CS739 Project 3 Writeup

Architecture

Our replicated block store uses two nodes on the server side. These two nodes are set up using the primary-backup paradigm. We created a simple client API with only read, write, and init functions exposed. In normal operation, the client library will communicate with the primary server, which will communicate with the backup server as needed. The two servers know that each other is alive via a simple heartbeat mechanism. Once per second, the primary will send an empty message to the server. If the RPC times out with no response, the primary server assumes the backup is dead. If the backup does not receive a heartbeat for two seconds, it assumes the primary is dead and begins taking over control. Both of these times are easily changed. Note that we are making many assumptions about the reliability and speed of the network, which were necessary due to the time constraint. We also assume that at most one of the servers will be dead at any time. This includes assuming that when the backup comes back online, the primary will not crash until the reintegration process is complete.

As mentioned above, in normal operation, the client communicates only with the primary. If the backup receives an RPC read or write request from the client library, it will respond with an error code indicating that it is not the primary, as well as the location of the primary. The library function will try again with the new client, hiding the mistake from the end client program. While the location information is not currently used, it could be used later if we wanted to adapt the system to use multiple servers. The servers and client library are coded in c++ using gRPC for communication. We used two gRPC services. First, the client->server service is exported on both the primary and backup and is used for read and write requests. Second, the primary->backup service is exported on only the backup and is used to relay writes to the backup, replay writes during reintegration, and the heartbeat. Both servers have a dedicated heartbeat thread, and the primary handles multiple gRPC requests concurrently.

The client library has three functions, read, write, and init. Init simply initializes gRPC. The read and write functions take as input a buffer of 4096 bytes and an offset. They both start by trying to send the appropriate request to the primary. If it is unable to reach the server, it tries again with the other server. This is repeated until it is able to communicate with one of the servers. Since connecting and reconnecting are handled by the library functions, server failures are transparent to the end user program. We planned on there being multiple clients connecting at once, both in the sense of multiple client machines and multiple end user programs concurrently on one machine. We have tested correctness and performance in both cases (see below). We attempted to create a test using FUSE and losetup to create a virtual block device that can be partitioned and formatted like any other block device, but unfortunately ran out of time.

Our servers export a single 256GB volume that can be accessed in 4K chunks. We store each 4K block of the volume in a separate file to allow us to do atomic writes as in P2. We don't create the files until a block is actually accessed. Despite each block being in a separate file, we do not require reads and writes to be aligned to block boundaries, only that they are 4K consecutive bytes. We assume that aligned operations will be the common case however. As the primary server is the only one that actually handles requests, it determines the ordering for strong consistency and simply forwards writes to the backup. The primary is only stateful when the backup is dead, but this is acceptable because we are assuming only one server will be dead at once. The backup, and the primary in normal operation are both stateless.

Primary detection and failovers

Primary detection and failover follow the same mechanism, utilizing the heartbeat as described above. Both servers launch in a backup state, exporting the backup service, client service, and the timeout thread. On launch the timeout period is extended to a nine second initialization phase. During our experiments this provided ample time for an existing primary to establish a communication channel with the new server, but is adjustable as necessary to avoid primary contention. We found on system startup that manually launching servers sequentially allowed enough time for one server to advance to primary and establish a heartbeat with the second, keeping it as a backup. If we launched servers programmatically we ensured a 1 second delay to avoid contention.

Heartbeats from the primary are sent at 1 second intervals. The backup times out after two seconds and checks for an up to date heartbeat. We felt these time intervals struck a good balance, limiting network traffic, while still allowing for a quick failover. These values are configurable if changes are needed to meet the needs of certain environments. If the heartbeat times out the backup server is promoted to primary. The server's state changes to 'initializing', and the backup service and timeout thread are terminated. The heartbeat thread is launched and the state is updated to 'single server' allowing the client service to begin handling requests. Within this state the server will accept and ack request, but will log writes for when the backup comes online. This process is described in more detail below.

