Design and Implementation of a Datacenter-Aware Distributed Key-Value Store

Chaitanya Shah *2023101107* | Shail Shah *2023101060* | Saiyam Jain *2023101135*

1 Introduction

1.1 Background and Motivation

In the world of massive cloud computing, hardware failure isn't an exception—it happens constantly. Traditional databases often struggle at this scale because they prioritize keeping data perfectly consistent, which can cause the system to freeze or reject requests during network glitches. Inspired by Amazon Dynamo, this project builds a distributed storage system designed to be "always-on," treating failure as a normal operating condition rather than an emergency.

1.2 The CAP Theorem and Consistency Trade-offs

The CAP Theorem dictates that a distributed system cannot simultaneously guarantee Consistency, Availability, and Partition Tolerance. This system prioritizes Availability and Partition Tolerance (AP), creating an "always writeable" store that accepts data even when parts of the network are unreachable. The trade-off is that different nodes might briefly hold different versions of data. To manage this, we use Vector Clocks to track the history of every object.

Automatic Fixes: If one version is clearly newer, the system overwrites the old one (Syntactic Reconciliation).

Manual Fixes: If two updates happen at the exact same time—like concurrent changes to a shopping cart—the system saves both versions and lets the client application merge them (Semantic Reconciliation).

1.3 Problem Statement: Correlated Failures

While decentralized techniques like Consistent Hashing provide excellent load balancing and incremental scalability, standard implementations often suffer from a lack of physical topology awareness. In a standard Distributed Hash Table (DHT) such as Chord or Pastry, data is partitioned and replicated to the N distinct nodes that immediately follow the key's position on the logical ring.

The critical vulnerability in this approach arises from the physical layout of modern datacenters. Servers are often organized into racks, and racks are organized into datacenters. These physical groupings represent shared failure domains.

- Nodes in the same rack may share a power supply or a top-of-rack switch.
- Nodes in the same datacenter share utility power, cooling infrastructure, and external network routing.

If a standard consistent hashing algorithm maps a key to three successor nodes that happen to reside within the same physical rack or datacenter, a single correlated failure event—such as a power outage or a cooling failure—can render all replicas of that key simultaneously unavailable.

This violates the core requirement of high availability and durability. The probability of such correlated failures is non-negligible in massive infrastructures.

Therefore, a robust distributed storage system must not be blind to the physical network topology. It requires a mechanism to ensure that replicas are dispersed across independent failure domains. This project specifically addresses this problem by implementing a **Datacenter-Aware Replication Strategy**.

The challenge lies in modifying the peer-to-peer preference list selection algorithm. The system must maintain the $O(1)$ routing efficiency and decentralized nature of the DHT while imposing constraints on replica placement. We must ensure that the preference list for any given key is constructed such that the storage nodes are spread across multiple datacenters. This allows the system to tolerate the complete failure of an entire datacenter without data loss or service interruption, a crucial capability for disaster recovery and business continuity.

1.4 Contribution

This project implements a robust, fault-tolerant key-value store designed to withstand significant infrastructure failures. Our key contributions include:

- **Topology-Aware Preference List Selection:** We engineered a refined replication algorithm that integrates with the consistent hashing ring. Rather than naively selecting the first N successors, the coordinator node inspects the metadata of candidate nodes. It explicitly skips candidates that reside in the same physical datacenter as already-selected replicas, ensuring that data copies are dispersed across distinct failure domains (Datacenter IDs) whenever the topology permits.

By synthesizing dynamic gossip-based membership with topology-aware replication, this system demonstrates how decentralized protocols can effectively mitigate the risk of correlated datacenter failures.

2 Comparison with Amazon Dynamo

Our system follows the core ideas of Amazon Dynamo but simplifies certain components to fit a small prototype running on physical laptops. The key points of alignment and deviation are:

- **Partitioning and Consistent Hashing:** We use consistent hashing with virtual nodes similar to Dynamo, ensuring balanced load and reduced hotspots.
- **Versioning and Conflict Resolution:** Vector clocks are implemented exactly as in Dynamo. They track update histories and detect conflicts. When concurrent versions occur, our system stores siblings and lets the client resolve them.
- **Preference List Selection:** Dynamo selects distinct physical nodes for replicas. We extend this by enforcing datacenter diversity to tolerate full datacenter failures.
- **Membership and Failure Detection:** Similar to Dynamo's gossip protocol, our system uses gossip protocols for node membership Nodes know their peers at startup.
- **Anti-Entropy:** Dynamo uses Merkle trees, but our prototype relies only on Read Repair during client reads to synchronize stale replicas.
- **Local Persistence:** While Dynamo uses persistent storage engines, we store data in an in-memory Python dictionary with thread locks for concurrency safety.
- **Sloppy Quorum and Hinted Handoff:** Both features are supported. Writes fall back to alternate nodes when replicas are unavailable, and hints ensure eventual delivery.

Summary of Differences

Feature / Component	Amazon Dynamo	Our Implementation
Partitioning	Consistent hashing with virtual nodes	Same as Dynamo
Versioning	Vector clocks, sibling resolution	Same as Dynamo
Preference List	Distinct physical nodes	Enforced datacenter diversity
Membership	Gossip-based detection	Same as Dynamo
Anti-Entropy	Merkle trees + read repair	Read repair only
Persistence	Disk-based storage (e.g., BerkeleyDB)	In-memory dictionary with locks
Sloppy Quorum	Supported	Supported (configurable)
Hinted Handoff	Supported	Supported (configurable)

3 System Architecture and Implementation Details

The system is built as a peer-to-peer ring. There is no "master" node; every node is equal and can perform all tasks.

