A Distributed Key-Value Store with Consistent Hashing and Quorum-Based Replication

Ojas[1], R Hitesh[2], R Shreevatsan S S[3], Aniket Kumar[4], Meena Belwal[5]
Department of Computer Science and Engineering, Amrita School of Computing, Bengaluru,
Amrita Vishwa Vidyapeetham, India
bl.en.u4cse22281@bl.students.amrita.edu[1]
bl.en.u4cse22248@bl.students.amrita.edu[2]
bl.en.u4cse22255@bl.students.amrita.edu[3]
bl.en.u4cse22206@bl.students.amrita.edu[4]
b_meena@blr.amrita.edu[5]

***Abstract*—This paper introduces a lean, fault-tolerant peer-to-peer distributed storage system based on Amazon's Dynamo, with a Distributed Hash Table (DHT) design. The system effectively shuffles data into multiple nodes through consistent hashing such that every data item is tagged with a different key in a circular keyspace. For added reliability, data is replicated on several nodes such that redundancy is maintained and no data is lost when nodes fail. The system is also intended to provide dynamic join and leave operations of nodes with little disruption to in-progress processes, enabling scalability and flexibility continuously with a growing or shrinking number of nodes. The system also provides high availability and consistency even in the case of node failures, making use of vector clocks as a technique to handle conflicts among concurrent writes. This deployment seeks to offer a strong and effective solution for dealing with large-scale data in distributed systems, overcoming major challenges such as fault tolerance, load balancing, and data consistency, hence being ideal for applications that need high availability and resilience.**

***Index Terms*—Distributed Systems, Key-Value Store, Consistent Hashing, Replication, Quorum**

I. Introduction

Managing large volumes of data poses unique challenges, particularly when it must be distributed across multiple systems. Ensuring data safety combined with easy accessibility while maintaining proper distribution is the main objective. DHT stands for Distributed Hash Table, which serves as a common solution to handle this challenge. The system establishes an effective and reliable approach to data storage and handling. Data stored as key-value pairs on multiple machines enables Amazon's Dynamo system to achieve better performance and fault tolerance standards.

We designed a basic peer-to-peer storage framework that implements the Distributed Hash Table approach. Each data item in this system needs a distinct key that is organized through consistent hashing into a continuous loop. Each data item is replicated across multiple machines to ensure redundancy and fault tolerance. The setup allows data preservation through multiple machines so that data remains accessible even when a machine fails or departs from the network.

The system design, built like Dynamo, enables users to connect to any node to read or write data. According to Dasgupta and Misra [1], DHT systems achieve high availability and distribute data to achieve load balancing through data replication. The system automatically handles node join and leave operations, ensuring minimal disruption and consistent performance.

Numerous technical challenges may arise from node departures without warning or delay interruptions within communication lines. The system implements write acknowledgment functionality and feasibility checking occasions before starting any read/write operation. Such measures maintain the reliability and consistency of the system regardless of the conditions under which it operates.

Our system maintains three primary objectives by protecting data safety and enhancing fault tolerance features while establishing distributed data sharing access. To maintain proper functioning our system needed to enable smooth operations during machine network entry or departure.

The key contributions of the proposed system are as follows:

- High fault tolerance through data replication and quorum-based operations
- Efficient and balanced data distribution using consistent hashing
- Dynamic scalability with seamless node addition and removal

The literature review is provided in Section II, which studies existing systems alongside their difficulties. Section III gives an overview of the tools and components used to build our system. We explain how our system works step by step. Finally, Section IV shows the results of our method and discusses how well it performs.

II. Literature Survey

In this section, we examine related works that address distributed storage systems, key-value databases, and consistent hashing mechanisms. These studies highlight various approaches to data distribution, indexing, performance optimization, and fault tolerance. Each contribution offers insights into ongoing research challenges and possible solutions within distributed storage systems.

The paper by Aslantaş [1] discusses a distributed key-value store designed for IoT devices, aiming to improve the way data is managed by making it faster, more scalable, and more reliable. It uses techniques like dividing data and copying data to make access better. Still faces challenges in keeping data consistent and synchronized across different devices.

In [2], Dawei et al. introduce a hybrid database system that combines HBase and Hive to make big data processing faster. This system benefits from distributed storage, which makes data processing more efficient. The system requires an extra layer to connect different parts, which adds complexity to the system.

The paper by Niu et al. [3] presents kv2vec, a method that uses neural networks to represent key-value pairs as dense vectors, improving metadata searches in scientific datasets. This method works better than older models in finding relevant key-value pairs. Existing methods don't take the order of words in key-value pairs into account, which can lead to incorrect results.

Cai et al. [4] introduce BonsaiKV+, a key-value store that uses both fast DRAM and large non-volatile memory (NVM) to improve data indexing performance. They used a three-layer system to scale better and solve issues like memory congestion.

The paper by Wang et al. [5] presents NStore, a key-value store designed for persistent memory. It improves performance by reducing some memory accesses and making read/write operations more efficient. System still faces problems with remote and local memory writes, under the case of uneven workloads, which affects overall performance.

