

Dynamo: Amazon's Highly Available Key-value Store

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

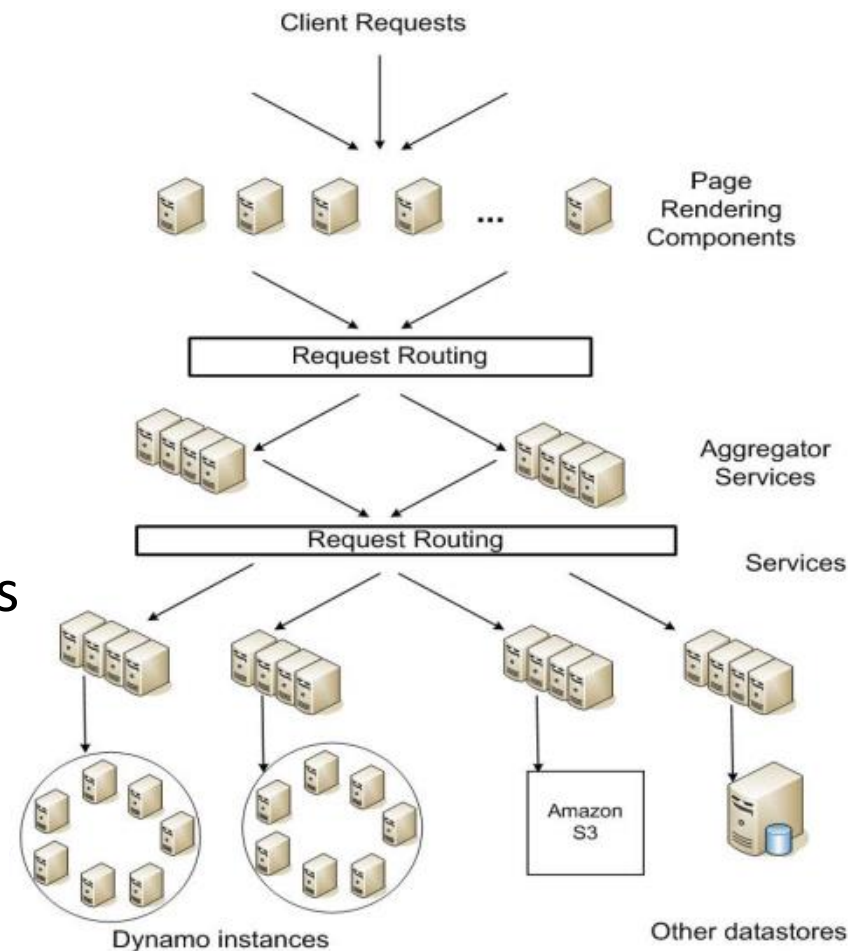
ECE1724

Authors: Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Many slides adapted from a talk by Peter Vosshall

Amazon's eCommerce Platform Architecture

- Loosely coupled, service-oriented architecture
- Stateful services manage their own state
- Stringent latency requirements
 - Services must adhere to formal SLAs
 - Measured at 99.9 percentile
- Availability is paramount
- Large scale, keeps growing
 - 10,000s servers worldwide



How does Amazon use Dynamo?

- Shopping cart
- Session information
 - E.g., recently visited products
- Product list
 - Mostly read-only, replicated for high read throughput

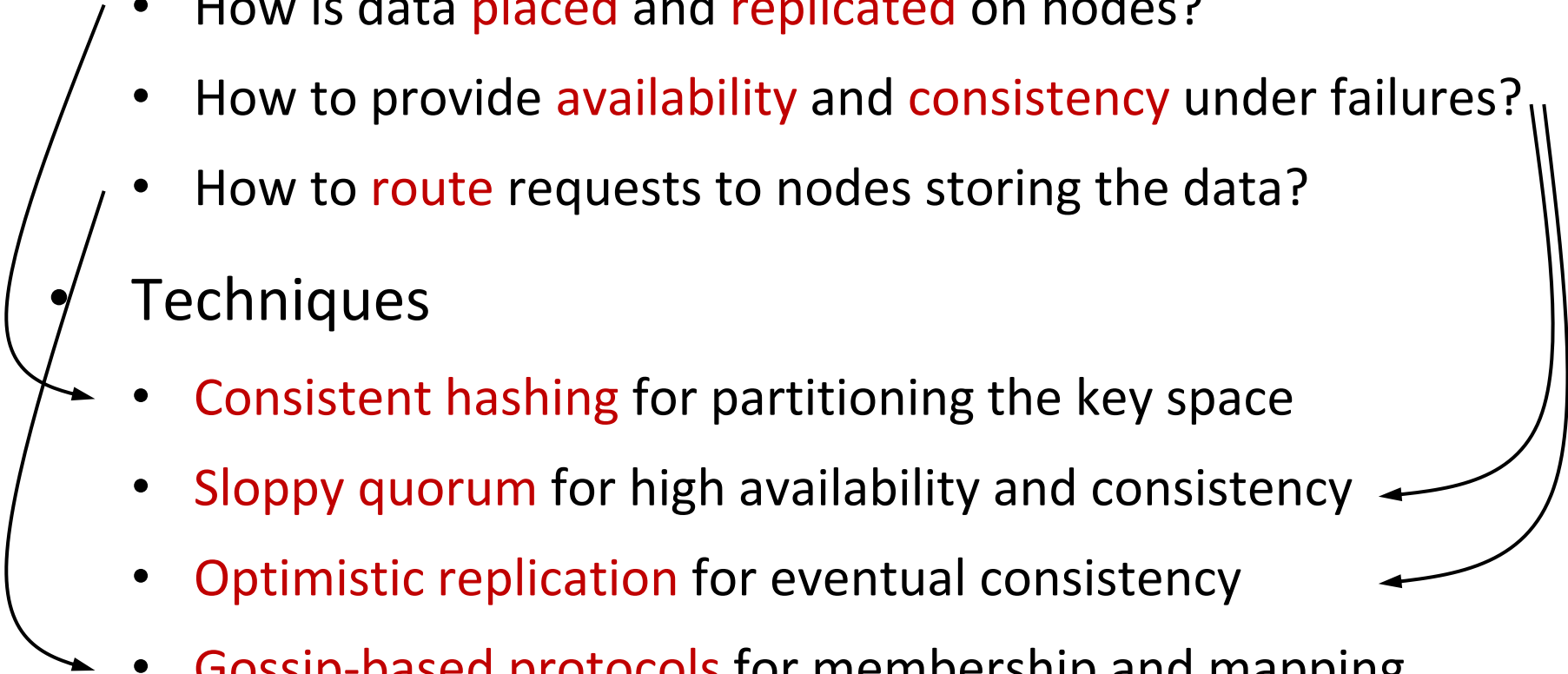
Motivation

- Need a highly available, scalable storage system
- Key-value storage is prevalent, powerful pattern
 - Data is mostly accessed by primary key
 - Data served is often self-describing blobs (not structured)
- RDMS is not a good fit
 - Most features are unused, e.g., query optimizer, stored procedures, triggers, etc.
 - Scales up, not out so easily
 - Strongly consistent, limits availability

Key Requirements

- High “always writable” availability is critical
 - Accept writes during failure scenarios
 - Total ordering not possible
 - Allow writes without prior context
 - Ordering a client’s writes may not be possible
- User-perceived consistency is also very important
 - Anomalies due to weak consistency should be rare
- Guaranteed latency, measured in 99.9 percentile
- Incremental scalability, reduces TCO
- Tunable latency, consistency, availability, durability

Design Overview

- Dynamo is a decentralized (peer-to-peer) replicated, distributed hash table
 - Key design questions
 - How is data **placed** and **replicated** on nodes?
 - How to provide **availability** and **consistency** under failures?
 - How to **route** requests to nodes storing the data?
 - Techniques
 - **Consistent hashing** for partitioning the key space
 - **Sloppy quorum** for high availability and consistency
 - **Optimistic replication** for eventual consistency
 - **Gossip-based protocols** for membership and mapping
- 

Dynamo API

- The get(k) and put(k, v) API includes a **context** that contains version information (discussed later)

```
// get returns one or more object versions, and a context.  
//  
object[], context = get(key)
```

```
// put supplies context returned by previous get.  
//  
put(key, object, context)
```

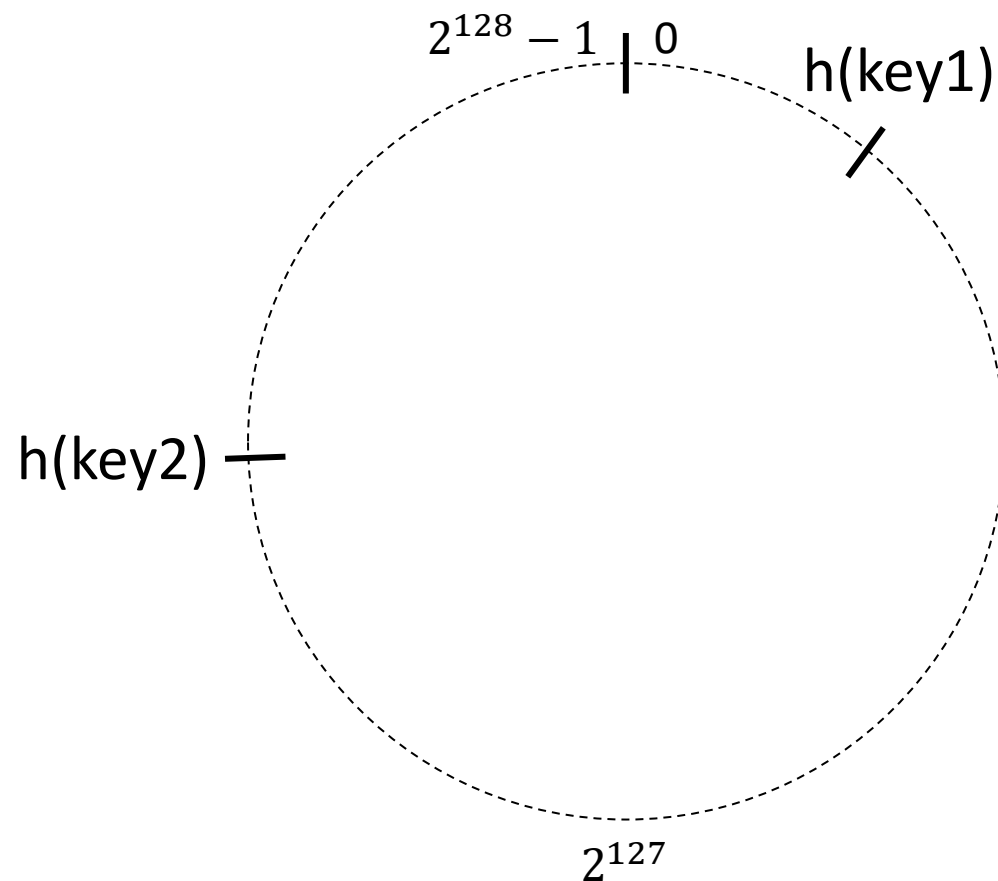
Consistent Hashing

Why Consistent Hashing?

