SLIIT UNI
THE KNOWLEDGE UNIVERSITY

Distributed Storage System

Final Report

SE2062 - Distributed Systems

T S D De Zoysa - IT23669062

L G A I De Silva - IT23632028

D S Wickramarachchi - IT23736450

D T D Wijetunga - IT23614062

Table Of Content

Requirements & objectives

Design an architecture for a distributed storage system that includes:

- ZooKeeper as a coordinator for node membership, leader discovery/election (the leader election can be specific to a Raft-like leader election behavior appropriate for the assignment), and a client discovery endpoint.

- Storage nodes:
    - Multi-master writes, with asynchronous replication (eventual consistency).
    - One passive cold-standby that can consume logs/snapshots, but can only promote to be serving traffic upon receiving a command.
    - Supports heartbeats to ZooKeeper for liveness detection.

- Recovery consists of promoting standby and replaying logs.

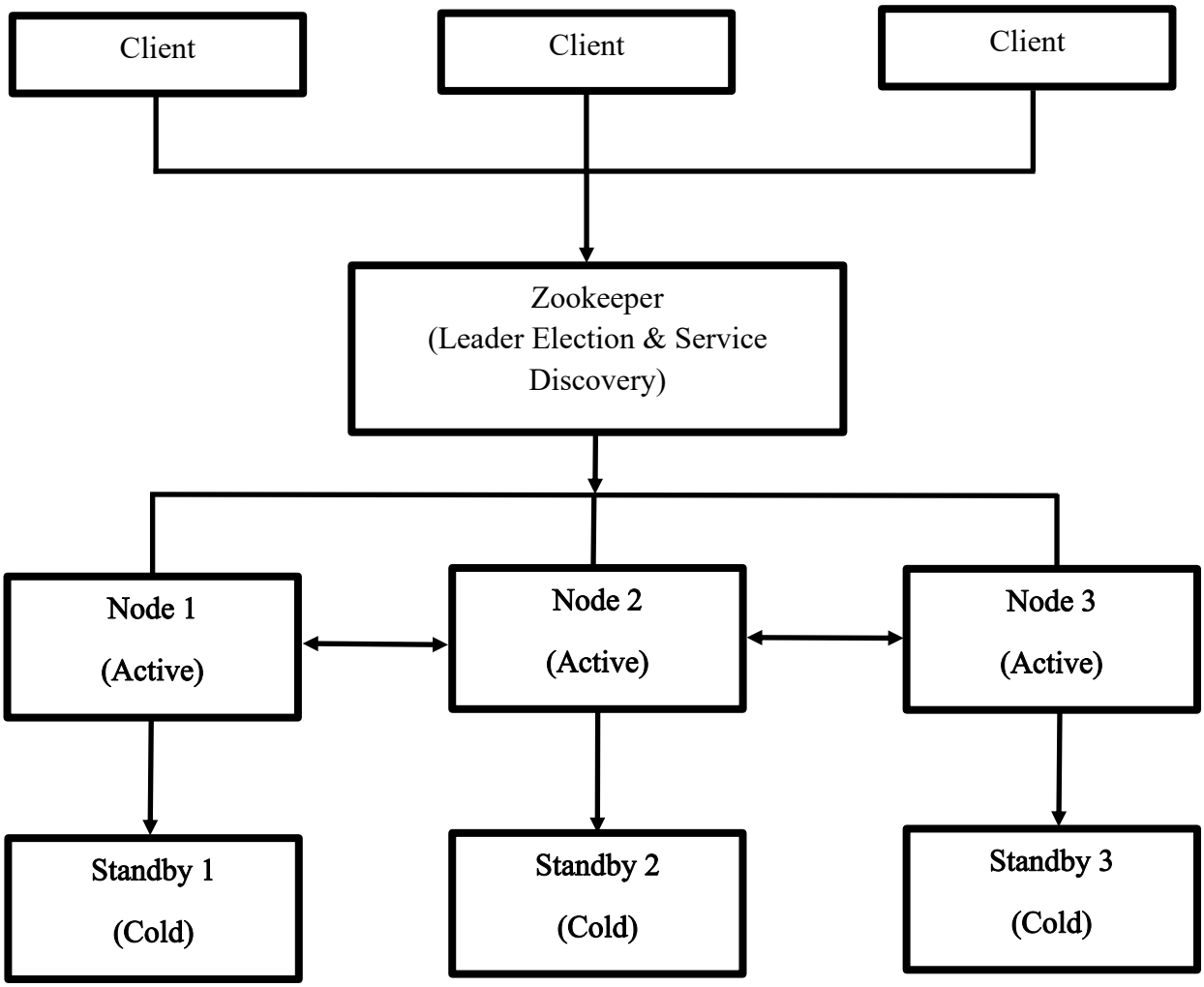
- Concurrency control uses timestamps synchronized by NTP (the ability to toggle simulated clock skew for tests) and deterministic last writable wins (LWW) using timestamp, fall back to node-id to tie-break.

