NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CE/CZ4052 Cloud Computing

Consistent Hashing and Distributed Database

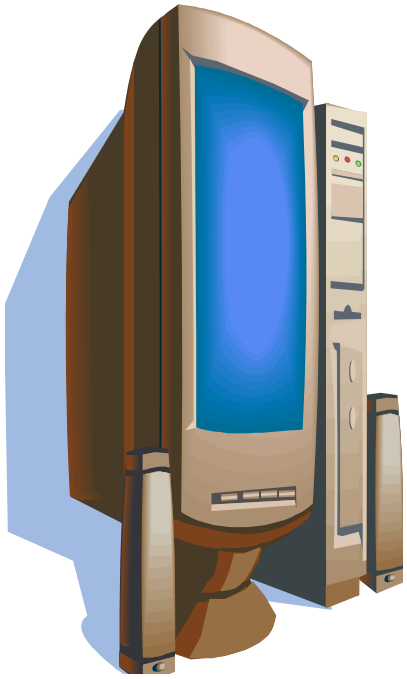
Dr. Tan, Chee Wei

Email: cheewei.tan@ntu.edu.sg

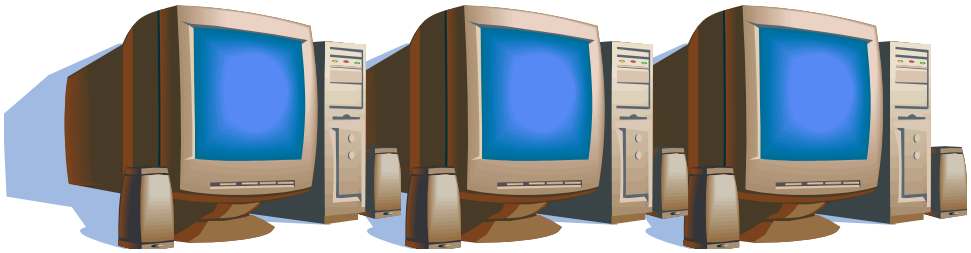
Office: N4-02c-104

Adapted from Princeton University Course COS 418: Distributed Systems "Scaling Out Key-Value Storage"

Horizontal or vertical scalability?



Vertical Scaling



Horizontal Scaling

Horizontal scaling is challenging

- 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!

Main challenge: Coping with constant failures

Today

1. **Techniques for partitioning data**
   - **Consistent Hashing**

2. Case study: Amazon Dynamo key-value store

Scaling out: Placement

- You have key-value pairs to be partitioned across nodes based on an id

- **Problem 1: Data placement**
  - **On which node(s)** to **place** each key-value pair?
    - Maintain mapping from data object to node(s)
    - Evenly distribute data/load

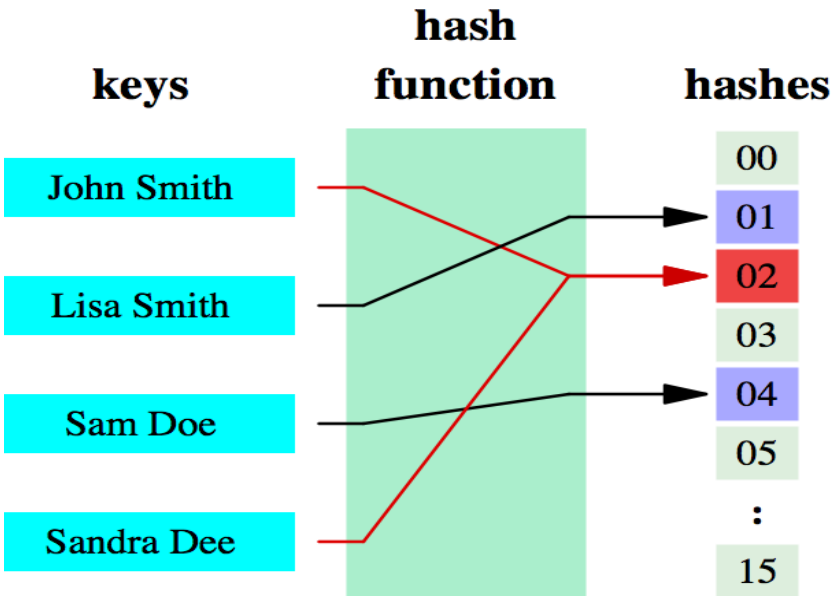
Scaling out: Partition Management

- **Problem 2: Partition management**
  - Including how to recover from node failure
    - *e.g.,* bringing another node into partition group
  - Changes in system size, *i.e.* **nodes joining/leaving**
  - Heterogeneous nodes

- **Centralized:** Cluster manager
- **Decentralized:** Deterministic hashing and algorithms

Hashing

- Hash function
  - Function that maps a large, possibly variable-sized datum into a small datum, often a single integer that serves to index an associative array
  - In short: maps n-bit datum into k buckets ($k << 2^n$)
  - Provides time- & space-saving data structure for lookup
- Main goals:
  - Low cost
  - Deterministic
  - Uniformity (load balanced)



Basic Hash Techniques

- Simple approach for uniform data
  - If data distributed uniformly over N, for N >> n
  - Hash fn = <data> mod n
  - Fails goal of uniformity if data not uniform

- Non-uniform data, variable-length strings
  - Typically split strings into blocks
  - Perform rolling computation over blocks
    - CRC32 checksum
    - Cryptographic hash functions (SHA-1 has 64 byte blocks)

Applying Basic Hashing

- Consider problem of data partition:
  - Given document X, choose one of k servers to use
- Suppose we use modulo hashing
  - Number servers 1..k
  - Place X on server *i = (X mod k)*
    - Problem? Data may not be uniformly distributed
  - Place X on server *i = hash (X) mod k*
    - Problem?
      - What happens if a server fails or joins (k → k±1)?
      - What is different clients has different estimate of k?
      - Answer: All entries get remapped to new nodes!

Modulo hashing

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

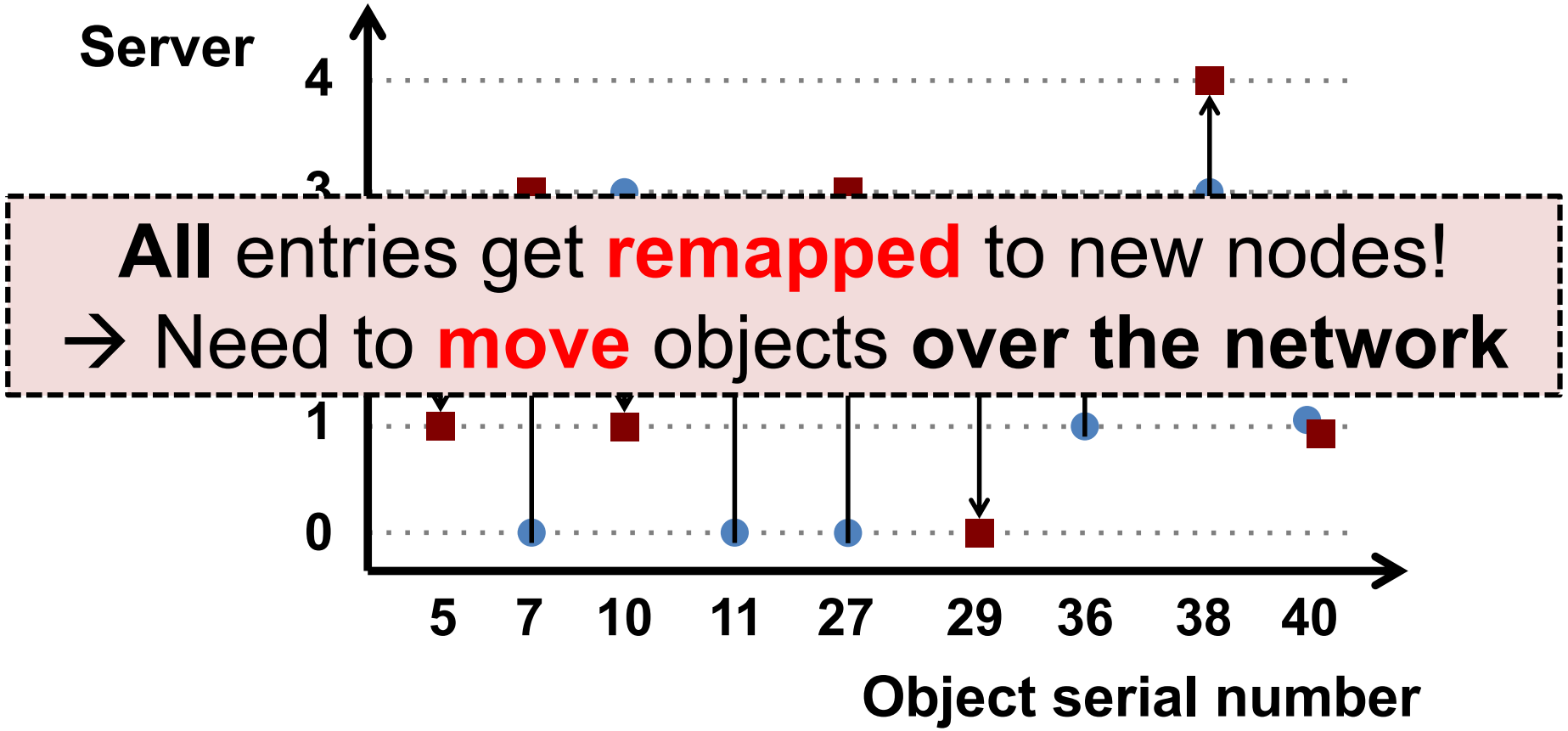
- Suppose instead we use **modulo hashing:**
  - Place ***X*** on server ***i* = hash(*X*) mod *k***

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

Problem for modulo hashing: Changing number of servers

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

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

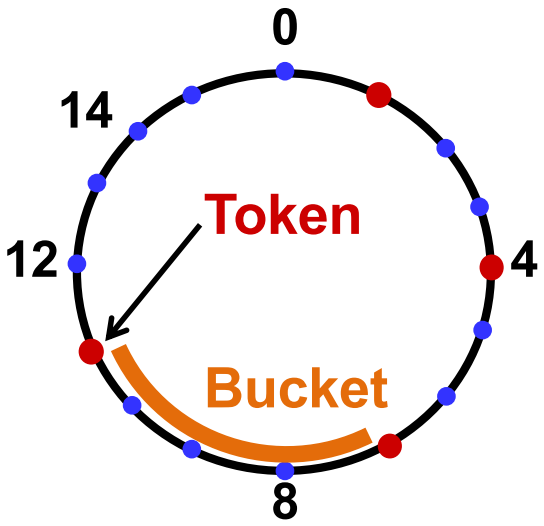


All entries get **remapped** to new nodes!

→ Need to **move** objects **over the network**

Consistent hashing

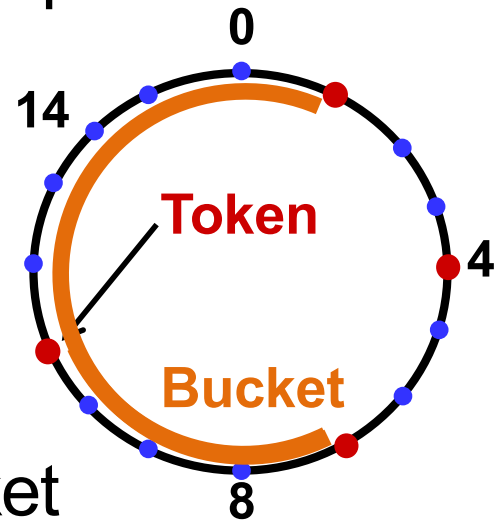
- Assign *n* ***tokens*** to random points on mod 2^k circle; hash key size = *k*
- Hash object to random circle position
- Put object in **closest clockwise bucket**
  - ***successor*** (key) → bucket



- **Desired features –**
  - **Balance:** No bucket has "too many" objects; E(bucket size)=1/ n^{th}
  - **Smoothness:** Addition/removal of token **minimizes object movements** for other buckets

Consistent hashing's load balancing problem

- Each node owns **1/nth** of the ID space in expectation
  - Hot keys => some buckets have higher request rate



- 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 implements v ***virtual*** nodes
  - Each **physical node** maintains **v > 1 token ids**
    - Each token id corresponds to a virtual node
    - Each **physical node** can have a different v based on strength of node (heterogeneity)

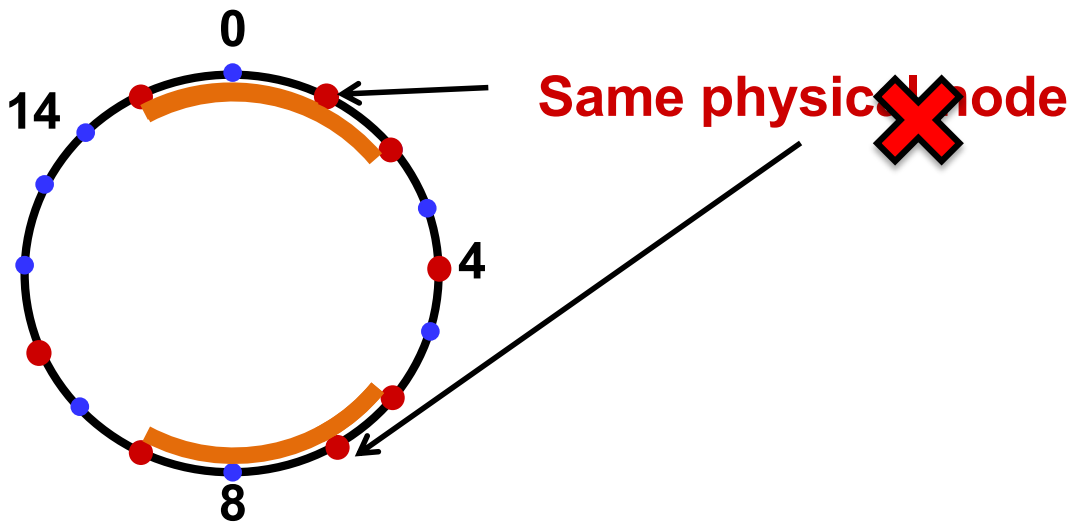
- Each virtual node owns an expected **1/(vn)th** of ID space

- **Upon a physical node's failure, v** virtual nodes fail
  - Their successors take over 1/(vn)th more
    - Expected to be distributed across physical nodes

Virtual nodes: Example

4 Physical Nodes

V=2



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

Key Value Storage

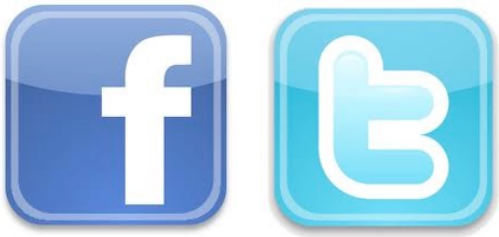
- Interface
  - **put**(key, value); // insert/write "value" associated with "key"
  - value = **get**(key); // get/read data associated with "key"

- Abstraction used to implement
  - File systems: value content → block
  - Sometimes as a simpler but more scalable "database"

- Can handle large volumes of data, e.g., PBs
  - Need to distribute data over hundreds, even thousands of machines

Key Values: Examples

- Amazon:
  - Key: customerID
  - Value: customer profile (e.g., buying history, credit card, ..)



- Facebook, Twitter:
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)

- iCloud/iTunes:
  - Key: Movie/song name
  - Value: Movie, Song

- Distributed file systems
  - Key: Block ID
  - Value: Block

System Examples

- Google File System, Hadoop Dist. File Systems (HDFS)

- Amazon
  - Dynamo: internal key value store used to power Amazon.com (shopping cart)
  - Simple Storage System (S3)

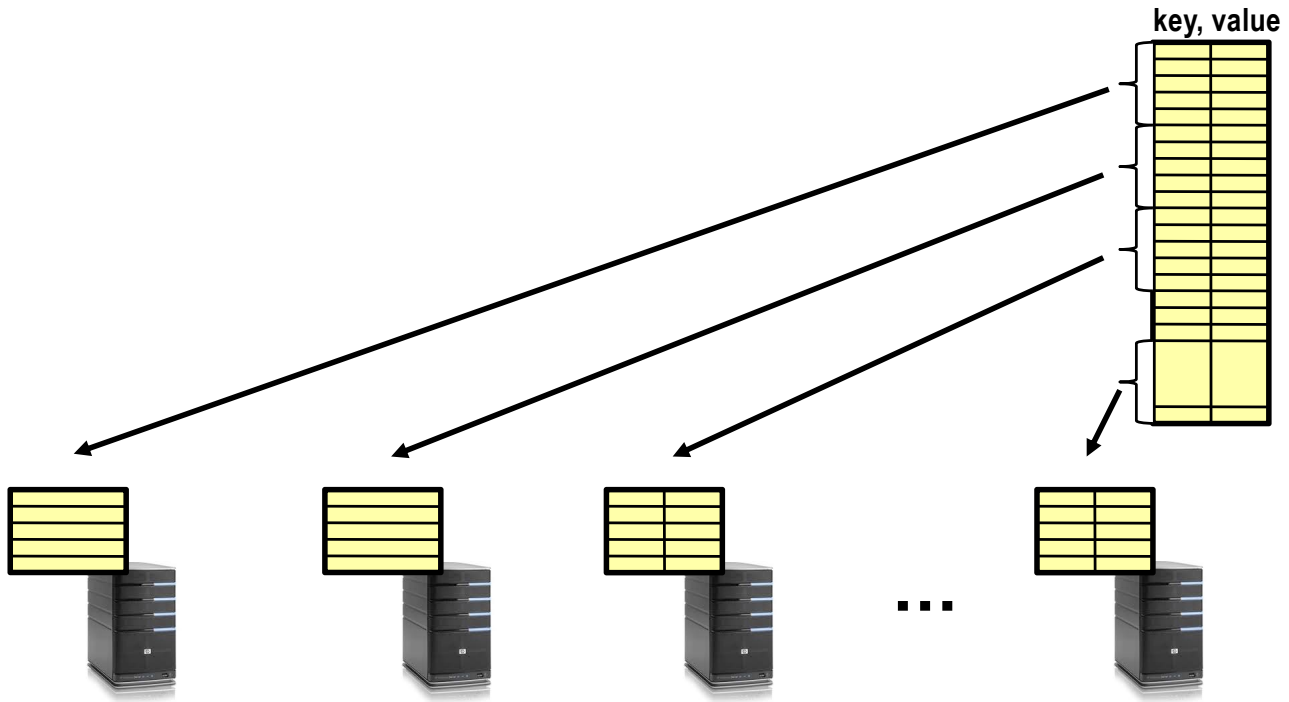
- **BigTable/Hbase:** distributed, scalable data storage

- **Cassandra**: "distributed data management system" (Facebook)

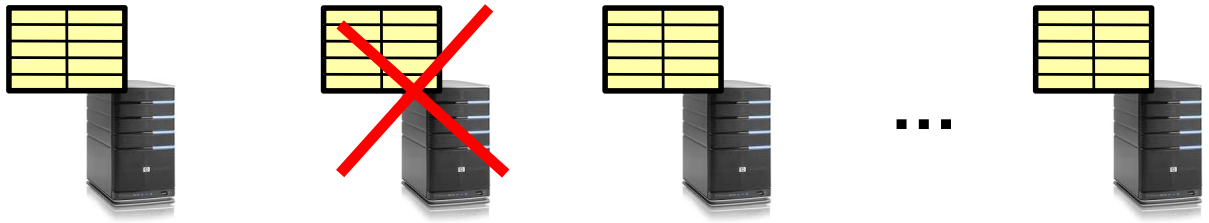
- **Memcached:** in-memory key-value store for small chunks of arbitrary data (strings, objects)

Key Value Store

- Also called a Distributed Hash Table (DHT)
- Main idea: partition set of key-values across many machines



Challenges



- **Fault Tolerance:** handle machine failures without losing data and without degradation in performance
- **Scalability:**
  - Need to scale to thousands of machines
  - Need to allow easy addition of new machines
- **Consistency:** maintain data consistency in face of node failures and message losses
- **Heterogeneity** (if deployed as peer-to-peer systems):
  - Latency: 1ms to 1000ms
  - Bandwidth: 32Kb/s to 100Mb/s

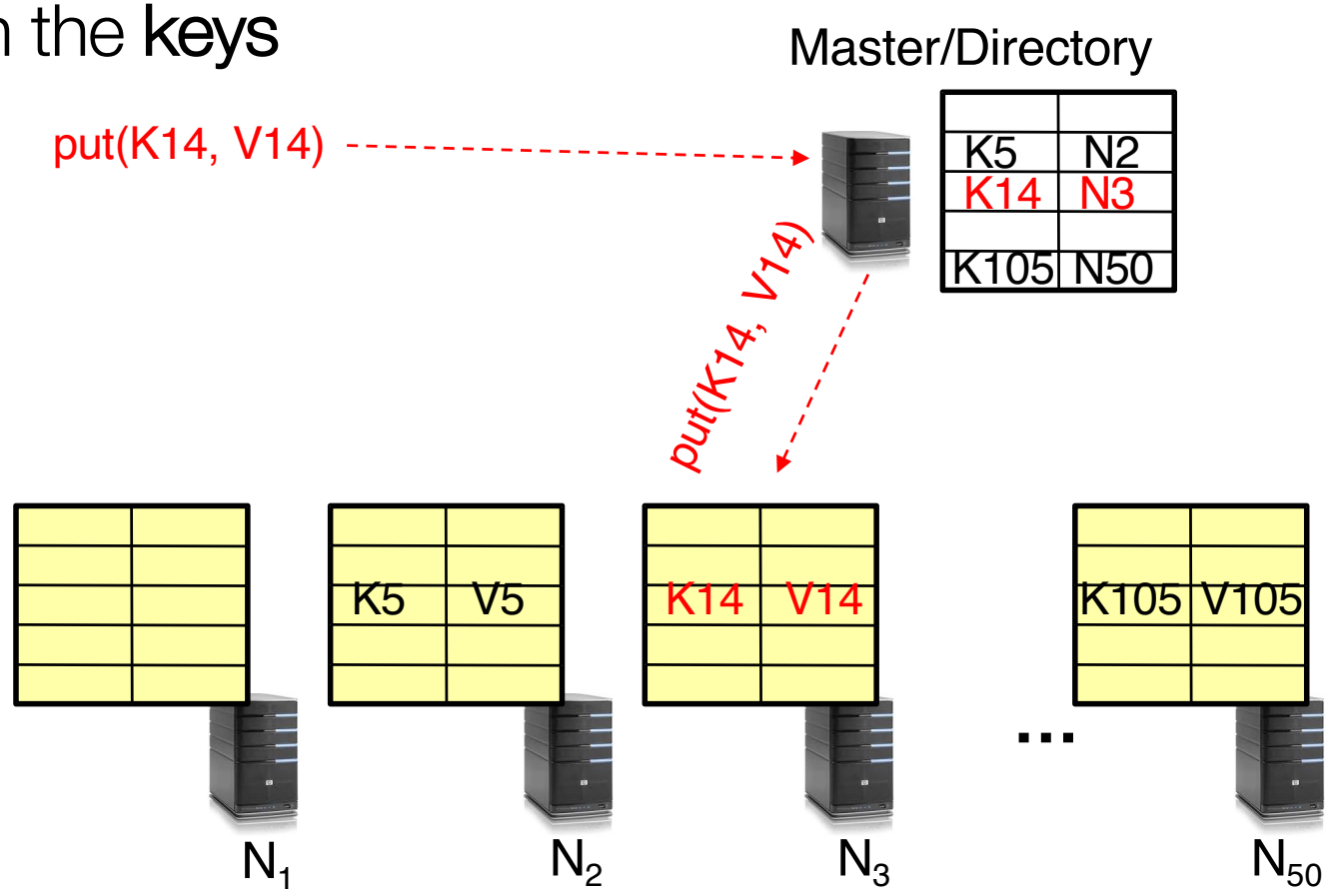
Key Questions

- put(key, value): where do you store a new (key, value) tuple?
- get(key): where is the value associated with a given "key" stored?

- And, do the above while providing
  - Fault Tolerance
  - Scalability
  - Consistency

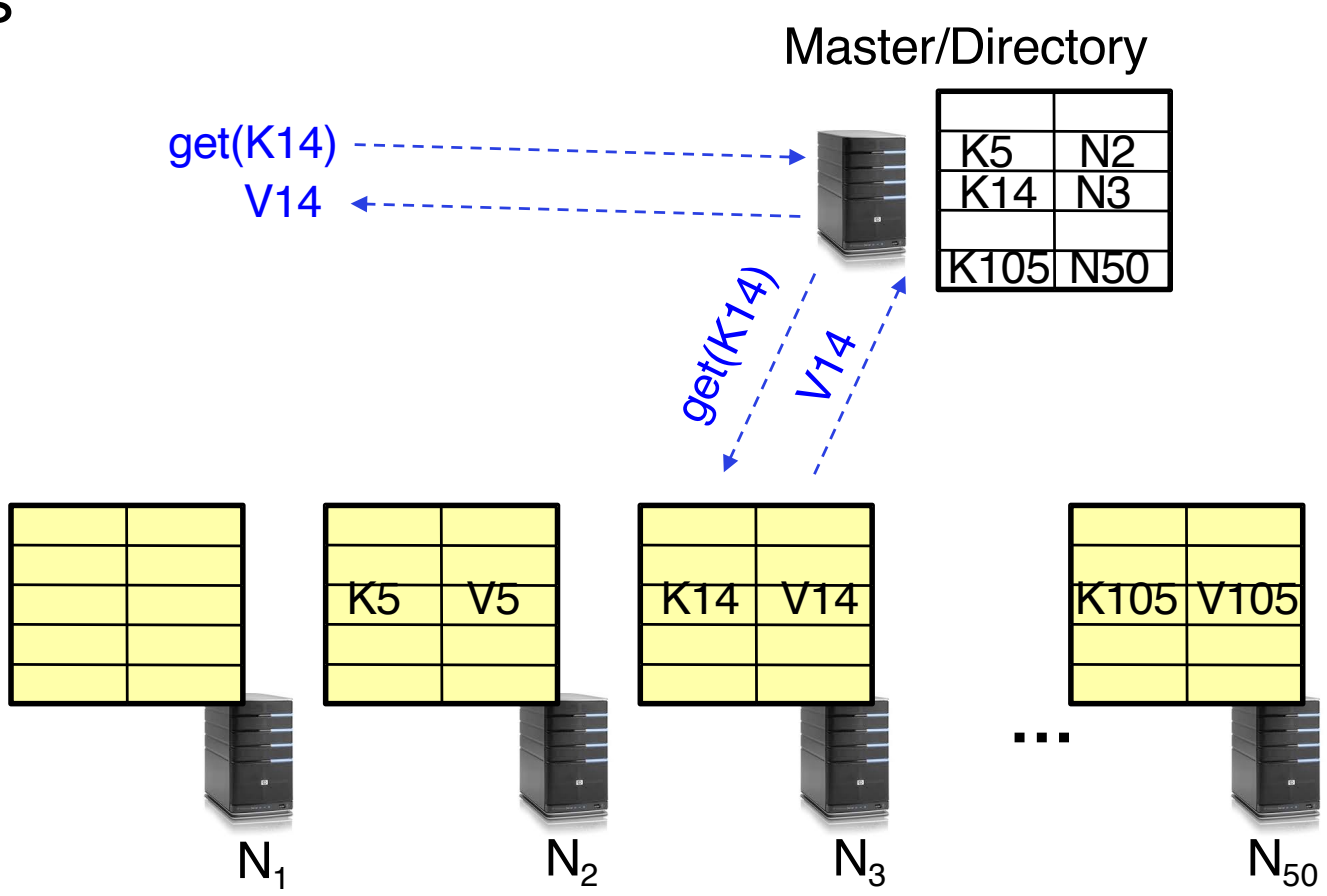
Directory-Based Architecture

- Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the **keys**



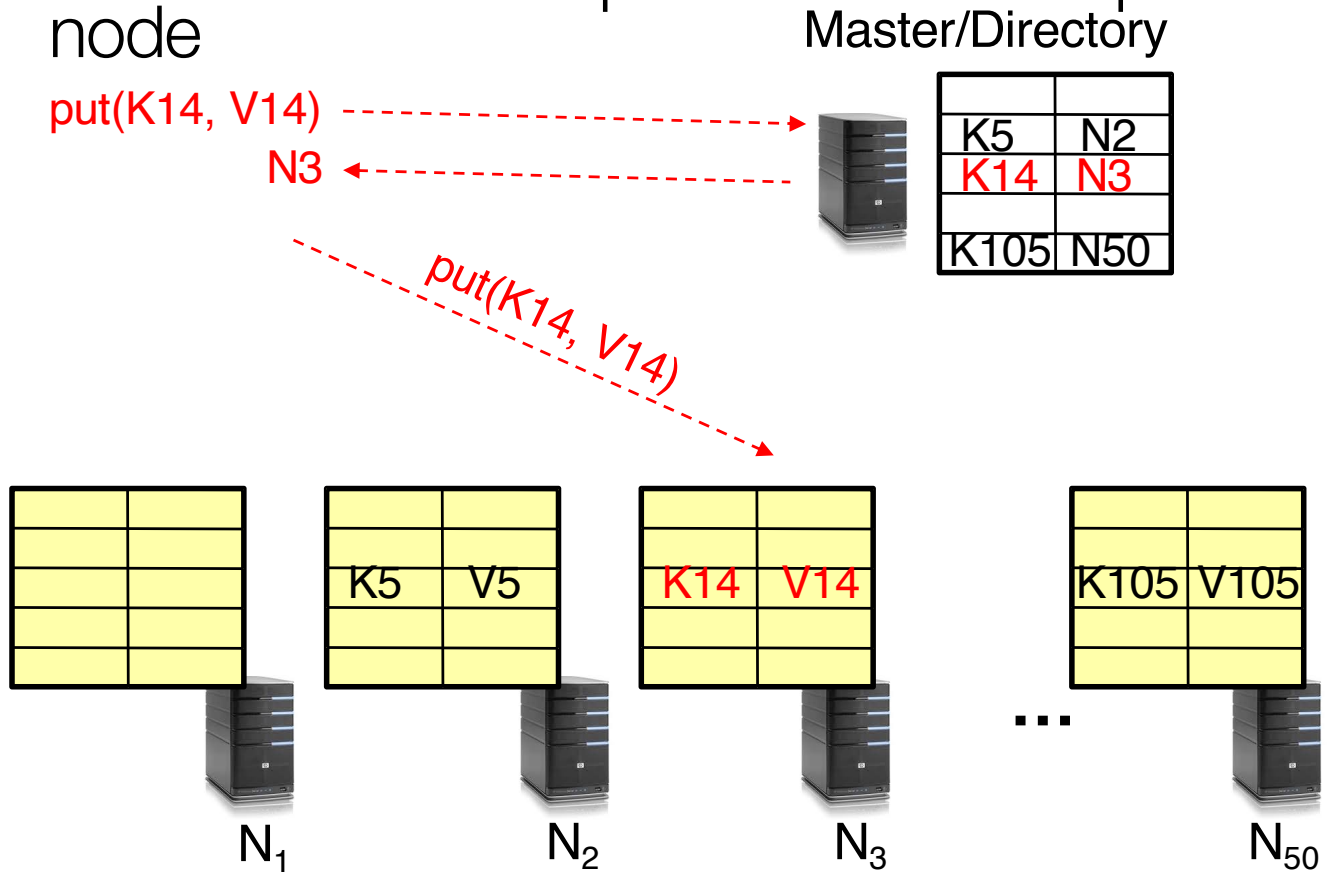
Directory-Based Architecture

- Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the keys



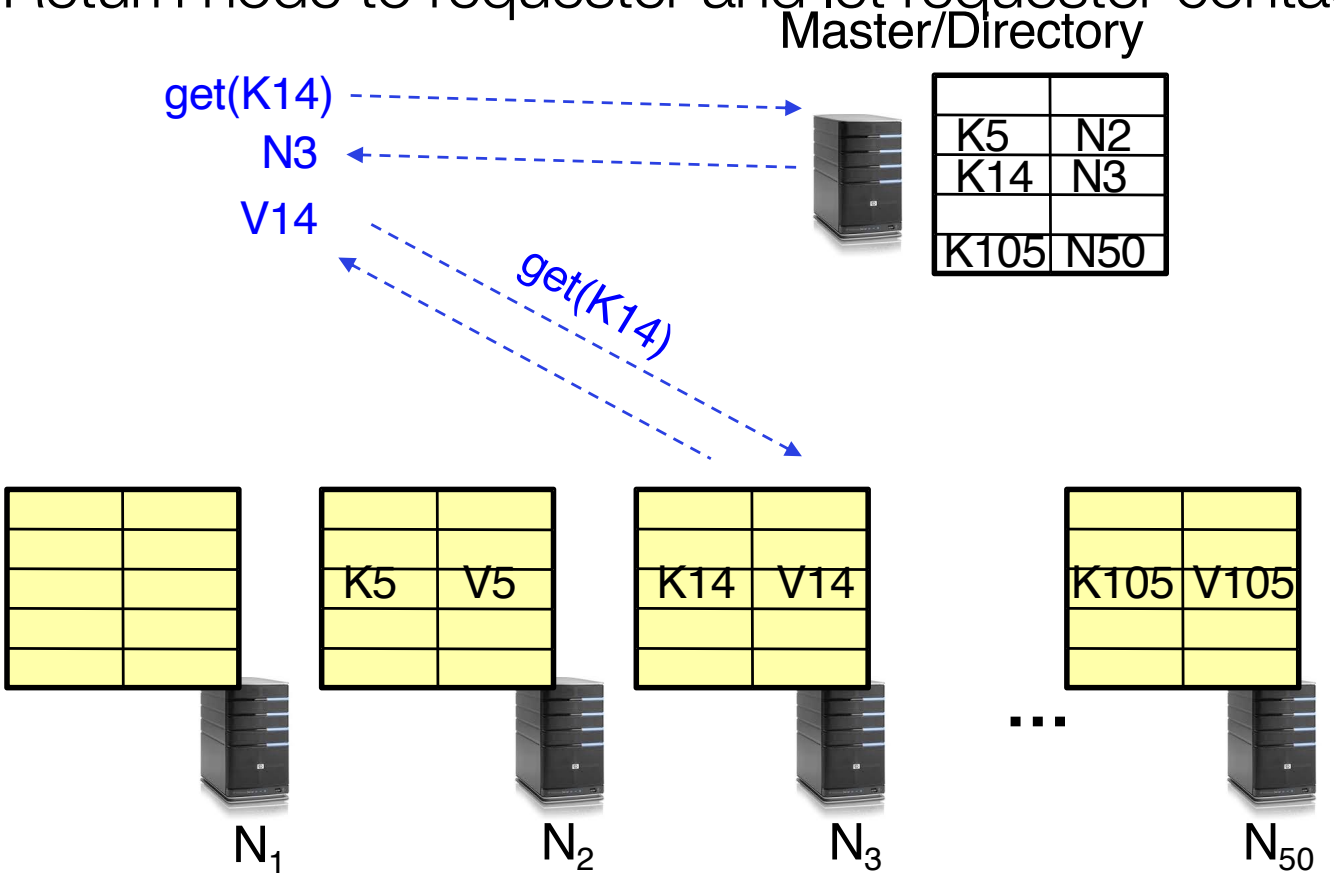
Directory-Based Architecture

- Having the master relay the requests → **recursive query**
- Another method: **iterative query** (this slide)
  - Return node to requester and let requester contact node

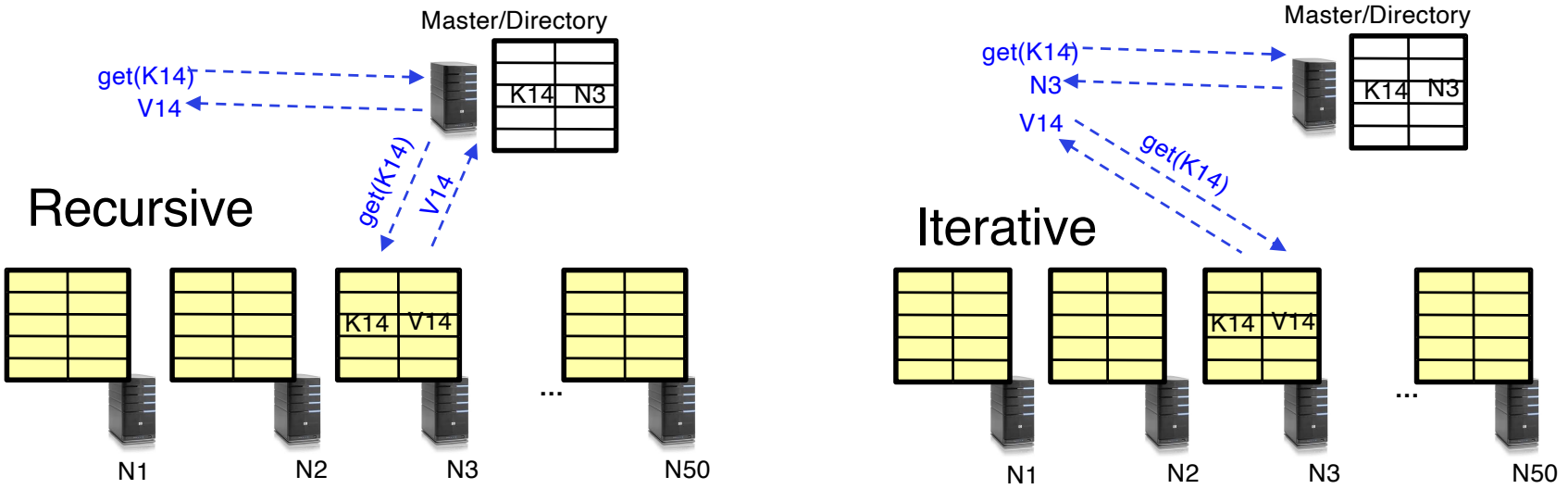


Directory-Based Architecture

- Having the master relay the requests → **recursive query**
- Another method: **iterative query**
  - Return node to requester and let requester contact node



