RecoverKV

CS 739 - Project 1 Report

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

Partner group

- Sujay Yadalam Sudarshan
- Surabhi Gupta
- Sambhav Satija

1 Design and Implementation

Our key-value store uses a simple client server model to serve requests to multiple clients. Our KV server is implemented in Golang [1] and has two components, the in-memory hash table and persistent back end storage. The hash table with key type `string` and value type `string` is a default map supported by the Go language. Although the map does not offer thread safety guarantees we still opted to use to due to its low lookup and write overhead. To make it thread safe we use mutual exclusion `sync.Mutex` provided in Golang standard library.

The server supports two public APIs, `GetValue(ctx context.Context, in *pb.Request)` which returns the value of a given key provided in request and `SetValue(ctx context.Context, in *pb.Request)` which sets or updates the key and value provided in request object. Both respond to the client with `successCode` of 0, 1 or -1 indicating the status of the request. All the key values are stored in the hash table at all times and we expect the table to not grow beyond the size of available memory. Both Get and Set operations directly modify the map before returning to client, however only during Set, we invoke the persistence module to store the key on disk. Each incoming request spawns a new thread at the server, so it is capable of handling a high number of concurrent connections.

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. However, with the combination of `NORMAL` and `WAL` our KV store is susceptible to OS or hardware crashes. One way to avoid this would be to use `FULL synchronous` mode which does `fsync` after each db write. Finally, once the server recovers from crash it loads the stored key values from the database to our in-memory map.

2 Protocol Specification

We used the Google RPC (gRPC) [2] opensource framework for communication between client and server. While the server was based on the *gRPC Go* plugin and the client interface was developed using *gRPC C++* plugin, we used a common protocol buffer configuration file implementing the `getValue` and `setValue` interfaces. The `getValue` method takes input a `key` object and returns the corresponding `(successCode, value)` pair. Whereas the `setValue` takes in `(key, value)` pair as input, and returns a `(successCode, oldValue)` pair.

It is worth mentioning here, that we use **Unary**

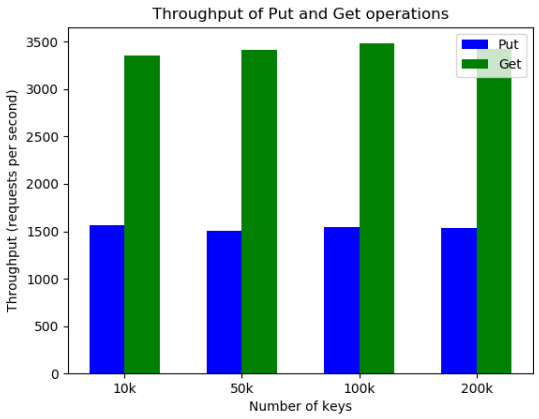


Figure 1: RecoverKV: Server throughput with varying number of keys

RPCs from gRPC, on a synchronous client. While asynchronous clients or Streaming RPCs could be used, we found that Unary RPCs are not only easier to implement, but also offer more flexibility while testing for multiple clients, and observing the bottlenecks.

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 [5]. 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.

3 Testing Methodology

We implemented the following three test suites:

3.1 Correctness Tests

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

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.

The second set of tests verifies if the put requests to the servers, followed by get requests, return the same set of values. Using a user controlled parameter of number of requests, we send a batch of put requests to the server where each value is derived from the `pid` of the test process. We test for both unique and repeating set of keys. The put requests are immediately followed by write requests, for validating if the values that were updated for the most recent set of keys, matches to what is received from the server. 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.

Finally, our third set of tests verifies the durability of key values inserted to the server. Here after inserting a range of key values to the server we prompt the user to kill the server manually and restart it. The test waits for a given period for the user to complete his actions after which it resumes to establish a connection and attempts to read all the keys it had stored prior to the crash. The values are then compared to the test's local copy and verify if they all match.

3.2 Single-Client Performance Tests

We implemented the single-client performance test suite with the aim to evaluate the performance of the key-value store when a single client is connected to the server. We evaluate the latency and throughput for `Put` and `Get` operations under scenarios of various workloads. Specifically, we use the following three workloads for this purpose.

