Distributed Systems

CS425/ECE428

Lecture 21

Adopted from Spring 2021

Our agenda for the next 2-3 classes

- Brief overview of key-value stores
- Distributed Hash Tables
  - Peer-to-peer protocol for efficient insertion and retrieval of key-value pairs.
- Key-value stores in the cloud
  - How to run large-scale distributed computations over key-value stores?
    - Map-Reduce Programming Abstraction
  - How to design a large-scale distributed key-value store?
    - Case-study: Facebook's Cassandra

Our focus today

- Brief overview of key-value stores
- Distributed Hash Tables
  - Peer-to-peer protocol for efficient insertion and retrieval of key-value pairs.
- Key-value stores in the cloud
  - How to run large-scale distributed computations over key-value stores?
    - Map-Reduce Programming Abstraction
  - How to design a large-scale distributed key-value store?
    - Case-study: Facebook's Cassandra

The Key-value Abstraction

- Examples of key → value pairs
  - twitter.com: tweet id → information about tweet
  - amazon.com: item number → information about it
  - kayak.com: Flight number → information about it (e.g., seats availability)
  - yourbank.com: Account number → information about it

The Key-value Abstraction (2)

- It’s a dictionary data-structure.
  - Insert, lookup, and delete by key
  - Implemented e.g. as hash table, binary tree
- But here, it is *distributed*.

Isn’t that just a database?

- *Yes, sort of.*
- Relational Database Management Systems (RDBMS) have been around for ages
  - MySQL, PostgreSQL, Oracle, MariaDB, MongoDB, …
- Data stored in structured tables based on *Schema*
  - Each row (data item) in a table has a primary key that is unique within that table.
- Queried using SQL (Structured Query Language).
  - Supports operations on tables and their records

Relational Database Example

users table

user_id	name	zipcode	blog_url	blog_id
101	Alice	12345	alice.net	1
422	Charlie	45783	charlie.com	3
555	Bob	99910	bob.blogspot.com	2

↑ Primary keys ↓

↑ Foreign keys

blog table

id	url	last_updated	num_posts
1	alice.net	5/2/14	332
2	bob.blogspot.com	4/2/13	10003
3	charlie.com	6/15/14	7

Example SQL queries

1. SELECT zipcode
   FROM users
   WHERE name = “Bob”

2. SELECT url
   FROM blog
   WHERE id = 3

3. SELECT users.zipcode, blog.num_posts
   FROM users JOIN blog
   ON users.blog_url = blog.url

Mismatch with today's workloads

- Data: Large and unstructured
- Lots of random reads and writes
- Sometimes write-heavy
- Foreign keys rarely needed
- Table joins infrequent

Key-value/NoSQL Data Model

- NoSQL = “Not Only SQL”
- Necessary API operations: get(key) and put(key, value)

- Tables
  - Like RDBMS tables, but …
  - May be unstructured: May not have schemas
    - Some columns may be missing from some rows
  - Don’t always support joins or have foreign keys
  - Can have index tables, just like RDBMSs

Key-value/NoSQL Data Model

- Unstructured
- No schema imposed
- Columns missing from some Rows
- No foreign keys, joins may not be supported

Key

Value

users table

user_id	name	zipcode	blog_url
101	Alice	12345	alice.net
422	Charlie		charlie.com
555		99910	bob.blogspot.com

Key

Value

blog table

id	url	last_updated	num_posts
1	alice.net	5/2/14	332
2	bob.blogspot.com		10003
3	charlie.com	6/15/14	

Our focus today

- Brief overview of key-value stores
- Distributed Hash Tables
  - Peer-to-peer protocol for efficient insertion and retrieval of key-value pairs.
- Key-value stores in the cloud
  - How to run large-scale distributed computations over key-value stores?
    - Map-Reduce Programming Abstraction
  - How to design a large-scale distributed key-value store?
    - Case-study: Facebook's Cassandra

Distributed Hash Tables (DHTs)

- Multiple protocols were proposed in early 1990s.
  - Chord, CAN, Pastry, Tapestry
  - Initial use-case: Peer-to-peer file sharing
    - key = hash of the file, value = file
  - Cloud-based distributed key-value stores reuse many techniques from these DHTs.

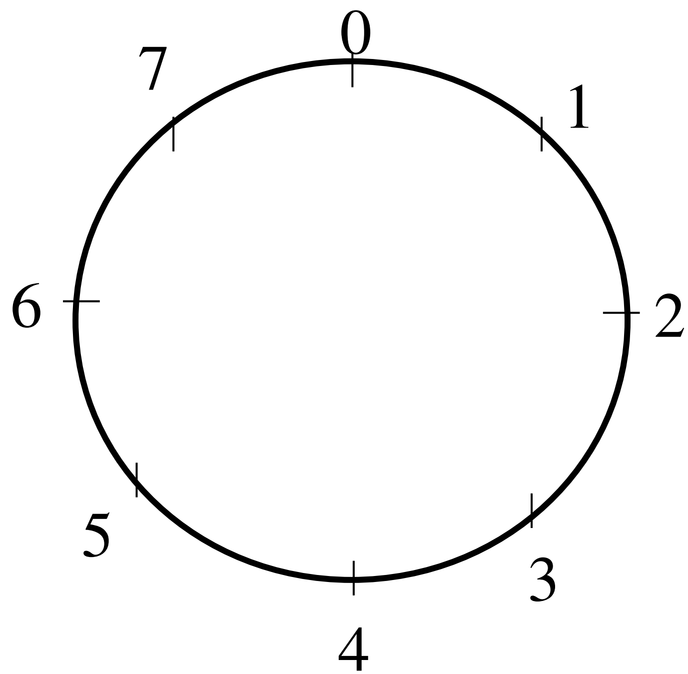
- Key goals:
  - Balance load uniformly across all nodes (peers).
  - Fault-tolerance
  - Efficient inserts and lookups.

