RecoverKV

CS739 Report 3

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

> In fact, it is among the simplest and most obvious of distributed algorithms.
>
> *Leslie Lamport on Paxos*

1 Design

In this report we present our **highly-available** and **strongly-consistent** distributed key-value store. We describe our design choices, availability and consistency guarantees, few implementation details and testing methodology. Finally, we report correctness and performance results of our KV store.

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 on failure scenarios.

All read and write requests from the clients goes through a *master server*. Upon receiving a write request, the master server concurrently sends the `PUT` query to every replica and also updates its own state. The PUT confirmation to the client is sent only when the master has received acknowledgement from all the replicas. Whereas for read requests, any of the replicas can be contacted for the latest update. A single master server ensues that all requests are always in-order by assigning every write query an unique id (uid). 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.

Each `PUT` query by the client is considered an immutable object which is replicated on all the servers along with the data, therefore, each `PUT` request gets associated to a unique uid. These immutable objects are maintained in an *unordered log* at each server, which serves as a backbone for our consistent recovery procedure.

By logging each `PUT` request, our system relies on immutable replicated logs much like Paxos [6] 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 replicas are consistent, in non-failure cases, when the system accepts GET requests.

At any given moment any server can be in one of four states/modes, `DEAD`, `ZOMBIE`, `READY` or `ALIVE`. A summary for each of these modes can be found in Table 1. Each server maintains it's state in memory, and contacts the master server if their mode changes. Therefore, the master server maintains the current state of every replica so that it can forward requests appropriately. Each 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. This is identified by periodically checking for holes in the database.

GET requests are sent to only `ALIVE` nodes in the system, while PUTs are forwarded to `ZOMBIE` nodes as well. The clients do a GET on a random server, so that the load is distributed uniformly. For every failed GET request the client retries for

MODES	PUT	GET	HEAL	ELECTION
DEAD	×	×	×	×
ZOMBIE	✓	×	×	×
ALIVE	✓	✓	✓	×
READY	×	×	✓	✓

Table 1: Server modes and supported features.

another different server until the majority of the servers failed to respond.

Failure Recovery. If a failed server restarts after a crash, it can immediately 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 maintain consistency during failure cases below.

Each server needs to maintain a peer list since, any node can communicate with any other. On resumption from failure, the server initiates recovery stage from each of the `ALIVE` and `READY` peers to gather the missing holes (*heal*). Essentially, the recovery stage can be divided into following steps:

1. Mark thyself as `zombie`. So that `GET` requests are blocked, `PUT` requests are enabled.
2. Find holes in log table and send them to all peers for processing. Peer `ZOMBIE` servers can not *heal*.
3. 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.
4. Load all persisted data store to in-memory hash map.
5. Reach out to peers to identify the current master. If master exists, mark yourself as `ALIVE` to enable GET requests.
6. If peer says master does not exist, mark yourself `READY`.

The recovering nodes fetch missed queries from **all** `ALIVE` peers because different servers might have different holes in their logs if they coherently failed. Since we maintain a global uid at the master server, 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.

Master Election. The master maintains the mode of each server in the cluster by sending periodic `Heartbeats`, so that it can avoid sending `PUT` requests to `DEAD` servers. From our observations 5 seconds seems reasonable for our 5 node cluster in terms of network load and performance. The master also sends `PUT` requests to the servers in parallel, to maximize write throughput. `READY` mode signifies if the server is actively participating in an election. This distinction is important because,

1. We don't want to handle GETs or PUTs in masterless environment.
2. This allows recovered servers to rejoin an ongoing election process, making sure no deadlocks exist in an iterative failure scenario during the election process itself. Guaranteeing progress in the system.

To handle master failures we use an algorithm for master election which relies on a static priority list of servers that is maintained at each server. This ordered list is decided when the servers initiate (the first server in this is list hard-coded to be the master when bootstrapping). The master election process is triggered when the current master does not send a `Heartbeat` RPC beyond 15 seconds, indicating that the current master is unavailable (or perceived dead by all others). Each `ALIVE` server enters the election stage,

1. First all replicas check with the old master if it is really `DEAD`. If not, stop election and mark old master as current master, if dead, mark yourself `READY` and continue with the following steps.
2. Each server sends its max local uid to all it's peers (can only send if it is in `READY` mode).

Servers maintain a timeout of 45 seconds before receiving the uid from the peers. Any uid received during this period will be considered for election, if none is seen then restart the election stage from step 1.

3. Next, determine the *minimum* uid value returned from the peers and identify the highest priority server with this uid in the list. Note, the uid can only differ by a single value at max.

4. Now using a push based mechanism, tell the identified server that they are the master. This is to avoid hard coding arbitrary timeouts.

