RecoverKV

CS739 Report 2

Vinay Banakar **Syed Suleman Ahmad** **Divyanshu Saxena**

Partner group

- Abigail Matthews
- Konstantinos Kanellis
- Chetna Sureka

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
>
> *Leslie Lamport*

1 Design

In this report we present our **highly-available** distributed key-value store. Our system is also **strongly-consistent** in both failure and non-failure cases, provided that there is no partition. We describe our design choices, availability guarantees, few implementation details and testing methodology. Finally, we report the results of both ours and partner's system against our test suite.

RecoverKV uses a quorum protocol where `R=1` and `W=N` hence, `R+W > N`. Writes go to all the servers, while reads can be done on any one server. Each server maintains an in-memory map which is used for fast look-ups and also logs updates to the key-value pairs in a SQLite3 table, which is used for recovery.

Writes to the servers are managed by a load balancer which ensures the writes are always in order by assigning every write query a unique id (uid). Each `PUT` query by the client is considered an immutable object which is replicated on all the servers along with the data. The uid itself acts as a logical clock in our system, a way for any server to evaluate it's own state and perform actions accordingly. Therefore, each PUT request in our system gets associated to a unique uid. These immutable objects are maintained in an *unordered log* at each server. These logs serve as the backbone for our consistent recovery procedure.

By logging each `PUT` request, our system relies on immutable replicated logs much like Paxos [5] and Raft [7]. The replicated log, inherently maintains the invariant that the maximum local uid on every server will always be equal to the highest committed (for which the response has been sent to the client) global uid. This ensures that all backend servers are consistent when the system accepts GET requests. However, when a failed server becomes available after a crash or partition, it can start accepting new `PUT` requests before recovering the ones missed during its down-time, which can create holes in its query log. Holes are a range of uids missing in the log table. There can also exist trailing holes, i.e if the server after recovery did not receive any new PUT requests but it still would be missing a range of uids compared to other nodes. We illustrate how we cater to this issue below.

At any given moment any server can be in one of three states/modes, `DEAD`, `ZOMBIE` or `ALIVE`. `DEAD` is when the server is unresponsive, `ZOMBIE` is when it isn't qualified to serve GET requests but can receive PUT requests and `ALIVE` is when it's healthy and can serve both GET and PUT requests. The server evaluates it's mode in two situations:

1. When a server starts, we take a pessimistic stance and begin in `ZOMBIE` mode. Unless it has tried to fill its holes from all of its peers, it can't transition to `ALIVE` mode.

2. Switches to ZOMBIE from ALIVE if any holes are created due to intermittent network or disk errors.

GET requests are forwarded to only ALIVE nodes in the system, while PUTs are forwarded to ZOMBIE nodes as well. The server need not maintain a peer list since in ZOMBIE mode it requests for ALIVE peers from the load balancer. On resumption from failure or after partition healing, the server initiates recovery stage from each of the ALIVE peers to gather the missing holes. Essentially, the recovery stage can be divided into following steps:

1. Register as zombie with load balancer. Block GET requests. PUTs can still be served.
2. Fetch all ALIVE peers from load balancer. If no ALIVE peers are found then transition itself to ALIVE state and start serving.
3. If Alive peers are found, find holes in log table and send them to peer for processing.
4. Once all requested log queries are received, persist them. Next, only apply the query to local data store if the corresponding $peer - uid$ is greater than the local uid for that key.
5. Load all persisted data store to in-memory hash map.
6. Register server ALIVE with load balancer to enable GET requests.

The recovering nodes fetch missed queries from **all** ALIVE peers because different servers might have different holes in their logs if they were partitioned or coherently failed. Since we maintain a global uid it is guaranteed that combining the logs of all the ALIVE servers will preserve the total-ordering property since it is guaranteed by our system that there wont be a query that is missed by all the servers, as the maximum number of tolerable failures for our system is at most $n - 1$, where n is the total number of nodes. Queries will only be missed when all nodes fail in the system.

