Amazon Dynamo

CS 475: Concurrent & Distributed Systems (Fall 2021)

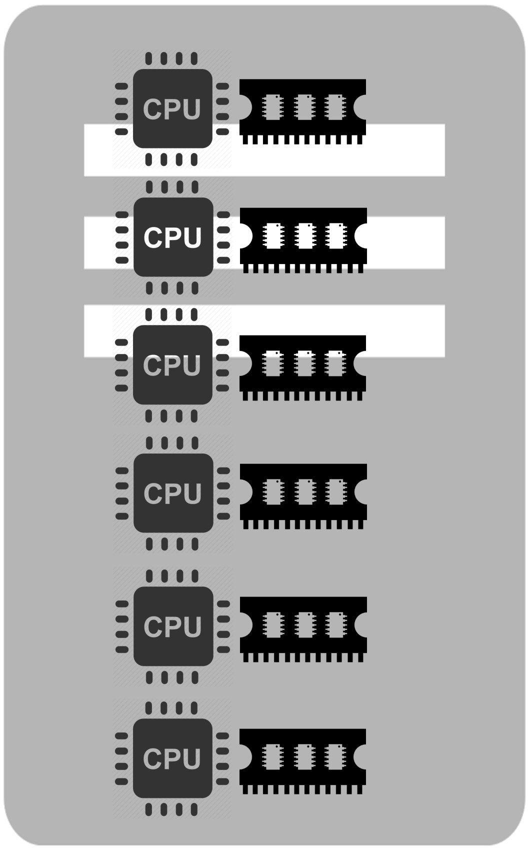
Lecture 13

Yue Cheng

Some material taken/derived from:

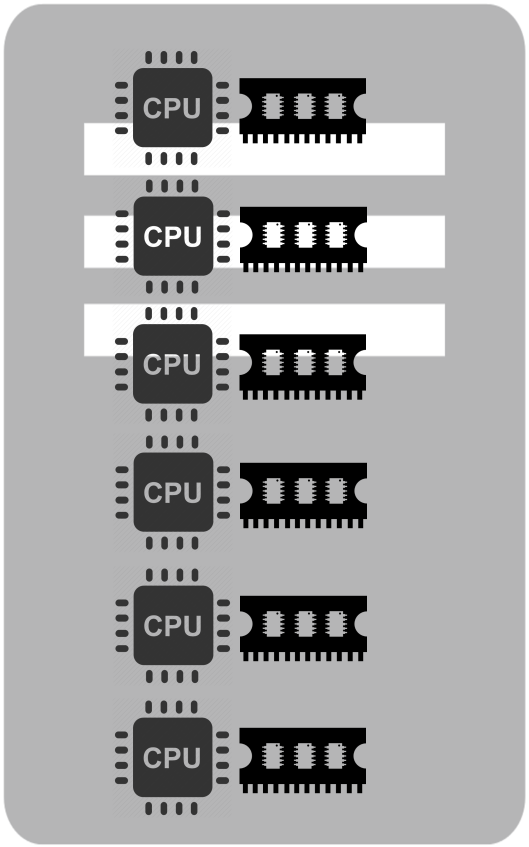
- Princeton COS-418 materials created by Michael Freedman.
- MIT 6.824 by Robert Morris, Frans Kaashoek, and Nickolai Zeldovich.

Horizontal or vertical scalability

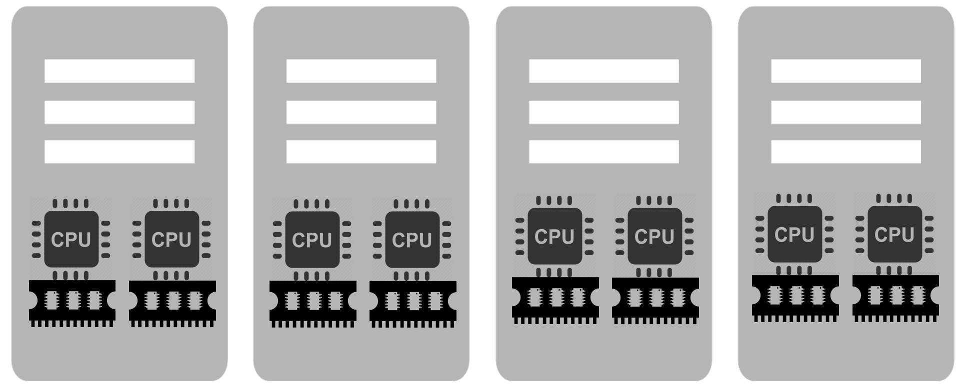


Vertical scaling
(Scaling-up)

Horizontal or vertical scalability



Vertical scaling
(Scaling-up)



Horizontal scaling
(Scaling-out)

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!

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

Outline

1. Techniques for partitioning data
   - Metrics for success

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
  - 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

Scaling out: Partition management

- Problem 2: Partition management
  - 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

Modulo hashing

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

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

Modulo hashing

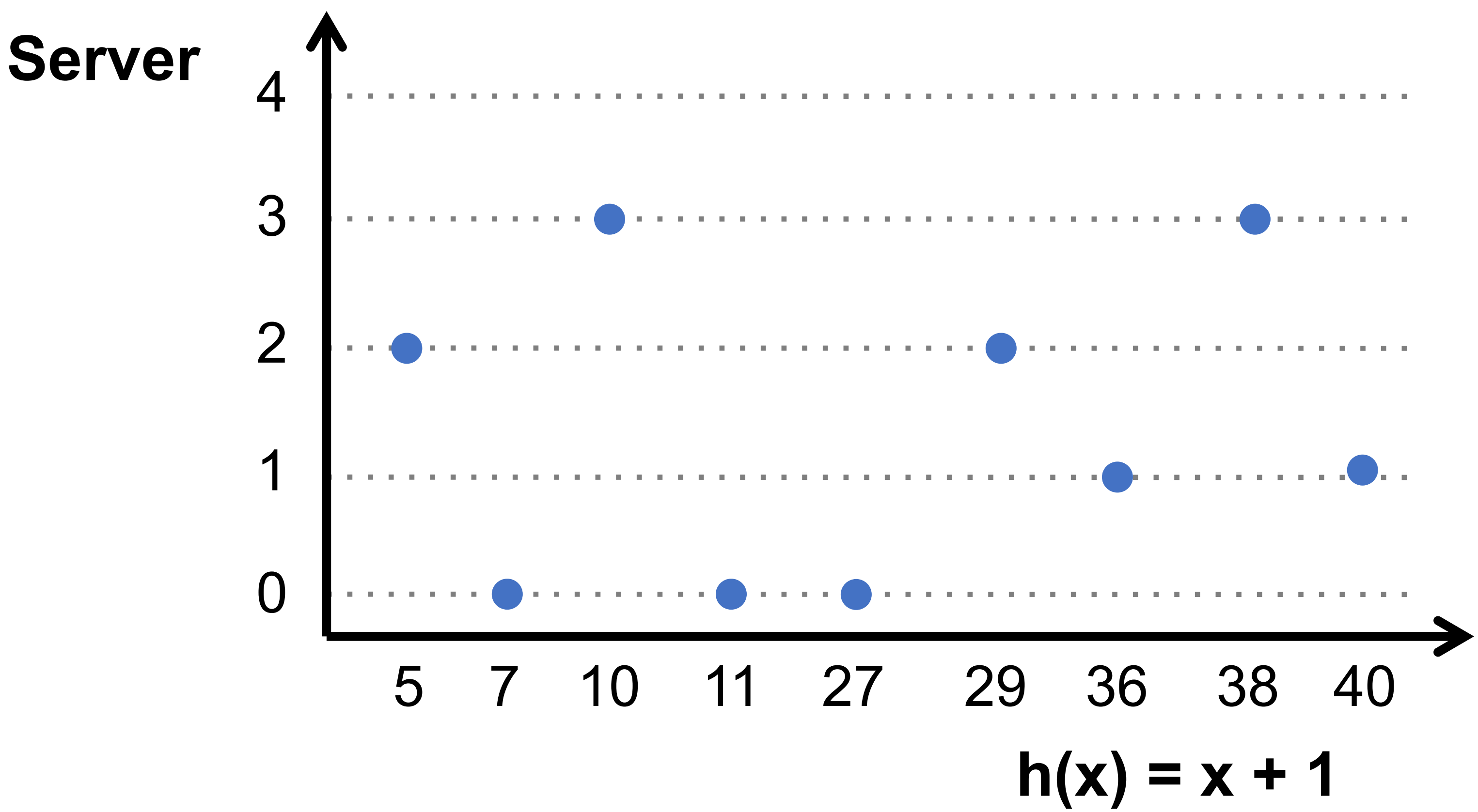
- First consider problem of data partition:
  - Given object id X, choose one of *k* servers to use

- Suppose we use modulo hashing:
  - Place *X* on server $i = \text{hash}(X) \bmod k$

- What happens if a server fails or joins ($k \leftarrow k \pm 1$)?
  - or different clients have different estimate of k?

