CS739 P4

LevelUpDB

A Cloud-Native version of LevelDB

Benita Kathleen Britto, Krishna Ganeriwal, Reetuparna Mukherjee, Shreyansh Sharma

1. Introduction

LevelUpDB is a cloud-native version of the key-value storage library LevelDB by Google. This project implements the following key specifications:

- LevelDB - Cloud key-value data storage.
- RAFT - to reach consensus.
- GRPC C++ - for RPC.
- Tunable Consistency - Strong and Eventual.
- Load balancer - to distribute load among participating nodes and maintain system state.
- Dynamic Resource Allocator - to support on demand addition or removal of nodes to the cluster.

This implementation is aimed to provide an available, scalable, durable system with tunable consistency of LevelDB. The tunable consistency helps trade performance for consistency and can hence be decided based on the client application.

2. Supported APIs

The following calls are supported by our cloud-native key-value data storage :

Operations	**Description**	**Return Value(s)**
get(key,consistencyLevel,quorum)	Reads value saved for key based on consistency level opted for and value for quorum	Returns the value(string) stored for the given key.
put(key,value)	Writes/Updates value for a given key.	Returns the response code indicating if the write is successful.
AddServer(kv_ip,raft_ip,lb_ip)	Adds a node (Follower) to the cluster	Returns the response code indicating if the addition is successful.
DeleteServer(raft_ip)	Removes node with given IP from the cluster	Returns the response code indicating if the deletion is successful.

3. Key Assumptions

We make the following assumptions:

.

- Put and Get APIs support only string data type.
- Batch Update/Creation is not supported.
- Put API is used for both KV Creation and Update.
- Load Balancer will always be up and has greater computing power than the rest of the nodes
- Sufficient memory for commit logs on each node.
- Cannot tolerate Byzantine Faults.
- Network partitions not supported
- Does not account for security.

4. System Design

4.1 Visibility Semantics

Writes are committed on logs of majority of nodes (N/2 + 1) before an acknowledgement is sent back to the client. When strong consistency is opted, read response is made available through the leader, whereas, when the consistency is eventual, then the load balancer requests all the nodes in the cluster (N) and waits for the first ‘R’ successful responses and sends back all the unique results to the client (if there are multiple values, then the client is expected to resolve the conflict).

4.2 Architecture

The project has 4 main components - **Load Balancer and Resource Allocator, Servers (consistency managed using RAFT), the Client and the key-value data storage library LevelDB.**

Load Balancer - Every server on joining the cluster registers itself with the load balancer using periodic heartbeats. Nodes do this using a bidirectional stream with load balancer as a keep-alive message and get to know the status of the overall system. Additionally, the load balancer also helps with routing the requests to the appropriate nodes based on the level of consistency opted by the client. For instance, during strong consistency - all reads and writes are routed to the leader node, whereas if the client opted for eventual consistency, then the read request would be made to all the ‘N’ nodes in the cluster and the load balancer would wait for ‘R’ successful responses (configured by the client) to respond back to the client. Moreover, the

leader node also asserts its leadership (registers status for that term) with the load balancer which is propagated to other nodes from here. Similarly, the leader learns about all its followers' IP from the load balancer, so it acts as a system state manager as well.

Had we only implemented strong consistency, the responsibility of the load balancer could have been shouldered by the master node. Adding eventual consistency to the system advocates the use of a loadbalancer which can maintain the state of the overall system and route requests accordingly, and to segregate this logic from the client library.

Resource Allocator - It is a process running on the nodes. It provides horizontal scalability to the client applications by allowing dynamic addition and removal of nodes to the cluster. Further, if too many nodes die at the same time (Not enough nodes alive for replication and election), it spawns new nodes automatically for reliability and availability, with no manual intervention needed.

Client - Client is dumb in this project and is agnostic of the state of the system. It just needs to know the address of the load balancer in its cluster to get started and make all requests. As mentioned above, the client is availed the procedures to add a node, remove a node, get key-value, set key-value, update key-value and finally configure the consistency level it desires between strong and eventual ('R' configurable reads supported, inspired by Dynamo DB). This allows the client to trade consistency for performance (based on the 'R' opted).

RAFT - The most critical part of the project, is the implementation of RAFT consensus algorithm. The system we have built supports replication and consensus helps making replicated systems fault tolerant. RAFT being a distributed consensus algorithm (easy to understand) helps solve this problem of getting multiple servers to agree on a shared state even in the face of failures.

LevelDB - We intended to provide a key-value store that is fast and supports an ordered mapping of string keys and string values. LevelDB by Google provides just that, thus, each server creates a LevelDB instance to replicate key-value pairs persistently.