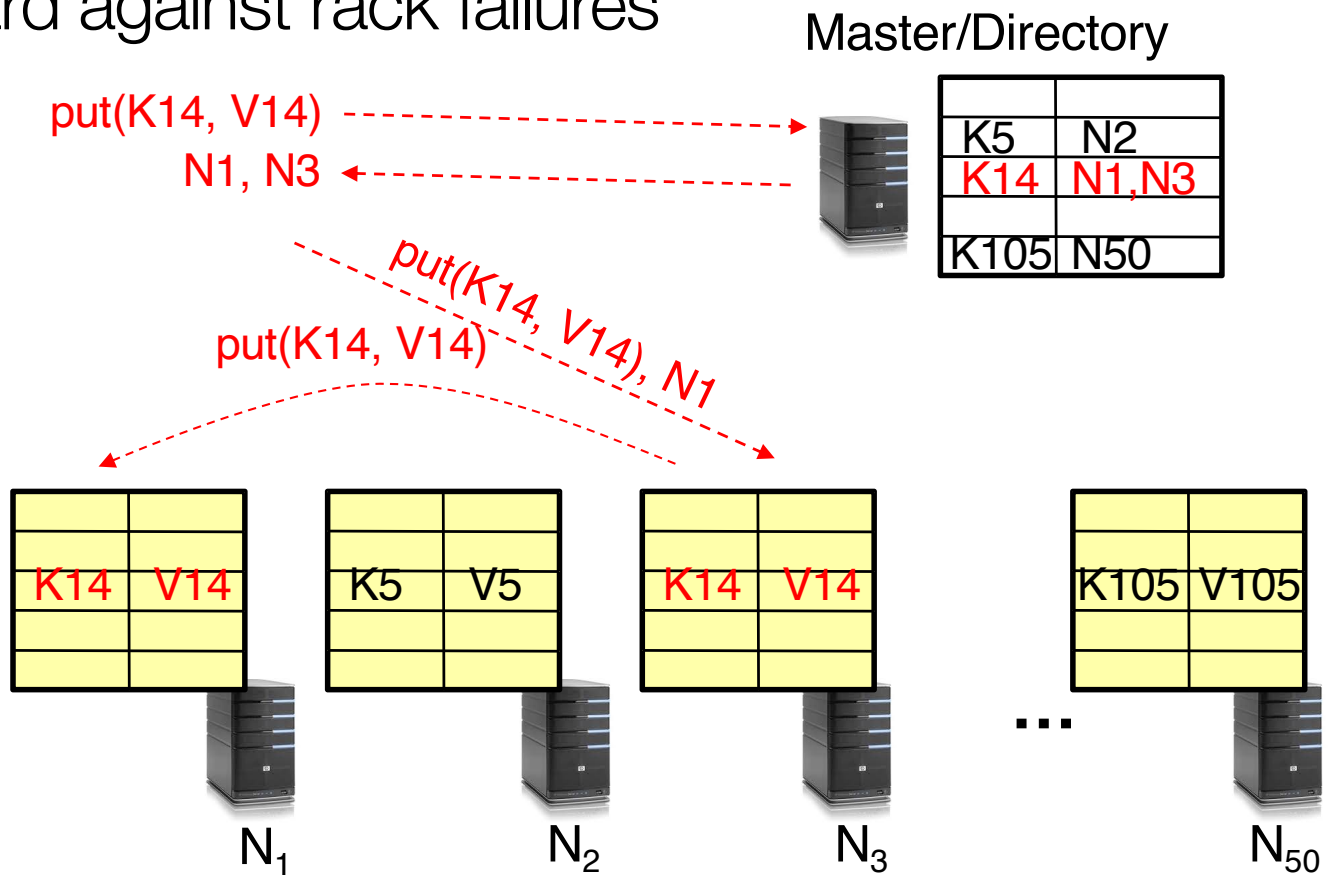
Discussion: Iterative vs. Recursive Query



- Recursive Query:
  - Advantages:
    - Faster, as typically master/directory closer to nodes
    - Easier to maintain consistency, as master/directory can serialize puts()/gets()
  - Disadvantages: scalability bottleneck, as all "Values" go through master
- Iterative Query
  - Advantages: more scalable
  - Disadvantages: slower, harder to enforce data consistency

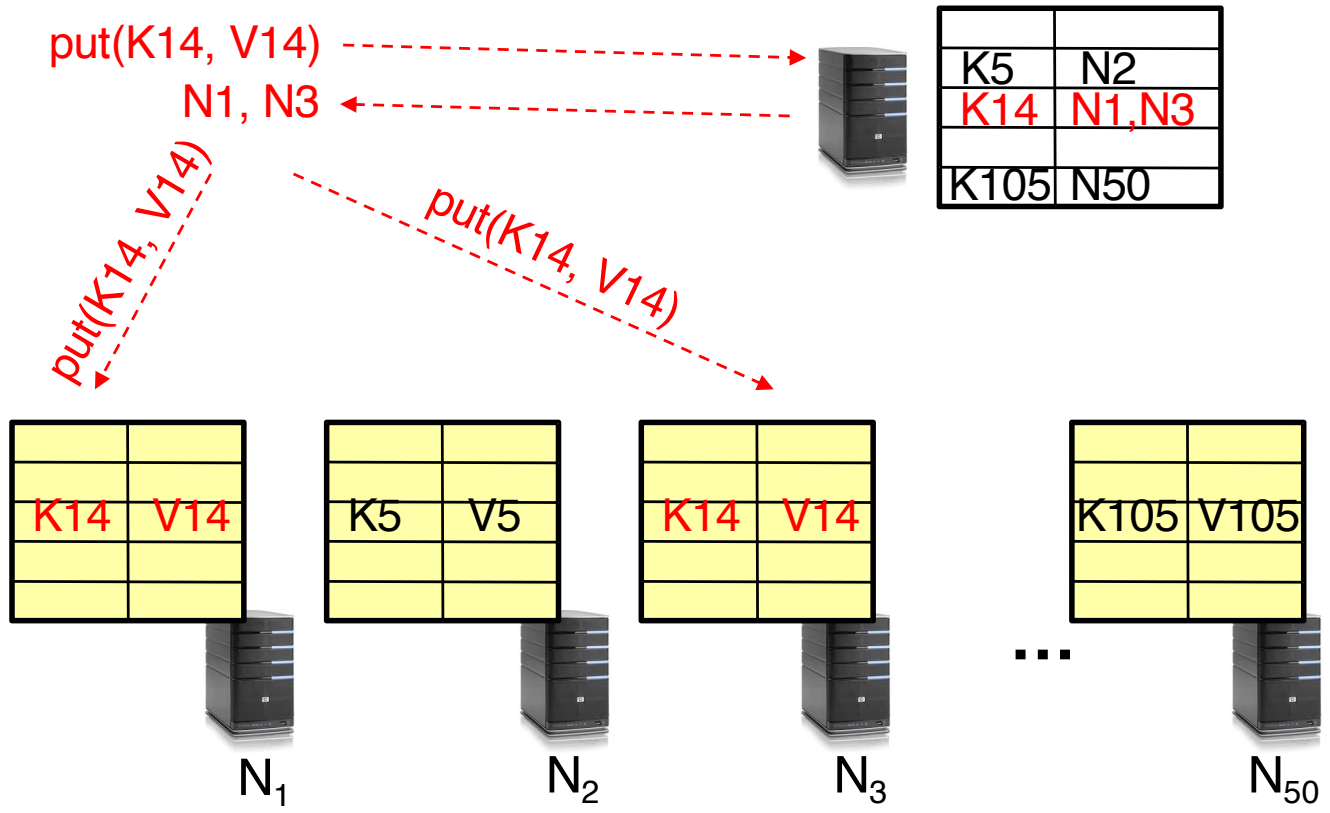
Fault Tolerance

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures



Fault Tolerance

- Again, we can have
  - **Recursive** replication (previous slide)
  - **Iterative** replication (this slide)



Scalability

- Storage: use more nodes

- Request throughput:
  - Can serve requests from all nodes on which a value is stored in parallel
  - Master can replicate a popular value on more nodes

- Master/directory scalability:
  - Replicate it
  - Partition it, so different keys are served by different masters/directories (see Chord)

Scalability: Load Balancing

- Directory keeps track of the storage availability at each node
  - Preferentially insert new values on nodes with more storage available
- What happens when a new node is added?
  - Cannot insert only new values on new node. Why?
  - Move values from the heavy loaded nodes to the new node
- What happens when a node fails?
  - Need to replicate values from fail node to other nodes

Replication Challenges

- Need to make sure that a value is replicated correctly

- How do you know a value has been replicated on every node?
  - Wait for acknowledgements from every node

- What happens if a node fails during replication?
  - Pick another node and try again

- What happens if a node is slow?
  - Slow down the entire put()? Pick another node?

- In general, with multiple replicas
  - Slow puts and fast gets

Consistency

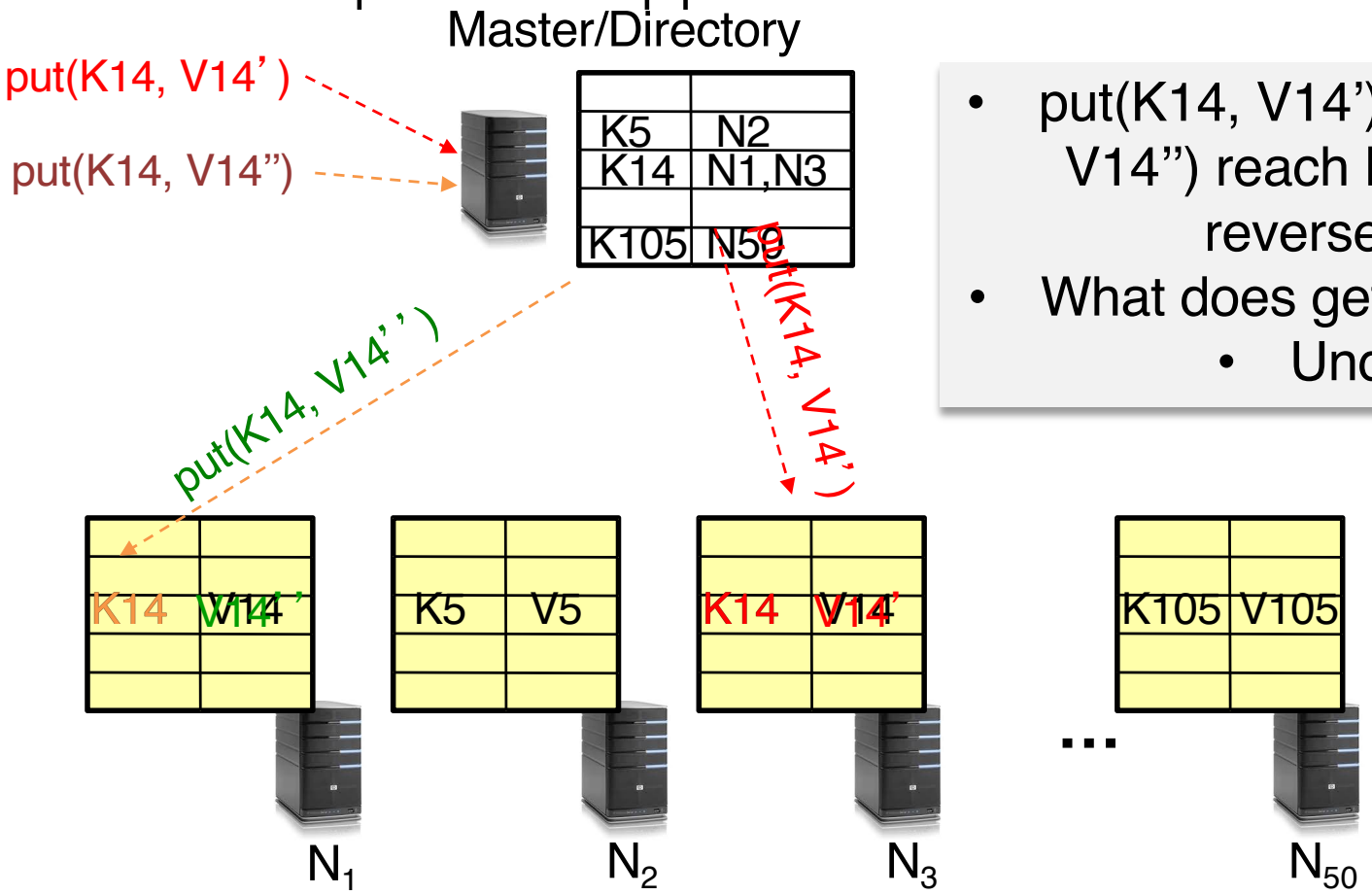
- How close does a distributed system emulate a single machine in terms of read and write semantics?
- **Q:** Assume **put(K14, V14')** and **put(K14, V14'')** are concurrent, what value ends up being stored?
- **A:** assuming **put()** is atomic, then either **V14'** or **V14''**, right?

- **Q:** Assume a client calls **put(K14, V14)** and then **get(K14)**, what is the result returned by **get()**?
- **A:** It should be V14, right?

- Above semantics, not trivial to achieve in distributed systems

Concurrent Writes (Updates)

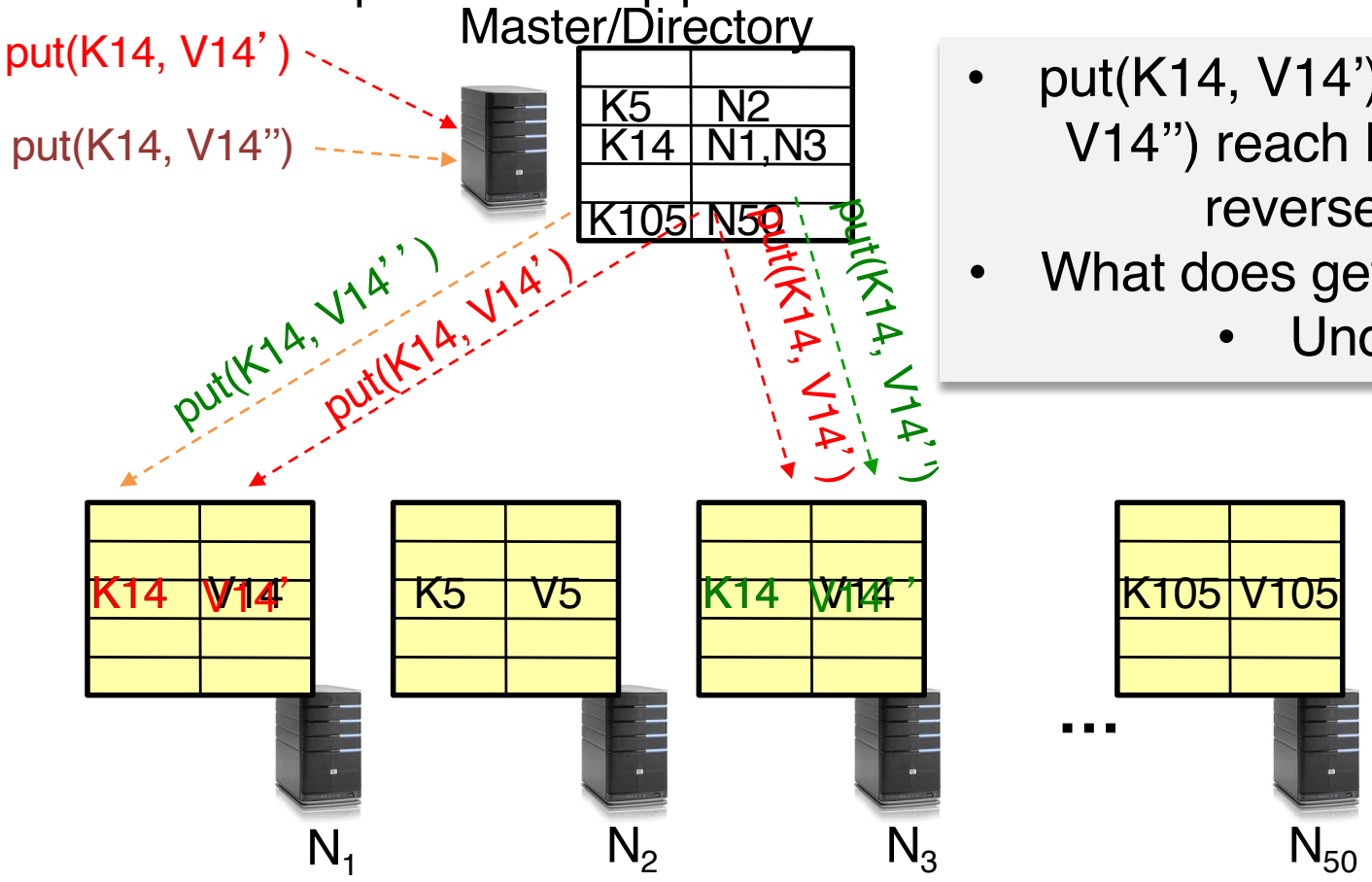
•If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- put(K14, V14') and put(K14, V14") reach N1 and N3 in reverse order
- What does get(K14) return?
  - Undefined!

Concurrent Writes (Updates)

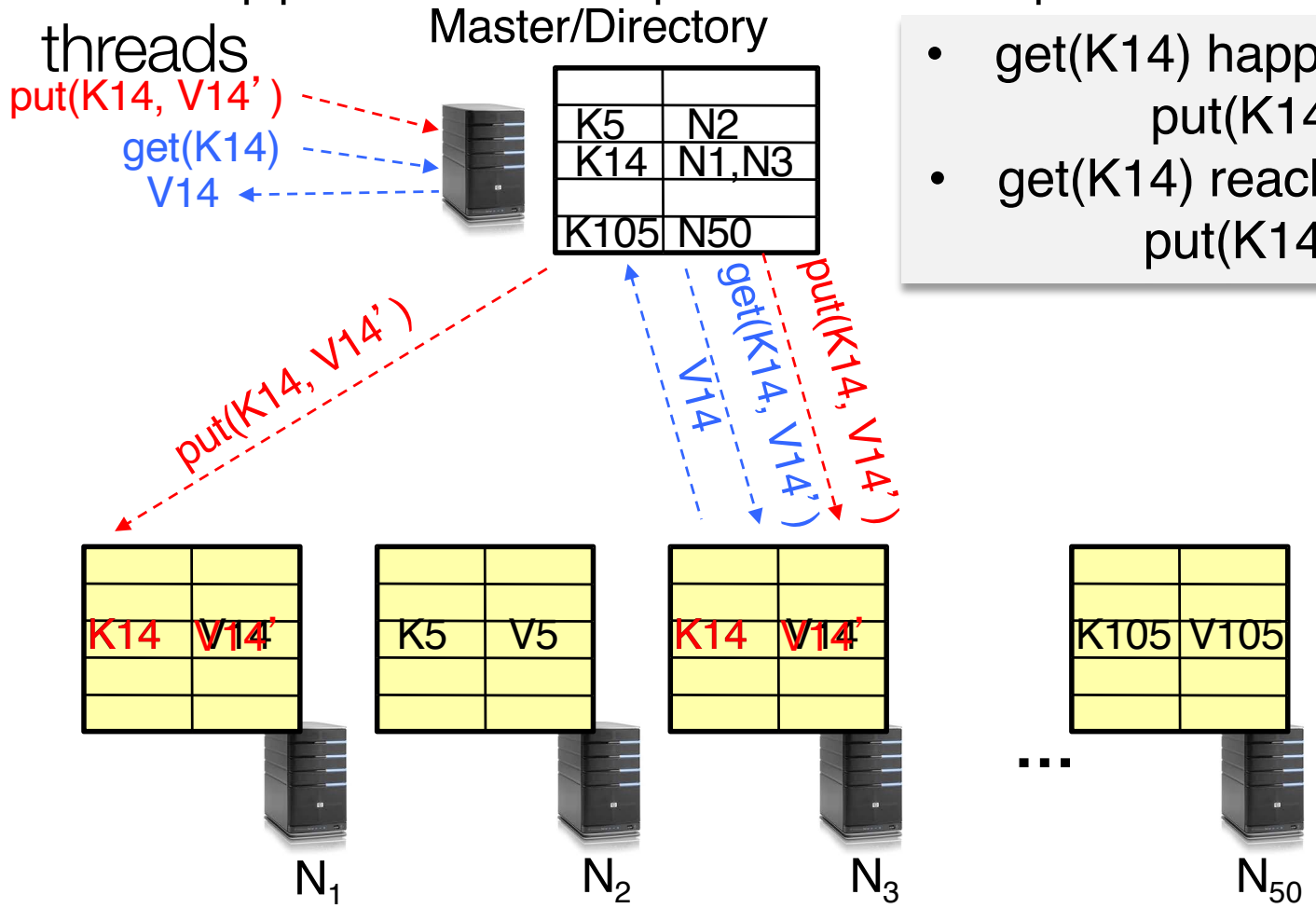
•If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- put(K14, V14') and put(K14, V14") reach N1 and N3 in reverse order
- What does get(K14) return?
  - Undefined!