Primary failure is transparent to the user, while the client itself is aware of the change as it must switch between two channels to communicate (client aware). We implemented and tested an alternative design in order to find the more efficient solution. Our second approach provided more transparency to the client by allowing client-server communication to take place over a single channel (client transparent). The backup and primary servers communicate with each other on their unique IP addresses, but on promotion to primary a shared IP address used by the client is toggled 'on' using an ip() system call. In order to implement this solution we had to adjust our startup flow so that the client service is not launched until failover (late launch). This approach simplified our client code, but ultimately proved to be notably slower. Average failover with the client-aware approach was 1.5s while the client-transparent approach averaged 9.4s. We were able to identify the late service launch as the primary bottleneck in this approach. We accomplished this by implementing the client-aware approach with the late service launch, resulting in the failover slowing down from 1.5s to 8.2s. The data can be found here.

Reads, Writes, and Logging

Clients can only submit valid read/write requests to the primary server; if the client attempts to use the backup for those operations while the primary is alive, the backup will respond telling the client to use the primary. When the primary server receives a write request from a client, it first acquires read/write locks (described below) for the relevant blocks. It then handles replicating the write, either by relaying the write request to the backup if it believes the backup is alive or by logging the write if the relay fails (indicating backup failure and switch to single-server mode) or if the primary was already in single-server mode. The write is then committed to disk once replication is confirmed, using atomic writing and swapping of temporary files. Finally, once the write is confirmed committed to disk, locks are released and the primary can send an acknowledgement to the waiting client. In the event of read requests, the primary server acquires read locks and reads and returns data without further operations on the backup server or the log.

Our log of single-server writes is stored in primary server memory since we assume the primary cannot crash when it's the only server or while the backup is recovering. It consists of a non-repeating list of dirty offsets which have been written to since the backup was last online. At backup recovery time, the primary forbids new write operations, reads the most recent data from disk for each dirty offset in the log, and sends the offsets and associated data to the backup where it replays the log by writing dirty 4K segments of data. Upon acknowledgement that backup log replay was successful, the primary clears the log. Our approach optimizes for repeated writes to the same offset by replaying only the most recent change.

Our strong consistency guarantees mainly depend on our design that clients retry unacknowledged writes, writes are confirmed on the backup or on a log before committing on the primary, writes are restricted by locking on the primary, and logs replay latest written data on every recovery attempt until a successful replay is confirmed. Our design ensures that a client will retry a write until receiving confirmation that it was committed on all known active servers, and that after successfully completing a recovery phase both the primary and backup must have identical up-to-date data. These guarantees hold as long as only one server can crash at a time, a primary is assumed not to crash while a backup is in recovery mode, and the primary does not process new writes during backup recovery.

Locks

As the primary is multithreaded and handles requests concurrently, we had to consider synchronization across threads. The three places we needed synchronization were per-block access, backup recovery, and primary initialization. All three of these are handled on the primary. For per-block access, we used a simple reader/writer lock for each block. The locks are acquired in order of block number to prevent deadlocks. We needed to ensure that writes and log replay do not happen at the same time so that new log entries are not added after the log has been sent, but before the backup is finished replaying it. To accomplish this, we again used a reader/writer lock. Before handling write requests, the thread acquires a shared lock and before starting to replay the log acquires the exclusive lock. Finally, when a server switches out of backup mode, it will immediately start accepting requests so the client need not resend the request, but it needs to wait until other parts of the primary process are done initializing to actually handle the request. This is accomplished using a condition variable.

Crash triggers

To be able to deterministically trigger crashes in the servers we created a special offset code that when seen by the primary or backup, causes them to crash. The first five bytes of the code is the string "CRASH". The sixth byte is the server mode we wish to trigger a crash in and the last two bytes are used to further distinguish crash points. To add a crash point to a server, it first checks if the first five bytes of the offset match "CRASH", then checks if the mode matches the server's current mode. Finally, it checks if the last two bytes match the code for that crash point. If the server does not crash, it handles the request assuming an offset of 0. When the primary is in single server mode, it adds the received offset (as opposed to 0) to the log so that we can trigger backup crashes during recovery. This allows us to trigger crashes deterministically from the client at almost any point in either server.

