SPS: An Efficient, Persistent Key-Value Store

Dominic Kao
dkao@mit.edu

Helen Xunjie Li
xunjieli@mit.edu

Ruwen Liu
ruwenliu@mit.edu

Patricia Adriana Suriana
psuriana@mit.edu

ABSTRACT

Our final project is an extention of the Lab4 ShardKV store that is both faster and more reliable. Our main areas of implementation were 1) The Mencius variant of the Paxos protocol, 2) A persistence architecture which uses periodic checkpointing, and 3) A deployment framework for testing a real-life distributed key-value store over Amazon's AWS.

1. INTRODUCTION

The main goal of the project is to design a key-value store that is resistent to failures and guarantees external consistency. Our second goal is to make this key-value store fast and efficient, while providing recovery in the failure cases listed in the default project requirements. Our final project is the Mencius variant of Paxos that backs up data to disk and a data checkpoint system for ShardKV. Our disk writes in both Mencius and ShardKV allow for fast recovery in the case of failure and loads data from other replicas in the case of total disk loss. Additionally, we tested the performance of this system when run on an actual distributed system. The latter half of our project involves porting the code to Amazon's AWS service and setting up a testing framework across multiple machines. At the end of the report, we include data of the performance of our implementation.

2. MENCIUS

Since we were asked to implement a more efficient Paxos protocol, we decided to build Mencius. While Mencius is described in detail in [1] and pseudo-code can be found in the technical report in [2], there are several factors that make building Mencius challenging: 1) despite being conceptually similar to Paxos (i.e., using striped instances, skipping turns using no-ops) applying these simple optimizations requires a complete re-design and re-building of the protocol from scratch, and 2) the pseudocode can hardly be used if we want to maintain the simple API from the labs as the separations of responsibility between library/mencius/client are substantially different. For example, the project member that worked on Mencius finished Lab4B in a day, whereas building Mencius alone took on the order of 5x the time/effort. First, we had to rebuild our Paxos library (Lab3A). We maintain the same API:

```
// px = paxos.Make(peers []string, me string)
// px.Start(seq, v) -- coordinator suggests instance
// px.Status(seq) (bool, v) -- get info about instance
// px.Done(seq) -- ok to forget all instances <= seq
// px.Max() int -- highest instance seq known, or -1
// px.Min() int -- instances before this seq forgotten
// px.Revoke(seq) -- non-coordinator wants to revoke seq
```

The only difference is an added API function called px.Revoke. Start is only called on a paxos instance that owns seq (and it skips the prepare phase), whereas Revoke is called when the paxos instance does not own seq (it tries to propose a no-op). Start begins a new goroutine, and as a first step sends Paxos.Accept messages, sending Paxos.Decided if it hears back from a majority. The tricky thing here is that Start (as defined in the labs) should always decide on an instance, if it doesn't it keeps trying. This is one major difference from the pseudocode found in [2] (in the technical report, the pseudocode in Appendix C, Mencius deals with retrying if a proposed op does not succeed for a particular instance, not the library, i.e. Coordinated Paxos in Appendix A). We handle this by checking how many iterations have occurred; after the first iteration (i.e., did not receive a majority upon sending Paxos.Accept), the loop reverts to regular Paxos. In Revoke, we start a new goroutine that tries to propose a no-op (servers can only propose no-op to instances they do not own), this is analogous to the prepare/accept/decided phases in the original Paxos.

The difficult part is not in implementing the general idea, but in the little details that cause small corner-case failures. For instance, it is absolutely crucial that new instances (ones seen for the first time by the library) are initialized with n_a equal to -1 (one of our first designs decided this was unnecessary and initialized it to 0 until realizing that doing this required major changes to the protocol). We built Mencius, while maintaining integration with the Lab3A API (piggybacked done values, status, min, max, etc.). When servers receive word of an instance higher than their px.done value, they automatically skip any instances they own in between px.done and the received instance. They do this by maintaining a map for each other server which lists which instances should be skipped. These messages are then piggybacked (then deleted) onto accept messages (Optimization 1 [1]). Skip messages are also piggybacked on propose messages (these are called suggest messages in Mencius, see Optimization 2 [1]).