Read after Write

- Read not guaranteed to return value of latest write
  - Can happen if Master processes requests in different threads



- get(K14) happens right after put(K14, V14')
- get(K14) reaches N3 before put(K14, V14')!

Consistency (cont'd)

- Large variety of consistency models (we've already seen):
  - Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
    - Think "one updated at a time"
    - Transactions
  - Eventual consistency: given enough time all updates will propagate through the system
    - One of the weakest form of consistency; used by many systems in practice
  - And many others: causal consistency, sequential consistency, strong consistency, …

Strong Consistency

- Assume Master serializes all operations

- Challenge: master becomes a bottleneck
  - Not addressed here

- Still want to improve performance of reads/writes → quorum consensus

Quorum Consensus

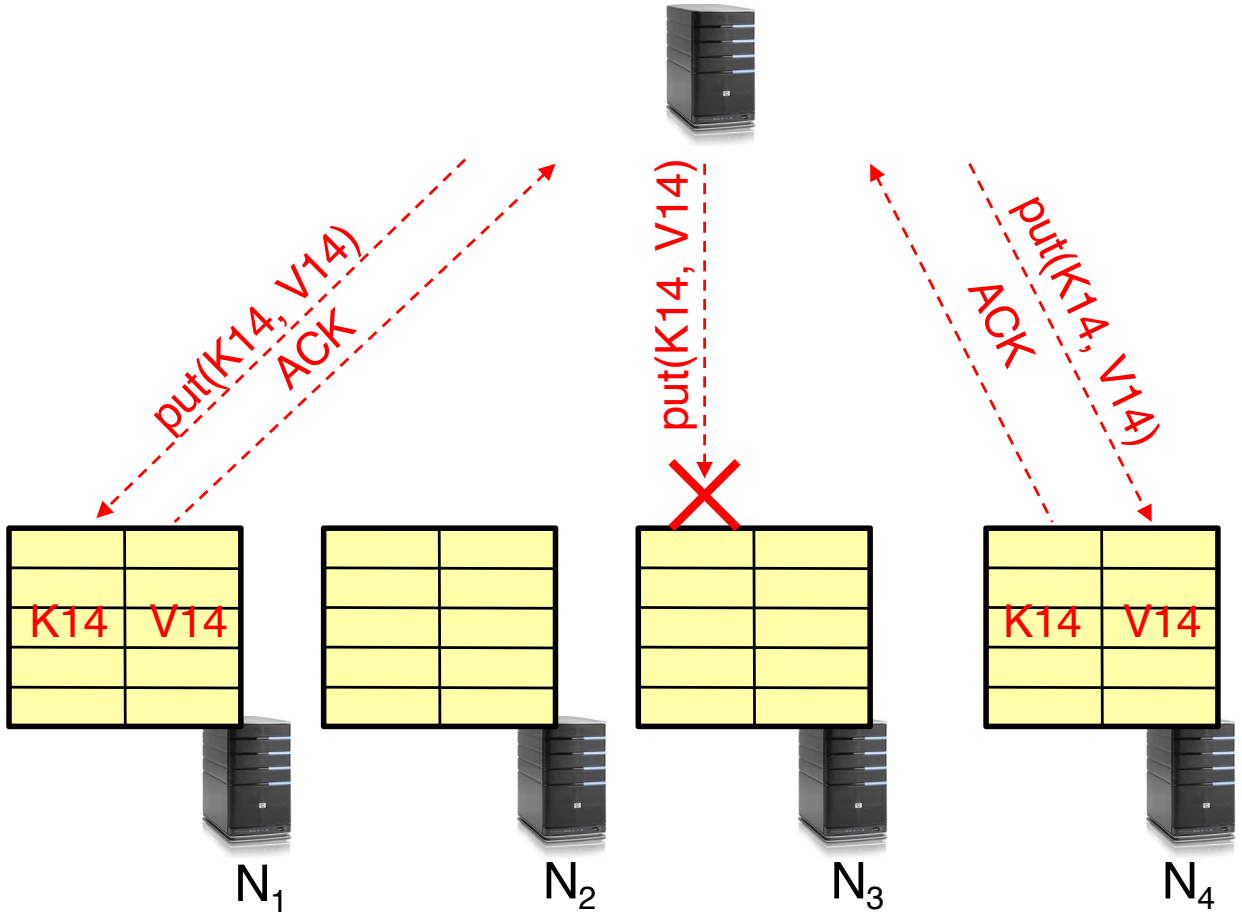
- Improve **put()** and **get()** operation performance

- Define a replica set of size N
- **put()** waits for acks from at least W replicas
- **get()** waits for responses from at least R replicas W+R > N

- Why does it work?
  - There is at least one node that contains the update

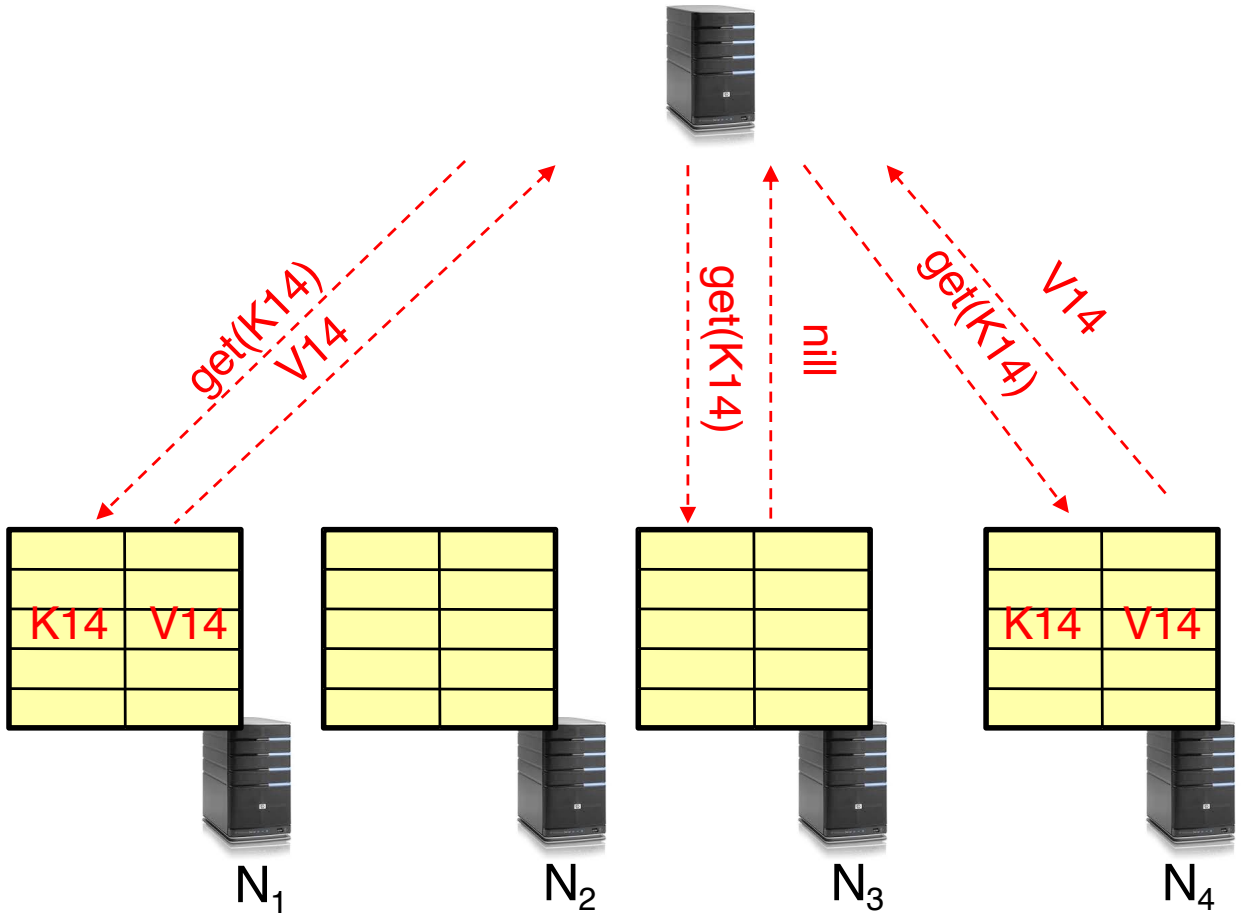
Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: {N1, N2, N4}
- Assume put() on N3 fails



Quorum Consensus Example

- Now, for get() need to wait for any two nodes out of three to return the answer



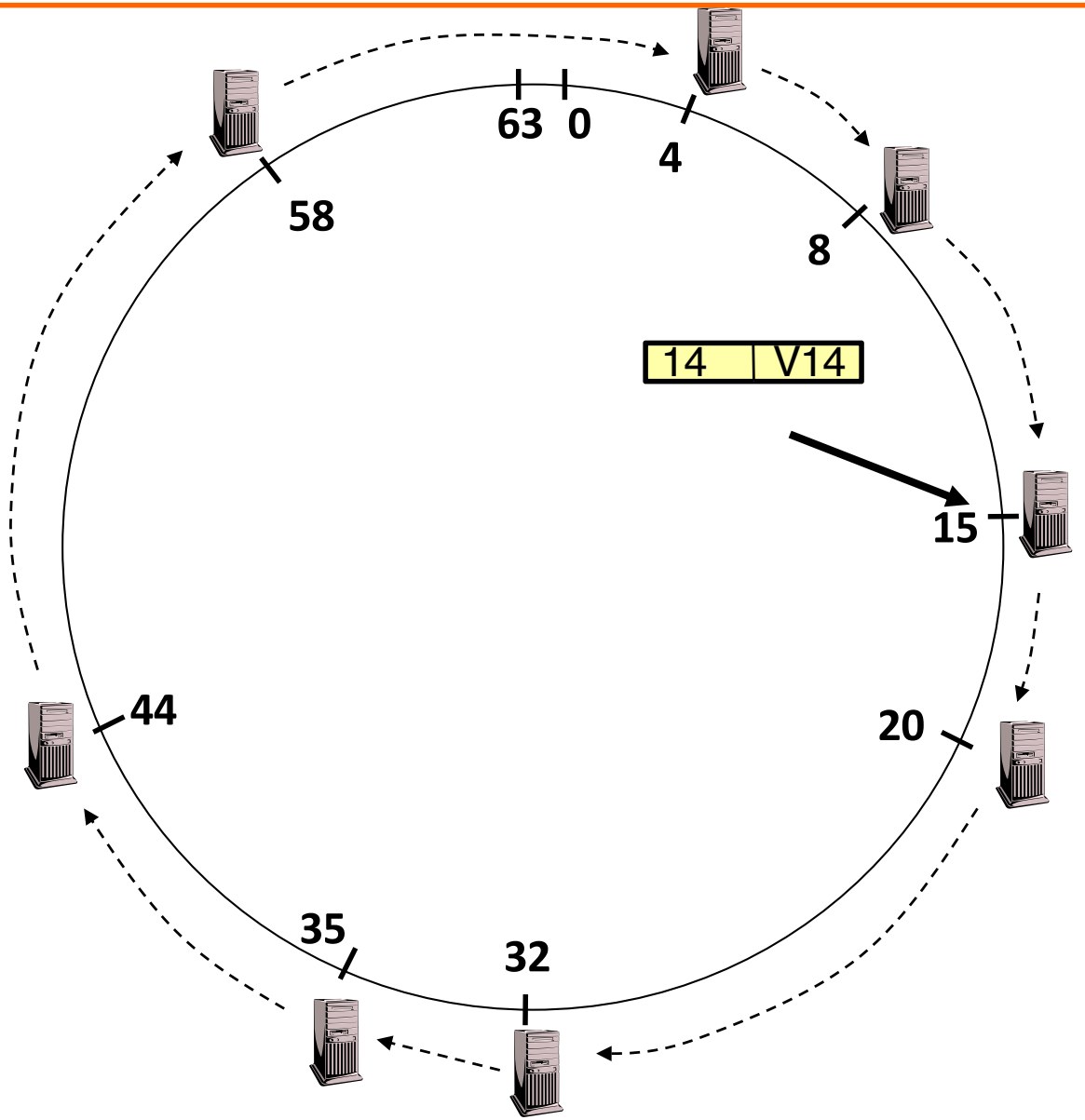
Scaling Up Directory

- Challenge:
  - Directory contains a number of entries equal to number of (key, value) tuples in the system
  - Can be tens or hundreds of billions of entries in the system!

- Solution: **consistent hashing**
- Associate to each node a unique *id* in an *uni*-dimensional space $0..2^m-1$
  - Partition this space across *M* machines
  - Assume keys are in same uni-dimensional space
  - Each (Key, Value) is stored at the node with the smallest ID larger than Key

Recap: Key to Node Mapping Example

- m = 8 → ID space: 0..63
- Node 8 maps keys [5,8]
- Node 15 maps keys [9,15]
- Node 20 maps keys [16, 20]
- …
- Node 4 maps keys [59, 4]



Scaling Up Directory

- With consistent hashing, directory contains only a number of entries equal to number of nodes
  - Much smaller than number of tuples

- Next challenge: every query still needs to contact the directory

Scaling Up Directory

- Given a **key**, find the **node** storing that key

- Key idea: route request from node to node until reaching the node storing the request's key

- Key advantage: totally distributed
  - No point of failure; no hot spot

Distributed Lookup (Directory) Service

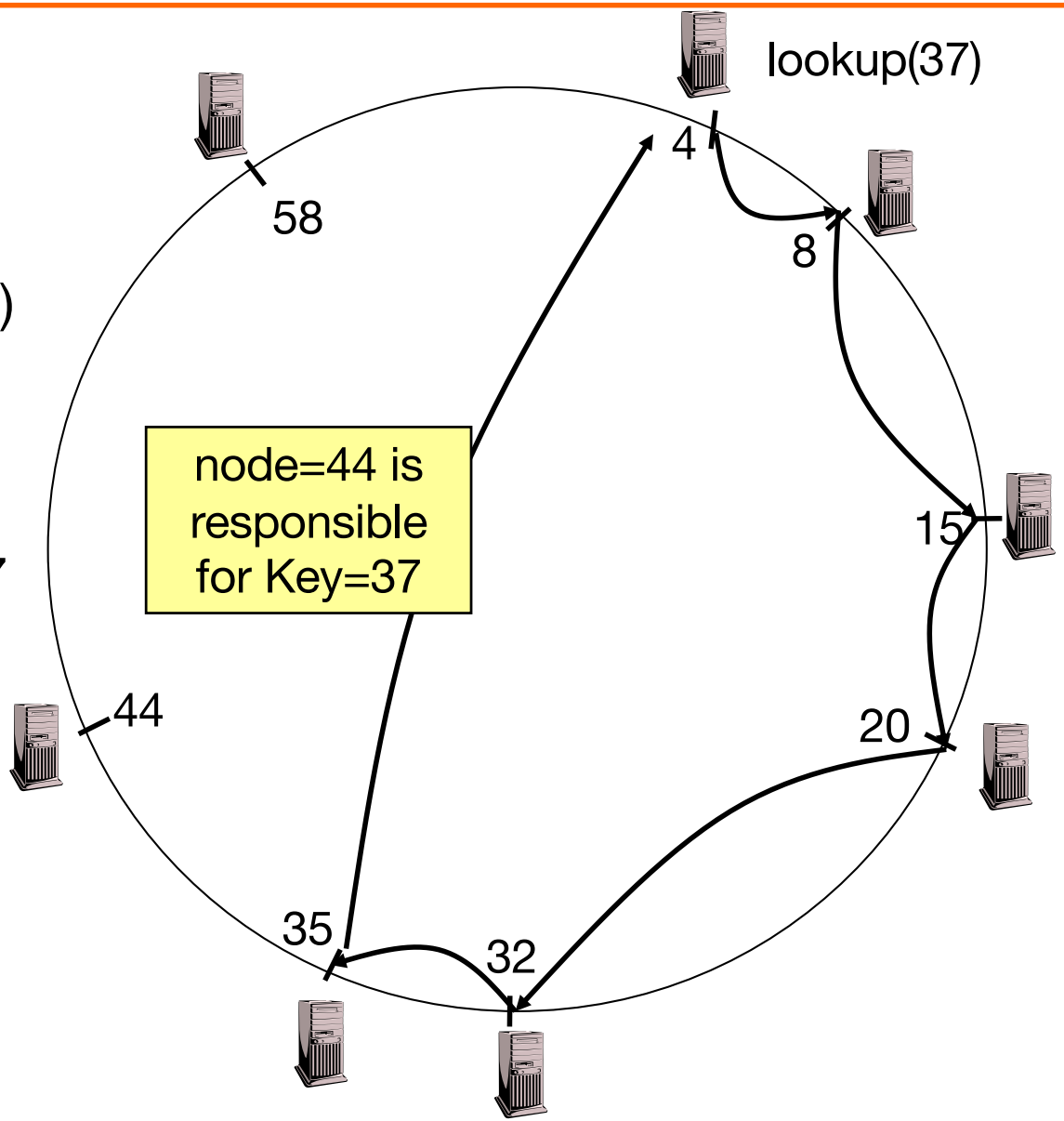
- Key design decision
  - Decouple correctness from efficiency

- Properties
  - Each node needs to know about O(log(*M*)), where *M* is the total number of nodes
  - Guarantees that a tuple is found in O(log(*M*)) steps

- Many other lookup services: Chord, Tapestry, Pastry, Kademlia, …

Lookup

- Each node maintains pointer to its successor
- Route packet (Key, Value) to the node responsible for ID using successor pointers
- E.g., node=4 lookups for node responsible for Key=37



Stabilization Procedure

- Periodic operation performed by each node n to maintain its successor when new nodes join the system

