LECTURE 11

DHTs and Amazon Dynamo

Scaling up

- Assumption so far: All replicas have entire state
  - Example: Every replica has value for every key
- What we need instead:
  - Partition state
  - Map partitions to servers

Partitioning state

- Modulo hashing
  - Apply hash function to key
  - Compute modulo to # of servers (N)
  - Store (key, value) pair at *hash(key) mod N*

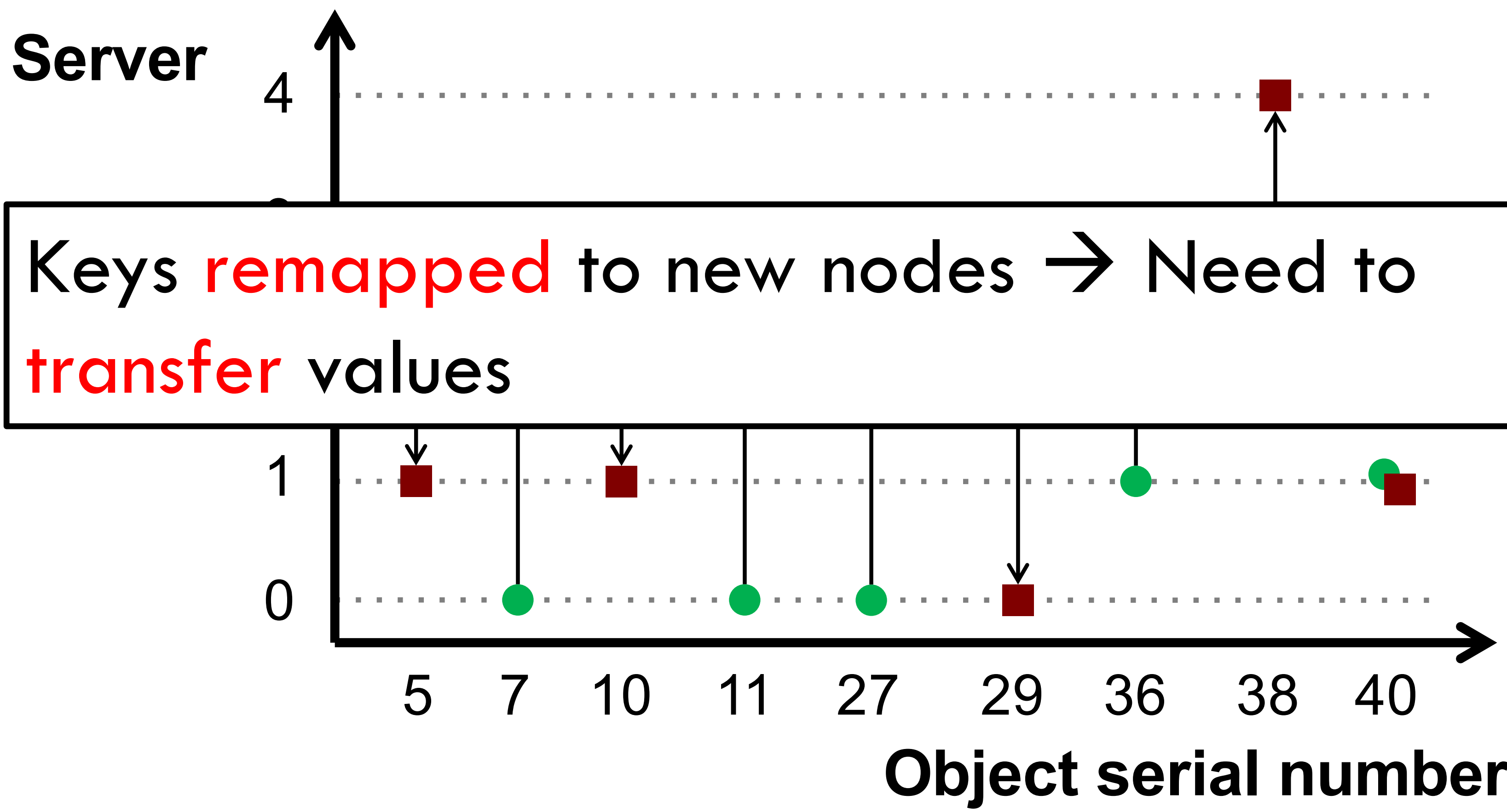
- Example:
  - Store student's transcripts across 4 servers
  - Hash function = (Year of birth) mod 4
  - Hash function = (Date of birth) mod 4

- Problem: Skew in load across servers

Problem for modulo hashing: Changing number of servers

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

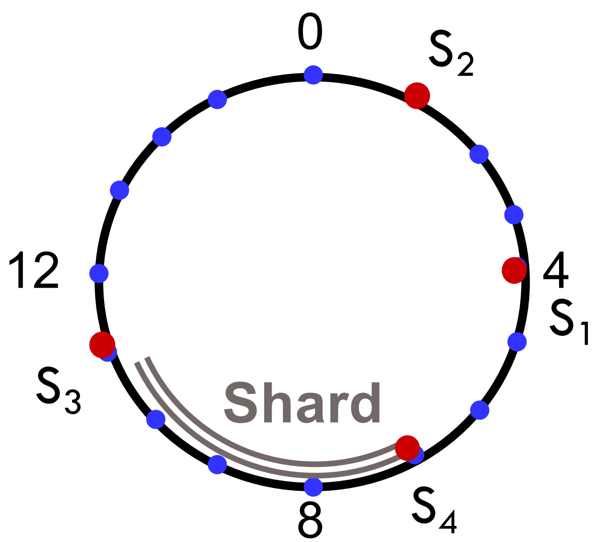
Add one machine: h(x) = x + 1 (mod 5)



Keys **remapped** to new nodes → Need to **transfer** values

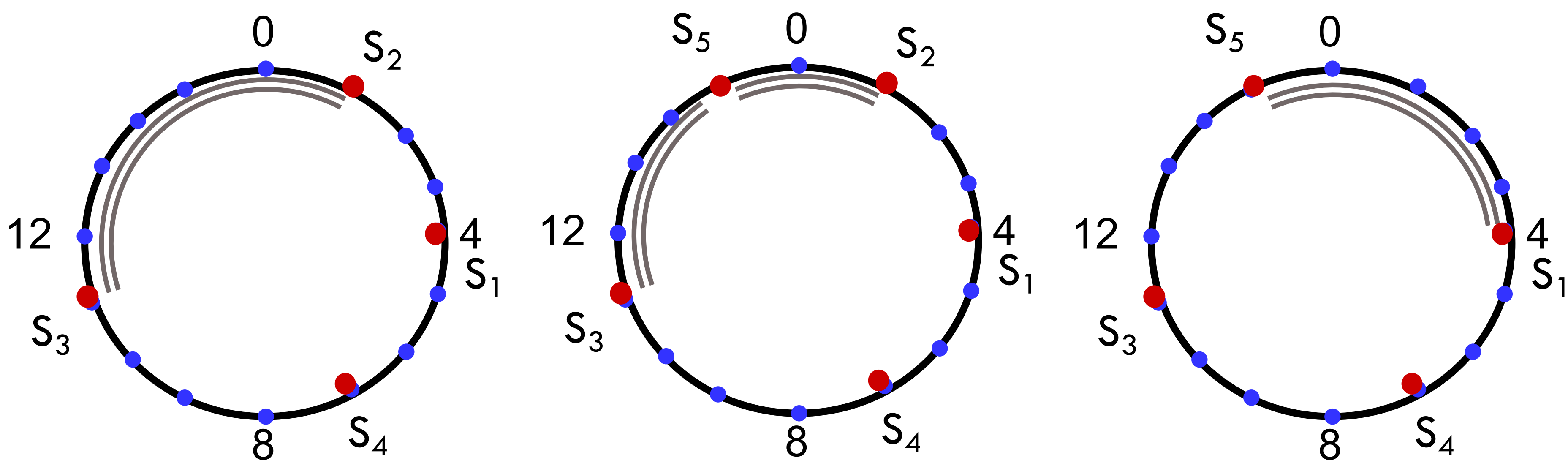
Consistent Hashing

- Represent hash space as a circle



- Partition keys across servers
  - Assign every server a random ID
  - Hash server ID
  - Server responsible for keys between predecessor and itself
- How to map a key to a server?
  - Hash key and execute read/write at successor