Consistency Tests

We were able to demonstrate consistency for several server states and failure points using our crash triggers and occasional manual restarts. These include: writing and reading consistent data at both aligned and unaligned offsets during normal operation, crashing between a series of writes to the original primary and a series of writes to the backup (now the takeover primary) then reading consistent data for all writes from the takeover server, writing in single-server mode then recovering a backup and crashing the primary before reading consistently from the recovered server, crashing in the middle of a write to primary then retrying the write and reading correctly from the backup turned primary, and crashing the backup during a log replay then restarting the backup correctly before a primary crash and reading consistent data from the replayed backup. These tests were run as client files which performed reads/writes and triggered crashes programmatically, occasionally paused and prompted the user to restart servers, and asserted that read data at different stages was consistent with expected written data. Video recordings of these demos can be found at the end of our presentation.

Recovery Time experiment

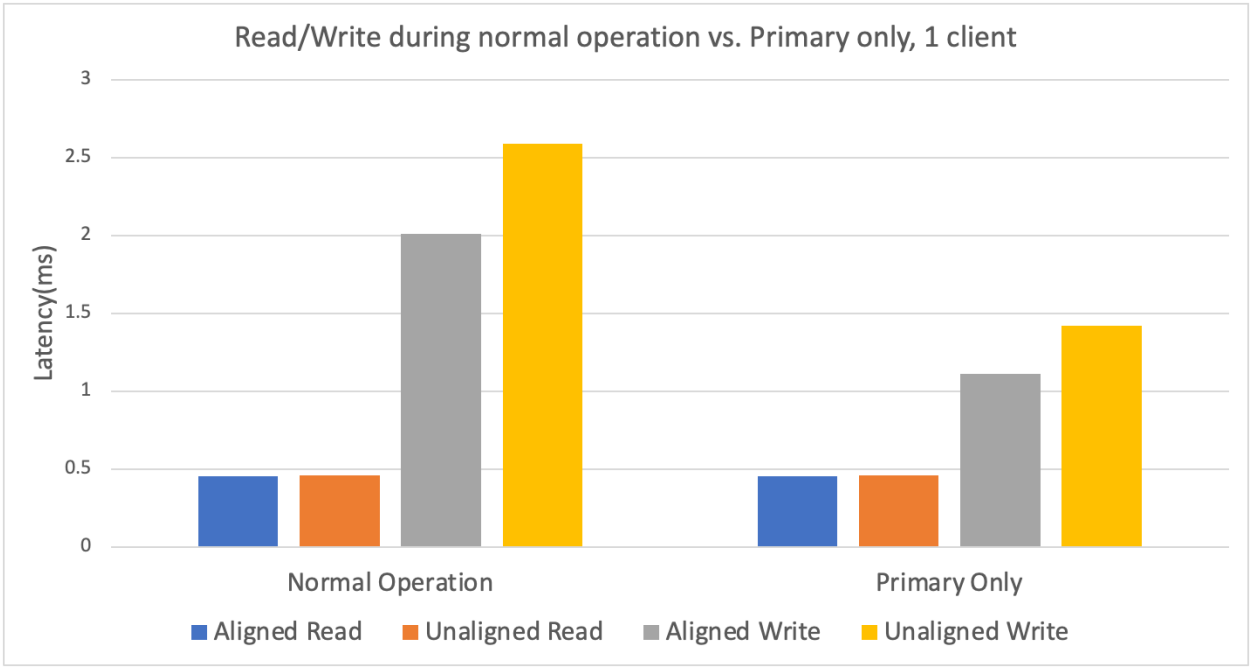
We measured the average time elapsed for each phase of the primary and backup startup procedure, including timing phases associated with failover and recovery in both servers. Detailed data can be found here, and timelines and graphs summarizing our timing measurement values can be found in the 'Startup/Recovery Time' slides of our presentation. For both servers, we measured the time elapsed to create the backup service, create the main server service, finish creating threads for services and heartbeats, and begin listening for heartbeats. These stages occur during initial startup for either server, and whenever a crashed server must restart as a backup. We also measured the time elapsed for the heartbeat timeout and transition to single-server mode the primary, which occurs during initial setup, backup crashes, and backup to primary fail-over; and the time for completing a heartbeat connection and replaying a log in the backup, which occurs during initial setup and backup recovery. We used the clock_gettime function with the CLOCK_MONOTONIC option to measure elapsed time in the server. Experiments were run with 5 repetitions, locally on a Kali VM with 2 GB memory and 2 Intel(R) Core(TM) i7-8750H @ 2.20GHz CPU cores running in Oracle VirtualBox, and remotely on a CloudLab setup using two Ubuntu 20.04 nodes with access to 157 GB memory and 40 Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz CPU cores.

