République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي والبحث العلمي

University of Abou Bekr Belkaid
Faculty of Science
Department of Computer Science

Thesis for the Master's Degree in Computer Science

Specialization: Networks & Distributed Systems

Designing a Distributed Network-Aware Key-Value Store

Author

Aymen BENYOUB

Supervisors

Mr. Paolo CASARI (UNITN)
Mr. Badr BENMAMMAR (UABT)

Defended on June 24th, 2026 before the following jury:

President: Mr. Abdelkrim BENAMAR
Examiner: Mr. Youcef BENMOUNA

Academic Year 2025–2026

Contents

List of Figures

List of Tables

Acronyms

KV Key-Value

AZ Availability Zone

HRW Highest Random Weight

vnode Virtual Node

IO Input/Output

RTT Round-Trip Time

RF Replication Factor

RPC Remote Procedure Call

gRPC Google Remote Procedure Call

API Application Programming Interface

FSM Finite State Machine

CAP Consistency, Availability, Partition Tolerance

EMA Exponential Moving Average

NIC Network Interface Card

NUMA Non-Uniform Memory Access

NVMe Non-Volatile Memory Express

CPU Central Processing Unit

RAM Random Access Memory

EC2 Elastic Compute Cloud

AWS Amazon Web Services

SQL Structured Query Language

ACK Acknowledgement

TLS Transport Layer Security

CLI Command-Line Interface

TCP Transmission Control Protocol

Abstract

Traditional distributed key-value stores use consistent hashing with virtual nodes (vnodes) but treat the network as a static black box. This leads to data skew and ignores real-time latency and resource variations. This thesis presents a network-aware distributed key-value store that dynamically adapts data placement. By decoupling control and data planes, the system monitors hardware metrics and RTTs to compute a deterministic cost function for vnode reallocation. Results show that the proposed topology eliminates hotspots and improves load balancing without degrading core storage performance.

Keywords: Distributed Key-Value Store; Network Awareness; Dynamic Data Placement; Virtual Nodes; Consistent Hashing

ملخص

تعتمد قواعد بيانات قيم المفاتيح الموزعة التقليدية على التجزئة المتسقة مع العقد الافتراضية، لكنها تتعامل مع الشبكة كصندوق أسود ثابت. يؤدي ذلك إلى انحراف البيانات وتجاهل تغيرات زمن الانتقال وحدود الموارد. تقدم هذه الأطروحة نظامًا موزعًا واعيًا بالشبكة يُكيّف موضع البيانات ديناميكيًا. من خلال فصل طبقة التحكم عن البيانات، يتابع النظام مقاييس العتاد وزمن الرحلة (RTT) لحساب دالة تكلفة حتمية لإعادة توزيع العقد الافتراضية. تُظهر النتائج أن هذه الطوبولوجيا تقضي على النقاط الساخنة وتوازن الحمل دون التأثير على أداء التخزين الأساسي.

الكلمات المفتاحية: قاعدة بيانات موزعة لقيم المفاتيح؛ الوعي بالشبكة؛ توزيع ديناميكي للبيانات؛ العقد الافتراضية؛ التجزئة المتسقة

Résumé

Les bases de données clés-valeurs distribuées traditionnelles s'appuient sur le hachage cohérent avec des nœuds virtuels, mais traitent le réseau comme une boîte noire statique. Cela cause des déséquilibres et ignore les variations de latence. Cette thèse propose un système clé-valeur distribué sensible au réseau qui adapte dynamiquement le placement des données. En séparant les plans de contrôle et de données, le système utilise métriques matérielles et RTTs pour calculer un score de coût déterministe. L'évaluation montre que cette approche élimine les points chauds et équilibre la charge sans dégrader les performances de stockage.

Mots-clés: Base de données clé-valeur distribuée; Sensibilité au réseau; Placement dynamique; Nœuds virtuels; Hachage cohérent

General Introduction

Context

Modern distributed databases replicate and partition data to ensure scalability and fault tolerance. A popular approach is using consistent hashing with virtual nodes to distribute data across a cluster, but treats the underlying physical network and node resources as a uniform, static substrate. While this abstraction simplifies design, it introduces practical limitations: data distribution becomes skewed when nodes differ in capacity, replica placement ignores the actual latency between nodes, and the system remains blind to real-time resource pressure until a node fails outright.

Problem Statement

Traditional placement algorithms assign virtual nodes based on static configuration or simple weight parameters that are set once at cluster initialization. As cluster conditions evolve, nodes become loaded, network links degrade, or new nodes join with different hardware profiles, the placement remains fixed. This static view leads to data hotspots, suboptimal replica locations, and degraded performance that persists until a human operator intervenes. There is no mechanism by which the system can autonomously detect that a given node is a poor placement choice and rebalance accordingly, without disrupting the primary data path.

Contribution

The goal of this research is to design and prototype a distributed key-value store that incorporates network-awareness and other load factors into its primary and replica placement decisions, handle data redistribution caused by different triggers, and evaluate its performance under multiple workloads and node failures, comparing it against traditional consistent-hashing-based systems. The proposed system continuously collects per-node hardware utilization metrics and inter-node round-trip

times, disseminates them through a gossip layer, and feeds them into a deterministic scoring function that drives vnode allocation. Placement decisions are committed through a Raft consensus layer, ensuring that all nodes converge on a consistent view without independently deriving conflicting assignments. The result is a system that adapts its topology to current cluster conditions while keeping the primary data path free of control-plane overhead.

Thesis Structure

- Chapter 1: An overview of foundational distributed storage concepts & Industry context.
- Chapter 2: Our system design & Implementation.
- Chapter 3: Evaluation & Analysis.
- Chapter 4: Discussion & Critical Reflection.

Chapter 1

Distributed Storage Overview

1.1 Introduction

Distributed storage is a decentralized architecture that spreads data across a cluster of independent nodes or geographically dispersed availability zones (AZs). By partitioning the data into granular units, the system achieves horizontal scalability, fault tolerance, and high availability. Unlike centralized storage, which presents a single point of failure, distributed systems utilize replication and consistency protocols to ensure data durability even in the event of partial node or regional failure.

To maintain performance and reliability at scale, distributed storage systems must orchestrate several interlocking mechanisms. While the specific implementation of these concepts varies across industry-standard systems, such as DynamoDB, Cassandra, and Etcd; they all share a common conceptual foundation. The following sections explore these core pillars.

1.2 Data Partitioning

Data partitioning involves dividing a large dataset into smaller, independent segments (called partitions or shards), allowing queries to be executed on those partitions, instead of full dataset scans, resulting in much better performance, and allows maintenance to be done on those segments. It can be done in two ways:

Vertical Partitioning: Divides the dataset into disjoint (or partially overlapping) subsets based on columns or attributes

Horizantal Partitioning: Divides the dataset into disjoint subsets based on records, determined by a partitioning function (e.g., hash, range)

For distributed storage, the method used is horizontal partitioning, called 'sharding' in this context, which involves putting those shards on different nodes in the cluster.

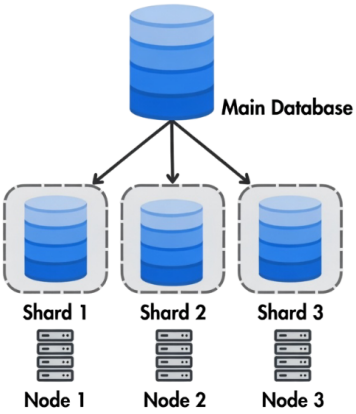


Figure 1.1: Sharding visualized [1].

As stated previously, the data contained in each shard can be determined in a number of ways, the two most popular being range-based sharding and hash-based sharding. (for KV systems and all storage types in general, the former being our focus).

1.2.1 Range-based Sharding

Range sharding works by splitting the data using key ranges (e.g., keys 0 to 999 go to shard-1, keys 1000 to 1999 go to shard-2 etc...), this allows for predictable partitioning, making it easy to know which shard might host a specific key, as well as much better performance for range queries, because they can be directed to a single or a subset of the shards [2].

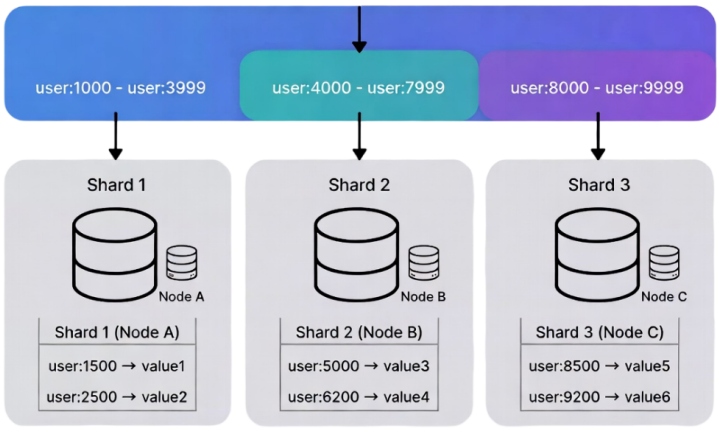


Figure 1.2: Sharding using key ranges

However, due to the contiguous and sorted nature of this technique, it can lead to 'hot shards', where a node gets overloaded with traffic due to frequent access to its key range, and the likelihood of receiving most if not all write traffic, in case it's holding the tail end of the key range, this forces the developer to attempt to manually fix this at the application level, by prepending a random hash to each new key for example, which in turn sacrifices the natural ordering of the keys, and lose out on its benefits.

This technique is used in Google's Spanner (SQL) and BigTable, both split their tables into chunks called splits and tablets respectively, and have a background process that handles dynamic rebalancing. Once a split or tablet reaches a certain size, or gets overloaded with traffic, the system splits it in two and moves one half to a different node in the cluster [3] [4].

1.2.2 Hash-based Sharding

Hash sharding works by passing the key to a cryptographic or non-cryptographic hash function, then using the output to map the key to one of the nodes, this has the advantage of uniform scaling, because the hash function scrambles the data, it prevents asymmetrical load growth as the dataset gets larger. and it also prevents any single shard from being overloaded with queries, but it lacks the range scan efficiency that range sharding has.

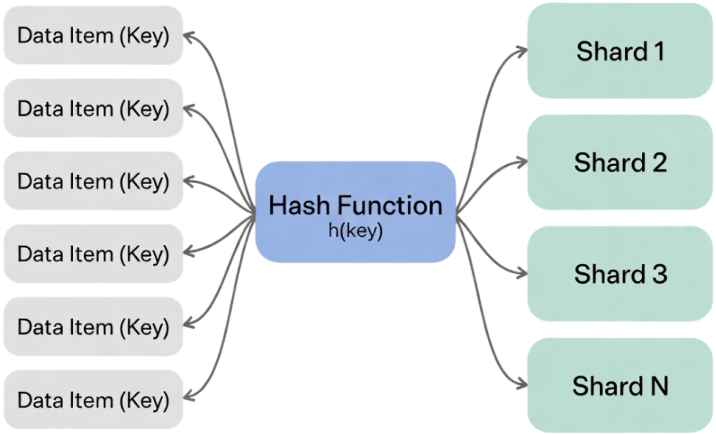


Figure 1.3: Sharding using hash functions

The key-to-node mapping can be done in a number of ways, the simplest one is to use a modulo division, and determining the shard based on the result:

$$shard = hash(K) \mod N$$

Where N is the number of nodes in the cluster.

While this method results in a fast mapping, it has the major drawback how of full remapping of all keys anytime a node joins or leave, causing massive data movement. Another method, called 'Rendezvous Hashing' [5], also known as Highest Random Weight (HRW hashing), for any given key, it calculates multiple scores by hashing a combination of the key and node identifiers (via string concatenation or byte interleaving), and assigning the key to the node with the highest score:

$$score_i = hash(\ K \ || \ node_i)$$

While this method is better for dynamic environments, its $O(N)$ complexity becomes a problem for very large clusters, though several optimized versions of it exist [6] [7].

Other simple methods include assigning based hash ranges, for example, we use a hash space of 2^{64} values, and split it into ranges, each belonging to a single node. This method works well for static clusters and gives reasonably even data distribution. However, it struggles in dynamic environments. Adding or removing a node requires reshuffling range boundaries, which typically causes massive data movement across the cluster.

Consistent Hashing was specifically designed to solve this problem.

Consistent Hashing & Virtual Nodes

Instead of dividing the hash space into fixed contiguous ranges, consistent hashing treats the hash space as a circle (a ring). Both keys and nodes are placed on this ring using the same hash function. Each key is assigned to the first node that appears clockwise from its position on the ring. When a node is added or removed, it only affects the keys in its immediate neighborhood on the ring — dramatically reducing the amount of data that needs to be moved [8].

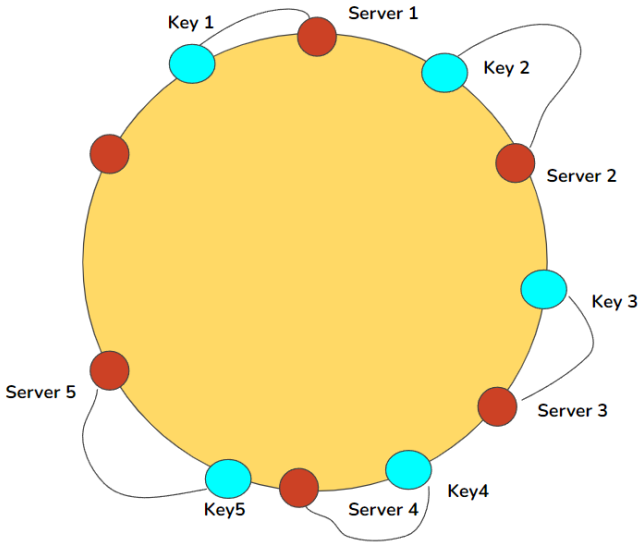


Figure 1.4: Consistent hashing ring topology [9].

The problem with this approach is that it doesn't consider node heterogeneity, because each node gets a single random position on the ring, nodes with smaller capacity maybe be given bigger portions of it, and vice versa. To solve this, the solution is to use 'virtual nodes' (or vnodes), which are multiple positions on the ring, all belonging to a single node, and to give different vnode counts to each node, proportional to their compute power, resulting in powerful nodes taking on more load in the system.

