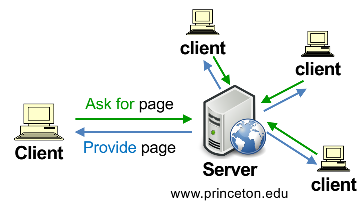
Peer-to-Peer Systems and Distributed Hash Tables

COS 418: Distributed Systems
Lecture 9

Mike Freedman

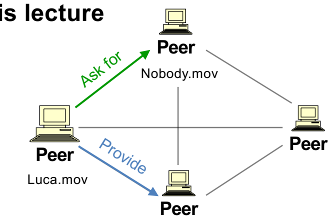
1

Distributed Application Architecture



Client-Server

This lecture



Peer-to-Peer

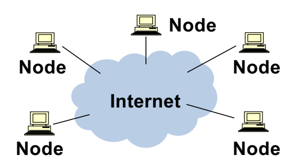
2

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

3

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

4

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

5

5

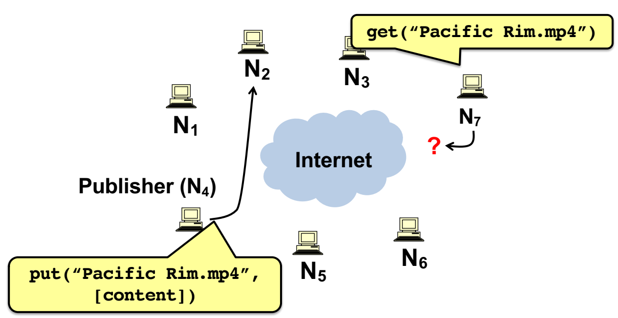
Why might P2P be a win?

- **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**
    - Both technically *and* legally

6

6

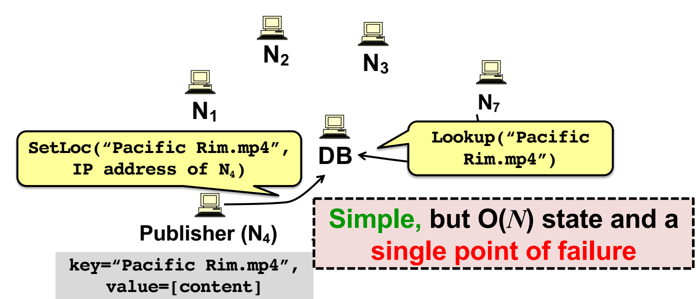
The lookup problem: locate the data



7

7

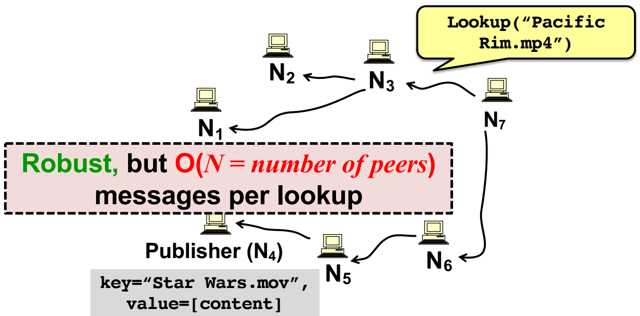
Centralized lookup (Napster)



8

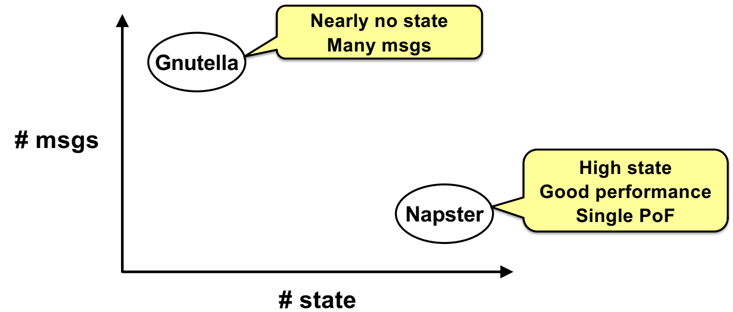
8

Flooded queries (original Gnutella)



