LOCATING DATA ON THE NETWORK: P2P NETWORKS, CHORD, AND DYNAMODB

Feb 17, 2022

George Porter

UC San Diego

ATTRIBUTION

- These slides incorporate material from:
  - Christo Wilson, NEU (used with permission)
  - Kyle Jamieson, Princeton
  - Tanenbaum and Van Steen, 3rd edition

ANNOUNCEMENTS

Project 5 is out

TA video on project 5 coming soon

Today: Locating data on the network

Tomorrow at 2pm!
Location: CSE 1242 (On ground floor near the building lobby)
Zoom Link: https://ucsd.zoom.us/j/91497330645

Title: What we talk about when we talk about networking

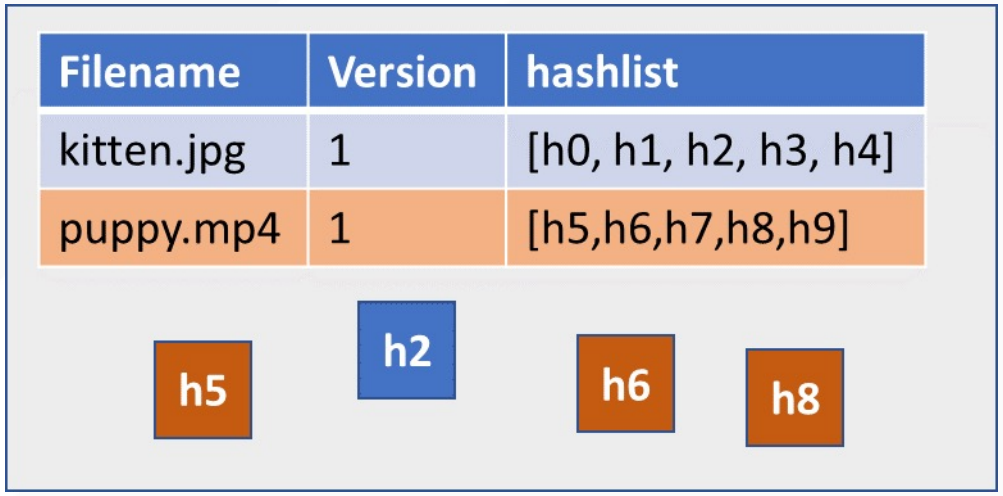
Abstract: Networks, and the applications they support, sometimes treat each other as strangers. By shaking things up a bit—expressing networked systems as compositions of small, pure functions and making their dataflow a first-class consideration—we can often achieve friendlier couplings across the stack, to the benefit of performance, robustness, and understandability. This approach has proved helpful in several contexts: networking algorithms learned "in situ," feeding data from deployment back into training; real-time video conferencing, especially for musicians and actors during the pandemic; image compression in a distributed network filesystem; and a serverless computing framework that lets software burst briefly to 10,000 cores. In ongoing work, we're building a "functional" operating system that enforces a separation between IO (declared to the OS) and computation (reproducible by default). If this system can support a broad range of computational tasks with visibility into their dataflow, we envision a new service model for cloud computing: "computation as a service."

Bio: Keith Winstein is an assistant professor of computer science and, by courtesy, of electrical engineering at Stanford University. (https://cs.stanford.edu/~keithw)

UC San Diego

LOCATING ITEMS (AT SCALE) IS A PRETTY HARD PROBLEM

- Consider our metadata store:



- Let's figure out about how many files a single server metadata store can store…

LET'S CHOOSE AN AWS INSTANCE TYPE

- General Purpose
- Compute Optimized
- Memory Optimized
- Accelerated Computing
- Storage Optimized
- Instance Features
- Measuring Instance Performance

LET'S PICK THE ARM-BASED MEMORY INSTANCE

R6g | R5 | R5a | R5b | R5n | R4 | X2gd | X1e | X1 | High Memory | z1d

Amazon EC2 R6g instances are powered by Arm-based AWS Graviton2 processors. They deliver up to 40% better price performance over current generation R5 instances for memory-intensive applications.

Features:

- Custom built AWS Graviton2 Processor with 64-bit Arm Neoverse cores
- Support for Enhanced Networking with Up to 25 Gbps of Network bandwidth
- EBS-optimized by default
- Powered by the AWS Nitro System, a combination of dedicated hardware and lightweight hypervisor
- With R6gd instances, local NVMe-based SSDs are physically connected to the host server and provide block-level storage that is coupled to the lifetime of the instance

MEMORY INSTANCE TYPES

Instance Size	vCPU	Memory (GiB)	Instance Storage	Network Bandwidth (Gbps)***	EBS Bandwidth (Mbps)
r6g.medium	1	8	EBS-Only	Up to 10	Up to 4,750
r6g.large	2	16	EBS-Only	Up to 10	Up to 4,750
r6g.xlarge	4	32	EBS-Only	Up to 10	Up to 4,750
r6g.2xlarge	8	64	EBS-Only	Up to 10	Up to 4,750
r6g.4xlarge	16	128	EBS-Only	Up to 10	4750
r6g.8xlarge	32	256	EBS-Only	12	9000
r6g.12xlarge	48	384	EBS-Only	20	13500
r6g.16xlarge	64	512	EBS-Only	25	19000
r6g.metal	64	512	EBS-Only	25	19000

Cost (per hour) of the r6g.16xlarge instance type: $3.2256

HOW MANY FILES CAN FIT INTO R6G.16XLARGE?

- 512GB of RAM
- Data requirements of each entry in the FileInfoMap?
  - Depends on size of the block...
  - Depends on distribution of file sizes…
    - Lots of small files? (e.g. C++, Java, Python, Go development)
    - Or big files? (audio or video files)
- Let’s see what the research literature says

TANENBAUM ET AL, 2004

File Size Distribution on UNIX Systems—Then and Now

Andrew S. Tanenbaum, Jorrit N. Herder*, Herbert Bos
Dept. of Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
{ast@cs.vu.nl, jnherder@cs.vu.nl, herbertb@cs.vu.nl}

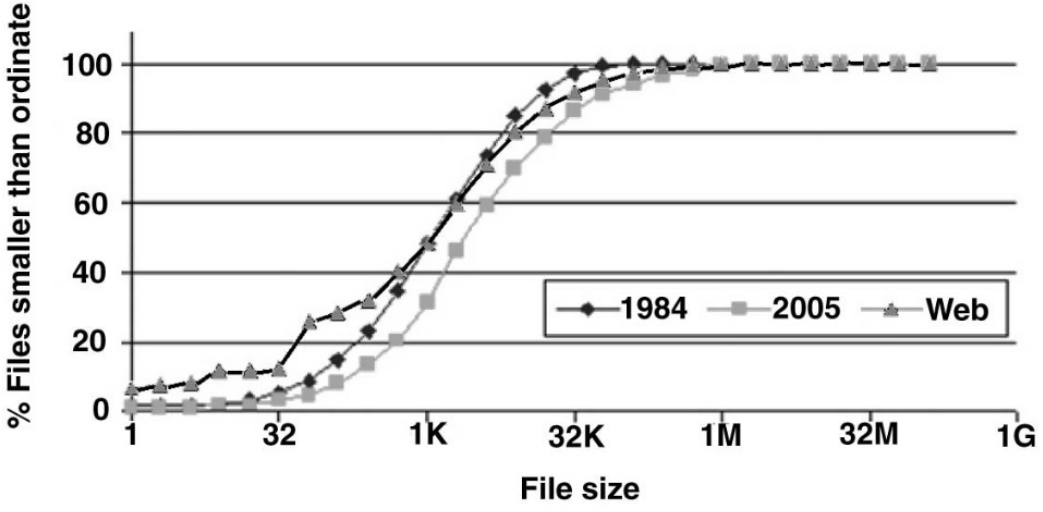


Fig. 2. Data of Fig. 1 shown graphically.

LIU ET AL, 2013

2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

Understanding Data Characteristics and Access Patterns in a Cloud Storage System

* *Ministry of*
† *Tsinghua National L*
ystem Science
cience and Technology

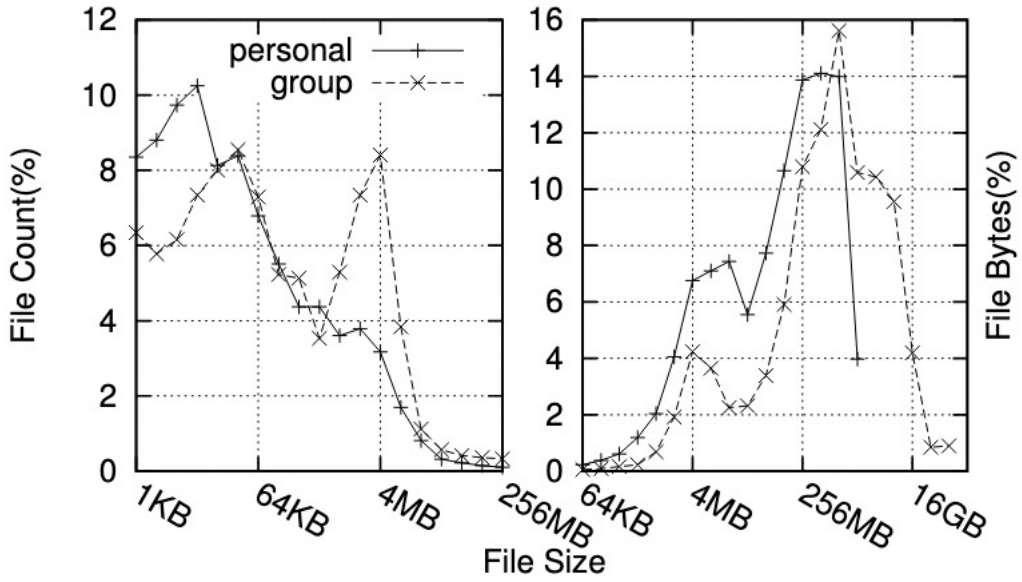


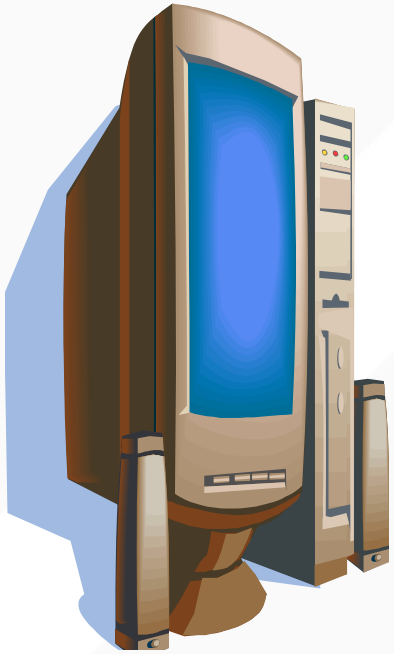
Figure 2. Bimodal file size distributions

A SIMPLE MODEL

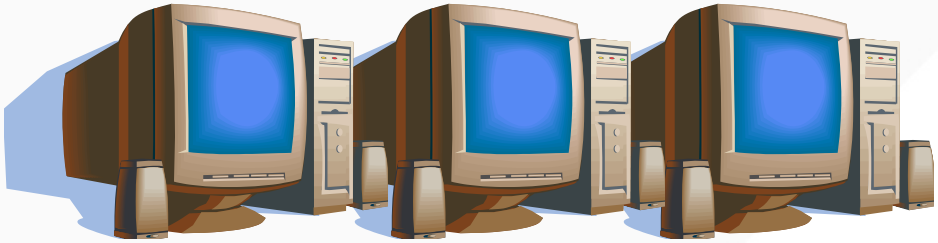
BUT WHAT IF YOU NEED MORE SPACE?

- What if you have more than 20 million files??
- You need *scale*

SCALING



Vertical Scaling
(bigger machines)



Horizontal Scaling
(more machines)

VERTICAL SCALING

- Get a machine with more RAM, more storage, a faster CPU, more CPUs, …
- Advantages:
  - Simple: Single machine abstraction
  - Simple: Only one IP address/hostname to consult
- Disadvantages:
  - Machines only get so big (have so much ram, etc)
  - What if the machine fails?