5. The elected master will acknowledge the votes. If it fails to do so, the peers will pick the next server with minimum uid in the priority list. The master once receiving majority of votes will start sending`Heartbeat` RPC to all servers (dead or alive).

6. Receiving the new masters' acknowledgement (heartbeat) the server marks itself from `READY` to `ALIVE`, marking the end of the election.

Post election, when client tries to communicate with the old master for PUTs, it will get a response of the new master's address for it to retry. A miss match in uid can arise when a master dies before propagating a PUT request to all servers, some servers can have higher uid to compared to others. This can happen because we don't employ two-phase commit [4]. Hence, the difference between uids during election can not be greater than 1, as for all other requests the uid is guaranteed to be consistent across the servers (as the master confirms to the client only when it has received confirmation from all replicas). Furthermore, we chose to pick the least uid server as the master during the election as this will ensure we will always have to rollback any server with higher uid than itself. Choosing max uid would have required us to always force recovery on servers with lower uid and would have added values in the system that the clients would not received a confirmation for. At the same time, picking the majority most common uid would have involve doing both rollback and recovery based on the situation. Our algorithm breaks split votes implicitly, when two servers return the same minimum uid, by referencing to the order of servers in the priority list. Our logic for picking a master with minimum uid, coupled with the priority list ordering guarantees that at any point in time, our system will always have a single master.

We have used timeouts as prudently as possible, so not to be sensitive to hardware and load changes. To avoid timeouts we use push based mechanisms to provide liveness to our system (Ex, peers automatically voting for master in an election). **In many ways, the crux of our consensus algorithm is to make decisions independently and only then check if those decisions apply to the real world.** Overall, the 45 seconds timeout value indicates an election term for our system, as after the timeout the election process re-instantiates, and the servers use a 15 seconds timeout to identify if the master is dead or unresponsive.

If a server finishes recovery and contacts a peer for master during election, the peer indicates the ongoing election process and asks the caller to mark itself as `READY` state. So now, the new recovered server can also participate in the election. However, if the server's participation is too late, it does not matter because the newly elected master sends heartbeats for this server to concede and it marks itself `ALIVE`.

Availability and Consistency. RecoverKV can tolerate up to $([n/2] - 1)$ failures in a system of n nodes, since we need a majority of servers to be alive for the election process to guarantee progress. The system is also strongly-consistent since we use a quorum of $W = N$, which ensures all writes are propagated to all the servers before a confirmation is sent back to the client. Having a single master server in the system makes sure that all requests are ordered so that we do not have worry about any conflicts on read requests. Furthermore, when a server crashes, on resumption it recovers and fills its log holes from its peers which ensures that a replicated log is maintained at each server before it can start serving client requests again. We make no assumptions on the type of failures cases, as our system caters to both replica and master failures.

Any new master elected, ensures that the replicas log is consistent with its log as all higher uid servers are roll-backed. This certifies that all PUT requests

that were not acknowledged to the clients are removed from the system. 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. For servers that crash during the election process, they can recover and join back if the new master is yet to be decided. Lastly, as discussed in the previous section, maintaining a static list of server priority, and using the minimum uid logic helps to decide upon a single master independently, we are guaranteed to have only one server elected as a master at any point in time. As long as the majority of servers are alive, our system assures both safety and liveness properties. This makes our system resilient to network errors such as duplication, and message re-ordering.

Although our algorithm closely resembles bully [3], we are vastly robust both during leader election and number of messages required to arrive at consensus. Example, we don't break if multiple servers claiming master arise during an election term. Our system provides same guarantees as Raft [7] does and even uses less number of messages per election for a same size cluster, assuming no partitions (although we don't yet see how we would fail during partition).

Assumptions

1. We do not deal with client or server-based partitioning
2. At most majority of the servers are ALIVE during the election process.
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.

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` [5] 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,

```
1  SELECT a AS id, b AS next_id FROM (
2    SELECT a1.uid AS a , MIN(a2.uid) AS b
3    FROM log_table AS a1 LEFT JOIN
         log_table AS a2
4      ON a2.uid > a1.uid
5    GROUP BY a1.uid) AS tab
6  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 three internal servers running on three different ports, first port for handling GET/PUT requests from the client while the second port is used for handling recovery requests from peers and the third to handle election mechanisms. We made these differentiation's 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`). Coupling a router with each server helped us to stay

modular and allows to test different consensus algorithms without any downtime. 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. The server then contacts it's router, which contains a static priority list, to request master info from a peer. During the election stage, the router is responsible for sending local max uid and receiving peers' uid for determining who is the next master.

Once router finds the master, it sends an RPC call indicating it's vote to it. It is also responsible for receiving `heartbeats` and sending a synchronous response. The server when elected as master, will use the router to send heartbeats to maintain state of the followers in the cluster. Each router in master state maintains, the current mode (`DEAD`, `ZOMBIE`, etc) of the servers and a global uid of the system (which each server strives to achieve). The master router also provides `rollback` rpc, which instructs it's followers to delete the PUT with a given uid. Router does not maintain any client state since, we don't consider client partition. The static peer list the server starts with is persistent and uid is anyway persistent in log and data table.

2.2 Client Interface

The client interface is implemented in C++, and uses gRPC to connect directly with the servers. Each client maintains a static list of all the servers in the system, and determines the current master server using gRPC calls to any random server in the list. The client interface sends all PUT requests to the master server whereas the GET requests are sent to any random server in the list. If the master or a server fails to respond within a 20 second timeout then another server, which has not been tried is contacted. The master server 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. All API's mentioned on the project specification were implemented.

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 Master

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

- **`ExternalService.GetValue`**: Sends a GET request to the server expecting a value or error code.
- **`ExternalService.SetValue`**: Sends a PUT request to the master where a key and value is passed. It returns the previous value of the key if any and an error code. It may also return who the current master is if the client contacts a replica instead of the master.
- **`ExternalService.StopServer`**: Kills the specified server the client had already init'ed with earlier. Returns error code.

3.2 Between Master and Replicas

The master server and the replicas utilize `InternalService` gRPC service, this constitutes the following functions,

- **`InternalService.SetValue`**: Forwards the PUT request from the Master 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.HeartBeat`**: 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.Rollback`**: Use by the newly elected master to send it max local uid so that all replicas are consistent with its log.
- **`InternalService.MasterStatus`**: This is used to ensure the unresponsive master (for 15 seconds) is really `DEAD`. If not, and it was only an intermittent network issue the master will respond back `ALIVE` and the replicas cancle the election.
- **`InternalService.IdentifyMaster`**: Tells the server that they are the new master, to which the newly elected master acknowledges.

3.3 Between Replicas

For server (replica) to server (replica) communication we utilize `InternalReplicaService` gRPC service, this constitutes the following functions,

- **`InternalReplicaService.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.
- **`InternalReplicaService.FetchMaster`**: Used by a recovering server to identify what is the current master in the system. Return the address of the master or indicate election process is happening.
- **`InternalReplicaService.SendMaxUID`**: During the election process all `Ready` servers send their max local uid value.

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.

For the third test, we insert a set of keys, kill the master server and wait for a specified timeout before requesting the same set of keys from a randomly chosen server from the peer list at the client. We check if these values are correct. Next we do new insert more keys to the newly identified master and do a get request to verify that the system as a whole is available after master failures.

