MenciusDB:

A fault-tolerant and high-performance key-value store built on the Mencius protocol

6.824 Spring 2014 Final Project
Quan Nguyen, Bradley Wu, Sherwin Wu, and Stephanie Yu

Note: All our code is available at the following public git repository:
http://github.mit.edu/styu/824PlaysGames

1 | Introduction

For our final project, we worked on the default final project and tailored it to create our own system for a specific use case. MenciusDB is a high-performance, fault-tolerant, key-value store that runs using the Mencius protocol for distributed consensus. It is designed to work best on systems that run on top of wide-area networks (WANs). It is fault-tolerant due to the highly replicated design that carried over from Labs 3 and 4, so it is robust to failures of machines. It is also high-performance through its modifications to the Paxos protocol to prevent two round trips for every consensus instance and its ability to scale up to work with multi-gigabyte size databases. In this write-up, we will present benchmarking data to demonstrate evidence of the high-performance scale of MenciusDB.

2 | Mencius Protocol

Because our system is tailored specifically to work well on wide-area networks, we decided to adapt the original Paxos protocol to use the Mencius protocol. The implementation of Mencius is based on that of Paxos, but Mencius peers do not have to contend to become the leader for every slot in the log. The peers instead are leaders for certain log entries that are "assigned" to it. For those specific log entries, if a peer has an operation to propose during its turn, it will send a special "suggest" message (which is the same as a "propose" message), thereby skipping the initial "prepare" message roundtrip. If other servers move ahead in the log, a server will notice and catch up by sending special "skip" messages (again, same as a "propose" message) to fill in its entry logs that it never got to "suggest" operations on. As a result, there would be less round trips per operation because the "prepare" phase in Paxos could be skipped since the leader is already determined. This also guarantees that Mencius peers always make progress, while Paxos peers could get stuck fighting to become the leader.

Our actual Mencius implementation follows directly from the Mencius paper and is an adapted version of our old original Paxos code. Our implementation follows the four rules presented in the Mencius paper:

1. Paxos servers now keep an additional `nextInstanceNum` state variable which keeps track of the number of the next instance the particular server is in charge of. When a client calls the

Start function, we start an instance of Paxos using `nextInstanceNum` and send out the "suggest" messages to all other servers.

2. In the `AcceptHandler` function, which is called when another server is trying to send a server an accept message, we detect if it the message is the special aforementioned "suggest" message. If it is, and if the instance number of the "suggest" message is greater than the current `nextInstanceNum` of the server, the Mencius server will send "skip" messages and increment its `nextInstanceNum` until it is finally greater than the instance number of the "suggest" message. This ensures that no servers that will fall behind in the log.
3. When a server believes that another server has fallen behind, it will try and send a No-Op to fill in the log. But it will propose the No-Op starting with the "prepare" phase. We handle this in the "Status" function; when a client wants the Status on an instance, and it hasn't been decided on even though the Paxos server has moved on to higher instance numbers, then we assume the server that was in charge of it has died, and propose the No-Op.
4. If a server suggested a non-No-Op operation and it doesn't succeed, then it will try again until it succeeds.

The Mencius protocol's main innovation is its striping of the agreed-upon Paxos log. In a system with three servers, server 1 would be a "distinguished proposer" for instances 0, 3, 6, etc…, while server 2 would be in charge of instances 1, 4, 7, etc…, and so on. On a certain agreement instance, only the "distinguished proposer" can propose any value other than a No-Op. When this "distinguished proposer" proposes its non-No-Op value, it can skip the entire first Prepare phase of the Paxos protocol because it knows that it is the only "distinguished proposer" on that instance. This allows us save on a single round trip call everytime we try to add something to the log, decreasing overall latency of placing a single operation into the Paxos log.

