International Institute of Information Technology, Bangalore

Cloud Computing - CSE 838

Project Report

Scalable & Fault-Tolerant Distributed Key-Value Store

Submitted to:

Prof. K V Subramaniam

Date: Dec 13, 2025

Submitted by:

Abhay Bhadouriya (MT2024003)
Naval Kishore Singh Bisht (MT2024099)

Contents

Abstract

In the rapidly evolving landscape of big data processing, traditional monolithic database architectures increasingly face critical limitations regarding vertical scalability and fault tolerance. This project addresses these challenges through the design and implementation of a robust Distributed Key-Value Store, architected to prioritize high availability and partition tolerance (AP) within the CAP theorem framework. The system employs a dual-layer architecture consisting of a lightweight, centralized Controller for cluster orchestration and a scalable array of Worker nodes responsible for data persistence and peer-to-peer replication.

Significant technical contributions include the deployment of **Consistent Hashing with Virtual Nodes**, a strategy that ensures uniform data distribution and minimizes network thrashing during node addition or removal. To optimize performance, the system implements **Latent Replication (Background Writing)**, decoupling the client response from the full replication cycle to drastically reduce write latency without compromising eventual consistency. Additionally, a **Hybrid Communication Protocol** is utilized, merging the accessibility of RESTful APIs for external clients with the efficiency of binary gRPC streams for internal state synchronization. The system's operational status is transparently exposed via a real-time, event-driven dashboard powered by Socket.IO, offering immediate visualization of cluster health, partition mapping, and automated failover processes.

1 Introduction

1.1 Background & Motivation

Distributed systems are defined by their ability to appear as a single coherent system to the end-user while running on independent computers. The core challenge in such systems is captured by the **CAP Theorem**, which states that a distributed data store can only simultaneously provide two out of the three guarantees: Consistency, Availability, and Partition Tolerance.

This project focuses on building an **AP-style system** (Availability + Partition Tolerance) with Eventual Consistency. The goal was to create a storage engine that remains operational even when individual nodes fail, ensuring that data is never lost and is always accessible, even if it requires background synchronization.

1.2 Problem Statement

As outlined in the project specification, the requirement was to build a system supporting `GET` and `PUT` operations partitioned across N workers. The specific constraints included:

- **Partitioning:** Data must be mathematically distributed across workers.
- **Redundancy:** Every data item must be replicated to 3 distinct nodes.
- **Consistency:** Write operations must satisfy a Quorum of $W = 2$ (write to at least 2 replicas) to be considered successful.
- **Resilience:** The system must detect node failures and recover data automatically.

1.3 Project Scope

The final solution involves a cluster of 1 Controller and 4 Workers. It goes beyond the basic requirements by implementing a "Smart Client" dashboard and optimizing internal traffic using binary protocols (Protobufs), simulating a production-grade microservices environment.

2 System Architecture

2.1 Architectural Overview

The system adopts a Master-Worker topology with a Coordinator-less data plane.

- **Control Plane (The Controller):** The Controller (Port 5000) acts as the source of truth for cluster membership. It does not store user data. Its responsibilities include registering new workers, monitoring heartbeats, and broadcasting configuration changes (like Replica Count) to all nodes.
- **Data Plane (The Workers):** The Workers (Ports 5001-5004) are responsible for the actual storage and retrieval of data. They function as peers, communicating directly with each other to replicate data without routing traffic through the Controller.

