DynamoMINI: A miniature DynamoDB with replication

[*]Distributed System project under Dr. Kishore Kothapalli

Pratham Thakkar
2021101077

Manav Shah
2021101090

***Abstract*—DynamoMINI is a toy version of Amazon's Dynamo, a highly available key-value storage system designed to provide scalability and low latency. The system replicates data across multiple nodes using consistent hashing and employs vector clocks for versioning and reconciliation of data. DynamoMINI ensures fault tolerance through replication and a gossip protocol for decentralised routing table management. The system prioritises high availability for write operations, allowing updates even during network partitions, and pushes the complexity of conflict resolution to the reads. DynamoMINI also utilises a quorum-like technique to maintain consistency among replicas. The project successfully demonstrates the key principles and challenges in designing a distributed storage system like Dynamo, highlighting the trade-offs between availability, consistency, and performance.**

This report will cover the design and implementation of DynamoMINI also discuss challenges encountered during development, solutions implemented, potential applications, and the testing methodology used to validate the system's functionality.

I. Introduction:

Amazon's DynamoDB is a highly available key-value storage system that prioritises scalability, high availability, and low latency. It is designed to handle massive workloads while ensuring consistent single-digit millisecond response times. DynamoDB utilises several key techniques: consistent hashing for data partitioning and replication, vector clocks for versioning and data reconciliation, and a quorum-like approach for maintaining consistency among replicas. The original system also incorporates features like hinted handoff and Merkle trees to further enhance availability and durability.

DynamoDB is crucial for various Amazon services, including their shopping cart service, due to its ability to provide an "always-on" experience. This high availability stems from DynamoDB's acceptance of eventual consistency, allowing updates to propagate asynchronously and pushing the complexity of conflict resolution to the reads. This approach ensures that write operations are never rejected, even during network partitions or node failures.

DynamoMINI is a simplified implementation of DynamoDB, focusing on demonstrating the fundamental principles of consistent hashing, data replication, vector clocks, and the quorum technique. It aims to provide a practical understanding of these techniques and their application in building a distributed storage system.

II. Core Concepts:

1) **Consistent Hashing**: Consistent hashing is a fundamental technique used to distribute data across nodes in a way that minimizes data movement when nodes are added or removed from the system. In DynamoMINI, a hash function maps both data items and nodes onto a virtual ring. When a data item needs to be stored or retrieved, its key is hashed, and the request is routed to the node responsible for the corresponding section of the ring. This approach ensures that only a small portion of data needs to be redistributed when the system topology changes.

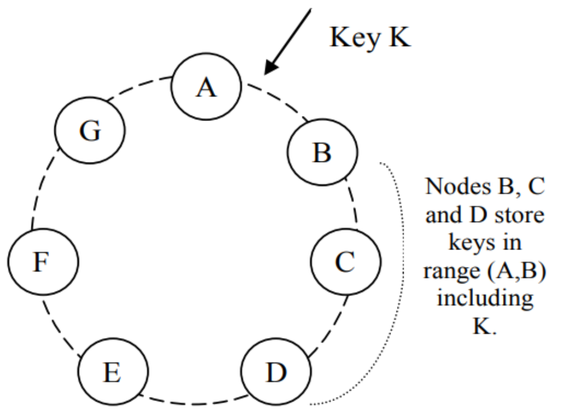

Fig. 1. **Partitioning and replication of keys in Dynamo ring.**

2) **Vector Clocks**: Vector clocks are employed for versioning and maintaining consistency within the decentralized routing table. Each node maintains a vector clock that tracks the latest known version of the routing table across all nodes. This information is exchanged during communication, allowing nodes to detect and resolve inconsistencies. We use version histories to resolve the conflicts.
3) **Quorum Technique**: The quorum technique is used to ensure data consistency in the distributed environment. The system defines a write quorum (W) and a read quorum (R), representing the minimum number of nodes that must participate in a write or read operation respectively. By ensuring that $R + W > N$ (where N is the