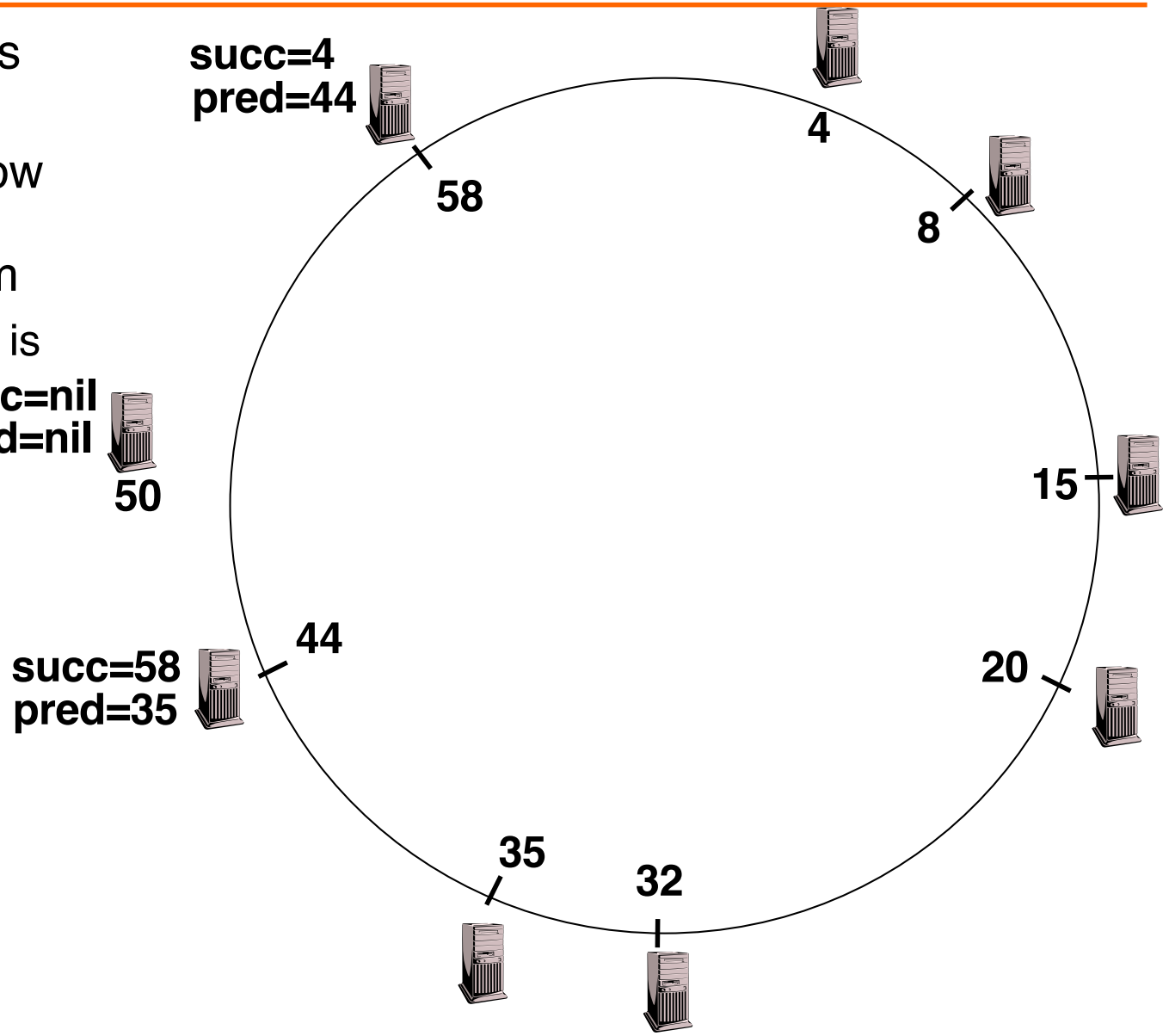
```
n.stabilize()
x = succ.pred;
if (x ∈ (n, succ))
    succ = x;      // if x better successor, update
succ.notify(n); // n tells successor about itself

n.notify(n')
if (pred = nil or n' ∈ (pred, n))
    pred = n';     // if n' is better predecessor, update
```

Joining Operation

Node with id=50 joins the ring

Node 50 needs to know at least one node already in the system

- Assume known node is 15

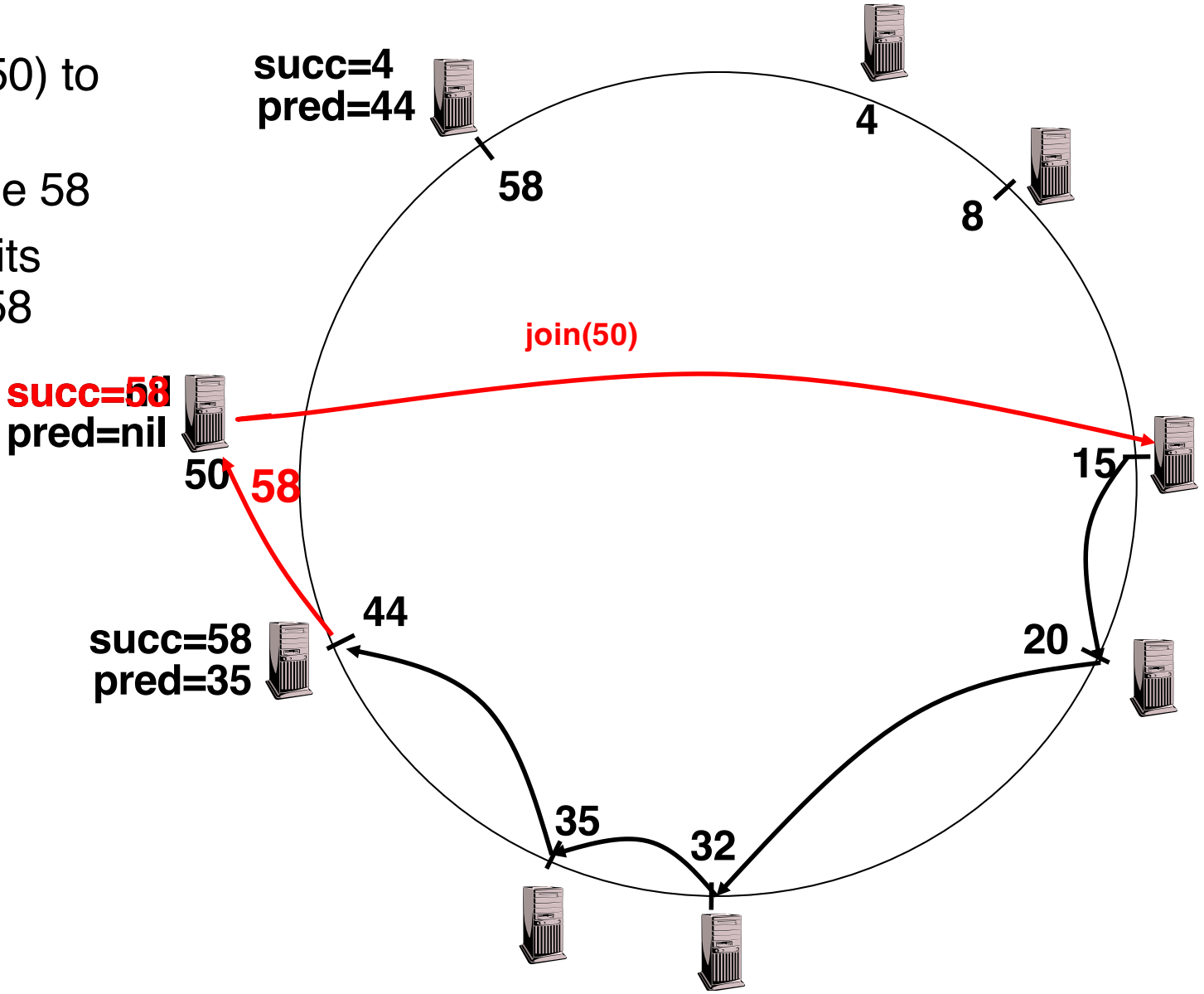


Joining Operation

n=50 sends join(50) to node 15

n=44 returns node 58

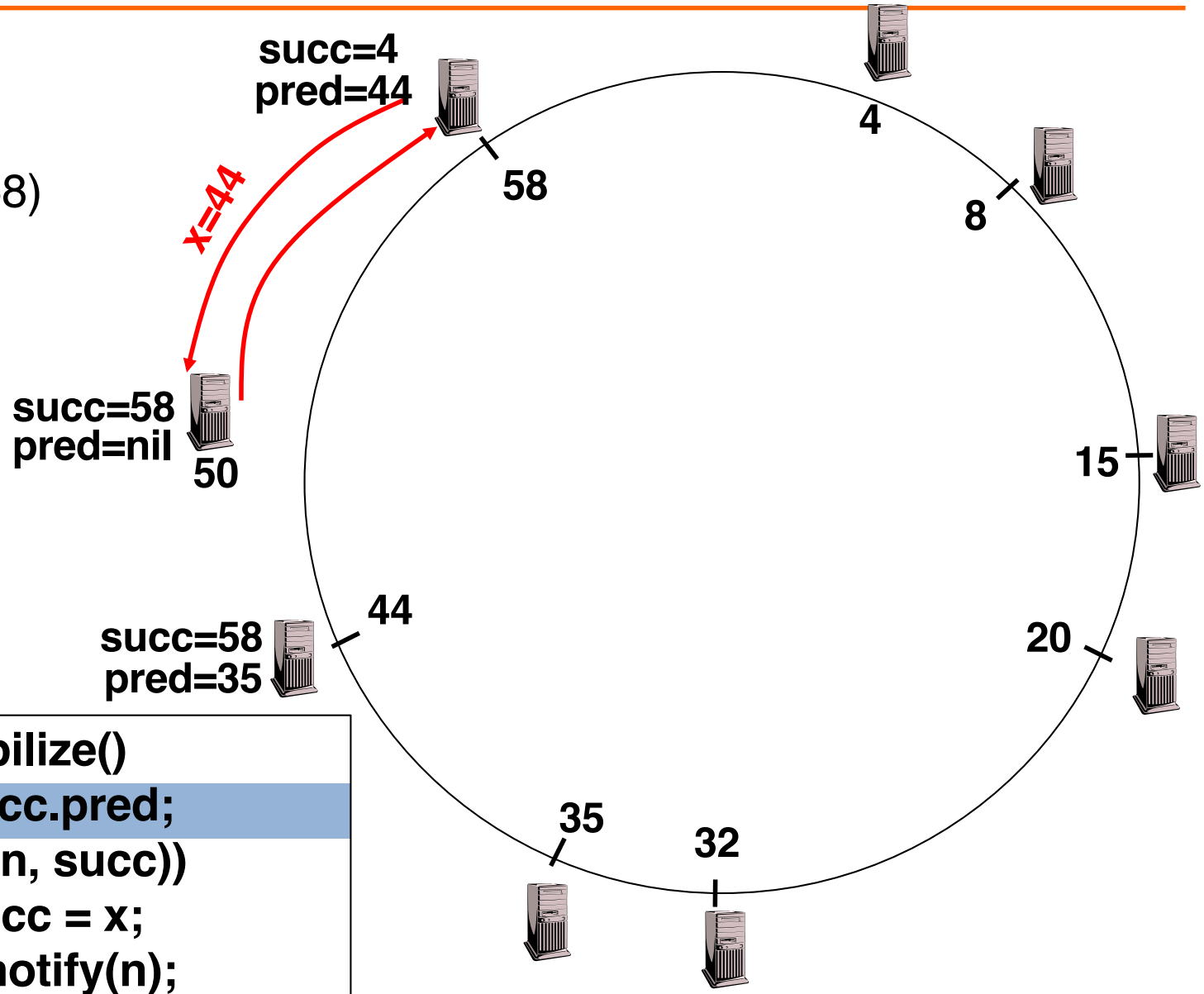
n=50 updates its successor to 58



Joining Operation

n=50 executes stabilize()

n's successor (58) returns x = 44



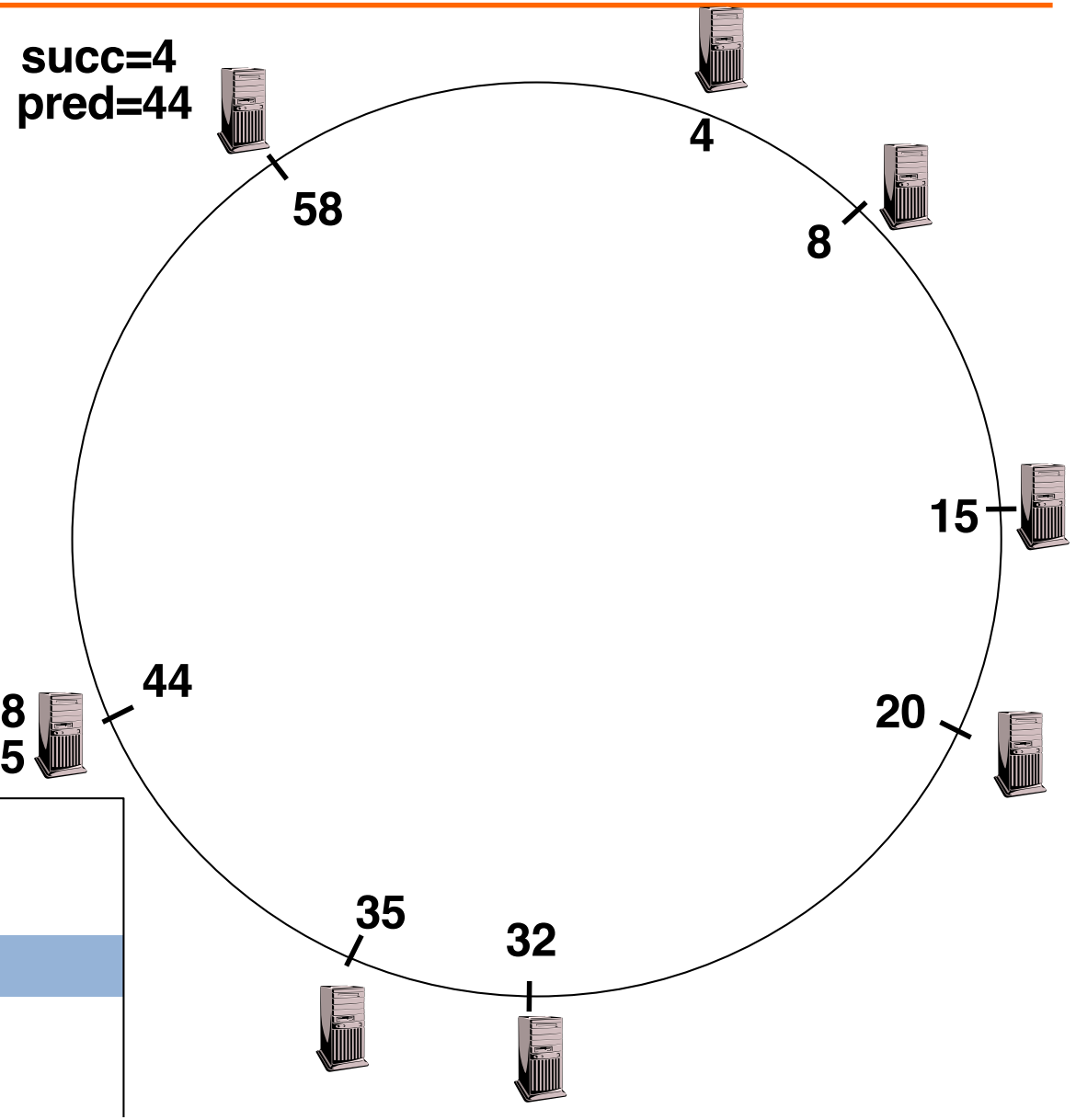

```
n.stabilize()
x = succ.pred;
if (x ∈(n, succ))
    succ = x;
succ.notify(n);
```

Joining Operation

n=50 executes stabilize()

x = 44

succ = 58




```
n.stabilize()
x = succ.pred;
if (x ∈(n, succ))
    succ = x;
succ.notify(n);
```

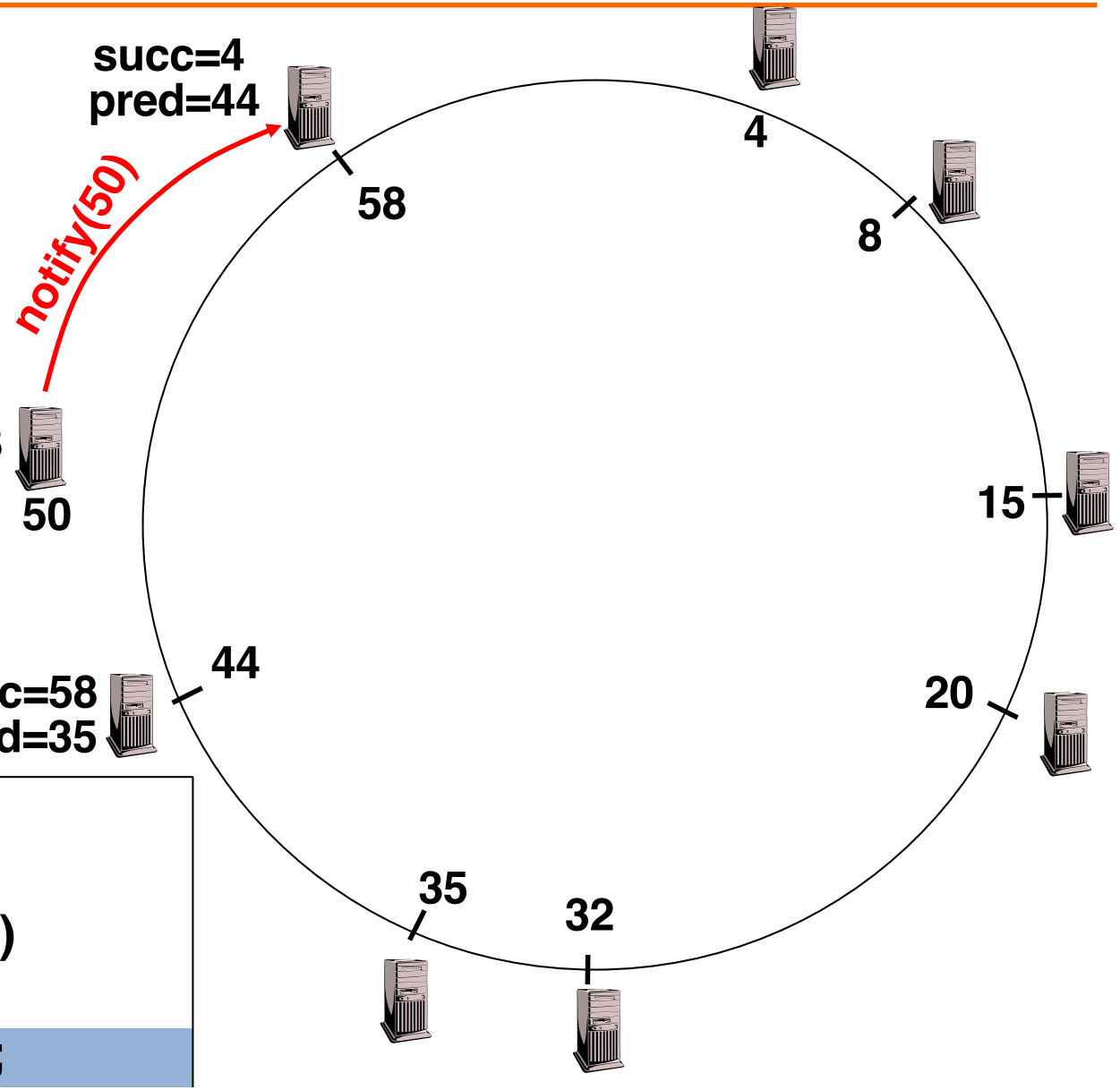
Joining Operation

n=50 executes stabilize()