The design and implementation of Mencius was done in close concert with an iterative design of test-cases to ensure that we were building a correct protocol. These test cases use the Lab3A test-cases, but all tests have been modified (some substantially) and new test-cases have been added. It now only makes sense to call Start when a paxos server owns seq (otherwise the paxos server ignores the call), so it is not possible to have duels with multiple start calls. It is possible to have multiple Revoke calls which duel (see TestRevoke), or a Start call dueling with Revoke (called a false suspicion in Mencius). Since non-owners of seq can only propose no-op, this means a Start/Revoke duel can end with an instance deciding the proposed value or no-op, and it is no longer safe to assume an instance will always decide on a particular value (as would be the case in Lab3A with dueling Start calls which propose the same value to the same instance). Moreover, the other labs (Lab4A and Lab4B) were modified to use Mencius, and it was of utmost importance that their respective tests still passed. The design and ensuring correctness for Mencius was a painstaking process, but of critical importance as it is a fundamental building block for the remainder of the project.

3. PERSISTENCE

Our system also handles failure by storing data onto disk. Our data is backed up using SQLite. Using the SQLite driver for Go, we wrote a suite of APIs to store ShardKV and Mencius state. In order to satisfy the strict consistency requirements, we implemented two schemes for persistence. For ShardKV, we kept a ledger of all the changed values of the ShardKV table, and ran a background process that periodically flushed the marked values into disk. For this, it is necessary to enforce strict all-or-nothing atomicity of the operations on the values being flushed and the associated client state. To address this, we made use of SQLite's built-in transactions system, and aborted any in-flight transactions at the time of disk failure.

Upon recovery of a replica, we would then load the key-value pairs back into disk using a similar transactional protocol. At the time of the benchmarks, we ran a ten-second checkpointing interval for ShardKV. This process saves the key-value map, the client-reply map, the processed pointer, and the current configuration. These are stored in a SQL record-wise fashion, except configurations which use Go's gob encoder to load and store binary blobs. However, this checkpointing scheme would not work for Mencius because of the stricter consistency requirements. In the case of persistence for Mencius, we flushed variables to disk as soon as they were altered in the RAM. This would prove to be a tremememdous bottleneck in our design, as every change in the paxos variables would incur a write to disk. However, in order to maintain the consistency of the paxos instances, it is a part of the latency that we could not remove. Our bottleneck analysis section will address this issue in more detail.

4. FAILURE RECOVERY

If a single replica crashes (with or without complete disk loss), it is able to recover completely and begin serving client operations. If all replicas crash simultaneously, upon rebooting, they will load data from disk and continue from where they left off. Handling the case of failure without disk loss is straightforward if persistence is designed correctly, a replica can load requisite data from the database on disk; therefore we limit our discussion here to the case of disk loss.

The naive approach to implementing recovery from disk failure is to query another peer that is reachable and ask it to transfer necessary data. This may seem OK at first glance, but certain pathological cases can arise. Consider a replica group consisting of replicas A, B and C. Packets to C are lost because of some type of network partition, but A and B are still able to serve operations, since they form a majority. The client sends several put operations, which are then decided by A and B. Replica B then crashes and loses its disk, and when it returns, it asks C for its disk. Replicas A and B had previously decided several instances, but being that B has lost its disk and replaced it with C's, it is now possible to re-decide previously decided instances with replicas B and C, if A were somehow partitioned. This is clearly a major problem.

In the previous example, if B had instead decided to recover from A, there would be no problems. This leads one to believe that it might be possible to pick some type of "most up-to-date" replica to retrieve on-disk data from. This happens to also not be possible: B and C may have also decided some instance that A is unaware of. Our solution to this scenario is simple: when a replica that has lost its disk reboots, it queries other replicas for their Paxos max (the highest instance they have knowledge of), it then requests that peers move their Paxos max to the highest such max; once peers have all processed up to and including at least this highest max, we request a disk transfer from any peer. We do not freeze operations during recovery; when we are waiting for peers to process operations, we do not care that they may still be deciding instances, we only care that they finish processing up to the highest Paxos max therefore, resolving the possible re-deciding of previously decided instances as a result of replica disk loss. Figure 1 depicts the recovery process for a ShardKV server in detail.

