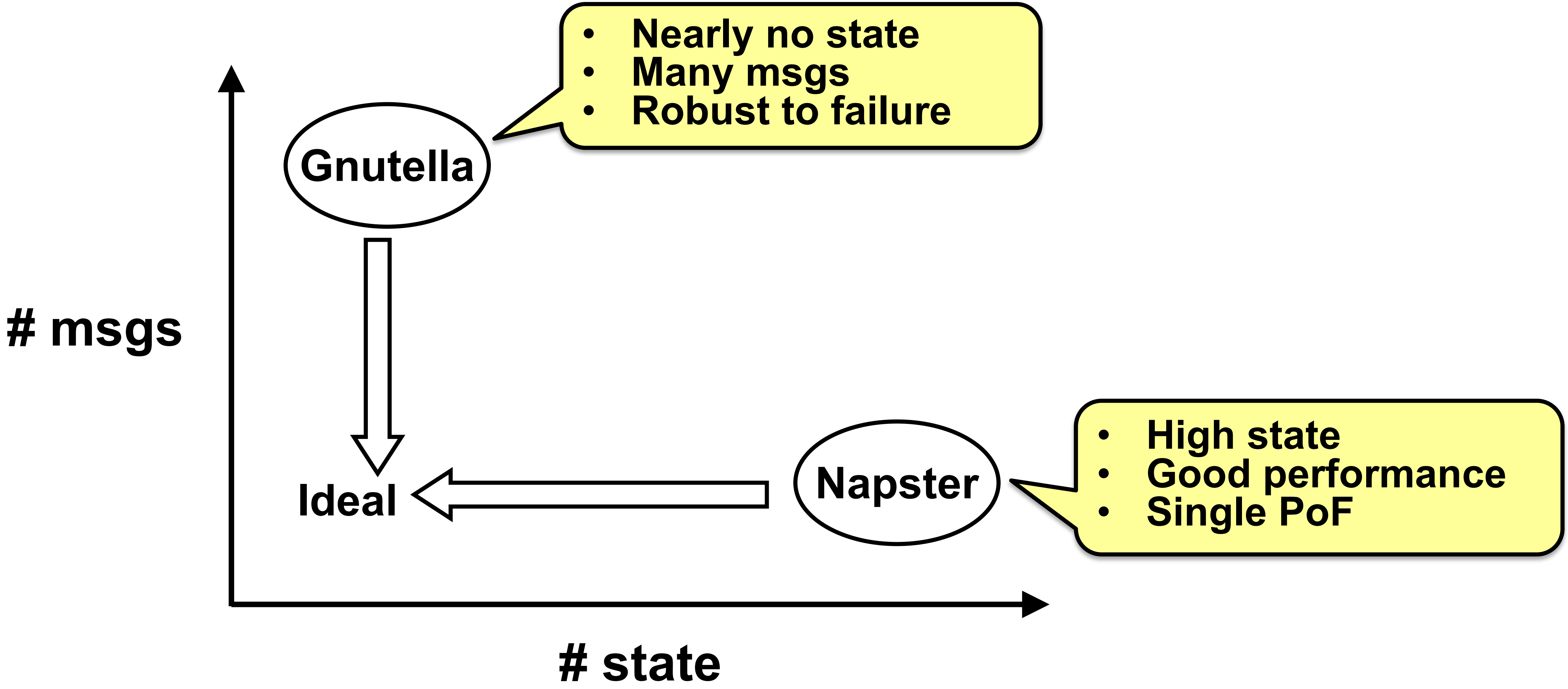
Robust, flexible, and more scalable

but still $O(N/k)$ msgs per lookup

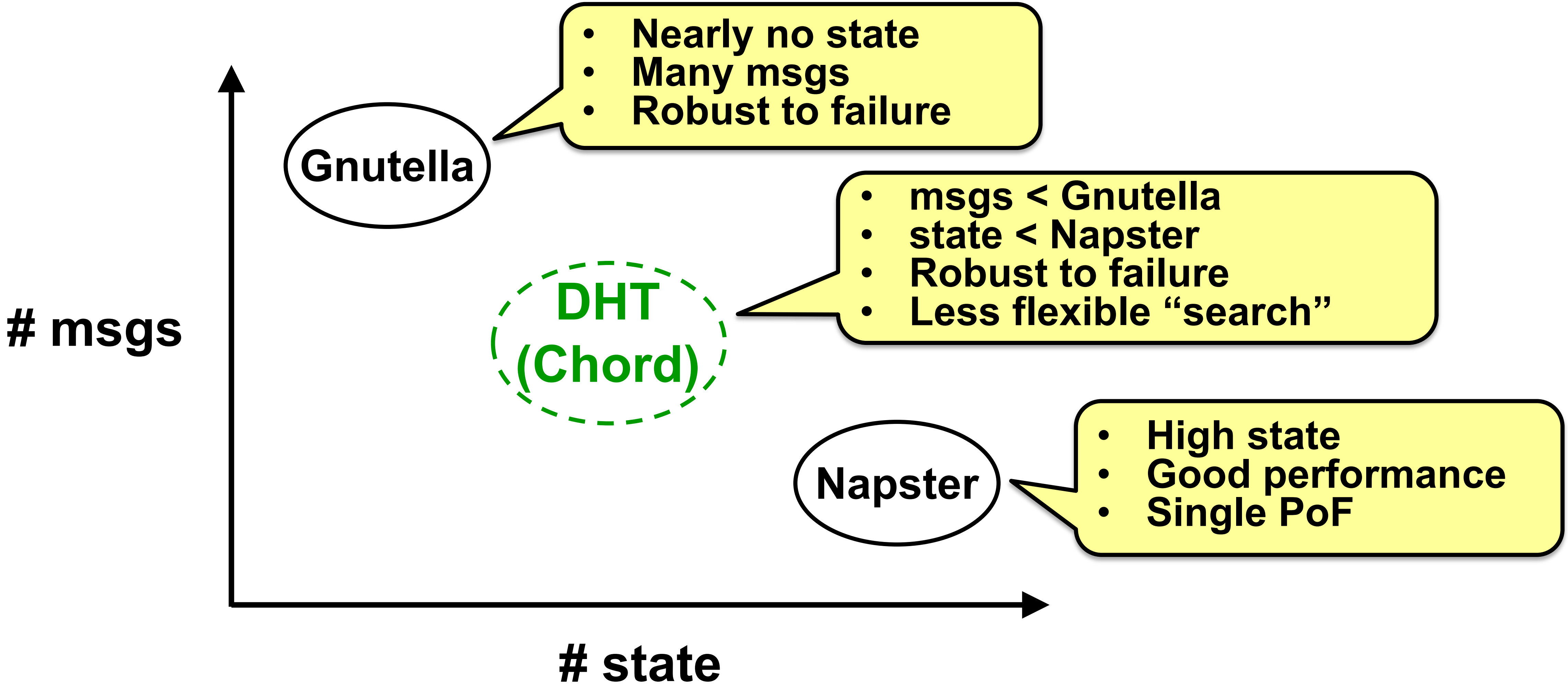
Tradeoffs in distributed systems



Tradeoffs in distributed systems



Tradeoffs in distributed systems



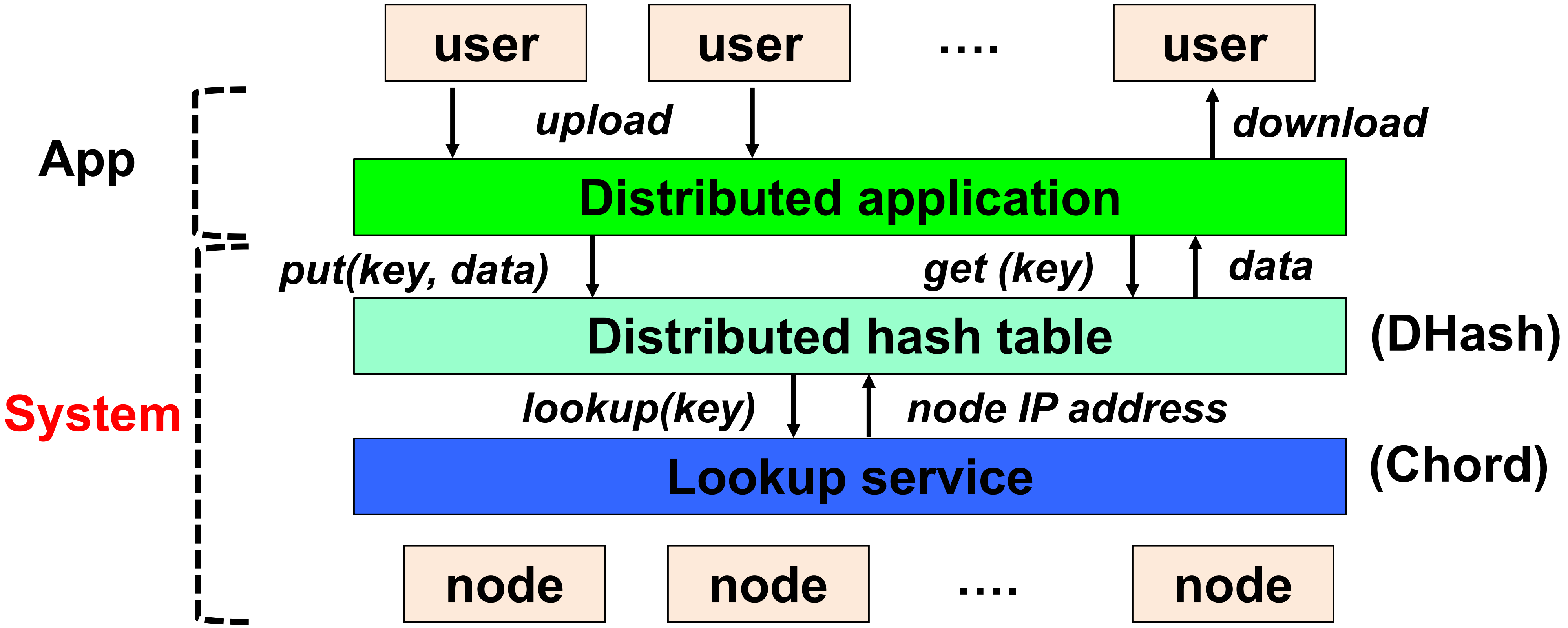
What is a DHT (and why)?

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

```
lookup(key) → IP addr    (Chord lookup service)
send-RPC(IP address, put, key, data_one)
send-RPC(IP address, get, key) → data_one
```

- **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) mod 2^{160}*
  - **Node identifier** = *SHA-1(IP address) mod 2^{160}*

- **What is "SHA-1"?**
  - SHA-1 is a cryptographic hash function that maps input to 160-bit output hash
  - Some properties:
    1. Output hashes looks randomly distributed across output space
    2. Given *hash1*, hard to find *input1* where *SHA1(input1) = hash1*
    3. Given *input1* and *hash1*, hard to find *input2* where *SHA1(input2) = hash1*
    4. Hard to find *input1* and *input2* where *SHA1(input1) = SHA1(input2*)

Chord identifiers

- **Hashed values (integers) using the same hash function**
  - **Key identifier** = *SHA-1(key) mod 2^{160}*
  - **Node identifier** = *SHA-1(IP address) mod 2^{160}*

- **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

Alternative: mod (n) hashing

- **System of n nodes: 1…n**
  - Node that owns key is assigned via *hash(key) mod n*
  - Good load balancing
- **What if a node fails?**
  - Instead of n nodes, now *n -1* nodes
  - Mapping of all keys change, as now *hash(key) mod (n-1)*

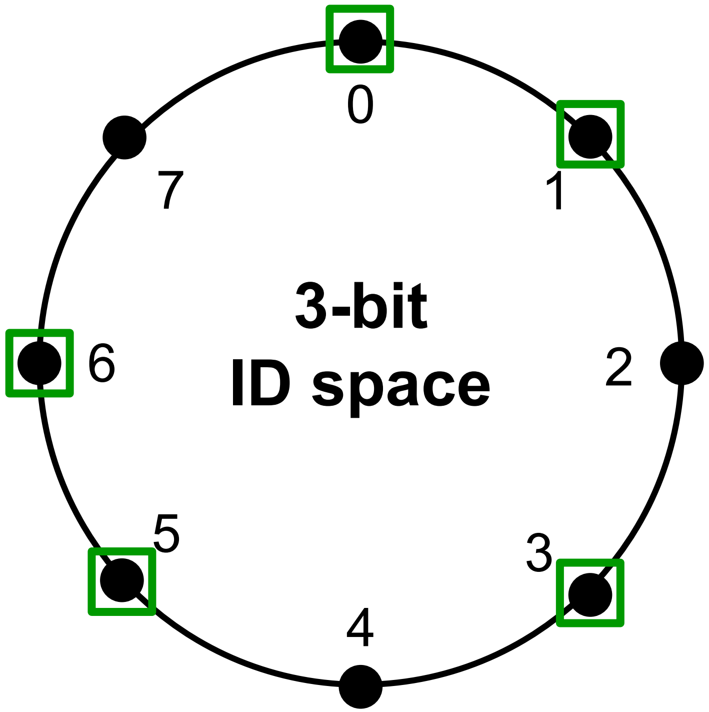
- **N = 5**
  - 12594 mod 5 = 4
  - 28527 mod 5 = 2
  - 816 mod 5 = 1
  - 716565 mod 5 = 0
- **N = 4**
  - 12594 mod 4 = **2**
  - 28527 mod 4 = **3**
  - 816 mod 4 = **0**
  - 716565 mod 4 = **1**

