DYNAMO: AMAZON’S HIGHLY AVAILABLE KEY VALUE STORE
Giuseppe DeCandia, et al. 2007

PRESENTED BY AFIYA NUSRAT
OUTLINE

• Introduction
• Motivation
• Design Overview
• System Objectives
• System Architecture
• Implementations
• Optimizations
• Conclusions
INTRODUCTION

• Two major challenges faced at Amazon
  – Reliability: slight outage results in huge financial loss
  – Scalability

• Amazon architecture is Highly Decentralized, Loosely Coupled and Service Oriented

• Need for always available storage technologies
  – Shopping Cart example

• Dynamo is highly available and scalable data store which can be configured as per application needs
MOTIVATION

• Traditional systems use relational databases
• Implementing RDBMS functionality requires expensive hardware, skilled personnel.
• Replication technologies limited
• Consistency chosen over availability
• Scaling and load balancing is difficult
• Dynamo addresses all the above problems efficiently!
DESIGN OVERVIEW

• Query Model
  – Simple read/write operations on data item identified by key
  – Targets application that store objects < 1 MB

• ACID
  – Relaxed ACID properties
  – Tradeoff on consistency to achieve availability
  – No isolation guarantees
DESIGN OVERVIEW (cont’d)

• Efficiency
  – System must be able to meet 99.9 percentile of reads and write to complete within a few ms
  – Services must be able to configure Dynamo to achieve desired latency and throughput

• Trusted Environment
  – No security related requirements

• Heterogeneity and Load distribution

• Symmetry

• Decentralization

• Incremental scalability
SYSTEM OBJECTIVES

• Guarantee eventual consistency
• Allow reads and writes even during node failures
• Target applications requiring key/value access with primary focus on high availability
• Target the storage space of an ‘always writeable’ data store by pushing conflict resolution to the reads.
SYSTEM ARCHITECTURE

• Core distributed system techniques in Dynamo
  – Partitioning
  – Replication
  – Versioning
  – Execution
  – Consistency
  – Handling failures
  – Membership and failure detection
  – Scaling
PARTITIONING

• Uses Consistent Hashing
  – MD5 hash on key to generate identifier
PARTITIONING (cont’d)

• Challenges faced:
  – Non uniform data and load distribution
  – Heterogeneity in node performance

• Solution
  – Consistent hashing using VIRTUAL NODES

Consistent Hashing $\rightarrow$ Incremental Scalability
REPLICATION

- $N =$ Number of Replicated machines
  - Coordinator Node: In charge of replication
  - Preference List: Nodes where object is replicated
- Coordinator replicates keys at $N-1$ clockwise successor nodes

Replication $\rightarrow$ availability, durability
VERSIONING

• Eventual Consistency: asynchronous propagation of updates
• Each version is treated as new and immutable
• Allows multiple versions to be present
• Data reconciliation
  – Syntactic : performed by the system
  – Semantic : performed by some business logic at the client side
VECTORING (cont’d)

- Vector clocks: to capture causality between different versions
  - (node, counter) pair
  - One clock per version per object
- Problem: increasing size of vector clocks
  - Solution: vector clock with timestamps
EXECUTION

- Any node can receive get() and put()
- Client selects node routing request through
  - Generic load balancer (max one hop)
  - Partition aware client library (zero hop)
- Coordinator for the request
  - Typically first node in preference list
EXECUTION (cont’d)

• To maintain consistencies among replicas:
  – Quorum like protocol
• Three configurable parameters:
  – N: number of replicas
  – R: min number of nodes that must read a key
  – W: min number of nodes that must write a key
• Setting $R+W>N \rightarrow$ Quorum like system
• R and W usually set less than N for better latency
HANDLING FAILURES

Temporary failures

- “Sloppy Quorum” so that reads/writes requests sent to first N healthy nodes
- Hinted handoff
  - When a intended node is down, hinted object is sent to another node
  - Object returned back to node once node is functionality
- Thus writes rejected only if all nodes are down

Permanent Failures

- Replica synchronization
  - Synchronize each replica periodically using Anti-entropy mechanism
  - Merkle trees used for faster, efficient detection of inconsistencies
  - Each node maintains separate Merkle trees for each key range
MEMBERSHIP AND FAILURE DETECTION

