CSCI 381/780
Cloud Computing

Key-value Store

Jun Li
Queens College

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, …)

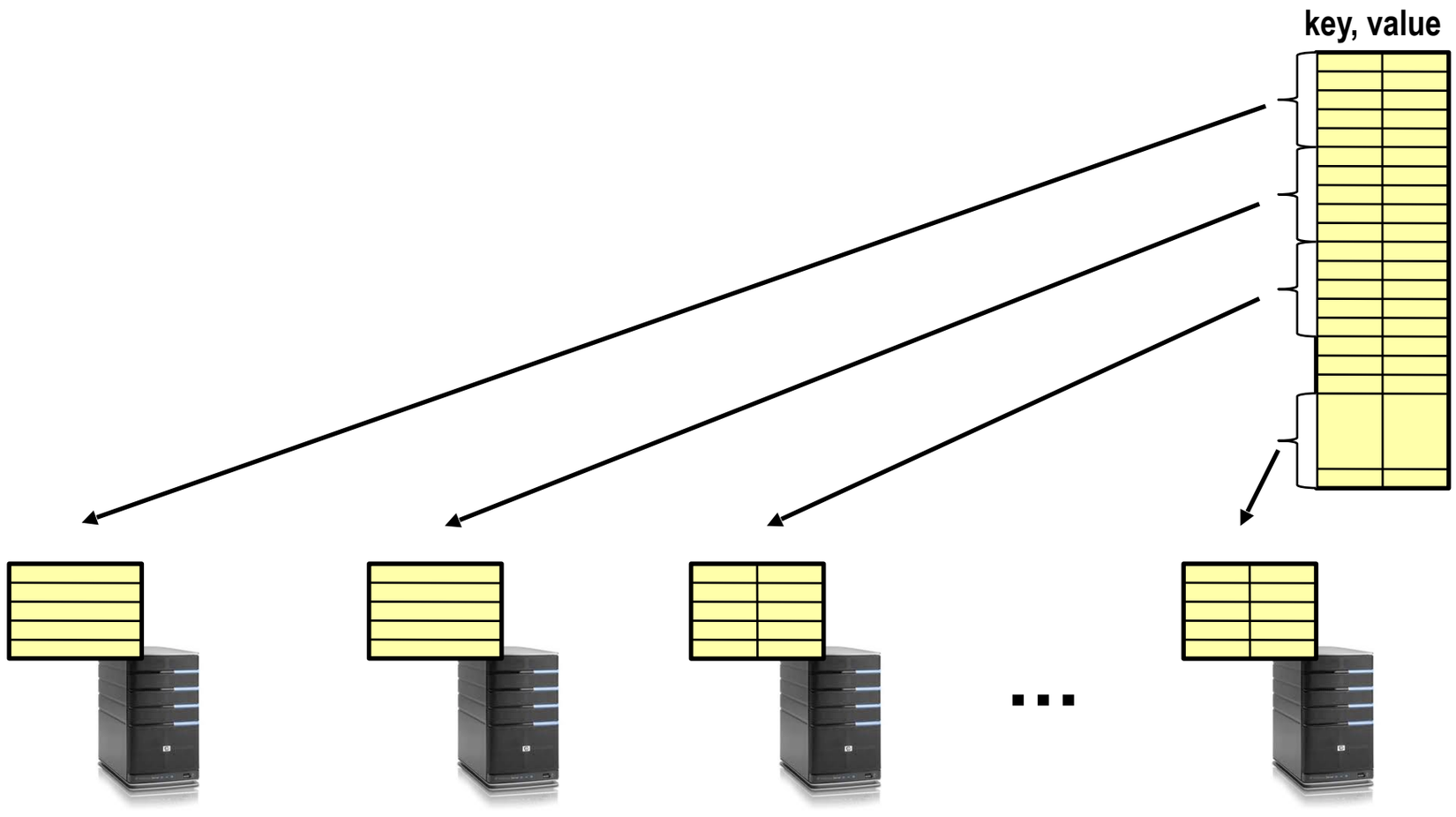
Distributed file systems

- Key: Block ID
- Value: Block

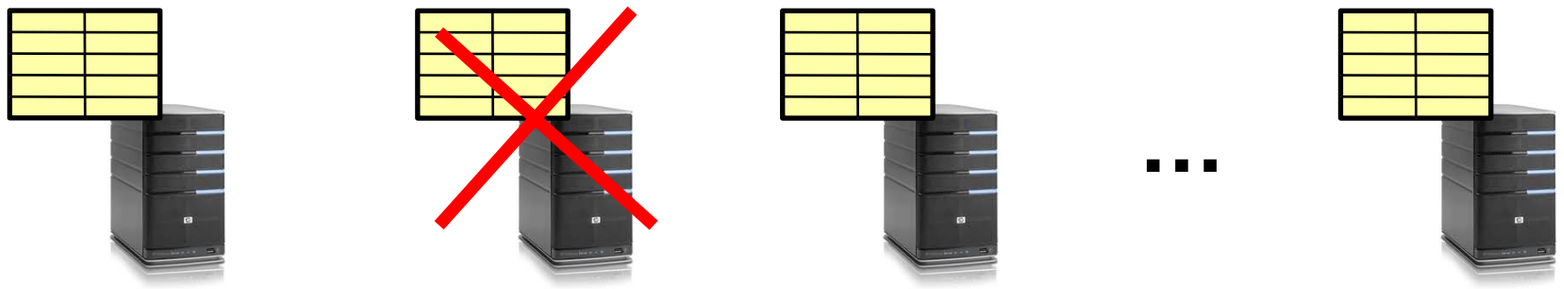
Key Value Store

Also called a Distributed Hash Table (DHT)

Main idea: partition set of key-values across many machines



Challenges



Scalability:

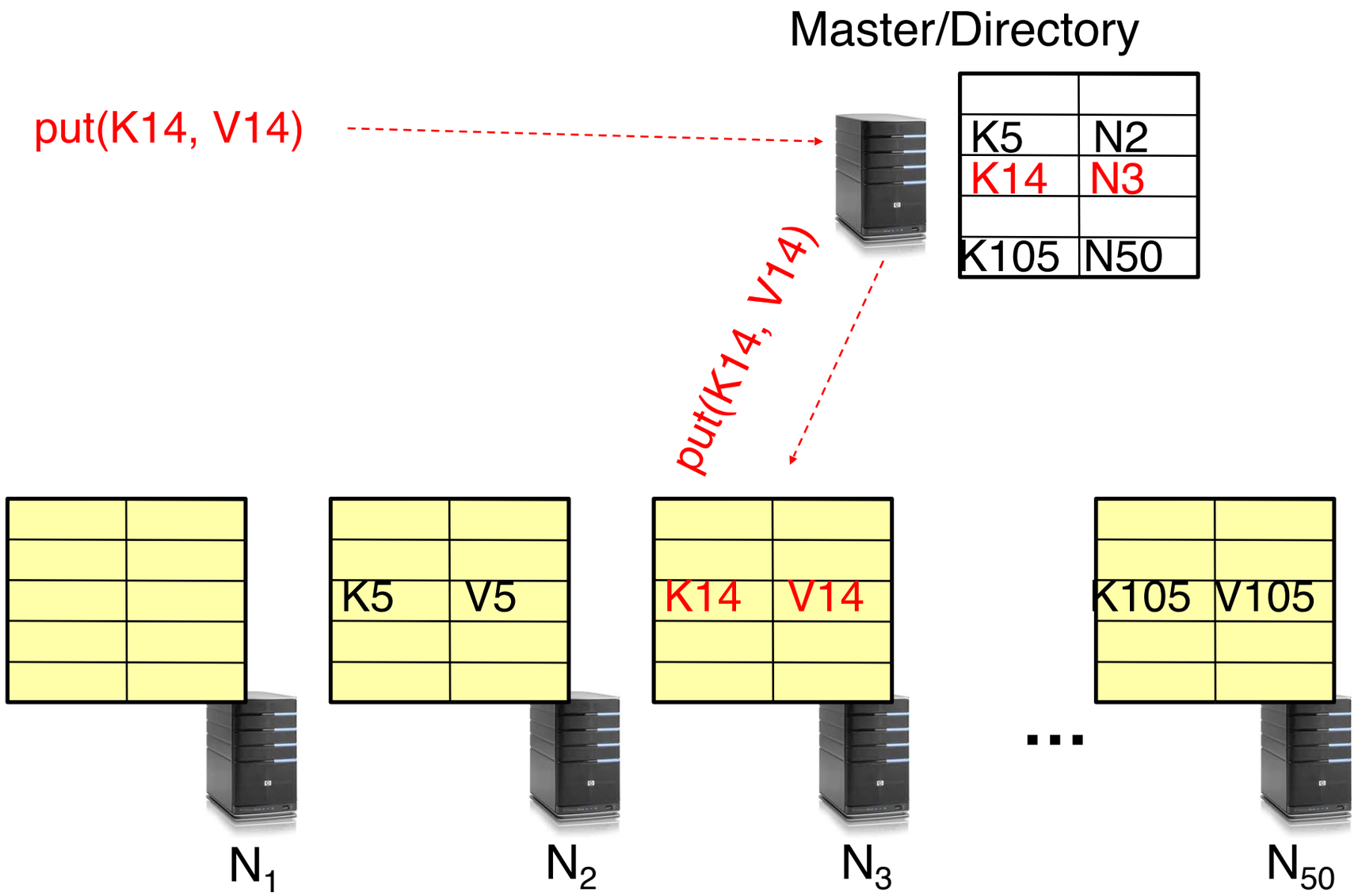
- Need to scale to thousands of machines
- Need to allow easy addition of new machines

Fault Tolerance: handle machine failures without losing data and without degradation in performance

Consistency: maintain data consistency in face of node failures and message losses