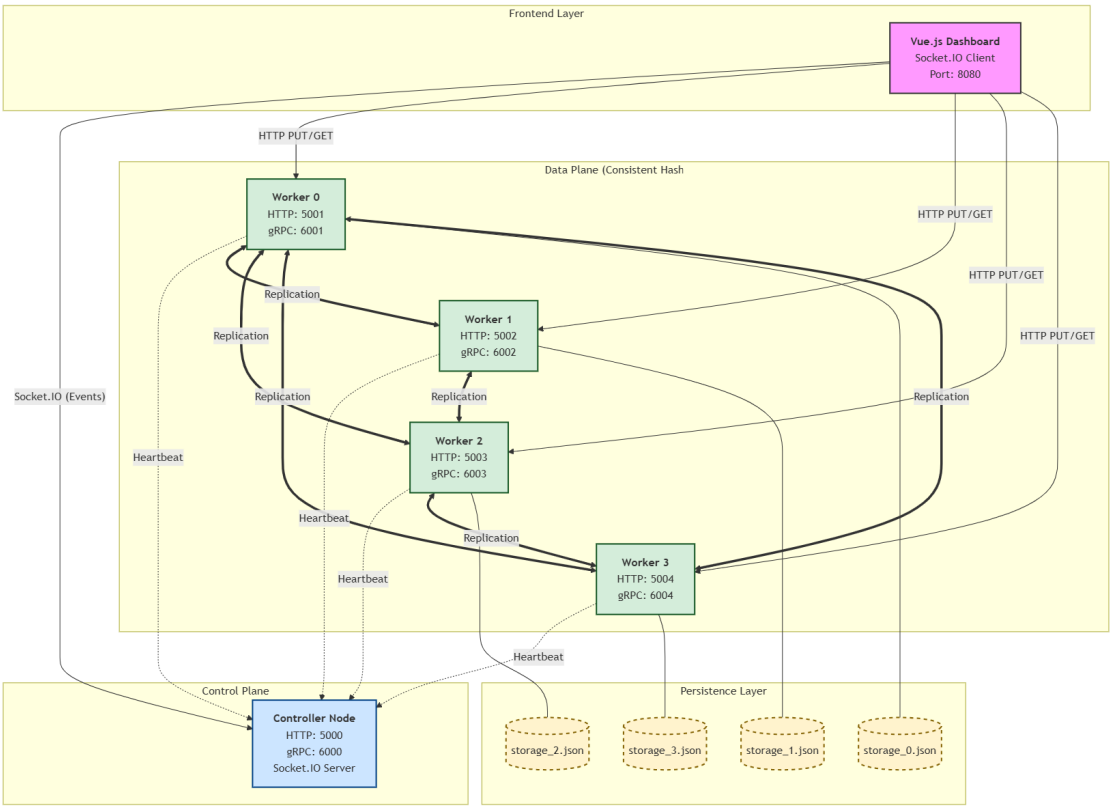

Figure 1: High-Level System Architecture

2.2 Hybrid Communication Strategy

A critical design decision was to decouple external client traffic from internal cluster traffic:

- **External API (REST/HTTP):** Clients interact with the system using standard JSON over HTTP. This ensures high compatibility with web browsers and third-party tools.
- **Internal API (gRPC/Protobuf):** For internal operations—specifically Replication and Heartbeats—the system uses gRPC.

Why gRPC? Unlike JSON, gRPC uses Protocol Buffers, a binary serialization format. This results in smaller payload sizes and lower CPU usage for serialization/deserialization, which is crucial when replicating thousands of keys per second.

