Dynamo: Amazon’s Highly Available Key-value Store

Josh Blum | 6.S897 | 09/28/2015

Introduction

- Amazon’s e-commerce platform serves **tens of millions** customers at peak times using **tens of thousands** of servers located in many data centers around the world.
- Need for a scalable and highly available key-value store
- Choose to focus on an eventually consistent store
  - Sacrifices consistency for availability

System Assumptions and Requirements

- Query Model
  - Data is uniquely identified by a key, stored as binary blob
  - No need for relational schema
- Efficiency
  - Runs on commodity heterogenous hardware infrastructure
  - Stringent latency requirements: SLA is 300ms for 99.9th percentile requests
- Other Assumptions
  - Security isn’t an issue

API

- `get(key)`
  - Returns a single object or a list of objects with conflicting versions along with a context
  - Conflicts are handled on reads, never reject a write
- `put(key, context, object)`
  - `context` refers to various kinds of system metadata

Data Partitioning

- Consistent hashing
  - Output range of a hash is treated as a ‘ring’.
  - Assign a key to each object (MD5 of 128-bit client supplied key)
    - MD5(key) -> node (position on the Ring)
  - Incrementally scalable: adding a single node does not affect the system significantly
- “Virtual Nodes”
  - Each node can be responsible for more than one virtual node.
  - Work distribution proportional to the capabilities of the individual node

Data Partitioning

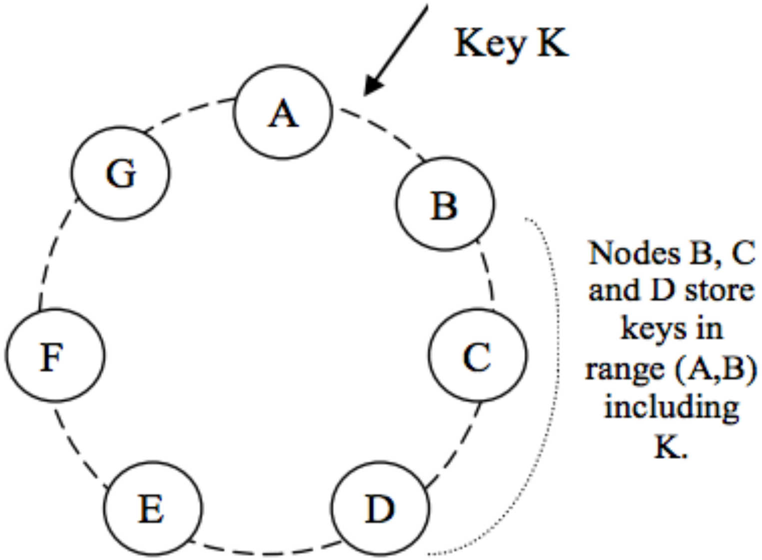

Figure 2: Partitioning and replication of keys in Dynamo ring.

Replication

Example: N=3

- Node B replicates the key k at nodes C and D in addition to storing it locally.
- Node D will store the keys in the ranges (A, B], (B, C], and (C, D].

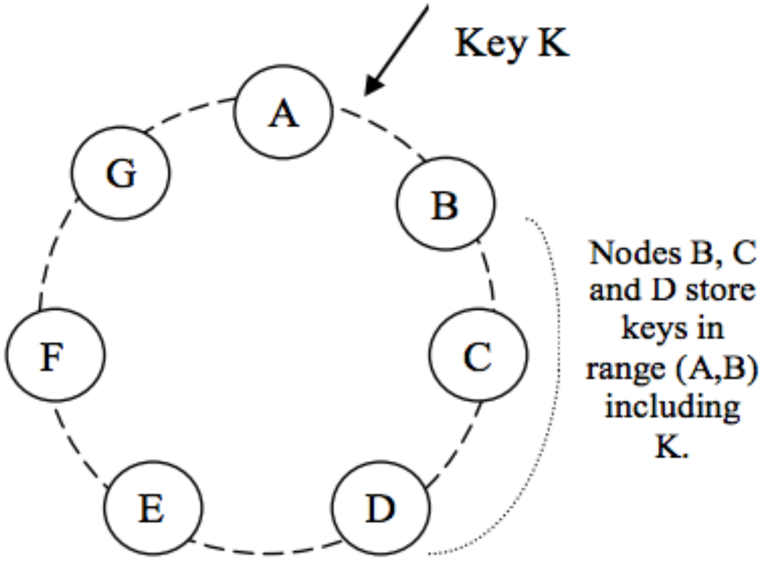

Figure 2: Partitioning and replication of keys in Dynamo ring.