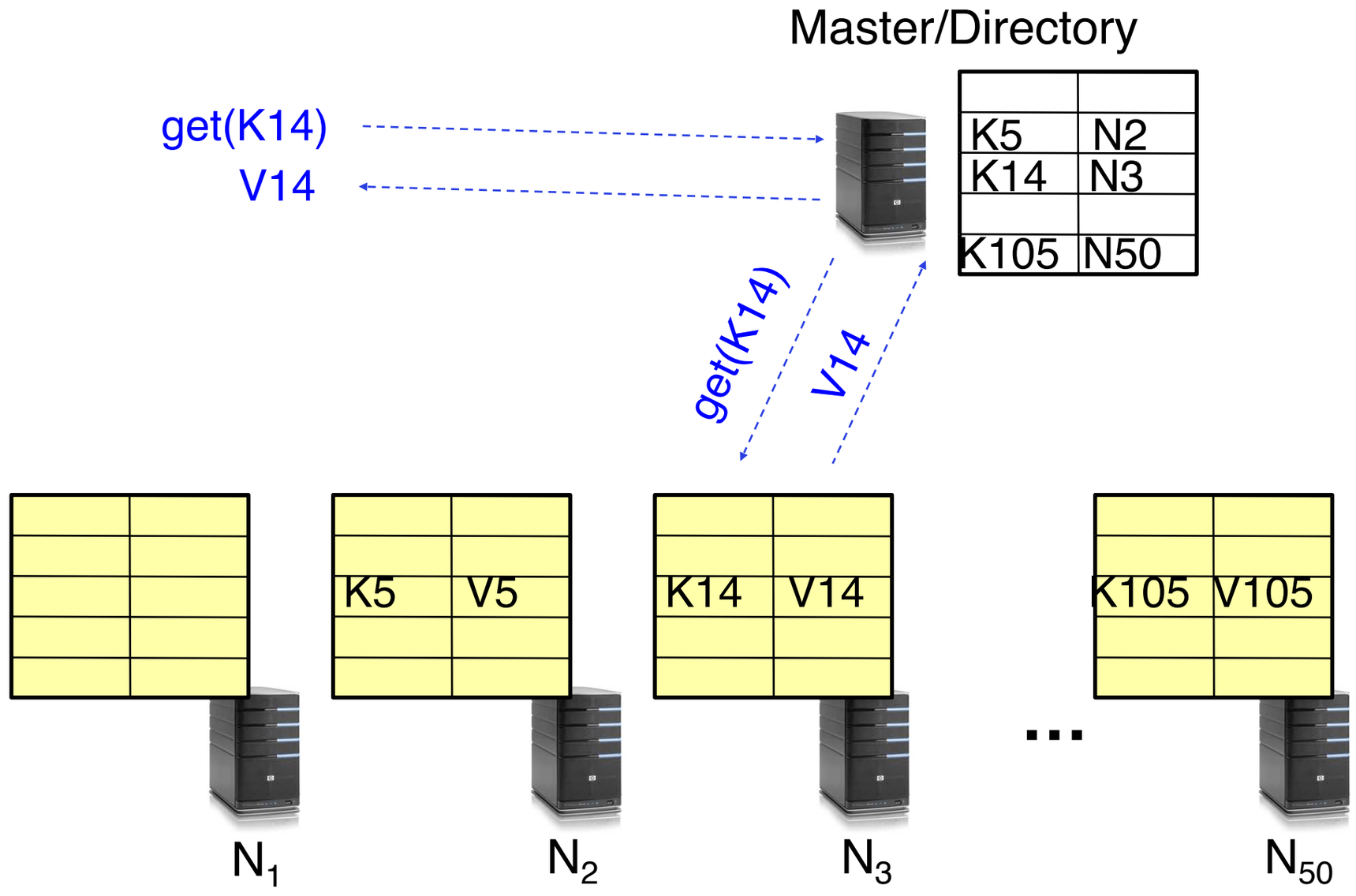
Directory-Based Architecture

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

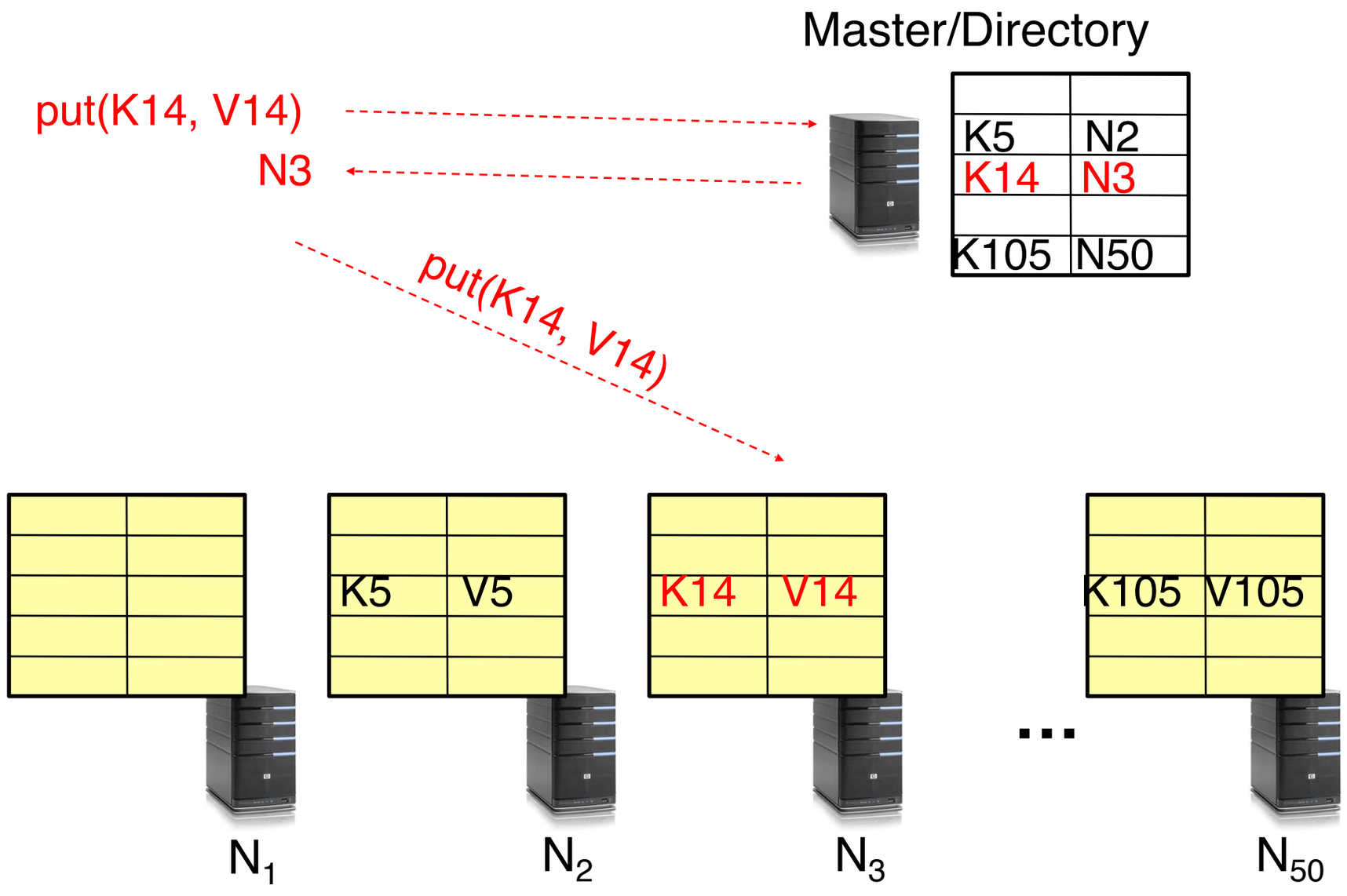


Directory-Based Architecture

Having the master relay the requests → recursive query

Another method: iterative query (this slide)

