Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements for some of the materials: Indy Gupta

Logistics

- Midterm 2 still ongoing.
  - Please refrain from any discussions until we officially release the assessment.
- HW4 due on Friday.
- HW5 (last homework!) will be released on Friday.
- MP3 due on May 1st.

Logistics

- Final exams via CBTF.
- Date range: May 7 – May 15
- Duration: 1 hr 50mins
- Please reserve a slot on PrairieTest if you haven't already done so.
- Syllabus: comprehensive
  - everything covered in class, including your midterm 1 and midterm 2 syllabus, and what's covered / will be covered after it.

Last few classes: Transactions Processing

- Transaction processing and concurrency control:
  - Multiple clients simultaneously access same set of objects.
  - How to maintain ACID properties.
  - Pessimistic and optimistic concurrency control.

- Distributed transactions:
  - Objects are sharded across multiple servers.
  - Two phase commit and distributed deadlock detection.

- **Next: when objects are *replicated* across multiple servers.**

Distributed Transactions

- *Sharding:* objects can be distributed across multiple (1000's of) servers
  - what we have been discussing so far.
  - Primary reason: load balancing and scalability.

- *Replication:* the same object may be replicated among a handful of nodes.
  - Primary reason: fault-tolerance, availability, durability.

Replication: Natural way to handle failures

- Node failures are common.

In each cluster's first year, it's typical that 1,000 individual machine failures will occur; thousands of hard drive failures will occur; one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours; 20 racks will fail, each time causing 40 to 80 machines to vanish from the network; 5 racks will "go wonky," with half their network packets missing in action; and the cluster will have to be rewired once, affecting 5 percent of the machines at any given moment over a 2-day span. And there's about a 50 percent chance that the cluster will overheat, taking down most of the servers in less than 5 minutes and taking 1 to 2 days to recover.

-- Jeff Dean (Google), source: cnet.com

Replication: Natural way to handle failures

- Node failures are common.
- What could happen if a node fails?
  - Objects unavailable until recovery.
  - 2PC "stuck" after coordinator failure
- Even worse: what happens if the drive fails.
  - no recovery!
- Replication provides greater availability and robustness to failures.
  - Geo-replication (spanning datacenters across the world) for greater robustness.

Replication

- Replication = An object has identical copies, each maintained by a separate server.
  - Copies are called "replicas"

- With *k* replicas of each object, can tolerate failure of any (*k*-*1*) servers in the system

Replication: Availability

- If each server is down a fraction f of the time
  - Server's failure probability

- With no replication, availability of object
  = Probability that single copy is up
  = $(1 - f)$

- With k replicas, availability of object
  = Probability that at least one replicas is up
  = 1 – Probability that all replicas are down
  = $(1 - f^k)$

Replication: Availability

- With no replication, availability of object =
  = Probability that single copy is up
  = $(1 - f)$
- With k replicas, availability of object =
  Probability that at least one replicas is up
  = 1 – Probability that all replicas are down
  = $(1 - f^k)$

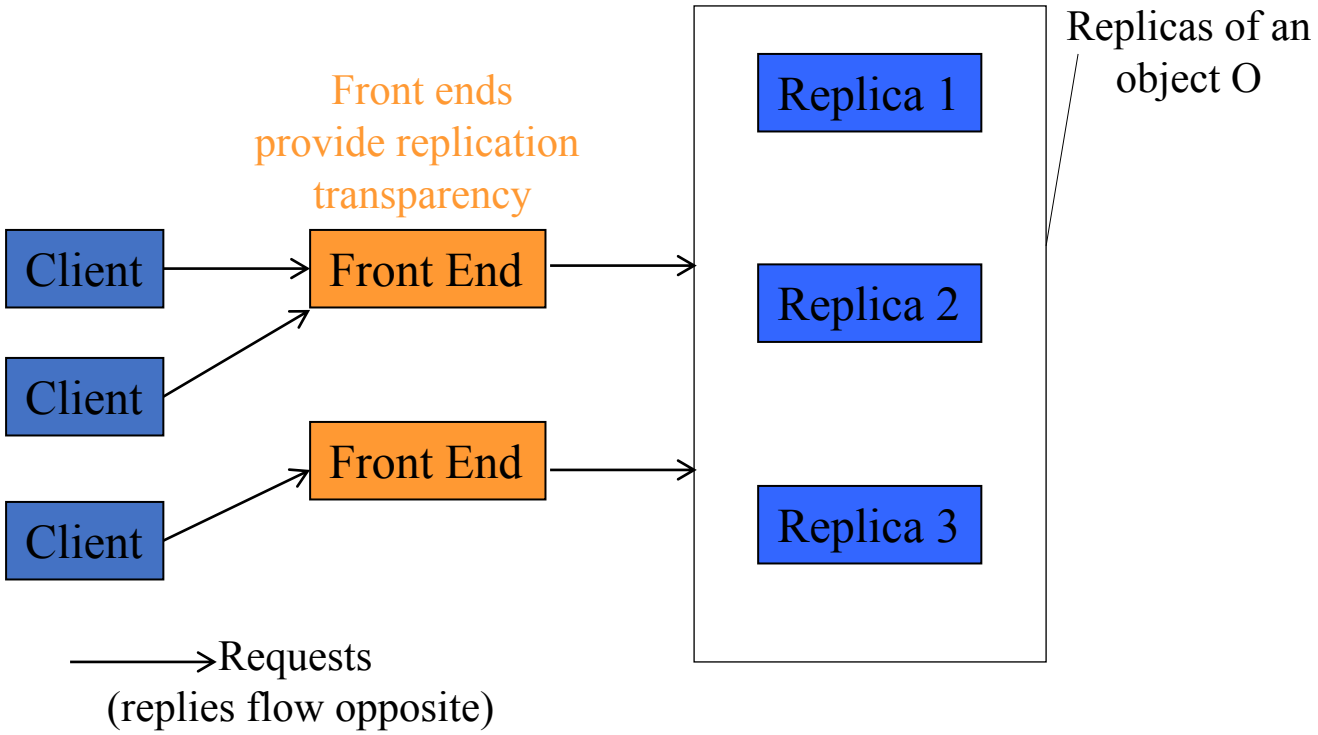
f=failure probability	No replication	*k*=3 replicas	*k*=5 replicas
0.1	90%	99.9%	99.999%
0.05	95%	99.9875%	6 Nines
0.01	99%	99.9999%	10 Nines

Replication: Challenges

1. Replication Transparency
   - A client ought not to be aware of multiple copies of objects existing on the server side

2. Replication Consistency
   - All clients see single consistent copy of data, in spite of replication
   - For transactions, guarantee ACID