- Enables partitioning the key space across nodes
- Handles adding and deleting nodes
 - If you use standard hashing, why would this be a problem?
 - Enables incremental scalability
- Handles data replication

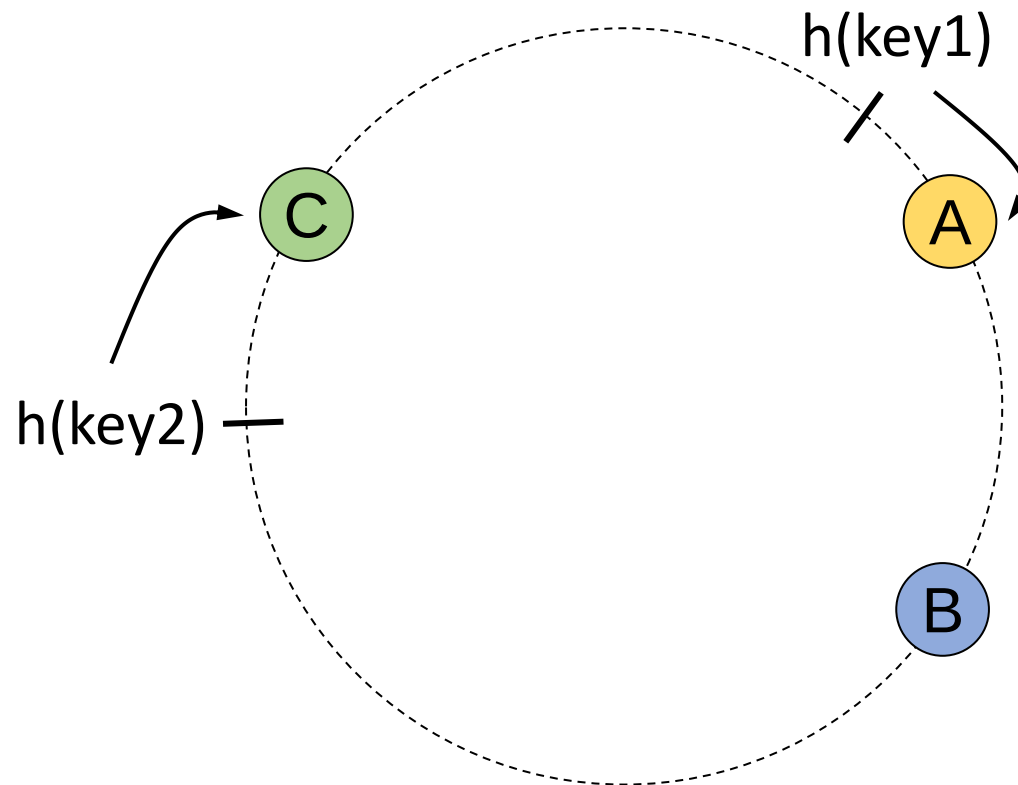
Hash ID

- Hash the key to a 128 bit ID
 - $ID = h(\text{key})$, where h is MD5
- ID lies in a circular key space



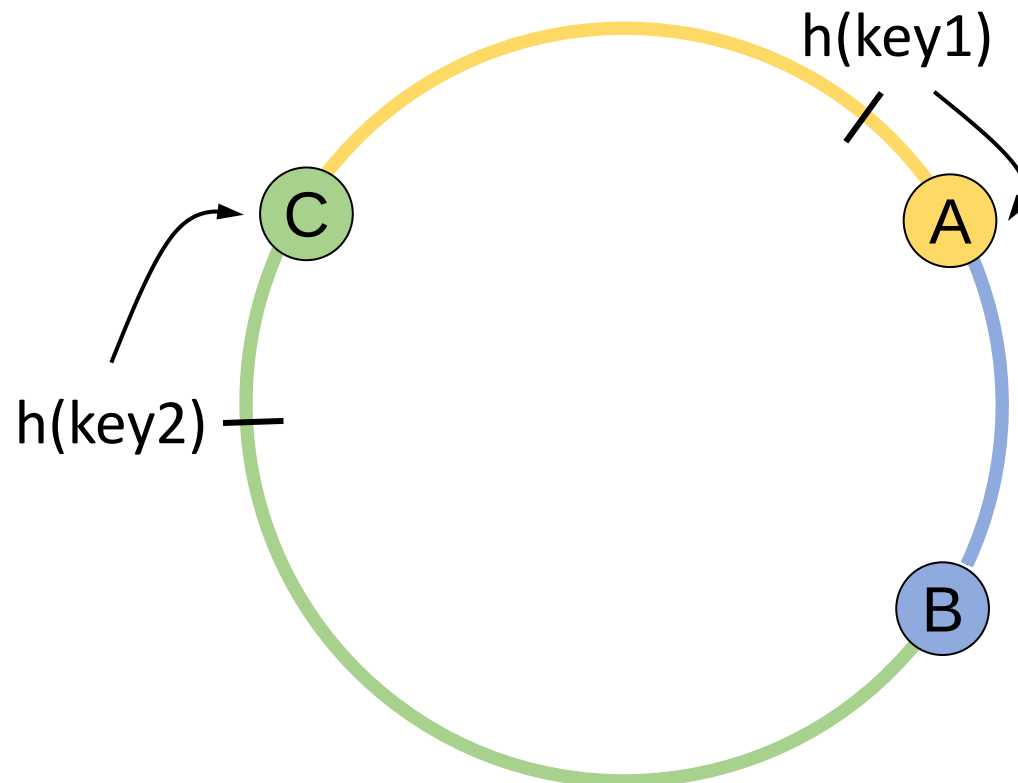
Node and Key Assignment

- Key idea of consistent hashing:
 - Each node is assigned an ID, e.g., $h(A)$, in the key space
 - Each key (based on its ID) is owned by first clockwise node