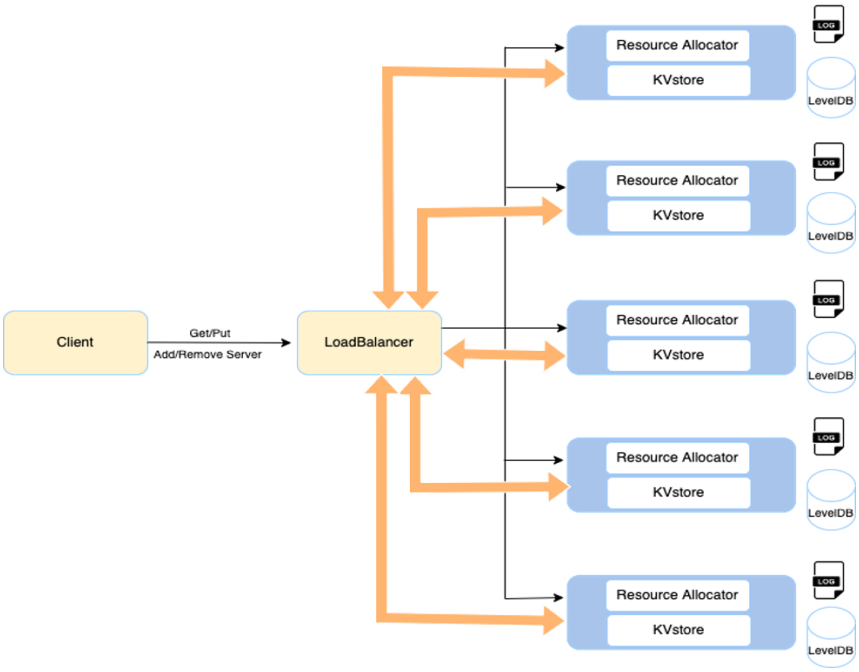

Fig. 1: System Design

4.3 Storage Structure

All writes at every node is stored both in-memory (unordered map) and on-disk (leveldb)

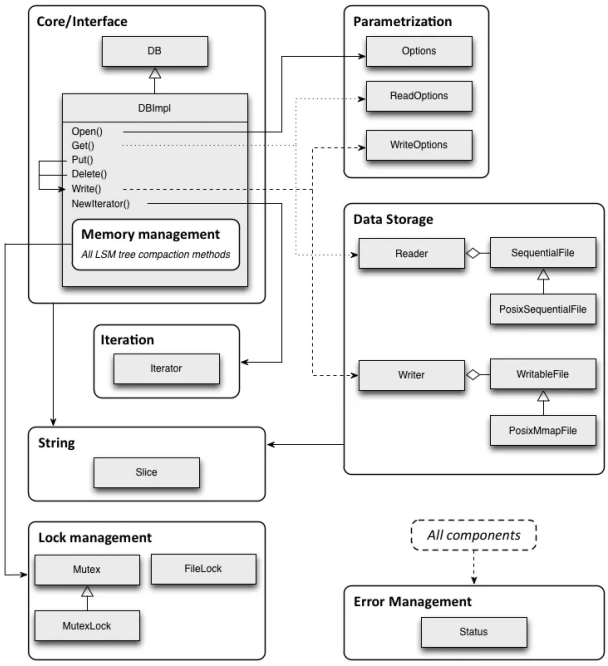

Fig. 2: LevelDB storage structure

- Key-Value Data Storage.
- Does not support SQL.
- No support for indexes.
- Supports Atomic operations.
- Supports bulk operations.
- Supports read-only snapshot.

Additionally, all writes are also stored as an unordered map in a volatile state in memory on each server to further facilitate fast reads.

4.4 Key Design Decisions

While architecting levelUpDB we made a few key design decisions which re listed as follows:

- **Read/Write -** The primary goal of this project is to create a cloud-native version of a distributed key-value store which supports replication and availability. Hence, to keep storage simple we have only supported string data types for keys and values to be stored.

- **Centralized System Monitor -** One of our key assumptions is that the load balancer never goes down (single server system for this project's scope). This server serves the following main purposes:
    - Interface between client application and the servers.
    - Source of liveness state monitor and address database for all servers.
    - Balances the load based on the consistency level opted by client.

- **Horizontal Scalability -** Supports dynamic addition of a new node or removal of an existing one from the RAFT cluster. Additionally, a key design feature includes automatically adding new nodes to the cluster if multiple existing nodes have failed and there isn't enough nodes alive (3 for this project to demonstrate RAFT).

- **Tunable Consistency -** RAFT inherently provides strong consistency, hence, our implementation by default is strongly consistent which means all read and write requests are made through the leader and the client is reverted back with a success when a successful replication/read consensus is reached.

  Further, we have designed our system in a way that, client applications can opt for a weaker consistency to trade it for a potentially better performance by choosing to wait for 'R' (configurable) reads and not the entire majority. This means the load balancer will contact all the live nodes for a read request but will wait only for 'R' successful responses and will relay all the unique responses back to the client for it to resolve conflicts (if any).

- **Write Locks -** Critical code including server logs, etc. are locked when written to, in order to avoid concurrent writes overwriting the same log index.

- **CA in the CAP theorem -** The system provides strong consistency and is highly available (replication,etc.) and we do not support partition in this implementation.
- **Homogeneity -** Current implementation of this project does not load any node more or less based on its computational specifications. Except leader/follower responsibilities, all nodes are loaded similarly (**except load balancer**).