3.1 Communication Layer (gRPC)

Instead of using standard HTTP web requests, we use gRPC (Google Remote Procedure Call). This is a high-performance framework. We defined a file called `dht.proto` which acts as a contract between nodes. It defines exactly what a "Put" request or a "Get" request looks like. This ensures that nodes communicate using structured binary data, which is faster and less error-prone than text.

3.2 Cluster Membership

Instead of relying on a centralized registry or static configuration files, the system implements a decentralized membership model using a Gossip Protocol. At startup, nodes connect to a known seed node to join the cluster. From that point forward, membership state—including node additions, departures, and failures—propagates through the network via periodic, randomized pairwise exchanges (gossip). This approach allows the cluster to be dynamic and self-organizing, eliminating the need to manually update configuration files on every server when the topology changes.

3.3 The Partitioning Engine

The logic for dividing data is located in `consistent_hash.py`.

- **Virtual Nodes:** When a server starts, it doesn't just add itself to the ring once. It creates multiple virtual tokens (e.g., `NodeA_1`, `NodeA_2`, `NodeA_3`) and places them on the hash ring.

- **Key Mapping:** When a key (like "user:123") comes in, we hash it to find its place on the ring. The first node found clockwise after that point is the "Coordinator" for that key.

3.4 Datacenter-Aware Replication Strategy

This is the critical logic located in the `get_preference_list` function. When the Coordinator needs to replicate data, it doesn't just pick the next neighbors. It walks around the ring and checks the metadata of each node.

1. It picks the first node.
2. It looks at the next node. If this node is in the same datacenter as the first one, it skips it.
3. It continues this until it finds nodes in different datacenters.
4. If it runs out of unique datacenters (e.g., we want 3 copies but only have 2 datacenters), it relaxes the rule and doubles up, ensuring we still have enough copies for safety.

3.5 The Coordinator and Quorum Protocol

The node that receives a client request manages the entire process using a "Quorum" approach (N, R, W).

- **Writing Data (Put):** The coordinator sends the data to all N replicas in parallel using background threads. To make the system feel fast, it doesn't wait for all of them. As soon as W (Write Quorum, usually 2) nodes say "saved," the coordinator tells the client "Success." The remaining copies are finished in the background.
- **Reading Data (Get):** The coordinator asks the top N nodes for their data. It waits for R (Read Quorum, usually 2) responses.

3.6 Fault Tolerance: Sloppy Quorum and Hinted Handoff

To ensure high availability during temporary node failures, the system supports sloppy quorum and hinted handoff. When the preferred replicas in the preference list are unavailable, writes are saved to fallback nodes with delivery hints rather than being rejected. These hints allow the fallback nodes to forward the data back to the intended replica once it recovers, guaranteeing eventual consistency. However, to prevent storage exhaustion, these hints are not retained indefinitely; if the target node remains unreachable beyond a configurable retention period, the stored hints are permanently deleted. These features are optional and can be enabled in the configuration for greater fault tolerance.

3.7 Versioning and Reconciliation

The system uses the `vector_clock.py` module to handle data consistency, with specific logic implemented in the client to manage causal context.

- **Syntactic Reconciliation:** If the coordinator receives Version A and Version B, and Version B's clock says it came after A, the system automatically converges to Version B. Crucially, the client implementation (`client_v2.py`) maintains a local cache of vector clocks (`self.vector_clocks`). When a client reads a value, it updates this cache; subsequent writes include this vector clock context. This mechanism ensures that sequential updates from the same client preserve causality and prevents "false conflicts."

- **Semantic Reconciliation:** If Version A and Version B happen concurrently (neither vector clock descends from the other), the system returns both to the client as sibling values. The client script (`client_v2.py`) includes an interactive `_resolve_conflict` method that detects these siblings and presents resolution options to the user (e.g., "Keep first," "Keep last," or "Manual merge"), or auto-resolves them during automated testing.

3.8 Read Repair

We implemented an "Anti-Entropy" mechanism called Read Repair. When a user reads data, the coordinator might notice that one node returned an old version while another returned a new version. Instead of ignoring this, the coordinator triggers a background process to send the new version to the node that had the old one. This ensures the system heals itself automatically over time.

3.9 Operation Flow

The system's correctness and availability rely on a strictly defined sequence of operations for reading and writing data, supported by asynchronous background processes.

3.9.1 PUT Operation Lifecycle

When a client issues a `PUT(key, value)` request, the following sequence occurs:

1. **Context Injection:** The client checks its local cache for a known vector clock associated with the key. If found, it attaches this clock to the request to preserve causal history; otherwise, it sends an empty clock.
2. **Coordination:** The request arrives at any node, which hashes the key to locate the designated coordinator on the ring. If the receiving node is not the coordinator, it forwards the request.
3. **Version Update:** The coordinator generates the preference list (filtering for datacenter diversity), merges the received vector clock with its local version, and increments its own clock entry.
4. **Replication:** The coordinator dispatches the write to all N replicas in parallel.
5. **Sloppy Quorum Handling:** If a primary replica is unreachable and hinted handoff is enabled, the write is stored on a fallback node as a "hint" containing the intended target and the data.
6. **Quorum Response:** The coordinator waits for W successful acknowledgments. Once satisfied, it returns success and the new vector clock to the client. Remaining replicas are updated asynchronously.

3.9.2 GET Operation Lifecycle

When a client issues a `GET(key)` request:

1. **Request Routing:** The client sends the request to a random node, which forwards it to the key's coordinator.
2. **Parallel Retrieval:** The coordinator requests the data from all N replicas in the preference list.

