6.824 Final Project Report
Eric Lubin, Jared Wong, Kevin King

Performant Donut Final Report

Introduction

We were excited to take on the default project, as well as make our own additions to it. We implemented a key-value store that functions in the presence of the following failures: dropped, reordered, or duplicated network packets; crashing (hard booting) machines; and hard disk failures. We make the guarantee that as long as a majority of the machines are reachable, the key-value store will maintain liveness; and as long as a majority of the machines' disks keep their integrity, no operations will be lost and the store will be consistent. Our key-value store also supports any key-value pair that fits in memory+swap, and has capacity equal to the capacity of the disk.

FileKV

FileKV is our custom storage solution. Rather than using a third-party database or storage system we decided to build our own all-or-nothing key-value database from scratch. At its heart is the simple idea of storing each key-value pair as a new file. Each database consists of a single directory and a file for each one of the key value pairs.

Atomic File Renames

To provide all-or-nothing semantics for our storage solution, we bootstrap atomicity using file renames. Each key-value pair is stored as a file name-file contents pair on disk. In order to provide all-or-nothing semantics, the key-value pair is initially written to a temporary file. The commit point for storing a key-value pair is when we rename the temporary file to its final name. Each file is named as the base64 encoding of its key, which allows both determinism for encoding the key, if provided again, and reversibility if FileKV needs to list out all the keys at once.

Memory Cache

Since FileKV must write all Puts to disk to achieve fault tolerance, we implement a Least-Recently Used in-memory cache to speed up its performance. If a key and its value are Put, the value is cached in memory, if it fits, after being written to disk. On Get, we first check whether a value exists in the cache for that key. If it does, we can successfully return and if not we resort to a disk lookup. The commit point of a Put remains the atomic rename of the file write. We limit the size of the cache to a predetermined value, and flush values of old keys that no longer fit.

Transactions

Some of the operations in other components of our system require transactions with all-or-nothing atomicity. To accomplish atomic transactions, we first log the entire series of puts to be committed into a transaction log. We then execute each of these puts (on their individual files). Finally, we delete the transaction log. The deletion of the transaction log is the commit point that signifies that each of these puts were successfully completed.

Upon recovery, FileKV looks for a transaction log. A transaction log will only be present if FileKV died before committing a transaction. If one is found, FileKV executes each of the logged puts it contains and then deletes the transaction log, finally resuming normal function.

Snapshots

We provide an efficient and lightweight snapshotting function that allows for near constant time “versions” of each FileKV database to created on the fly. We use this snapshotting tool in our ShardKV implementation to make a copy of the current set of shards that needs to be transferred to another paxos group.

The snapshotting process is as follows:

1. Create a new directory with a temporary name
2. Iterate through all the keys and make hard links for every key, value pair in the new database
3. Rename the database directory to its desired name

In this manner, snapshots take time in proportion to the number of key-value pairs in the database and are independent of the size of these values. If the old database goes to overwrite the value of a given key with a subsequent Put, our use of atomic rename means that the new file is written to a new empty inode before the inode pointer is changed. Thus the read-only snapshot remains unmodified as the reference count to the value is decremented.

Paxos

Bulk Prepare

Vanilla paxos requires both a prepare and accept phase for every paxos instance. When sending its first prepare, a paxos peer attempts a “bulk prepare” of a constant size (currently set to 25). When an acceptor receives a bulk prepare and has not had any prepares for the next 25 slots, the acceptor records the prepare for the current slot in the next 25 slots as well. If a peer’s bulk prepare is promised to by a majority, this peer will skip the prepare phase for these slots.

Biased Clients

To solve the dueling leaders problem, we bias our clients towards sending requests to the earliest servers in the socket list. This behaviour follows naturally from the way clients retry in the face of errors. Clients first try server zero, then on error proceed to the next server in the list. As a result, in relatively normal and reliable network connections, clients will generally be contacting the first paxos peer. In this way, the first paxos peer will in general successfully bulk prepare and not be competing against other peers. We see this as a pleasantly simple way to avoid dueling leaders since our paxos “clients” are also servers that are under our control.