Problem for modulo hashing: Changing number of servers

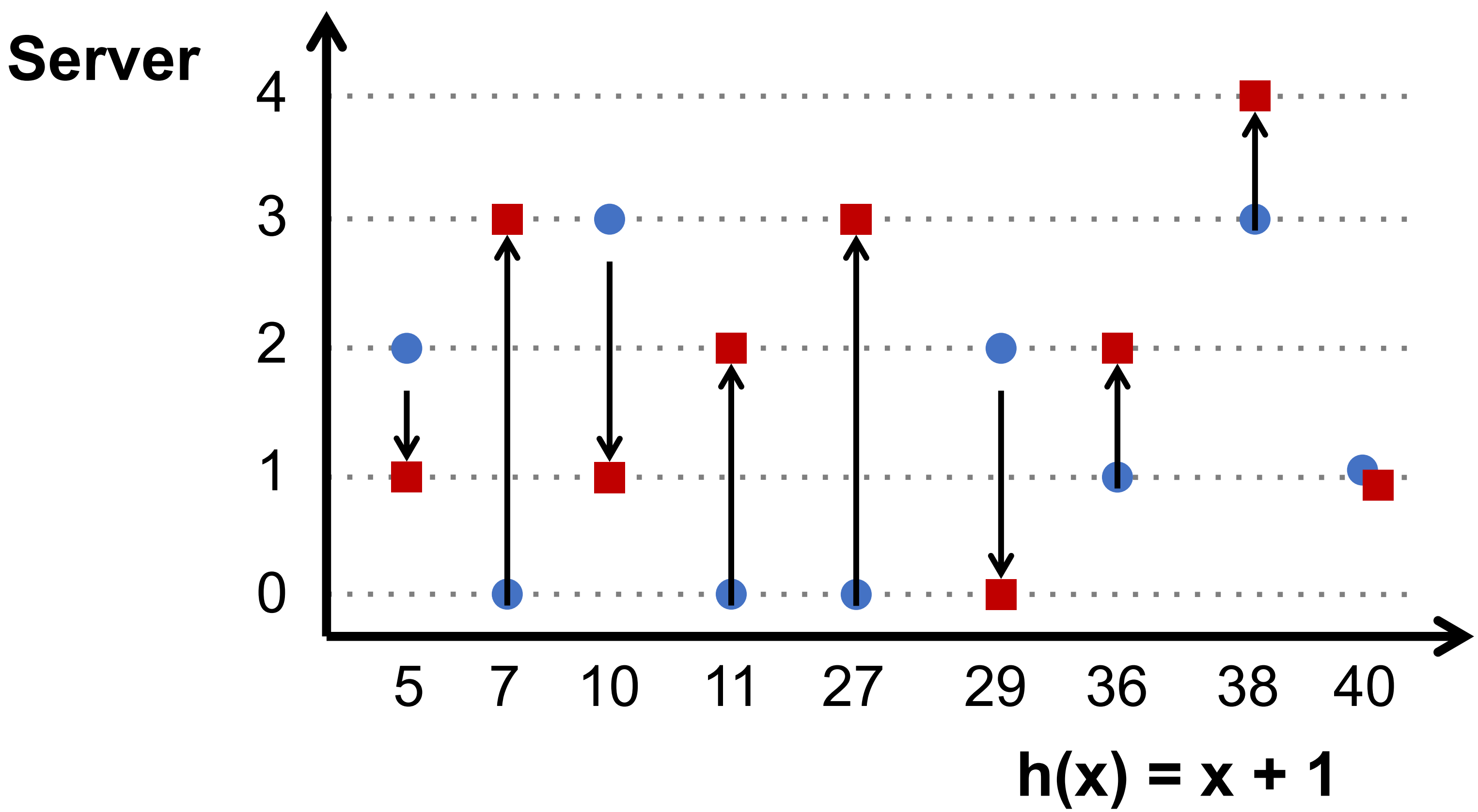
i = h(x) mod 4



Problem for modulo hashing: Changing number of servers

i = h(*x*) mod 4

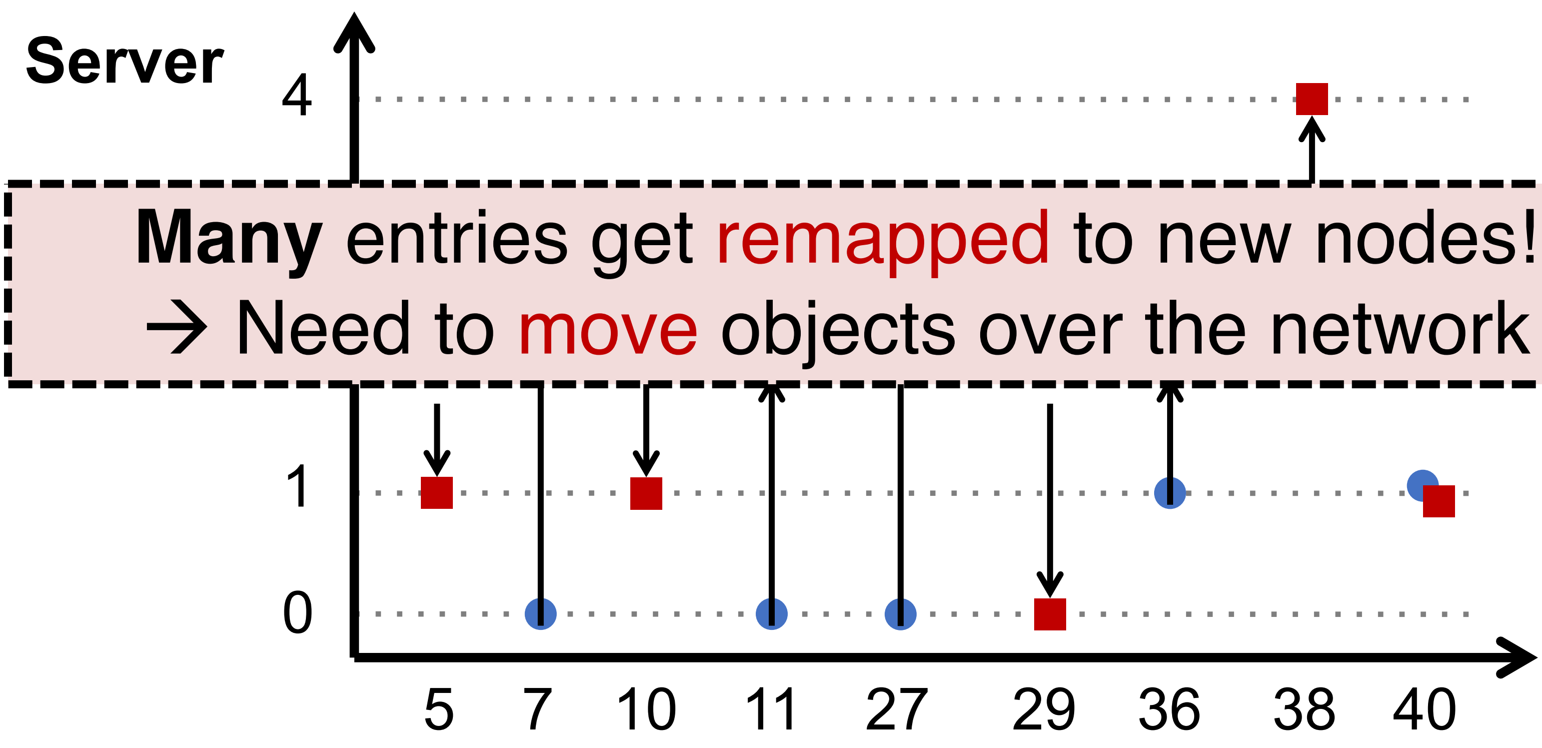
Add one machine: i = h(*x*) mod 5



Problem for modulo hashing: Changing number of servers

i = h(*x*) mod 4

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

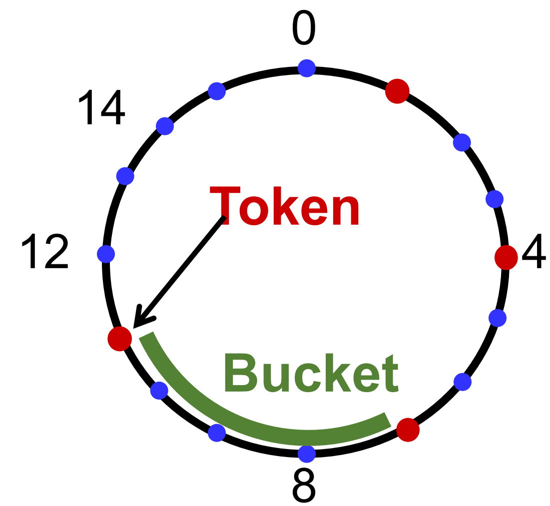


Many entries get remapped to new nodes!