Consistent hashing [Karger '97] Data partitioning

Identifiers have m = 3 bits

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

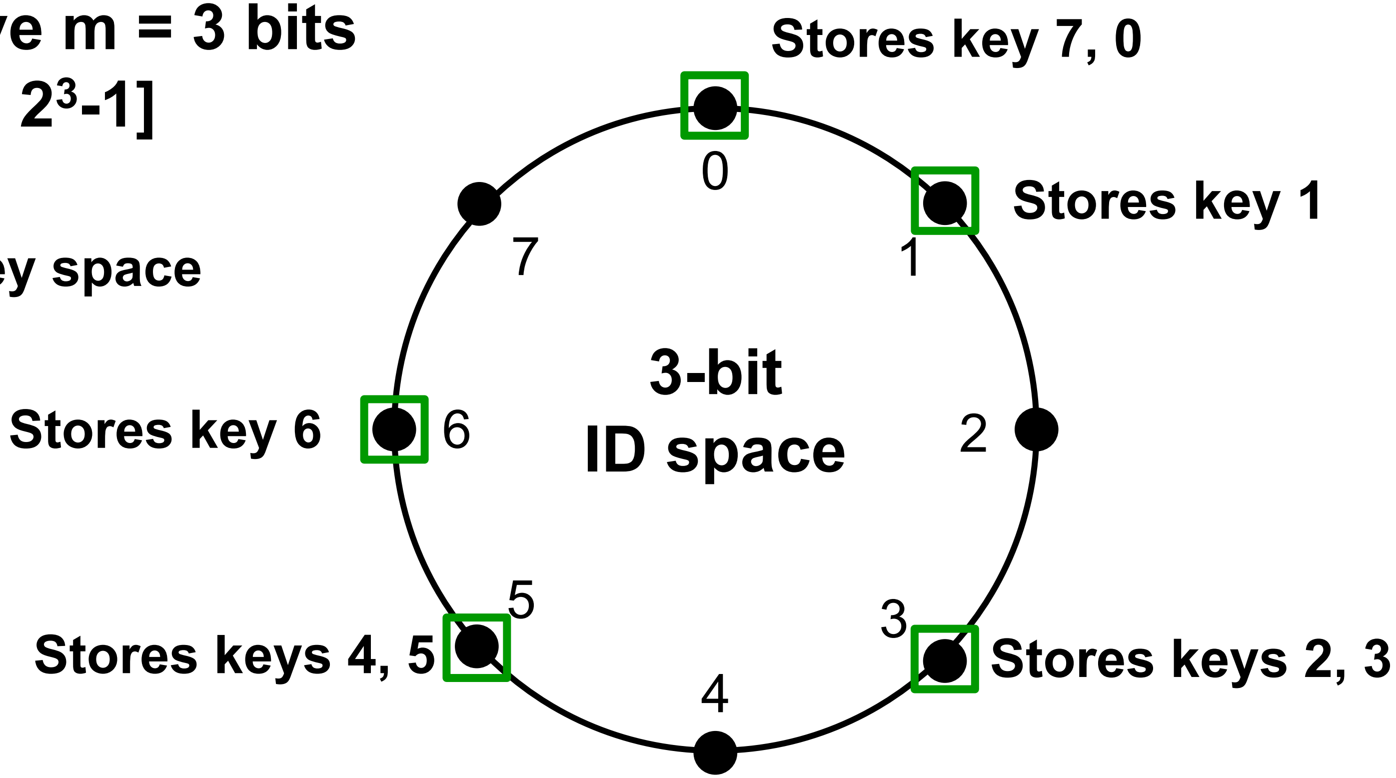
- Identifiers/key space
- Node



Consistent hashing [Karger '97]
Data partitioning

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

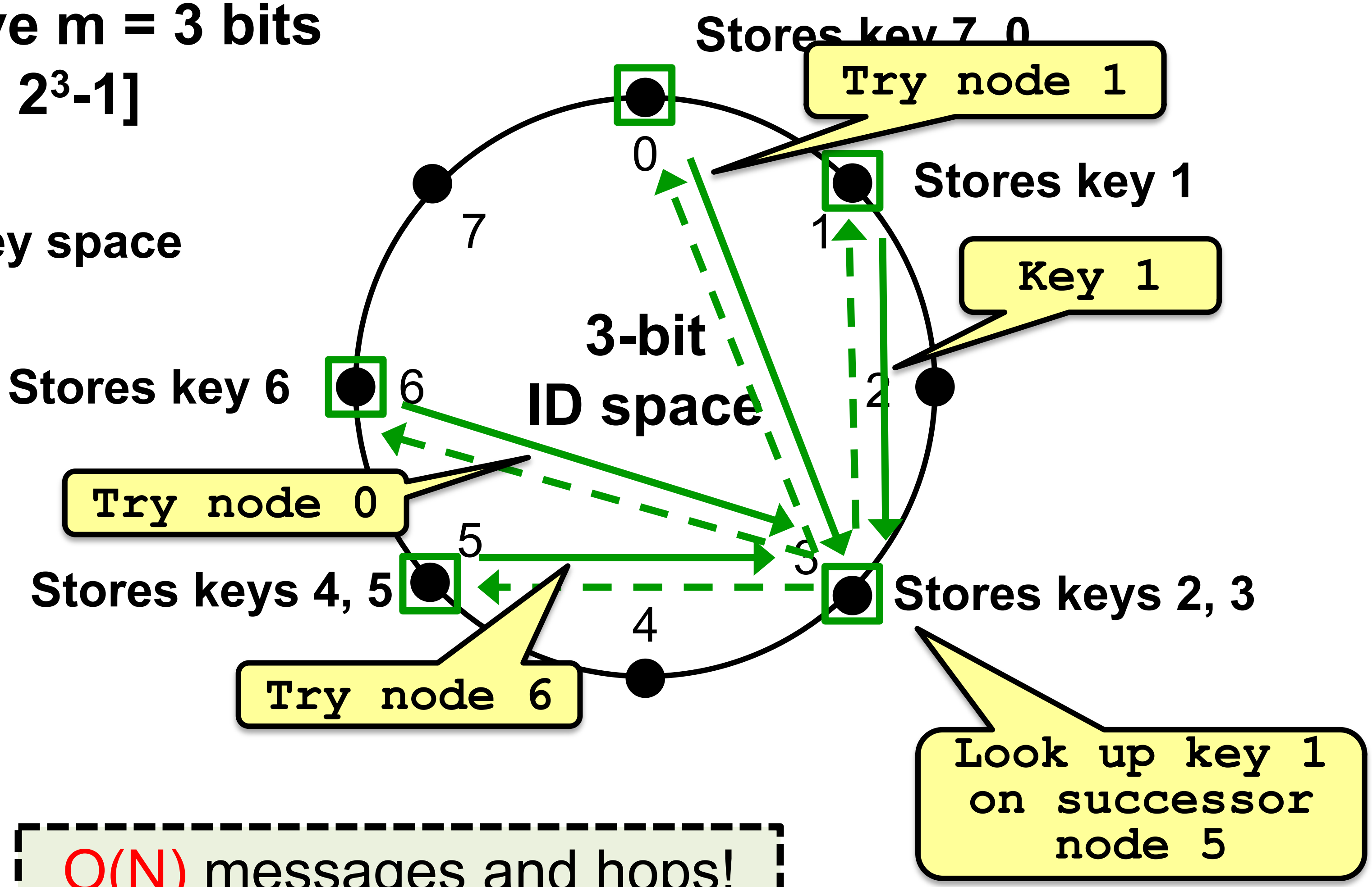
Consistent hashing [Karger '97]
Strawman lookup vis successors

Identifiers have m = 3 bits

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

● Identifiers/key space

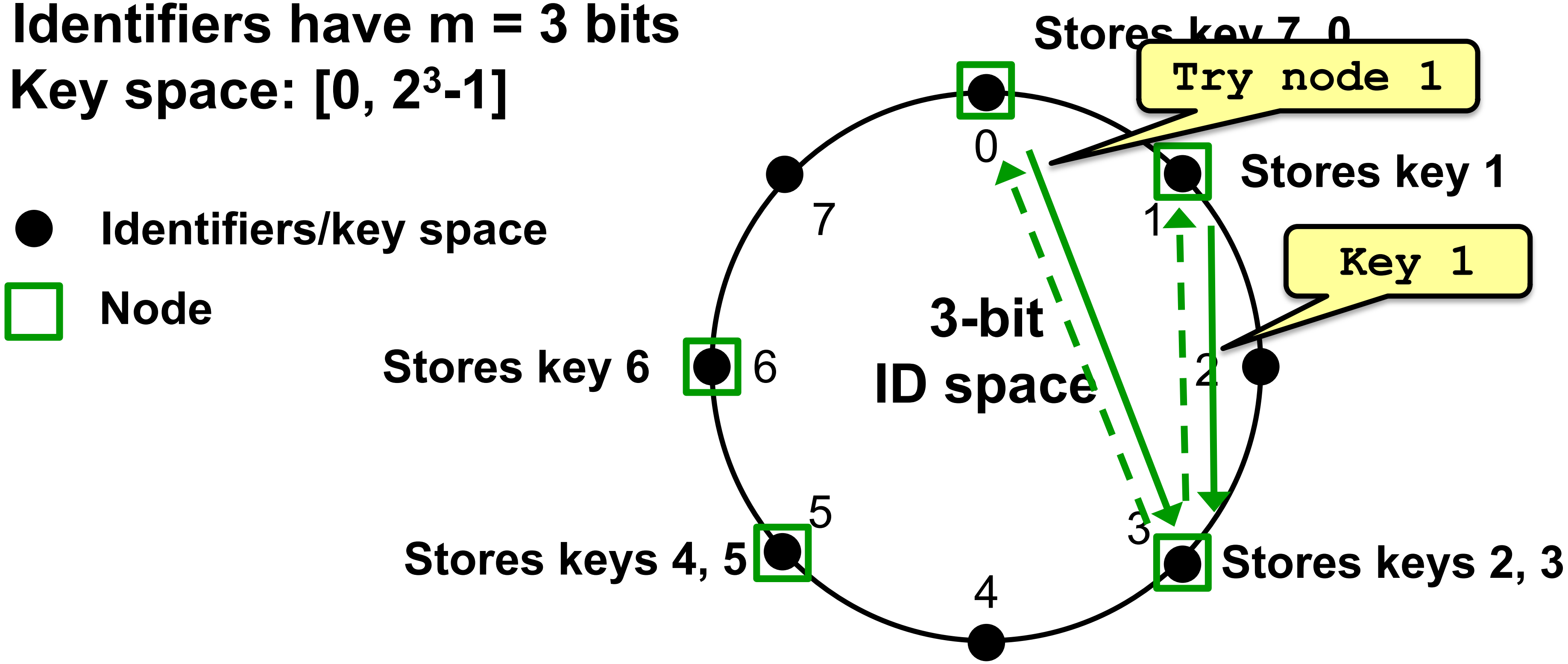
□ Node



O(N) messages and hops!

Consistent hashing [Karger '97]
Observation about last hop

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



Try to find key's predecessor node as fast as possible.
This precedessor will know key's successor (owner).

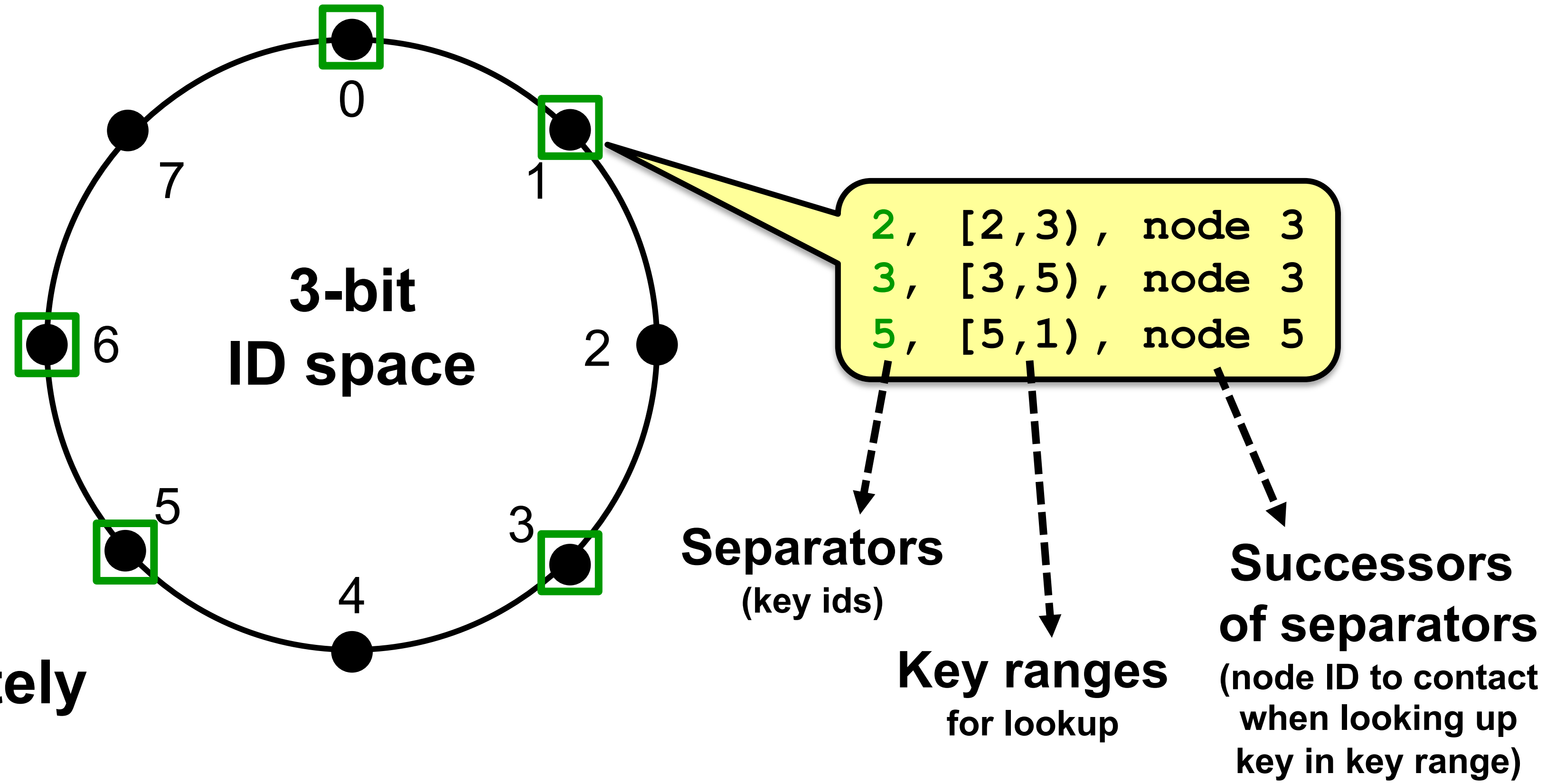
Chord – finger tables for *find_predecessor*

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

Example for node N = 1:

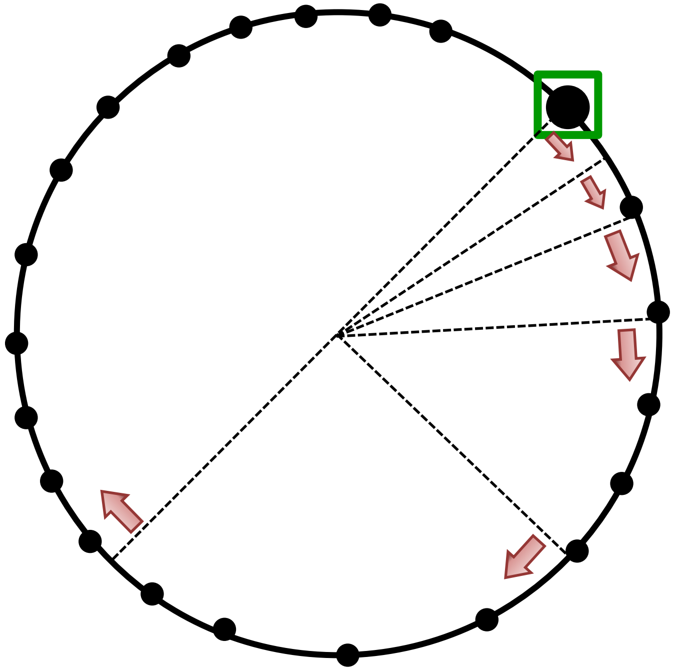
- $\boxed{1} + \boxed{1}$ mod 8 => 2
- $\boxed{1} + \boxed{2}$ mod 8 => 3
- $\boxed{1} + \boxed{4}$ mod 8 => 5

N 2^{k-1}



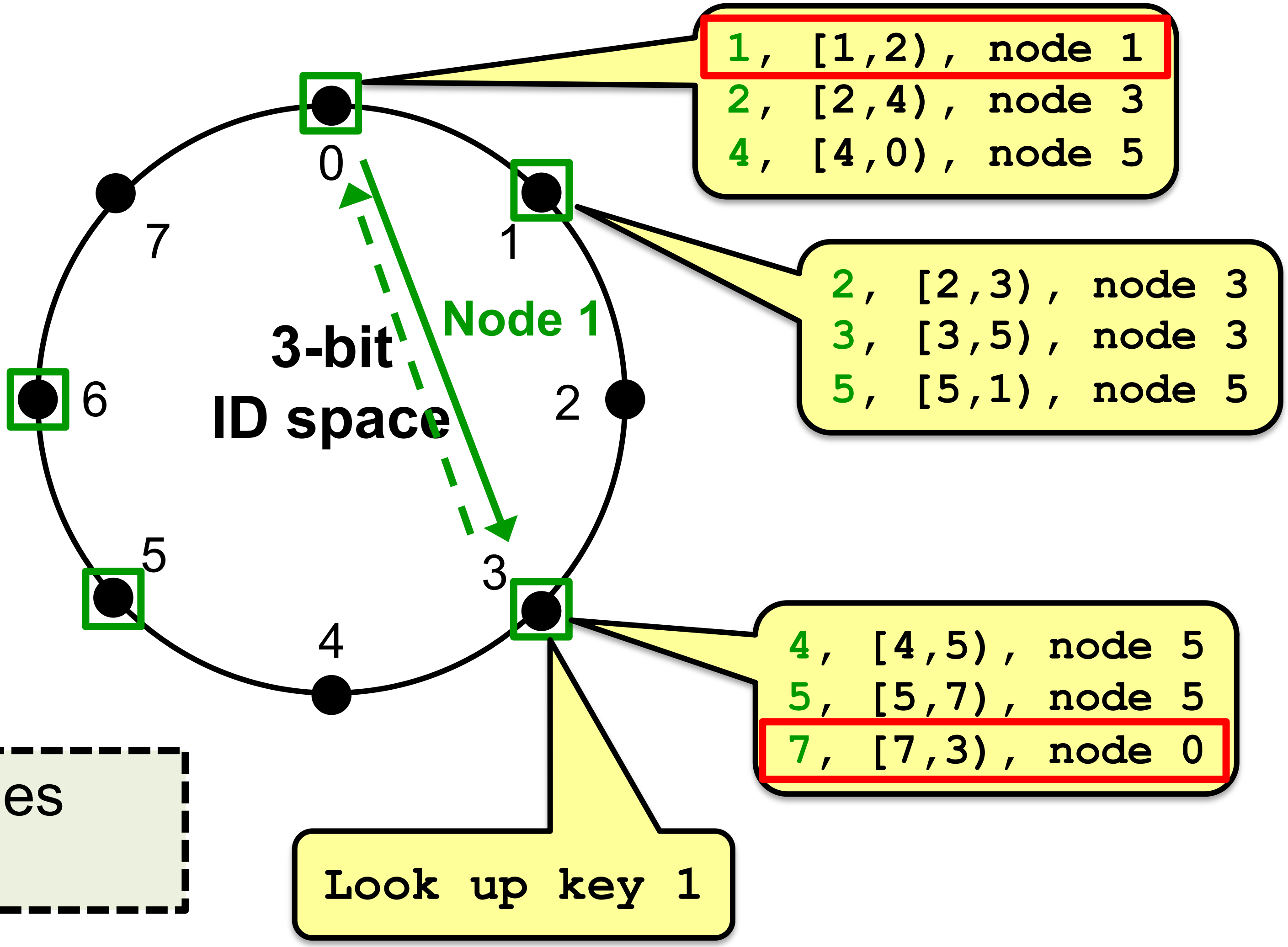
"Finger" is node immediately succeeding separator

Chord – finger tables for *find_predecessor*



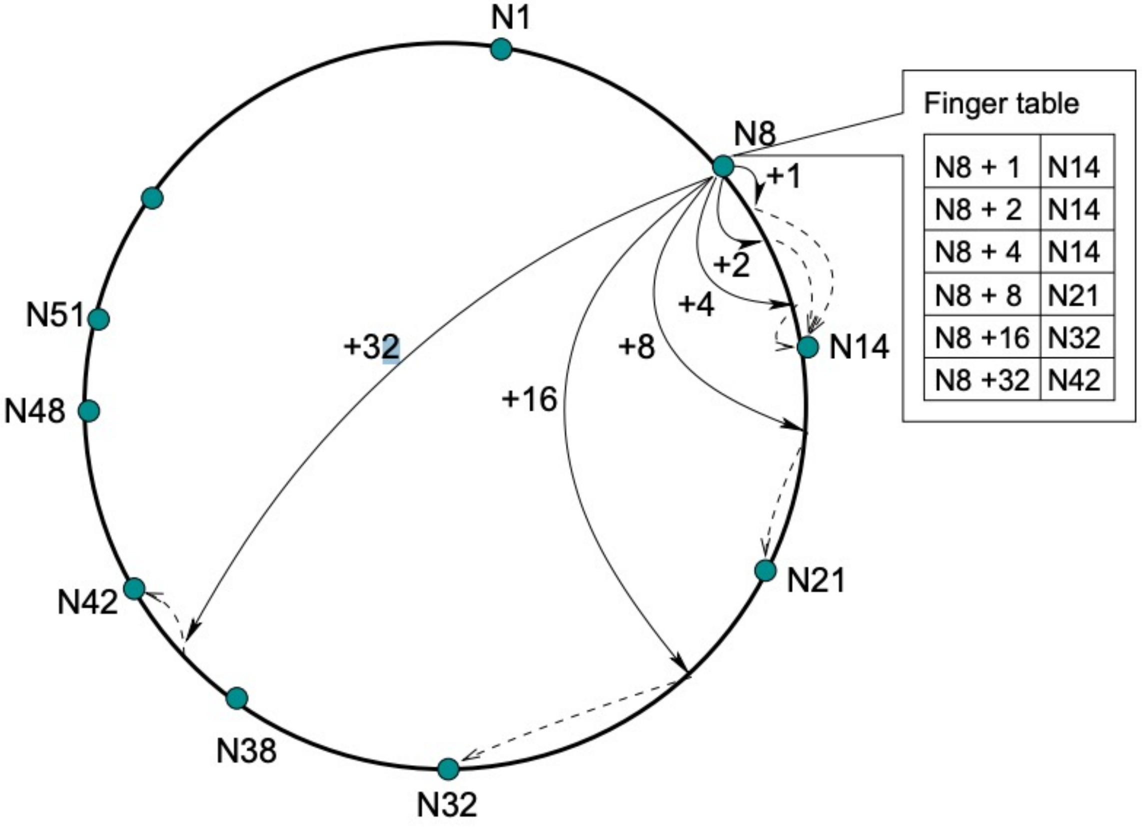
Chord – finger tables for *find_predecessor*

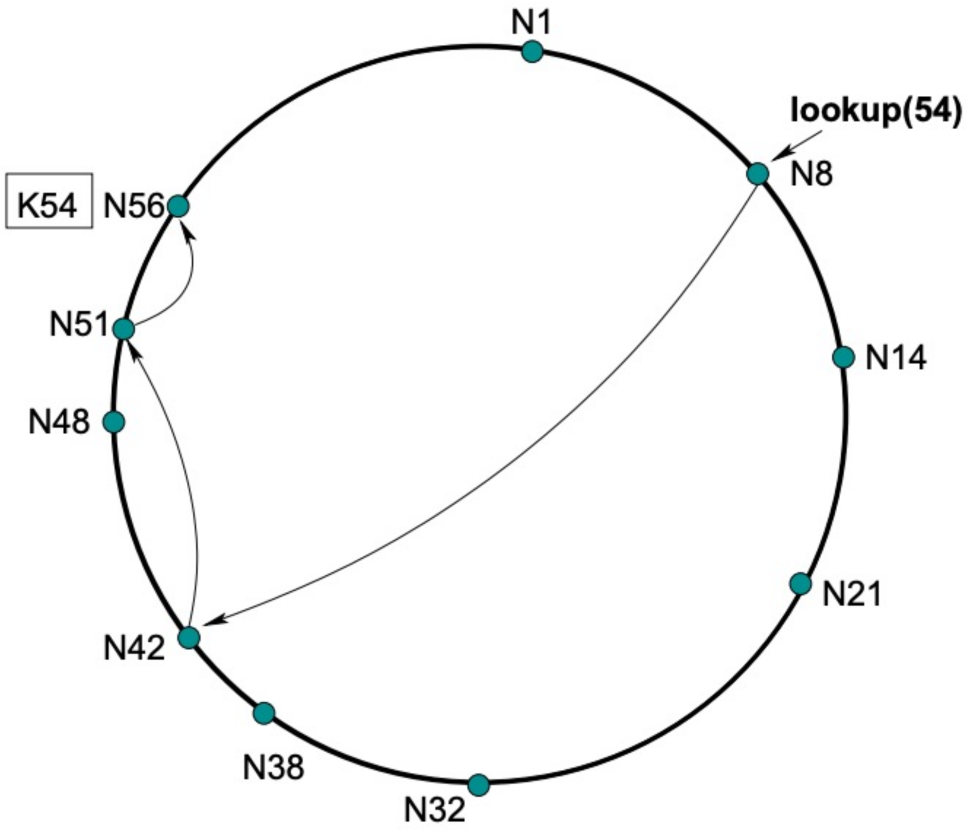
Each node keeps m states
Key space → m ranges via
$(N+2^{k-1}) \bmod 2^m$, $1<=k<=m$



O(logN) messages
and hops!

