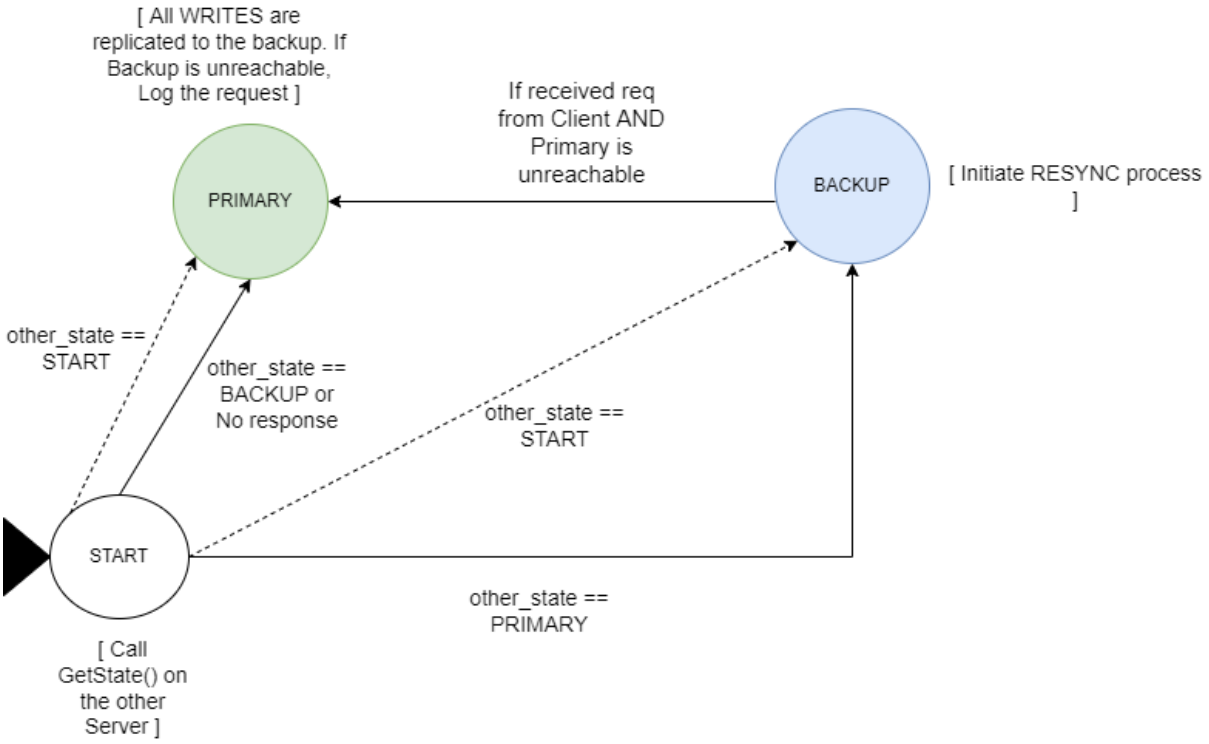
CS 739 P3 : Replicated Block Store

-Lincoln Spartacus James
-Madhav Kanbur
-Himanshu Pandotra
-Pavithran Ravichandiran

Our replicated block storage design is based on a primary-backup replication scheme. The design provides Strong Consistency by sending writes to the backup node(if alive) from the primary, and once completed on the backup, sends an acknowledgement back to the client . All the read/write requests are performed only by the primary node. The Backup redirects clients to the Primary (if it's alive) or promotes itself to the Primary (if Primary is dead). The client can have only one outstanding read or write request at a time. The API from the client provides support for 4KB read/write requests. It supports logical storage space of 256GB and allows unaligned addresses to be accessible. We assume that no network partition will occur between the Primary and Backup.