Data Versioning

- System is eventually consistent, thus a `get()` call may return stale data
- An object can have distinct version sub-histories, the system needs reconcile in the future
- Uses vector clocks in order to capture causality between different versions of the same object.

Vector Clocks

- A vector clock is a list of (node, counter) pairs.
- Every version of every object is associated with one vector clock.
- When a client wishes to update an object, it must specify which version it is updating.
- This is done by passing the “context” it obtained from an earlier read operation, which contains the vector clock information.

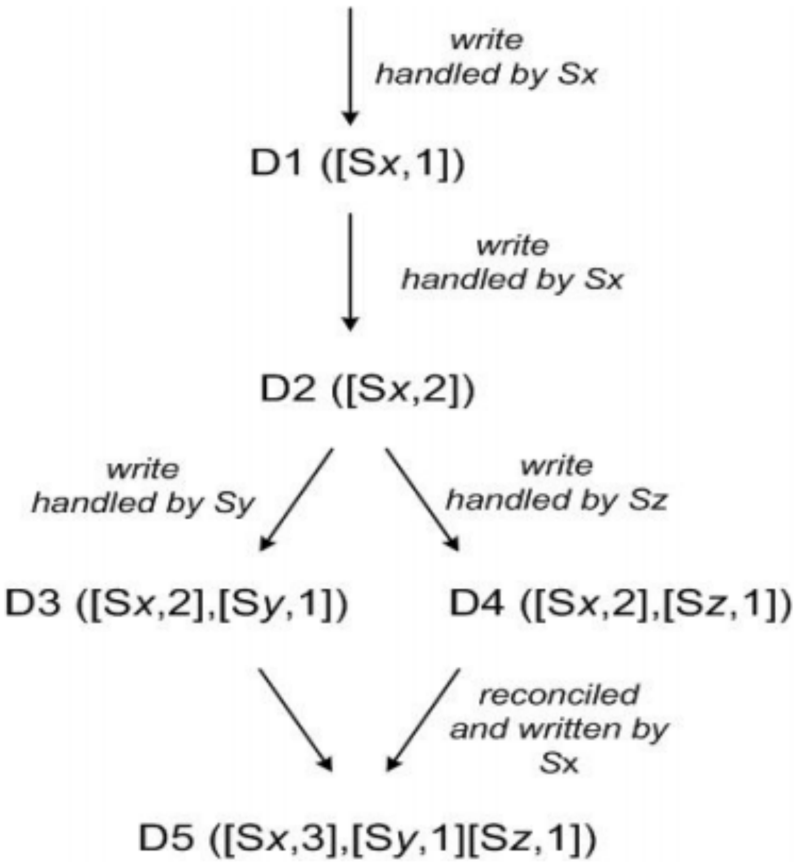

Figure 3: Version evolution of an object over time.

Sloppy Quorum

- `R`: minimum number of nodes that must participate in a successful read operation
- `W`: the minimum number of nodes that must participate in a successful write operation
- Setting `R + W > N` yields a quorum-like system.
- The latency of a `get()` (or `put()`) operation is dictated by the slowest of the `R` (or `W`) replicas
- `R` and `W` are usually configured to be less than `N`, to provide better latency.

Sloppy Quorum: `get()`

- `get()`: coordinator reads from `N` nodes; waits for `R` responses.
  - If they agree, return value.
  - If they disagree, but are causally related, return the most recent value
  - If they are causally unrelated apply reconciliation techniques and write back the corrected version

Sloppy Quorum: `put()`

- `put()`: the coordinator writes to the first `N` healthy nodes on the preference list.
  - Coordinator writes new version vector clock locally and forwards to `N` highest ranked reachable nodes
  - If `W-1` more writes succeed, the write is considered to be successful

`(N, R, W)` Configurations

- Typical: `(3, 2, 2)`
  - Balances performance, durability, and availability
- `W = 1`
  - Never reject a write as long as one node is alive
- Low values of W and R can increase the risk of inconsistency
  - Requests are successful before being processed by a majority of the replicas.
  - Introduces vulnerability window for durability for writes

Failures

- Like Google, Amazon has a number of data centers, each with many commodity machines.
  - Individual machines fail regularly
  - Sometimes entire data centers fail due to power outages, network partitions, tornados, etc.
- To handle failure of entire centers, replicas are spread across multiple data centers.
- Hinted handoff for transient failures
- Merkle trees for replica synchronization

Questions?