Case Study 4:
Dynamo: Amazon's Highly Available Key-value Store

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

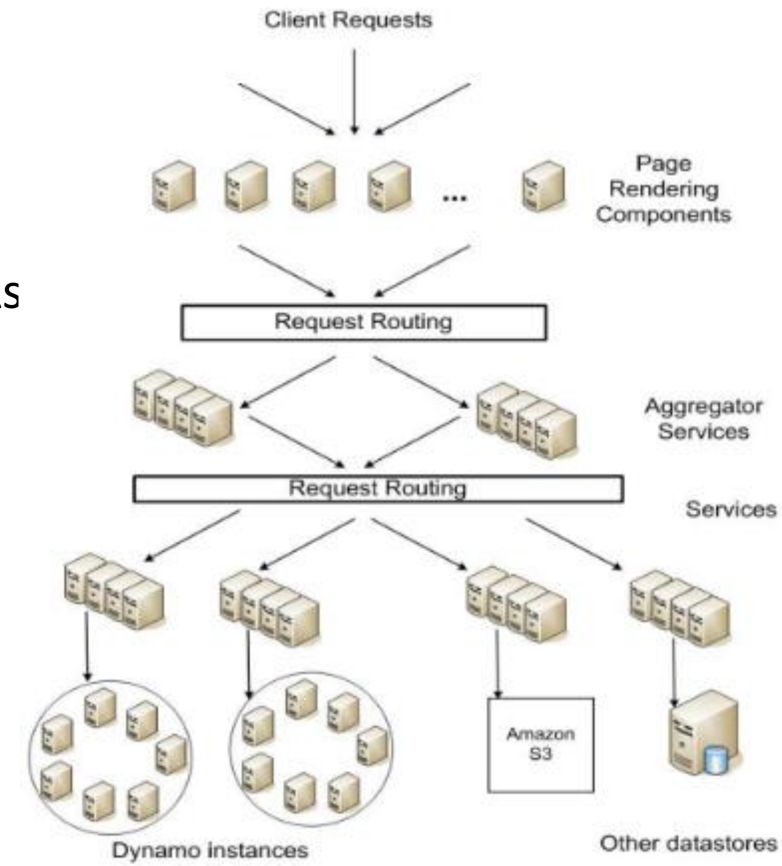
Distributed Systems
ECE419

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

- Loosely coupled,
  service-oriented architecture
- Stringent latency requirements
  - Services must adhere to formal SLAs
  - Measured at 99.9 percentile
    - 500 ms for client requests
    - 10-100 ms for core services
- 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, but scale out is not so easy
  - 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, e.g., after failure
    - 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