– Return node to requester and let requester contact node

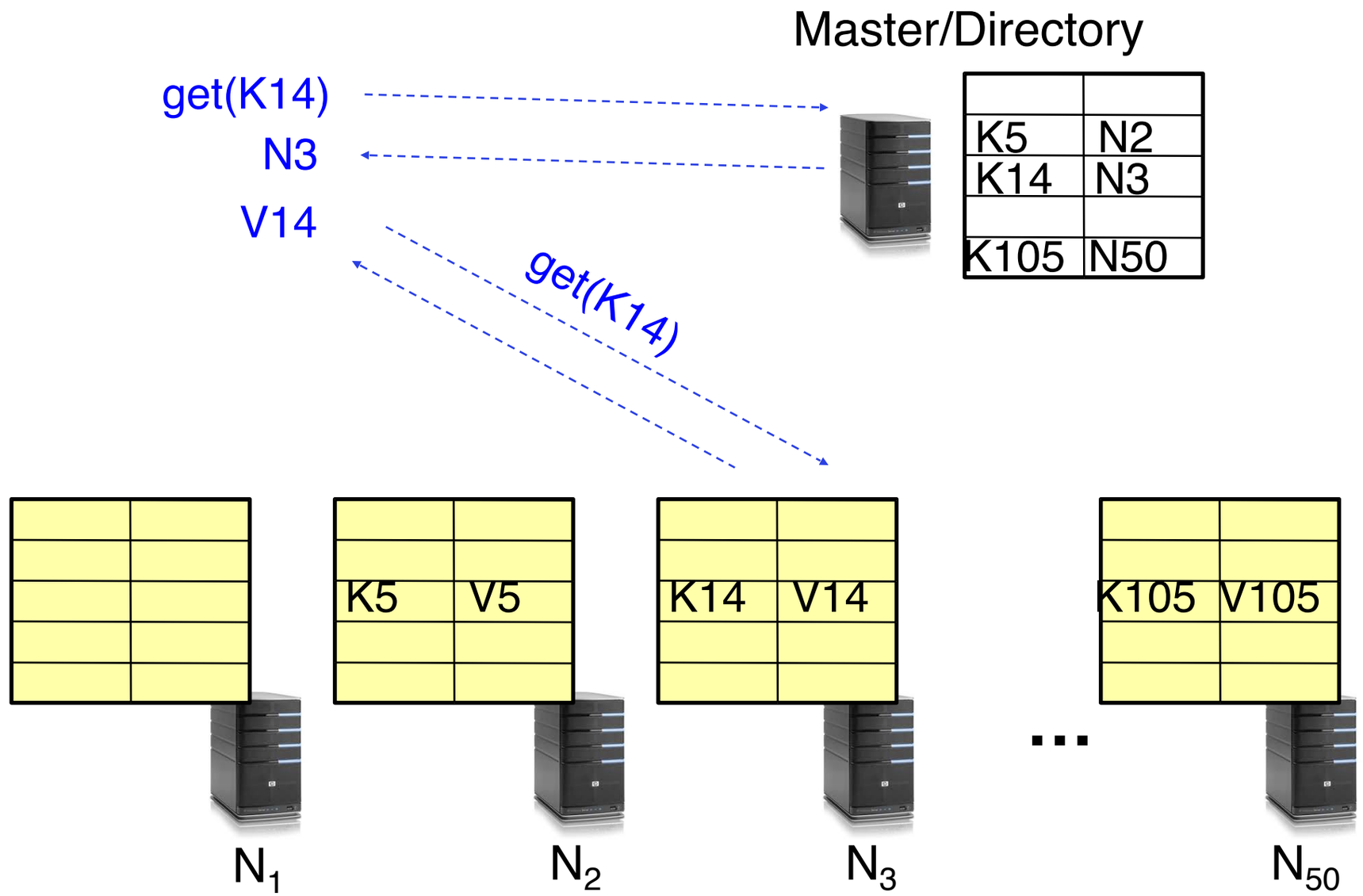


Directory-Based Architecture

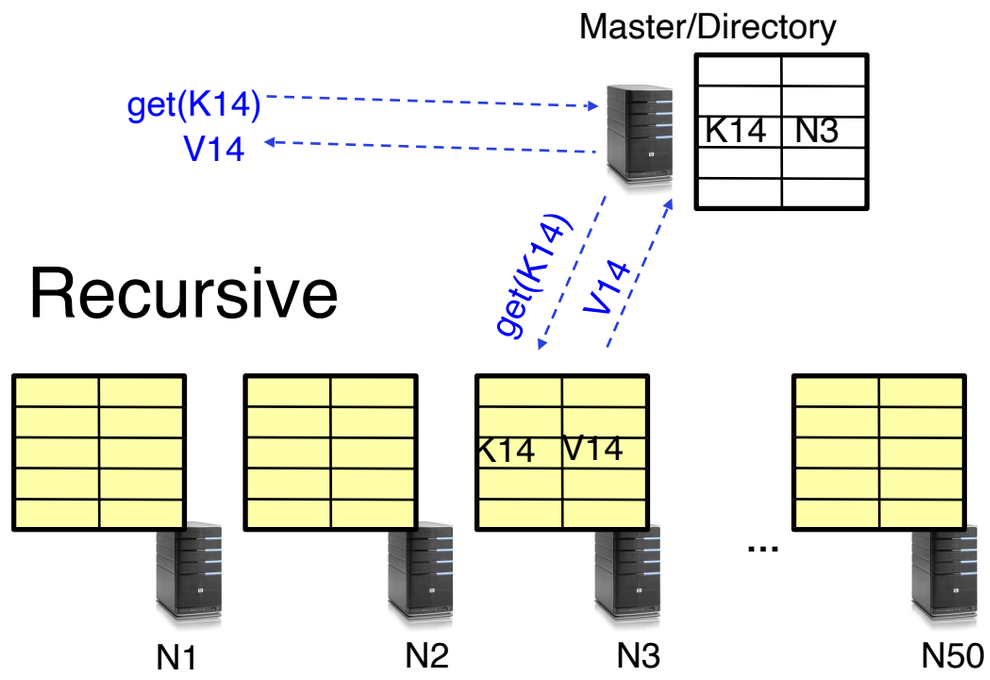
Having the master relay the requests → recursive query

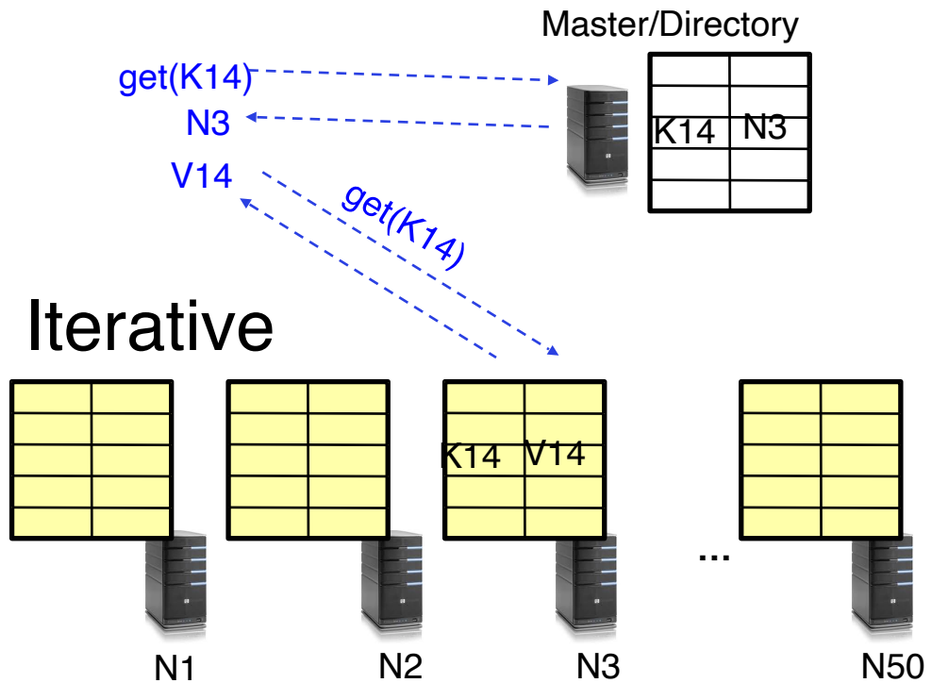
Another method: iterative query

– 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

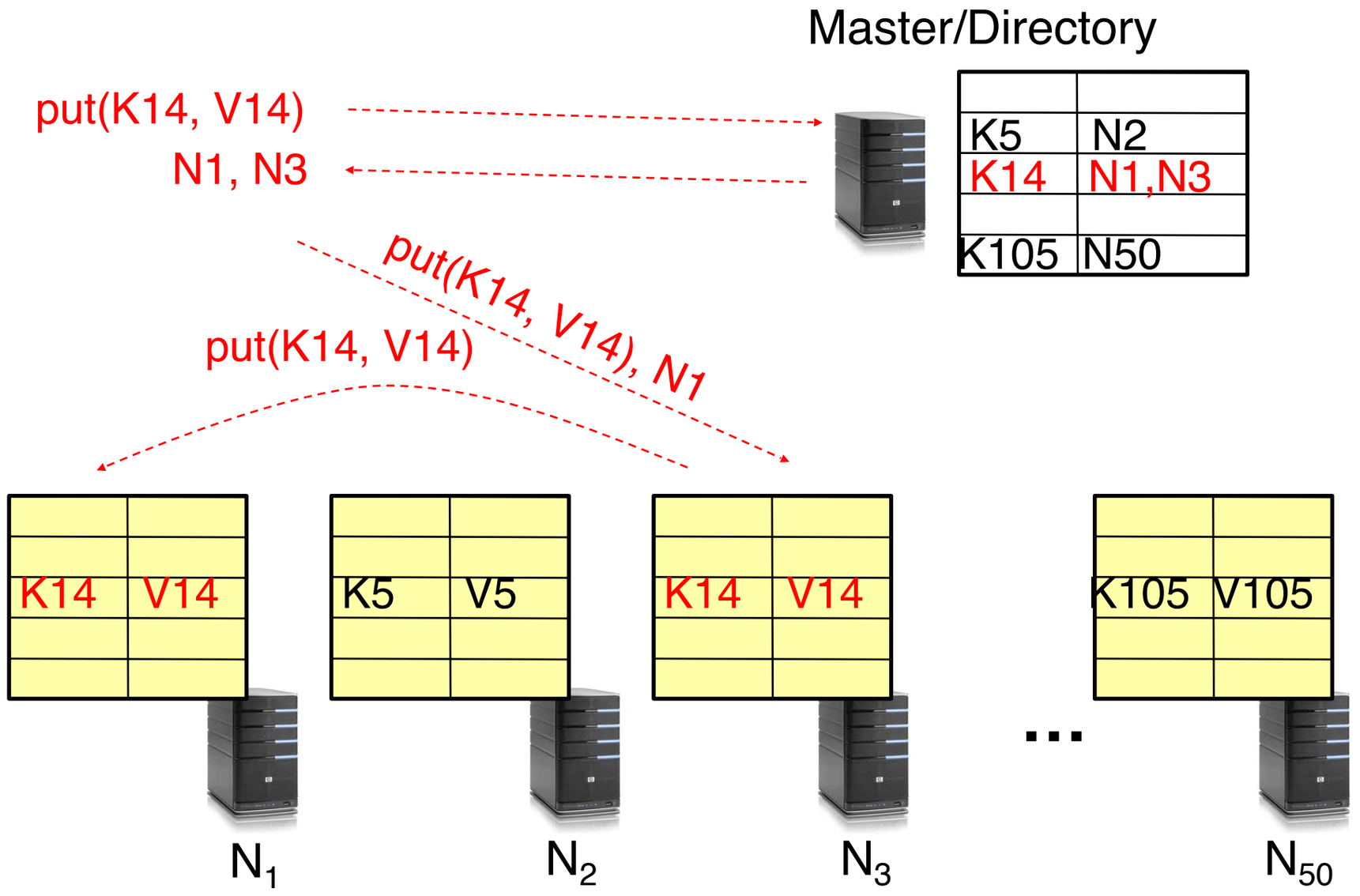
Iterative Query

- Advantages: more scalable
- Disadvantages: slower, harder to enforce data consistency