Wang and Sun [6] compare several algorithms on measures like lookup time, memory space, initialization time, resize time, balance in key distribution, and monotonicity. By implementation of the algorithms in Java and the benchmarking thereof on typical hardware, the research offers empirical evidence beyond theoretical complexity. It recognizes Jump, Anchor, and Dx as high performers, with the understanding that actual-world performance may vary from asymptotic expectations. Release of the code as open-source improves reproducibility and further research.

Zaouia et al. [7] consider data management in distributed photovoltaic (PV) stations—comparing centralized versus geographically distributed configurations—and propose a multi-replication consistent-hashing scheme that enhances fault tolerance and outperforms standard hashing in their experiments.

Yi et al. [8] survey cross-modal hashing, observing that batch methods don't generalize to streaming data and supervised methods may be computationally costly or sacrifice quantization accuracy. They introduce Online Label Consistent Hashing (OLCH), which online learns and updates hash codes to ensure semantic consistency and efficiency in an online learning regime.

Wang and Luo [9] discuss hash functions, mentioning weaknesses of MD5 and SHA-1 and of SHA-2/3 in embedded platforms. It summarizes previous efforts using chaotic maps for hashing, with some drawbacks like the restricted key space and inefficient processing.

Coluzzi et al. [10] consider issues related to managing big data and the scalability of distributed storage systems. They indicate that consistent hashing is limited by data skew with small nodes and suggest an enhanced algorithm applying virtual nodes to offer improved load balance and flexibility. Experimental results demonstrate its practical efficiency.

Massimo Coluzzi et al. [11] present MementoHash, a new constant-hashing algorithm with perfect balance under arbitrary node insertion/removals, with a very low memory footprint and constant time updates. MementoHash improves over AnchorHash and other contemporary schemes in lookup rate and scalability

John Chen el at. [12] Extends the traditional consistent-hashing model by imposing explicit load limits per node, removing hotspots under skewed workloads. Their scheme provides provable balance guarantees and enables efficient reconfiguration of the hash ring

Masoomeh Javidi Kishi el at. [13]Introduces SSS, a distributed KV store that provides external consistency for all transactions and never aborts read-only transactions. It leverages vector clocks and a new "snapshot-queuing" mechanism to order-serialize transactions client-observed, delivering up to 7× throughput over 2PC baselines

Yang Shi el at. [14] Introduces KVSwitch, a programmable-switch-based load balancer for distributed KV stores that dynamically directs requests to underloaded servers. Exhibits up to a 2× decrease in tail latency under highly skewed query streams

Chao Dong el at. [15] Introduces DxHash, which saves up to 98.4% metadata overhead over AnchorHash yet maintains quick lookups and constant-time reconfiguration. Empirical analysis demonstrates better performance in terms of both memory consumption and update latency

M. Singh el at. [16] Suggests a low-overhead end-to-end security architecture for resource-limited IoT edge devices, such as secure boot, FOTA, and secure key storage means, resulting in an efficiency gain of 20% compared to the prior art

B. Pusti and el at. [17] Introduces an adaptive VM-allocation and secure-link MEC-server model with 6.43% decreased energy consumption when it imposes data-at-rest encryption and access control between IoT devices and gateways H. Manohar Reddy el at. [18] Creates an energy-profiling framework for homomorphic encryption (HE) on Raspberry Pi 4 IoT nodes, quantifying CPU, memory, execution time, and showing up to 70% energy cost—best practices for HE optimization on edge devices A. Aswathy el at. [19] Reports a distributed data-ingestion and analytics pipeline for real-time Twitter streams that utilizes edge nodes for preprocessing and minimizing end-to-end latency in mission-critical applications that enhance the user experience.

III. System Design

The core architecture of our system draws inspiration from Amazon's Dynamo and follows a peer-to-peer topology with decentralized control.

A. System Architecture

Our distributed key-value store consists of multiple independent nodes organized in a logical ring. Each node in the system runs an Akka-based actor system and maintains local storage for key-value pairs. The nodes communicate with each other using asynchronous message passing over Akka's actor model.

Every node is identified by a 32-bit signed ID obtained by hashing its listen address (`IP: port`). These IDs are placed on a circular hash ring using our Java `HashUtil.hash(...)` function. Likewise, each client key is hashed to the same space, and the key coordinator is the first node whose ID is greater than or equal to the key's hash (wrapping around at the end of the ring).

Figure 1 illustrates the logical ring and how nodes manage ranges of keys in a distributed fashion.

