Paul Hemberger
Andres Romero
Kevin White
6.824 Final Project

Default Project, yet another.

We implemented the default project of a persistent, fault-tolerant, and high-performance key-value store. We increased average response times by 34%, achieved persistence across a wide range of failures and implemented a leader election protocol for Multi-Paxos.

We will summarize our approach to each aspect of the project, as well as highlight areas for further improvement.

Persistence

Persistence was implemented with JSON encoding and modifications to Paxos and ShardKV. For the Paxos log, a FileManager interface was written as an API for reading and writing Paxos RPCs as they arrived. The ShardKV had a separate interface named PersistenceMap, which functioned as a caching out-of-memory map.

We wanted an uncomplicated persistence scheme that would guarantee full recovery of a crashed server, regardless of when it failed. It is particularly important that no changes are ever lost, since Paxos and ShardKV maintain several invariants over their state variables, and losing any of these states would break their correctness.

In order to create a persistent Paxos system, we created the FileManager interface. The FileManager performs CRUD operations on Paxos log files. JSON encoding was used in order to marshal and unmarshal Paxos instances between memory and the hard disk. On recovery, when Paxos is a part of a larger application, that application is responsible for passing the last known done value to the Paxos instance. From there, the FileManager will grab the corresponding files and recreate the Paxos log for that particular replica. Recalculating the Min also erases old log files, which are separated by groups of 100 sequence numbers.

In ShardKV we started by writing every *(key, value)* pair to disk on writes, and reading directly from disk on reads. This was functional, but not quite as fast as we would like on Gets. To improve performance, we added a transparent cleaning cache. On a Put or a Get, the *(key,value)* pair would be placed inside the cache. The cache is given a memory limit by the Go runtime, and when cache.Clean() is called it deletes members of the in-memory map to keep the memory of our Go instances in check. This allows us to hold arbitrarily large out of memory shards, (such as the specified 100Gb shards) without breaking our memory use. Recently used keys would still return for Gets just as quickly as in the original lab 4b.

To avoid unnecessary overhead, we removed the ability to move shards within the system. Once shards are not guaranteed to fit in-memory, transfer of shards becomes difficult, and would have required an overhaul of the entire ShardKV system. Instead, we opted to use a cryptographically secure hashing function to separate the keys into shards. This should, in theory, even the load between shards entirely, and therefore remove some of the need to move them. This is a place where the system could be improved in the future.

Multi-Paxos

Paxos was optimized using the suggested Multi-Paxos changes. A distinguished leader is elected that proposes all values for the Paxos log. All other replicas forward any proposals they receive to the distinguished leader. The distinguished leader can then run an abbreviated Paxos algorithm by sending Prepares to all servers once elected, and then immediately Accepting any further proposals.

Leader election can be a cause for concern in Multi-Paxos, since in the case of a leader failure, many peers may want to propose an election. To keep this number sane, each peer who notices a leader failure performs two steps: first, it pings all servers in its replica group; second, if the lowest responding server (ordered by IP) is itself, and it received a majority of responses, it will propose itself as the new leader. This potentially reduces the number of elections to one, from the lowest IP server in the majority.

Leader failure is observed through two channels: periodic pings, and successive failures to forward proposals. Each peer sends heartbeat pings to the leader, and if the leader fails to respond for too long, they will attempt an election. Leader failure is also observed when a peer tries to forward a proposal to the leader, but fails several times in a row. These two mechanisms help keep replicas up to date regardless of whether or not they are actively trying to reach the leader.

Leader elections are entered into the same Paxos log as all other values. When a Learner observes an election, it updates its known leader. Placing the elections in the same log as all other values has one very beneficial property: when a new election is run, which is only when a peer in a majority is convinced the leader has died, the highest proposal number observed increases. This means that when the dead leader comes back, any fast-tracked Accept messages it sends will be rejected since its proposal number is too low. This design allows for the leadership to change transparently during failure.

Some care must also be taken to handle instances that were proposed before a leader election. A leader will move any proposal it receives that has a sequence number less than when it became elected to traditional two-phase Paxos. Peers similarly will tell the leader to use traditional Paxos if they already have values for the proposed sequence, implying that some

agreement may have occurred before the new leader's election, that the leader may be unaware of. This prevents the leader from ever overwriting previously accepted values, despite the omission of the Prepare phase.

Fast Reads

A further optimization we made for performance was the fast reads protocol from Google's MegaStore[1]. Fast reads allows any sufficiently up-to-date replica to respond immediately to Get requests, and skips placing the request into the Paxos log.

To reach this goal, a ProposalMaster is assigned to each replica group. The ProposalMaster keeps track of the highest seen committed Paxos log entry. Upon receiving a Get request, a replica checks the ProposalMaster to see if its highest committed entry is greater than the ProposalMaster's. If it is, then the replica must be fully to up to date, and can immediate service the Get, without need of a Paxos entry.

The ProposalMaster stays up to date by having each replica send its latest interpreted log entry immediately after completion.

For our purposes, we assume as in the labs for the viewserver, that the ProposalMaster service is stable. In a real-world system, the ProposalMaster could be designed with a primary-backup scheme, or alternatively, if the ProposalMaster fails, the replicas could fall back on traditional log entries for Gets.

Testing & Benchmarking

There are unit tests for every previously mentioned new module in our project. These were made to isolate problem areas and ease problem debugging. Additionally, we created integration tests for each aspect of the project. These include the suggested test scenarios for persistence, as well as a leader-aware Multi-Paxos test.

Finally, we added benchmarking tests to compare our final solution with the original Lab 4b. The results are included in the following table:

Environment	Average Response Time: Concurrent Puts and Gets	Average Response Time: Concurrent Gets Only	Average Response Time: Concurrent Puts Only
Lab4b	85 ms	85 ms	78 ms
Final Project	56 ms	23 ms	126 ms

Environment	Min (ms)	Median (ms)	Max (ms)
Lab 4b	Put: 47 Get: 58	Put: 135 Get: 143	Put: 956 Get: 429
Final Project	Put: 60 Get: 1.2	Put: 144 Get: 1.3	Put: 803 Get: 50

Notice that Gets have a sizeable speed up, but the overall system doesn't outperform Lab 4b by a large margin. This is because we have the increased overhead of persistence, every Paxos decision must have at least 3 writes to disk, and every Put RPC must have a Paxos decision and an additional write.

Areas of Improvement

In both Paxos and the ShardKV server there are locks and writes that tend to be greedy and span multiple variables. This was done for development convenience, but separation would improve performance. A Golang CPU profiling run indicated we spent 10% of our CPU time waiting on locks, and another 10% on writes, both the top two CPU spenders. Changing to read-write locks, and using locks with greater granularity could improve our locking bottleneck.

Even though Multi-Paxos is better performing than the original Paxos, it does introduce the bottleneck of the leader. If a slow machine is elected leader then it could even become slower than leaderless Paxos would. This was not shown directly by our local tests, but we acknowledge it as an area of improvement of this design.

The leader election protocol is dependent on developer-provided parameters that decide the time required to observe and elect a leader. For instance, the current design will attempt a leader election if a forwarded proposal fails five times successively, or fails to ping it over 3 seconds. Decreasing these values would increase the speed of leadership change, but at the cost of more bandwidth consumed. There is no one-size-fits-all way to decide these values, since they are dependent on the network's bandwidth and latency. As such, performance in our tests could be improved by optimizing these values, but may not improve performance if we deployed elsewhere.

[1] http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf