Data Models
Key/Value: Amazon Dynamo

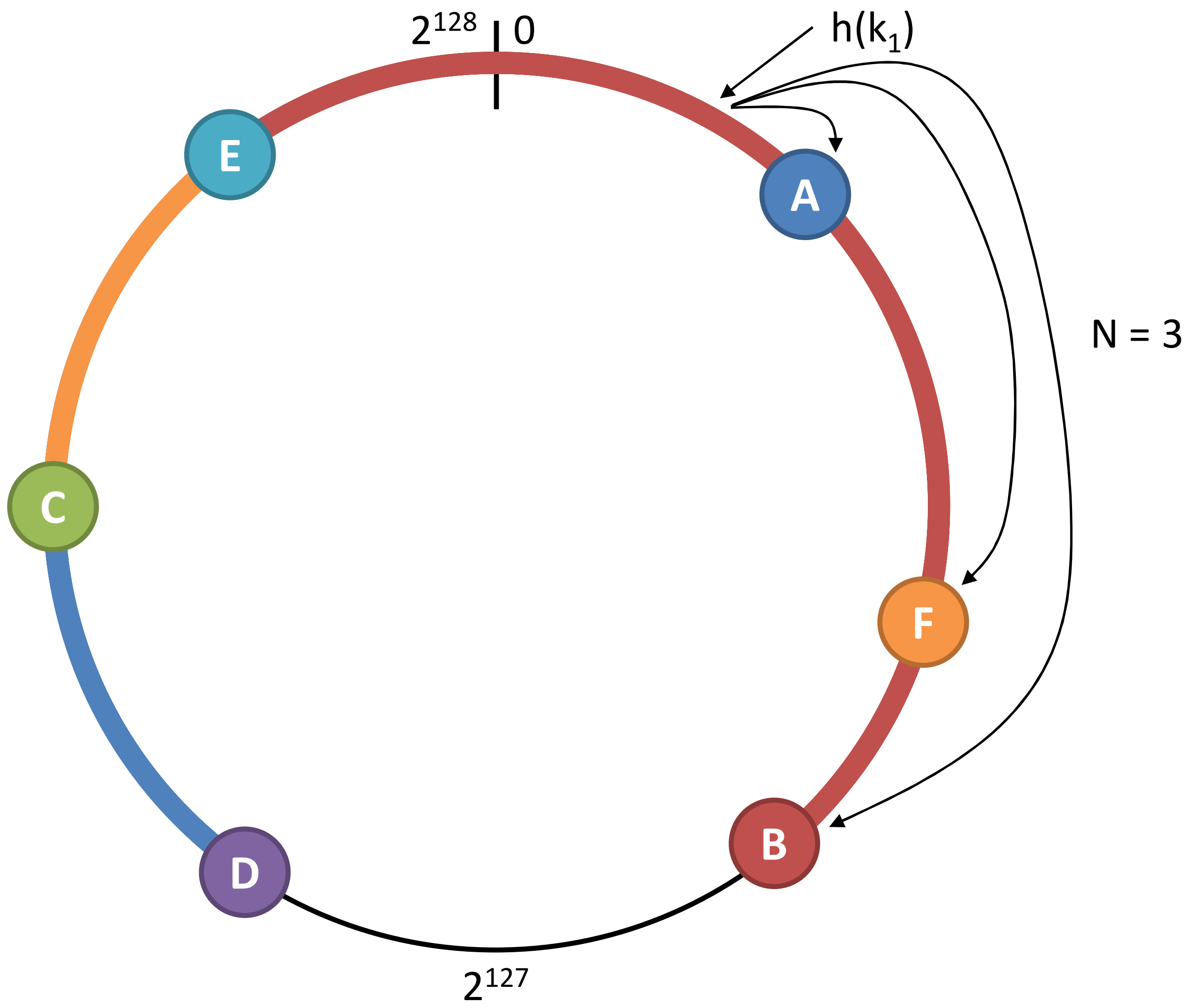
Part 2

Thanks to Dave Maier, M. Grossniklaus & K. Tufte

Replication

- Replication is used to achieve high availability and durability
- Every data item is replicated at *N* hosts
  - *N* is configured "per-instance" of Dynamo
  - each key *k* is assigned a **coordinator** host that handles write requests for *k*
  - coordinator is also in charge of replication of data items within its range
- Algorithm
  1. coordinator stores data item with key *k* locally
  2. coordinator stores data item at *N*-1 clockwise successors nodes
- Every node is responsible for the region of the ring between itself and its N^{th} predecessor
- Preference list
  - enumerates nodes that are responsible for storing a key *k*
  - contains more than *N* nodes to account for node failures

Replication



Data Versioning

- Eventual consistency by asynchronous updates of replicas
  - put() may return to caller before all replicas have been updated
  - subsequent get() may return objects that have not been updated yet
- "Always writable" design
  - result of each modification is a new and immutable version of the data
  - multiple versions may be exist in the system at the same time
  - vector clocks capture causality between different versions of an object
- Version reconciliation
  - **system-based**: most versions simply subsume the previous version
  - **client-based**: if failures combined with concurrent updates lead to branches, the client (app) needs to collapse multiple branches into one
- Context passed between get and put operations contains vector clock information

Data Versioning Evolution

1. Client *A*
   – writes new object D
   → node *Sx* writes version D1
2. Client *A*
   – updates object D
   → node *Sx* writes version D2
3. Client *A*
   – updates object D
   → node *Sy* writes version D3
4. Client *B*
   – reads and updates object D
   → node *Sz* writes version D4
5. Client *C*
   – reads object D (i.e., D3 and D4)
   – client performs reconciliation
   → node *Sx* writes version D5

