Dynamo: Amazon’s highly available key value store

(By: Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall & Werner Vogels at Amazon.com)

Published in SOSP 2007 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles

Presented by: Prerak Trivedi
Amazon Dynamo is a highly available key-value storage system that uses hash trees and non-relational database. Similar to Google’s BigTable.

Build a highly decentralized distributed system to provide reliability.

Features:

I. Scalable
II. Loosely coupled
III. Highly Available

Performance + Efficiency + Reliability = $$$

Downtime effects:
- Financial Loss
- Impacts Customer Trust
Query Model: Simple Read/Write operations to a data item uniquely identified by a key

ACID Properties:
- Atomicity – Guarantees “All or nothing” principle
- Consistency – Ensures truthfulness of database
- Isolation – All transactions are performed exclusively
- Durability – Recover committed transactions in case of failures

Efficiency: Services must be able to perform under latency to consistently satisfy throughput requirements

Others: Non-hostile environment, No need for security mechanisms like authorization & authentication
Service Level Agreements (SLA)

- SLA Guarantees application delivers its functionality in a bounded time
- Clients and services agree on several system-related characteristics as shown
- Each service initializes a distinct instance of Dynamo
- Example: Guarantee response within 300 ms for 99.9% of its requests for a 500 request per second instance
Design Considerations

- Sacrifice strong consistency for high data availability
- Eventual consistent data using replication algorithms
- “Always writeable” data store
- Other Implementation principles:
  - Incremental Scalability
  - Symmetry
  - Decentralization
  - Heterogeneity
## Techniques in Dynamo and their advantages

<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 for writes</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 Quorum and hinted handoff</td>
<td>Provides high availability and durability guarantee when some of the replicas are not available</td>
</tr>
<tr>
<td>Recovering from permanent failures</td>
<td>Anti-entropy using Merkle trees</td>
<td>Synchronizes divergent replicas in the background</td>
</tr>
<tr>
<td>Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection</td>
<td>Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information</td>
</tr>
</tbody>
</table>
System Interface:

- **Get(Key):**
  - Get(key) operation to locate object replicas
  - Return object list with the conflicting version and context

- **Put(Key, context, object):**
  - Determine where replicas should be placed for associated key
  - Write the replicas to the disk

- **Context:**
  - Encodes system metadata that the caller is not aware of
  - Includes versioning information
Partitioning algorithm:

- Dynamically partition the data over the set of storage nodes
- Relies on consistent hashing to distribute load across multiple nodes and hash function is treated as fixed ring
- Concept of virtual nodes for graceful handling of failures, easy accommodation of a new node
Replication:

- To achieve high availability and durability
- Data replicated among $N$ hosts (where ‘$N$’ is a parameter configured “per-instance”) and stored in preference list.
- Each Node is responsible for region between it and its $N^{th}$ predecessor.
Data Versioning:

- To achieve eventual consistency allowing updates through all replicas asynchronously.
- Get() and Put() suspect if the update is done or not.
- Syntactic or Semantic reconciliation performed by system or client respectively.
- Use of clock truncation scheme to detect difference between versions.
Execution of get() and put() operations:

- Route the requests through generic load balancer that selects a node based on the load information – Need not be code specific
- Use of client library that knows the partitions and routes request directly to coordinator – Achieves lower latency
- The coordinator handles the read/write operations.
- Use of a consistency protocol similar to quorum systems to maintain the consistency of data
Handling failures – Hinted Handoff:

- Use of “sloppy quorums” instead of traditional quorum approach
- All read and write operations are performed on the first N healthy nodes from the preference list, which may NOT always be the first N nodes encountered while walking the consistent hashing ring
- Ensures that read and write operations are not failed due to network failures
- Each object is replicated across multiple data centers, which are connected through high-speed network links
Handling permanent failures – Replica Synchronization:

- Dynamo implements an anti-entropy protocol to keep the replicas synchronized.
- Use of Merkle trees to detect inconsistencies between replicas faster.
- Merkle trees minimize the amount of data that needs to be transferred for synchronization and reduce the number of disk reads performed during the anti-entropy process.
- Disadvantage – When a new node leaves or joins, the trees need to be recalculated.
Merkle Tree:

- A Merkle tree is a hash tree where leaves are hashes of the values of individual keys

- Parent nodes higher in the tree are hashes of their respective children

Advantages:
- Each branch can be checked independently without requiring nodes to download the entire data set
- Help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas
Membership and Failure Detection:

- Ring Membership - explicit mechanism to initiate the addition and removal of nodes from a Dynamo ring.
- Use of Seed nodes to prevent partition
- A gossip-based protocol propagates through the ring and discovers other nodes.
- Failure detection is done in a similar gossip protocol to avoid attempts to communicate with unreachable peers
Implementation

Storage Node

Request Coordination
- Built on top of event-driven messaging substrate
- Implemented using Java
- Coordinator executes client read & write requests by collecting data from nodes
- State machines contain logic for handling the requests, responses, etc.

Membership & Failure Detection
- State machine waits for small period to receive any outstanding responses.
- 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

Chosen based on application’s object size distribution
Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages.
Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.
Thank You