→ Need to move objects over the network

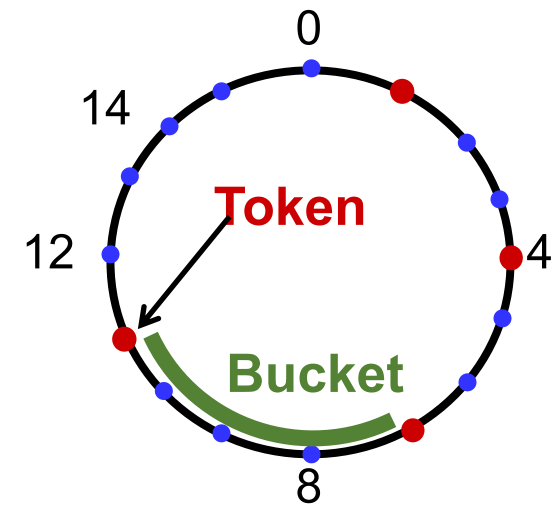
Consistent hashing

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



Consistent hashing

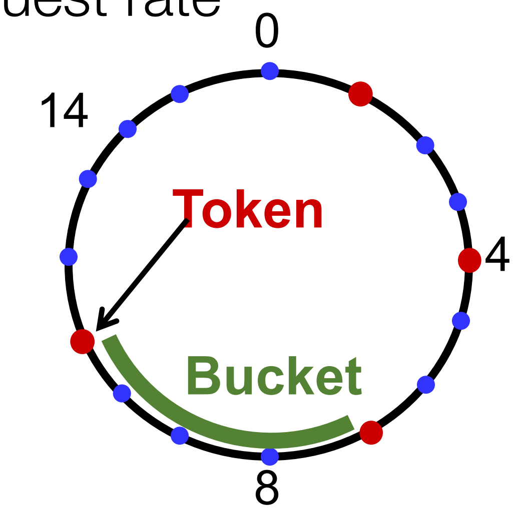
- Assign n *tokens* to random points on mod 2^k circle; hash key size = k
- Hash object to random circle position
- Put object to **closest clockwise** bucket
  - *successor* (key) → bucket



- **Desirable 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

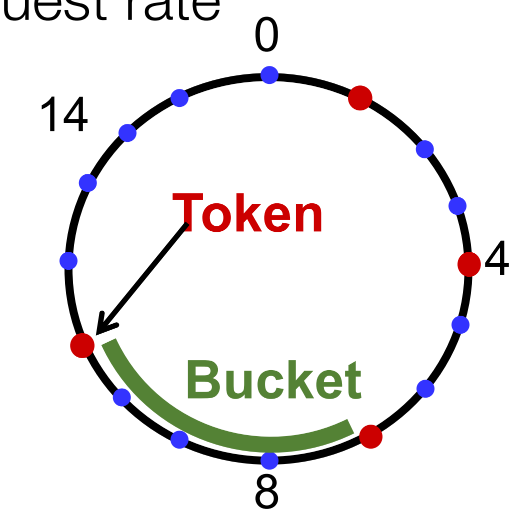
Consistent hashing's load balancing problem

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



Consistent hashing's load balancing problem

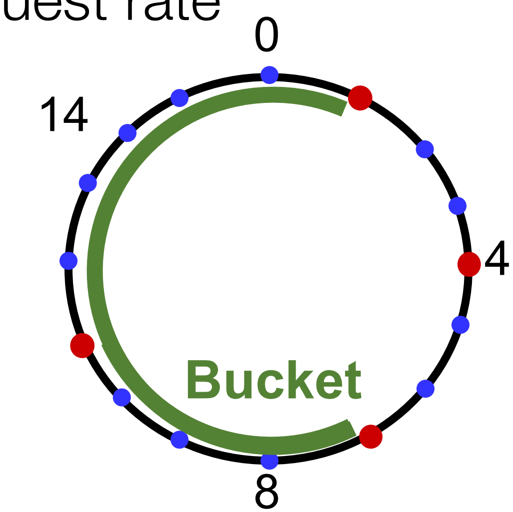
- Each node owns 1/nth of the ID space in expectation
  - Hot keys → some buckets have higher request rate



- 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

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

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)

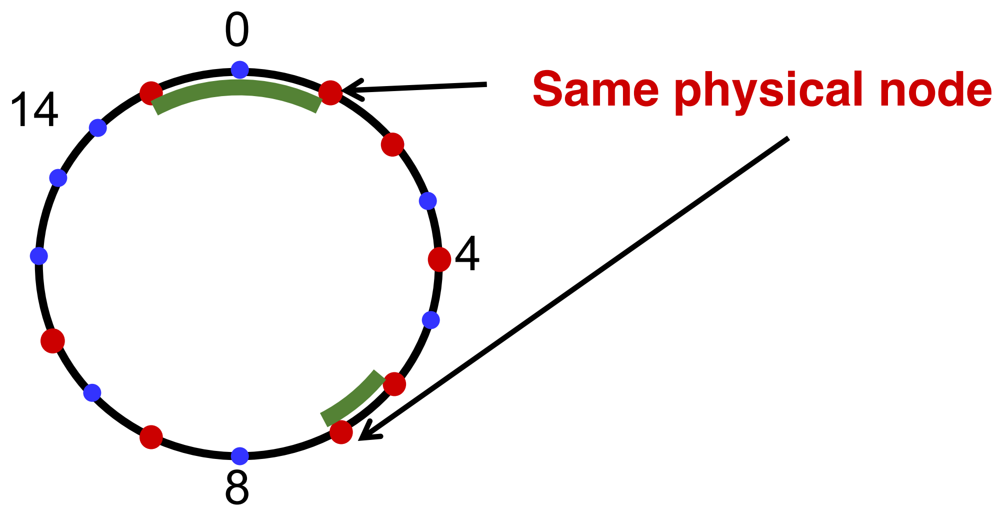
- Each virtual node owns an expected $1/(vn)^{th}$ of ID space

- 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

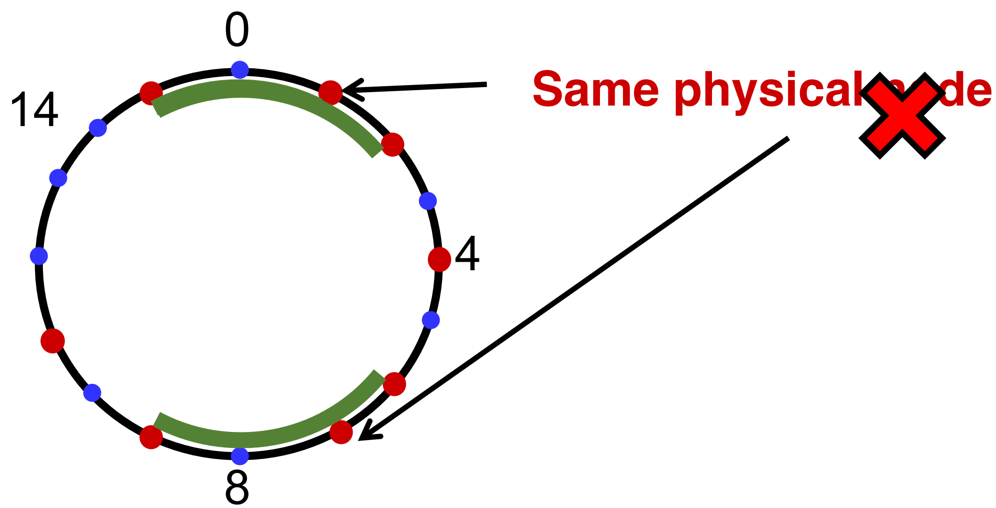
V=2



Virtual nodes: Example

4 Physical Nodes

V=2



Result: Better load balance with larger v

Outline

1. Techniques for partitioning data
   - Metrics for success

2. Case study
   - Amazon Dynamo key-value store

Dynamo: The P2P context

- Chord and DHash intended for wide-area P2P systems
  - Individual nodes at Internet's edge, file sharing

Dynamo: The P2P context

- Chord and DHash intended for wide-area P2P systems
  - Individual nodes at Internet's edge, file sharing

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

Dynamo: The P2P context

- **Chord** and **DHash** intended for **wide-area P2P systems**
  - Individual nodes at Internet's edge, file sharing

- 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

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” *etc.*?
- Product list
  - Mostly read-only, replication for high read throughput

How does Amazon use Dynamo?

