Distributed Systems

CS425/ECE428

April 10 2023

Instructor: Radhika Mittal

Acknowledgements for some of the materials: Indy Gupta

Logistics

- HW4 is due today!
- HW5 has been released.
- Final exam: May 4-11.
  - Please reserve a slot on PrairieTest.
  - Same format as your midterm, but longer.
  - It will be comprehensive – everything covered in class.
- I am traveling next week:
  - April 17: Lecture will be delivered by Sarthak Moorjani.
    - Topic: MapReduce (part of your exam syllabus)
  - April 19: Office hour/QA session with the course TAs in DCL 1320.

Our agenda for the next 3-4 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

- (Business) Key → Value
  - (twitter.com) tweet id → information about tweet
  - (amazon.com) item number → information about it
  - (kayak.com) Flight number → information about flight, e.g., 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
  - E.g., hash table, binary tree

- But *distributed*.

Isn't that just a database?

- *Yes, sort of.*
- Relational Database Management Systems (RDBMSs) have been around for ages
  - e.g. MySQL is the most popular among them
- Data stored in structured tables based on a *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 joins.

Mismatch with today's workloads

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

- 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

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 usecase: 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.

Distributed Hash Tables (DHTs)

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

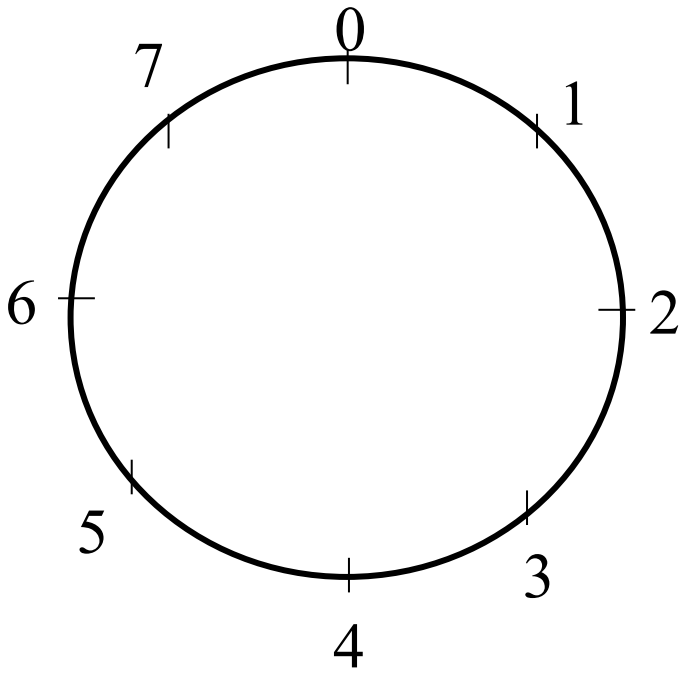
- Key goals:
  - Balance load uniformly across all nodes (peers).
  - Fault-tolerance
  - Efficient inserts and lookups.

Chord

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

Chord: Consistent Hashing

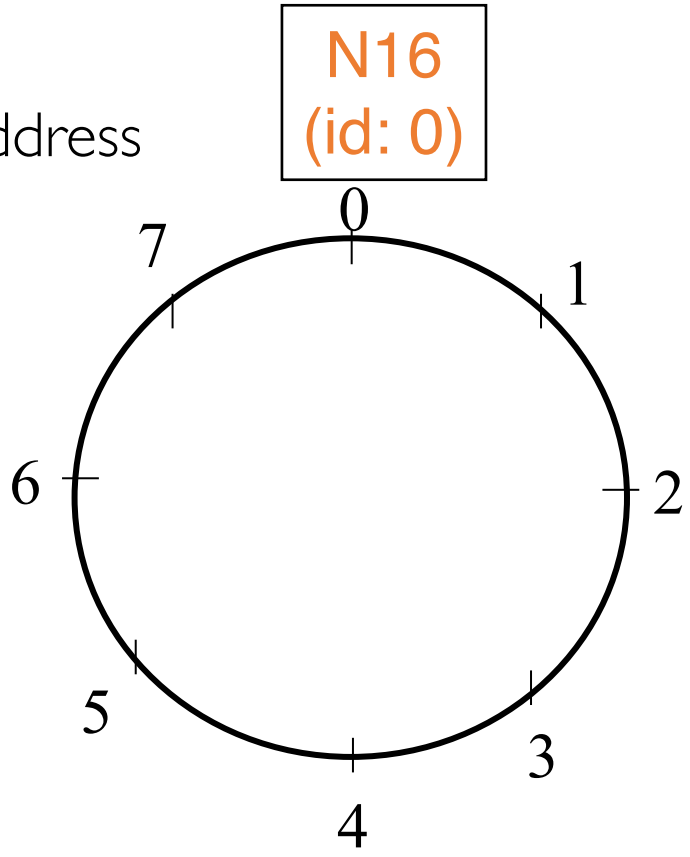
- Uses *Consistent Hashing* on node's (peer's) address
  - SHA-1 (ip_address,port) → 160 bit string
  - Truncated to **m** bits (modulo 2^m)
  - Called peer id (number between 0 and 2^m-1)
  - **m** chosen such that negligible chance of id conflicts
  - Can then map peers to one of 2^m logical points 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)
  - **m** chosen such that negligible chance of id conflicts
  - Can then map peers to one of 2^m logical points on a circle

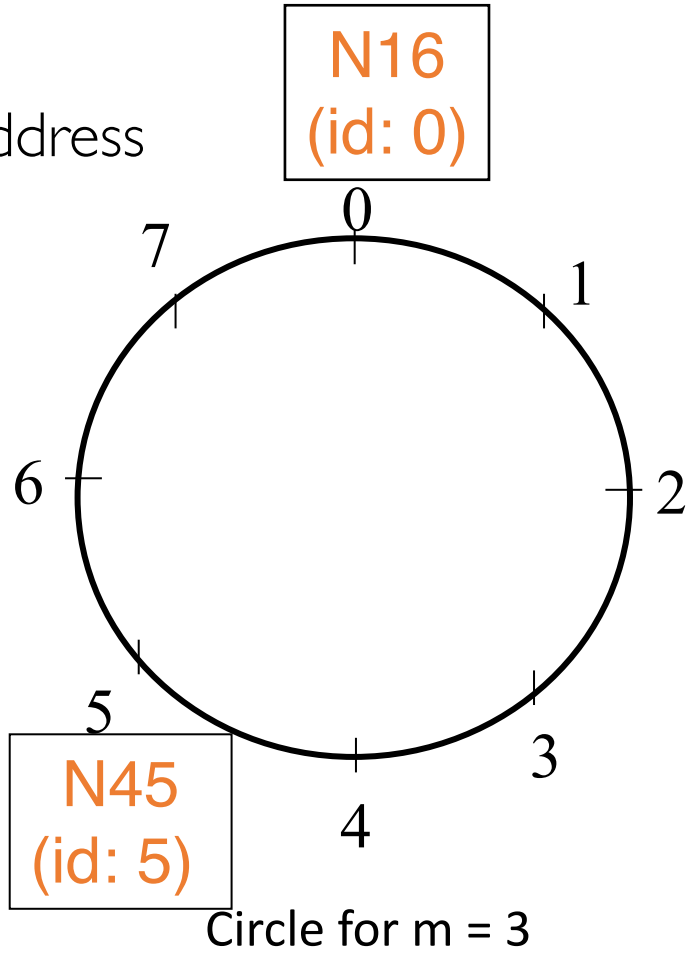


Circle for m = 3

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)
  - **m** chosen such that negligible chance of id conflicts
  - Can then map peers to one of 2^m logical points on a circle

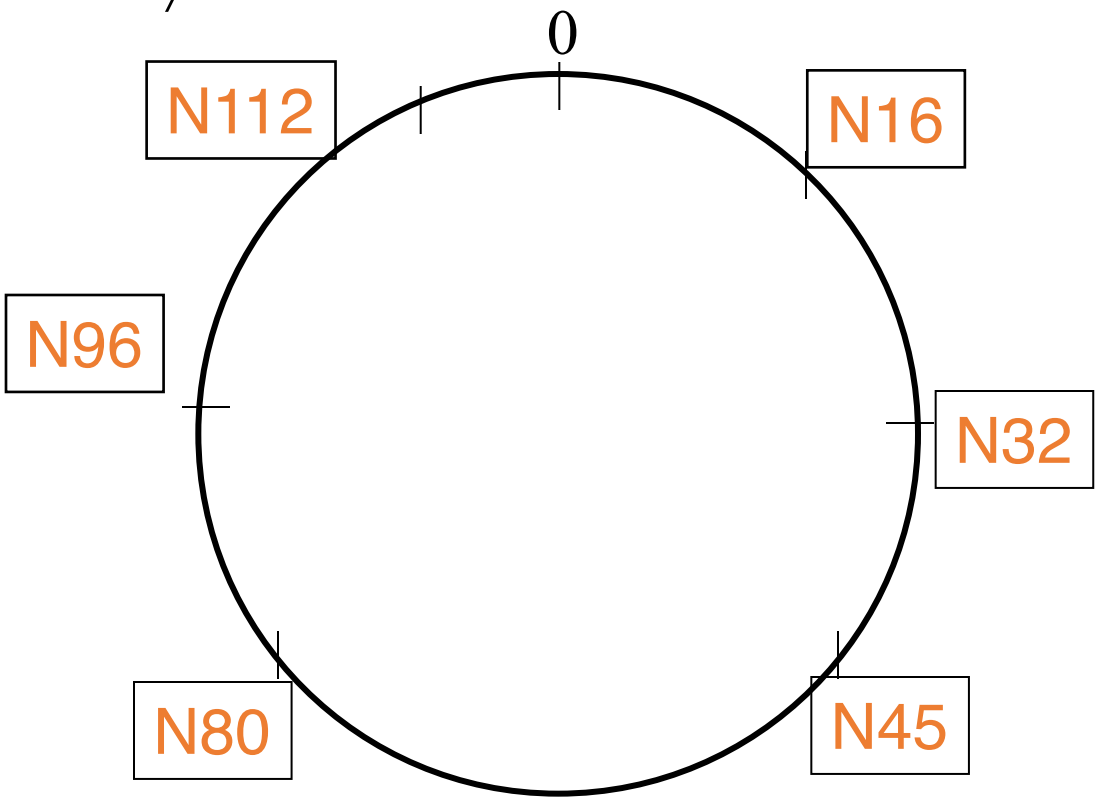


Circle for m = 3

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.

