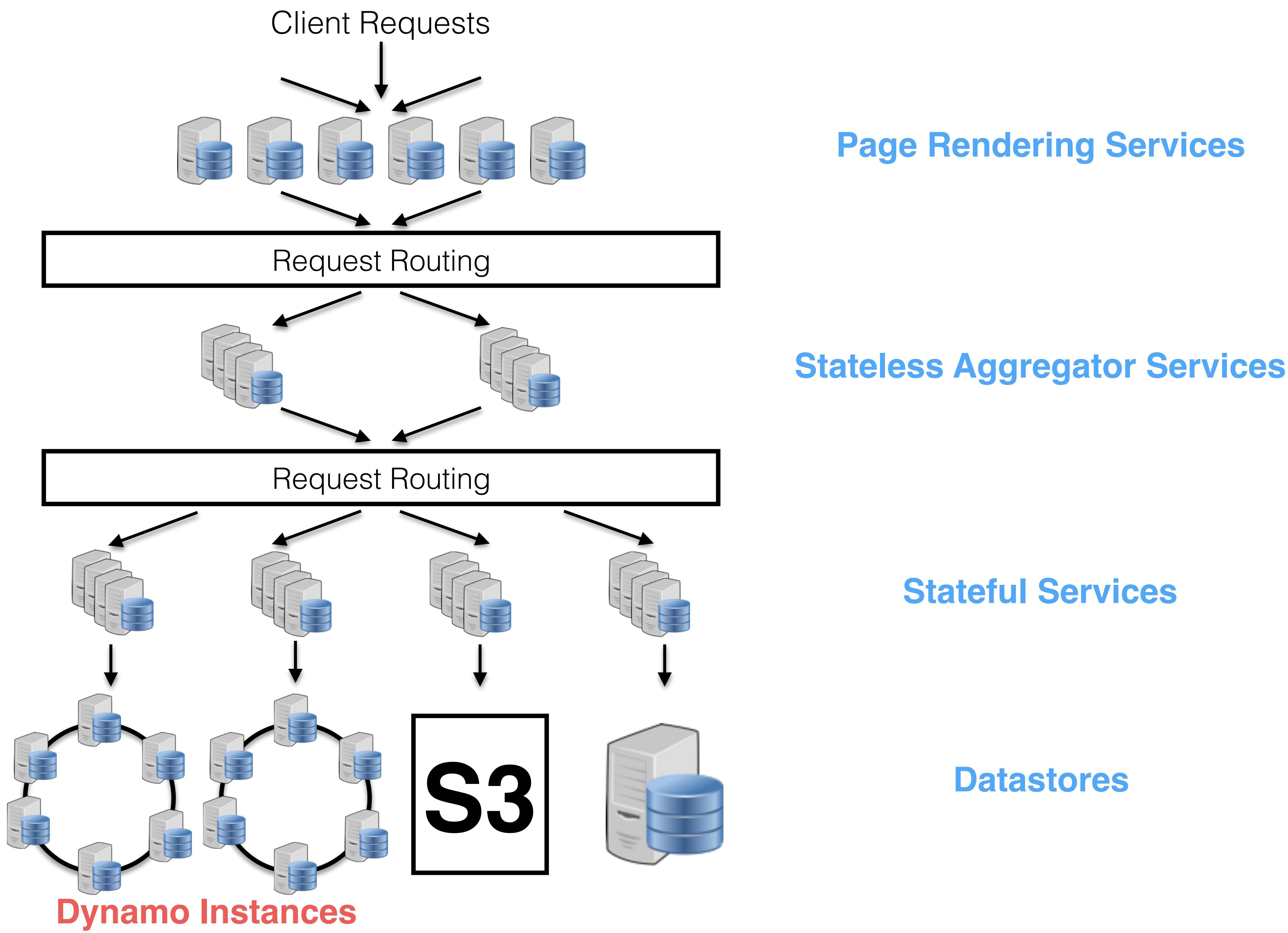
Dynamo

- Service providing persistent data storage to other services running on Amazon's distributed computing infrastructure.
- Dynamo: Amazon's Highly Available Key-value Store, SOSP 2007
- First paper discussing implementation of weak consistent, highly available data replication service

Amazon's Platform Architecture



Design Issues

- Replication for high availability and durability
  - replication technique: synchronous or asynchronous?
  - conflict resolution: when and who?
- Dynamo’s goal is to be “always writable”
  - rejecting writes may result in poor Amazon customer experience
  - data store needs to be highly available for writes, e.g., accept writes during failures and allow write conversations without prior context
- Design choices
  - optimistic (asynchronous) replication for non-blocking writes
  - conflict resolution on read operation for high write throughput
  - conflict resolution by client for user-perceived consistency