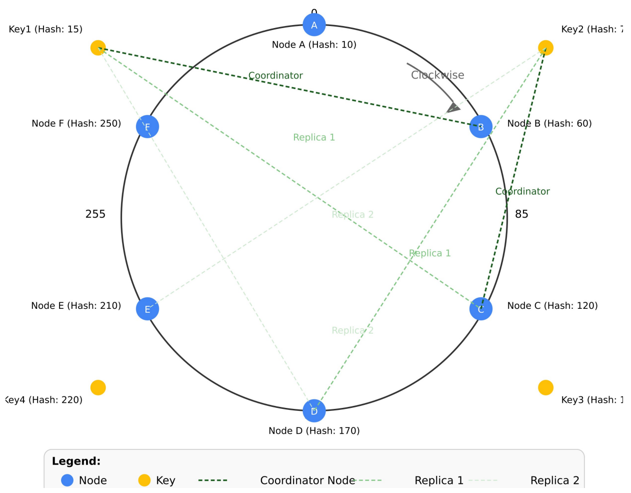


Fig. 1. Consistent Hash Ring with Node Responsibility

B. Components and Communication

The system is composed of the following components:

- **Node Actor:** The primary actor in each node that receives and processes client requests and coordinates replication with neighboring nodes.
- **Coordinator Node:** The node responsible for a particular key, determined by consistent hashing. It initiates read and write operations.
- **Storage Manager:** A file-based engine responsible for storing key-value pairs locally on each node using a CSV-backed format.
- **Client:** An independent process that sends read/write requests to any node in the system.

Nodes communicate via Akka messages, including:

- `ClientReadRequest`, `ClientReadResponse`, `ClientUpdateRequest`, `ClientUpdateResponse`
- `ReadRequest`, `ReadResponse`, `WriteRequest`, `WriteResponse`
- `JoinRequestMessage`, `NodesListMessage`, `JoinDataMessage`, `JoinMessage`
- `LeaveDataMessage`, `LeaveMessage`, `TimeoutMessage`

The ring's current view is maintained by each node, allowing decentralized coordination.

C. Consistent Hashing

The consistent hashing mechanism makes sure that there is minimal data movement when nodes join or leave the ring. The ring is divided based on the hash space. Each key is hashed and assigned to the first node whose ID is greater than or equal to the key's hash.

If a node leaves, only a small portion of the keys need to be reassigned. Similarly, when a node joins, it takes responsibility for part of its successor's keyspace. **Deterministic Replication:** We replicate each key on the next $N-1$ nodes clockwise in the ring (where N is the replication factor)

D. Request Routing

When a client sends a request to any node (consider Node A for this example), the following steps occur:

1) Node A hashes the key and determines the coordinator node using the current ring view.
2) It forwards the request to the coordinator.
3) The coordinator performs quorum-based replication (more details in Section III) and responds once quorum is satisfied.
4) If the coordinator fails to respond within the client timeout (10 s), the client receives an error. In practice, clients may retry against another node or back off and retry the same node.

This approach ensures fault tolerance and low latency without needing a centralized load balancer.

E. Failure Detection and Timeout

To handle partial failures, nodes use a 3s quorum timeout (‘QUORUM_TIMEOUT_SECONDS‘) and clients use a 10s request timeout (‘CLIENT_TIMEOUT_SECONDS‘). If a quorum is not reached in time, an error is returned.

IV. Replication and Consistency

To ensure high availability and fault tolerance, the system employs a replication mechanism that distributes each key-value pair to multiple nodes in the network. Inspired by Amazon's Dynamo, this approach uses a combination of consistent hashing, replica placement, and quorum-based coordination to manage consistency and availability in the presence of node failures.

A. Replication Strategy

Each key-value pair is stored on a total of N nodes, where N is the replication factor. The node to which a key is mapped via consistent hashing is termed the **coordinator node**. The next $(N-1)$ successor nodes in the clockwise direction on the hash ring serve as replica holders.

For instance, if $N = 3$, the system ensures that a key is stored on:

- The **coordinator** node (responsible for the key)
- The **first successor** (Replica 1)
- The **second successor** (Replica 2)

This placement guarantees data redundancy and improves fault tolerance by ensuring that at least two replicas remain available even if a single node fails.

B. Quorum-Based Consistency Model

To maintain consistency across replicas while ensuring high availability, the system uses a quorum-based approach for both read and write operations. This model is defined by three parameters:

- N: Total number of replicas (replication factor)
- R: Minimum number of nodes that must respond to a read operation
- W: Minimum number of nodes that must acknowledge a write operation

Consistency is guaranteed under the condition:

$$R + W > N$$

This ensures that the read and write quorums intersect, allowing at least one replica to have the most recent value during concurrent or conflicting operations.

A typical configuration in our system uses $N = 3$, $R = 2$, and $W = 2$. This allows the system to tolerate node failures while maintaining eventual consistency.

C. Version Control Using Vector Clocks

Due to the decentralized nature of the system, it is possible that concurrent writes to the same key may occur across different replicas. To detect and resolve such conflicts, we employ **vector clocks** for versioning.

Each version of a key is associated with a vector clock that tracks the causal history of write operations. When a node receives multiple versions of the same key, it uses vector clocks to:

- Identify and suppress older versions
- Detect concurrent versions that must be resolved

If conflicting versions are detected, the system can either:

1) Return all conflicting versions to the client for manual resolution
2) Employ application-specific logic for automatic conflict resolution

This mechanism helps maintain eventual consistency in the presence of concurrent updates and network partitions.

V. Data Management

Our system uses a modular, file-based storage engine combined with consistent hashing to manage data placement, retrieval, and fault tolerance.