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 the master server itself, and then see if all them had consistent data for PUT requests they missed.

Additionally, we also crafted some tests to check for inconsistency, by sending out GET requests to every servers after a PUT to the master. We also wrote extended tests to see if the server returns inconsistent data if the master is made to die while processing PUT requests. As our system is strongly-consistent, it does not returns in-consistent data to client, and all GETs receive the latest value.

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.

5 Results & Discussion

All our experiments are performed on Cloudlab machine, whose specifications are provided in Table 2. For our evaluation we configured our cluster to have 5 servers. In the results all server backends and associated routers are on the same machine but we can deploy our system on multiple nodes without needing changes as all our communication happens via gRPC.

5.1 Correctness Tests

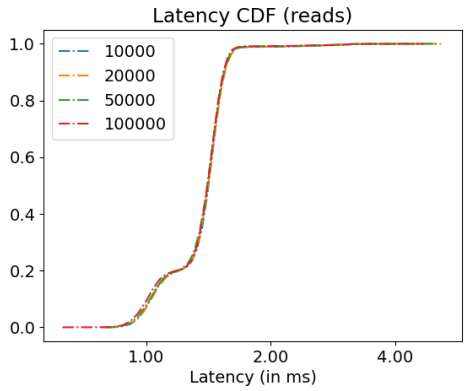
All our correctness tests were successfully passing on the system.

5.2 Varying Key Store Sizes

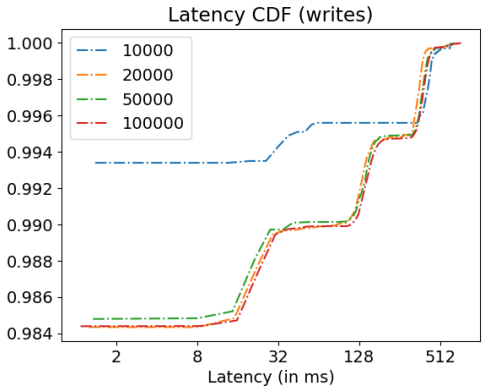
Figure 1 shows the cumulative distribution of latencies of PUT and GET operations on RecoverKV for varying number of keys put onto/read from the data store (10k, 20k, 50k and 100k). We make three important observations from the graphs.
First, from figure 1a, we observe a clear plateau for the GET operations. As per design, we randomly use any of the backend servers for serving GETs. However, accessing the master backend requires one less RPC call, compared to the other replicas (because of redirection via a router). Second, we see that the performance of the system is constant for GETs 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). Finally, the latencies for writes are also roughly similar irrespective of the key sizes (indicated by the fact that the CDF graph is shown from the 98.4th percentile). We claim that the small levels observed in figure 1b are because of network congestion mechanisms utilized by gRPC - and hence, at some threshold above 10k requests,

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 2: Machine Specifications



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



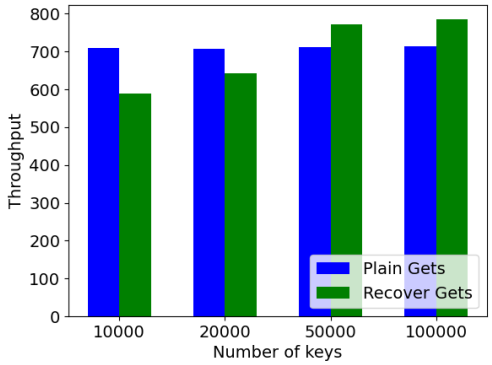
(b) Latency of PUT operations with varying number of keys

Figure 1: RecoverKV: Cumulative distributions of Latencies for PUT and GET operations for varying number of keys. Note that the Write CDF has a lower bound of 0.984, which means the plots for all key sizes is a steep slope at 2ms.

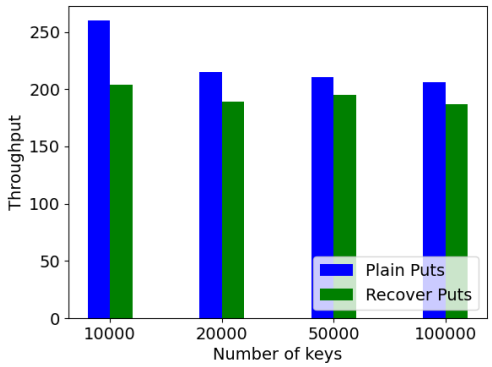
the protocol shows the behaviour as observed in the figure.

5.3 Normal Operation vs Server Failure Condition

Next, we evaluate 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 2 shows the throughput performance of the system under failures. 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



(a) Throughput of GET operations



(b) Throughput of PUT operations

Figure 2: RecoverKV: Comparison of PUT and GET operations under normal and failure conditions

that duration. From figure 2a, we observe that `GET`s are not adversely impacted by the failure of a single node - in line with our design goal of a failure-tolerant service. Further, in case of a remote replica failing, more requests are directed towards the local backend at the master - which (as shown in fig 1a), give a better throughput. For `PUT`s, the throughput is roughly similar compared to the no failure case.

5.4 PUTs vs GETs

A quick comparison of Figures 2a and 2b show that `PUT`s 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.

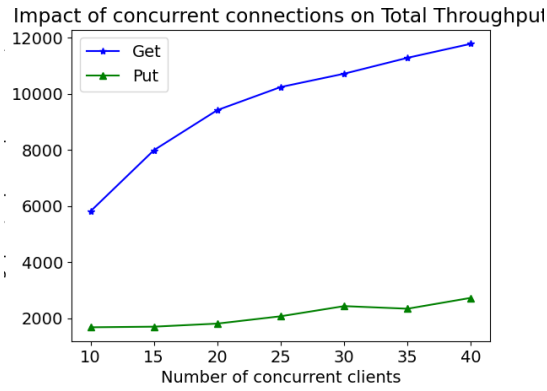


Figure 3: Throughput of PUTs and GETs for multiple concurrent clients

5.5 Multi-client Performance Tests

From Section 5.2, 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`.

6 Conclusion

In this report, we described RecoverKV, a highly available and strongly consistent key value store, that can pro-actively recover from failures. We conclude that there is a cost of ensuring consistency while maintaining availability - as we observe a decrease in the throughput compared to the RecoverKV implemented in the previous project. Contrasting to our LB based approach in P2, our system might have higher MTTR but provides a higher MTBF. Further, we note that there is a cost that we pay, by routing all PUT requests via the master, in terms of the throughput of the system (due to additional hops) - 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] Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, 1982.

[4] S. Gupta and M. Sadoghi. Easycommit: A non-blocking two-phase commit protocol. In M. H. Böhlen, R. Pichler, N. May, E. Rahm, S. Wu, and K. Hose, editors, *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 157–168. OpenProceedings.org, 2018.

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

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

[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.