Adding/Removing Nodes



- Minimizes migration of state upon change in set of servers
  - Server addition: New server splits successor's shard
  - Server removal: Successor takes over shard

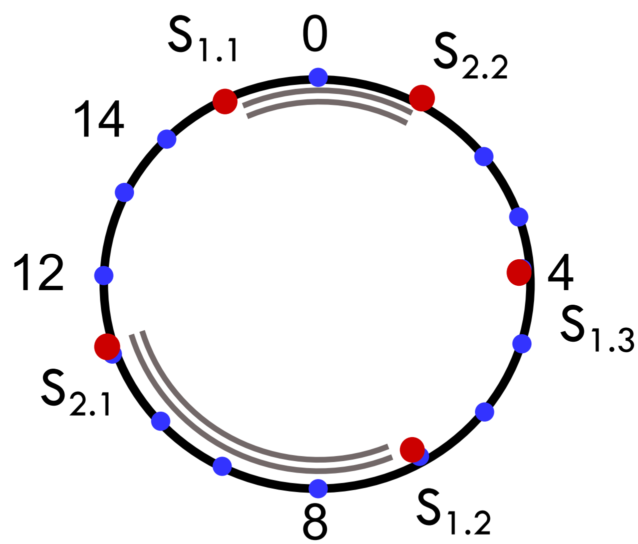
Virtual nodes

- Each server gets multiple (say v) random IDs
  - Each ID corresponds to a virtual node

- If N servers with v virtual nodes per server, each virtual node owns $1/(vN)^{th}$ of hash space

- Larger v → better load balancing
  - Vary v across servers to account for heterogeneity

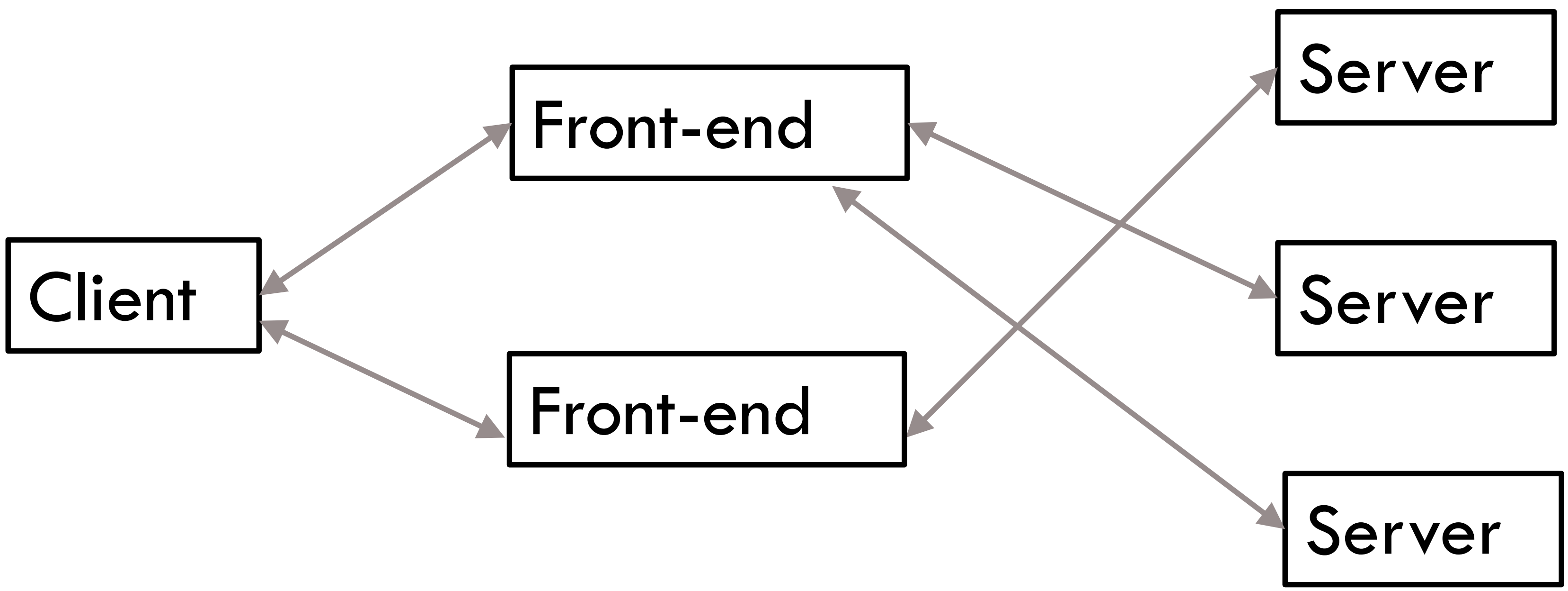
Virtual nodes



- What happens upon server failure?
  - v successors take over
  - Each now stores **(v+1)/v×1/N**th of hash space

Using Consistent Hashing

How does client map keys to servers?

