6.824 Final Project Report

Geoffrey Lalonde (glalonde@mit.edu) Andrew Huang (kahuang@mit.edu)
Project Repository: https://github.com/jefesaurus/6824final

In our final design and implementation of a persistent, fault-tolerant, high-performance key/value store we decided on several of the goals stated in the project assignment. Namely, using multi-paxos with leader election to remove dueling leaders under high load, removing the prepare phase of paxos when operating under a consistent leader, better memory utilization for our shardkv service's configuration snapshots, persistence for our paxos library, and persistence for our shardmaster service.

The generic paxos library that we implemented in lab3a requires some modifications and optimizations in order to streamline the consensus protocol for highly-performant applications. We address two main issues: dueling leaders under high client load and reducing the necessary round-trip delays by one by removing the paxos prepare phase.

Multi-paxos is a variant of paxos where one paxos peer acts as the leader of the replica group. The leader is the only peer who can push paxos instances to agreement, and as a result all peers who are not the leader forward their requests to the peer that they believe is the leader. The leader is elected with the following protocol:

1. All peers ping each other every *PING_INTERVAL* seconds

2. Every *PING_INTERVAL* seconds, each peer determines who the leader is: the highest index peer whom they have received a ping from in the last PING_INTERVAL * 5 seconds (which may be themselves).

3. A peer can determine that it's the leader iff it has contacted a quorum of peers in the last *PING_INTERVAL * 5* seconds.

Assuming that peers' clocks are reasonably synced, this protocol ensures at most a single leader at any given time. As an optimization, if an instance is already decided and a client starts a proposal, the leader ignores the client and returns. Because of this optimization and the fact that only the leader can push an instance to agreement, some peers may not hear of decided values if the network is unreliable. Therefore, when a non-leader peer is contacted by a client to start an instance, it spawns a goroutine that tries to determine the decided value once the instance has been agreed on.

The prepare phase of paxos provides consistency guarantees that are trivial for multi paxos when there is a consistent leader. However, when the leader dies and a new leader is elected, these consistency issues become problematic without a prepare phase. In order to maintain correctness and consistency of our paxos library, we revert back to our original paxos implementation for the set of instances that the previous leaders had controlled. We determine the maximum paxos instance by contacting a majority of peers and taking the maximum of their highest known instances. For any instances above this maximum, we are able to skip the prepare phase again.

With the addition of these two modifications, we were able to approximately double the throughput of our paxos library. Our three throughput tests consisted of the following: 50 concurrent operations, 10,000 operations done in 200 batches of 50 operations, and again 10,000 operations done in 200 batches of 50 operations but with simulated network propagation delays for rpcs. The results are shown in Figure 1.

	50 Ops	10,000 Ops	10,000 Ops w/ Delay
Paxos	165.121886ms	25.700564218s	1m3.028979029s
Multi-Paxos	76.615355ms	14.715619431s	46.097839948s

Figure 1

In order to handle hundreds of gigabytes of data, it was necessary to reduce the memory overhead of our shardkv implementation. To handle the movement of shards across replica groups, our implementation stored snapshots of shards per configuration number. This scales very poorly with both configuration number and shard size, so we implemented a forgetting scheme. Each shardkv replica group determines the minimum configuration number of the replica group by using the min function of its paxos peer.. Each shardkv server then contacts at least one server in each of the other shardmaster groups and exchanges minimum configuration numbers. When this minimum is advanced, a replica group can delete the snapshot of shards for that configuration number. If all replica groups are up to date with the latest configuration, we incur no extra memory overhead.

Our final addition to this project was to make our paxos library and shardmaster service persistent under two cases: having a minority of peers crash but maintain disk and all peers dying and the service coming up again. It was important to us to make these persistent as our paxos library can be used for a variety of applications and the shardmaster service is a single critical point in the design of our system. We implemented persistence with the external library LevelDB (https://code.google.com/p/leveldb-go/) which is a performant database key value store. We initially chose it because it is actually written in Go, as opposed to most others, which we hoped would make the interface more natural and easy to use. For the most part this turned out to be correct, but in retrospect, we regret choosing this particular implementation because it is quite young, has almost no documentation, and is missing some major features. One example is that it isn't possible to iterate over all of the entries in the database. The result was that in several cases we had to serialize a golang map as a keyset in one of the actual keys, and then manually keep it fresh with batch write and deletes.

Our persistence strategy was to identify the critical state and then replace all of the in-memory data structures with calls to a database on disk. The idea was that the database would be able to maintain its own read/write logs and rollback procedures to deal with atomicity for writes to multiple rows or for large values. It sounded simple enough at the start but ended up causing a fair amount of trouble because neither of us have any practical experience with databases. For paxos we persisted to disk only the bare minimum of data required to start up the process from the database. This mostly consisted of a few pieces of static data about the overall state of that paxos replica and instance data. In order to be able to rejoin a paxos consensus group we stored the addresses of the peers and the name of the actual paxos member. For each peer, we persisted the minimum instance number, maximum instance number. Most importantly we also store all of the paxos instance data known after min. Upon crashing and restarting, we check our database for the metadata to provide our peer with context in order to communicate with our peers. Upon starting up, the peer looks for a database, and does some consistency checks to make sure the required data actually exists. After that it reenters the main startup sequence.

The shardmaster persistence was predicated on our paxos persistence. That is, we stored some static metadata for context, but most of that values were in a few key/value name spaces for configurations, and query replies.

Through implementing multi-paxos, removing the prepare phase of our paxos, forgetting old snapshots of shardkv shards, and persisting our paxos peers and shardmaster service we were able to achieve a high-performance, fault-tolerant, and persistent distributed key/value store system.