Incase of partitions, we might be not able to find any ALIVE peers to recover the holes in our local log. Since, we wanted to focus more on availability rather than consistency, the server registers itself as ALIVE and starts serving GET requests as well. Client might see inconsistent results until the partitions are healed, on which the server will immediately recover by interacting with peers.

The load balancer (LB) maintains the mode of each server and also maintains the partition information, i.e. which ZOMBIE server can communicate to which set of ALIVE servers. This state along with global uid is kept durable by persisting data on a separate storage engine in WAL Journaling (Write-Ahead Log) mode. A cron-job periodically pings for LB health, and restarts it on failure. We can further improve the resiliency of the LB by having a hot backup, but we leave that for future work. LB allows us to scale and reconfigure the servers easily. The LB also sends PUT requests to the servers in parallel, to maximize write throughput. Scaling the LB itself is left to be worked on in the next project.

RecoverKV is not limited by the lower bound of asynchronous consensus [6]. That is, we don't need to have $2f + 1$ nodes to support f failures. We just require $f + 1$ since each write is guaranteed to exist in all ALIVE nodes. This makes our system resilient to network errors such as duplication, message re-ordering and partitioning. Servers can crash while recovering, but it does not corrupt the data in its store. Every time it tries to recover, it will replay the same replicated log it missed queries for and reach the same replicated stable state.

On partition, we have chosen availability over consistency but this is purely a configuration choice. *Nonetheless, we are both **strongly-consistent** and **available** even after network partition but these guarantees remain only if the partitioned server is alive.* Once, a partitioned server comes back after crash, it will try to fill it's holes from reachable peers but if it cannot reach any peer after multiple attempts it will transition itself to ALIVE from ZOMBIE so that it can start serving GET requests as well. We note that, at this point in time the client might see inconsistent data, but once the partition heals, the partitioned servers immediately runs recovery process, syncing in with all reachable ALIVE peers to become consistent again. Due to CAP theorem [3], we believe this is an inherent limitation as we cannot be both consistent and available during partitions.

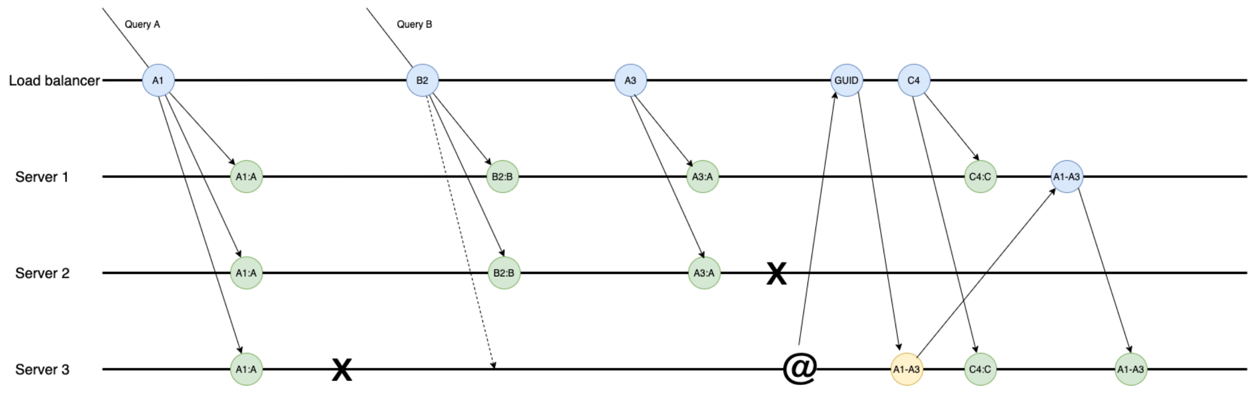


Figure 1: Event diagram of LB with 3 servers and 2 failure cases.

If there is a partition between the load balancer and a server, the server is again considered to be in `DEAD` state by LB and no requests will be forwarded to it. This also means that no other peer server will be contacting it to recover it's holes, if needed. Once partition is healed, the server immediately runs recovery process to sync itself back from reachable peers.