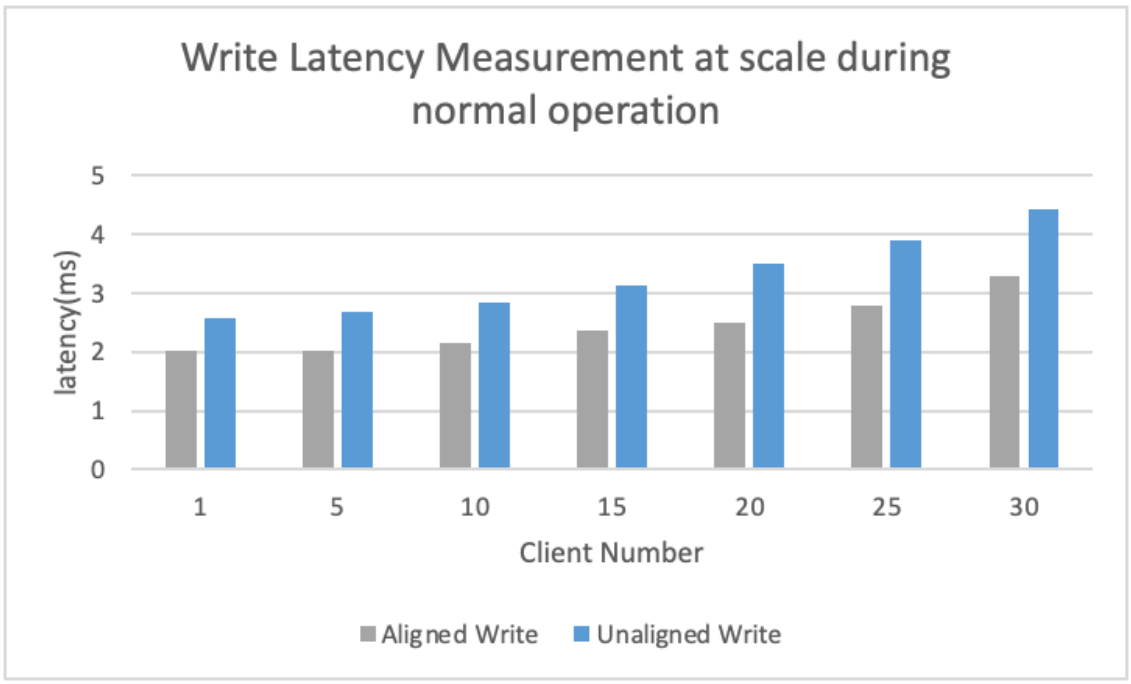
We found that server startup and recovery times are dominated by the time required for heartbeat functions in primary detection, failover, and recovery. Creating and exposing gRPC services take between 0.5-2.0 ms, and other thread creation and state change phases take only 0.01-0.1 ms, whereas successfully connecting over heartbeat during backup startup takes over 500 ms and heartbeat timeouts during primary creation take a minimum of 8 seconds. We also found that creating and exposing gRBC services did take longer on remote experiments with cross-node communication than on localhost. We also measured the time elapsed for replaying a log of writes on a backup during initialization and recovery with logs of size 0, 2, 8, 32, 128, and 512 entries, and found that the time required for replaying logged writes increased approximately linearly with the number of entries, approaching 1000 ms for 512 entries.