Replication Transparency

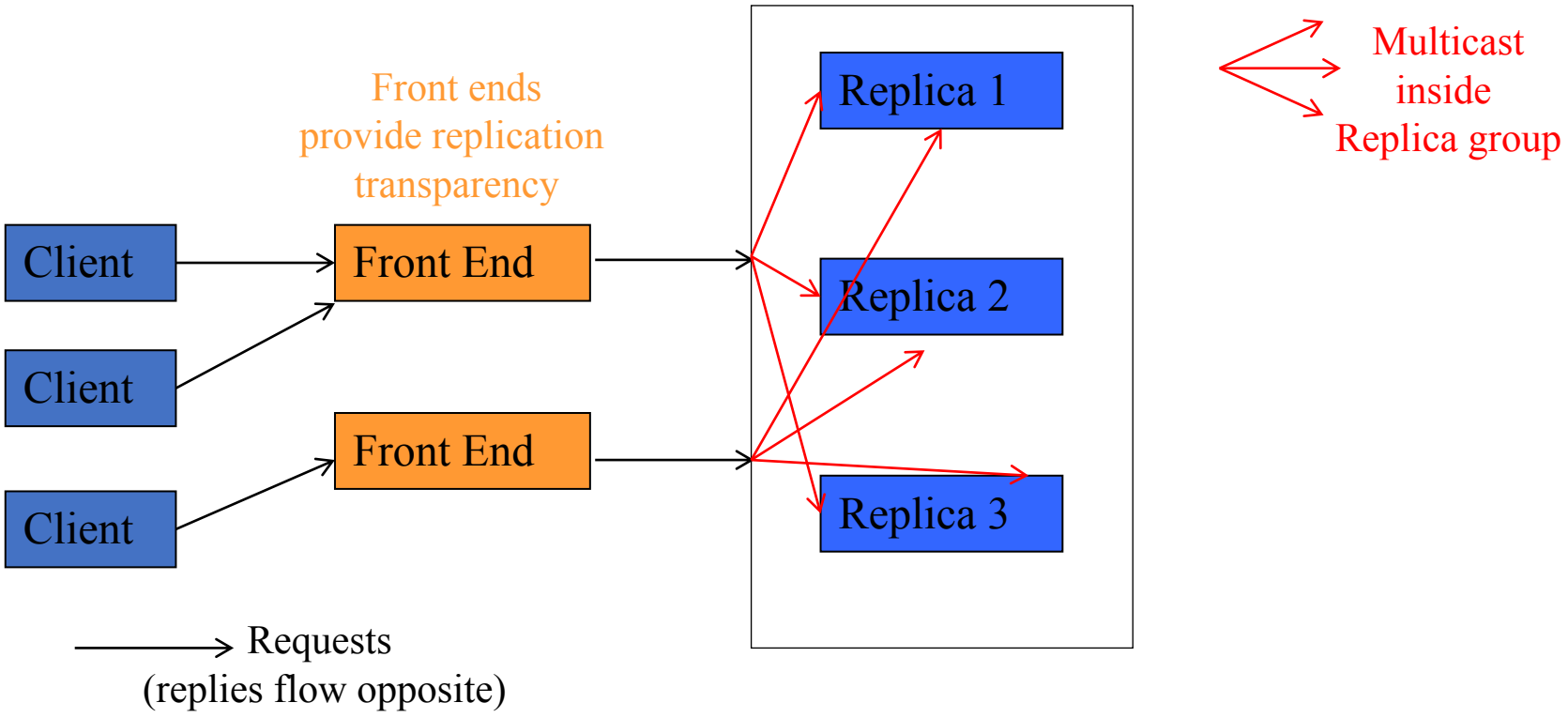


Replication Consistency

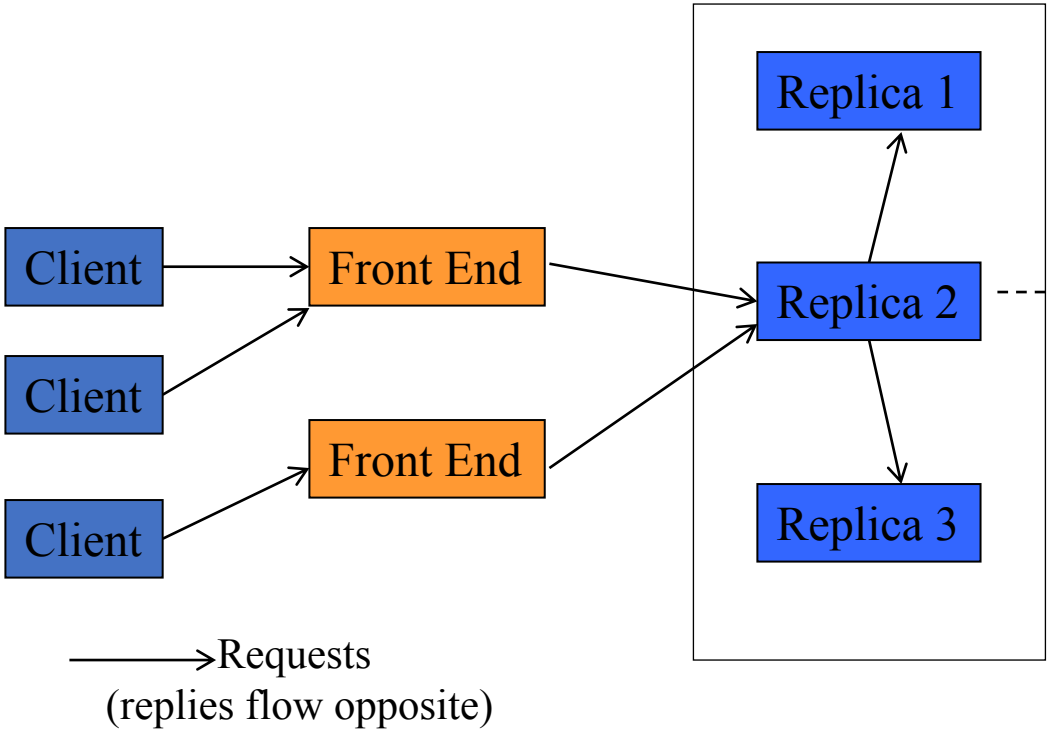
- Two ways to forward updates from front-ends (FEs) to replica group
  - Active Replication: treats all replicas identically
  - Passive Replication: uses a primary replica (leader)

- Both approaches use the concept of "Replicated State Machines"
  - Each replica's code runs the same state machine
  - *Multiple copies of the same State Machine begun in the Start state, and receiving the same Inputs in the same order will arrive at the same State having generated the same Outputs.* [Schneider 1990]

Active Replication



Passive Replication



- Leader => total reliable ordering of all updates
- On leader failure, run election

Transactions and Replication

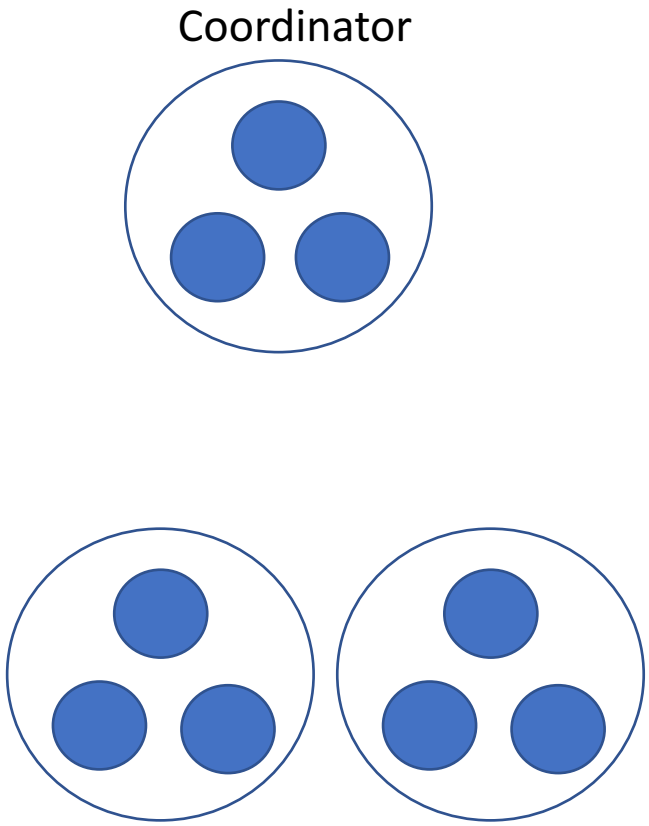
- One-copy serializability
  - *A concurrent execution of transactions in a replicated database is one-copy-serializable if it is equivalent to a serial execution of these transactions over a single logical copy of the database.*
  - (Or) The effect of transactions performed by clients on replicated objects should be the same as if they had been performed one at a time on a single set of objects (i.e., 1 replica per object).
- In a non-replicated system, transactions appear to be performed one at a time in some order.
  - Correctness means serial equivalence of transactions
- When objects are replicated, transaction systems for correctness need one-copy serializability.

Transactions and Replication

- Objects distributed among 1000's cluster nodes for load-balancing (sharding)
- Objects replicated among a handful of nodes for availability / durability.
  - Replication across data centers, too
- Two-level operation:
  - Use transactions, coordinators, 2PC per object
  - Use Paxos / Raft among object replicas
- Consensus needed across object replicas, e.g.
  - When acquiring locks and executing operations
  - When committing transactions

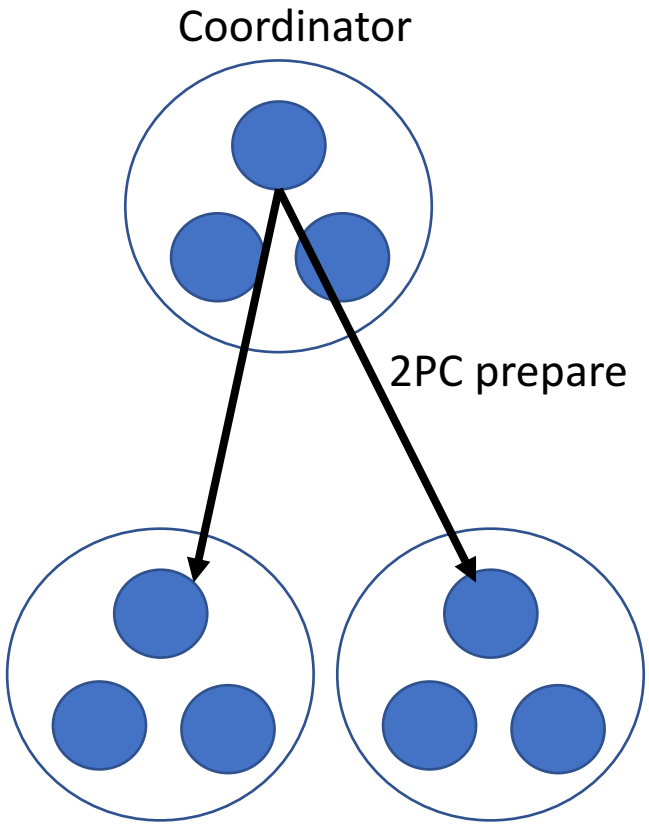
2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group



