6.824 Project Report [Default Project]

May 11, 2014

Peter Krafft, Jeremy Sharpe, Andrew Wang

We extended Lab 4 to handle server crashes and restarts with and without losing disk. We implemented the Mencius protocol to optimize Paxos agreement, and we tried to handle a large KV store by keeping important data structures on disk. In this report, the first section presents our design for these features. The second section describes the additional test cases we aimed to pass. The final section shows our benchmark analysis.

1. Design

Our code base is built on top of Lab 4, which includes the Paxos, Shardmaster, and ShardKV packages. We assume the Shardmaster is reliable, so we apply our modifications to the Paxos library and the ShardKV state machine. Handling persistent state is relevant to both components. The Mencius protocol mainly affects Paxos, plus minor changes to how ShardKV accesses underlying Paxos instances. Finally, dealing with large data requires proper memory management within both Paxos and ShardKV.

To recover from crashes, we identified the Paxos and ShardKV state necessary for continued operation. Wherever this state is updated in memory as part of an operation, we atomically commit it to disk too. When we kill a server, we either wipe the disk or keep it around, and during restarts, we check whether a disk is available, whether locally or from a peer.

The relevant Paxos state includes the done vector and individual instances. A Paxos peer's done vector enables forgetting by listing the latest minimum instance it knows about everyone. Both the proposer and acceptor threads update this state, and therefore write it to disk, each time they send or receive an RPC. Each Paxos instance also contains both proposer and acceptor state, but only the acceptor state needs to be saved for correct recovery. The proposer state doesn't need to be saved because Paxos peers can always reissue smaller proposals again.

Our ShardKV state machine maintains four pieces of state, which are updated by the operations get(), put(), and reconfigure(). First, we keep track of which configuration we are operating in. This is updated only by reconfigure(). Then, the actual KV state includes the key-value map and the client state for each shard, plus the execution point in the log this state reflects. All operations must update this state. Even when processing a duplicate client request, it occupies a space in the log, so the execution point must be incremented. Finally, our Lab 4 design kept old shards to be transferred during reconfiguration, in case the receiving group was slow to reach that reconfiguration.

During server restarts, disk recovery is initiated from ShardKV. If StartServer() finds that its local disk is not empty, it simply continues with loading ShardKV state, and it initiates its local Paxos

to do the same in Make(). But if its local disk is empty, then it must copy disk from a peer in its group. StartServer() cycles the request through all its peers, until someone confirms a successful disk transfer. It is important that both ShardKV and Paxos state are transferred together, since the state machine is tied to what the local Paxos has proposed or learned about. When a peer receives a disk transfer request, it must halt all operations to prevent them from modifying disk during disk transfer.

There is the special case of initial startup when no one has any disk state. Peers will still try to request disk from each other, but in this case they may explicitly reply that they have no disk to offer. Only when all other peers have responded as such, may one assume a clean initialization.

With persistent Paxos and ShardKV in place, we implemented “round-robin” Mencius to reduce the number of RPC messages. Our implementation is based on the OSDI 2008 paper; the rules and optimizations in sections 4.3 and 4.4 served as guidelines for modifying our Paxos library. The additional required Paxos state for a peer are a pointer to the next log instance it owns, the list of instances it may skip, and whether it thinks its other peers have failed. For persistence, this state is maintained similarly as the rest of the Paxos state.

In Mencius each peer issues implicit Prepares on all the instances that it owns. Therefore, it may send Accepts directly on those instances, saving an RPC round trip. We simulate the implicit Prepares by initializing instances with a non-zero proposal number. Because each peer’s subsequent Accepts jump over several instances, and peers operate asynchronously, holes may appear in the log. Mencius avoids this bottleneck by allowing peers to skip their next instance if they have heard about another peer’s proposal on a higher instance. When a peer skips one of its owned instances, it inserts the instance number in its skip list, promising to issue only a No-op for that instance. This skip list is piggy-backed and gossipped on all RPCs, so when other peers hear about it, they may incorporate it into their own list and immediately apply the No-ops.

The remaining piece of our Mencius implementation is dealing with peer failures. Failed peers can no longer issue skips on their instances, so other peers have to take ownership of them. Each peer must independently detect others’ failures. This is accomplished by gossipping each peer’s next instance number through the RPCs as well. When a peer’s own instance number has advanced too far beyond another peer’s advertised number, it assumes the other peer has failed, and it proposes No-ops on that peer’s instances. Regular Paxos is required here because all peers are performing failure detection independently, which may not be reliable, and a peer that is supposedly dead but actually alive needs to be able to issue its own Accepts.

The last piece of our project is to handle a large KV store that would not fit in memory. We assume each shard is not unreasonably large, and adapt Lab 4 to accept a large variable number of shards. Since the full KV store will not fit in memory, we must store all shards on disk. In absence of an in-memory cache, for every operation in ShardKV, we have to load a

shard from the disk, and save it back when we are done. We assume shards are small enough that reading/writing a single shard is fast.

2. Test cases

We extended the tests of labs Paxos and ShardKV to verify the correctness of our added features. Most of our tests centered around persistence. We hand designed a number of failure scenarios for both Paxos and Shardkv, and we also had a number of other tests that incorporated random failures and restarts. Our basic cases tested whether servers who crashed and restarted but kept their disk would remember the correct information that they had before crashing. Our more complicated cases tested failures with and without disk loss that occurred during agreement and reconfigurations. We also modified many of the original test cases, such as testing many concurrent operations and unreliable network connections, to include random failures and restarts. In these test, peers failed with probability 0.1 about every 100 milliseconds and recovered with probability 0.9 10 milliseconds after they crashed and with increasing probability longer periods of time after they crashed. Failed peers also lost their disk contents with probability 0.1, but we constrained the tests so that only one peer from each group would lose disk in each test. Our persistent and low-memory implementations passed these tests successfully.

We also tested Mencius and large amount of data with less success. Our Mencius implementation was able to pass all the original Paxos tests, and some of the original ShardKV tests, but it had some trouble with persistence. Our low memory implementation of ShardKV had a memory leak which made large amounts of data difficult to deal with. Even though the Paxos log was deleting instances and the shard data structures in ShardKV were empty, the memory usage of our implementation still grew too quickly.