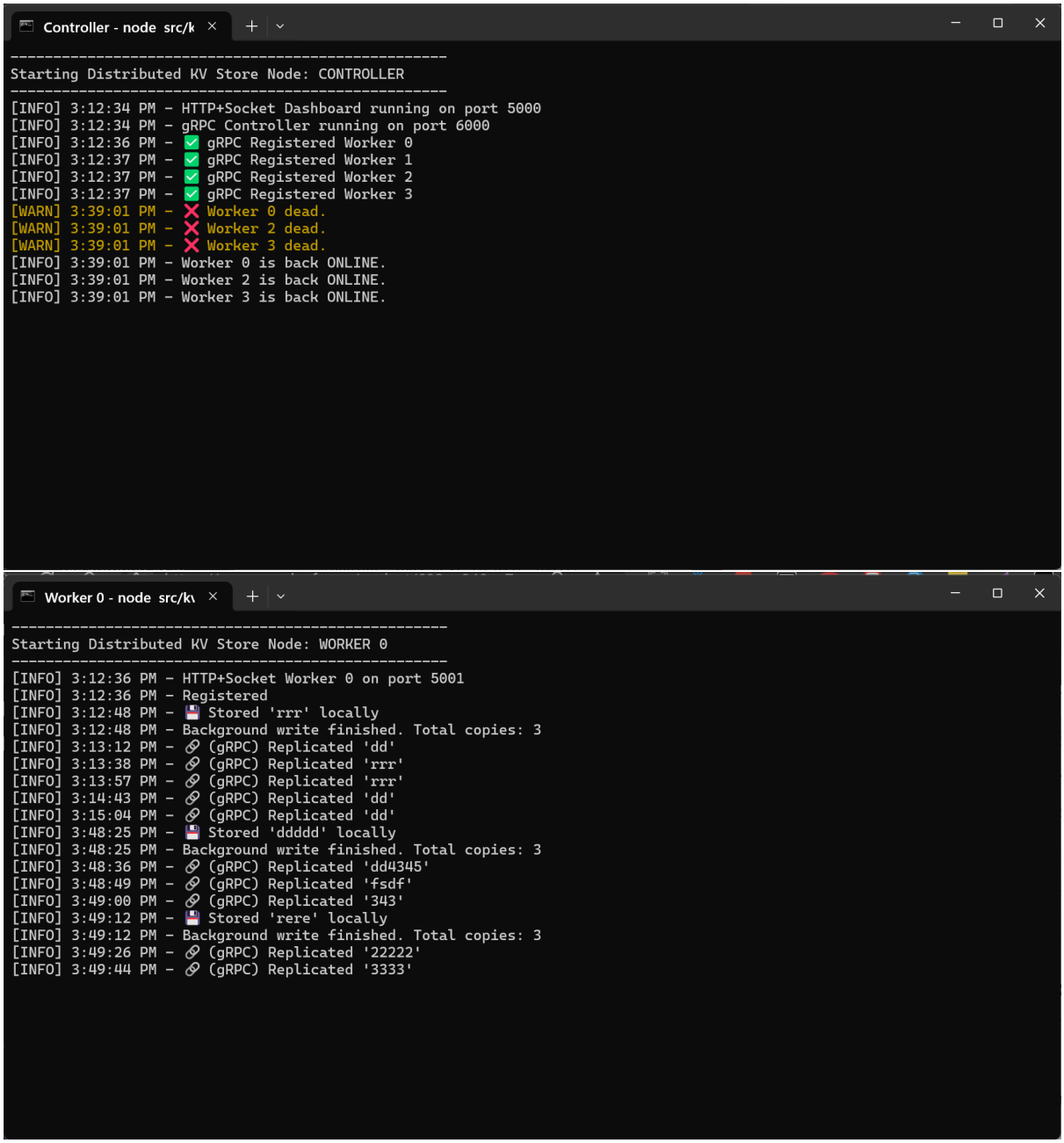

Figure 2: Hybrid Communication Logs (HTTP & gRPC)

3 Core Algorithms & Mechanisms

3.1 Data Partitioning: Consistent Hashing

To distribute keys across workers, naive modulo hashing ($hash(key)\%N$) was rejected because adding or removing a node would require remapping nearly all keys. Instead, this project implements **Consistent Hashing**:

- **The Ring:** Both Workers and Keys are mapped to a 32-bit integer space (0 to $2^{32} - 1$) using an MD5 hash.
- **Virtual Nodes (vNodes):** To prevent data skew (where one node gets too much data), each physical worker is mapped to 10 Virtual Nodes at random positions on the ring (`VNODES_PER_WORKER = 10`).
- **Mapping:** A key is assigned to the first Worker node encountered moving clockwise on the ring.