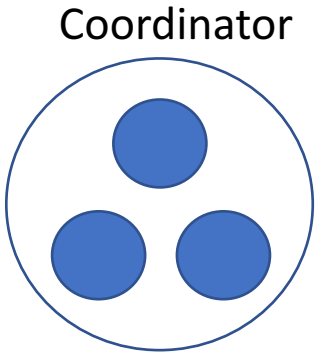
2PC and Paxos

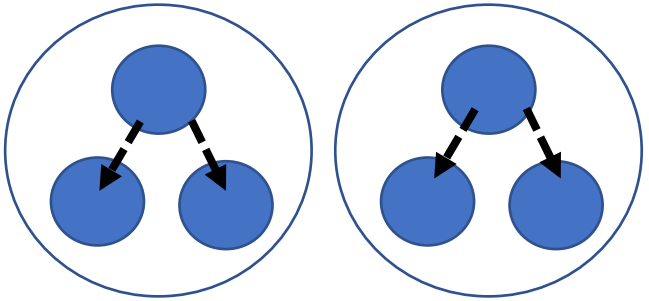
- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs



2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs

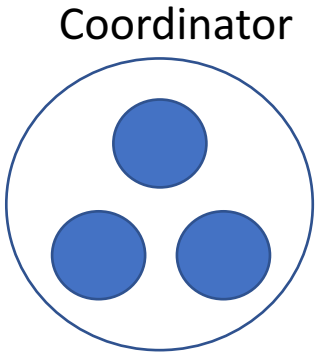


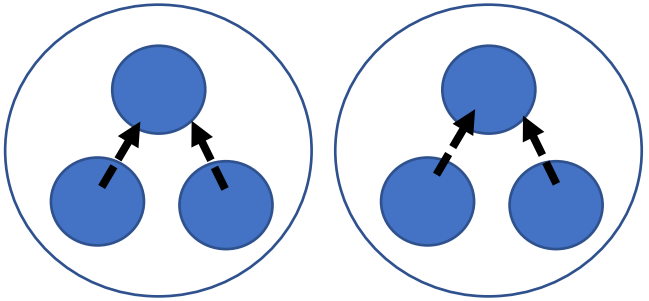


Paxos Prepare

2PC and Paxos

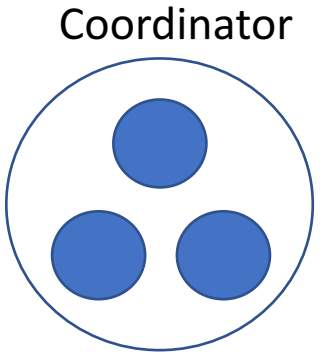
- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs

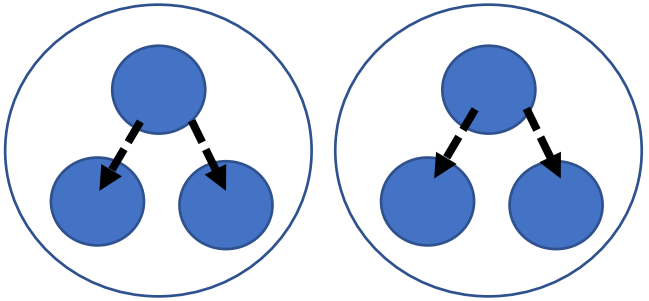




2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs

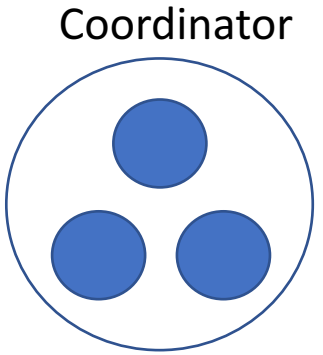


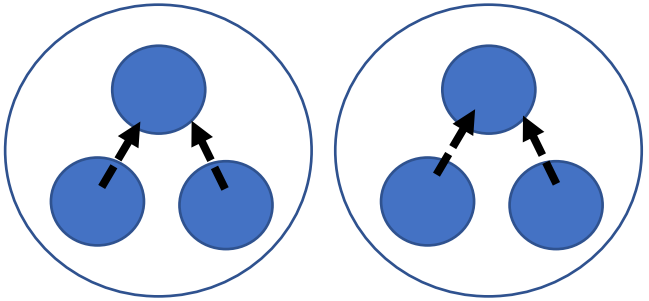


Paxos Accept

2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs

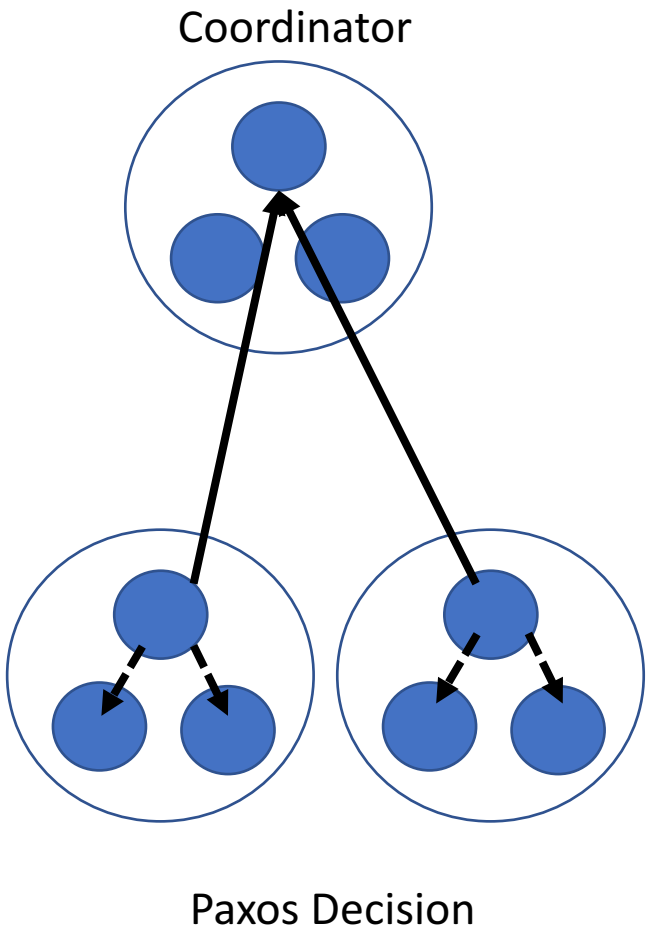




Paxos relay accept to leader
(distinguished learner)

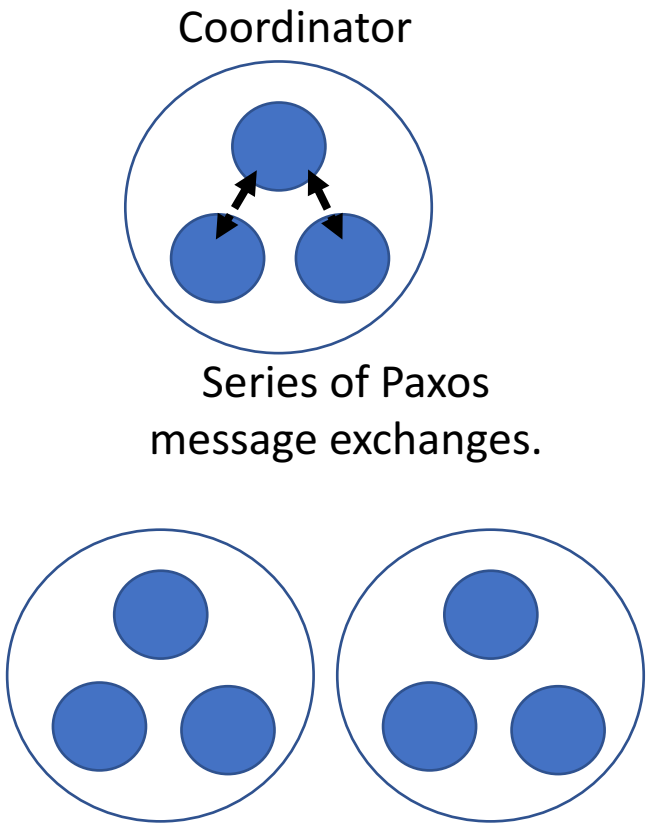
2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader



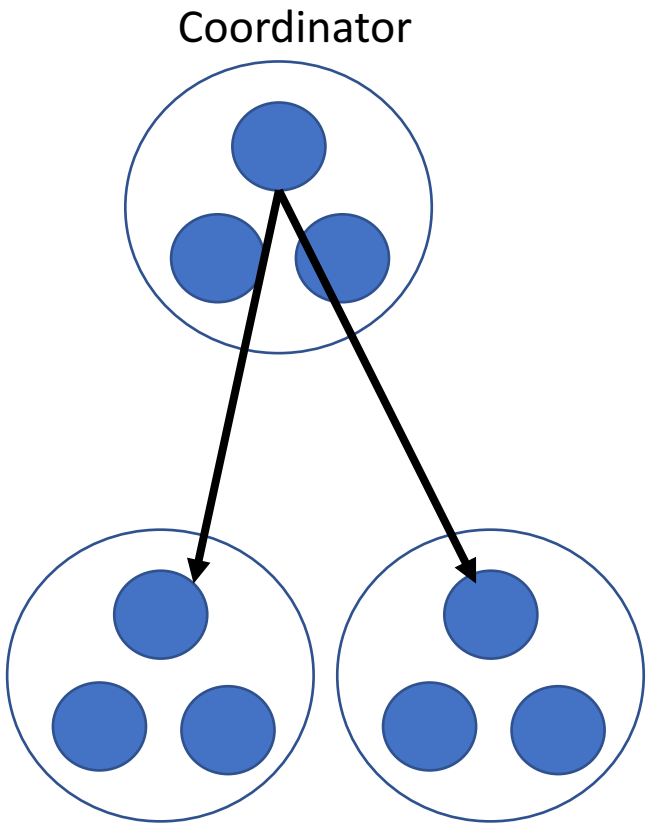
2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader
  - Coordinator leader uses Paxos to commit decision to its group log.