Chord – finger tables





From Chord ToN paper

Implication of finger tables

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

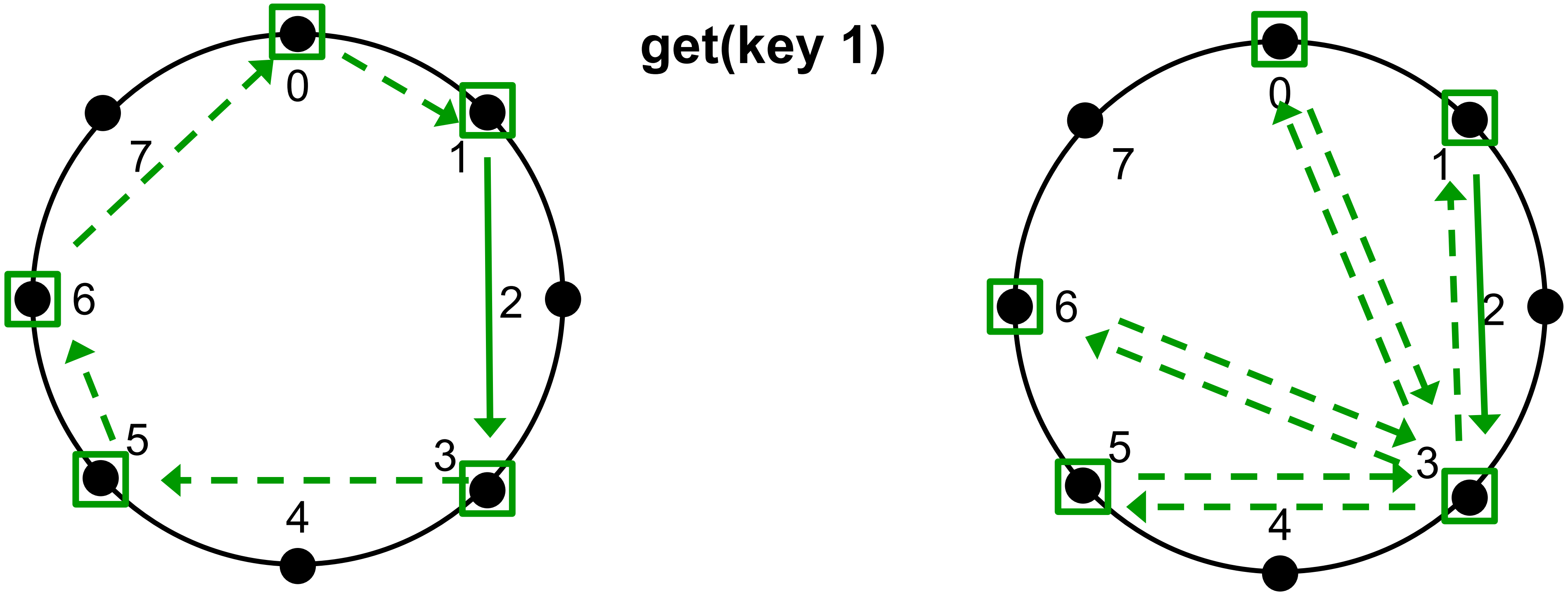
- Better than arranging nodes in a single tree
  - Every node acts as a root
    - So there's **no root hotspot**
    - **No single point** of failure
    - But a **lot more state** in total: N nodes each have O(log N)

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 – Recursive vs. Iterative Lookup

get(key 1)



Recursive Lookup

Iterative Lookup

System Dynamics

- Handling node joins
- Handling node failures
  - Rebuilding lookup structures
  - Ensure data durability

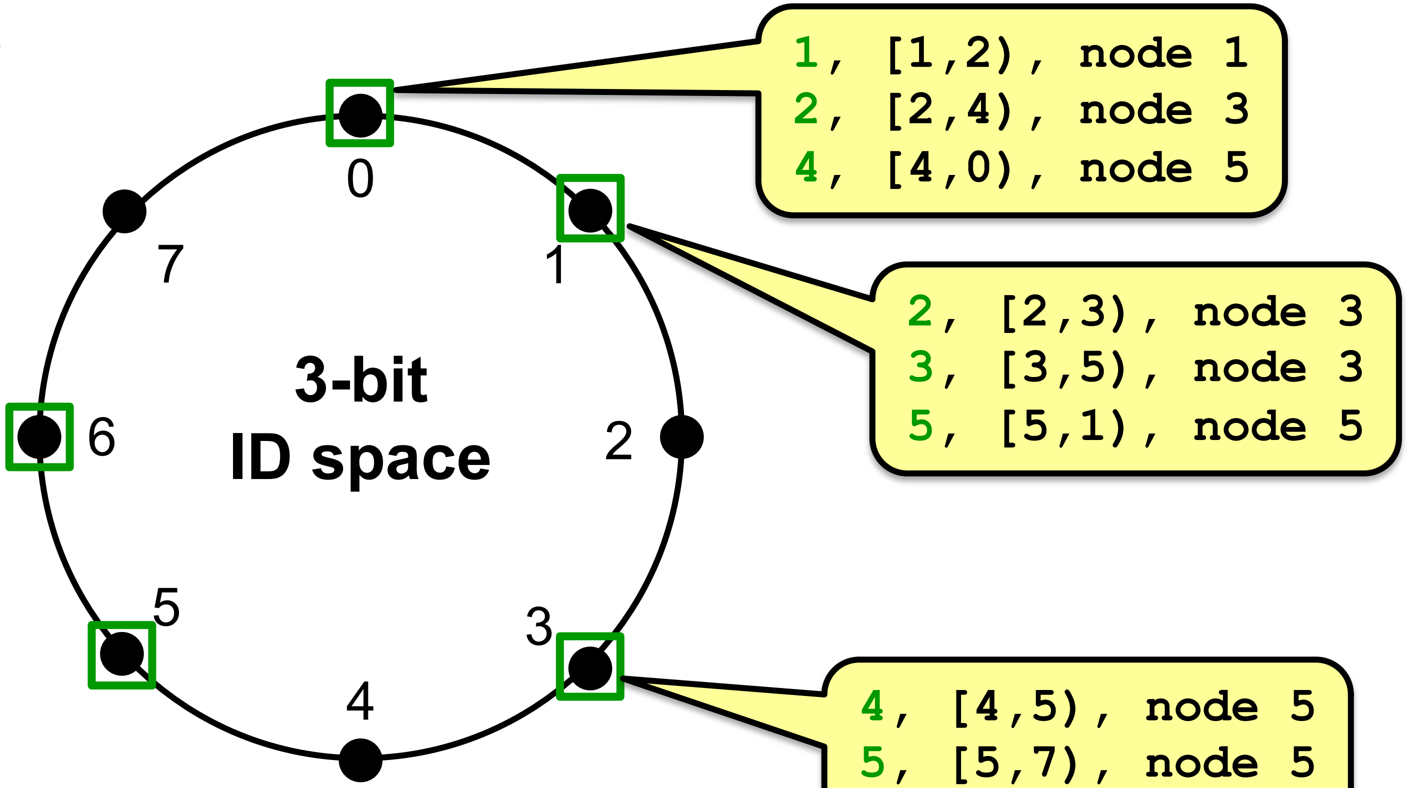
Chord – finger tables

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

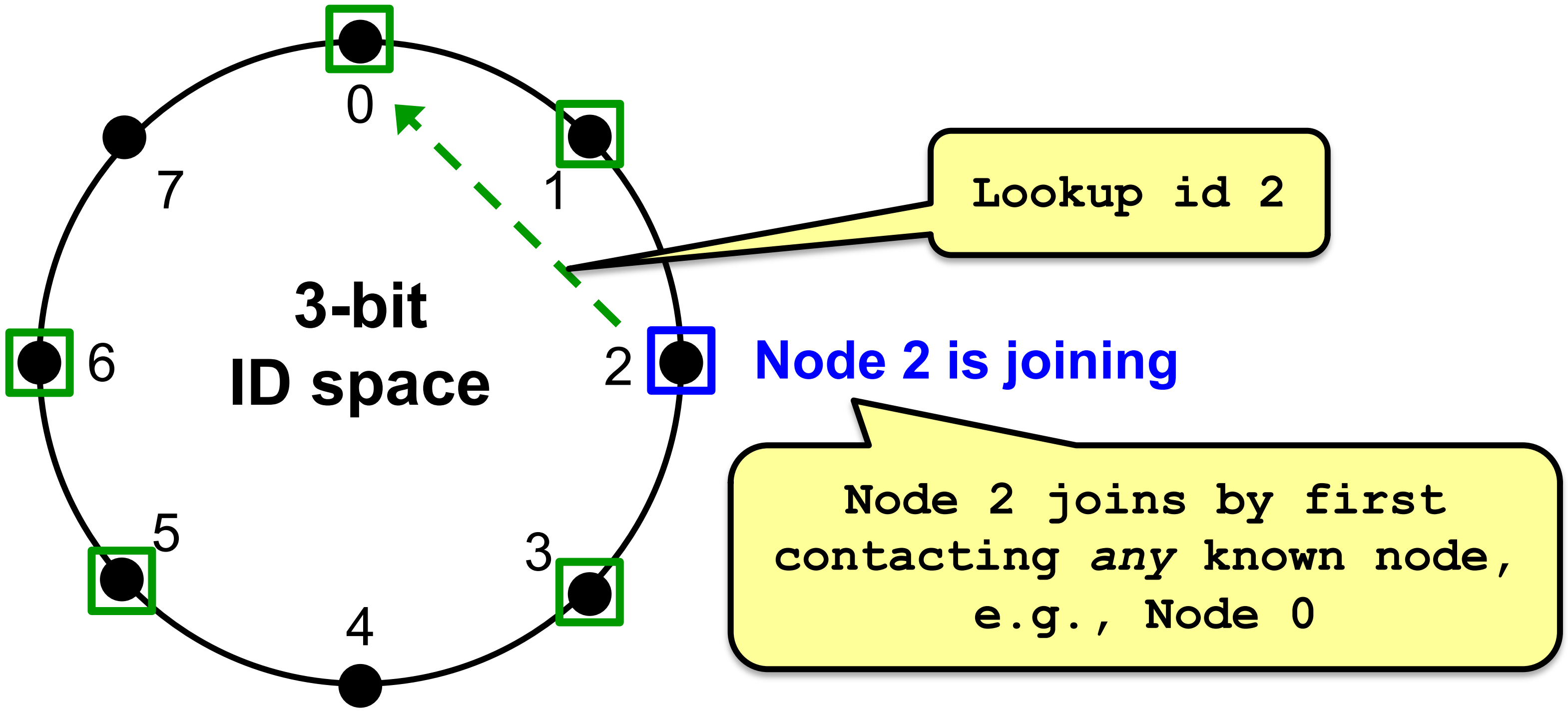
"Finger" is node immediately succeeding separator



Chord – node joining

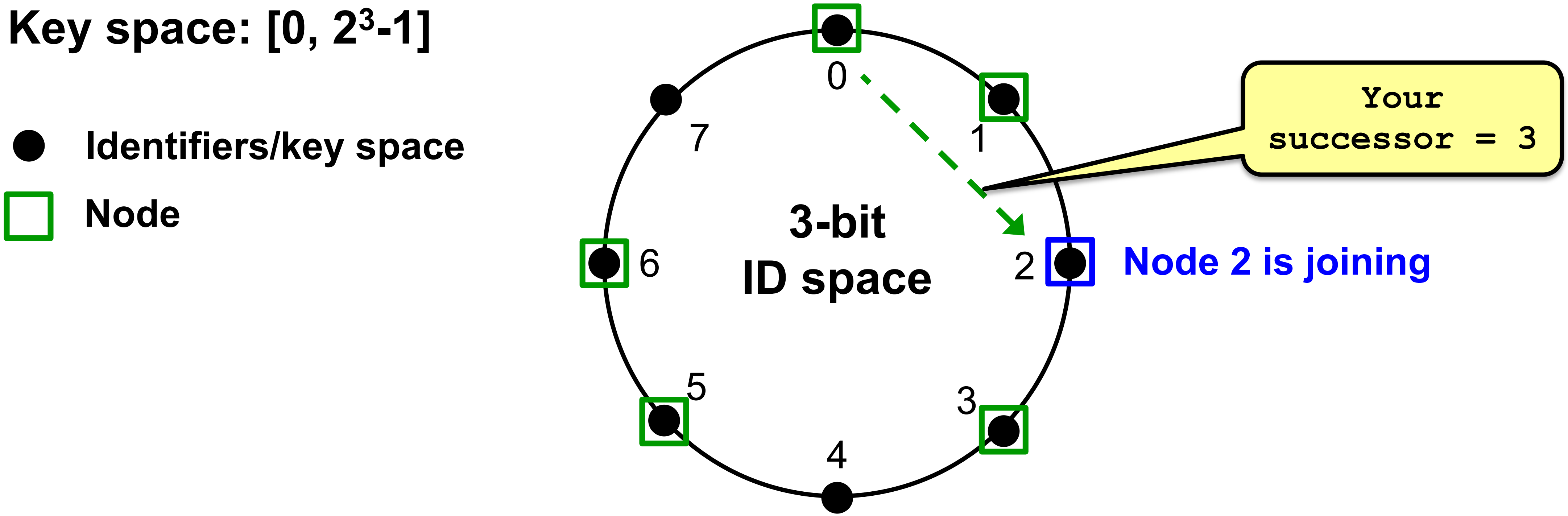
Identifiers have m = 3 bits
Key space: [0, 2^3-1]

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



Chord – node joining

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

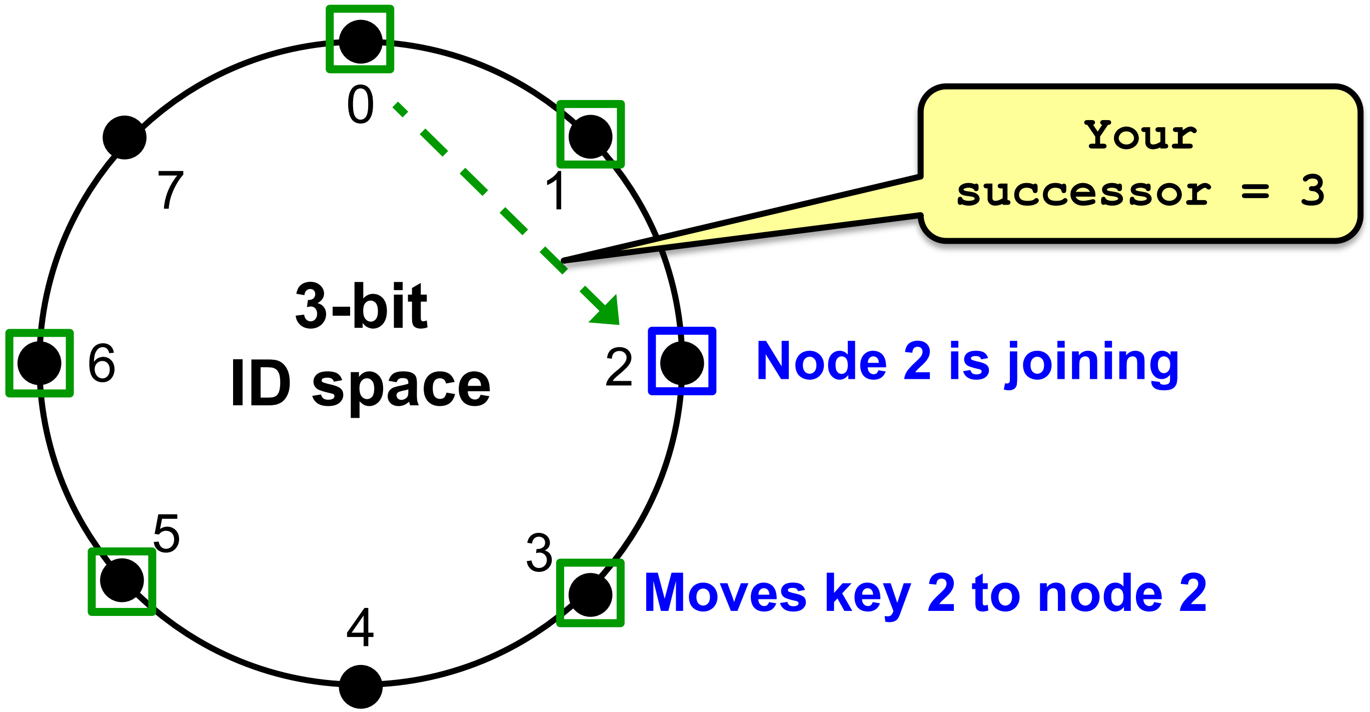


Chord – node joining

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

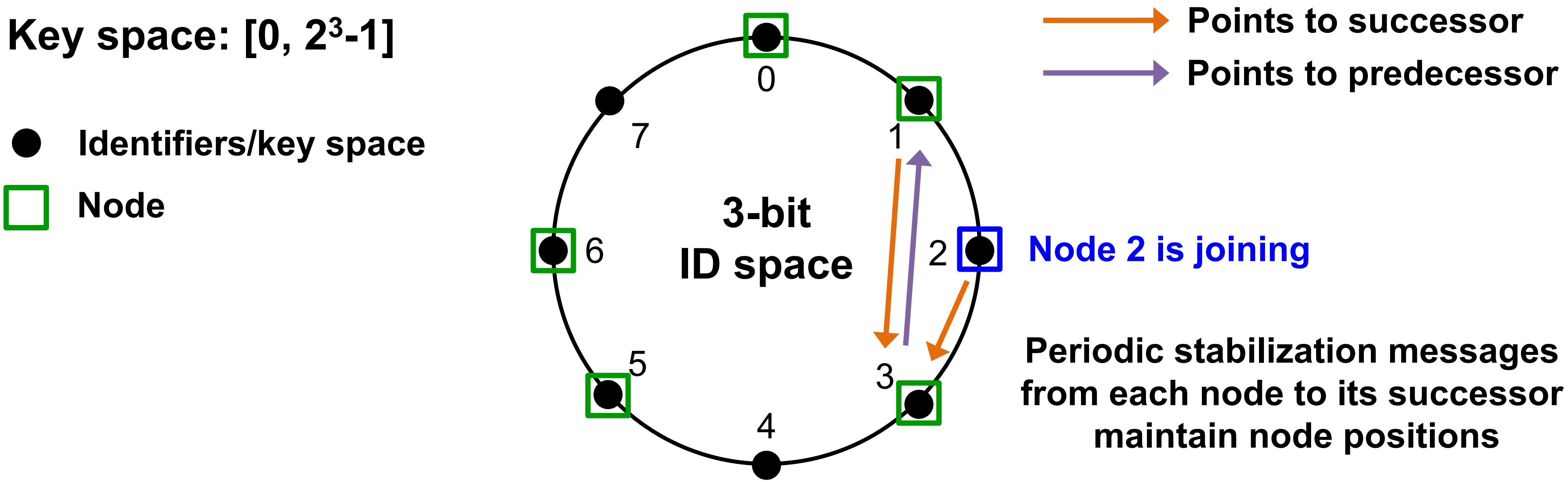
● **Identifiers/key space**

□ **Node**



Chord – node joining

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

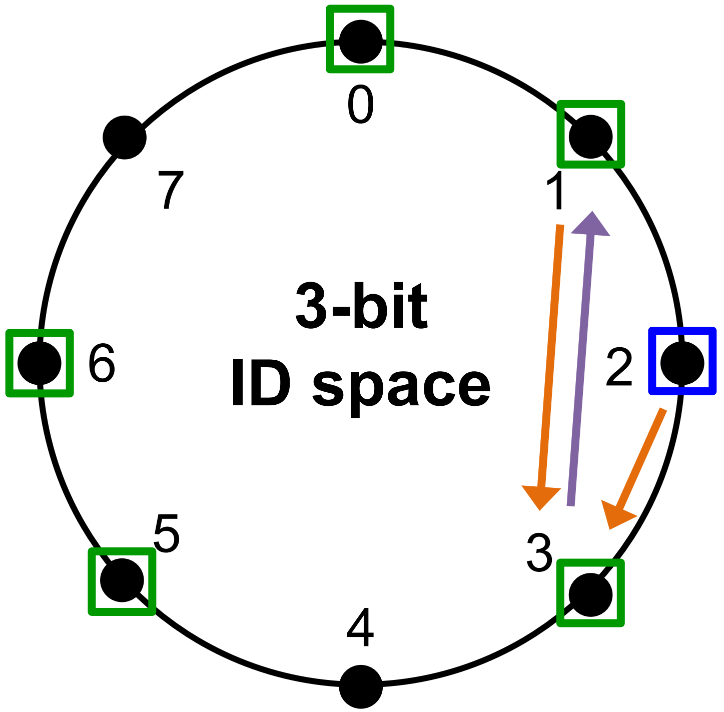


Periodic stabilization messages from each node to its successor maintain node positions