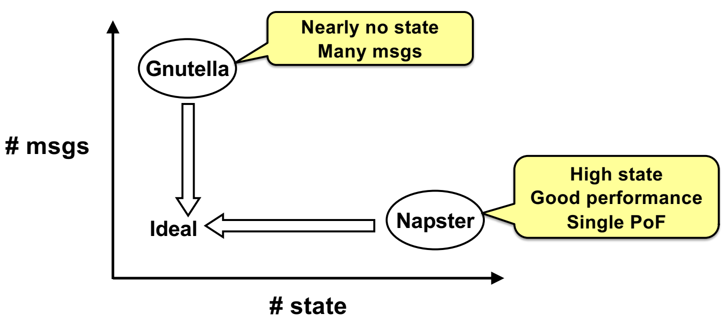
9

Tradeoffs in distributed systems



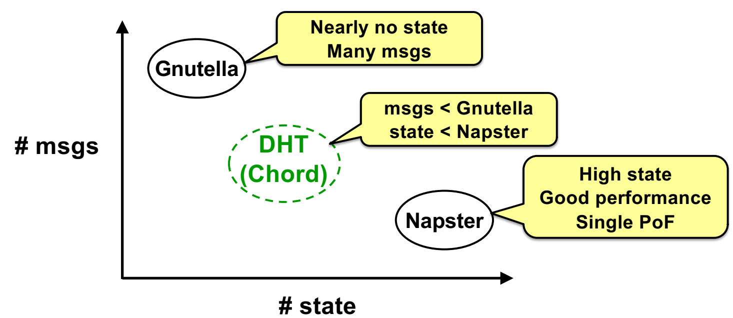
10

Tradeoffs in distributed systems



11

Tradeoffs in distributed systems



12

Today

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

13

13

What is a DHT (and why)?

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

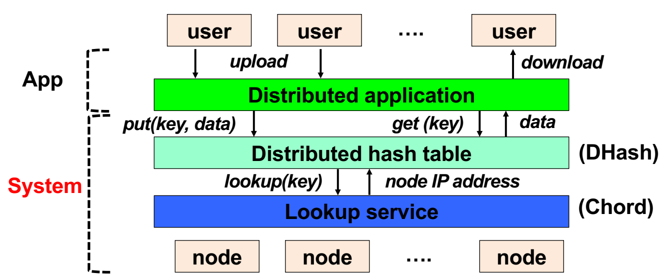
  `key = hash(data)`
  `lookup(key)` → **`IP addr`** (Chord lookup service)
  `send-RPC(IP address,` **`put,`** `key, data)`
  `send-RPC(IP address,` **`get,`** `key)` → `data`

- **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

14

14

Cooperative storage with a DHT



15

15

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

16

16

Today

1. Peer-to-Peer Systems

2. Distributed Hash Tables (DHT)

3. **The Chord Lookup Service**

17

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

18

Alternative: mod (n) hashing

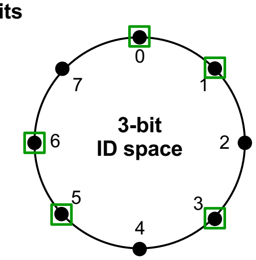
- **System of n nodes: 1…n**
  - Key identifier = *hash(key) mod n*
  - Good load balancing
- **What if a node fails?**
  - Instead of n nodes, now *n -1* nodes
  - Mapping of all keys change, as now *hash(key) mod (n-1)*
    - **N = 5**
      - 12594 mod 5 = 4
      - 28527 mod 5 = 2
      - 816 mod 5 = 1
      - 716565 mod 5 = 0
    - **N = 4**
      - 12594 mod 4 = **2**
      - 28527 mod 4 = **3**
      - 816 mod 4 = **0**
      - 716565 mod 4 = **1**

19

Consistent hashing [Karger '97] – data partition

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

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

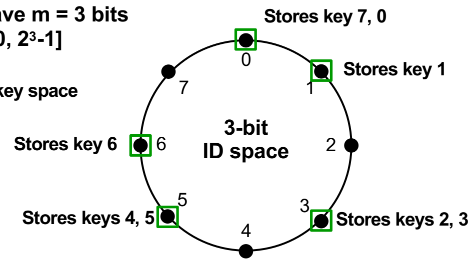


20

Consistent hashing [Karger '97] – data partition

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

● Identifiers/key space
□ Node

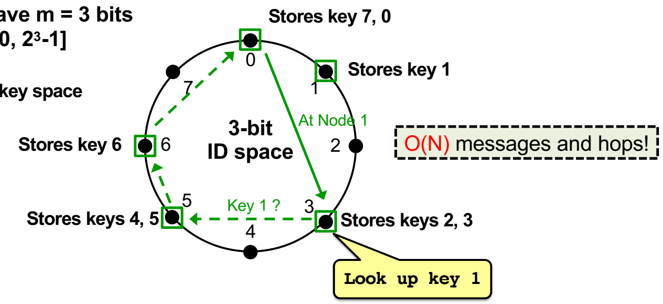


Key is stored at its **successor:** node with next-higher ID

Consistent hashing [Karger '97] – basic lookup

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

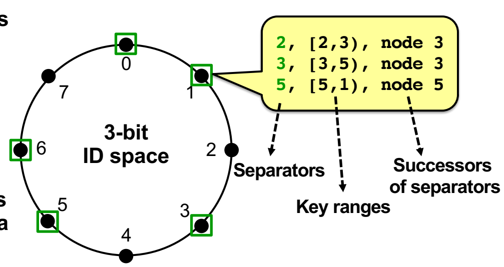
● Identifiers/key space
□ Node
Successor pointer