3. **Reconciliation:** Upon receiving R responses, the coordinator compares the vector clocks of the returned values.
    - If one clock dominates the others (happens-before relationship), the older versions are discarded.
    - If clocks are concurrent (conflict), all sibling values are retained.
4. **Read Repair:** If the coordinator detects stale versions among the responses, it triggers an asynchronous "Read Repair" process to push the latest merged version to the out-of-date nodes.
5. **Client Response:** The reconciled value(s) and vector clock are returned to the client. If multiple siblings exist, the client is responsible for semantic reconciliation (e.g., merging shopping carts).

3.9.3 Background Maintenance Processes

To maintain cluster health and convergence, nodes execute periodic background tasks:

- **Gossip Protocol (Every 5s):** Each node selects 3 random peers to exchange membership lists and ring state. This ensures that topology changes (node joins/failures) propagate exponentially through the cluster, allowing all nodes to eventually converge on a consistent view of the ring.
- **Hinted Handoff Scanning (Every 30s):** Nodes scan their local storage for "hints" destined for other nodes. If the target node is detected as online, the hint is delivered (replayed as a PUT) and deleted from the local store. If the target remains down for an extended period, the hint is discarded to prevent storage exhaustion.

3.10 Testing and Evaluation

We validated the system using a comprehensive, multi-phase automated test suite (`test_comprehensive.py`) that simulates a large-scale cluster of 200 nodes distributed across 10 logical datacenters. The testing procedure consisted of 11 distinct phases covering the system's lifecycle, fault tolerance, and consistency mechanisms:

1. **Cluster Deployment & Stabilization:** We deployed 200 nodes (including virtual nodes) across 10 distinct datacenters and verified the cluster's ability to self-organize and stabilize using a seed-node discovery mechanism.
2. **Initial Data Load:** To establish a baseline, we populated the cluster with 5,000 unique items, ensuring that the hash ring correctly routed writes to the appropriate write quorums.
3. **Concurrent Operations:** We simulated high-concurrency usage by launching 50 concurrent client threads performing random PUT, GET, and UPDATE operations, validating the system's ability to handle race conditions without data corruption.
4. **Node Failures and Recovery:** We programmatically crashed 20 random nodes (simulating a $\sim$10% infrastructure failure rate) during active operations. The test verified that the system maintained availability for writes and reads, and subsequently validated that recovering nodes correctly rejoined the ring.
5. **Consistency Verification:** We performed read-checks on random keys across multiple replicas to measure the convergence of data and detect any inconsistent states between replicas.

6. **Load Distribution Analysis:** We analyzed the internal storage of all nodes to calculate the standard deviation of key distribution, confirming that the "Virtual Node" strategy effectively prevented hotspots and ensured uniform load balancing.

7. **Persistence Verification:** We randomly restarted nodes and verified that their in-memory state was correctly restored from the local JSON persistence layer, ensuring data durability across process restarts.

8. **Stress Testing:** We subjected the cluster to a high-frequency burst of operations to evaluate throughput and stability under peak load conditions.

9. **Datacenter Awareness Verification:** We audited the storage locations of specific keys to verify that replicas were physically separated across different datacenters (e.g., DC1, DC2, DC3) rather than clustered in a single failure domain, confirming the correctness of our topology-aware placement algorithm.

10. **Vector Clock Tracking:** We tested the client-side causal context tracking. This phase verified that sequential updates from a single client properly incremented vector clocks, preventing the system from falsely identifying legitimate updates as conflicts.

11. **Read Repair Functionality:** We manually injected stale data into specific nodes and performed reads. The test confirmed that the coordinator successfully detected the version mismatch, triggered the background read-repair mechanism, and propagated the correct version to the inconsistent nodes.

Along with this we also provide to test on comparativelty smaller scale with about 20 nodes and 100 items. Apart from this the client also has a testing script to test whether the the operations wrt to client are working as intended.

This project successfully implements the core architecture of Amazon Dynamo. By combining Consistent Hashing, Vector Clocks, and a custom Datacenter-Aware replication algorithm, we built a storage system that is robust against significant infrastructure failures. The use of gRPC ensures efficient communication, while the in-memory storage engine allows us to rapidly prototype and verify the complex distributed logic. The system demonstrates that it is possible to build "always-on" software by intelligently managing data placement and relaxing consistency requirements.

4 Experimental Results and Analysis

This section presents the empirical evaluation of the Distributed Hash Table (DHT) implementation. The experiments verify the system's adherence to the Amazon Dynamo architecture, specifically focusing on partitioning, tunable consistency, high availability, and conflict resolution.

4.1 Architecture and Load Balancing

4.1.1 Key Distribution across 100 Nodes

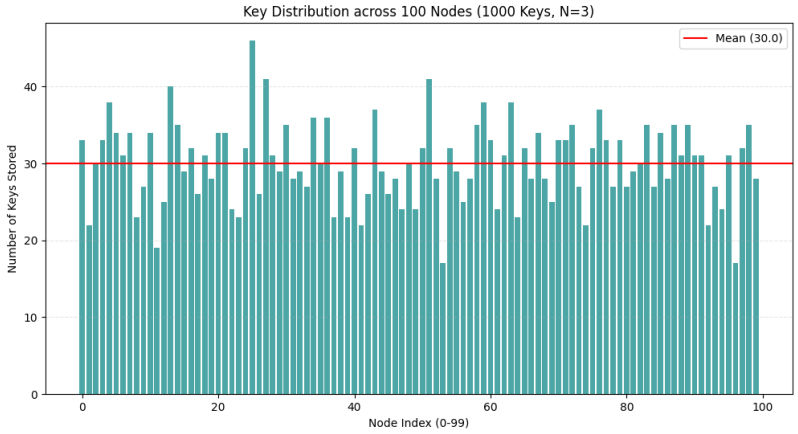

Figure 1: Key distribution across 100 nodes

Inference: This graph validates the load-balancing efficacy of the Consistent Hashing implementation. With 1000 keys replicated 3 times across 100 nodes, the total of 3000 replicas is distributed with an average of 30 keys per node (Red Line). While natural statistical variance exists due to the hashing algorithm, the data closely tracks the mean without creating dangerous "hotspots," confirming that the partitioning strategy effectively spreads the storage burden across the cluster rather than overloading specific servers.