We have implemented a new suite of test cases in ShardKV to convince ourselves that we handle failure recovery correctly. These tests target specific replica groups (in the case of single-replica crash tests) by performing puts using keys targeting the random shards. We then kill one of the replicas inside of this replica group. Performing puts before, during and after the crash, and validating that all of these key value pairs exist on the restarted replica some time after reboot; this demonstrates that a replica has caught up. In the case of a disk loss test, the replica also loses its disk (the SQL databases containing our persistent data) when it crashes. Separate test cases use larger (100MB) keys with several GBs of total data in key value pairs. The situation when all servers crash is simulated by performing a set of puts, killing all servers and rebooting them, performing more puts, and ensuring that every individual replica contains data consistent with the puts. The re-decided case described earlier (with servers A, B and C) is tested by deafening a random minority of replicas within a random replica group. Large numbers of puts are then processed by this same replica group. One of the peers which were not made deaf then crashes and loses its disk. The entire replica group is made undeaf and the crashed replica boots up. We check that all replicas within the replica group contain all previous key value pairs and that their key value maps are exactly identical using the reflection package. We ensure that the replica can't just "get lucky" and retrieve disk state from a peer that wasn't deaf by ensuring that replicas are made deaf in the same order that a crashed replica will walk the peer array to try to retrieve disk state from. Every test also has its unreliable counterpart where RPC calls and replies are

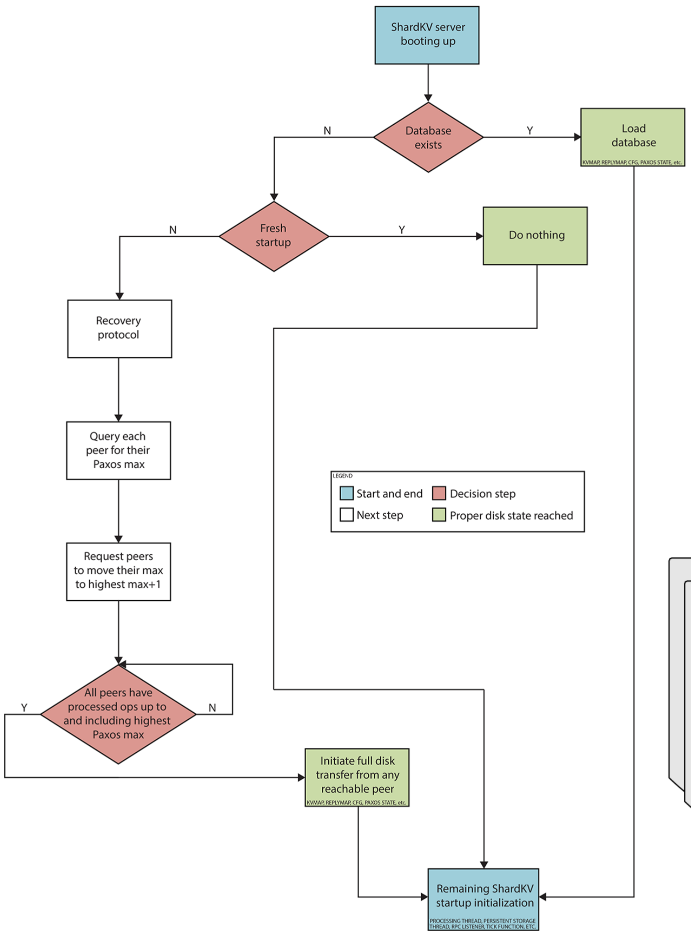
ShardKV server booting up
Database exists
N
Y
Load database
KVMAP, REPLYMAP, CFG, PAXOS STATE, etc.
Fresh startup
N
Y
Do nothing
Recovery protocol
Query each peer for their Paxos max
Request peers to move their max to highest max+1
All peers have processed ops up to and including highest Paxos max
Y
N
Initiate full disk transfer from any reachable peer
KVMAP, REPLYMAP, CFG, PAXOS STATE, etc.
Remaining ShardKV startup initialization
PROCESSING THREAD, PERSISTENT STORAGE THREAD, RPC LISTENER, TICK FUNCTION, ETC.
LEGEND
Start and end
Decision step
Next step
Proper disk state reached

Figure 1: Recovery Protocol

randomly dropped, and all tests perform data consistency checks across all replicas within the replica group(s) under scrutiny.