Chord – node joining

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

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



Points to successor
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
```

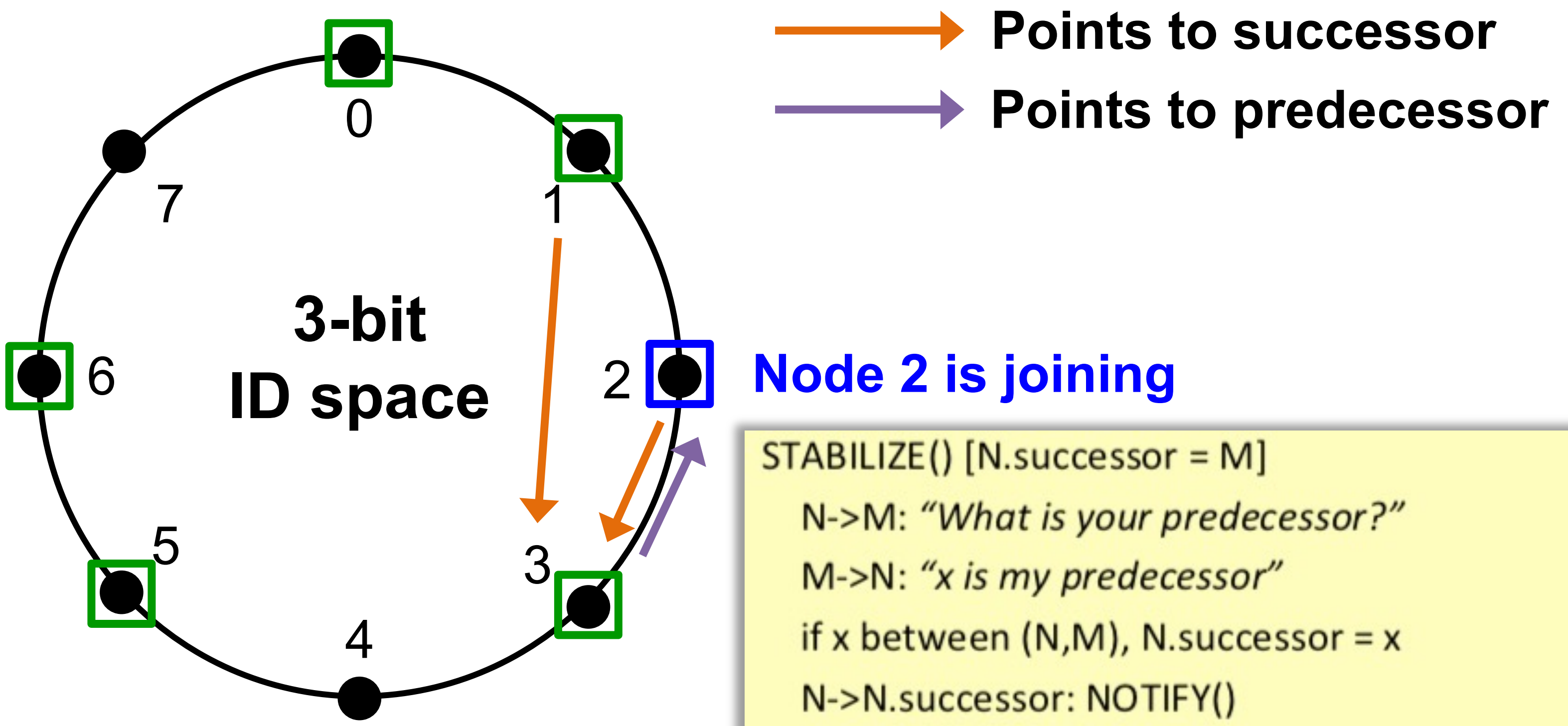
Pseudocode from Rodrigo Fonseca's lecture notes

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:

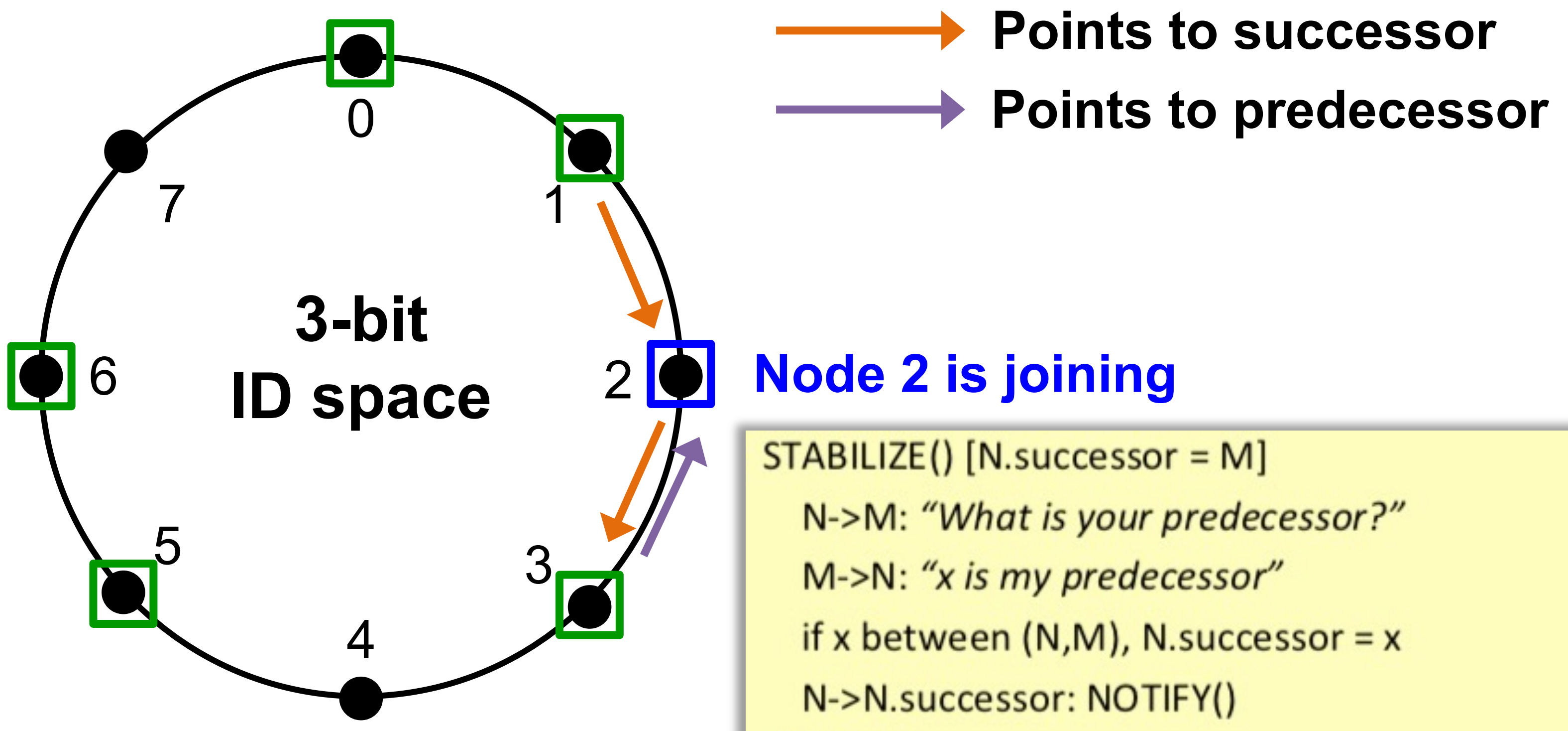
    If (N between (M.predecessor, M))

        M.predecessor = N

Chord – node joining

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

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

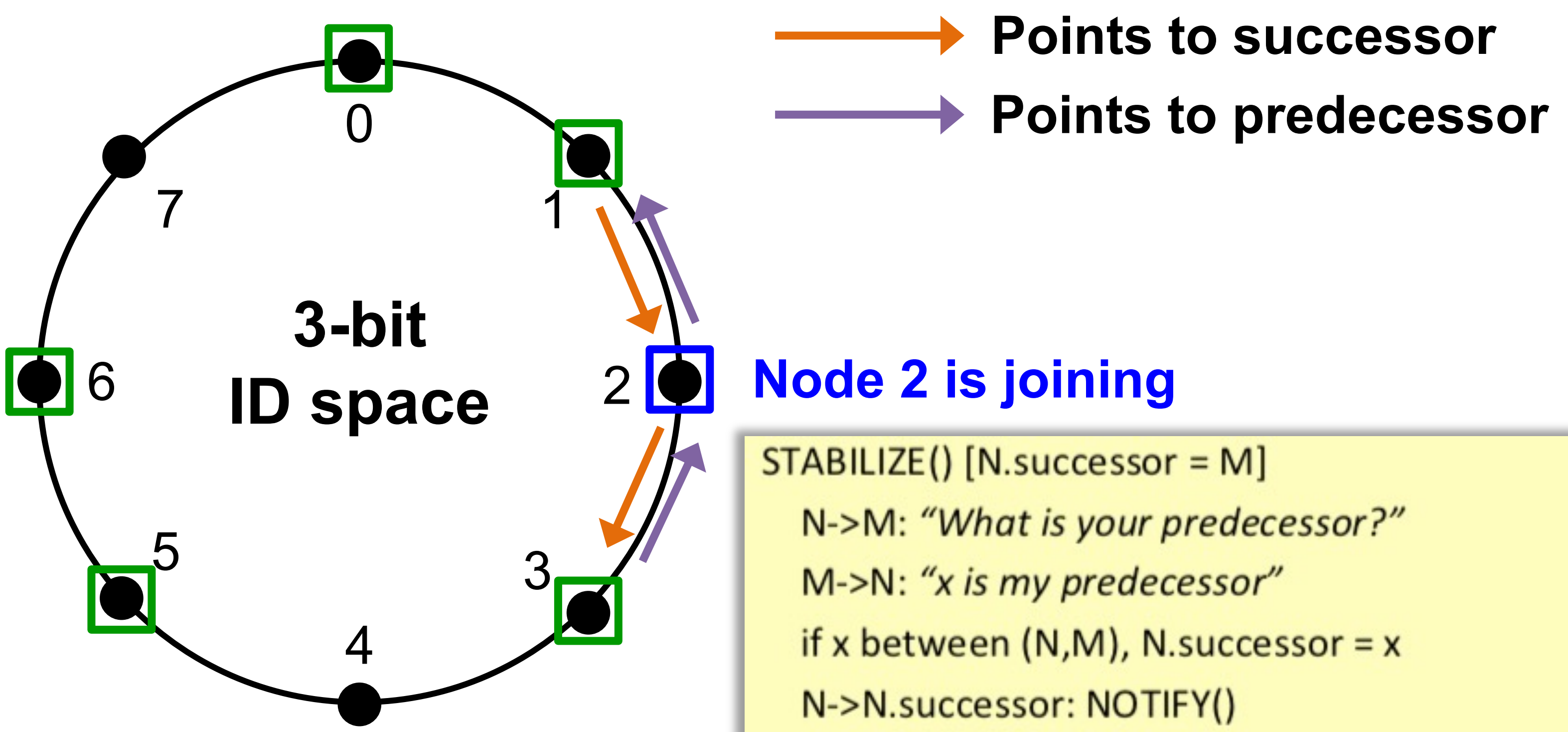


STABILIZE() [N.successor = M]
    N->M: *"What is your predecessor?"*
    M->N: *"x is my predecessor"*
    if x between (N,M), N.successor = x
    N->N.successor: NOTIFY()
NOTIFY()
    N->N.successor: *"I think you are my successor"*
M: upon receiving NOTIFY from N:
    If (N between (M.predecessor,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

Chord – node joining