Chord

- Developed at MIT by I. Stoica, D. Karger, F. Kaashoek, H. Balakrishnan, R. Morris, Berkeley and MIT
- Key properties:
  - Load balance:
    - spreads keys evenly over nodes.
  - Decentralized:
    - no node is more important than others.
  - Scalable:
    - cost of the key lookup is O(logN), N = no. of nodes.
  - High availability:
    - automatically adjusts to nodes joining and nodes leaving.
  - Flexible naming:
    - no constraints on the structure of keys to look up.

Chord: Consistent Hashing

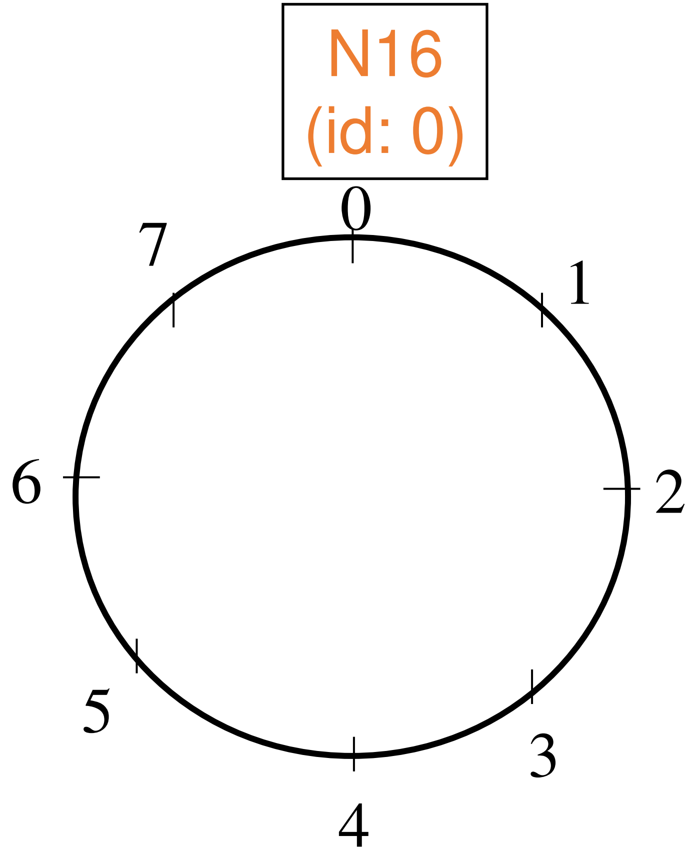
- Uses Consistent Hashing on node's (peer's) address
  - SHA-1(ip_address,port) →160 bit string
  - Truncated to m bits (modulo 2^m)
  - Called peer id (number between 0 and 2^m-1)
  - Not unique but id conflicts very unlikely
  - Can then map peers to one of 2^m logical points on a circle



Circle for m = 3

Chord: Consistent Hashing

- Uses Consistent Hashing on node's (peer's) address
  - SHA-1(ip_address,port) →160 bit string
  - Truncated to m bits (modulo 2^m)
  - Called peer id (number between 0 and 2^m-1)
  - Not unique but id conflicts very unlikely
  - Can then map peers to one of 2^m logical points on a circle

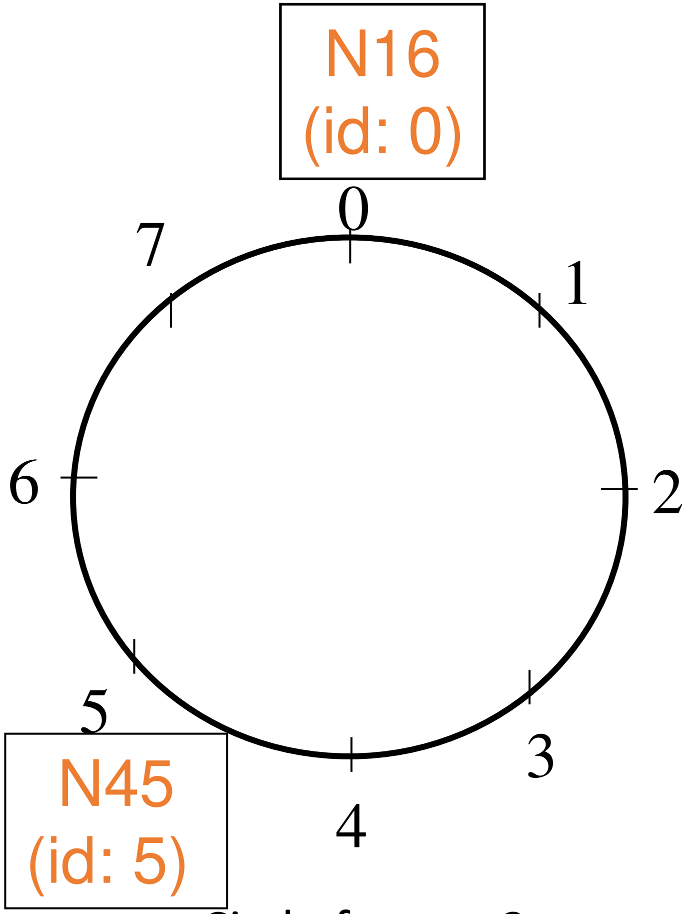


Circle for m = 3

Where will N16 be placed on this circle?

Chord: Consistent Hashing

- Uses Consistent Hashing on node's (peer's) address
  - SHA-1(ip_address,port) →160 bit string
  - Truncated to m bits (modulo 2^m)
  - Called peer id (number between 0 and 2^m-1)
  - Not unique but id conflicts very unlikely
  - Can then map peers to one of 2^m logical points on a circle

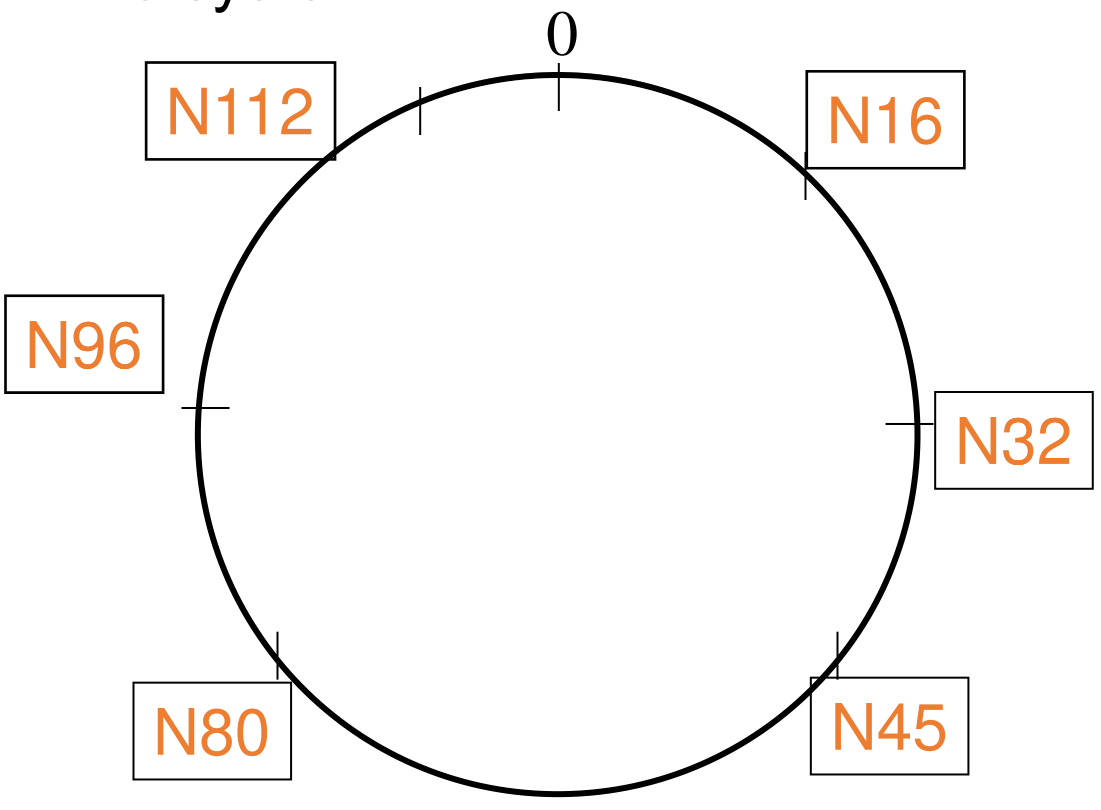


Circle for m = 3

Where will N45 be placed on this circle?

Ring of Peers: Running Example

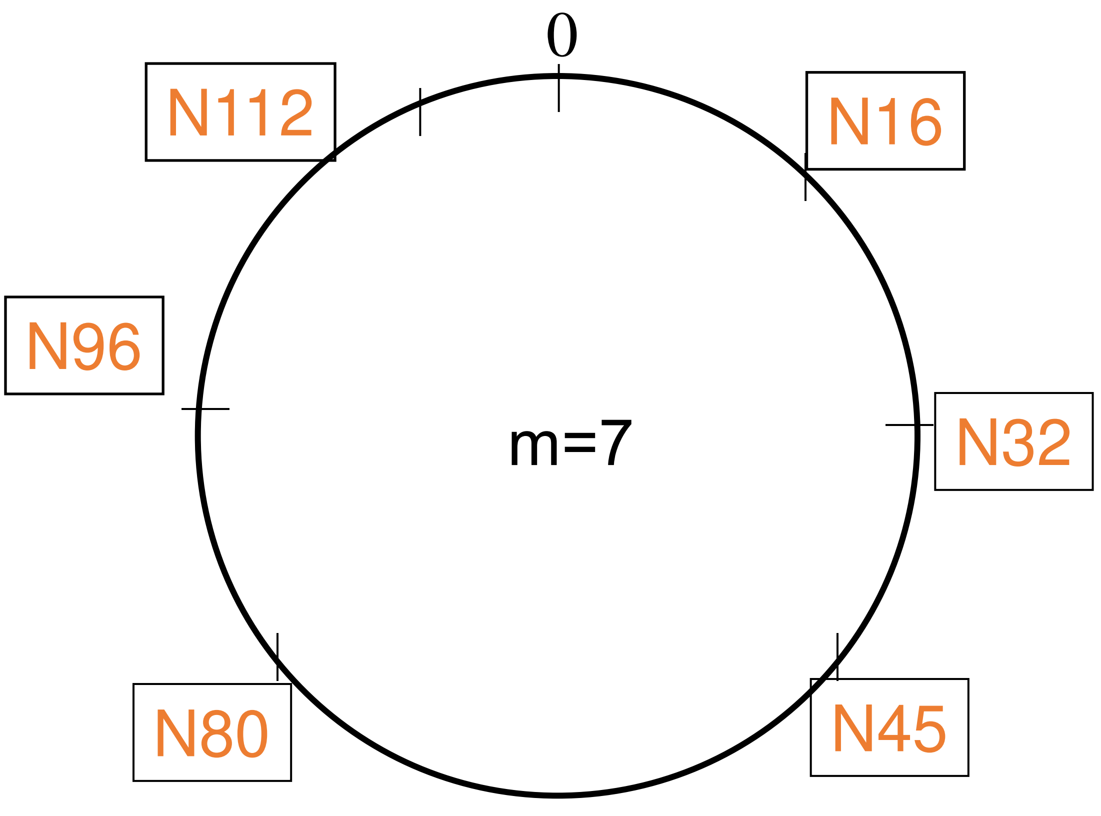
- Say m=7 (128 possible points on the circle – not shown)
- 6 nodes in the system.



