Dynamo DB: Effectiveness of Anti-Entropy and Gossip Protocol

Karan Kumar Gangadhar
New York University, Courant
kk5409@nyu.edu

Satyanarayana Chillale
New York University, Courant
sc9960@nyu.edu

Abstract

We present a distributed key-value storage system inspired by Amazon's Dynamo DB, implementing eventual consistency through several key mechanisms: consistent hashing with virtual nodes for scalable data partitioning, gossip-based anti-entropy for replica synchronization and membership change propagation, vector clock versioning for conflict detection and resolution, and hinted handoff for temporary failures. The system uses a sloppy quorum approach allowing configurable read/write replica counts. We evaluate the system with three sets of experiments. First, we demonstrate how the gossip protocol and anti-entropy mechanisms reduce replica divergence and inconsistencies. Second, we measure the incidence of stale reads under different read/write quorum configurations (R and W values) to quantify the consistency–availability trade-off. Third, we investigate the impact of varying gossip and anti-entropy synchronization intervals on data consistency. Our results show that enabling periodic gossip and anti-entropy drastically lowers inconsistency, that stronger quorum settings (higher R and/or W) reduce stale reads, and that more frequent anti-entropy synchronization yields fewer stale reads. Notably, we observe that under induced node failures, a system with enabled failure-recovery protocols (gossip, anti-entropy, hinted handoff) can achieve fewer stale reads than a scenario with no failures but with these protocols disabled, underscoring the critical role of replica synchronization mechanisms in maintaining consistency. We discuss the design trade-offs, relate our findings to the original Dynamo system and other consistency models, and highlight how our implementation validates key concepts in weakly consistent distributed storage.

1 Introduction

Modern distributed systems often prioritize high availability and partition tolerance over strong consistency, embracing *eventual consistency* as a practical compromise Vogels [2009]. Amazon's Dynamo DeCandia et al. [2007] is a seminal example of this tradeoff: it sacrifices immediate consistency to ensure that the system remains always writable and available, even during network partitions or node failures.

Dynamo pioneered a set of now widely adopted techniques—*consistent hashing*, *sloppy quorums*, *vector clocks*, *gossip protocols*, *Merkle-tree-based anti-entropy*, and *hinted handoff*—to manage data in a decentralized cluster while tolerating partial failures. These mechanisms allow Dynamo to balance the CAP theorem tradeoffs in a configurable manner.

This project implements a Dynamo-inspired decentralized key-value store to explore these distributed consistency techniques in practice in a simulated environment. Our system replicates each key across N nodes and provides tunable consistency via configurable read (R) and write (W) quorums. By adjusting R and W, we can favor either consistency or availability: for instance, choosing $R + W > N$ ensures strong consistency under certain assumptions Gilbert and Lynch [2002], while lower quorum values favor lower latency and fault tolerance at the cost of potential stale reads.

Our implementation includes the following core components:

- **Partitioning and Replication:** We use consistent hashing with virtual nodes to evenly distribute keys and to minimize data movement during membership changes Karger et al. [1997].
- **Quorum-based Replication:** Each key is replicated to N nodes, and a write is considered successful after acknowledgments from W replicas. Reads query R replicas and reconcile versions using vector clocks.
- **Versioning and Conflict Resolution:** We associate each value with a vector clock Fidge [1988], allowing detection of concurrent updates. Conflicting versions are reconciled via timestamps.
- **Gossip Protocol:** Nodes periodically gossip membership and metadata with peers to disseminate updates and keep data consistent and up-to date.
- **Merkle Tree Anti-Entropy:** Nodes periodically synchronize their data using Merkle trees DeCandia et al. [2007] to efficiently identify and repair divergent replicas.
- **Hinted Handoff:** If a target replica is unreachable, the coordinator node temporarily stores a hinted version and delivers it when the node becomes available DeCandia et al. [2007].

We evaluate our system through a series of experiments measuring the effectiveness of these consistency protocols. Specifically, we:

1. Examine the role of gossip and anti-entropy in reducing inconsistencies across replicas.
2. Analyze the impact of quorum settings (R and W) on the percentage of stale reads observed by clients.
3. Investigate how the frequency of synchronization (e.g., anti-entropy intervals) influences the degree of data divergence.
4. Inject faults such as node failures and message drops to test the resilience of the system under adverse conditions.

Our findings confirm several key insights: frequent synchronization significantly reduces stale reads even under faults, and choosing quorums such that $R + W > N$ greatly reduces inconsistencies. Notably, a system with active anti-entropy and gossip mechanisms can outperform a stable but unsynchronized system—highlighting that continuous repair can be more critical to eventual consistency than mere system uptime.

The rest of the report elaborates prior work, our system architecture, experimental methodology, and results.

2 Related Work

Our work is directly inspired by Amazon's Dynamo DeCandia et al. [2007], which introduced a pragmatic design for an eventually consistent key-value store. Several core techniques in our system—including consistent hashing, vector clocks, sloppy quorums, hinted handoff, and Merkle-tree-based anti-entropy—are drawn from Dynamo. It was one of the first production systems to demonstrate that high availability and partition tolerance can be achieved by relaxing strong consistency, relying instead on conflict reconciliation for correctness.

Following Dynamo, several systems adopted similar architectural patterns. Apache Cassandra Lakshman and Malik [2010] extends Dynamo's decentralized design and quorum-based consistency but replaces vector clocks with last-write-wins using timestamps for simpler conflict resolution. While this reduces versioning overhead, it sacrifices the ability to detect concurrent updates.

Other Dynamo-inspired systems include Project Voldemort (LinkedIn) and Riak (Basho), both of which adopted vector clocks and tunable consistency levels. Riak, in particular, maintained configurability and reconciliation mechanisms, making it a close operational cousin to our system. Our project can be viewed as a simplified reimplementation of such systems for pedagogical evaluation.

Prior to Dynamo, Bayou Terry et al. [1995] explored eventual consistency through anti-entropy and user-defined merge procedures. It also introduced session guarantees, such as monotonic reads and

read-your-writes, which improve usability under weak consistency. Werner Vogels' work Vogels [2009] advocates for eventual consistency in practice, emphasizing that in many real-world applications, strict consistency is unnecessary. These ideas are grounded in the CAP theorem Gilbert and Lynch [2002], which formalizes the impossibility of achieving consistency, availability, and partition tolerance simultaneously in distributed systems.

Finally, our use of Merkle trees for anti-entropy builds on prior work by Merkle Merkle [1980], which Dynamo later adopted. Efficient state reconciliation using hash trees remains a relevant research area, especially for large-scale data systems and blockchains.

3 System Design

3.1 Consistent Hashing with Virtual Nodes

To partition the key space across nodes, we use consistent hashing Karger et al. [1997], a technique that assigns each node a position on a hash ring. Each key is placed on the ring based on its hash and is stored at the first node with a hash position greater than the key's hash. Replication is achieved by forwarding the key to the $N - 1$ immediate successors.

To improve load balance, we adopt virtual nodes (VNodes). Each physical node owns multiple positions on the ring, ensuring that its failure distributes load across many nodes rather than concentrating it on a single successor. Similarly, when a new node joins, it takes over partitions from several existing nodes, smoothing load distribution. The number of virtual nodes per physical node is configurable.

Each node maintains a preference list of successors for each key range, which supports replication and hinted handoff. Membership changes (e.g., join/leave) are disseminated via gossip, ensuring that the ring structure eventually converges.