Server:
State Machine:



- Initially both servers will boot into STATE_START as shown in the state diagram.
- GetState RPC is sent to check the state of the other server.
- If it sees the other server state as primary, then it sets itself to the backup state.
- If the other is in backup, then it becomes the primary node.
- If the other is STATE_START, then both of them pick up the default_state passed in the compiler options to classify one as primary and other as backup.
- If the backup receives the read/writes, it first checks if the primary is still alive by sending a GetState RPC. If primary is not available, it promotes itself to primary. If

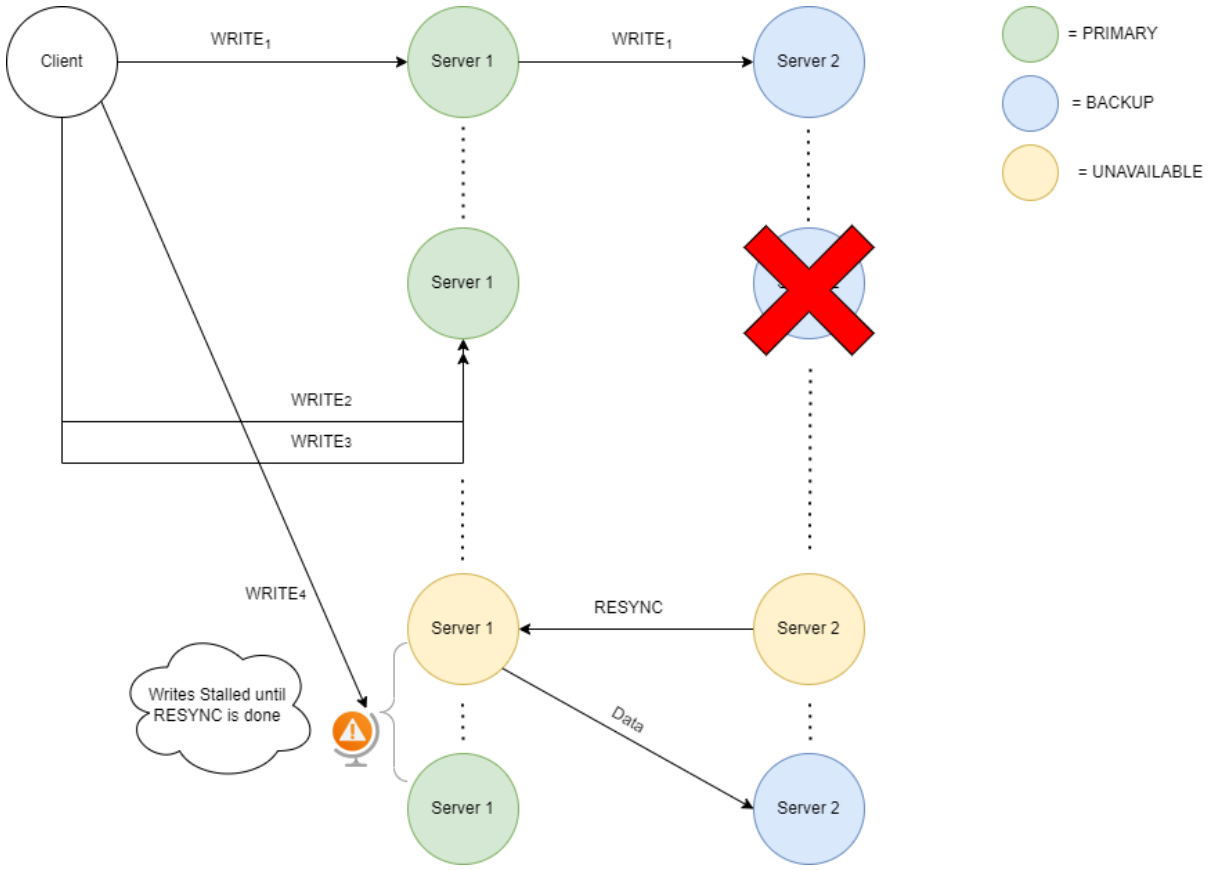
the RPC returns a success state, it implies the primary is still alive and the response code to the client is set to REDIRECT, which will cause the client to contact the other server.

- Primary services all requests. If the Backup is unavailable, it logs all affected block numbers in a std::set so that it can be used during RESYNC.