A. Key Placement & Local Store

Each key is hashed into a 32-bit space and assigned to its *coordinator* node on the consistent-hash ring. Each node persists its share of the keyspace in a simple CSV file (space-delimited) with entries:

```
<key> <value> <version>
```

A write-through in-memory cache accelerates reads and writes. On node joins/leaves, each node calls `dropOldKeys()` to remove or accept only the keys it remains responsible for.

B. PUT and GET Operations

- **PUT(key,value):**
  1) Client sends `ClientUpdateRequest` to any node.
  2) Coordinator hashes the key, reads current versions from R replicas, picks a new version, then issues `WriteRequest` to all N replicas.
  3) Waits for W `WriteResponse` acknowledgements before replying.
- **GET(key):**
  1) Client sends `ClientReadRequest`.
  2) Coordinator hashes the key and sends `ReadRequest` to N replicas in parallel.
  3) Gathers R `ReadResponse` messages, picks the highest-version value, and returns it.

Both operations proceed asynchronously and use a 3s quorum timeout by default.

C. Failure Handling

We handle failures via quorum timeouts:

- If fewer than R or W replicas reply in time, the coordinator aborts and returns an error to the client.
- On graceful node leave, a node redistributes its data to its successors before shutdown.

This design achieves a clean separation between placement, storage, and replication, with all key I/O confined to the CSV engine and quorum logic in the NodeActor.

VI. Node Lifecycle

Dynamic membership is handled entirely in-band, via Akka messages and on-disk handoff, without external services. As showed in the figure 2

A. Joining

A new node N_{new} executes:

1) *Bootstrap contact:* send `JoinRequestMessage` to any existing node.
2) *Ring view update:* receive `NodesListMessage` and update local ring.
3) *Data handoff:* request `DataRequestMessage` from its predecessor; receive a `JoinDataMessage` containing exactly those records it now owns.
4) *Announce presence:* multicast a `JoinMessage` so everyone adds N_{new} to their ring view.

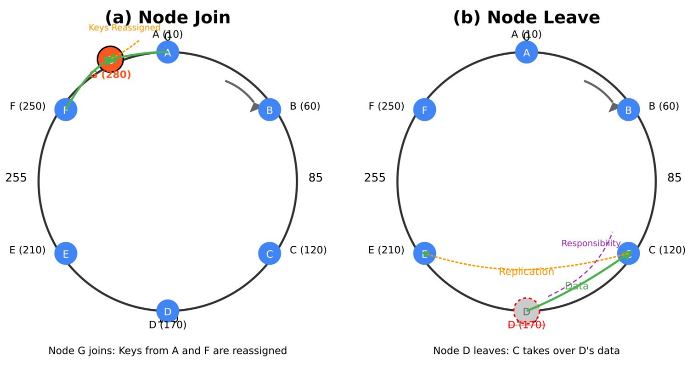


Fig. 2. Sequence when a node joins or leaves: ring update and data handoff.

B. Leaving

When a node N_{leave} gracefully departs:

- It computes, for each key it holds, which $N - 1$ new successors now own that key.
- It sends a `LeaveDataMessage` to each of those successors with the exact subset they must import.
- It multicasts a `LeaveMessage` so everyone removes N_{leave} from their ring view.
- Finally it deletes its local storage and stops.

C. Failure & Recovery

On an unexpected crash, there is no explicit handoff or anti-entropy yet:

- Peers detect missing quorum replies (via a 3s timeout) and continue serving other requests.
- A recovering node uses `JoinRequestMessage` + `JoinDataMessage` exactly as in a fresh join, then multicasts `ReJoinMessage` so everyone updates its ActorRef.

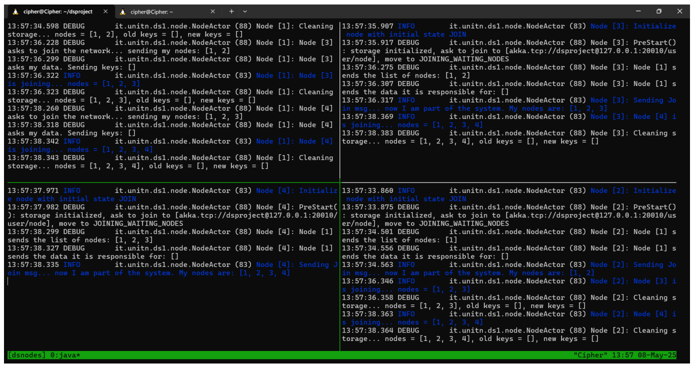
D. Membership View

Each node keeps a full but decentralized view of the ring (its map of IDs→ActorRefs). All updates—joins, leaves, re-joins—are propagated by simple multicast of the three message types above.

VII. Simulated Evaluation

To assess the correctness of our system, we conducted a series of controlled experiments on a local testbed. Four nodes were deployed on separate Java Virtual Machines (JVMs), each maintaining independent storage, routing metadata, and participating in replication and quorum-based operations. Figure 3 illustrates the logical ring and how nodes manage ranges of keys in a distributed fashion.