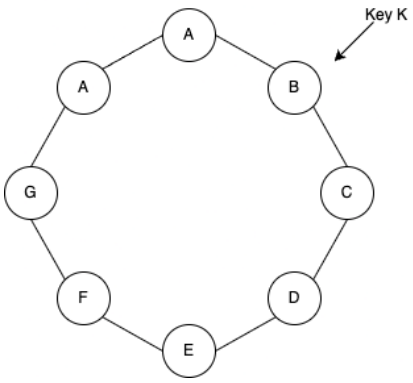

Figure 1: Partitioning and replication of keys

Figure 1 illustrates consistent hashing in a system with 4 physical nodes, each assigned 2 virtual nodes (total of 8 positions on the ring). Each data item, such as Key K, is hashed to a position on the ring and replicated to its next 3 successors (write quorum $W = 3$)

3.2 Sloppy Quorums and Replica Node Selection

We implement sloppy quorums following Dynamo. A write is successful once W healthy nodes out of N replicas acknowledge it, and a read returns upon receiving R responses. If $R + W > N$, the system ensures that at least one replica overlaps between any read and write, increasing the chance of consistent reads. If $R + W \leq N$, stale reads can occur under failure or delay.

The coordinator (usually the primary replica) orchestrates read and write operations. For writes, it sends the updated value with a vector clock to all N replicas and waits for W acknowledgments. Non-responding nodes receive the update later via hinted handoff or anti-entropy. For reads, it collects

R versions and compares vector clocks to detect conflicts, returning all concurrent versions to the client for resolution.

3.3 Data Versioning with Vector Clocks

To track causality and detect concurrent writes, each value is tagged with a vector clock Fidge [1988]. A vector clock maintains a counter per node. When a node updates a key, it increments its own entry and propagates the updated clock.

Upon reads, versions are compared: if one vector clock dominates another, it supersedes the older version. If vector clocks are incomparable, the versions are concurrent (siblings), and the timestamps in both are compared and the latest one is used for resolution. Vector clocks grow with the number of writers. In practice, we observed minimal overhead since our cluster sizes were small and stable.

3.4 Gossip Protocol for Membership and State Dissemination

Each node runs a periodic gossip protocol to disseminate membership and metadata updates van Renesse et al. [2003]. Every T_{gossip} seconds, a node selects a random peer and exchanges a digest that includes: (a) membership status with timestamps, and (b) version summaries for keys.

During a gossip exchange, discrepancies trigger fetches of missing or outdated data. This ensures eventual convergence despite failures. For failure detection, we use gossip-based heartbeats. If a node fails to gossip within a timeout, it is suspected down. This integrates with hinted handoff logic to route writes during outages.

3.5 Anti-Entropy via Merkle Trees

While gossip disseminates updates, it does not guarantee full convergence. We implement periodic anti-entropy using Merkle trees Merkle [1980]. Each node maintains a Merkle tree for each key-range it is responsible for. The tree's leaves are hashes of key-value pairs; internal nodes hash their children recursively.

Replica pairs periodically exchange Merkle roots. If the roots differ, the trees are traversed to locate mismatching subtrees, and only the differing keys are exchanged. For performance, we sometimes use a simplified scheme that hashes small key groups or directly compares key lists when ranges are small. Anti-entropy ensures that replicas eventually reconcile all missed updates, even in the presence of extended failures.

3.6 Hinted Handoff for Temporary Failures

Hinted handoff handles transient failures gracefully. If a write's target replica is unreachable, the coordinator buffers a hint: a record that replica X is supposed to store value V for key K. The write succeeds if W acknowledgments are received (including the coordinator itself).

Hints are delivered once the target node is marked alive (via gossip or anti-entropy). Our implementation integrates hint delivery into both gossip exchanges and background threads. This ensures high write availability even during node failures, without sacrificing durability.

4 Implementation Details

Our prototype is implemented in **Elixir**, a functional programming language that runs on the Erlang VM. Elixir offers lightweight concurrency and efficient message-passing semantics, making it well-suited for building distributed systems.