HORIZONTAL SCALING

- Form a *cluster* of 10, 100, 1000… servers that work together
- Advantages:
  - No one machine has to be very expensive/fancy
  - A failure of one machine doesn't result in everything being lost
- Disadvantages:
  - How to find the data you're looking for??
  - Performance is hard to reason about (subject of a future lecture, in fact)

HORIZONTAL SCALING ISSUES

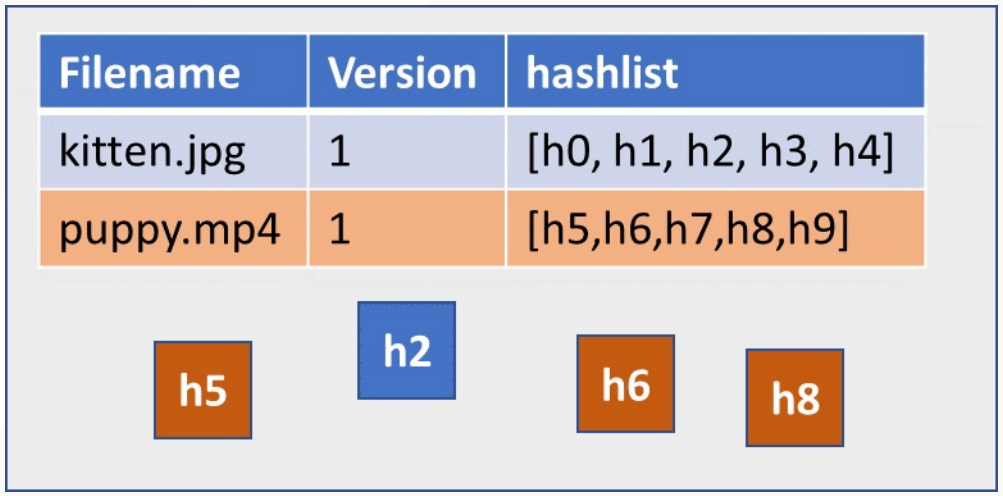
- Probability of any failure in given period = $1-(1-p)^n$
  - p = probability a machine fails in given period
  - n = number of machines

- For **50K machines**, each with **99.99966% available**
  - **16%** of the time, **data center experiences failures**

- For **100K machines, failures 30%** of the time!

THE LOCATION PROBLEM

- Given a cluster C of *N* servers, how do we locate the specific server C_i responsible for a data item?

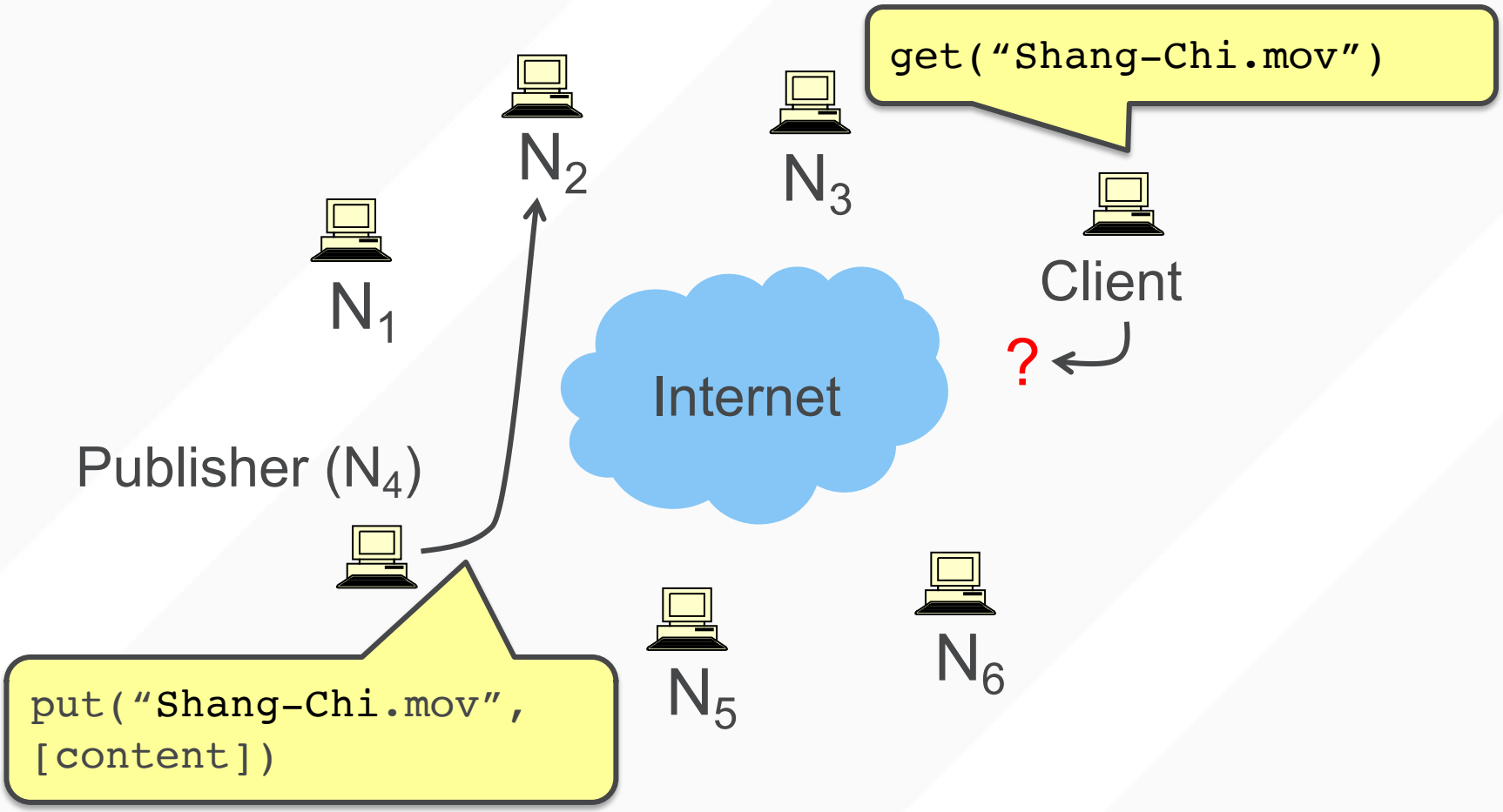


- E.g. For a logical metadata storage *service* spread across N machines, which machine has the hash list for kitten.jpg? For puppy.mp4?

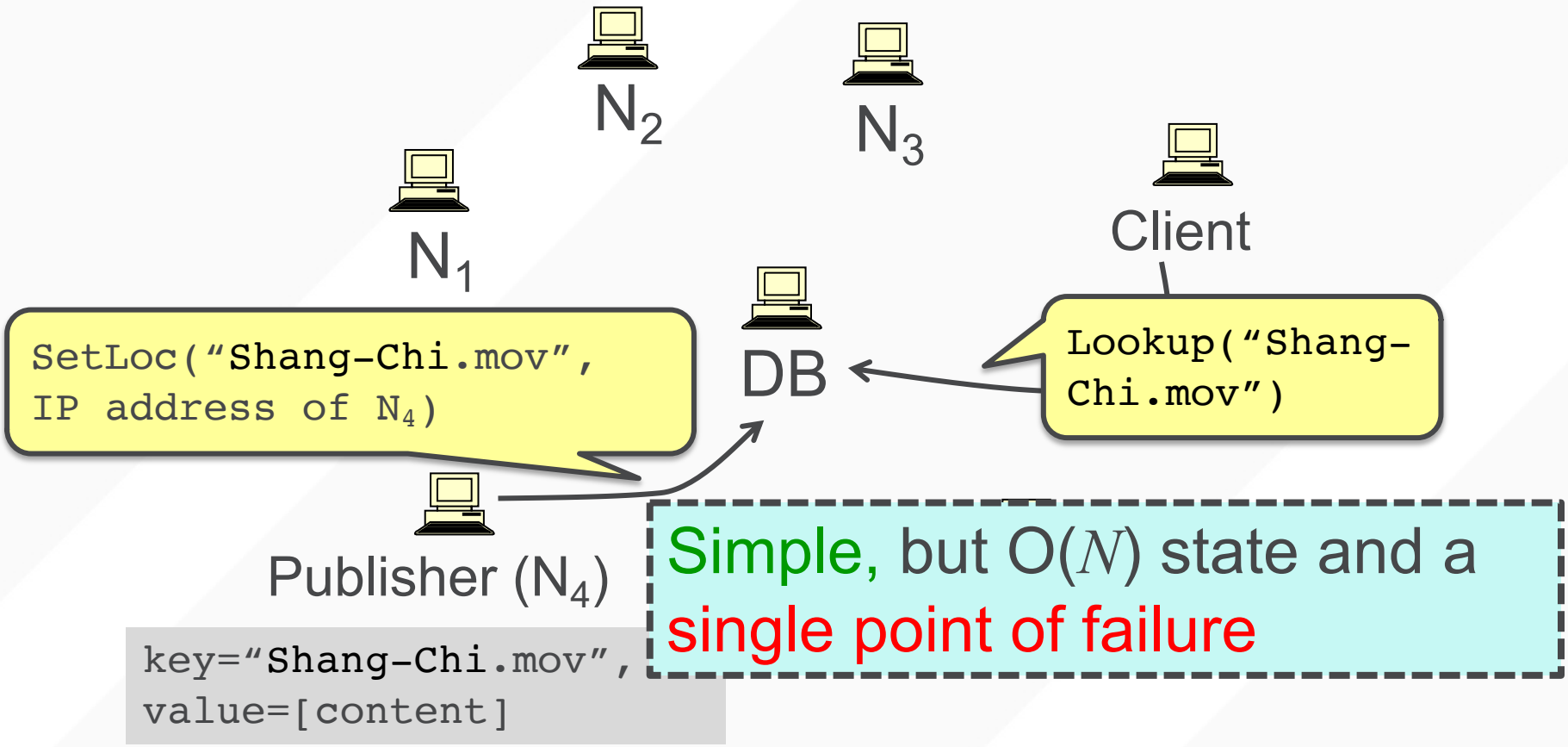
WHAT IS “FLAT” NAMING?

- The name doesn’t give you an indication of where the data is located
- Flat:
  - MAC address: 00:50:56:a3:0d:2a
- Vs hierarchical:
  - IP address: 206.109.2.12/24
  - DNS name: starbase.neosoft.com

FLAT NAME LOOKUP PROBLEM



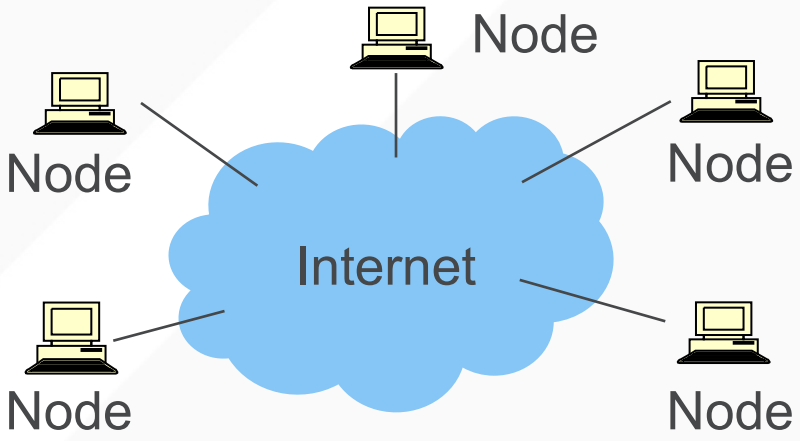
CENTRALIZED LOOKUP (NAPSTER)



Outline

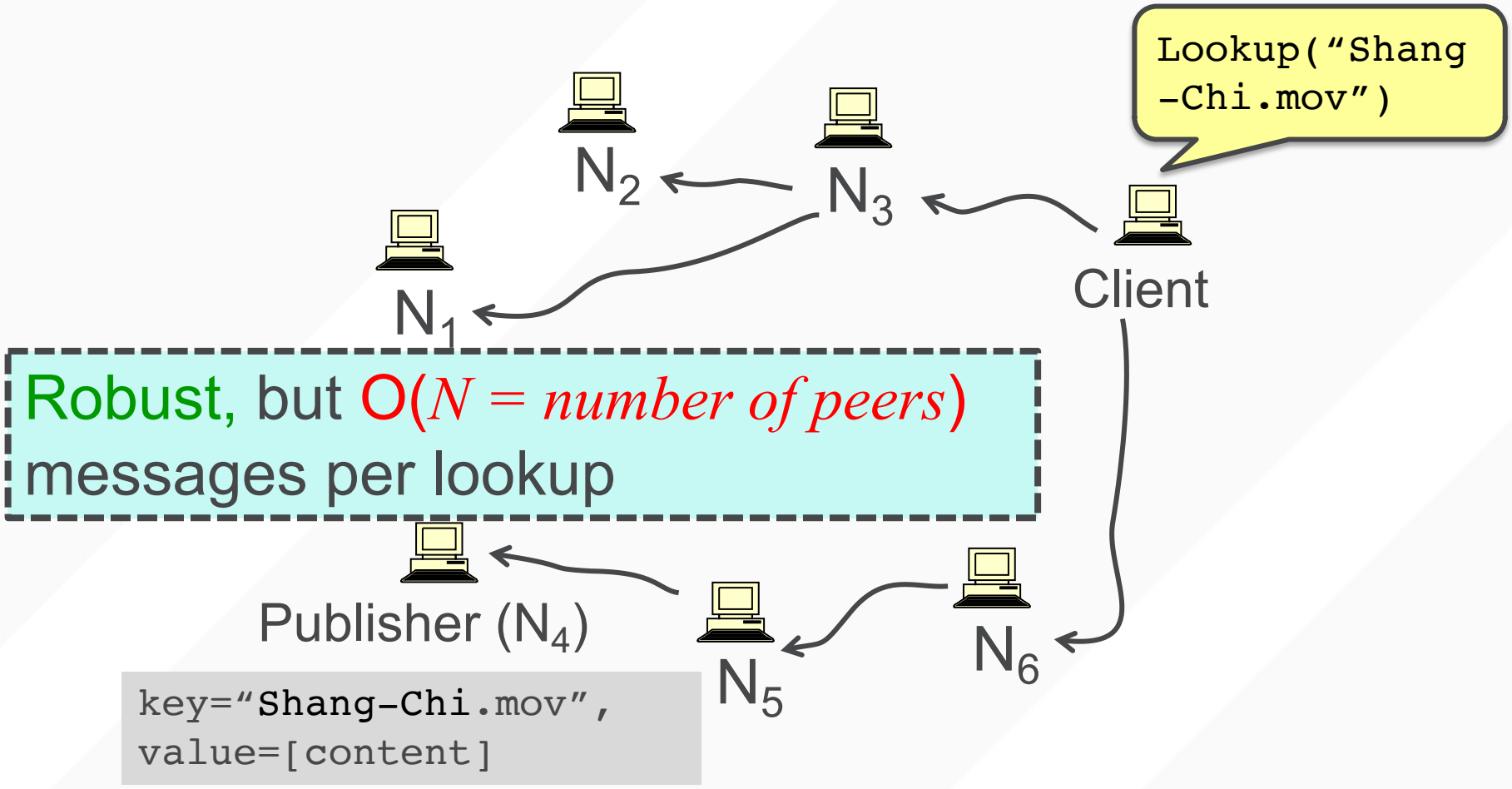
- Peer-to-peer networks
- Chord DHT
- DynamoDB DHT

PEER-TO-PEER (P2P) NETWORKS

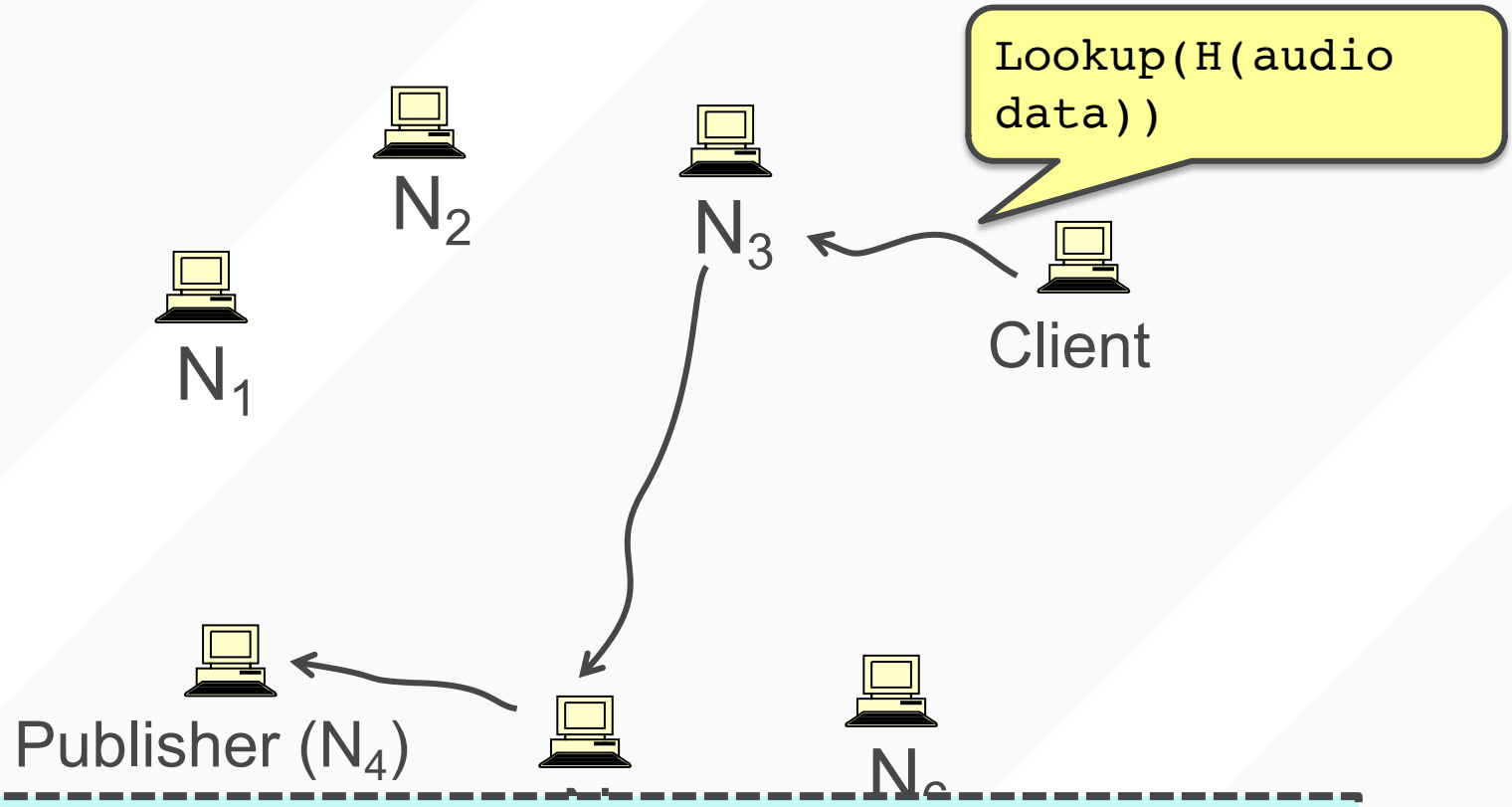


- A **distributed** system architecture:
  - **No centralized control**
  - Nodes are **roughly symmetric** in function
- **Large** number of **unreliable** nodes (could be reliable too)

FLOODED QUERIES (ORIGINAL GNUTELLA)



ROUTED DHT QUERIES (CHORD AND DYNAMODB)



Can we make it robust, reasonable state, reasonable number of hops?

Outline

- Peer-to-peer networks
- Chord DHT
- DynamoDB DHT

SYSTEMATIC FLAT NAME LOOKUPS VIA DHTS

- Local hash table:

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

- **Service:** Constant-time insertion and lookup

How can I do (roughly) this across millions of hosts on the Internet or within a giant datacenter application?
Distributed Hash Table (DHT)

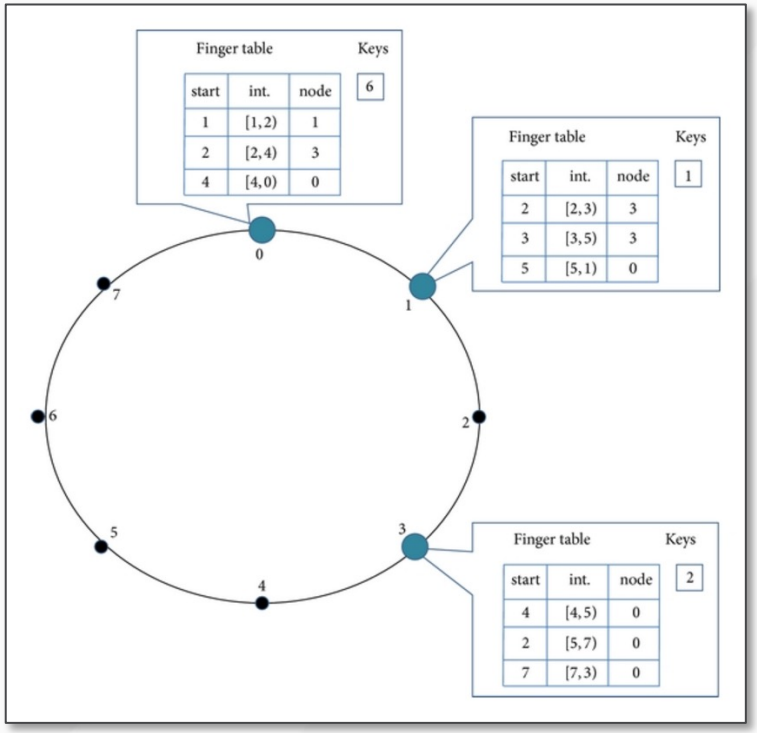
WHAT IS A DHT (AND WHY)?

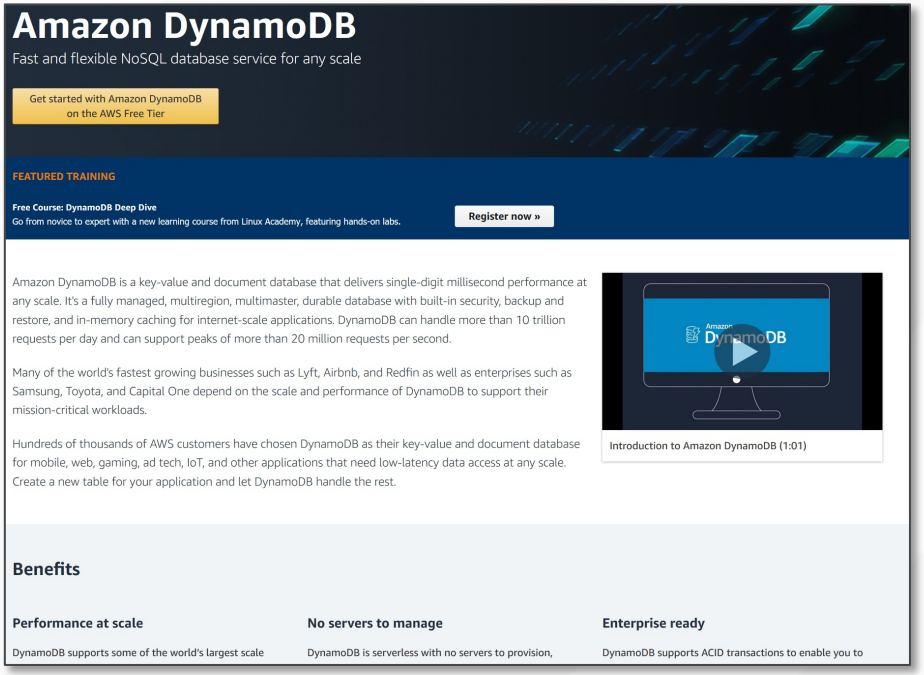
- Distributed Hash Table:

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

- **Partitioning data** in truly **large-scale distributed systems**
  - Tuples in a global database engine
  - Data blocks in SurfStore
  - Files in a P2P file-sharing system

TWO EXAMPLES OF DHTS





- Chord
  - Fully decentralized
  - Over wide-area Internet
  - Designed for millions of end points
- DynamoDB
  - Managed within a single datacenter
  - Some centralization
  - 10s to 100s of end points

STRAWMAN: MODULO HASHING (E.G. HASHMAP)

- Consider problem of data partition:
  - Given **object id X**, choose one of ***k*** servers to use

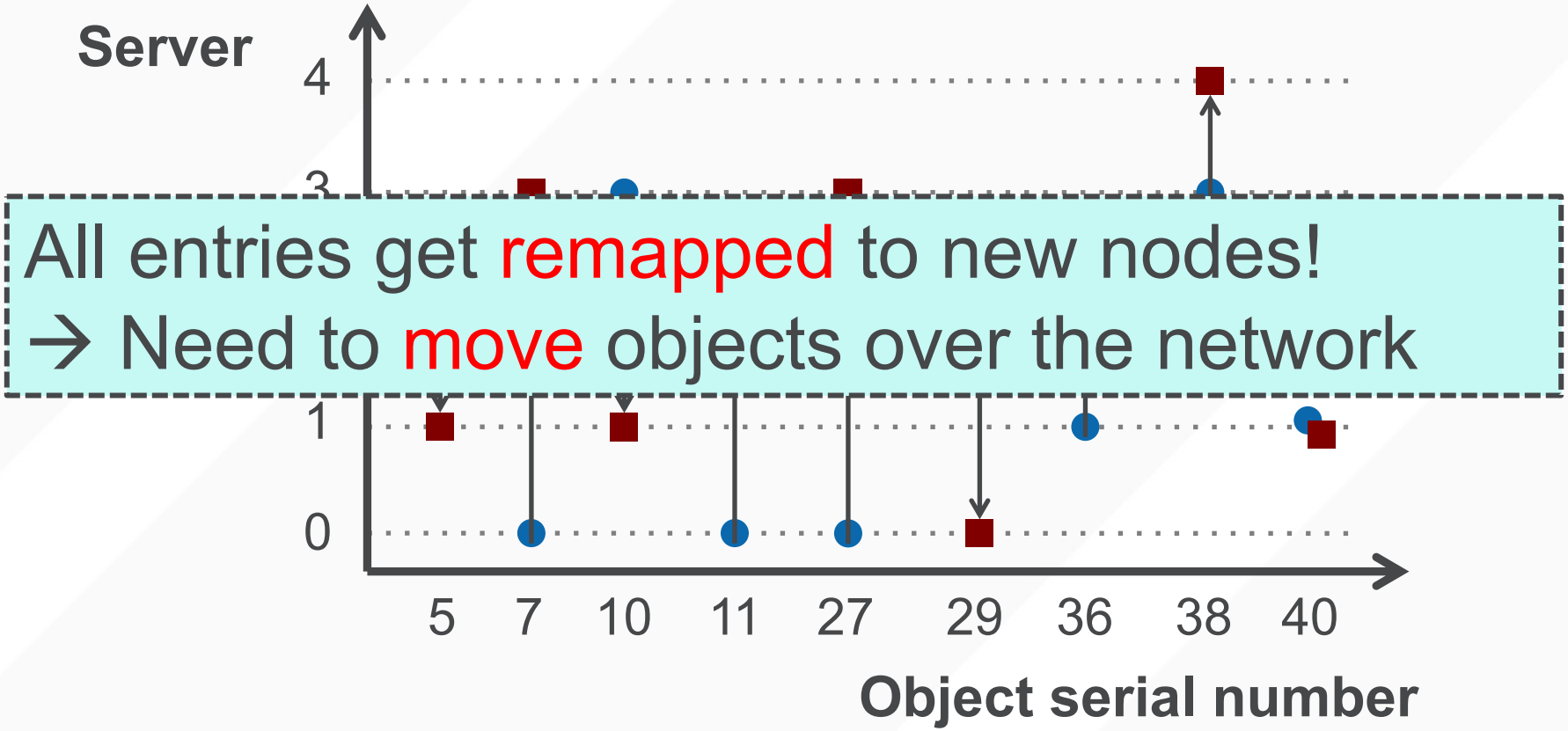
- Suppose instead we use **modulo hashing:**
  - Place ***X*** on server ***i* = hash(*X*) mod *k***

- What happens if a server fails or joins (k ← k±1)?
  - or different clients have **different estimate** of k?

PROBLEMS WITH MODULO HASHING

$h(x) = x + 1 \pmod 4$

Add one machine: $h(x) = x + 1 \pmod 5$



All entries get remapped to new nodes!

→ Need to move objects over the network

CHORD LOOKUP ALGORITHM PROPERTIES

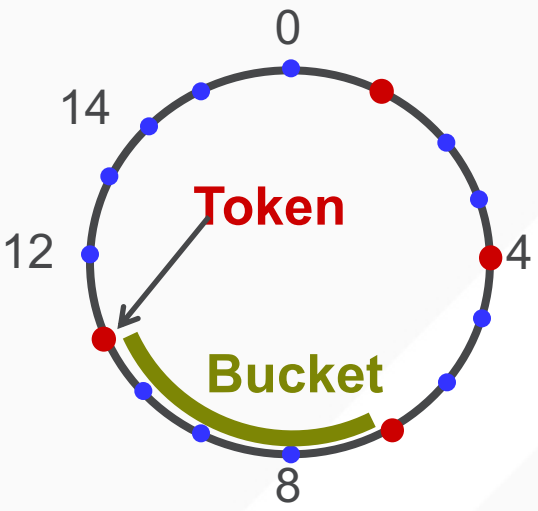
- **Interface:** lookup(key) $\rightarrow$ IP address
- **Efficient:** O(log N) messages per lookup
  - N is the total number of servers
- **Scalable:** O(log N) state per node
- **Robust:** survives massive failures

CHORD 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

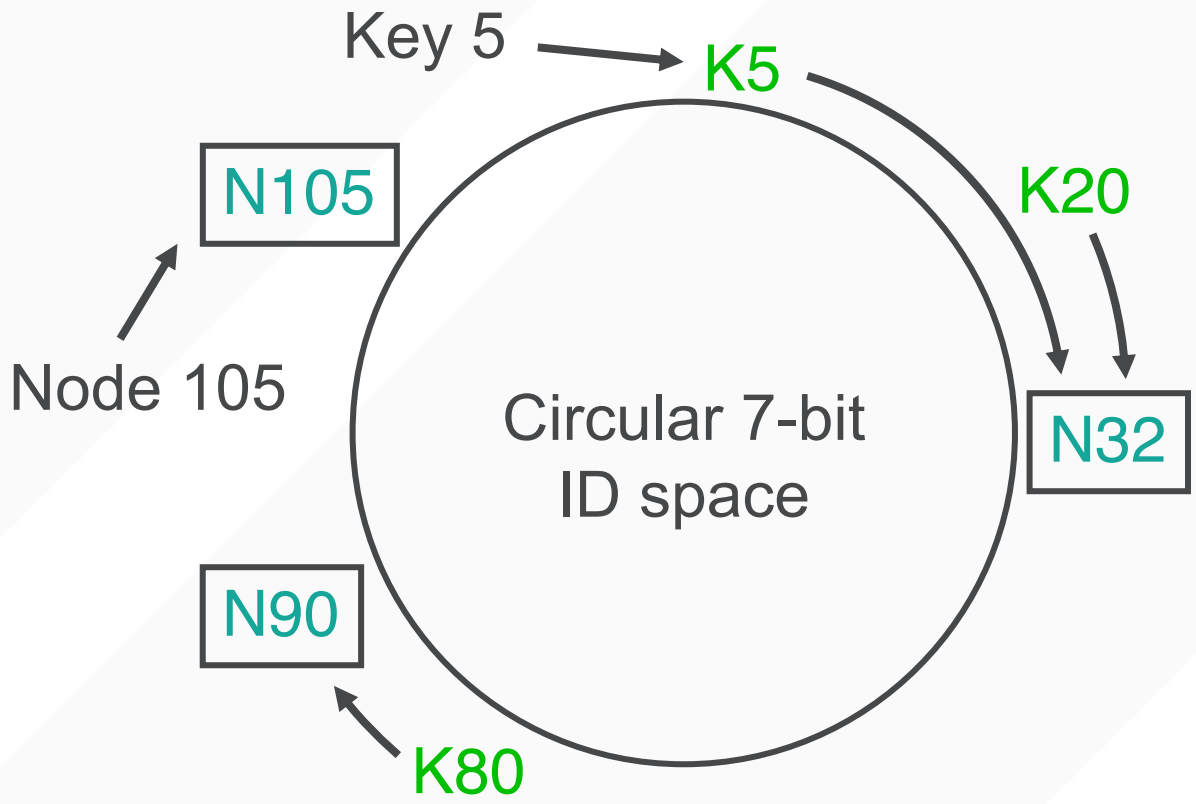
CONSISTENT HASHING

- Assign *n tokens* to random points on mod 2^k circle; hash key size = *k*
- Hash object to random circle position
- Put object in closest clockwise bucket
  - *successor* (key) → bucket



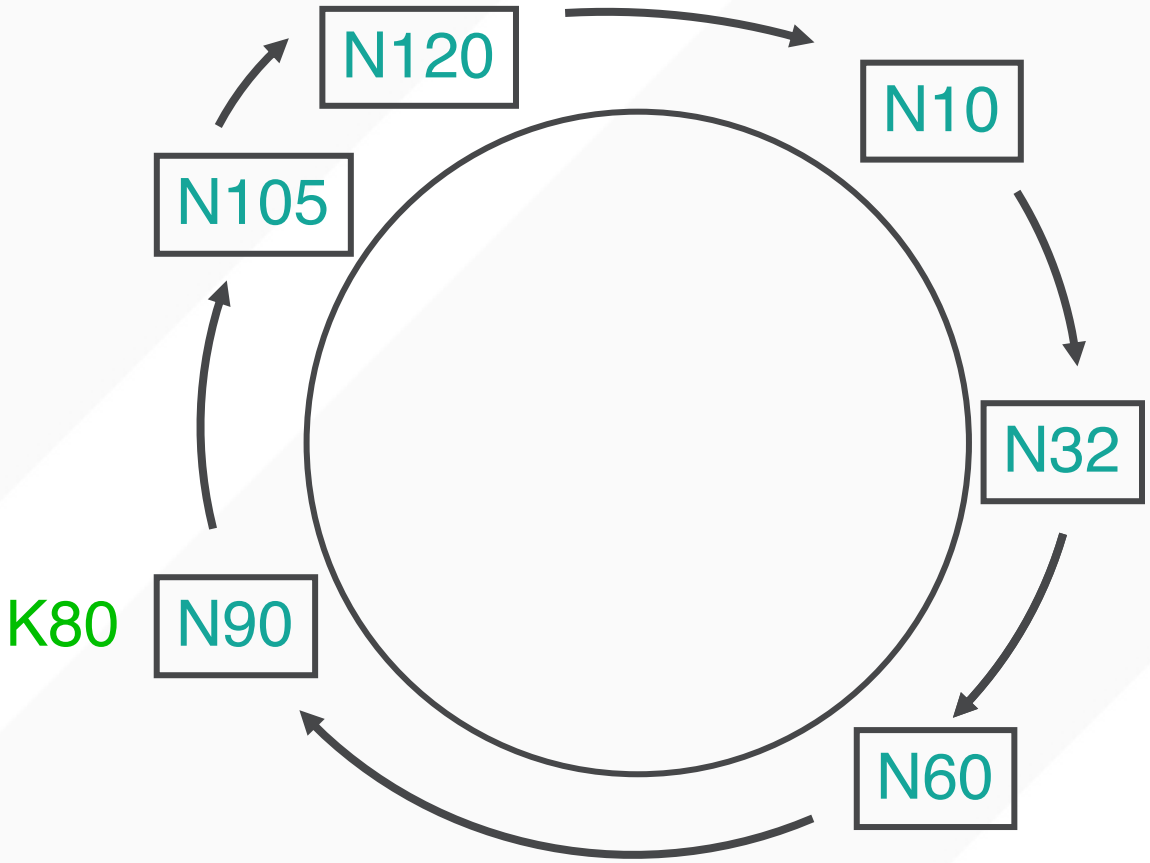
- Desired features –
  - Balance: No bucket has "too many" objects
  - Smoothness: Addition/removal of token minimizes object movements for other buckets