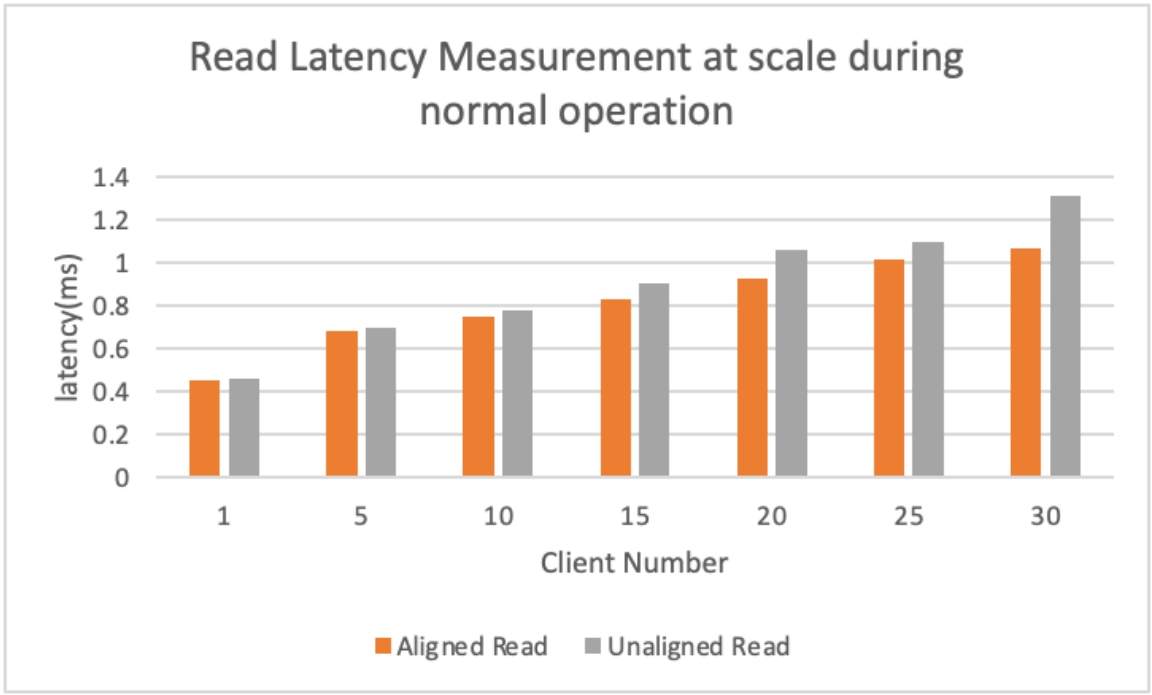
Latency experiment

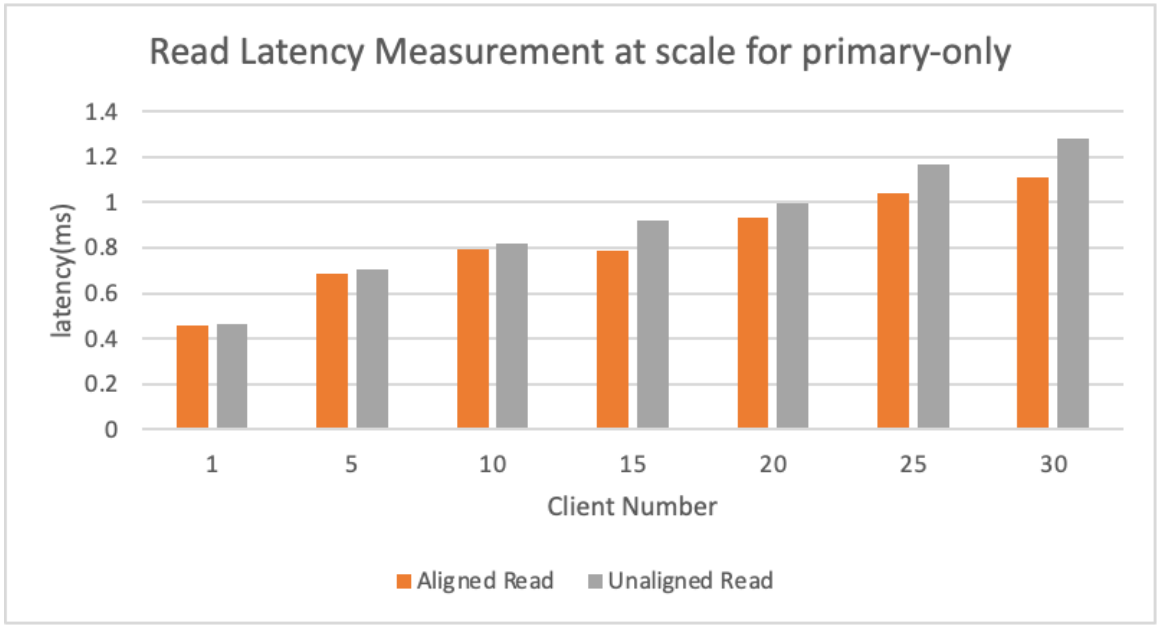
Experiment is conducted using cloudlab. 32 nodes(a mix of type c6525-25g and m510) in the Utah cluster were used.
One node is used to run the primary server, one node is used to run the backup, one node is used as the benchmark machine that takes measurements for different operations. The rest of the nodes are used to simulate concurrent workload. The simulated workload is a continuous request of read and write from a number of clients. Measurements are taken for client number(including the measurement node) 1, 5, 10, 15, 20, 25 and 30.