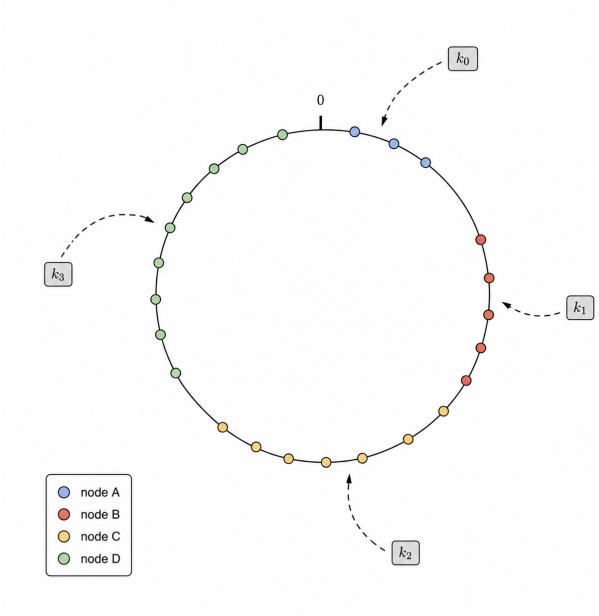


Figure 1.5: Consistent hashing with virtual nodes.

This approach is used in multiple industry systems, such as Amazon's Dynamo [10], Apache's Cassandra [11] (originally developed by Facebook) and Riak KV, each of these assigns multiple points (or tokens) on the ring for each node, Riak also pre-divides the hash space into fixed partitions, and assigns each of them to a single vnode.

1.3 Replication & Consistency

In any distributed data store, data must be replicated to other nodes after write operations, to ensure that it's durable (fault-tolerance), in the case of node crashes or data losses, and to scale the throughput of the system (read performance), as a single machine has limited I/O bandwidth; replicating the data allows the system to handle more read requests concurrently, however, because network propagation takes time, having two or more copies of a single piece of data in different locations, has the risk of divergence, which introduces two fundamental types of replication updates: **synchronous** and **asynchronous** replication.

1.3.1 Synchronous Replication

In this type of replication, the primary node (the one receiving the write, and responsible for the data being written) blocks the client's request until all replicas confirm that the update has been written to disk/memory, this guarantees perfect consistency, because all nodes hold the same data, and grants immediate failover to the replicas, allowing them to take over traffic without having to catch up with the primary. But, it introduces increased latency, as writes speeds are bound by the slowest network link or disk, and the primary must wait on all replicas to confirm the write, which also results in decreased throughput, because each write takes more time and resources[12].

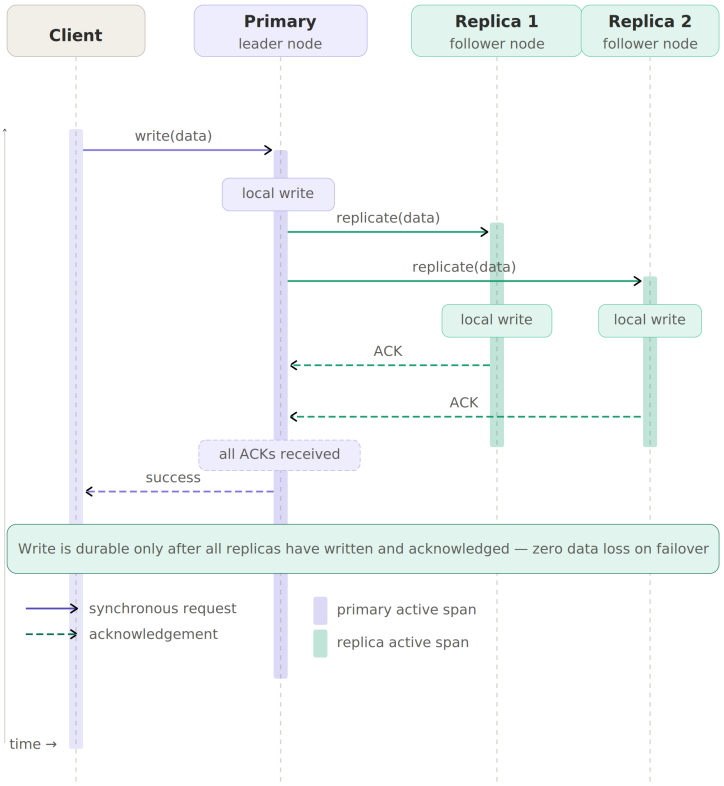


Figure 1.6: Synchronous replication flow

1.3.2 Asynchronous Replication

In this type of replication, the primary node acknowledges the client's write request immediately after writing to its local storage, without waiting for replicas to confirm the update, this allows for much lower latency and higher throughput, as the primary can process more writes concurrently, but it introduces the risk of data loss in case of a primary failure before the replicas have received the update, and it also results in eventual consistency, as replicas may temporarily hold stale data until they receive the latest updates from the primary[12].

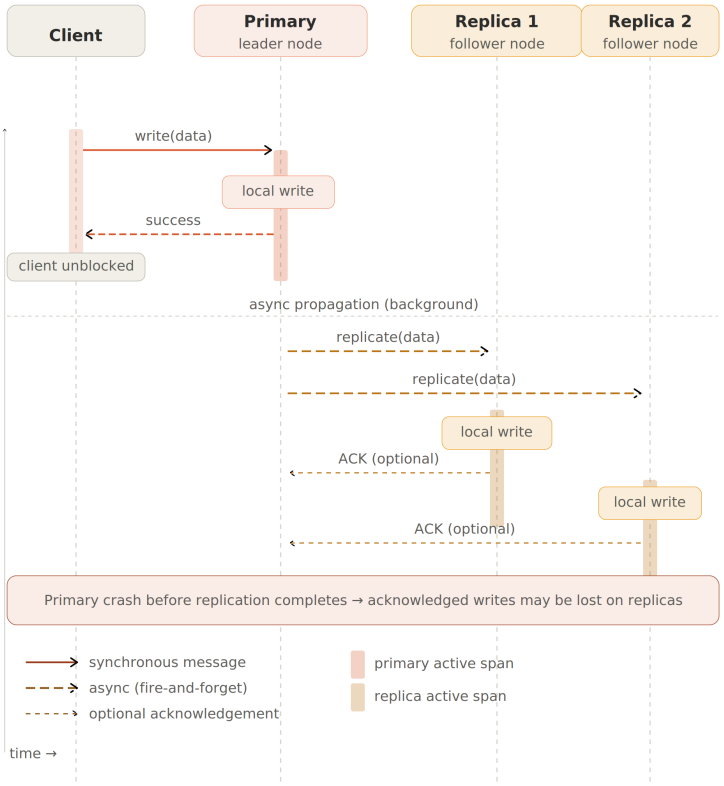


Figure 1.7: Asynchronous replication flow

Some systems that opt for this replication method, such as Dynamo, Cassandra and Riak, implement a quorum-based approach to mitigate the consistency issues, where the client can specify the number of replicas that must acknowledge a read or write operation before it's considered successful, allowing for a tunable balance between consistency and availability [10] [11] [13].

1.3.3 CAP Theorem & System Guarantees

The CAP theorem states that in the presence of a network partition, a distributed system can only guarantee two out of the following three properties: Consistency, Availability, and Partition Tolerance. But, in practice, all distributed systems must be designed to handle partitions, as they are inevitable in real-world networks, so the real trade-off is between consistency and availability. Systems that prioritize consistency (CP) will reject requests if consistency cannot be acheived (meaning the system uses synchronous replication), while systems that prioritize availability (AP) will accept all requests, even if they may return stale data (meaning the system uses synchronous replication)[14].

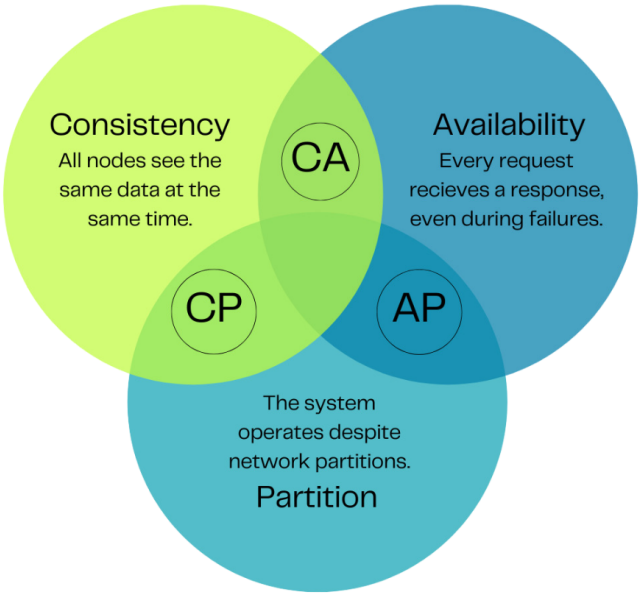


Figure 1.8: CAP theorem trade-offs [15].

1.4 Consensus & Membership

In a distributed system, it's crucial to maintain a consistent view of the cluster's state across all nodes, especially when it comes to membership (which nodes are part of the cluster) and leadership (which node is responsible for coordinating certain operations). Consensus algorithms are used to achieve agreement among distributed nodes on a single data value or a sequence of values, even in the presence of failures.

1.4.1 Paxos Algorithm

The Paxos algorithm is a family of protocols for solving consensus in a network of unreliable processors. It works by electing a leader (or proposer) that proposes a value to the other nodes (acceptors), which then vote on whether to accept the proposal. The algorithm ensures that once a value has been chosen, it cannot be changed, and that all nodes eventually agree on the same value, even if some nodes fail or messages are lost. Paxos is known for its robustness and fault tolerance, but it is also complex to implement and understand,which led to the creation of the Raft algorithm [16].

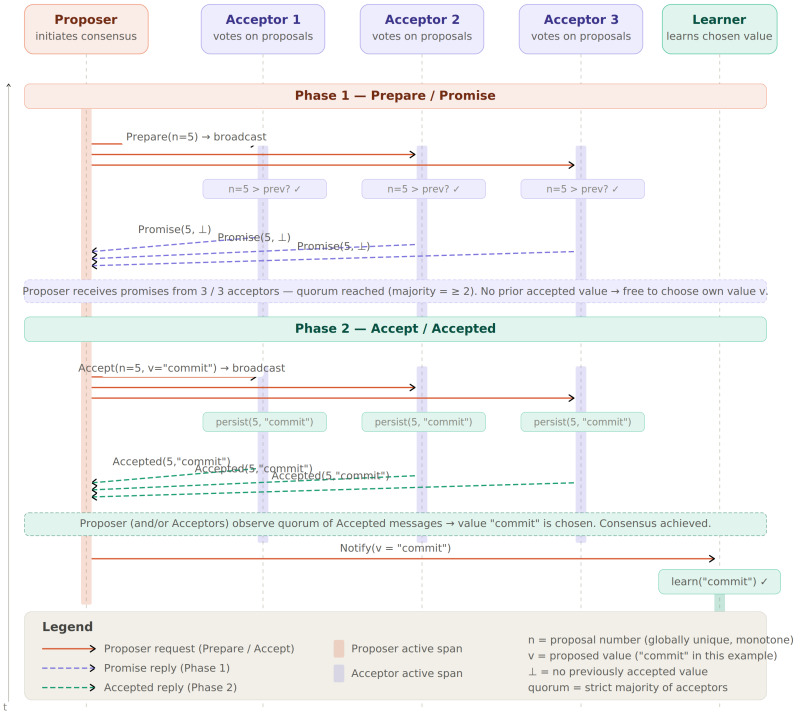


Figure 1.9: Paxos algorithm flow (single decree).

1.4.2 Raft Algorithm

Raft is a consensus algorithm designed to be more understandable than Paxos while providing the same guarantees. It works by electing a leader that manages the replication of log entries to follower nodes. The leader handles all client requests and ensures that log entries are replicated in a consistent order across the cluster. Raft simplifies the consensus process by breaking it down into three main components: leader election, log replication, and safety. It has become popular in distributed systems due to its clarity and ease of implementation[17].

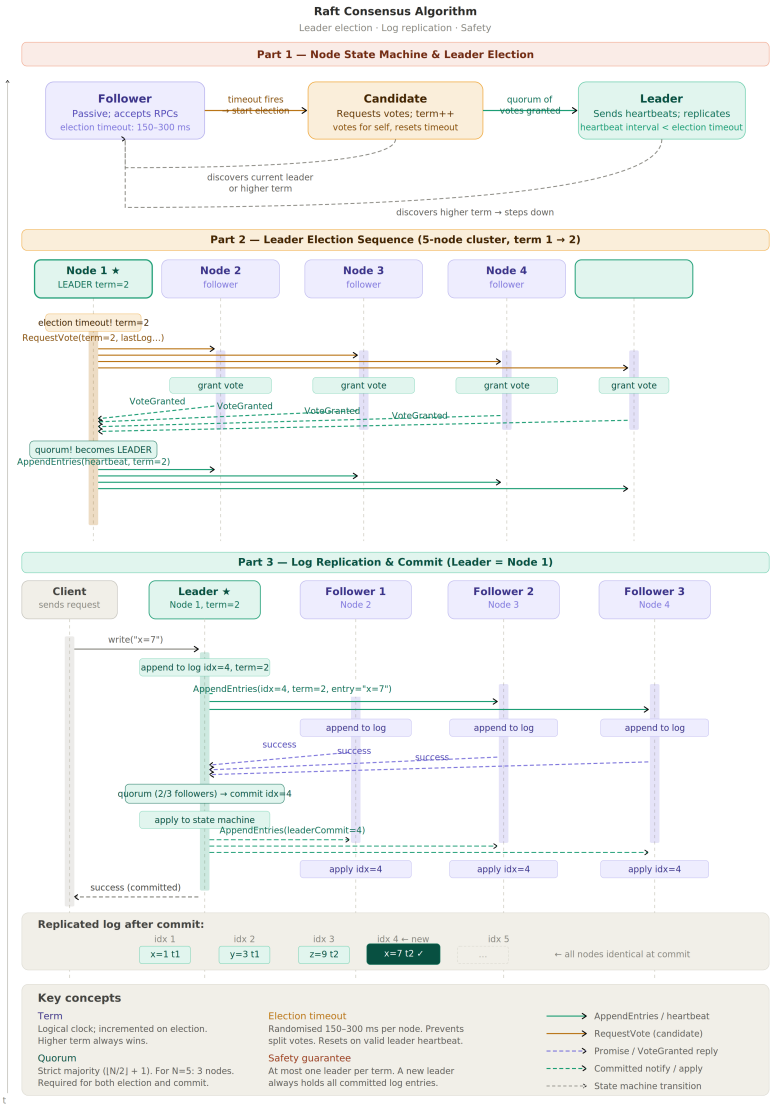


Figure 1.10: Raft algorithm flow

1.4.3 Gossip protocols

Gossip protocols are a class of communication protocols used for disseminating information across a distributed system. They work by having nodes periodically exchange information with a randomly selected subset of other nodes, similar to how gossip spreads in social networks. This approach allows for efficient and scalable dissemination of information, such as membership updates or configuration changes, without the need for a central coordinator. Gossip protocols are often used in distributed databases and peer-to-peer systems to maintain cluster membership and ensure eventual consistency.