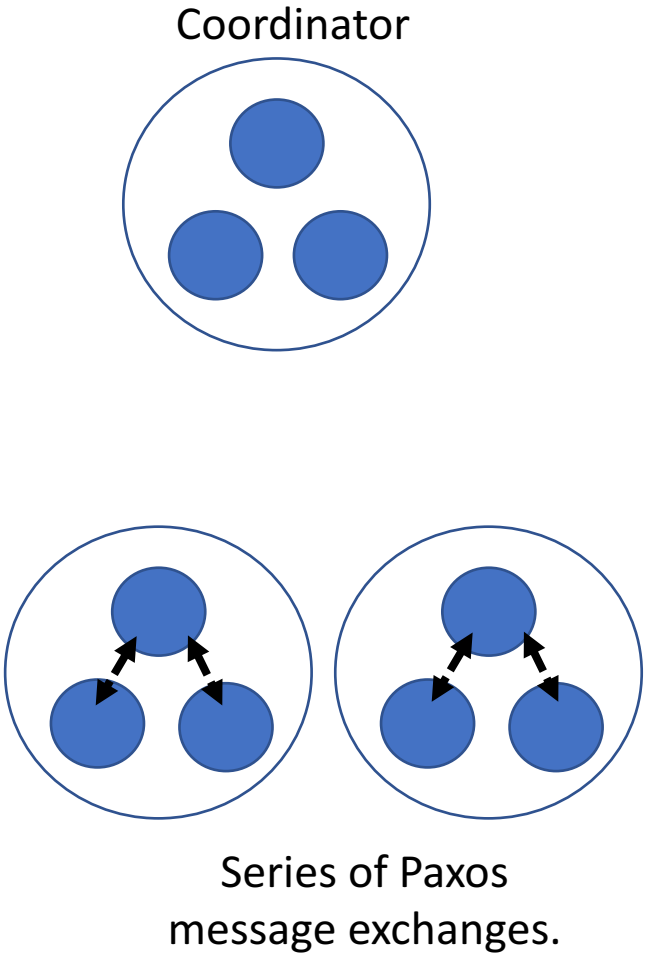
2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader
  - Coordinator leader uses Paxos to commit decision to its group log.
  - Coordinator leader sends Commit message to leaders of each replica group.



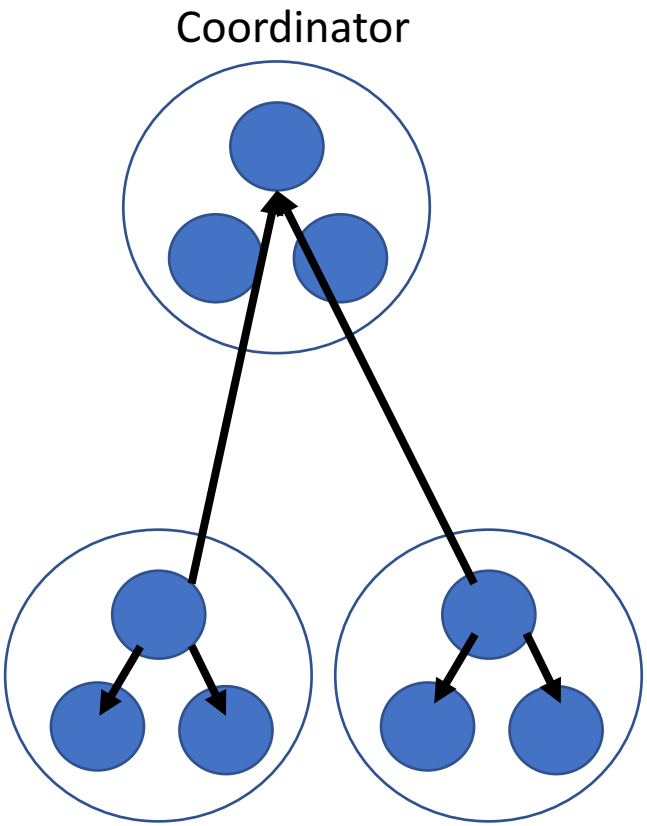
2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader
  - Coordinator leader uses Paxos to commit decision to its group log.
  - Coordinator leader sends Commit message to leaders of each replica group.
  - Each replica leader uses Paxos to process the final commit.



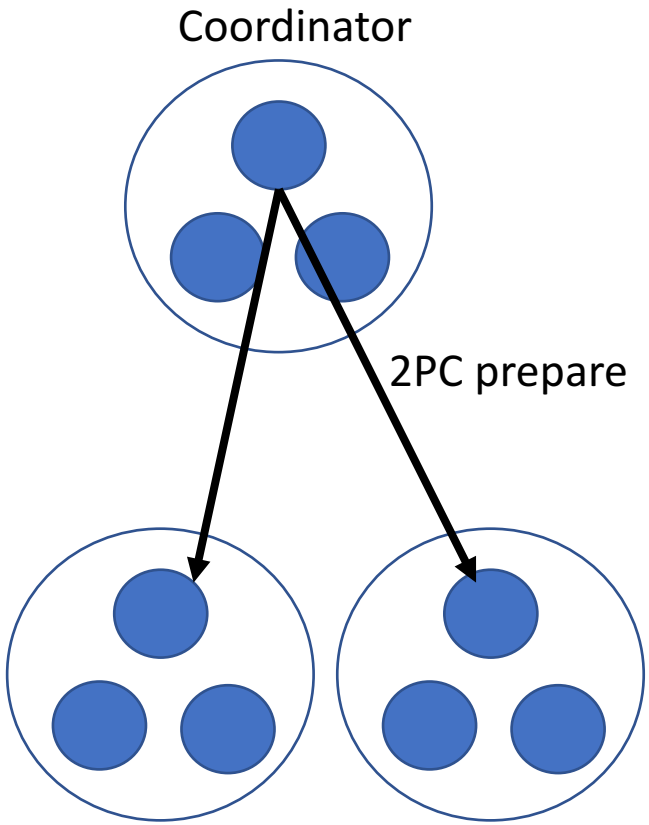
2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader
  - Coordinator leader uses Paxos to commit decision to its group log.
  - Coordinator leader sends Commit message to leaders of each replica group.
  - Each replica leader uses Paxos to process the final commit.
  - Replica leader send the "commit ok / have committed" message back to coordinator.



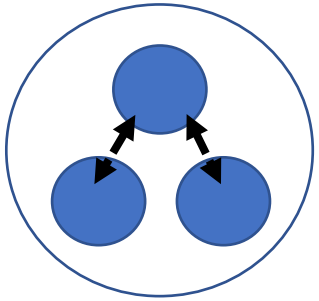
2PC and Paxos

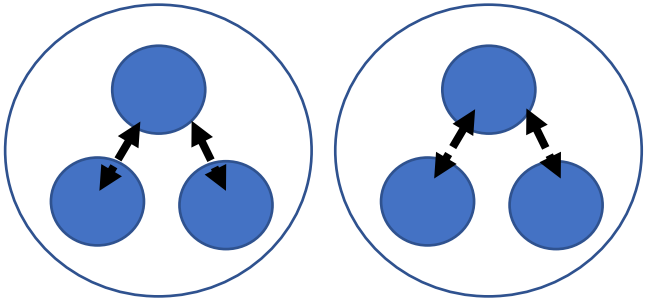
- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs



2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs



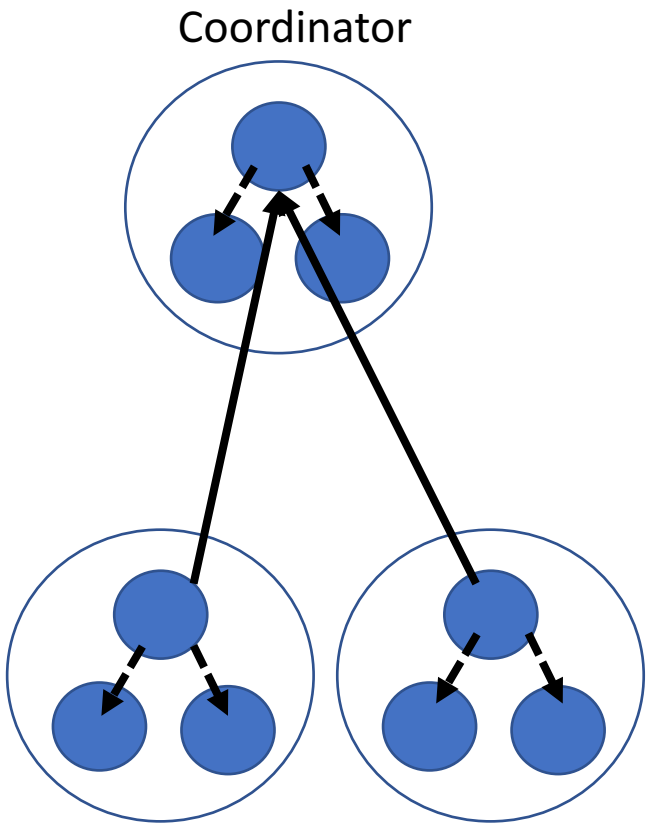


Series of message exchanges for Paxos

Coordinator can also be a participant

2PC and Paxos

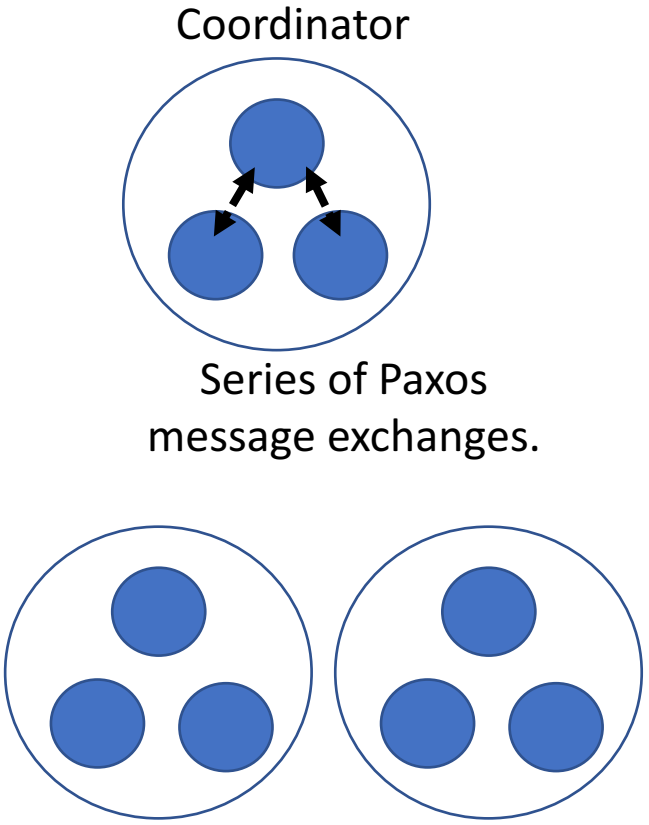
- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader



Coordinator can also be a participant

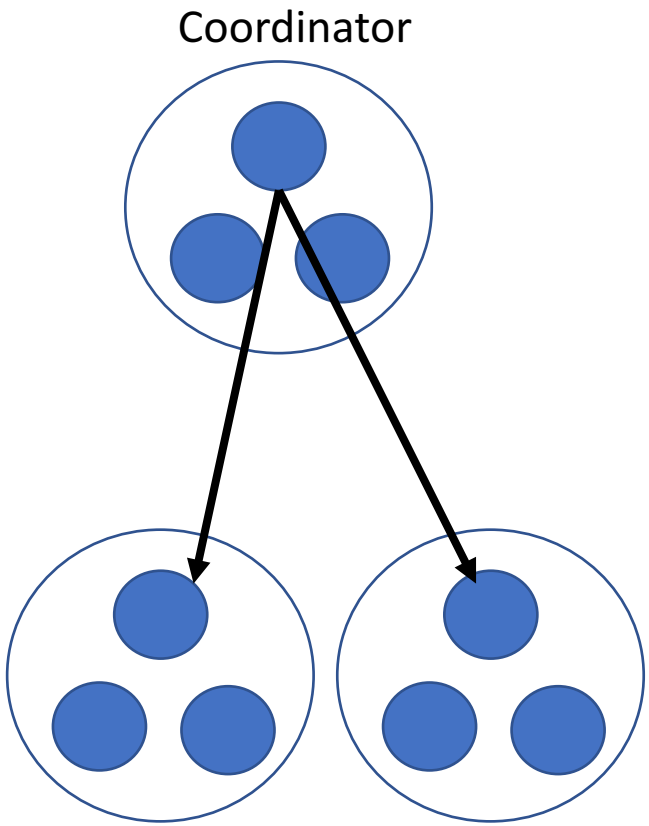
2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader
  - Coordinator leader uses Paxos to commit decision to its group log.



2PC and Paxos

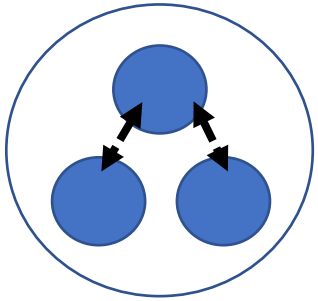
- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader
  - Coordinator leader uses Paxos to commit decision to its group log.
  - Coordinator leader sends Commit message to leaders of each replica group.

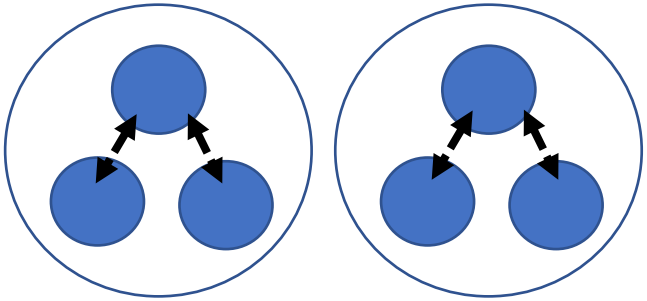


Coordinator can also be a participant

2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader
  - Coordinator leader uses Paxos to commit decision to its group log.
  - Coordinator leader sends Commit message to leaders of each replica group.
  - Each replica leader uses Paxos to process the final commit.



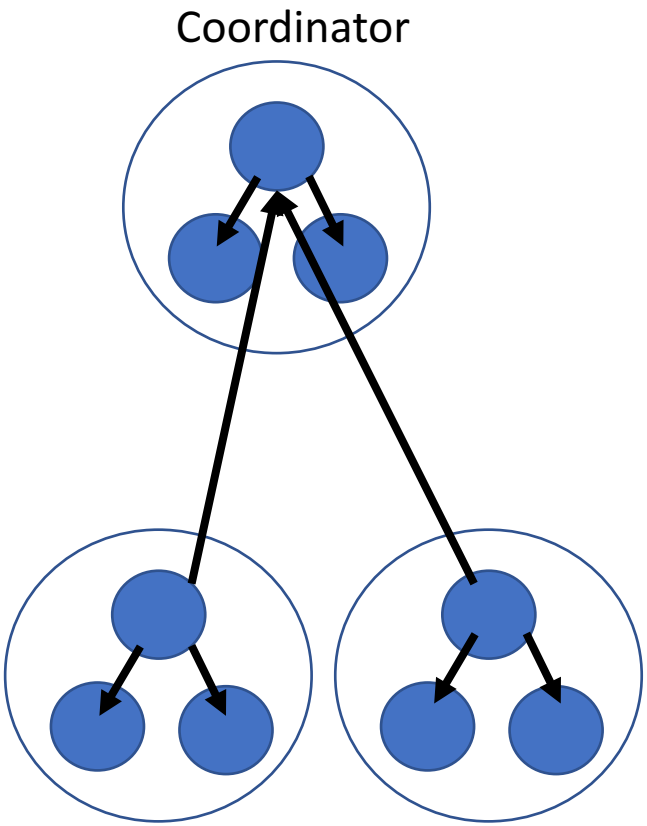


Series of Paxos message exchanges.

Coordinator can also be a participant

2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader
  - Coordinator leader uses Paxos to commit decision to its group log.
  - Coordinator leader sends Commit message to leaders of each replica group.
  - Each replica leader uses Paxos to process the final commit.
  - Replica leader send the "commit ok / have committed" message back to coordinator.



Optional Self-Study (not in syllabus)

- "Spanner: Google's Globally-Distributed Database", OSDI'12

- Makes use of:
  - Two-phase locking
  - Two-phase commit
  - Paxos consensus
  - Time synchronization with tight synchronization bounds for "external consistency"

- Paper and talk are available online.