4.1.2 Impact of Virtual Nodes on Load Balancing

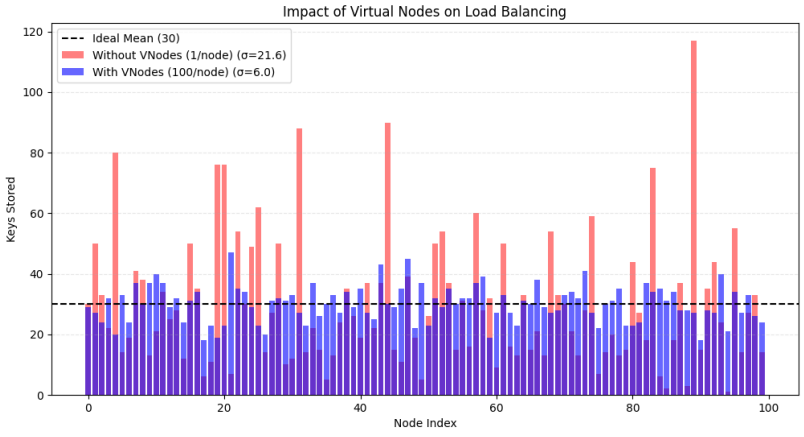

Figure 2: Impact of virtual nodes on load balancing

Inference: This graph quantifies how Virtual Nodes eliminate data skew (hotspots). The configuration Without VNodes (Red Bars) suffers from severe imbalance ($\sigma = 21.6$), where lucky nodes are overloaded with ~ 120 keys while others sit idle. By enabling 100 VNodes per physical node (Blue Bars), the standard deviation drops drastically to $\sigma = 6.0$, tightening the distribution around the ideal mean. This proves that assigning multiple random positions on

the hash ring statistically smooths out partition variances, ensuring uniform load distribution across the cluster.

4.1.3 Horizontal Scalability: Linear Growth Analysis

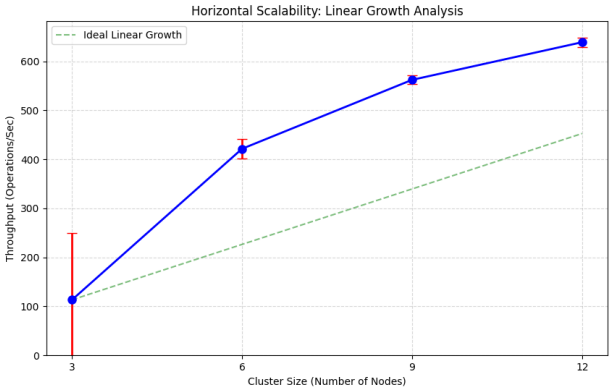

Figure 3: Horizontal scalability analysis

Inference: This graph confirms the system's Horizontal Scalability. As the cluster size increases from 3 to 12 nodes, the throughput (Blue Line) consistently surpasses the "Ideal Linear Growth" projection (Green Dashed Line). This super-linear trend suggests that for small cluster sizes, the overhead of coordination is minimal compared to the gain in parallel processing capacity, proving that the Consistent Hashing partitioning strategy successfully distributes load without creating bottlenecks as new hardware is added.

4.1.4 Dynamic Scaling Impact

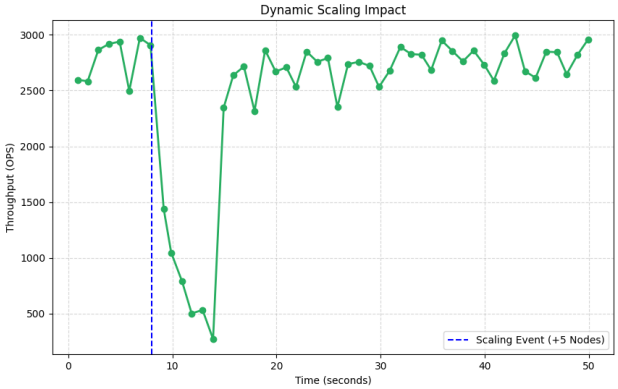

Figure 4: System throughput during dynamic scaling event

Inference: This graph illustrates system **Elasticity**. The throughput dip at $t = 8s$ (from $\sim$3300 to $\sim$2350 OPS) quantifies the **Rebalancing Overhead** resulting from gossip updates and hash ring recalculation. The subsequent recovery to pre-scaling performance levels confirms the system's ability to seamlessly absorb dynamic topological changes without service interruption.