Our design assumes no client based partitions since the project specification only listed server to server based partition in its API format. Nevertheless, considering the scenario where clients are also in the same partition as the servers, we would only need to change the design choice of `ZOMBIE` mode servers to also start requesting GETs (for availability over consistency) and running recovery stage in-parallel periodically. We would also limiting PUT requests to the servers to which client can interact with. Currently, our system makes no such differentiation since,

1. Although the client can specify the set of servers it wants to interact with, the client interface or the LB or those set of servers themselves, can route the requests to any set of servers of their choice at the back-end.

2. The whole distributed system should be seen as a single coherent service by a client.

3. We do not support byzantine fault tolerance.

4. Finally, we do not support malicious clients. Since there is no concept of permissions in our system.

Figure 1 shows a high level view of how our system tolerates general failures. The $blue$ event indicates propose and $green$ event indicates data commit to the log. Next, `x` indicates server crashed and `@` says server is back up but in `ZOMBIE` mode. Query A as sent by client will be assigned a uid at LB and forwarded to all `ALIVE` servers. The LB sends these requests in parallel waits for ACK from all servers before responding to client. When S3 crashes, S1 and S2 continues to be in sync and data is replicated for integrity. Case `A3` shows that even exact same queries can have different uids (because they were submitted at different times to the system), this is needed to avoid arbitrarily creating holes in replicated log. Next S2 is partitioned but S3 comes back from `DEAD`, so it's mode is now `ZOMBIE`. S3 initiates recovery by first asking the LB for latest global uid, with this it can creates the set of missing holes, `A1 to A3`. Now, S3 requests the set from server 1, since it is the only `ALIVE` peer it can communicate with. S1, finds these queries and streams them to S3.

2 Implementation

2.1 Server

Our KV server is implemented in Golang and has two components, the in-memory hash table for serving GET requests and a persistent back end storage for handling PUT requests. All PUT requests are persisted on database before sending back the response. For persistence we implemented a wrapper around `Sqlite3` [4] database. We configure the database in NORMAL synchronous mode implying the writes are batched before committing to the disk. This batching of writes will not cause any durability issues since we also configure the storage engine in `WAL Journaling` (Write-Ahead Log) mode. In this mode the engine uses a Write-Ahead log which commits data to disk when the db connection is closed or a size threshold of the log is met. If the application crashes abruptly, the data written to WAL will be persisted since its the operating system's responsibility to flush page caches to disk.

We maintain two tables in each server,

`data_table <key string, value string, uid int64>`

`log_table <uid int64, query string>`

The data table is for storing key/value pairs and the log table for replicated query logs. The immutable logs are maintained in log-table and data in data-table. The holes by a server in its logs are identified by the following JOIN query,

```
SELECT a AS id, b AS next_id FROM (
  SELECT a1.uid AS a , MIN(a2.uid) AS b
  FROM log_table AS a1 LEFT JOIN
      log_table AS a2
    ON a2.uid > a1.uid
  GROUP BY a1.uid) AS tab
WHERE b > a + 1
```

Since uid is the primary key of log-table, we take advantage of the already built index during these queries. The healer (peer node who sends data to recovering node) simply does a local `SELECT` query for the range of requested uids by the recovering node, and sends the result to it using gRPC data streams.

Each server in fact has two internal servers running on two different ports, first port for handling GET/PUT requests from the client while the second port is used for handling recovery requests from peers. We made this differentiate to ensure the client request throughput should not be hampered because of queue buffering, during recovery handling (as a healer) or during recovery stage (as a `ZOMBIE`). Finally, once the server recovers from crash and marks itself `ALIVE`, it loads the stored key values from the database to our in-memory map and is now ready for serving GET requests. All servers connect with the LB on startup using gRPC channels.

2.2 Load balancer

Our load balancer is a standalone process written in Golang. Clients and servers communicate with it via gRPC. All the requests must go through load balancer which is managed by the client interface. The load balancer keeps into account which servers are `ALIVE`, in `ZOMBIE` recovering state, or `DEAD`. PUT requests from clients are forwarded to every non-`DEAD` server concurrently, while GET requests are forward to any one randomly chosen `ALIVE` server. Any `DEAD` server is automatically restarted by the LB. It also maintains server's partition state, which is refereed to during recovery stage of a crashed or already partitioned server. Finally, it maintains the current global uid of the system, which each server strives to achieve to complete its log. For durability, all data is persisted on disk using `sqlite`. A cron-job periodically pings the load balancer for its health every 100ms and restarts the LB if it fails to respond.