Our agenda for the next few 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 and cloud job scheduling
  - How to run large-scale distributed computations over key-value stores?
    - Map-Reduce Programming Abstraction
  - How to schedule jobs in the cloud?
  - How to design a large-scale distributed key-value store?
    - Case-study: Facebook's Cassandra

Our agenda for the next few 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 and Cloud job scheduling
  - How to run large-scale distributed computations over key-value stores?
    - Map-Reduce Programming Abstraction
  - How to schedule jobs in the cloud?
  - 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

- 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 Data Stores

- Necessary API operations: get(key) and put(key, value)
- Also called NoSQL Data Stores
  - NoSQL = "Not Only SQL"

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 2000s.
  - 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.

Distributed Hash Tables (DHTs)

- Multiple protocols were proposed in early 2000s.
  - **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.

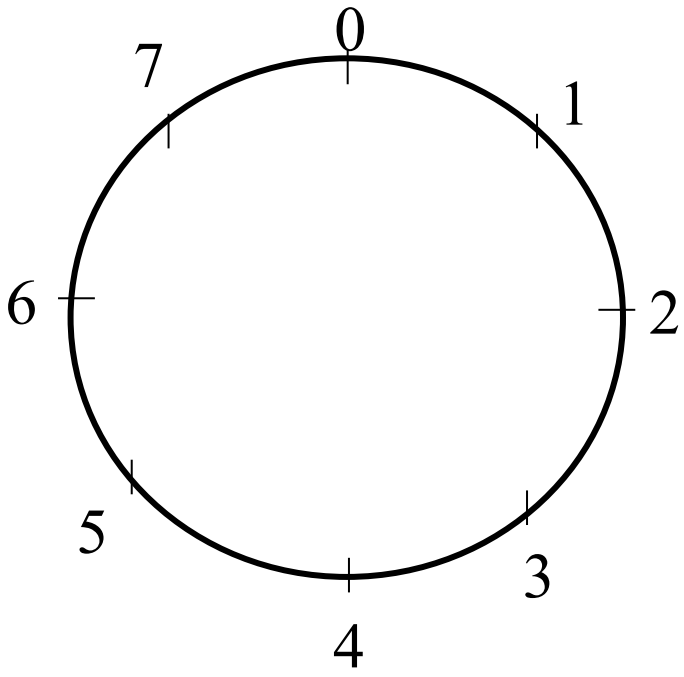
- Chord: Developed at MIT by I. Stoica, D. Karger, F. Kaashoek, H. Balakrishnan, R. Morris.

Chord

- Key properties:
  - Load balance:
    - spreads keys evenly over nodes (peers).
  - 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

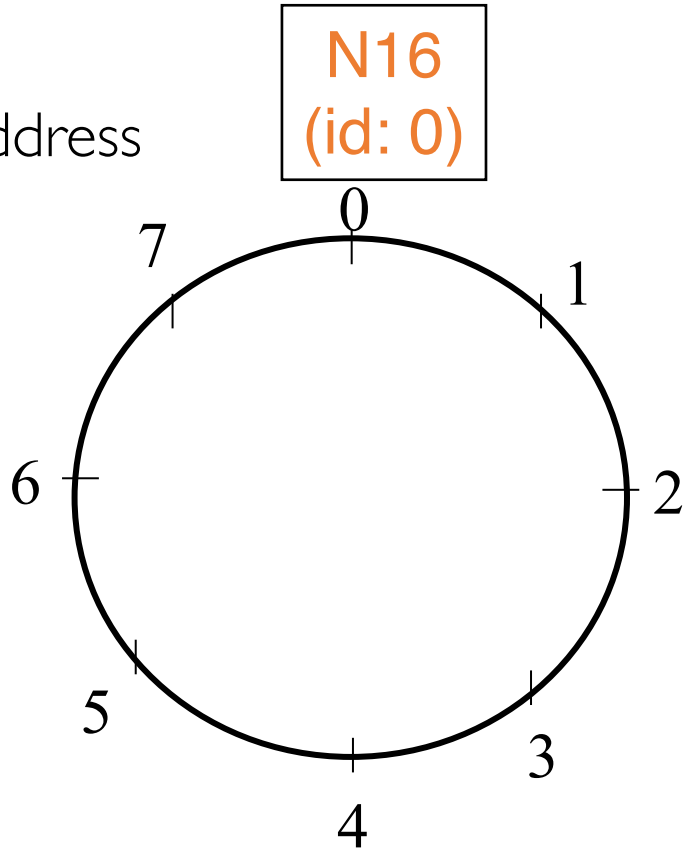
- Uses *Consistent Hashing* on node's (peer's) address
  - SHA-1 (ip_address,port) → 160 bit string
  - Truncated to **m** bits (modulo 2^m)
  - Called peer id (number between 0 and 2^m-1)
  - **m** chosen such that negligible chance of id conflicts
  - Can then map peers to one of 2^m logical points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

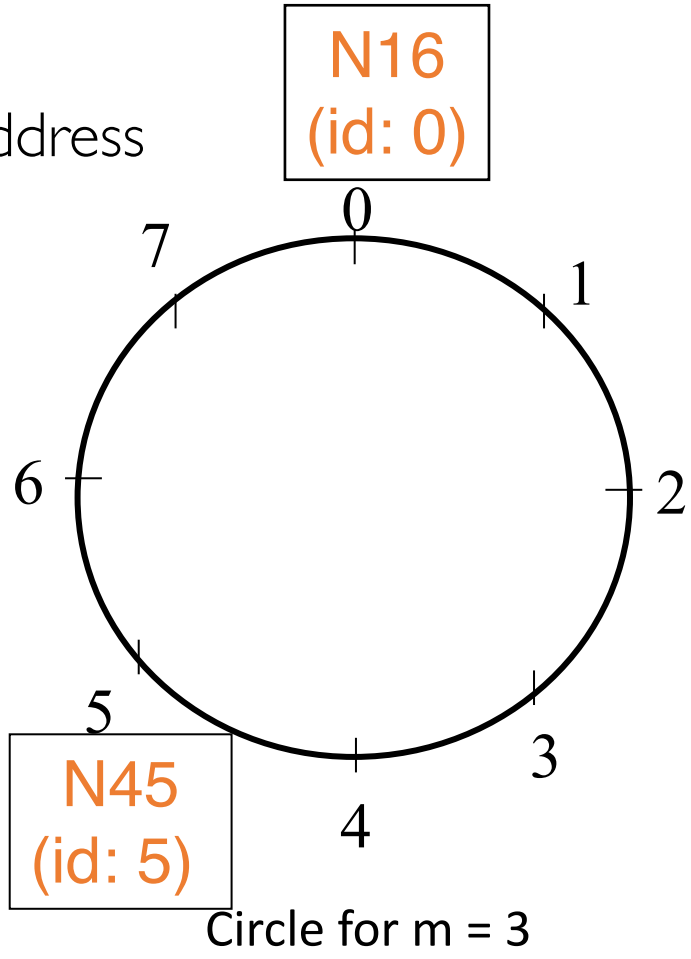


Circle for m = 3

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

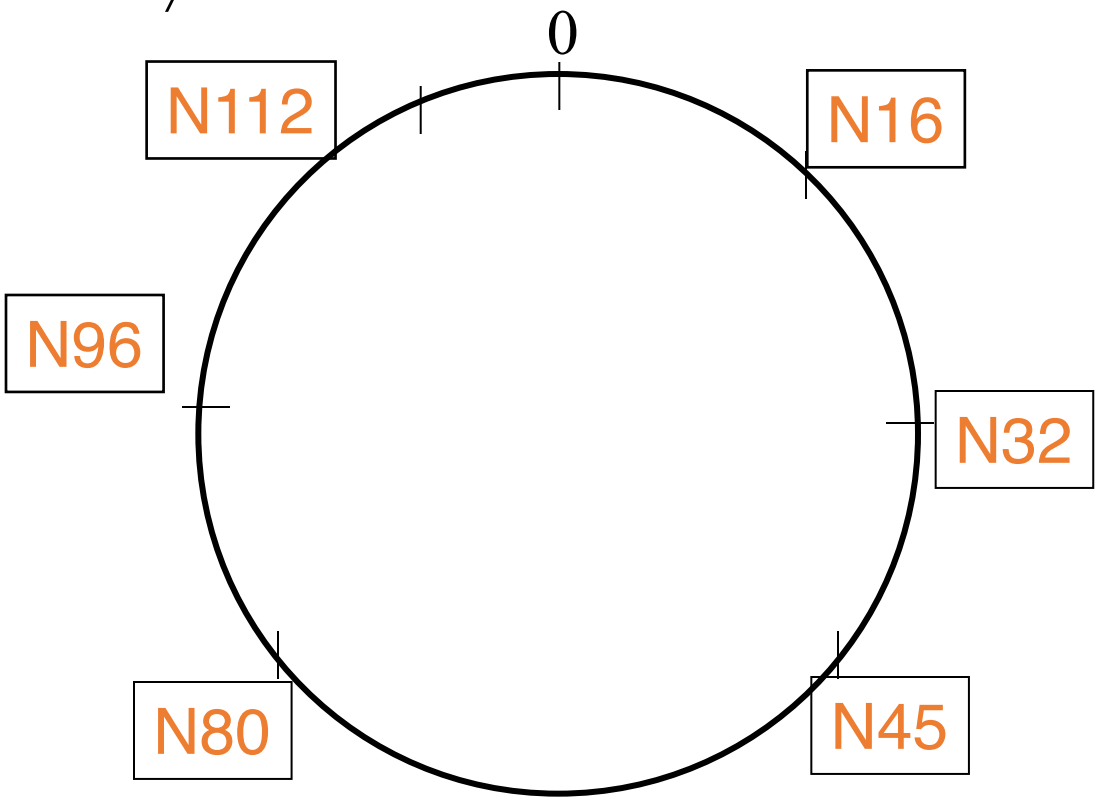


Circle for m = 3

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.