Fault Tolerance

Replicate value on several nodes

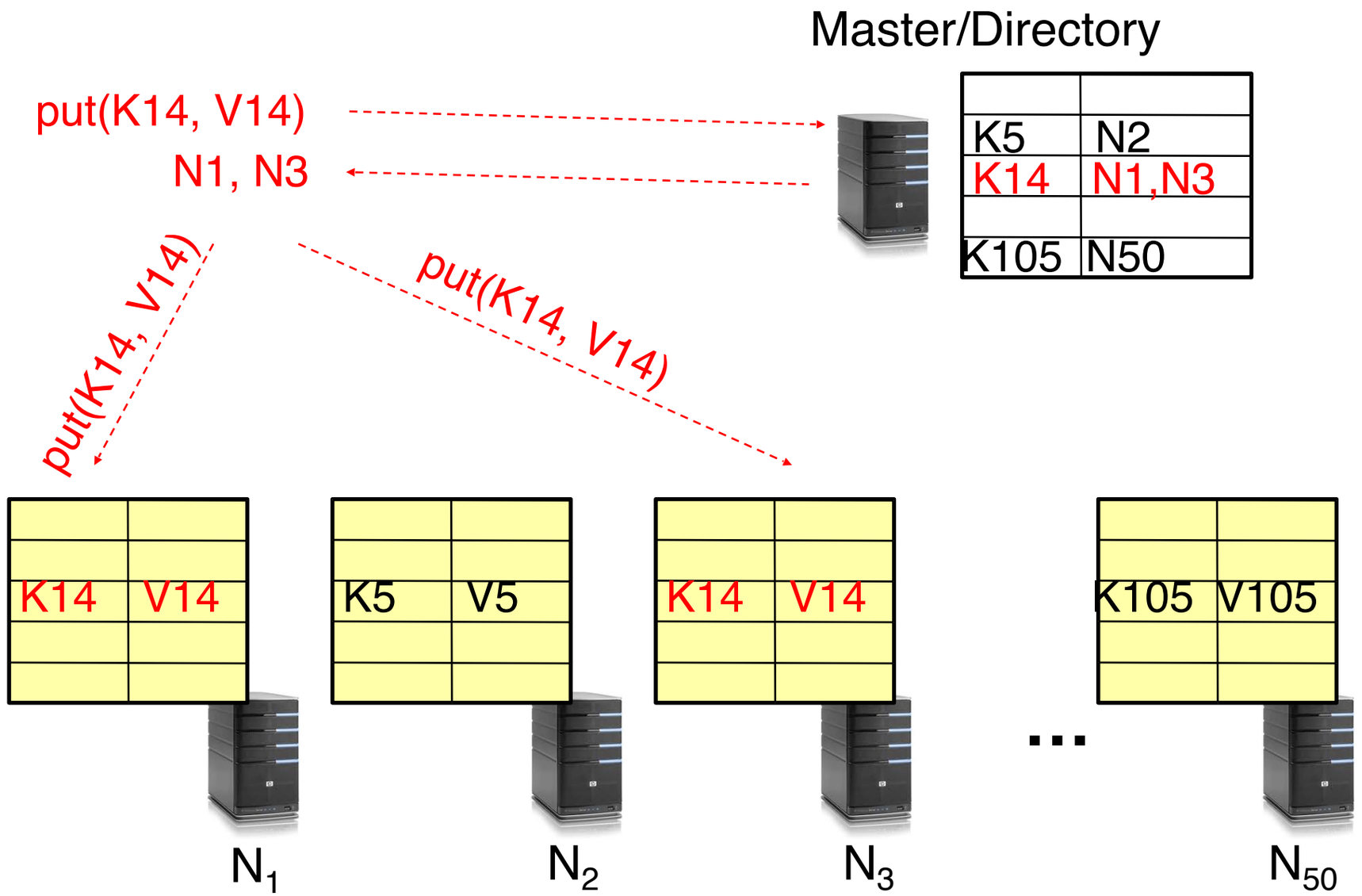
Usually, place replicas on different racks in a datacenter to guard against rack failures



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?

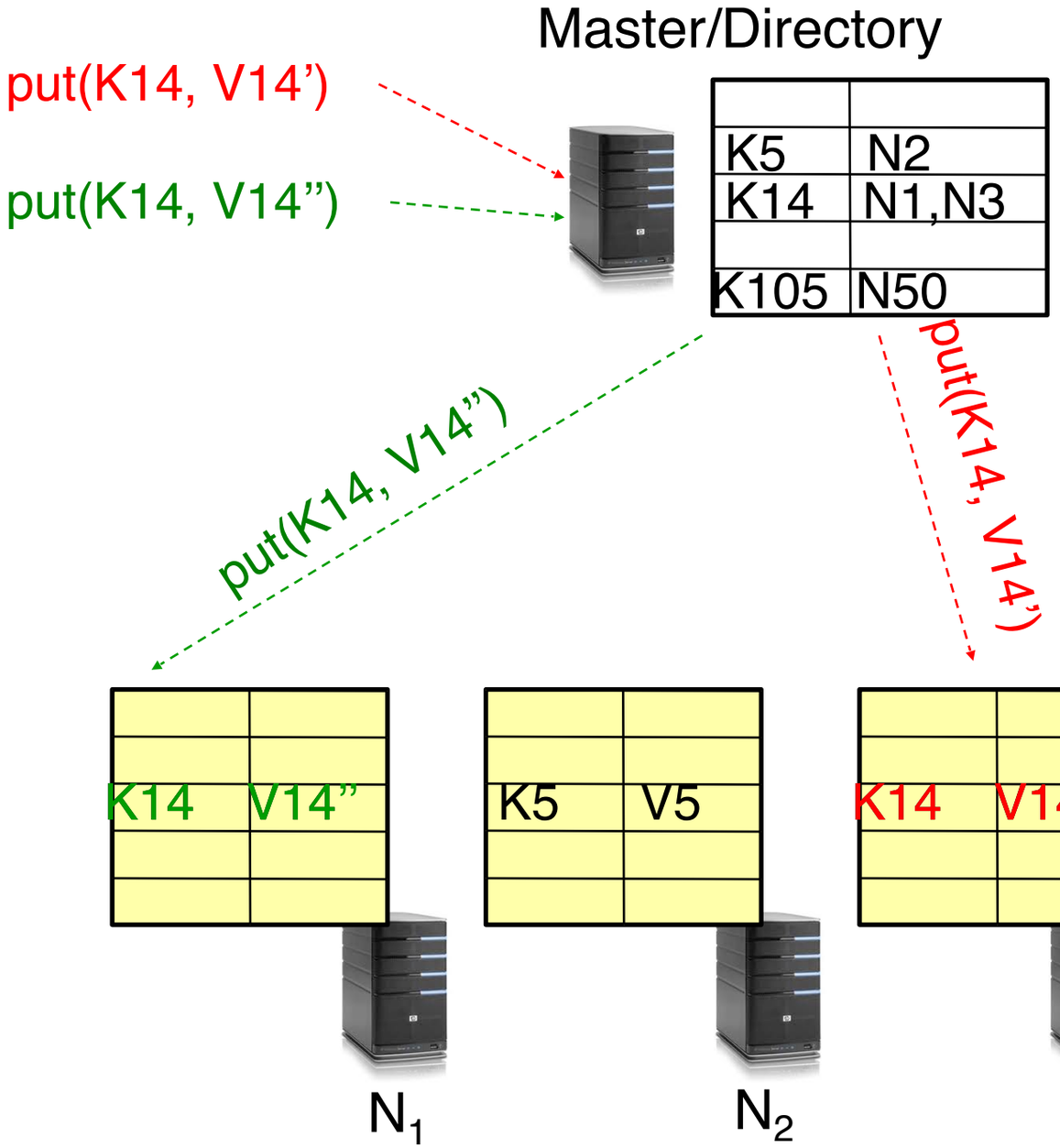
Q: Assume a client calls put(K14, V14) and then get(K14), what is the result returned by get()?

A: It should be V14, right?