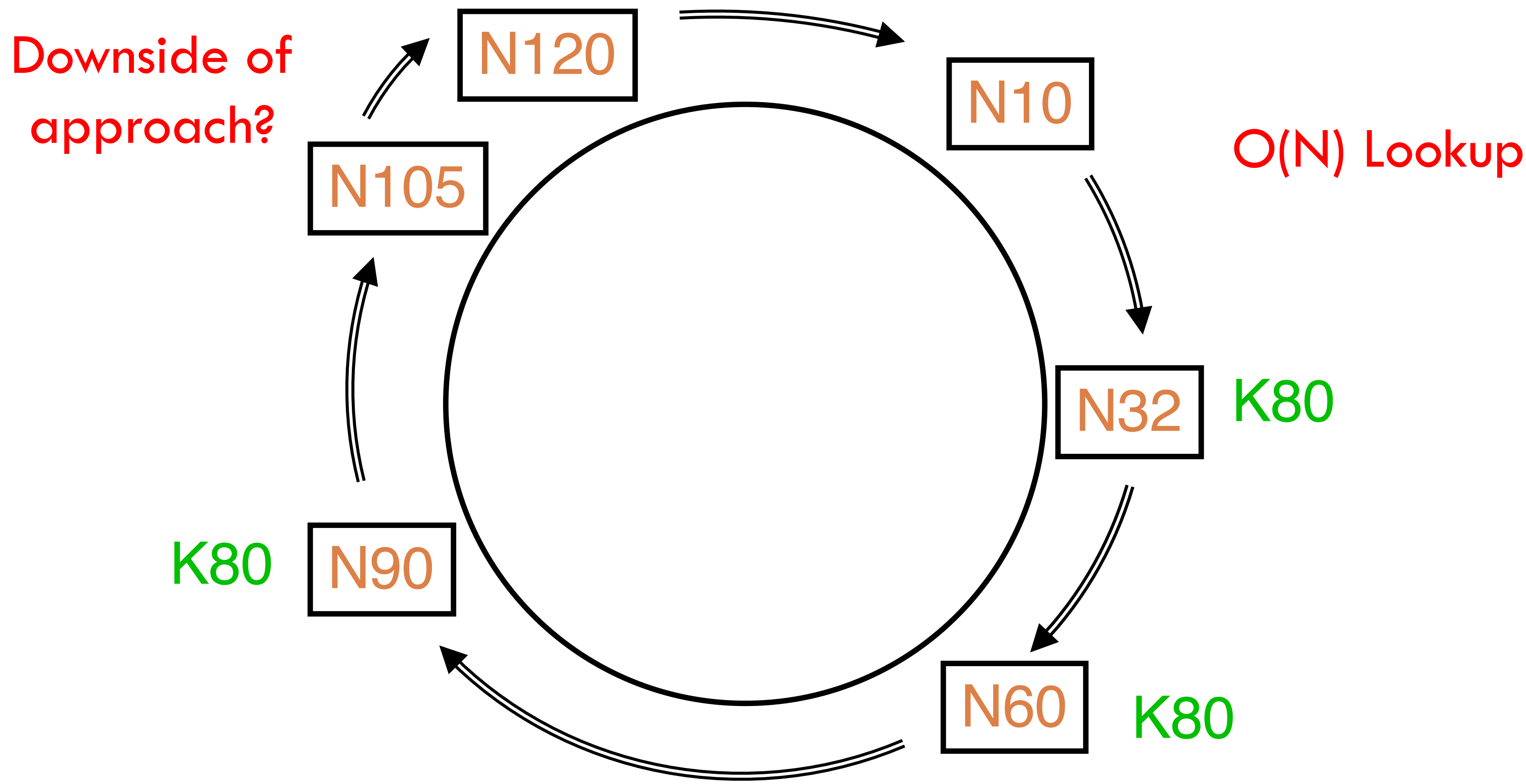


Front-ends must agree on set of active servers

Distributed Hash Table

- Scalable lookup of node responsible for any key
  - Scale to thousands (or even millions) of nodes
  - No one node knows all nodes in the system
- Example usage:
  - Trackerless BitTorrent
  - Key = File content hash
  - Value = IP addresses of nodes that have file content

Successor pointers



- If you don't have value for key, forward to succ.

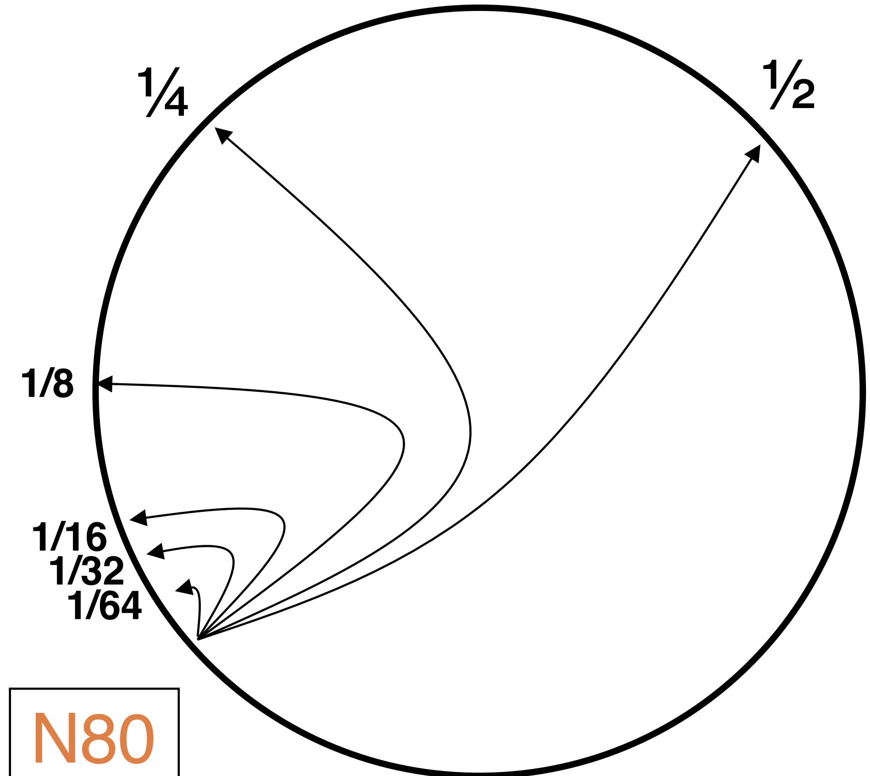
Efficient lookups

- What's required to enable O(1) lookups?
  - Every node must know all other nodes

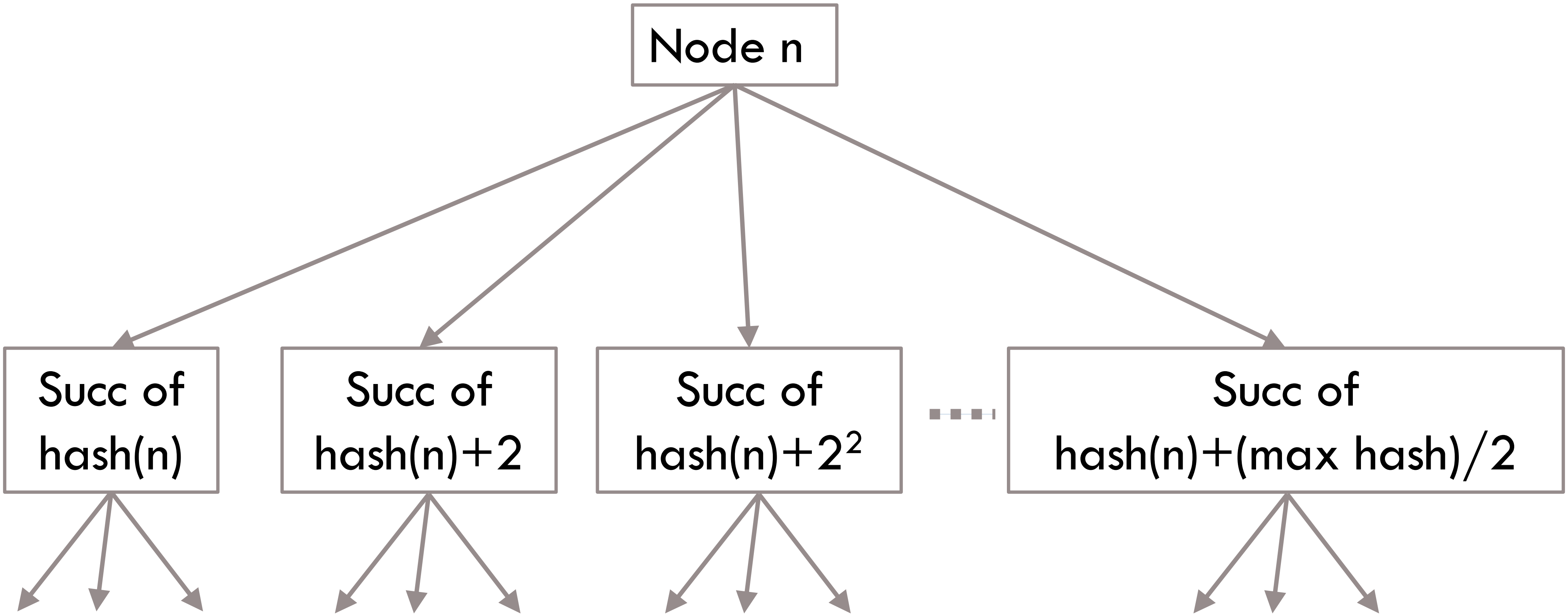
- Need to convert linear search to binary search
- Idea: Maintain log(N) pointers to other nodes
  - Called finger table
  - Pointer to node ½-way across hash space
  - Pointer to node ¼-way across hash space
  - …

Finger tables

- i'th entry at node n points to successor of hash(n)+2^i
  - # of entries = # of bits in hash value
- Binary lookup tree rooted at every node
  - Threaded through others' finger tables