Design details:

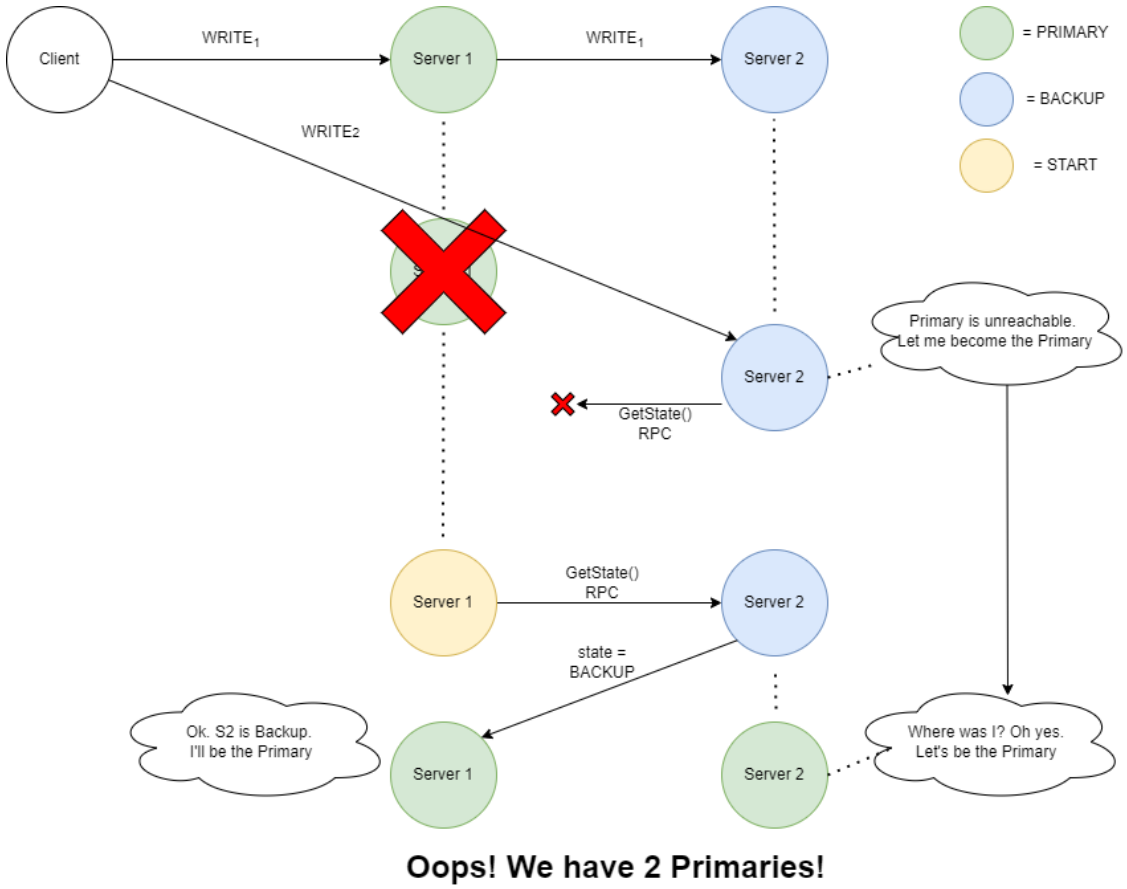
- pread/pwrite are performed on read/write RPC requests from the client or from primary.
- The primary/back-up node reads/writes on a single 256GB file. Handling aligned/unaligned accesses is made simple by using a single file and also avoids opening/closing of files on every read/write RPC request if multiple files are used.
- fysnc() is performed on each pwrite().
- In-mermory set (std::set) is used to keep track of all the updated block numbers when the backup is dead and the <block numbers, block data> pair will be streamed back into the backup during the reintegration/resync process.
- The duplicate write requests (writes to the same addresses) can be avoided by using the set and only the last write will be sent to the backup on reintegration.
- gRPC is used between client-server and primary-backup nodes.

Resync/Reintegration:



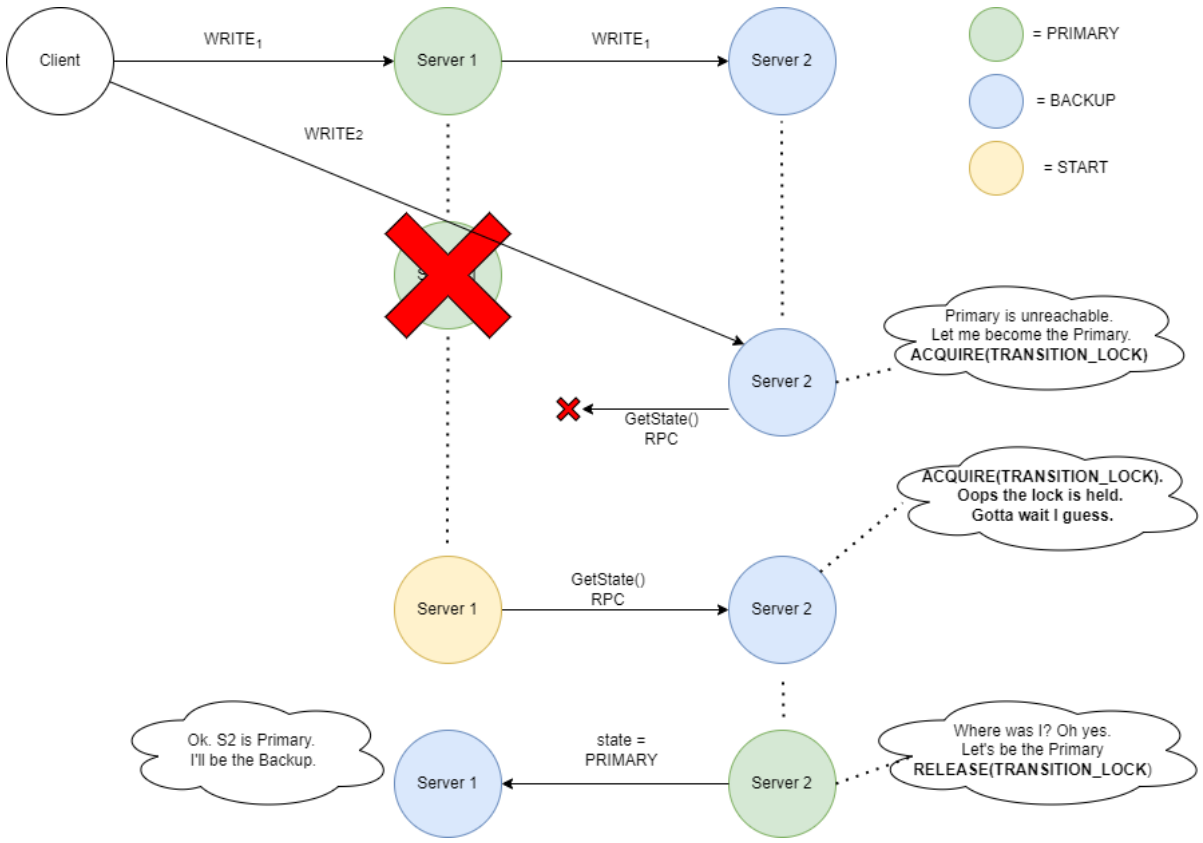
The diagram above explains the process of reintegration. Initially both the primary and backup nodes are alive and all the write requests are forwarded to the backup before sending acknowledgement to the client (write1). Once the backup has crashed, all the block numbers of the subsequent write requests(write2,write3) will be stored in a set (Block number = address/4096). In case of unaligned accesses, the next block number will also be stored. Once the backup comes alive, it initiates the resync process by sending a RESYNC RPC to the primary node. Primary node on seeing the resync request, streams all the write requests stored in the set along with its data by reading from its own file. During this time, no write requests(write4) are serviced by the primary to avoid race conditions.

Backup-Transition Race condition:



During our design, we carefully examined various race conditions which can cause the 2 servers to boot in primary or backup mode and avoided it. The above diagram explains such a case. Initially, both the primary and backup are alive and the client requests are handled as seen in WRITE1. Once the primary has crashed, the client detects the failure and forwards the WRITE2 request to the backup. The backup on receiving the write request sees that it is in backup mode and checks if the primary is still alive by GetState RPC. This RPC fails since the primary has crashed and the backup decides to promote itself to primary. During this time, the primary can come back to life and see that the other server is in the backup state and can also promote itself to the primary state. This kind of race condition will cause 2 primaries to exist in the design which can be catastrophic.

In order to avoid the race condition mentioned above, we use a mutex lock (TRANSITION lock). The backup node acquires the lock once it decides to become primary. During this time, the GetState RPC calls will not be serviced and the response will be sent only when it promotes itself to a definitive primary state, causing the other server to boot in backup mode. The same is shown in the below diagram.