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

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

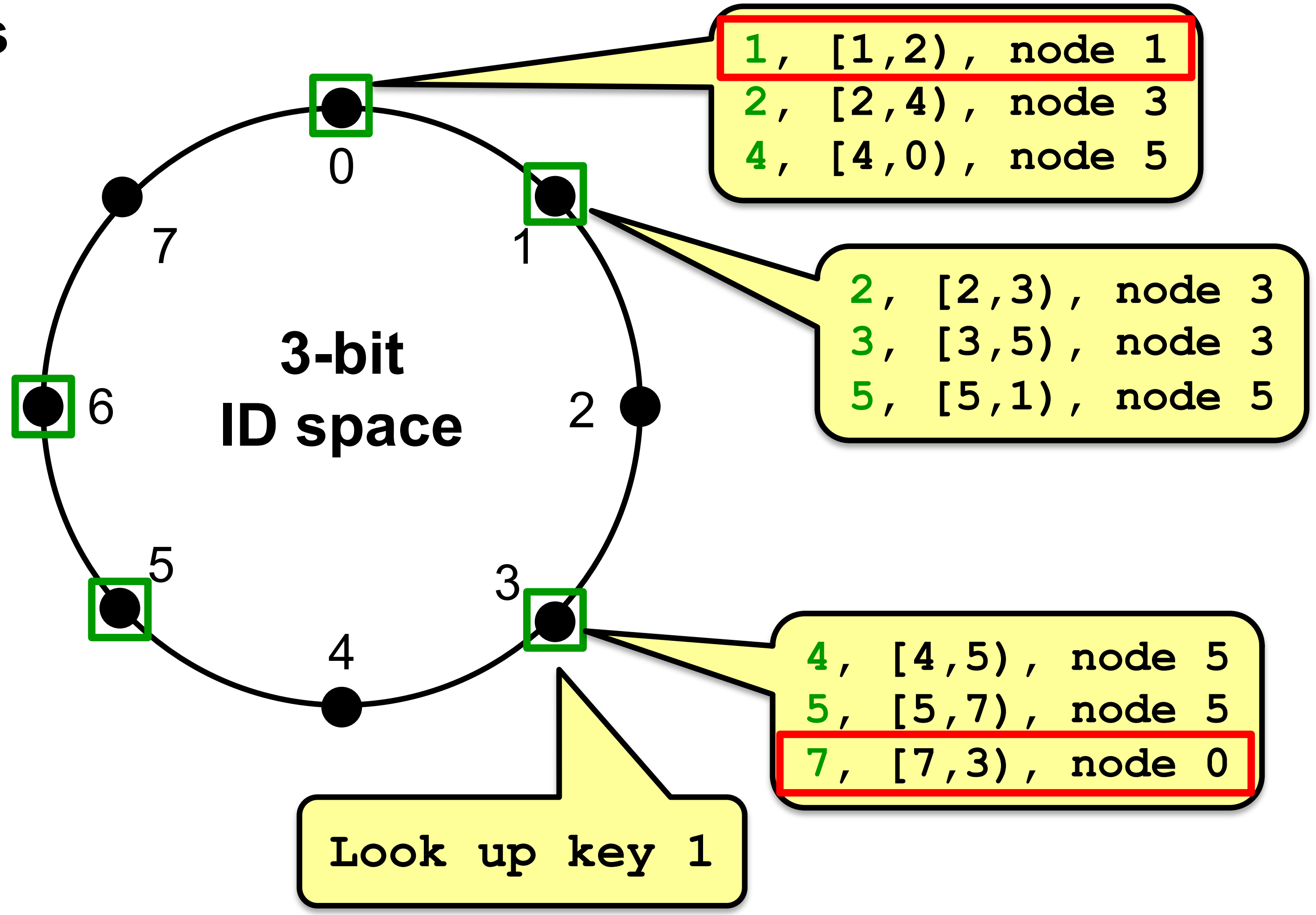


STABILIZE() [N.successor = M]
N->M: *"What is your predecessor?"*
M->N: *"x is my predecessor"*
if x between (N,M), N.successor = x
N->N.successor: NOTIFY()
NOTIFY()
N->N.successor: *"I think you are my successor"*
M: upon receiving NOTIFY from N:
If (N between (M.predecessor, M))
M.predecessor = N

Chord – failures and successor list

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

- Identifiers/key space
- Node

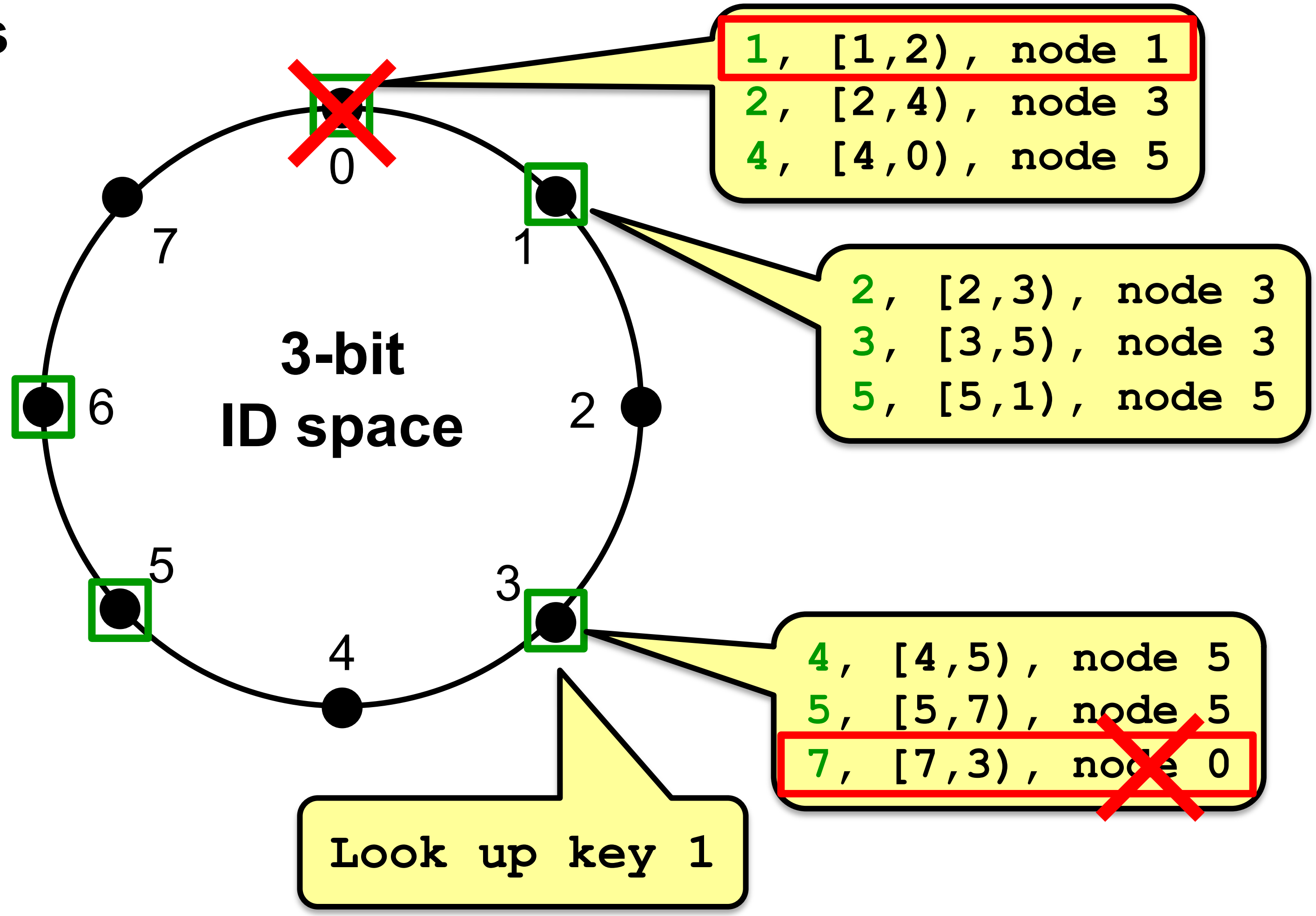


Chord – failures and successor list

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

Identifiers/key space

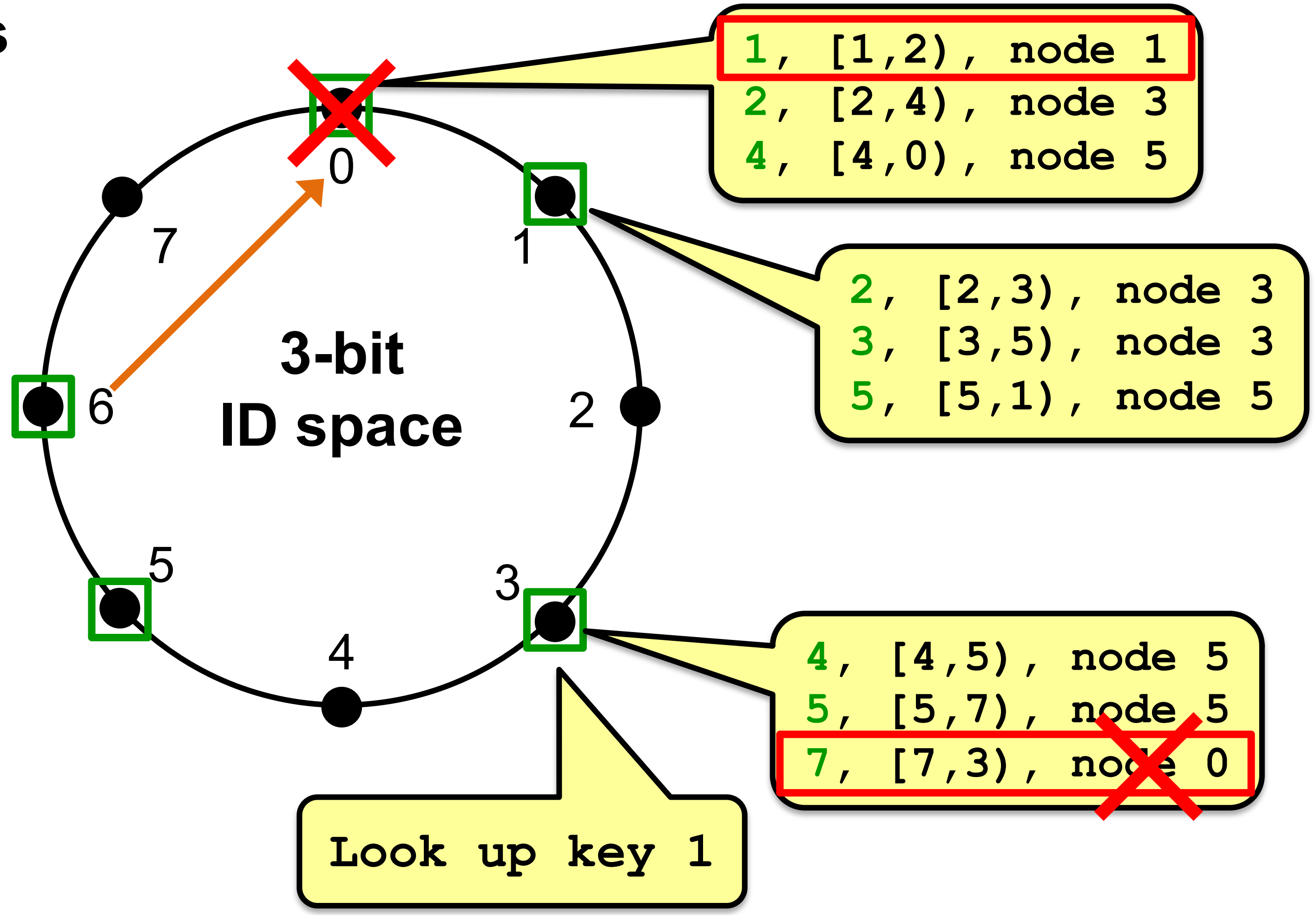
Node



Chord – failures and successor list

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

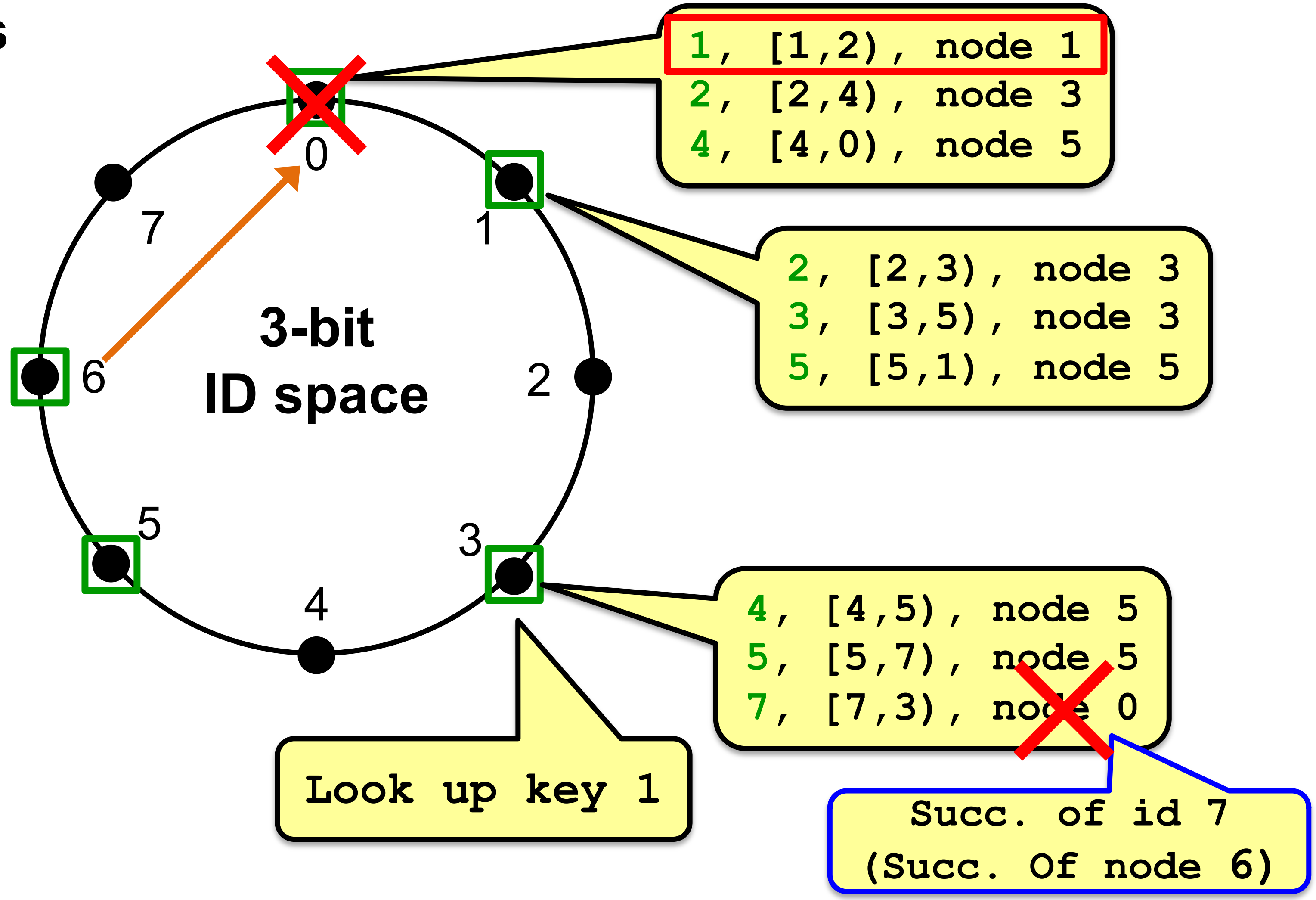
- Identifiers/key space
- Node
- Points to successor



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

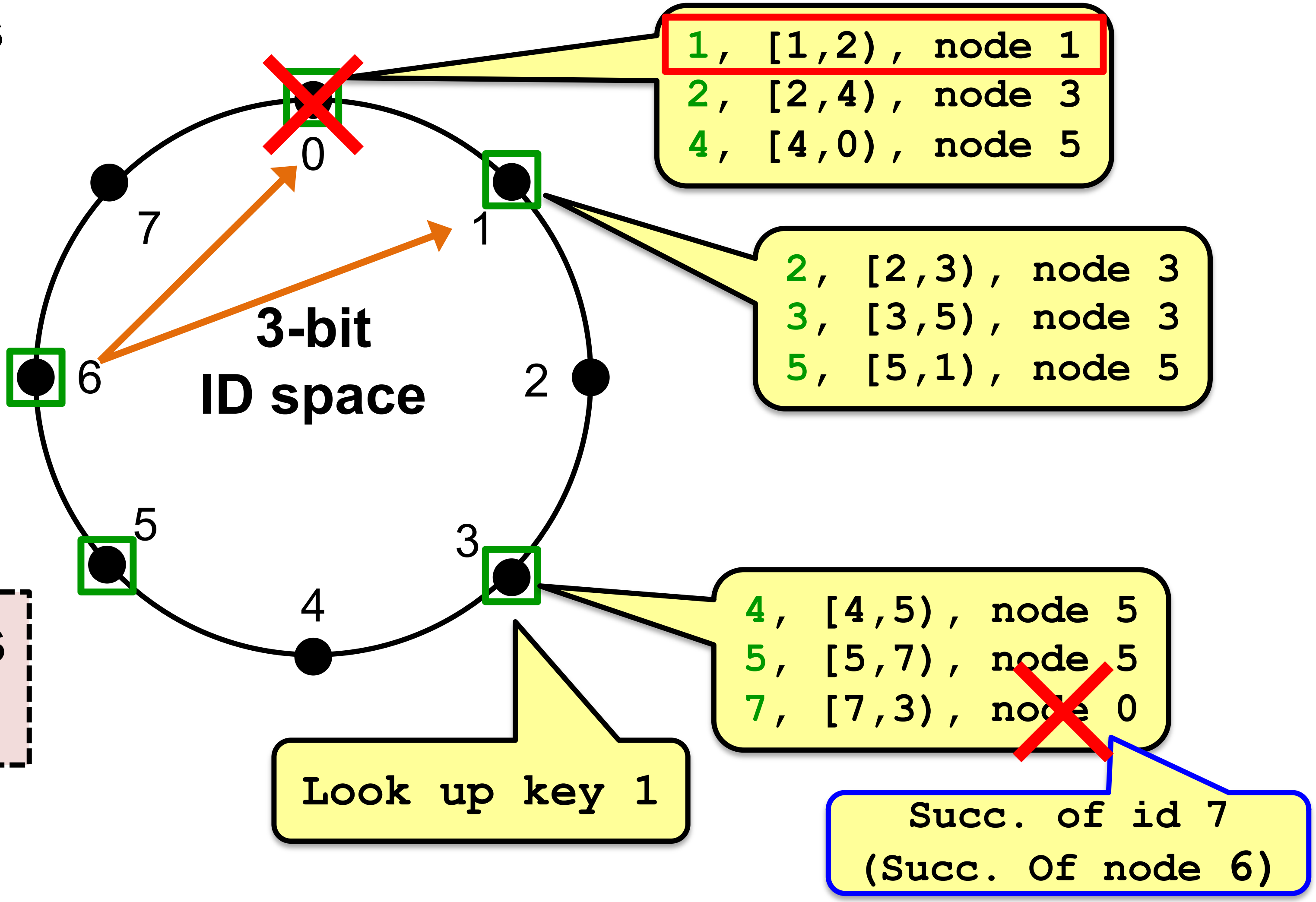