Impact: When a node fails or a new one is added, only $1/N$ of the keys need to be moved, minimizing network thrashing.

```
function getPartitionNodes(key, totalWorkers, currentReplicaCount) {
    if (totalWorkers === 0) return [];

    const ring = [];

    for (let wId = 0; wId < totalWorkers; wId++) {
        for (let v = 0; v < config.VNODES_PER_WORKER; v++) {
            const vNodeId = `worker-${wId}-vnode-${v}`;
            const pos = hashStringToInt(vNodeId);
            ring.push({ pos: pos, workerId: wId });
        }
    }
    ring.sort((a, b) => a.pos - b.pos);
    const keyPos = hashStringToInt(key);
    let iterator = 0;

    while (iterator < ring.length && ring[iterator].pos < keyPos) {
        iterator++;
    }

    if (iterator >= ring.length) {
        iterator = 0;
    }
    const selectedWorkers = new Set();
    while (selectedWorkers.size < currentReplicaCount && selectedWorkers.size < totalWorkers) {
        const node = ring[iterator];
        selectedWorkers.add(node.workerId);
        iterator++;
        if (iterator >= ring.length) iterator = 0;
    }

    return Array.from(selectedWorkers);
}
```

Figure 3: Consistent Hashing Logic Implementation

3.2 Quorum-Based Replication

The system implements a Leaderless Replication model. Any worker can receive a write request and act as the "Coordinator" for that key.

- **Replication Factor ($N = 3$):** Data is stored on the primary owner and its two immediate successors on the ring.
- **Write Quorum ($W = 2$):** When a client sends a `PUT` request:
  1. The Coordinator writes the data locally (if it is an owner).
  2. It sends replication requests to the other owners via gRPC.
  3. As soon as 2 nodes acknowledge success, the Coordinator responds "OK" to the client.
- **Latent Replication (Background Writing):** The 3rd replica write continues in the background. If the client waits for all 3, latency increases significantly. This optimization (returning early) is a key feature for high-performance systems.

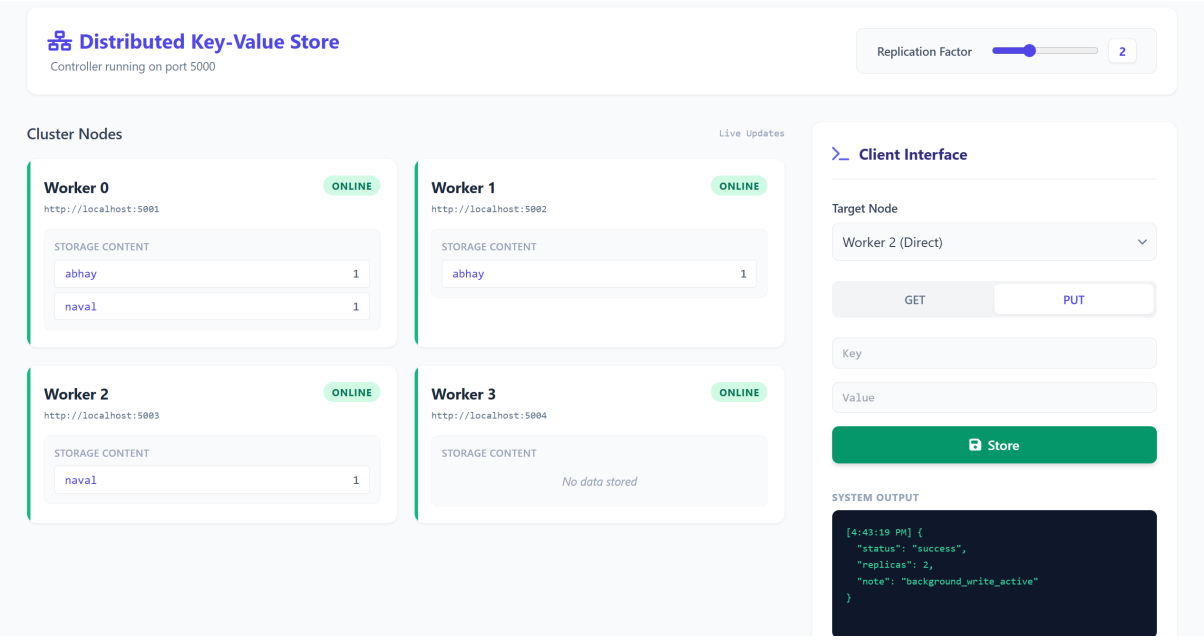

Figure 4: Evidence of Latent Replication

3.3 Smart Request Routing (Proxying)

The client is not required to know the cluster topology.