Demo Videos :

- Availability - Crash hidden from user - https://docs.google.com/file/d/1UmBQEqQj2KiWeosc7DhD2qFE1iZgOBXX/preview
- Strong Consistency - Read back latest value - https://docs.google.com/file/d/14NXFGE0vecb-l6I4U7HZlxUZ4Vib3WIh/preview
- RESYNC process - https://docs.google.com/file/d/1fqPV0cyLNjER_dgmSKBd9NvxN-tLg5Hg/preview
- Fail-over process - https://docs.google.com/file/d/1uUf8F3JcSq4kuZT8S4bBL0BvT1p-eo6-/preview

Experiments and Performance Results:

We evaluate our replicated storage implementation by measuring the latency of block read and write operations under various scenarios. Figure 1. shows the read and write latency when both - the primary and backup servers are operational. As seen from the plots the median read latency is much lower than median write latency. This behaviour is expected in our design as reads are served by the primary directly while writes are persisted to the primary's disk as well as replicated with the backup and only then is an acknowledgment sent back to the client.

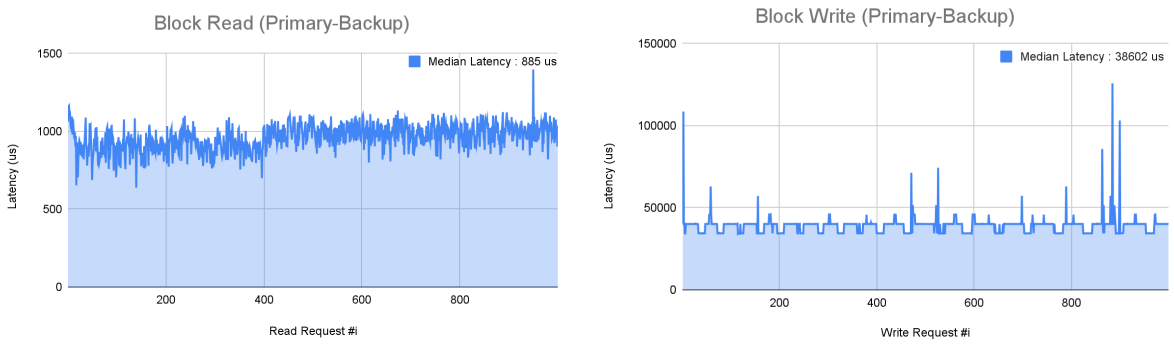


Figure 1.Measurement of Latency for Block Read and Write commands

Figure 2. compares the latency results for block storage servers with primary-backup replication and without any replication. As seen from latency results for block write, a primary-backup system sacrifices some latency to improve availability.

Figure 2. Comparison of Latency in Two nodes vs Single node

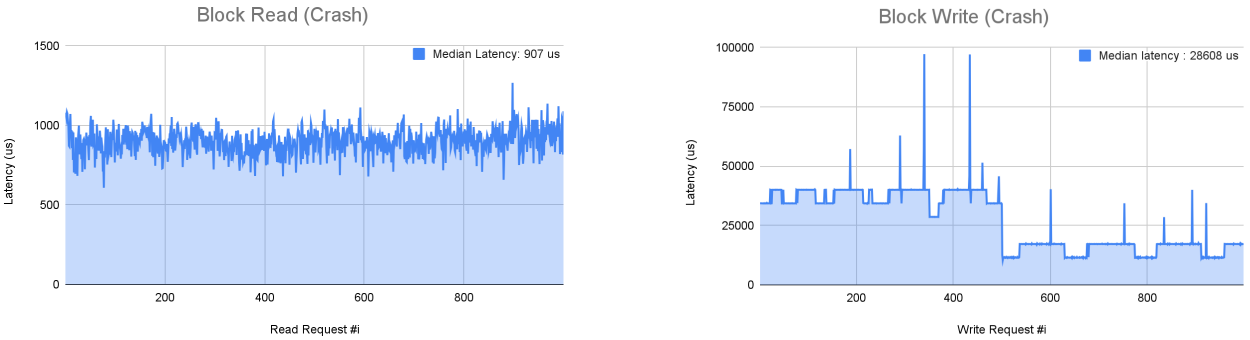


Figure 3. Measurement of Latency for Block Read and Write commands(with crash)

In Figure 3, we simulate a scenario when the primary server crashes while the clients are performing read/write operations on the server. As we can see from the plots, the read latency is unaffected after primary crashes as the backup server starts serving the read commands. The write latency is reduced as now we only have one server in the storage system and writes do not need to be propagated to a backup, thereby reducing the latency. Even though, now the availability is compromised.