Identifiers have m = 3 bits
Key space: [0, 2^3-1]

● Identifiers/key space

□ Node

→ Points to successor

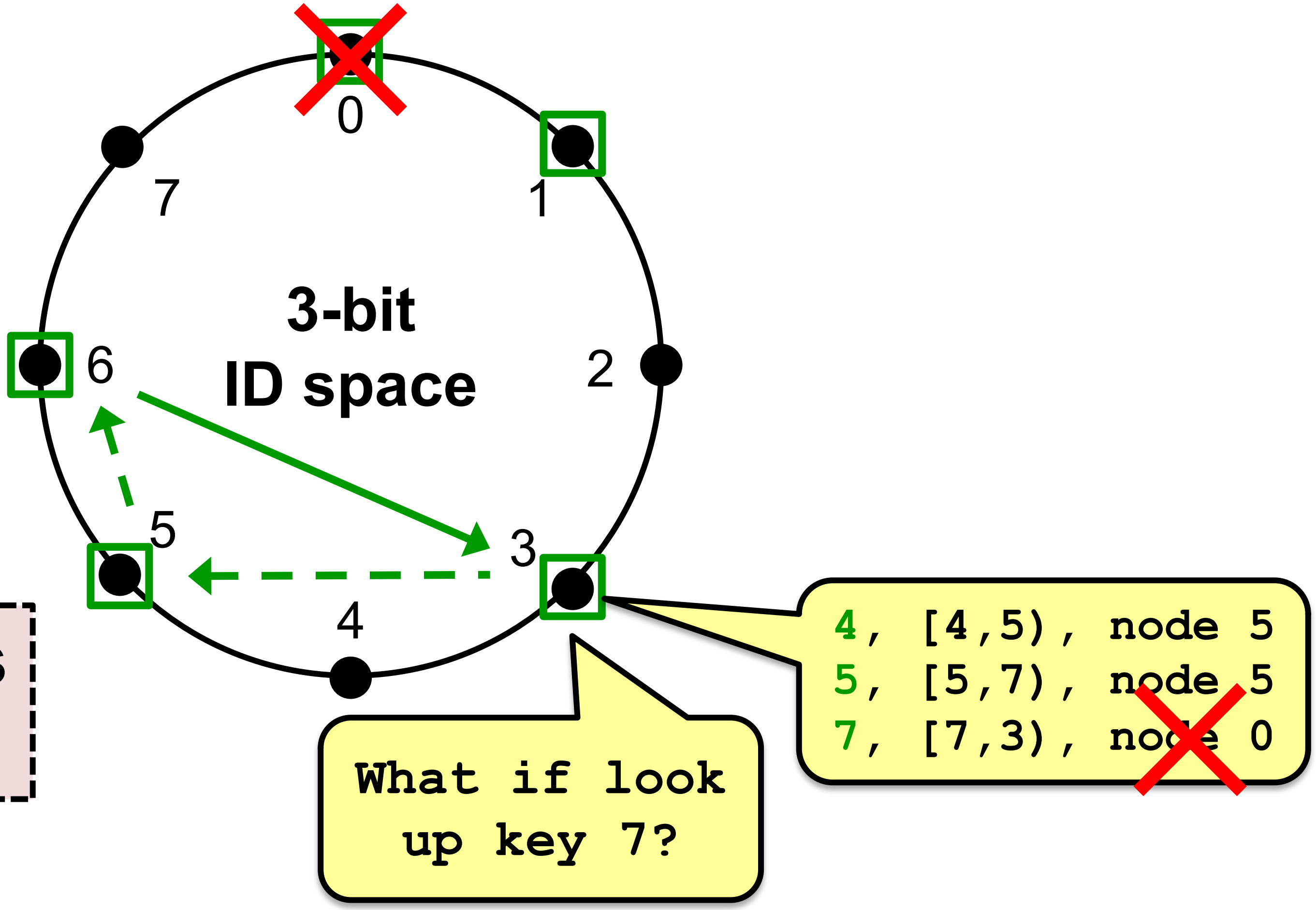
r-nearest successors
(r = logN)



Chord – failures and successor list

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

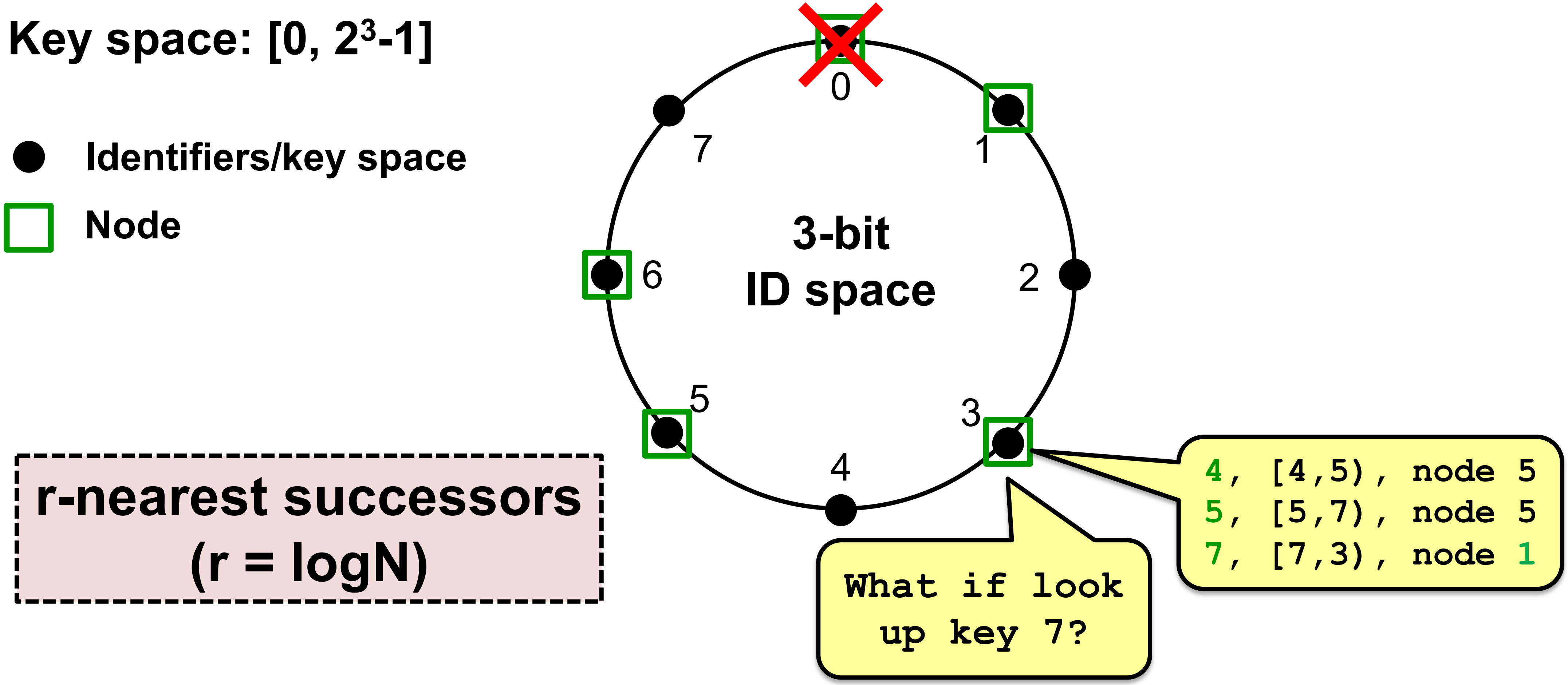
● Identifiers/key space
□ Node



r-nearest successors (r = logN)

Chord – failures and successor list

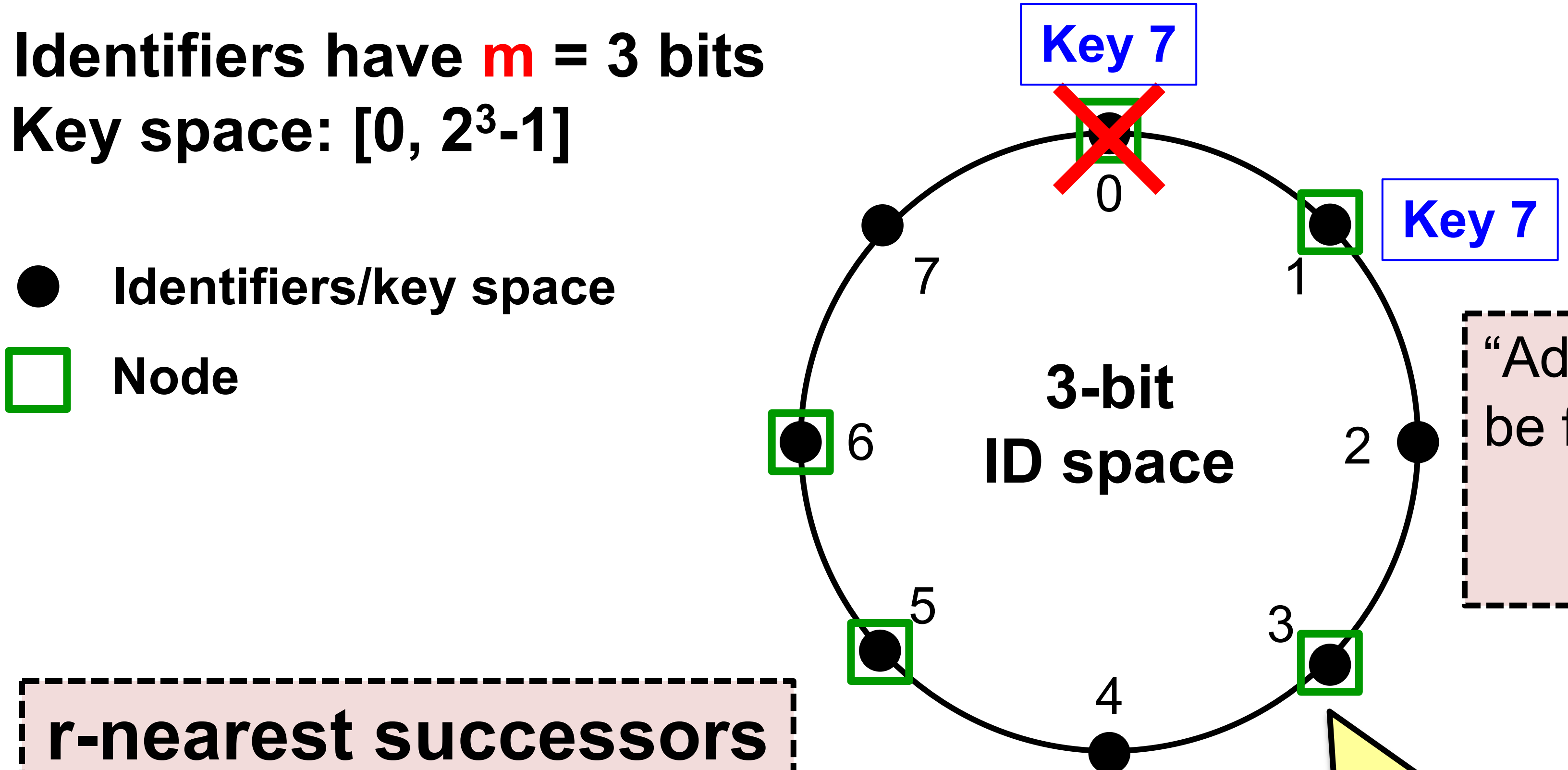
Identifiers have m = 3 bits
Key space: [0, 2^3-1]



r-nearest successors
(r = logN)

DHash replicates data 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)

```
Get data
under key 7
```

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?

- **High latency and limited bandwidth** between peers (vs. intra/inter-datacenter, client-server model)
  - 1 M nodes = 20 hops; 50 ms / hop gives 1 sec lookup latency (assuming no failures / slow connections…)
- User computers are **less reliable** than managed servers
- **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