Chord – finger tables

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

● Identifiers/key space
□ Node

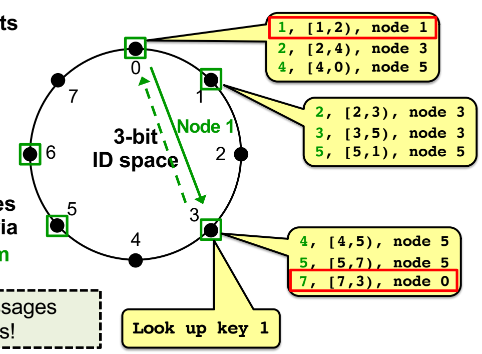


Each node keeps m states
Key space → m ranges via
$(N+2^{k-1}) \bmod 2^m$, 1<=k<=m

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

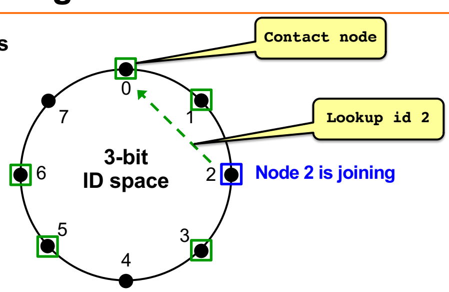
Chord lookup algorithm properties

- **Interface:** lookup(key) → 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]

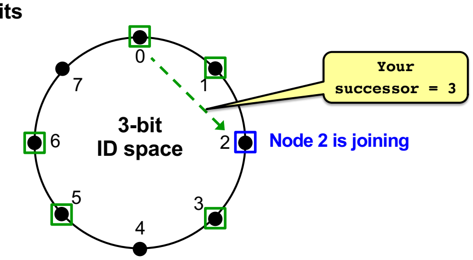
● **Identifiers/key space**
□ **Node**



Chord – node joining

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

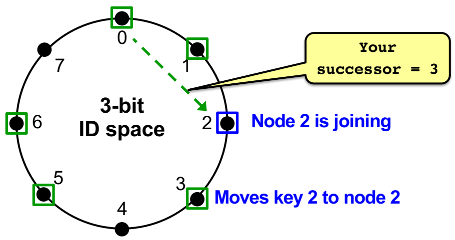
● **Identifiers/key space**
□ **Node**



Chord – node joining

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

- **Identifiers/key space**
- **Node**



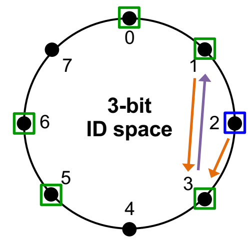
31

31

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

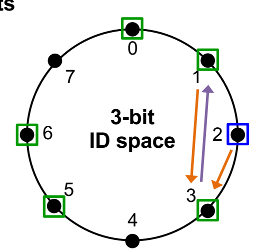
32

32

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

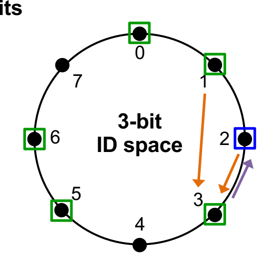
33

33

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
```

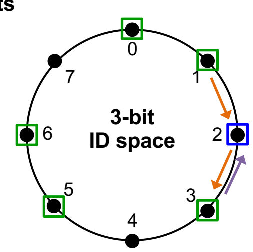
34

34

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
```

