FireStore: Strongly Consistent Block Storage

Aditya Kaushik Kota	Akshat Sinha	Himanshu Sagar
kota@cs.wisc.edu	asinha32@cs.wisc.edu	hsagar2@cs.wisc.edu

Section 1: Introduction

We design a simple block storage system (FireStore: our strong desire for consistency), with two nodes. One node acts as Primary, while the other starts as the Backup node. Clients only connect to Primary and all read/write requests are forwarded to Primary. The nodes send a heartbeat request(Ping request) to each other and receive a Pong response. The Heartbeat response also contains a boolean value to indicate if it is from a Primary Node. This helps in detecting if the connected node is in Primary mode. The client connects to the backup in case of failure at the primary node, crash at backup is transparent to clients. We handle node failure using an active queue to store new write requests, which are replayed on the node during recovery. We design a correctness test to check for consistency with a single client. We compare the result of the consistency check with and without fsync to see the effect of fsync on our correctness.

We can summarize the features of our system as:

- Two nodes design; can handle failure of 1 node
- Heartbeat mechanism to check node failure
- Correctness test with a single client
- Client automatically switches to backup in case of primary storage failure.

We describe our design in more detail in Section 2, followed by testing and measurements in Section 3.

Section 2: Design & Implementation

2.1: Replication Strategy

Our system uses two nodes, one acts as Primary, while the other is in Backup mode. The system guarantees failure tolerance of one node and remains available during this time. Since we offer consistency and availability, we are giving up on the ability to handle network partitions.
The client connects to the primary for all read and writes operations and in case of failure, it will try to connect to the backup node. Figure 1, shows read and write protocol.

- Write protocol: When a write request is received at primary, it first writes it to its file and then sends it to the backup to store the write it to its file. Once it receives the ack back from the backup node, it sends replies success to the client. This whole process consists of 2 RTTs, 1 for the client to primary, and the second RTT for replication between primary and backup. We also call fsync after every write, to flush the contents of individual writes to the disk.
- Read protocol: The primary node replies to read requests from its file storage for read requests. This only takes 1 RTT as compared to the write requests.

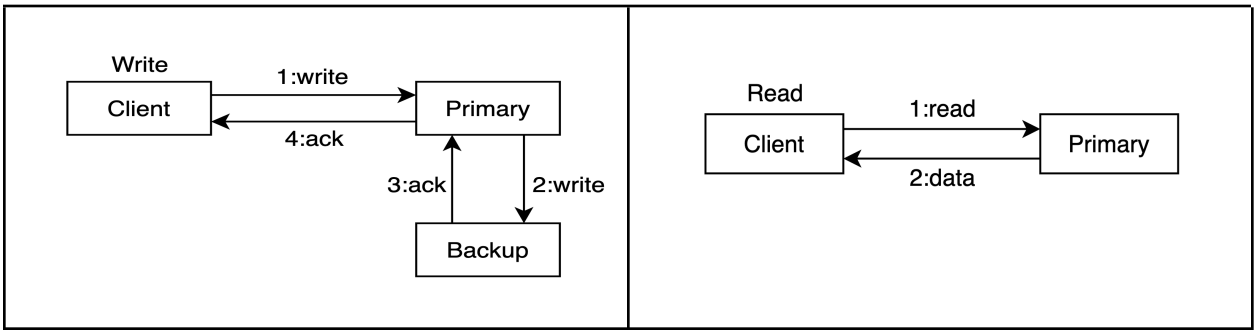

Figure 1: Read and write protocol.

2.2: Durability

The primary and backup nodes use a single large allocated file for persisting the data. On server startup, the nodes open the file in read/write mode and all threads share the same global file descriptor for file operations. This sharing of the file descriptor ensures ordering between different threads without using any synchronization mechanism.

2.3 Heartbeat and Crash Detection

The system uses a heartbeat mechanism for crash detection. Both threads send heartbeat requests to each other using a separate thread created at startup. The heartbeat thread sends a request every 100 ms, and a failure of three consecutive requests marks the other node in-active. We use a condition variable to wait for the failed node to come back alive. This helps us avoid wasting resources while the other node is crashed. When the failed node is restarted, its heartbeat requests signal the condition variable to start back the heartbeat thread of the non crashed server. The heartbeat response also contains information if the node is currently in primary mode. We use it to check whether a node needs to start in backup or primary mode.

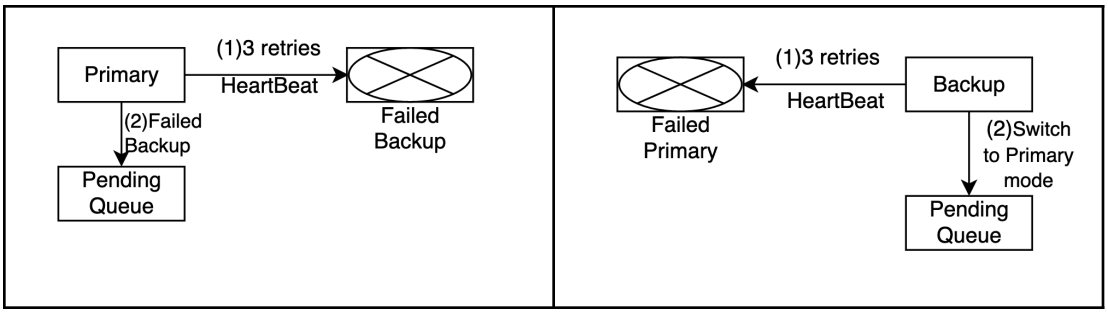

Figure 2: Failure at primary is transparent, while backup changes its mode to primary.

Once a crash is confirmed, we start operating in single server mode and there is a small difference in protocol depending on the failed node. The primary node continues to be in its current mode while the backup changes itself to primary. Both nodes start storing new write requests in a pending queue, which is used to replay writes when the failed nodes re-join the system. Figure 2 shows this subtle difference in the single server mode.
For clients, primary node failure causes its writes and reads to fail, which also connects to backup after three consecutive failures, meanwhile, the failure of a backup node is transparent to the client.

2.4: Crash Recovery Protocol

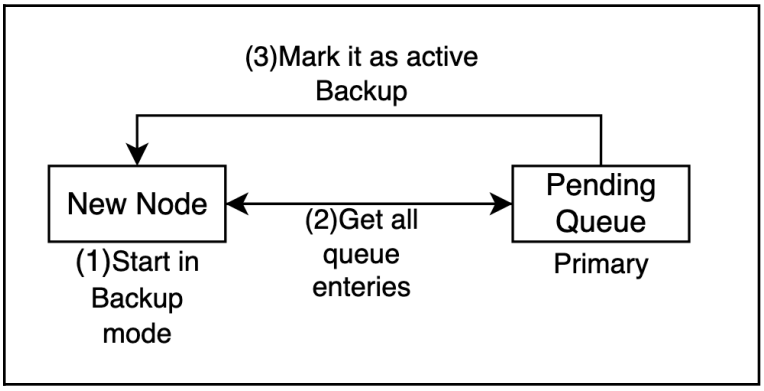

Figure 3: Failed nodes start back as a backup node, and get all pending new writes from the current primary node. The primary marks it active once the queue is finished.

The consecutive three failures of heartbeat requests mark the crash of the other node. To ensure the safety of the system, we first check if the current running node is in primary mode. This helps us eliminate corner cases, where the primary server is restarted immediately after a crash and the backup server has yet not transitioned to primary mode. After starting as backup node, the server replays all writes that were completed while it was in the failure state. Once the pending queue is empty, the primary marks the backup as active and starts sending it new write requests. Figure 3 shows the complete state diagram transitions for the crash recovery protocol. There is no communication with clients in the recovery phase.

Section 3: Testing & Measurements
We measured the system for both correctness and performance.

3.1: Correctness

3.1.1: Availability
The system remains available during failure at any one of the primary or backup nodes. We

tested the availability of our system by inducing crashes at several points of write requests. The complete lists of crashes are mentioned in Table 1.

```
enum CRASH_POINTS
{
    NO_CRASH = 0,
    PRIMARY_AFTER_WRITE_REQ_RECV = 1,
    PRIMARY_AFTER_WRITE = 2,
    BACKUP_AFTER_WRITE_REQ_RECV = 3,
    BACKUP_AFTER_WRITE = 4,
    PRIMARY_AFTER_ACK_FROM_BACKUP = 5,
    PRIMARY_AFTER_ACK_TO_CLIENT = 6,  };
```

Table 1: Complete list of all crash points used to evaluate availability of the system.

We used a small mix of read and write requests workload. Reads are done before and after crashes, while writes are done while the system has a failed node to check the correctness of the system. Client auto-switches to the other backup node and blocks until the write/read request has succeeded. Our test randomly crashes the client among any of the crash points mentioned in Table 1 and checks whether the client's request gets completed. We report total correctness in these scenarios.