5. DEPLOYMENT

We deployed our system both locally and on the Amazon Web Service (AWS). There are two branches of our repository. The main code is the same in two branches. However, the test code in the master branch is mainly written to test the correctness of our implementation. The code on the aws branch is modified such that our system can be deployed across different machines on AWS to test real-world throughput and latency.

We used Elastic Cloud Computing (EC2) machines to deploy our system. In addition to Mencius, ShardMaster, and ShardKV, our EC2 deployment framework consists of Supervisor and ClerkServer class. Supervisor acts as a manager of one particular EC2 machine and is responsible for starting and killing Mencius, ShardMaster, or ShardKV instances via our API interface. In our framework, only one Supervisor exists in one EC2 machine. The Supervisor keeps maps of Mencius / ShardMaster / ShardKV it has started and deletes the instance in the maps on Kill. Supervisor also supports query of how many requests the ShardKVs living on the same machine as the Supervisor have served so far. This RPC is particularly useful when measuring the throughput of our system.

ClerkServer is similar to Supervisor in that there is only one ClerkServer per one EC2 machine and it acts as a manager of that particular machine. The difference is that ClerkServer is only responsible for Clerk instances (the ShardKV's client). That way, we are able to start many different clerks on different machines to increase the parallelism of our requests. To support this functionality, we also modified the Clerk class slightly by creating an additional init() method, namely MakeClerkServer(). MakeClerkServer() initializes the clerk and starts a go thread which continuously sends requests to the ShardKV's server while it is not terminated.

The setup is illustrated in the following diagram. The ClerkServer is omitted for simplicity.

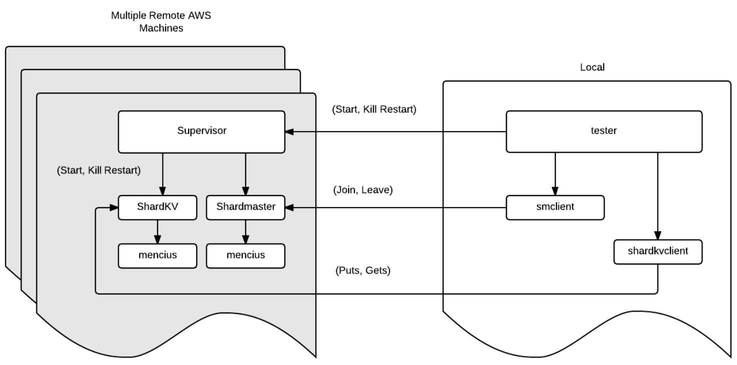

Figure 2: Deployment Setup

We created two test suites to measure the latency and throughput. The throughput test lives in shardkv_test.go file, while the latency test lives in latency_test.go on the aws branch of our code repository. TestThroughput first proceeds by initializing the Supervisors. Once all the Supervisors have been successfully initialized on the EC2 machine, our test suite contacts all the Supervisors to start the ShardKV servers and the ShardMaster servers. Next, the test suite initializes the ClerkServers, which will initialize the clerks who will start firing requests at ShardKV servers. After some predetermined time interval, we stop all the execution and query the Supervisors for the total number of requests processed within that period. The throughput is defined as the total requests processed divided by the time interval.

We measure the latency under 2 conditions: without contention and with contention. Latency is defined as the time it takes for the client to finish executing one request call. The setup process of our latency test is the same as that of the throughput test. For the latency test without contention, we only start one clerk to send requests to ShardKV. For the latency test with contention, we start many clerks on several different machines and direct them to send concurrent requests. For both tests, we take a sample of the latency measurements by polling one of the clerks for its latency data.

6. PERFORMANCE

Our performance is measured in terms of RPC calls made by our Paxos implementation, and in terms of latency and throughput of our key-value store deployed on AWS.

6.1 Paxos Performance

We created a benchmarking package specifically to compare Mencius and Paxos. These benchmarks compare the number of RPC calls with an increasing number of instances to decide. These benchmarks also look at the number of RPC calls under failure of a replica (see Figure 3). We also leverage Go's built-in benchmarking to compare offline performance of Mencius and Paxos across the tests from Lab3A. From our tests, we can see that under normal conditions, Mencius uses roughly 2/3 of the RPC calls that Paxos does (the prepare phase in Mencius is skipped). Paxos though degrades more gracefully under failure than Mencius does. This is because of a replica fails in Paxos, there are just less RPC calls (but still a majority of replicas). In Mencius on the other hand, the instances that were previously owned by the failed replica now must be revoked by the other replicas, and revocations constitute of the entire regular Paxos phases (prepare, propose and decided). The number of RPC calls under a replica failure in Mencius is still just under those for Paxos without failures. This is because for some instances owned by the replicas that have not crashed we are still able to skip the prepare phase.

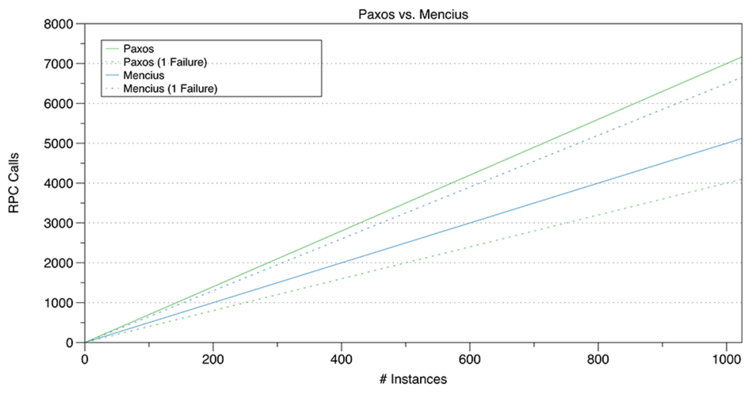

Figure 3: RPC Calls

Using the built-in benchmarking package in Go, we tested the processing time for a variety of Lab3A tests (see Figure 4). These benchmarks were run locally on a Macbook Pro. Some of the tests for Mencius were modified for reasons described earlier which may account for speedup in some cases. In other tests, the total sleep time eclipses any difference in performance between the two (e.g., in TestManyForget there is a total of 7 second sleeps). These tests were run many times, and we show the 95% confidence interval using the mean and standard deviation of these runs. We can say from these results that Mencius does not run slower than Paxos, and that there is a likely performance benefit in using this optimized Paxos variant.

6.2 System Performance

We created tests to measure the performance of our entire system deployed on AWS. We used compute optimized machines rather than general purpose machines to utilize

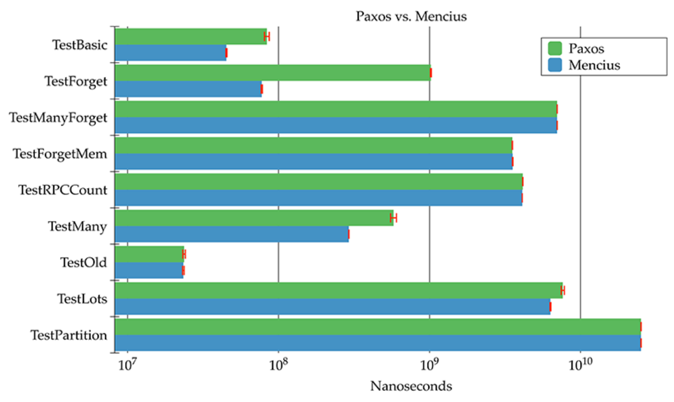

Figure 4: Paxos Performance

a higher network performance (1̃0ms packet latency within the data center using internal IP addresses). These machines communicate through Transmission Control Protocol (TCP). We tested several configurations. However, due to a limited number of available AWS machines, we are only able to test up to 4 replica groups. Figure 5 shows the throughput of our system. Each replica group is deployed on one machine. There are 3 paxos peers for each replica group. ShardMaster is deployed on a separate machine, and is replicated using 3 paxos peers. As we can see in Figure 5, our performance improves as number of replica groups increases. We have conducted average throughput runs, we show the 95% confidence interval using the mean and standard deviation of these runs.

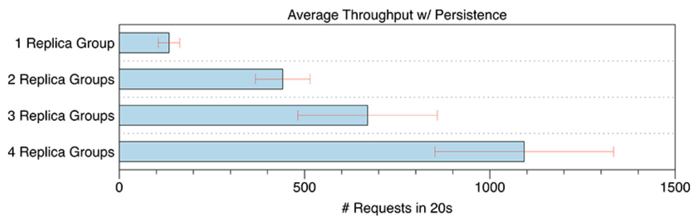

Figure 5: Avg. Throughput w/ Persistence

6.3 Performance Bottlenecks

Our project was implemented with consistency as the foremost goal. As such, we found it difficult to relax parts of our implementation. Upon analysis, we realize that the most prominent bottleneck piece of our design is in the persistence of Mencius states. We can see this in the partial persistence graph below.