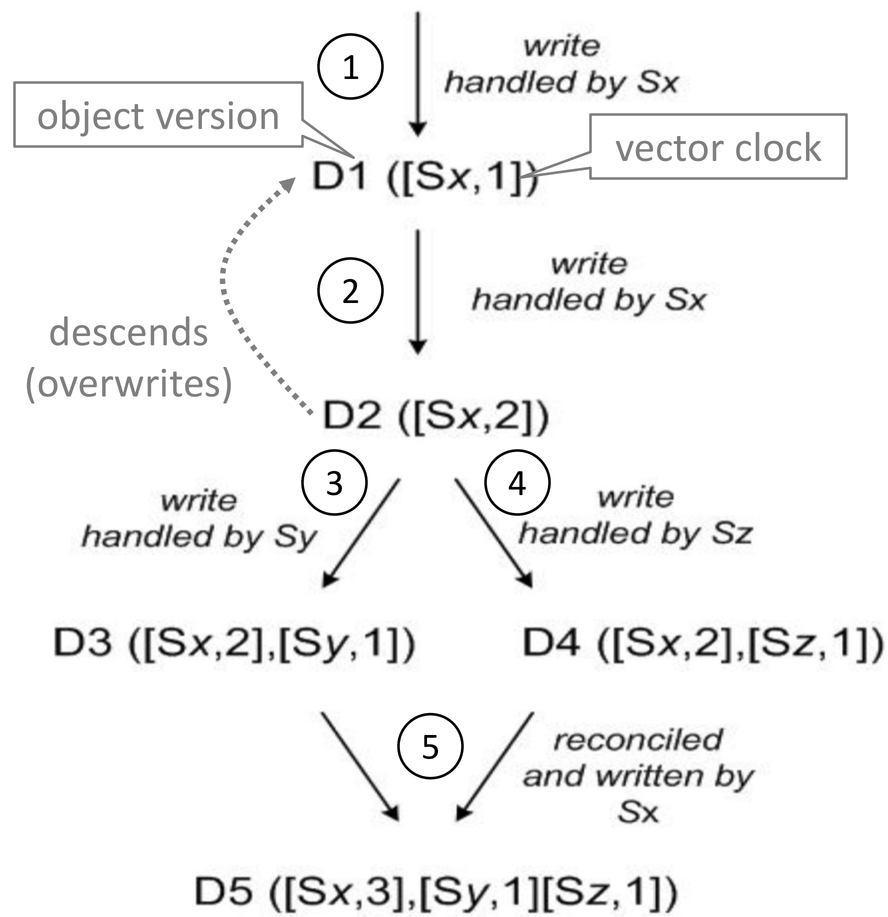


Figure Credit: G. DeCandia et al.: Dynamo: Amazon's Highly Available Key-Value Store, Proc. SOSP, 2007.

PQ1: How Would You Reconcile Two Carts?

Item	Qty	Price-Each
Mixer	1	199.00
Tea	3	7.50
Teapot	1	23.25
Spatula	4	3.30
Strainer	1	12.80

Item	Qty	Price-Each
Mixer	1	199.00
Tea	2	7.50
Spatula	4	3.30
Strainer	2	12.80
Kettle	1	22.65

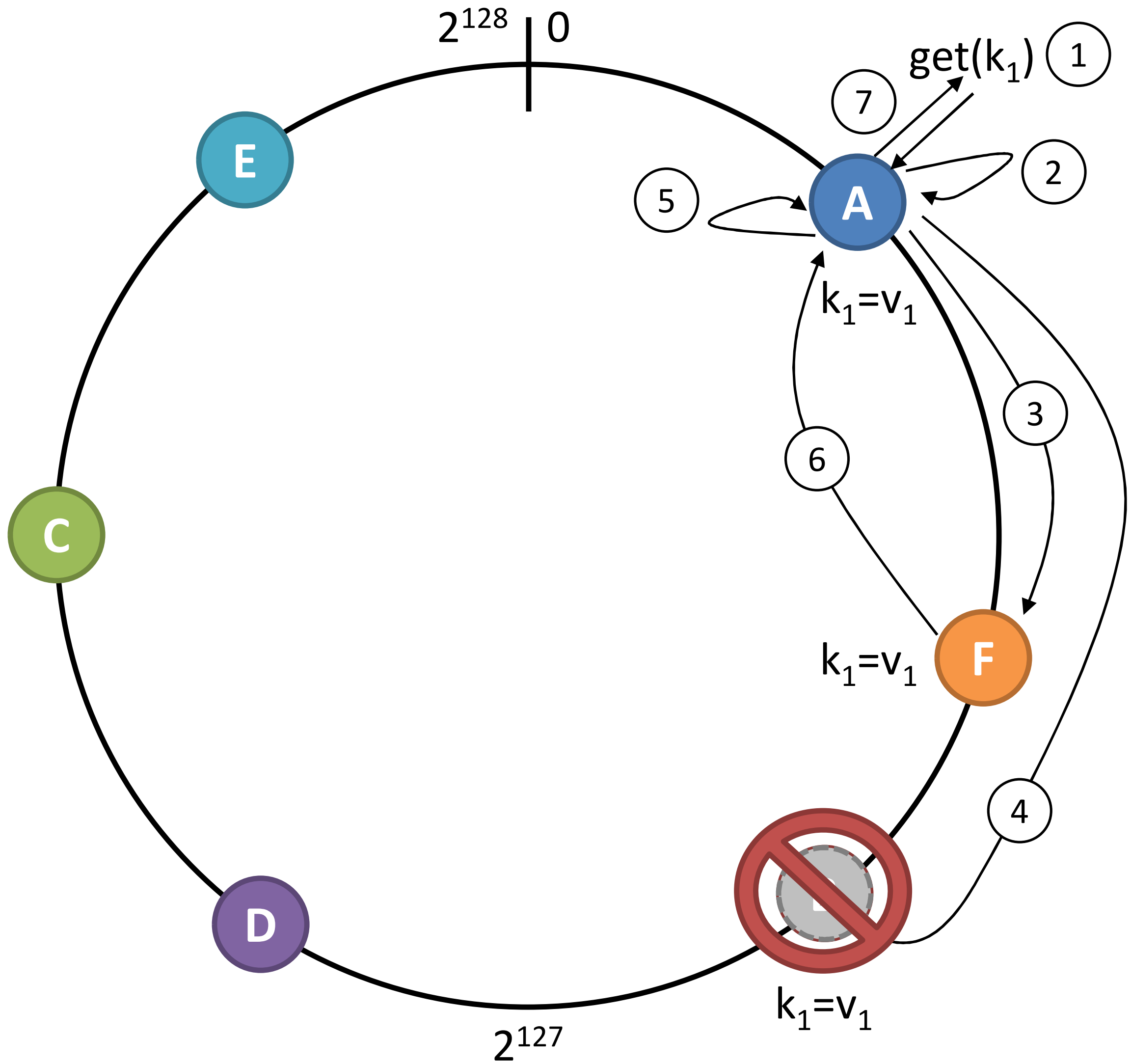
What was your general strategy?

Execution of Get and Put Operations

- **Coordinator**: node that handles read or write operation
  - typically the first of the top *N* nodes of the preference list for a write
  - requests that hit a node that is not in the top *N* of the requested key's preference list are forwarded to the appropriate coordinator
- Quorum-like Protocol
  - R: minimum number of nodes that must participate in successful read
  - W: minimum number of nodes that must participate in successful write
- "Sloppy Quorum"
  - read and write operations are performed on the first *N* **healthy** nodes
  - not guaranteed be the first *N* nodes encountered while walking the ring
- Hinted Handoff
  - if a node is unreachable, a **hinted** replica is sent to next healthy node
  - nodes receive hinted replicas keep them in a separate database
  - hinted replica is delivered to original node when it recovers

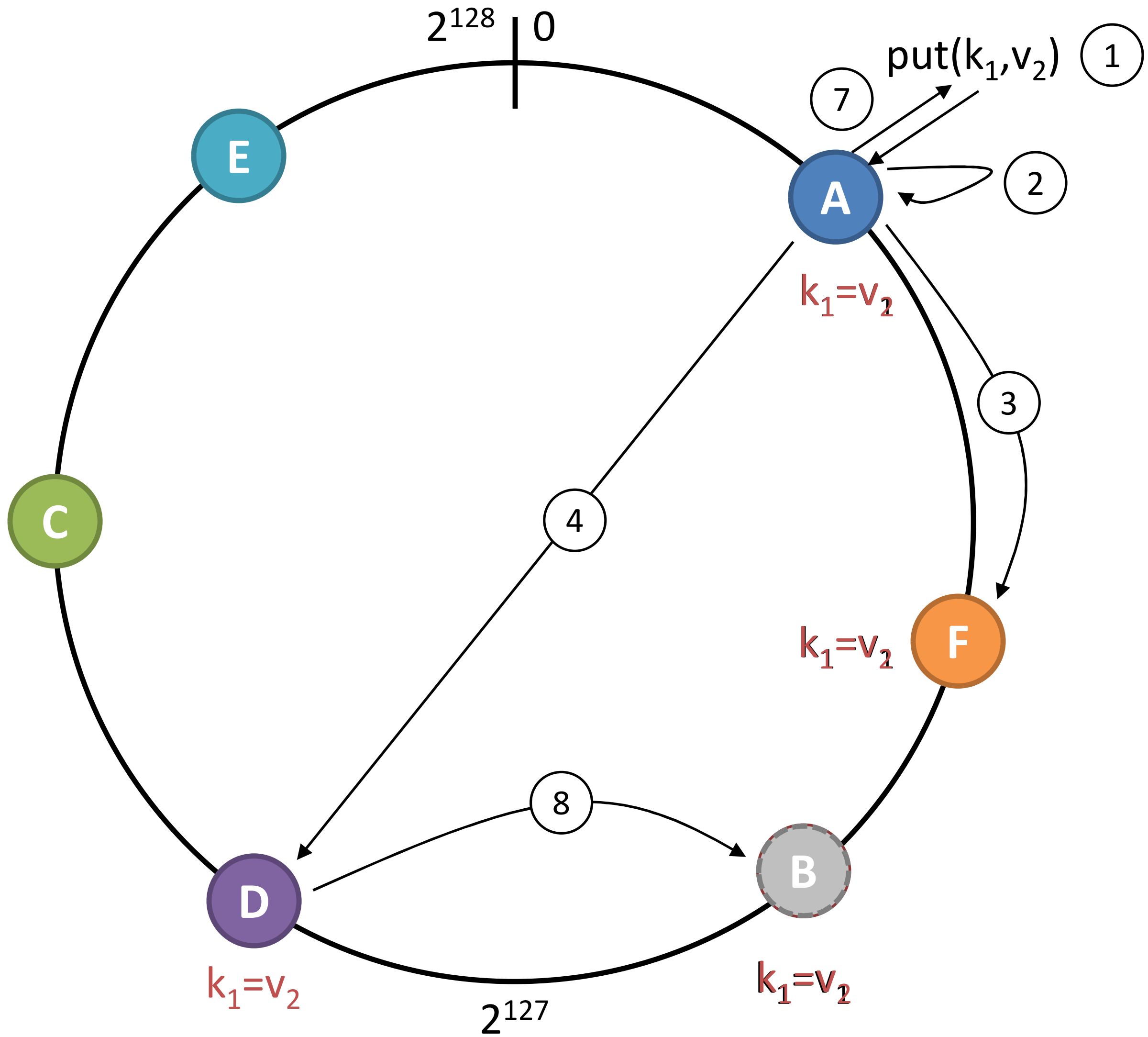
Execution of Get Operation

N = 3
R = 2
W = 2



Execution of Put Operation

N = 3
R = 2
W = 2

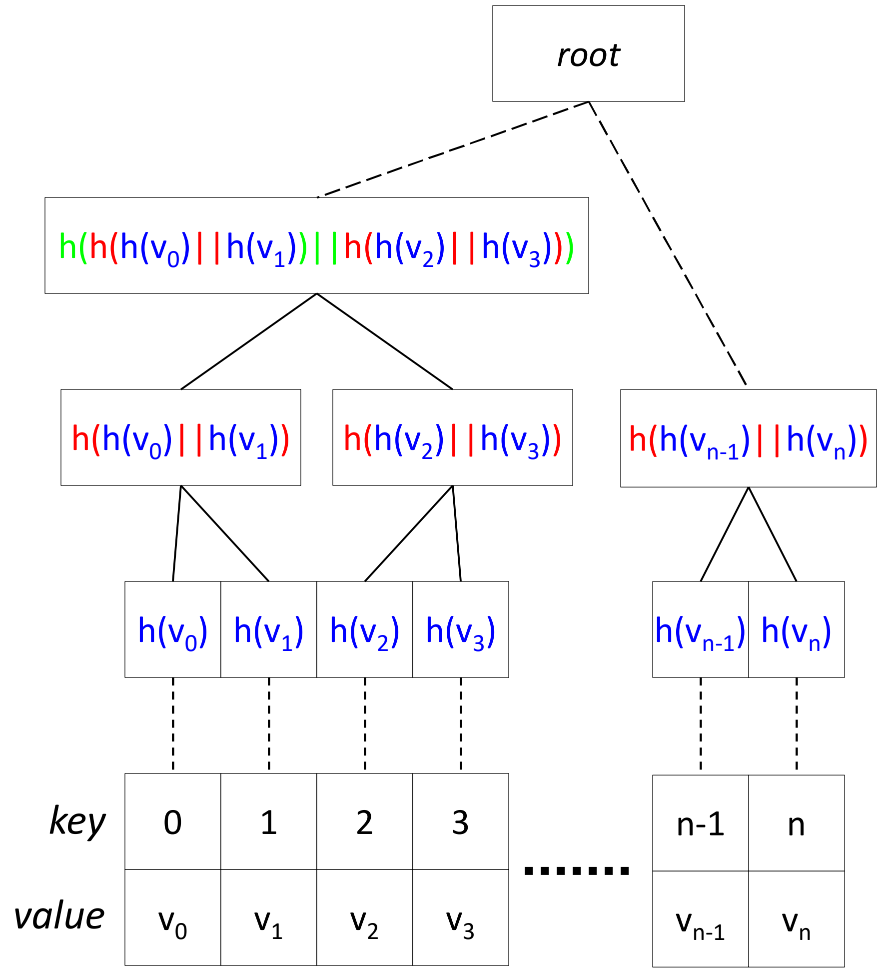


Replica Synchronization

- Scenarios exist where hinted replicas become unavailable before being returned to the original node
  - Dynamo implements an anti-entropy (replica synchronization) protocol
  - Dynamo uses Merkle trees to detect inconsistencies between replicas
- Algorithm
  - every physical node maintains a separate Merkle tree for each hosted key range, i.e., the set of keys covered by a virtual host
  - to check if their key ranges are up-to-date, two nodes exchange the roots of the Merkle tree of the key ranges they have in common
  - if the value of the roots are equal, the key ranges are up-to-date
  - if not, they recursively exchange the values of the children until they reach the leaves of the Merkle tree
  - at that point, the inconsistent keys are identified and the corresponding values can be exchanged

Merkle Tree

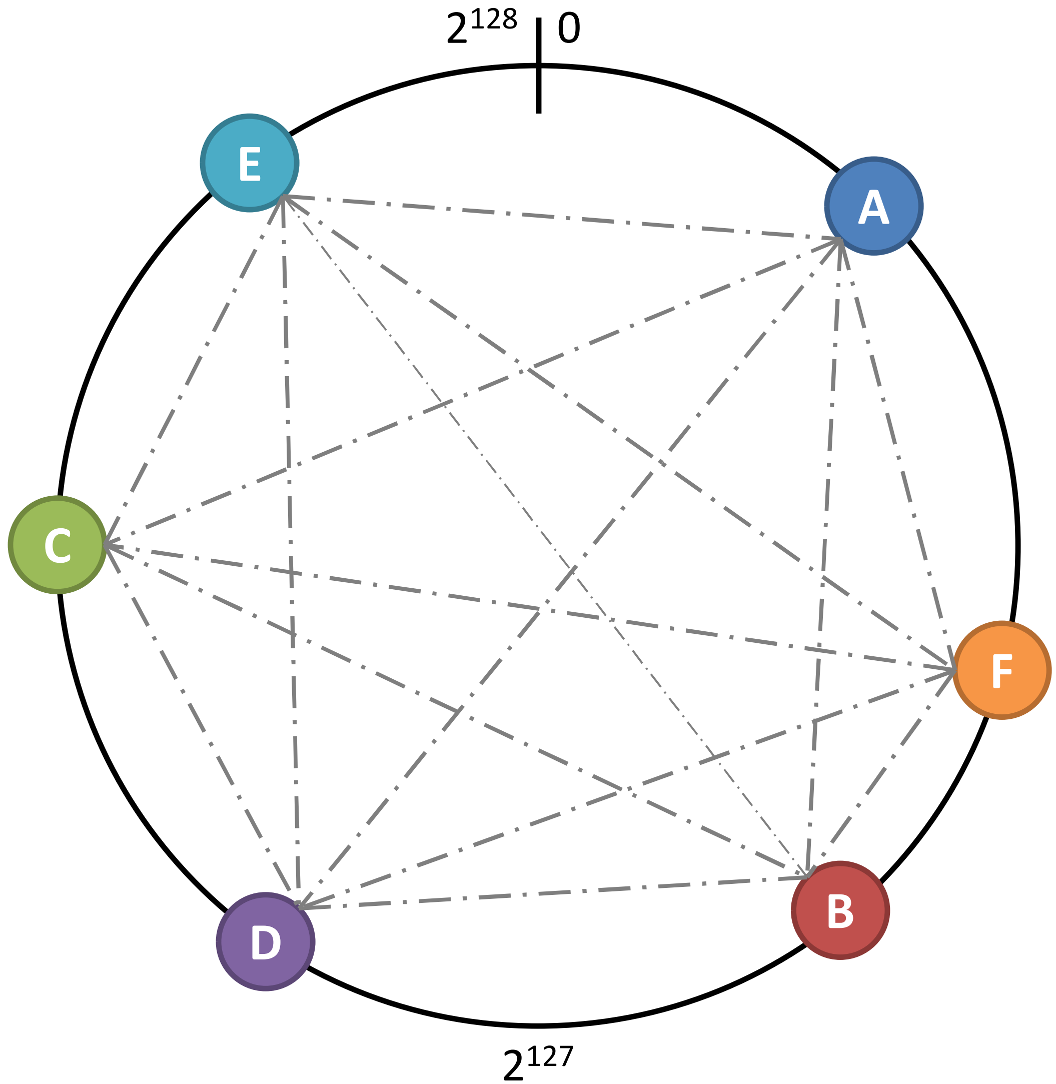
- Hash tree
  - **leaves** are hashes of the values of individual keys
  - **non-leaves** are hashes of their respective children nodes
- Properties
  - efficient verification of large data structures
  - every branch can be processed independently
- Applications
  - ZFS file systems
  - Git revision control system
  - NoSQL systems (e.g., Dynamo, Cassandra, and Riak)



Membership and Failure Detection

- Explicit mechanism to initiate addition and removal of nodes
  - outage or failure rarely signifies permanent departure of node
  - manual error could result in unintentional startup of node
  - these events should not result in rebalancing of partition assignment or repair of unreachable replica
- Full membership model
  - administrative command issued to join/remove a node to/from ring
  - node that handles request updates its persistent membership table
  - gossip-based protocol propagates membership changes
- Failure detection
  - communication failures avoided based on a purely local failure notion
  - globally consistent view on failure state is not required thanks to explicit join/remove commands

Full Membership Model



PQ2: Membership Management and Failure Detection

Your team is managing a **distributed online ticket booking system** that relies on DynamoDB.

A **server failure** has been detected, and you must decide how to handle **membership changes and failure recovery**.

Discuss the following:

1. **Why does Dynamo use an explicit join/remove command instead of automatic failure detection for managing membership?**
2. **What advantages does a gossip-based protocol offer for updating membership across nodes?**
3. **How does hinted handoff help maintain availability during temporary node failures? What happens if the hinted replica is lost?**

Prepare to present your team's insights.

Implementation

Three main software components

1. Local persistence engine
   - plug-in architecture supports different storage engines
   - storage engine chosen based on application’s object size distribution
   - *BerkeleyDB*: objects in the order of tens of kilobytes
   - *MySQL*: larger objects
2. Request coordination
   - built on top of an event-driven messaging infrastructure
   - communication implemented using Java NIO channels
   - client requests create a state machine on node that received request
   - each state machine handles exactly one client request
3. Membership and failure detection

Configurability

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

Figure Credit: Peter Vosshall, Amazon.com, 2007.

Typical Usage Patterns

- Business logic-specific reconciliation
  - each data object is replicated across multiple nodes
  - client applications performs reconciliation in case of divergent versions
  - **example**: merging different versions of a customer's shopping cart
- Timestamp-based reconciliation
  - Dynamo performs simple timestamp-based reconciliation
  - "last write wins", i.e., object with largest physical times is selected
  - **example**: maintaining a customer's session information
- High-performance read engine
  - R = 1, W = N
  - high read request rate with only a small number of updates
  - **example**: management of product catalog and promotional items