3.1.2: Strong Consistency

3.1.2.A: Consistency check for overlapping requests:

Our system supports writing at arbitrary addresses and not just 4KB aligned addresses. We tested it by running a sequential address workload of write and read requests. We keep the expected read in-memory and compare it with the read replied from the system by calculating hashes of both memory chunks. We report total correctness in these scenarios.

3.1.2.B: Overall Consistency check

Correctness tests are an important aspect of strongly consistent systems. We generate a mixed read and write sequence of requests with both overlapping and non-overlapping addresses to create a deterministic workload. To measure the correctness of our system we run the generated workload on a local file as well as our system. During the test, we manually introduced crashes randomly at both primary and backup nodes. After each read request, we compare the checksum of the affected portion of the local file, primary and backup. Initially. We got some checksum errors, this helped in debugging our system. In our final version of the system, the checksum of every write matches for read requests performed on the same address.

We also experimented with turning on and off fsync while running the correctness experiment. Although our system supports switching fsync per write request, our correctness experiment runs either with fsync on or fsync off.

3.1.3: Testing Strategy for Concurrent Writes

We formalized a concurrency test to measure our system against concurrent requests where the dependency of all write requests are not guaranteed. This is inspired by Tao paper covered in class. However, we did not get enough time to run this test, hence we cannot comment on the outcome of it. We intend this section to be entirely a theoretical discussion of testing the strong consistency of our proposed distributed system.

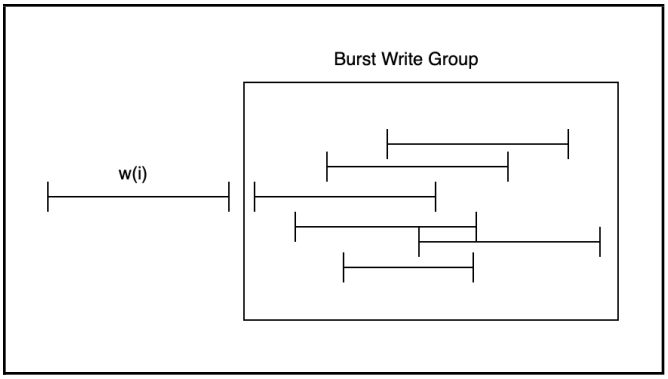

Figure 4: A sample write requests set showing bursty traffic.

The test is initialized with a large number(say 100) of concurrent clients, and they do bursty write requests. All writes are restricted in an 8KB block to create overlapping writes while being concurrent to each other. The clients logs a 4 value tuple value for each write request:

- Start time
- End time
- Operation address
- 4KB Data

This will generate a set of overlap requests, which can be ordered by start time. Figure 4, shows a sample of such requests. We intend to do two read requests, one before and one after each overlapped write execution. Since our system is not doing any heavy computation or I/O operations, the number of concurrent writes will be less. Given this assumption, we can generate all possible sequences after all *w(i*) in a group and read requests after bursty traffic should match one of them. If there is any interleaving of write requests, read after the bursty traffic will NOT match the generated possibilities, failing the strong consistency test.

3.2: Performance

We performed several experiments outlined below to test correctness, scaling capabilities

3.2.1A: Comparison with Single Server (no replication)

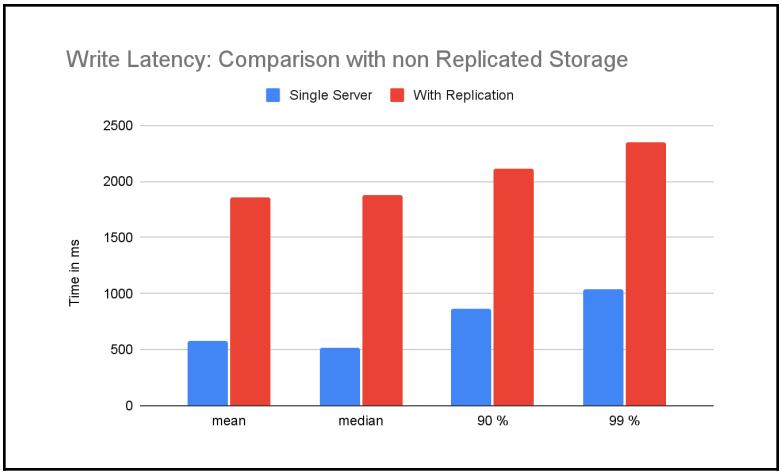

Figure 5: Write Latency with and without Replication

This experiment demonstrates the impact of replication to backup i.e, sending data to a single remote server(no replication) vs our system. We can observe from the graph that in the case of replication, the latency is more than double that of a single server setup. We attribute this due to one extra RTT and one extra fsync call at the backup node.