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

Each instance contains **a few hundred** servers

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

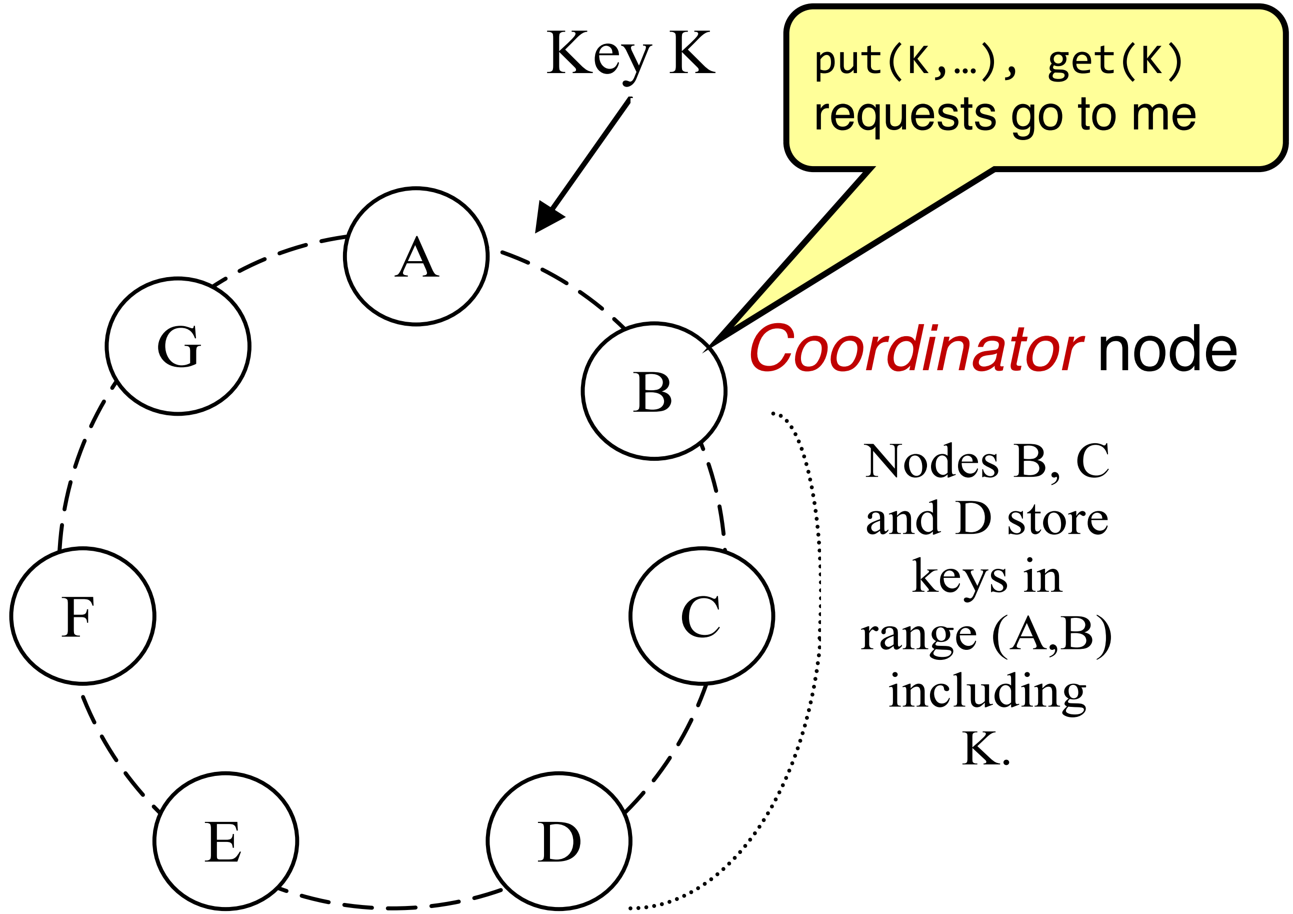
- Place replicated data on nodes with consistent hashing
- Maintain consistency of replicated data with vector clocks
  - Eventual consistency for replicated data: prioritize success and low latency of writes over reads
    - And availability over consistency (unlike DBs)
- Efficiently synchronize replicas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

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 tradeoffs: Response time vs. consistency vs. durability

Data placement



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

Data replication

- A key-value pair → key's *N* successors (*preference list*)
  - **Coordinator receives a put** for some key
  - Coordinator then replicates data onto nodes in the key's **preference list**

Data replication

- A key-value pair → key's *N* successors (*preference list*)
  - **Coordinator receives a put** for some key
  - Coordinator then replicates data onto nodes in the key's **preference list**

- Writes to more than just *N* successors in case of failure

Data replication

- A key-value pair → key's *N* successors (*preference list*)
  - **Coordinator receives a put** for some key
  - Coordinator then replicates data onto nodes in the key's **preference list**
- Writes to more than just *N* successors in case of failure
- For robustness, the preference list **skips tokens** to ensure distinct physical nodes

Gossip and lookup

- Gossip: Once per second, each node contacts a randomly chosen other node
    - They exchange their lists of known nodes (including virtual node IDs)
- Assumes all nodes will come back eventually, doesn't repartition
- Each node learns which others handle all key ranges

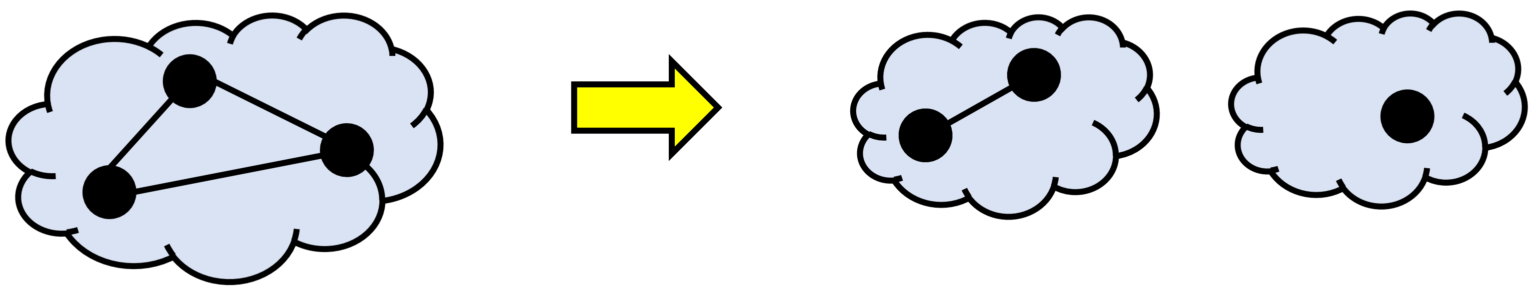
Gossip and lookup

- Gossip: Once per second, each node contacts a randomly chosen other node
  - They exchange their lists of known nodes (including virtual node IDs)
- Assumes all nodes will come back eventually, doesn't repartition
- Each node learns which others handle all key ranges

  - Result: All nodes can send directly to any key's coordinator ("zero-hop DHT")
    - Reduces variability in response times

Partitions force a choice between availability and consistency

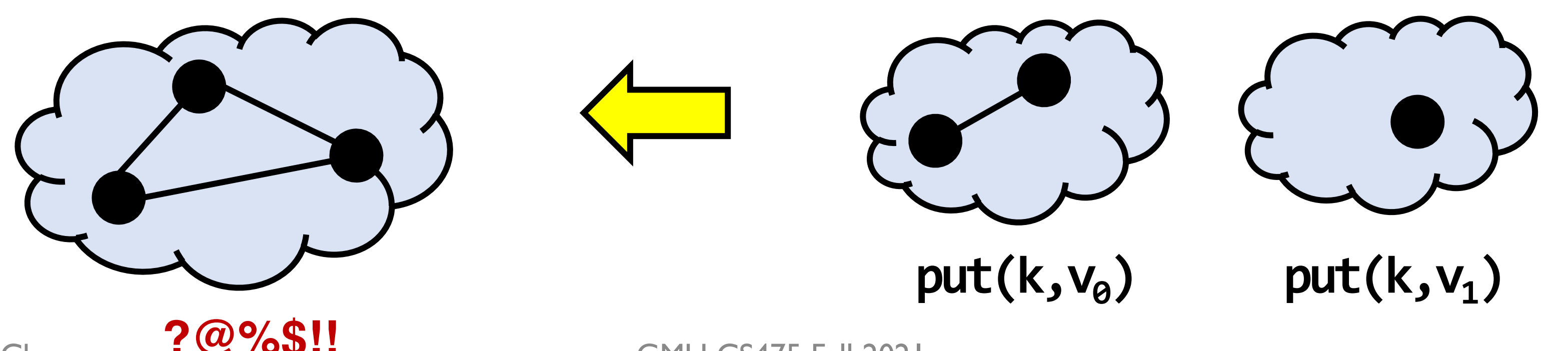
- Suppose three replicas are partitioned into two and one