- **Nodes:** 4 JVMs on localhost (ports 20010, 20020, 20030, 20040)
- **Replication factor** $N = 3$, **Read quorum** $R = 2$, **Write quorum** $W = 2$
- **Timeouts:** quorum timeout = 3 s, client timeout = 10 s
- **Failure injection:** manual `tmux send-keys C-c` on the third node after the first, and termination as shown in Figure 4



```
13:57:34.598 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [1]: Cleaning
storage... nodes = [1, 2], old keys = [], new keys = []
13:57:36.228 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [1]: Node [3]
asks to join the network... sending my nodes: [1, 2]
13:57:36.299 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [1]: Node [3]
asks my data. Sending keys: []
13:57:36.322 INFO        it.unitn.ds1.node.NodeActor (83) Node [1]: Node [3]
is joining... nodes = [1, 2, 3]
13:57:36.323 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [1]: Cleaning
storage... nodes = [1, 2, 3], old keys = [], new keys = []
13:57:38.260 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [1]: Node [4]
asks to join the network... sending my nodes: [1, 2, 3]
13:57:38.318 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [1]: Node [4]
asks my data. Sending keys: []
13:57:38.342 INFO        it.unitn.ds1.node.NodeActor (83) Node [1]: Node [4]
is joining... nodes = [1, 2, 3, 4]
13:57:38.343 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [1]: Cleaning
storage... nodes = [1, 2, 3, 4], old keys = [], new keys = []
```

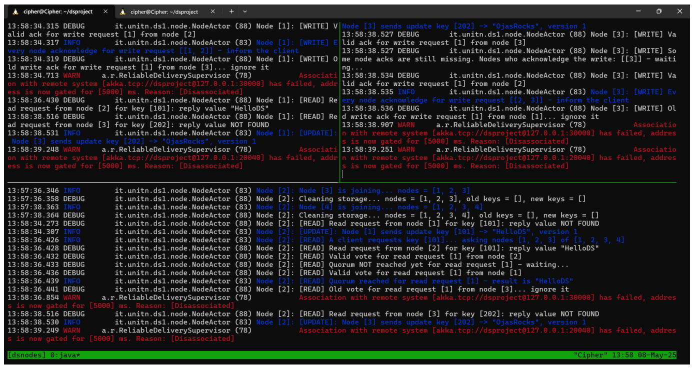
```
13:57:35.907 INFO        it.unitn.ds1.node.NodeActor (83) Node [3]: Initialize
 node with initial state JOIN
13:57:35.917 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [3]: PreStart()
: storage initialized, ask to join to [akka.tcp://dsproject@127.0.0.1:20010/us
er/node], move to JOINING_WAITING_NODES
13:57:36.275 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [3]: Node [1] s
ends the list of nodes: [1, 2]
13:57:36.307 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [3]: Node [1] s
ends the data it is responsible for: []
13:57:36.317 INFO        it.unitn.ds1.node.NodeActor (83) Node [3]: Sending Jo
in msg... now I am part of the system. My nodes are: [1, 2, 3]
13:57:38.369 INFO        it.unitn.ds1.node.NodeActor (83) Node [3]: Node [4] i
s joining... nodes = [1, 2, 3, 4]
13:57:38.383 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [3]: Cleaning s
torage... nodes = [1, 2, 3, 4], old keys = [], new keys = []
```

```
13:57:37.971 INFO        it.unitn.ds1.node.NodeActor (83) Node [4]: Initializ
e node with initial state JOIN
13:57:37.982 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [4]: PreStart(
): storage initialized, ask to join to [akka.tcp://dsproject@127.0.0.1:20010/
user/node], move to JOINING_WAITING_NODES
13:57:38.299 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [4]: Node [1]
sends the list of nodes: [1, 2, 3]
13:57:38.327 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [4]: Node [1]
sends the data it is responsible for: []
13:57:38.335 INFO        it.unitn.ds1.node.NodeActor (83) Node [4]: Sending J
oin msg... now I am part of the system. My nodes are: [1, 2, 3, 4]
```

```
13:57:33.860 INFO        it.unitn.ds1.node.NodeActor (83) Node [2]: Initialize
 node with initial state JOIN
13:57:33.875 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: PreStart()
: storage initialized, ask to join to [akka.tcp://dsproject@127.0.0.1:20010/us
er/node], move to JOINING_WAITING_NODES
13:57:34.501 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: Node [1] s
ends the list of nodes: [1]
13:57:34.556 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: Node [1] s
ends the data it is responsible for: []
13:57:34.563 INFO        it.unitn.ds1.node.NodeActor (83) Node [2]: Sending Jo
in msg... now I am part of the system. My nodes are: [1, 2]
13:57:36.346 INFO        it.unitn.ds1.node.NodeActor (83) Node [2]: Node [3] i
s joining... nodes = [1, 2, 3]
13:57:36.358 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: Cleaning s
torage... nodes = [1, 2, 3], old keys = [], new keys = []
13:57:38.363 INFO        it.unitn.ds1.node.NodeActor (83) Node [2]: Node [4] i
s joining... nodes = [1, 2, 3, 4]
13:57:38.364 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: Cleaning s
torage... nodes = [1, 2, 3, 4], old keys = [], new keys = []
```

```
[dsnodes] 0:java*                                        "Cipher" 13:57 08-May-25
```