Design Principles

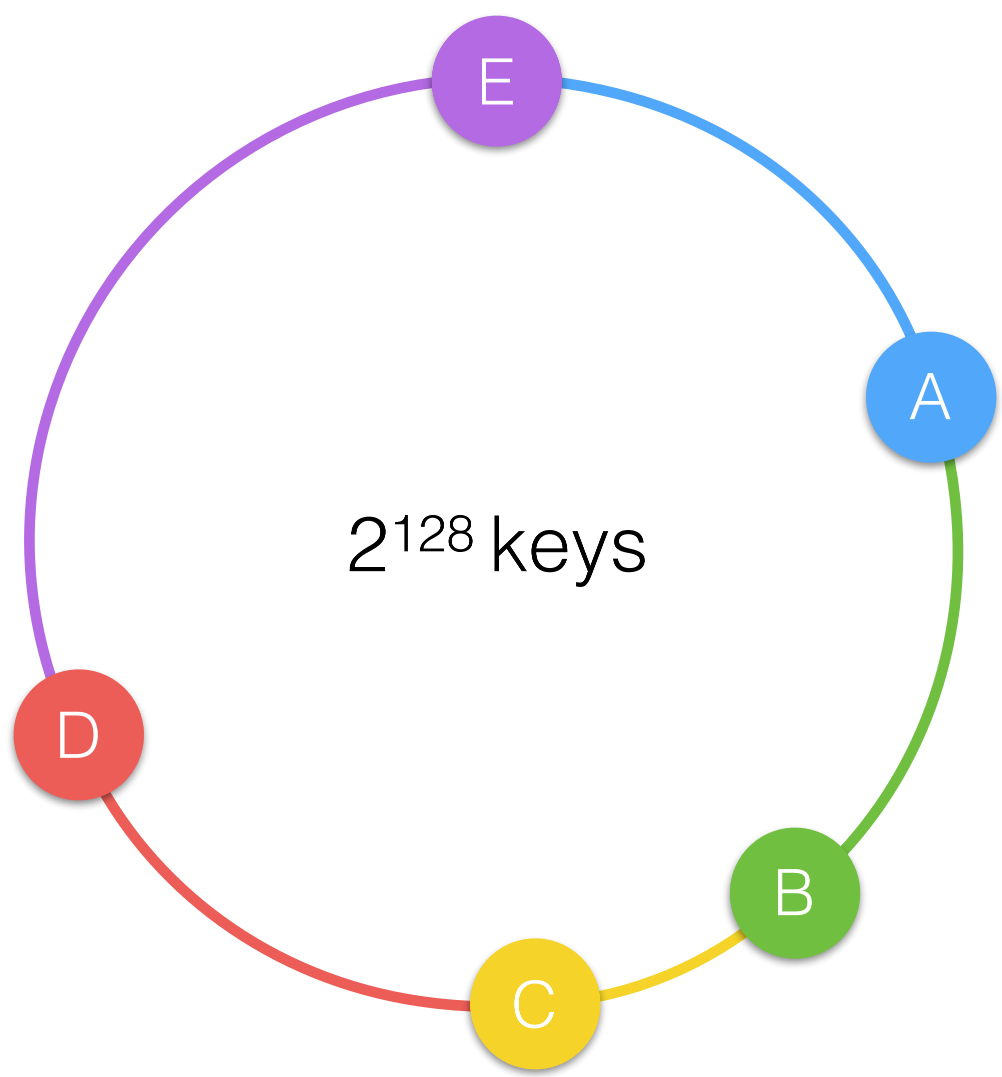
- Incremental Scalability
  - scale out (and in) one node at a time
  - minimal impact on both operators of the systems and the system itself
- Symmetry
  - every node should have the same set of responsibilities
  - no distinguished nodes or nodes that take on a special role
  - symmetry simplifies system provisioning and maintenance
- Decentralization
  - favor decentralized peer-to-peer techniques
  - achieve a simpler, more scalable, and more available system
- Heterogeneity
  - exploit heterogeneity in the underlying infrastructure
  - load distribution proportional to capabilities of individual servers
  - adding new nodes with higher capacity without upgrading all nodes

Interfaces

- Data objects (blobs) are identified through an hash key
- Two single object operations, no group operations
- Read operation: GET(K) → [OBJECT(S), CONTEXT]
    - Locate object replicas associated with key k
    - Return single object or multiple conflicting version of the same object with context
- Write operation: PUT(K, OBJECT, CONTEXT)
    - Determine where the replicas should be placed based on key k
    - Write the replicas to disks
- Context
    - System metadata, opaque to the caller
    - Include information on object, such as versioning