Mapping Keys to Nodes

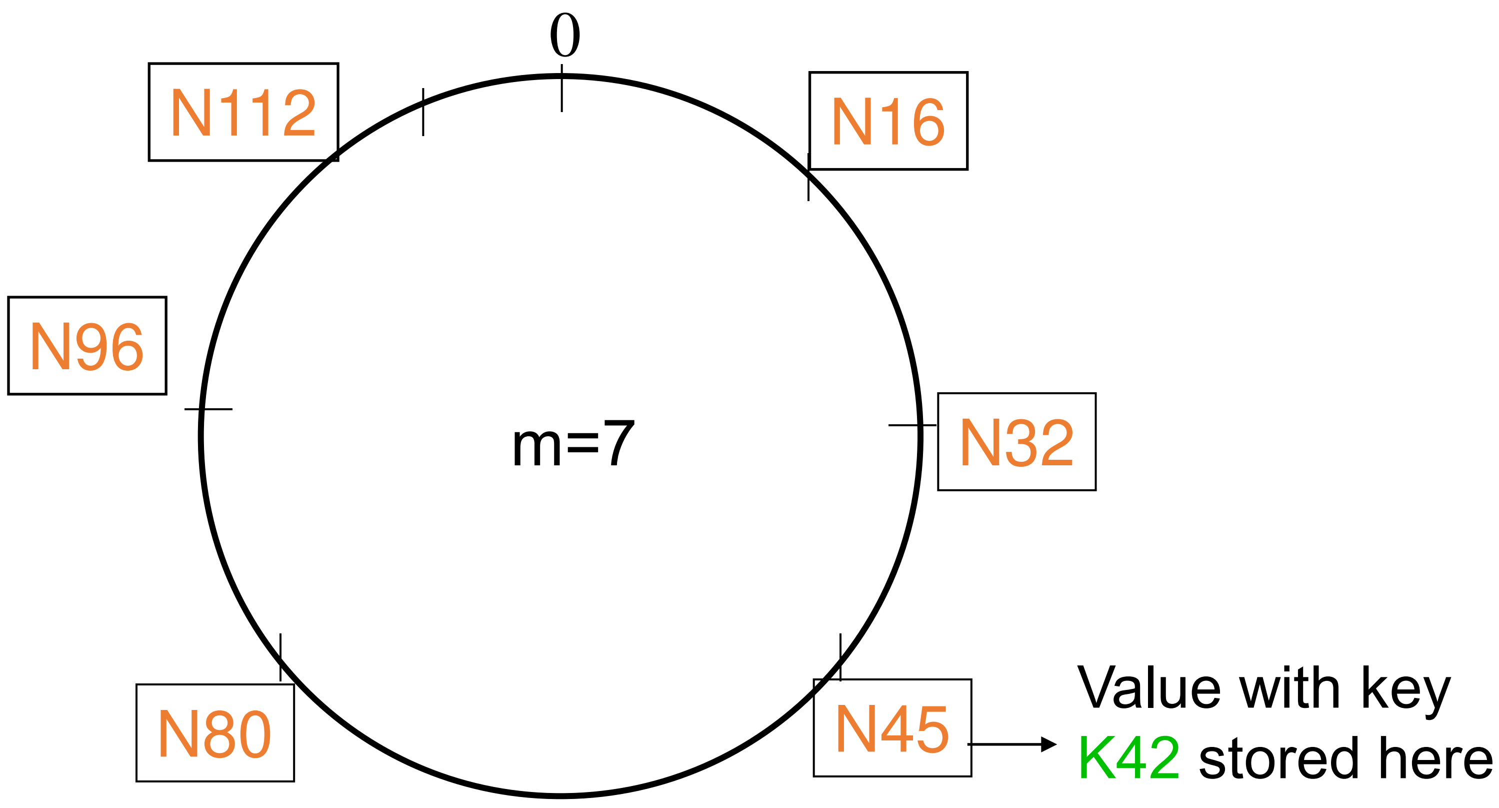
- Use the same consistent hash function
  - SHA-1(key) → 160 bit string (key identifier)
    - Henceforth, we refer to SHA-1(key) as *key*.
  - The key-value pair stored at the key's *successor* node.
  - *successor(key) = first peer with id greater than or equal to (key mod 2^m)*
    - *Cross-over the ring when you reach the end.*
      - *0 < 1 < 2 < 3 …….. < 127 < 0 (for m=7)*
- Consistent Hashing => with K keys and N peers, each peer stores O(K/N) keys. (i.e., c *x* K/N, for some constant c)