x = 44

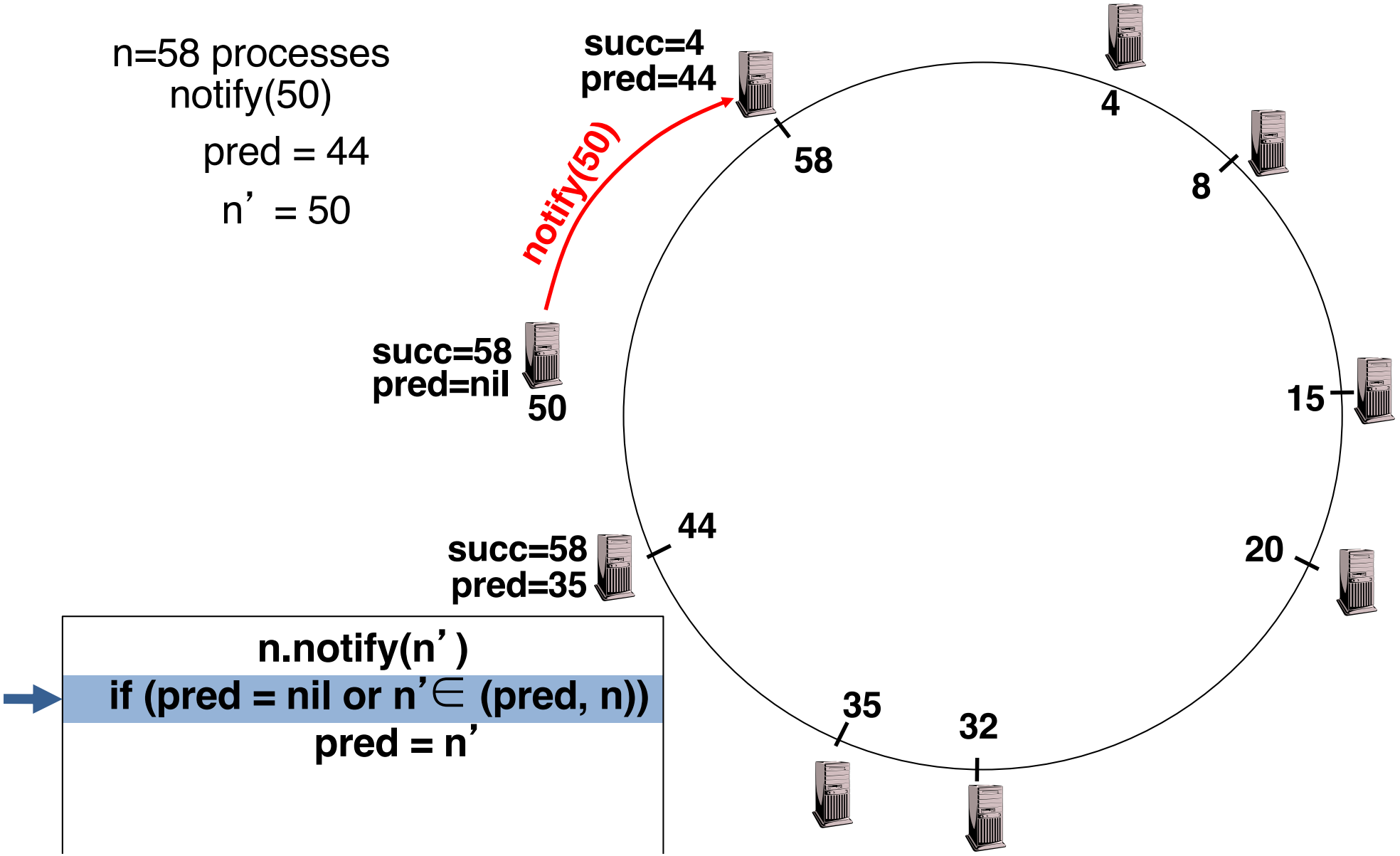
succ = 58

n=50 sends to it's successor (58) notify(50)




```
n.stabilize()
x = succ.pred;
if (x ∈(n, succ))
    succ = x;
succ.notify(n);
```

Joining Operation



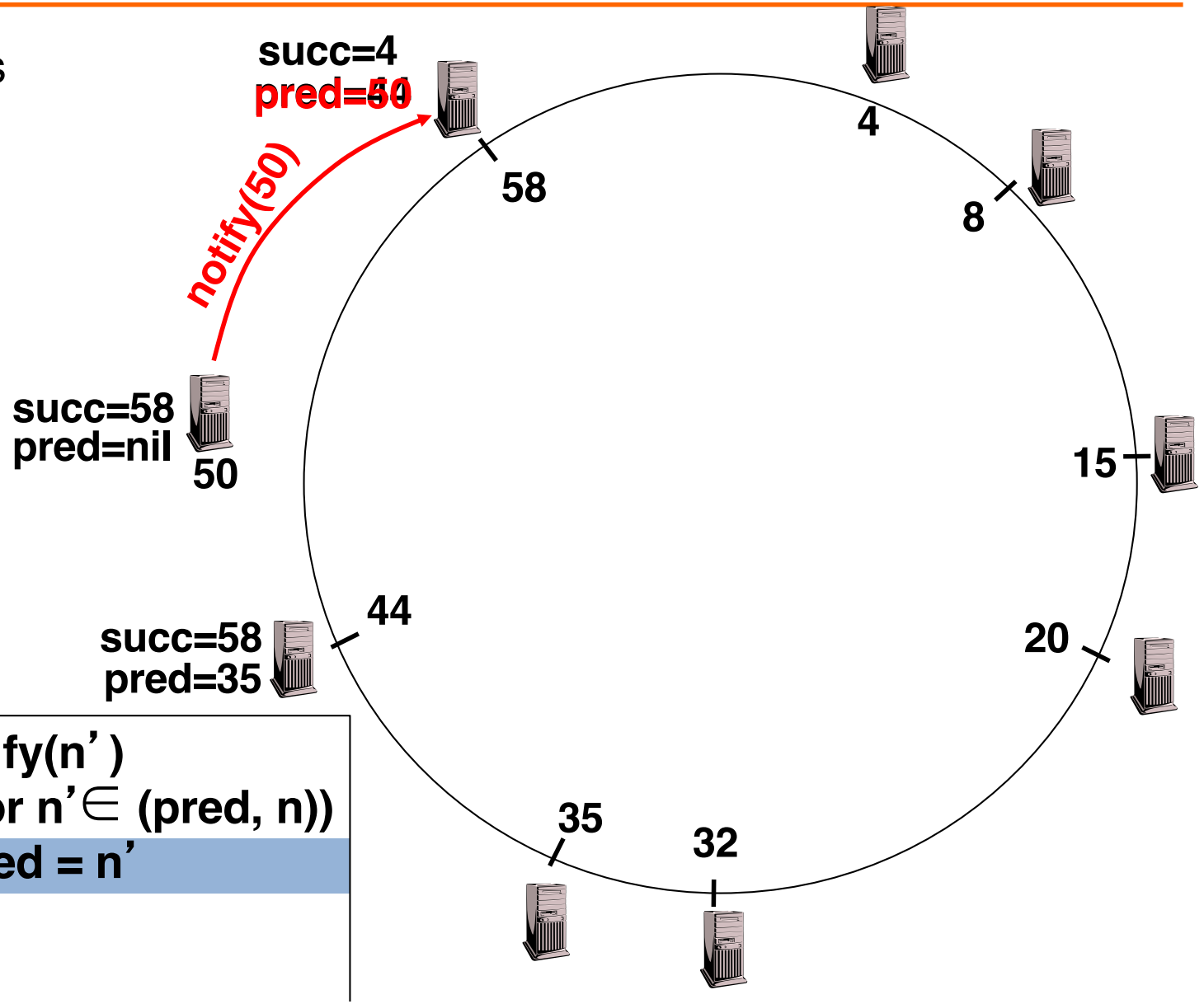
Joining Operation

n=58 processes
notify(50)

pred = 44

n' = 50

set pred = 50

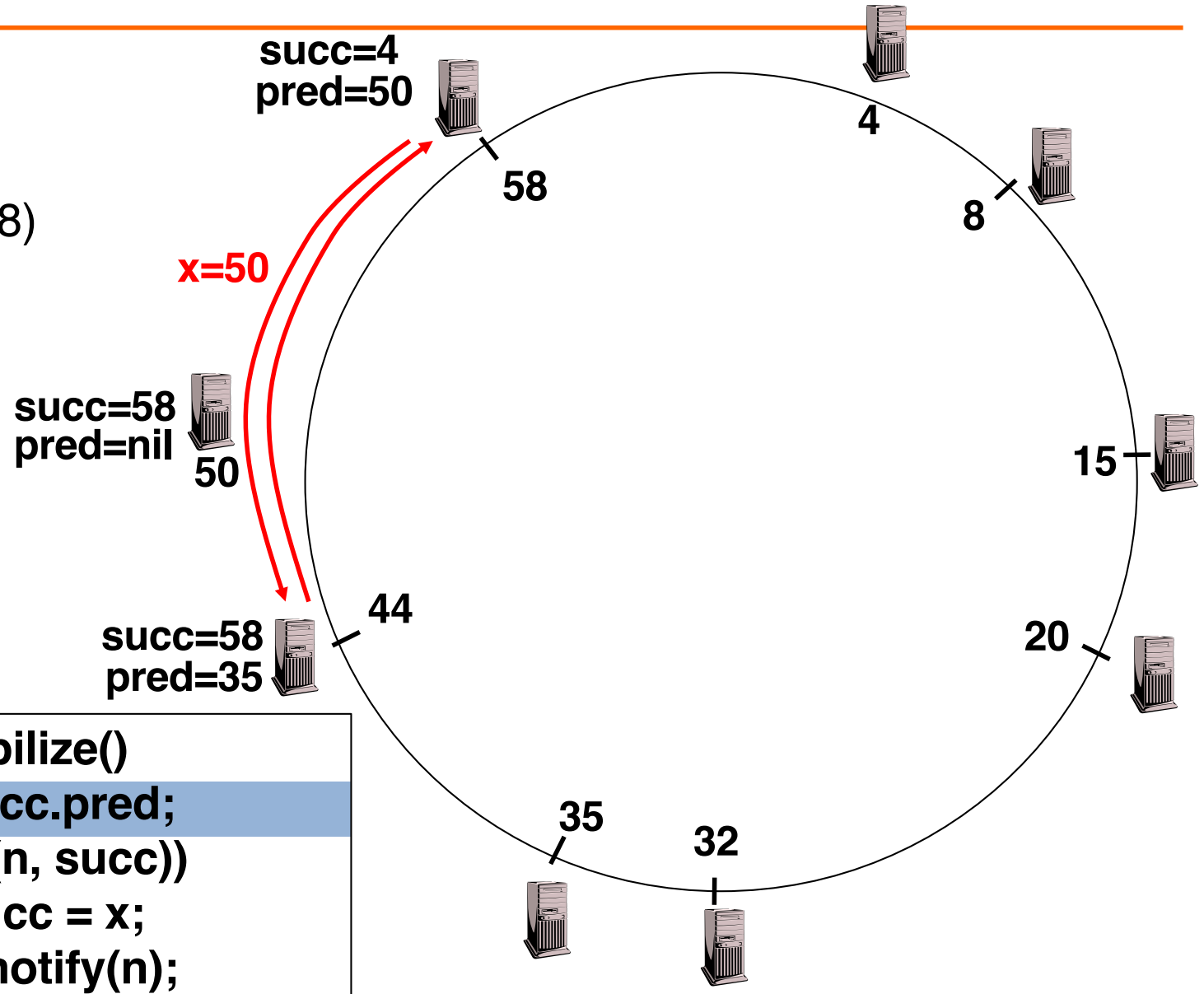



```
n.notify(n')
if (pred = nil or n' ∈ (pred, n))
    pred = n'
```

Joining Operation

n=44 runs stabilize()

n's successor (58) returns x = 50



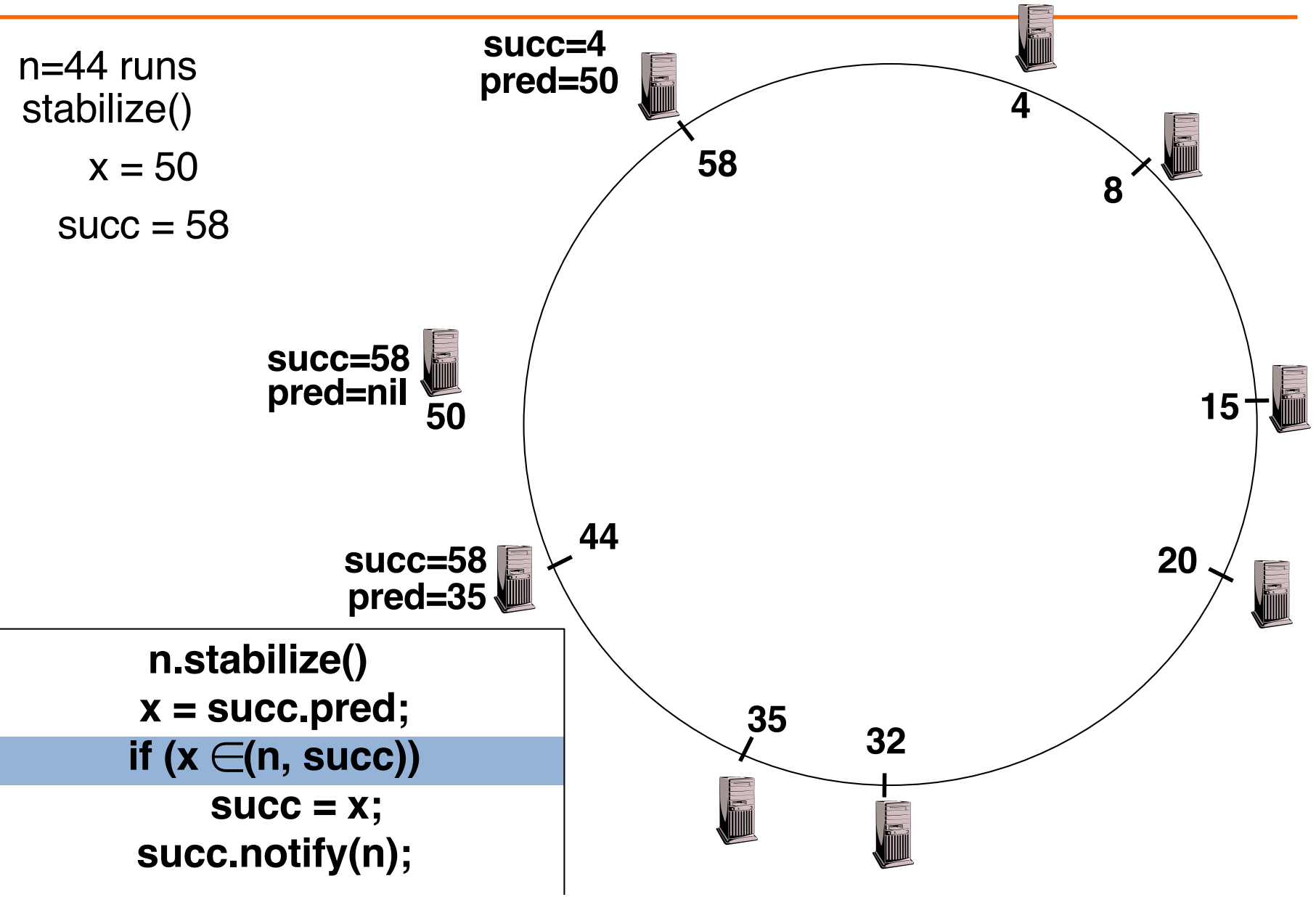

```
n.stabilize()
x = succ.pred;
if (x ∈ (n, succ))
    succ = x;
succ.notify(n);
```

Joining Operation

n=44 runs stabilize()

x = 50

succ = 58




```
n.stabilize()
x = succ.pred;
if (x ∈(n, succ))
    succ = x;
succ.notify(n);
```

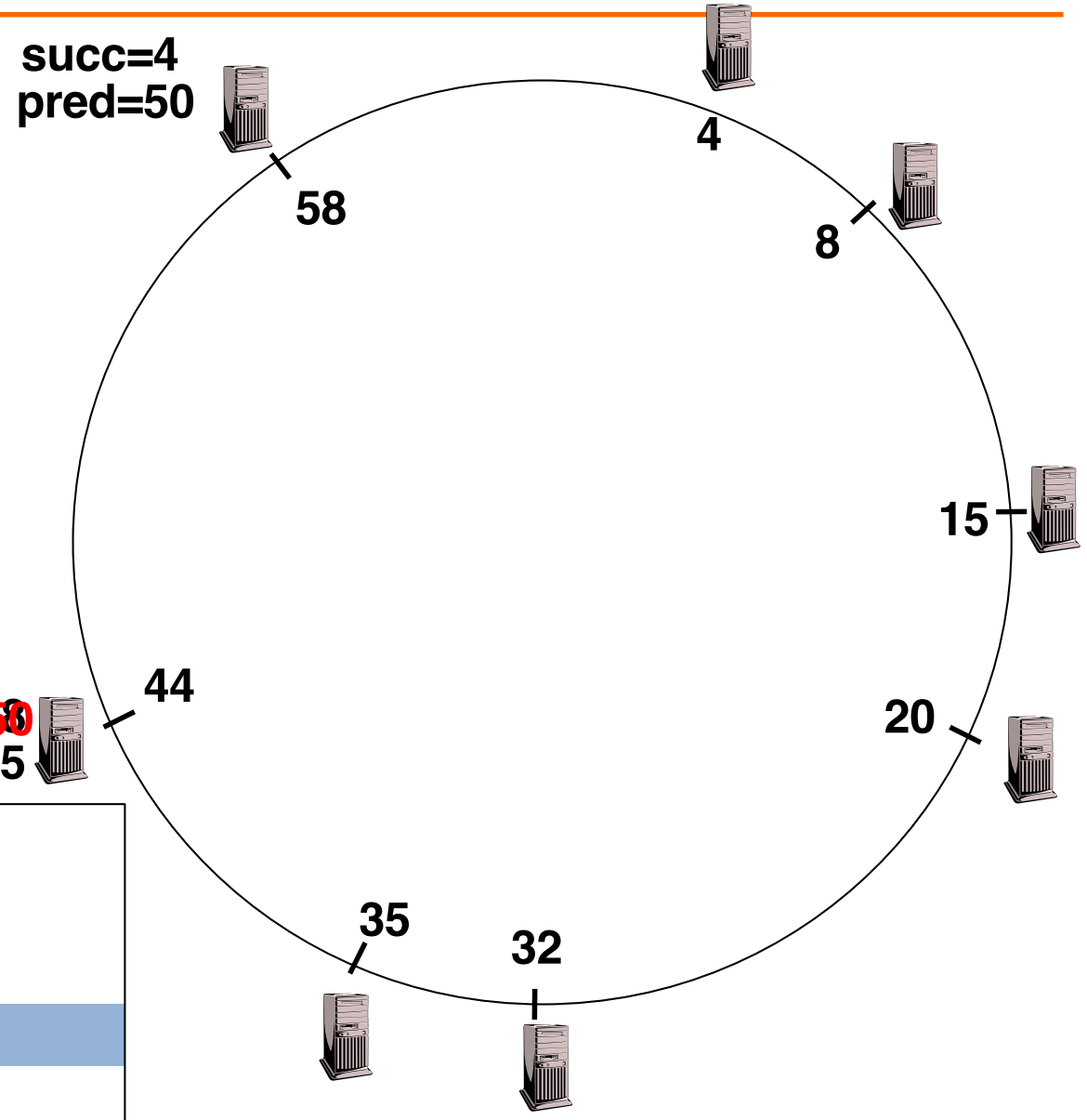
Joining Operation

n=44 runs stabilize()