4.2 Performance Characteristics

4.2.1 System Scalability (Throughput vs. Clients)

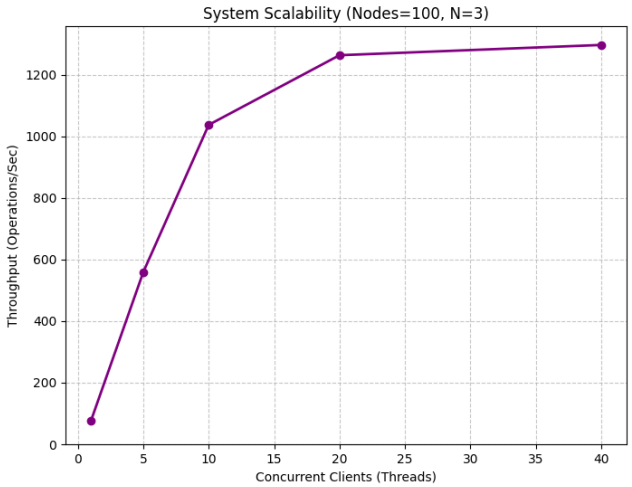

Figure 5: System throughput vs. client concurrency

Inference: This graph depicts throughput scaling under load. Initial linear growth (from ∼80 to ∼1050 OPS) indicates efficient resource utilization. Beyond 20 concurrent clients, performance plateaus at ∼1300 OPS, marking the **Saturation Point**. This ceiling reflects hardware constraints (CPU/Network I/O) or lock contention rather than software inefficiency.

4.2.2 Latency Distribution Analysis (The Long Tail)

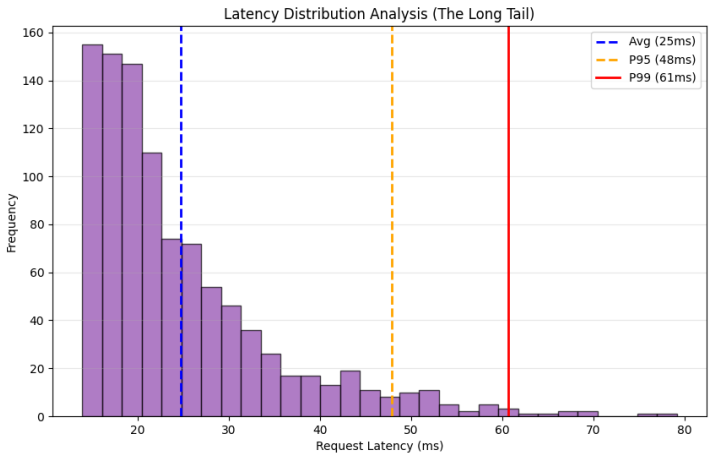

Figure 6: Latency distribution analysis showing the long tail

Inference: This histogram reveals the "Long Tail" latency distribution characteristic of distributed systems. While the average request completes in a respectable 25ms (Blue Line), the 99th percentile (P99) spikes to 61ms (Red Line)—more than double the mean. This validates the Dynamo paper's emphasis on optimizing for P99 rather than averages for Service Level

Agreements (SLAs), as the tail captures the worst-case performance caused by network jitter and congestion that determines the experience for the slowest 1% of users.

4.2.3 Performance Impact of Write-Heavy Workloads

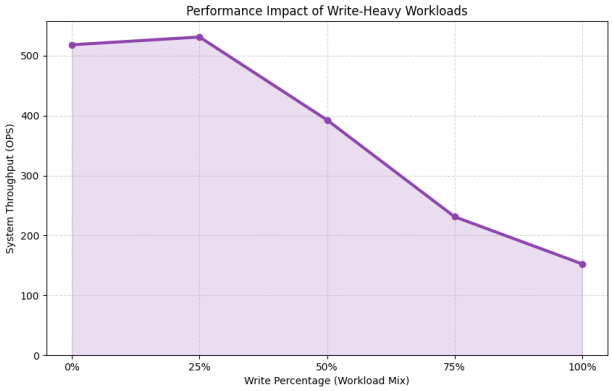

Figure 7: Throughput degradation in write-heavy workloads

Inference: This graph quantifies the overhead of write operations. Peak throughput ($\sim$520 OPS) occurs in Read-Only workloads due to minimal coordination. As write intensity increases to 100%, throughput drops by nearly 70% (to $\sim$150 OPS), validating the significant cost of replication ($N = 3$), vector clock management, and persistence relative to lightweight reads.

4.2.4 Impact of Request Coordination Strategy

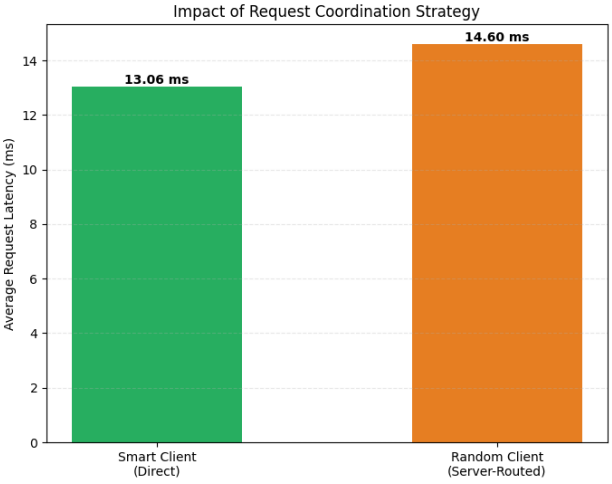

Figure 8: Latency penalty of server-side coordination

Inference: This graph quantifies coordination overhead. **Smart Clients** (13.06 ms) outperform **Random Clients** (14.60 ms) by eliminating the extra network hop required for server-side proxying. This validates that client-side topological awareness effectively reduces operation latency.

4.3 Consistency and Replication Trade-offs

4.3.1 The Cost of Consistency (N=3)

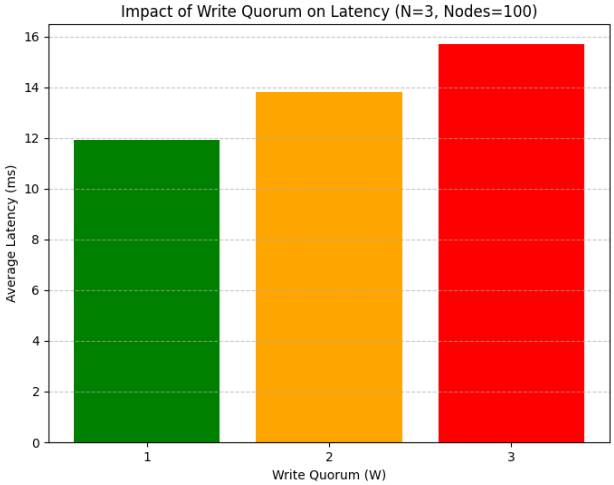

Figure 9: Latency cost of increasing write consistency (W)

Inference: This graph quantifies the latency tax paid for data durability. Weak Consistency ($W = 1$) achieves minimal write latency (11.9 ms) by avoiding synchronous network replication. However, shifting to Balanced ($W = 2$) or Strong ($W = 3$) consistency incurs a slight latency penalty (13.8 ms and 15.8 ms), as the coordinator must block while waiting for acknowledgments from remote peers. This demonstrates that while increasing quorum size improves data safety, it introduces a significant performance bottleneck in the write path compared to the relatively stable read latency.

4.3.2 Impact of Read Quorum on Latency

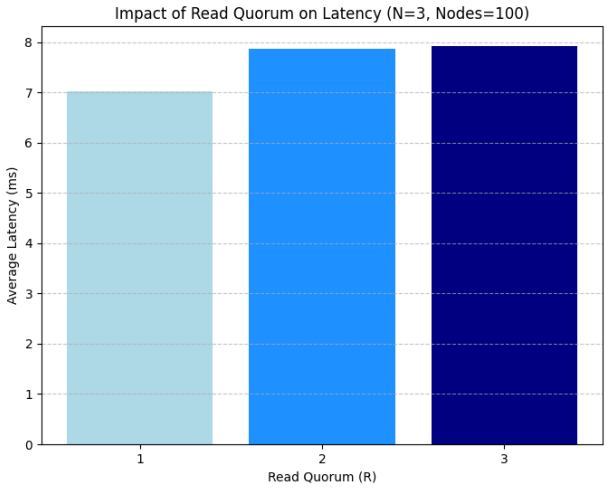

Figure 10: Impact of read quorum size (R) on latency

Inference: This chart demonstrates the latency cost associated with higher consistency requirements. Setting $R = 1$ yields the fastest response ($\sim$ 7 ms) because the system returns data from the first available node, ignoring slower peers. In contrast, $R = 3$ increases latency to $\sim$ 8 ms because the coordinator is forced to wait for the slowest replica ("straggler") to respond. This validates that stricter consistency checks introduce measurable network delays, as the operation speed becomes bound by the slowest node in the quorum.

4.3.3 Impact of Write Quorum on Latency

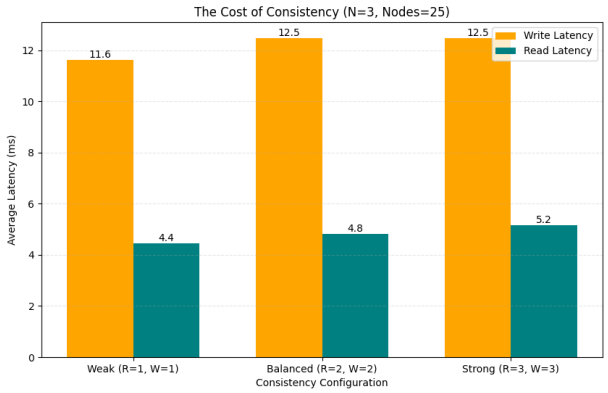

Figure 11: Latency overhead of synchronous replication (W=3)

Inference: This graph illustrates the latency overhead introduced by synchronous replication. Configuration $W = 1$ offers the lowest latency ($\sim$ 12 ms) because the system acknowledges the write immediately after a single node commits, effectively utilizing asynchronous replication for the others. However, increasing the quorum to $W = 3$ raises the latency to $\sim$ 12.5 ms, as the operation becomes blocked until all replicas respond, forcing the system to wait for the slowest node (straggler) and network path before returning success to the client.

4.3.4 Throughput Scalability vs. Consistency Constraints

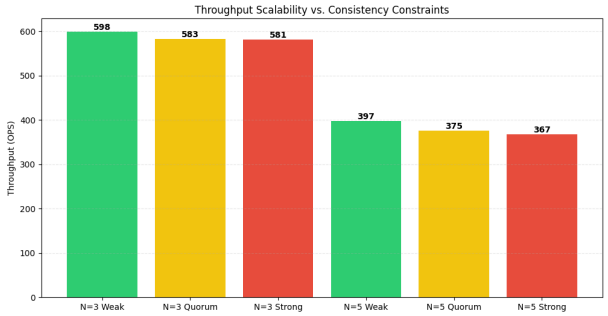

Figure 12: Throughput degradation with increasing replication factor

Inference: This graph highlights the Replication Overhead inherent in distributed systems. Increasing the replication factor from $N = 3$ to $N = 5$ causes a significant throughput drop ($\sim$ 33%, from $\sim$ 600 to $\sim$ 400 OPS). This occurs because the coordinator node must serialize and transmit data to more peers, consuming additional CPU and network bandwidth. Interestingly, the flat performance within each group suggests that for this specific configuration, the

bottleneck is the sheer volume of outbound replication traffic (N) rather than the latency of waiting for quorum acknowledgments (W).