Emulation Framework. Our system leverages emulation framework from the distributed systems course written in Elixir using the Erlang VM's lightweight processes and message-passing model to simulate distributed systems. The core module `ComBase` manages process registration, message delivery, and time tracking. It supports simulated network behaviors such as message delays, drops, and reordering through a configurable fuzzing pipeline in the `Fuzzers` module. The `Emulation`

wrapper module simplifies context management and exposes a high-level API for spawning processes, broadcasting messages, and scheduling timers. This framework enables deterministic and reproducible evaluation of our system's behavior under various consistency and failure scenarios.

Gossip protocols and quorum-based `GET`/`PUT` operations are implemented using asynchronous message passing. For consistent hashing, we use the SHA-1 hash function to map both nodes and keys onto a circular identifier space, enabling balanced key distribution and alignment with Dynamo's original design. We add a `vnodes` configuration parameter to specify the number of virtual nodes per physical node. Each value is versioned using vector clocks, implemented as maps from node identifiers to integer counters. Writes increment the local counter and merge clocks to capture causality and resolve conflicts.

The system supports configurable replication and tunable consistency through quorum parameters: replication factor N, read quorum R, and write quorum W. Operations are routed to coordinator nodes, which communicate with a preference list of replicas to fulfill requests. A test harness exercises the system by issuing read/write operations under varying quorum values and simulated network conditions. Failure scenarios, such as message loss, node crashes, and partitions, are injected via fuzzers to evaluate robustness. This framework enables detailed analysis of eventual consistency, anti-entropy convergence, and stale read behaviors, as discussed in the subsequent sections.

5 Experiments

We evaluated our Dynamo-inspired system on a cluster of multiple logical storage nodes, each configured to replicate keys with a configurable replication factor of $N = 4$. We focused on three primary experiments:

1. **Effectiveness of Gossip and Anti-Entropy in Reducing Inconsistencies**
2. **Stale Reads under Different R/W Quorum Settings**
3. **Impact of Synchronization Frequency on Consistency**

Each experiment ran a workload of 100 interleaved `GET` and `PUT` operations over a small keyspace, with configurable drop of 0.05. Node failure were simulated by marking nodes as failed in the local map. A stale read is defined as one that does not return the latest value written prior to the read in the operation history.

5.1 Experiment 1: Effectiveness of Gossip and Anti-Entropy Mechanisms

This experiment compared two configurations:

- **No Repair Protocols**: Gossip, anti-entropy, and hinted handoff are disabled. Message drops simulate partial write failures.
- **With Repair**: Gossip, anti-entropy, and hinted handoff are enabled. Node failures and message drops are injected via fuzzers.

We ran the same workload under both configurations while varying message drop rates (1%, 5%, and 10%).

5.2 Experiment 2: Stale Reads vs. R/W Quorum Settings

To isolate the effect of quorum settings on consistency, we tested four (R, W) configurations:

- R=1, W=4
- R=2, W=3
- R=3, W=4
- R=4, W=1

Gossip and anti-entropy were disabled to avoid background repair. After each write, a read was issued with a short delay to simulate overlapping behavior. Each setting ran for 100 write-read cycles on a single key.

5.3 Experiment 3: Effect of Gossip/Anti-Entropy Sync Interval

We varied the synchronization interval T_{sync} used for both gossip and anti-entropy processes, to assess its effect on consistency. The tested intervals were:

- 20 ms
- 50 ms
- 100 ms
- 100,000 ms (effectively no sync)

We fixed quorum at R=1, W=3 to maximize the chance of observing stale reads. We introduced message drops and delays to simulate missed updates.

6 Results and Analysis

6.1 Experiment 1: Repair Protocols Versus Message Drops

Setup. We fixed the replication factor at $N = 4$ and compared two quorum choices— (R=2, W=3) and (R=1, W=4)—under three independent message-drop probabilities (1 %, 5 %, 10 %). Each configuration was exercised for 100 interleaved `PUT`/`GET` operations on a hot key. We ran the workload twice: once with *all background repair disabled* (gossip, anti-entropy, hinted handoff) and once with *all repair mechanisms enabled*. Stale reads were counted when the value returned by a read did not match the most recent completed write in the global history.