2.3 Client Interface

The client interface is implemented in C++, and uses gRPC to connect directly with the LB. All API's mentioned on the project specification are implemented. Since, all clients connect to the same LB, the LB needs to distinguish clients connections so that it can make sure a client cannot interact with a server it did not `init` for. Therefore, the client interface generates a unique user id for each client connection and forwards it to the LB, which the LB uses for maintaining per client partition and permissions states.

Therefore, the LB can connect to multiple clients but all requests from the clients are ordered and assigned a unique id. Each client's GET request is evenly divided amongst all `ALIVE` servers to avoid any hot spots.

3 Protocol specification

We used the Google RPC (gRPC) [1] opensource framework for communication between client and server. Our choice of using gRPC over REST or socket based implementation was based on the ease of use and scalability of the gRPC framework. Since gRPC is encoding agnostic, we had to choose between *JSON* or *Protocol Buffer* communication formats. We went forward with *Protocol buffers* for better performance because JSON formatted data is relatively much slower to encode and decode. The other advantage of using gRPC is it maintains connections in a thread pool allowing clients and server to concurrently send requests although they are single threaded. By maintaining this pool it also alleviates connection creation and tear down cost.

Since all our communication happens via gRPC, it allows us to instantiate our services on multiple nodes. Communications happen via two main services between three interfaces as described bellow,

3.1 Between Client and LB

The client and LB utilize `ExternalService` gRPC service, this constitutes the following functions,

- **`ExternalService`.InitLBState**: Inits the LB and registers the servers client can communicate with. Returns error code.
- **`ExternalService`.GetValue**: Sends a GET request to the LB expecting a value or error code.
- **`ExternalService`.SetValue**: Sends a PUT request to the LB where a key and value is passed. It returns the previous value of the key if any and an error code.
- **`ExternalService`.FreeLBState**: When the client wishes to shuts down it's connection, it will invoke this function so that LB can clear it's state. Returns an error code.
- **`ExternalService`.StopServer**: Kills the specified server the client had already init'ed with earlier. Returns error code.
- **`ExternalService`.PartitionServer**: Invoked to partition the servers passes as argument by the client. Returns error code.

3.2 Between LB and Servers

The client and LB utilize `InternalService` gRPC service, this constitutes the following functions,

- **`InternalService`.GetValue**: Forwards the GET request from the LB to any one server. Captures the response or any error code.
- **`InternalService`.SetValue**: Forwards the PUT request from the LB to all servers. Captures the response or any error code.
- **`InternalService`.StopServer**: Tells the server to shutdown. On clean shutdown, return confirmation from server, otherwise, none.
- **`InternalService`.PartitionServer**: Forwards the request to server to re-run the recovery phase, so that the server is consistent within the partition. Returns a response that recovery has started or an error code.
- **`InternalService`.PingServer**: Sends an empty object to the server for periodically checking if the server has successfully restarted and has good health. Returns server health response or error code.
- **`InternalService`.MarkMe**: Used by the servers to communicate the current state to the LB (whether they are operating in `ZOMBIE` or `ALIVE` mode). Returns the global *uid* from the LB.
- **`InternalService`.FetchAlivePeers**: Used by the servers to fetch the list of all `ALIVE` peers it can contact with, from the LB. Returns a list of servers, if any, or returns an error code.

3.3 Between Servers

For server to server communication we utilize `InternalServerService` gRPC service, this constitutes the following functions,

- **`InternalServerService`.FetchQueries**: This is a special function where we use gRPC streams [2] to respond back a stream of queries from the replicated log to a peer server which requested it. The peer will call `FetchQueries` with the range of missing holes as a parameter. Along with each stream this returns an error code.

4 Testing Methodology

We implemented the following three test suites:

4.1 Correctness Tests

Our correctness and consistency testing methodology can be broken down into five categories.