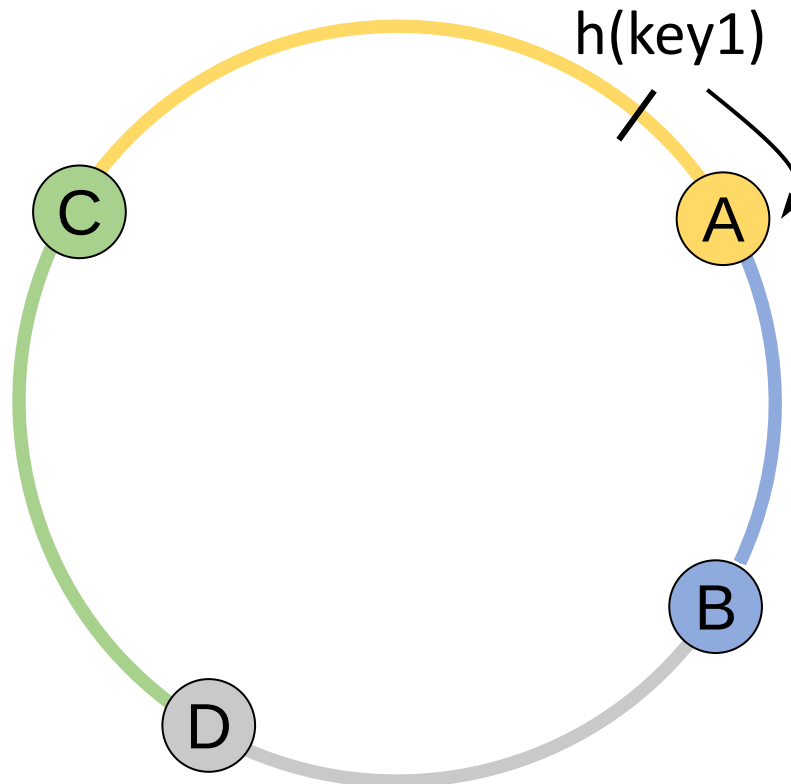
Nodes Store Key Ranges

- Each node owns keys in the range between its predecessor and itself



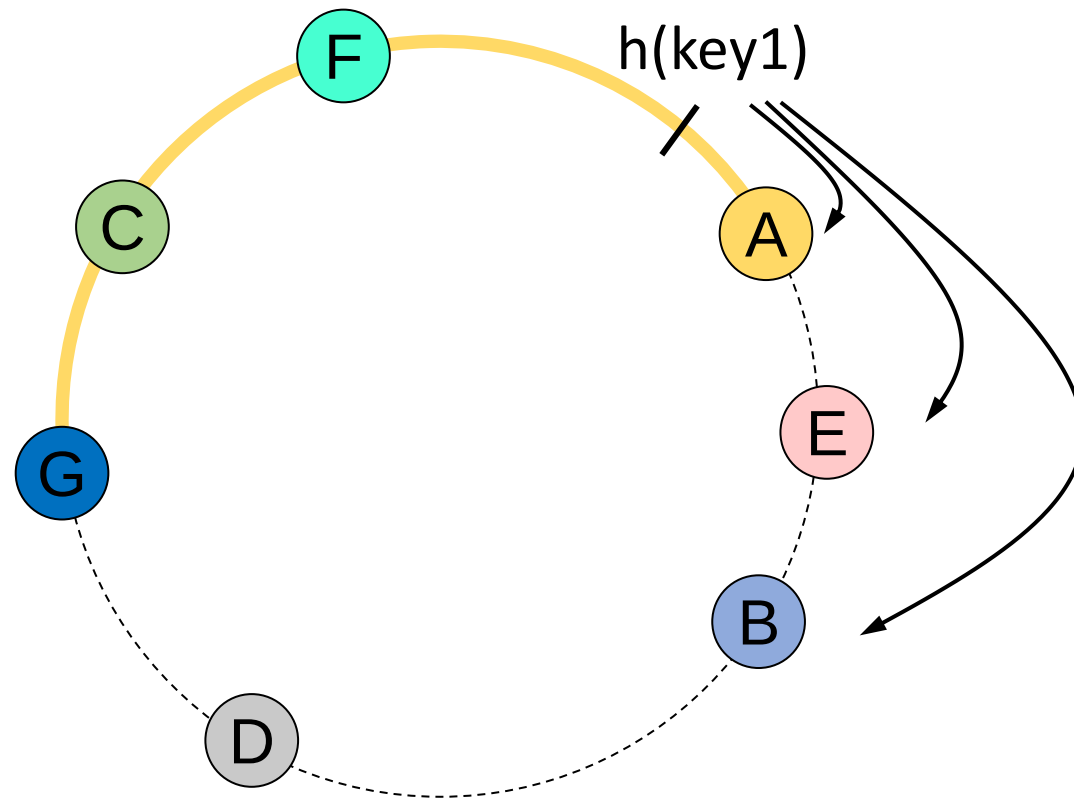
Node Addition/Deletion

- Adding or removing a node affects only a part of the key range



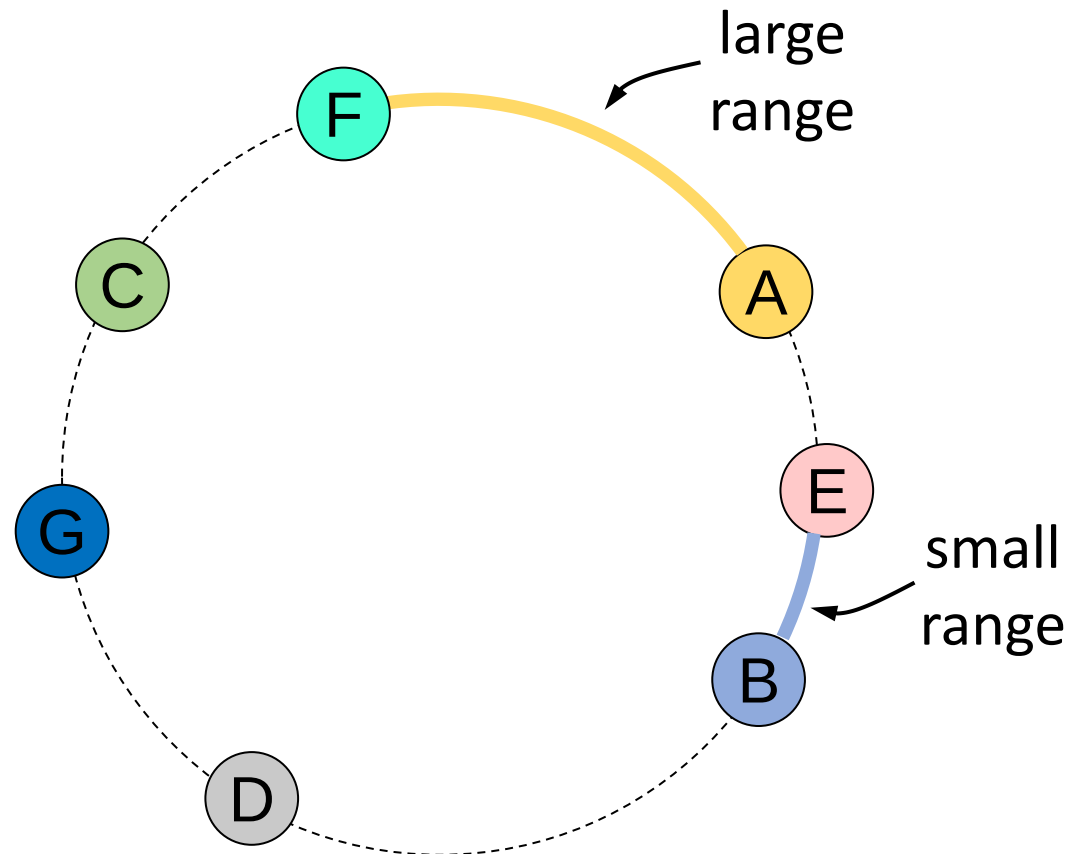
Replication

- A key is replicated at the first 3 clockwise nodes
- Each node stores key ranges between its 3rd predecessor and itself



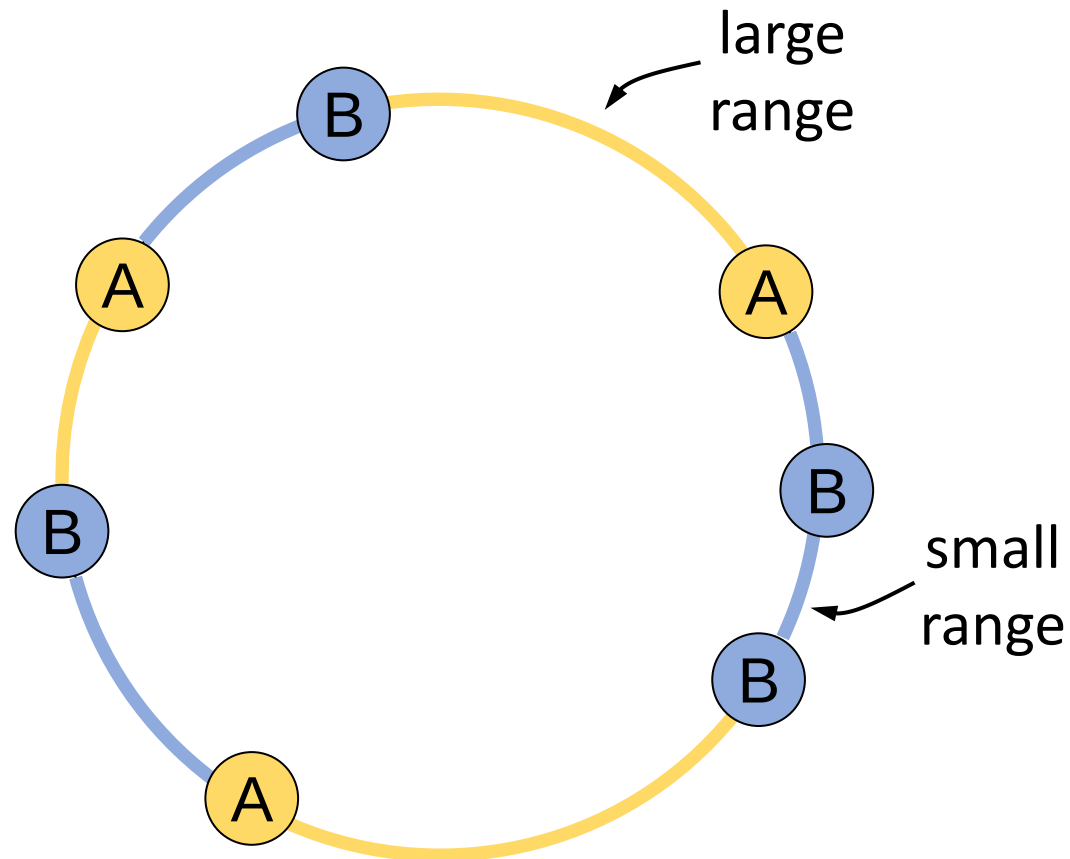
Key Load Imbalance

- Key range can be unbalanced



Load Balancing via Virtual Nodes

- Map each physical node to multiple virtual nodes
 - Pros: reduces key range skew across physical nodes
 - Cons: increases membership size



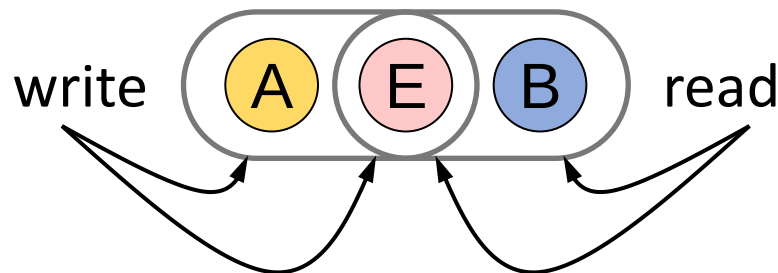
Sloppy Quorum

Why Sloppy Quorum?

- Challenge is to ensure both **high availability** and **user-perceived consistency**, with two goals:
 - Data should be always writable
 - Avoid anomalies due to weak consistency with high probability
- Solution: Be available
 - Consistent during normal operation, sloppy during failures

Majority Quorum Protocol

- Sloppy quorum builds on majority quorum protocol
- Basic Majority Quorum protocol
 - Assume
 - N: Number of nodes (or replicas) storing a key
 - R: Successful read involves at least R nodes
 - W: Successful write involves at least W nodes
 - Choose: $R + W > N$
 - Since reads and writes overlap at least one replica, majority quorum ensures reads will read the latest data
 - Example:
 - $N = 3, R = 2, W = 2$

