CS 739 Mini-Project 3 Report

Kalyani Unnikrishnan Basava Kolagani Adil Ahmed Vyom Tayal

8th April 2022

Contents

1 Overview

We implement a distributed block storage system, Strongly-Consistent Redundancy-based Utilitarian Block Store (SCRUBS) that is designed to be robust to failures while providing strong consistency. Our architecture is based on the primary-backup redundancy model with writes served only by the primary server but reads distributed across both servers for improved performance. We provide strong consistency in the face of complex and co-ordinated failures. We also implement a new "Handshake" mechanism for servers to elect a primary and backup amongst themselves without any manual intervention. We instrument the code to run custom failure scenarios and crash tests. We also run performance tests for various workloads to benchmark our system. We present the details of the design, testing and evaluations in this report.

2 Design

2.1 Key Principles

Our key design goals are as follows. All design decisions are made to conform to these targets.

1. The system must provide strong consistency and durability. We implement failure logs and write queues to ensure consistency and ordering.

2. The system must optimize for performance where possible. We distribute reads across the servers. We also perform the local write on the primary and the remote write from the primary to backup concurrently. Further, we use "piggybacking" to reduce extraneous network calls.

3. The servers must have a selection mechanism that is robust to human error or server failure. We implement a "Handshake" mechanism to ensure this.

4. The client library is not to be relied on or trusted to maintain consistency. The protocol is designed in a way that servers achieve consistency amongst themselves even in the face of unexpected / malicious behavior from the client.

5. Failures are abstracted away from the client. The client simply sees the final result of the operation without any intervening details.

The assumptions we make about the system are:

1. The client library knows the IPs of the servers. No master or view service is required.

2. Both servers will not fail at the same time.

3. A server will not fail during a recovery operation.

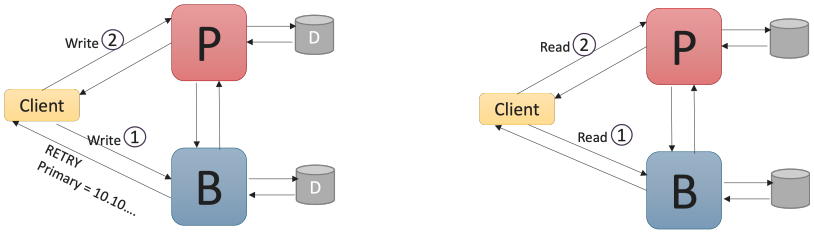

Figure 1: Client Operations - Read and Write

2.2 Regular Operation

The API supports two operations, `read(address)` and `write(address, buffer)`. All operations use a constant block size of 4 KB. The address refers to the offset within the block store at which data needs to be read/written. We support both 4-KB aligned and unaligned offsets. The client library is implemented in C++ along with a Python wrapper to call the APIs. The C++ functions accept a read buffer to populate with the contents of the block while the Python API wrapper simply returns the buffer upon providing the address. The client maintains some state, namely the identity of the primary server and whether the server is running in failover (no backup) mode. The state is updated dynamically and not provided during initialization. The client operation works as below:

1. The client library randomly picks a server in its list and sends a read / write call based on the client request.

2. If the server is down, the client library retries the request with the other server, simultaneously updating the primary IP as well as the failover mode.

3. If both servers are up, the client alternates the read requests between the servers.

4. If both servers are up and the client sends a write call to the backup, it will receive a RETRY request from the backup. This reply will contain the identity of the primary. All further writes are sent to the primary until further state updates, as shown in Figure 1

5. The backup also sends the IP of the primary in the READ response, so the client can update its state and avoid extraneous writes.

6. When one server crashes, the read/write responses also contain the failover information so the client can stop distributing reads and avoid extraneous calls. This additional information is referred to here as "piggybacking" state in the response of regular read/write calls.

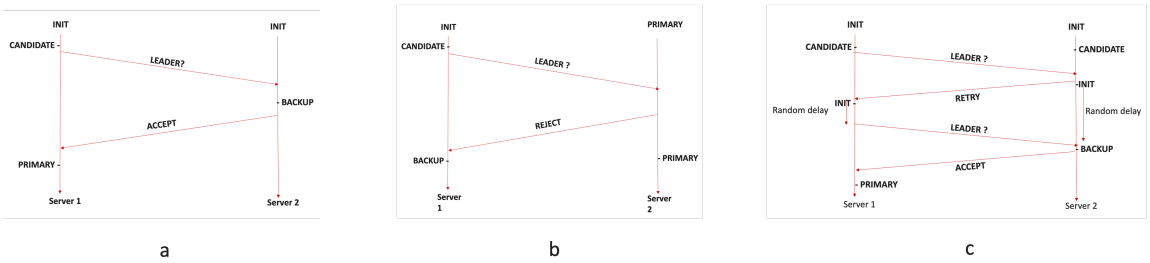

Figure 2: Handshake Scenarios: a) First server sends the election RPC while second is still in the INIT stage. b) First server sends the election RPC while second is already PRIMARY c) Both servers are in the CANDIDATE stage when the Election RPC is sent

2.3 Server Initialization

Upon startup, each server initiates a handshake mechanism as described below. A server can be in 4 states at any given time, namely INIT, CANDIDATE, PRIMARY, BACKUP. The procedure is described in Figure 2

1. At startup, a server is in the INIT state.
2. After waiting for a randomized time, the server appoints itself a CANDIDATE and sends an Election RPC to the second server asking to be appointed by the leader.
3. If the second server is in the INIT state, it accepts the election request and sets it's own state to BACKUP. The first server appoints itself as the PRIMARY upon receiving the accept request.
4. If the second server is already in the PRIMARY state, it rejects the request. The first server then sets itself to be the BACKUP.
5. If the second server is also in the CANDIDATE stage, it sets itself back to the INIT stage and replies with a RETRY RPC, upon which the first server also resets to the INIT stage. The operation is then retried after a randomized timeout to come to a consensus.
6. If the Election RPC call fails, the second server is assumed to be down and the first server sets itself as the PRIMARY.

Mutexes are used to ensure that the operations are thread-safe.

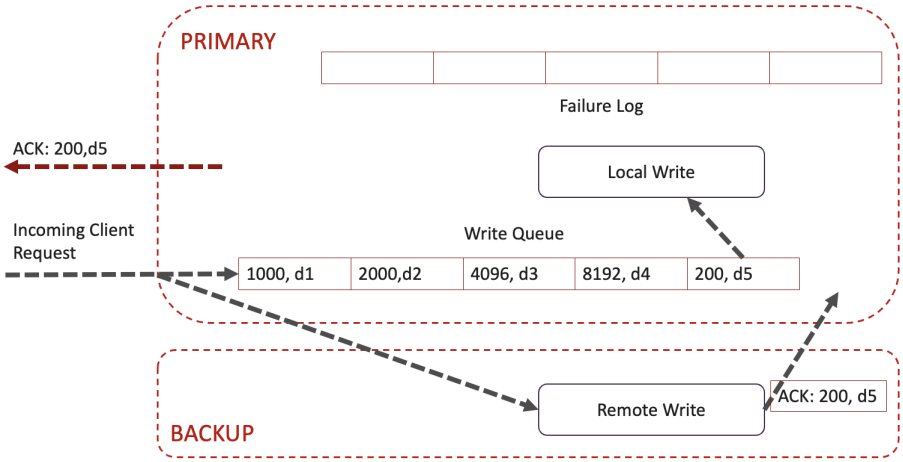

Figure 3: Write Commit Protocol

2.4 Server Commit Protocol

Writes are served only by the primary server while reads are served by both servers. The writes are performed according to the protocol described below, illustrated in Figure 3

1. A `write_queue` is maintained on the primary, in which incoming writes are ordered.
2. Whenever an additional item is added to the queue, the `local_write` and `remote_write` operations are triggered concurrently. The `local_write` writes the data to disk on the primary while the `remote_write` sends an RPC to the backup server to persist the data to its disk.
3. Once both calls return successfully, the operation is popped from the queue and an acknowledgement is sent to the client.
4. On the server side, all writes are written to a large flat file. The `pread` and `pwrite` system calls are used to read from or write to the offset.
5. A heartbeat mechanism is implemented wherein the backup periodically pings the primary to detect if it is operational.

2.5 Crash Recovery

Primary Server Crash. If the primary server crashes, the backup gets no response to its heartbeat signal. When this occurs, the handshake is re-initiated at the conclusion of which the backup is appointed the primary. When the former primary server comes back up, it initiates the handshake and is subsequently appointed the backup - as per the protocol. It is now ready to serve reads and accept remote writes from the new primary.

Backup Server Crash. If the backup crashes, the primary does not receive an acknowledgement of the `remote_write` operation and appends that write to the failure log. When the backup comes up, it initiates the handshake at the end of which it is appointed the backup again. It then initiates a `Sync` operation by sending an RPC to the primary. The primary streams all pending writes in the failure log to the backup. Once the log is cleared, the backup is now ready to receive client read requests.

2.6 Strong Consistency and Fault Tolerance

The primary only returns an acknowledgement to the client when the write has successfully completed at both the primary and backup. Some interesting cases arise that we create special rules for and ensure consistency.

Concurrent Reads and Writes to identical/overlapping addresses It may occur that the primary is completing a write to a specific offset say 2000, while a read request comes in at the offset 1000. This overlaps with the operation currently being performed. If we serve the read, we may be in a situation where the read returns different values based on which server serves the request - violating our consistency guarantees. To avoid this, the primary responds with a RETRY whenever a read overlaps with an ongoing write - at which point the client performs the read from the backup. If the backup has the old data, this remains consistent as the write is not complete yet. If it has the new data, it is still consistent as the write has been completed and the data is up-to-date.

Backup Logs As we perform both `local_write` and `remote_write` concurrently, it may occur that the primary crashes during a write which successfully completes at the backup - leaving the servers in an inconsistent state. To avoid this, the backup maintains the last-completed write operation in a data sructure known as the `transition_log`. If the primary crashes and the backup takes over, the transition log retains this write in its failure log and replays it on the primary when it comes back as the new back-up.

Storing Last Write Address It may also occur that the write completes at the primary but the primary crashes before the RPC is ever sent to the backup. When the backup is promoted to primary, it is never aware of this ghost write - leaving the servers inconsistent when the primary comes back up as the new backup.

To avoid this, we persist the offset of the last completed write at the primary to disk. When the server crashes and comes back up, it requests the data at the stored offset from the new primary to ensure consistency.

3 Implementation

We implement both our client and server in C++. We use the gRPC library to set up the RPC between clients and server. We also implement a python wrapper of the C++ functions for ease of testing.

4 Testing

We define 7 different crash modes as below. We pass the mode as an argument from the client, based on which the server crashes at a deterministic point. The crash modes are listed below. We inject faults in each stage of our workflow and observe the resulting state when both servers are back up. Our protocol is consistent to all faults except a fault that occurs during recovery / sync operations. This falls within our assumptions for the design of the system. We develop a consistency matrix to show fault injection at different points of the workflow, and the end result - as shown in Figure 4.

1. `NO_CRASH`
2. `PRIMARY_CRASH_BEFORE_LOCAL_WRITE_AFTER_REMOTE`
3. `PRIMARY_CRASH_AFTER_LOCAL_WRITE_AFTER_REMOTE`
4. `PRIMARY_CRASH_BEFORE_LOCAL_WRITE_BEFORE_REMOTE`
5. `PRIMARY_CRASH_AFTER_LOCAL_WRITE_BEFORE_REMOTE`
6. `BACKUP_CRASH_BEFORE_WRITE`
7. `BACKUP_CRASH_AFTER_WRITE`
8. `NODE_CRASH_READ`

Client	Primary					Backup				State
	Hand-shake	Read	Local Write	Remote Write	Sync	Hand-shake	Read	Local Write	Sync	Consistent/ Inconsistent
-	X	-	-	-	-	✓	✓	✓	✓	✓
Read	✓	X	-	-	-	✓	✓	✓	✓	✓
Write	✓	✓	X	✓	✓	✓	✓	✓	✓	✓
Write	✓	✓	✓	X	✓	✓	✓	✓	✓	✓
Write	✓	✓	✓	✓	✓	✓	✓	X	✓	✓
-	✓	✓	✓	✓	✓	✓	✓	✓	X	X
-	✓	✓	✓	✓	X	✓	✓	✓	-	X

Figure 4: Fault Injection points vs Consistency Guarantees

4.1 Storage on Server

We use a flat file named `bs` present in a static directory on the ext4 filesystem to store the blocks of data.

5 Evaluation

We ran several benchmarks and conducted measurements to evaluate our final implementation.

5.1 Experimental Setup

Our experiments are conducted on 4 nodes on CloudLab [1], which is a large cloud infrastructure built mainly for academic research. Each node runs Ubuntu 18.04 LTS and uses 2 Intel Xeon Silver 4114 10-core CPUs at 2.20 Ghz. Each node has 192 GB of memory.

5.2 Experimental Results

We measure and evaluate how various implementations perform on several key metrics. We note the performance impact in each case and reason about the differences observed.

5.3 Throughput

We run the primary and backup on two separate nodes and spawn an increasing number of client threads on a third node using threads. The clients do simultaneous reads or simultaneous writes. We measure the throughput for reads and writes both with and without distributing the reads across both servers.

Clients	Total time	Bandwidth (MBps)
1000	1.91	2.41
2000	3.39	2.4

Figure 5: Write Bandwidth

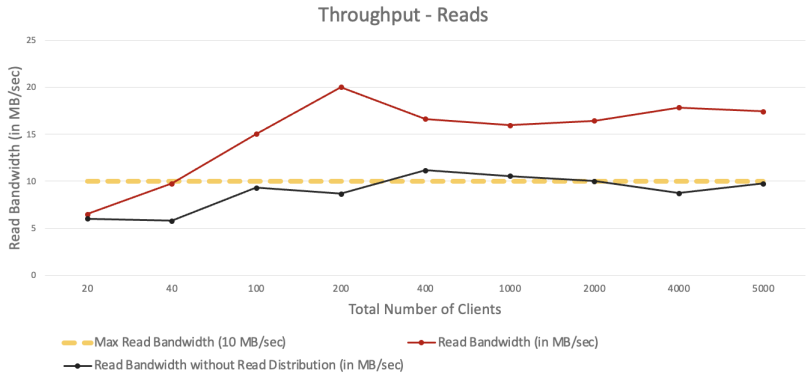

Figure 6: Read throughput comparison for distributed vs single-server writes

As visible in Figure 5, the write bandwidth peaks at 2.4 MBps as they are executed sequentially in a queue on the server. For reads, we observe a bandwidth of 10 MBps when reads are only sent to the primary and a peak bandwith of 20 MBps when reads are distributed, demonstrating the performance improvements resulting from the read distribution, as shown in Figure 6.

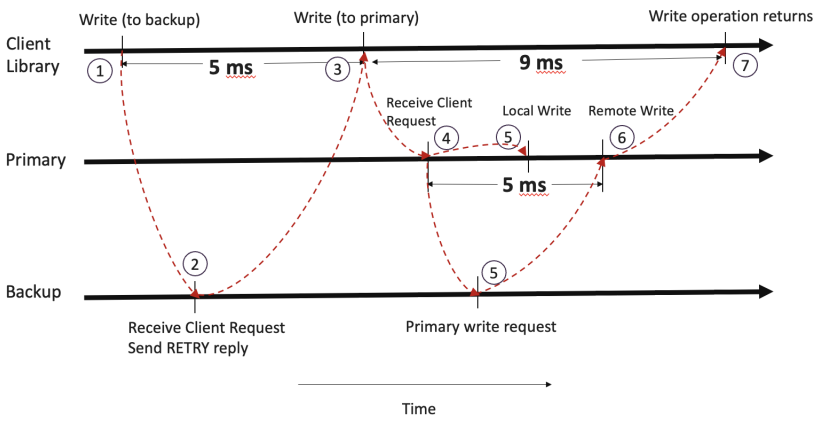

Figure 7: Timeline of Operations

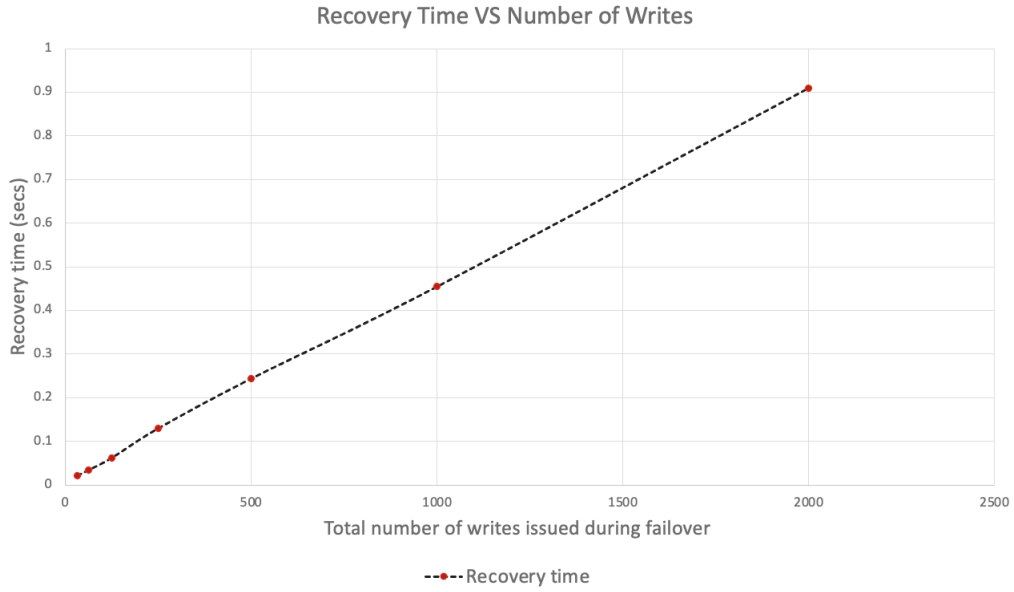

Figure 8: Recovery Time vs Size of Failure log

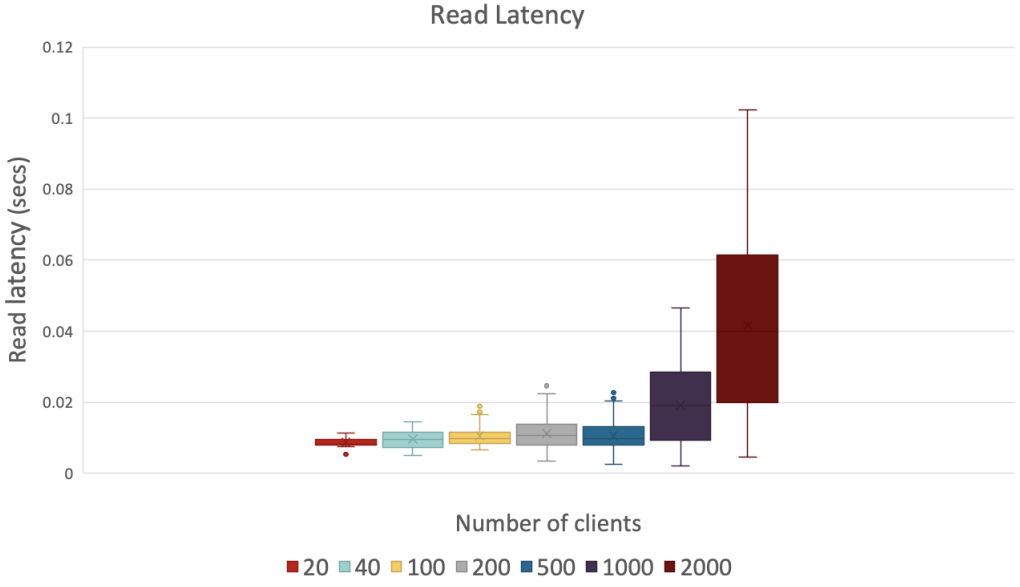

Figure 9: Read Latency for individual clients

5.4 Latency

We chart the worst-case delay in our workflow, when the client first attempts to connect to the backup for writes and needs to retry the request to the primary. We observe an average latency between 9 ms and 14 ms for the write call - as seen in Figure 7. When running an increasing number of clients in parallel, we chart the latency observed by each client for writes and reads. We observe that tail latency for reads increases after 500 clients, due to the link becoming congested - as shown in Figure 9.

5.5 Recovery overhead

We observe that the time taken for server initialization is negligible compared to the time taken to synchronize failover writes from the primary to backup. The recovery time is linear to the number of pending writes that need to be synced, as shown in Figure 8

6 Conclusion

We develop a robust block store, SCRUBS- that is resistant to multiple failures and optimizes performance while maintaining consistency. We implement a handshake mechanism for servers to reach a consensus, queues to order the writes and failure logs to maintain consistency. We run evaluations and crash tests to demonstrate the correctness of the protocol.

References

[1] Cloudlab. `https://www.cloudlab.us/`, 2018.