P3 Replicated Block Store

Group 14: Aakarsh Agarwal, Muhammad Asad Khan, Bijan Tabatabai, Salman Munaf

Table of Contents

1. Design and Implementation

1.1 Replication Strategy

To replicate data, we chose to arrange our servers in a Primary/Backup configuration. In this setup, the client only communicates with the current primary. If the client sends an RPC to the backup, the backup will reject the request and return an error code indicating that it is not the primary. In order to keep the data on the two servers consistent, the primary forwards all writes to the backup and does not send an acknowledgment to the client until it has confirmation that the backup has persisted the write or that the backup has failed.

1.2 Durability

To ensure strict consistency, we must also ensure that data is stored to disk durably. We do this with our take on undo logging. When a write arrives at the primary, it copies the data that will be overwritten into a .undo.tmp file. Then, after we know this data has been persisted the file is renamed to only have the extension .undo. We do this to ensure that we don't accidentally "undo" corrupted data into our blockstore if the server happens to crash while writing to it. Then, the primary forwards the write to the backup where a similar process occurs. When the backup signals to the primary that it has persisted the write, the primary deletes its undo file and finally acknowledges that the write has been persisted to the client.

If an undo file exists when a server starts, that means that the server crashed in the middle of a write. To make sure the blockstore is in a consistent state on startup, the server checks its local directory for undo files. If it finds any, the server rewrites the data in the undo file into the appropriate location. After an undo file is processed, it is deleted. The server only begins listening for incoming RPCs after it has finished processing its undo files.

1.3 Crash Recovery Protocol

Every second, the primary sends the backup a heartbeat message. If the backup does not receive a heartbeat from the primary in over five seconds, it assumes that the original primary has failed and promotes itself to the primary. If the client sends an RPC in the time between when the original primary fails and the backup becomes the primary, the client library will retry the request until either the promoted primary returns or a timeout much greater than five seconds elapses. This "failover" strategy is more or less transparent to the client other than increased latency for the request. Our choices of a one second heartbeat period and five second promotion timeout are arbitrary and could likely be tuned to have better performance.

Heartbeats have an added benefit of signaling to the primary that the backup has crashed or has recovered. If the backup does not respond to the primary's heartbeat or forwarded write, the primary assumes the backup has crashed. When the primary believes the backup is unavailable, it stops attempting to forward new writes to the backup. Instead, it keeps a queue in memory of the writes the backup has missed. If the backup restarts, it will begin responding to heartbeats again. When this happens, the primary empties its queue to bring the backup back up-to-date. After this is done, the primary and backup revert to normal operation.

2 Performance and Measurements

2.1 Correctness

We provide video demonstrations of our system correctly tolerating crashes at https://drive.google.com/drive/folders/1qoISpaF0hyk2rpJqUxGmOJYN-Fg_hEwg?usp=sharing.

Crash_primary.mp4 shows that if the client sends a request after the primary crashes, it will attempt to send messages to the primary and backup which are rejected, either because the primary is done or because the backup is not the primary, until the backup identifies that the primary is down and promotes itself. After it does so, it responds to the client's request.

Crash_secondary.mp4 demonstrates our strategy of having the primary keep a list of write requests that occur after the backup has failed. In that video, after we crash the backup, the primary receives a write request, which it logs. When the backup comes back up, the primary forwards the write it missed to maintain consistency between the two data stores.

Undo_log_demo.mp4 demonstrates our crash consistency. It shows that even if the primary crashes in the middle of the write and corrupts the data store, the corrupted data will be recovered by replaying the undo log. To simulate the corrupted state and server crash, our server checks what value is being written to the system after the undo file is created. If that value is equal to our seminal value “hello_world” we write corruption to our data file and exit the server.

Finally, multiple_cmds.mp4 shows that our system can synchronize writes to overlapping regions. For five iterations, we send 10 writes and 2 reads in parallel. We see that in each iteration, the reads only return data values that have possibly been written because the parallel writes were properly synchronized.

2.2 Performance

Experimental Setup: To measure READ/WRITE latency, we used 3 CloudLab C220g1 nodes connected on a 1Gbps network link. We made sure to run client, primary, and backup on different nodes to have realistic readings.

READ Latency: The READ latency in our project is similar for both replication and single server cases as we service READ requests only from the primary. We found the average latency of READ requests to be 910 microseconds.
In case of primary crash, there wasn’t that much effect on READ latency other than the failover time as the time it takes to service READ request is very small (910 microseconds).

In case of backup crash, there wasn't any affect on READ latency at all since READ requests are only serviced from the primary.