x = 50

succ = 58

n=44 sets succ=50

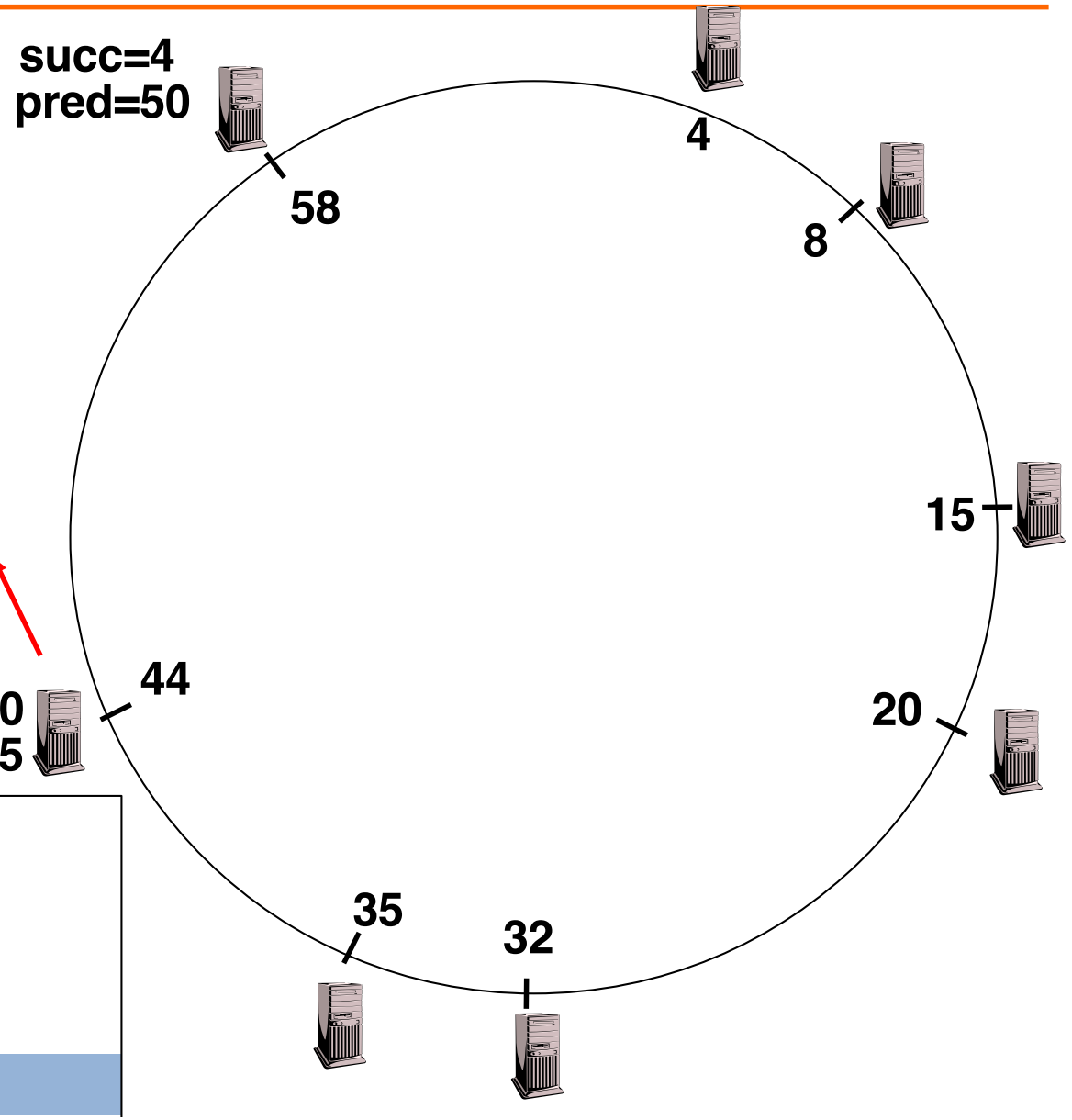



```
n.stabilize()
x = succ.pred;
if (x ∈(n, succ))
    succ = x;
succ.notify(n);
```

Joining Operation

n=44 runs stabilize()

n=44 sends notify(44) to its successor

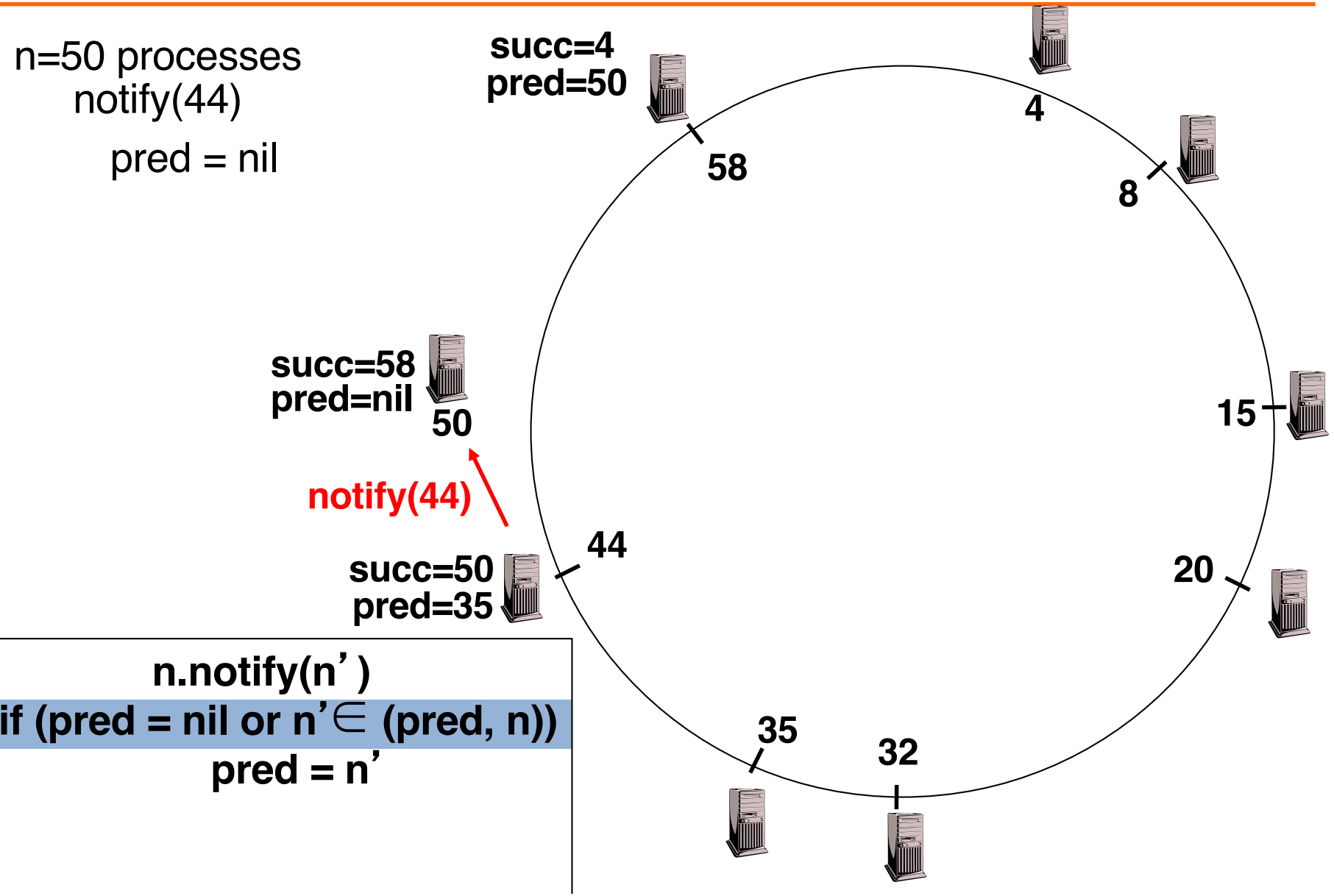



```
n.stabilize()
x = succ.pred;
if (x ∈(n, succ))
    succ = x;
succ.notify(n);
```

Joining Operation

n=50 processes
notify(44)

pred = nil

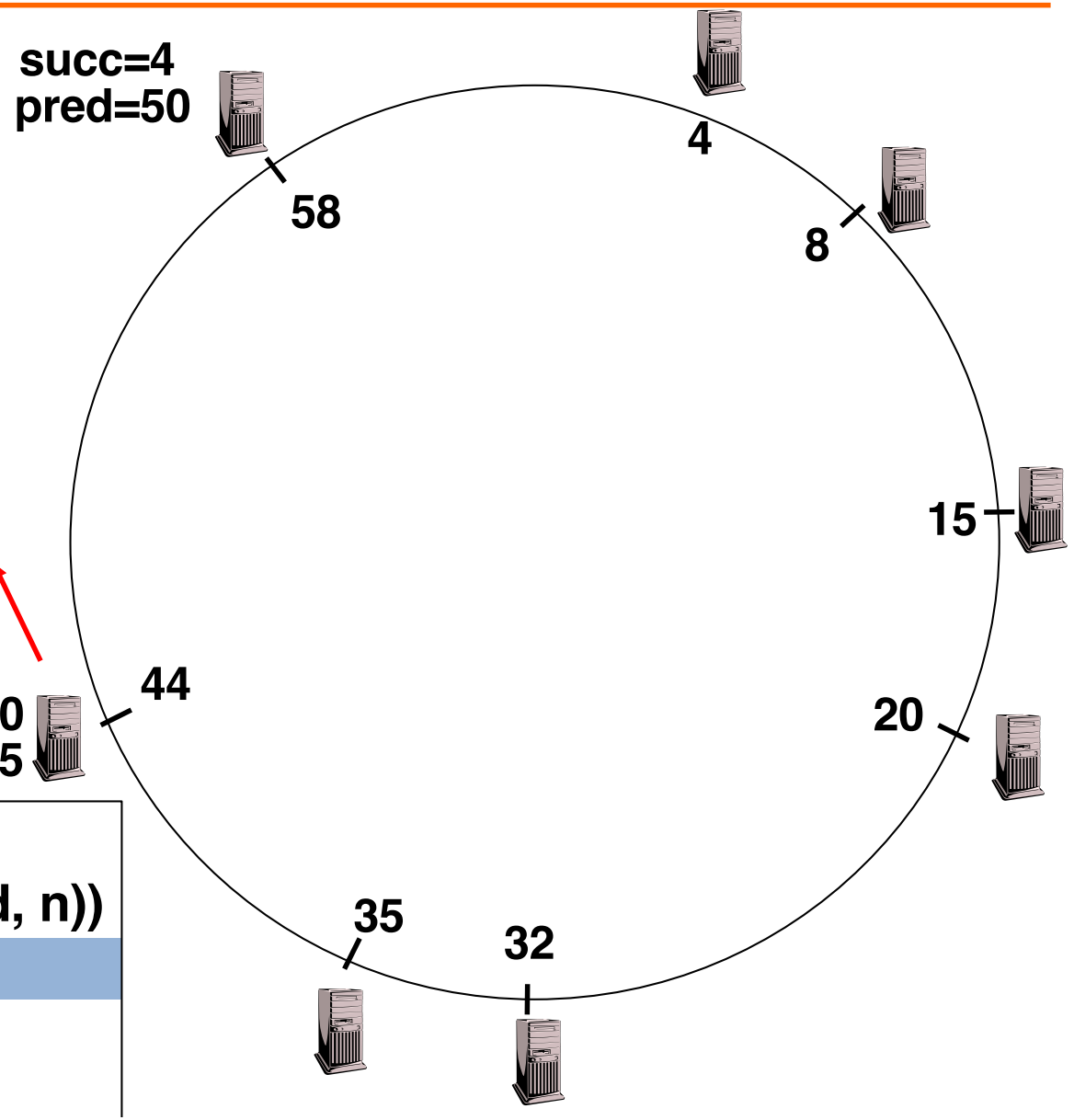


Joining Operation

n=50 processes
notify(44)

pred = nil

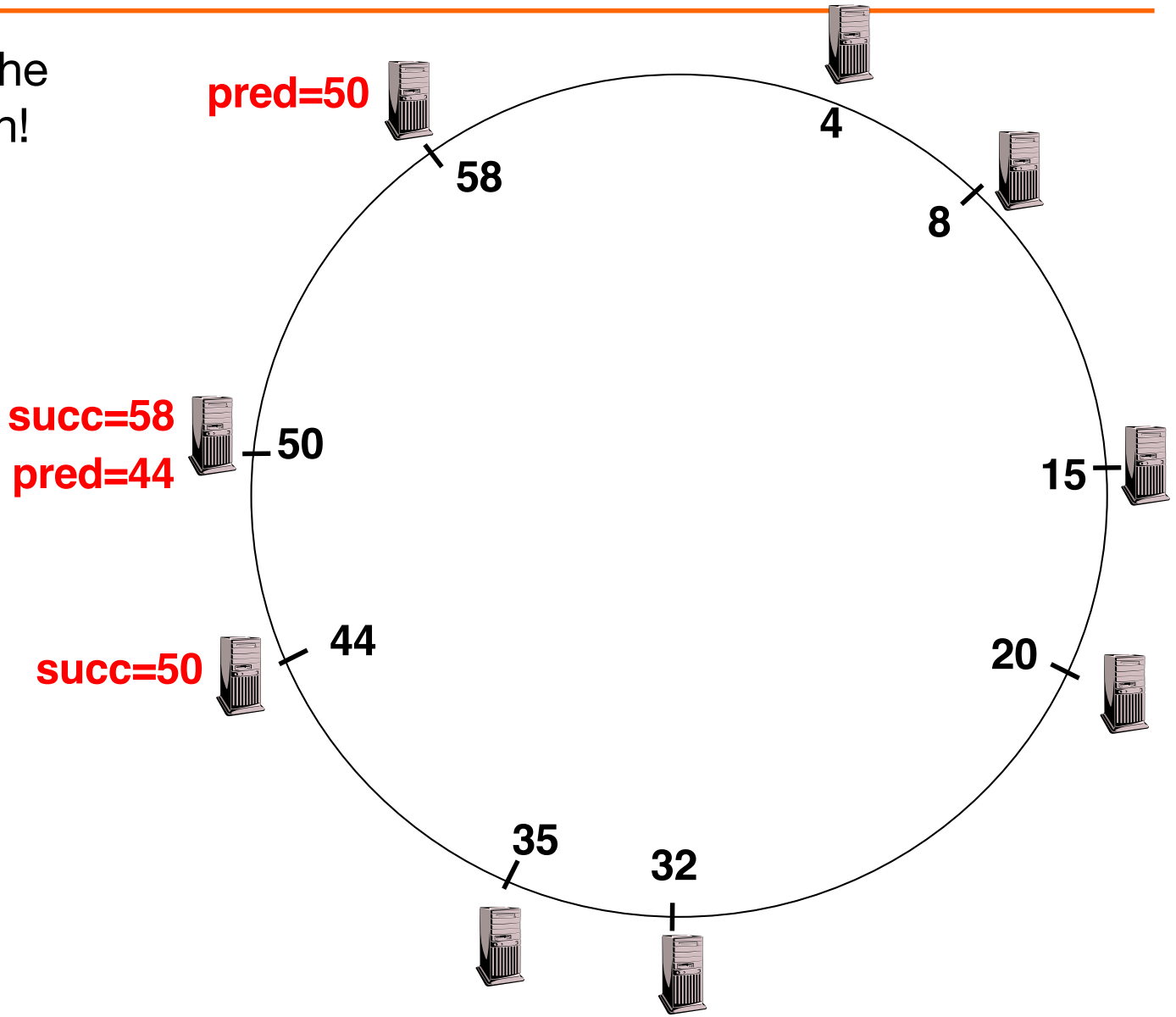
n=50 sets pred=44




```
n.notify(n')
if (pred = nil or n'∈ (pred, n))
    pred = n'
```

Joining Operation (cont' d)

This completes the joining operation!



Achieving Fault Tolerance for Lookup Service

- To improve robustness each node maintains the k (> 1) immediate successors instead of only one successor

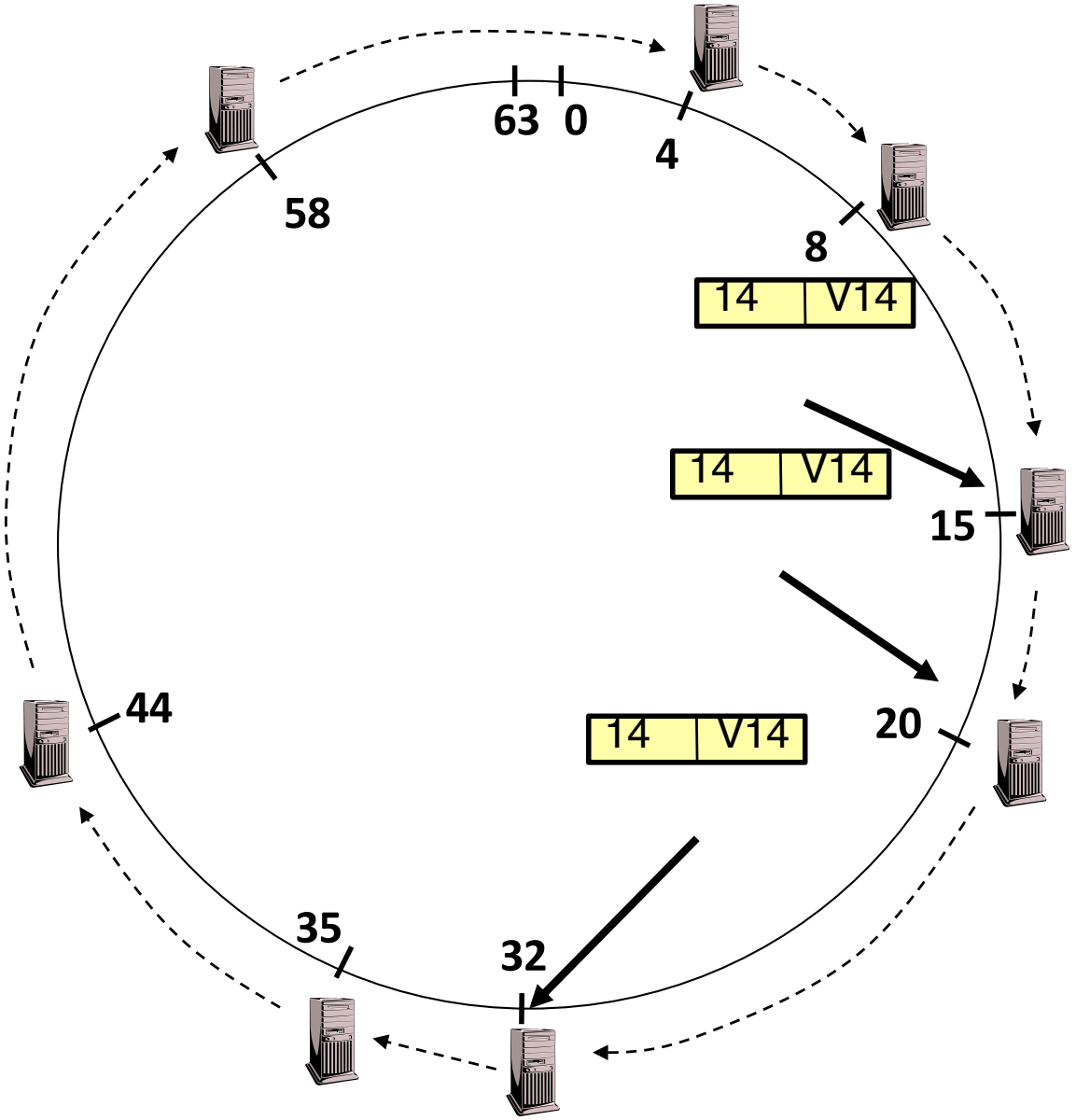
- In the pred() reply message, node A can send its k-1 successors to its predecessor B

- Upon receiving pred() message, B can update its successor list by concatenating the successor list received from A with its own list

- If k = log(M), lookup operation works with high probability even if half of nodes fail, where M is number of nodes in the system