4. Possible Read and Write Scenarios

Given RAFT being a well defined algorithm, there aren't any strange write scenarios that haven't been covered by the algorithm. The write process is hence, straight forward. To summarize the write in RAFT driven multi-server distributed storage:

Write Process:

Each write request made by the client is stored in the Leader's log. This log is then replicated to followers. The Leader node broadcasts AppendEntries RPCs to all other servers(Followers) to sync/match up their logs with the current Leader. The Leader keeps sending the RPCs until all the Followers safely replicate the new entry in their logs.

There is a concept of entry commit in the algorithm. When the majority of the servers in the cluster successfully commit an entry, the Leader also commits it in its log to mark successful replication. After the entry is committed, the leader applies the entry and responds back with the acknowledgement to the client. These entries are executed in the order they are received.

A peculiar case is if the Leader crashes, the logs may become inconsistent. Quoting the Raft paper : "*In Raft, the leader handles inconsistencies by forcing the followers' logs to duplicate its own. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log.*"

The Leader node will look for the last matched index number in the Leader and Follower, it will then overwrite any extra entries further that point(index number) with the new entries supplied by the Leader. This helps in Log matching the Follower with the Leader. The AppendEntries RPC will iteratively send the RPCs with reduced Index Numbers so that a match is found. When the match is found, the RPC succeeds.

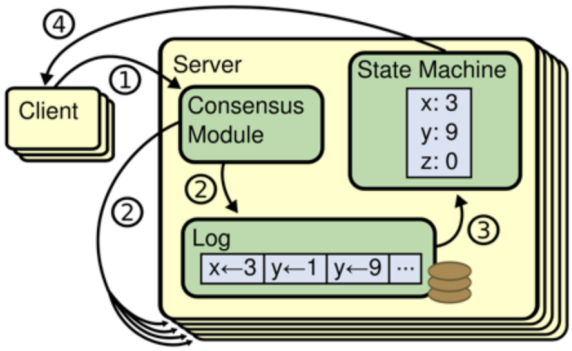

Fig. 3: RAFT Write Consensus

Read Process:

The tunable consistency feature in the project made read process a very interesting and important feature. It helps the client applications to tightly control the tradeoffs between availability, consistency and performance. The two different read options possible due to tunable consistency are:

1. Read from Leader
2. Read from 'N' nodes and wait for 'R' (configurable) successful responses (N >= R)

Case #1: Strong Consistency - Read from Leader

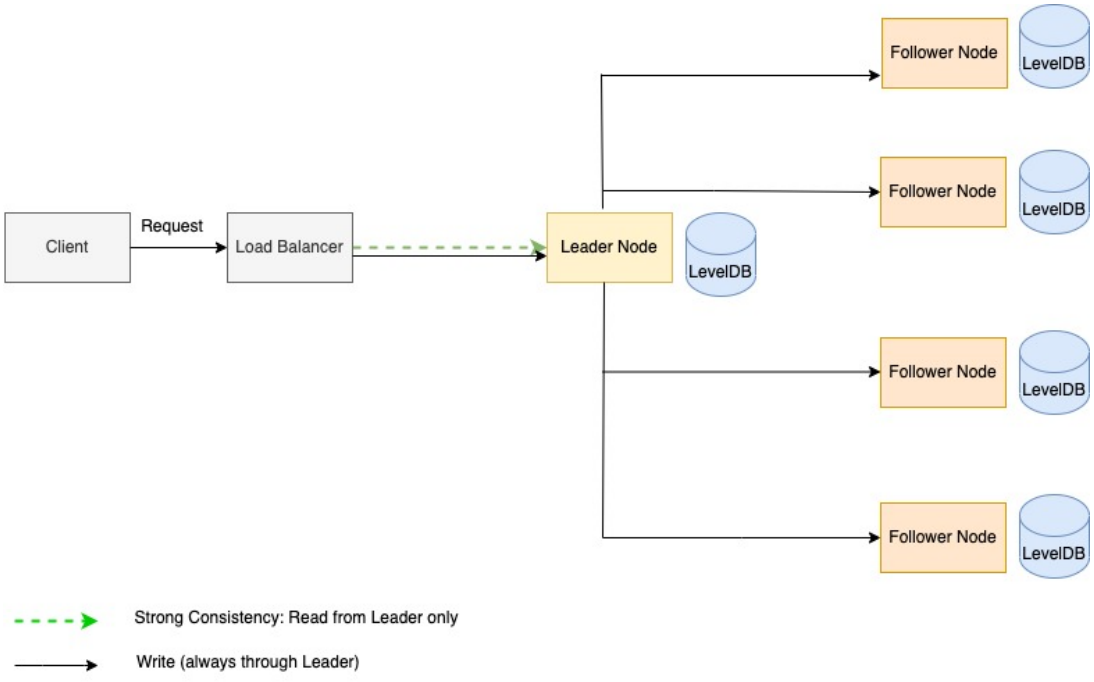

Fig. 4: Case #1 Strong Consistency - Read from Leader

Description: Strong Consistency opted by client

Implementation: The Load Balancer requests only the Leader for read requests (the Leader node could be overloaded).