WRITE Latency: The WRITE latency in case of replication was found to be 64 ms and in case of single server, it was around 31 ms.

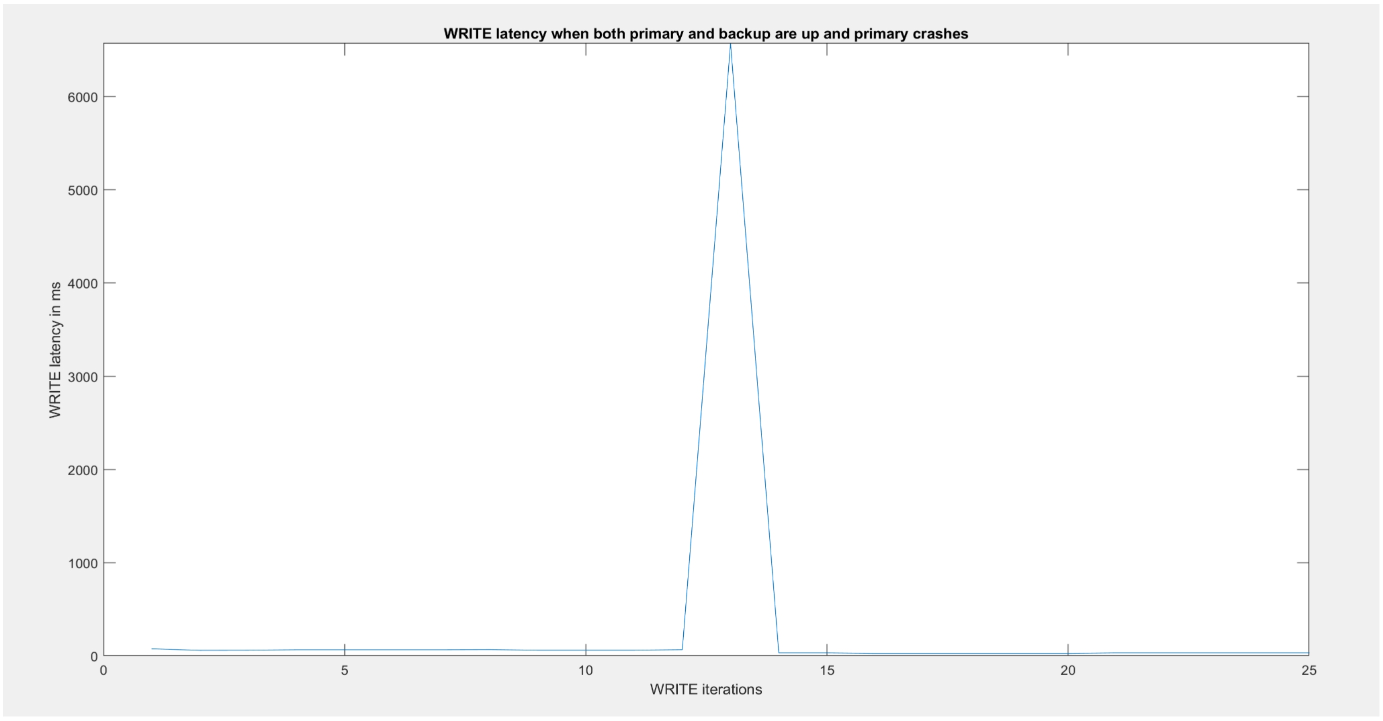

Fig. 1

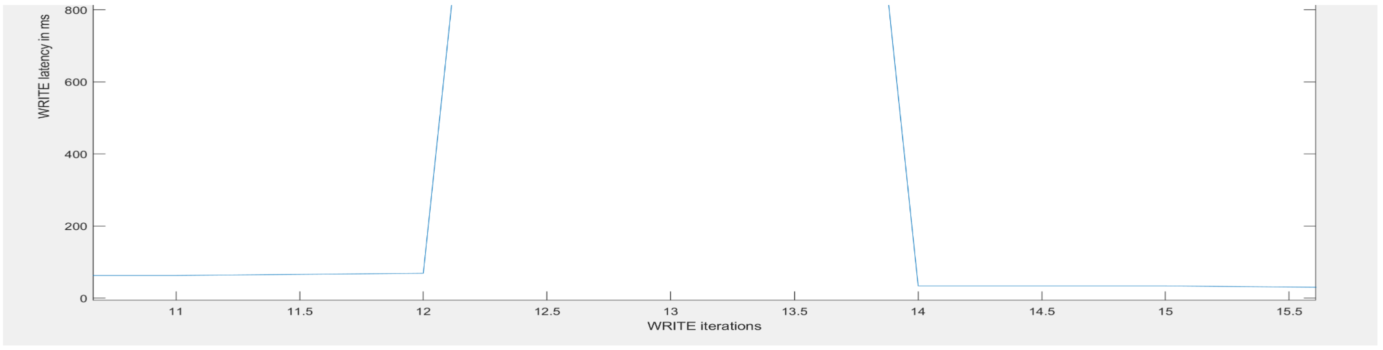

Fig. 2 (zoomed in version of Fig. 1)