4.1.1 API Structure

Our first set of tests validate the client API structure, verifying that all edge cases are properly catered to, for example, we check for calling `kv739_get()` or `kv739_put()` before initializing the client instance using `kv739_init()`, validate whether `kv739_shutdown` actually breaks down the connection, and if return code for all the methods are returned as expected.

4.1.2 Batched Updates

The second set of tests verifies if the put requests to the servers, followed by get requests, return the same set of values. We test for both unique and repeating set of keys - whether the values received from the store are the most recent ones or not. We further extended the tests to have inter-leaving writes and reads, and similar to previous case verified successful updates to the latest set of keys. Additionally, we also checked for valid response codes, such as whether the server handles erroneous requests for non-existent keys.

4.1.3 Durability

These tests verify the durability of key/value pairs inserted at the servers. For each server started at the backend, we `init` for them individually. This makes sure the client requests are routed to that particular server. We make two sets of tests for durability.

We insert a range of key/value pairs, and wait after every PUT so that we are sure they are registered at the server. Then following all PUTs, the test kill the server using clean `die` API call. After the server is restarted and the connection is resumed, we attempt to read all the keys we had stored prior to the crash. The values are then compared to the test's local copy and are verified if they all match.

In the second case, we do not wait between PUTs, and concurrently send multiple PUT requests and wait for the acknowledgements. Once the acknowledgments are received we immediately kill the server using an unclean `die` call. Using the same methodology as above we then verify whether all GET requests return the correct values after server restarts. This test ensures that there where no in-progress operations at the servers that were acknowledged but not persisted. Any acknowledgement received by the client should be deemed persistent.

4.1.4 Availability and Recovery

These tests make sure if the system can tolerate server failures, and the service is available.

We randomly pick half of the servers initialized by the client, and register the `die` API call on them. We test for both clean and unclean types. After the servers have died and before they are restarted again we send interleaving PUT/GET requests to see if the system is still responding correctly. This verifies that the system can tolerate both clean and unclean failures, even if half of the cluster is down. We also carry this test by killing all except one node in the system, to see if service is still responsive.

Lastly, we check if the system can successfully recover from failures during the recovery phase itself. For our specific system, once the server is respawned it tries to fetch all missed queries from its peers. We have two sets of tests for testing this.

For the first test, we kill the server during the recovery phase to check if it will again become alive. We init separately for a specific server, kill it using the `die` API with clean death argument. Following immediately, we submit another die call (unclean) after a one second wait. Since, our system takes some initial recovery time we are sure it will be in recovery phase when the second `die` API call is made. After a specific time period, we then send GET/PUT requests to the server to see if it successfully restarted again.

For the second test, we make sure that we only have one server instance running, and see if after killing it, it still becomes responsive even though it might not have alive peers to recover from. This ensures even that even though the recovery is not possible, the system is still configured to be available.

4.1.5 Consistency

For data consistency, we need to check if the data placed on a specific server, is reflected on every server.

We start off by first registering an `init` for each server separately. We generate a key, and go over each server to PUT a uniquely different value, for that key. The test then reads that value from each server, and ensures that the value equals the last updated value by the test. This ensures the system is consistent in non-failure cases.

In term of failures, we `init` for all servers. Then the test randomly picks one server, and registers a `die` command on it (we test with both clean and unclean types). Immediately following it, we send PUT requests for new keys to the system so that the server that has just died misses the updates. After a specific timeout, the test then calls `init` for the server that was killed, and sends GET requests for the new set of keys, and verifies the returned values. This makes sure that the system will become consistent after some time (specified by the timeout value).

We further extend the above tests, by killing servers while the system is still serving requests. We iteratively go over $n - 1$ set of servers and call the `die` API, while concurrently sending inter-leaving PUT/GET requests. When only one instance remains, we check if the last set of written values are *correctly* retrieved using GET requests. This test ensures the system is still consistent, even when it is encountering failures while simultaneously serving clients.

As an extension, we also carry out the test to kill multiple servers and then see if all them had consistent data for PUT requests they missed.