Ring of Peers: Running Example



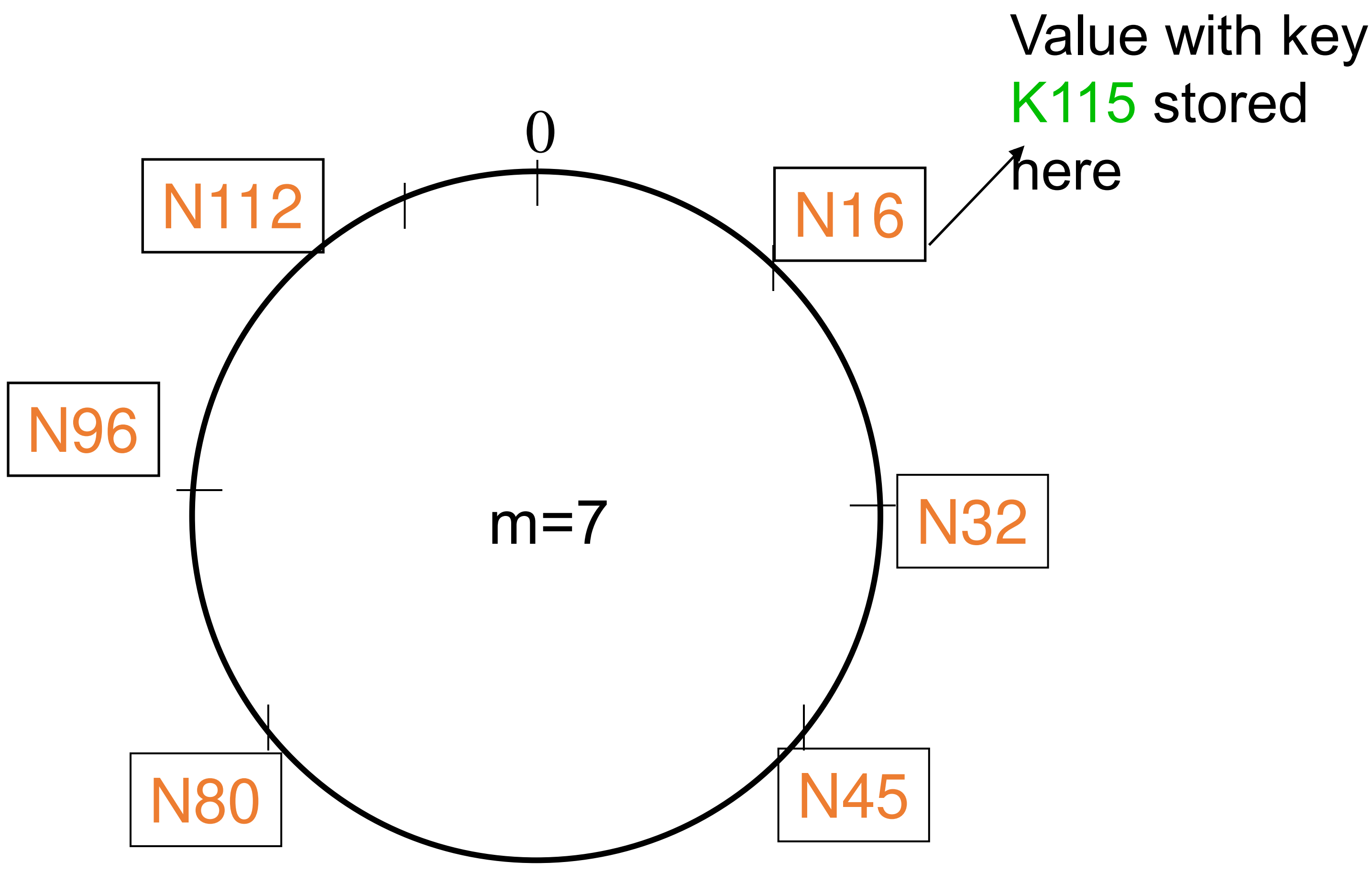
Where will the value with key 42 be stored?

Ring of Peers: Running Example



Where will the value with key 42 be stored?