Fig.1 represents the case when primary crashes. The impulse in the middle of the graph is due to primary being crashed and backup taking over. As can be seen from Fig. 2, the WRITE latency before primary crash is more than the latency after primary crash. It is due to the fact that after primary crashes, there is only one server and thus no replication. We found our WRITE latency to be 64 ms before primary crashed, and 31 ms after primary crashed, as expected due to no replication in the latter case.

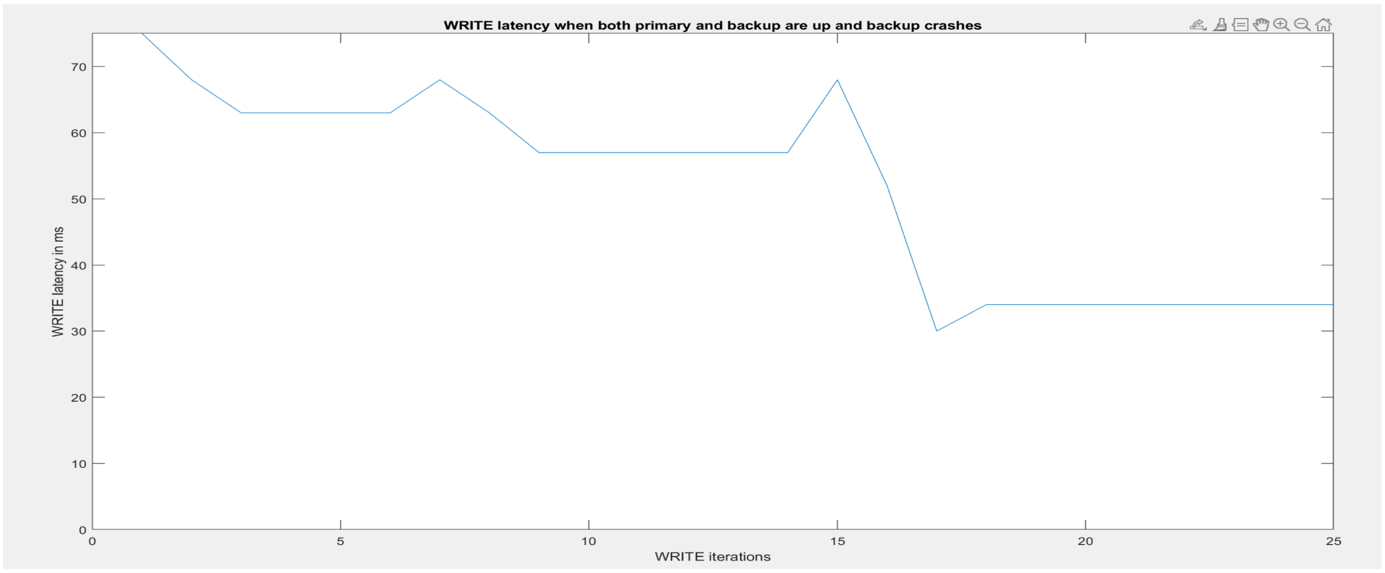

Fig. 3

Fig. 3 represents the latency incase of the Backup crash. The drop in graph represents the instant when Backup crashed. As expected, we can see the latency dropped after the backup crashed because of no replication.

Recovery Timeline: We found our failover time to be 5548 ms.
The average time it took to reintegrate 20 WRITE requests was 679 ms. Once the backup crashes, the primary starts logging all WRITE requests. And as soon as Backup comes back, this log is transferred to Backup. This reintegration time of 679 ms represents this case.

Aligned vs Non-Aligned READ/WRITE:

Read	
Aligned (µs)	Unaligned (µs)
917	908

Write	
Aligned (ms)	Unaligned (ms)
62	71

The above table shows the READ and WRITE latencies in case of aligned vs non-aligned requests. There is not much difference in READ and WRITE latencies in align vs non-aligned case because the method is similar for both types with the exception of using two locks instead of one in case of non-aligned requests.