- 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:



Mechanism: Sloppy quorums

- If no failure, reap “consistency” benefits of single master
  - Else sacrifice “consistency” to allow progress
- Dynamo tries to store all values put() under a key on first N live nodes of coordinator’s preference list

Mechanism: Sloppy quorums

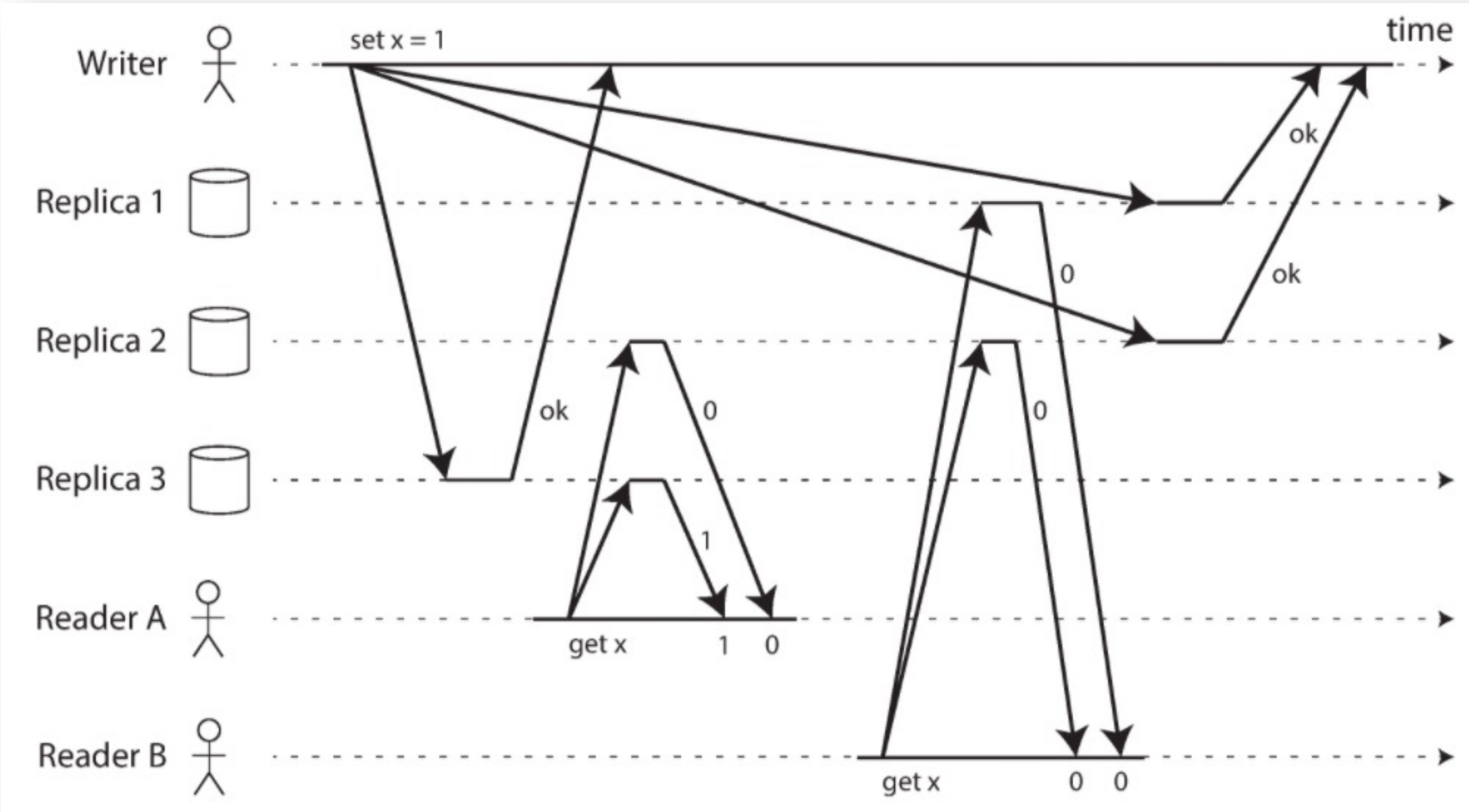
- If no failure, reap "consistency" benefits of single master
  - Else sacrifice "consistency" to allow progress

- Dynamo tries to store all values put() under a key on first N live nodes of coordinator's preference list

- BUT to speed up `get()` and `put()`:
  - Coordinator returns "`success`" for `put` when **W** < N replicas have completed **write**
  - Coordinator returns "`success`" for `get` when **R** < N replicas have completed **read**

Consistency under sloppy quorums != linearizability

Sloppy quorum of (N=3, W=3, R=2)



*: https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ (Page 334)

Sloppy quorums: Hinted handoff

- Suppose coordinator doesn't receive *W* replies when replicating a `put()`
  - Could return failure, but remember goal of high availability for writes…

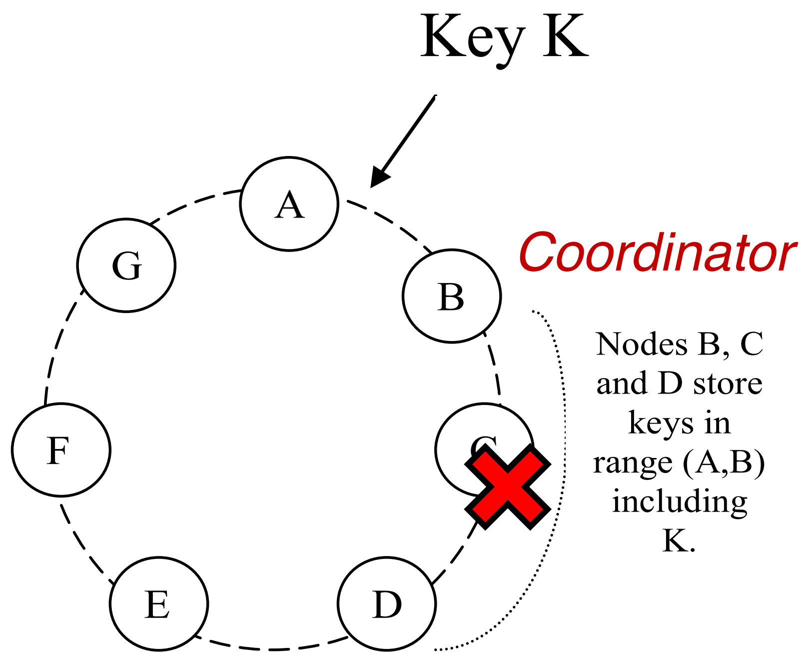
Sloppy quorums: Hinted handoff

- Suppose coordinator doesn't receive *W* replies when replicating a `put()`
  - Could return failure, but remember goal of high availability for writes…

- Hinted handoff: Coordinator **tries further nodes** in preference list (**beyond first *N***) if necessary
  - Indicates the **intended replica node** to recipient
  - **Recipient** will periodically try to forward to the **intended replica node**

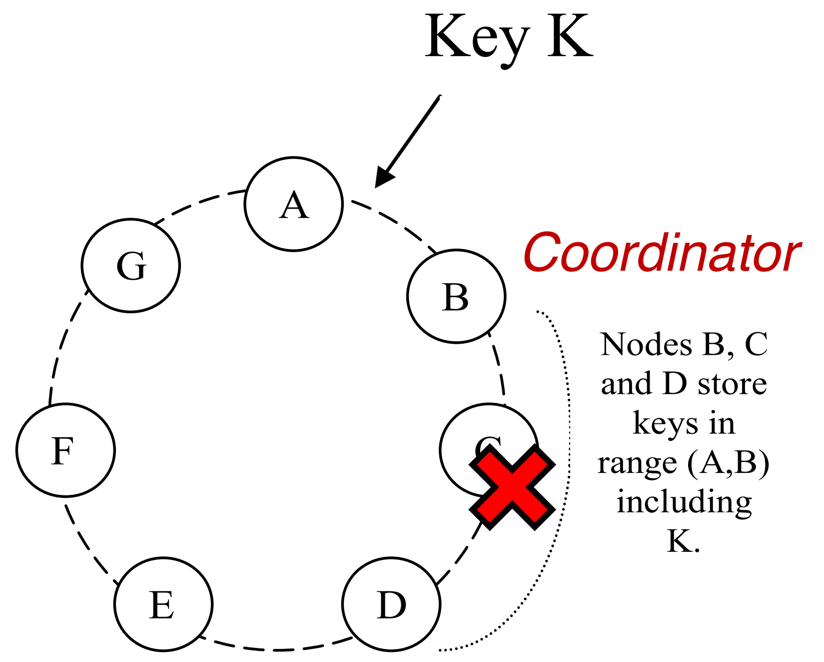
Hinted handoff: Example

- Suppose C fails
  - **Node E** is in **preference list**
    - Needs to receive replica of the data
  - Hinted Handoff: replica at **E** points to node **C**; **E** periodically forwards to **C**