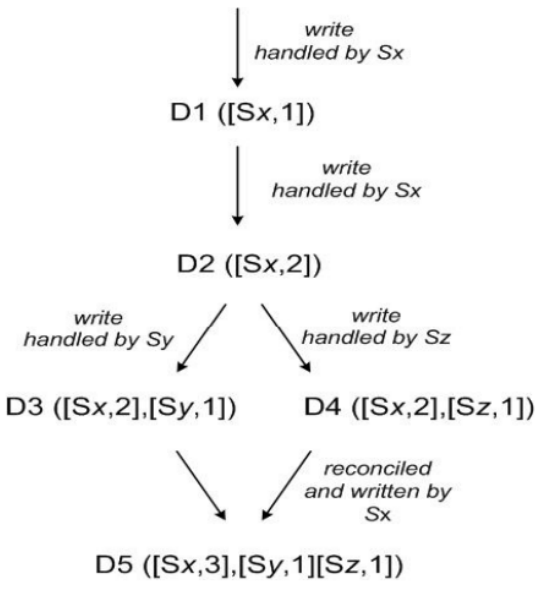

Fig. 2. **Version Evolution of an object over time.**

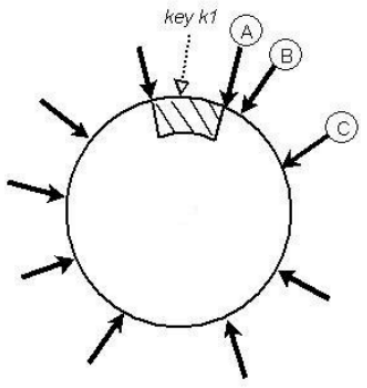

Fig. 3. **Partitioning and placement of keys. A, B, and C depict the three unique nodes that form the preference list for the key k1 on the consistent hashing ring (N=3). The shaded area indicates the key range for which nodes A, B, and C form the preference list. Dark arrows indicate the token locations for various nodes.**

replication factor), the system guarantees that any read operation will see the effects of a previously completed write operation. This helps maintain consistency even in the presence of network partitions or node failures.

III. Basic Assumptions:

DynamoMINI is designed as a distributed key-value store, where data is partitioned and replicated across multiple nodes to ensure fault tolerance and high availability. It operates under the following assumptions:

- **Unique Identification**: Each data item is uniquely identified by a key.
- **No Cross-Item Operations**: Operations are confined to single data items.
- **Trusted Environment**: Security concerns are out of scope; assume a trusted environment.

IV. Design:

1) **Replication and Fault Tolerance**: DynamoMINI leverages asynchronous replication to strengthen fault tolerance. When a write request is received, the system writes the data to W nodes before responding to the client. The remaining (N-W) replicas are updated asynchronously, thereby achieving **eventual consistency** within a finite time. For reads, data is retrieved from R of the N nodes storing the replicas, ensuring that the most recent version is returned. This **quorum-based approach** balances read and write availability while tolerating up to (W-1) node failures. As a result, the system remains **reliable** even in the presence of node or network failures and is **highly available**.
2) **Routing Table Management**: Each node in DynamoMINI maintains a routing table that maps key ranges to the responsible nodes. The routing table is structured as a dictionary, where the keys represent the start of a key range, and the values contain information about the corresponding node. When a client requests a key, the node consults its routing table to identify the coordinator node for that key range.
3) **Reconciliation**: Reconciliation is a crucial process that ensures eventual consistency across all replicas. DynamoMINI employs a periodic reconciliation thread that compares data across different replicas and resolves any inconsistencies. The reconciliation algorithm is customized to handle our scenarios, where we resolve the conflicting writes based on timestamps.
4) **Decentralization with Gossiping Protocols**: DynamoMINI is designed with a decentralized architecture that leverages **peer-to-peer communication**, where each node shares **symmetric responsibilities**, ensuring uniform load distribution. To manage its routing table, DynamoMINI utilizes a gossip protocol, which plays a crucial role in its scalability and robustness. This protocol allows nodes to periodically exchange routing table information with a random subset of other nodes. These interactions enable nodes to reconcile their routing tables, ensuring updates are disseminated throughout the network without reliance on a centralized authority. This decentralized, gossip-driven approach ensures a consistent view of the network topology and data distribution, enhancing both system resilience and scalability.
5) **Scalability:** DynamoMINI is designed to scale efficiently, accommodating increases in both the number of nodes and the volume of data with minimal overhead. By leveraging consistent hashing, the system dynamically adjusts partition assignments as nodes join or leave the network. This allows for seamless horizontal scaling, ensuring that the system maintains its performance and availability without requiring significant reconfiguration or manual intervention.

V. Implementation:

In DynamoMINI, any storage node can receive and process client requests for any key, determined by the hashed value of the key on the ring. Typically, a coordinator node is selected from the key's preference list, which consists of the N nodes responsible for storing replicas of that key, as determined by consistent hashing.

Clients can leverage a partition-aware interface to route requests directly to the appropriate coordinator, optimizing request handling and ensuring efficient data access.

A. Get Operation:

- **Coordinator Selection**: The client's get(key) request is routed to a coordinator node, typically the first in the key's preference list. If the request is handled by a non-preferred node, it will be forwarded to the appropriate coordinator.
- **Data Retrieval**: The coordinator requests all existing versions of the data associated with the key from the N highest-ranked reachable nodes in the preference list. It waits for at least R responses, where R is a configurable parameter representing the minimum number of nodes required for a successful read operation.
- **Version Handling**: If the coordinator receives multiple versions of the data, it returns all causally unrelated versions to the client for semantic reconciliation. If the versions can be reconciled syntactically using vector clocks, the coordinator returns a single, authoritative version.
- **Read Repair**: After responding to the client, the coordinator checks for stale versions in the received responses. If any are found, it updates those nodes with the latest version, ensuring consistency among replicas