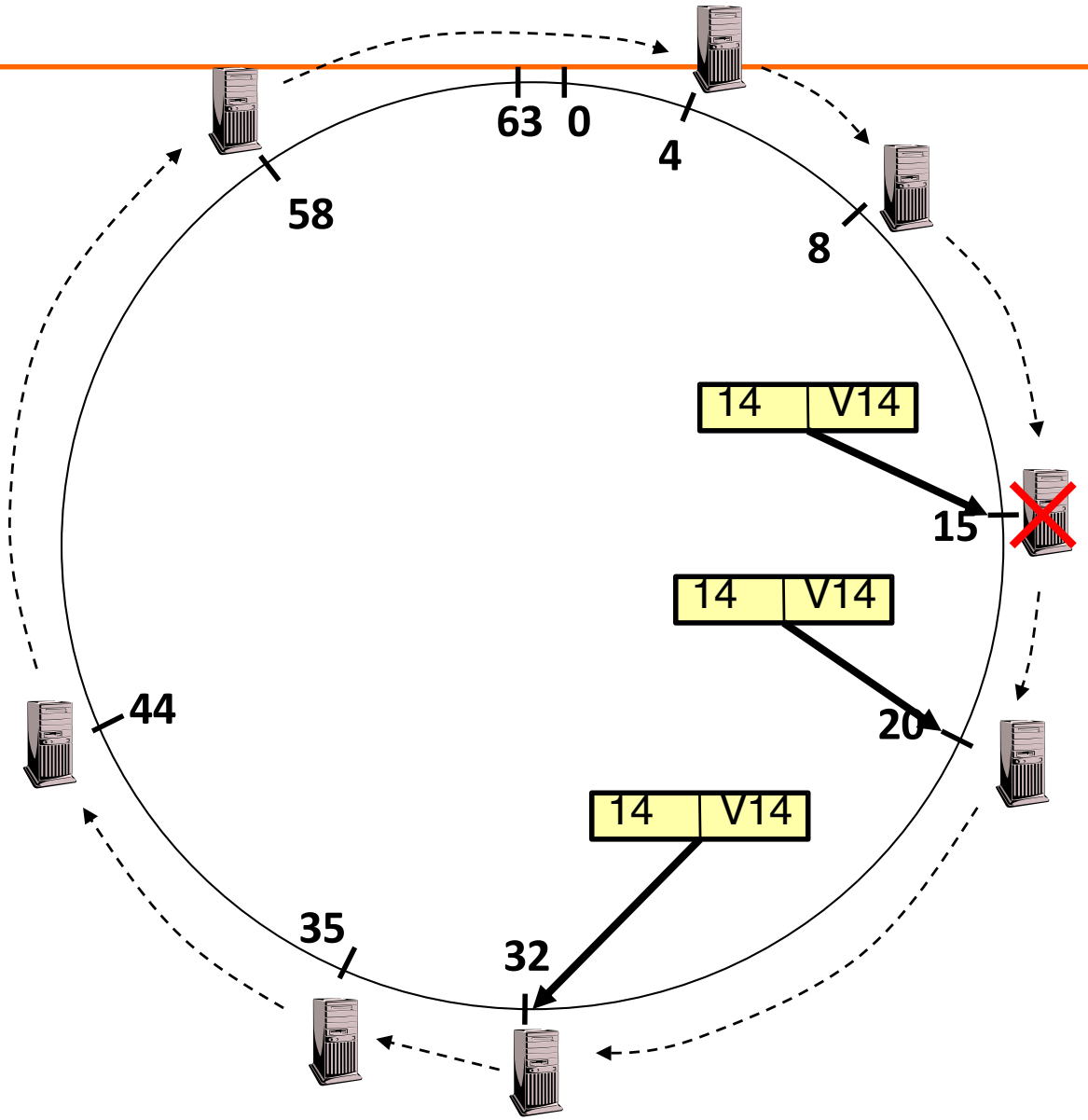
Storage Fault Tolerance

- Replicate tuples on successor nodes
- Example: replicate (K14, V14) on nodes 20 and 32



Storage Fault Tolerance

- If node 15 fails, no reconfiguration needed
  - Still have two replicas
  - All lookups will be correctly routed
- Will need to add a new replica on node 35



Today

1. Techniques for partitioning data

2. **Case study: the Amazon Dynamo key-value store**

Amazon Dynamo

Motivation

- Build a distributed storage system:
  - Scale
  - Symmetry: every node should have same functionality
  - Simple: key-value
  - Highly available
  - Heterogeneity: allow adding nodes with different capacities
  - Guarantee Service Level Agreements (SLA)

System Assumptions and Requirements

- Central challenge: low-latency key lookup with high availability
  - Trades off **consistency** for **availability** and **latency**

- **Techniques:**
  - **Consistent hashing** to map keys to nodes
  - **Vector clocks** for conflict resolution
  - **Gossip** for node membership
  - **Replication** at successors for availability under failure

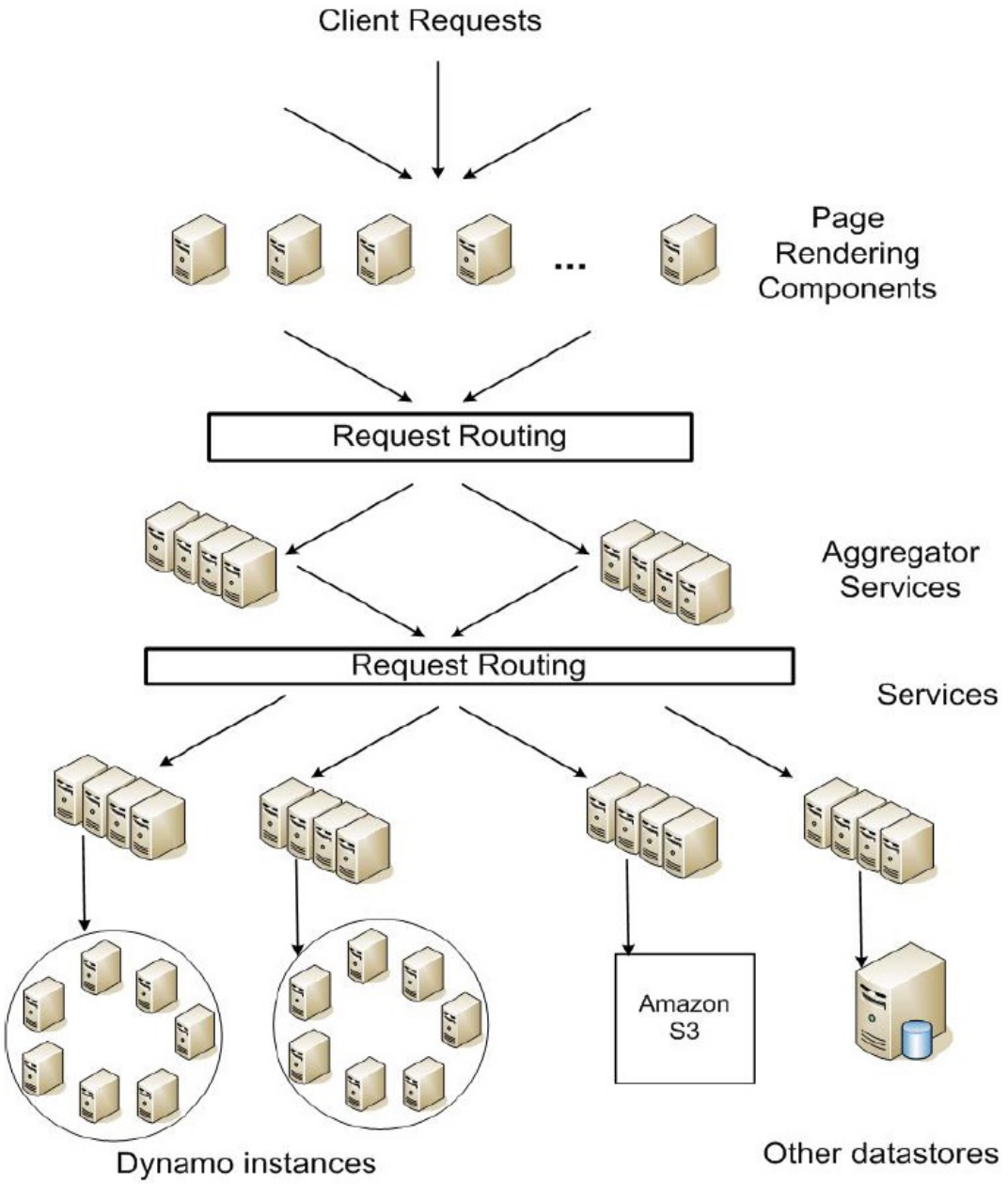
- *SLA* (Service Level Agreement)*:* 99.9% performance guarantees
  - E.g., 500ms latency for 99.9% of its requests for a peak client load of 500 requests per second
  - average, median, variance not representative for user's experience

•Other Assumptions: internal service, no security related requirements

Architecture

- Service oriented architecture: modular, composable

- Challenge: end-to-end SLAs
  - Each service should provide even tighter latency bounds



Design Consideration

- Sacrifice strong consistency for availability

- Conflict resolution is executed during ***read*** instead of ***write***, i.e. "always writeable".

- Other principles:
  - Incremental scalability
  - Symmetry
  - Decentralization
  - Heterogeneity

Amazon's workload (in 2007)

- **Tens of thousands** of servers in globally-distributed **data centers**
- **Peak load:** Tens of millions of customers
- **Tiered** service-oriented architecture
  - **Stateless** web page rendering servers, atop
  - **Stateless** aggregator servers, atop
  - **Stateful** data stores (**e.g. Dynamo**)
    - **put( ), get( ):** values "usually less than 1 MB"

How does Amazon use Dynamo?

- **Shopping cart**
- **Session info**
  - Maybe “recently visited products” *et c.*?
- **Product list**
  - Mostly read-only, replication for high read throughput

Dynamo requirements

- **Highly available writes** despite failures
  - Despite disks failing, network routes flapping, "data centers destroyed by tornadoes"
  - Always respond quickly, even during failures → replication
- **Low request-response latency:** focus on **99.9%** SLA
- **Incrementally scalable** as servers grow to workload
  - Adding "nodes" should be seamless
- Comprehensible **conflict resolution**
  - High availability in above sense implies conflicts

Design questions

- How is data **placed and replicated?**
- How are **requests routed and handled** in a replicated system?
- How to cope with temporary and permanent **node failures?**

Dynamo's system interface

- Basic interface is a key-value store
  - **get(k)** and **put(k, v)**
  - Keys and values opaque to Dynamo

- get(key) → value, **context**
  - Returns one value or multiple conflicting values
  - Context describes version(s) of value(s)

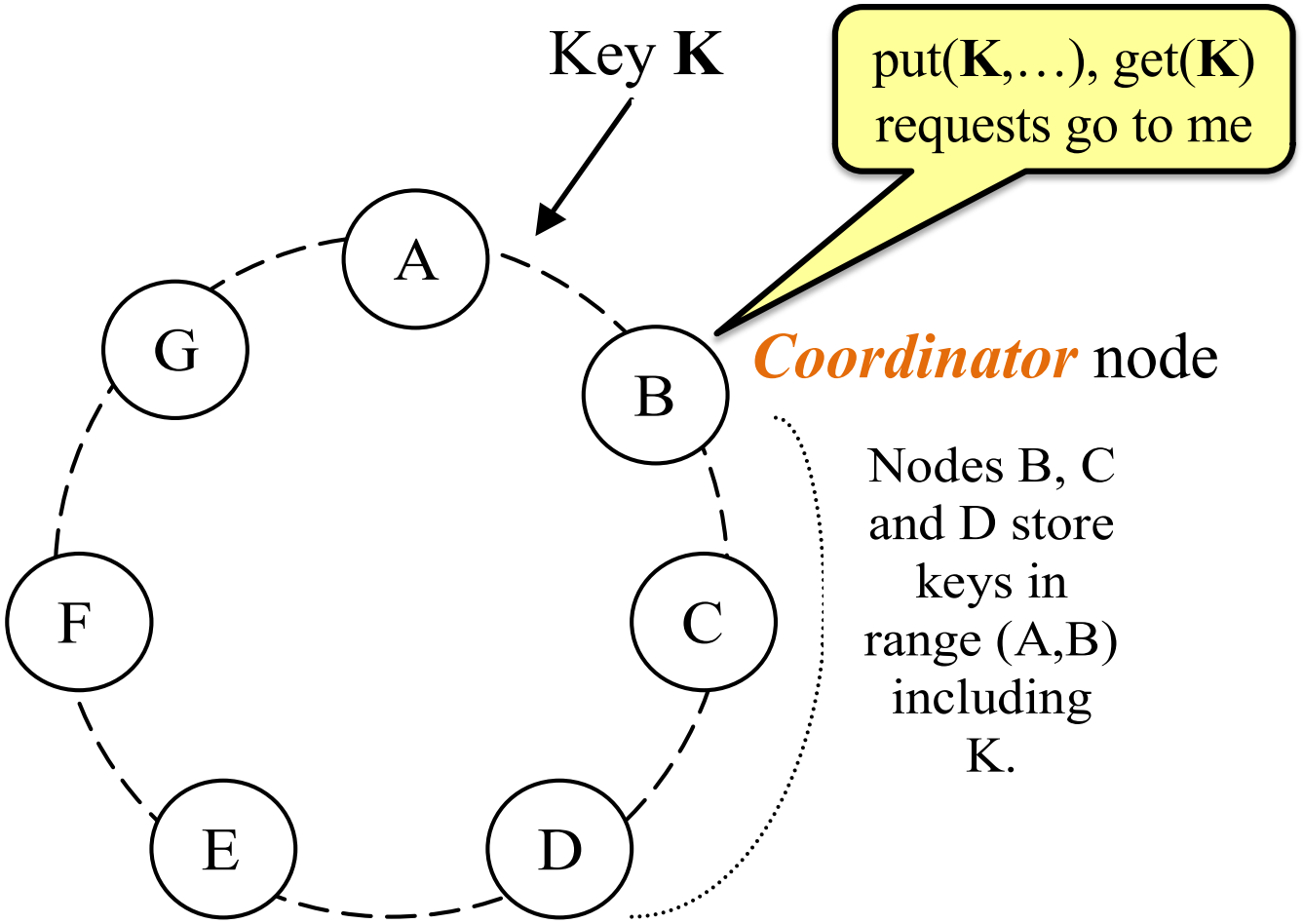
- put(key, **context**, value) → "OK"
  - **Context** indicates **which versions** this version supersedes or merges

Dynamo's techniques

- **Place** replicated data on nodes with **consistent hashing**
- Maintain consistency of replicated data with **vector clocks**
  - **Eventual consistency** for replicated data: prioritize success and low latency of writes over reads
    - And availability over consistency (unlike DBs)
- Efficiently **synchronize replicas** using **Merkle trees**

Key trade-offs: Response time vs. consistency vs. durability

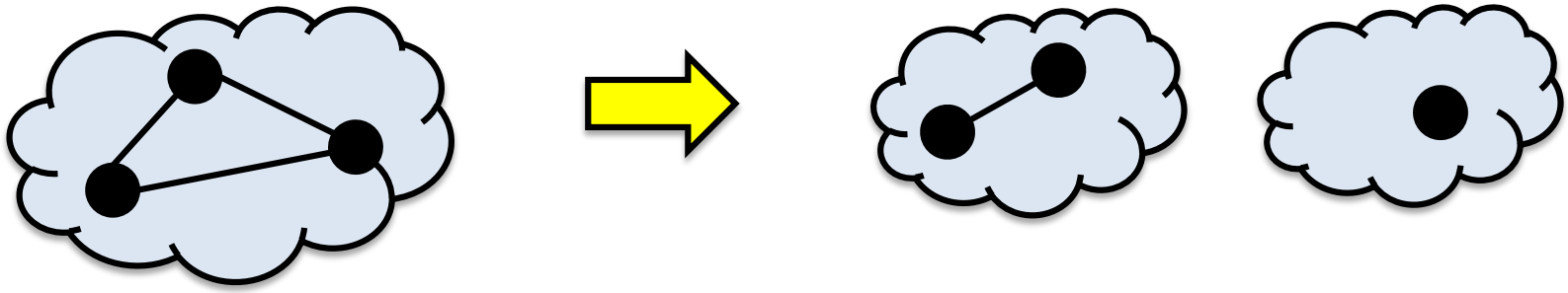
Data placement



Each data item is **replicated** at ***N*** virtual nodes (e.g., $N = 3$)

Partitions force a choice between availability and consistency

- Suppose **three** replicas are partitioned into **two and one**

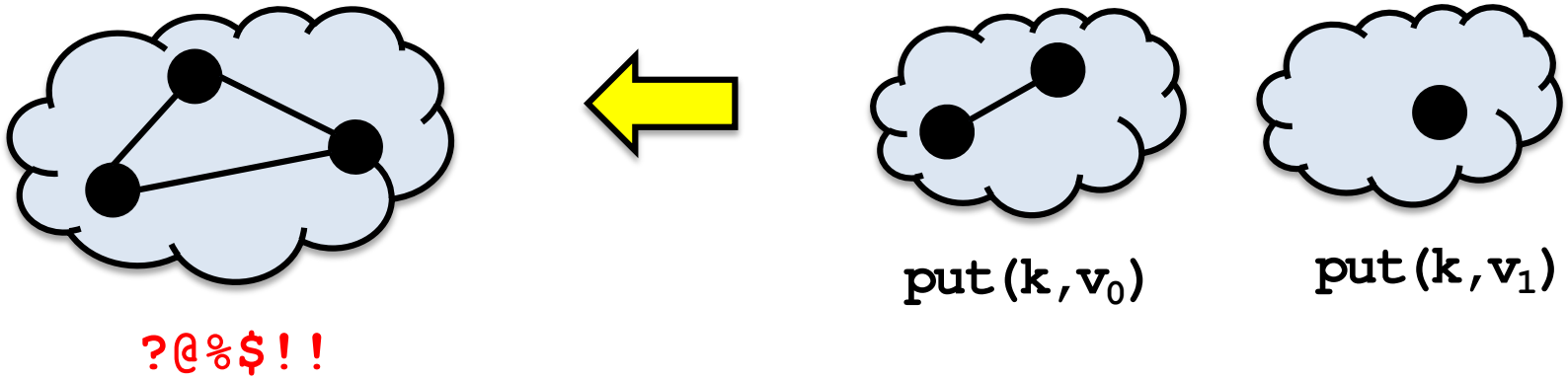


- If one replica fixed as master, no client in other partition can write

- Traditional distributed databases emphasize consistency over availability when there are partitions

Alternative: Eventual consistency

- Dynamo emphasizes **availability over consistency** when there are partitions
- Tell client write complete when only some replicas have stored it
- Propagate to other replicas in background
- **Allows writes in both partitions**…but risks:
  - Returning **stale data**
  - **Write conflicts** when partition heals:



Dynamo: Take-away ideas

- Consistent hashing broadly useful for replication—not only in P2P systems
- Extreme emphasis on **availability and low latency,** unusually, at the **cost of some inconsistency**
- Eventual consistency lets writes and reads return quickly, **even when partitions and failures**

Today's Papers

- Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web,
  - David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, Rina Panigrahy
  - https://www.cs.princeton.edu/courses/archive/fall09/cos518/papers/chash.pdf

- Dynamo: Amazon's Highly Available Key-value Store,
  - Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan, Sivasubramanian, Peter Vosshall, and Werner Vogels, SOSP'07
  - www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf