6.824 Final Project Design Document

Sumit Gogia, Kevin Y. Chen, Michael Xu, Tommy Zhang

May 2014

1 Introduction

The problem addressed in this project was that of building a persistent, fault-tolerant, performant key-value store. Serving as a foundational storage mechanism for many different large-scale applications, as evidenced by the vast number of applications dependent on services providing key-value storage such as MongoDB and Redis, it is essential that a key-value storage system satisfy the properties listed above.

Link to code: simple.mit.edu/6.824/

2 Design

Our system employs established techniques in distributed systems to provide the properties desired. Persistence is accomplished with a logging system that allows machines that have failed to reestablish their state by examining the log; large data management is provided both through sharding the store across many machines in memory and on disk; performance and fault-tolerance obtained through a memory-handling system similar to caching which helps serve common requests very quickly, along with an optimized version of the Paxos protocol, Multi-Paxos, to improve the communication bottleneck present when using Paxos for replication. The design and implementation details of each of these systems follows.

2.1 Fault-Tolerance and Persistence

The system provides persistence by maintaining, for each server, logs on disk recording information for each operation the server performs. On recovery, so long as its disk is maintained, a server then only needs to roll back state to the last operation which was fully executed. The system strives to provide strong persistence, at some cost of throughput and recovery time. Much of the persistence comes from logs on disk, which utilize Google Go's gob library for performance and efficient memory usage.

Paxos is made persistent with two standard Write-Ahead-Logging systems. A logging directory is made for each Paxos peer. In this directory, a separate file is created for each Paxos instance, to store necessary variables. The largest proposal number, accept number and accept value are written to this log befo re responses are given to RPC's. This ensures that all relevant Paxos state is persistent before other peers are notified; preserving correctness. A single file is used to log all Paxos decisions. Each time a decision is made, the result and sequence number are appended to the decision log file. Upon restart, all Paxos instance variables and all decisions on disk are loaded back into memory. In handling incomplete writes to log, the server catches an error thrown by the log decoder and tags the operation as partially committed. Invalid decisions and instance logs are not loaded into memory.

The ShardKV server persists several items to disk: 1) A write-ahead log of applied operations 2) The key-value store 3) Snapshots of the store's state upon reconfiguration. Any operation that the ShardKV server performs is fully logged. Operations are both logged and applied to a persistent database in a single thread. This thread ensures that the operation log and persistent data store state differ by at most one operation. This property allows the server to recover from main memory loss in a relatively simple manner. The key-value store exists in persistent storage, along with snapshots of the server state at different configuration versions. A dangerous fault only occurs if main memory is lost in the middle of a write to disk. In this case, an operation must be invalidated. Before an operation is committed to persistent memory, any state required to undo the operation is logged in an `Undo Entry`. The entry indicates that an operation is about to be committed to memory. After write-through, an *Operation Entry* is written to the log. An `Operation Entry` comprises a complete description of the operation (Type, Key, Value, etc.). An operation is fully committed when both `Undo Entry` and `Operation Entry` are present in the log. A missing or invalid `Operation Entry` indicates an incomplete operation, and results in a rollback. For Put and Get operations, rollback involves resetting the affected values in the key-value store and deduplication map; the data is available in the logged `Undo Entry`. A reconfigure operation, which requires the movement of entire shards, is surprisingly simpler to invalidate.

Incomplete reconfigurations are dealt with by resetting server state to the previous *snapshot*. A *snapshot* stores all shards and deduplication state up to any reconfiguration. Since a *snapshot* is stored before any shard movement begins, this strategy is correct, if not terribly performant.

The Shardmaster does not keep a local log. Since the Shardmaster simply creates new configurations for each operation, rather than allowing modifications as ShardKV does, persistent storage of the configurations is not required. To recover from main memory loss, the Shardmaster reconstructs the list of configurations using the Paxos log. All logged operations are executed sequentially, resulting in a duplicate of the pre-crash state. Paxos's persistence is enough to ensure Shardmaster persistence and fault-tolerance.

Along with main memory loss, all systems can tolerate the loss of a single servers disk contents, given that there are at least three Paxos peers. Since each entry in the Paxos log is guaranteed to be in the persistent storage of at least two of the three peers, the union of any two machines is guaranteed to contain the entire operation history.

2.2 Large Data Management

Applications often require very large storage space, so that each server providing the storage needs to be able to serve hundreds of gigabytes of data. Sharding can help to provide fast access to large amounts of data, but as each server will only have a few gigabytes of memory, it does not completely solve the issue of needing to serve hundreds of gigabytes per server. The method by which the system solves this issue is by allowing the disk for each server to be used as storage space alongside memory, with a caching mechanism determining which key-value pairs to serve quickly from memory, and which key-value pairs are less important and can be stored on disk.

