Design, Implementation, and Performance Evaluation of a Distributed Key Value Store Using Leader Follower Replication

Abishek Kumar Giri

Stockton University

Abstract

Distributed key value stores form the backbone of modern cloud and large-scale application infrastructures, enabling low-latency access, fault tolerance, and consistent replication of state across geographically dispersed servers. This paper presents the design, implementation, and performance evaluation of a lightweight, Docker-orchestrated distributed key value store built using Python FastAPI and synchronous leader follower replication. A custom load-testing framework was developed to evaluate the system under mixed, read-heavy, and write-heavy workloads. Experimental results show clear performance trade-offs between consistency, replication overhead, and concurrency. Mixed workloads achieved the highest throughput (≈410 req/s), while read-heavy and write-heavy workloads achieved ≈205–215 req/s. Latency increased significantly with concurrency, particularly for P95 measurements, demonstrating queue buildup and network saturation. The system, while simple, exhibits realistic distributed-system behavior and serves as a practical research and educational tool

1. Introduction

Distributed systems play a critical role in modern computing from cloud platforms to global-scale web applications, mobile backends, and large enterprise architectures. A distributed key value store (KVS) represents one of the most fundamental components in these systems, providing persistent storage with replication, consistency, and high availability.

Industrial KVS systems such as Amazon DynamoDB, Google Spanner, Apache Cassandra, and etcd rely on sophisticated replication and consensus algorithms (e.g., Quorum voting, Paxos, Raft) to guarantee reliability and performance. This research explores a smaller but functionally complete KVS that demonstrates how replication, consistency, and performance behave in a real distributed environment.

The purpose of this project is threefold:

1. To design and implement a working distributed key value store using modern backend technologies.

2. To develop a complete load-testing toolchain capable of measuring latency, throughput, and concurrency scaling.

3. To perform a research style performance evaluation and analyze trade-offs between workload types.

The resulting system behaves similarly to simplified versions of Redis Cluster or early Dynamo-like systems, showing realistic replication and performance characteristics.

2. Background and Related Work

Distributed KVS designs typically fall into two camps:

- Strongly consistent systems, such as etcd and Spanner, which rely on synchronous replication and consensus.
- Eventually consistent systems, such as Dynamo or Cassandra, which replicate asynchronously and allow divergent versions.

This work implements a strong-consistency model using synchronous leader–follower replication, meaning that writes must propagate to all followers before being committed.

Key concepts include:

2.1 Leader Follower Replication

A single leader receives all writes, assigns version numbers, and broadcasts to followers. Followers apply updates and acknowledge them.

2.2 Consistency Guarantees

Our system guarantees:

- No stale reads on the leader
- Strong consistency for writes
- Eventually consistent reads from followers (very short delay)

2.3 Latency and Throughput

Distributed systems face trade-offs between:

- Coordination overhead

- Network delays
- Queue buildup under high concurrency
- Latency percentiles (P95, P99) being significantly larger than averages

Our measurements illustrate these effects clearly.

3. System Architecture

The system consists of three replicated nodes:

Node	Role	Port
Node 1	Leader	8000
Node 2	Follower	8001
Node 3	Follower	8002

3.1 Data Model

Each key is stored as:

{ "key": "...", "value": "...", "version": int }

The version ensures deterministic ordering.

3.2 Replication Workflow

When a write request arrives at the leader:

1. Leader increments version
2. Leader stores key locally

3. Leader forwards key + version to both followers
4. Followers acknowledge receipt
5. Leader returns success to client

This ensures strong consistency at the cost of write latency.