We evaluate how our system handles aligned and unaligned read/write instructions. We implement our storage system by writing to a single large file on an EXT4 filesystem. Hence access to any block, aligned or unaligned address doesn't need any separate processing. This is shown by our results below. As we can see in Figure 4, both read and write latencies are similar for unaligned and aligned cases.

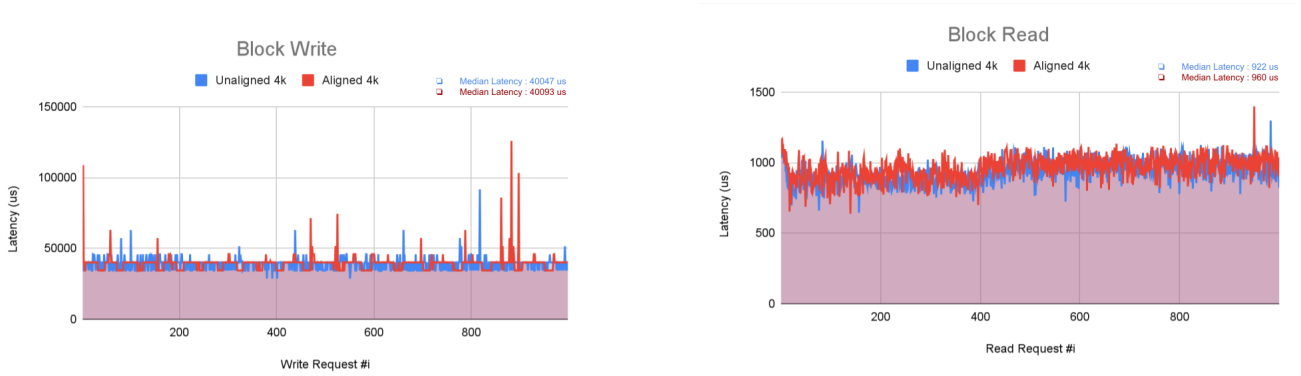


Figure 4. Comparison of latency in 4K aligned and unaligned Block Read and Write commands

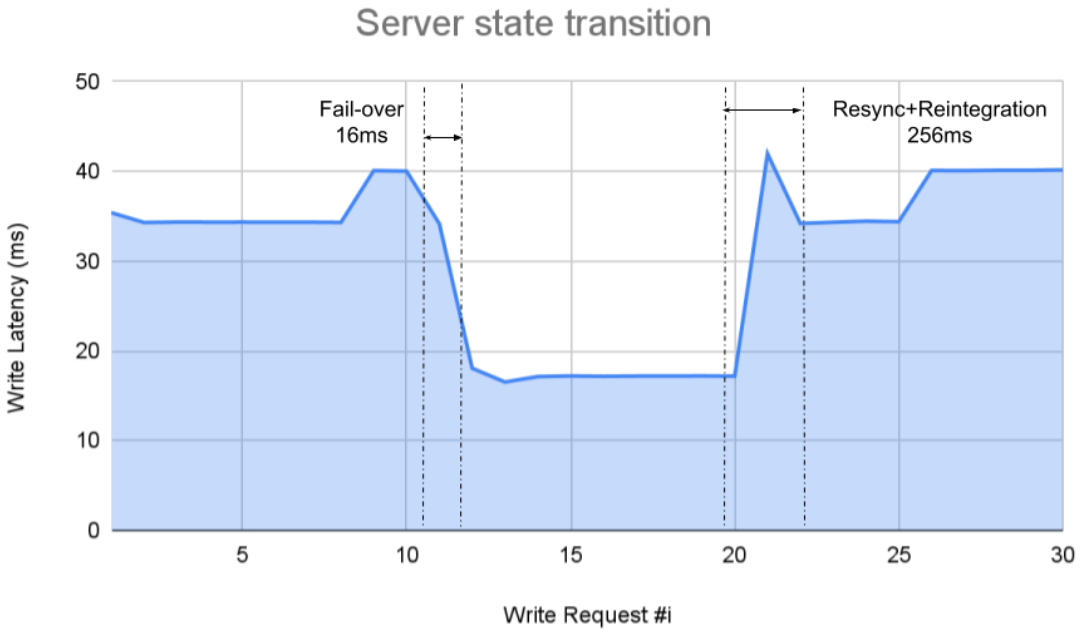


Figure 5. Server state transition (observed write latencies from client side)