Consistent Hashing

Dynamo API

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

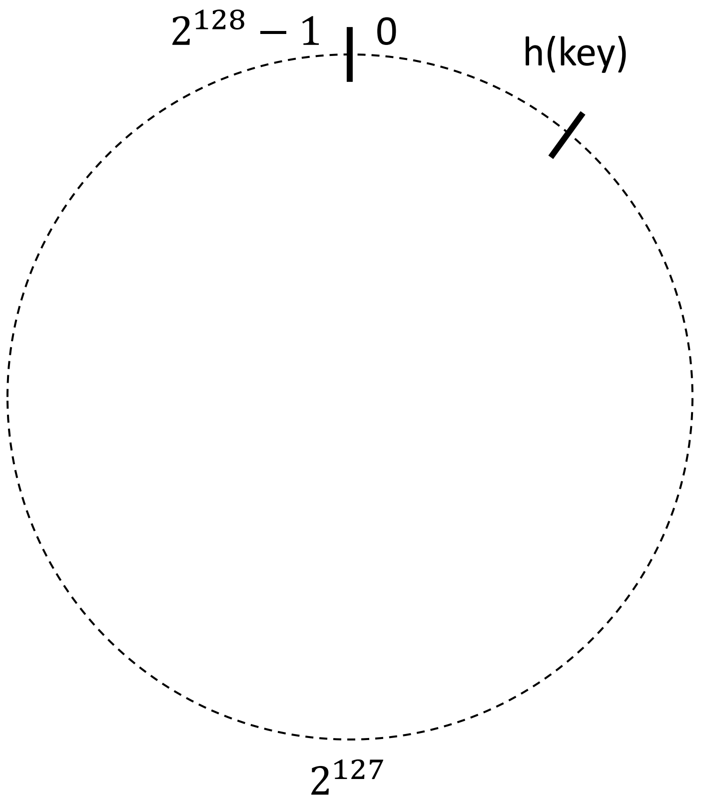
```
// get returns one or more object versions, and a context.
//
object[], context = get(key)

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

Why consistent hashing?

- Enables partitioning (sharding) 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

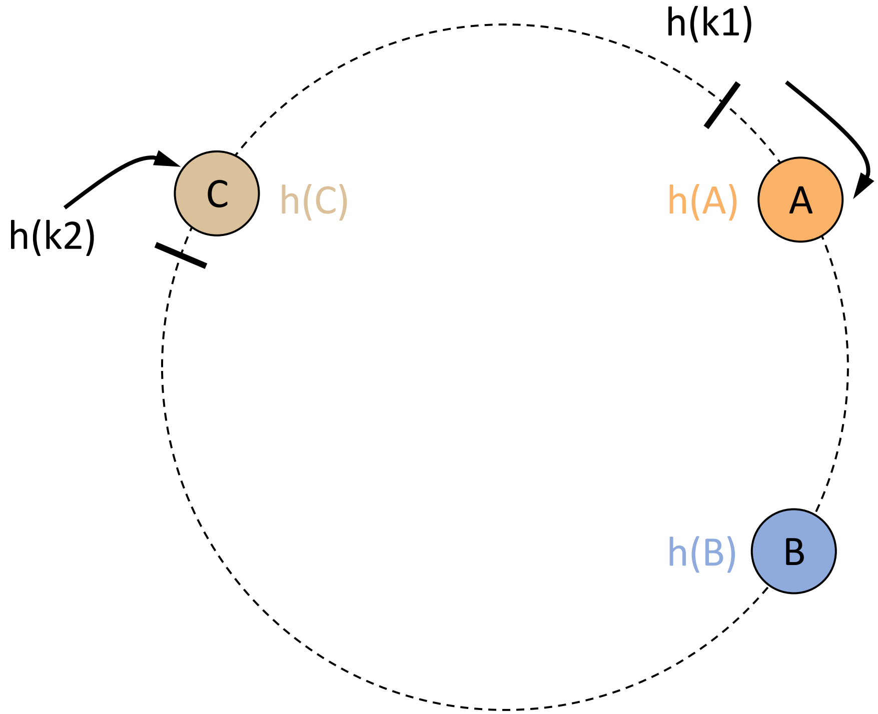
Hash ID

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



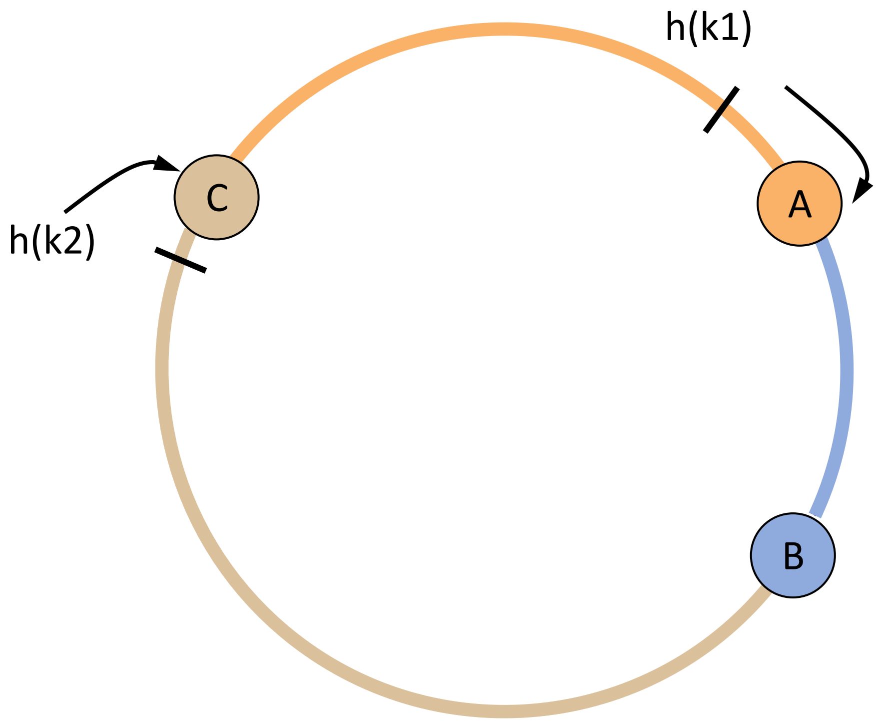
Node and key assignment

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



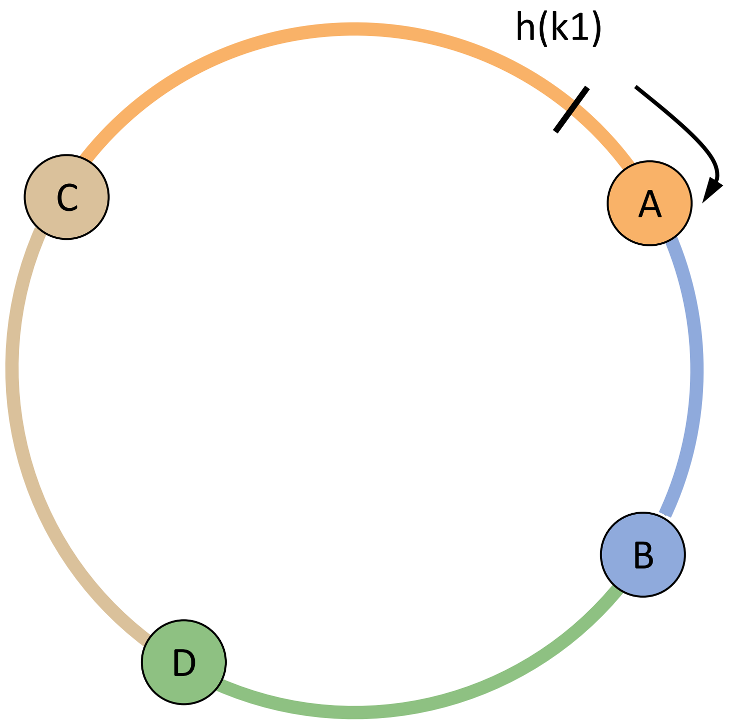
Nodes store key ranges

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



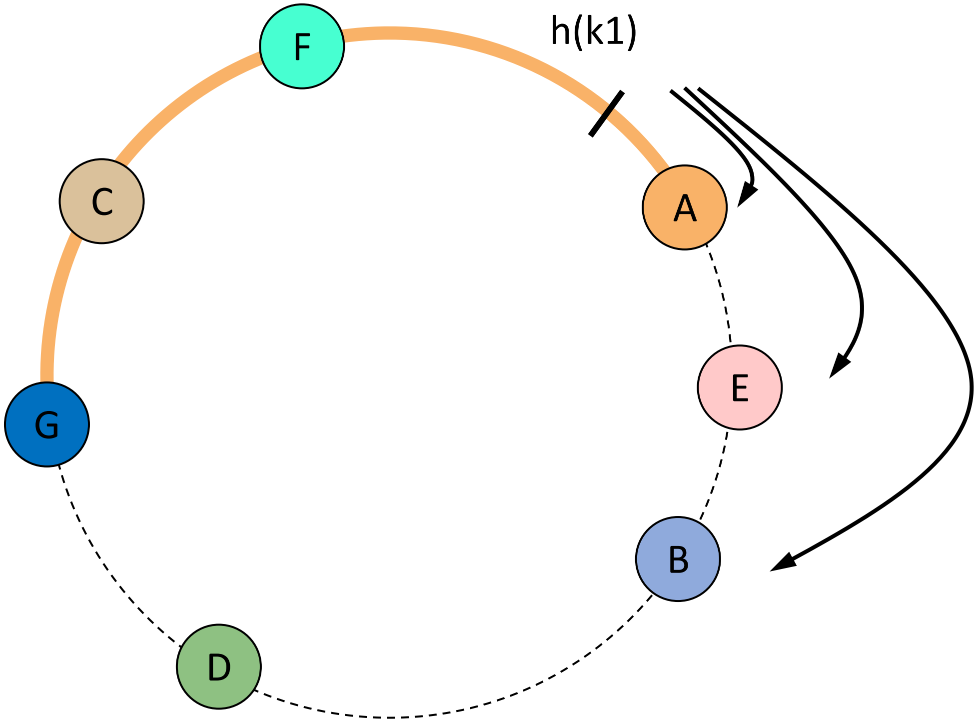
Node addition/deletion

- Adding or removing a node only affects a part of the key range, i.e., successor's key range