- HTTP API as follows: PUT /key, GET /key, DELETE /key, POST /sync (internal use).

- Persistence: append-only replication logs on Disk, a simple JSON key-value DB.

- Testing: unit & integration tests, scripts to inject failures, a Docker Compose to run 3 nodes + 1 cold-standby + ZooKeeper.

High-level architecture



Component Design

1. **ZooKeeper (Coordinator / Discovery)**

   - Utilizes the official Docker image for ZooKeeper during deployment.

   - Keeps a record of storage nodes with ephemeral znodes (heartbeat updates).

   - Provides an http endpoint (simple light-weight wrapper service included in zookeeper/) to answer "who is the current leader?" - the leader is the storage node with the highest elected leadership score computed by using ZooKeeper ephemeral znodes + stored metadata. For the assignment, we show a Raft-like leader election using ZooKeeper as the central coordinator and using the ephemeral node sequence numbers: the node with the smallest sequence number is elected leader (ZooKeeper's ephemeral sequential znodes paradigm).

   - ZooKeeper steers promotion of cold-standby by writing to a control znode that standby nodes observe.

2. **Storage node (Java)**

   ❖ HTTP API for clients:

      - PUT /kv/{key} — JSON body {"value":"..."}; store a value with the current NTP timestamp.

      - GET /kv/{key} — returns {"value": "...", "timestamp": <ts>, "nodeId": <id>} or 404.

      - DELETE /kv/{key} — treat as tombstone with timestamp.

❖ Internal API:

- POST /sync — endpoint to receive a batch of replication entries: [{operation: "PUT"/"DELETE", key, value?, timestamp, nodeId, seq}].

❖ Persistence:

- An append-only replication log file named replication.log contains each local write as a JSON line with a unique incremental seq number.

- The key-value store is persisted in a JSON file named kvstore.json, which is updated each time an entry is applied (with writes applied in a way that preserves idempotence when timestamp+nodeId is the same).

- Snapshot files named snapshot-{seq}.json will be created periodically.

❖ Replication Worker:

- A background thread batches up new log entries for an in-memory cursor and asynchronously POSTs chunked to the peers' /sync endpoints.

- Retries will be attempted with exponential backoff in the event a peer cannot be reached to POST a new log entry.

❖ Standby Behavior:

- If the mode is standby, the node will accept the /sync command but will reject client PUT/DELETE/GET commands (or it will only return the read-only state on a GET after it is promoted to active mode). The design chooses to only allow GET if it is explicitly allowed.

- When a node is promoted to active mode (due to a ZooKeeper signal), it transitions from standby mode to active mode, services clients, and replays all logs/snapshots to catch up.

❖ Heartbeat Client:

- The heartbeat client will periodically update an ephemeral znode in ZooKeeper (or calls the ZooKeeper wrapper HTTP) with a heartbeat payload: {nodeId, ip, port, mode, lastAppliedSeq, timestamp}.

3. Client CLI

- A small command line utility, written in Java, that:
    - Queries the ZooKeeper discovery endpoint to get the leader address.
    - Performs PUT, GET, DELETE operations on the selected node or leader.

Data Model & Persistence

- Replication Entry


```
{
  "seq": 123,
  "op": "PUT",
  "key": "user:42:name",
  "value": "Alice",
  "timestamp": 169123,
  "nodeId": "node-A"
}
```


- An append-only log is referred to as replication.log (one JSON entry per line). Local seq numbers increase monotonically for each node.