Figure 5 shows how write latencies change as the system transitions from the state when both servers (primary-backup) are alive to one server alive and then back to both servers alive.

The Fail-over time (time for backup to take over when primary crashes) is around 16ms.

Once the server comes back alive, it joins as a backup server and all the logged writes are synced with this server to bring it to the same state as the primary. The time taken by this resync and reintegration process depends on the size of the write log. In Fig 5, 10 write requests are sent when the backup is down; the resync process was found to take 256ms. After this point, both servers operate in normal mode with write latency of around 40ms.

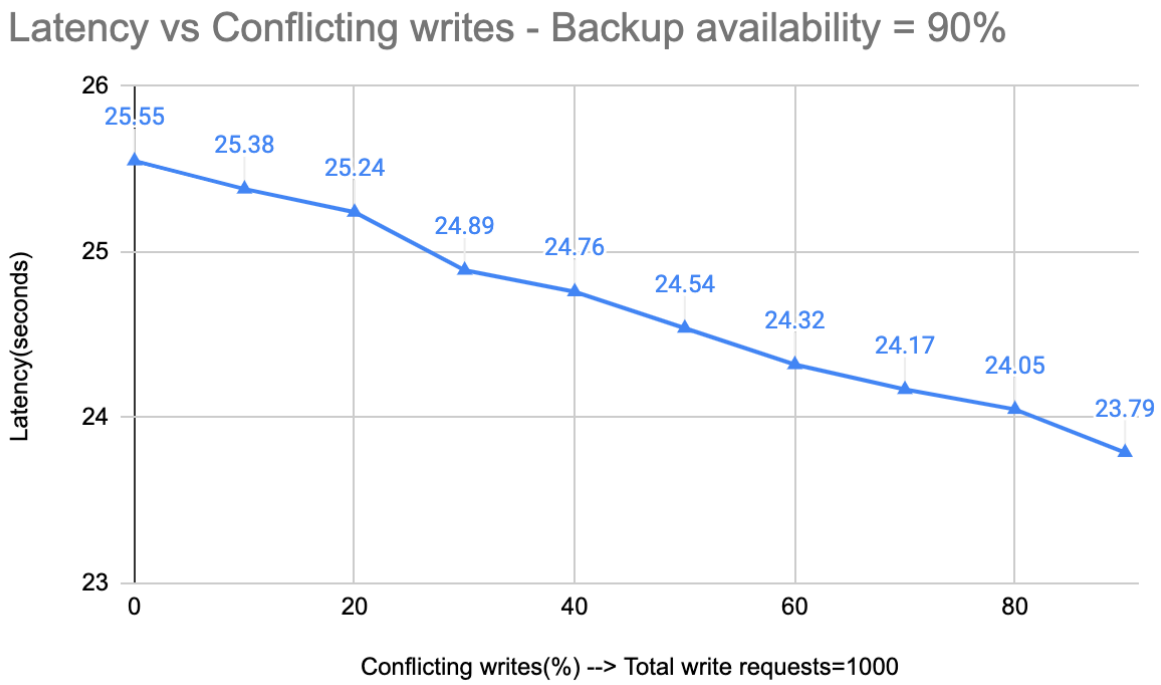


Figure 6 - Latency of Writes for varying % of conflicts

In Figure 6, we explore the latency of write operations for varying percentages of conflicts in our Write set. This experiment was performed in the presence of a flaky Backup server that crashes frequently and maintains 90% availability. A high conflict percent indicates that more writes map to the same block number. We observe that as the conflict percentage increases, the time taken to perform writes decreases. This is because the time to RESYNC our backup falls as more writes map to the same block number(s). This is expected, as our design transmits only one copy - the latest copy of a block instead of re-doing every single write during the RESYNC process.