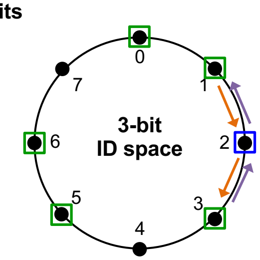
35

35

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
```

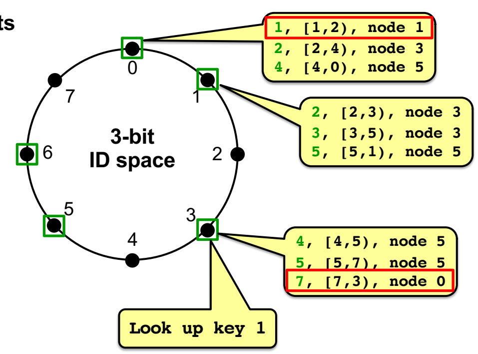
36

36

Chord – failures and successor list

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

● Identifiers/key space
□ Node



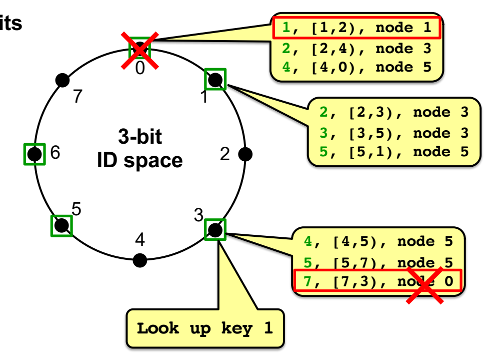
37

37

Chord – failures and successor list

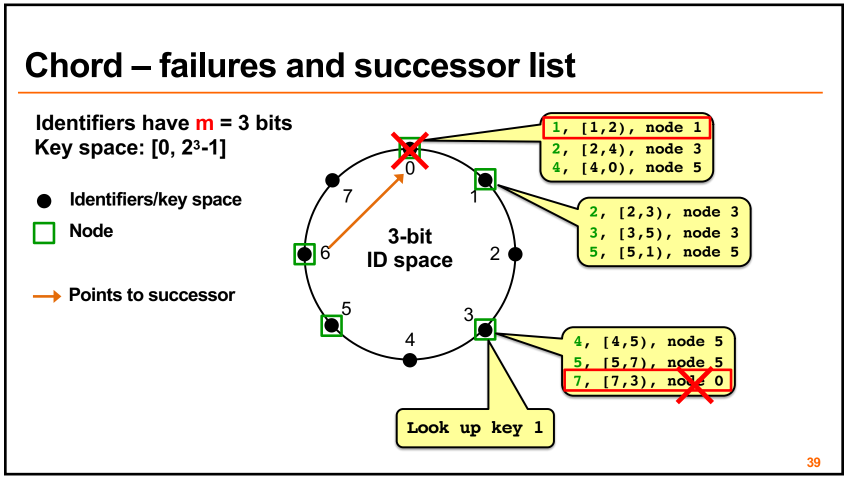
Identifiers have m = 3 bits
Key space: [0, 2^3-1]

● Identifiers/key space
□ Node

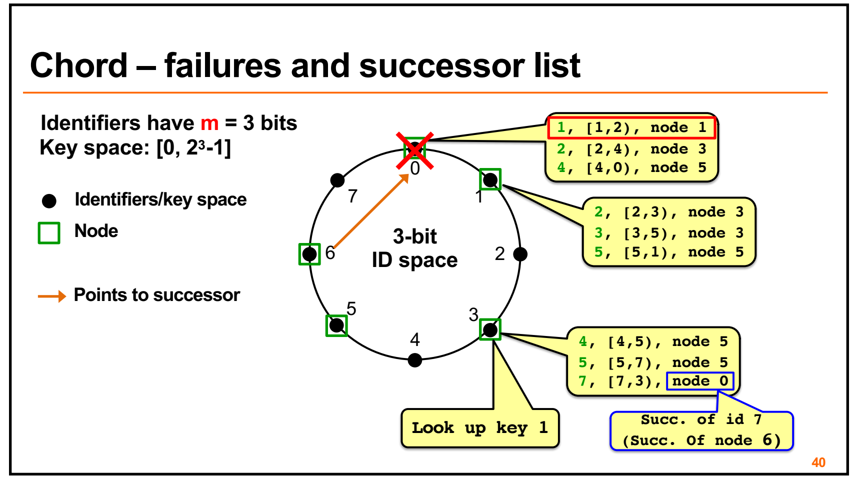


38

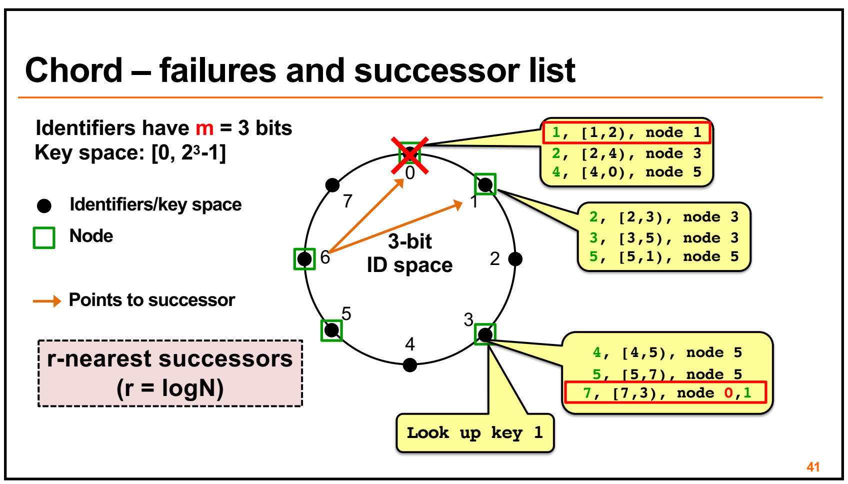
38


Chord – failures and successor list
Identifiers have m = 3 bits
Key space: [0, 2^3-1]
Identifiers/key space
Node
Points to successor
3-bit ID space
0
1
2
3
4
5
6
7
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
39

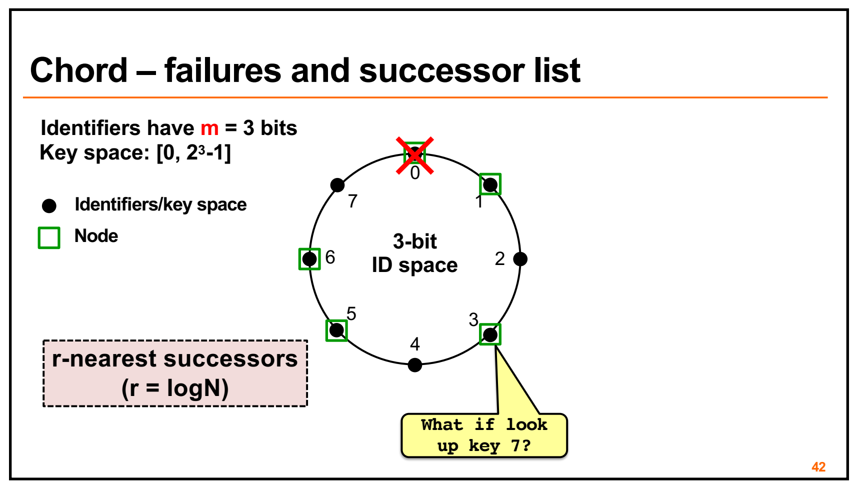
39


Chord – failures and successor list
Identifiers have m = 3 bits
Key space: [0, 2^3-1]
Identifiers/key space
Node
Points to successor
3-bit ID space
0
1
2
3
4
5
6
7
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
Succ. of id 7
(Succ. Of node 6)
Look up key 1
40

40

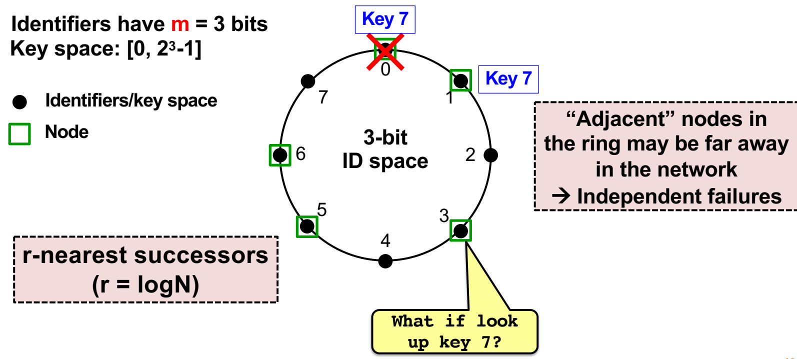

Chord – failures and successor list
Identifiers have m = 3 bits
Key space: [0, 2^3-1]
Identifiers/key space
Node
Points to successor
r-nearest successors
(r = logN)
3-bit ID space
0
1
2
3
4
5
6
7
1, [1,2), node 1
2, [2,4), node 3
4, [4,0), node 5
2, [2,3), node 3
3, [3,5), node 3
5, [5,1), node 5
4, [4,5), node 5
5, [5,7), node 5
7, [7,3), node 0,1
Look up key 1
41

41


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)
3-bit ID space
0
1
2
3
4
5
6
7
What if look up key 7?
42

42

DHash replicates blocks at *r* successors



43

43

Today

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

44

44

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

45

45

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

46

46

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

47