- The key-value store is *kvstore.json*, mapping a key to *{value, timestamp, nodeId, deleted}*. Applying logic is idempotent; just compare incoming timestamp+nodeId to existing and apply if it is newer (last writer wins).

- A snapshot is a full *kvstore.json* saved as snapshot-*{lastSeq}.json* every N applied entries, truncating replication logs pre-snapshot.

Replication Design

- Multi-master:
  - Any node that is active can accept writes. On write:
    - Node gets NTP timestamp (or a local simulated version).
    - It appends a replication entry to its local replication.log.
    - It applies the entry locally to kvstore using LWW.
    - A background replication worker will batch recent entries and asynchronously POST /sync to peers.

- Asynchronous, eventual:
  - Writes are considered durable on the node that accepted them (after a local append and fsync can be configured).
  - Replication to peers is best-effort and asynchronous - the system prefers availability.

- Background worker:
  - Pads entries by sequence range and pushes them to each peer.
  - Peers will accept those batches and append to their local log, then apply.
  - The worker maintains per-peer replication state and, on reconnect, will redeliver missing ranges.

Cold Standby, Snapshotting, Compaction

❖ **Cold-standby:**

- Cold-standby status means a standby node has the mode=standby and has registered to serve standby workloads in ZooKeeper. It receives replication batches passed in via /sync, appends them to its replication.log and applies the replication logs towards kvstore. A standby node does not serve reads or writes from clients.

- A standby node uses lastAppliedSeq to stay close to current.

❖ **Promotion:**

- when ZooKeeper detects a failure in an active node (meaning it has missed heartbeats for longer than the set threshold), it will promote the standby node by updating a control znode in ZooKeeper. The standby observes the control znode update, flips its ZooKeeper mode=active and registers with ZooKeeper as an active node.

- On a node's promotion, the promoted node will guarantee that there is no divergence of state. It will pull any missing log files (as a PR mechanism) from its peers, and then replay it until its lastAppliedSeq >= its peers.

❖ **Snapshotting:**

- Raft supports SNAPSHOTs many nodes will periodically write out a snapshot (e.g., every N entries or after some time period). After writing snapshots, all log entries prior to this snapshot checkpoint may be truncated (e.g., compaction).

- Snapshots are also Provide enhancements for cold-standby catch-up and over for disk utilization.

Heartbeat & Membership, Failure Handling

- **Heartbeat:**
  - Every node sends out a heartbeat (ephemeral znode data, or HTTP wrapper) to ZooKeeper that contains the following information: lastAppliedSeq, uptime, mode.
  - ZooKeeper watcher observes the heartbeats and records which ones are gone.

- **Failure detection:**
  - When a node has not sent an update for its heartbeat beyond the HEARTBEAT_TIMEOUT (configurable), ZooKeeper will indicate to the cluster that the node is down and will begin the standby promotion process related to roles that were assigned to that node

- **Promotion and Recovery:**
  - Promotion includes:
    - ZooKeeper writing a promote command.
    - Standby changes to active, pulls missing logs/snapshot from peers, applies the logs, and then declares itself as active.

  - The failed node re-joins the cluster: after the container has been restarted, it re-registers itself, downloads the latest snapshot from an active peer (a pull), replays the logs since that snapshot, and transitions to standby or active depending on the policy for the cluster.

Concurrency and Conflict Resolution

❖ **Timing & Ordering:**

- When a write is performed, it is stamped with a timestamp that was derived from NTP, with a precision of milliseconds. As part of the local testing, we have made available the option to toggle simulated clock skew so that tests may inject clock skew.

- Primary ordering key: (timestamp, nodeId) with a deterministic comparison:
    - The higher timestamp always wins. If timestamps are equal, then the nodeId is compared lexicographically (or as a numeric node priority) as a tie-breaker.

❖ **Last Write Wins (LWW):**

- When a node applies a replication entry, it looks up the existing {value, timestamp, nodeId} for that particular key:
    - If the incoming (timestamp, nodeId) is greater than the existing {value, timestamp, nodeId}, then the node applies the new write.
    - If it is older, then the entry is discarded.
- Deletes are tombstones with timestamps, and follow the same last write rules.

❖ **Why do we use deterministic?**

- This guarantees that under any message arrival order all nodes will converge toward the same and consistent state, as it uses (timestamp, nodeId) to enforce total order.

Observability and logging

- The metrics endpoint: GET /metrics delivers JSON data containing the following attributes:
    - uptime, mode, lastAppliedSeq, replicationLagPerPeer (est.), lastTimestampReceived

- Structured logs (JSON lines) include entries for:
    - Local writes appended.
    - Outgoing replication batches (target peer, seq range).
    - Incoming /sync batches (source node, entries count).
    - Conflict resolution decisions (applied vs rejected with reason).
    - Heartbeat events, promotions.

- Monitoring:
    - In order to obtain some basic observability for demo purposes, you can use docker stats and /metrics.

Testing & Failure Injection

❖ Unit Testing (Java JUnit):

- LWW (Last Write Wins) conflict resolution with different timestamps/nodeIds.
- Log application idempotency (applying the same log entry twice).
- Snapshot creation and application.

❖ Integration testing (using Docker Compose):

- Run ZooKeeper + 3 active nodes + 1 standby.
- Concurrent write: two clients (the same key) writing to different nodes; after the replication window, assert that the final value after replication is the LWW result across all nodes.
- Failure test: kill one active node container within failures/kill_node.sh. Confirm that ZooKeeper promotes the standby node and that the cluster continues to accept writes. Confirm that when the killed node comes back online it rejoins as a standby node and has caught up.
- Partition simulation: *failures/simulate_partition.sh* drops connectivity between nodes A and B and tests divergence and eventual reconciliation when the partition heals.

❖ Clock skew tests:

- Simulate skew through environment variable SIMULATE_SKEW_MS on the nodes; perform writes with skewed clocks; verify with a deterministic resolution that the order of writes via NTP (or simulated) + tie-break determined past increments.

❖ Failure injection scripts:

- failures/kill_node.sh <node> — stops container via docker stop.
- failures/simulate_partition.sh <nodeA> <nodeB> — uses docker network disconnect to drop packets between two containers (Linux host required).

Deployment & Runbook

Sathindu needs to check this part so leave it for him

Results and Expected System Behavior

- Any write to an active node is accepted and applied at the active node right away.

- Within a configurable replication batch window of seconds, peers receive replication entries and apply them locally. They will eventually converge according to LWW rules.

- For killing an active node, the standby node will be promoted and begin serving client requests after catching up with the logs/snapshots. Clients may find the new leader by using ZooKeeper discovery.

- For a partition, it is possible for the two sides to diverge while partitioned, and when the partition is healed, replication will guarantee that the entries are merged deterministically based on the timestamp + node-id tie-break. Depending on the timestamps, if a conflicting write occurred, one will win as soon as possible (LWW). Deleted keys will follow the same LWW semantics.

Limitation & Future Work

❖ **Limitations**

- Although LWW makes conflict resolution easier, it loses some concurrent writes (last write wins) which may be inappropriate in some semantics of the application.

- It does not provide guarantees of linearizability or strong consistency (it sacrifices that for complexity and availability).

- ZooKeeper acts as the single source of the truth for membership; a more rigid Raft cluster or ectd may better be suited for a production configuration.

- There is no authentication/authorization in the prototype; therefore it is not suitable for production.

- Network partitioning is taken care of using LWW, but the partitioning only gets resolved when some complex object graph are merged which may not be appropriate.

- The snapshot transfer and log pull are rudimentary and have not been optimized for large datasets.

❖ **Future Work**

- Adding causal metadata (vector clocks) would help preserve causality and allow for user defined merge functions.

- Implement quorum-based writes (when configured) for a more consistent mode.

- Add TLS/authentication via signing requests for hardened production deployments.

- There should be no longer use of the ZooKeeper wrapper and should replace it with a native client to ZooKeeper to get the strongest semantics possible.

Conclusion

This project showcases a working demonstration of a distributed storage architecture focused on availability. The system achieves a fault-tolerant, eventually-consistent state through the use of ZooKeeper coordination of multi-master replication and cold-standby failover.