Case #2: Eventual Consistency - Wait for ‘R’ successful responses (N >= R)

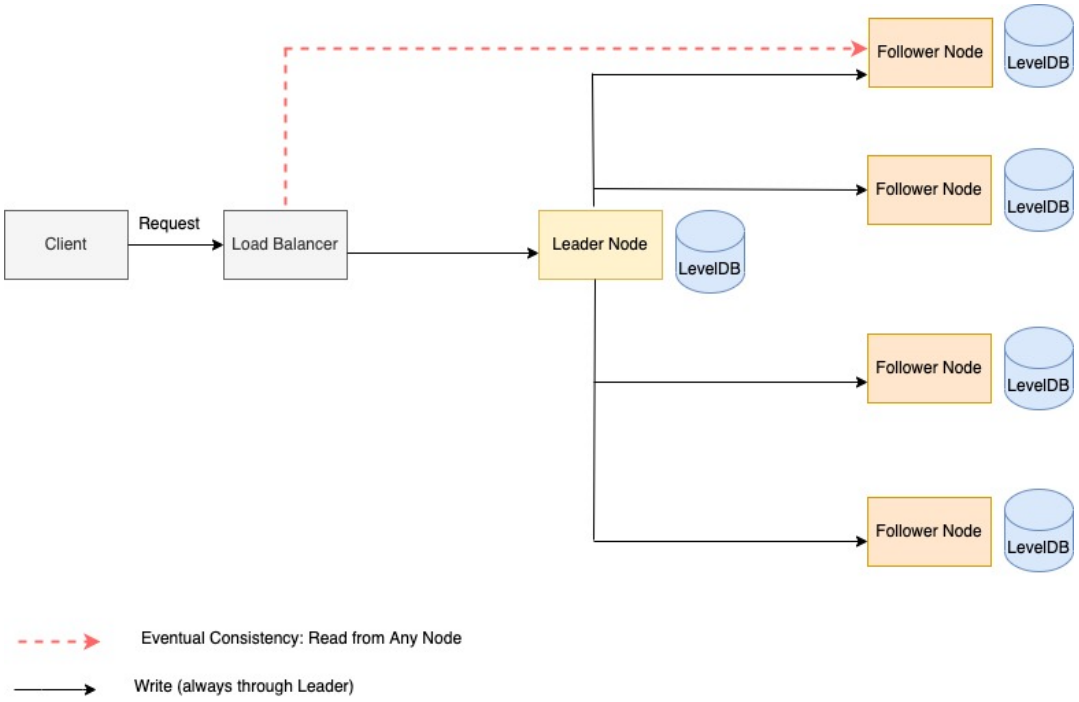

Fig. 5: Eventual Consistency - Wait for ‘R = 1’ successful responses (N >= R)

Description: Eventual Consistency - Read from ‘N’ nodes and wait for ‘R’ (configurable) successful responses (N >= R)

Implementation: The client should have configured ‘R’ with the Load Balancer. ‘R’ here should be strictly less than or equal to ‘N’ nodes in the cluster. ‘R’ should be configured based on what level of consistency and performance requirements might the application have. The Load Balancer contacts all the ‘N’ nodes in the cluster and waits for the first ‘R’ successful responses. Since, there might be some nodes that might not have the last write for that key, there is a possibility that there could be more than 1 unique responses for the same key, which is a conflict. The resolution of such conflicts in our implementation is left to the client application. The load balancer in this case would simply return all the unique values it received from the ‘R’ first responses.

5. Crash Recovery Protocol

Whenever a node crashes - Leader, Follower or Candidate and it comes up again the procedure in RAFT for recovery remains the same. If a follower or candidate crashes, then future RequestVote and AppendEntries RPCs sent to it will fail. Raft handles these failures by retrying indefinitely; if the crashed server restarts, then the RPC will complete successfully. If a server crashes after completing an RPC but before responding, then it will receive the same RPC again after it restarts. Raft RPCs are idempotent, so this causes no harm.

For example, if a follower receives an AppendEntries request that includes log entries already present in its log, it ignores those entries in the new request.

6. Failover

In RAFT based systems, when a follower node fails, then it might not affect the functioning of the system as long as the number of followers are still in majority for the leader and big enough to handle certain number of failures. Whereas, leader failover is basically the case of leader election in RAFT.

In order to maintain authority as a Leader of the cluster, the Leader node sends heartbeat to assert leadership to other Followers. A leader election takes place when a Follower node times out while waiting for a heartbeat from the Leader. At this point of time, the timed out node changes it state to Candidate, votes for itself and issues RequestVote RPC to establish majority and attempts to become the Leader. The election can go the following three ways:

- The Candidate node becomes the Leader by receiving the majority of votes from the cluster nodes. At this point of time, it updates its status to Leader and notifies the loadbalancer via AssertLeadership RPC. It then starts sending heartbeats (AppendEntries RPC) to notify other nodes.

