CS739 P3: Replicated Block Store

Benita Kathleen Britto, Hemalkumar Patel, Reetuparna Mukherjee

1. Introduction

This project was motivated by the Amazon EBS. Our implementation targeted simple read(4K) and write(4K) operations. It provides a highly available, scalable, durable and strongly consistent system. It offers uniform load balanced read operations to squeeze the maximum bandwidth. We have extensively used gRPC C++ library and its rich features like bidirectional streaming.

2. Key Assumptions

We make the following assumptions:

- We only support fail-stop failures
- There are no disk corruptions in our stable storage
- There are no packet drops (stable network)
- A unresponsive node means that the node has crashed
- One node will always be up
- Primary does not crash when the backup is recovering
- No failures during crash recovery

3. System Design

3.1 Architecture

We have 2 servers, Primary and Backup. Primary serves both reads and writes. Backup only serves reads. Both nodes have stable storage where actual data is stored and log where operations are recorded. Upon start, node registers itself with load balancer as either primary and backup and sends a heartbeat after every X seconds. Nodes maintain a bidirectional stream with load balancer as a keepalive message and get to know the status of the overall system. The backup node gets to know in case the primary fails, and elects itself as a primary. This is all hidden from the client which leads to high availability.

The reason behind using a load balancer was to effectively route requests on dynamic addition and removal of nodes. It also helps make the client agnostic of the system state. The client is relieved of the responsibility of distributing load across the server nodes.

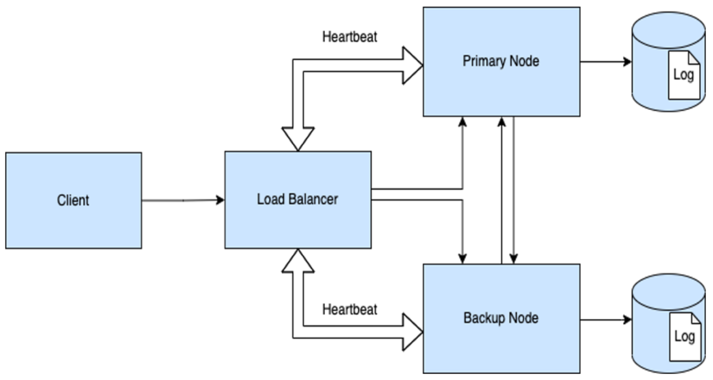


Fig. 1: System Design

3.2 Storage Structure

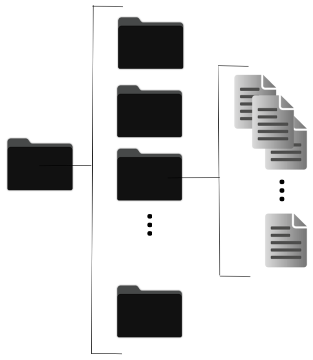


Fig. 2: Replicated block store structure in Stable Storage

- Block Storage Size: 1GB
- 1024 directories, each having 256 files of 4KB size
- Easy scale up/down (not included in this project)
- Can partition data across storage (not included in this project)

4. Possible Primary-Backup Implementations for Writes

This was the interesting part of the project. Read operations do not matter from a consistency perspective as they don't modify the state of the storage, but writes do. We came up with 3 approaches with each having its own tradeoffs.

1. Primary Commits First, Backup Second
2. Backup Commits First, Primary Second
3. Primary Prepares, Backup Commits, Primary Commits (Implemented Approach)

Approach #1: Primary Commits First, Backup Second

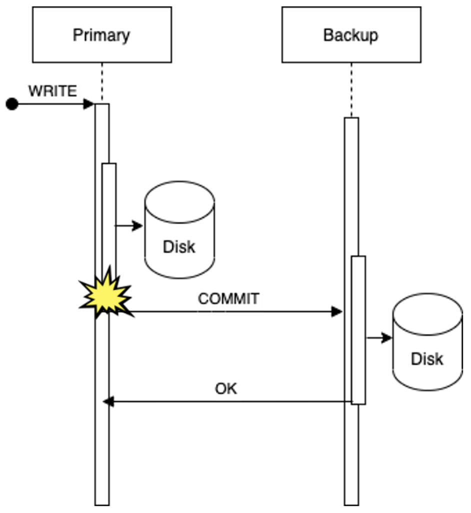


Fig. 3: Approach #1 for Primary-Backup Write

Description: On write request, Primary writes (commits) data first. Primary contacts backup to write. Backup commits data.