Finger tables



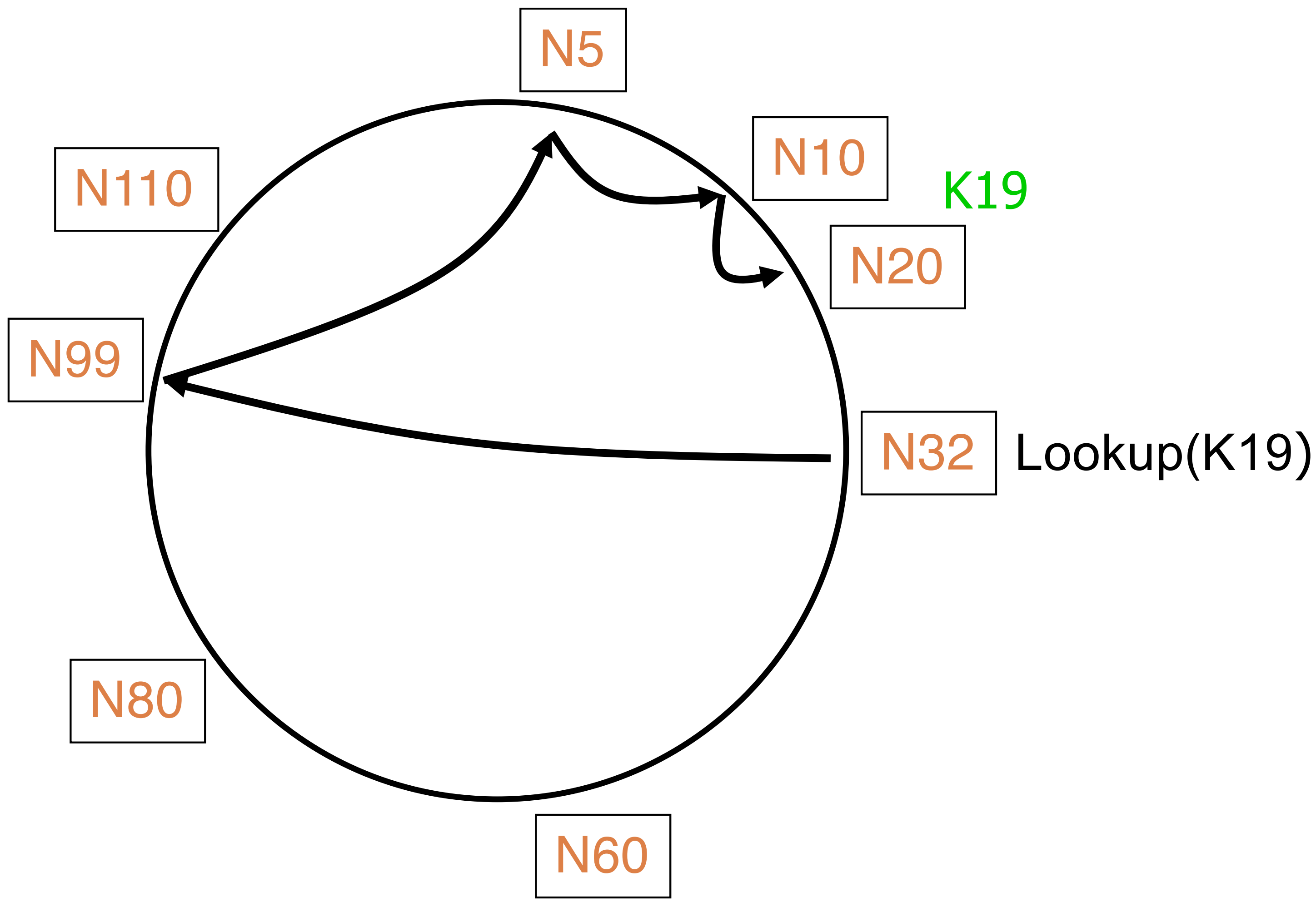
How to recursively use finger tables to locate node for key k?

Lookup with finger table

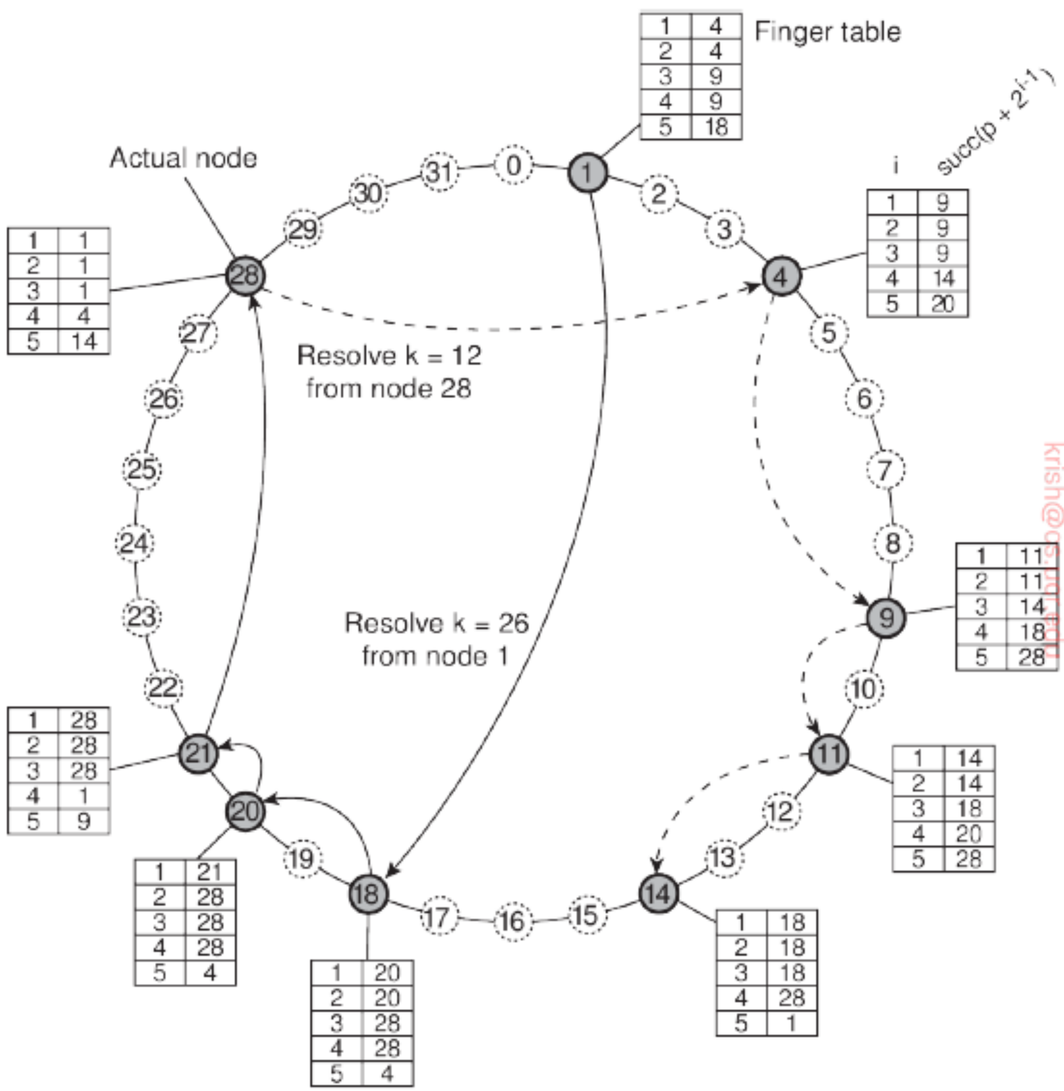

```
Lookup(key k, node n)
  look in local finger table for
       highest f s.t. hash(f) < hash(k)
  if f exists
       call Lookup(k, f)            //next hop
  else
       return n's successor         //done
```

Modulo arithmetic

Lookups take O(log *N*) hops



Example



Resolving key 26 from node 1 and key 12 from node 28 using DHTs in Chord (using finger tables)

Is log(N) lookup fast or slow?