1. Simple Writes and Reads (SimpleRW): The first workloads consists of a set of put requests, sent consecutively followed by a set of get requests, one for each key that was put into the KV-store. The number of keys to put on the store is taken in from a user argument. We compute the latency of each

request and report the average latencies for `Put` and `Get` requests separately. Further, we evaluate the throughput for the two operations as well. The throughput is calculated as follows:

$$Throughput, \mathbf{T} = \frac{R}{t} \quad (1)$$

where R is the total number of requests served and t is the total time, recorded for executing R requests.

2. Exponential Reads (ExpR): The second workload is aimed to evaluate the performance of the KV-store under skewed workloads. A set of `Put` requests writes a user-provided number of keys on the KV-store. We use an exponential distribution to sample the keys that are read in the `Get` requests. Specifically, we use an exponential distribution with parameter 3.5, to sample numbers between 0 and 1. This number is then multiplied by the total number of keys, to obtain the index of the key to be sampled. We measure the latency and throughput for the `Get` requests using the same methodology as described in **SimpleRW**.

3. Batch Writes and Reads (BatchRW): The third workload is aimed to evaluate the impact of caching on the metrics of latency and throughput. For this workload, the total set of generated keys is divided into batches - the batch size being a user-controllable argument. The workload consists of multiple cycles of batch `Put` and `Get` requests. A batch of keys is chosen, and `Put` requests are sent for each of the keys in this batch. This is followed by a batch of `Get` requests, for the set of keys that were written in the current batch. The cycle is repested for disjoint batches, until the entire set of keys have been used. The latency and throughput are measured using the same methodology as described in **SimpleRW**.

3.3 Multi-Client Performance Tests

We also evaluate the performance of our KV-store under conditions of multiple clients sending requests simultaneously. The workload is same as the **SimpleRW** workload described in section 3.2 - a set of `Put` The number of concurrent connections to the server can be provided by a command line argument. Further, the user provides the total number of keys that would be used for the workload. The entire set of keys is partitioned between all the concurrent clients in a round robin manner. Once all the clients have completed their writes, then a set of `Get` operations is performed, again partitioned equally among all the clients. The latency for each request is recorded separately and a master thread evaluates the observed throughput of the KV-Store, by measuring the total time for all the concurrent requests to finish. Here as well, the metrics are evaluated separately for `Put` and `Get` operations.

Cloudlab (`c240g5`)	
CPU	2x Intel(R) Xeon Silver 4114 2.20GHz 10-core
Memory	187GB DDR4 1866 MHz
Storage	447GB SATA SSD 100GB 10K RPM HDD
OS	Ubuntu 20.04 LTS

Table 1: Machine Specifications

4 Results & Discussion

All the bellow tests are performed on Cloudlab [3] machine, whose specification is provided in Table 1

4.1 RecoverKV Results

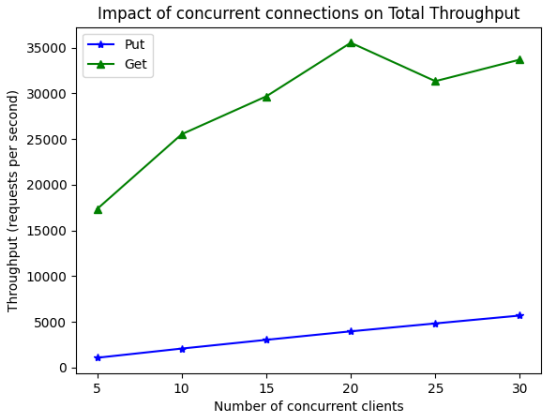
We carried out the above described performance and correctness on our server.