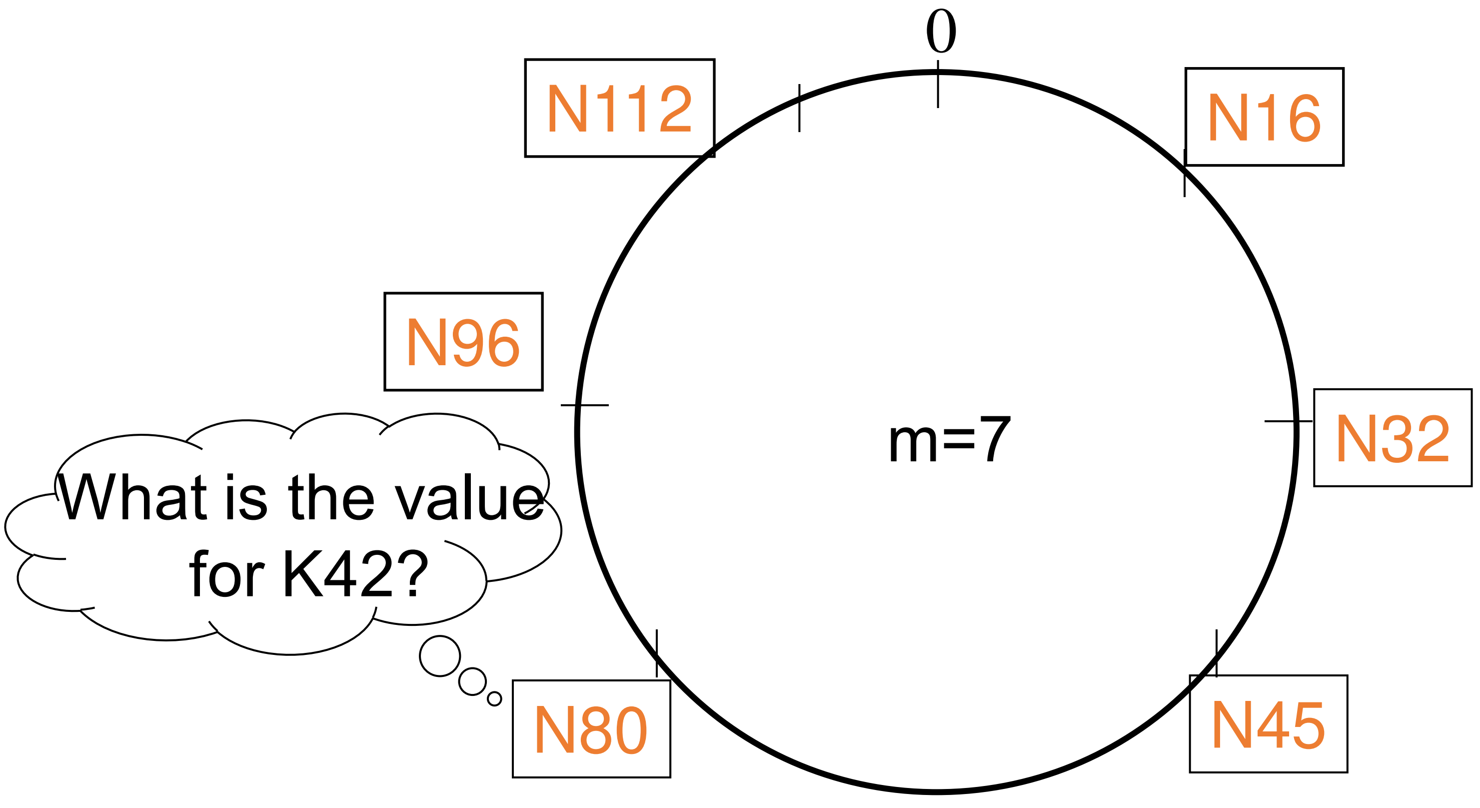
Ring of Peers: Running Example



Where will the value with key 115 be stored?

Performing Lookups

Suppose N80 receives a request to lookup K42.



Need to ask the successor of K42!

Performing Lookups

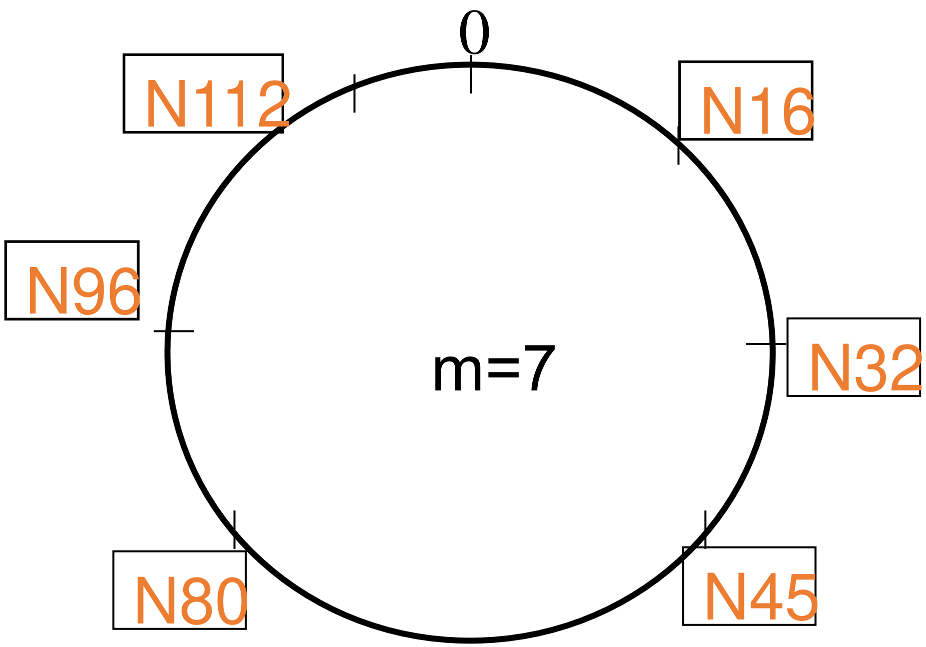
- Option 1: Each node is aware of (can route to) any other node in the system.
  - Need a very large routing table.
  - Poor scalability with 1000s of nodes.
  - Any node failure and join will require a *necessary* update at all nodes.
- Option 2: Each node is aware of only its ring successor.
  - O(N) lookup. Not very efficient.
- Chord chooses a middle-ground.