Problem: As shown in the diagram, What happens if the primary crashes before contacting the backup? Backup has no knowledge of the request. Client will see the old data and assume that the request failed. Backup cannot trust its commit log to distinguish the real and failed writes.

Solution: Contact backup for all txns. But this becomes way more expensive.

Approach #2: Primary Commits First, Backup Second

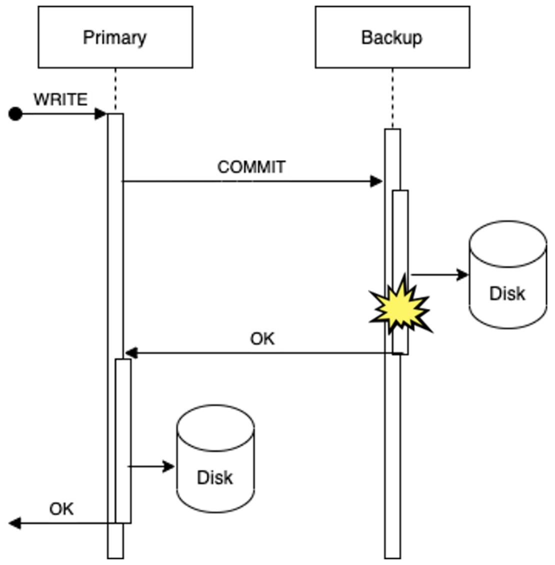


Fig. 4: Approach #2 for Primary-Backup Write

Description: On write request, Primary contacts backup to write. Backup commits data and then Primary commits later.

Problem: As shown in the diagram, What happens if backup crashes before responding to the primary? Primary can commit/abort the txn. It is the same problem as way-1, but in reverse order.

Solution: Contact Primary for all txns. But this becomes way more expensive.

Approach #3: Primary Commits First, Backup Second (Implemented Approach)