4.1.6 Partitioning

For our system, partitioning is defined by the inability for servers to talk to each other due to network errors, therefore these tests aim to verify this claim.

First, we make sure if the partitioning API is working as expected. We create complete partitioning in the system such that no server is reachable from any other server. We put some key-value pairs on the store and then, we randomly pick a server and kill it using the API. Immediately following, we send a set of PUT requests updating the keys value, making sure that the update requests get registered before the killed server is restarted. Therefore, the server cannot recover by contacting its peers because the system is totally partitioned. We verify this by sending GET requests to the restarted server.

Secondly, we also verify if the system comes back to consistent state when the partition heals. Similar to previous test, we completely partition the system, and a server is killed that missed the a set of updates. On healing the partition, all servers are reachable from each other. We wait for a specific time period, to give the server time to acquire updates which it had missed. We then issue GET requests on the same set of keys as those in the last test on the server that was originally killed. If all the returned values match the latest set of updates (which the killed server missed), the test passes as the server is again in consistent state.

4.1.7 Client Permissions

In these tests we check if the client can interact with a server it did not run the `init` command for. We `init` for a subset of servers, and try to register `die` and `partition` API calls on the other set of servers. If the API calls return an error codes, then these tests pass.

4.2 Single-Client Performance Tests

We aim to evaluate the performance of our system for `PUT` and `GET` operations, and analyze the cost that recovery mechanisms have to pay, in terms of the performance of the system. We measure the latency of each operation by a single client, under the following workloads:

- **Normal Operation**: We measure the latency and throughput of each operation, for a total of X writes and X read operations, where X is the number of keys to be put on the store, configurable as a user argument.

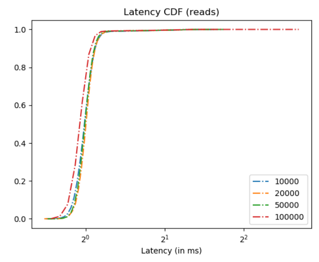
- **Failure Recovery**: The test is the same as the previous one, with the only difference that one server is randomly killed in between the writes, and then, another server is killed during the reads. This helps us motivate the impact of recovery on performance.

4.3 Multi-Client Performance Tests

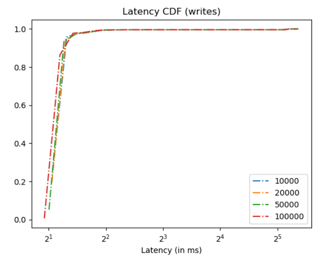
With the aim to evaluate how the performance of our system is impacted by concurrent requests coming in from multiple clients, we evaluate the latency and throughput for `PUT` and `GET` operations individually.

Cloudlab (`c240g2`)	
CPU	2x Intel(R) Xeon E5-2660 v3 2.20GHz 10-core
Memory	157GB DDR4 RDIMMs @ 2133 MT/s
Storage	447GB SATA SSD
OS	Ubuntu 20.04 LTS

Table 1: Machine Specifications



(a) Latency of GET operations with varying number of keys



(b) Latency of PUT operations with varying number of keys

Figure 2: RecoverKV: Cumulative distributions of Latencies for PUT and GET operations for varying number of keys

5 Results & Discussion

All our experiments are performed on Cloudlab machine, whose specifications are provided in Table 1. For our evaluation we configured our cluster to have 3 servers and 1 load balancer. In the results Both server and load balancer are on the same machine but we deployed our system on multiple nodes without needing changes as all our communication happens via gRPC. Our partner's cluster also has 3 servers and their system is evaluated on an identical machine.

5.1 RecoverKV Results

5.1.1 Correctness Tests

All our correctness tests were successfully passing on the system.

5.1.2 Varying Key Sizes

Figure 2 shows the cumulative distribution of latencies of PUT and GET operations on RecoverKV. From the graphs, we see that the performance of the system is constant even against increasing number of keys. We claim that the bottleneck in the system, is hence, the communication between the various components. Otherwise, we would expect the performance to degrade with increasing number of keys on the KV Store (and hence, longer time to seek from in-memory Maps). Figure 3 shows even for varying keys the communication between client and the load balancer is constantly 30% of the overall latency for a single GET request.

5.1.3 Normal Operation vs Server Failure Condition

Next, we compare the performance of RecoverKV in the face of server failures. More specifically, we aim to evaluate if failures have any adverse impact on the performance of the system. Figure 4 shows the throughput performance of the system under failures.

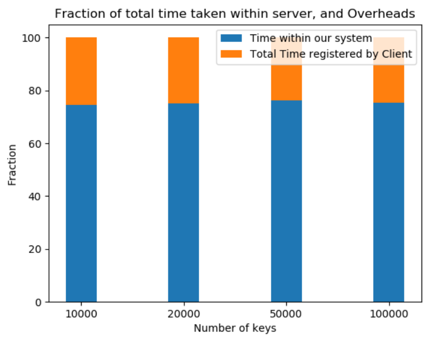
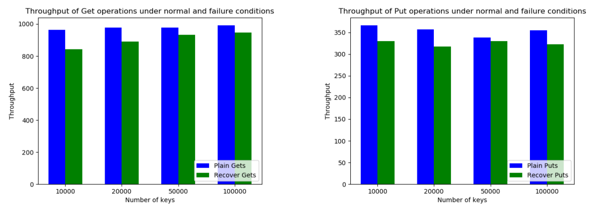


Figure 3: Client to LB communication cost



(a) Throughput of GET operations

(b) Throughput of PUT operations

Figure 4: RecoverKV: Comparison of PUT and GET operations under normal and failure conditions

The throughput is measured over the entire set of keys read/written on the store and any failures in between impact the net throughput of the system in that duration. From the figures, we show that the system does not suffer huge performance losses and as described earlier, still maintains consistent operations. This slow down in GETs compared to P1 are due to arbitrary locks and conditions that we haven't optimized for since our focus had been on correctness for P2. We will be improving this in P3 and expect GET throughput to increase under both plain and recovery conditions.

5.1.4 PUTs vs GETs

A quick comparison of Figures 2a and 2b, and figures 4a and 4b show that `PUTs` are significantly slower. The reason being the fact that our write quorum is the entire node set, while the read quorum is only 1. Hence, for writes, the time is determined by the slowest server, while in reads, only a single server is used. Nevertheless, our PUTs are concurrently sent to all servers. We observe that our SQL queries can be further improved for performance, this will be handled in P3.

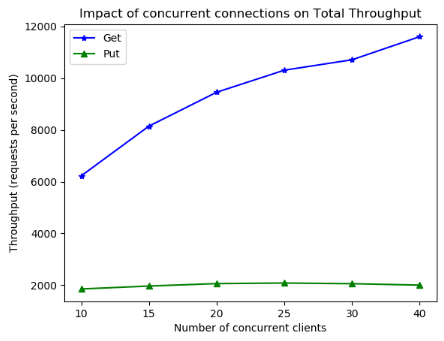


Figure 5: Throughput of PUTs and GETs for multiple concurrent clients

5.1.5 Multi-client Performance Tests

From Section 5.1.2, we saw that the bottlenecks are on the communication of the system. Upon deeper probing, we found that gRPC calls in Golang are blocking for the client. Hence, until one RPC has been completed, the client cannot register another RPC call. This implied that the system evaluation is missing on the throughput. Hence, we present the results of our KV Store under cases of several concurrent clients. We see that the throughput saturates at around 40 concurrent clients, with maximum `PUT` throughput reaching as high as `2050 rps` and `GET` throughput reaching as high as `12000 rps`.

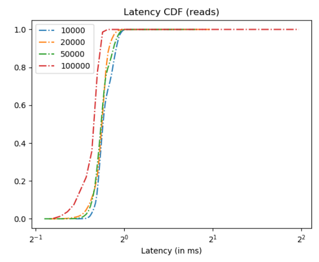
5.2 Partner KV Store

5.2.1 Correctness Tests

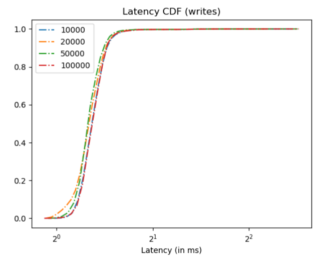
Our tests identified some bugs in the partner group system, which they were able to patch up quickly. Eventually, all our correctness tests passed for their system as well (We did not run any strict consistency check for the partner group).