Partitioning

- Consisten Hashing
  - Load distribution
  - Arrival and departures have neighbour impact



Replication

- 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
- Coordinator stores data item with key *k* locally
- Coordinator stores data item at *N-1* clockwise successors nodes
- Every node is responsible for the region of the ring between itself and its predecessor
- *Preference list* enumerates nodes that are responsible for storing a key *k* – contains more than *N* nodes to account for node failures

Data Versioning

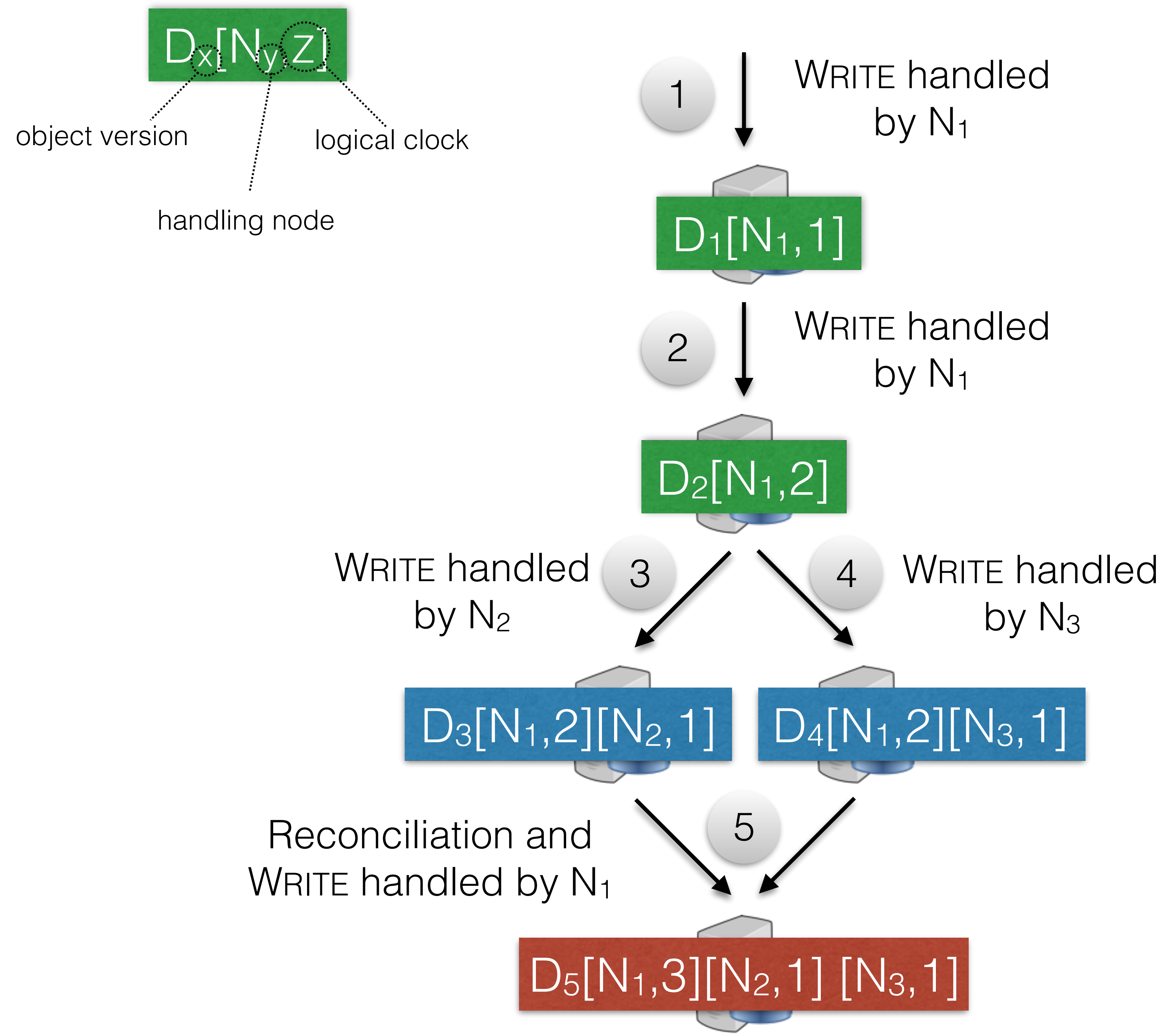
- Replication performed after a response is sent to a client
  - This is called asynchronous replication
  - May result in inconsistencies under network partitions
- But operations should not be lost
  - “add to cart” should not be rejected but also not forgotten
  - If “add to cart” is performed when latest version is not available it is performed on an older version
  - We may have different versions of a key/value pair
- Once a partition heals versions are merged
  - The goal is not to lose any “add to cart”
- Data versioning technique: Vector clocks
  - Capture causality between different versions of an object