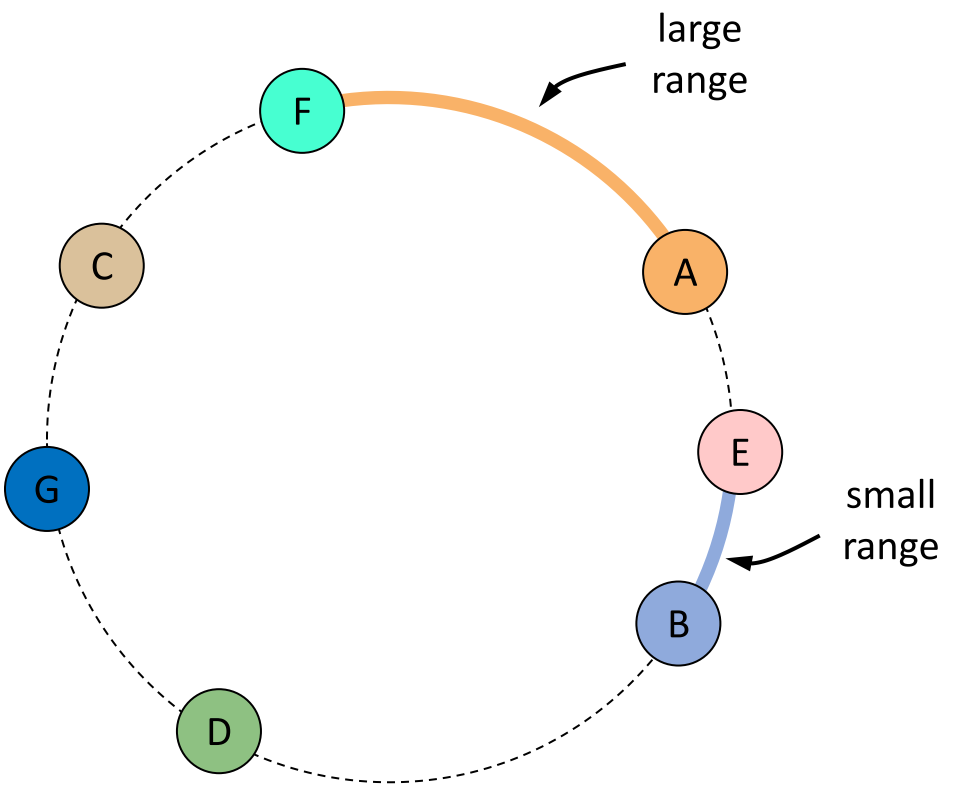
Replication

- A key is replicated at the first N (e.g., 3) clockwise nodes
- Each node stores key ranges between its 3rd predecessor and itself



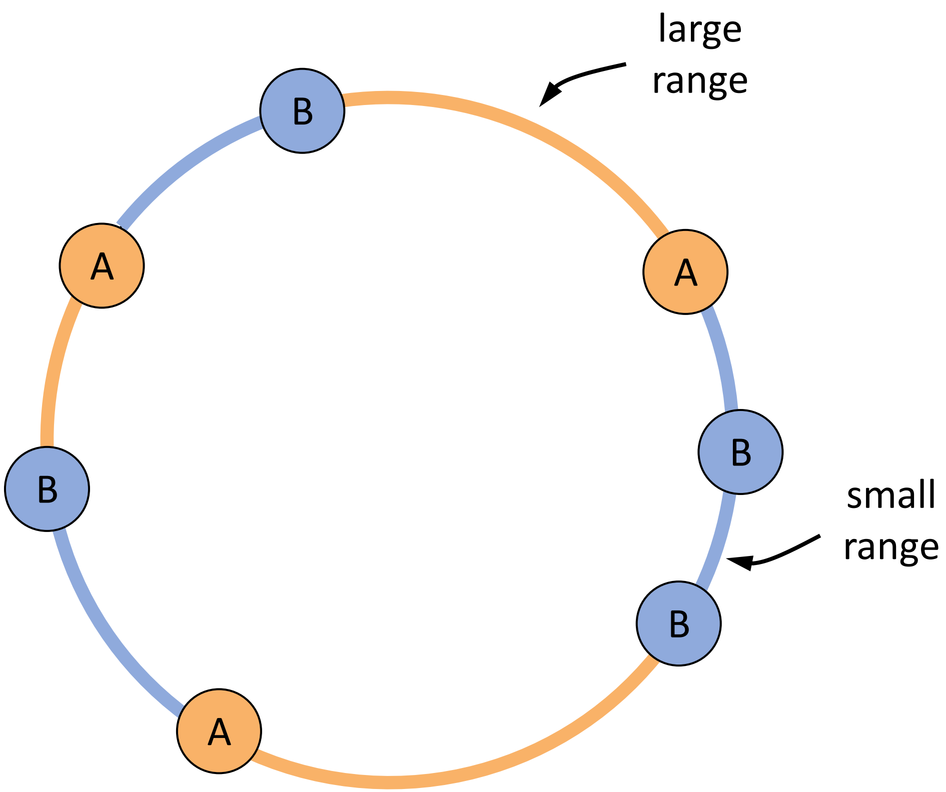
Key load imbalance

- While node IDs are relatively random, key range may be unbalanced => some nodes may store many more keys



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



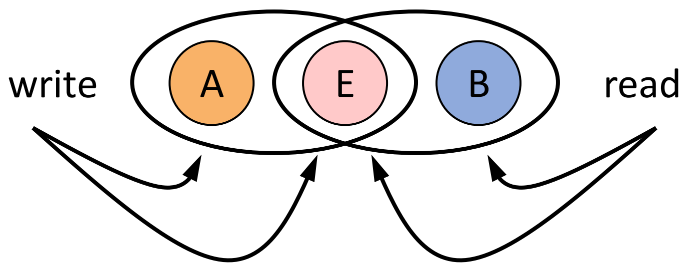
Sloppy Quorum

Why sloppy quorum?

- Goal is to provide both high availability and user-perceived consistency
  - Data should always be 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

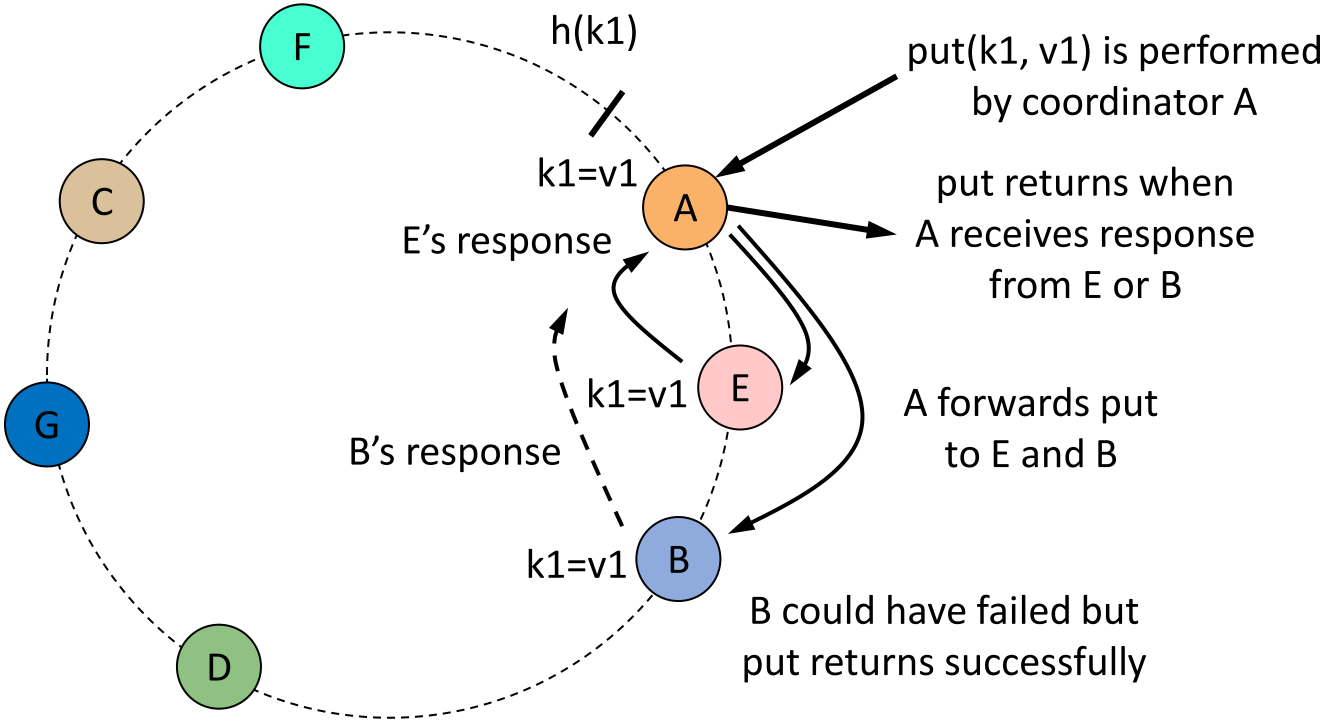


Majority quorum in Dynamo

- Assume N = 3, R = 2, W = 2
- put(k, v)
  - Coordinated by a node that stores key k
    - Typically, first replica is chosen as coordinator