Above semantics, not trivial to achieve in distributed systems

Concurrent Writes (Updates)

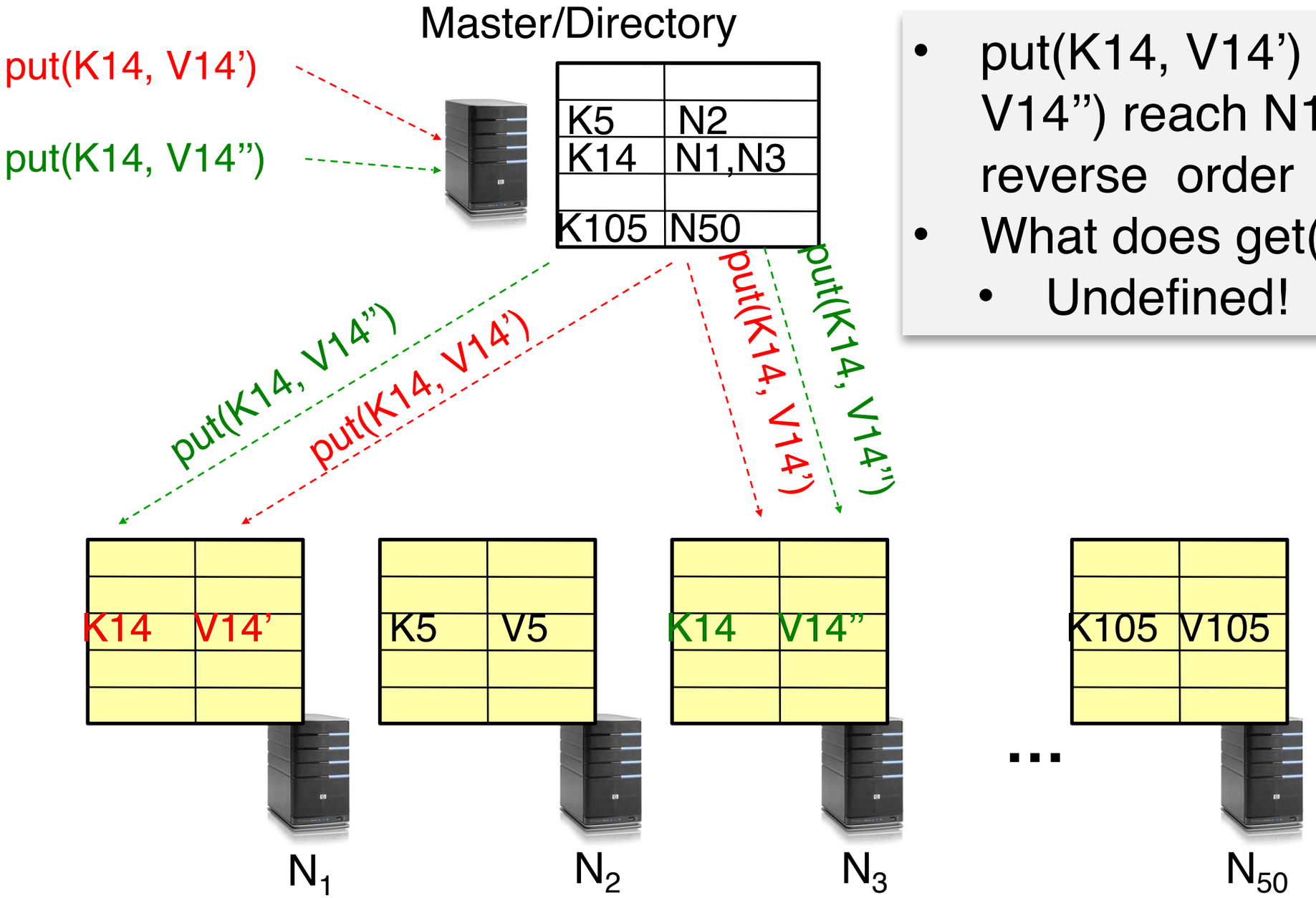
If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- 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

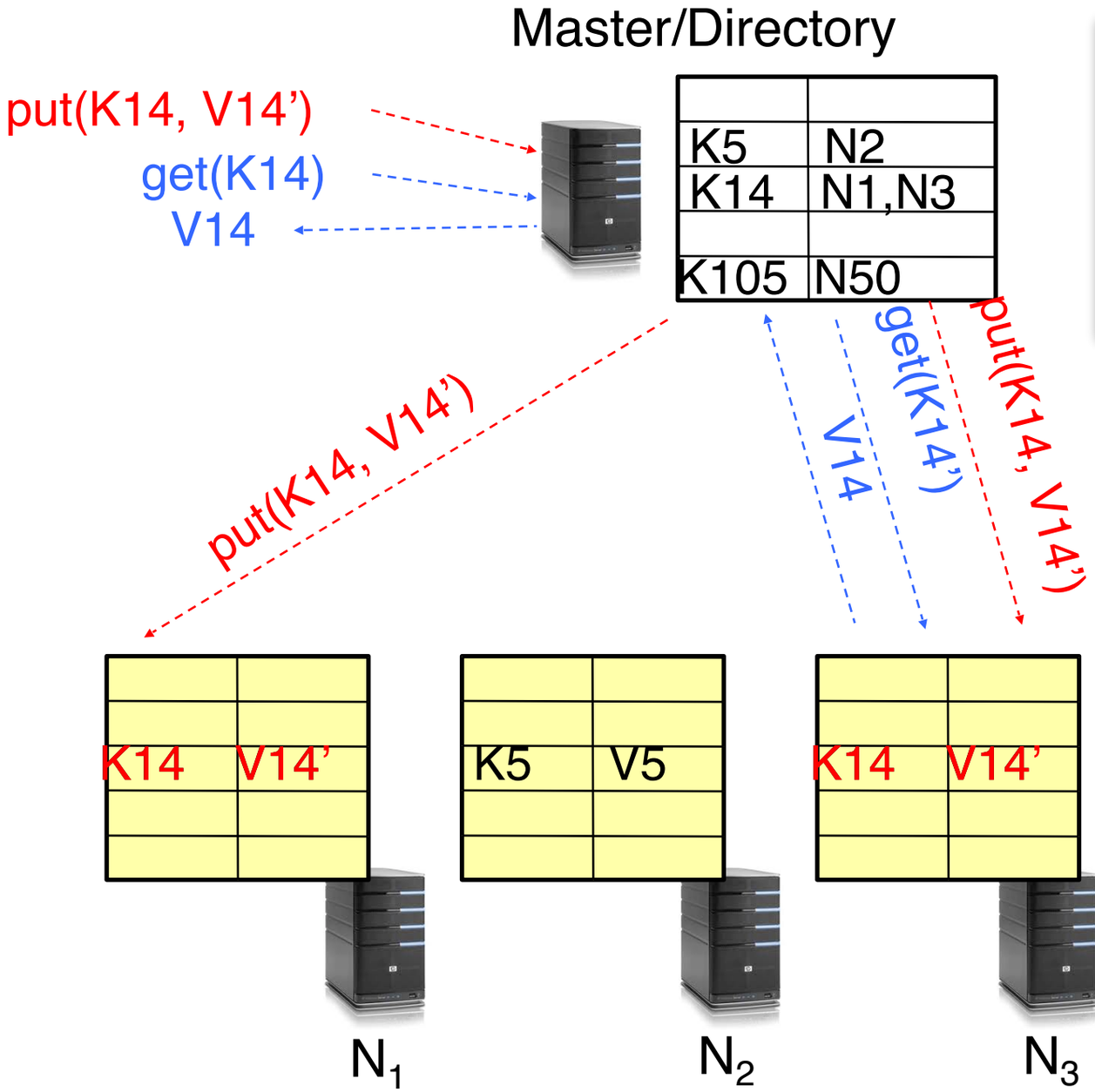


- put(K14, V14') and put(K14, V14") reach N1 and N3 in reverse order
- What does get(K14) return?
  - Undefined!

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')!

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?

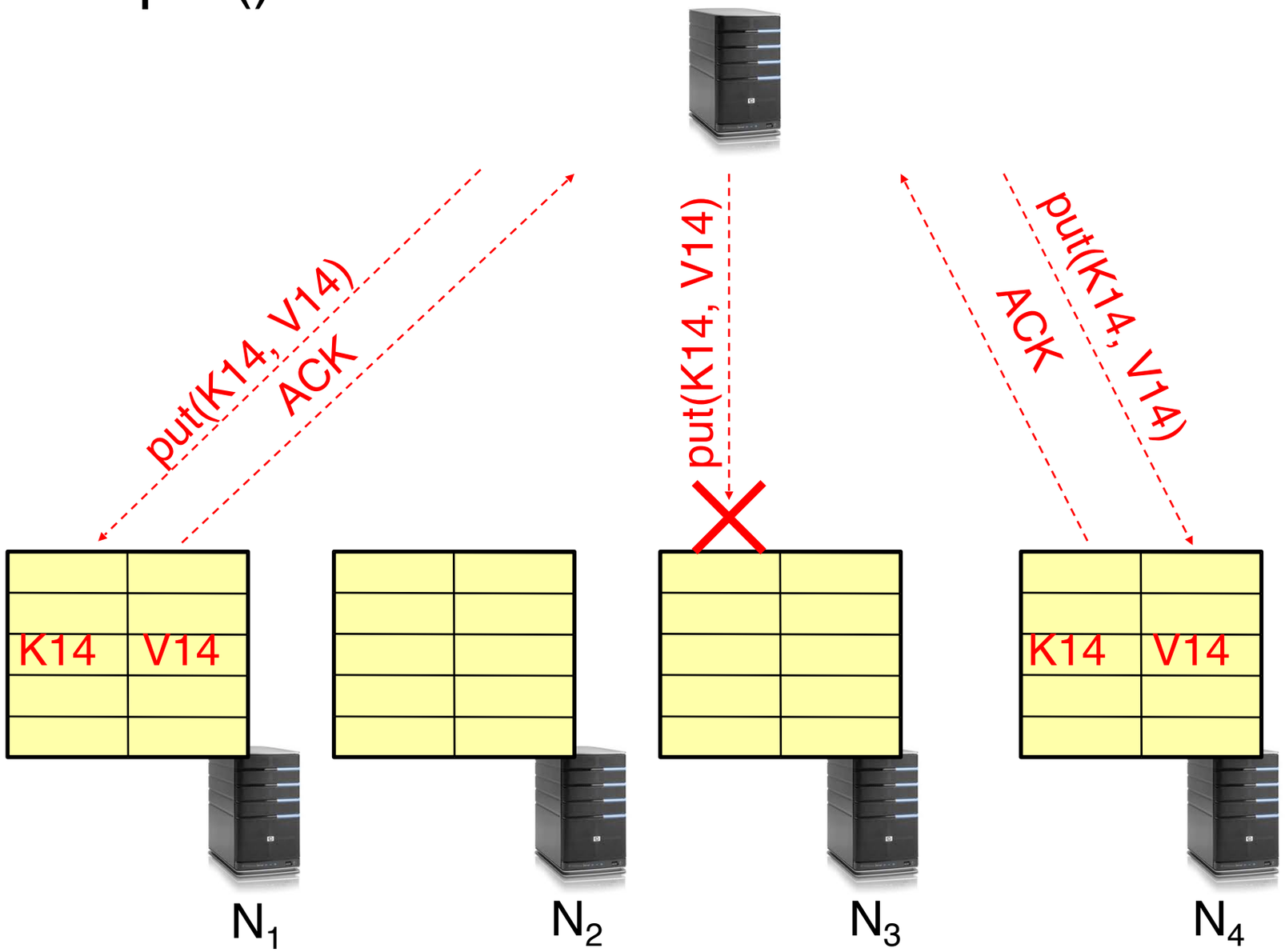
– There is at least one node that contains the update