4.1.1 Varying Key Size

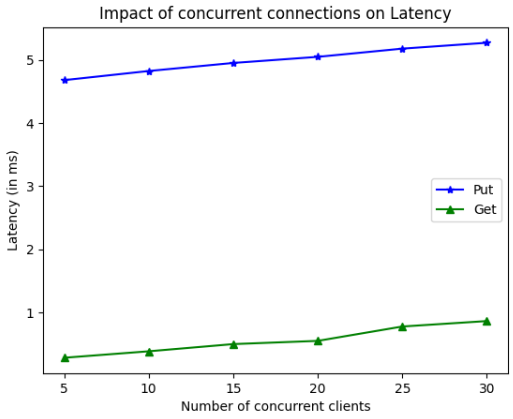
Here we increased the number of keys by 50 thousand on each iteration to assess if hash table performance varies in relation to the number of keys stored in it. First figure 1 shows the average GET throughput is **4200 ops/sec** and PUT is **1500 ops/sec**. The figure also demonstrates that the hash table performs consistently for different sizes and we even see our GET throughput slightly increasing as the number of keys increase.

4.1.2 Varying Batch Size

Figure 3 shows the server throughput on varying the total count of GET and PUT requests (batch



(a) Throughput vs clients



(b) Latency vs clients

Figure 2: Partner KV: Throughput and Latency behavior with multi-clients

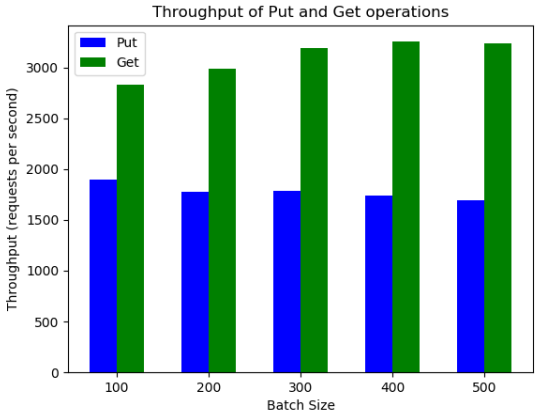


Figure 3: RecoverKV: Server throughput with varying request batch size

size). As the total number of requests are increased, we see that our GET throughput slightly increases, whereas we see a slight drop in the PUT throughput. The GET latency reduces from 0.35ms to 0.30ms on increasing the batch size from 100 to 500, while the PUT latency increased from 0.52ms to 0.59ms.

4.1.3 Multi-Client

We cautiously chose Golang as our server code due to its well touted concurrency performance and easy RPC support. This along with our mindful use of mutex locks has resulted in reasonable performance. This suggests that a single instance of our KV server can scale well as we add more cores to it. Figure 4 (a) shows there is 4x overall throughput gain with just 6x more clients, illustrating that the server can support large number of synchronous clients at any point. However, Figure 4 (b) shows the gradual increase in latency as more clients access the server concurrently. We should also node that the relative percentage of throughput gains decrease as the number of clients increase while latency observed by each client steadily increase.

4.1.4 Access Distribution

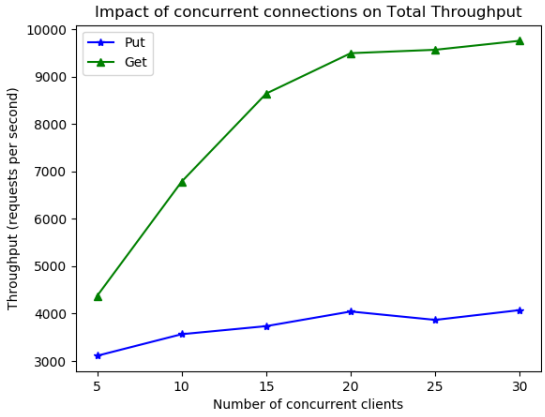
As shown in Figure 5, we observed that varying client read distributions from uniform to zipf workloads does not have any significant impact on the server GET response performance.

4.1.5 Correctness Tests

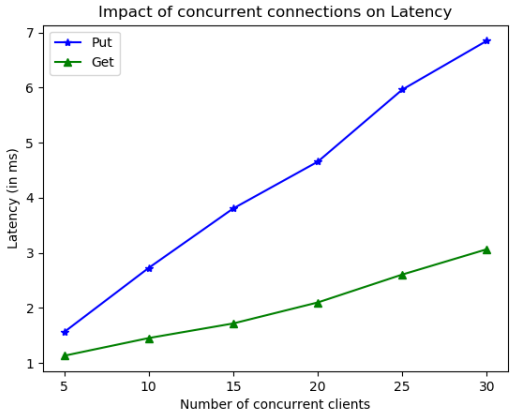
We described our correctness tests in section 3.1. All of the defined tests were passing correctly. For durability and recovery tests, the server was able to load the key and value pairs successfully from the persistent store into the in-memory index structure.