    - 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 put(k1, v1)



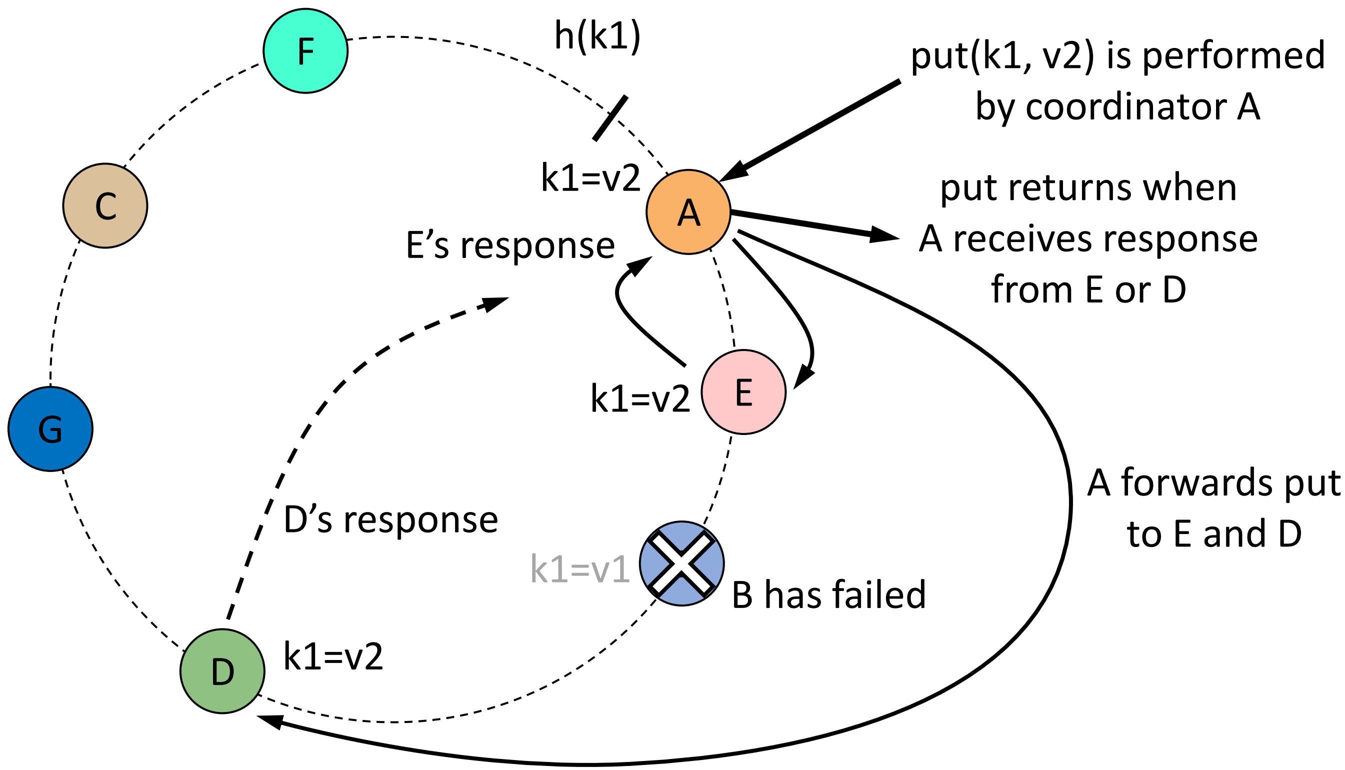
Sloppy quorum

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

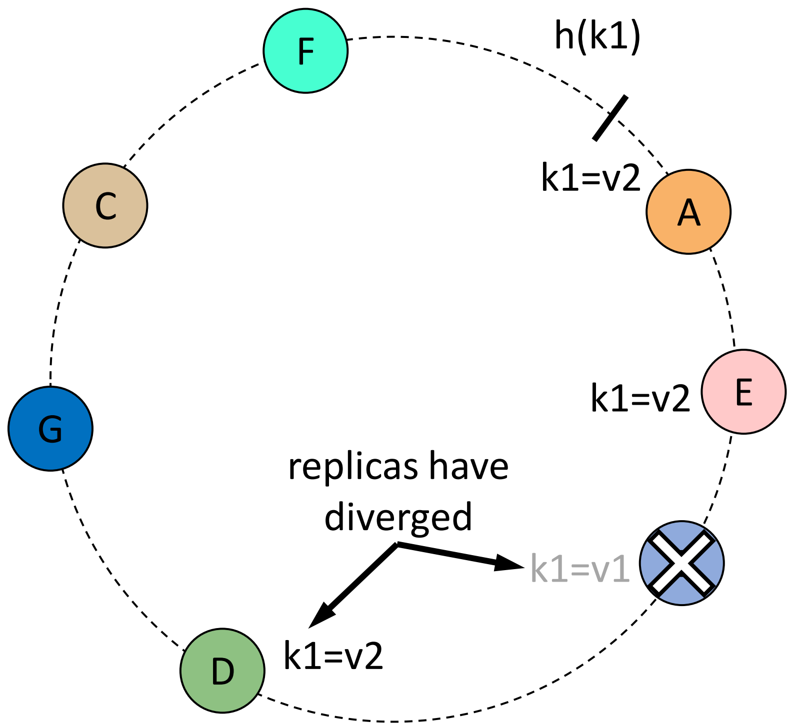
Sloppy quorum example

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



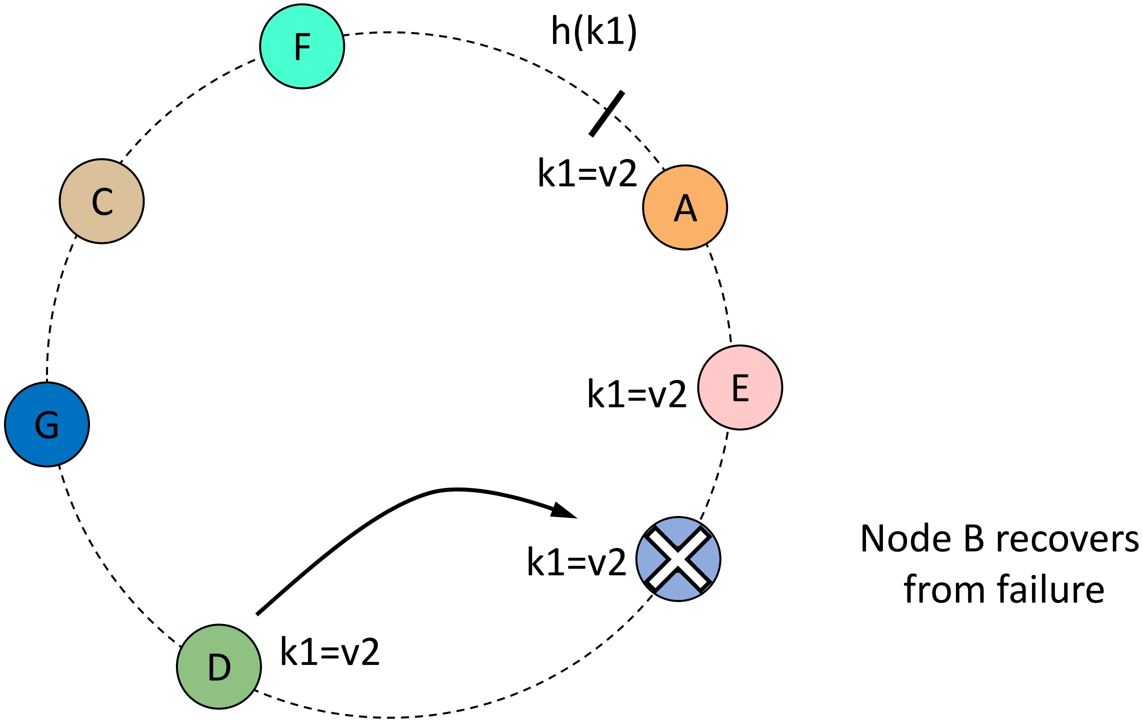
Sloppy quorum and replica divergence

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



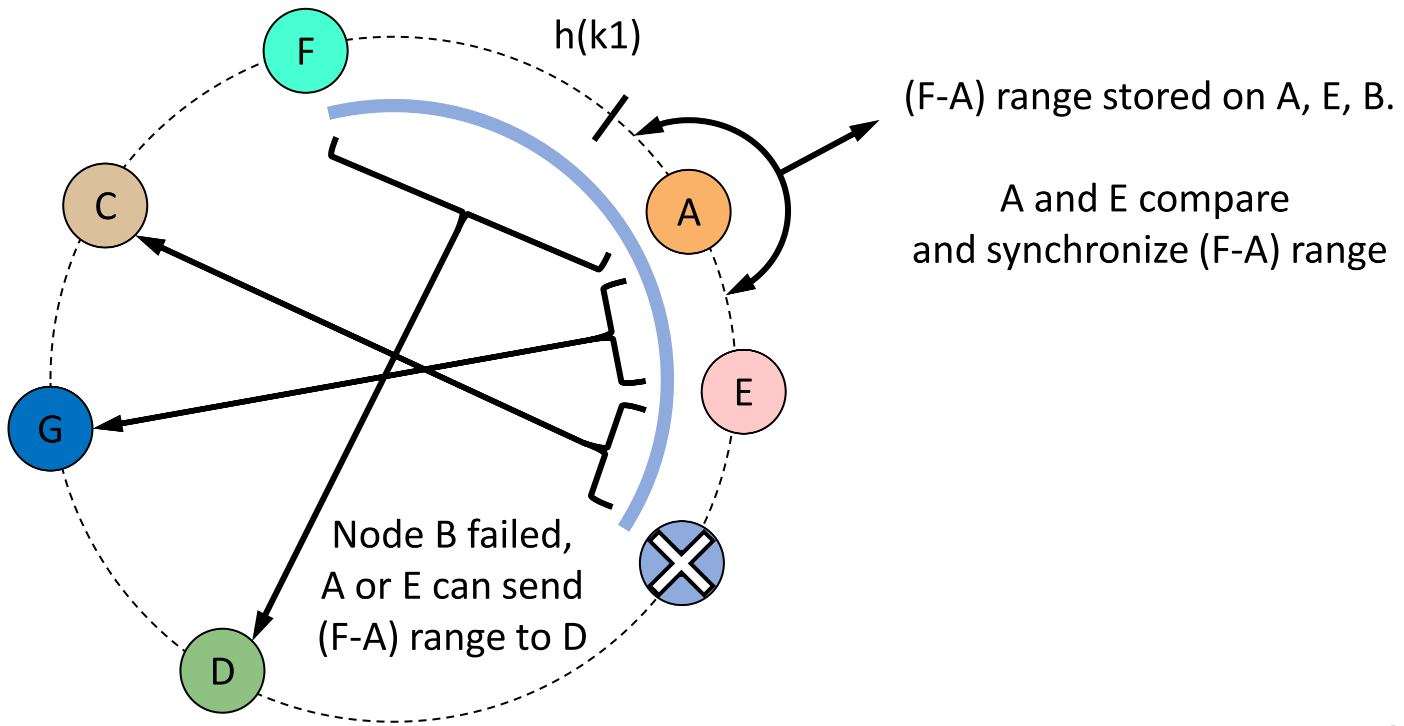
Sloppy quorum and failure recovery

- 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 synchronize key ranges 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

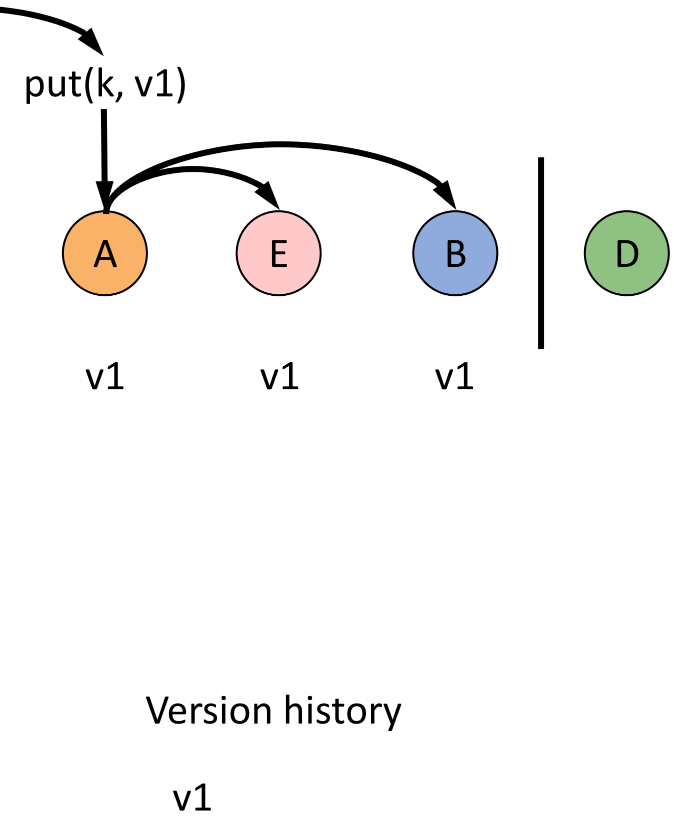
Optimistic Replication

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

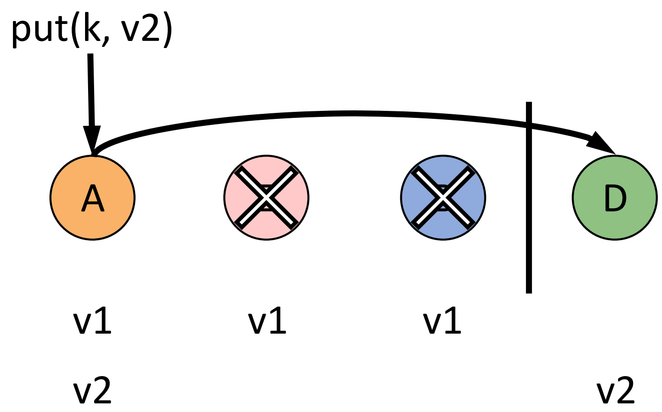
Optimistic replication example

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

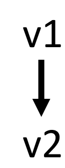