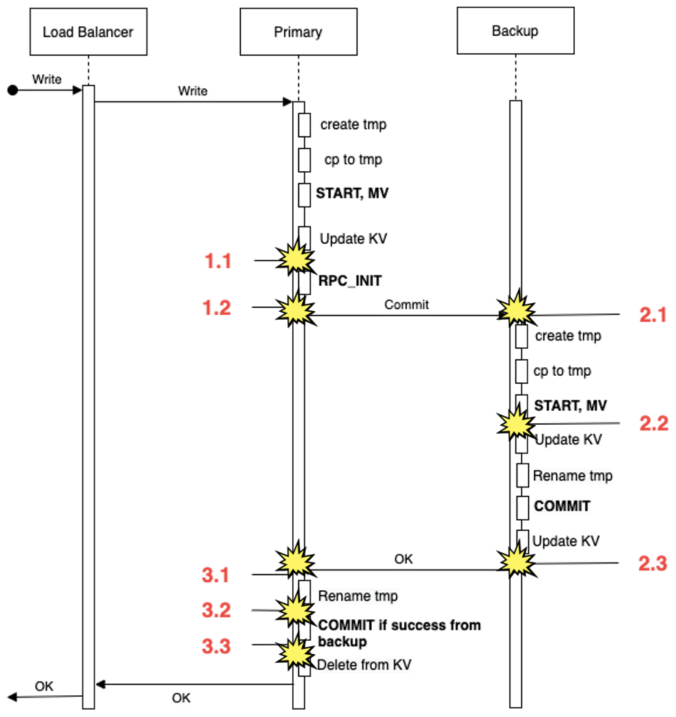
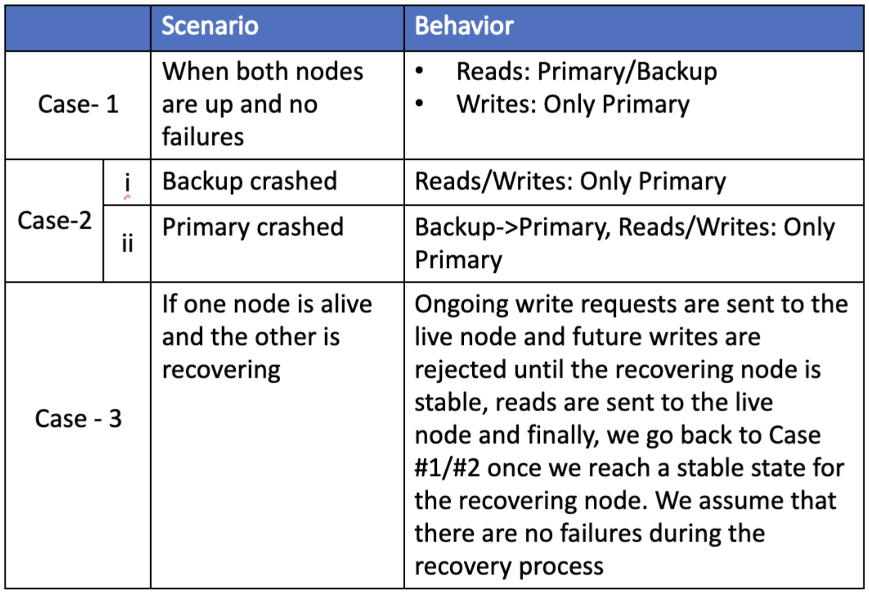


Fig. 5: Approach #3 for Primary-Backup Write

Description: This approach is inspired from the 2 Phase Commit, however it is modified considering we have just 2 nodes. The key distinction of this approach from the previous one is that we are using **prepare** statement to know that we have a transaction. We also added the **RPC_INIT** statement on Primary to know that we have initiated the replication. So on crash Primary can Commit/Abort based on the backup state. We have shown the possible crash points and we describe the recovery protocol in the forthcoming sections. The crash recovery system runs in 2 phases, bringing the transactions in the log to a conclusion and applies all pending writes as needed. We will describe how crash recovery works in the later sections.

System Behavior



		Scenario	Behavior
Case- 1		When both nodes are up and no failures	• Reads: Primary/Backup • Writes: Only Primary
Case-2	i	Backup crashed	Reads/Writes: Only Primary
	ii	Primary crashed	Backup->Primary, Reads/Writes: Only Primary
Case - 3		If one node is alive and the other is recovering	Ongoing write requests are sent to the live node and future writes are rejected until the recovering node is stable, reads are sent to the live node and finally, we go back to Case #1/#2 once we reach a stable state for the recovering node. We assume that there are no failures during the recovery process

Table 1: Our Replicated Storage Behavior in different Scenarios

5. Crash Recovery Protocol

Our crash recovery protocol consists of 4 steps:

- Parsing logs - The backup on reboot parses its own log to create an in-memory key value store representing the state of all the transactions in the log.
- Get pending writes - The backup initiates an RPC request to receive all the writes that have been committed on the primary but are pending replication. This is achieved through the RPL_PENDING state on the primary's log.
- Recover from other states - For any transaction on the backup's log for which it is unaware of the final state, the backup initiates an RPC to get the state of the particular transaction on the primary and apply the changes accordingly.
- Cleanup - All the .tmp files and the log are cleaned up on reboot

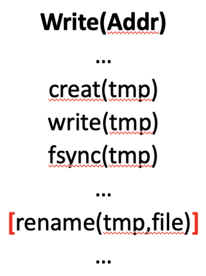
When the backup is recovering, the primary stops servicing writes until the backup has reconciled its state.