Hinted handoff: Example

- Suppose C fails
  - **Node E** is in **preference list**
    - Needs to receive replica of the data
  - Hinted Handoff: replica at E points to node **C**; E periodically forwards to **C**



- When C comes back
  - E forwards the replicated data back to **C**

Wide-area replication

- Last ¶,§4.6: **Preference lists always** contain nodes from **more than one data center**
  - **Consequence:** Data likely to **survive failure** of **entire data center**

Wide-area replication

- Last ¶,§4.6: **Preference lists always** contain nodes from **more than one data center**
  - **Consequence:** Data likely to **survive failure** of **entire data center**

- Blocking on **writes to a remote data center** would incur unacceptably high latency
  - **Compromise:** $W < N$, eventual consistency
  - Better **durability**, **latency** but worse **consistency**

Sloppy quorums and `get()`s

- Suppose coordinator doesn't receive *R* replies when processing a **`get()`**
  - Penultimate ¶,§4.5: "*R* is the min. number of nodes that must participate in a successful read operation."
    - Sounds like these **`get()s`** fail

- Why not return whatever data was found, though?
  - As we will see, consistency not guaranteed anyway…

Sloppy quorums and freshness

- Common case given in paper: N = 3; R = W = 2
  - With these values, do sloppy quorums guarantee a **`get()`** sees all prior **`put()s`**?

Sloppy quorums and freshness

- Common case given in paper: N = 3; R = W = 2
  - With these values, do sloppy quorums guarantee a **`get()`** sees all prior **`put()`s**?

- If no failures, yes:
  - Two writers saw each **`put()`**
  - Two readers responded to each **`get()`**

Sloppy quorums and freshness

- Common case given in paper: N = 3; R = W = 2
  - With these values, do sloppy quorums guarantee a `get()` sees all prior `put()`s?

- If no failures, yes:
  - Two writers saw each `put()`
  - Two readers responded to each `get()`
  - Write and read quorums must overlap!

Sloppy quorums and freshness

- Common case given in paper: N = 3; R = W = 2
  - With these values, do sloppy quorums guarantee a `get()` sees all prior `put()`s?

- With node failures, no:
  - Two nodes in preference list go down
    - `put()` replicated outside preference list; Hinted handoff nodes have data

  - Two nodes in preference list come back up
    - `get()` occurs before they receive prior `put()`

Conflicts

- Suppose N = 3, W = R = 2, nodes are named A, B, C
  - 1st `put(k, …)` completes on A and B
  - 2nd `put(k, …)` completes on B and C
  - Now `get(k)` arrives, completes first at A and C

Conflicts

- Suppose N = 3, W = R = 2, nodes are named A, B, C
  - 1st `put(k, …)` completes on A and B
  - 2nd `put(k, …)` completes on B and C
  - Now `get(k)` arrives, completes first at A and C

- Conflicting results from A and C
  - Each has seen a different put(k, …)

Conflicts

- Suppose N = 3, W = R = 2, nodes are named A, B, C
  - 1st `put(k, …)` completes on A and B
  - 2nd `put(k, …)` completes on B and C
  - Now `get(k)` arrives, completes first at A and C

- Conflicting results from A and C
  - Each has seen a different put(k, …)

- Dynamo returns both results; what does client do now?

Version vectors (vector clocks)

- *Version vectors:* List of (**coordinator node, counter**) pairs
  - *e.g.*, [(A, 1), (B, 3), …]

- Dynamo stores a version vector with **each stored** key-value **pair**

- Tracks causal relationship between different versions of data stored under the same key **k**

Version vectors (VV) in Dynamo

- **Rule:** If vector clock comparison of v1 < v2, then the first is an ancestor of the second – Dynamo can forget v1

- Each time a `put()` occurs, Dynamo increments the counter in the V.V. for the coordinator node

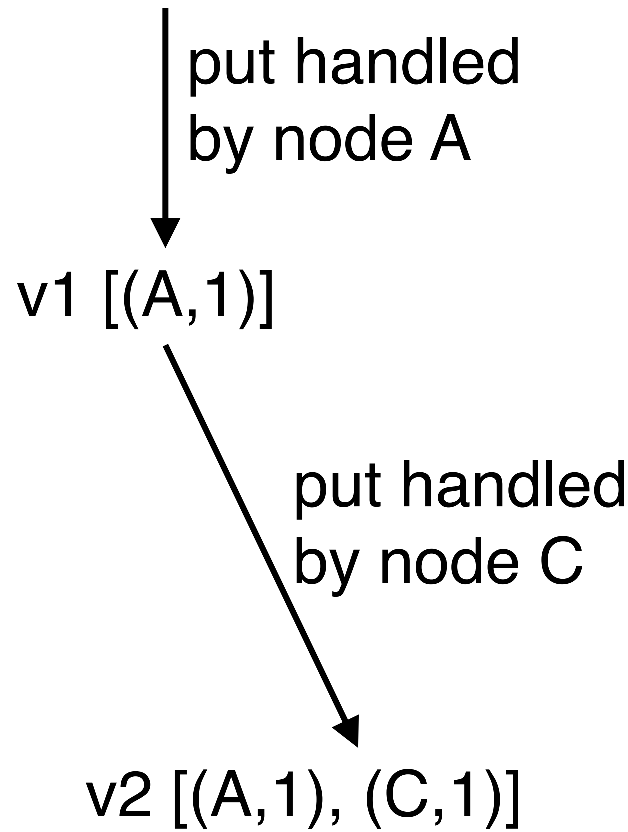
- Each time a `get()` occurs, Dynamo returns the V.V. for the value(s) returned (in the "context")
  - Then users must supply that context to `put()`s that modify the same key

Version vectors (auto-resolving case)

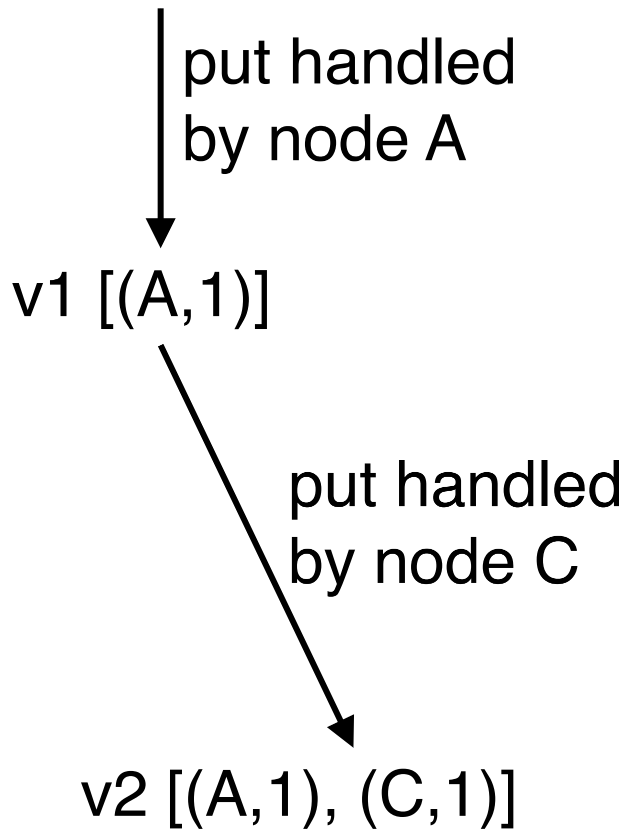
put handled
by node A

v1 [(A,1)]

Version vectors (auto-resolving case)