Vector Clocks

- Each write to a key *k* is associated with a vector clock VC(k)
- VC(k) is an array (map) of integers
  - In theory: one entry VC(k)[i] for each node i
- When node i handles a write of key k it increments VC(k)[i]
  - VCs are included in the context of the put call
- In practice:
  - VC(k) will not have many entries (only nodes from the preference list should normally have entries), and
  - Dynamo truncates entries if more than a threshold (say 10)

Data Versioning

1. Client A
   - writes new object D
   - node N_1 writes version D_1
2. Client A
   - updates object D
   - node N_1 writes version D_2
3. Client A
   - updates object D
   - node N_2 writes version D_3
4. Client B
   - reads and updates object D
   - node N_3 writes version D_4
5. Client C
   - reads object D (i.e., D_3 and D_4)
   - client performs reconciliation
   - node N_1 write version D_5



Handling PUT/GET

- Any node can receive GET/PUT request for any key.
- This node is selected by
  - Generic load balancer
  - By a client library that immediately goes to coordinator nodes in a preference list
- If the request comes from the load balancer
  - Node serves the request only if in preference list
  - Otherwise, the node routes the request to the first node in preference list
- Each node has routing info to all other nodes
- Extended preference list: N nodes from preference list + some additional nodes (following the circle) to account for failures
  - Failure-free case: nodes from preference list are involved in get/put
  - Failures case: first N alive nodes from extended preference list are involved

R and W

- R = number of nodes that need to participate in a GET
- W = number of nodes that need to participate in a PUT
- R + W > N (a quorum system)
- Handling PUT (by coordinator) // rough sketch
  1. Generate new VC, write new version locally
  2. Send value and VC to N selected nodes from preference list
  3. Wait for W -1
- Handling GET (by coordinator) // rough sketch
  1. Send read to N selected nodes from preference list
  2. Wait for R
  3. Select highest versions per VC, return all such versions
  4. Reconcile/merge different versions
  5. Write back reconciled version

Handling Failures

- N selected nodes are the first N healthy nodes
  - Might change from request to request
  - Hence these quorums are <u>“Sloppy” quorums</u>
- Sloppy vs. strict quorums
  - sloppy allow availability under a much wider range of partitions (failures) but sacrifice consistency
- Also, important to handle failures of an entire data center
  - Power outages, cooling failures, network failures, disasters
  - Preference list accounts for this (nodes spread across data centers)
- Hinted Handoff
  - if a node is unreachable, a <u>hinted replica</u> 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

Replica Synchronization

- Using vector clocks, concurrent and out of date updates can be detected.
  - Performing read repair then becomes possible
  - In some cases (concurrent changes) we need to ask the client to pick a value.
- What about node rejoining?
  - Nodes rejoining the cluster after being partitioned
  - When a failed node is replaced or partially recovered
- Replica synchronization is used:
  - to bring nodes up to date after a failure
  - for periodically synchronizing replicas with each other

Replica Reconciliation

- A node holding data received via “hinted handoff” may crash before it can pass data to unavailable node in preference list
- Need another way to ensure each (k, v) pair replicated N times
- Nodes nearby on ring periodically compare the (k, v) pairs they hold, and copy any they are missing that are held by the other

Merkle Trees

- Idea: hierarchically summarize the (k, v) pairs a node holds by ranges of keys
- Leaf node: hash of one (k, v) pair
- Internal node: hash of concatenation of children
- Compare roots; if match, done
- If don’t match, compare children; recur…
- Benefits
  - minimize the amount of data to be transferred for synch
  - reduce the number of required disk reads

Anti Entropy in Dynamo

- Dynamo uses Merkle trees for anti-entropy as follows
  - each node maintains a separate Merkle tree for each key range it hosts
  - two nodes exchange the root of the Merkle tree of the key ranges that they host in common
  - using the synch mechanism described before, the nodes determine if they have any differences and perform the appropriate synch

Techniques used in Dynamo

Problem	Technique	Advantage
Partitioning	Consistent hashing	Incremental scalability
High availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates
Handling temporary failures	Sloppy quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available
Recovering from permanent failures	Antil-entropy using Merkle trees	Synchronizes divergent replicas in the background
Membership and failure detection	Gossip-based membership protocol and failure detecIon	Preserves symmetry and avoids having a centralized registry for storing membership and node liveliness informaIon