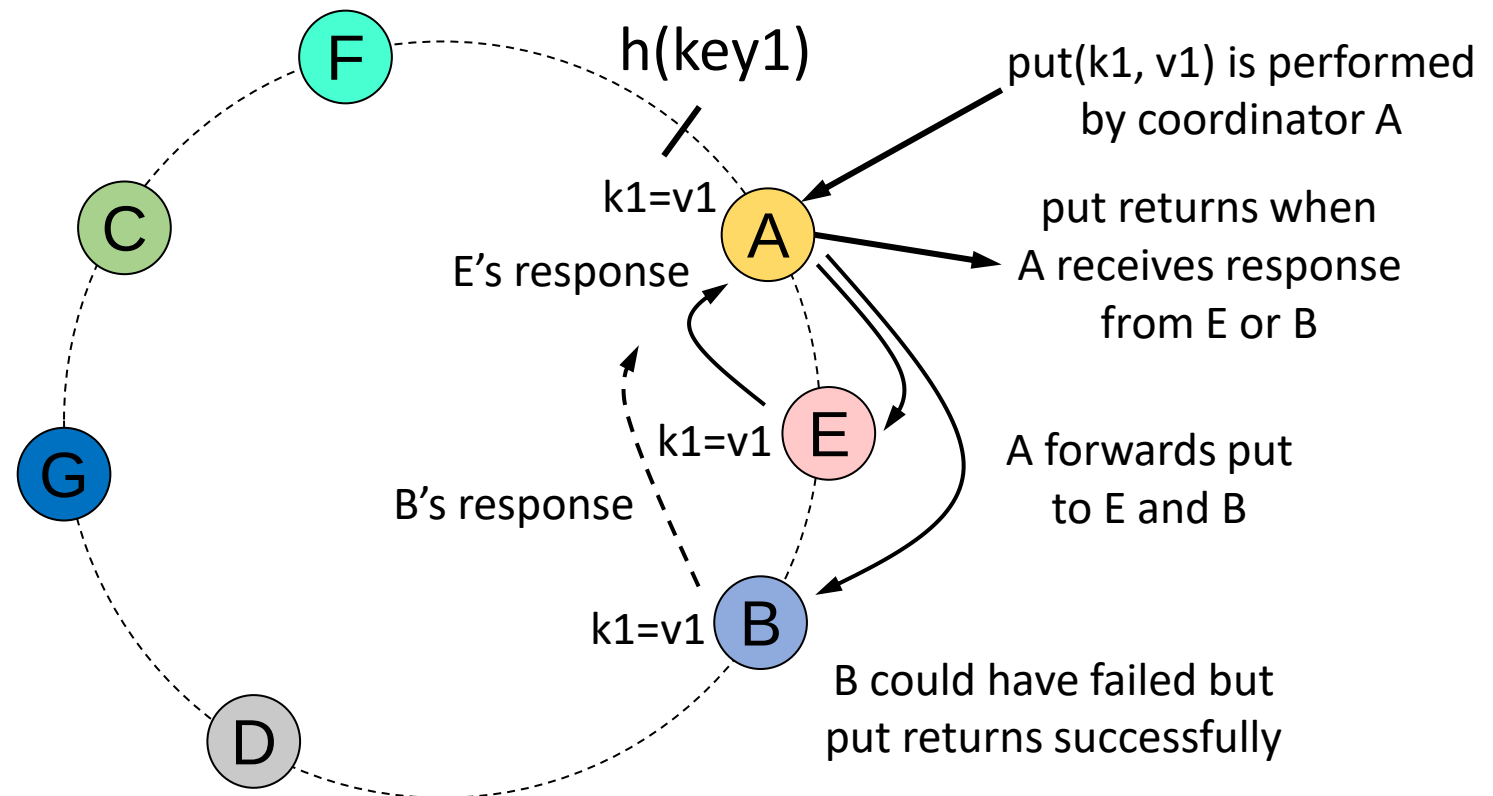


Majority Quorum Example

- Assume $N = 3$, $R = 2$, $W = 2$
- `put(k, v)`
 - Coordinated by a node that stores key k
 - Typically, first replica is chosen
 - However, other replicas may also be chosen for load balancing
 - Returns when at least $W=2$ replicas update key and respond to the coordinator
- `get(k)`
 - Coordinated by any node (whether node stores k or not)
 - Returns when at least $R=2$ replicas respond with the value of key to the coordinator

Majority Quorum Example

- $N = 3, R = 2, W = 2$
- Assume client performs $\text{put}(k1, v1)$



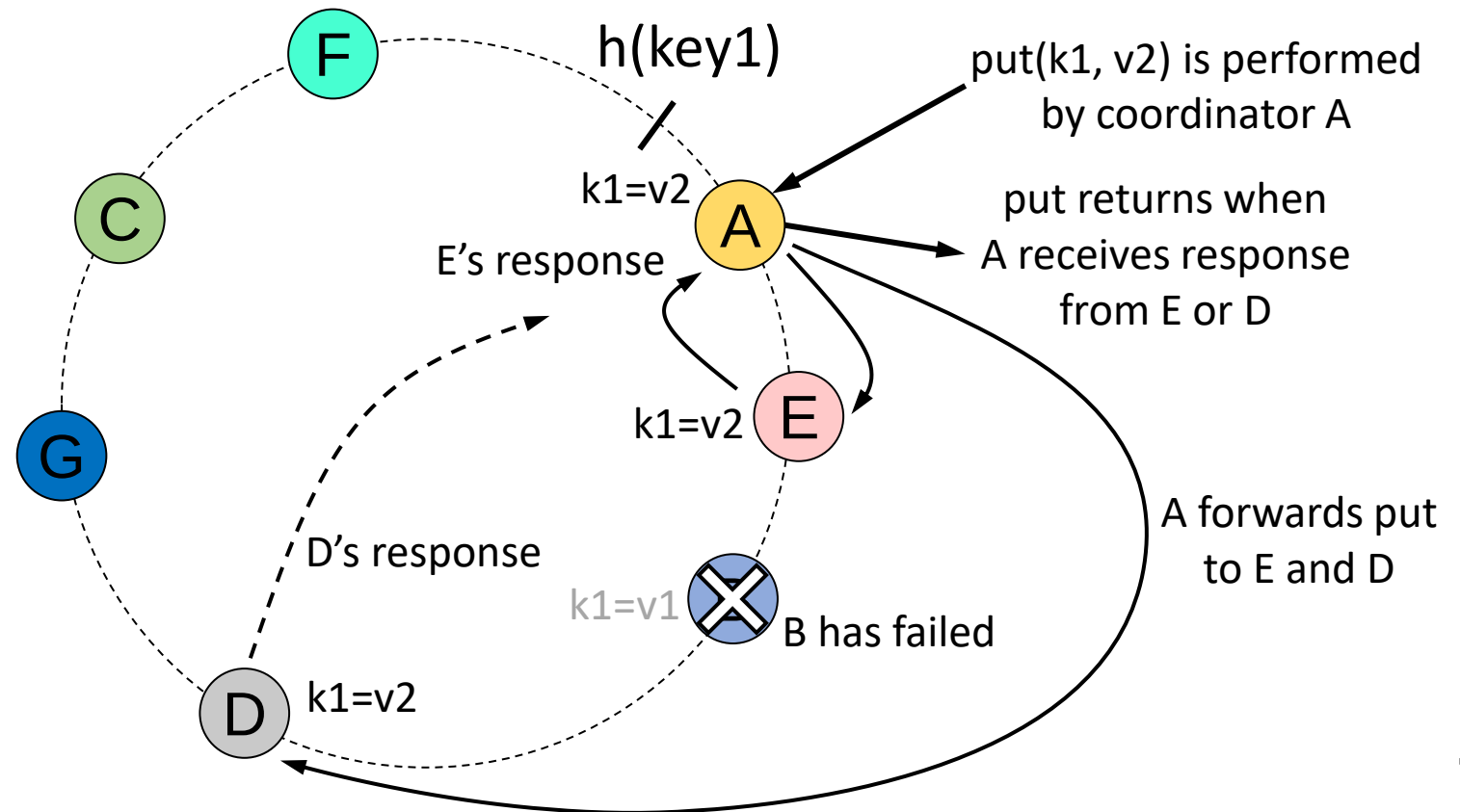
Sloppy Quorum

always writable operation

- When a node is not available, writes sent to a new node
- Reads and writes are performed on N **healthy** nodes
 - So failed nodes are skipped
 - Sloppy: $R+W > N$ does not guarantee that reads, writes overlap
- However, reads still often read the latest data

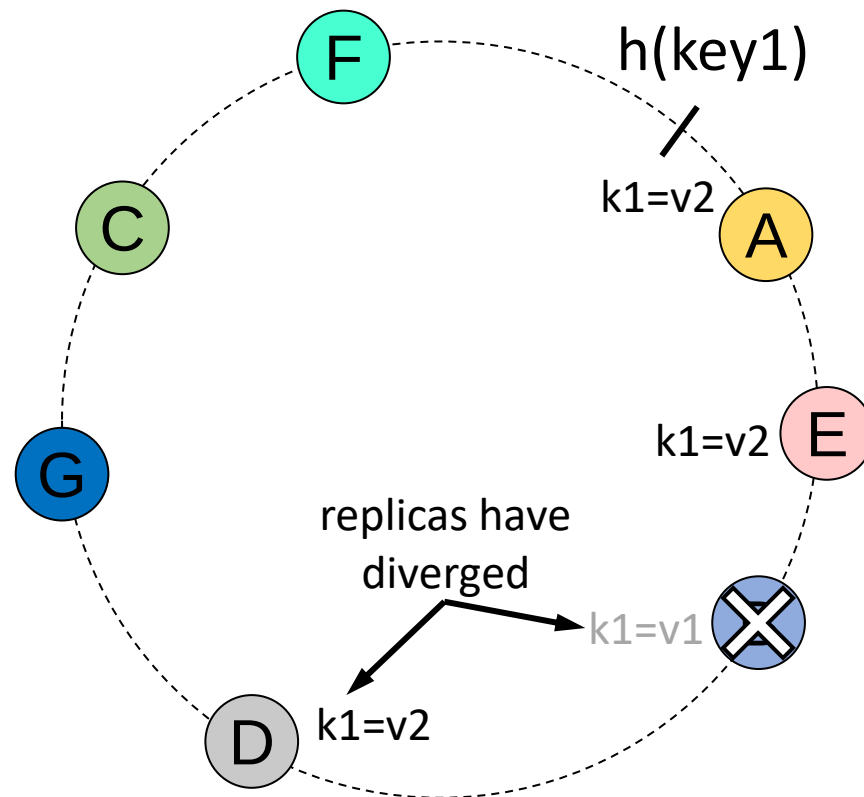
Sloppy Quorum

- Assume client performs $\text{put}(k1, v2)$
- If B fails, A forwards $\text{put}(k1, v2)$ to D (**temporary replica**)
- Even if B restarts, $\text{get}(k1)$ often returns latest version