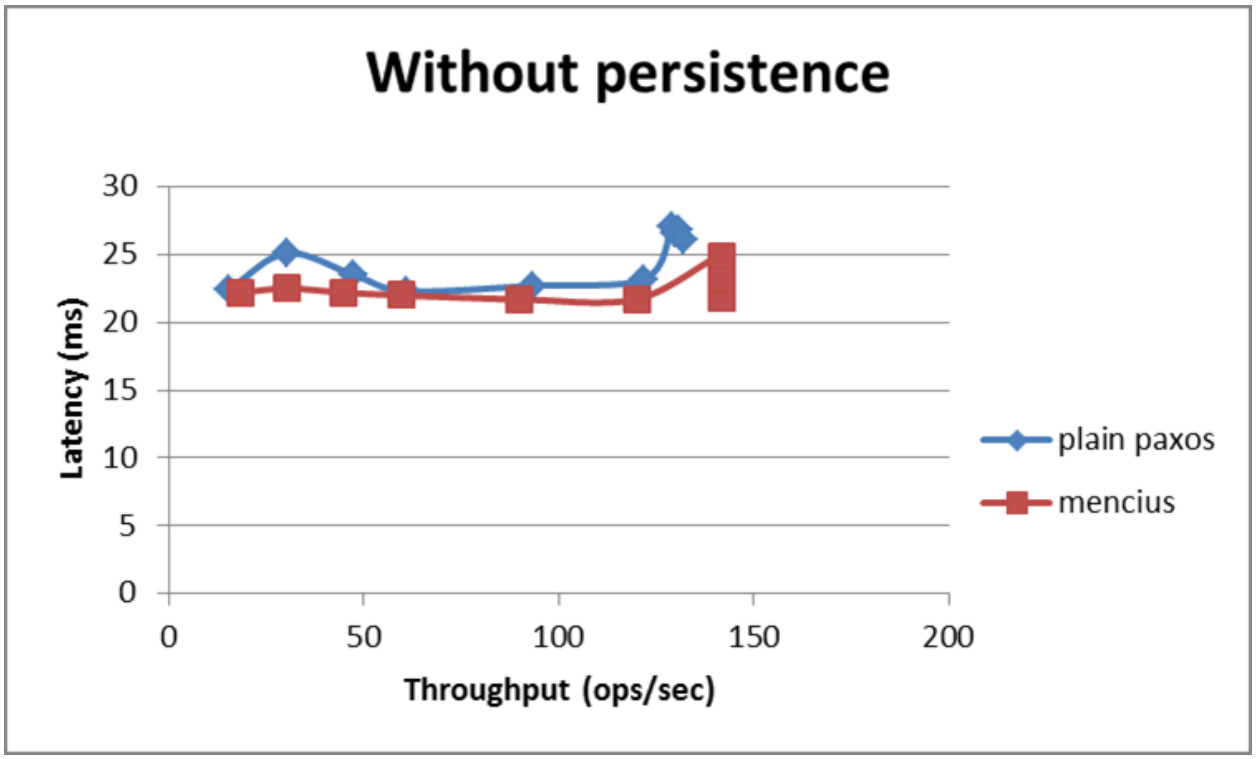
3. Benchmark analysis

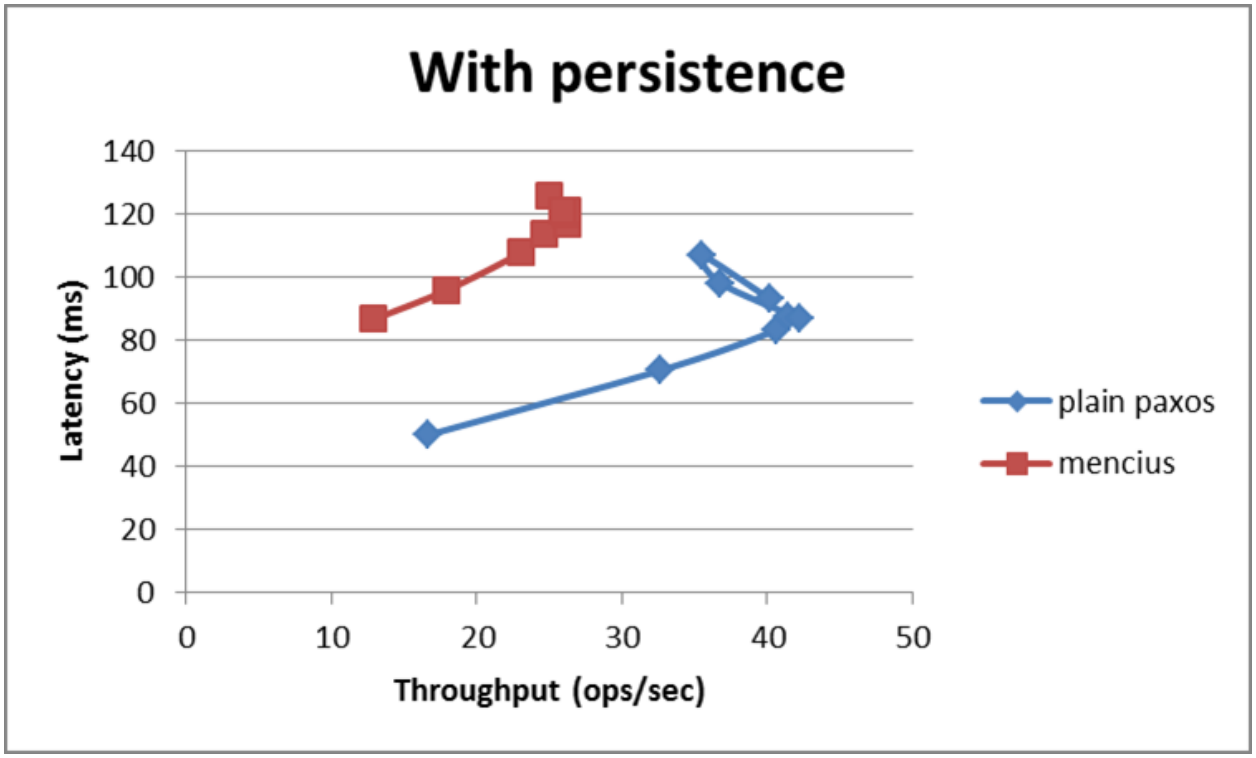
By running our code through go's pprof tool, we found that the most cpu time and memory allocation are spent in gob encoding/decoding and the RPC library. (See Appendix A.1.) Not only does RPC use gob, but our persistent implementation also uses gob to read and write disk. The the gob library accounts for the bulk of the processing, even the when we are not keeping persistent state.

Our Mencius implementation reduced the RPC count significantly. Running the RPCCount test reported 2224 RPCs for plain Paxos versus 1048 for Mencius.

The two figures below plot latency over throughput, comparing plain Paxos against the use of persistence and Mencius. Without persistence, Mencius has at least slightly higher performance

than plain Paxos. Though the difference is slight, Mencius's latency is consistently lower, and it is able to sustain higher throughput before bottlenecking, at nearly 150 ops/sec vs around 130 ops/sec. With persistent state, however, the tables are turned. Mencius's latency is now 20-30 ms longer than Paxos's, and it bottlenecks at just over half the throughput. Persistent Mencius requires additional state to be saved, and in additional functions, at the Paxos level, where most of the RPCs are occurring. Comparing the two plots, one sees the simple addition of writing to disk causes latency to rise by a factor of at least 2 to 3, and the max throughput is correspondingly decreased.





A. Appendix

A.1. pprof output

```
(pprof) top50 -cum
Total: 4527 samples
        0   0.0%   0.0%       2397  52.9% System
        0   0.0%   0.0%       2129  47.0% runtime.gosched0
        1   0.0%   0.0%       1252  27.7% encoding/gob.(*Decoder).Decode
        3   0.1%   0.1%       1247  27.5% encoding/gob.(*Decoder).DecodeValue
     1218  26.9%  27.0%       1225  27.1% syscall.Syscall
        9   0.2%  27.2%       1121  24.8% encoding/gob.(*Decoder).decodeValue
        2   0.0%  27.2%        887  19.6% encoding/gob.(*Decoder).decodeTypeSequence
        3   0.1%  27.3%        853  18.8% encoding/gob.(*Decoder).getDecEnginePtr
       18   0.4%  27.7%        809  17.9% encoding/gob.(*Decoder).compileDec
        1   0.0%  27.7%        787  17.4% encoding/gob.(*Decoder).recvType
        0   0.0%  27.7%        778  17.2% net/rpc.(*Server).ServeConn
        0   0.0%  27.7%        740  16.3% net/rpc.(*Server).ServeCodec
        0   0.0%  27.7%        691  15.3% net/rpc.(*Server).readRequest
      234   5.2%  32.9%        663  14.6% runtime.mallocgc
        3   0.1%  33.0%        593  13.1% net/rpc.(*Client).input
       17   0.4%  33.3%        588  13.0% encoding/gob.(*Decoder).decOpFor
        0   0.0%  33.3%        414   9.1% net/rpc.(*gobClientCodec).ReadResponseHeader
        1   0.0%  33.4%        379   8.4% net/rpc.(*Server).readRequestHeader
        1   0.0%  33.4%        374   8.3% net/rpc.(*gobServerCodec).ReadRequestHeader
       28   0.6%  34.0%        368   8.1% runtime.new
        0   0.0%  34.0%        350   7.7% encoding/gob.(*Encoder).Encode
```

```
       3   0.1%  34.1%       347   7.7% encoding/gob.(*Encoder).EncodeValue
       344   7.6%  41.7%     344   7.6% runtime.futex
       1   0.0%  41.7%       340   7.5% paxos.func·001

(pprof) top50
Total: 6.5 MB
       3.0  46.2%  46.2%     3.0  46.2% paxos.(*Paxos).GetInstance
       1.5  23.1%  69.2%     1.5  23.1% reflect.unsafe_New
       1.0  15.4%  84.6%     1.0  15.4% newdefer
       0.5   7.7%  92.3%     0.5   7.7% runtime.allocm
       0.5   7.7% 100.0%     0.5   7.7% runtime.convT2E
       0.0   0.0% 100.0%     1.0  15.4% encoding/gob.(*Decoder).Decode
       0.0   0.0% 100.0%     1.0  15.4% encoding/gob.(*Decoder).DecodeValue
       0.0   0.0% 100.0%     1.0  15.4% encoding/gob.(*Decoder).decodeInterface
       0.0   0.0% 100.0%     1.0  15.4% encoding/gob.(*Decoder).decodeStruct
       0.0   0.0% 100.0%     1.0  15.4% encoding/gob.(*Decoder).decodeValue
```