6. Failover

When the primary goes down, the load balancer is notified of the broken stream which it relays to the backup in response to its heartbeat. The backup then registers itself as the new primary and the load balancer starts routing all requests to it. The crashed node on reboot registers itself as the new backup, and reconciles its state with the primary. This happens in 4 steps as mentioned under the Crash Recovery Protocol.

7. Durability

To ensure atomicity of updates (writes), we perform updates in 2 phases i.e. we copy the file contents to a temporary file and make updates only to the temporary file. We then perform an atomic rename operation that replaces the temporary file with the original file.



```
Write(Addr)
…
creat(tmp)
write(tmp)
fsync(tmp)
…
[rename(tmp,file)]
…
```

Fig. 6: ALICE diagram

8. Testing Strategy

- CRC for comparing block content
- Macros for predefined crash points
- system(kill) and dereferencing NULL to simulate a crash

9. Performance Results

9.1 Latency

To collate latencies results we run scripts that access different addresses, run on 100 iterations and as single-threaded requests.

Read Latency without Crash

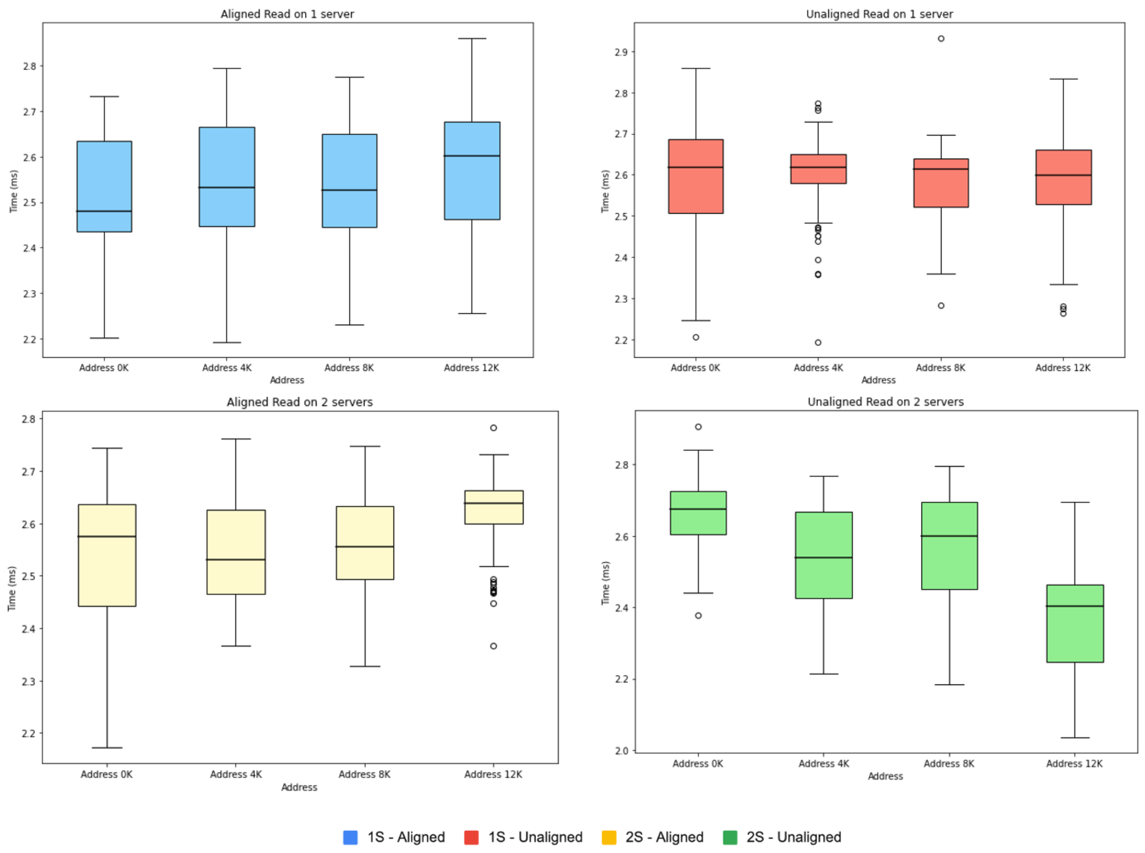


Fig. 7: Read Latencies on 1 and 2 servers with no crash

We observe that reads on 1 server and 2 servers perform the same because reads can be performed on any node (primary/backup). We also observe that unaligned reads have a higher latency than aligned reads because unaligned reads must access 2 different files in comparison to accessing a single file on aligned reads.

Write Latency without Crash

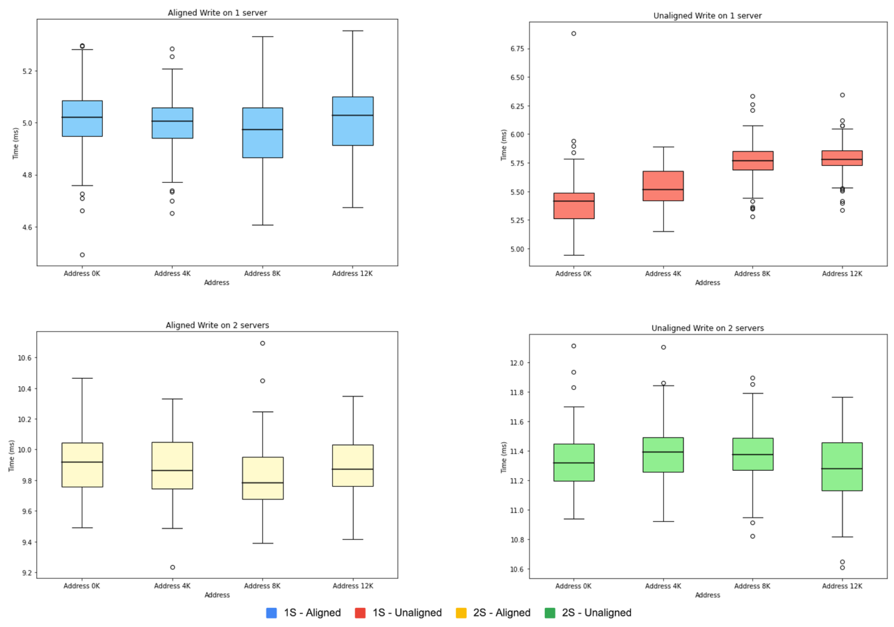


Fig. 8: Write latencies on 1 server and 2 servers with no crash

Write latencies on 2 servers are close to double of the latencies on 1 server as the write must be replicated on 2 different nodes. The latencies on 2 servers are less than double of the latencies on a single server because there is no additional RPC from the client to the load balancer (i.e. 1 server write = RPC (client->load balancer) + RPC (client->node); 2 server write = RPC (client->load balancer) + RPC (client->node) + RPC (node->node). We observe a similar trend of unaligned writes having higher latencies than aligned writes, per our expectation.

Read and Write Latency with Crash

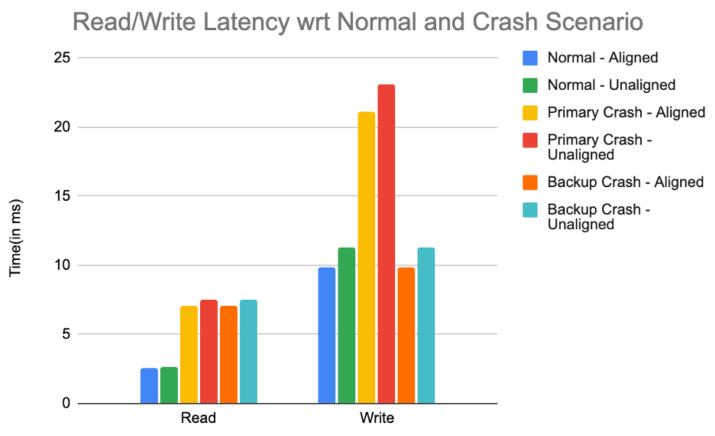


Fig. 9: Latencies of Read and Writes with crash

The normal case is when two servers are up. For calculating read latencies with crash, we crashed the primary and the backup and show the latencies for both. We observe that latencies when there is a crash on the primary or the backup are the same. This is because the client library retries the request and the request will be routed to the next server (round robin). Thus, the latencies with crash are higher than the normal case due to the retry with exponential backoff mechanism on the client library.

Write latencies on primary crash are higher than backup crash because the primary is responsible for performing the write on itself and replicating it on the backup. When the primary crashes on a write, we need to invoke the failover process wherein the backup becomes the primary and the client library will retry the request and is routed to the new primary. However, when a backup crashes, the write will be performed by the primary and there is no disturbance in the write process. Thus, writes with crashes on the backup perform like the normal case.

9.2 Scalability

Read Performance

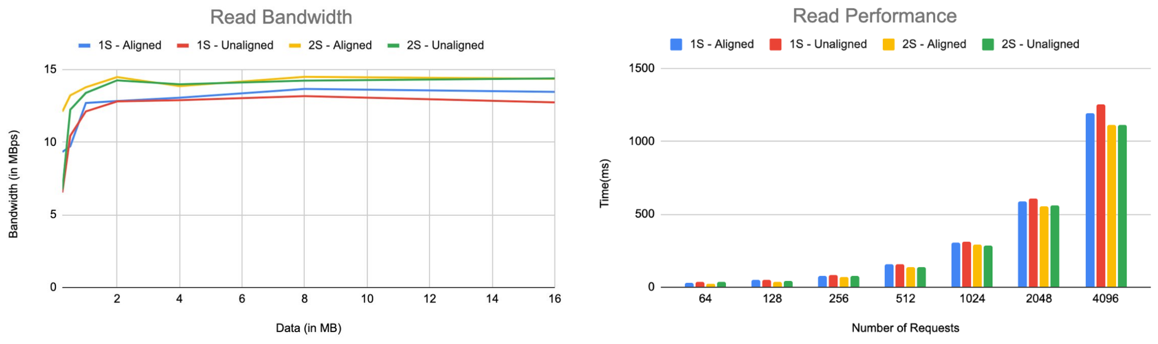
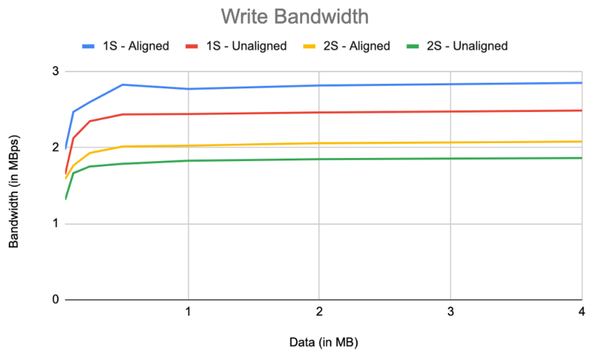


Fig. 10: Read Scalability in terms on Bandwidth (left) and Time taken (right)

Key Observations:

- For 1 server, Aligned vs Unaligned difference is more noticeable when number of request increases (Reason: 2x file reads per request).
- For 2 Servers, Aligned vs Unaligned difference decreases when number of request increases (Reason: Load Balancer + load distribution over 2 servers).
- Bandwidth: 2 Server > 1 Server; Aligned > Unaligned