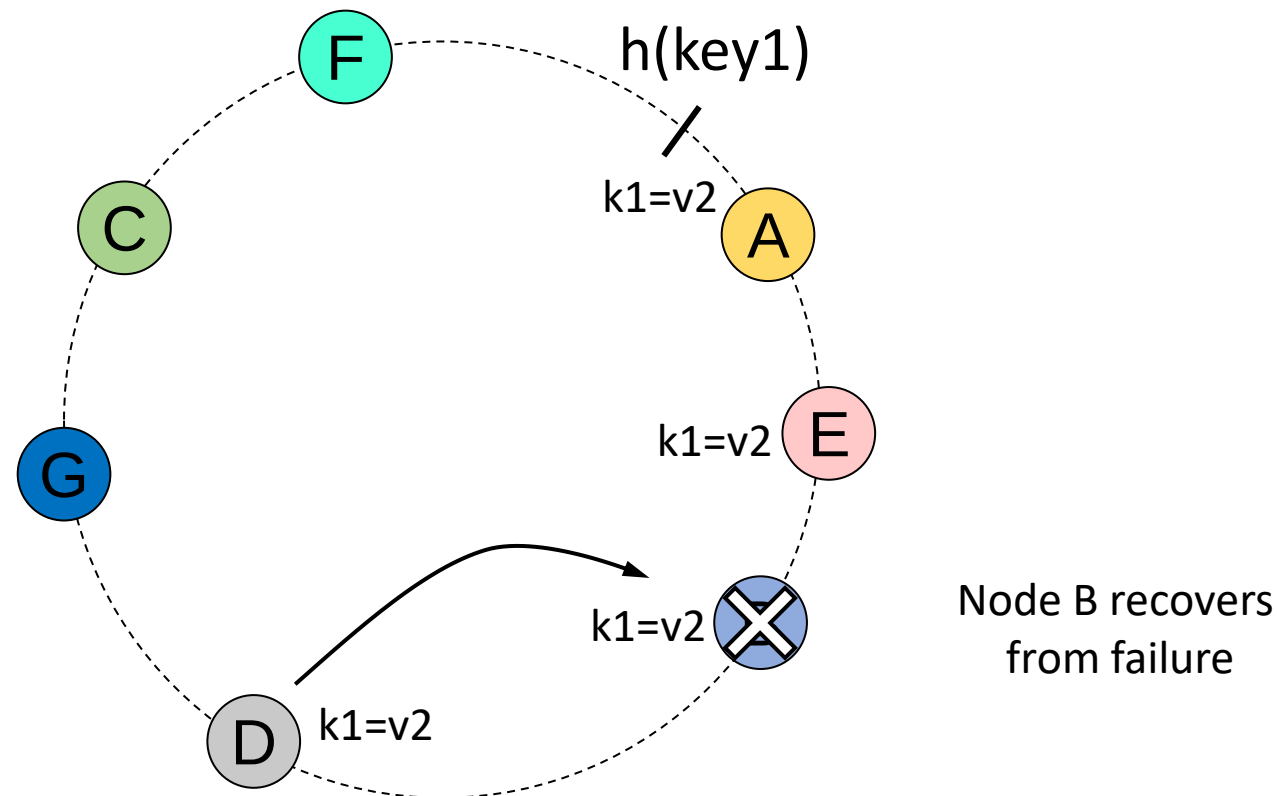
Sloppy Quorum and Replica Divergence

- After node B fails, it will have a stale replica



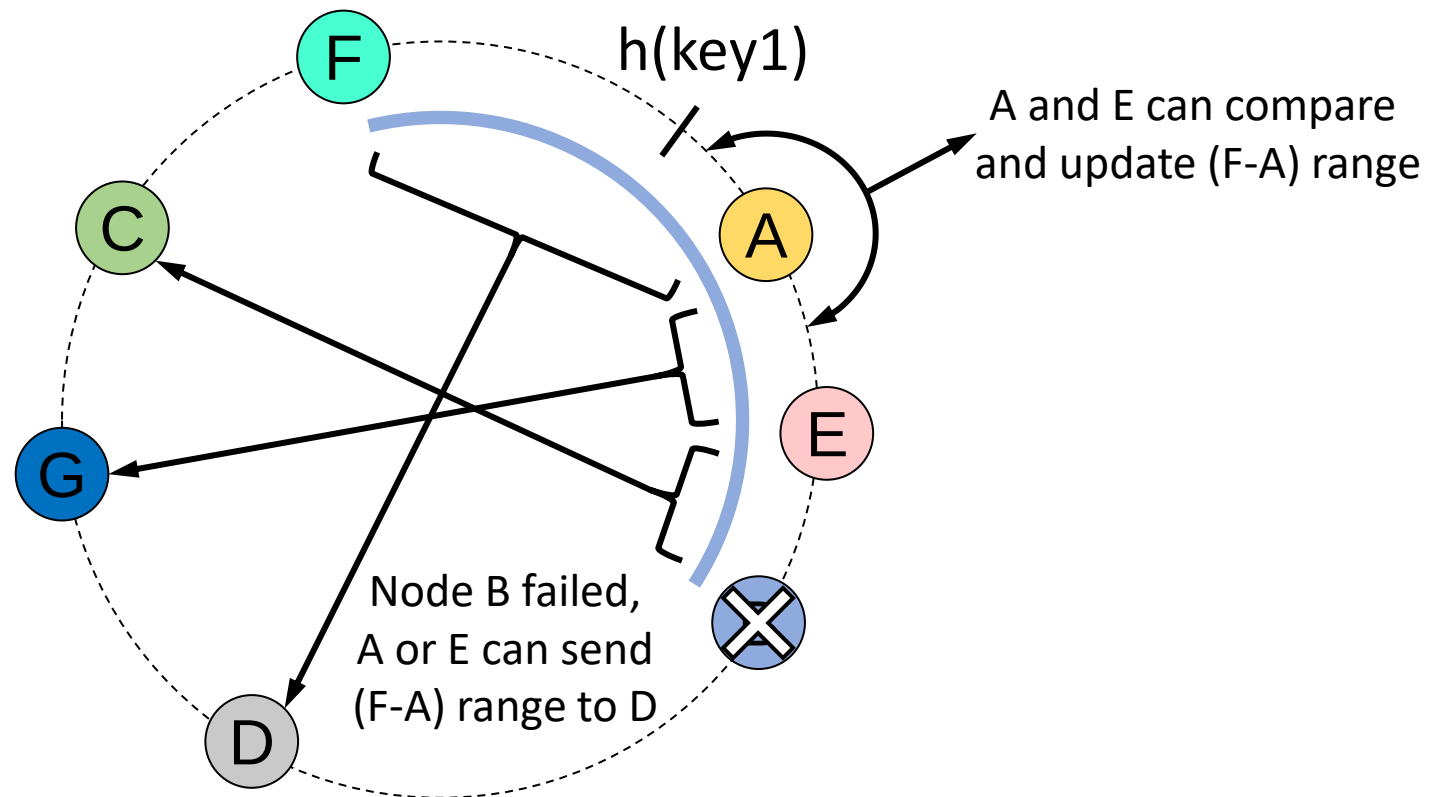
Sloppy Quorum and Failure Recovery

- After node B fails, it will have a stale replica
- When temporary replica D finds that B has recovered:
 - D sends v2 to B, and may delete v2 from its store



Replica Synchronization

- Nodes may have stale replicas, leave or fail permanently
- Replicas of key ranges are synchronized with an efficient anti-entropy protocol that uses Merkle trees



Sloppy Quorum Configuration

N	R	W	Application
3	2	2	Consistent, durable, user state (typical configuration)
N	1	N	High performance read engine
1	1	1	Distributed web cache

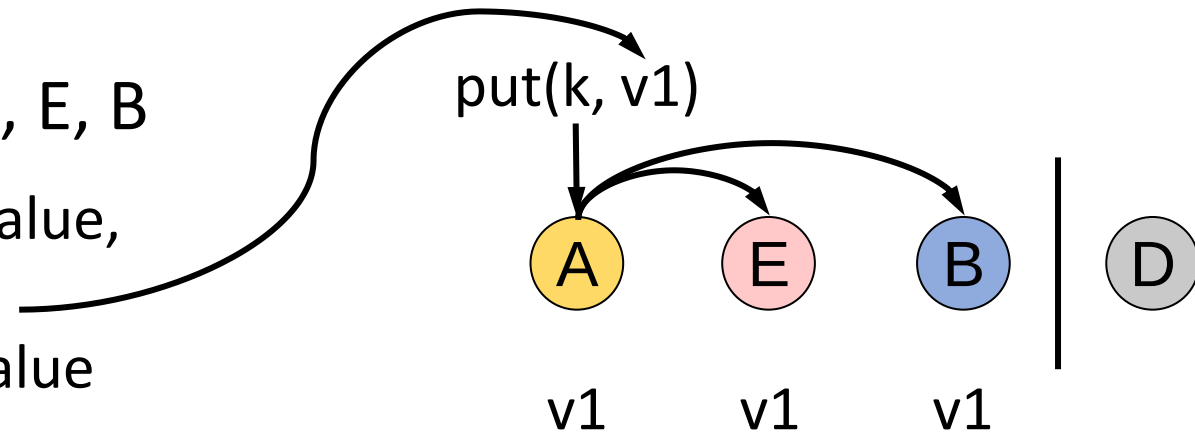
Optimistic Replication

Why Optimistic Replication?

- With sloppy quorum, replicas may be stale or conflicting
 - Stale replica: replica has old version
 - Conflicting replica: process wrote to a stale replica
- Optimistic replication is used to
 - **Detect** stale and conflicting replicas
 - **Synchronize** them so replicas become **eventually consistent**
- Dynamo implements optimistic replication using immutable versions and version histories
 - put() creates new, **immutable object version**
 - Each node tracks **version history**, i.e., version information for each object version and how they are related

Optimistic Replication Example

- `put(k, v1)` writes to A, E, B
 - Assume `v1` is both a value, and a new version associated with the value

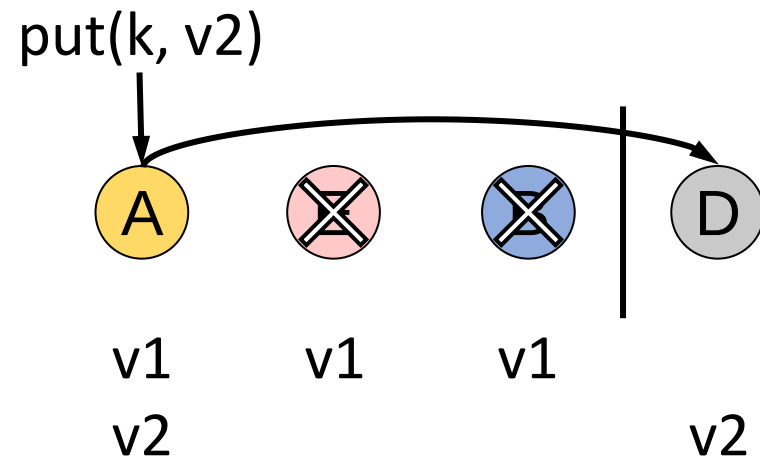


Version history

`v1`

Example

- B and E fail
- $\text{put}(k, v2)$, based on $v1$, writes to A and D
 - D is a temporary replica
- $v1$ is an ancestor of $v2$ in version history

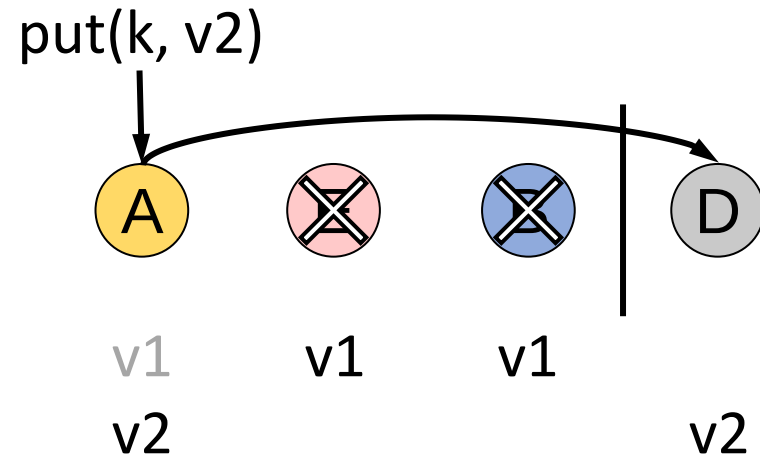


Version history

$v1$
↓
 $v2$

Example

- B and E fail
- $\text{put}(k, v2)$, based on $v1$, writes to A and D
 - D is a temporary replica
- $v1$ is an ancestor of $v2$ in version history
- A removes $v1$ (stale version)