- For a million nodes, it's 20 hops
- If each hop takes 50 ms, lookups take **a second**
- If each hop has 10% chance of failure, it's a couple of timeouts
- So log(N) is better than O(N) but **not great**

Handling churn in nodes

- Need to update finger tables upon addition or removal of nodes
- Hard to preserve consistency in the face of these changes

Amazon Dynamo

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

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service

- Added to "Hall of Fame" at SOSP'17
- Rumored to be underpinning of Amazon S3's architecture

Dynamo settings

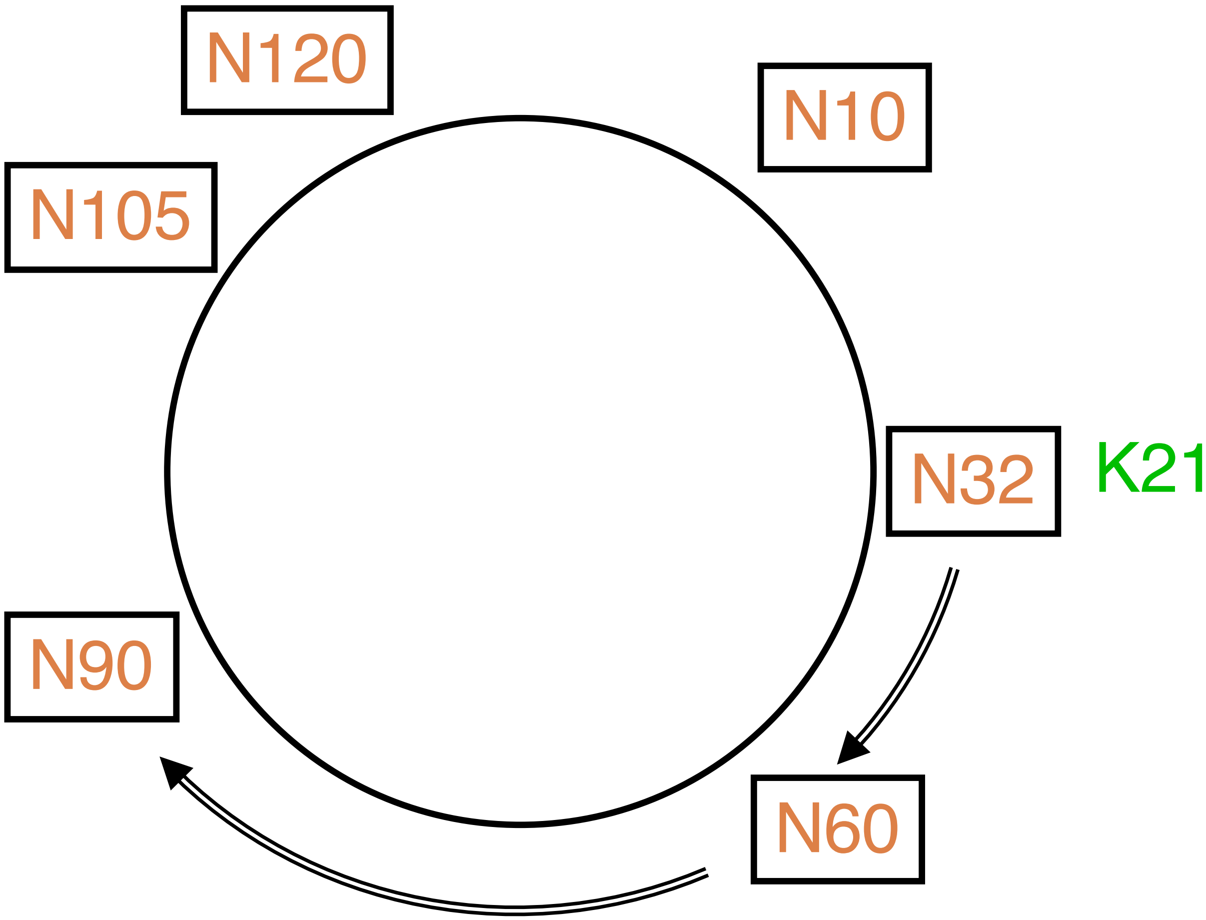
- Setting:
  - Tens of millions of customers
  - Data spread across tens of thousands of servers

- Example use case: Store shopping carts
- Goals:
  - High availability
  - Low latency
    - Consistency takes a hit

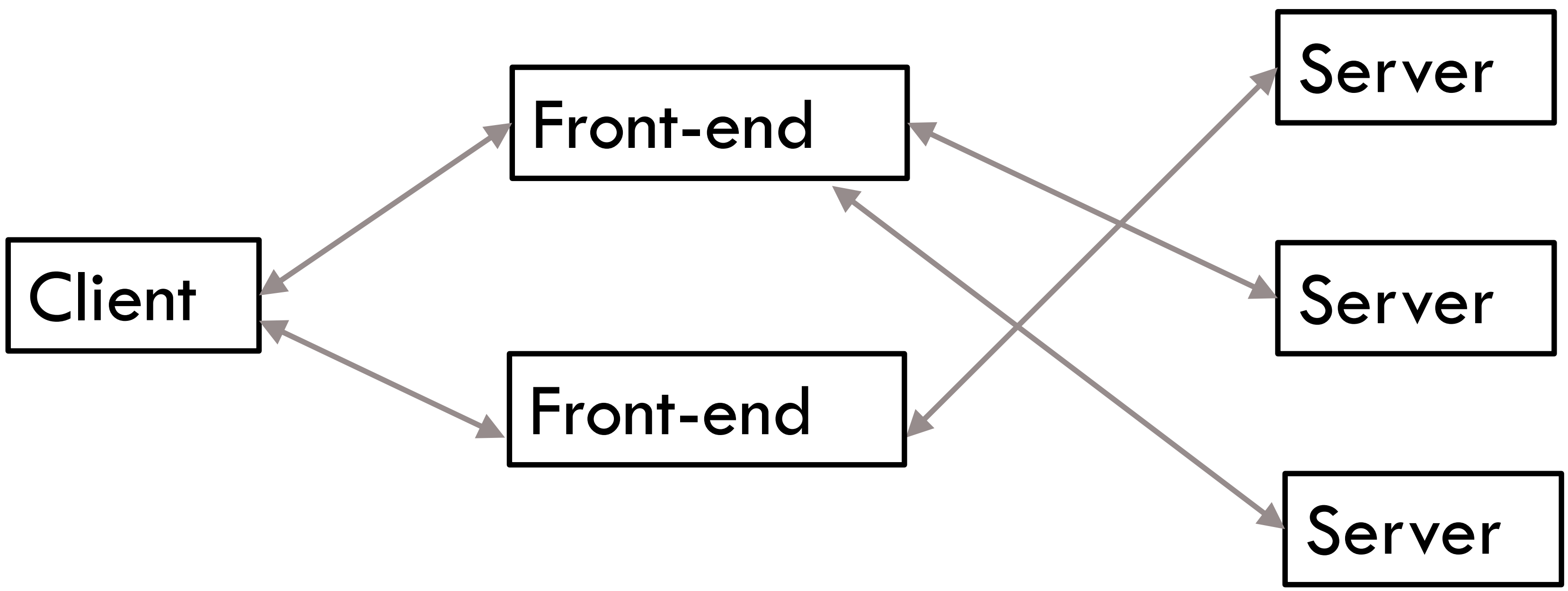
Consistent Hashing in Dynamo

- Recall: Consistent hashing maps value for key to successor in hash space
- Replicate value for every key at N nodes
  - N clockwise successors of key
- Execution of writes
  - Write received by coordinator (successor of key)
  - Coordinator forwards to successors