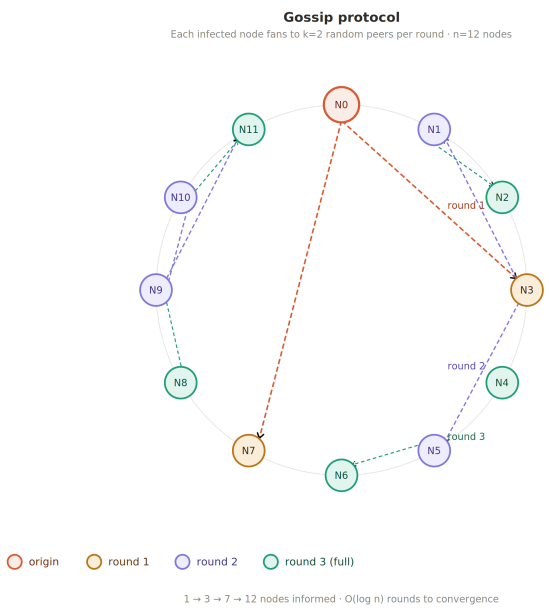


Figure 1.11: Gossip protocol flow

Though gossip and consensus algorithms are different, they are often used together in many large scale distributed systems, such as Cassandra and Riak, where gossip is used for membership and failure detection, while Raft or Paxos are used for leader election and ensuring consistency among replicas [11] [13].

1.5 Conclusion

In this chapter, we provided an overview of the fundamental concepts and mechanisms that underpin distributed storage systems. We explored data partitioning techniques, with a focus on consistent hashing and virtual nodes, which enable scalable and efficient data distribution across a cluster. We also delved into replication strategies, highlighting the trade-offs between synchronous and asynchronous replication, and how they relate to the CAP theorem. Finally, we discussed consensus algorithms like Paxos and Raft, as well as gossip protocols, which are essential for maintaining cluster membership and ensuring consistency in distributed environments. Understanding these core principles is crucial for designing and evaluating distributed storage systems, and we will be exploring how they're used in the design of our own system in the next chapter.

Chapter 2

System Design & Implementation

2.1 Introduction

This chapter describes the design and implementation of our network aware KV store. We will cover the main components, their interactions, and the rationale behind key design decisions. The implementation is done in Go, and the source code is available at [18].

2.2 Tools Used

Software / Tool / Dependency	**Category**	**Role in the Project**
Go	Language / toolchain	Primary implementation language for the KV store, cluster logic, RPC services, and command-line tools.
gRPC	Communication framework	RPC layer used for client requests and internal service-to-service communication.
Protocol Buffers	Serialization format	Defines the service interfaces and structured messages exchanged over gRPC.
HashiCorp Raft	Dependency	Provides replicated log and consensus support for fault-tolerant coordination.
HashiCorp memberlist	Dependency	Implements gossip-based membership discovery and failure detection.
gopsutil	Dependency	Collects system and process statistics for diagnostics and runtime insight.
Docker / Docker Compose	Tooling	Containerization and local multi-node orchestration during development and testing.
tc	Tooling	Linux traffic control utility used to emulate latency, loss, and network impairment during experiments.
iperf3	Tooling	Network performance measurement tool used to saturate bandwidth and measure throughput during experiments.
Bash / tmux	Tooling	Supports local automation and multi-terminal development workflows.
AWS EC2	Cloud platform	Provides virtual machines for distributed deployment and performance testing of the KV store.

Table 2.1: Main software, tools, and dependencies used in the project.

Test Environment

- Local multi-node testing is performed using Docker Compose, which allows us to easily spin up multiple instances of our service on a single machine and

simulate network conditions using ‘tc‘.

- For more realistic distributed testing, we use a small cluster of cloud instances from AWS ... which consists of **7 EC2 c7i.flex.large instances** (2 vCPU, 4 GB RAM) running Ubuntu 22.04 x86, which host the storage nodes of our system, and one being used a client, all located in different AZs within eu-west-3

2.3 Design Goals

The implementation targets the following goals:

- **Scalability:** support horizontal scaling via data partitioning (vNodes) and stateless RPC frontends.
- **Availability:** tolerate node failures using replication and membership/gossip for fast failure detection.
- **Simplicity of reasoning:** keep per-node components small and well-separated (EventLoop, Store, Replicator, Raft placement).
- **Network & Load awareness:** keep track of node-local metrics, and use them to make dynamic placement decisions for primary and replica vNode assignments.

2.4 High-level Architecture

Our system splits responsibilities across a few cooperating subsystems:

- **Core runtime (‘core‘):** Represents the data plane, implements the in-memory store, durability log, batched event loop, and the gRPC server that exposes the KV API.
- **Cluster manager (‘cluster‘):** The control plane of the system, handles cluster membership , leader election, runtime node evaluations, placement decisions, replication clients, and vnode migration.
- **CLI and load generator (‘cmd/cli‘, ‘cmd/loadgen‘)** — clients used for testing and evaluation.

Figure 2.1 shows the logical component layout for a single node and its interaction with peers.

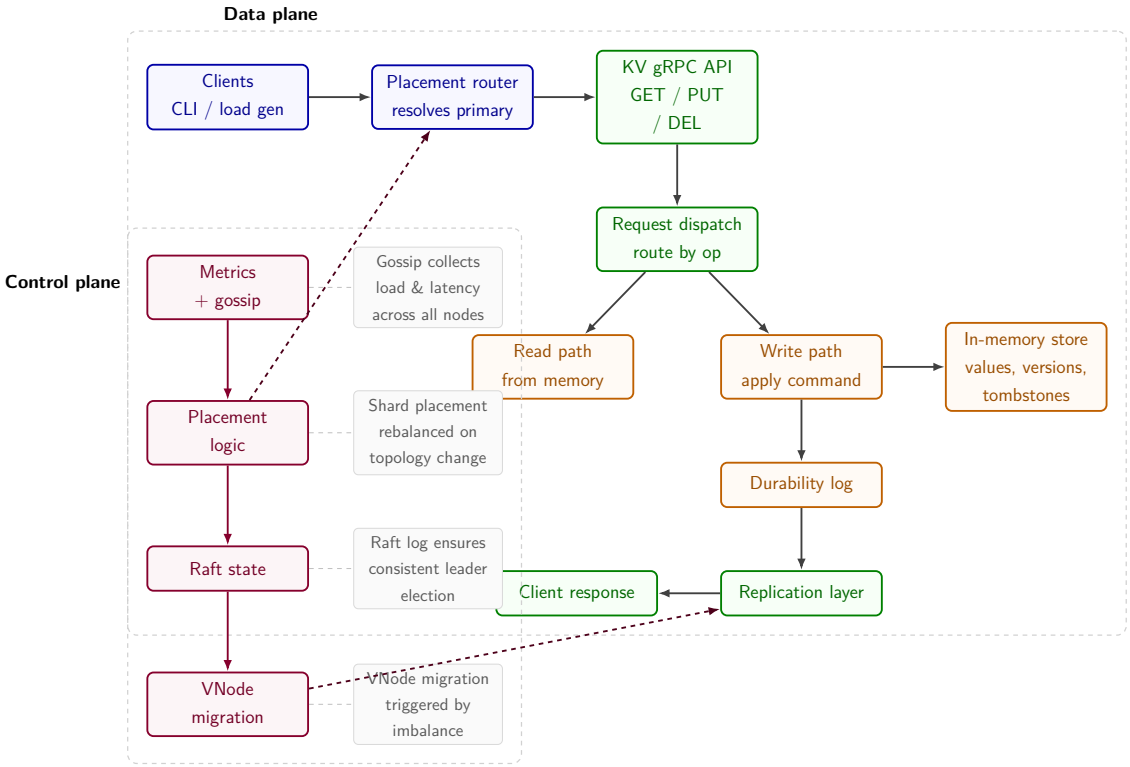


Figure 2.1: System design overview of the network-aware KV store.

2.5 Storage Internals

In this section we dive into the core storage internals (or the data plane components), including the event loop, durability log, store layout, gRPC server semantics.

2.5.1 gRPC Server and RPC semantics

Each node runs a gRPC server that exposes ‘Get‘, ‘Put‘, and ‘Delete‘ RPCs . For write operations, the server verifies local ownership (via ‘Replicator.CheckOwnership‘) and rejects writes to non-owners (for extra safety, as clients are already aware of ownership), then attaches a version to the incoming key, and sends it off to the event loop (read requests also go through the event loop). After a successful local apply, the server asks the ‘Replicator‘ to send replication requests to remote replicas, using the ‘ReplicateStream‘ RPC, which opens a bidirectional stream between the node and the remote nodes it’s targeting, and sends the replication requests in batches.

Figure 2.2 shows a breakdown of the data-plane RPC layer.

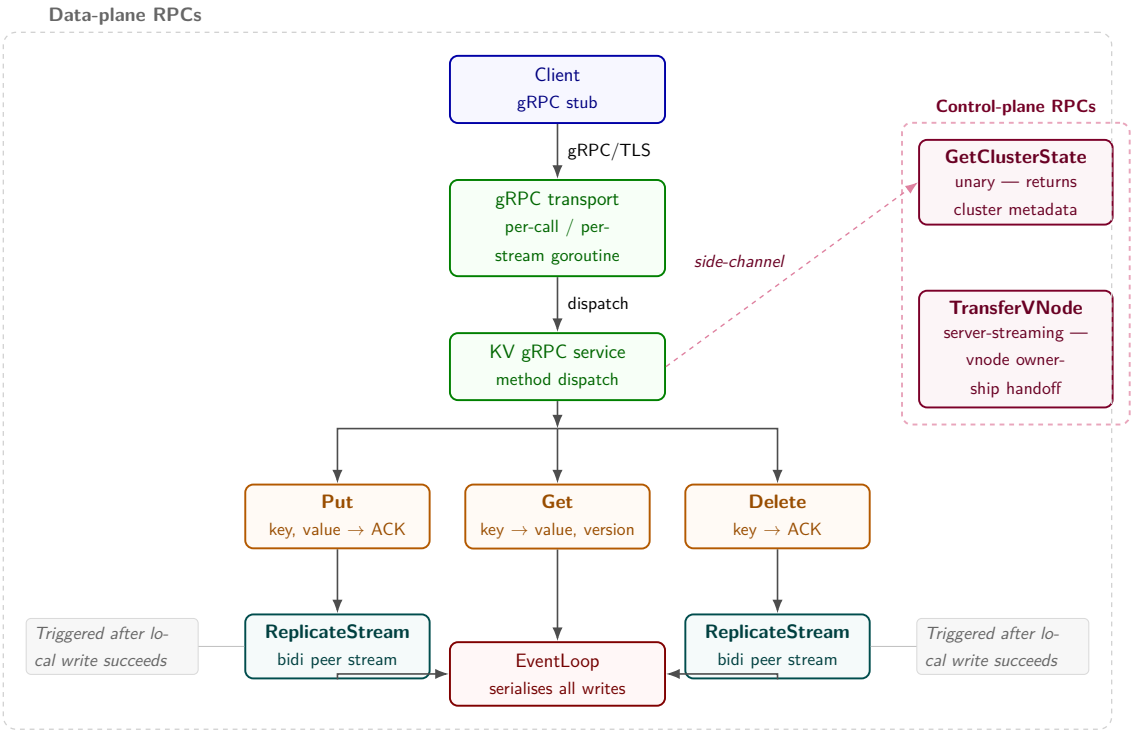


Figure 2.2: gRPC surface and call flow of the KV node.

2.5.2 The EventLoop

An event-loop is a single long-running thread or routine that repeatedly dequeues and dispatches incoming events or commands from a queue, executing them one at a time so handlers that modify shared state run serially. This pattern guarantees a single, well-defined order of state changes and makes reasoning about concurrency simpler, because handlers need not acquire fine-grained locks to coordinate with other handlers running in the same loop[19].

In this system the `EventLoop` is implemented as a single goroutine that serializes all state-changing requests arriving from the gRPC surface. Specifically:

- Non-GET requests are collected into short write batches (grouped by either batch size or a small timeout).
- For each command in a batch, the loop applies the change to the in-memory store and then enqueues the batch to the durability log for asynchronous flushing.
- The loop hands the batch to a dedicated acknowledgment worker that waits for the configured durability boundary before replying to clients. This design keeps the event loop focused on processing new requests.

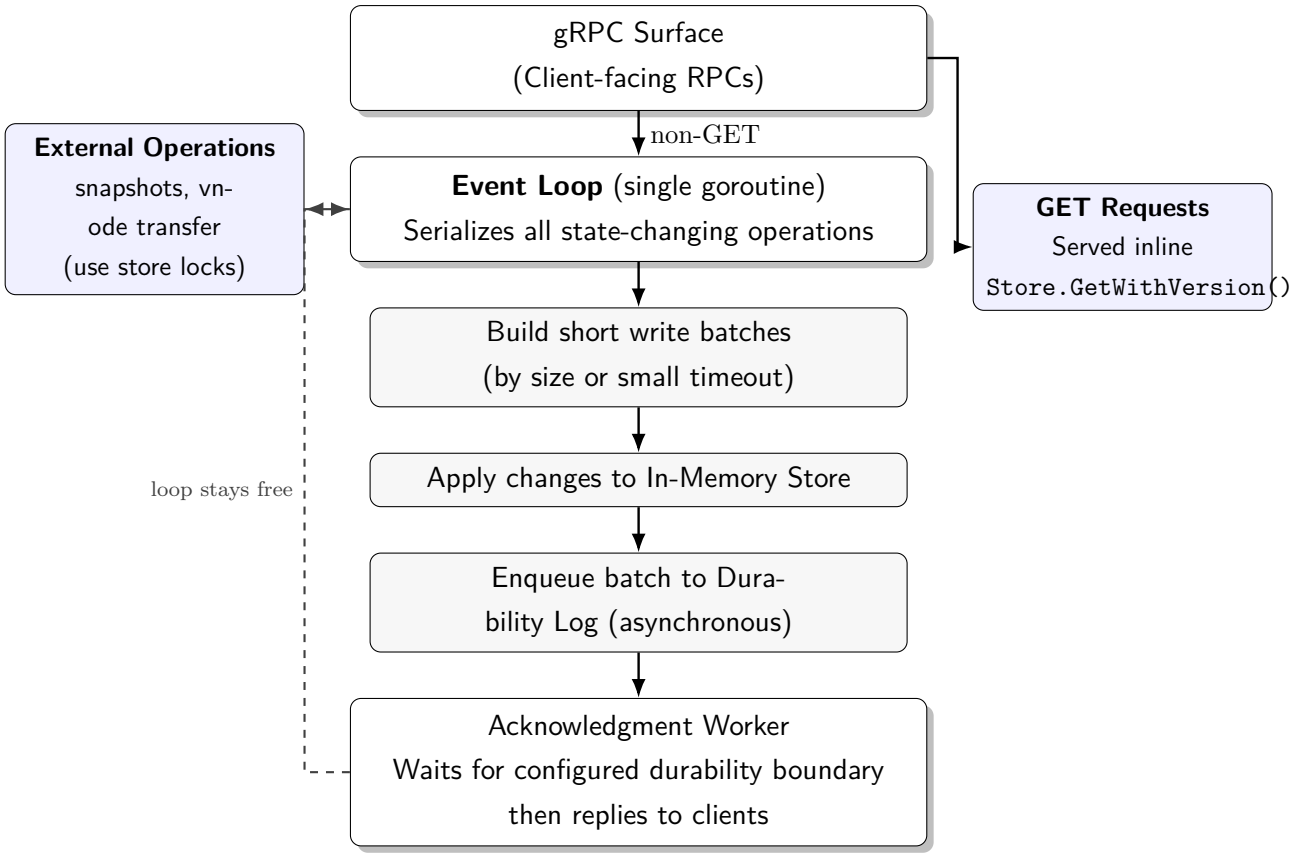
- GET requests are served inline using a direct `Store.GetWithVersion()` call and return immediately.
- Operations invoked outside the event loop (such as snapshots and vnode transfers) continue to use locks on the store.

As a result, the event loop eliminates most concurrency complexity while maintaining clean integration with the system's existing durability and migration subsystems. The pseudocode in Figure 2.3 below illustrates the main flow of the event loop.

```
    while true:
        request ← wait for next gRPC request

        if request is GET:
            value, version ← Store.GetWithVersion(
               request.key)
            reply to client(value, version)

        else:  // Put, Delete, etc.
            add request to current_batch

            if batch is full or timeout reached:
                for each cmd in current_batch:
                    apply cmd to In-Memory Store

                enqueue batch to Durability Log (
                   async)
                hand batch to Acknowledgment Worker

                reset current_batch
```

Figure 2.3: Pseudocode for the Event Loop (single goroutine).

The figure 2.4 below also shows the architecture of the event loop.



The Event Loop eliminates most concurrency complexity while integrating with the durability log and migration subsystems.

Figure 2.4: Event-loop flow.

2.5.3 The durability log

The Durability Log is an append-only background subsystem that guarantees committed write batches survive crashes. It operates as follows:

- Callers use the log's `Append` method to non-blockingly enqueue an encoded batch and immediately receive a monotonically-increasing identifier.
- A dedicated writer goroutine drains the enqueue queue, encodes entries, groups them into I/O batches, writes them to the log file, and issues the platform-specific `flush`/`fsync` calls required by the configured durability policy.
- Each command is encoded inline as a compact per-command record, shown in Figure 2.5. Batches are formed by simple concatenation of these records with no extra framing.

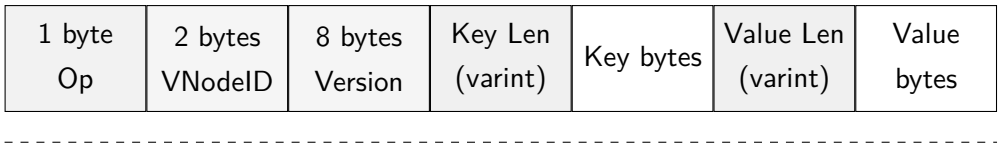


Figure 2.5: On-disk format of a single command record in the durability log.

- The writer continuously tracks the highest durable identifier it has flushed to disk.
- Other parts of the system observe durability by calling the log's `Wait(id)` helper, which blocks until the writer has advanced the durable identifier to be at least `id`.
- In normal operation, the `EventLoop` applies commands to the in-memory store, calls `Append` for the batch, and an Acknowledgment Worker performs `Wait` before responding to clients once the configured durability boundary is reached.
- Three acknowledgment modes are supported:
  - `AckAfterEnqueue` — acknowledges immediately after enqueueing (weakest guarantee),
  - `AckAfterFlush` — acknowledges after the data has been flushed to the OS page cache,
  - `AckAfterFsync` — acknowledges only after a successful `fsync` (strongest guarantee, ensures data reaches persistent storage).

Overall, the log cleanly separates fast in-process enqueueing from the slower on-disk durability work. It uses batching and varint length encoding for I/O efficiency and exposes a simple, crash-surviving durability frontier. This frontier can be waited on by higher layers and is used by the startup replay routine to restore state.

2.5.4 In-memory Store

The Store is the canonical, vnode-aware in-memory key/value layer. It enforces versioned state, tombstone semantics, and efficient per-vnode iteration for migration. It is structured and operates as shown below 2.6.

- Values are stored in a `map[string][]byte`, while authoritative per-key metadata (logical Version and Tombstone flag) resides in a separate meta structure.
- Ownership indices are maintained as `vNodeIdx` (vnode → set of live keys) and `tombstoneIdx` (vnode → set of delete markers). Lightweight counters

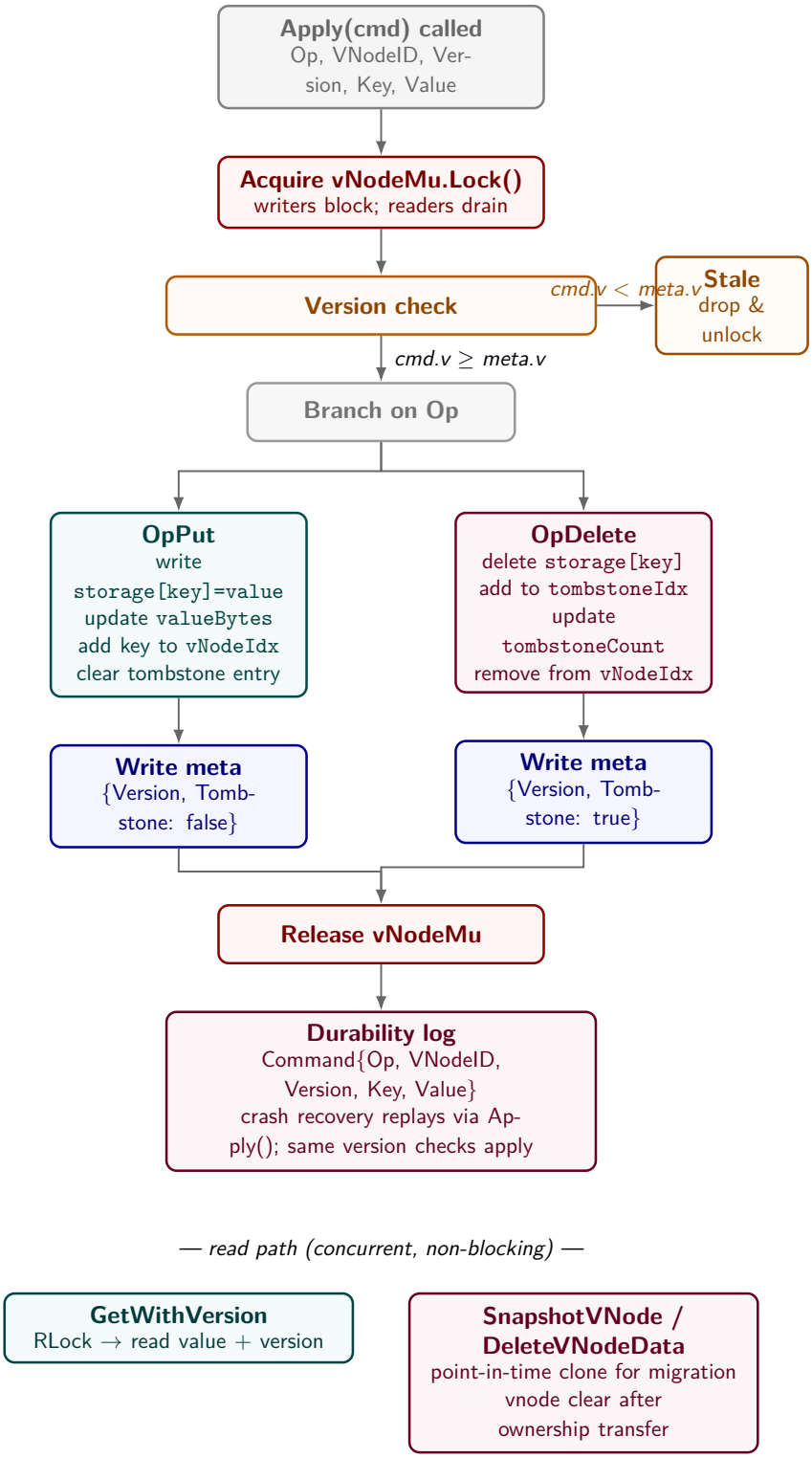


Figure 2.6: Lifecycle of a command through the Store

(`valueBytes`, `tombstoneCount`, `vnodesWithData`, `vnodesWithTombstones`) support observability.

- All mutations that touch storage, meta, or the vnode/tombstone indices are performed under the single `vNodeMu` lock (writers use `Lock`, readers use `RLock`). Combined with the Event Loop's batched single-goroutine serialization, this provides atomic cross-structure updates with low read contention.
- The `Apply(cmd)` method implements monotonic versioning and idempotence: a zero `cmd.Version` is locally incremented as a legacy fallback; any `cmd.Version` less than the current meta version is ignored for stale-write safety.
- On `OpPut`, the store replaces the value, updates `valueBytes`, ensures the key appears in `vNodeIdx`, clears any existing tombstone, and records `meta{Version , Tombstone:false}`.
- On `OpDelete`, it removes the value, registers a tombstone in `tombstoneIdx`, updates `tombstoneCount`, and records `meta{Version, Tombstone:true}`. All operations occur under the lock to maintain invariants.
- Reads via `GetWithVersion` acquire an `RLock`, return a cloned value and the logical version (or `ErrKeyNotFound` if the key is absent or tombstoned).
- `SnapshotVNode` produces a point-in-time clone of all live entries and tombstones for a given vnode (used during transfer), while `DeleteVNodeData` atomically clears a vnode's data after ownership transfer.
- The Store is tightly coupled with the durability layer: the durability log replays the same `Command{Op, VNodeID, Version, Key, Value}` layout, and replayed commands are applied via `Apply` using identical version checks to guarantee crash-recovery idempotence.

Operationally, the design trades modest extra memory for the `vNodeIdx` and `tombstoneIdx` structures to achieve linear-time migration and inexpensive per-vnode snapshots, while keeping the common Get/Put/Delete paths O(1).

2.6 Distributed Architecture

This section explores the control plane design, which is where our 'network-aware' contribution lies. The main responsibilities of the control plane are:

- **Data Placement:** deciding which node should be primary for each vnode, how many of them it should hold, and where to replicate to, all based on

runtime metrics and network conditions.

- **Membership and Failure Detection:** keeping track of which nodes are alive and their current load/latency metrics to inform placement decisions.
- **Rebalancing & Migration:** changing primary and replica assignments around the cluster as conditions change, and performing the necessary data transfer to move vnode ownership.

Our system handles sharding using a 2^{64} hash space evenly partitioned among a fixed number of vnodes (1024 in our implementation) that are assigned to physical nodes by a placement algorithm. Each vnode has one primary owner and a configurable number of replicas. The placement algorithm takes into account node-local metrics and inter-node latency to make informed decisions about where to place primaries and replicas for optimal performance and load distribution.

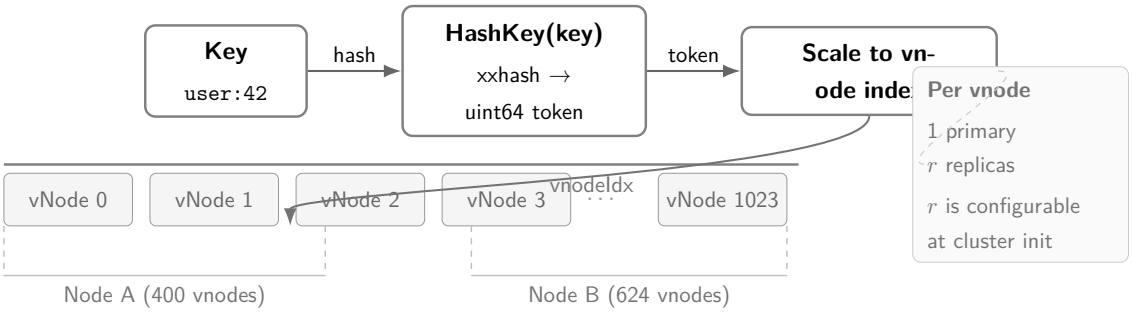


Figure 2.7: Hash space partitioning

To find the corresponding vnode for a key, the key is first hashed to a 64-bit token using xxhash, then scaled down to a vnode index using a fast multiplication-based scaling method (similar to Lemire's fast range scaling[20]), which avoids expensive modulo operations and provides a faster mapping from token to vnode index. This design allows for efficient lookups and straightforward data partitioning across the cluster.

2.6.1 Primary & Replica Placement

The placement layer is driven by a runtime metrics pipeline that collects local resource information from each node, disseminates those measurements through the cluster, and converts them into deterministic vnode ownership decisions. These decisions are made by a single elected leader and committed through Raft, ensuring that all nodes eventually converge on the same placement view rather than independently deriving conflicting assignments. Each node periodically reports a metrics

vector

$$m_i = \left(\text{AvgRTT}_i, \text{CPUUsage}_i, \text{MemUsage}_i, \text{NetUsage}_i, \text{DiskUsage}_i, \text{CPUCores}_i, \text{MemGB}_i, \text{BandwidthMbps}_i, \text{DiskGB}_i\right),$$

where the usage terms are normalized to $[0, 1]$, and the remaining terms represent the node's static hardware capacity. The collector computes AvgRTT_i from peer latency samples using an exponential moving average. For a latency sample x_{ij} to peer j, the update rule is

$$\text{RTT}_{ij}^{(t)} = \begin{cases} x_{ij}, & \text{if this is the first sample for peer } j, \\ 0.8\,\text{RTT}_{ij}^{(t-1)} + 0.2\,x_{ij}, & \text{otherwise.} \end{cases}$$

The node-wide RTT used by placement is the arithmetic mean over the currently known peer estimates,

$$\text{AvgRTT}_i = \frac{1}{|P_i|} \sum_{j \in P_i} \text{RTT}_{ij},$$

with a fallback of 1.0 ms when no samples are available. These metrics are gossiped as part of the cluster state, allowing every node to evaluate placement inputs consistently.

Given the reported metrics, the system computes a placement score for each node. The score is constructed so that smaller values indicate a better placement candidate. First, the static capacities are normalized against the cluster-wide averages:

$$RS_{\text{CPU},i} = \frac{\text{CPUCores}_i}{\overline{\text{CPUCores}}}, \qquad RS_{\text{MEM},i} = \frac{\text{MemGB}_i}{\overline{\text{MemGB}}},$$

$$RS_{\text{NET},i} = \frac{\text{BandwidthMbps}_i}{\overline{\text{BandwidthMbps}}}, \quad RS_{\text{Disk},i} = \frac{\text{DiskGB}_i}{\overline{\text{DiskGB}}}.$$

Disk pressure is modeled using a penalty that rises sharply as disk utilization approaches saturation:

$$P_{\text{disk},i} = \frac{1}{1.01 - \min(\text{DiskUsage}_i,\, 0.99)}.$$

The final placement score is then

$$S_i = w_{\text{RTT}} \frac{\text{AvgRTT}_i}{\overline{\text{AvgRTT}} + \varepsilon} + w_{\text{CPU}} \frac{\text{CPUUsage}_i}{RS_{\text{CPU},i}} + w_{\text{MEM}} \frac{\text{MemUsage}_i}{RS_{\text{MEM},i}} + w_{\text{NET}} \frac{\text{NetUsage}_i}{RS_{\text{NET},i}} + w_{\text{Disk}} \frac{P_{\text{disk},i}}{RS_{\text{Disk},i}},$$

where

$$w_{\text{RTT}} = 0.31, \quad w_{\text{CPU}} = 0.15, \quad w_{\text{MEM}} = 0.16, \quad w_{\text{NET}} = 0.24, \quad w_{\text{Disk}} = 0.14,$$

and $\varepsilon = 10^{-9}$ is used for numerical stability. In effect, the score favors nodes with lower latency, lower utilization, and greater available capacity.

The score is not used directly as a vnode count. Instead, it is inverted into a weight so that better nodes receive a larger share of the vnode space:

$$W_i = \frac{1}{S_i + \varepsilon}.$$

Let

$$W_{\text{tot}} = \sum_{k=1}^{n} W_k$$

be the total weight across the cluster and let $V = 1024$ be the total number of vNodes. The ideal vnode allocation for node i is

$$T_i^* = \frac{W_i}{W_{\text{tot}}} V.$$

The actual target count is obtained by taking the floor of each T_i^* and then distributing the remaining vNodes to the nodes with the largest fractional remainders. This yields an integer allocation that is deterministic, proportional to the measured node quality, and still sums exactly to 1024.

Once the target vnode counts are known, the leader reassigns ownership. The current vnode map is first scanned to count how many vNodes each node owns. Any vnode whose owner is no longer present, or whose owner exceeds its target count, is revoked and marked as orphaned. The orphaned vNodes are then assigned to nodes that still have placement deficit. To keep this reassignment deterministic, each orphaned vnode v is paired with each candidate node p through a hash-based scatter key,

$$H(p, v) = \text{xxhash}(p \,\|\, \text{":"} \,\|\, v),$$

and the node with the best score among the remaining deficit nodes is chosen as the new primary. This produces a stable, reproducible placement decision without requiring a full global optimization step.

Replica assignment uses a related but separate cost function. For each vnode with primary p, the system evaluates every other node $r \neq p$ as a replica candidate and computes

$$C(p, r) = \text{RTT}_{p,r} \cdot \frac{1}{1.01 - \text{NetUsage}_r} \cdot \frac{1}{\text{BandwidthMbps}_r + \varepsilon}.$$

The first term captures the pairwise latency between the primary and the candidate replica. The second term penalizes network saturation on the replica node, and the

third term prefers nodes with higher physical link capacity. Candidates are sorted in ascending order of cost, and the first $rf - 1$ nodes are selected as replicas for that vnode, where rf is the replication factor. In this way, the system places primaries according to global node health and capacity, then places replicas according to pairwise communication cost and available network headroom.

In summary, the placement pipeline is straightforward but expressive: runtime metrics are collected locally, normalized and gossiped cluster-wide, converted into node scores, translated into vnode target counts, and finally used to assign both primaries and replicas. The result is a deterministic, capacity-aware placement policy that reacts to current cluster conditions while remaining stable enough for practical rebalancing.

2.6.2 Replication & Quorums

Replication is responsible for propagating each primary write to the replica nodes selected by the committed placement. The placement itself is generated by a single elected leader and committed through Raft, so the replication layer always acts on a consistent vnode map rather than a locally inferred one. For a key k, the placement returns the vnode owner and its replica set:

$$(\text{primary}(k), \text{replicas}(k)) = \text{Placement.GetNodesForKey}(k).$$

The replication layer then removes the local node from the target set, deduplicates the remaining replica identifiers, and sends the write only to those remote nodes that are currently eligible to receive it.

Each write is represented as

$$r = (\text{op}, \text{key}, \text{value}, \text{vnodeID}, \text{version}),$$

where $\text{op} \in \{\text{PUT}, \text{DELETE}\}$, vnodeID identifies the vnode owning the key, and version is the logical timestamp assigned at the RPC layer. The vnode identifier is carried through replication so that the receiving node can apply the update in the correct vnode context, and the version ensures that replicated writes remain monotonic and compatible with the store's stale-write checks. In particular, the primary first applies the write locally, and only then propagates it to the remote replica set.

The system enforces a quorum-style commit rule. If the configured replication

factor is rf, the effective factor is

$$rf_{\text{eff}} = \min\bigl(\max(rf, 1),\, N\bigr),$$

where N is the number of currently visible cluster members. The required acknowledgment threshold is then

$$q = \left\lfloor \frac{rf_{\text{eff}}}{2} \right\rfloor + 1.$$

The primary contribution counts as one acknowledgment immediately after the local write succeeds. If the number of available targets is insufficient to reach q, the write fails early instead of being accepted in a partially replicated state. This keeps durability semantics explicit: a write is not considered committed unless a quorum can be reached.

To keep the replication path efficient, the implementation does not issue one RPC per write. Instead, each target node has a dedicated batch worker that accumulates pending replication requests into batches of size

$$B = 512$$

or flushes them after a small time window

$$\Delta t = 1 \text{ ms}.$$

When a batch is flushed, the worker sends a `ReplicationBatchRequest` over a bidirectional streaming RPC and waits for a corresponding `ReplicationBatchAck`. This amortizes network overhead while preserving the ordering guarantees needed by the write path. The batch acknowledgment reports how many entries were applied and may include an error string if the remote side failed to process the batch. In effect, replication is executed as a stream of ordered, batched writes with per-target backpressure and explicit completion tracking.

The public RPC surface mirrors this behavior. The single-write `Replicate` method and the batched `ReplicateBatch` and `ReplicateStream` methods all feed into the same internal request path used by primary writes. As a result, replicated operations do not bypass the event loop or the store's versioning rules; they are serialized through the same mutation pipeline as client-originated updates. This is important because it keeps the primary path, replica path, and store semantics aligned: the same vnode ID, operation type, and version are used everywhere, and the same apply logic determines whether the update is accepted.

Overall, replication in this system combines three ideas: placement determines

the replica set, the quorum rule determines when a write may be considered successful, and batched streaming RPCs keep the fan-out efficient under load. This gives the replication layer a clean separation of concerns while still fitting tightly into the placement-driven vnode architecture.

2.6.3 Placement Re-evaluation, Hysteresis, and Migration

The placement layer is not recomputed continuously. Instead, it is evaluated at two different timescales: a long-term evaluation that may produce a new vnode assignment for the entire cluster, and a short-term evaluation that performs limited role swaps when a primary node becomes degraded but a replica can safely take over. Both paths are protected by hysteresis checks so that small metric fluctuations do not trigger unnecessary churn. In all cases, the resulting placement is committed by Raft, which means that a single elected leader proposes the updated vnode map and the rest of the cluster converges on the same committed view.

The long-term evaluation is the main placement update path. It begins by collecting the current cluster membership and filtering the reported metrics to active nodes only. The scoring function then produces a ranking

$$S_i = f(\text{AvgRTT}_i,\ \text{CPUUsage}_i,\ \text{MemUsage}_i,\ \text{NetUsage}_i,\ \text{DiskUsage}_i,\ \text{CPUCores}_i,\ \text{MemGB}_i,\ \text{BandwidthMbps}_i,\ \text{DiskGB}_i), \quad (2.1)$$

where lower scores indicate better placement candidates. These scores are converted into target vnode counts, which are then used to reassign vnode primaries and replicas. The resulting vnode map is compared against the previous placement before it is committed. Let

$$M_P = \#\{v \mid \text{Primary}_{\text{new}}(v) \neq \text{Primary}_{\text{old}}(v)\}$$

be the number of vNodes whose primary owner changes, and let

$$M_R = \#\{v \mid \text{Replicas}_{\text{new}}(v) \neq \text{Replicas}_{\text{old}}(v)\}$$

be the number of vNodes whose replica set changes. The system only commits the new placement if the drift is large enough to justify the update. Otherwise, the old epoch is preserved and the cluster avoids unnecessary redistribution. In the implementation, the primary drift threshold is

$$T_P = \lfloor 0.12 \cdot V \rfloor,$$

where $V = 1024$ is the number of vNodes, and the replica drift threshold is

$$T_R = \lfloor 0.20 \cdot V \cdot (rf - 1) \rfloor ,$$

where rf is the replication factor. If

$$M_P < T_P \quad \text{and} \quad M_R < T_R,$$

then the placement change is considered negligible and the epoch is left unchanged. This is the main hysteresis mechanism for the long-term path: small score changes do not immediately translate into a cluster-wide reconfiguration.

When the long-term drift exceeds the thresholds, the new placement is committed through Raft. The committed vnode map may then cause a node to gain ownership of some vNodes and lose ownership of others. For every gained vnode, the new owner performs a full vnode transfer from the previous primary or, if necessary, from one of the replicas. For every lost vnode, the old owner keeps the data only for a grace period and then drops it. This is the full migration path: it is triggered only when a committed placement change changes vnode ownership, and it ensures that state is moved in a controlled manner rather than abruptly discarded.

The short-term evaluation is more conservative and is intended to react to node degradation without waiting for a full placement recomputation. It examines the current placement and identifies primaries that have remained severely degraded for several evaluation cycles. A node is considered severely degraded if it is overloaded or slow, for example when

$$\text{CPUUsage}_i \geq 0.90, \qquad \text{DiskUsage}_i \geq 0.90, \qquad \text{or} \qquad \text{AvgRTT}_i \geq \max(120,\ 2{\cdot}\overline{\text{RTT}}),$$

where $\overline{\text{RTT}}$ is the cluster average RTT among active nodes. Degradation is not acted upon immediately. Instead, the node must remain degraded across multiple short-term samples, and in the implementation the node is only considered sustained-degraded after six consecutive detections. This streak requirement prevents one-off spikes from causing role swaps.

Once a vnode is identified whose primary is sustained-degraded, the short-term evaluator inspects its replicas and looks for a healthier candidate. A replica is eligible only if it is not itself degraded, and the swap is accepted only when the replica is sufficiently better than the current primary. In the implementation, the replica must improve on the primary's score by at least 10%, which can be written as

$$S_{\text{replica}} < 0.90\, S_{\text{primary}}.$$

If such a candidate exists, the vnode performs a role swap: the replica becomes the new primary and the old primary is moved into the replica position. This allows the system to react to localized node degradation without reassigning the entire vnode distribution. The result is a much smaller movement set than a full placement update, while still preserving availability under adverse conditions.

The migration pipeline follows from whichever evaluation path produced the new placement. A full placement change leads to vnode transfers for newly gained primaries and cleanup of vnode state on nodes that lost ownership. A short-term role swap does not move the vnode's data elsewhere in the cluster; instead, it changes which replica is responsible for serving as the primary. In both cases, Raft acts as the coordination boundary, so the cluster does not independently invent conflicting ownership maps. This is important because it separates *placement computation* from *placement commitment*: the former may be speculative, but the latter is authoritative.

Taken together, the system uses hysteresis at two levels. At the cluster level, long-term placement updates must exceed primary and replica drift thresholds before they are committed. At the node level, short-term role swaps require sustained degradation before a replica can be promoted. These two filters keep the cluster stable under transient noise while still allowing it to respond to persistent load imbalance or failure pressure. The practical effect is that the system only performs expensive full vnode transfers when the committed placement truly changes, and otherwise prefers the cheaper role-swap path when a primary merely becomes temporarily unhealthy.

2.6.4 Gossip and Raft

Gossip and Raft together form a two-layer control plane that balances timeliness and correctness. Gossip (implemented with `memberlist`) supplies a low-latency, best-effort view of cluster liveness, RPC addresses and smoothed resource metrics; Raft (the `RaftNode` wrapper and placement FSM) supplies a durably ordered, linearizable sequence of control commands (notably placement epochs and membership changes). By separating these responsibilities we keep the per-key data path (durability log + single-goroutine EventLoop) lightweight while ensuring that cluster-wide metadata is consistent and recoverable.

Gossip is used as an operational signal: nodes publish periodic metric samples and probe results which the leader consumes (see `Replicator.StartMetricsReporter` and the `MetricsCollector`). These metrics feed the placement scoring and short-term swap heuristics executed by the leader in `Replicator.UpdatePlacement` and

`UpdatePlacementShortTerm`. Because gossip is eventual and may be partitioned or delayed, it is never treated as authoritative for control-plane changes, its role is to inform candidate placements and to surface fast failure signals.

Raft is the authoritative sequencer for control state that must be globally agreed and durable. Placement proposals computed by the leader are proposed with `RaftNode.ApplyPlacement`; once a Raft log entry commits the new epoch becomes canonical and every node executes the deterministic apply path in `ApplyPlacementFromRaft`. Thus Raft ensures the following invariant for placement epochs: if a node observes epoch e then it has observed all epochs $e' \leq e$ in the same order (Raft linearizability). Importantly, Raft is not used to order per-key writes, the system relies on the durability log and the EventLoop for per-key ordering and durability to keep the write fast-path inexpensive.

The interaction model is intentionally staged. The leader samples gossip, computes a candidate placement, and proposes it through Raft. Commitment of an epoch is synchronous with respect to the control plane and immediately changes vnode ownership as seen by routing and placement queries; data movement necessary to make the new owner fully serving is performed asynchronously after commit. The apply path computes delta sets (gained/lost primaries, replica changes) and launches non-blocking catch-up transfers via the `TransferVNode` RPC (`transferVNodeFrom`). This design creates a controlled timing gap: placement commit (control plane) precedes data availability (data plane). The implementation mitigates the gap using a configurable *GracePeriod* before dropping lost vnode data, by exposing migration hints (`GetMigrationSourceForKey`), and by instrumenting transfer progress so operators can detect stalled migrations.

Several operational and safety concerns follow from this division of labor. First, because gossip is only a hint, placement algorithms include hysteresis and conservative thresholds to avoid oscillation from noisy signals. Second, because placement is authoritative but transfers are async, clients or internal routing should expect that ownership change does not necessarily imply immediate local availability; routing may consult migration hints or fall back to remote reads while catch-up completes. Third, it is essential that transferred snapshot entries be applied through the normal write pipeline: the `SetTransferApplier` hook should invoke the EventLoop so entries undergo durability log encoding, version checks and the same ack semantics as local writes.

Finally, the codebase provides a reconciliation loop that bridges gossip and Raft: `ReconcileRaftWithMembership` adds nodes observed via gossip into the Raft configuration (via `AddVoter`) and, after conservative timeouts, removes long-missing peers to

protect quorum safety. For robustness we recommend monitoring placement epoch progression, active migration metrics, and transfer latency, and implementing the noted version-aware merge on transfers (compare streamed `version` fields against local versions before applying) to avoid overwriting concurrent newer writes.

Together, gossip and Raft implement a pragmatic control plane: gossip supplies rapid operational evidence for adaptive placement, while Raft supplies the durable linearization necessary for safe vnode ownership and membership changes.

2.7 Throughput Testing

To exercise the distributed data plane we built a configurable, single-binary workload driver ("loadgen") that targets the placement-aware RPC surface. The driver is intended to produce repeatable, production-like load while measuring both throughput and latency distributions under realistic routing, placement and migration behaviors.

Architecturally the driver opens a pool of gRPC client connections to a placement router `PlacementRouterWithPoolSize` and runs a configurable number of concurrent worker goroutines. Each worker repeatedly selects an operation (PUT, GET, DELETE) according to an explicit mix and picks a key from a selectable key distribution. Supported distributions include uniform, Zipfian (skew parameter s and offset v), and a 'latest' bias that favors recently created keys. These distributions allow the experimenter to stress uniform storage, hotspot contention, and recency-biased workloads respectively.

The driver supports three lifecycle phases that mirror standard benchmarking practice: (1) a prefill stage which populates a portion of the logical keyspace in parallel to establish baseline data; (2) a warmup phase (no measurement) so caches, connection pools and placement caches stabilize; and (3) a measured phase which collects the metrics used for reporting. Prefill is parallelized across the configured 'Clients' workers and uses a deterministic seed to facilitate reproducible conditions. The router is refreshed once at startup so that worker requests are routed according to the current placement; this ensures the load traverses the real distributed paths and triggers replication/transfer behaviours when placement changes occur.

2.8 Conclusion

This chapter presented the design decisions of our distributed KV store with a focus on network-aware control plane decisions. The system is built around a vnode-based

sharding architecture, a durability log for crash recovery and persistence, and a control plane that uses runtime metrics to drive placement and replication decisions. The design emphasizes separation of concerns: the durability log handles per-key persistence, the in-memory store manages versioned state, and the control plane orchestrates cluster-wide placement and replication based on observed conditions. The result is a system that can adapt to changing load and failure conditions while maintaining consistency guarantees.

Chapter 3

Evaluation & Analysis

3.1 Introduction

This chapter evaluates the system using a controlled workload driver that exercises the real placement-aware RPC path. The goal is to measure end-to-end throughput, latency, and placement behavior under several request mixes and key-access patterns, while keeping the experimental protocol fixed so that differences in the results can be attributed to the system rather than to the benchmark setup.

3.2 Experimental Methodology

All experiments use a fixed value size of 512 bytes and the same acknowledgement policy. In the implementation, acknowledgement policy 1 corresponds to `AckAfterFlush`, meaning that a write is acknowledged after it has been flushed to the operating system, but not necessarily fully `fsync`'d to stable storage. This provides a realistic durability setting without forcing the strongest and slowest write path on every request. We also use a default replication factor of 3 for most experiments, which is a common default in production systems and allows us to evaluate the placement and replication behavior without confounding it with changes in the replication factor. The workload driver is implemented as a separate client process that generates requests according to the specified mixes and distributions, and measures the resulting throughput and latency, in most of our tests, we used 200 concurrent workers to generate load, with a runtime of 8 minutes per run. (preceeded by a prefill stage and a 2 minute warmup stage, which are not included in the runtime)

General throughput and latency workload The main workload suite is designed to represent three common access profiles: read-heavy, balanced, and write-heavy. Each workload is executed twice, once with a uniform key distribution and once with a latest-biased distribution. Uniform traffic spreads requests evenly across the keyspace, while latest-biased traffic concentrates requests on recently used keys and exposes hotspot behavior, temporal locality, and cache sensitivity.

Workload	GET	PUT	DELETE	Distribution
Read-heavy	85%	11%	4%	Uniform, Latest
Balanced	50%	45%	5%	Uniform, Latest
Write-heavy	25%	65%	10%	Uniform, Latest

Table 3.1: Request mixes used in the main throughput and latency evaluation.

The read-heavy workload emphasizes lookup performance and routing overhead, the balanced workload approximates a mixed transactional setting, and the write-

heavy workload stresses the event loop, durabilty-log, and replication pipeline under sustained update pressure. The latest-biased distribution is especially useful for revealing whether placement and replication amplify contention when access is concentrated on a smaller hot set.

We also ran this benchmark on two different cluster sizes, a 4-node and a 6-node configuration (which uses 300 concurrent workers instead of the default 200), to observe how the system scales horizontally under the same workload conditions. The results are summarized in Section 3.3.

Keyspace and prefill The keyspace must be large enough to avoid a tiny working set that repeatedly hits the same few keys, but not so large that the benchmark becomes dominated by cold inserts and one-time accesses. For this reason, the benchmark uses a fixed logical keyspace K together with an initial prefill size P.

A practical choice is to treat P as a fraction of K rather than as an arbitrary absolute number. In practice, a prefill ratio in the range $0.3 \leq P/K \leq 0.5$ is usually a reasonable compromise: it leaves enough free space to generate new writes during the measurement phase, while still establishing a warm steady-state dataset for reads. The exact values can be chosen according to the target machine capacity, but the important property is the ratio, not the raw count. The keyspace should be much larger than the number of concurrent workers so that the benchmark measures distributed storage behavior rather than trivial single-key contention.

Micro-benchmarks In addition to the main workload suite, the evaluation includes several focused micro-benchmarks that isolate specific design choices.

Micro-benchmark	**Variable Under Test**	**Primary Question**
VNode allocation fairness	Placement score and vnode target count	How evenly are primaries distributed across nodes under a fixed cluster state?
Replica assignment sensitivity	RTT and bandwidth inputs to replica scoring	Which nodes are preferred as replicas when latency and load differ?
Replication factor effects	Replication factor RF	How does increasing/reducing the replication factor affect throughput and latency?

Table 3.2: Focused micro-benchmarks used to isolate system behavior.

The vnode allocation fairness experiment measures how the placement layer distributes primaries across nodes for a fixed cluster snapshot. Here the key output

is the per-node share of primary vnodes, compared against the target vnode counts computed from the node scores. This shows whether the allocation logic is balanced and whether high-capacity, low-cost nodes receive a proportionally larger share of the keyspace.

The replica assignment sensitivity experiment focuses specifically on how node-to-node RTT and bandwidth affect replica placement. Because replica selection uses both link latency and bandwidth-related penalties, this test can show whether the system prefers fast and lightly loaded nodes as replicas, and whether the chosen replica set remains reasonable when some nodes are slower or more saturated than others.

The replication factor effects experiment varies the replication factor RF and measures how throughput and latency change in response. This is important for understanding the cost of replication and quorum semantics, and for confirming that the replication pipeline scales as expected when more replicas are added.

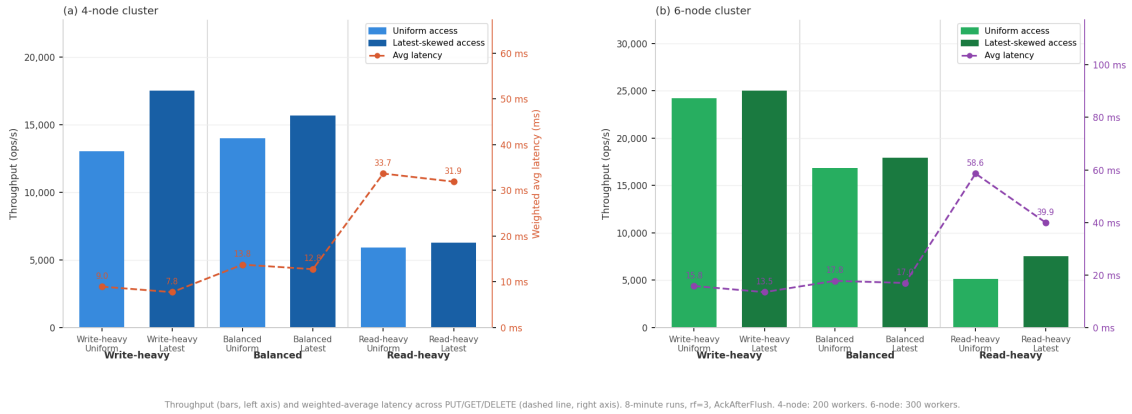
All micro-benchmarks were run on the 4-node cluster configuration, which is sufficient to observe the behaviour of the system.

Experimental protocol Each run should begin with a prefill stage that populates the chosen portion of the keyspace, followed by a warmup interval and then a measured interval. The warmup interval is important because the system under test performs asynchronous placement updates, caching, and replication activity; measuring immediately after startup would overemphasize initialization effects rather than steady-state behavior. During the measured interval, the benchmark records per-operation success and failure counts together with latency histograms, which can later be summarized as throughput, average latency, and tail latency.

The workload driver is intentionally designed to exercise the actual distributed path rather than an isolated in-memory path. This means that the measurements naturally include the effects of vnode routing, placement decisions, write batching, and replication behavior. As a result, the numbers reflect the behavior of the system as deployed, rather than a simplified microservice surrogate.

3.3 General benchmark results

The benchmark results for both the 4-node and 6-node configurations are summarised in Figure 3.1. Each panel shares the same workload structure: three profiles crossed with two key-access distributions, allowing a direct visual comparison of how the cluster scales.



Throughput (bars, left axis) and weighted-average latency across PUT/GET/DELETE (dashed line, right axis). 8-minute runs, rf=3, AckAfterFlush. 4-node: 200 workers. 6-node: 300 workers.

Figure 3.1: Benchmark results for the main workload suite. with two different cluster sizes.

3.3.1 Write-heavy workload.

Under sustained write pressure (65% PUT, 25% GET, 10% DELETE), the 4-node cluster delivers **13,038 ops/s** under uniform access and **17,517 ops/s** under latest-biased access, with weighted-average latencies of **8.96 ms** and **7.75 ms** respectively. The 6-node cluster scales this substantially, reaching **24,210 ops/s** (uniform) and **25,019 ops/s** (latest) at latencies of **15.83 ms** and **13.50 ms**. The throughput gain from adding two nodes is approximately 1.86× under uniform access, consistent with the additional primary vnodes absorbing write fan-out. The latency increase on the 6-node cluster is attributable to the higher replication fan-out across more nodes and the increased quorum coordination overhead at larger scale. It is worth noting that the write-heavy latest run on the 4-node cluster recorded a failure rate of 0.18% (15,091 failures out of 8.4M operations), likely caused by a transient placement rebalancing event during the run; all other configurations maintained a 100% success rate.

3.3.2 Balanced workload.

At a 50/45/5 GET/PUT/DELETE split, the 4-node cluster achieves **14,017 ops/s** (uniform) and **15,663 ops/s** (latest), with notably low latencies of **13.77 ms** and **12.77 ms**. The 6-node cluster reaches **16,859 ops/s** (uniform) and **17,950 ops/s** (latest) at **17.79 ms** and **16.98 ms**. The scaling gain here is more modest ($\approx 1.20\times$) than in the write-heavy case, reflecting the higher GET share: reads are served inline without batching and are distributed across more nodes, so additional nodes help but are not as impactful as for write-dominated traffic.

3.3.3 Read-heavy workload.

The read-heavy workload (85% GET, 11% PUT, 4% DELETE) produces the most pronounced contrast between configurations and access patterns. Under uniform access, the 4-node cluster achieves **5,938 ops/s** at **33.68 ms**, while the 6-node cluster actually drops to **5,117 ops/s** at **58.62 ms**. This counter-intuitive result is explained by the increased replication overhead and cross-node routing cost with more nodes: under uniform access, reads are spread evenly across the keyspace and all 1024 vnodes, so each additional node introduces more inter-node coordination without concentrating the benefit. Under latest-biased access the picture reverses: the 4-node cluster reaches **6,267 ops/s** at **31.91 ms**, while the 6-node cluster reaches **7,517 ops/s** at **39.90 ms**, a 1.20× throughput gain. Here the hot set is small enough that the additional nodes help distribute the concentrated read traffic across more primaries without proportionally increasing coordination cost.

3.3.4 Scaling behaviour and hardware context.

Across the write-heavy and balanced workloads, adding two nodes yields throughput gains of 1.20×–1.86×, which is sub-linear but expected for a system with rf=3 replication: each write must still reach a quorum of two remote nodes regardless of cluster size, so the marginal benefit of additional nodes is bounded by the replication cost. The read-heavy uniform result is a practical reminder that horizontal scaling in a distributed KV store benefits write-dominated and locality-biased workloads more than it benefits uniformly distributed read traffic at small cluster sizes. As with the original results, the absolute throughput figures should be interpreted in the context of the hardware: all nodes are shared-tenancy `c7i.flex.large` EC2 instances (2 vCPU, 4 GB RAM), not dedicated bare-metal servers, and the client workload generator runs on an instance of the same type. The numbers reflect realistic cloud deployment conditions rather than peak theoretical throughput.

Takeaways The two-cluster comparison confirms that the system scales horizontally for write-heavy and locality-biased workloads, with the placement layer correctly redistributing vnodes across the larger node set without manual intervention. The read-heavy uniform case highlights the known trade-off between replication overhead and read scalability at small cluster sizes. Zero-failure operation is maintained across all configurations with the exception of one transient event in the 4-node write-heavy latest run.

3.4 Micro-benchmarks results

3.4.1 Primary distribution

To see the effect of the placement layer on primary distribution, and whether it's fair and well-proportioned, we simply read the server logs on the leader node and the other members, we launched the cluster and waited for the membership state to stabilize, initially, the leader node owns all 1024 vnodes as primaries, but once membership settled in, and the placement evaluation interval elapsed, we see the following logs:

```
[placement-long] score snapshot: node-10000=0.5564, node-10002=0.5575, node
-10001=0.5685, node-9999=0.5702
[placement-long] target vnode counts: node-10000=259, node-10001=254, node-10002=258,
 node-9999=253
```

which show the calculated scores (lower is better) and the corresponding target vnode counts for each node. To confirm that the actual distribution matches the target, we wait until the ownership transfer completes and the placement is commited, then we see the following logs on the leader:

```
[placement-raft] epoch=1->2: gained=0 lost=771 primary-replica-changes=253
new-replicas=0
```

which confirms the leader only kept 253 primaries and transferred 771 to the other nodes, and on the other nodes, we see in order:

```
[migration] epoch=1->2: completed transferring, gained 259 vnodes, lost 0 vnodes

[migration] epoch=1->2: completed transferring, gained 254 vnodes, lost 0 vnodes

[migration] epoch=1->2: completed transferring, gained 258 vnodes, lost 0 vnodes
```

confirming that the other three nodes received their target shares of primaries. This shows that the placement layer correctly computes scores, translates them into target vnode counts, and that the allocation logic distributes primaries according to those targets.

3.4.2 Replica assignment

To observe the effects of node-to-node RTT and bandwidth on replica assignment, we'll use tc to artificially manipulate the network conditions between nodes. In this scenario,we use a replication factor of 2 ($RF = 2$), meaning each write only gets

replicated once, and let the cluster stabilize with normal conditions, which gives us this replica map:

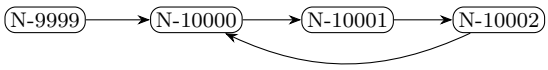


Figure 3.2: Initial state for replica write targets. ($RF = 2$)

then we add 50ms and 100ms of latency to the network interfaces for node-10000 and node-10001, leaving node-10002 untouched, and after the next placement evaluation interval, the replica map changes to:

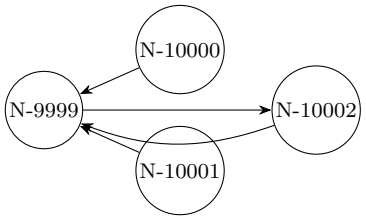


Figure 3.3: Replica assignment changes with added RTT

now node-9999 and 10002 become the only replication targets, given they got no added latency, which shows that the system correctly prefers lower latency nodes as replicas. Now on top of those added latencies, we saturate the NIC on node-10002, using iperf3 from the client node with 12 parallel streams,which maxes out the net usage on it, and after the next evaluation interval, the replica map changes to:

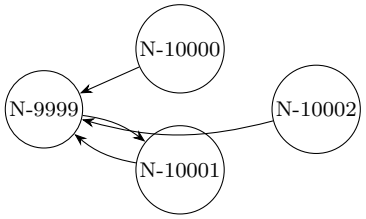


Figure 3.4: Replica assignment changes with added bandwidth saturation

now N-10001 takes its place, given it has less latency than N-10000, and lower NIC usage, which shows that the system correctly penalizes saturated nodes and prefers less loaded nodes as replicas. This confirms that both latency and bandwidth inputs are effectively influencing replica placement decisions.

And to further show how these metric changes affect primary distribution, the figure below shows the changes in scores, and vnode counts per node in each placement epoch.

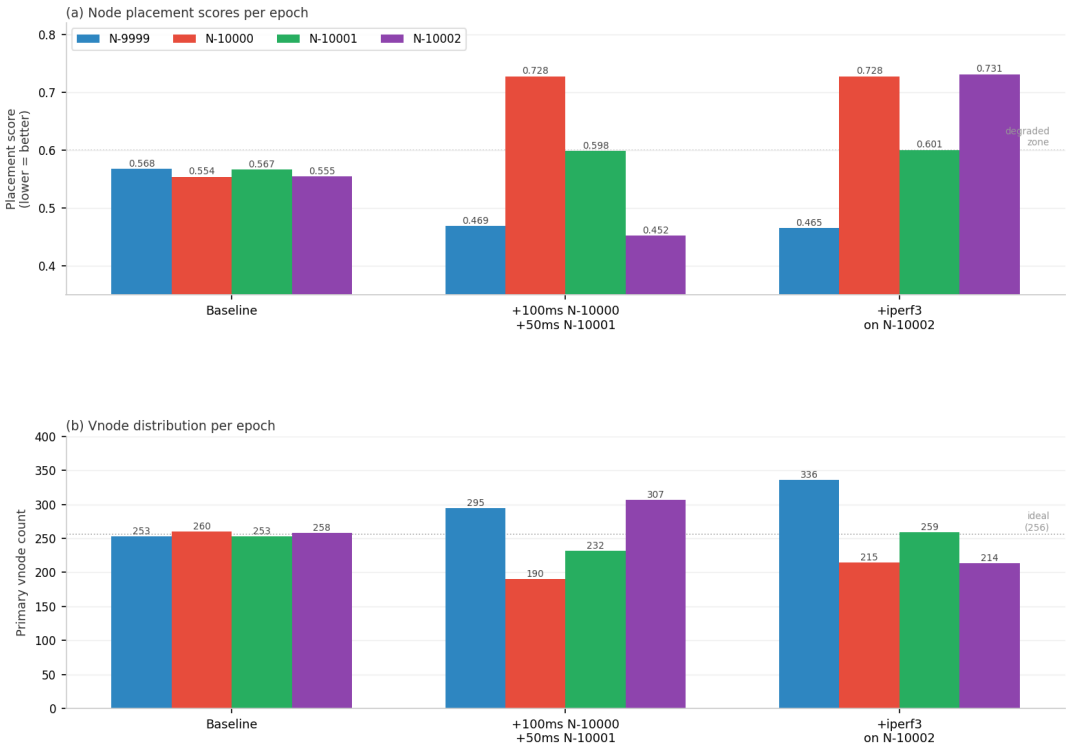


Figure 3.5: Placement sensitivity to network metrics

3.4.3 Replication factor sensitivity

Figure 3.6 shows throughput and weighted-average latency for the balanced uniform workload across $rf \in \{1, 2, 3, 4, 5\}$. The dominant cost driver is the quorum threshold $q = \lfloor rf_{\text{eff}}/2 \rfloor + 1$, not the fan-out count: throughput drops steeply from **15,099 ops/s** at $rf = 1$ to **11,804 ops/s** at $rf = 2$ (-22%) as q increases from 1 to 2, then more gradually to **10,896 ops/s** at $rf = 3$ (-8%) as only the fan-out grows. At $rf = 4$ and $rf = 5$ throughput plateaus at roughly **8,500 ops/s** because $rf_{\text{eff}} = \min(rf, N) = 4$ in both cases, the four-node cluster saturates the replication target set and both configurations are effectively identical. Overall, $rf = 3$ retains **72%** of the no-replication throughput at a latency overhead of approximately **5 ms**, confirming that the replication pipeline introduces no disproportionate cost beyond what the quorum semantics inherently require.

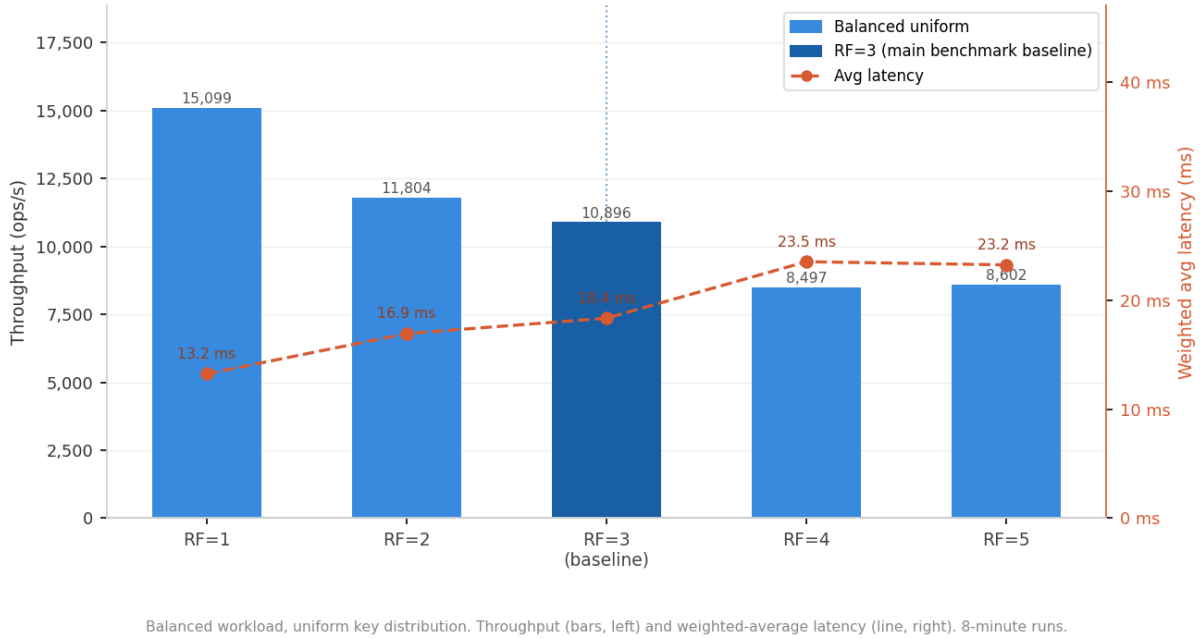


Figure 3.6: Throughput and latency for varying replication factors

3.5 Conclusion

In this chapter we evaluated the system under a variety of workloads and configurations, demonstrating that it scales horizontally for write-heavy and locality-biased traffic, while also confirming that the placement layer effectively distributes primaries and selects replicas based on network conditions. The micro-benchmarks provided additional insight into specific design choices, showing that the system's dynamic placement logic responds appropriately to changes in latency and load, and that the replication pipeline scales as expected with increasing replication factors. Overall, the results confirm that the proposed network-aware placement strategy successfully balances load and optimizes performance without manual intervention.

Chapter 4

Discussion & Critical Reflection

4.1 Introduction

This chapter reflects on the design decisions made throughout the project, interprets the evaluation results in a broader context, discusses the limitations of the current prototype, and identifies directions for future work. The goal is not to restate what was measured, but to examine what the results reveal about the design choices and what a more mature version of this system would need to address.

4.2 Design Decisions and Their Trade-offs

4.2.1 Gossip for Metrics, Raft for Placement

The decision to separate the metrics dissemination layer from the placement commitment layer is perhaps the most consequential architectural choice in this system. Gossip provides low-latency, best-effort propagation of runtime metrics, which is exactly what an adaptive placement algorithm needs: frequent, approximate signals rather than perfectly consistent snapshots. Raft, on the other hand, provides the linearizable, durable sequencing that placement decisions require, since two nodes computing conflicting vnode maps would produce split-brain ownership and undefined replication behavior.

The practical cost of this separation is a timing gap between when a metric change is observed and when it influences a committed placement epoch. During this window, the cluster may continue using a suboptimal placement. In most cases this gap is short, but under sustained load with frequent metric fluctuations, the gossip-to-Raft pipeline may lag. The hysteresis thresholds mitigate unnecessary churn, but they also widen the reaction window. Tuning these thresholds is therefore a deployment concern that this prototype does not fully resolve.

4.2.2 Single-Goroutine EventLoop

Serializing all mutating operations through a single goroutine simplifies reasoning about concurrency considerably. The write path has no fine-grained lock contention between commands, and the durability log can append entire batches atomically. The trade-off is that the EventLoop becomes the primary throughput bottleneck for write-heavy workloads, which is clearly visible in the benchmark results: write throughput is stable and insensitive to key-access patterns, confirming that the loop, not routing or placement, is the binding constraint.

For the scale of this prototype, this is an acceptable trade-off. A production

system targeting higher per-node throughput would likely need to partition the EventLoop by vnode range, allowing multiple goroutines to process disjoint keyspace segments in parallel. This would require more careful coordination during vnode migration, since a transfer would need to drain and pause the relevant partition of the loop rather than the entire serialization path.

4.2.3 Vnode Count and Granularity

The fixed vnode count of 1024 was chosen to give the placement algorithm sufficient granularity to produce well-proportioned target allocations across a small cluster, while keeping the state tracked by the placement FSM compact. In a four-node cluster, 1024 vnodes translates to roughly 256 primaries per node under balanced conditions, which gives the scoring function enough resolution to reflect small score differences as meaningful allocation differences.

The limitation of a fixed vnode count becomes apparent when the cluster scales significantly. A cluster of 50 nodes with 1024 vnodes would have approximately 20 primaries per node, reducing the granularity of placement decisions. A production design would either use a larger vnode space or support dynamic vnode splitting, at the cost of a more complex migration protocol.

4.2.4 Placement Score Weights

The weights in the placement scoring function — $w_{\text{RTT}} = 0.31$, $w_{\text{NET}} = 0.24$, $w_{\text{MEM}} = 0.16$, $w_{\text{CPU}} = 0.15$, $w_{\text{Disk}} = 0.14$ — were chosen to reflect the thesis's central argument that network conditions should be the dominant placement signal in a distributed KV store. The relatively high weight on RTT and network usage reflects the observation that disk and CPU pressure on a single node is easier to address by scaling vertically, whereas network latency between nodes is a structural property of the topology that cannot be fixed by upgrading hardware.

These weights, however, were not derived from a systematic optimization process. They represent an informed prior rather than a calibrated result. A more rigorous approach would involve running the placement algorithm under a variety of synthetic cluster conditions and measuring how different weight configurations affect hotspot elimination and load balance convergence time. This remains an open direction.

4.3 Interpretation of Results

4.3.1 Write Path Insensitivity to Access Pattern

The near-identical throughput and latency figures across uniform and latest-biased distributions for write-heavy and balanced workloads are a direct consequence of the EventLoop design. Because all writes are serialized through a single goroutine and batched before hitting the durability log, the access pattern of the keys being written does not change the fundamental processing cost. This holds across both cluster sizes: the 4-node cluster shows a 34% gap between uniform and latest in the write-heavy workload, but this is attributable to the transient failure event recorded in that run rather than a genuine access-pattern effect; the 6-node cluster, which completed both runs cleanly, shows less than 4% difference between the two distributions. This is a positive result in the sense that the system is predictable under different write profiles, but it also means that the placement layer's benefits are not directly observable in write throughput; the bottleneck is structural, not placement-driven.

4.3.2 Read-Heavy Workload Variance

The most practically interesting result is the behaviour of the read-heavy workload across both cluster sizes and access patterns. Under latest-biased access, both configurations benefit from temporal locality: the 4-node cluster achieves 6,267 ops/s at 31.91 ms and the 6-node cluster reaches 7,517 ops/s at 39.90 ms, a modest but consistent scaling gain. Under uniform access the picture is more nuanced. The 4-node cluster achieves 5,938 ops/s at 33.68 ms, while the 6-node cluster drops to 5,117 ops/s at 58.62 ms, a counter-intuitive regression that is explained by the increased inter-node routing cost under uniform scatter: with more nodes, a larger fraction of reads must cross node boundaries without any locality benefit to offset the added RTT and coordination overhead. This result exposes a genuine limitation of the current design: the system does not serve reads from replicas, so additional nodes add replication cost without adding read capacity for uniformly distributed traffic. A more complete implementation would either route reads to the nearest replica or maintain a node-local read cache to absorb uniform read traffic before it hits the cross-node path. The earlier variance observed across multiple read-heavy uniform runs on the original cluster (one outlier reaching approximately 315 ms against a mean of 103.3 ms across five runs) is consistent with a placemen rebalancing event coinciding with a saturated read path, and similarly points to the placement layer needing to account for current read load before committing a new epoch rather than

acting on metric drift alone.

4.3.3 Replication Factor Saturation

The plateau in throughput at RF=4 and RF=5, caused by the four-node cluster saturating the replication target set ($rf_{\text{eff}} = \min(rf, N) = 4$ in both cases), is a useful reminder that replication factor and cluster size are coupled parameters. The result confirms that the system correctly clamps the effective replication factor to the number of available nodes, but it also highlights that benchmarking a replication-sensitive design on a four-node cluster limits the observable range of the replication factor axis. A larger cluster would be needed to separate the effects of quorum threshold increases from fan-out increases at higher replication factors.

4.4 Limitations

4.4.1 Test Cluster Size

All experiments were conducted on clusters of EC2 `c7i.flex.large` instances (2 vCPU, 4 GB RAM each), using a 4-node and a 6-node configuration.While the two-cluster comparison provides a scaling data point, both configurations remain small and relatively homogeneous, and several results are artifacts of this scale. The read-heavy uniform regression on the 6-node cluster, and the occasional spurious placement epoch bumps on the 4-node cluster are all phenomena that would likely behave differently on a larger, heterogeneous deployment. The placement scoring function is designed to differentiate nodes based on runtime metrics, but when all nodes run on identical hardware in the same availability zone and receive similar load, scores converge and the algorithm has limited differentiation signal to act on. The micro-benchmark experiments with artificially injected latency and bandwidth saturation were specifically designed to compensate for this, but they do not substitute for a heterogeneous production environment spanning multiple instance types or availability zones, which remains the natural next validation step.

4.4.2 Absence of Macro Topology Awareness

While the placement algorithm incorporates network latency and bandwidth metrics, it does not explicitly model the cluster topology. For example, it does not recognize that certain nodes are co-located in the same availability zone or rack, which could have implications for failure domains and correlated performance degradation. A more advanced design might incorporate topology labels into the scoring function

or enforce anti-affinity rules to ensure that replicas are distributed across failure domains.

4.4.3 Metrics Collection Overhead

The placement scoring pipeline relies on periodic RTT probing between nodes and local collection of CPU, memory, disk, and network utilization via `gopsutil`. The overhead of this collection is not measured or reported in the evaluation. In a production system with tight latency budgets, the interference between metrics collection goroutines and the data path goroutines could be non-trivial, particularly for disk I/O metrics which may themselves trigger I/O operations on some platforms.

4.4.4 Incorporating Topology-Aware Placement

Integrating explicit topology information into the placement algorithm would allow it to make more informed decisions about replica distribution. For example, the scoring function could include a penalty for placing multiple replicas of the same vnode on nodes that share a failure domain, or it could prioritize placing primaries on nodes with lower latency to the majority of clients. Which gives rise to a more complex but potentially more robust placement strategy that accounts for both performance and reliability considerations.

4.5 Conclusion

Taken together, the design trade-offs, result interpretations, and limitations discussed in this chapter point to a consistent picture: the core architectural choices of this prototype are sound for the scale at which they were tested, but several of them: the EventLoop's single-threaded write path, the absence of replica-aware reads, the static placement weights, were deliberate simplifications rather than final answers. The evaluation demonstrates that network-aware placement is a viable foundation for this class of system, while also making clear where a production-grade version would need to diverge from this prototype. These open questions, along with concrete directions for addressing them, are taken up in the concluding chapter.

General Conclusion & Future Work

General Conclusion

This thesis presented the design, implementation, and evaluation of a distributed network-aware KV database that integrates runtime network and resource awareness into vnode placement. By combining adaptive placement with gossip-based monitoring and Raft-backed coordination, the proposed system improves load distribution while maintaining scalability and fault tolerance. The experimental results demonstrate the practicality of this approach and provide a solid foundation for future improvments, including adaptive placement tuning, read-aware rebalancing policies, higher per-node concurrency, more extensive failure evaluation, and validation on larger heterogeneous clusters.

Future Work

Adaptive Placement Tuning Replace the fixed placement weights with an adaptive optimization technique, such as reinforcement learning or Bayesian optimization, to automatically tune placement decisions for different deployment environments.

Read-Aware Rebalancing Incorporate cluster-wide read-rate metrics into the placement logic to defer rebalancing during periods of high read activity, reducing the impact of control-plane operations on request latency.

Partitioned EventLoop Partition the EventLoop across vnode ranges to increase per-node write throughput by exploiting multi-core processors while preserving correctness during vnode migration.

Failure Injection Evaluation Extend the evaluation using controlled node failures and network partitions to measure detection time, replica promotion latency, and service availability under adverse conditions.

Larger Heterogeneous Clusters Evaluate the system on larger clusters with heterogeneous hardware to further validate the placement algorithm and assess its scalability in more realistic scenarios.

References

[1] Hayk Simonyan. *Database Replication & Sharding Explained*. 2024. URL: https://hayksimonyan.substack.com/p/database-replication-and-sharding.

[2] CoVaib DeepLearn. *System design concept: Database Sharding and Partitioning*. 2025. URL: https://medium.com/@shivanimutke2501/database-sharding-and-partitioning-e3c173746b55.

[3] James C. Corbett et al. "Spanner: Google's Globally Distributed Database." In: *ACM Trans. Comput. Syst.* 31.3 (Aug. 2013). ISSN: 0734-2071. DOI: 10.1145/2491245. URL: https://doi.org/10.1145/2491245.

[4] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data." In: *ACM Trans. Comput. Syst.* 26.2 (June 2008). ISSN: 0734-2071. DOI: 10.1145/1365815.1365816. URL: https://doi.org/10.1145/1365815.1365816.

[5] David Thaler and Chinya V Ravishankar. "A name-based mapping scheme for rendezvous." In: *Electrical Engineering and Computer Science Department, The University of Michigan, Ann Arbor, Michigan* (1996), pp. 1–31.

[6] Yongjie Guan. "Local Rendezvous Hashing: Bounded Loads and Minimal Churn via Cache-Local Candidates." In: *arXiv preprint arXiv:2512.23434* (2025).

[7] Mankamana Mishra et al. "Weighted HRW and its applications." In: 2019. URL: https://api.semanticscholar.org/CorpusID:131909750.

[8] David Karger et al. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web." In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 1997, pp. 654–663.

[9] Pratima Upadhyay. *Consistent Hashing*. 2022. URL: https://www.linkedin.com/pulse/consistent-hashing-pratima-upadhyay/.

[10] Giuseppe DeCandia et al. "Dynamo: amazon's highly available key-value store." In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: Association for Computing Machinery, 2007, pp. 205–220. ISBN: 9781595935915. DOI: 10.1145/1294261.1294281. URL: https://doi.org/10.1145/1294261.1294281.

[11] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system." In: *ACM SIGOPS operating systems review* 44.2 (2010), pp. 35–40.

[12] Alexander Patino. *What is synchronous replication?* URL: https://aerospike.com/blog/what-is-synchronous-replication/.

[13] Riak. *Riak KV: flexible data model for many unstructured use cases including user, session and profile data*. URL: https://riak.com/posts/technical/why-riak/products/riak-kv/.

[14] Eric Brewer. "CAP twelve years later: How the "rules" have changed." In: *Computer* 45.2 (2012), pp. 23–29. URL: https://ucsb.edu.

[15] Hazelcast. *What is the CAP Theorem?* URL: https://hazelcast.com/foundations/distributed-computing/cap-theorem/.

[16] Leslie Lamport. "The part-time parliament." In: *Concurrency: The Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 277–317. ISBN: 9781450372701. URL: https://doi.org/10.1145/3335772.3335939.

[17] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm." In: *2014 USENIX annual technical conference (USENIX ATC 14)*. 2014, pp. 305–319.

[18] Aymen Benyoub. *Source code for the project*. 2026. URL: https://github.com/AymenBenyoub/nawst.

[19] ITU Online Editorial Team. *What Is the Event Loop? A Practical Guide*. 2024. URL: https://www.ituonline.com/tech-definitions/what-is-event-loop/.

[20] Daniel Lemire. *A fast alternative to the modulo reduction*. 2016. URL: https://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/.

Abstract

Traditional distributed key-value stores use consistent hashing with virtual nodes (vnodes) but treat the network as a static black box. This leads to data skew and ignores real-time latency and resource variations. This thesis presents a network-aware distributed key-value store that dynamically adapts data placement. By decoupling control and data planes, the system monitors hardware metrics and RTTs to compute a deterministic cost function for vnode reallocation. Results show that the proposed topology eliminates hotspots and improves load balancing without degrading core storage performance.

Keywords: Distributed Key-Value Store; Network Awareness; Dynamic Data Placement; Virtual Nodes; Consistent Hashing

ملخص

تعتمد قواعد بيانات قيم المفاتيح الموزعة التقليدية على التجزئة المتسقة مع العقد الافتراضية، لكنها تتعامل مع الشبكة كصندوق أسود ثابت. يؤدي ذلك إلى انحراف البيانات وتجاهل تغيرات زمن الانتقال وحدود الموارد. تقدم هذه الأطروحة نظامًا موزعًا واعيًا بالشبكة يُكيّف موضع البيانات ديناميكيًا. من خلال فصل طبقة التحكم عن البيانات، يتابع النظام مقاييس العتاد وزمن الرحلة (RTT) لحساب دالة تكلفة حتمية لإعادة توزيع العقد الافتراضية. تُظهر النتائج أن هذه الطوبولوجيا تقضي على النقاط الساخنة وتوازن الحمل دون التأثير على أداء التخزين الأساسي.

الكلمات المفتاحية: قاعدة بيانات موزعة لقيم المفاتيح؛ الوعي بالشبكة؛ توزيع ديناميكي للبيانات؛ العقد الافتراضية؛ التجزئة المتسقة

Résumé

Les bases de données clés-valeurs distribuées traditionnelles s'appuient sur le hachage cohérent avec des nœuds virtuels, mais traitent le réseau comme une boîte noire statique. Cela cause des déséquilibres et ignore les variations de latence. Cette thèse propose un système clé-valeur distribué sensible au réseau qui adapte dynamiquement le placement des données. En séparant les plans de contrôle et de données, le système utilise métriques matérielles et RTTs pour calculer un score de coût déterministe. L'évaluation montre que cette approche élimine les points chauds et équilibre la charge sans dégrader les performances de stockage.

Mots-clés: Base de données clé-valeur distribuée; Sensibilité au réseau; Placement dynamique; Nœuds virtuels; Hachage cohérent