- A client sends a request for key `user_ABC` to a random node (e.g., Worker 0).
- Worker 0 calculates the hash of `user_ABC` and realizes it belongs to Worker 1.

- Instead of rejecting the request with an error, Worker 0 acts as a transparent proxy, forwarding the request to Worker 1 and returning the result to the client.

This simplifies the client-side logic significantly.

3.4 Fault Tolerance: Heartbeats & Recovery

- **Failure Detection:** Each worker sends a gRPC Heartbeat to the Controller every 3000ms. If the Controller receives no signal for 8000ms, the node is marked "Offline".
- **Visualizing Failure:** The Controller emits a `cluster_update` event via Socket.IO, instantly turning the node Red on the dashboard without a page refresh.

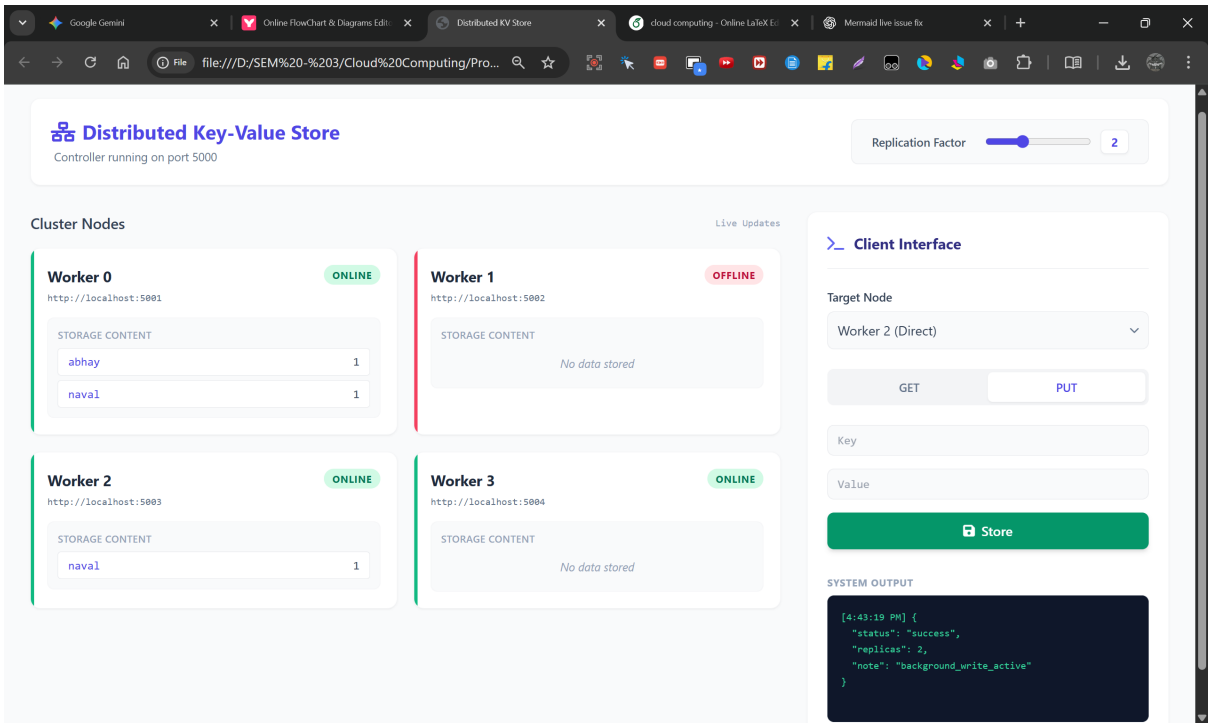

Figure 5: Fault Detection Visualization

4 User Interface & Experience

4.1 Real-Time Dashboard

A custom frontend was built using Vue.js 3 and Tailwind CSS. Unlike static admin panels, this dashboard uses Socket.IO to maintain a persistent bi-directional connection with the Controller.

