Dynamo: Amazon’s Highly Available Key-value Store

Slides by: Parker Misch
Schedule

Introduction

System Architecture

Implementation

Evaluations
Introduction

GOALS

● Create Key-Value store
  ○ <1 MB objects

● Reliability
  ○ Lose $$ when down
  ○ Customers angry

● Scalability

● Acceptable latency at 99.9th percentile (<300ms)
Introduction

SYSTEM REQUIREMENTS

- Query Model
  - Simple read/write interface with key
- ACID Properties (Atomicity, Consistency, Isolation, Durability)
  - Weaken consistency --- increase availability
- Efficiency

ASSUMPTION

- Non-hostile environment
Introduction

DESIGN QUESTIONS

- Strong consistency or high availability?
- Conflict resolution
  - During reads or writes?
  - By data store or application?

DESIGN PRINCIPLES

- Incremental scalability
- Decentralized
- Heterogeneity
System Interface

- **get( key )**
  - Return: { [object], context }

- **put( key, context, object )**
Partitioning

- Consistent hashing
- Key is hashed
- Object stored at next node
  - K0 - Node B
  - K1 - Node G
  - K2 - Node G
- Virtual nodes
  - Scaling
  - Heterogeneity

MD5(K0) = 49
MD5(K1) = 0
MD5(K2) = 127
Replication

- Replication at N hosts
  - Configured per instance
  - Yellow nodes replicate K0
- Each key has coordinator
  - B for K0
- Preference list created
  - B, C, D, E, ...
  - Positions skipped with virtual nodes

MD5(K0) = 49
Data Versioning

- Different versions can exist on each node
- Objects have vector clocks
  - \{node, counter\} pairs
- Reconciliation during reads
  - Keeps data “always writeable”
  - By application
Data Versioning - Reconciliation Example

- **Get(key)**
  - Returned D3 and D4
  - Context object contains version metadata
- **Put(key, context, object)**
  - Context is same as returned by Get
  - New object consolidates VC of D3 and D4
Consistency Protocol

- Configured with R and W
- R - min nodes participating in a read
- W - min nodes participating in a write
- Latency dictated by slowest replica in R or W
  - $R + W \leq N$ to increase latency
- Requests sent to N highest-ranked nodes
  - $R - 1$ responses necessary for successful read
  - $W - 1$ responses necessary for successful write
Hinted Handoff

- Node E stores obj
- obj stored in separate DB
- Will resend to A once back up

OTHER FAILURES

- Whole data center down
  - Object is replicated across data centers
  - Preference list is constructed with nodes in multiple centers
Anti-Entropy for Permanent Failures

- Merkle Tree for each key range
- Nodes exchange roots to compare
  - Consolidate if different
Example Case

- Application1: Get (K0)
- Application2: Get (K0)
- Application1: Put(K0, context, obj_1)
  - B and D store
- Application2: Put(K0, context, obj_2)
  - B and C store
- Version Count B:2, C:1, D:1
- Application3: Get(K0)
  - Return by obj_1 and obj_2

$N = 3$
$R = 2$
$W = 2$
Use Patterns

- **Business logic reconciliation**
  - Decided by application
  - Shopping Cart on Amazon.com

- **Timestamp based reconciliation**
  - Decided by data store
  - User sessions storage

- **High performance read engine**
  - Small number of updates, large number of reads
  - $R = 1, W = N$
  - Product catalog
Load Distribution Strategy

1. $T$ random tokens per node - partition by token
Load Distribution Strategy

2. T random tokens per node equal partitions
Load Distribution Strategy

3. \( \frac{Q}{S} \) random tokens per node - equal sized partitions

\( Q = \text{number of partitions}, \ S = \text{number of nodes in system} \)
Load Distribution Strategy

- Strategy 3 was most efficient (load balancing efficiency)
- Why?
  - Generally - symmetry
  - Faster bootstrapping/recovery
  - Easy of archival
Thanks!

Questions?