Persistence Strategy

We extensively make use of FileKV for all operations that modify state to ensure that all state changes are persisted to disk in case of failures. In addition, we add some further error-handling in the case that a call to Start() is completed, and then the server crashes before the value has been decided (because propose runs as a separate go routine). In Start(), we write the propose value for the sequence number to disk using FileKV, and on recovery we check for any values that were proposed but did not complete, in order to restart that go routine that was halted.

Donut

The Donut is our paxos-backed fault-tolerant load balancer that dynamically reassigns shards to groups as a function of the current and historical traffic on the network. The load balancing is sensitive to both a high number of requests to a given key, as well as the size of the value being stored for the key.

Traffic Reporting

For ShardKV replicas, time is divided up into units of Epochs. Within each epoch, each paxos group tracks the total amount of traffic to each shard. Traffic is considered a function of both the number of requests to any keys within that shard as well as the size of the values stored for that key. On every request sent to the group that owns that shard in which the key resides, we add a constant REQUEST_COST, along with the size of the value, to a counter for the given shard. At the end of every epoch, each paxos group sends their traffic information to one of the servers in the Donut paxos group, then flushes the traffic array and begins again.

Rebalance Strategy

The Donut server is a paxos backed set of servers responsible for processing the traffic updates from the shardkv servers and sending configuration updates to the shardmaster. Time is divided up into Eons, every Eon is configured to consist of one or more Epochs, and a configuration update is guaranteed to be sent to the shardmaster only once per Eon. In our current implementation, Epochs last 3 seconds each and Eons last 15 seconds each.

Donut keeps track of an exponentially weighted moving average of the traffic to each shard, which is updated every Epoch. We define α to be the weight of the most recent traffic update in the moving average. At the end of every Eon, Donut attempts to rebalance shards in order to best even out the average traffic to each group.

Our rebalancing algorithm is as follows:

1. For every group, sort its shards in increasing order by traffic
2. Sort every group by the sum of the traffic to all its shards
3. Consider moving shards to the group, x, with the smallest total, x.Total
4. For each group, y, starting with the one with the most traffic, move the shard with the lowest average traffic, indicated by value v, to x if y.Total - x.Total > v
5. Every time a shard is moved, return to step 2 and repeat

During shard transfers, Eons are halted for a predetermined amount of time to give the server enough time for the shard transfer to take place without having an abnormally low amount of traffic for that period.

Persistence Strategy

We make use of FileKV to get and set the current Eon number, the exponentially weighted moving average of each shard's traffic, the highest epoch number reported for each group ID, and the last committed paxos sequence number. These values, along with the persisted state of Paxos, are enough to recover from a crash.

ShardKV

Incremental Shard Transfers

The biggest challenge in supporting more data than can fit into memory was making shard transfers incremental. In the original implementation, all of the keys of a shard were serialized in a single RPC, transferred over the wire all at once, and deserialized into memory in a single function call. Since a shard can have many key-value pairs (and client request history records), our design had to change if we were to support key-value pairs up to the full memory size.

Our new design sends either one key-value pair or one client request history record at a time, and then sends a final "shard transfer completed" message to signify that the receiving replica group now "owns" the respective shard. Extra care had to be taken to provide delivery guarantees under crashing processes. A shardkv peer does not advance its commit point until it has disowned the shard, snapshotted its current state, and queued the snapshot as a pending shard transfer (all recorded to disk using FileKV).

A subtle detail is that since shard transfers happen across multiple RPCs, it is possible for a peer to receive two shard merges for the same shard with differing versions before either has completed (due to reordered and duplicated requests). We resolve this problem by recording both the highest version of a shard currently being merged and the highest version of a shard already merged. In this way, peers know whether a shard merge RPC is applicable to their current state.

Persistence Strategy

At the core of ShardKV's resilience to crashing is the persisted paxos log. If a hard disk fails, a ShardKV peer will refill its entire paxos log on the first request and as a result completely rebuild its state. This recovery would obviously take a long time, but is only for the worst case failure. To facilitate recovery, we persist a paxos commit point for each ShardKV peer that is only advanced upon the complete processing of an operation found in a paxos instance.

For some operations, it is not sufficient to process the operation and then bump the commit point, since a crash could occur between these two operations. In this case, we use a FIleKV transaction (with all-or-nothing atomicity) and couple the commit point increment with the other persisted data structure modifications (the key-value pair data map and client request history map).

Upon recovery, a ShardKV peer continues from the persisted paxos commit point. The peer also checks if there are any pending shard transfers, and if so, processes these transfers using the stored snapshots.