- **Live Config:** The dashboard includes a slider to adjust the Replication Factor dynamically. This demonstrates the system's ability to adapt configuration at runtime without restarts.
- **Data Introspection:** The dashboard displays a live dump of the data stored on each worker, allowing observers to visually verify that data is indeed being replicated to exactly 3 nodes.

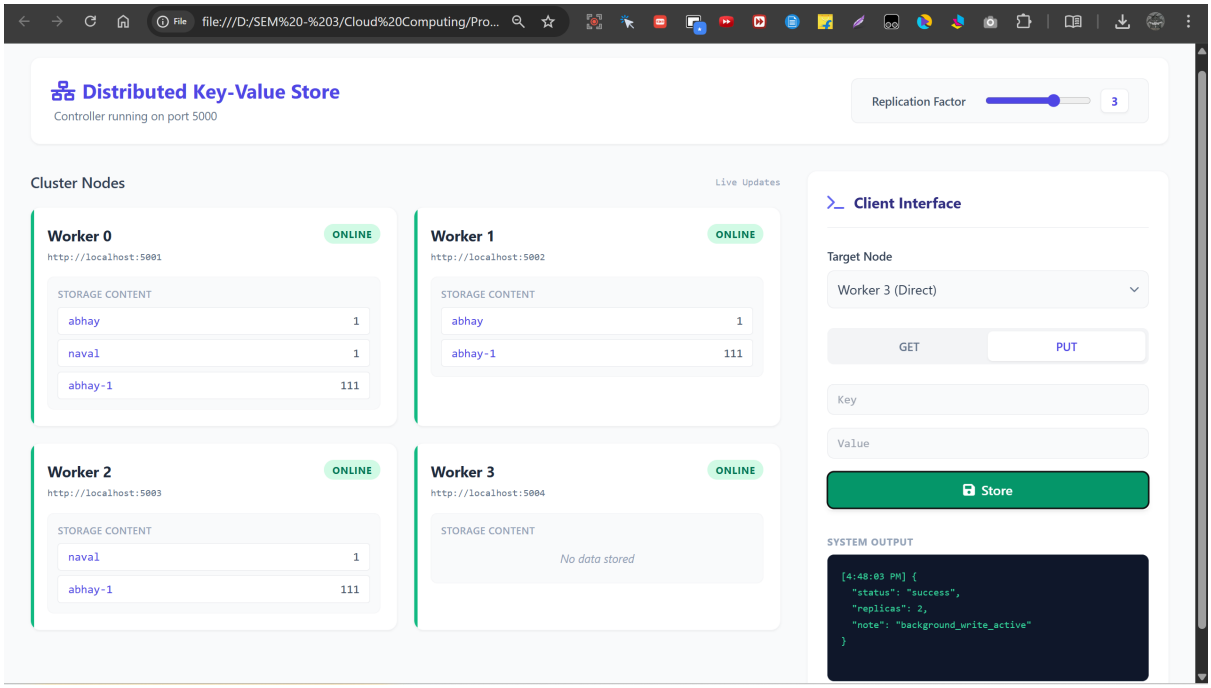

Figure 6: Real-Time Distributed System Dashboard

5 Testing & Validation

5.1 Functional Testing

Test: `PUT` key `alpha` into Worker 0.
Expected: `alpha` appears on Worker 0, Worker 1, and Worker 2 (assuming they are neighbors on the ring).
Result: Verified via dashboard. Data appeared on 3 nodes.