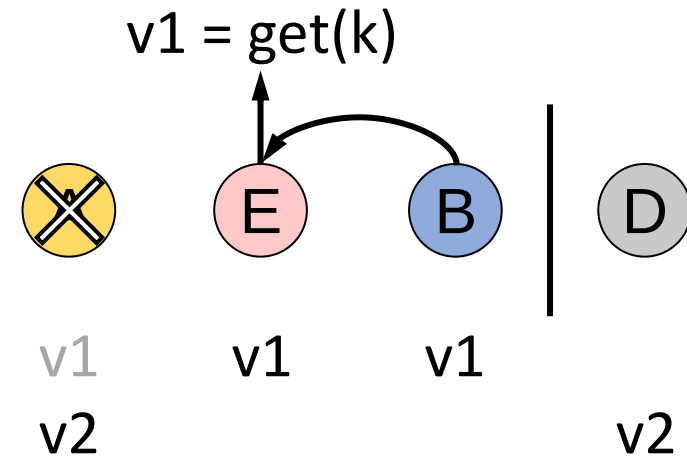


Version history

$v1$
↓
 $v2$

Example

- B and E recover
- A fails
- `get(k)` reads `v1` from E and B
 - `v1` is a stale version

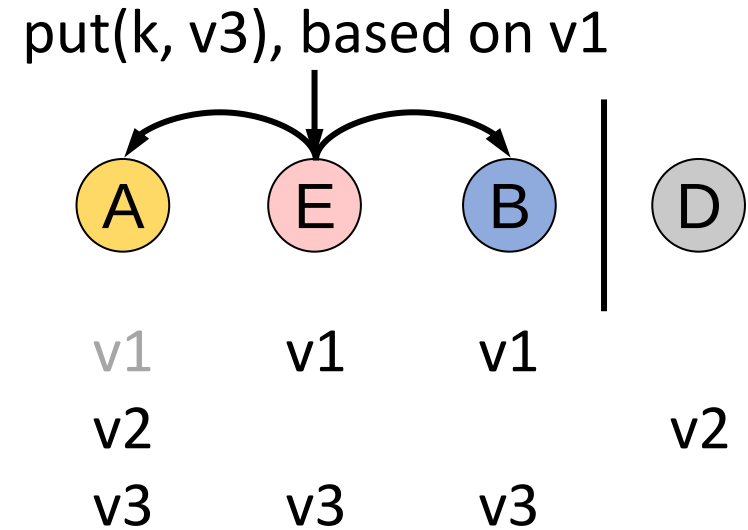


Version history

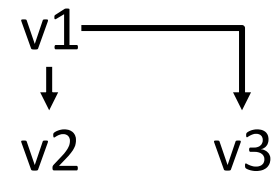
v1
↓
v2

Example

- A recovers
- `put(k, v3)`, based on `v1`, writes to E, A, B
 - Creates branch in history, since `put()` performed based on stale version `v1`

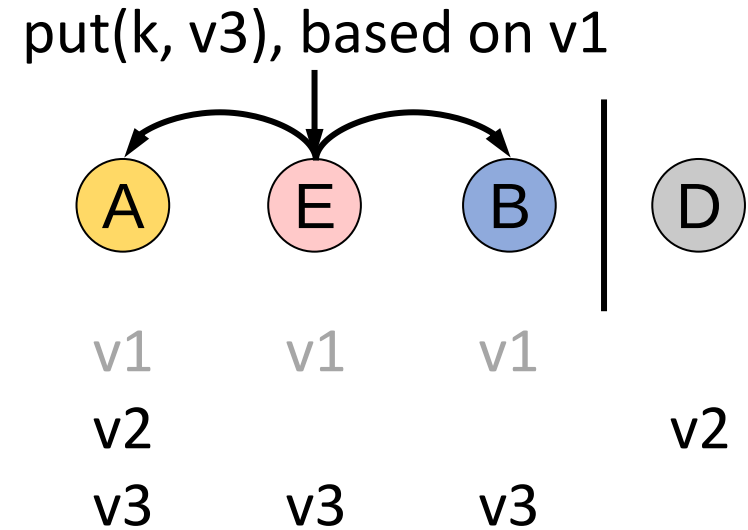


Version history

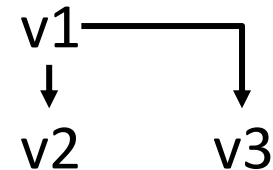


Example

- A recovers
- `put(k, v3)`, based on `v1`, writes to E, A, B
 - Creates branch in history, since `put()` performed based on stale version `v1`
- Nodes store versions that are leaves in version history
 - E and B remove `v1`, ancestor of `v3`
 - A stores `v2` and `v3`, since they conflict

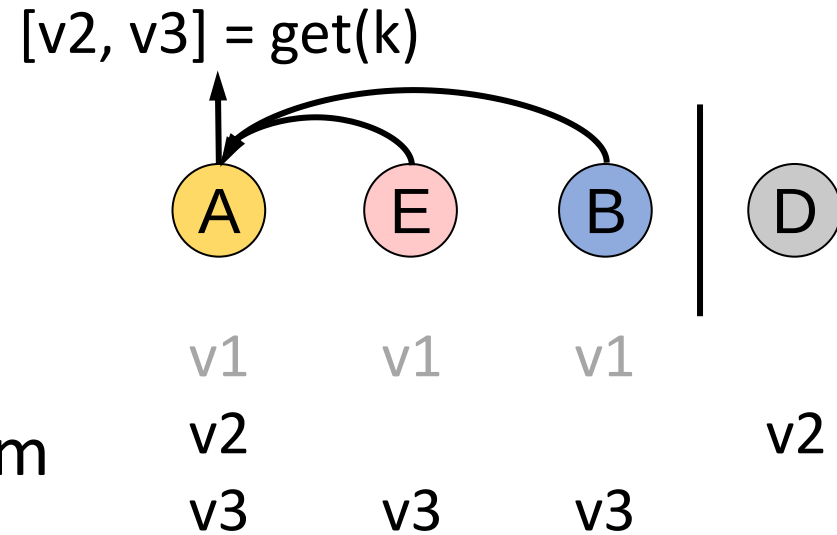


Version history

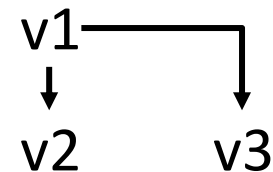


Example

- $\text{get}(k)$ reads conflicting $[v2, v3]$ from A, E, B
- Dynamo provides all conflicting versions to client, since client knows best how to reconcile them
 - E.g., app can merge two conflicting shopping carts



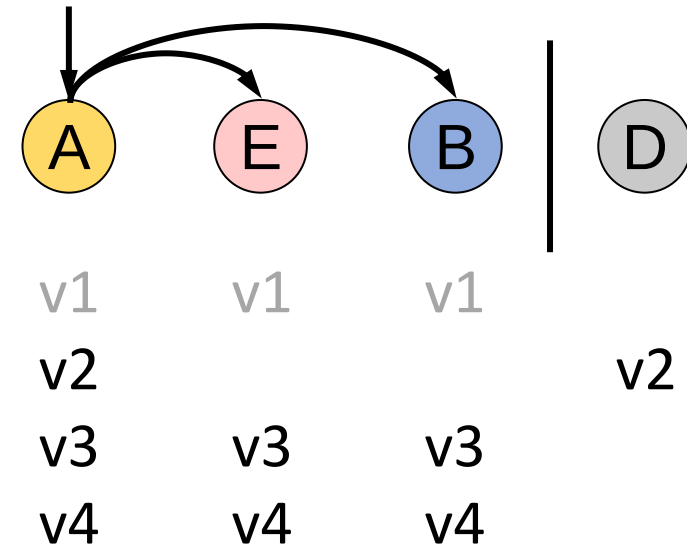
Version history



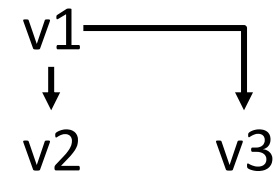
Example

- `put(k, v4)`,
based on `[v2, v3]`,
writes to A, E, B
- Dynamo expects app
reconciled `[v2, v3]`
when it created `v4`

`put(k, v4)`, based on `[v2, v3]`



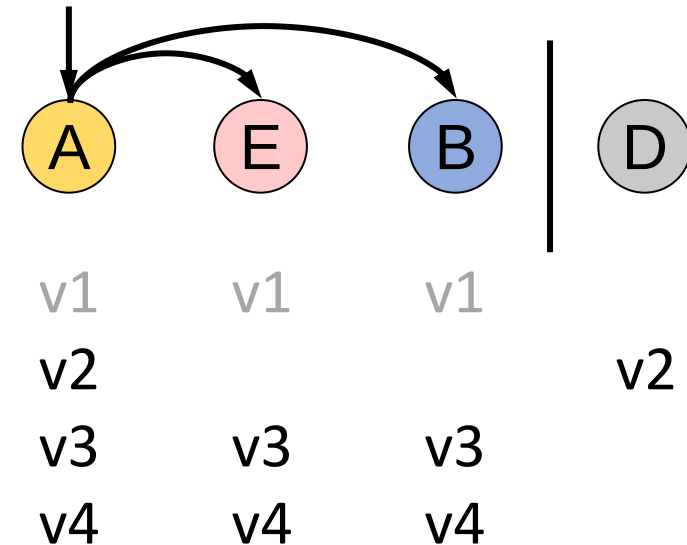
Version history



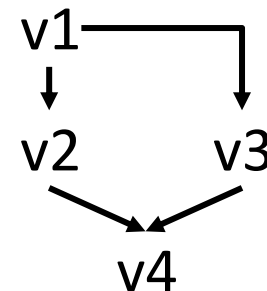
Example

- `put(k, v4)`,
based on `[v2, v3]`,
writes to A, E, B
 - Dynamo expects app
reconciled `[v2, v3]`
when it created `v4`
- `put()` merges conflicting versions
into single new version
 - Version history has single head

`put(k, v4)`, based on `[v2, v3]`



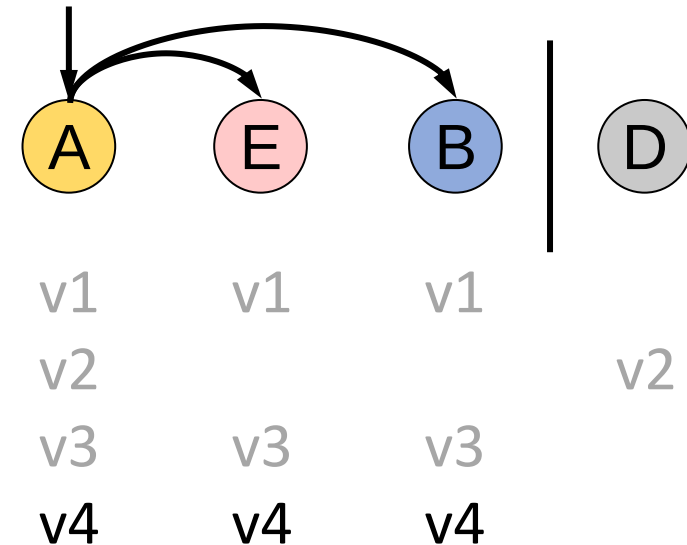
Version history



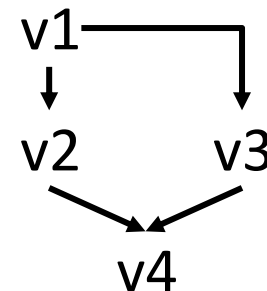
Example

- `put(k, v4)`,
based on `[v2, v3]`,
writes to A, E, B
 - Dynamo expects app
reconciled `[v2, v3]`
when it created `v4`
- `put()` merges conflicting versions
into single new version
 - Version history has single head
- A, E, B and D can remove
stale versions `v2` and `v3`