Quorum Consensus Example

N=3, W=2, R=2

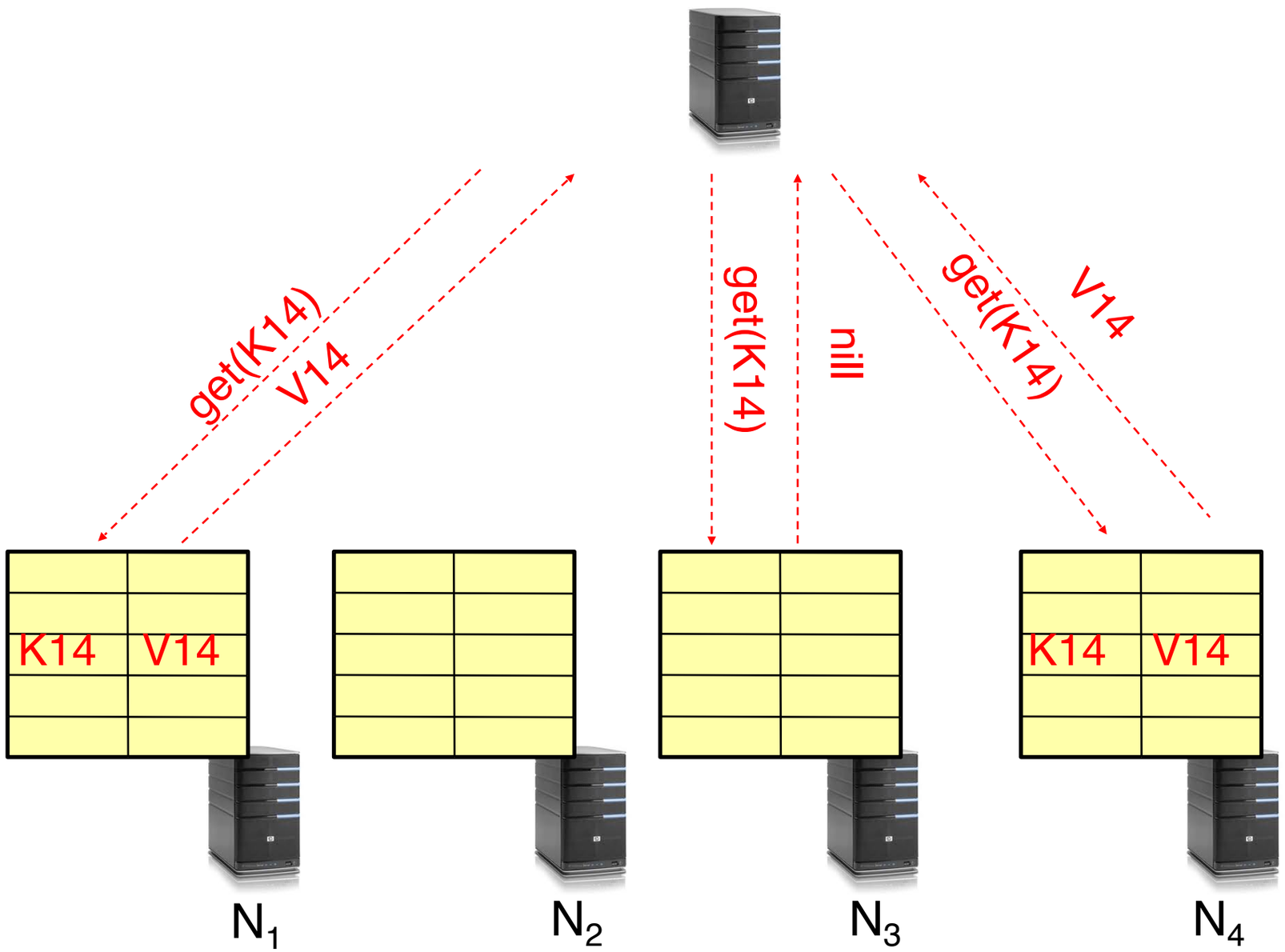
Replica set for K14: {N1, N2, N4}

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

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***

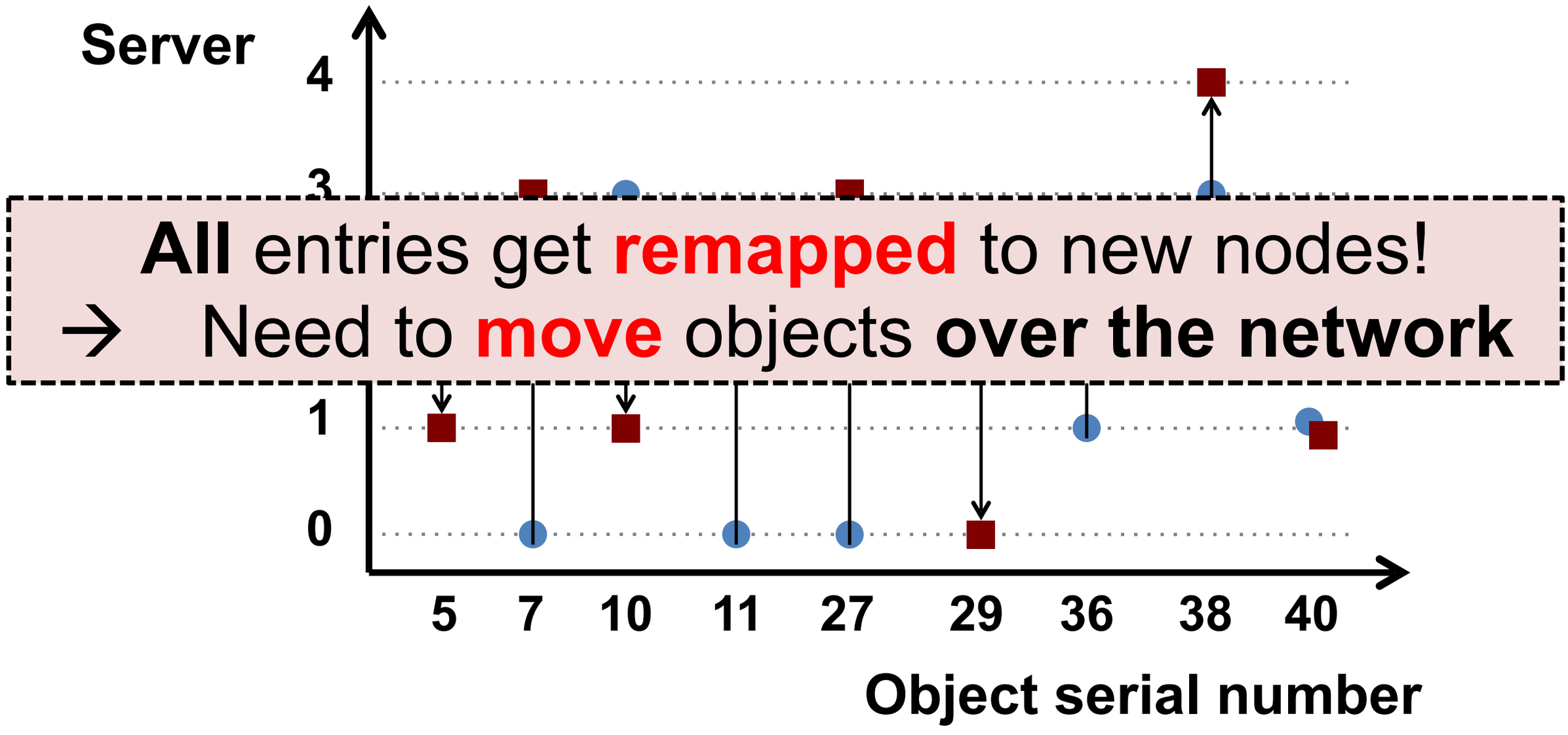
What happens if a server fails or joins (k ← k±1)?

– or different clients have **different estimate** of k?