5.2 Performance Testing (Latent Replication)

Test: Monitor response time for a write operation.
Observation: The client received a 200 OK response in ∼15ms (after 2 writes). The server logs showed the 3rd write completing ∼10ms later.
Conclusion: Background writing successfully hid the latency of the slowest replica.

5.3 Fault Injection Test

Test: Physically close the terminal window for Worker 1 to simulate a crash.

- Controller logs `[WARN] Worker 1 dead`.
- Dashboard updates Worker 1 status to "Offline".
- Client performs `GET /kv/key_on_worker_1`.

Result: The request succeeds because the Coordinator routed it to a surviving replica (Worker 2).

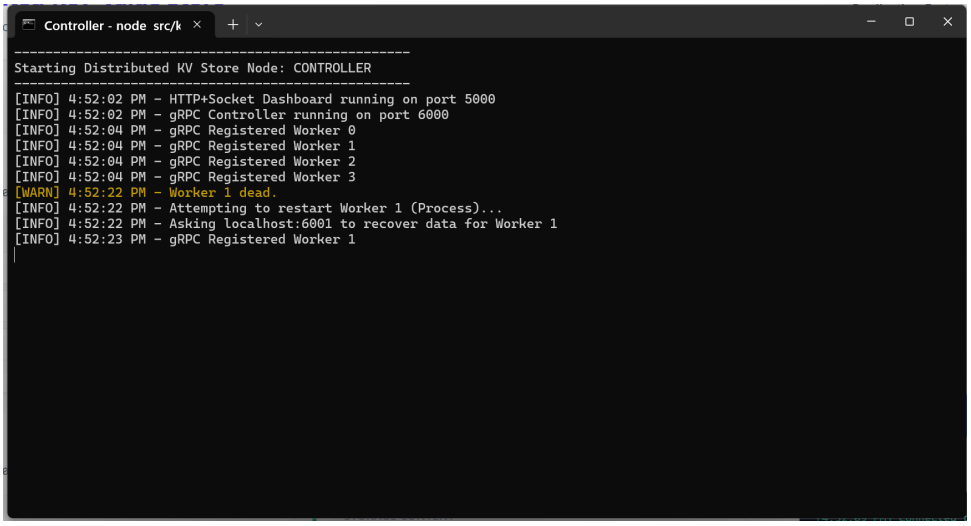

Figure 7: Fault Injection Demonstration

6 Technology Stack Selection

The selection of technologies for this distributed key-value store was driven by the specific requirements of the CAP theorem, specifically targeting high availability and partition tolerance. Each component was chosen to optimize for latency, throughput, or developer ergonomics.

6.1 Backend Runtime: Node.js

Node.js was selected as the core runtime environment for both the Controller and Worker nodes.

- **Event-Driven Architecture:** Distributed systems rely heavily on network I/O (waiting for replication acknowledgments, heartbeats, and client requests). Node.js's non-blocking I/O model ensures that a worker node is never halted while waiting for a disk write or a network response, allowing it to handle thousands of concurrent operations on a single thread.
- **Unified Language:** Using JavaScript across the entire stack (Backend and Frontend) simplified the sharing of logic, particularly configuration constants and validation routines.

6.2 Internal Communication: gRPC & Protocol Buffers

While external clients interact with the system via standard HTTP/REST for compatibility, internal node-to-node communication utilizes gRPC.

- **Performance:** Unlike JSON, which is text-based and verbose, gRPC uses **Protocol Buffers** (Protobuf), a binary serialization format. This results in payloads that are 30-40% smaller and significantly faster to serialize/deserialize, reducing the network overhead during high-volume data replication.
- **Strong Typing:** The use of `.proto` files defines a strict contract between nodes. This prevents classing errors where a Worker might send a malformed heartbeat to the Controller, ensuring system stability.