Fig. 3. Node Joining in the Consistent Hash Ring.



```
13:58:34.315 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [1]: [WRITE] V
alid ack for write request [1] from node [2]
13:58:34.317 INFO        it.unitn.ds1.node.NodeActor (83) Node [1]: [WRITE] E
very node acknowledge for write request [[1, 2]] - inform the client
13:58:34.319 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [1]: [WRITE] O
ld write ack for write request [1] from node [3]... ignore it
13:58:34.713 WARN     a.r.ReliableDeliverySupervisor (78)             Associati
on with remote system [akka.tcp://dsproject@127.0.0.1:30000] has failed, addr
ess is now gated for [5000] ms. Reason: [Disassociated]
13:58:36.430 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [1]: [READ] Re
ad request from node [2] for key [101]: reply value "HelloDS"
13:58:38.516 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [1]: [READ] Re
ad request from node [3] for key [202]: reply value NOT FOUND
13:58:38.531 INFO        it.unitn.ds1.node.NodeActor (83) Node [1]: [UPDATE]:
 Node [3] sends update key [202] -> "OjasRocks", version 1
13:58:39.248 WARN     a.r.ReliableDeliverySupervisor (78)             Associati
on with remote system [akka.tcp://dsproject@127.0.0.1:20040] has failed, addr
ess is now gated for [5000] ms. Reason: [Disassociated]
```

```
Node [3] sends update key [202] -> "OjasRocks", version 1
13:58:38.527 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [3]: [WRITE] Va
lid ack for write request [1] from node [3]
13:58:38.527 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [3]: [WRITE] So
me node acks are still missing. Nodes who acknowledge the write: [[3]] - waiti
ng...
13:58:38.534 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [3]: [WRITE] Va
lid ack for write request [1] from node [2]
13:58:38.535 INFO        it.unitn.ds1.node.NodeActor (83) Node [3]: [WRITE] Ev
ery node acknowledge for write request [[2, 3]] - inform the client
13:58:38.536 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [3]: [WRITE] Ol
d write ack for write request [1] from node [1]... ignore it
13:58:38.907 WARN     a.r.ReliableDeliverySupervisor (78)             Associatio
n with remote system [akka.tcp://dsproject@127.0.0.1:30000] has failed, addres
s is now gated for [5000] ms. Reason: [Disassociated]
13:58:39.251 WARN     a.r.ReliableDeliverySupervisor (78)             Associatio
n with remote system [akka.tcp://dsproject@127.0.0.1:20040] has failed, addres
s is now gated for [5000] ms. Reason: [Disassociated]
```

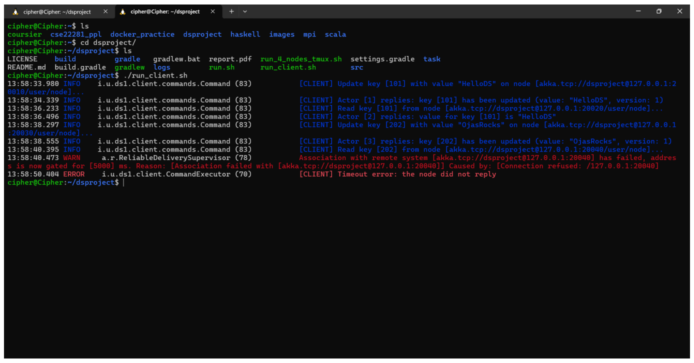
```
13:57:36.346 INFO        it.unitn.ds1.node.NodeActor (83) Node [2]: Node [3] is joining... nodes = [1, 2, 3]
13:57:36.358 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: Cleaning storage... nodes = [1, 2, 3], old keys = [], new keys = []
13:57:38.363 INFO        it.unitn.ds1.node.NodeActor (83) Node [2]: Node [4] is joining... nodes = [1, 2, 3, 4]
13:57:38.364 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: Cleaning storage... nodes = [1, 2, 3, 4], old keys = [], new keys = []
13:58:34.273 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: [READ] Read request from node [1] for key [101]: reply value NOT FOUND
13:58:34.307 INFO        it.unitn.ds1.node.NodeActor (83) Node [2]: [UPDATE]: Node [1] sends update key [101] -> "HelloDS", version 1
13:58:36.426 INFO        it.unitn.ds1.node.NodeActor (83) Node [2]: [READ] A client requests key [101]... asking nodes [1, 2, 3] of [1, 2, 3, 4]
13:58:36.428 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: [READ] Read request from node [2] for key [101]: reply value "HelloDS"
13:58:36.432 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: [READ] Valid vote for read request [1] from node [2]
13:58:36.433 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: [READ] Quorum NOT reached yet for read request [1] - waiting...
13:58:36.436 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: [READ] Valid vote for read request [1] from node [1]
13:58:36.439 INFO        it.unitn.ds1.node.NodeActor (83) Node [2]: [READ] Quorum reached for read request [1] - result is "HelloDS"
13:58:36.441 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: [READ] Old vote for read request [1] from node [3]... ignore it
13:58:36.854 WARN     a.r.ReliableDeliverySupervisor (78)             Association with remote system [akka.tcp://dsproject@127.0.0.1:30000] has failed, addres
s is now gated for [5000] ms. Reason: [Disassociated]
13:58:38.516 DEBUG       it.unitn.ds1.node.NodeActor (88) Node [2]: [READ] Read request from node [3] for key [202]: reply value NOT FOUND
13:58:38.530 INFO        it.unitn.ds1.node.NodeActor (83) Node [2]: [UPDATE]: Node [3] sends update key [202] -> "OjasRocks", version 1
13:58:39.249 WARN     a.r.ReliableDeliverySupervisor (78)             Association with remote system [akka.tcp://dsproject@127.0.0.1:20040] has failed, addres
s is now gated for [5000] ms. Reason: [Disassociated]
```

```
[dsnodes] 0:java*                                        "Cipher" 13:58 08-May-25
```

Fig. 4. Node failure simulation by manual termination.

We performed these from the client side as an independent node as shown in Figure 5.

1) Write key 101 → “HelloDS”
2) Read key 101
3) Kill the node that held one of the replicas
4) Read key 101 again
5) Write key 202 → “OjasRocks”
6) Read key 202



```
cipher@Cipher:~$ ls
coursier  cse22281_ppl  docker_practice  dsproject  haskell  images  mpi  scala
cipher@Cipher:~$ cd dsproject/
cipher@Cipher:~/dsproject$ ls
LICENSE    build         gradle   gradlew.bat  report.pdf  run_4_nodes_tmux.sh  settings.gradle  task
README.md  build.gradle  gradlew  logs         run.sh      run_client.sh        src
cipher@Cipher:~/dsproject$ ./run_client.sh
13:58:33.980 INFO    i.u.ds1.client.commands.Command (83)           [CLIENT] Update key [101] with value "HelloDS" on node [akka.tcp://dsproject@127.0.0.1:2
0010/user/node]...
13:58:34.339 INFO    i.u.ds1.client.commands.Command (83)           [CLIENT] Actor [1] replies: key [101] has been updated (value: "HelloDS", version: 1)
13:58:36.233 INFO    i.u.ds1.client.commands.Command (83)           [CLIENT] Read key [101] from node [akka.tcp://dsproject@127.0.0.1:20020/user/node]...
13:58:36.496 INFO    i.u.ds1.client.commands.Command (83)           [CLIENT] Actor [2] replies: value for key [101] is "HelloDS"
13:58:38.297 INFO    i.u.ds1.client.commands.Command (83)           [CLIENT] Update key [202] with value "OjasRocks" on node [akka.tcp://dsproject@127.0.0.1
:20030/user/node]...
13:58:38.555 INFO    i.u.ds1.client.commands.Command (83)           [CLIENT] Actor [3] replies: key [202] has been updated (value: "OjasRocks", version: 1)
13:58:40.395 INFO    i.u.ds1.client.commands.Command (83)           [CLIENT] Read key [202] from node [akka.tcp://dsproject@127.0.0.1:20040/user/node]...
13:58:40.473 WARN     a.r.ReliableDeliverySupervisor (78)           Association with remote system [akka.tcp://dsproject@127.0.0.1:20040] has failed, addres
s is now gated for [5000] ms. Reason: [Association failed with [akka.tcp://dsproject@127.0.0.1:20040]] Caused by: [Connection refused: /127.0.0.1:20040]
13:58:50.404 ERROR    i.u.ds1.client.CommandExecutor (70)           [CLIENT] Timeout error: the node did not reply
cipher@Cipher:~/dsproject$ |
```

Fig. 5. Client operations.

A. Metrics Observed

1) **Write Availability:** Both writes (keys 101, 202) were acknowledged by at least two replicas before and after failure—100% success under simulated node crash.
2) **Read Consistency:** Reads before and after node failure always returned the most recent value (no stale reads), thanks to $R = 2$ quorum satisfying intersection with the write quorum.
3) **Replication Correctness:** On each write, the key was stored on the coordinator and its two clockwise successors. After killing one replica, the remaining two still satisfied the quorum for subsequent operations.

B. Failure Recovery Simulation

When we terminated one replica immediately after writing key 101, subsequent reads for 101 still succeeded by contacting the two surviving replicas. This demonstrates that our simple quorum mechanism provides uninterrupted availability despite node crashes. (Automatic rejoin and hinted-handoff were not exercised in this run.)

C. Discussion

Our small-scale experiment confirms:

- *Availability:* Writes and reads continue through single-node failures.
- *Consistency:* Quorum intersection guarantees the latest version is always returned.
- *Durability:* Data remains accessible on surviving replicas until full ring membership is restored.

While limited in scale, the behavior aligns with expectations from Dynamo-style systems and indicates that our implementation is functionally correct under node churn.

VIII. Conclusion and Future Work

We presented a peer-to-peer, Akka-based key-value store that uses consistent hashing, $N = 3$ replication, and $R = W = 2$ quorums to balance availability and consistency. Through local experiments with failure injection, we showed that the system continues to serve both reads and writes without data loss under single-node crashes.