Findings. Figure 2 shows that disabling repair makes the system highly sensitive to even minimal packet loss. At 1 % drops, stale-read rates already reach **5 %** for $(2, 3)$ and **7 %** for $(1, 4)$; at 10 % drops these rates climb to **10 %** and **14 %**, respectively. Enabling gossip, anti-entropy, and hinted handoff slashes those numbers to **1.7 %** and **3.8 %** at 1 % drops and keeps them below **7 %** even at 10 % drops in the worst case.

Interpretation. Two effects stand out. *(i)* Overlapping quorums matter: $(2, 3)$ never exceeds the staleness of $(1, 4)$ in any run because every read in $(2, 3)$ contacts at least one replica that acknowledged the write. *(ii)* Continuous repair is more valuable than apparent node stability: a deployment that experiences failures but heals itself outperforms an identical cluster with zero failures but no healing. These results echo Dynamo's original claim that "always-on" anti-entropy is a first-class design knob for eventual consistency systems.

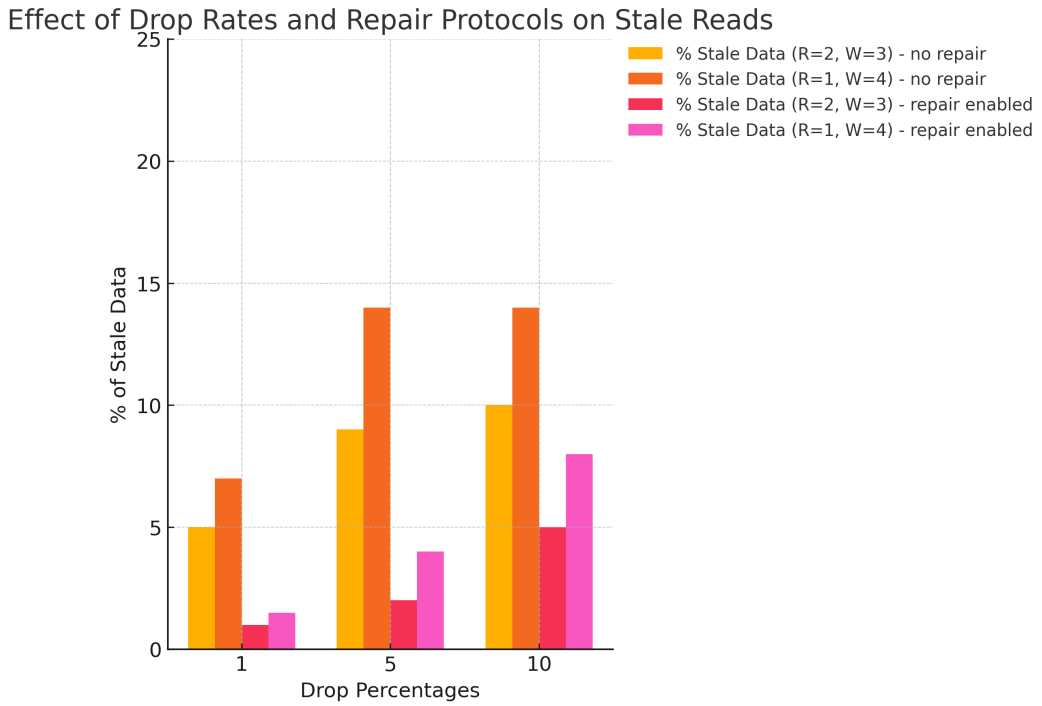

Figure 2: Impact of message-drop rate and background repair on stale reads (N=4). For each drop percentage (1 %, 5 %, 10 %) the four bars show, from left to right: *(i)* R=2, W=3 with repair *disabled*, *(ii)* R=1, W=4 with repair *disabled*, *(iii)* R=2, W=3 with gossip/anti-entropy *enabled*, and *(iv)* R=1, W=4 with repair *enabled*. Enabling gossip, anti-entropy, and hinted handoff slashes stale-read rates to under 2 % at 1 % drops and to below 7 % even at 10 % drops.