- The Candidate node fails to receive the majority of votes and returns to Follower state. The term ends with no Leader.

- If the term number of the Candidate node requesting the votes is less than other Candidate nodes in the cluster, the AppendEntries RPC is rejected and other nodes retain their Candidate status. If the term number is greater, the Candidate node is elected as the new Leader.

To quote the RAFT paper to explain server timeouts: "*Raft uses randomized election timeouts to ensure that split votes are rare and that they are resolved quickly. To prevent split votes in the first place, election timeouts are chosen randomly from a fixed interval (e.g., 150–300ms). This spreads out the servers so that in most cases only a single server will time out; it wins the election and sends heartbeats before any other servers time out. The same mechanism is used to handle split votes. Each candidate restarts its randomized election timeout at the start of an election, and it waits for that timeout to elapse before starting the next election; this reduces the likelihood of another split vote in the new election.*"

Hence, even if a Leader node fails, there would be another Follower node which would escalate to become a Candidate and later a Leader.

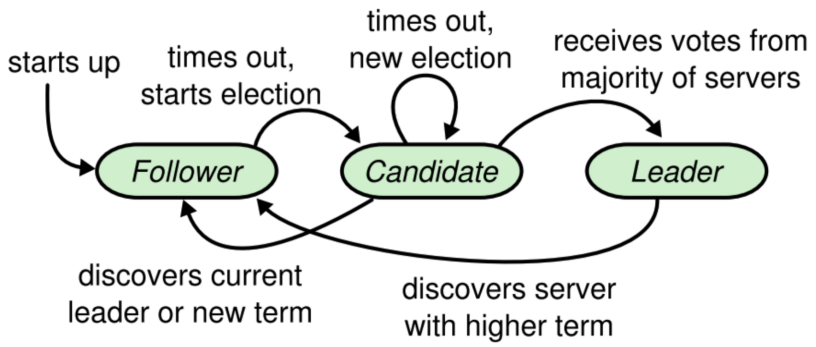

Fig. 7: RAFT Election

7. Durability

To ensure data logged remains non-volatile and exists even after a crash, our system persists some of the important parts on disk to maintain durability. The parts persisted on disk are:

- **currentTerm**: latest term the node has seen,
- **votedFor**: ID of the candidate that received vote in the current term.
- **Log[]** : log entries containing term and key-value pair.

8. Testing Strategy

Raft Correctness

We implemented Raft from scratch and so, we used the following scenarios to test the correctness of our implementation.

- To test the **correctness of leader election** (also shown in Demo 5), we prefill the replicated log files of Server1 and Server2 to be ahead of the Server3 (assuming that we have a system of 3 servers). We observe that Server3's Request Vote RPCs get rejected by Server1 and Server2 and one of Server1 and Server2 are elected as the leader.
- To maintain the leadership of a server, we spawn a new thread after a server becomes the leader which will be responsible for periodically send the Append Entries RPC with an empty entries log.
- To test the **consistency of the replicated logs** across servers is reconciled through Append Entries RPC, we start 3 servers and invoke a Put request. After the Put request has been acknowledged to the client, we spawn another server. We observe that the new server will receive the put request due to the continuous invocation of Append Entries by the leader i.e. the leader will retry Append Entries until the logs are consistent.

- To test the **replication of Raft** (also shown in Demo 1), we start 3 servers and raise a Put request. Later, we invoke a Get request and observe that the value is as per the value in the Put request.
- For split vote, we ensure that the election timeout is randomized for each server (using rand() and seed()).
- For crash points, we set the CRASH_TEST macro to 1 and place the function crash() on the line of code we want it to crash (source code file where the macro exists: src/util/common.h).

Performance Evaluation

Steps to run the test scripts are mentioned in the README.
Test scenarios covered (both strong and eventual consistency options):

- Single client read
- Single client write
- Concurrent reads on same key
- Concurrent writes on same key
- Concurrent reads on different keys
- Concurrent writes on different keys

For writes, we keep the size of the key 16 bytes, while changing the size of the value from 64 bytes to 16 KB. We collect the results for a system containing 3 servers and 5 servers. Additionally, we also test the overhead of replication by running the write tests on 1 server (single node LevelDB) and 3 servers.

9. Performance Results

9.1 Read

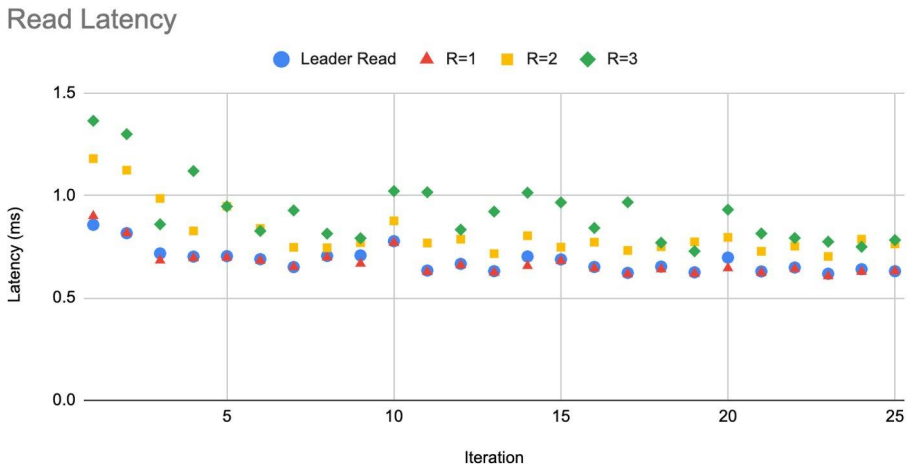

Fig. 8: Read Latency

Leader Read and R=1 have similar read latency, further as quorum size increases latency increases (more nodes to consider).