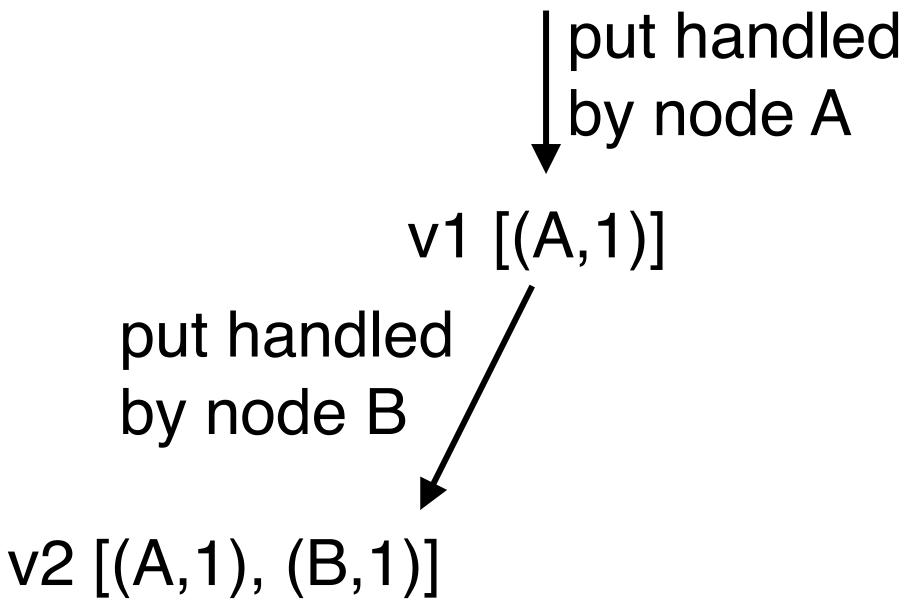


Version vectors (auto-resolving case)

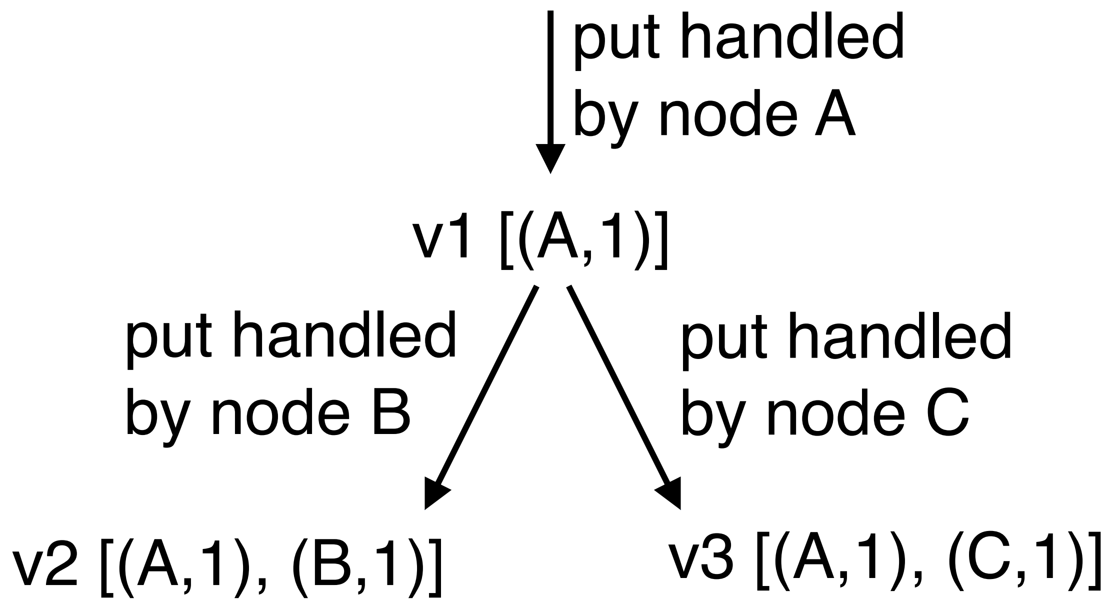


v2 > v1, so Dynamo nodes automatically drop v1, for v2

Version vectors (app-resolving case)



Version vectors (app-resolving case)



Version vectors (app-resolving case)

put handled
by node A

v1 [(A,1)]

put handled
by node B

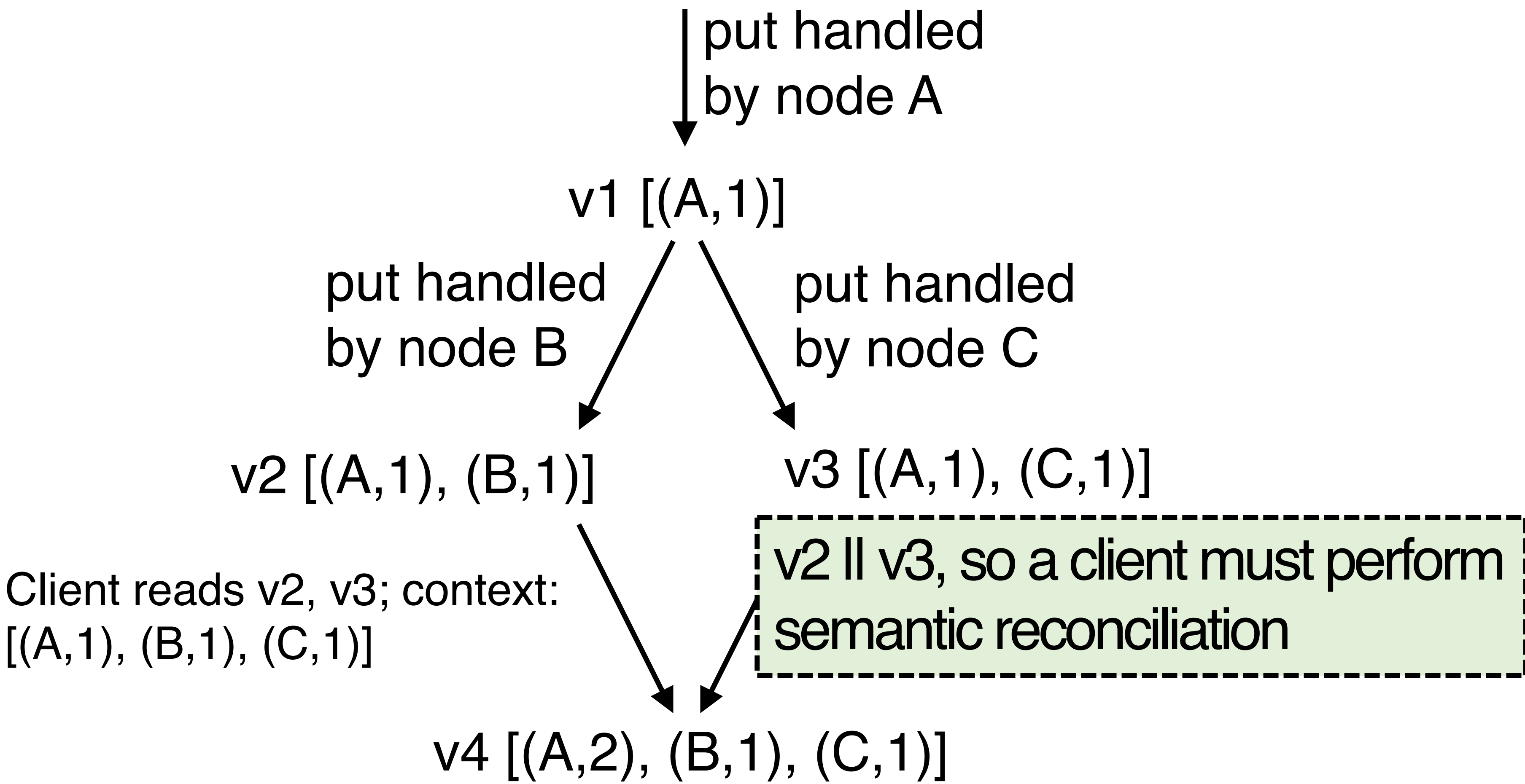
put handled
by node C

v2 [(A,1), (B,1)]