Problem for modulo hashing: Changing number of servers

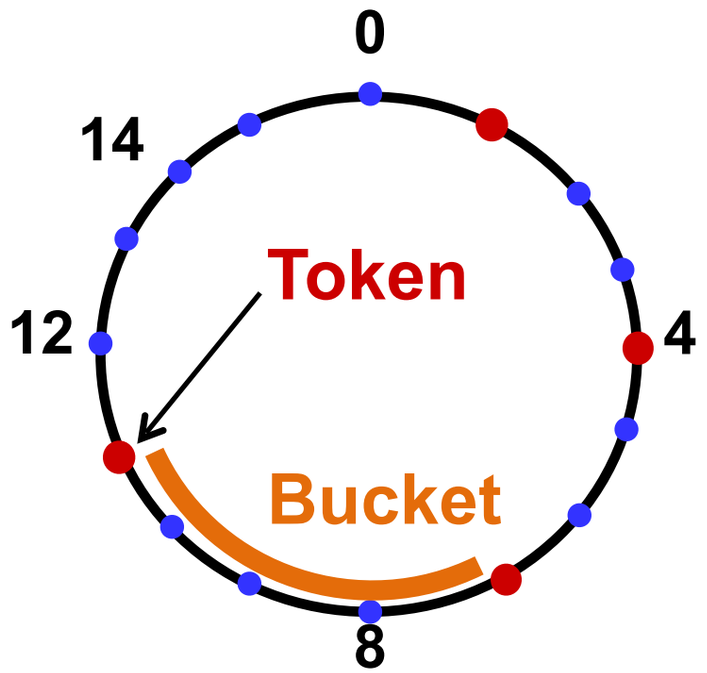
h(*x*) = *x* + 1 (mod 4)

Add one machine: h(*x*) = *x* + 1 (mod 5)



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

Recap: Key to Node Mapping Example

k = 8 → ID space: 0..63

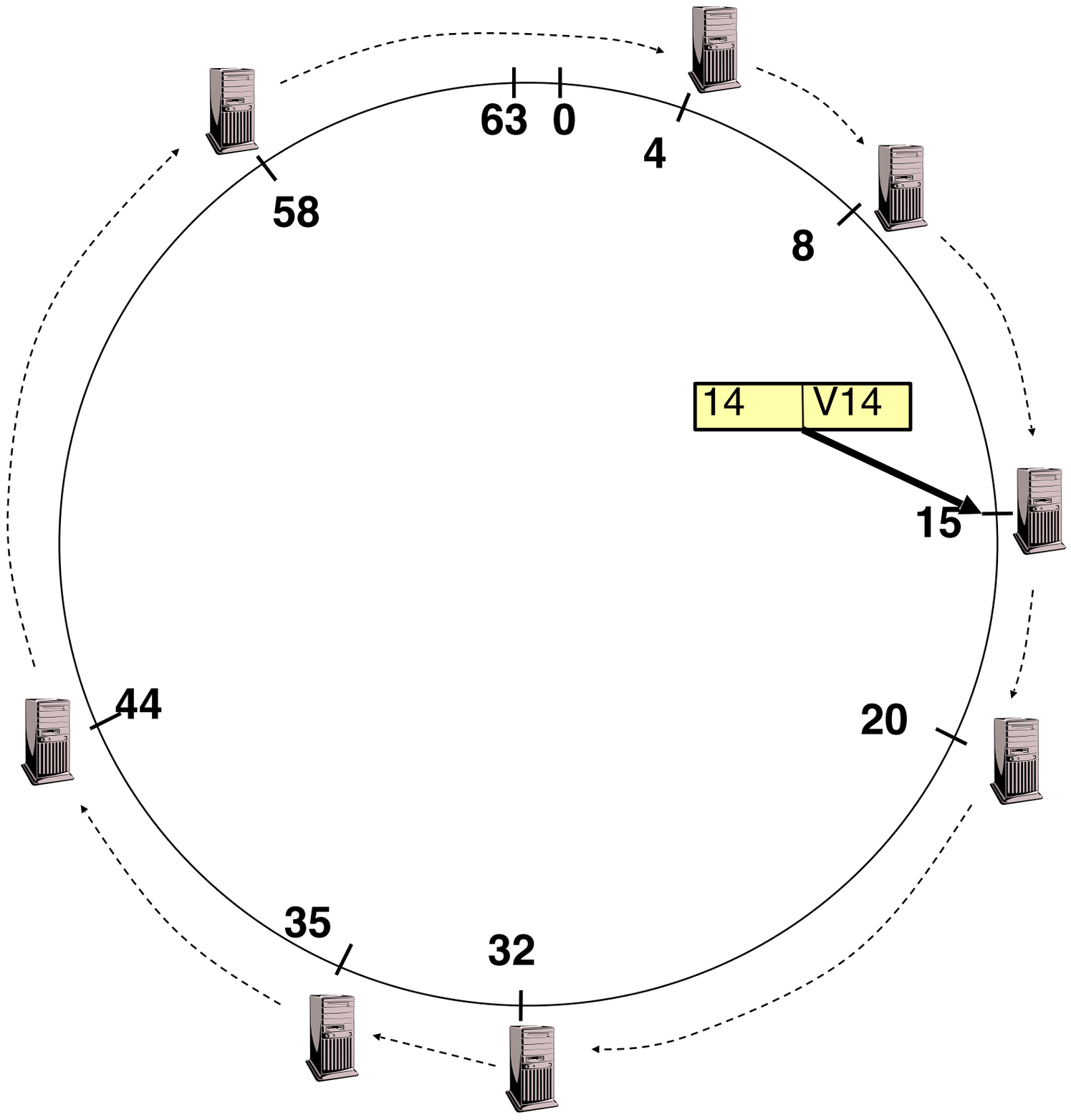
Node 8 maps keys [5,8]

Node 15 maps keys [9,15]

Node 20 maps keys [16, 20]

…

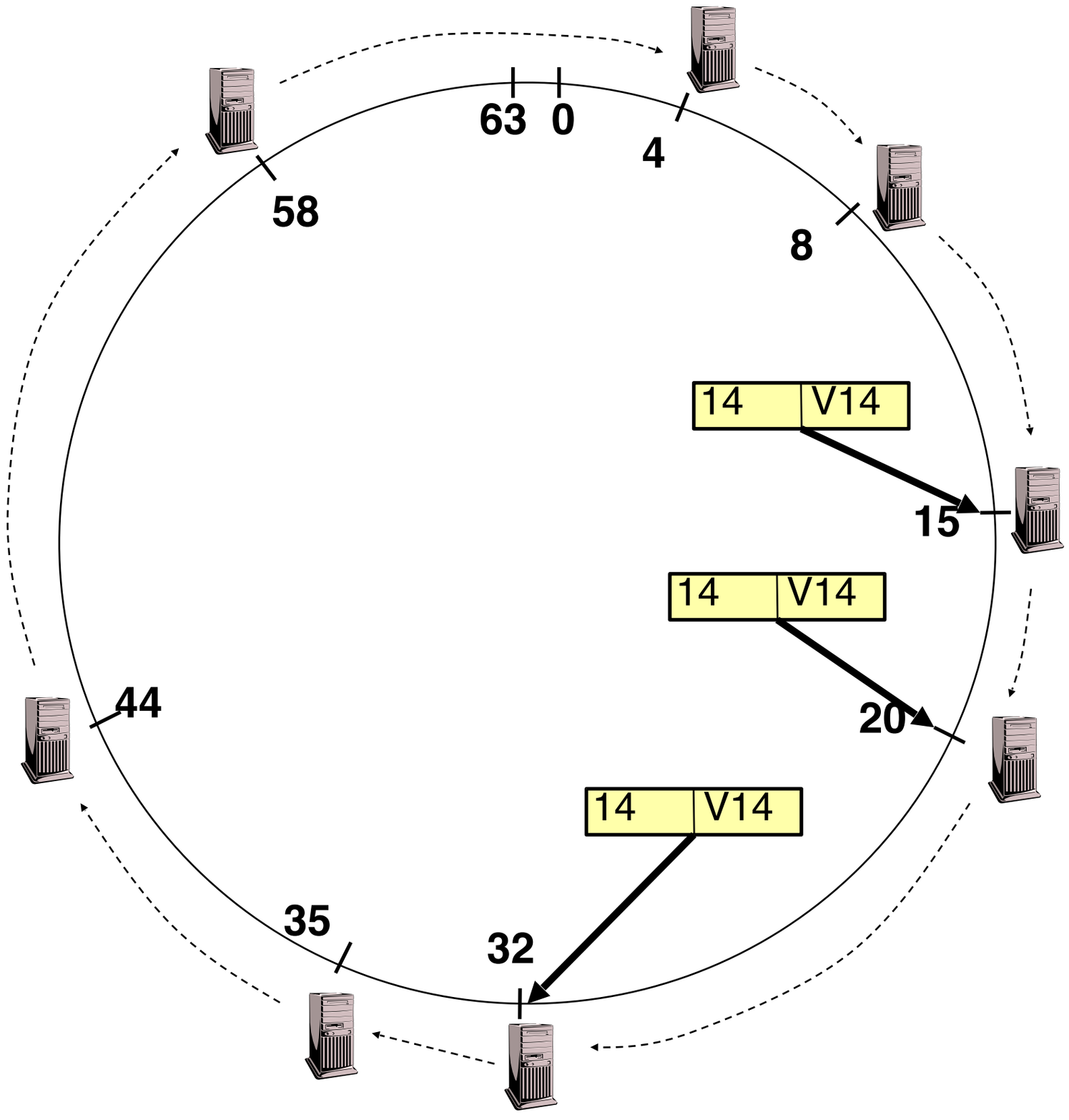
Node 4 maps keys [59, 4]