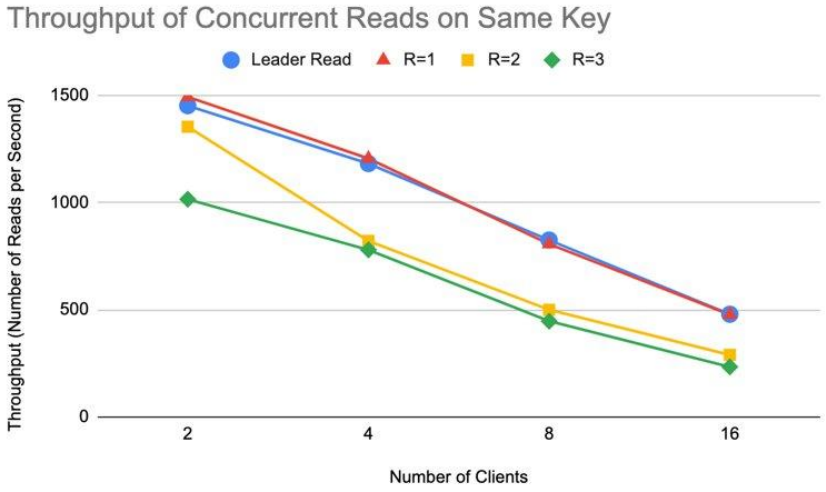

Fig. 9: Throughput of concurrent reads on same key

As the quorum size increases (more nodes to consider) and as the number of clients increases the load increases, the throughput decreases.

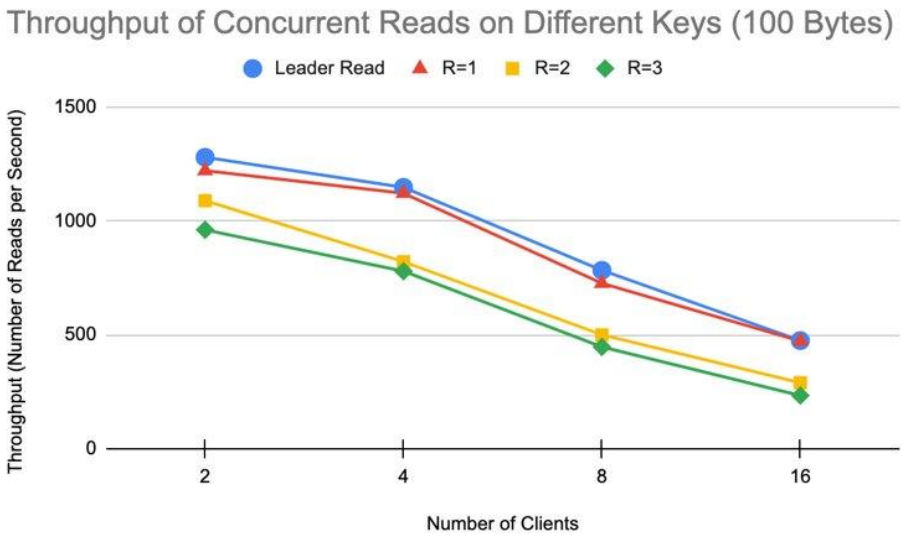

Fig. 10: Throughput of concurrent reads on different keys

The throughput trend for reads on different keys was observed to be similar to that of the case for similar key reads (slightly lower throughput though)

9.2 Write

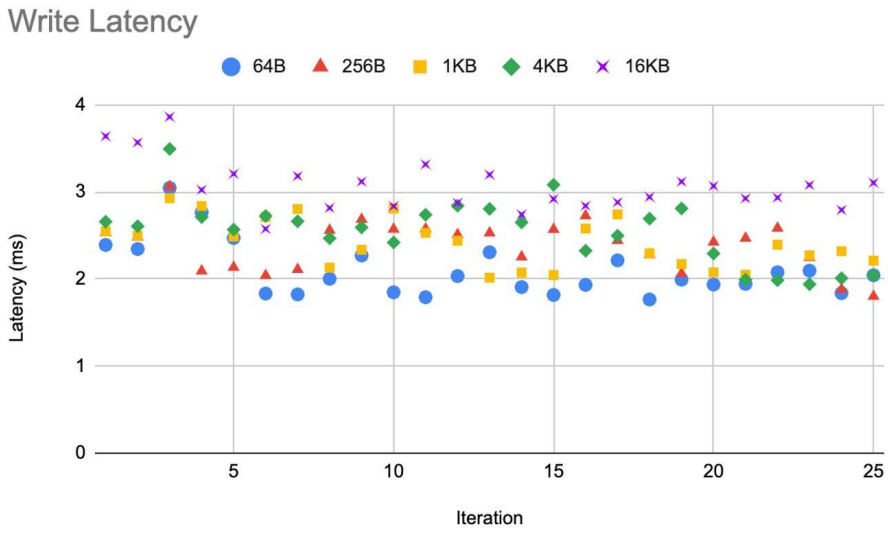

Fig. 11: Write Latency

As the write size increases, latency increases (with some minor outliers).

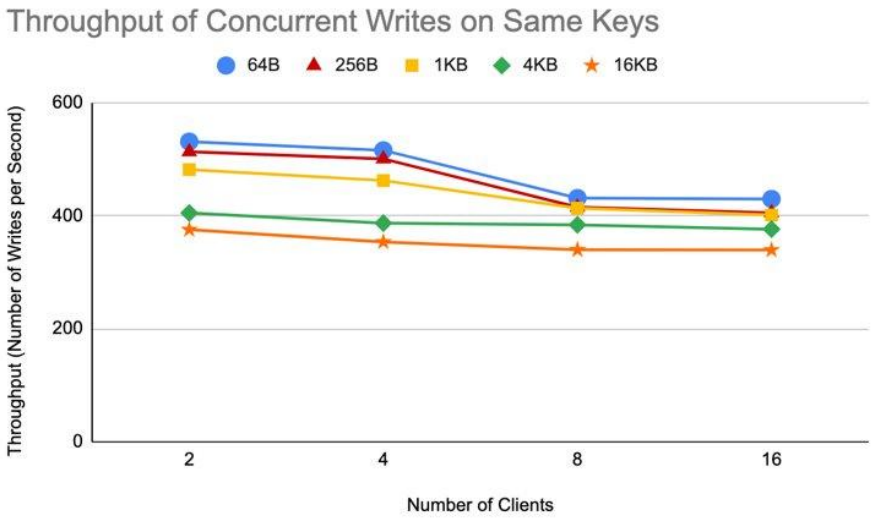

Fig. 12: Throughput of concurrent writes on same key

As the write size increases and as the number of clients increases (the load increases) the throughput decreases.

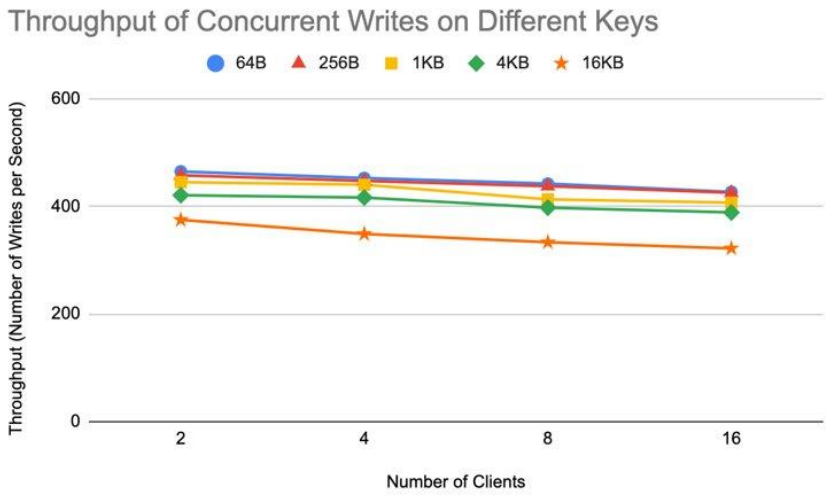

Fig. 13: Throughput of concurrent writes on same key

The throughput trend for writes on different keys was observed to be similar to that of the case for similar key writes (slightly lower throughput though).

9.3 Replication Overhead

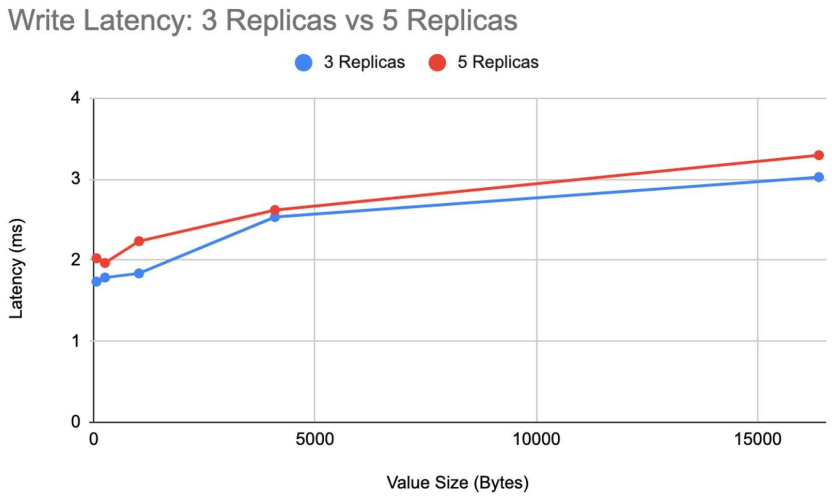

Fig. 14: Write Latency vs Replication

Replication comes with its own overhead, write latency increases as the number of replicas increases (more nodes to consider).