CONSISTENT HASHING [KARGER '97]

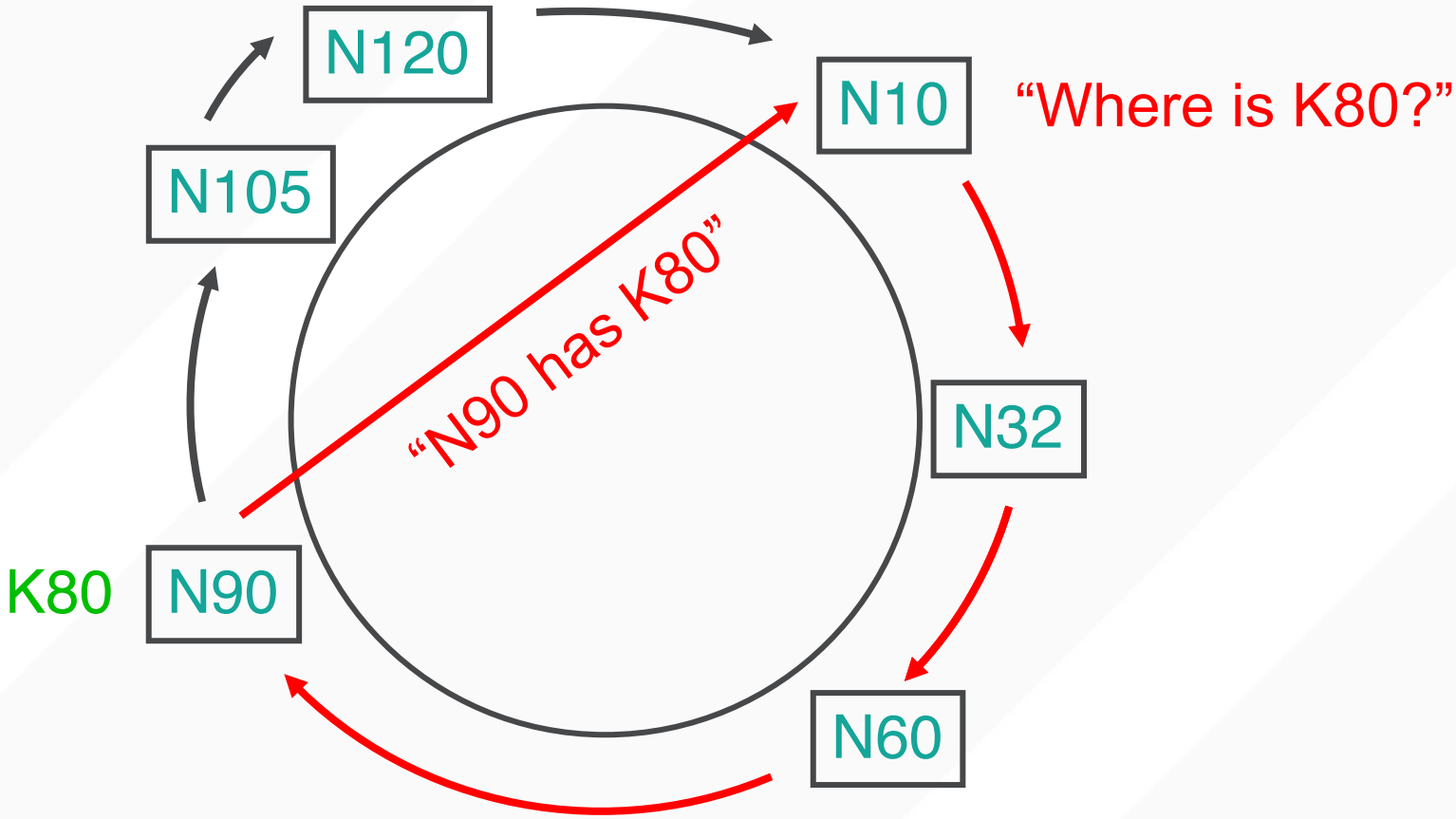


Key is stored at its successor: node with next-higher ID

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**

CONSISTENT HASHING AND LOAD BALANCING

- Each node owns **1/nth** of the ID space in expectation
  - Says nothing of request load per bucket

- If a node fails, its successor takes over bucket
  - **Smoothness goal ✓:** Only localized shift, not O(n)

  - But now successor owns **two** buckets: **2/nth of key space**
    - The failure has **upset the load balance**

VIRTUAL NODES

- **Idea:** Each **physical node** now maintains **v > 1 tokens**
  - Each token corresponds to a ***virtual node***

- Each virtual node owns an expected **$1/(vn)^{th}$** of ID space

- **Upon a physical node's failure, v** successors take over, each now stores **$(v+1)/v \times 1/n^{th}$** of ID space

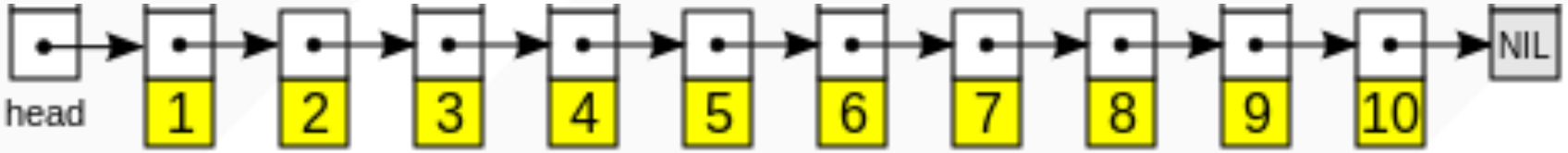
- **Result: Better load balance** with larger v

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

CHORD INTUITION

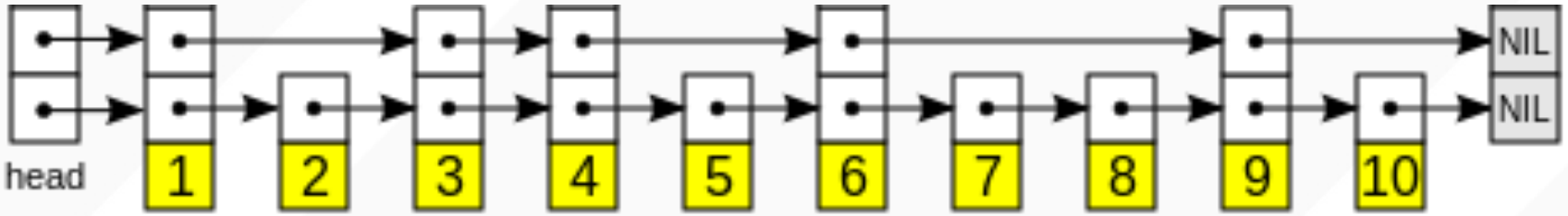
- Skip Lists (Pugh, 1989)
- Consider a linked list:



- Lookup time: O(n)

CHORD INTUITION

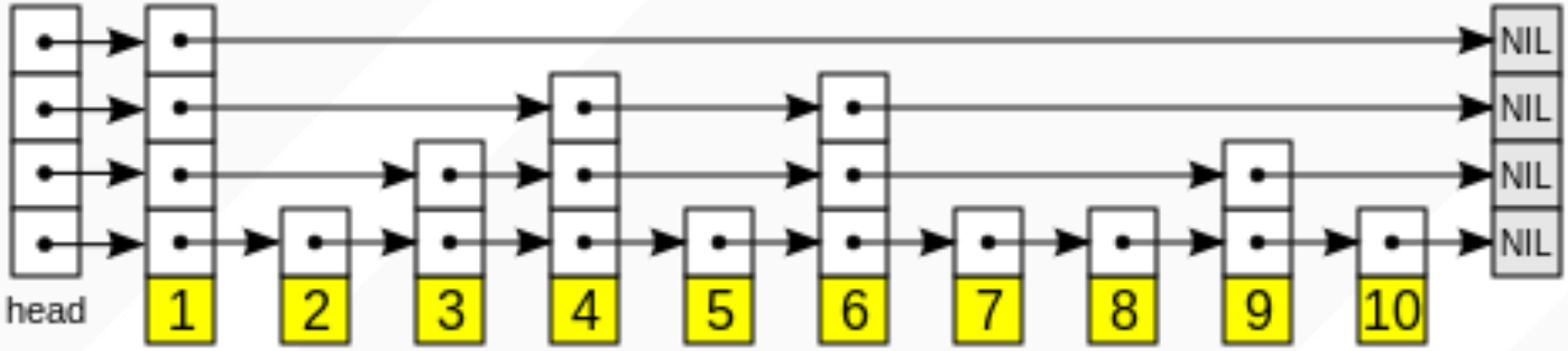
- Skip Lists (Pugh, 1989)
- Consider a linked list:



- Add 2nd row of pointers spaced further apart
  - Still O(n), but more efficient
  - Use 2nd row to get as close as possible without going over
  - Then last row to get to the desired element

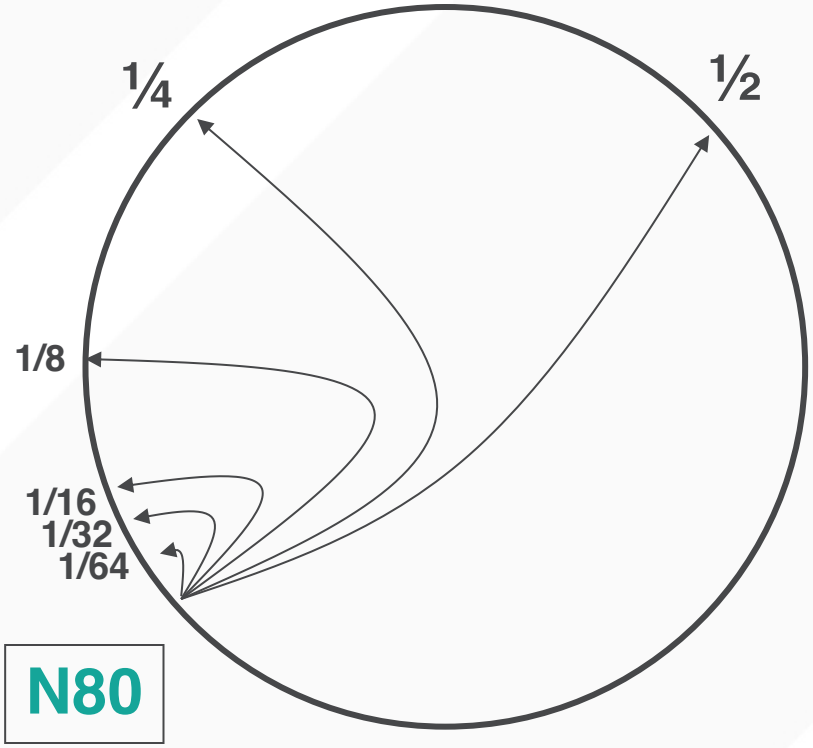
CHORD INTUTION

- Skip Lists (Pugh, 1989)
- Consider a linked list:

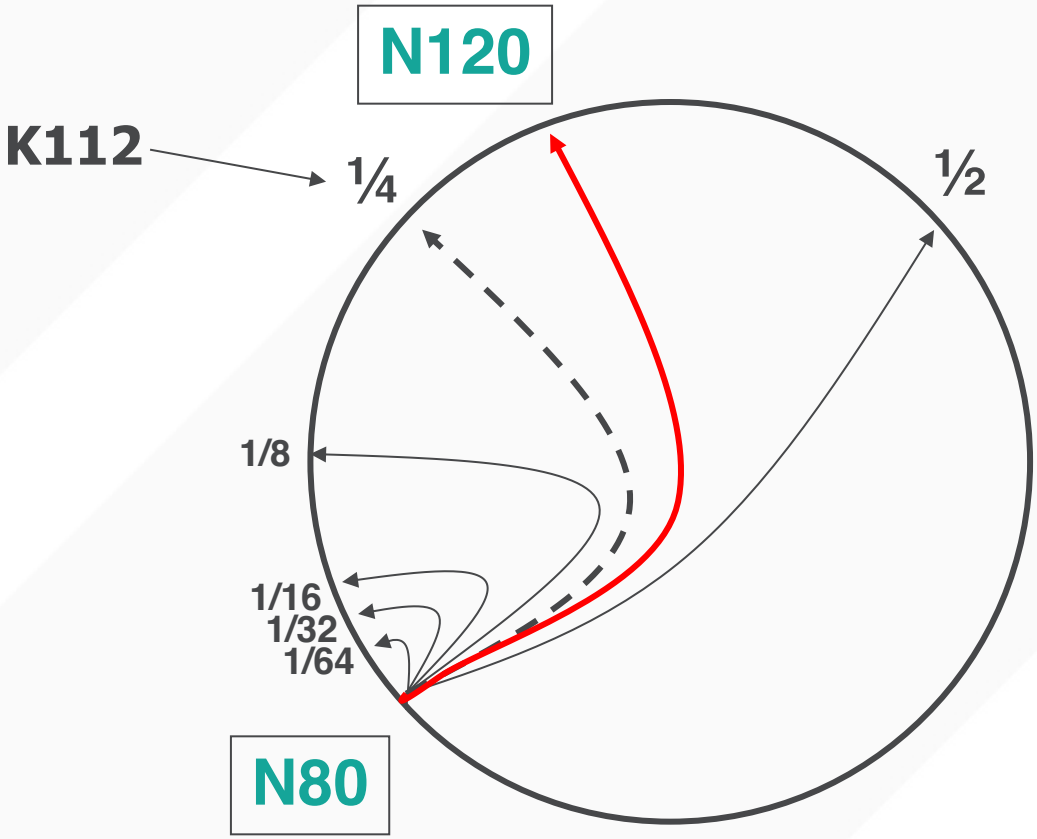


- Add log(N) rows
  - Get as close as possible on top row, then drop down a row, then drop down another row, until the bottom row
  - O(log N) lookup time

“FINGER TABLE” ALLOWS LOG N-TIME LOOKUPS



FINGER I POINTS TO SUCCESSOR OF $N+2^I$



IMPLICATION OF FINGER TABLES

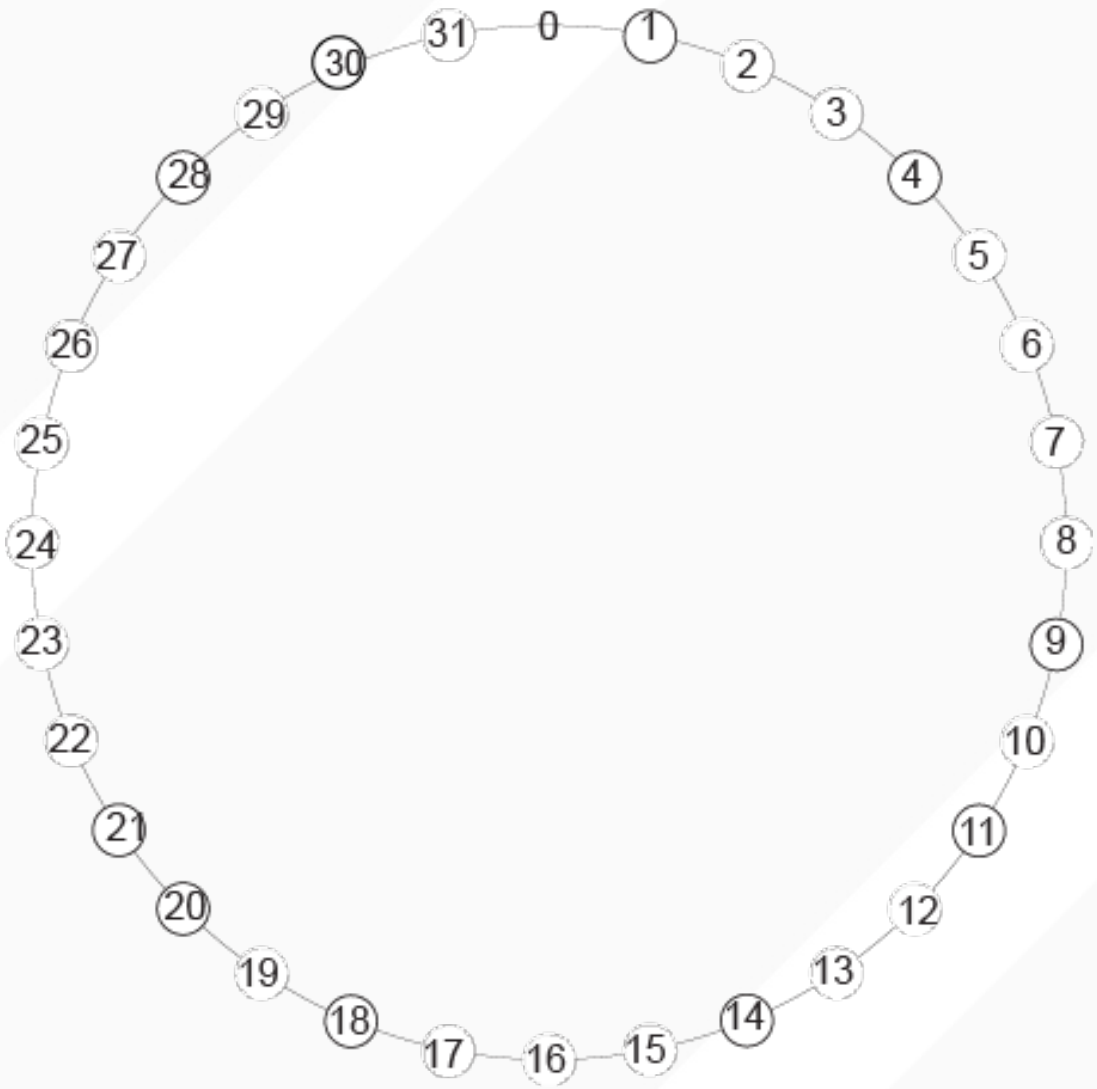
- A **binary lookup tree** rooted at every node
  - Threaded through other nodes' finger tables

- 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

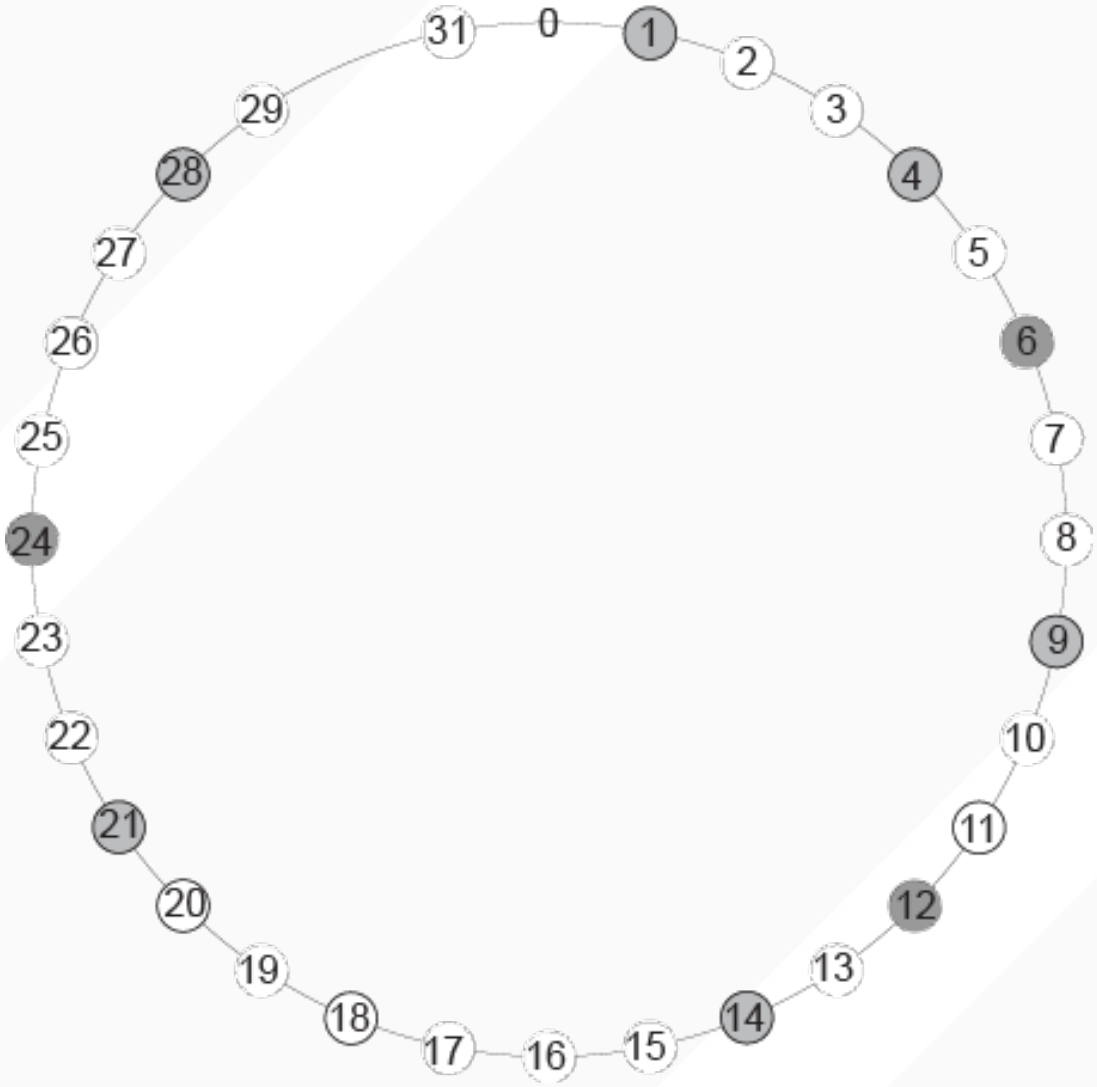
LOOKUP WITH FINGER TABLE

```
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
```

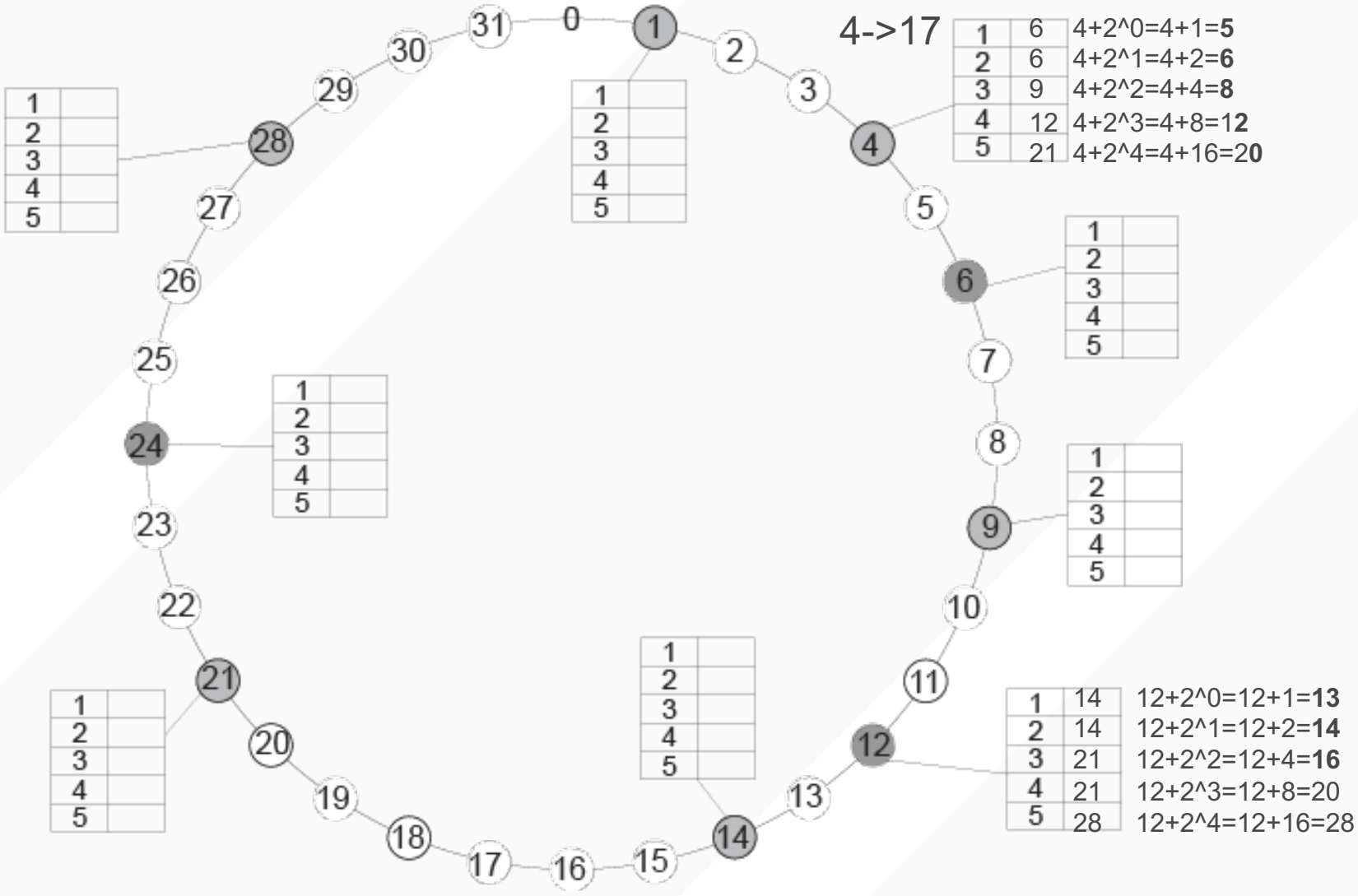
THE CHORD RING (2^5=32)



CHORD RING WITH SERVERS {1,4,6,9,12,14,21,24,28}



ADDING FINGER TABLES



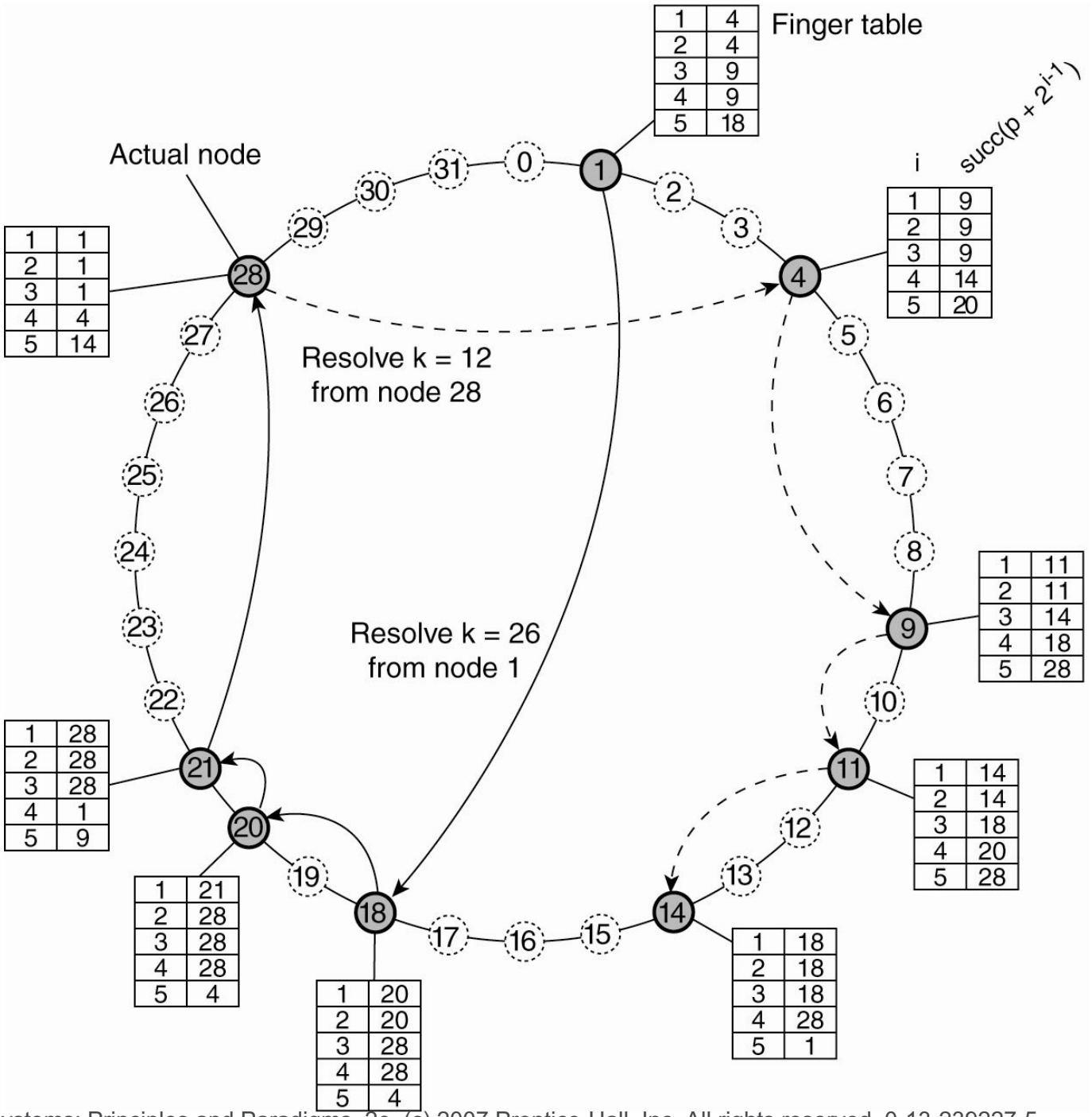
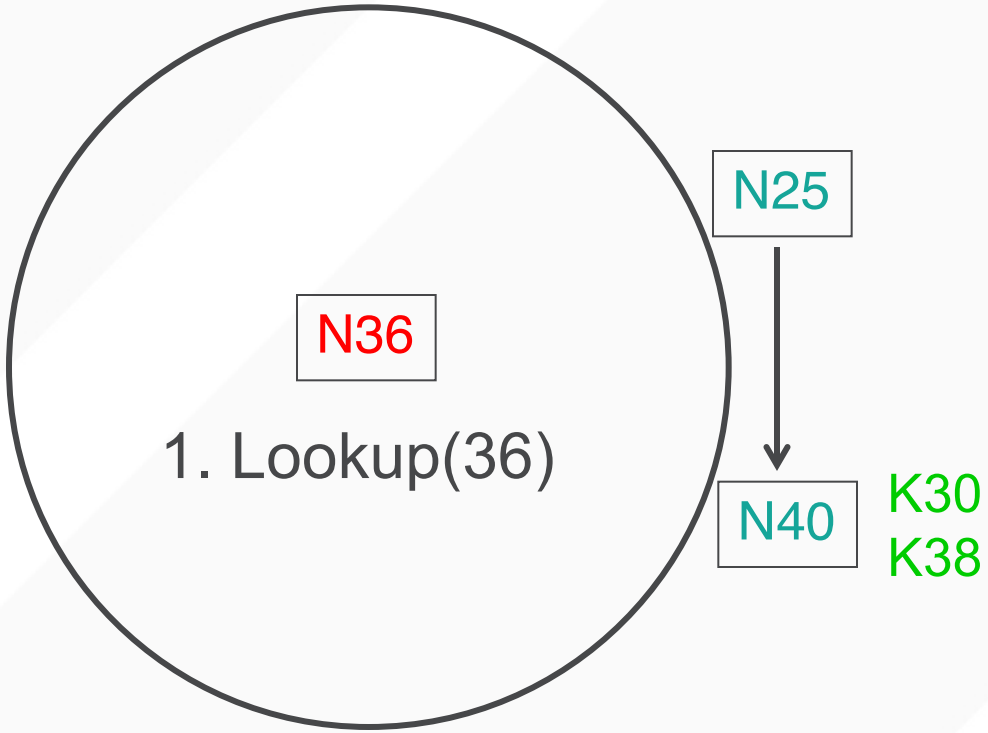


Figure 5-4. Resolving key 26 from node 1 and key 12 from node 28 in a Chord system.

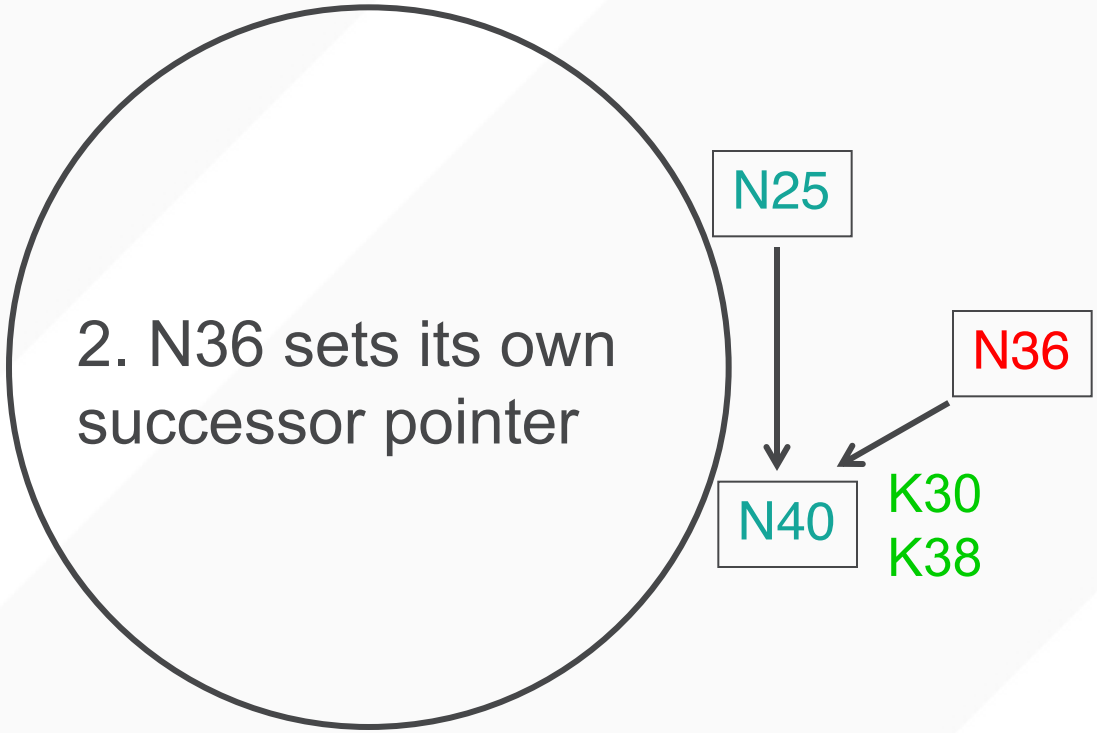
AN ASIDE: IS LOG(N) FAST OR SLOW?