6.2 Experiment 2: Quorum Strength and Staleness

Setup. To isolate quorum effects we disabled all repair traffic and executed 100 write–read pairs on a single key while varying (R, W) over the four combinations that sum to $N{=}4$: {(1,4), (2,3), (3,4), (4,1)}. A short delay (uniform 0–5 ms) between each write and its subsequent read creates the window for observing stale replicas.

Findings. Figure 3 reveals a near-monotone decline in staleness as the read quorum grows: $R{=}1$ incurs **15 %** stale reads, $R{=}2$ only **5 %**, $R{=}3$ a residual **3 %**, and a full read quorum ($R{=}4$) eliminates inconsistency entirely. Notice that $W{=}4$ alone is not a silver bullet: with $(R{=}1, W{=}4)$ the write commits to *all* replicas, yet a read issued before the final write ACK can still hit a lagging replica and return the previous version.

Interpretation. The classical rule $R + W > N$ guarantees at least one overlapping replica between a read and a write, but our results show the guarantee is *probabilistic* when operations overlap in time. Increasing R shrinks the vulnerability window linearly because each additional replica polled reduces the chance that *all* R are stale. Conversely, pushing W above $N/2$ yields diminishing returns once the write is already durable on a majority quorum.

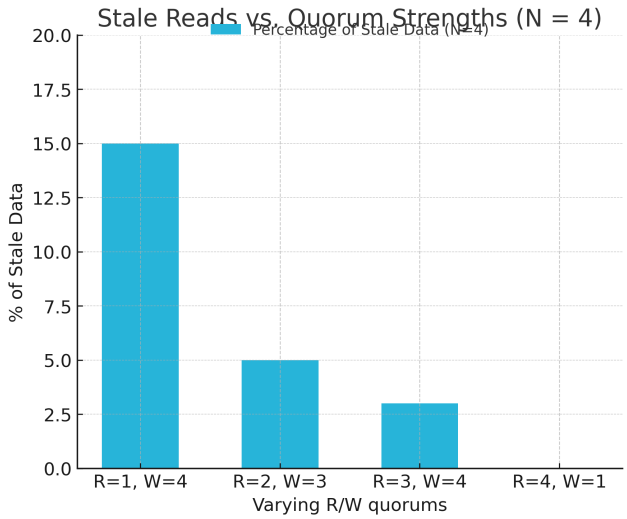

Figure 3: Stale reads as a function of R/W quorum configuration ($N = 4$, no background repair). With a full read quorum (R=4, W=1) no stale reads occur. Reducing the read quorum increases the window for reading a replica that has not yet received the most recent write: R=3, W=4 yields 3 % staleness, R=2, W=3 5 %, and R=1, W=4 roughly 15 %. The results illustrate that while $R + W > N$ is necessary for strong consistency, it is not sufficient once writes and reads overlap in time.

6.3 Experiment 3: Synchronisation Frequency

Setup. We fixed the quorum at $(R{=}1, W{=}3)$—a deliberately weak read setting that amplifies staleness—and varied the common sync interval T_{sync} used by both gossip and anti-entropy. Four settings were tested: 20 ms, 50 ms, 100 ms, and "No sync" (anti-entropy disabled, modelled as $T_{\text{sync}} = 100\,000$ ms). Each run reused the same 100-operation workload with 5 % message drops.

Findings. As plotted in Figure 4, staleness tracks the sync period almost linearly for realistic values: **4 %** at 20 ms, **5 %** at 50 ms, and **6 %** at 100 ms. Disabling periodic repair altogether balloons staleness to **12 %**. The marginal improvement between 50 ms and 20 ms (1 pp) is small compared to the jump between 100 ms and 50 ms (2 pp), indicating diminishing returns as the interval approaches typical RTTs.