This portion of the system was implemented as an LRU cache holding the portion of the key-value store in memory, which communicates with a local instance of MongoDB that holds our on-disk storage. Each time get or put operations are serviced, the results are brought into the cache; in the case of puts, results are also placed into a write buffer which is written to disk by a background thread. It is assumed that the time required for any writes in the write buffer to be applied is less than the time it takes for the results of that operation to move out of the cache, since the cache is assumed to span a memory size on the order of gigabytes, and most accesses to the storage are on a relatively-small set of keys.

There were also several changes that had to be made to the initial strategies developed in lab 4b for handling the necessary RPC calls. While get and put requests could be handled by just replacing the memory accesses with corresponding requests to the data structure handling disk and memory accesses, the reconfiguration operations required significant changes, since sending an entire shard, consisting of both memory and on-disk contents, would require reading the entire shard into memory before sending. To surmount this, the system has the receiver ask for fixed-size pieces of a shard until the server it is receiving from indicate that the shard has finished sending. A server can ask for different pieces of a shard from different servers; since we keep an ordering on the key-value pairs, the receiver can request according to the amount of key-value pairs it has received, and any sender will know which portion of the shard needs to be sent next.

2.3 Performance Optimizations

2.3.1 Paxos

The system makes heavy use of the Paxos protocol for decision making and consensus within the shardmaster and replica groups. In the basic Paxos protocol, each Paxos agreement round requires at least 2 round trip messages between the proposer and acceptors. In order to decide on a value, the proposer must first send out a round of prepare requests before sending out the accept requests (and wait for acknowledgements in both cases). The prepare requests are necessary in order to maintain correctness of the protocol; if proposers skip the prepare phase, then the value decided on may not be unique.

However, the Paxos protocol retains correctness as long as at most one proposer skips the prepare phase. This means that in the case that values are only proposed by a single proposer, the Paxos protocol can be optimized by requiring only 2 round trip messages for each consensus round. We implemented a version of multi paxos which makes use of this observation by allowing elected leaders to propose values by skipping the prepare phase.

Leaders are elected every `LeaderLifetime` rounds (the default lifetime is 20 rounds). Every round r with $r \equiv 0 \mod$ `LeaderLifetime` is designated as an election round. The Paxos peer which wins an election round r is then designated as the leader for the next `LeaderLifetime` rounds. During the rounds in which a Paxos peer is the designated leader, it is allowed to propose values by skipping the prepare phase, whereas all other Paxos peers must propose values using the normal Paxos protocol.

To implement this, we augment the consensus protocol to decide on a tuple of the form `(value , id)` containing both a decision value and the id of the Paxos peer who proposed it originally. The Paxos peer is considered the winner of a round if its id was decided upon.

Note that Paxos agreement rounds are not guaranteed to proceed in sequential order (i.e. the Paxos decision for some sequence number $i+1$ is allowed to start before round i is complete). In order for a Paxos peer to determine whether it is the leader in some round i, it checks to see if round r was completed and it was the winner of round r, where r is the latest election round.

In order to make use of the multi paxos protocol, clients try to contact the leader of the Paxos family in order to propose a value, and try other Paxos peers only if the leader is unreachable. For example, in order to perform a read/write operation on a shard, clients will first attempt to contact the leader of the replica group responsible for the shard before contacting the other servers in the group. The key value servers include the leader of the replica group in their response to the client so that the client learn the leader of each replica group.

With this implementation, Paxos agreement rounds generally require only 2 round trips of messages since clients will all attempt to contact the leader of a replica group. In the event of failed links or network partitions, the Paxos agreement will revert back to the basic protocol, but this optimization means that the steady state in a stable network configuration is much faster.

3 Results and Discussion

3.1 Fault-Tolerance and Persistence

The system was shown to be very capable of handling many different types of failure, with the only exception being the case when there multiple disk failures within the same replica group. Tests were made to determine the effectiveness of the system in the cases of network partitions, network failure and unreliability, as well as crash and memory losses. These tests were created by using the tests for lab4b, as well as adding hard resets throughout these tests which would also reset the memory of the working server. The logging protocol, coupled with Paxos, allowed the system to handle all the scenarios well; however, this was at the price of performance, since logging requires many disk accesses. Without any other of the improvements on lab 4b in place, the system slowed to nearly 10% of the speed it previously had, those these tests were run all on a local machine. It is expected that the relative performance to the case where persistence was not implemented would actually improve in the case that the system was tested on multiple machines, as the overhead of communication between the servers would also be heavy.

3.2 Large Data Management

Our system was also shown to handle large amounts of data well through our tests. Tests were developed to measure the correctness of the system under the case of all operations that were present in lab 4b, applied in heavy amounts; the system passed these without suffering significant performance drops due to the fact that it writes to disk via a background process. Performance of course suffered when the requested keys were not in memory, since the disk accesses required far more time than the memory access - however, we do expect this scenario to be infrequent, indicating that the overall performance of the system should be good. In conjunction with the fault-tolerance and persistence system, this system did not slow performance significantly.