Cody: A Highly Available Block Store Implementation

Cody Tseng Ping-Han Chuang Wuh-Chwen Hwang

1. Design

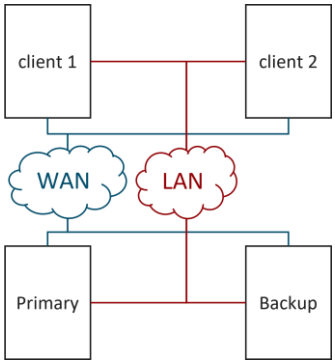
We decided to use primary-backup as our replication scheme. The system is designed to be highly available, and data should be strongly consistent. According to the CAP theorem, a system cannot be consistent, highly available, and network-partition tolerable. Therefore, although failures of the entire network are tolerable, our system cannot withstand network partitions.

1.1. Failure Assumptions

To achieve our guarantees, we have the following assumptions about failures:

- At most one server can fail
- When one server resumes from failure, the other server cannot fail before the resumed server fully recovers
- servers connect to two networks, and at most one network can fail

1.2. System architecture



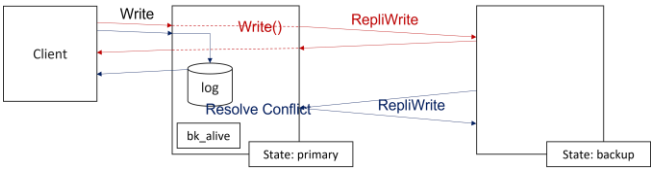
During startup, one server is designated as primary and the other as backup. Their role may change if failures occur. Only the primary can handle client requests. If the backup receives requests from clients, it simply rejects them and lets the client (library) figure out the right thing to do. Primary sends heartbeat messages to backup periodically to maintain its primacy. If the backup does not hear from the primary for a while, it times out and takes over as the new primary. All communications are first conducted through the default network (LAN, private IP) and retried through the other network (public IP) if the first attempt fails.

1.3. Client-Server Communications

A client library, which includes both servers' two IP addresses, is provided. Clients can interact with the service with two interfaces:

- Read: `(uint address)->{uint status, string data}`
- Write: `(uint address, string data)->{uint status}`

Both reading and writing return status. Currently, the status is 0 or 1, indicating whether the operation is accepted or rejected (because the server being contacted is backup). The client library will delegate the request to a server. If the server times out, the client library will send the request to the same server through the other network. If a client cannot connect to that server through either network or if the server rejects the request, the client library sends the request to the other server.



1.4. Inter-Server Communications

As previously mentioned, the primary must regularly send heartbeat messages to the backup to keep itself primary. In addition, if the primary receives write requests from the clients, it needs to replicate the write to the backup (hereinafter: replicated-write message). Both communications share the same remote procedure call:

RepliWrite: `(optional uint address, optional string data)->{uint status}`

Heartbeat messages are RepliWrite calls without argument. The actual write replication requests are RepliWrite with address and data content filled. On receiving RepliWrite calls, the backup should reset the timer, possibly write content to the device, and return a normal status. If a primary receives a RepliWrite call, which means both servers think themselves are primary, it compares its node number against the other server, and the smaller node number server always wins, becoming the primary.

1.5. Server states and data structures

i_am_primary

The server maintains this state to indicate whether it is the primary or the backup and acts appropriately. It will change when:

- The backup does not receive a heartbeat message. Backup turns to primary.
- Primary receives a RepliWrite request (double primary), and its node number is larger than the other. This server turns to backup.
- The return status of RepliWrite is not normal. This happens only when the other server thinks it is primary. This server turns to backup.

Under our failure model, the primary-to-backup transition should never happen. We formed these rules when designing a system that has a single network. After network failure and recovery, both servers may claim themselves to be primary, and these rules help the system transform back to a correct primary-backup state. However, we found that there could be a vulnerable window between the resuming of the network and the servers receiving each other's heartbeat. If different clients send write requests to different servers during that short period, two servers may have conflicting data content. Therefore, we changed our assumption to "at most one out of two network fails."

is_backup_alive (used by the primary)

By checking the return status of the heartbeat message, the primary can understand whether the backup is alive. This state is used to expedite the write request–the primary does not issue replicate-write if it knows the backup is not alive.

log (a queue of write requests) (used by the primary)

When the primary sees the backup is not alive, or when the replicated-write message times out (that is, the backup turned down before is_backup_alive state changes), the primary keeps the client's write request in the log queue, so that the recovery procedure can send them to the resumed backup in a later timepoint.

1.6 Client request handling

When the primary receives a read request, it simply reads the content and returns it. When a write request comes to the primary, it writes to and flush the local device, and sends a replicated-write RPC to the backup. If the backup is unreachable, the server pushes the request to the log queue. Write requests return only after the replicated-write RPC successfully returns, or that the request is pushed to the log queue. To simplify the system, write requests are handled sequentially.

As previously mentioned, the backup always rejects client requests.

1.7 Persistence

Servers write to a raw SSD partition which is 260 gigabytes in size. To ensure persistence, `fsync` is called immediately after the write. While the lack of write-ahead logging may cause some non-atomic effect in case of device failure, we believe this will hardly happen if the device's atomic write unit is a 4K page and the write is page aligned, which is mostly true. In addition, as the provided write interface is idempotent, the client can retry the write request and correct the state (although some readers may read incorrect data before such write is issued). In exchange, since the system directly writes to the device, it should have better performance by eliminating the overhead given by the file system.

1.8 Recovery

Regardless of its previous status, a server always resumes as a backup. When the backup resumes, the primary detects it by the heartbeat status and initiates the recovery procedure. The primary asynchronously fetches write requests from the log queue and sends replicated-write RPC to the backup. A log entry is popped from the log queue only after the backup successfully persists the request. The is_backup_alive state is changed to true only after the log queue is cleared.

During the recovery, the primary can still serve client requests. Instead of sending replicated-write requests, it pushes the write requests to the log queue. Theoretically, if the incoming write request rate is higher than the recovery rate, the system can never finish the recovery. We believe this is not a problem in practice, since the client write requests are sequentially handled, it should not run faster than the recovery procedure.

Primary cannot fail when the backup recovery has not finished. For one, the log queue state is not persisted, so it will not know where it left off. More importantly, the backup will time out and take over, thus accepting new write requests (before it entirely recovers), thus invalidating the strong consistency guarantee.

Since the log entries are popped after the replicate-write successfully returns, backup failure during recovery is fine. The primary simply waits for the backup resume and picks up the recovery procedure where it left off.

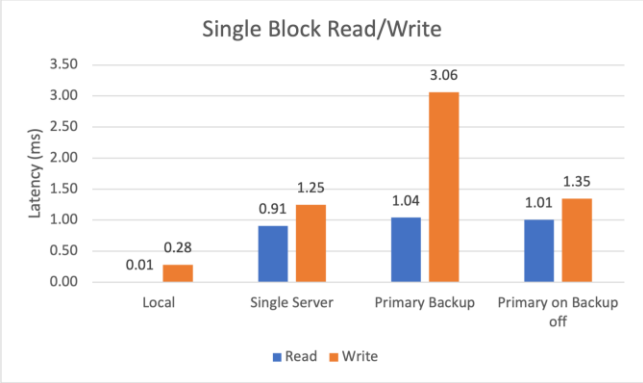
2. Performance Measurements

This section shows the performance measurements of the system. Measurements are done on the CloudLab c220g1 machines, which have a CPU with 32 cores @2.40GHZ, 128 GB 1866MHZ DDR4 memory, Intel DC S3500 480 GB 6G SATA SSDs, 10 Gbps LAN, and 1 Gbps WAN. Three measurements are performed. The first one is the measurement of read/write latency. The second one is to measure the latency of two kinds of requests: 4k-address-aligned and unaligned. The last one is the measurement of recovery time when a server crashed.

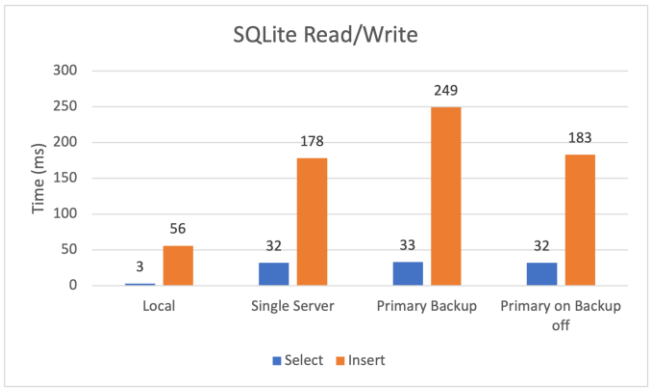
2.1. Latency of Read and Write under Different Scenarios

To test reading and writing performance, we prepared two workloads. One is to read and write a block on the system and the other is to do selection and insertion on SQLite which ran inside FUSE backed by a simple log-structured file system that communicates with our block store service. These two workloads are further run on four machine settings: Local, Single Server, Primary Backup, Primary On Backup Off. Local means we simply read and write (insert or select) the raw device without client and server. It can be a performance reference to the other three settings. The next setting, Single Server, contains a client and a server rather than a primary backup. Primary Backup is the normal operation of our system. Then, the last setting, Primary On Backup Off, shows the performance when the backup crashes.

The following figure shows the measurement results of the workload of single block read and write. For Local, its latency is lowest because no client or server is communicating with each other. Comparing Local and Single Server, we know that RPC calls increase latency a lot and they dominate latency. Single Server is the simplest model. It doesn't have overhead like heartbeat messages, extra write to the log queue, or extra write to backup, so its latency is lower than the following two settings. Primary Backup has the highest latency in write because it must write a request to the backup which includes an extra RPC call. Write in Primary On Backup Off is slightly larger than Single Server because it needs an extra write to log into its disk rather than sending RPC calls to the backup.

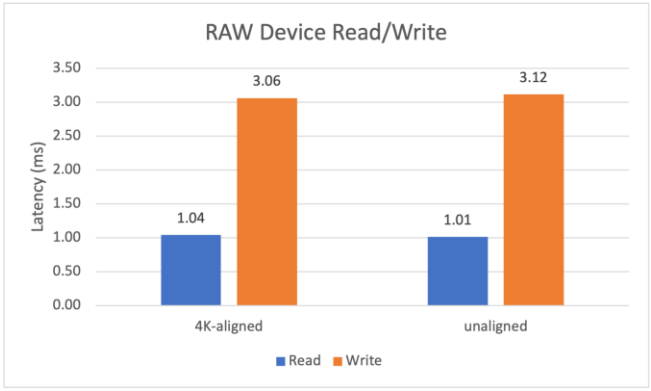


The next figure shows the performance of selection and insertion on SQLite. With this workload, performance comparisons among four settings are similar, which meets our expectations. The reason why the number in this situation is higher is that SQLite selection and insertion access is more than a block.



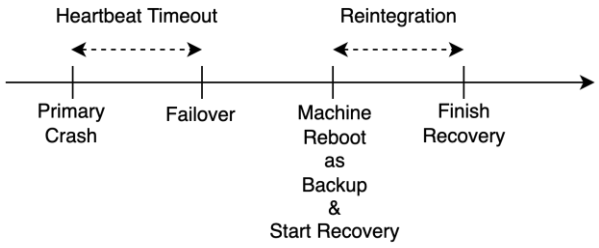
2.2. 4k-Address-Aligned Request and Unaligned Request

The next measurement is for 4k-address-aligned requests and unaligned requests. The result is shown in the following figure. The difference between these two requests is quite small because the latency is dominated by RPC calls.

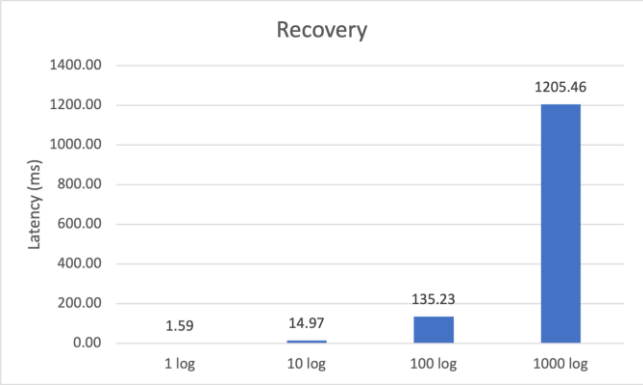


2.3. Recovery

Lastly, we measure the recovery time of the system. Before showing the numbers, we introduce what will happen between primary crashes and recovery finishes.



After primary crashes, it takes some time for the backup to discover and takeover. Backup knows primary crashed by noticing there is no heartbeat for a while. We set this heartbeat timeout to 1 second. Then, when the crashed machine reboots as backup, the recovery procedure will start. The overall time for the recovery procedure is proportional to the number of logs in the primary. The following figure shows the measurement results on different numbers of logs. The results meet our expectations.



3. Testing & correctness

3.1. Testing framework

We implemented a testing framework with Python. The framework will go into different nodes and control our server and client programs using SSH. We also modified our server programs so that when they reach a critical time point (e.g., starting recovery procedure, transition to a primary server …etc.), they will emit some "magic" strings. By capturing these magic strings, the testing framework can know the state of the servers and it can also wait for specific events to happen.

3.2 Test implemented

The following is a list of tests we implemented and passed on our block store service.

- Test single block, Test unaligned blocked
  - The system can write a single block no matter if the block is aligned or not
- Test FUSE
  - The filesystem based on the block store APIs that we wrote should work as expected.
- Test two servers
  - The primary server can successfully propagate a write to the backup server
- Test backup die, Test backup die during the recovery
  - The system should continue operation even when the backup server has failed. And the recovery protocol should eventually succeed after the backup server comes back online even if an earlier recovery attempt failed.
- Test primary die
  - The system should continue operation when the primary server has failed. The backup server should take over and become the new primary and the client library should be able to find it.
- Test network failure
  - The system should continue operation when one of the two connected networks has failed. We simulate network failure by setting up a firewall on the network interface.
- Test overwrite
  - The client should always read the latest value even if some servers have crashed.

3.3 Demo video

https://uwprod-my.sharepoint.com/:v:/r/personal/ctseng27_wisc_edu/Documents/CS739-Project3-demo.mp4?csf=1&web=1&e=vbNHbd