There are also other positive implications of Mencius's striping mechanism. The Multi-Paxos protocol, where all operations must pass through a single leader, aims to achieve the same goal as Mencius, which is to reduce the number of roundtrips and ensures that the system as a whole make progress by not having the Paxos peers contending for every single slot. Multi-Paxos, however, has many disadvantages compared to Mencius. In a high load environment, the single leader is a bottleneck on network bandwidth if messages are large in size and a CPU bottleneck on the single leader if the messages are small in size. Further, if a single leader is geographically distant from a client, there is significant latency in sending the message. However, in Mencius every server will have an opportunity to add something to the log. Therefore, a client can send their operation to the closest geographical server, whichever it is, and it will be inserted into the log. Further, the striping of the log allows for higher throughput and lower latency overall for operations given to the server. The Mencius protocol is limited, however, in that the system works best when there is an approximately even load on the servers in the system. We leave this up to the user of our system to strategically place servers such that the load on each of them is approximately equal.

3 | Data Persistence

Another aspect of our fault tolerant protocol is that we persist key-value data on disk. Our system is designed to support a large amount of data that may exceed the size of any one server's RAM. To do so, we store the entire key-value map as a set of flat files in a server's file system. From a base file path specified at server startup, a directory exists for every configuration ever maintained by the shardkv server, and within this directory, we have a file for each key-value pair in the server's hash table, with the file name as the key and the file's contents as the corresponding value. For example, if a client submits a put request for key, *myKey*, and value, *myValue*, and the current server reconfiguration number is *3*, the server would be create a file with content, *myValue*, on disk at the file path, */<base path>/3/myKey.*

In order to try and reduce file I/O operations, we supplemented our on disk key-value storage with an in memory cache implemented as a LRU cache. A read operation only needs to check the file system if the requested key does not exist in cache. For write operations, it is important to persist changes to disk in case of system crashes. However, due to the cost of file write operations, we want to avoid writing to disk on every incoming put request. Instead, we implement a *writeBuffer* to maintain an ordered list of operations that have been applied to the in-memory cache but not yet saved on disk. The contents of the write buffer are periodically flushed to disk, currently implemented in the Tick() function, independent of incoming client requests.

For recovery purposes, we also save all meta data pertaining to the state of the server, which includes the paxos state, current configuration, and data structures for detecting duplicate requests. The structures are encoded using Go's *gob* package and then written to the directory corresponding to the current configuration. On startup or restart, a shardkv server will check its file system and load any pre-existing state from disk. Because we flush batch updates to the key-value structure on disk and not on every client request, it is possible that a server crashes before writing all updates to disk. The recovered server state may trail behind the live system, but is able to catch up as per normal operation.

4 | Benchmarks

We ran a number of benchmarks comparing the performance of disk reads/writes versus in-memory reads/writes and observed a significant performance overhead. Furthermore, we benchmarked the performance of a combination of the original shardkv server, the shardkv server with our Mencius implementation, and a shardkv server with both Mencius and disk writes implemented. Although we had hoped Mencius would yield better performance, we realized there is overhead associated with the increased number of rpc calls made when broadcasting operations to all servers in a Mencius group instead of just the majority (an optimization we had made to the original paxos protocol), which offsets the benefits of reducing the paxos agreement to a single round trip. Additionally, due to the overhead of the reads/writes to disk, we observed an expected performance deterioration in the final implementation. However, because of the fault tolerance that the disk writes allows in recovering from server crashes, we think this is an acceptable loss in performance.

5 | Conclusion

Mencius DB is a key-value store that is meant to be used on wide-area networks with load approximately equally distributed amongst its servers. It is fault-tolerant and highly failure resistance through its consensus protocols and data persistence. It is also high-performance through our use of the optimized Mencius consensus protocol. The result is a highly robust data storage system that can survive through machine failures, power losses, and other disasters without losing service to the end user.

6 | References

[1] Yanhua Mao, Flavio P. Junqueira, Keith Marzullo. *Mencius: Building Efficient Replicated State Machines for WANs. <http://www.sysnet.ucsd.edu/sysnet/miscpapers/mencius-osdi.pdf>*