Write Performance



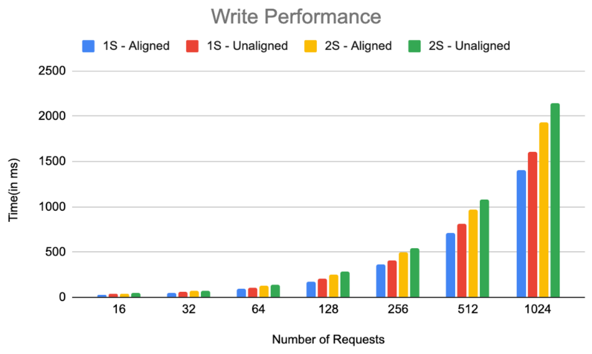


Fig. 11: Write Scalability in terms on Bandwidth (left) and Time taken (right)

Key Observations:
Bandwidth: 1 Server > 2 Server and Aligned > Unaligned

9.3 Recovery

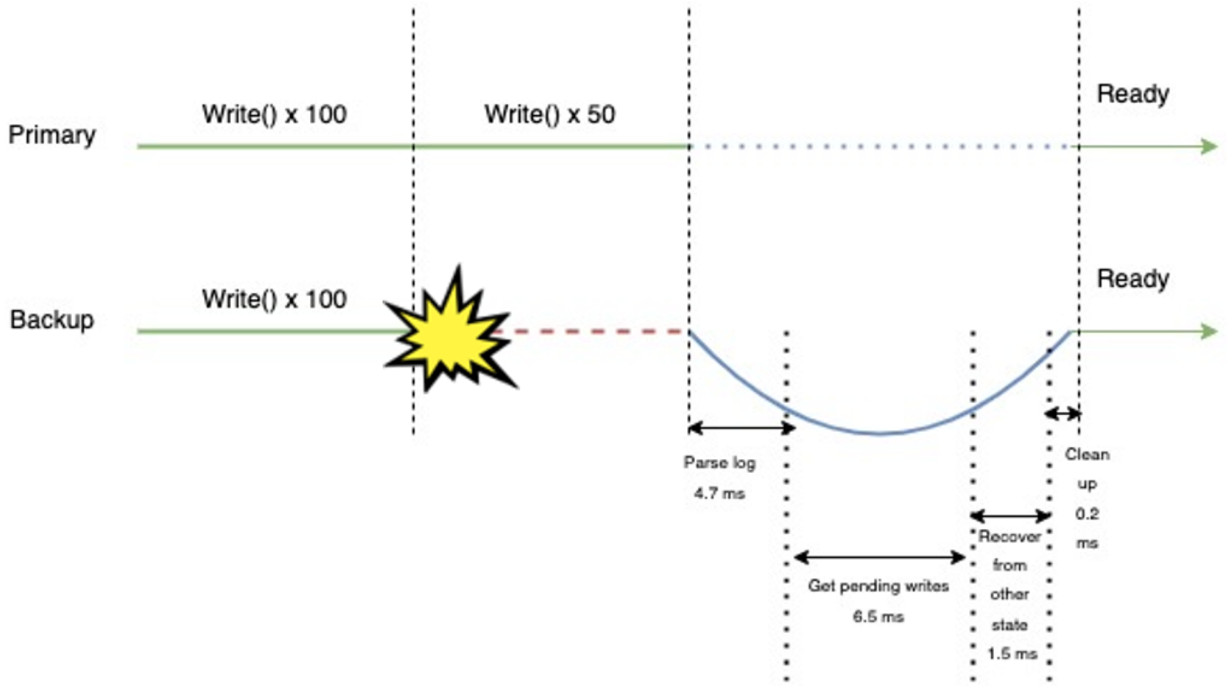


Fig. 12: Crash Recovery Timing on the 4 steps of Recovery

10. Demos

Link to onedrive

11. References:

1. https://aws.amazon.com/ebs/
2. https://grpc.io/docs/languages/cpp/basics/
3. Two-phase commit protocol - Wikipedia
4. Replicated Data Consistency Explained Through Baseball - Microsoft