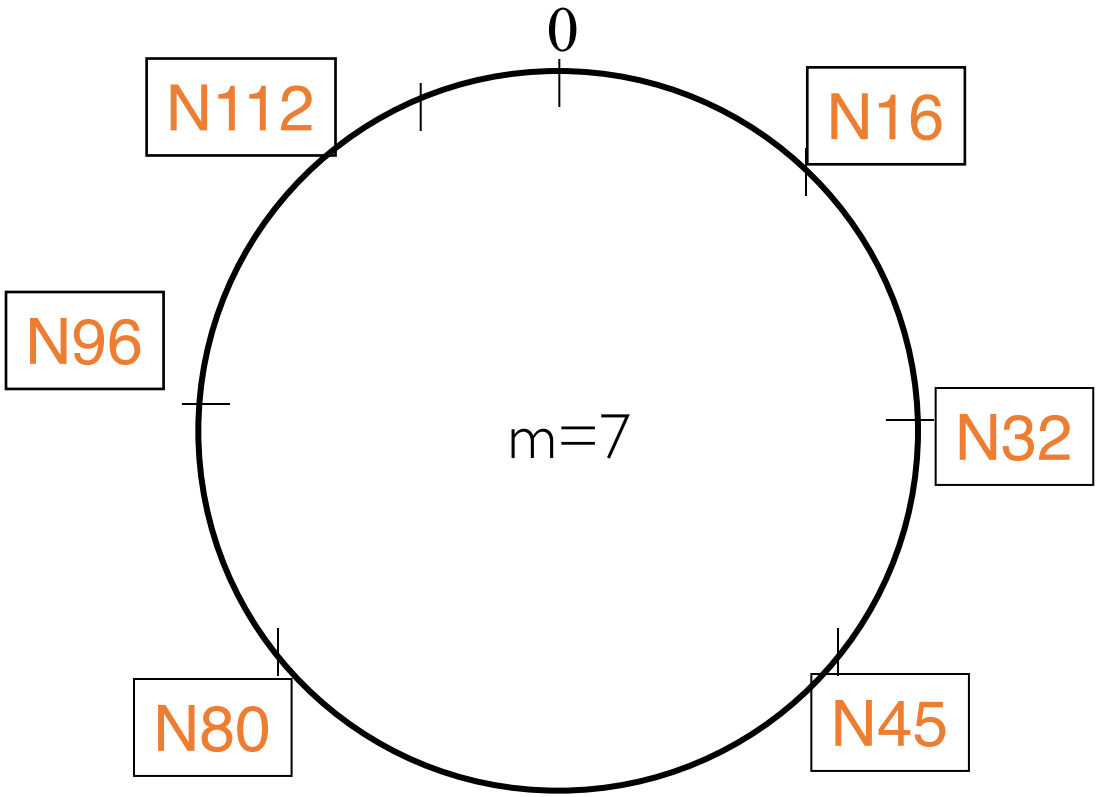


Mapping Keys to Nodes

- 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 node 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)*

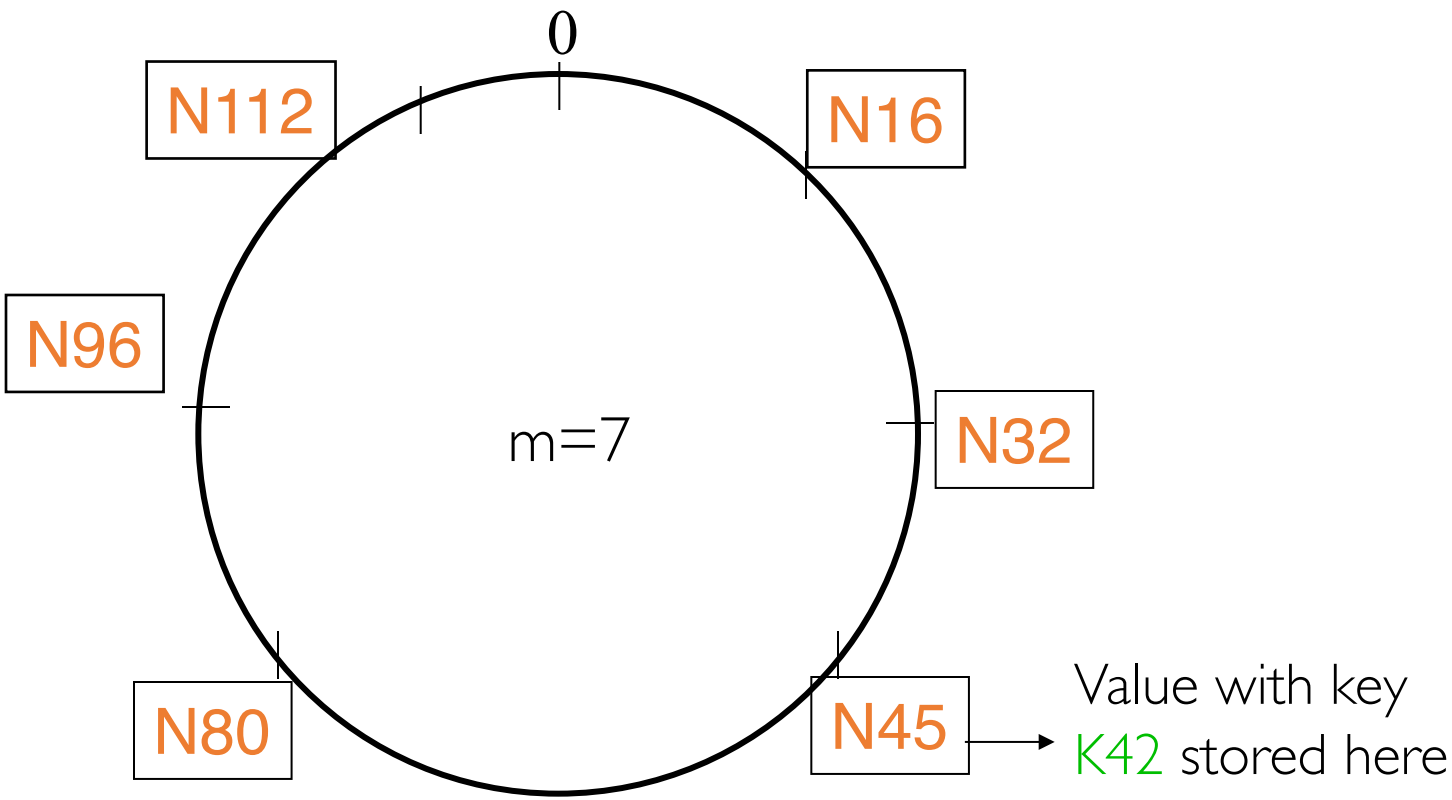
- Consistent Hashing => with K keys and N node, each node stores O(K/N) keys. (i.e., < c.K/N, for some constant c)