B. Put Operation:

- **Coordinator Selection**: The client's put(key, value) request is handled by a coordinator node within the key's preference list. Coordinator is generally the node first to the right of our key in the Hash Ring.
- **Version Generation and Local Write**: The coordinator generates a new vector clock for the updated data and writes the new version locally.
- **Replication**: The coordinator sends the updated data and vector clock to the N highest-ranked reachable nodes in the preference list. The write is considered successful once at least W - 1 nodes respond, where W represents the minimum number of nodes required for a successful write operation.
- **Durable Write**: For improved performance, the W nodes within the N replicas are chosen to perform a "durable write", ensuring data is persisted even if the other nodes experience transient failures.

For the implementation we have used Python, RPyC, Redis as backend key-value storage & We have done a thorough manual testing.

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	Quorum-based algorithm & using replicas	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Using a Down Routing Table to maintain the set of down nodes	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

TABLE I
A SUMMARY OF THE LIST OF TECHNIQUES OUR DYNAMOMINI USES AND THEIR RESPECTIVE ADVANTAGES.

VI. Key Challenges:

Building a distributed system like DynamoMINI presents various challenges related to consistency, fault tolerance, and dynamic membership. This section addresses some of the key challenges encountered during development and the solutions implemented to overcome them.

1) Routing Table Updates:
   Problem: Handling failed nodes without permanently removing them.
   Solution: Two separate routing tables are maintained, one for active nodes and another for inactive nodes. This approach allows the system to track potentially recoverable nodes and periodically check their status without completely removing them from the system.
2) Achieving eventual consistency across all replicas, especially in the presence of network partitions:
   Problem: Data divergence during network partitions.
   Solution: A dedicated reconciliation thread periodically compares data across different replicas and applies reconciliation algorithms to resolve any inconsistencies. This ensures that data converges to a consistent state eventually, even if temporary inconsistencies arise due to network issues or node failures.
3) Ensuring that the routing table remains consistent across all nodes in a decentralized manner:
   Solution: Each node actively participates in maintaining routing table consistency by periodically exchanging information with other nodes. This decentralized approach, often referred to as a gossip protocol, helps to propagate routing table updates quickly and efficiently throughout the system, converging in approximately log(n) steps.
4) Node Addition & Deletion:
   Problem: Fetching key ranges for a new node.
   Solution: The new node fetches data from its successors

based on the updated routing table.

VII. Use Cases:

- **Shopping Cart**: Key-value pairs represent items and their quantities. Supports fault-tolerant, write-available services.
- **Instagram Post Likes**: Store like counts efficiently with real-time updates across replicas.
- **General Applicability**: DynamoMINI is a natural choice to all those systems requiring high availability, fault tolerance, low latency for reads and writes, and which allow for eventual consistency.

-

VIII. Conclusion and Future Scope:

DynamoMINI successfully implements a simplified version of Amazon's Dynamo, showcasing the power of consistent hashing for data partitioning and replication across multiple nodes. By employing vector clocks for versioning and the quorum technique for consistency, DynamoMINI ensures high availability and fault tolerance. The system's ability to handle dynamic load balancing, failure detection and recovery, and eventual consistency through reconciliation algorithms makes it a robust and reliable storage solution. The gossip protocol employed for decentralised routing table management further enhances its scalability and robustness by allowing nodes to efficiently share and reconcile routing information.

The project, however, identifies areas for future development, drawing inspiration from the full-fledged Dynamo system described in the sources. These include:

- **Merkle Trees for Efficient Reconciliation**: Incorporating Merkle trees, as suggested in the original paper, would enhance the efficiency of data reconciliation. Merkle trees allow for the verification of data integrity and the identification of inconsistencies between replicas using a hierarchical hashing structure. This approach minimises the amount of data that needs to be transferred during reconciliation, reducing network overhead and improving performance.
- **Hinted Handoff for Enhanced Availability**: Implementing a hinted handoff mechanism would further bolster DynamoMINI's availability during temporary node failures. In a hinted handoff, if a node is unavailable during a write operation, the data is temporarily written to an alternative node. Once the original node recovers, the alternative node transfers the data to ensure all replicas are consistent. This mechanism ensures that write operations are not rejected due to transient failures, improving the system's overall resilience.

By incorporating these future enhancements, DynamoMINI can evolve into a more comprehensive and feature-rich distributed storage system.

IX. Acknowledgement:

We are grateful for the opportunity to this project as part of a Distributed Systems course taken by Dr. Kishore Kothapalli.

References

[1] Dynamo: Amazon's highly available key-value store: Giuseppe DeCandia and Deniz Hastorun and Madan Jampani and Gunavardhan Kakulapati and Avinash Lakshman and Alex Pilchin and Swaminathan Sivasubramanian and Peter Vosshall and Werner Vogels.

[2] Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service.