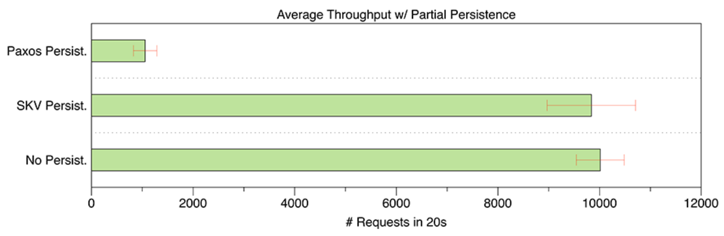

Figure 6: Avg. Throughput w/ Partial Persistence

The bottom bar is the throughput we obtain when we turn off all writes to disk. The second bar is the throughput when we turn on disk writes for ShardKV, and the top bar is the throughput when we turn on disk writes for Mencius. As we can see, the ShardKV writes to disk have been optimized well through the use of checkpoints. However, turning on the Mencius disk writes causes a severe decline in performance because of an inability to batch the continous writes to disk. In the face of disk writes this frequent, the impact of network latency becomes less significant. This bottleneck was something that we could not remove, because we used a pre-packaged logging structure, SQLite, to broker writes to disk for us.

Furthermore, the out-of-order nature of paxos operations makes it hard for the writes to be cache-friendly. Because of this, we recognize that Paxos might even not be the best solution to a disk-crash tolerant version of a key-value store. For future extensions of this system, we recommend 1) not persisting anything non-essential for recovery from failure (we are storing some values that optimize recovery time but perhaps unnecessary. like Mencius skipped instances which are later piggybacked), and 2) the study of a more lightweight, cache-efficient logging system.

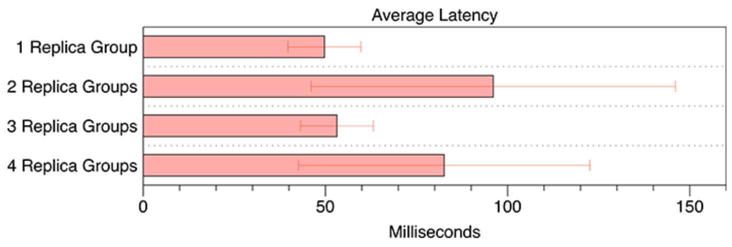

Figure 7: Avg. Latency w/ No Contention

Another part of our system bottleneck is the contention between clerks. Contention happens when there are multiple clerks issuing Puts/Gets to the same ShardKV group. The effect is most pronounced when we are measuring the latency with multiple clerks, as shown in Figure 8. The horizontal axis indicates the average request latency in milliseconds and the vertical axis indicates the clerk group size. As we increase the clerk group size, the contention increases, which in turn increases the average latency of our system. One possible way to reduce contention is to increase the number of replica groups to service more clerks. Another solution is to implement a more sophisticated ShardMaster which dynamically rebalances shards based on observed hotspots.

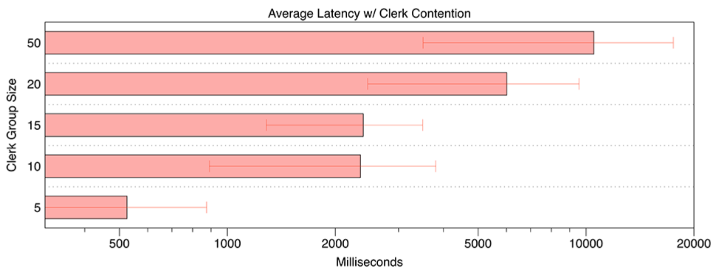

Figure 8: Avg. Latency w/ Contention

7. CONCLUSIONS

As seen in the Performance section, our system can be easily scaled to handle large volumes of data, if we increase the number of replica groups. However, to drastically improve the performance, we would have to relax consistency guarantees of our system, particularly in our Paxos implementation.

8. REFERENCES

[1] Y. Mao, F. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. *OSDI*, pages 369–384, 2008.

[2] Y. Mao, F. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. Technical report, 2008.