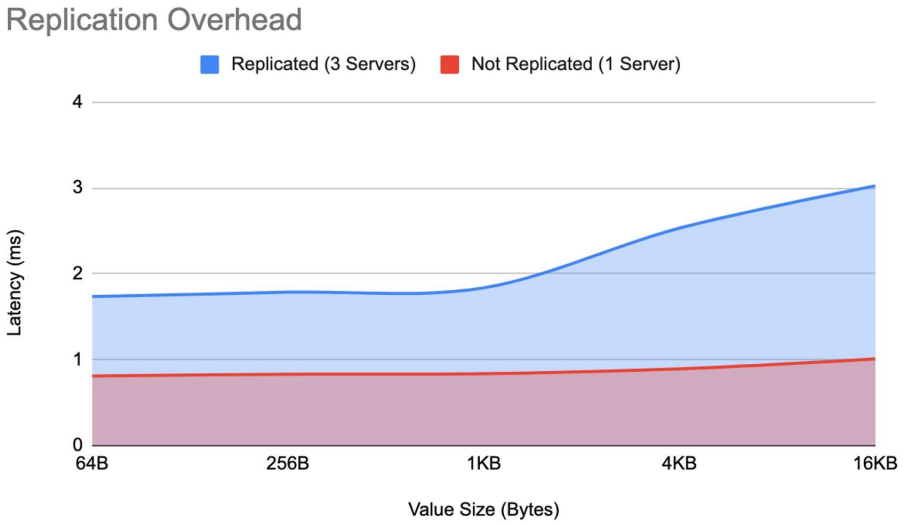

Fig. 15: Replication Overhead

Latency increases as the number of nodes to replicate on and the size of the write (value) increases.

9.3 Consistency

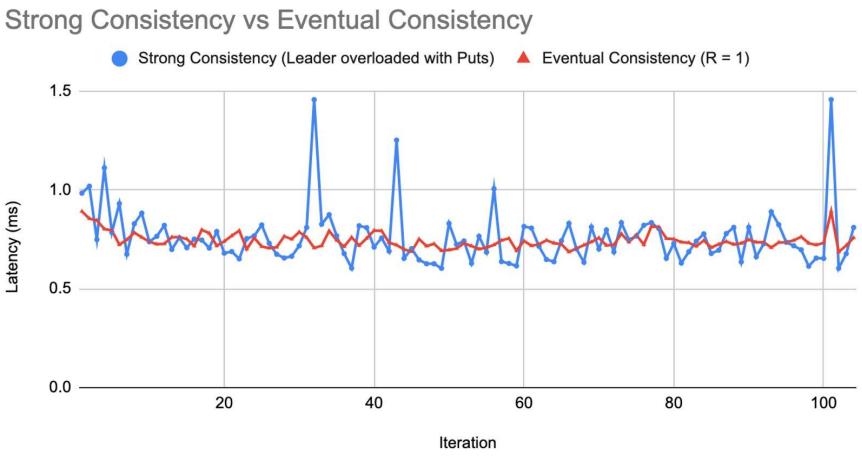

Fig. 16: Latency of strong vs eventual consistency read on a quorum R=1

10. Presentation Remarks

As per the Raft paper, we cannot have more that one leader in the system and return stale results. However, in our implementation, we can have more than one leader because of two reasons:

- Load balancer
- Autoscaling feature

Consider the following scenario:
We have a system with 5 servers and 1 load balancer. A Put request is served by a leader (say Server 1) and Servers 2 and 3 also receive this request. The Put request is acknowledged back to the client. A network partition arises such that the load balancer and Server 4 and 5 are in one partition while Server 1, 2 and 3 are in the other. The load balancer thinks that Server 1 and 2 have crashed and so set's their status to DEAD. Servers 4 and 5 will contest in an election because they do not have a leader in their partition. The load balancer can additionally spawn more nodes due to the autoscaling. Thus, we have 2 leaders in the system (Server 1 in the other partition and one of Server 4 or 5 in this partition). Additionally, if we receive a Get request, it could get a stale value violating strong consistency. This is why we assume that we do not have a network partition.

11. Demos

- DEMO GOOGLE DRIVE LINK
    - Demo 1: Raft In Action
    - Demo 2: Leader Crash
    - Demo 3: Autoscale
    - Demo 4: Tunable Consistency

- Demo 5: Leader Election Correctness

12. References

1. In Search of an Understandable Consensus Algorithm
2. LevelDB documentation
3. gRPC
4. Leader or Majority: Why have one when you can have both? Improving Read Scalability in Raft-like consensus protocols