Replication in Dynamo

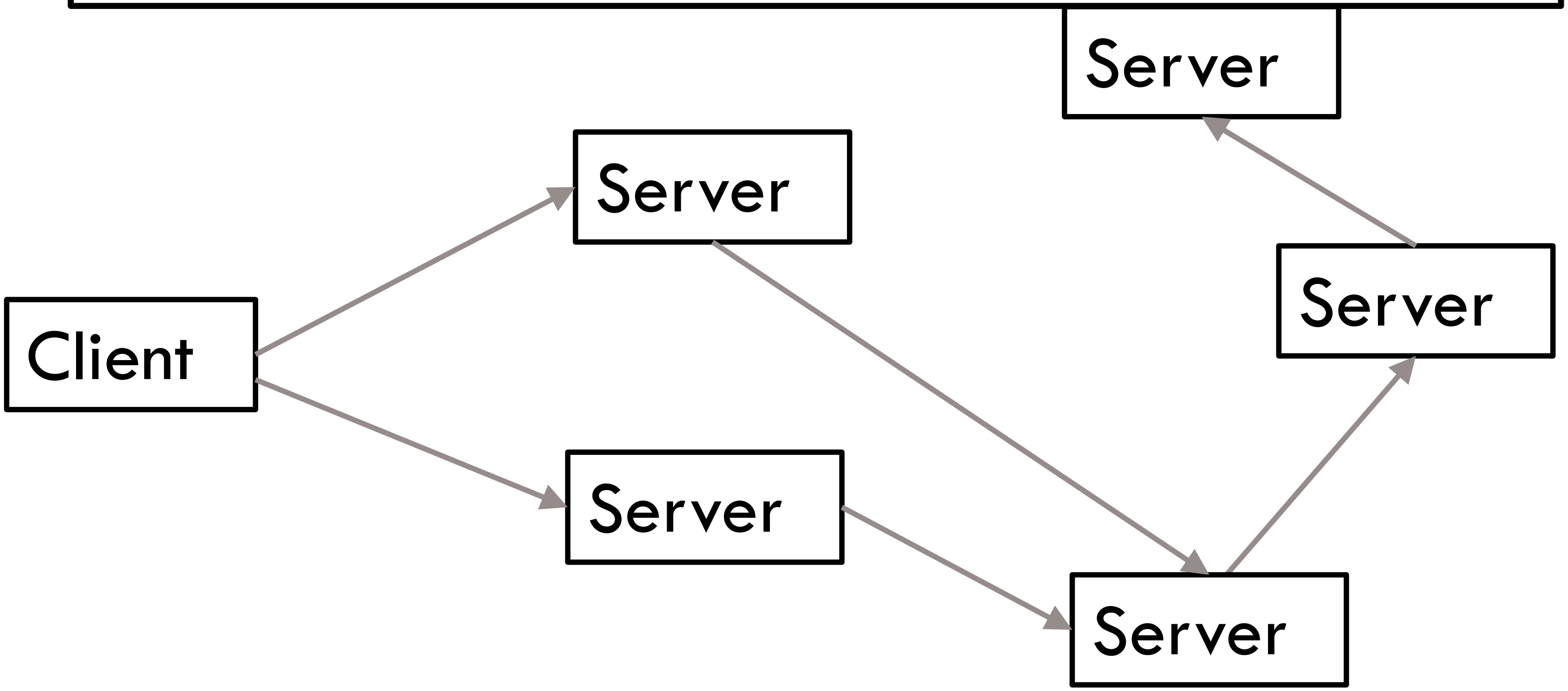


Using Consistent Hashing



Consistent Hashing in Dynamo

What would it take to make this work?



1-hop DHT

Gossip

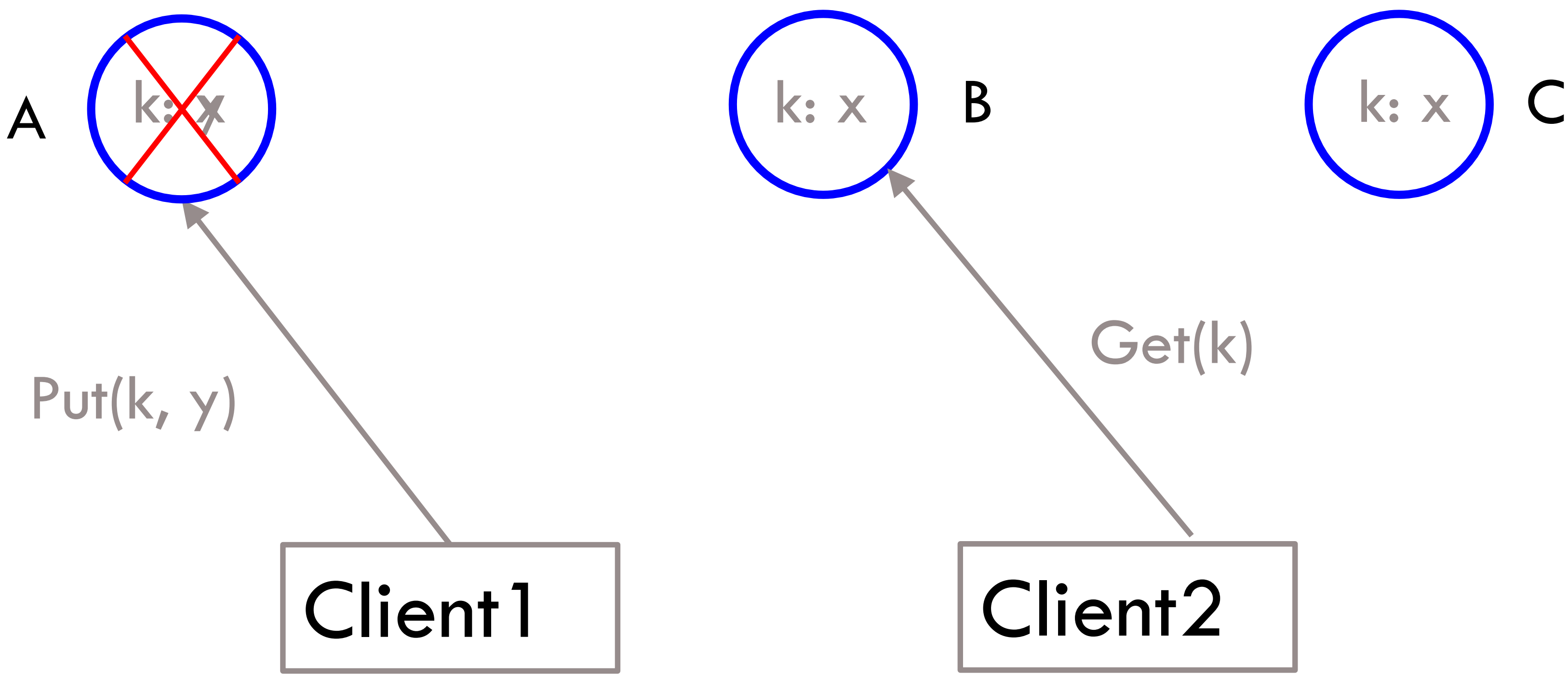
- Once per second, each server contacts a randomly chosen other server
- Servers exchange their lists of known servers
  - Including virtual node IDs

Sloppy quorums

- N replicas for every key
  - Higher durability with greater N
- Serving reads and writes:
  - Coordinator forwards request to first N-1 reachable successors
  - Waits for response from R or W to replicas
- How to maximize availability/minimize latency?
  - Low R and/or low W
- How to ensure read sees last committed write?
  - R+W > N