Storage Fault Tolerance

Replicate tuples on successor nodes

Example: replicate (K14, V14) on nodes 20 and 32

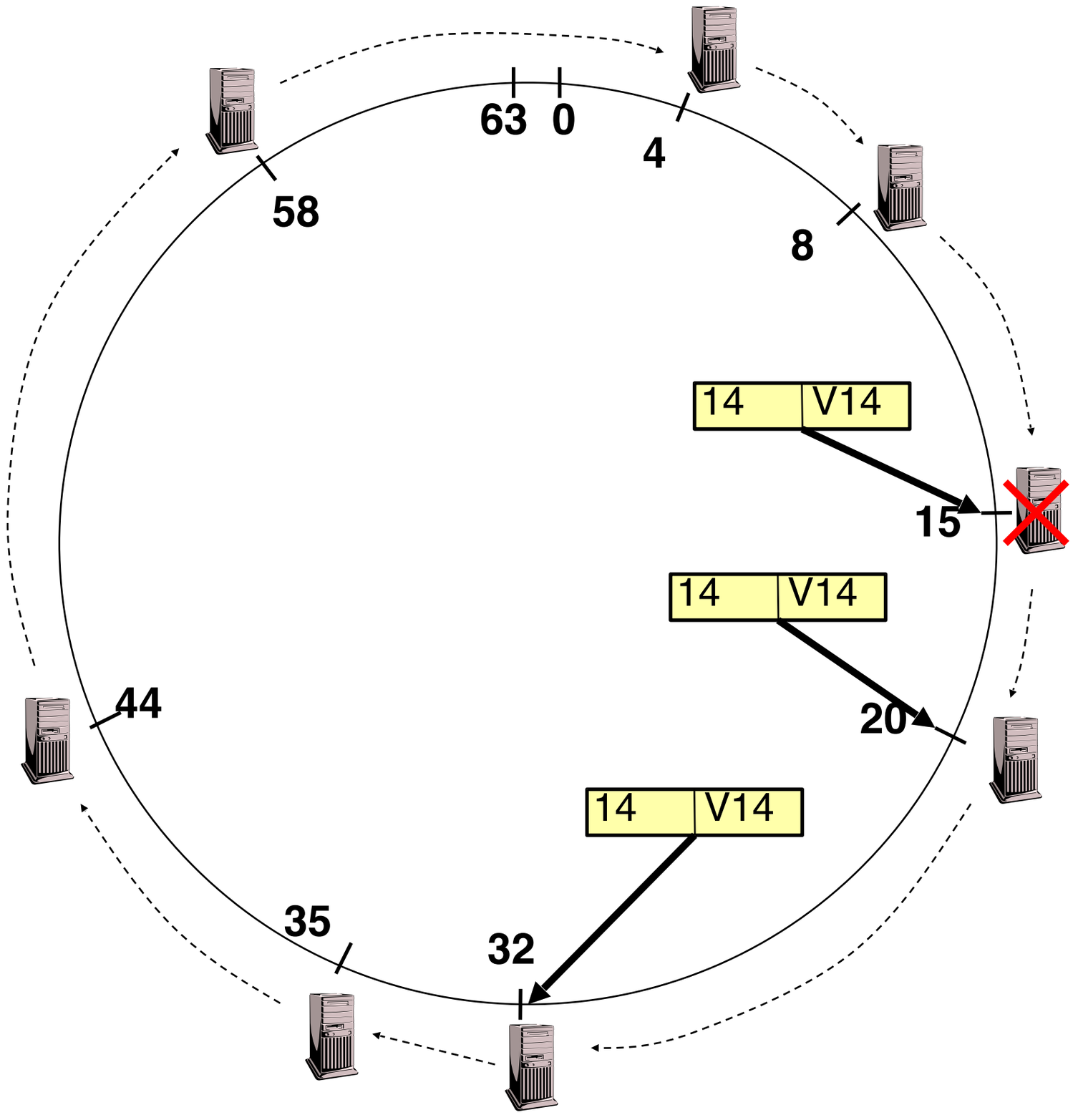


Storage Fault Tolerance

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



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

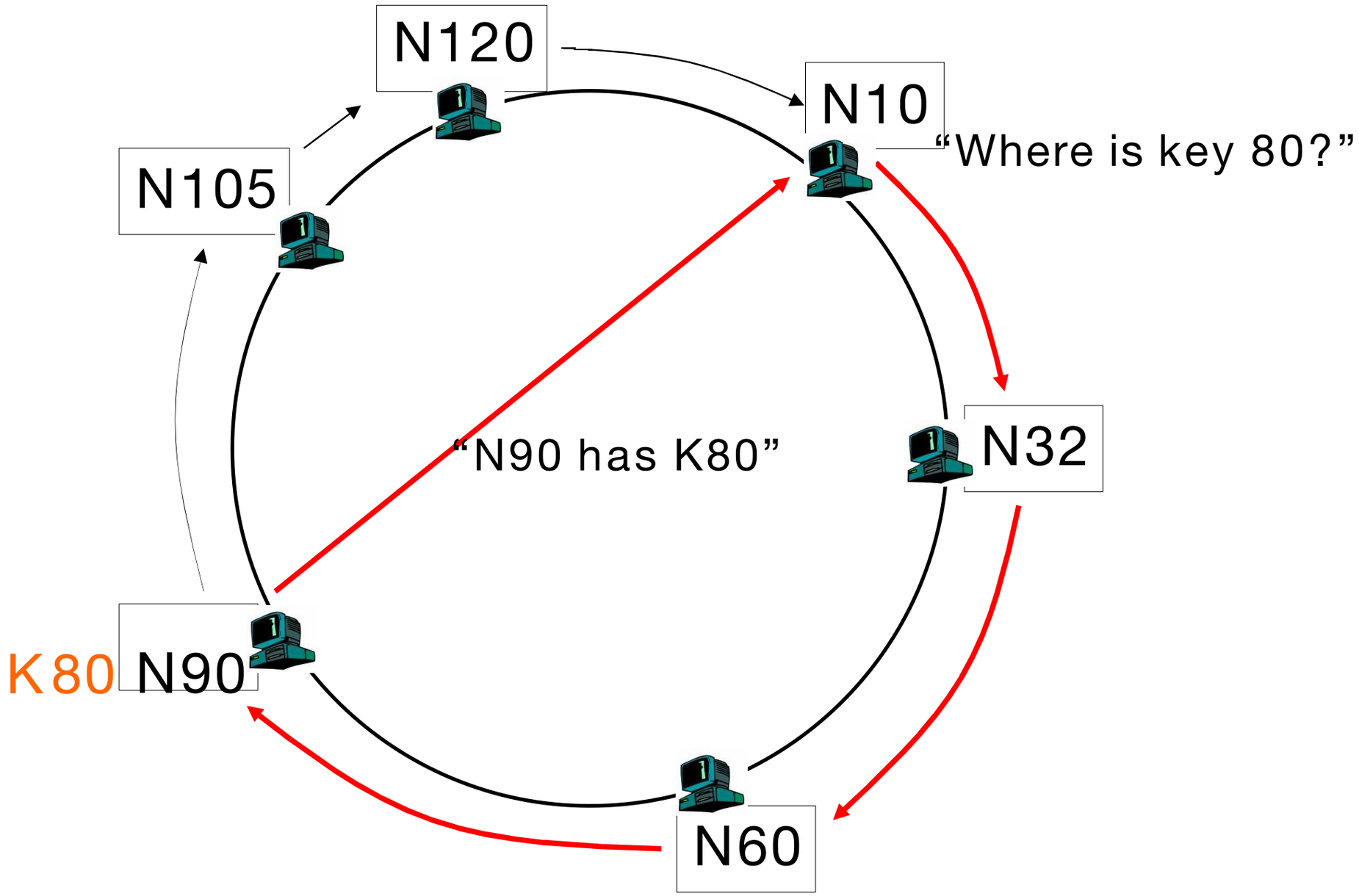
Given a key, find the node storing that key

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

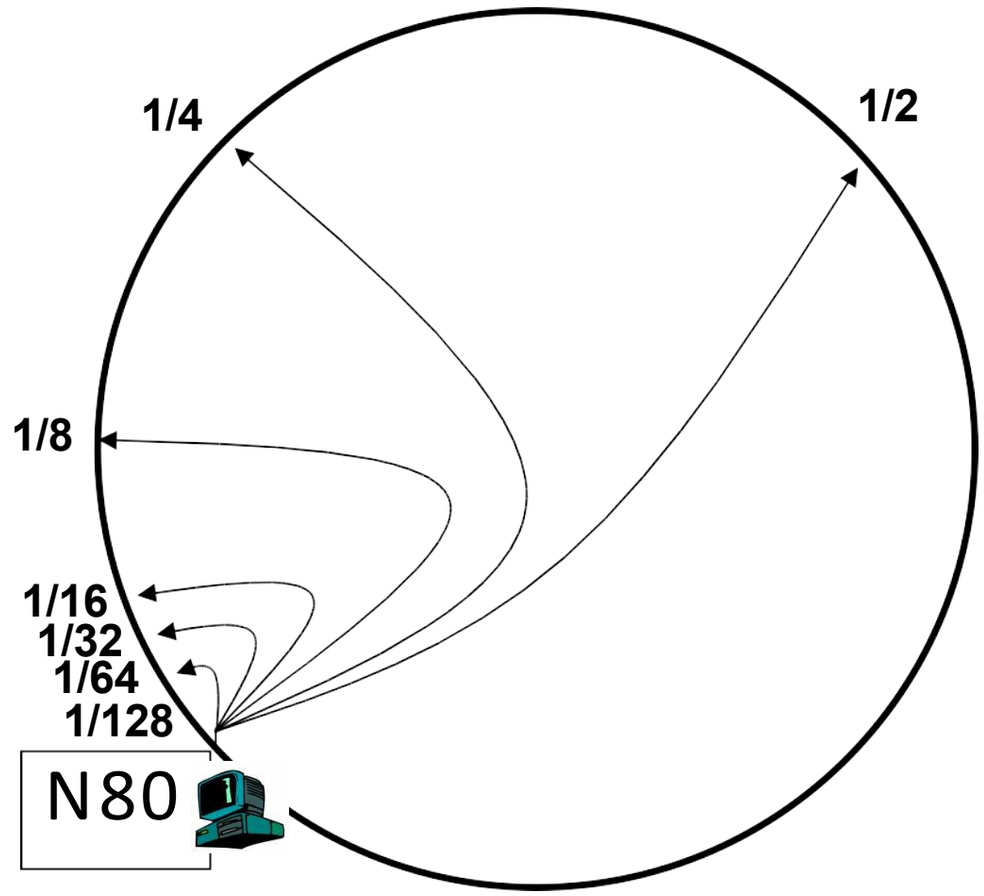
Key advantage: totally distributed

– No point of failure; no hot spot

DHT: Chord Basic Lookup



DHT: Chord "Finger Table"



- Entry i in the finger table of node n is the first node that succeeds or equals $n + 2^i$
- In other words, the ith finger points $1/2^{n-i}$ way around the ring