• Membership
  – Explicit addition/removal of nodes
  – Gossip based protocol propagates membership changes
  – Seed nodes: to prevent logical partitioning

• Failure detection
  – Purely local view of failure detection
  – If A cannot contact B, it considers B failed
  – Periodically retries communication
<table>
<thead>
<tr>
<th>PROBLEM</th>
<th>TECHNIQUE</th>
<th>ADVANTAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTITIONING</td>
<td>CONSISTENT HASHING</td>
<td>INCREMENTAL SCALABILITY</td>
</tr>
<tr>
<td>HIGH AVAILABILITY</td>
<td>VECTOR CLOCKS WITH RECONCILIATION DURING READS</td>
<td>VERSION SIZE IS DECOUPLED FROM UPDATE RATES</td>
</tr>
<tr>
<td>HANDLING TEMPORARY FAILURES</td>
<td>SLOPPY QUORAM HINTED HANDOFF</td>
<td>HIGH AVAILABILITY AND DURABILITY GUARANTEE</td>
</tr>
<tr>
<td>RECOVERING FROM PERMANENT FAILURES</td>
<td>ANTI ENTROPY USING MERKLE TREES</td>
<td>SYNCHRONIZES DIVERGENT REPLICAS</td>
</tr>
<tr>
<td>MEMBERSHIP AND FAILURE DETECTION</td>
<td>GOSSIP BASED PROTOCOL</td>
<td>PRESERVES SYMMETRY AVOIDS CENTRALIZED REGISTRY</td>
</tr>
</tbody>
</table>
Implementation

Storage Node

Request Coordination
- Built on top of event-driven messaging substrate
- Uses Java NIO
- Coordinator executes client read & write requests
- State machines created on nodes serving requests

Membership & Failure Detection
- Each state machine instance handles exactly one client request
- State machine contains entire process and failure handling logic

Local Persistence Engine

Pluggable Storage Engines
- Berkeley Database (BDB) Transactional Data Store
- BDB Java Edition
- MySQL
- In-memory buffer with persistent backing store
OPTIMIZATIONS

• Comparison between average and 99.9\textsuperscript{th} latencies exhibit diurnal pattern
• Write latencies higher than read
• 99.9\textsuperscript{th} percentile latencies higher than the average
OPTIMIZATIONS (cont’d)

• Optimization solution: Buffered writes
• Result → Reduction in 99.9th percentile latency by factor of 5 during peak traffic
• Durability tradeoff for performance hence, use Durable Writes
PARTITIONING STRATEGIES

**Strategy 1**
T random tokens per node and partition by token value

**Strategy 2**
T random tokens per node and Q equal-sized partitions (Q is large)

**Strategy 3**
Q/S tokens per node, equal-sized partitions (S machines)

In terms of load distribution efficiency, Strategy 3 > Strategy 1 > Strategy 2
CONCLUSIONS

• Dynamo has provided the desired level of availability
• Successful in handling failures
• Incrementally scalable
• Dynamo allows service owners to customize their storage systems using tunable parameters
• Successfully combines various decentralization technologies to build single system
Dynamo vs. BigTable

• Two different approaches to storage systems
  – BigTable is more tuned for scalability, consistency and Google’s specific needs
  – Dynamo is more tuned for SLA’s.

• Dynamo scales to only a couple hundred machines whereas BigTable is built to store petabytes of data across thousands of servers

• Dynamo is more flexible due to tune-able parameters and correlates to academic research
SIMILAR WORK

• DynamoDB
  – Similar data model but different underlying implementation

• Apache Cassandra
  – Combination of Dynamo and BigTable

• RIAK
  – Open source implementation of Dynamo paper

• Project Voldemort
  – Key value store used at LinkedIn
LET’S DISCUSS!

• Problems with scalability
  – Nodes need to maintain complete routing information for active gossip

• Amazon’s quality metric
  – Interesting that they measure latency at 99.9\textsuperscript{th} percentile!

• Semantic reconciliation increases complexity at client side

• Internet resources