4.2 Partner Group Results

The following set of results are derived by testing our partner group's KV-store, using our test suites.



(a) Throughput vs clients



(b) Latency vs clients

Figure 4: RecoverKV: Throughput and Latency behavior with multiple clients

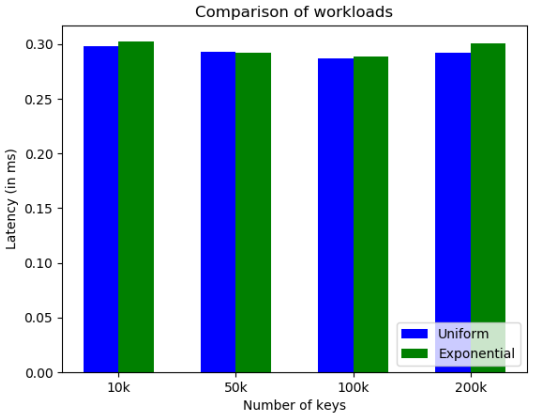


Figure 5: RecoverKV: Server latency with varying read workload distributions

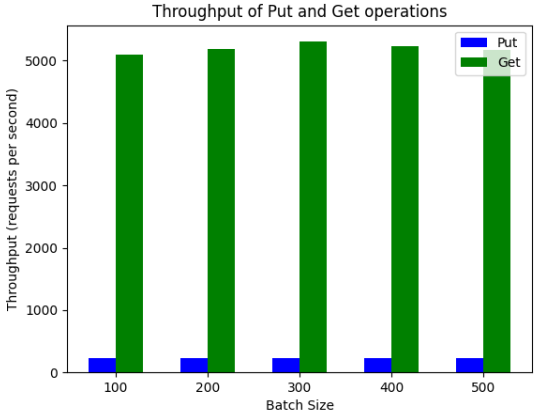


Figure 6: Partner Group: Server throughput with varying request batch size

4.2.1 Varying Key Size

We observed as the key sizes increase their write throughput increases, where as read throughput have gone down.

4.2.2 Varying Batch Size

Figure 6 shows our partner group server throughput on varying the total count of GET and PUT requests. There seems to be a significant difference between the read and write throughput numbers. The PUT throughput is around 200ops/s whereas the GET throughput is approximately 5000ops/s. This suggests that our partner group server implementation favors more GET performance, while their PUT request latency is comparably very high (4.5ms).

4.2.3 Multi-Client

Overall, Figure 2 shows their system scale better than our implementation. However, they do saturate at 20 clients for GET request throughput. Their per clients latency grows steadily as more connections are added.

4.2.4 Access Distribution

Their zipf did better than uniform reads distribution. Since their write throughput is extreemly slow, our tests were not able to finish in time to verify if holds true for larger key sizes.

4.2.5 Correctness Tests

All of our correctness tests successfully passed on the partner groups server instance.

5 Conclusion

We foresee our server performance can be improved if we avoid more locks on our data structures and perhaps build a batched writer to the file system instead of using SQLite for our storage engine. We expect using `syscalls` would do better than SQL queries. Nevertheless, in comparison with our partners system, we demonstrated that our KV server provides extremely good write throughput even during high load. Our single threaded reads are comparable with their implementation although their multi client read performance is better. This could largely be attributed to the performance difference between locks in Golang vs C++17.

References

[1] GoLang. `http://golang.org/`, 2021.

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

[3] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, et al. The design and operation of cloudlab. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 1–14, 2019.

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

[5] S. Popić, D. Pezer, B. Mrazovac, and N. Teslić. Performance evaluation of using protocol buffers in the internet of things communication. In *2016 International Conference on Smart Systems and Technologies (SST)*, pages 261–265, 2016.