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

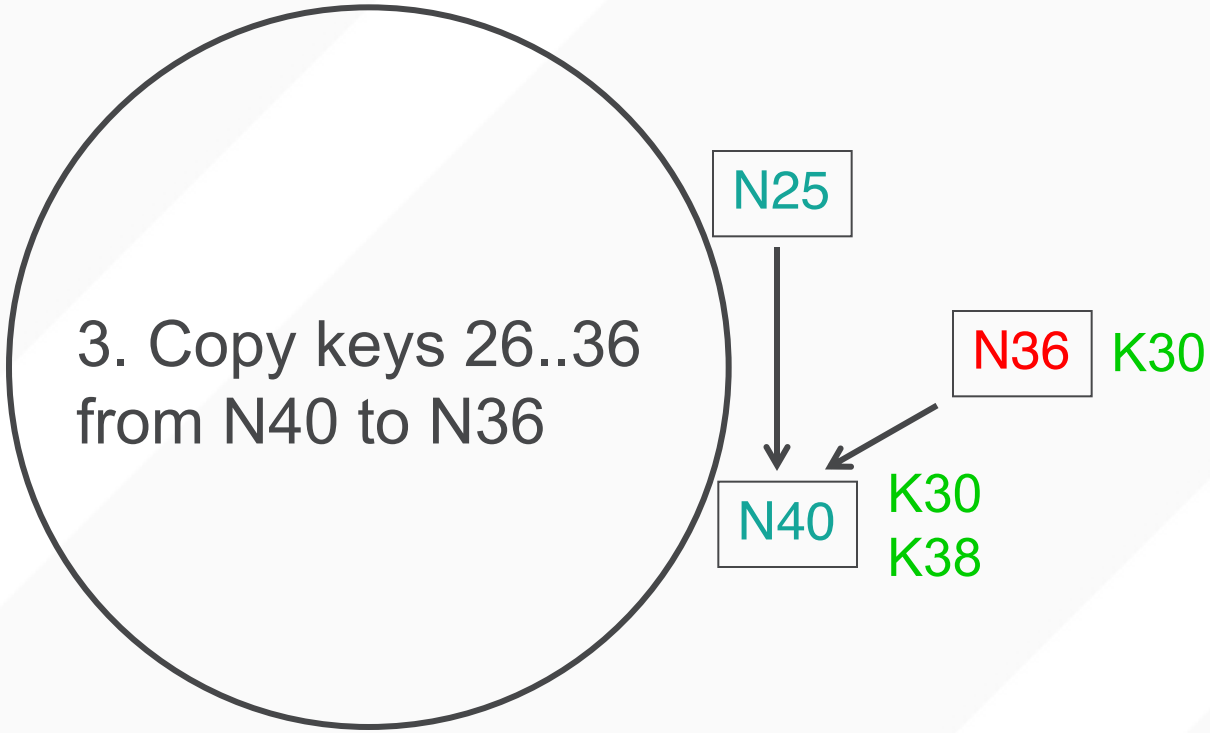
JOINING: LINKED LIST INSERT



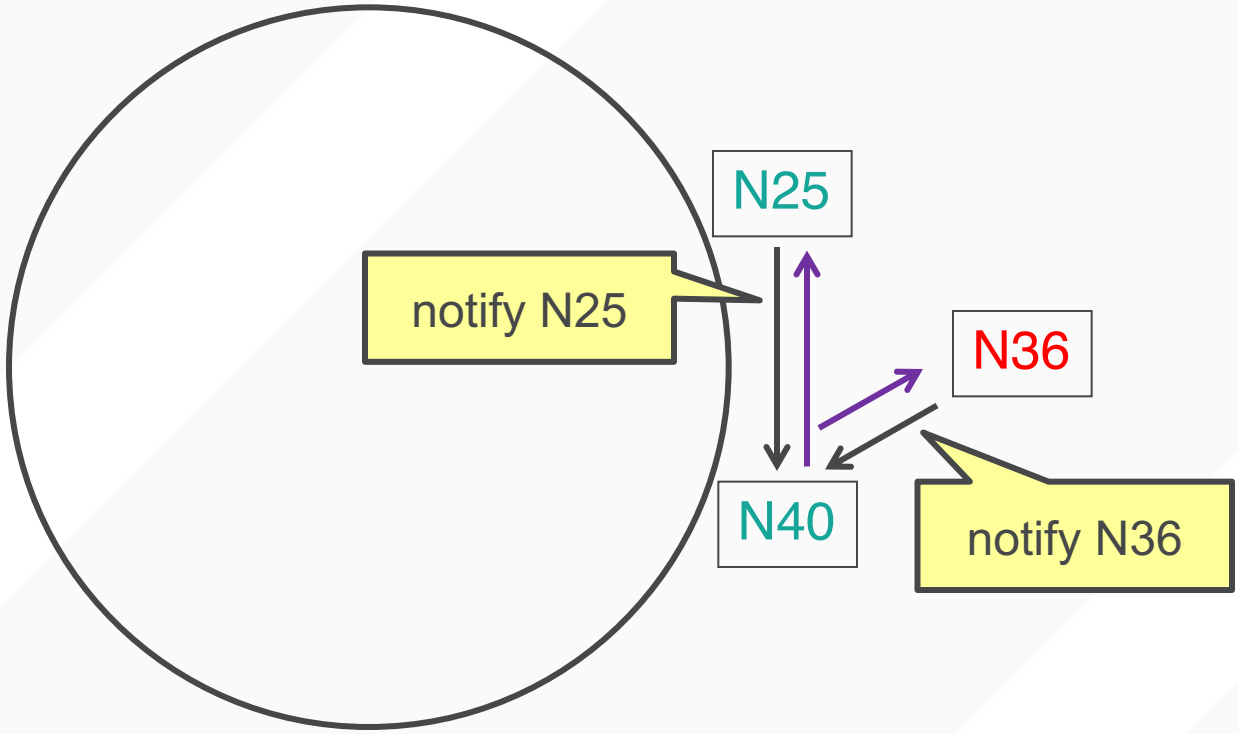
JOIN (2)



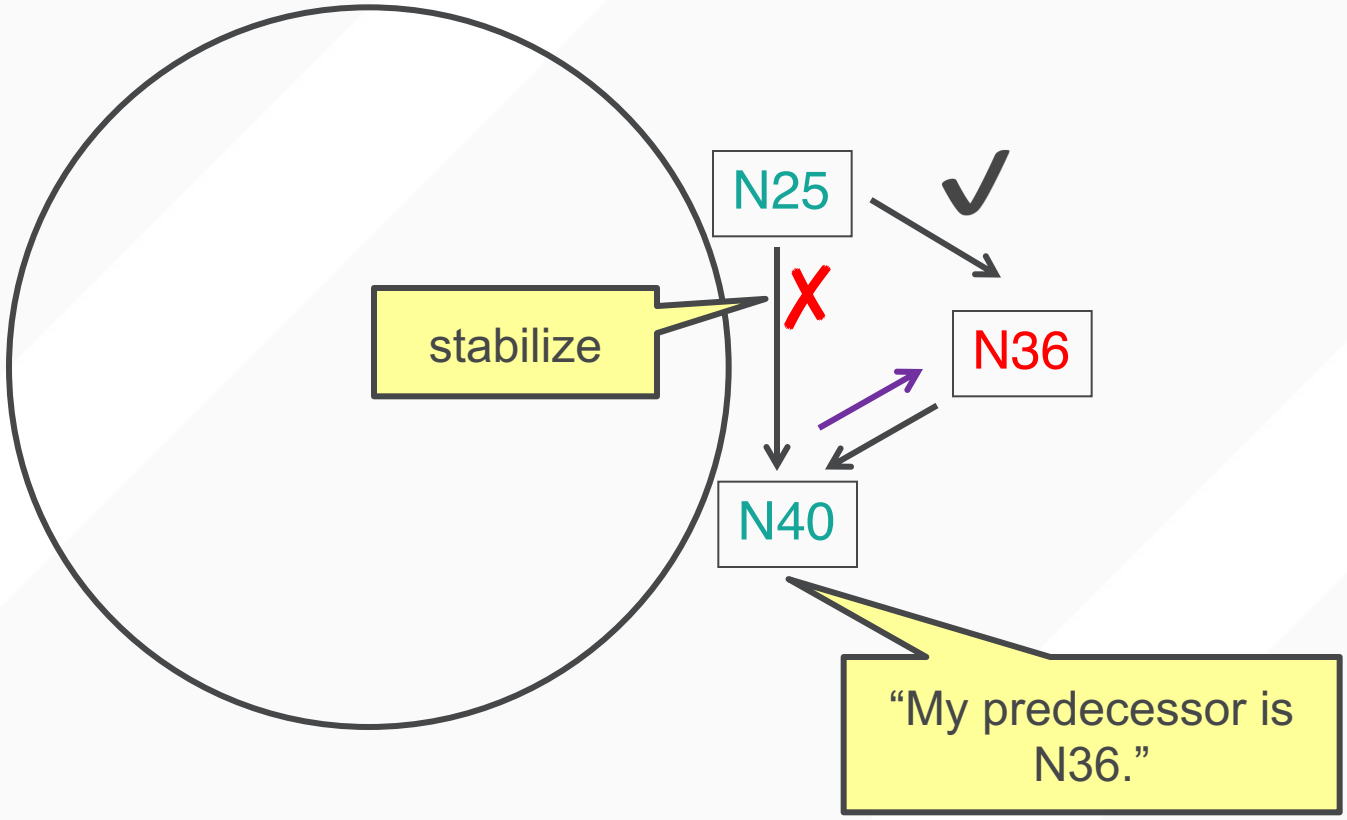
JOIN (3)



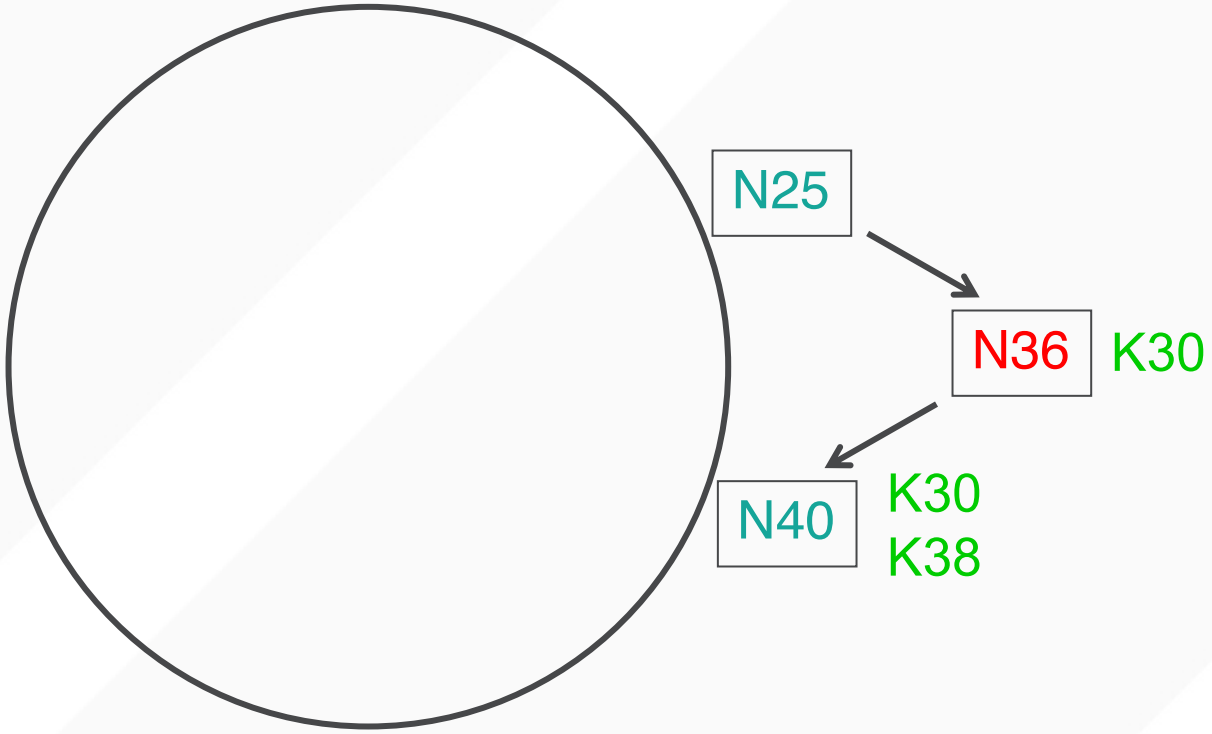
NOTIFY MESSAGES MAINTAIN PREDECESSORS



STABILIZE MESSAGE FIXES SUCCESSOR



JOINING: SUMMARY



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

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 after node failure
- **Incremental scalability:** “add nodes, capacity increases”
- **Self-management:** minimal configuration
- **Unique trait:** no single server to shut down/monitor

UC San Diego