Testing

The hardest aspect of a server to test is what happens when a server is spontaneously killed at any point in the flow of execution. Using methods of lab 3 and lab 4 are insufficient for testing what happens when a server is spontaneously killed and later restarted. In order to test crashing servers, it is only appropriate to create separate processes for each logical server and kill the processes using SIGKILL. In order to accomplish this goals, we created a framework for testing using separate processes.

Our framework is in the package `testutils`, and it provides code to setup and teardown the necessary components of a server. In order to get a server up and running we usually need to create 4 things:

- Server Binary - A binary that simply runs a new instance of the server. This is important for starting the server in a separate process. Note that fork really isn't an option in Go.
- Interprocess Communication Socket - To be able to communicate with our server from our tests
- Server Port - To listen for incoming requests from peer server, for example in a paxos group or shardkv replica group
- FileKV - Our storage solution backs every server we create so that we can handle killing

and restarting servers.

Coordinating the creation of these four things for each server can be quite complicated. Our framework facilitated the creation of these parts of the server, starting servers, killing servers, and cleaning up after servers.
In order to create proper binaries for each of our servers, we had to create separate RPC systems just for the interprocess communication.

Testing our systems individually and together in order to test killing servers at arbitrary points in time, and to test losing disk, was relatively straightforward given the primitives provided by our testing framework.

Each of the packages that end with _test, have our persistence tests.

Benchmarks

Benchmarking a persistent distributed system with a minimum of 15 separate servers (processes running separate servers) on a single computer is mostly futile. Each of the servers is backed by persistent storage and requires writing to disk for nearly every state change. We don't have access to battery backed memory, so we require that we write to disk on every single update. When 15 separate processes all try to write relatively small chunks of data to disk very frequently, the single largest factor impeding performance will be disk writes.

In order to properly benchmark this system, we should have set up different computers for each server.

Benchmark Data

Bulk Prepare vs. Vanilla Paxos

Test: Put 1000 10-KByte values

	Vanilla Paxos	Bulk Prepare
Avg. Latency	112ms	61ms
Total Time	120.47s	65.12s

We saw positive results from our paxos bulk prepare protocol. Recall that Vanilla Paxos executes both the prepare and accept phase for every paxos instance and Bulk Prepare avoids the prepare phase on most paxos instances.

FileKV MemCache

Test: Put 100 10-KByte values and then execute 500 random Get requests

	Without Caching	With 2GB Cache

Avg. Put Latency	37.69ms	60.21ms
Avg. Get Latency	31.13ms	37.38ms

Here we observe that our cache actually drastically reduces performance. We conjecture that this is because every client operation requires a new paxos agreement and advancement of the shardkv commit points. Since both of these events cannot be cached, we are probably better off letting the operating system cache disk reads for us.

Paxos Biased Clients

Test: Agree on 1000 consecutive values.

	Biased Access Towards *peer[0]*	Access *peer[i]* for Instance *i*
# RPCs	8157	11360
Total Time	6.421s	11.647s

Our idea of biasing clients to the same peer proved to work well in avoiding dueling leaders, drastically decreasing the number of RPCs and time required to agree on 1000 values.

Concurrent Throughput

Test: Put 100 values 1MiB in size, 3 shardmasters, 3 shardkv groups, 3 replicas per shardkv group

Concurrent Requests	Throughput (MiB/s)
1	3.163805
2	1.977247
3	1.420348
4	1.001179

The throughput decreases drastically as we increase the number of concurrent clients. This effect is probably because all servers share the same disk. As the contention over the disk goes up, the throughput will decrease.

Conclusion

Making our entire system persistent was indeed a challenge. Although our “abuse” of the UNIX file system may be a questionable idea, we enjoyed rolling our own persistent store. Building persistent primitives and then abstracting around them worked very well from a software engineering point of view. Persisting the paxos log first provided us, for free, the recovery of every state machine built on top of it. We definitely wish we had more time to deploy our system into a production environment, especially to test the donut component. Since all shards are on the same “machine” (the laptop we are using to test), moving shards between groups to load balance only had a negative effect because of the shard transfers. Overall, this project was interesting in both the software engineering and distributed systems manner. To say the least, all three of us respect the UNIX file system much more after completing this project.