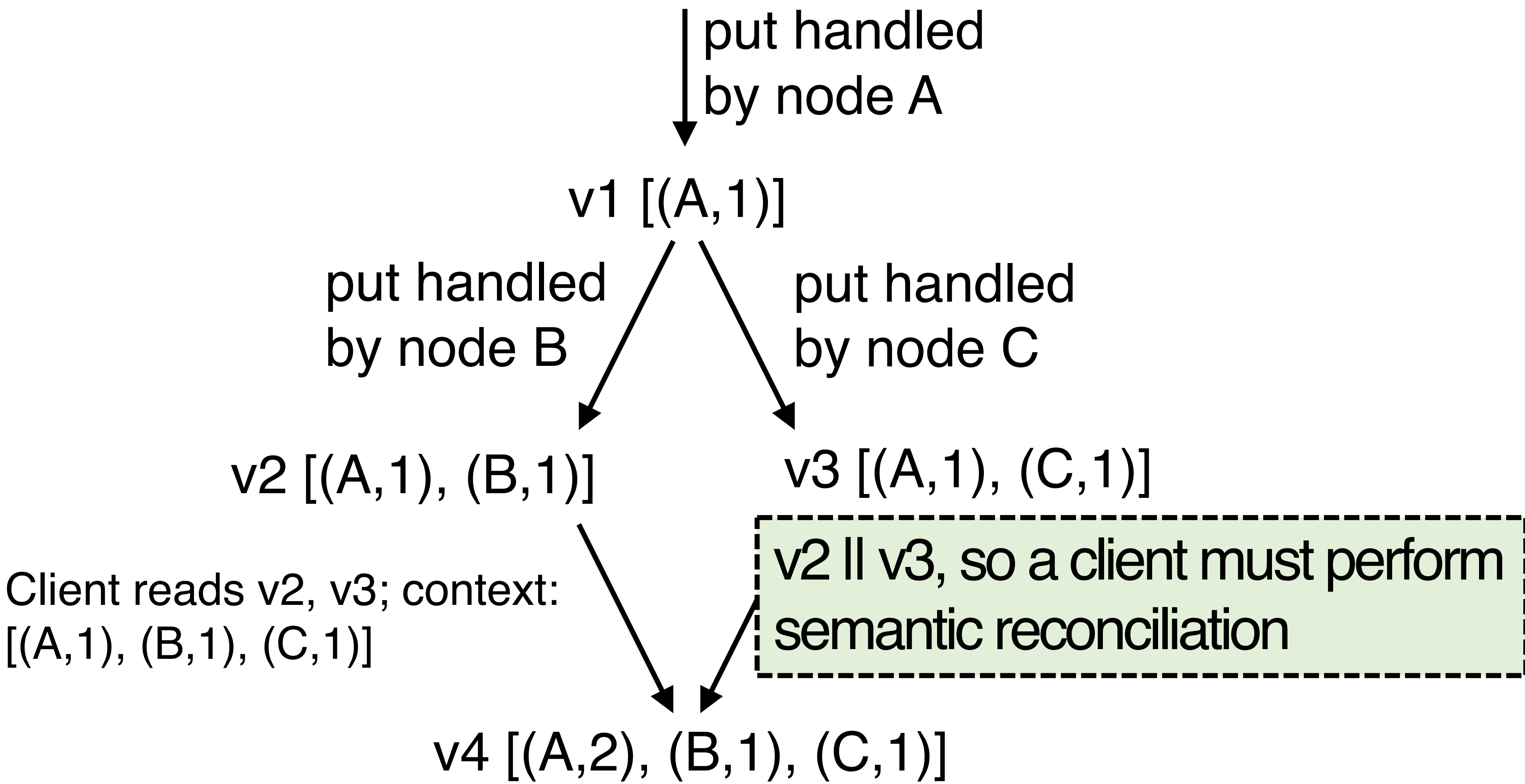
v3 [(A,1), (C,1)]

v2 || v3, so a client must perform semantic reconciliation

Version vectors (app-resolving case)



Version vectors (app-resolving case)



Client reconciles v2 and v3; node A handles the put

Trimming version vectors

- Many nodes may process a series of `put()`s to same key
  - Version vectors may get long – do they grow forever?

Trimming version vectors

- Many nodes may process a series of `put()`s to same key
  - Version vectors may get long – do they grow forever?
  - In practice, unlikely: unless failures, upper limit of N

Trimming version vectors

- Many nodes may process a series of `put()`s to same key
  - Version vectors may get long – do they grow forever?
  - In practice, unlikely: unless failures, upper limit of N

- Dynamo also uses a clock truncation scheme
  - Stores time of modification with each V.V. entry
  - When V.V. > 10 nodes long, V.V. drops the timestamp of the node that least recently processed that key

Impact of deleting a VV entry

put handled
by node A

v1 [(A,1)]

Impact of deleting a VV entry

put handled
by node A

v1 [(A,1)]

put handled
by node C

v2 [~~(A,1),~~ (C,1)]

Impact of deleting a VV entry

put handled
by node A

v1 [(A,1)]

put handled
by node C

v2 [~~(A,1),~~ (C,1)]

v2 || v1, so looks like application resolution is required

Concurrent writes

- What if two clients concurrently write w/o failure?
  - *e.g.* add **different items** to same cart at **same time**
  - Each does `get-modify-put`
  - They both see the same initial version
    - And they both send `put()` to same coordinator

- Will coordinator create two versions with conflicting VVs?

Concurrent writes

- What if two clients concurrently write w/o failure?
  - *e.g.* add **different items** to same cart at **same time**
  - Each does **`get-modify-put`**
  - They both see the same initial version
    - And they both send **`put()`** to same coordinator

- Will coordinator create two versions with conflicting VVs?
  - We want that outcome, otherwise one was thrown away
  - Paper doesn't say, but coordinator could detect problem via **`put()`** context

Removing threats to durability

- Hinted handoff node crashes before it can replicate data to node in preference list
  - Need another way to ensure that each key-value pair is replicated N times

Removing threats to durability

- Hinted handoff node crashes before it can replicate data to node in **preference list**
    - Need another way to ensure that each key-value pair is replicated N times

- **Mechanism:** replica synchronization
    - Nodes nearby on ring periodically gossip
        - Compare the (k, v) pairs they hold
        - Copy any missing keys the other has

Removing threats to durability

- Hinted handoff node crashes before it can replicate data to node in preference list
  - Need another way to ensure that each key-value pair is replicated N times

- Mechanism: replica synchronization
  - Nodes nearby on ring periodically gossip
    - Compare the (k, v) pairs they hold
    - Copy any missing keys the other has

How to compare and copy replica state quickly and efficiently?

Efficient synchronization with Merkle trees

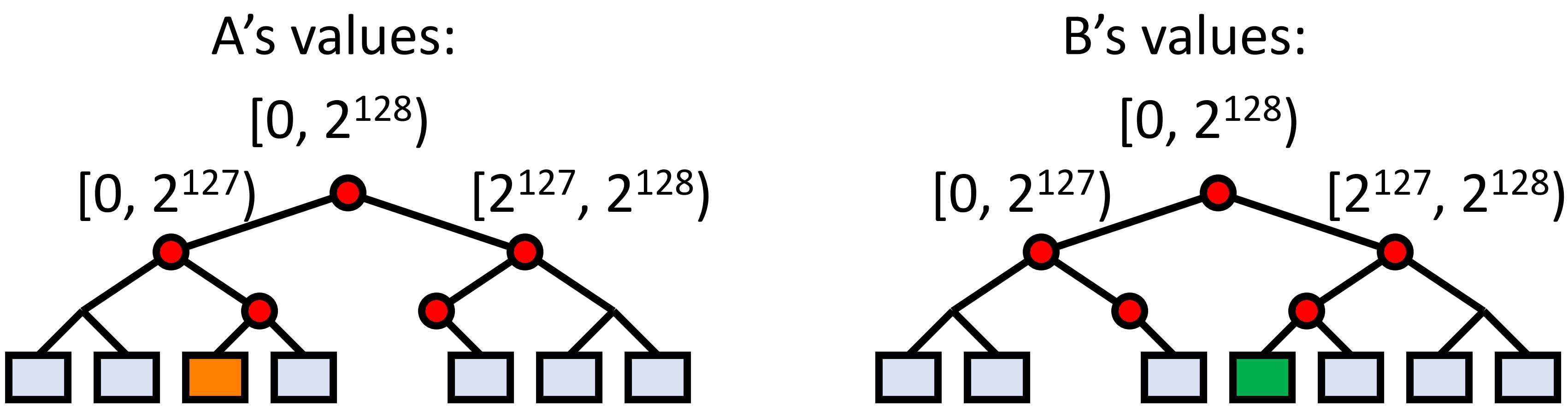
- Merkle trees hierarchically summarize the key-value pairs a node holds

- One Merkle tree for each virtual node key range
  - Leaf node = hash of one key's value
  - Internal node = hash of concatenation of children

- Compare roots; if match, values match
  - If they don't match, compare children
    - Iterate this process down the tree

Merkle tree reconciliation

- B is missing orange key; A is missing green one
- Exchange and compare hash nodes from root downwards, pruning when hashes match



Finds differing keys quickly and with minimum information exchange

How useful is it to vary N, R, W?

N	R	W	Behavior
3	**2**	**2**	**Parameters from paper:** **Good durability, good R/W latency**
3	3	1	Slow reads, **weak durability,** **fast writes**
3	1	3	**Slow writes,** strong durability, fast reads
3	3	3	More likely that **reads see all prior writes**?
3	1	1	Read quorum **doesn't overlap** write quorum

Dynamo: Take-aways

- 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
- **Version vectors** allow some conflicts to be resolved automatically; others left to application