Performing Lookups

- Chord chooses a middle-ground.
  - Each node is aware of m other nodes.
  - Maintains a *finger table* with m entries.
  - The i-th entry of node n’s finger table = successor($n + 2^i$)
    - i ranges from 0 to m-1

Finger Tables

Compute the finger table for N80

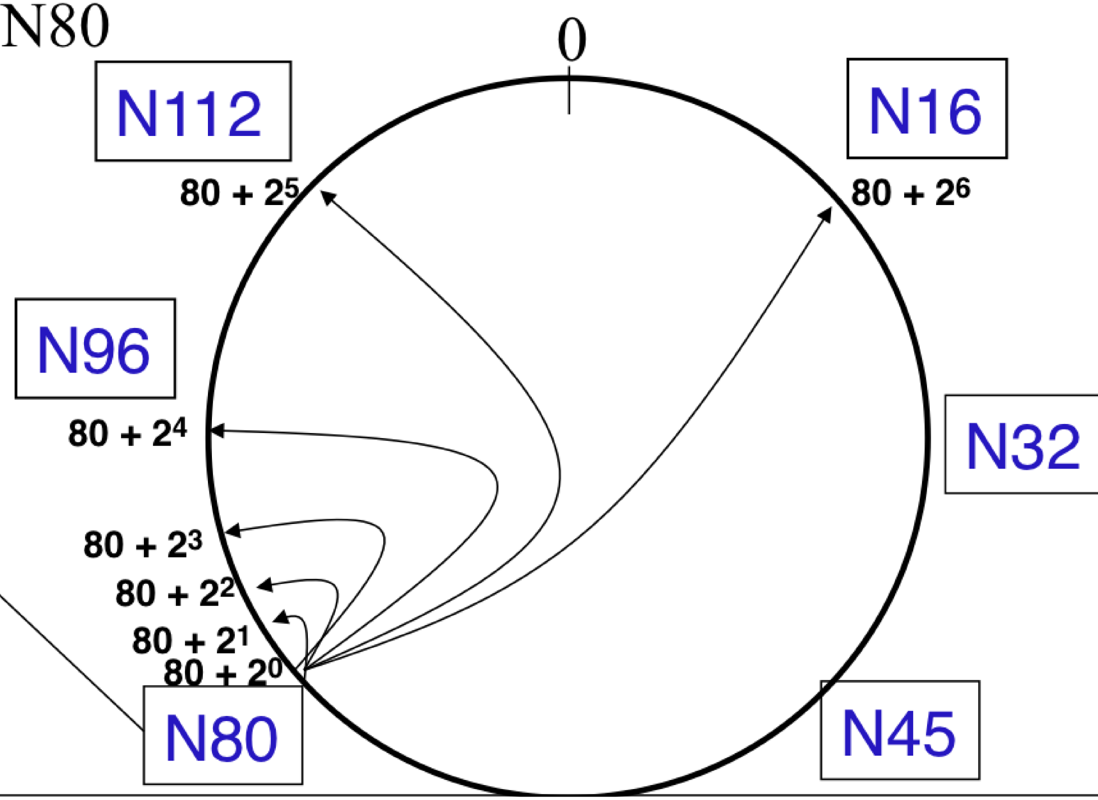


Finger Tables

Say $m=7$

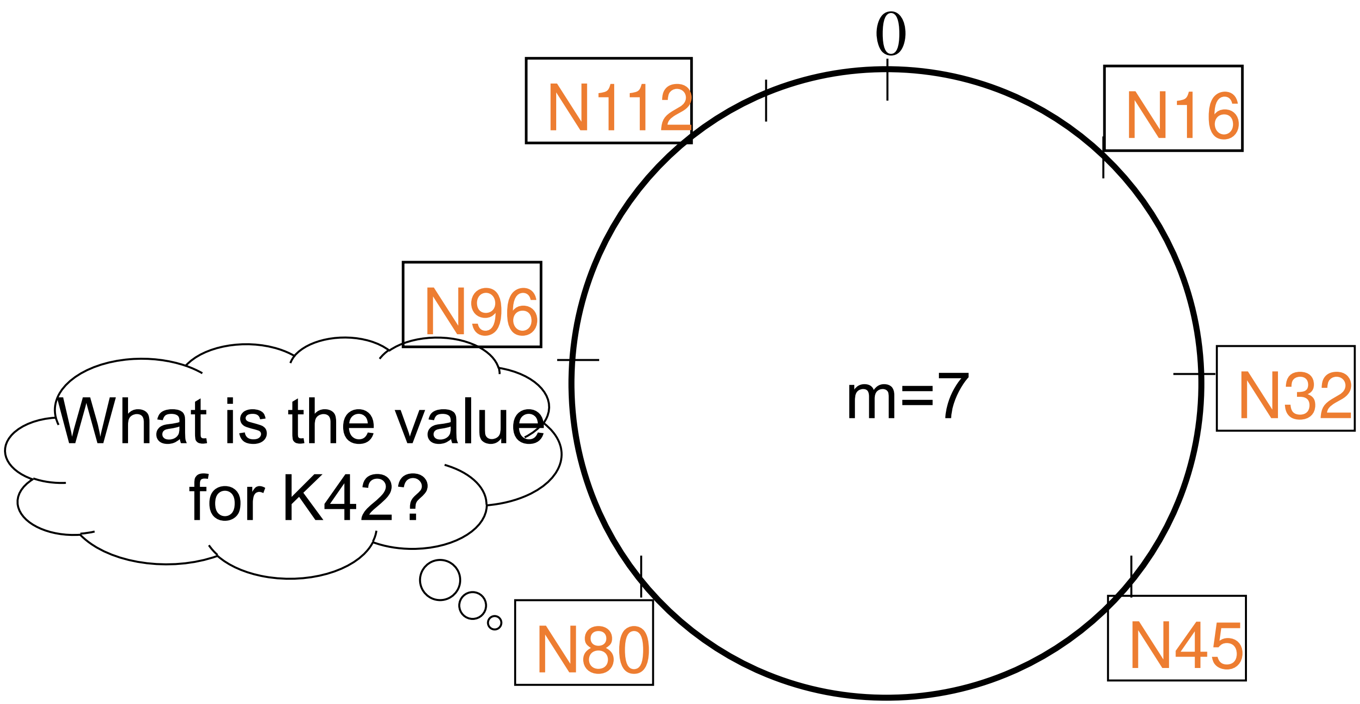
Finger Table at N80

i	$ft[i]$
0	96
1	96
2	96
3	96
4	96
5	112
6	16



Performing Lookups

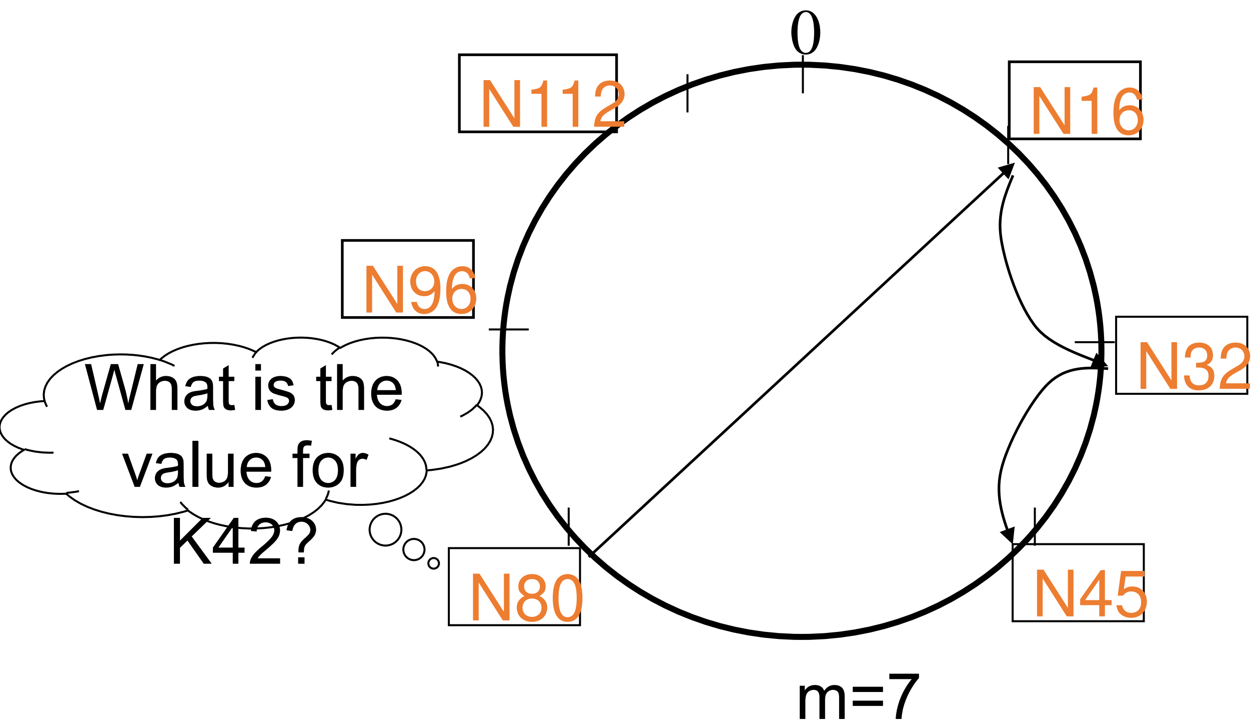
Suppose N80 receives a request to lookup K42.



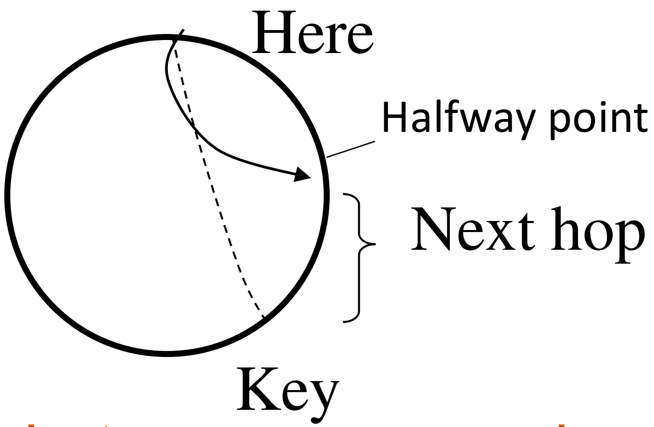
Need to locate successor of K42!
Forward the query to the most promising node.

Search for key k at node n

At node n, if k lies in range (n, next(n)], where next(n) is n's *ring successor,* then next(n) = successor(key), so send the query to next(n). Else, send the query to largest finger entry <= k



Analysis



The search for key takes O(log(N))

Proof Intuition:

- (intuition): at each step, distance between query and peer-with-file reduces by a factor of at least 2 (why?)
- (intuition): after log(N) forwardings, distance to key is at most $2^m/2^{\log(N)} = 2^m / N$
- Expected number of node identifiers in a range of $2^m / N$:
  - ideally one
  - O(log(N)) with high probability (for consistent hashing)

So using ring successors in that range will use another O(log(N)) hops. Overall lookup time stays O(log(N)).

Analysis

- O(log(N)) search time holds for file insertions too (in general for routing to any key)
  - "Routing" can thus be used as a building block for
    - all operations including: insert, lookup, delete
- O(log(N)) time is true only if the finger and successor entries are correct
- When might these entries be wrong?
  - When you have failures
    - Next class!