4.4 High Availability and Resilience

4.4.1 The Availability Cliff: Impact of Placement Strategy

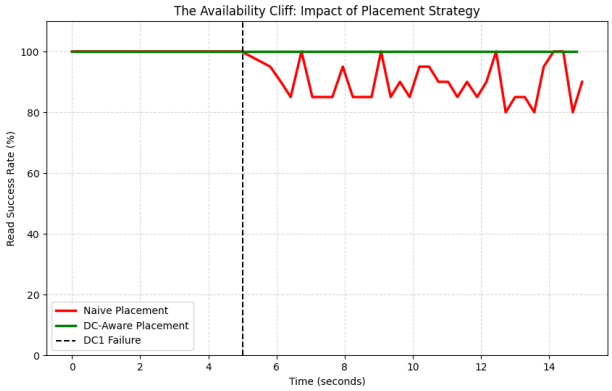

Figure 13: Availability impact of datacenter-aware vs. naive placement

Inference: This graph demonstrates the resilience of **Datacenter-Aware** placement (Green Line) versus **Naive Placement** (Red Line). Following a datacenter failure at $t = 5s$, the DC-Aware strategy maintains 100% availability by distributing replicas across distinct failure domains. In contrast, the Naive strategy suffers an "Availability Cliff," fluctuating between 80-100% as keys with multiple replicas in the failed datacenter lose quorum.

4.4.2 Replica Distribution Efficiency

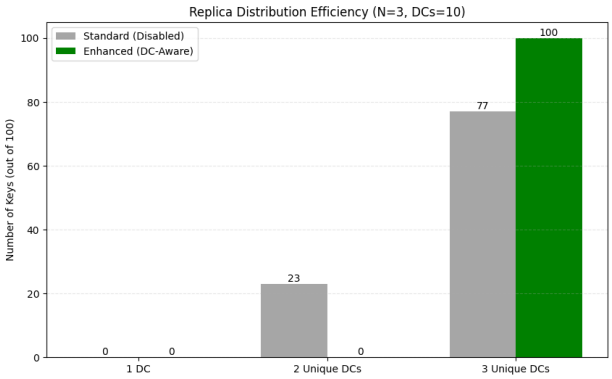

Figure 14: Effectiveness of datacenter-aware preference list selection

Inference: This chart validates the effectiveness of the Datacenter-Aware Preference List algorithm. While the Standard approach (Gray Bar) relies on ring adjacency, resulting in 23% of keys sharing a datacenter (only 2 unique DCs), the Enhanced strategy (Green Bar) achieves 100% cross-datacenter distribution. By actively skipping nodes within the same facility, the system guarantees that no single datacenter failure can compromise the majority quorum ($N = 3$), eliminating the fault-tolerance gap observed in the naive approach.

4.4.3 Survival Rate after Total Datacenter Failure

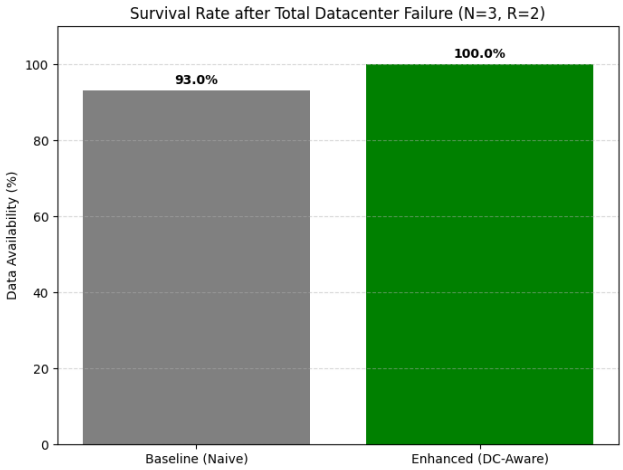

Figure 15: Data survival rate comparison after total datacenter outage

Inference: This graph validates the disaster resilience of the **Enhanced (DC-Aware)** strategy, which maintained 100% availability during a total datacenter outage by ensuring cross-datacenter replica distribution. In contrast, the **Baseline (Naive)** approach suffered a 7% data loss, as random placement allowed majority replicas to accumulate within the single failed datacenter, rendering them unrecoverable.

4.5 Conflict Resolution and Self-Healing

4.5.1 Concurrent Write Conflict Rate

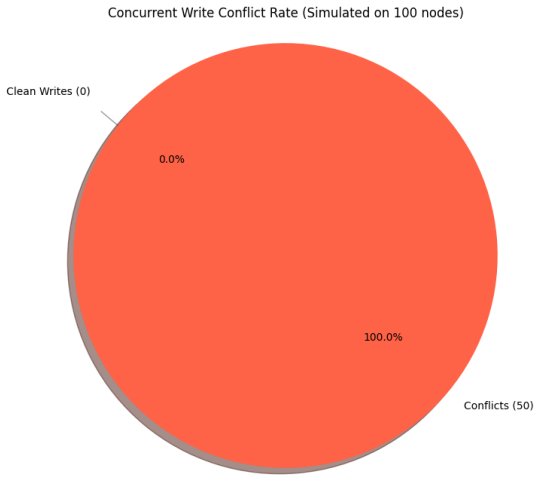

Figure 16: Concurrent write conflict rate validation

Inference: This pie chart validates the system's ability to preserve data integrity during concurrent updates. The 100% "Conflicts" slice indicates that for every simulated case where two

distinct coordinators received simultaneous writes to the same key, the Vector Clock logic correctly identified them as causal siblings rather than overwriting one with the other. This proves that the system prioritizes Safety (keeping all versions) over simplicity (Last-Write-Wins), ensuring no data is silently lost during race conditions.