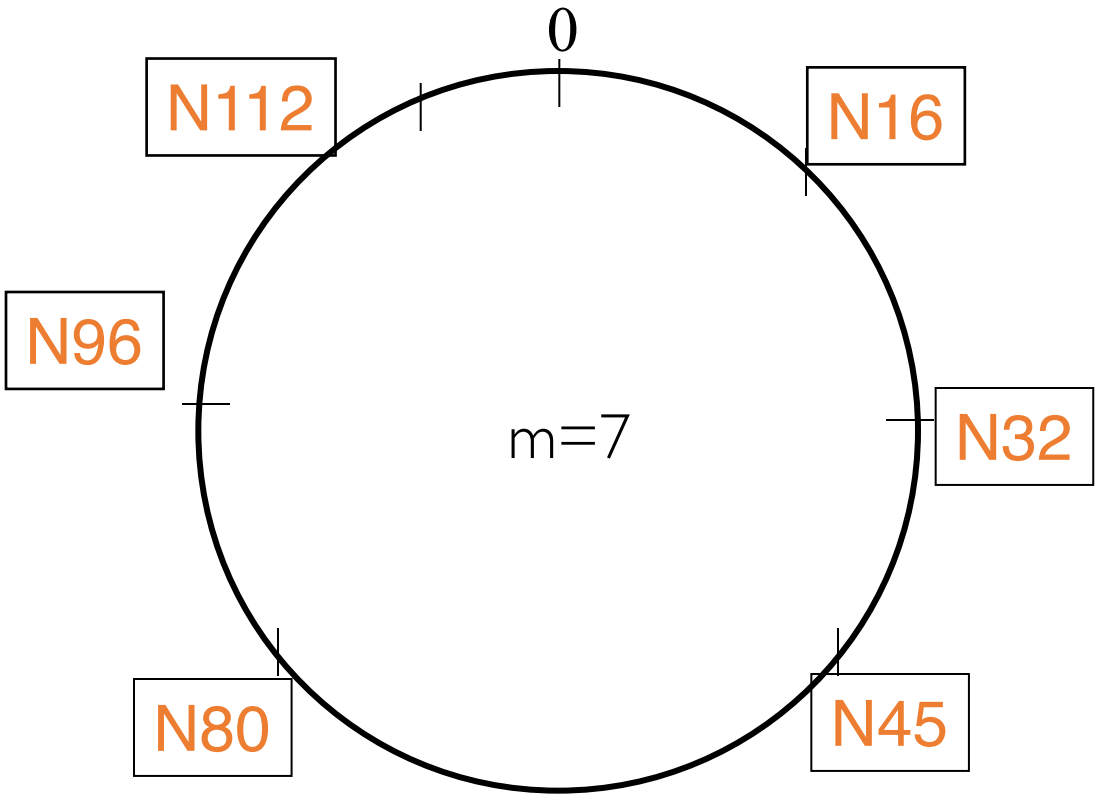


Mapping Keys to Nodes

- 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)*

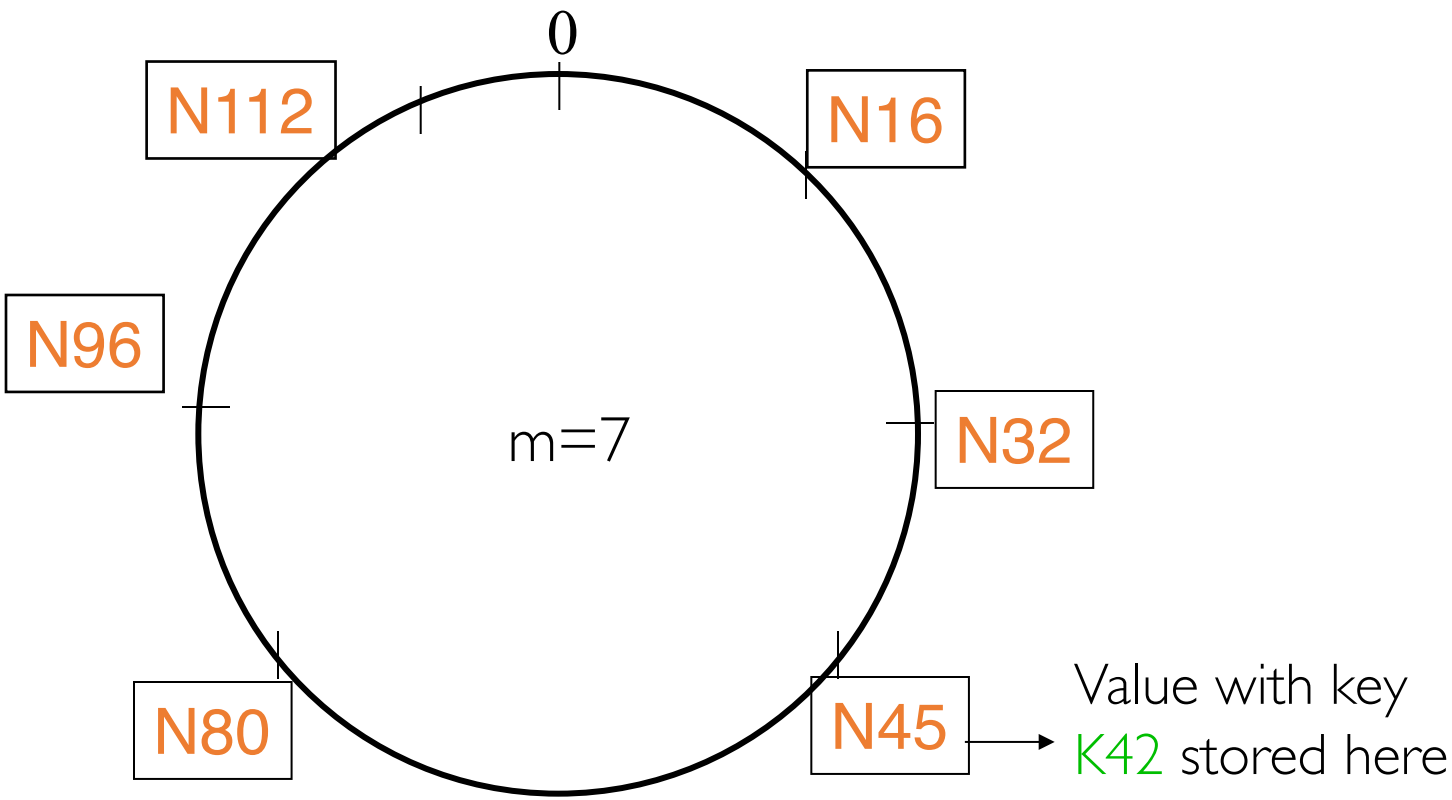
- Consistent Hashing => with K keys and N peers, each peer stores O(K/N) keys. (i.e., < c.K/N, for some constant c)

Ring of Peers: Running Example



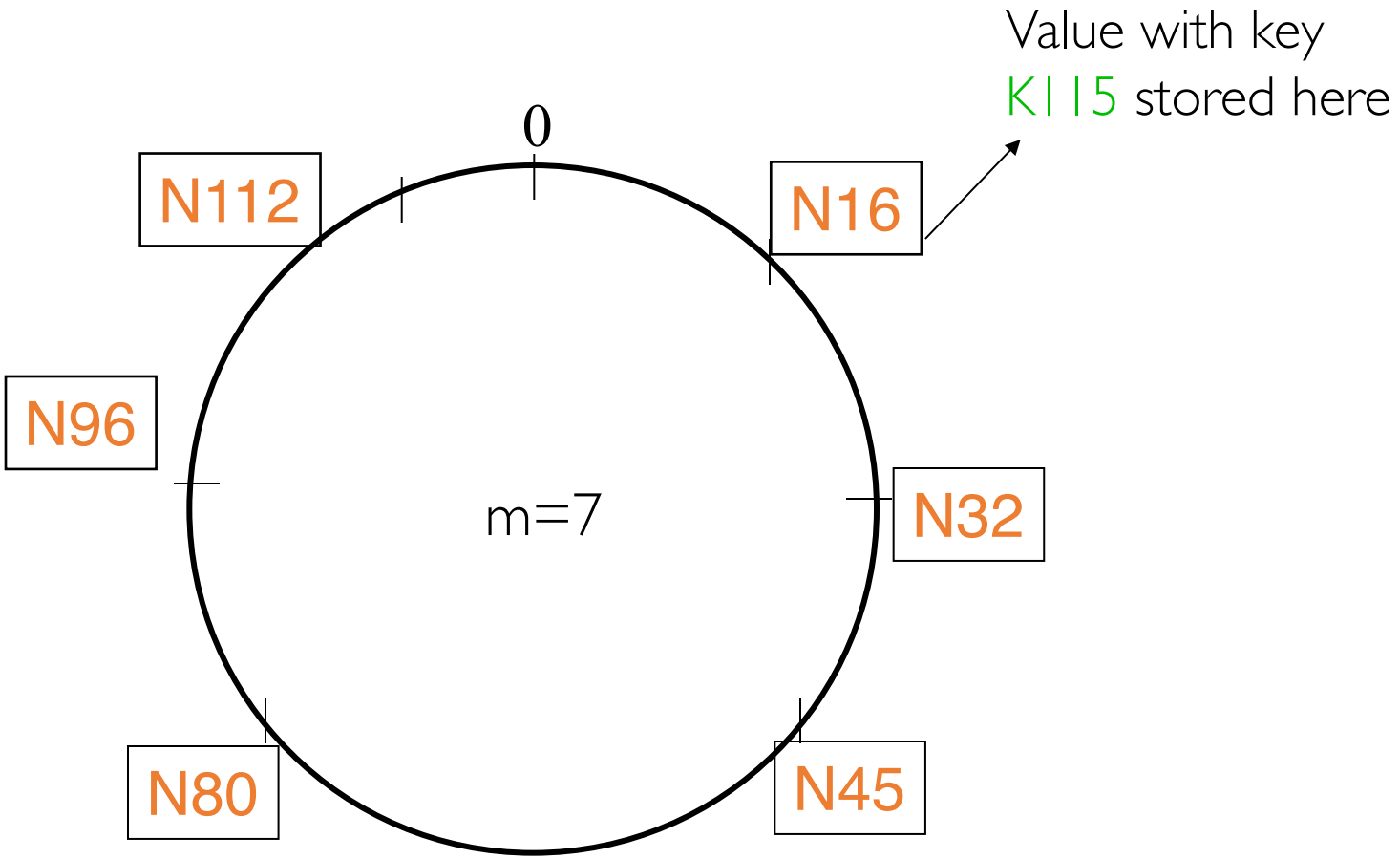
Where will the value with key 42 be stored?

Ring of Peers: Running Example



Where will the value with key 42 be stored?

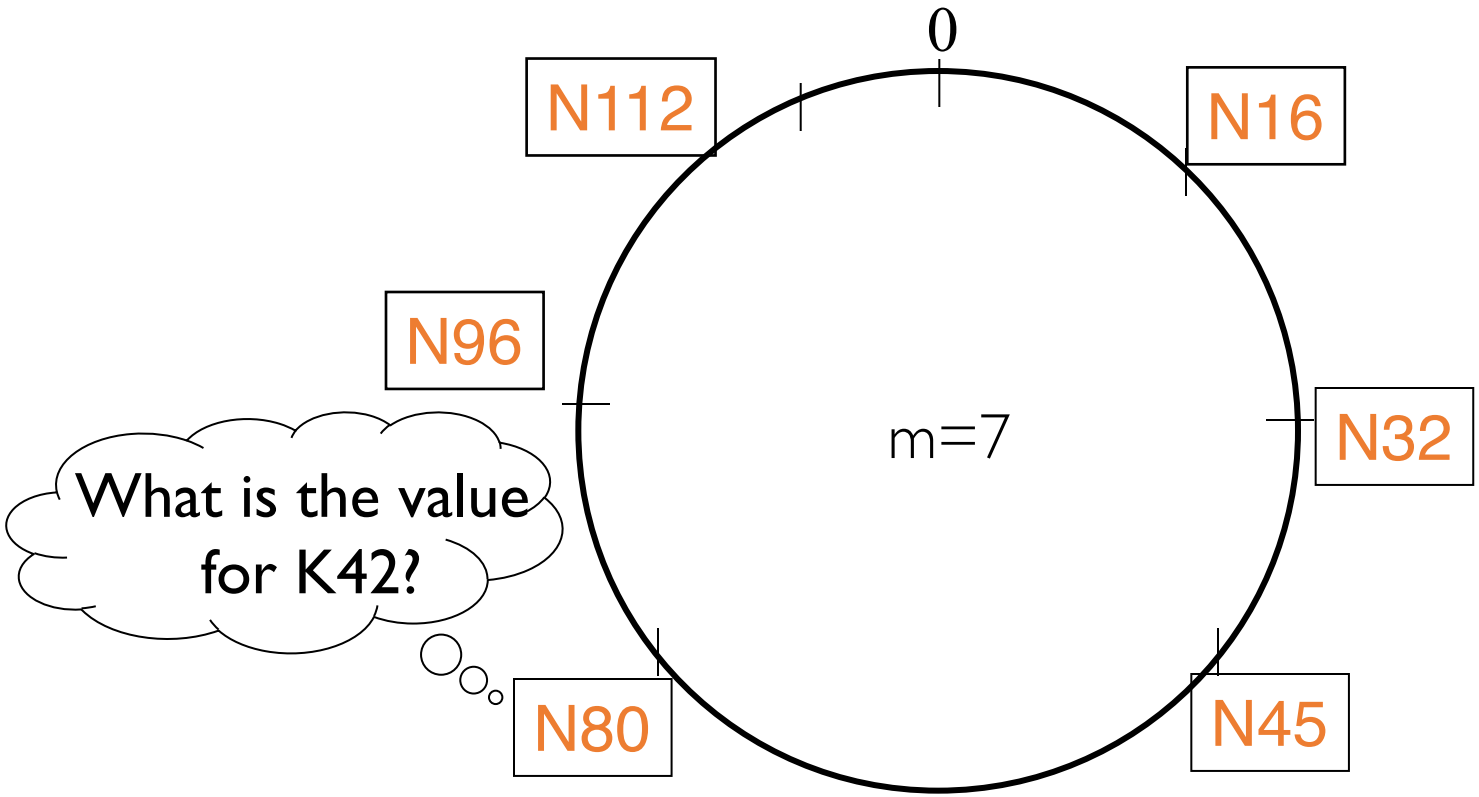
Ring of Peers: Running Example



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 sweet middle-ground.

Performing Lookups

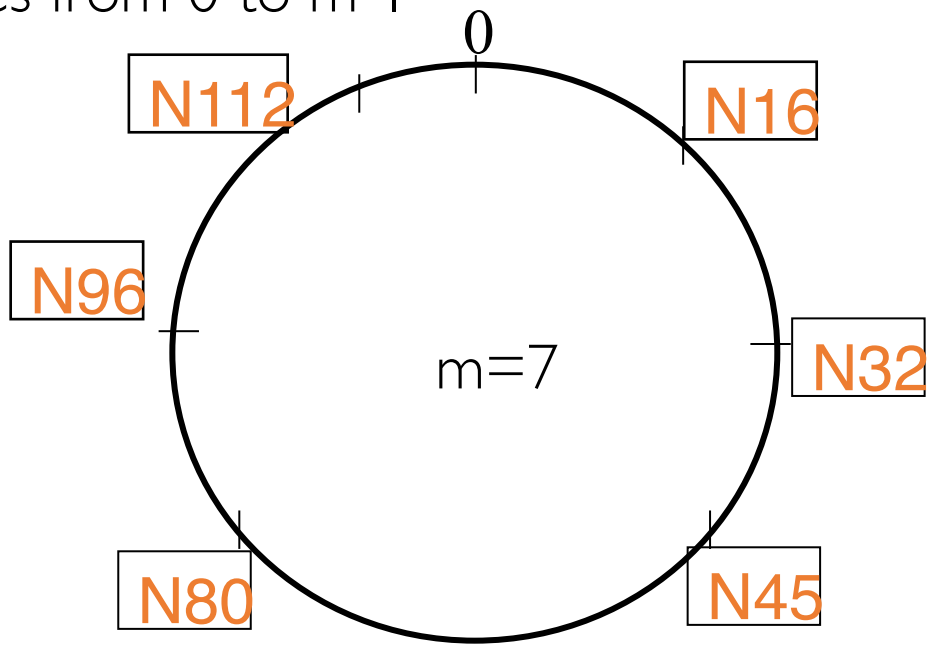
- Chord chooses a sweet middle-ground.
  - Each node is aware of ~**m** other nodes.
  - Maintains a *finger table* with **m** entries.
  - The ith entry of node n's finger table = successor($n + 2^i$)
    - i ranges from 0 to m-1

Finger Tables

Compute the finger table for N80

ith entry of node n's finger table = successor(n + 2^i),

i ranges from 0 to m-1

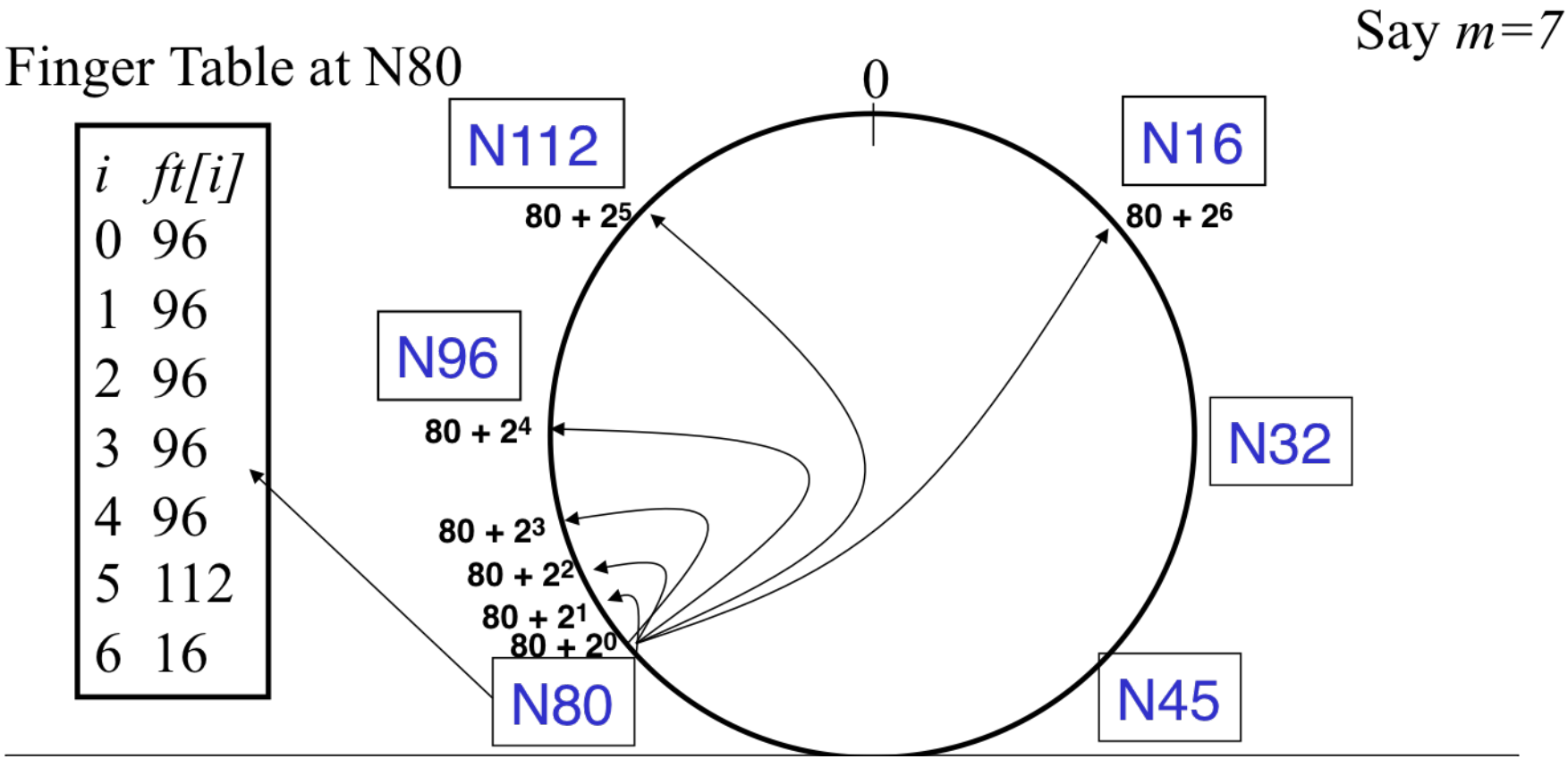


0 : 96
1 : 96
2 : 96
3 : 96
4 : 96
5 : 112
6 : 16
~~7 :~~

80 + 64 = 144

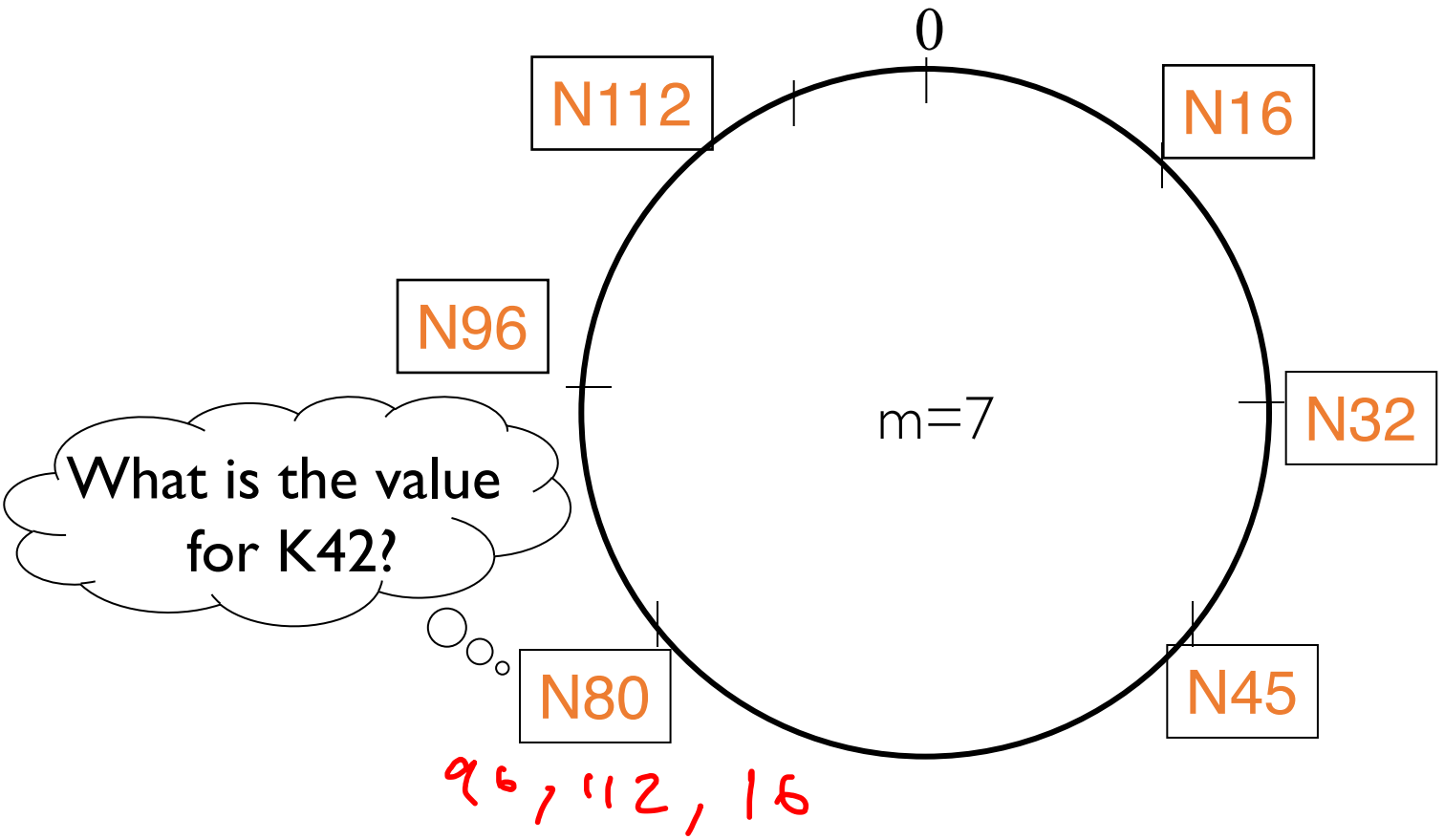
144 - 128 = 16

Finger Tables



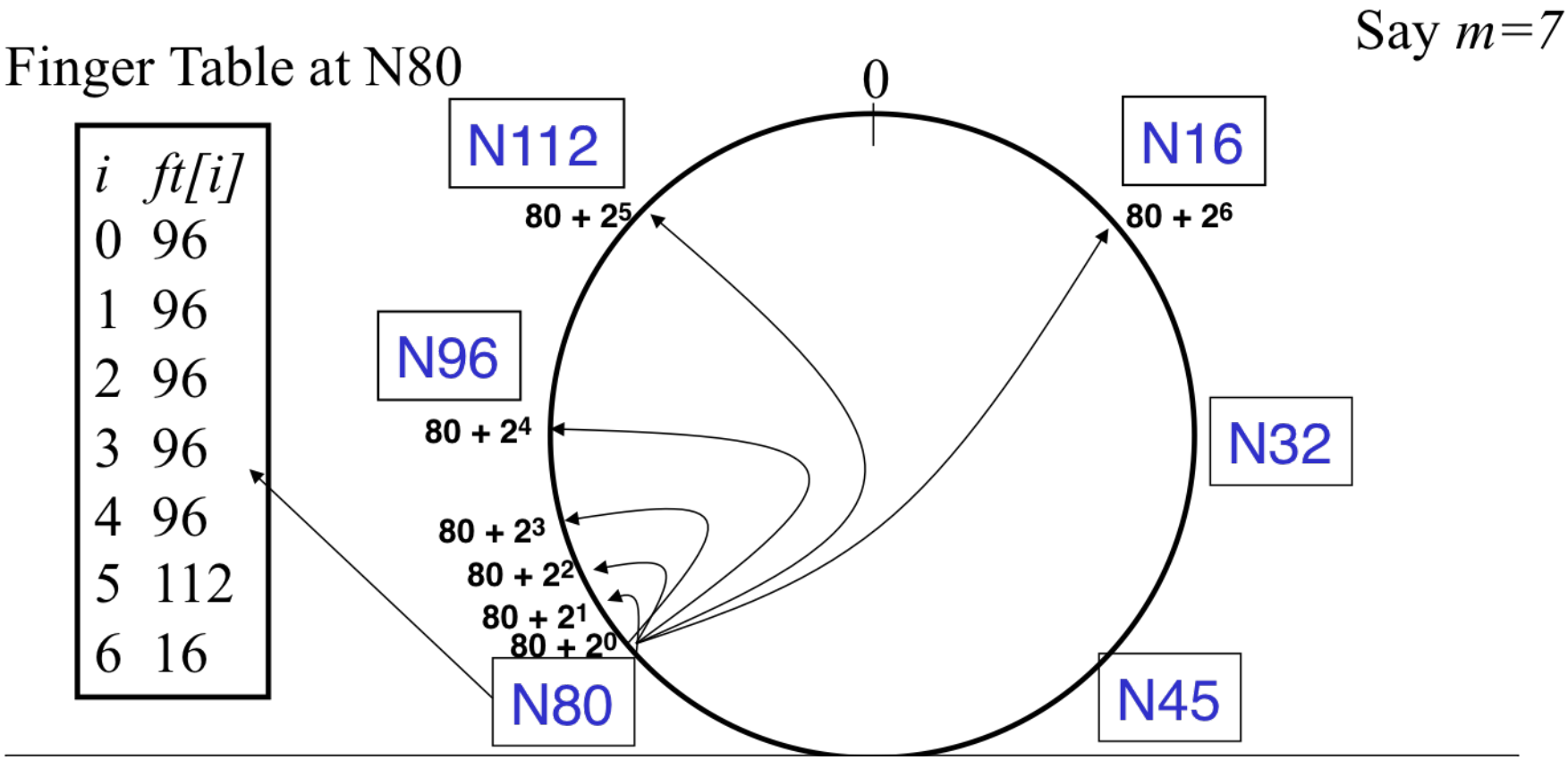
Performing Lookups

Suppose N80 receives a request to lookup K42.



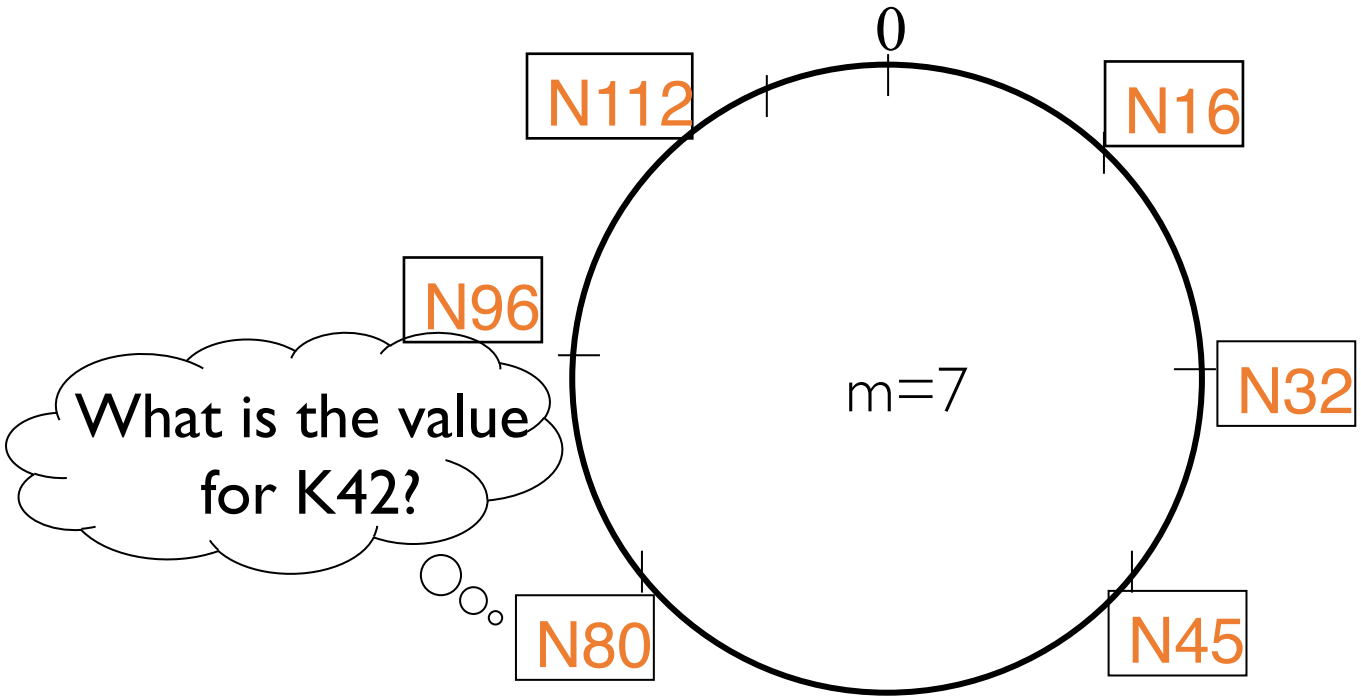
Need to locate successor of K42!

Which nodes is N80 aware of?



Performing Lookups

Suppose N80 receives a request to lookup K42.

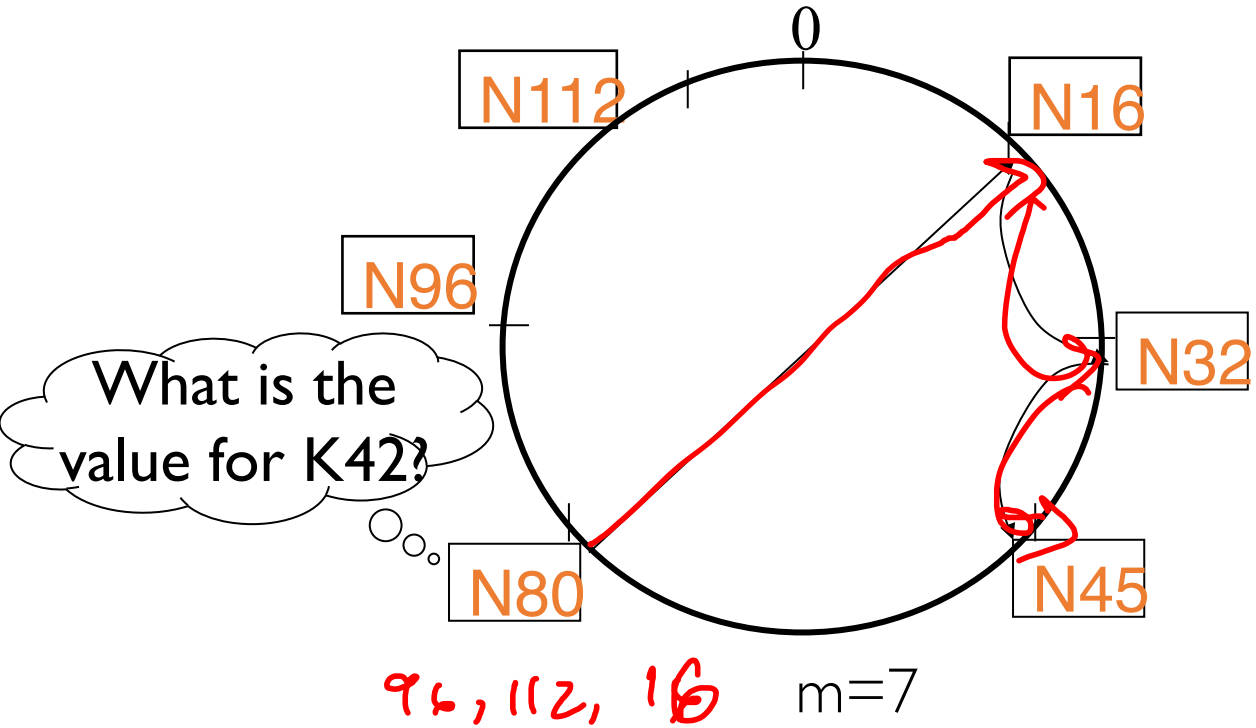


Need to locate successor of K42!

Forward the query to the most promising node you know of.

Search for key k at node n

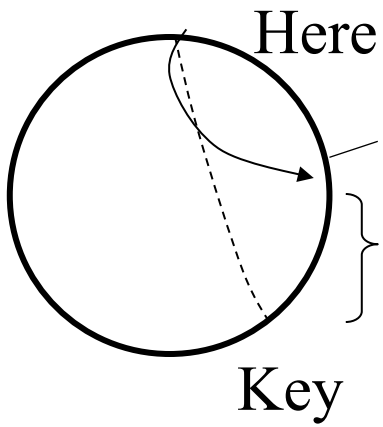
At node n, if k lies in range (n, next(n)], where next(n) is n's *ring successor* then next(n) = successor(key). Send query to next(n)
Else, send query for k to largest finger entry <= k

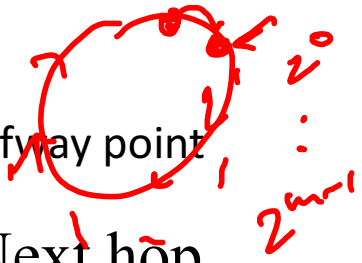


96, 112, 16

0 32
1 32
2 32
3 32
4 32
5 80
6 80

Analysis

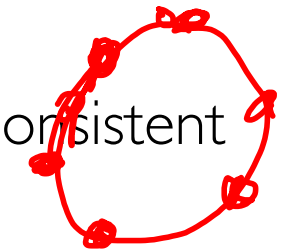




Search takes O(log(N)) time

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 (by properties of consistent hashing)

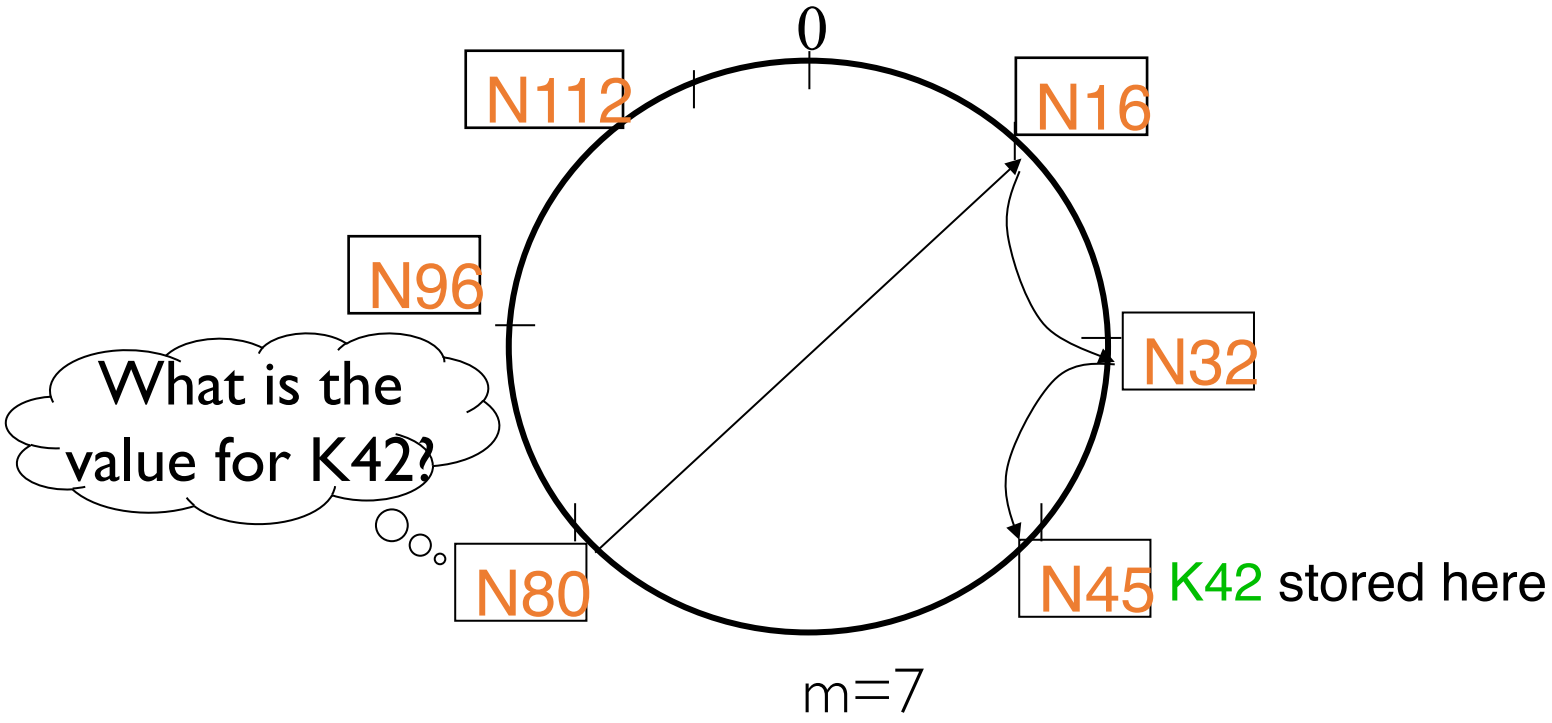


- So using ring successors in that range will use another O(log(N)) hops. Overall lookup time stays O(log(N)).

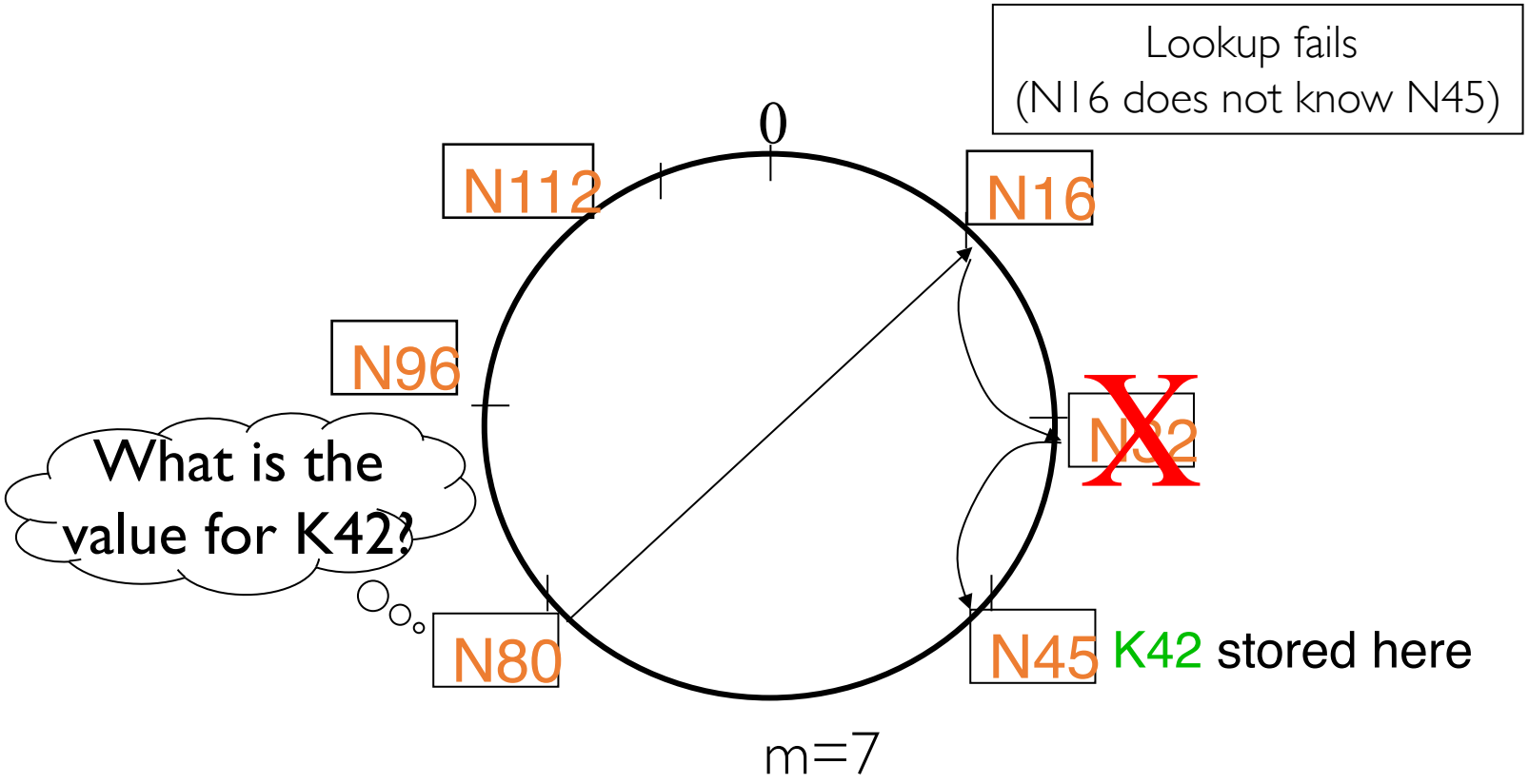
Analysis

- 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: insert, lookup, delete
- O(log(N)) time true only if finger and successor entries correct
- When might these entries be wrong?
  - When you have failures
    - Coming up next!

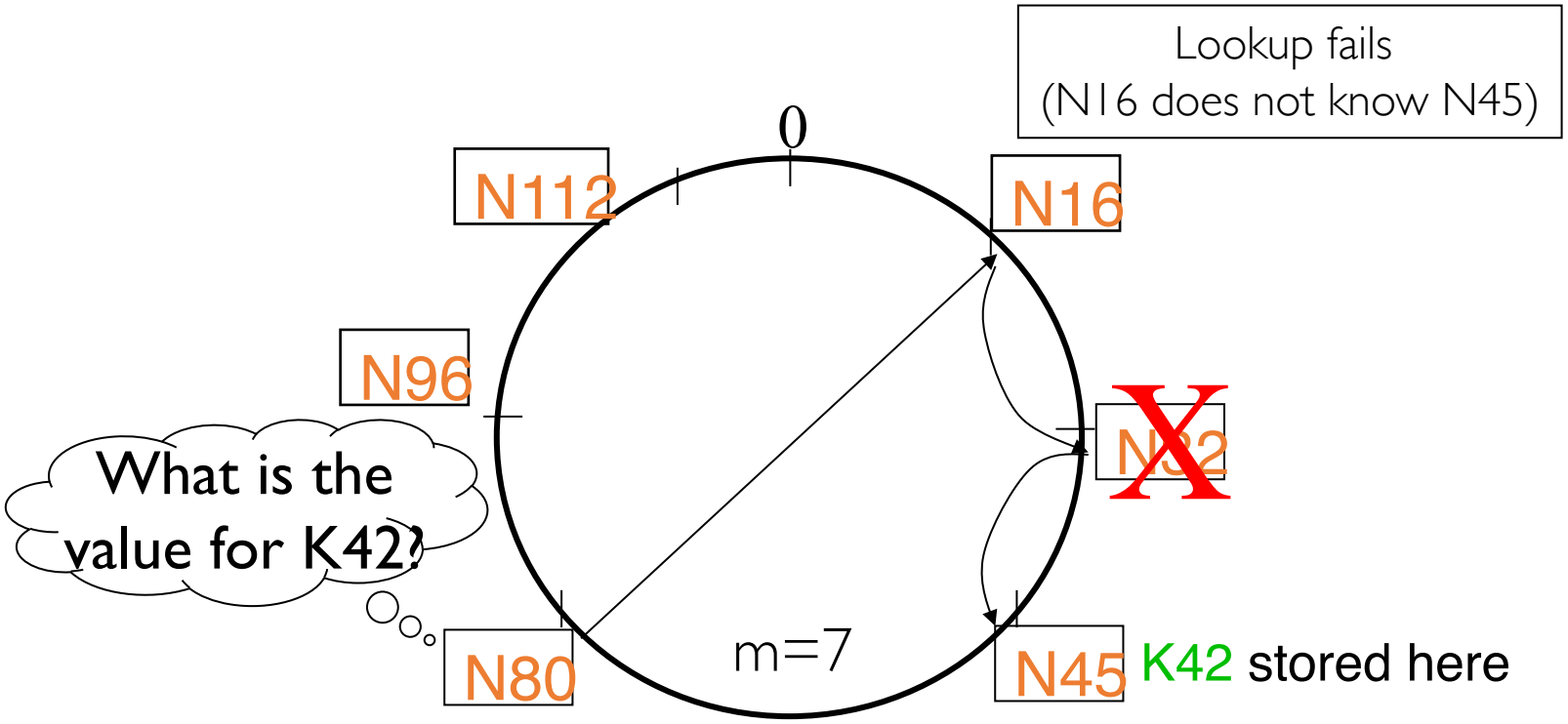
Search for key k at node n



If a node fails



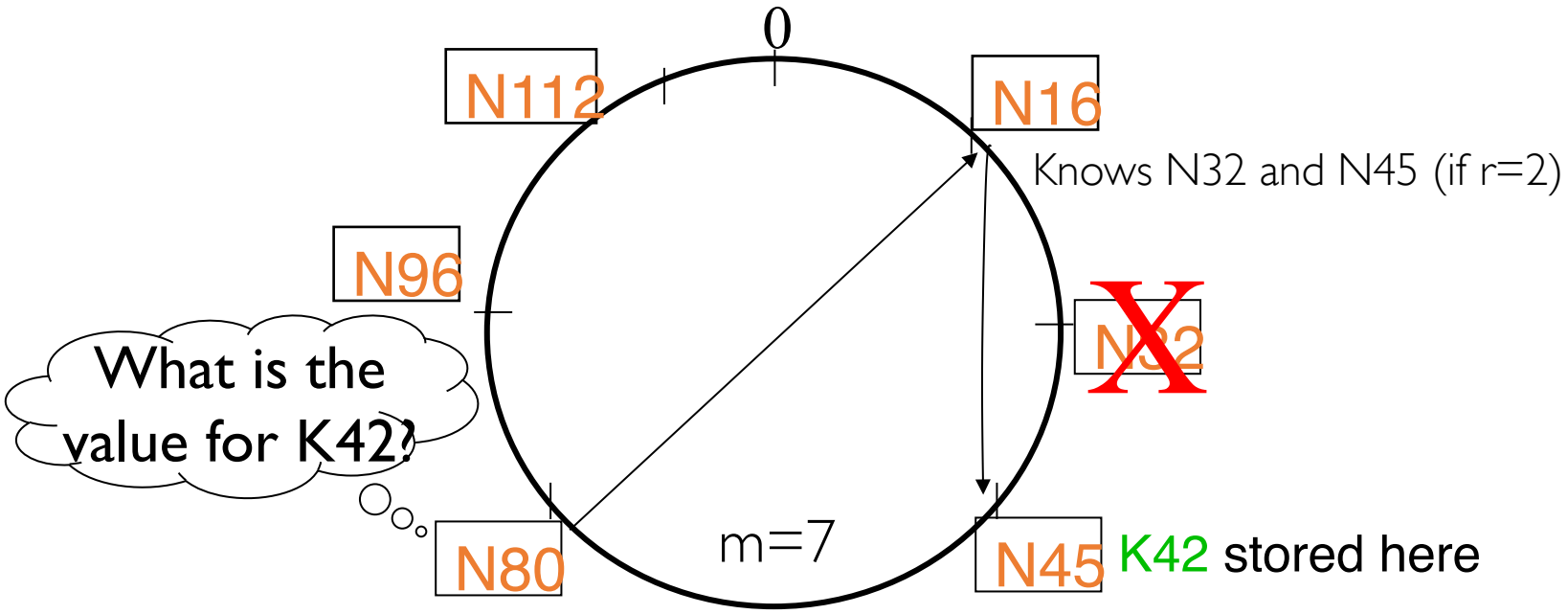
If a node fails



How do we handle this?

If a node fails

One solution: maintain r multiple ring successor entries
In case of failure, use another successor entries



Search under node failures

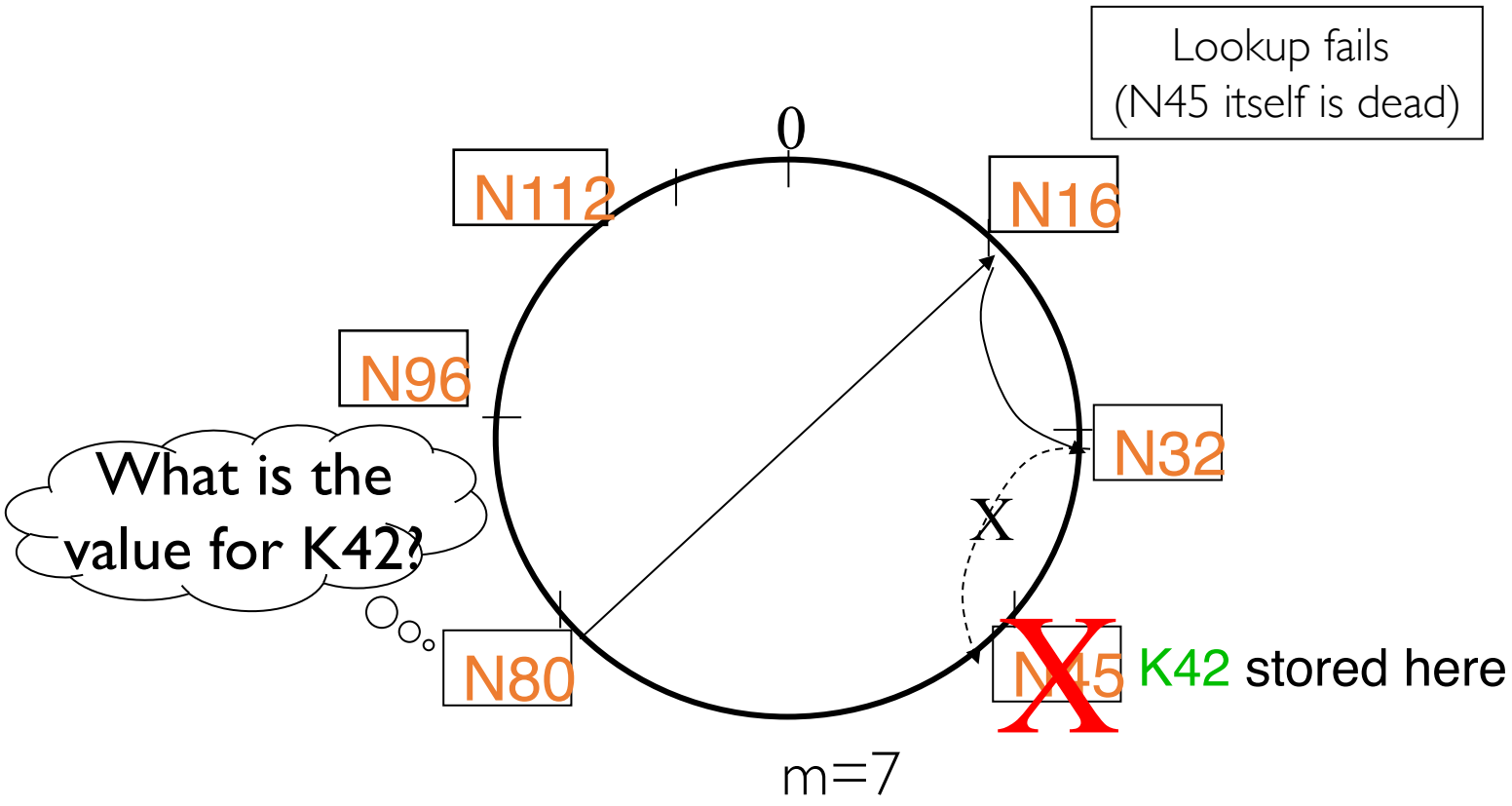
- If every node fails with probability 0.5, choosing r=2log(N) suffices to maintain lookup correctness (i.e. keep the ring connected) with high probability.
  - Intuition:
    - Pr(at given node, at least one ~~successor~~ predecessor alive)=

$$1-(\frac{1}{2})^{2\log N}=1-\frac{1}{N^2}$$

    - Pr(above is true at all alive nodes)=

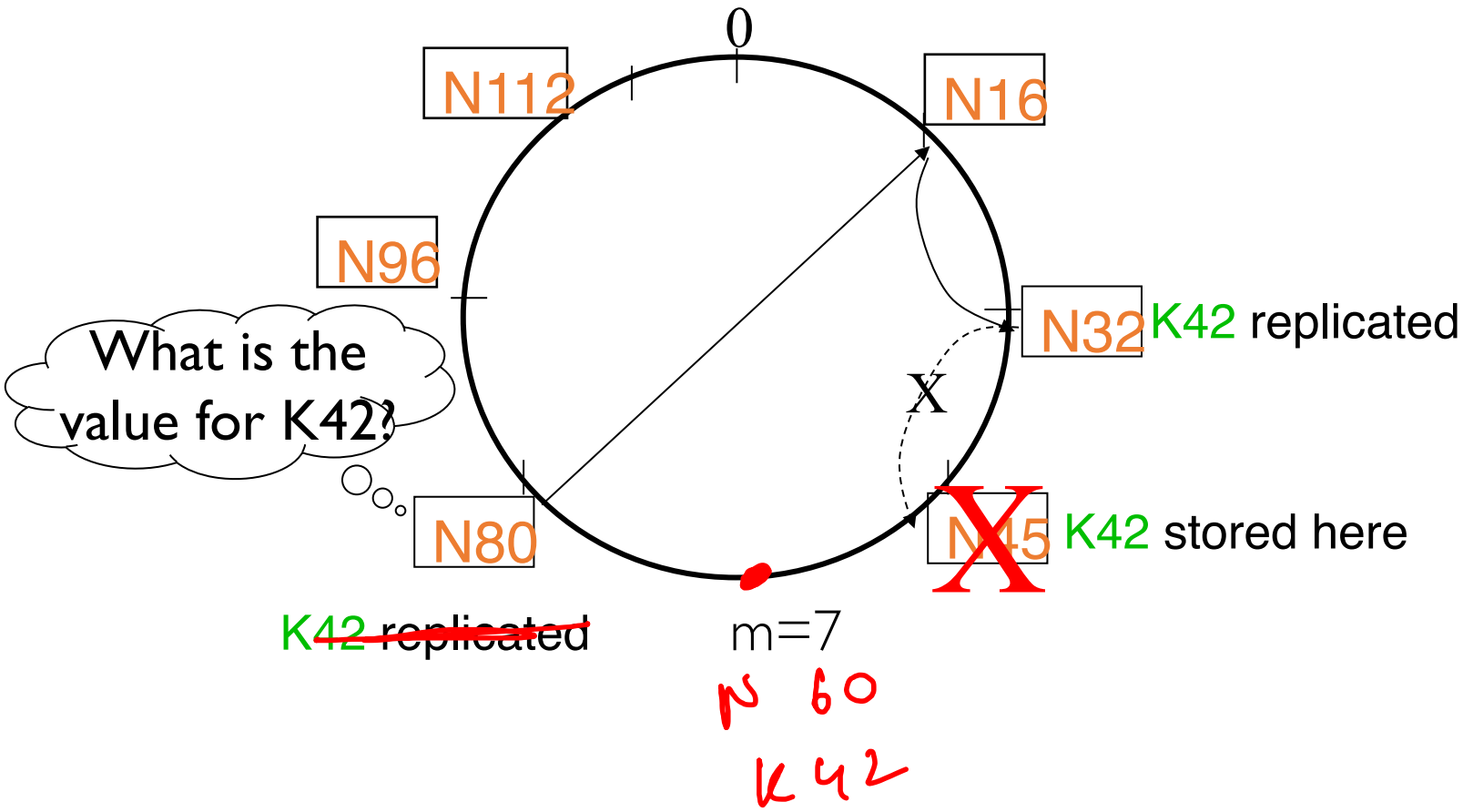
$$(1-\frac{1}{N^2})^{N/2}=e^{-\frac{1}{2N}}\approx 1$$

If a node fails



If a node fails

One solution: replicate key-value at r successors and predecessors



Need to deal with dynamic changes

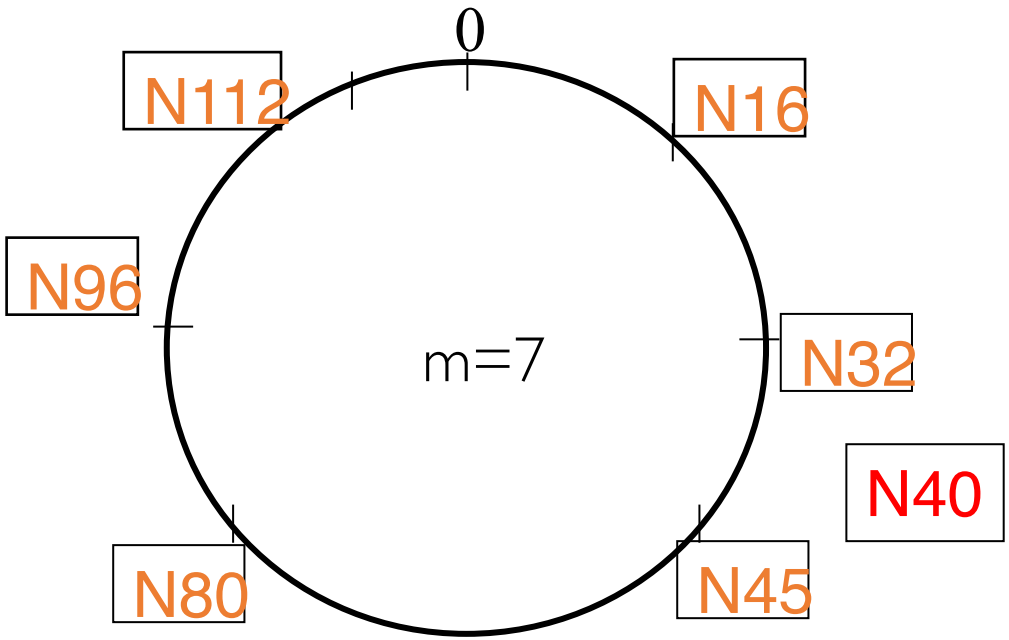
- ✓ Nodes fail
- • New nodes join
- • Nodes leave

So, all the time, need to:

→ Need to update successors and fingers, and copy keys

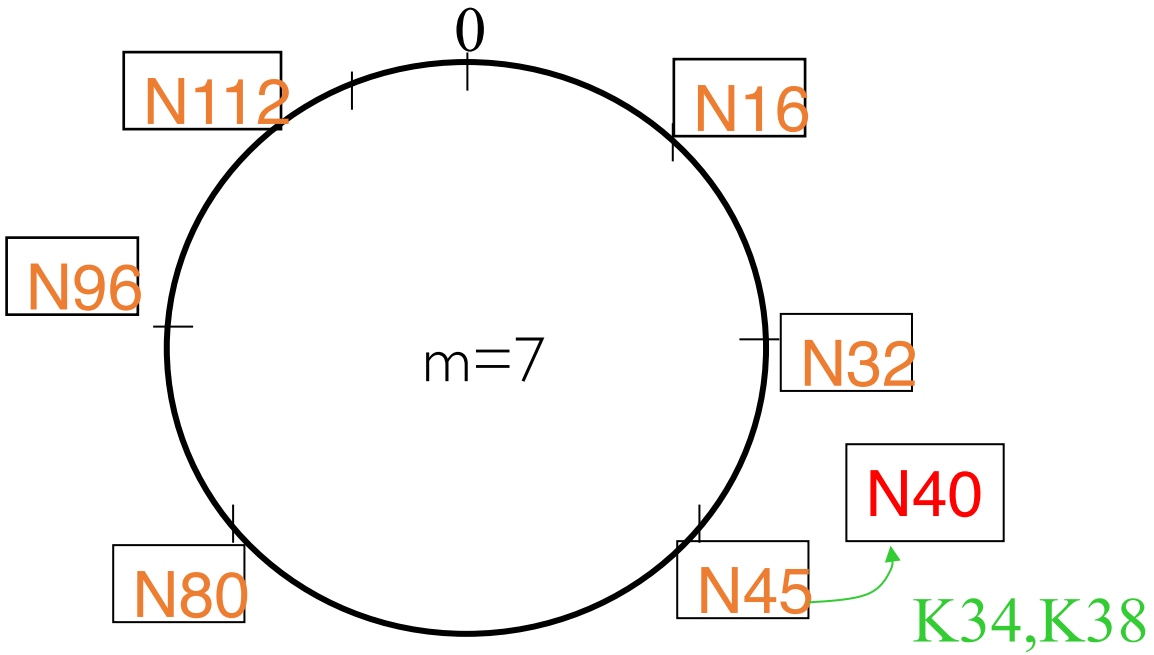
New node joins

New node contacts an existing Chord node (introducer).
Introducer directs N40 to N45 (and N32).
N32 updates its ring successor to N40.
N40 initializes its ring successor to N45, and initializes its finger table.
Other nodes also update their finger table.



New node joins

N40 may need to copy some files/keys from N45
(files with fileid between 32 and 40)



New node joins

- A new peer affects O(log(N)) other finger entries in the system, on average.
- Number of messages per node join (to initialize the new node's finger table) = O(log(N)*log(N))
- Proof in Chord's extended TechReport.

Concurrent Joins

- Aggressively maintaining and updating finger tables each time a node join can be difficult under high *churn*.
  - E.g. when new nodes are concurrently added.
- Correctness of lookup does not require all nodes to have fully "correct" finger table entries.
- Need two invariants:
  - Each node, n, correctly maintains its ring successor (*next(n)*)
    - First entry in the finger table.
  - The node, successor(k), is responsible for key k.
- Next class: stabilization protocol.