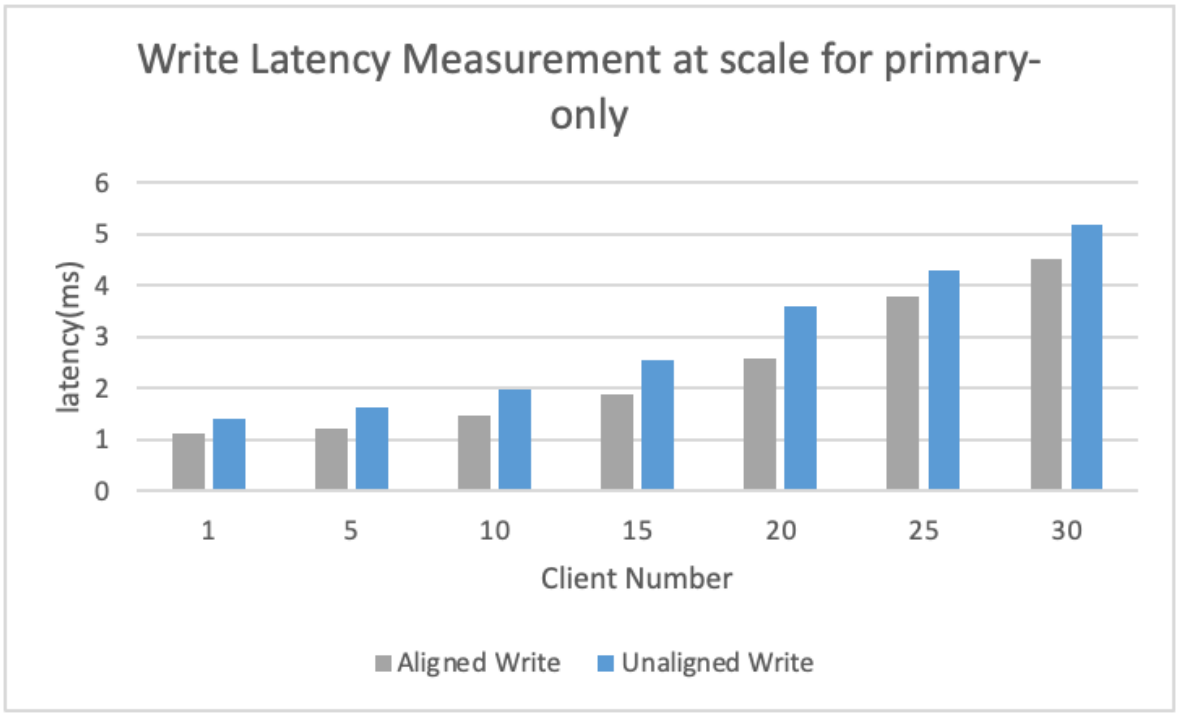
Overall we did not find any trend that contradicts our expectation: the latency of unaligned read/write is slightly longer than their aligned counterpart, but the difference is not significant since most of the latency is dominated by overheads and other than the disk read/disk write. And for the scaling performance, latency degrades at a linear rate for different operations as the client number increases.
Data
Graphs:




Write Latency Measurement at scale during normal operation
latency(ms)
0
1
2
3
4
5
1
5
10
15
20
25
30
Client Number
Aligned Write
Unaligned Write


Read Latency Measurement at scale during normal operation
latency(ms)
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1
5
10
15
20
25
30
Client Number
Aligned Read
Unaligned Read





[Benchmark measurement code]
[Workload simulation code]
[Primary server node]
[Backup server node]
[Measurement node]

Availability Demo

We modified a crash consistency benchmark to show the server failures are transparent to the user. We ran the benchmark 3 times and moved crashes to the beginning. On the first iteration there was no crash, on the second there was a primary failure, and on the third there was a backup failure. The demo has one large terminal on the left showing the client screen, and two on the right showing the servers. The upper right shows the server that starts as primary, the bottom right shows the backup. While running you can watch the crashes of the primary, then backup on the upper right screen. You will see that the client responds no differently to any of the scenarios, but that for the primary crash the run took much longer, as it had to wait for the backup to get promoted to primary. The demo can be found here.

Conclusion

We developed an implementation of a primary-backup replicated server, using the primary-backup protocol from our paper reading as a guide and making modifications to performance and consistency methods for simplicity. We found that following the protocol allowed us to provide a strong consistency guarantee with the addition of a few additional locking and heartbeat failover mechanisms which were required to deal with our handling of file volumes and gRPC calls. Although we were unable to fully implement certain features like IP takeover and FUSE integration, our basic server successfully functions as a replicated block store with good availability and consistency against single-node crashes. We found that it performs as we would expect from a primary-backup system, with optimized latency for read requests, scalable multi-client writes, and backup recovery under common write behaviors.

Response to demo question about failure models and durability

During our presentation we spoke about how we used a temp file and atomic rename on writes to help ensure crash consistency. We were asked about unaligned writes, and realized that our approach did not fully account for this as we could not atomically save two blocks at once. We discussed this issue after the presentation and felt that given the different failure models, our approach would always provide strong consistency, but that there may be temporary inconsistency between the primary and backup for short periods of time, and that at least part of that inconsistency required the client to resend the request (this *was* expected behavior) to resolve.

Expectations

- Only one server can crash at a time
- The backup server does not crash during recovery,

Protocol

- On write() the primary first sends write to backup server
- If there is no response from backup, the write is logged

- Only after write to backup or log is complete does the primary write to disk
- On primary failure, the backup is promoted to primary, on server recovery it comes up as backup
- The backup is not promoted to primary until after it recovers
- Writes on the primary lock and ensure no read is processed until write is complete

Fail-models

- The primary gets data and crashes
    - No issue. No writes have occurred. There is no inconsistency. The client never gets an ack and can retry
- The primary gets data, sends to the backup, the backup crashes
    - No issue. The primary will not get an ack from backup, it will log the request. There is a brief inconsistency, but it is handled by the server. When the backup recovers it will get the log and be brought to a consistent state with the primary
- The primary gets data, sends to the backup, gets an ack, and crashes before or during the write
    - This scenario is problematic because the primary will be inconsistent with the backup and will not have a log to recover from. The backup in this scenario will have the correct state, and will be promoted to primary resulting in strong consistency from a client perspective, in that a future read to the backup will return the correct data. Because the client does not get an ack, it will retry the request. In this case the new primary will redo the write, log it, and send it to the old primary on recovery
    - This scenario is dangerous because it relies on the client to resend the request. After discussion we realized that in order to avoid this reliance, we could use a transactional log on the primary that is persisted to disk prior to sending the write to the backup. After writing to disk the log would be updated with a checkpoint. On recovery the server could check for writes that were not checkpointed and replay them. This approach would work well for multi-failure scenarios as well.