Example

- Say, B and E fail
- 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

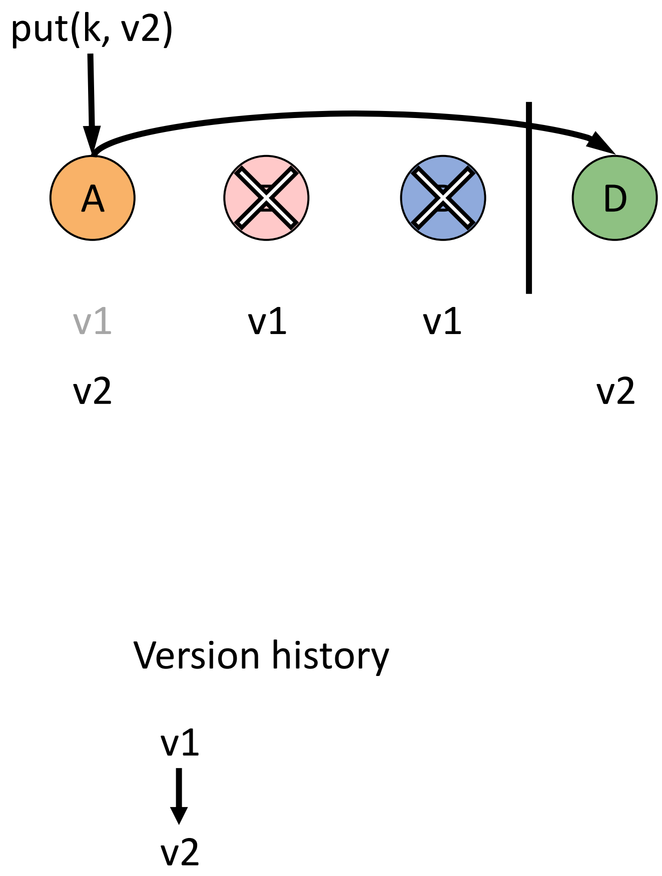


Version history



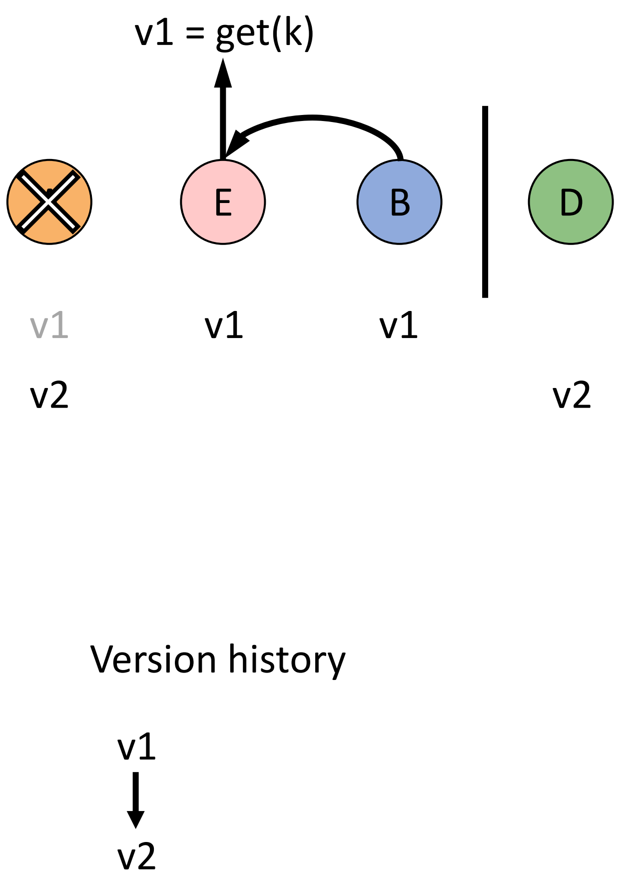
Example

- Say, B and E fail
- 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)



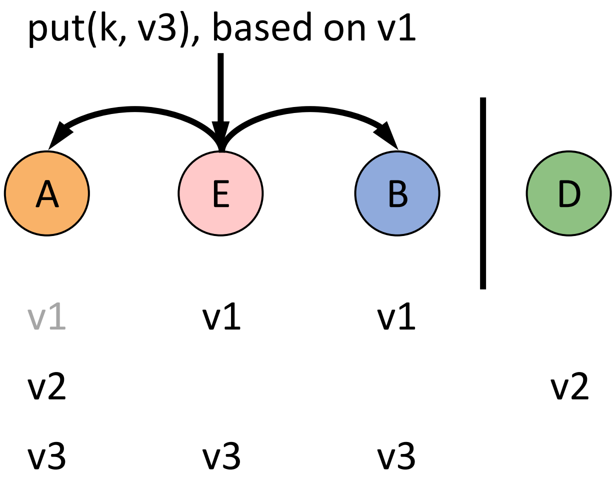
Example

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

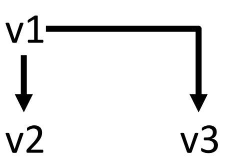


Example

- 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

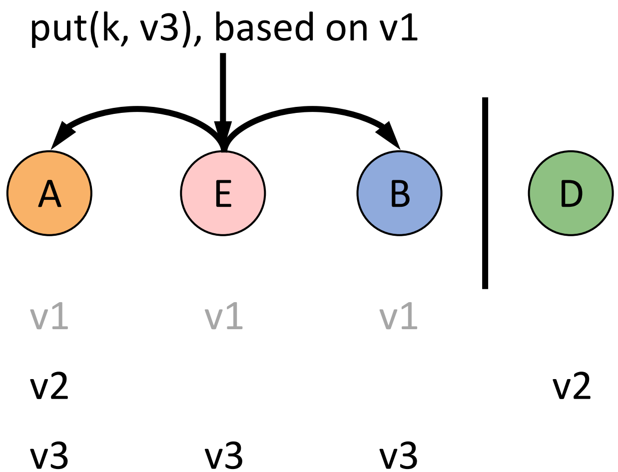


Version history

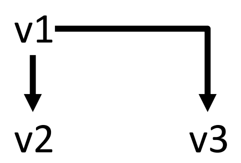


Example

- 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 only store leaf versions in version history
  - E and B remove v1, ancestor of v3
  - A stores v2 and v3, since they conflict

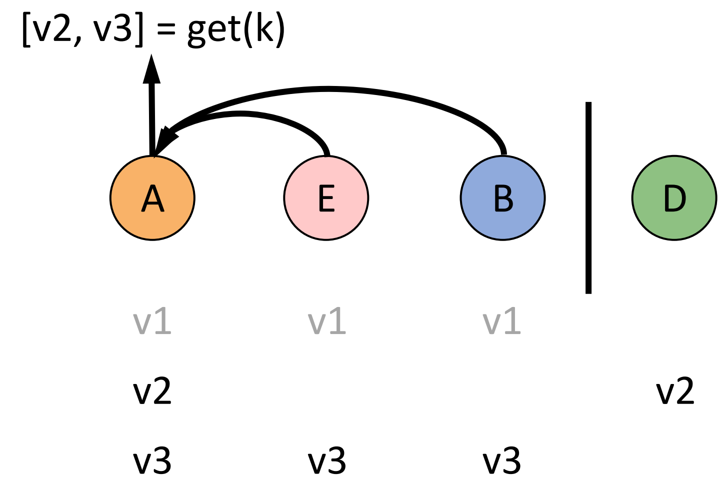


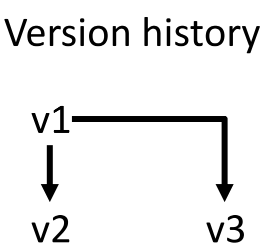
Version history



Example

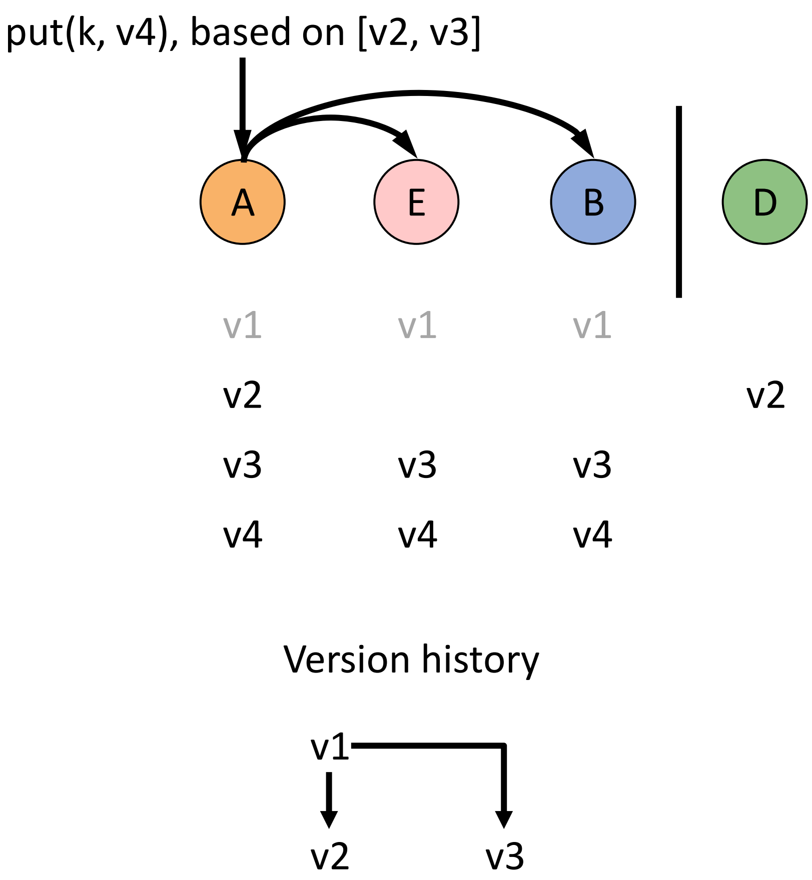
- get(k) reads conflicting [v2, v3] from A, E, B
- Dynamo provides all conflicting versions to client, since client knows best how to reconcile them
  - E.g., app can merge two conflicting shopping carts





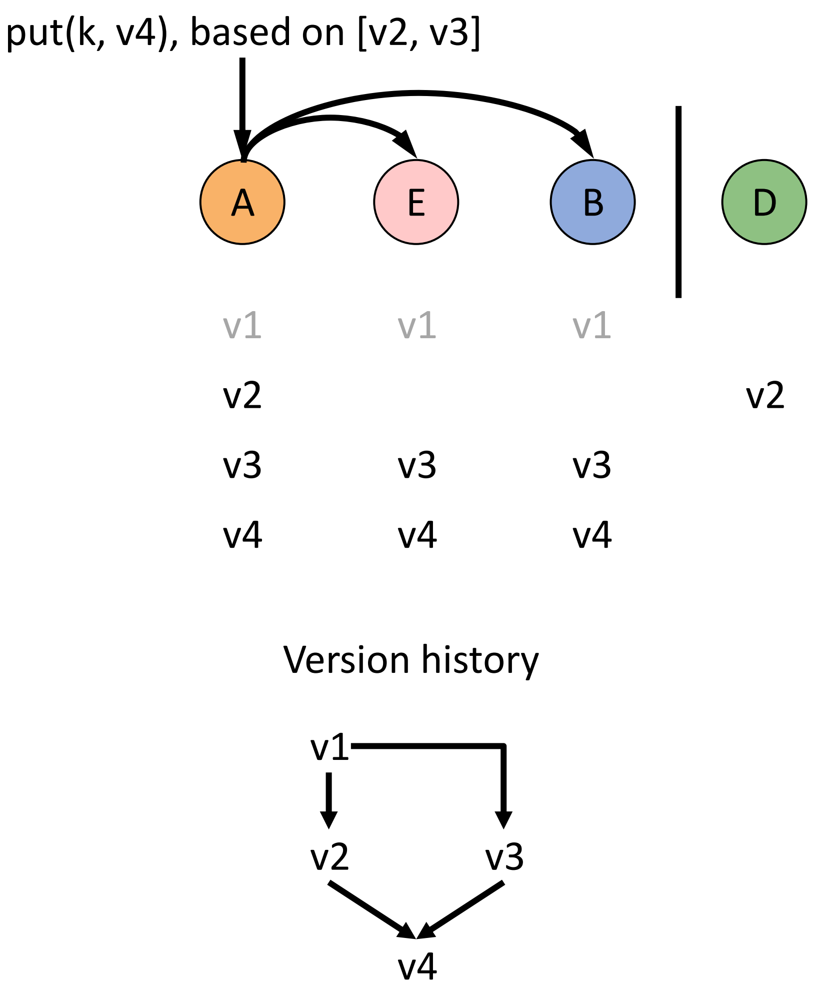
Example

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



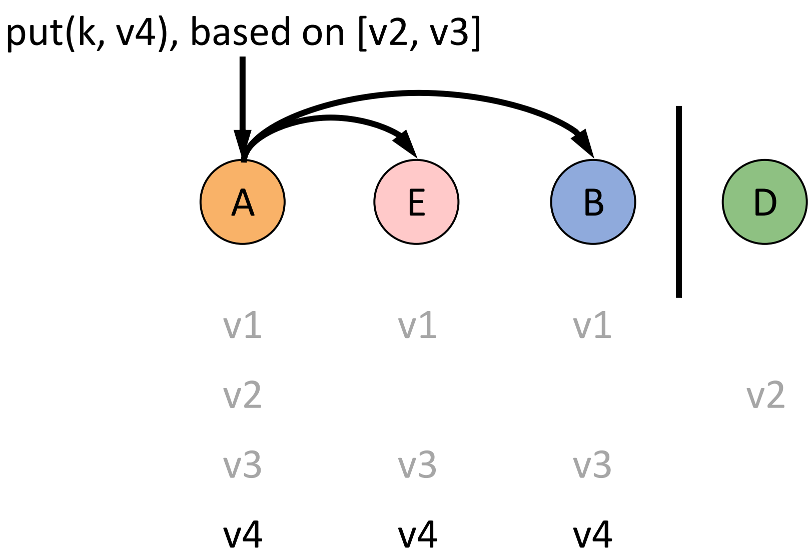
Example

- 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

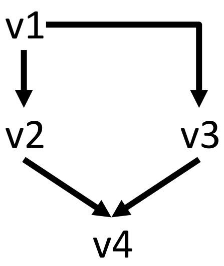


Example

- 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

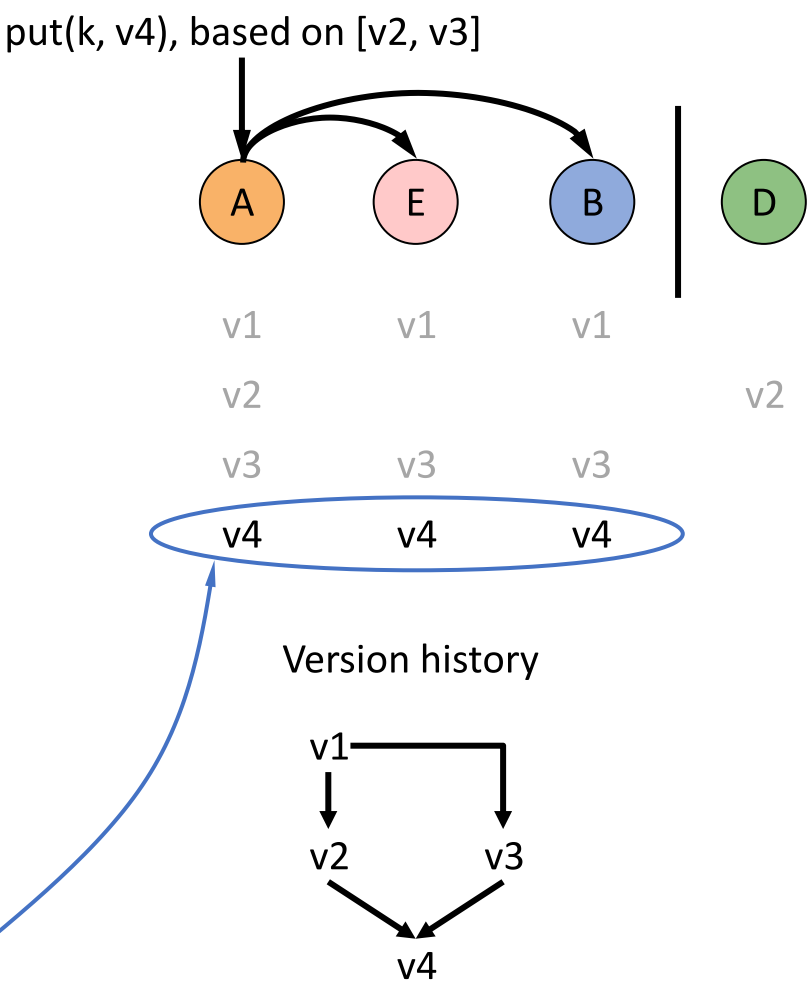


Version history



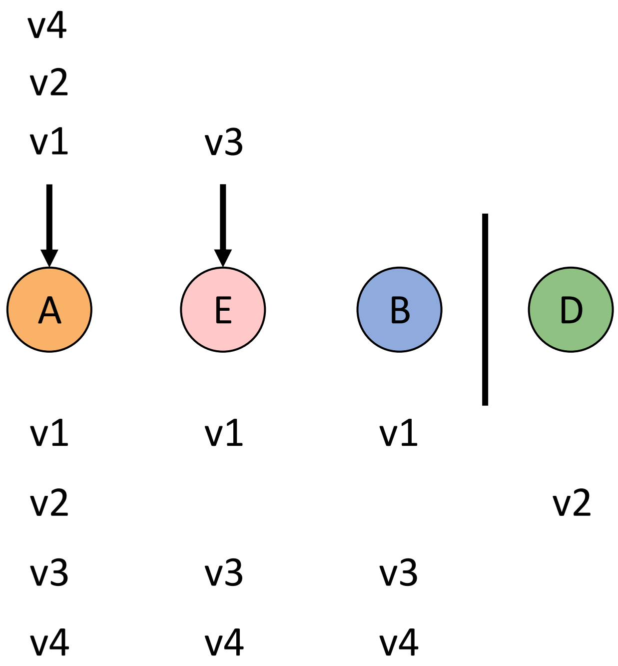
Example

- 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



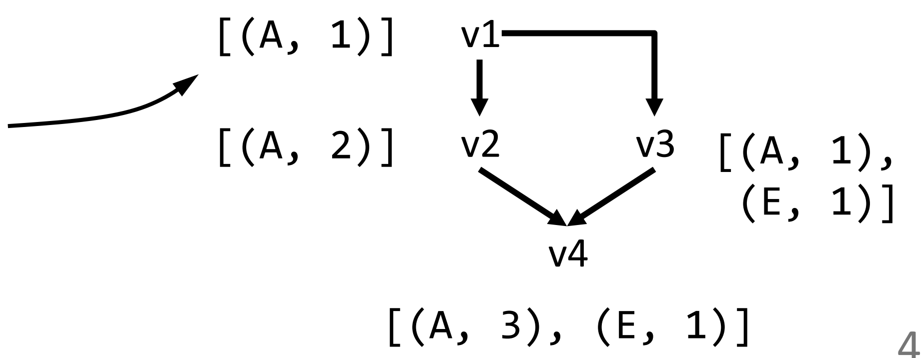
Version history with vector clocks

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




```
VC: [(node1, #updates1),
     (node2, #updates2), …]
```

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 conflicting object versions, and a context.
// context contains vector clock for each returned version.
object[], context = get(key)

// put supplies context returned by previous get.
// context helps generate vector clock for new object version.
put(key, object, context)
```

Gossip-Based Protocols

Membership and mapping

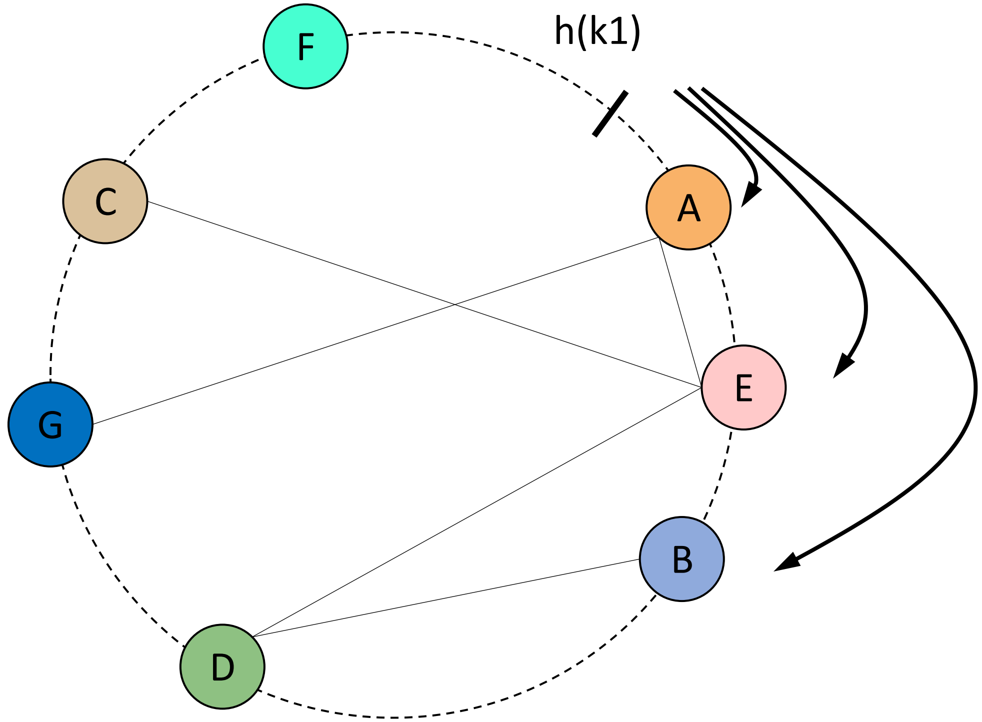
- Dynamo uses gossiping to propagate membership, mapping information
- Administrator explicitly adds and remove nodes
- Membership information: nodes communicate with each other to eventually learn about an added/deleted node
- Mapping information: nodes also learn about node mappings, i.e., the key ranges stored on a node
- What is an alternative method for propagating this information?

Why gossip-based protocols?

- Gossip protocols exchange information between nodes in a peer-to-peer (symmetric) manner
  - A<->B: A and B learn about each other's state
  - B<->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 to become eventually consistent
- Tradeoffs:
  - Pros: avoid need for a coordinator, provide higher availability
  - Cons: nodes may have stale information for a while, limited scaling

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)

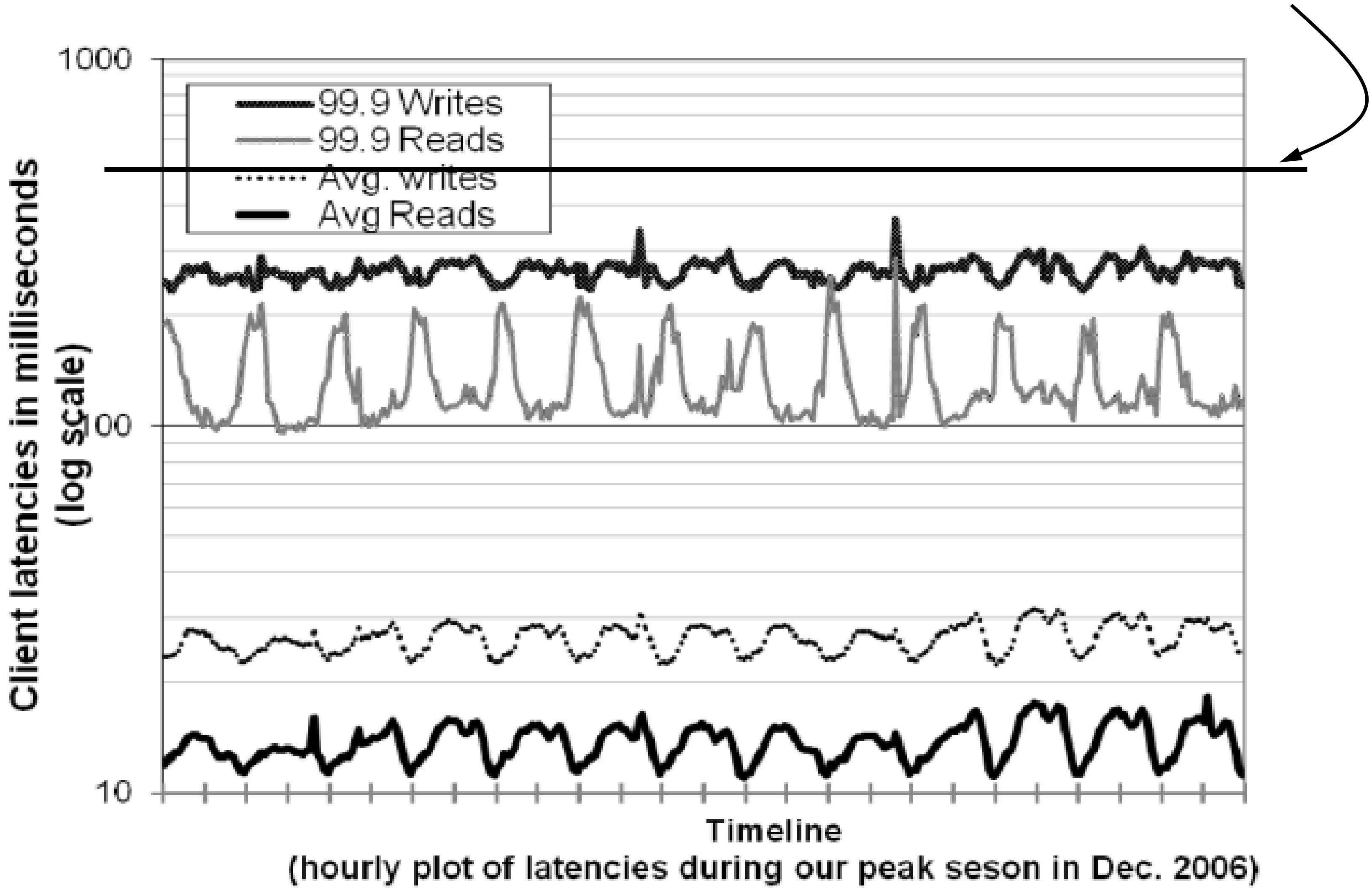


Failure detection

- 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 doesnt respond to a ping message
  - A periodically checks if B is alive again
  - In the absense of requests, A doesn’t need to know if B is alive

Evaluation

500 ms SLA for storage system
for shopping cart application

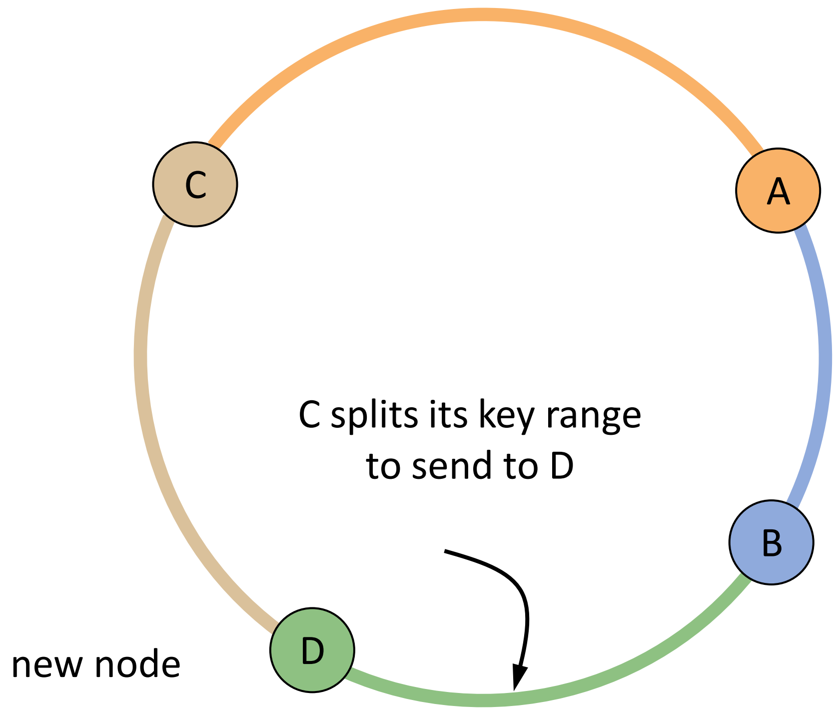


Lessons learned: tail latency

- 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

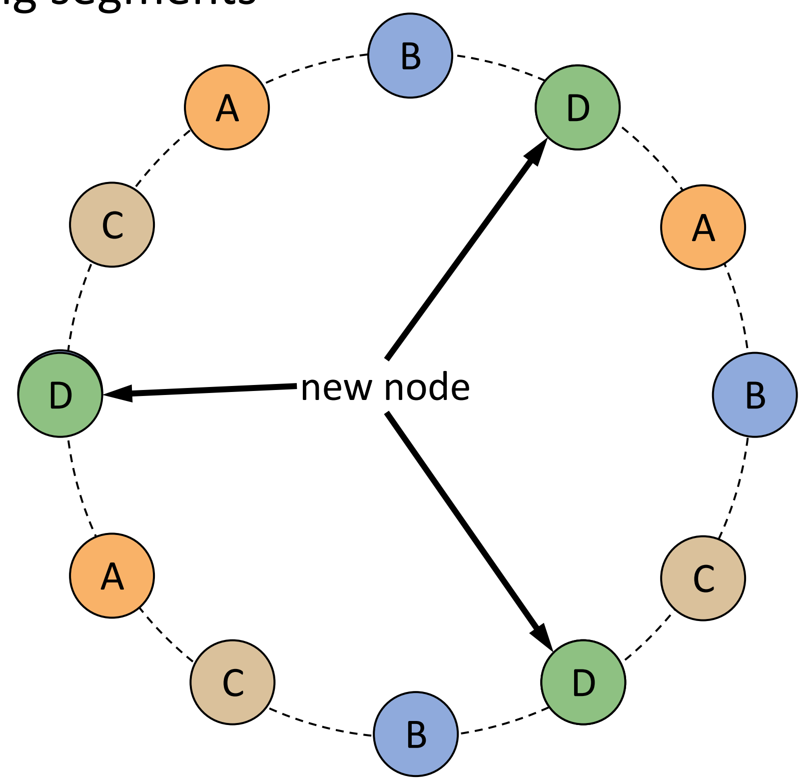
Lessons learned: repartitioning

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

- 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