`put(k, v4)`, based on `[v2, v3]`



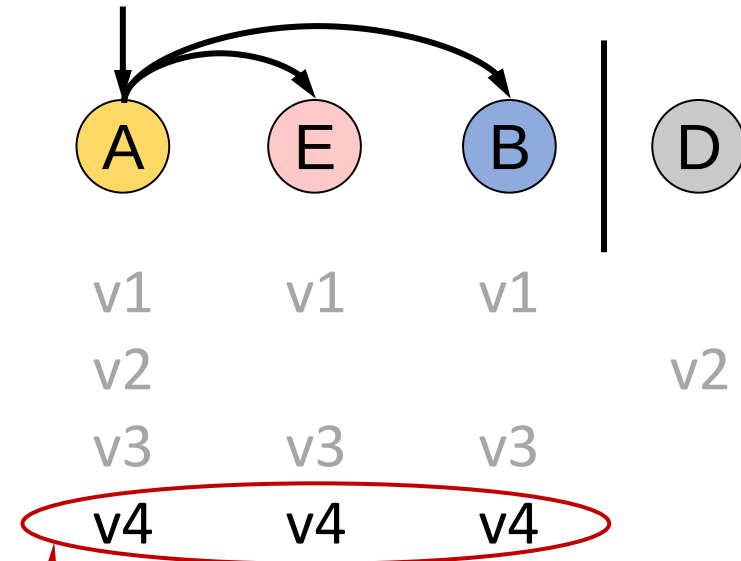
Version history



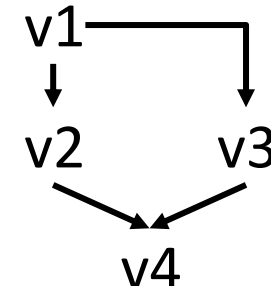
Example

- `put(k, v4)`,
based on `[v2, v3]`,
writes to A, E, B
 - Dynamo expects app
reconciled `[v2, v3]`
when it created `v4`
- `put()` merges conflicting versions
into single new version
 - Version history has single head
- A, E, B and D can remove
stale versions `v2` and `v3`
 - Object is **eventually consistent**

`put(k, v4)`, based on `[v2, v3]`



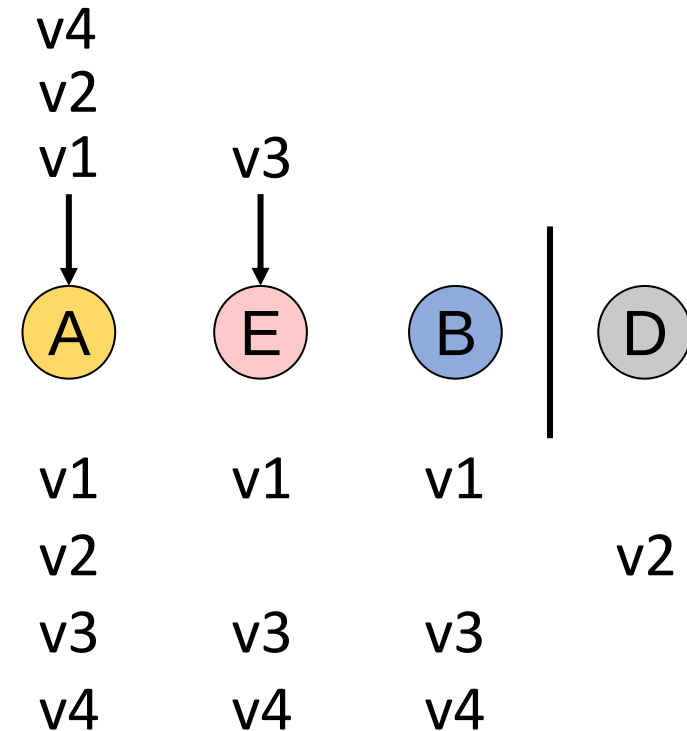
Version history



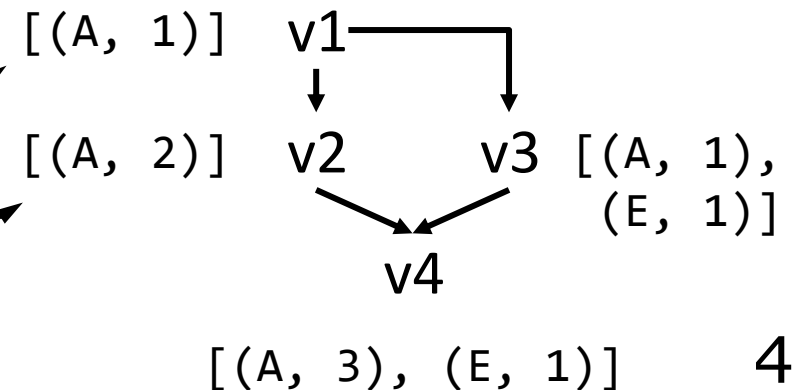
Implementing Version History With Vector Clocks

- Dynamo uses vector clocks to implement version history
- Efficiently capture causality
 - Stale versions can be forgotten
 - Concurrent versions are conflicting, require reconciliation
- Each object version stores a vector clock:

$[(\text{node1}, \# \text{updates1}), (\text{node2}, \# \text{updates2}), \dots]$



Version history



Dynamo API With Vector Clocks

- The get(k) and put(k, v) API includes a **context** that contains version information (**vector clock**)

```
// get returns one or more object versions, and a context.  
// context contains version information for each returned version.  
object[], context = get(key)  
  
// put supplies context returned by previous get.  
// context helps generate version information for new object version.  
put(key, object, context)
```

Gossip-Based Protocols

Why Gossip-Based Protocols?

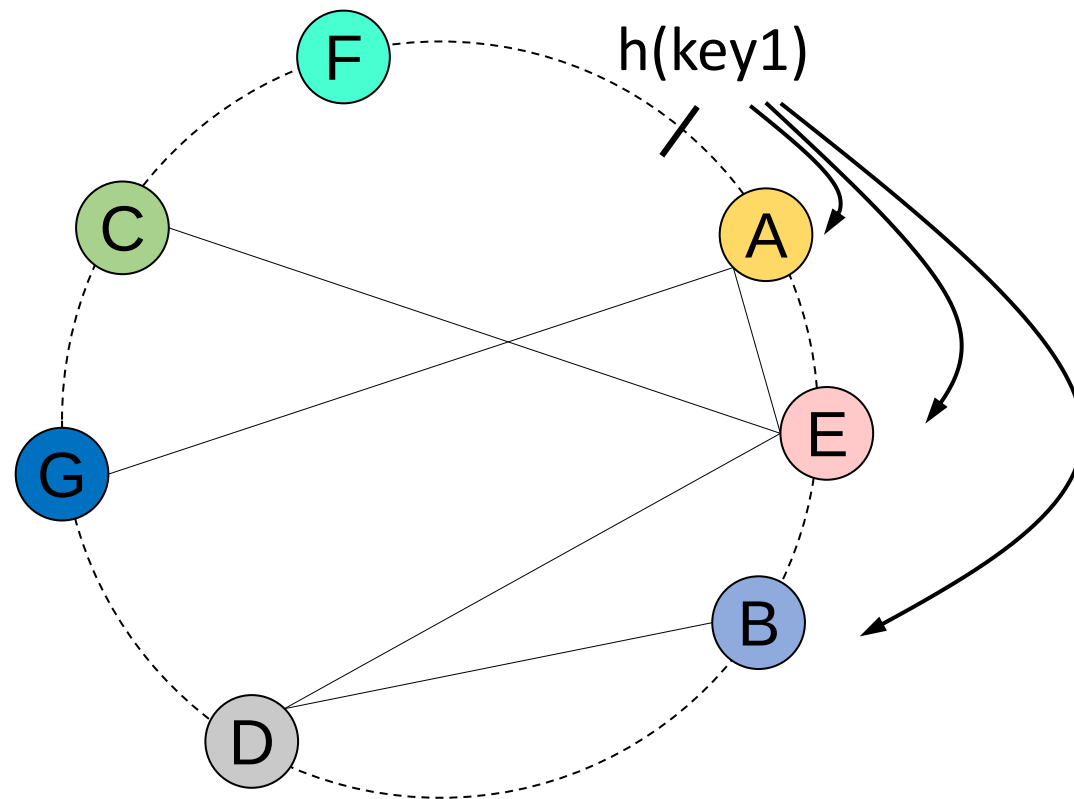
- Gossip protocols exchange information between nodes in a peer-to-peer (symmetric) manner
 - $A \leftrightarrow B$: A and B learn about each other's state
 - $B \leftrightarrow C$: B and C learn about each other's state, so C learns about A's state as well
- In general, these protocols enable nodes to
 - Learn about the state of other nodes
 - Use version history of state to become eventually consistent
- Tradeoffs:
 - Pros: avoid need for a coordinator, provide higher availability
 - Cons: nodes may have stale information for a while

Membership and Mapping

- Dynamo uses gossiping to propagate **membership, mapping information**
- Administrator explicitly adds and remove nodes
- **Membership:** After that, nodes communicate with each other to eventually learn about an added/deleted node
- **Mapping information:** Nodes also learn about node mappings, i.e., the key ranges stored on a node

Routing Key Lookup

- With gossiping, each node knows about 1) all other nodes, and 2) the key ranges each node stores
- Allows one-hop routing (critical for low latency)