Latency/availability over consistency

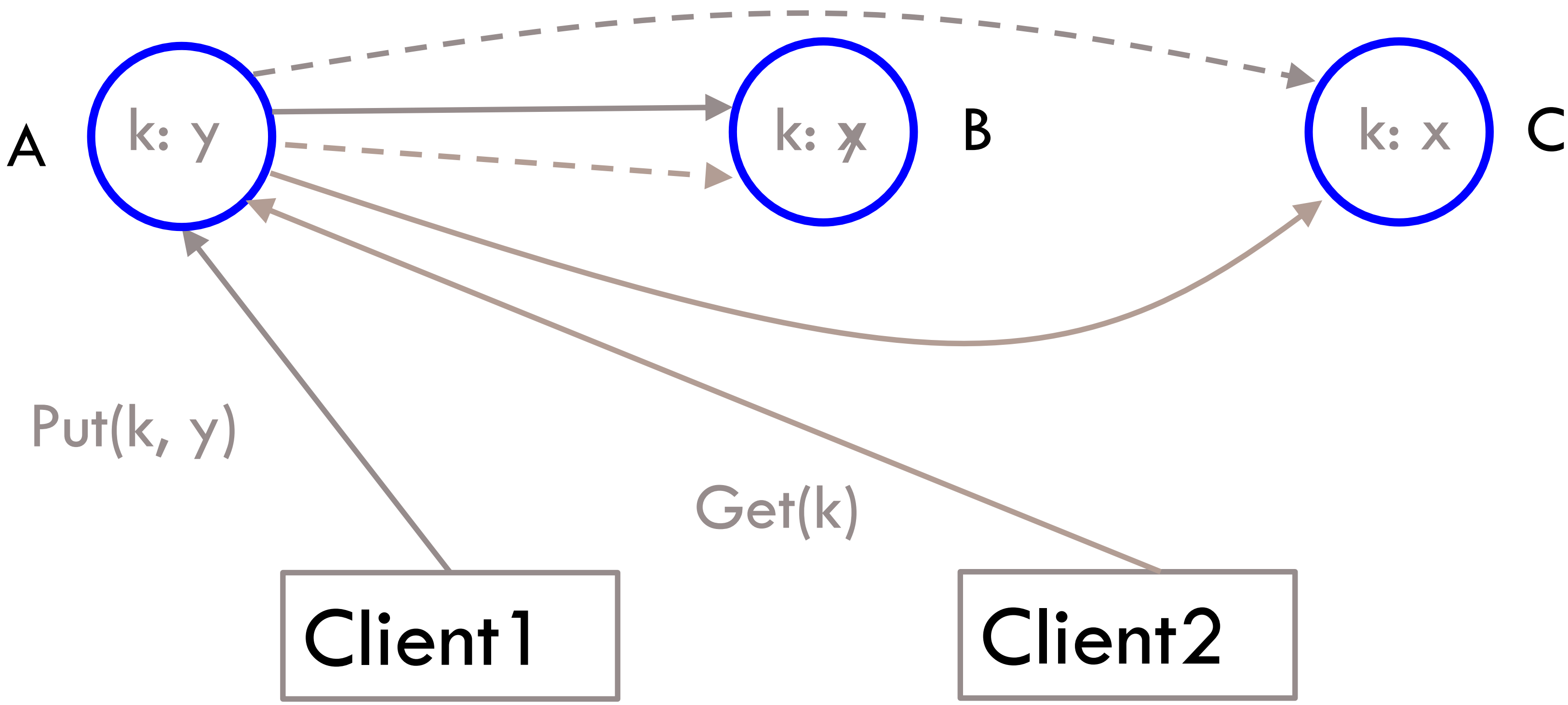
N = 3, W = 1, R = 1



Consistency over latency/availability

N = 3, W = 2, R = 2

How to tell which of R copies read is latest version?



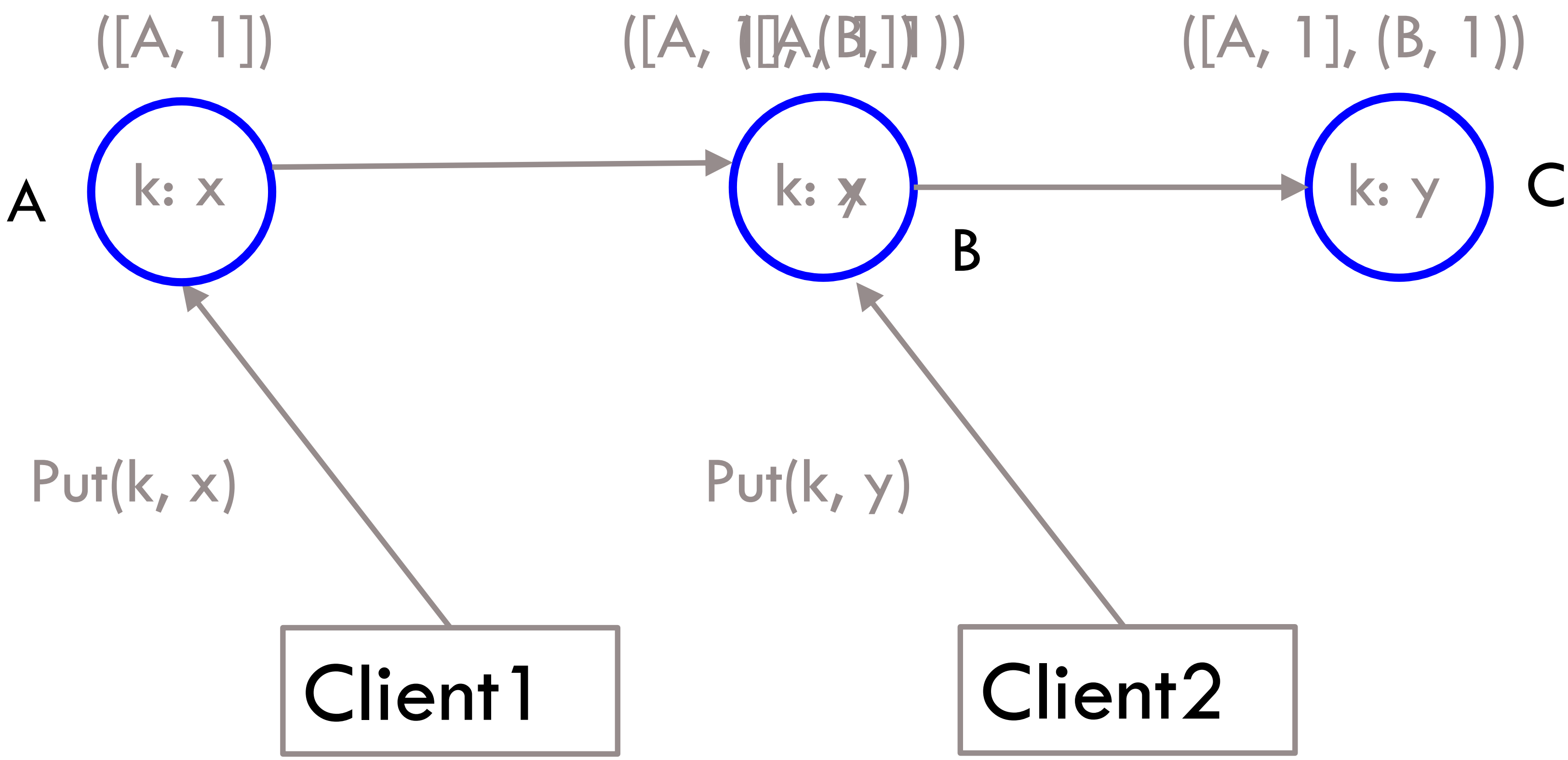
Vector clocks

- Store a vector clock with each key-value pair
- What we have discussed previously:
  - Vector with # of components = # of servers
  - Not scalable

- Dynamo's adaptation of vector clocks:
  - List of (coordinator node, counter) pairs
  - Example: [(A, 1), (B, 3), …]

Vector clocks

N = 3, W = 2, R = 2

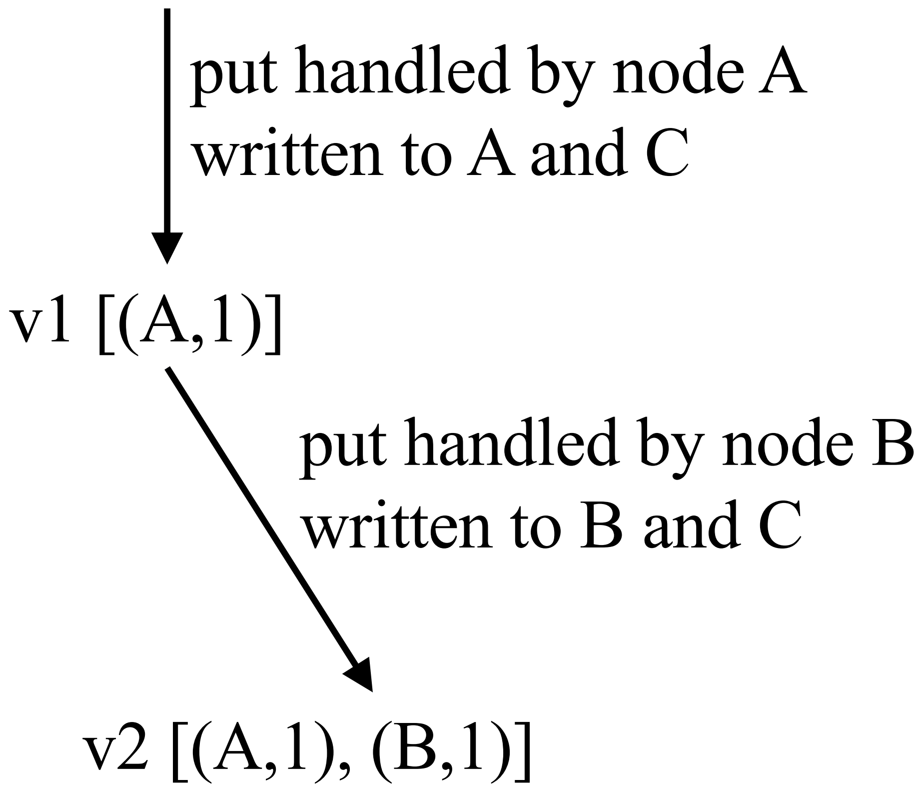


Vector clocks in Dynamo

- Consider following scenario:
  - Client1 executes PUT(k, v1)
  - Client2 executes GET(k) and gets v1
  - Client2 executes PUT(k, v2)
- How can vector clocks help in recognizing that okay to garbage collect v1?

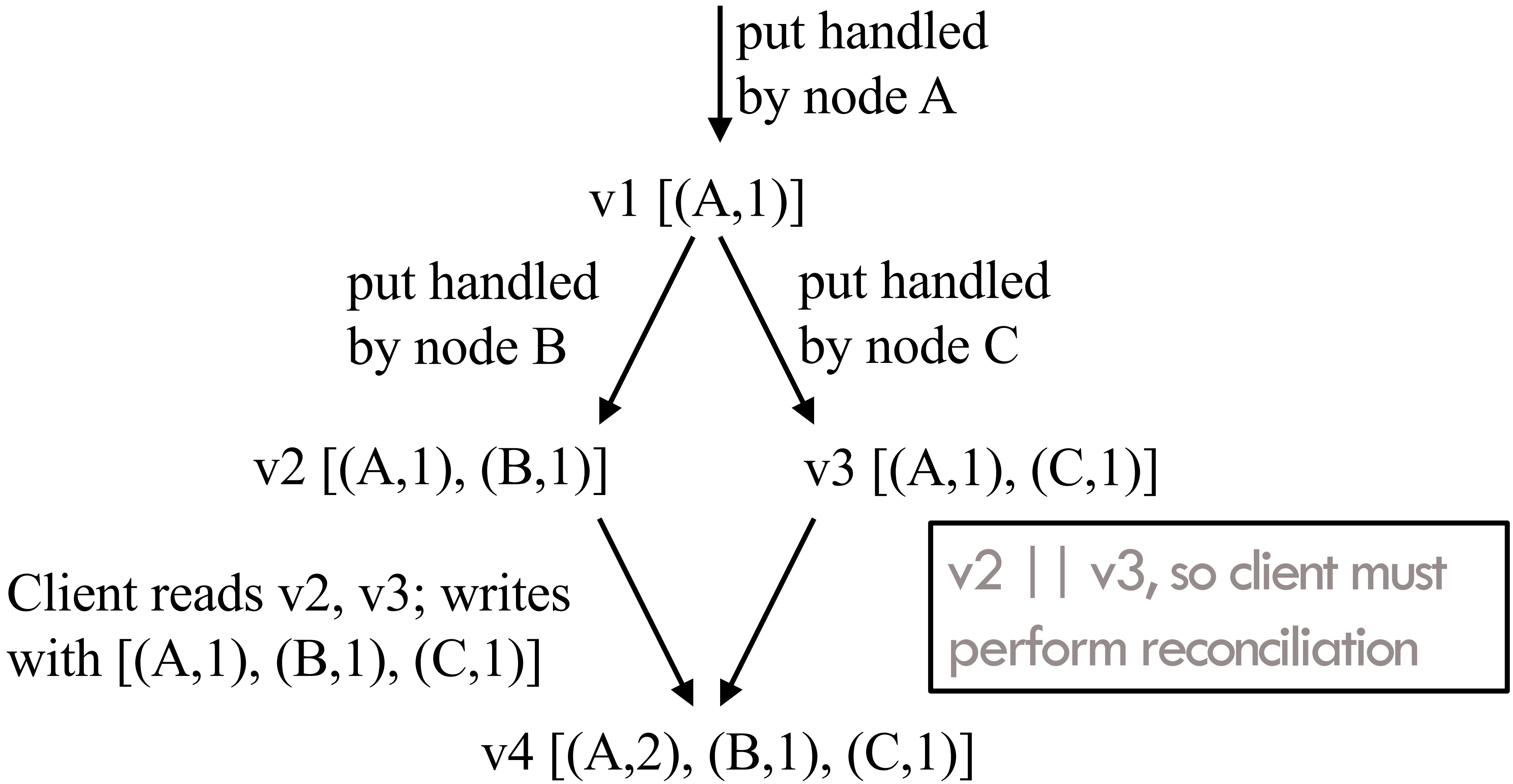
- When responding to a GET, Dynamo returns the vector clock for value returned
- Client includes vector clock in subsequent PUT

Automatic conflict resolution



v2 > v1, so Dynamo automatically drops v1 at C

App-specific conflict resolution



Dynamo's client interface

- Client interface:
  - Get(key) → value
  - Put(key, value)

- Get(key) → List of <value, context> pairs
  - Returns one value or multiple conflicting values
  - Context describes version(s) of value(s)
- Put(key, value, context)
  - Context indicates which versions this version supersedes or merges

Trimming version vectors

- Many nodes may process Puts to same key
  - Version vectors may grow arbitrarily long
- Dynamo's clock truncation scheme
  - Dynamo stores time of modification with each version vector entry
  - When version vector > 10 nodes long, Dynamo drops node that least recently processed key
- Problems with truncation?
  - False concurrency

Impact of clock truncation