Ring of Peers: Running Example



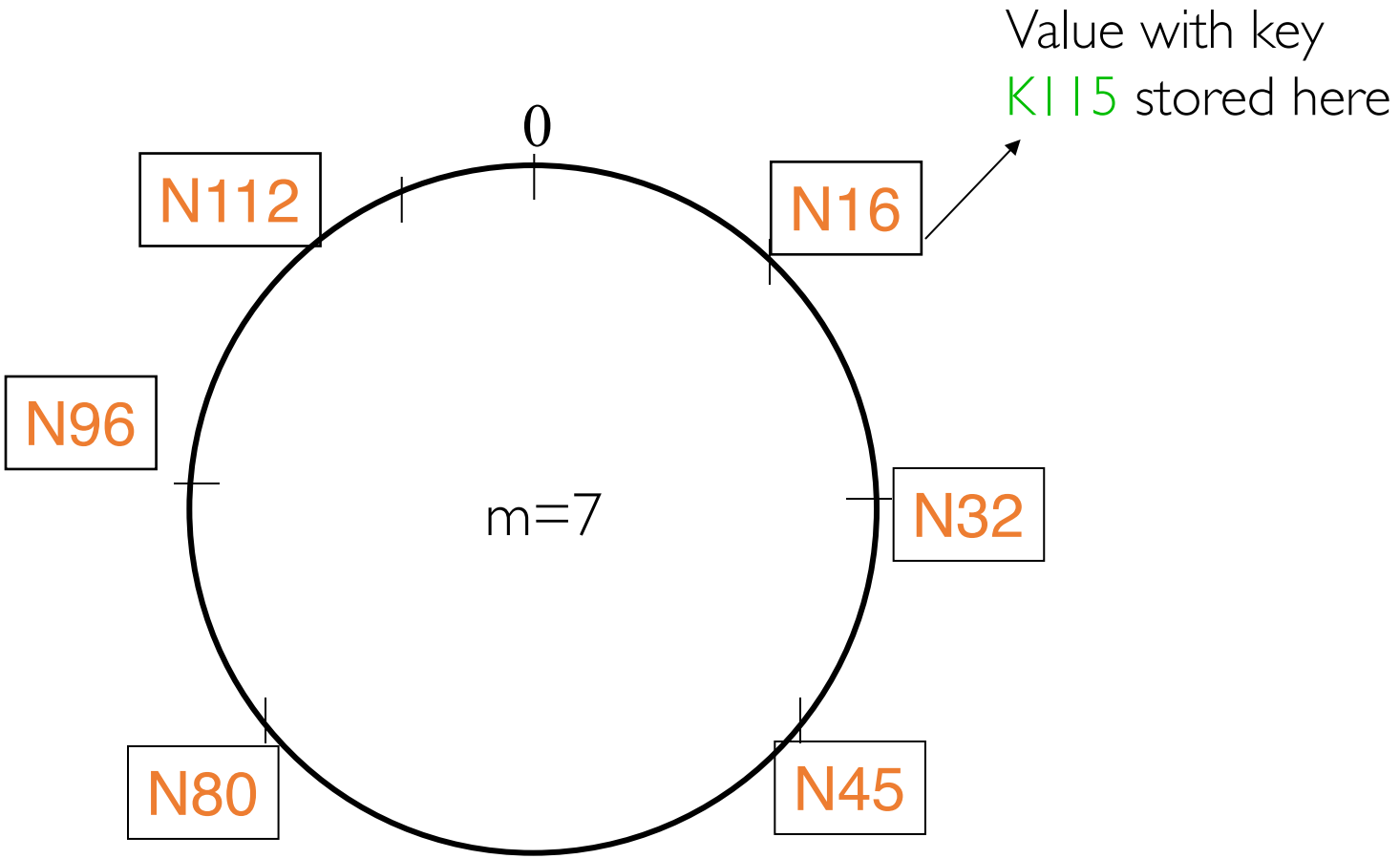
Where will the value with key 42 be stored?

Ring of Peers: Running Example



Where will the value with key 42 be stored?

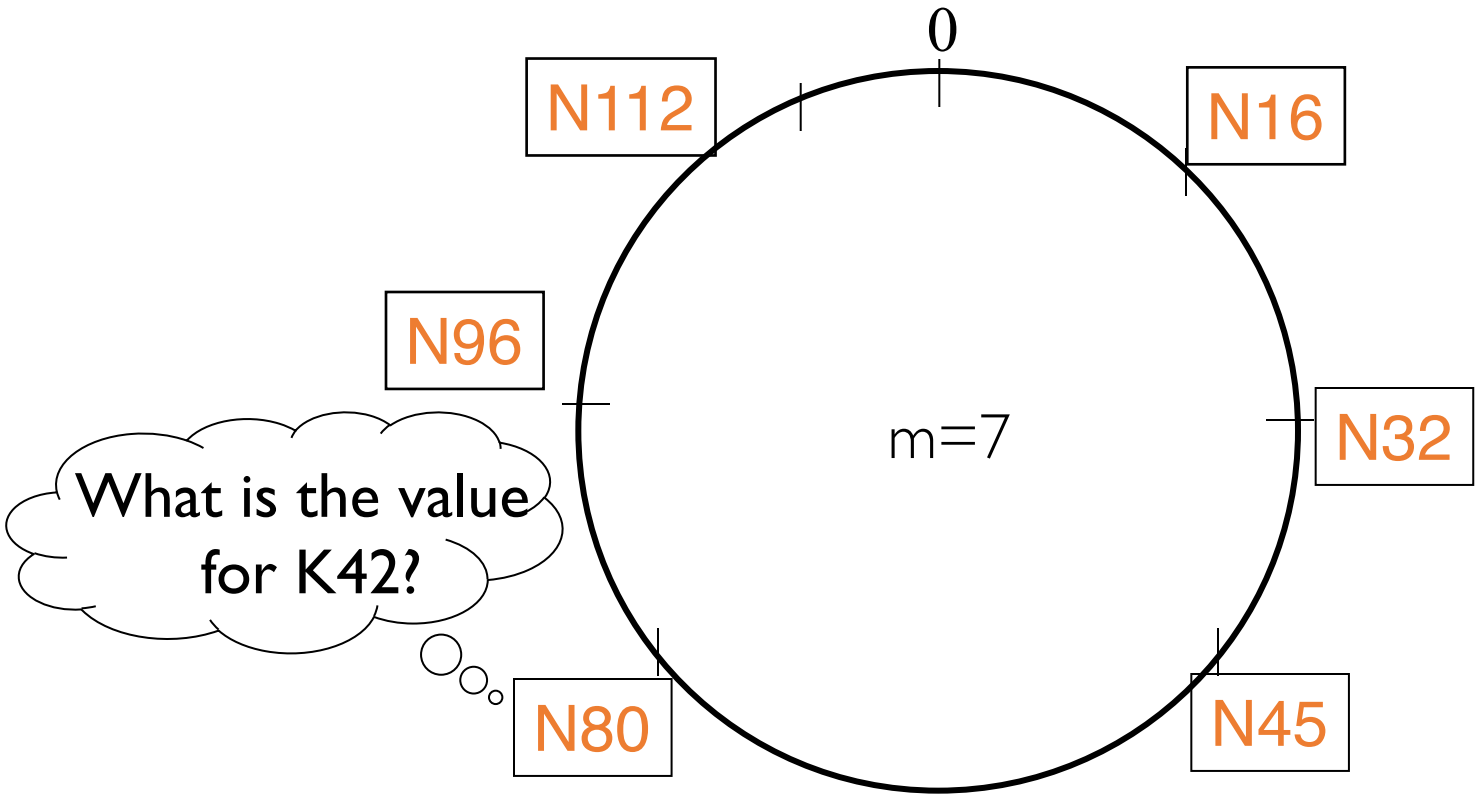
Ring of Peers: Running Example



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 (the next node in the ring).
  - O(N) lookup. Not very efficient.
- Chord chooses a sweet middle-ground.

Performing Lookups

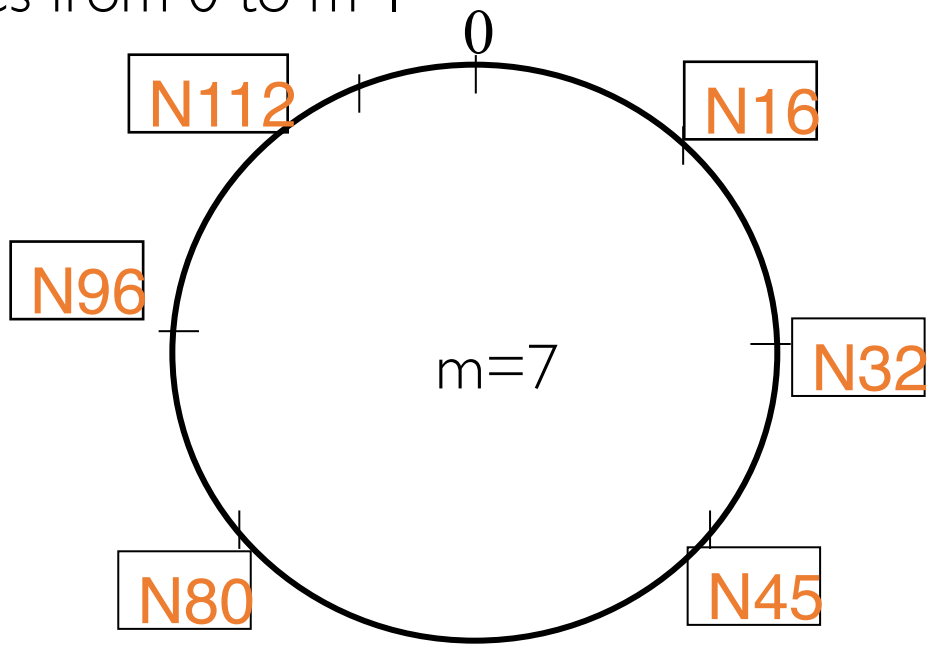
- Chord chooses a sweet middle-ground.
  - Each node is aware of ~**m** other nodes.
  - Maintains a *finger table* with **m** entries.
  - The ith entry of node n's finger table = successor($n + 2^i$)
    - i ranges from 0 to m-1
    - Recall: *successor(x) = first node with id greater than or equal to ($x \bmod 2^m$)*

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



To be continued in next class…..