Distributed Key-Value Store Report

Team Member : Rovin Singh (24290010) , Naveen Tiwari(24210068)

Implementation Discussion

We have built a distributed key-value store designed for scalability, fault tolerance, and configurable consistency. Below is an overview of the implementation details for both the client and server.

Server Implementation

- Uses **consistent hashing** to distribute keys across nodes, reducing data movement when nodes join or leave.
- Employs a **Hybrid Logical Clock (HLC)** for timestamp-based operation ordering.
- Supports **replication** with a configurable factor, using **quorum-based writes** for strong consistency and **read repair** for eventual consistency on reads.
- Includes **Write-Ahead Logging (WAL)** for durability, **snapshots** for compaction, and **reconnection logic** for fault tolerance.

Client Implementation

- Connects to multiple servers, using **consistent hashing** to route Get and Put requests.
- Falls back to querying all servers if the **ring state** is unavailable.
- Handles **failures** with retries and reconnection attempts.

Consistency Guarantees

- **Writes:** Eventual consistency via **quorum writes**, which succeed only if a majority of replicas acknowledge.
- **Reads:** Eventual consistency, with **read repair** fixing stale data in the background.**Last Write Wins (LWW)** - When conflicts occur between multiple writes to the same key, the system resolves them by selecting the write with the highest HLC timestamp.

Protocol Specification

The client and server communicate using **gRPC**, defined in a kvstore.proto file. Below are the key services and methods:

KVStore Service

- Get(GetRequest) returns (GetResponse): Fetches a value by key.
- Put(PutRequest) returns (PutResponse): Stores a key-value pair.

NodeInternal Service (for server-to-server communication)

- Replicate(ReplicateRequest) returns (ReplicateResponse): Replicates data across nodes.
- AddNode(AddNodeRequest) returns (Empty): Shares new node info via gossip.
- Heartbeat(Ping) returns (Pong): Monitors node health.
- GetRingState(RingStateRequest) returns (RingStateResponse): Retrieves the current ring state.

Key Message Formats

- **GetRequest:** { string key }
- **GetResponse:** { string value; uint64 timestamp; bool exists }
- **PutRequest:** { string key; string value }
- **PutResponse:** { string old_value; bool had_old_value }
- **ReplicateRequest:** { string key; string value; uint64 timestamp }
- **ReplicateResponse:** { bool success; string error }

Performance Benchmark Tests

Our benchmarks evaluated system performance across varying conditions:

- **Workload Patterns:** Tested read-heavy (80%), write-heavy, and mixed workloads
- **Access Patterns:** Compared uniform vs. Zipfian (hot/cold) key distributions
- **Deployment Scenarios:** Evaluated single-client, multi-client, and failure conditions
- **Data Sizes:** Measured performance with 1KB to 1MB values

Results: Zipfian distribution outperformed uniform access (5,940 vs. 3,715 ops/sec for read-heavy workloads). Multi-client testing demonstrated excellent scalability (22,687 ops/sec), with minimal replication latency (0.199ms).

Storage Subsystem Tests

We validated core storage functionality through targeted tests:

- **Basic Operations:** Verified Store/Get/Delete correctness
- **Consistency:** Used timestamp ordering tests to validate operation sequencing
- **Durability:** Tested WAL recovery and snapshotting for persistence
- **Concurrency:** Ran 35 concurrent goroutines to expose race conditions
- **Fault Handling:** Simulated various corruption scenarios

Results: All 16 tests passed with high performance under stress (310,000 ops/sec). The system demonstrated excellent recovery capabilities (95% recovery during corruption) and robust concurrency handling.

Correctness Tests

We focused on distributed systems properties:

- **Basic Operations:** Verified Put/Get functionality across nodes
- **Consistency:** Ensured eventual consistency across the cluster
- **Fault Tolerance:** Confirmed operation during node failures
- **Self-Healing:** Validated read repair for rejoining nodes
- **Test Infrastructure:** Developed automated harness for server lifecycle management

Results: Tests confirmed strong eventual consistency, uninterrupted operation with node failures, and effective self-healing through read repair. The client retry mechanisms successfully handled transient failures during node transitions.

System Model and Assumptions

The system model defines the environment and operational constraints under which the distributed key-value store operates. The following assumptions are made:

Network Assumptions

- **No Network Partitions**
- **Reliable Communication**

Node Assumptions: Persistent Disk Storage: Each node has access to persistent storage (e.g., a disk) to store data durably. This ensures that data remains available even after a node restarts. **Crash-Stop Failures**: Nodes may fail by crashing (stopping entirely) but do not exhibit malicious or unpredictable behavior (e.g., Byzantine failures). This assumption simplifies fault tolerance strategies.

Consistency Model: Eventual Consistency for Reads: If no updates occur to a key, all reads will eventually return the same value, prioritizing availability over immediate consistency. **Strong Consistency for Writes with Asynchronous Replication:** Write operations are acknowledged to clients after being committed to the primary node but before replication completes. Replication to secondary nodes happens asynchronously in the background.Strong consistency is maintained despite asynchronous replication because: Each key has a deterministic primary node (via consistent hashing) . All writes to a key go through its primary node. Reads are directed to the primary node or follow the consistent hash ring.The no-partition assumption ensures clients can always reach the primary

Data Model: Key-Value Pairs: Data is stored as key-value pairs, where keys are unique strings and values are arbitrary strings.**Time-to-Live (TTL)**: Keys can optionally have a TTL, after which they are automatically removed from the store.

Cluster Management: Dynamic Membership: Nodes can join or leave the cluster dynamically, with the system rebalancing data accordingly. **Gossip Protocol**: Cluster membership changes are propagated using a gossip protocol, ensuring all nodes maintain an up-to-date view of the cluster.