Future Work

Planned enhancements include:

- **Gossip-based Membership:** Replace static bootstrap with a decentralized failure detector and ring dissemination.
- **Hinted Handoff & Anti-Entropy:** Implement hinted handoff queues and background Merkle-tree reconciliation to fully repair lost replicas.
- **RESTful API Layer:** Expose operations over HTTP for easier integration.
- **Adaptive Quorum Tuning:** Dynamically adjust R and W based on node health and workload.
- **Security:** Add TLS and authentication between nodes and clients.

Our prototype confirms that quorum-based replication on a consistent-hash ring can achieve high availability and durability in the face of node failures, laying the groundwork for production-grade extensions.

References

[1] B. Aslantaş, E. N. Pektaş and Ş. Baydere, "Distributed Key Value Store for IoT Edge Devices," 2024 9th International Conference on Computer Science and Engineering (UBMK), Antalya, Turkiye, 2024.

[2] Y. Dawei, Y. Hengxiang, H. Meihui and M. Jun, "Research on the Application of Distributed Key-Value Storage Technology in Computer Database Platform," 2022 IEEE 2nd International Conference on Power, Electronics and Computer Applications (ICPECA), Shenyang, China, 2022.

[3] C. Niu, W. Zhang, S. Byna and Y. Chen, "Kv2vec: A Distributed Representation Method for Key-value Pairs from Metadata Attributes," 2022 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2022.

[4] M. Cai, J. Shen, Y. Yuan, Z. Qu and B. Ye, "Scaling Persistent In-Memory Key-Value Stores Over Modern Tiered, Heterogeneous Memory Hierarchies," in IEEE Transactions on Computers, vol. 74, no. 2, pp. 495-509, Feb. 2025.

[5] Z. Wang et al., "NStore: A High-Performance NUMA-Aware Key-Value Store for Hybrid Memory," in IEEE Transactions on Computers, vol. 74, no. 3, pp. 929-943, March 2025.

[6] M. Wang and Q. Sun, "Application of Consistent Hash Based on Virtual Node in Traffic Big Data," 2022 7th International Conference on Intelligent Computing and Signal Processing (ICSP), Xi'an, China, 2022.

[7] N. Zaouia, K. Djouzi and M. Daoui, "Collisions-Resistant Hash Function Based on a Logistics Map," 2023 IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Paris, France, 2023.

[8] J. Yi, X. Liu, Y. -m. Cheung, X. Xu, W. Fan and Y. He, "Efficient Online Label Consistent Hashing for Large-Scale Cross-Modal Retrieval," 2021 IEEE International Conference on Multimedia and Expo (ICME), Shenzhen, China.

[9] Z. Wang and T. Luo, "An Improved Consistent Hashing-Based Data Indexing Method for Distributed Photovoltaic Stations on Highways," 2024 IEEE 22nd International Conference on Industrial Informatics (INDIN), Beijing, China, 2024.

[10] M. Coluzzi, A. Brocco, P. Contu and T. Leidi, "A survey and comparison of consistent hashing algorithms," 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Raleigh, NC, USA, 2023.

[11] M. Araujo, D. Oliveira and F. Travassos, "MementoHash: A Stateful, Minimal-Memory, Best-Performing Consistent Hashing," arXiv preprint arXiv:2306.09783, 2023.

[12] J. Chen, B. Coleman and A. Shrivastava, "Revisiting Consistent Hashing with Bounded Loads," in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, no. 15, pp. 13 711–13 719, 2021.

[13] M. Javidi Kishi, S. Peluso, H. Korth and R. Palmieri, "SSS: Scalable Key-Value Store with External Consistent and Abort-free Read-only Transactions," arXiv preprint arXiv:1901.03772, 2019.

[14] X. Liu, Y. Shi and K. Li, "KVSwitch: Balancing Distributed Key-Value Stores with Efficient In-Network Load Management," Electronics, vol. 8, no. 9, 1008, 2019.

[15] Chao Dong, Fang Wang and Dan Feng, "DxHash: A Memory-Saving Consistent Hashing Algorithm," in Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23), 2023.

[16] M. Singh and S. Sankaran, "Lightweight Security Architecture for IoT Edge Devices," in Proc. 2022 IEEE International Symposium on Smart Electronic Systems (ISES), pp. 455–458, 2022.

[17] B. Pusti and S. Sankaran, "Security and Energy-Aware Resource Allocation in Mobile Edge Computing (MEC)," in Proc. 2022 IEEE International Symposium on Smart Electronic Systems (iSES), 2022.

[18] H. Manohar Reddy, S. P. C. Sajimon and S. Sankaran, "On the Feasibility of Homomorphic Encryption for Internet of Things," in 2022 IEEE 8th World Forum on Internet of Things (WF-IoT), pp. 1–8, 2022.

[19] A. Aswathy, R. Prabha, L. S. Gopal, D. Pullarkatt and M. V. Ramesh, "An Efficient Twitter Data Collection and Analytics Framework for Effective Disaster Management," in 2022 IEEE Delhi Section Conference (DELCON), pp. 1–6, 2022.