5.2.2 Varying Key Sizes

Figure 6 shows the cumulative distribution of latencies of PUT and GET operations on Partner KV. Comparing with Figure 2, we see that the trends are similar to what we observe, which go on to show



(a) Latency of GET operations with varying number of keys



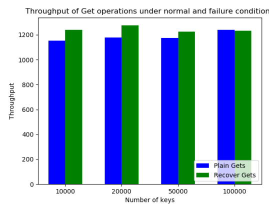
(b) Latency of PUT operations with varying number of keys

Figure 6: Partner KV: Cumulative distributions of Latencies for PUT and GET operations for varying number of keys

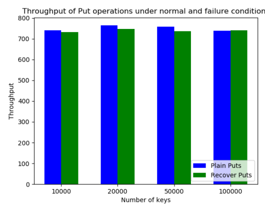
that for their system as well, the communication costs are the major bottlenecks.

5.2.3 Normal Operation vs Server Failure Condition

Figure 7 shows the throughput performance of the Partner KV system under failures.



(a) Throughput of GET operations



(b) Throughput of PUT operations

Figure 7: Partner KV: Comparison of PUT and GET operations under normal and failure conditions

We observe that the Partner KV also does not suffer any huge losses in performance under conditions of failure.

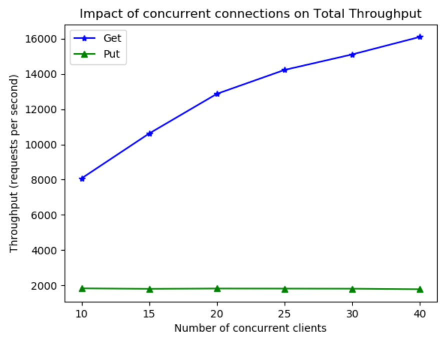


Figure 8: Throughput of PUTs and GETs for multiple concurrent clients

5.2.4 Multi-client Performance Tests

From figure 8, we observe that the throughput of the Partner KV saturates at around 40 concurrent clients as well, with maximum `PUT` throughput reaching as high as `1850 rps` and `GET` throughput reaching as high as `16000 rps`.

5.2.5 Observations

We observe that the Partner KV store records a higher Write throughput for a single client, while it saturates before RecoverKV for multiple concurrent clients. We claim that this is the result of blocking RPCs from the client to the LB in our system. In case of multiple clients, the LB can concurrently serve multiple RPCs from different clients (each blocking at the client) - and that measures the real throughput of RecoverKV. Our system is fully capable to be deployed on a multi-node cluster machines however we observe it's not the same for our partner's system and hence postpone the multi node evaluation. From manual analysis with tools such as `iostat` and `htop` we noticed the partner's system is highly I/O bound, to prove this we conducted above similar experiments on HDD and observed that RecoverKV did **10x** better in throughput for PUT requests compared to theirs.

6 Conclusion

In this report, we described RecoverKV, a highly available and strongly consistent key value store, that can pro-actively recover from failures. Further, RecoverKV can be strongly consistent unless there are no failures during partitions. We conclude that there is a cost of ensuring availability while maintaining consistency - as we observe a decrease in the throughput compared to the single instance server we had implemented in the previous project. Further, we note that there is a cost that we pay, by routing all requests via the Load Balancer, in terms of the throughput of the system - but at the same time it also allows RecoverKV to be flexible and manageable.

References

[1] gRPC. `https://grpc.io/`, 2021.

[2] gRPC Streams. `https://grpc.io/docs/what-is-grpc/core-concepts/#server-streaming-rpc`, 2021.

[3] S. Gilbert and N. Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.

[4] R. D. Hipp. SQLite3, 2020.

[5] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[6] L. Lamport. Lower bounds for asynchronous consensus. Technical Report MSR-TR-2004-72, July 2004.

[7] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.