Interpretation. Anti-entropy acts as a moving "broom" that sweeps away divergent versions; the shorter the sweep interval, the shorter the window in which a client can read stale data. For latency-sensitive applications the results suggest two tuning levers: either raise R slightly (cf. Experiment 2) or reduce T_{sync}— both deliver similar consistency improvements, but their cost profile differs (added read latency vs. background bandwidth). Choosing the cheaper lever depends on workload mix and network budget.

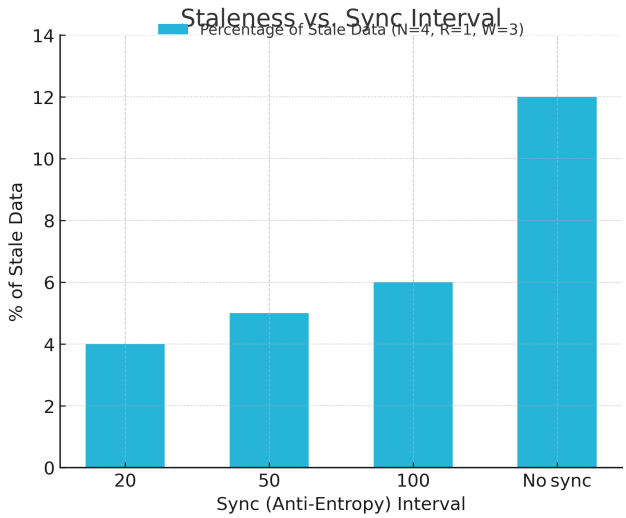

Figure 4: Stale reads as a function of R/W quorum configuration ($N = 4$, no background repair). With a full read quorum (R=4, W=1) no stale reads occur. Reducing the read quorum increases the window for reading a replica that has not yet received the most recent write: R=3, W=4 yields 3 % staleness, R=2, W=3 5 %, and R=1, W=4 roughly 15 %. The results illustrate that while $R + W > N$ is necessary for strong consistency, it is not sufficient once writes and reads overlap in time.

7 Limitations

While our system and experiments provide meaningful insights into quorum behavior and failure-recovery mechanisms, several limitations exist in our current implementation:

7.1 Limitations

- **Single-machine simulation:** All nodes were simulated as concurrent processes on a single machine. This limits the realism of network latency, bandwidth variation, and independent node failures.
- **Simplified failure model:** Our failure injection only simulated message drops and measured node failure to simulate failure scenarios.
- **No real-time latency measurements:** We focused primarily on stale read percentages and did not record end-to-end latency or throughput under load.
- **Basic anti-entropy logic:** Our reconciliation compares entire key-version maps linearly without optimization. Real-world systems often use Merkle trees to minimize synchronization overhead.

8 Conclusion

In this project, we designed and implemented a Dynamo-inspired distributed key-value store to study the trade-offs between consistency, availability, and failure recovery. Our focus was on evaluating how quorum settings and recovery protocols such as anti-entropy, gossip, and hinted handoff—affect the rate of stale reads under both ideal and failure conditions.

Through systematic experiments, we observed that background synchronization protocols significantly improve consistency, often outperforming stable but static systems. Notably, our results show that a system with failures but equipped with recovery mechanisms can achieve better consistency than a stable system without them.

These findings reinforce the importance of designing systems that not only tolerate failures but actively recover from them through decentralized protocols. Our work highlights that eventual

consistency, when paired with effective recovery, can yield practical consistency even in imperfect environments.

Future work includes extending our implementation to run in truly distributed settings, optimizing anti-entropy with Merkle trees, and exploring stronger or hybrid consistency models.

References

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

Colin Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, 1988.

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In *ACM SIGACT News*, volume 33, pages 51–59, 2002.

David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663, 1997.

Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. In *ACM SIGOPS Operating Systems Review*, volume 44, pages 35–40, 2010.

Ralph C. Merkle. Protocols for public key cryptosystems. *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 172–183, 1995.

Robbert van Renesse, Kenneth Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. In *ACM Transactions on Computer Systems*, pages 103–145, 2003.

Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.