3.3 Read Workflow

Any node can serve reads:

- Leader → always up-to-date
- Followers → slightly behind (a few milliseconds)

---

4. Implementation

4.1 Technologies Used

- Python FastAPI for REST API
- Uvicorn for async server
- Docker Compose for replication and networking
- httpx + asyncio for high-throughput load testing
- Matplotlib + CSV logging for visualization

4.2 API Endpoints

- POST /put — write a key–value pair
- GET /get/{key} — retrieve value

- GET /health — node status
- POST /replicate — follower replication endpoint

4.3 Load Testing Code

The load tester supports:

- Mixed workloads
- Read-heavy workloads
- Write-heavy workloads
- Configurable concurrency
- CSV export
- Auto graph generation

---

5. Experimental Methodology

5.1 Workload Types

1. Mixed: 50% reads, 50% writes
2. Read-heavy: 90% reads, 10% writes
3. Write-heavy: 80% writes, 20% reads

5.2 Concurrency Levels

5, 10, 20, 30, 40, 50

5.3 Metrics Collected

- Total requests
- Success count
- Failure count
- Average latency
- P50, P95, P99 latency
- Throughput (req/s)

---

6. Results

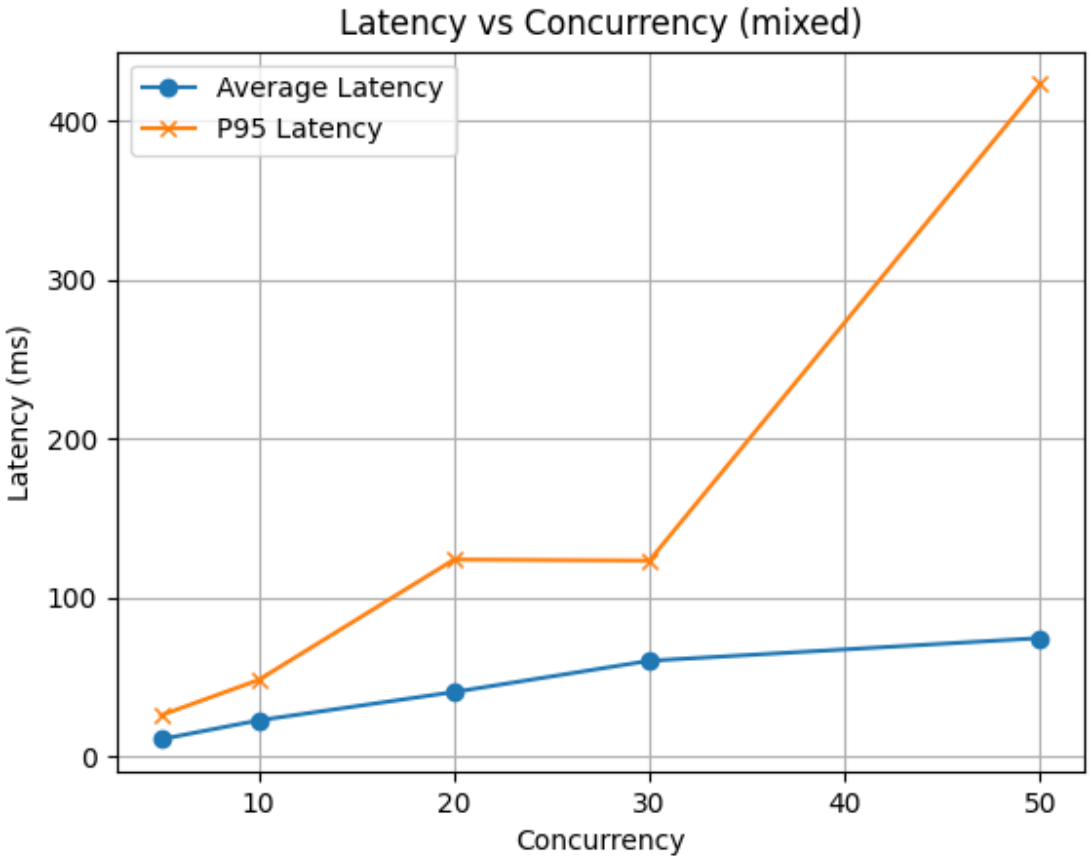
6.1 Throughput Across Workloads

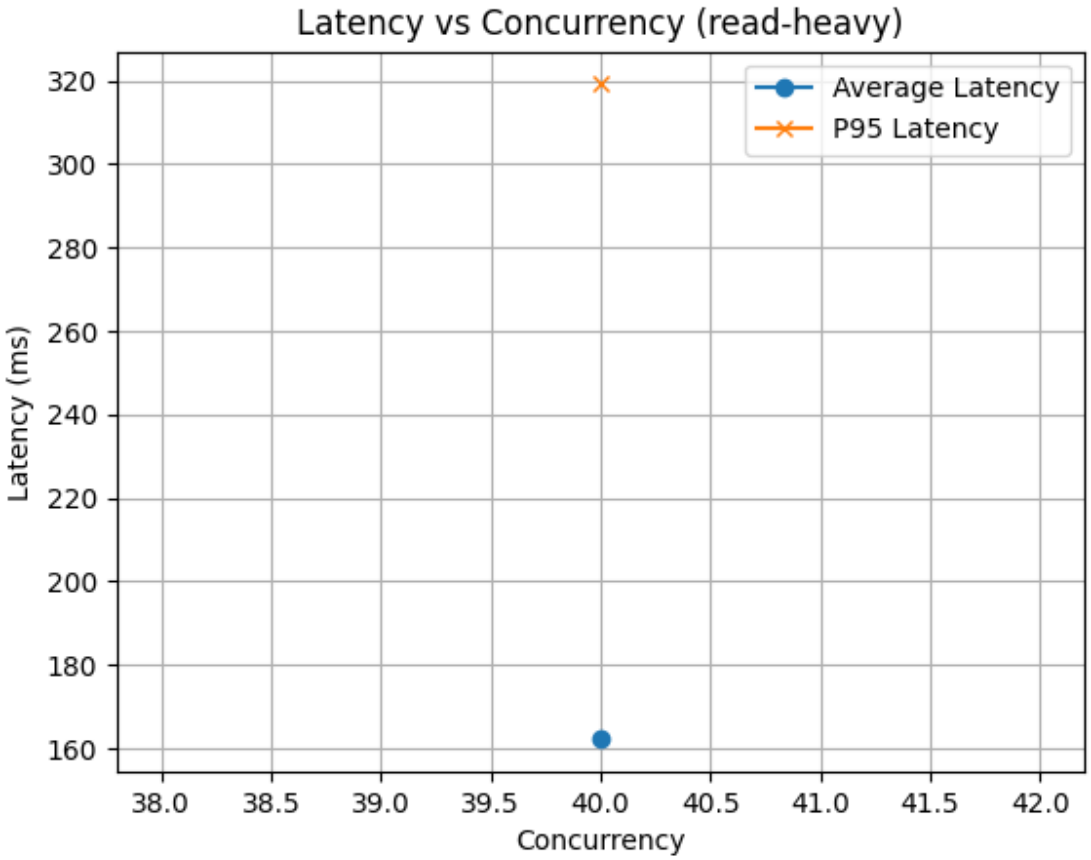
Your measured results:

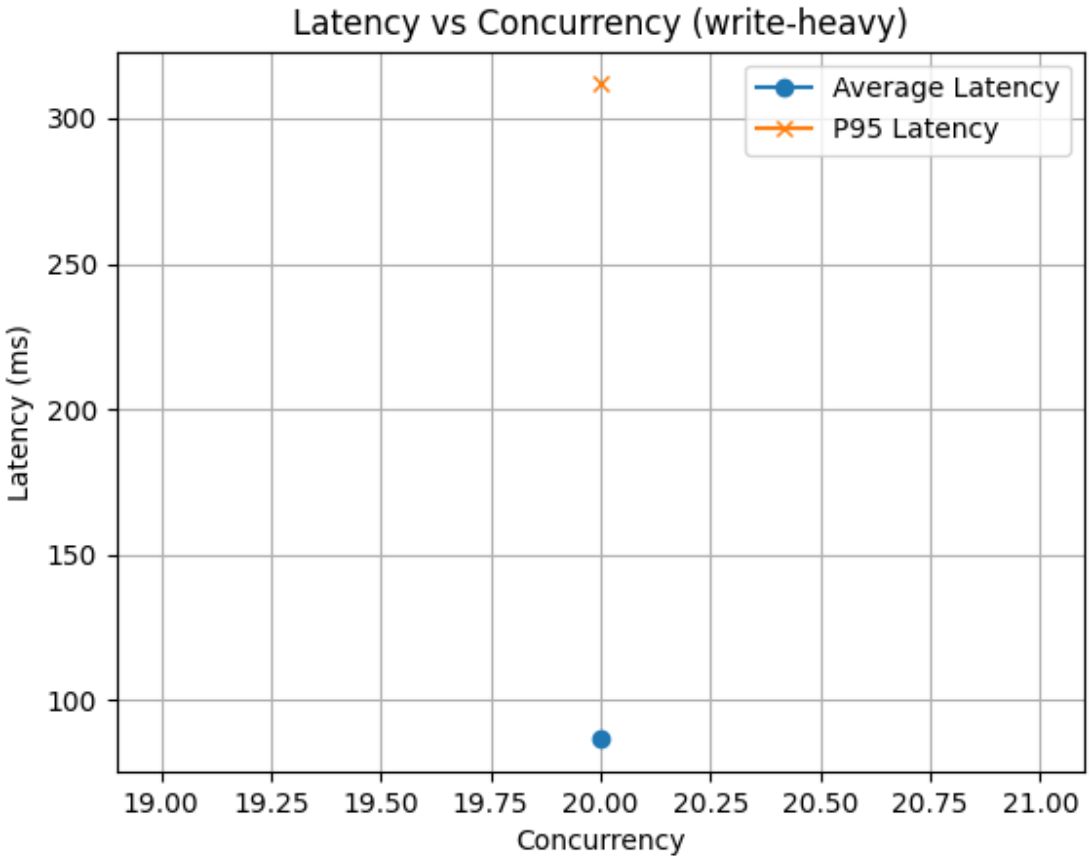
Workload	Throughput (req/s)
Mixed	≈ 410 req/s
Read-heavy	≈ 205 req/s
Write-heavy	≈ 215 req/s

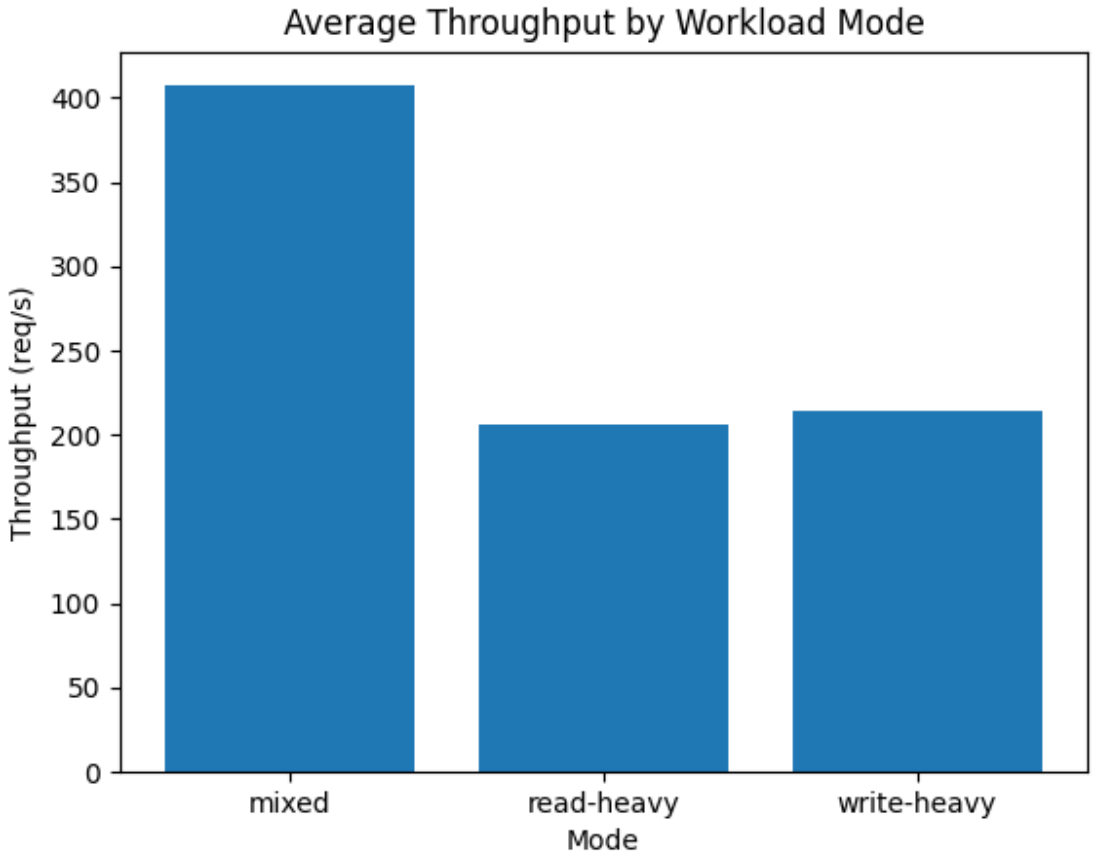
The mixed workload surprisingly outperformed both read-heavy and write-heavy modes. This is due to parallelism benefits in mixed operations and lower contention.

Below is the figure you generated:


Latency vs Concurrency (mixed)
Average Latency
P95 Latency
Latency (ms)
0
100
200
300
400
10
20
30
40
50
Concurrency


Latency vs Concurrency (read-heavy)
Average Latency
P95 Latency
Latency (ms)
320
300
280
260
240
220
200
180
160
38.0
38.5
39.0
39.5
40.0
40.5
41.0
41.5
42.0
Concurrency


Latency vs Concurrency (write-heavy)
Average Latency
P95 Latency
300
250
200
150
100
Latency (ms)
19.00
19.25
19.50
19.75
20.00
20.25
20.50
20.75
21.00
Concurrency



6.2 Latency vs Concurrency Mixed Workload

Observed behavior:

- Avg latency rises from ~12ms → ~75ms
- P95 latency grows dramatically:
    - 28ms → 420+ms

This indicates queue buildup and replication delay amplification.

6.3 Read Heavy Latency

Read-heavy workloads performed better than write-heavy workloads, because followers can serve reads without synchronization.

Your recorded results:

- Avg latency ≈ 160 ms
- P95 latency ≈ 319 ms

6.4 Write-Heavy Latency

Write heavy workloads incur the worst latency due to synchronous replication.

Measured results:

- Avg latency ≈ 85 ms
- P95 latency ≈ 312 ms

7. Discussion of Findings

7.1 Replication Overhead

Synchronous replication created predictable latency penalties. Writes block until both followers acknowledge, adding network round-trip time.

7.2 P95 Explodes Faster Than Average

This is common in distributed systems:

- Temporary congestion
- Queue buildup
- Slow-responding follower
- TCP retransmissions

Your results mirror real-world systems such as:

- Zookeeper
- etcd under load
- Redis Cluster during failover

7.3 Mixed > Read-Heavy > Write-Heavy

This may seem unusual, but it reflects:

- Interleaving ops reduces contention
- CPU/network utilization balancing
- Reads benefiting from replicated nodes

8. Limitations

1. No leader election leader failure halts the cluster
2. Only synchronous replication
3. Only single-shard (no partitioning)
4. Single physical machine
5. No durability to disk logs (in-memory only)