6.3 Real-Time Synchronization: Socket.IO

To fulfill the requirement of a "Live Dashboard" without relying on inefficient short-polling, Socket.IO was integrated.

- **Event Propagation:** The Controller acts as an event emitter. When a node's status changes (e.g., from Online to Offline), a message is pushed instantly to the frontend.

- **Visualizing Partitioning:** Socket.IO enables the dashboard to reflect the current replica count configuration in real-time, providing immediate visual feedback when the user adjusts the consistency parameters.

6.4 Frontend Framework: Vue.js 3 & Tailwind CSS

- **Reactivity System:** Vue 3's Composition API allows for fine-grained reactivity. When a Socket.IO event arrives, Vue automatically updates only the specific DOM elements (like a single worker's status card) without re-rendering the entire page.
- **Rapid UI Development:** Tailwind CSS provided utility-first classes that sped up the styling process, ensuring a clean, modern interface that focuses on readability of complex system data.

6.5 Hashing Algorithm: MD5

- **Uniform Distribution:** For Consistent Hashing to work effectively, keys must be spread evenly across the integer space (0 to 2^{32}). MD5 acts as a chaotic mixing function that provides excellent uniformity.
- **Speed vs. Security:** While MD5 is no longer considered cryptographically secure against collision attacks, it is significantly faster than SHA-256. In the context of database partitioning, speed and distribution uniformity are prioritized over cryptographic resistance.

Component	**Technology**	**Justification**
Runtime	Node.js	Asynchronous, event-driven architecture is ideal for I/O-heavy distributed systems.
Internal RPC	gRPC + Protobuf	Binary serialization provides strict typing and better throughput than text-based REST.
Real-Time	Socket.IO	Enables instant state propagation to the frontend without polling.
Frontend	Vue.js 3	Reactive data binding makes it easy to reflect complex cluster states.
Hashing	MD5	Provides a uniform distribution of keys across the 32-bit integer space.

Table 1: Summary of Technology Stack Selection

7 Conclusion & Future Scope

7.1 Conclusion

This project stands as a comprehensive implementation of a fault-tolerant distributed key-value store, successfully meeting and exceeding the core design requirements. By transcending basic functionality, the system integrates advanced architectural patterns typically found in industrial-grade NoSQL solutions. The adoption of **Consistent Hashing** ensures equitable data distribution and minimizes reorganization overhead during scaling events. Furthermore, the hybrid communication strategy—leveraging **gRPC** for high-performance internal node synchronization and REST for external client compatibility—demonstrates a sophisticated understanding of network optimization.

The implementation of **Latent Replication** critically balances the trade-off between write latency and data durability, offering a user experience that remains responsive under load. Finally, the real-time, event-driven dashboard transforms abstract distributed computing concepts into observable phenomena, providing immediate visual verification of the system's resilience and self-healing capabilities.

7.2 Future Scope

To further harden the system for production-grade workloads, the following enhancements are proposed:

- **Hinted Handoff:** Currently, temporary node failures result in a transient dip in replication factor. Implementing Hinted Handoff would allow the coordinator node to temporarily queue write operations intended for the offline node. Upon the node's recovery, these "hints" would be automatically replayed, significantly accelerating the cluster's return to a consistent state and preventing data staleness.

- **Merkle Trees (Anti-Entropy):** To optimize the data recovery process, the system could incorporate Merkle Trees. Rather than exchanging full datasets to identify discrepancies, nodes would compare hierarchical hash trees. This would allow the system to pinpoint exactly which key ranges differ with minimal network overhead, making the synchronization process far more bandwidth-efficient as the dataset grows.

- **Persistent Storage Engine:** While the current file-based JSON storage serves as a functional proof-of-concept, it faces I/O bottlenecks with large datasets due to the need for full-file parsing. Transitioning to a Log-Structured Merge-tree (LSM-tree) engine, such as **LevelDB** or **RocksDB**, would unlock high-throughput sequential writes and efficient range queries, enabling the system to handle data volumes far exceeding available RAM.