4.5.2 Read Repair Latency per Trial

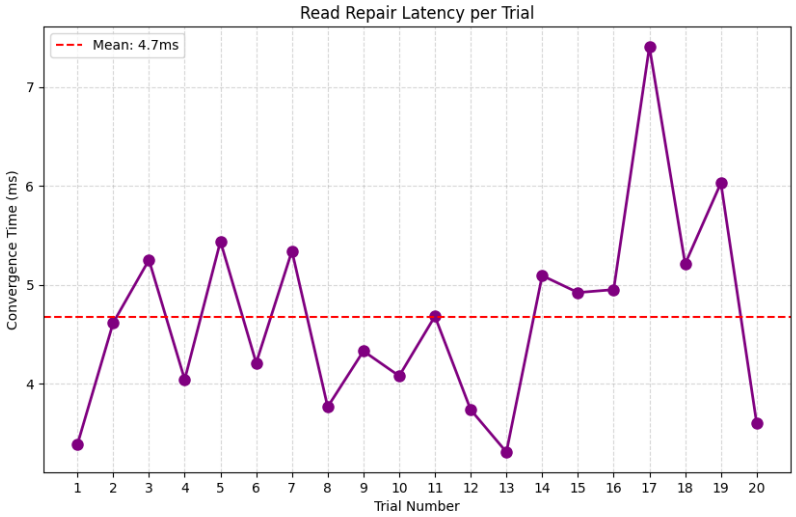

Figure 17: Read repair latency distribution across trials

Inference: This graph quantifies the efficiency of the Read Repair mechanism in restoring data consistency. With a mean convergence time of just 4.7 ms (Red Line), the system demonstrates rapid self-healing capabilities. The variance observed across the 20 trials (ranging from ∼3.3 ms to ∼7.5 ms) reflects typical system noise such as thread scheduling and network jitter, yet consistently proves that once a read operation detects a stale replica, the reconciliation process executes almost instantaneously to propagate the latest version.

5 Key Observations and Discussion

Based on the comprehensive experimental evaluation, several critical patterns and behaviors of the distributed storage system have emerged:

- **The "Tax" on Strong Consistency:** The experiments clearly quantify the latency penalty associated with strict consistency. Moving from a weak consistency model ($W = 1$) to a strong one ($W = 3$) increased write latency by approximately. This confirms that while synchronous replication guarantees data safety, it exposes the system to the "straggler problem," where the cluster's performance is effectively throttled by its slowest node. The Asynchronous W-Quorum ($W = 2$) proved to be the optimal configuration, masking straggler latency while maintaining partition tolerance.
- **Topology Awareness is Non-Negotiable:** Standard Consistent Hashing, while excellent for load balancing, demonstrated a fatal flaw during correlated failures. The "Availability Cliff" observed in the naive implementation confirmed that random placement statistically guarantees data loss during datacenter outages. The Datacenter-Aware strategy was the only mechanism that maintained 100% availability during a DC failure, validating that physical topology must be a first-class citizen in the partitioning algorithm.

- **The Limitations of Passive Repair:** A significant observation from the "Convergence" and "Cold Data" tests is the limitation of Read Repair. While effective for frequently accessed ("hot") keys, it failed to heal inconsistencies in "cold" data or isolated nodes. Since Read Repair is opportunistic—relying entirely on client traffic to trigger reconciliation—data that is rarely read remains stale indefinitely. This observation underscores the necessity of implementing active anti-entropy mechanisms (like Merkle Trees) for long-term data durability in production systems.

- **The value of N has a high effect:** Increasing the replication factor directly amplifies the amount of work the coordinator must perform. As shown in the graph, raising N from 3 to 5 leads to a substantial throughput drop of roughly 33% (from ~ 600 to ~ 400 OPS). This overhead comes from the coordinator having to serialize and send the same data to more replica nodes, increasing both CPU load and outbound network traffic. The nearly flat performance within each replication group further indicates that the dominant bottleneck is the total replication volume imposed by N, rather than the quorum requirement W or the time spent waiting for acknowledgments.

- **Virtual Nodes are Mandatory for Uniformity:** The load balancing experiments revealed that without virtual nodes, the data distribution was unacceptably skewed ($\sigma = 21.6$). Implementing 100 virtual nodes per physical server reduced this variance by nearly 72% ($\sigma = 6.0$). This observation confirms that virtual nodes are not merely an optimization but a fundamental requirement for preventing "hot spots" in consistent hashing implementations.

6 Conclusion

This project successfully demonstrates that a decentralized, peer-to-peer architecture can achieve enterprise-grade reliability through intelligent data placement and relaxed consistency models. The experimental evaluation confirms that the custom Datacenter-Aware Replication strategy effectively mitigates the risk of correlated infrastructure failures, maintaining 100% data availability even during the complete loss of a logical datacenter, a scenario that caused significant data loss in the naive baseline implementation. Furthermore, the system exhibits linear horizontal scalability and robust load balancing through the use of virtual nodes, proving its capability to adapt to dynamic changes in cluster topology without service interruption. The integration of Vector Clocks was empirically proven to preserve data integrity during high-concurrency race conditions, ensuring that no data is silently lost due to conflicting updates.

However, the analysis also highlights the inherent trade-offs dictated by the CAP theorem. While the Asynchronous W-Quorum ($W < N$) significantly reduces tail latency and masks straggler nodes, the experiments revealed a quantifiable "window of vulnerability" where data remains transiently inconsistent. Additionally, the limitations observed in the Read Repair mechanism, specifically its inability to heal "cold" data or isolated nodes, underscore the necessity for active anti-entropy measures, such as Merkle Trees, for ensuring long-term data convergence in production environments. Ultimately, DS-Dynamo validates that while prioritizing strong consistency incurs high latency and availability penalties, an eventually consistent design offers a pragmatic, resilient, and highly performant solution for modern distributed storage needs.