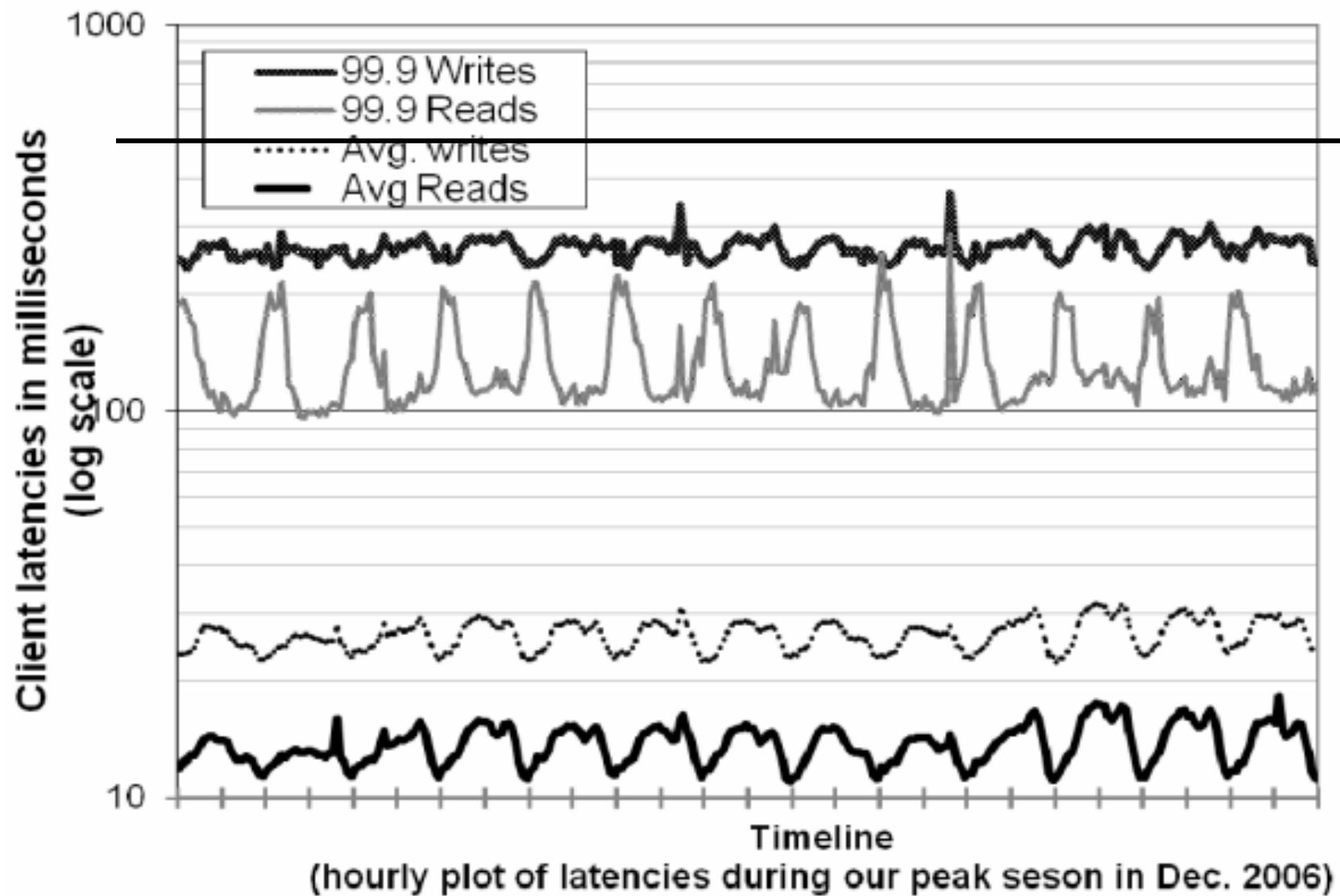


Failure Detection

- Initially implemented node failure detection via gossip
- Not needed due to explicit node add/remove
 - No need to distinguish between temporarily failed/recovering nodes versus removed/added nodes
- Simple failure detection
 - A detects B as failed if it doesn't respond to a ping message
 - A periodically checks if B is alive again
 - In the absence of requests, A doesn't need to know if B is alive

Evaluation

500 ms SLA for storage system
for shopping cart application

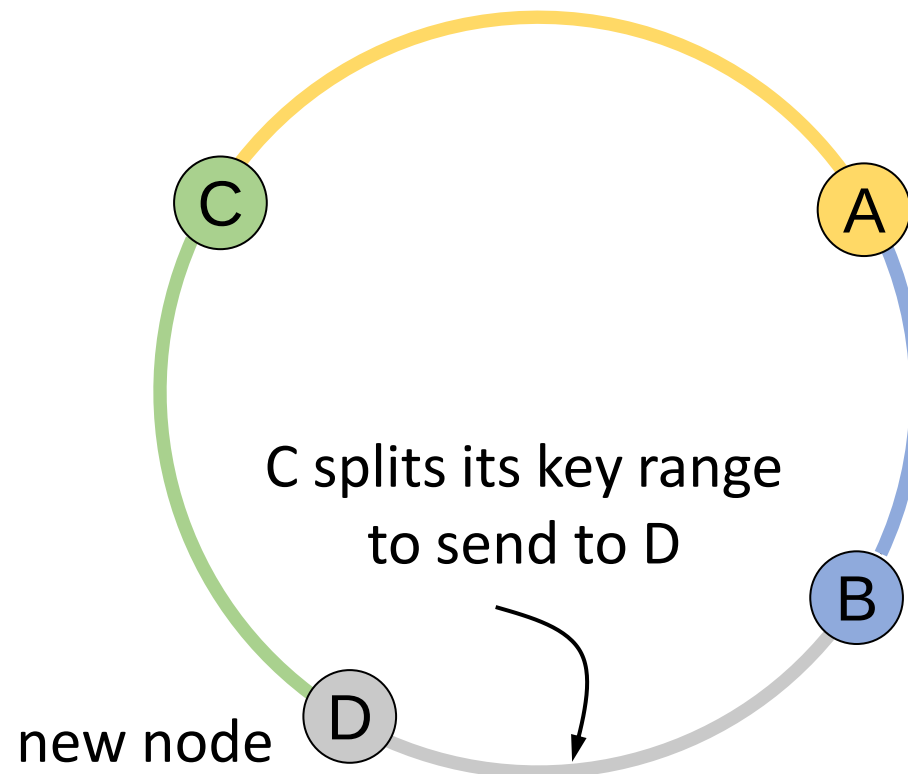


Lessons Learned: Tail Latency

- 99.9 percentile is a high bar
 - Packet losses, waiting on disk, accessing large objects, JVM garbage collection, ...
- Techniques used to reduce tail latency
 - Use buffered writes to avoid waiting on disk
 - Need to deal with version consistency, e.g., if version number is increased on disk, but failure loses the object version
 - Lazy removal of stale versions
 - Adaptive throttling of background operations based on observed foreground operation latency

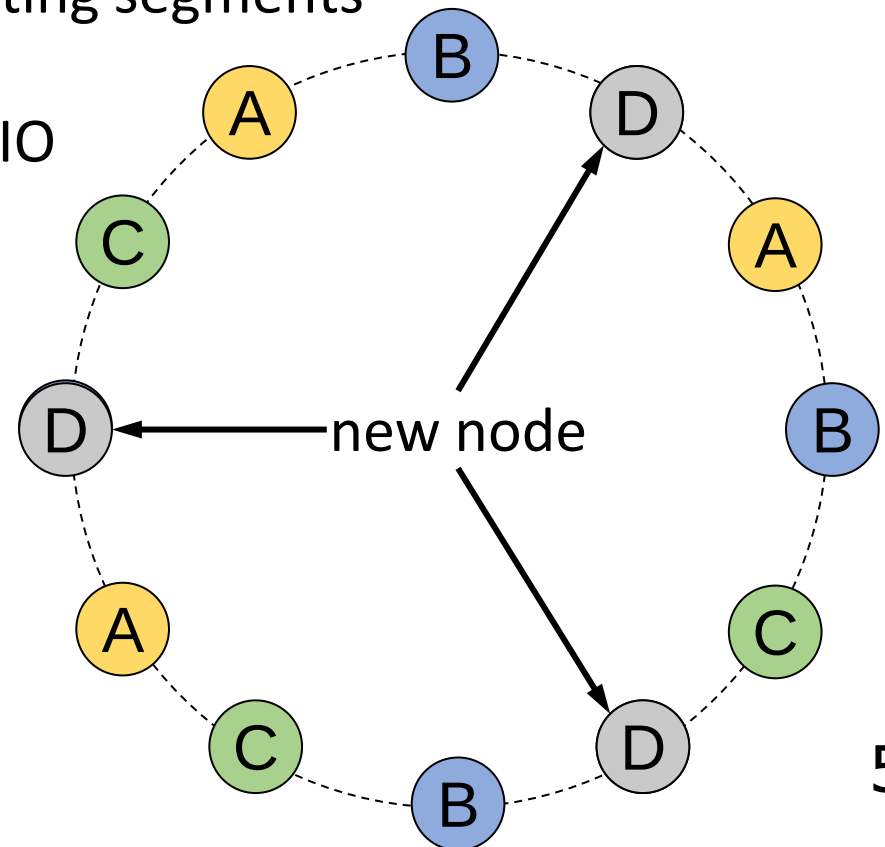
Lessons Learned: Repartitioning

- Slow repartitioning
 - Successor (C) splits key range to bootstrap new node (D)
 - Requires ordered key traversal (scan), causes heavy random disk I/O at C, with throttling, takes hours/days to finish



Lessons Learned: Repartitioning

- Use fixed arcs strategy
 - Divide hash ring into many fixed key ranges called segments
 - **Coordinate assignment of segments to nodes**
 - New node (D) steals entire existing segments from other nodes, allowing simple file transfer, sequential IO
- Scales better
- However, moves away from decentralized principle



Dynamo: Pros and Cons

- Pros
 - Highly available - 99.9995% request success over one year
 - Meets tight latency requirements
 - Incrementally scalable
 - Tunable consistency, durability
- Cons
 - No transactional semantics
 - More challenging programming model, e.g., handling conflicts
 - Doesn't support ordered key operations, streaming operations
 - Not appropriate for large (> 1MB) objects

Conclusions

- Highly scalable, replicated, eventually consistent key-value store
- Decentralized (peer-to-peer) techniques can be used for building highly available system
 - High availability: provides an "always-on" experience
 - Mostly consistent: clients rarely see conflicting versions
- Highly influential
 - Apache Cassandra builds on Dynamo's design

Discussion

Q1

- What design constraints are imposed by the “always writable” requirement?

Q2

- How would you compare Dynamo against Bigtable in terms of:
 - API
 - Workloads
 - Availability
 - Consistency

Q3

- Say dynamo is heavily loaded, i.e., many of the nodes are loaded, and so the dynamo administrator decides to add a node. Would that help reduce load on all the nodes?

Q4

- Under what scenarios can a client read multiple conflicting versions of an object? Why is this unlikely in Dynamo?

Q5

- What are the scalability limitations of Dynamo?