3.2.1B: Latency of Operations with Concurrent Clients

Following grid shows various combinations of client and ops for RR(Read Random), RS(Read Sequential), WR (Write Random) and WS(Write Sequential) workloads. We report mean, median, 90 percentile and 99 percentile values for all of the workloads.

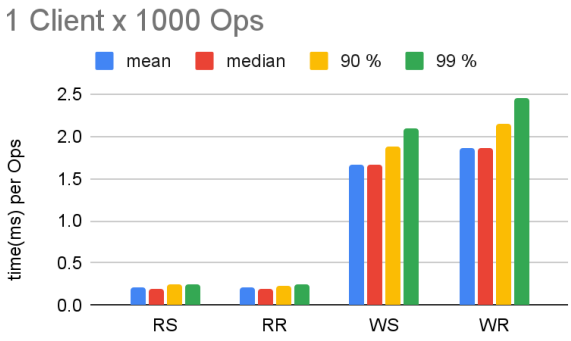

Figure 6a: Graphs showing statistics of 1 client doing 1000 ops with time in milliseconds (ms)

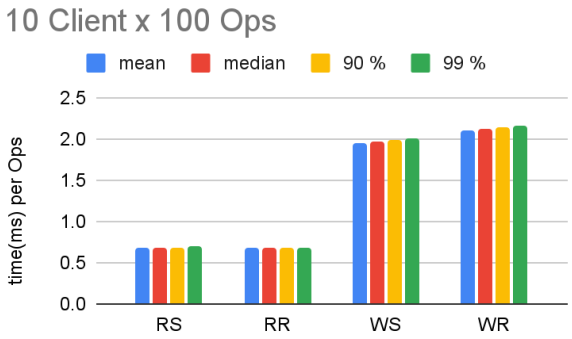

Figure 6b: Graphs showing statistics of 10 client doing 100 ops with time in milliseconds (ms)

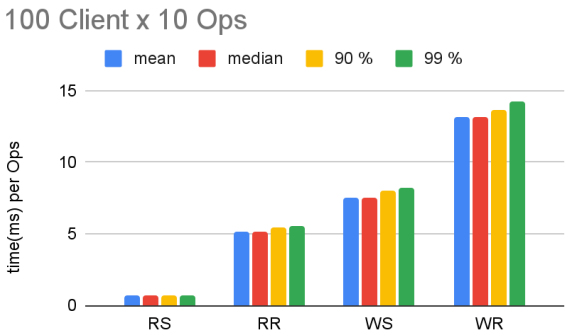

Figure 6c: Graphs showing statistics of 100 client doing 10 ops with time in milliseconds (ms)

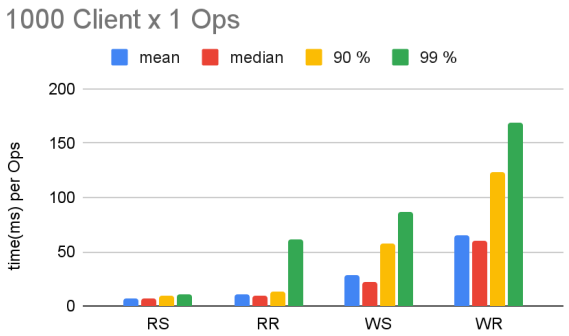

Figure 6d: Graph showing statistics of 1000 client doing 1 ops with time in milliseconds (ms)

In order to measure the performance of our replicated block storage (FireStore) we increased the number of concurrent clients from 1 to 1000 while simultaneously decreasing the number of ops from 1000 to 1. This experiment attempts to show our performance with a variable number of clients and concurrent ops. We observe:

1. With 10 clients, high latency differences exist between read ops and write ops. This can be explained by 2 RTTs involved in replicating data over primary and backup servers. Moreover, since we have one FD and only one client is being served, both ops - reads and writes from one FD are really quick.
2. With 10 clients, RR and RS have similar performance, possibly because single FD is being moved around enough among concurrent clients to mitigate performance gains of sequential workload. This argument also holds for writers.
3. With 100 clients, we see near linear increase from read sequential to read random workloads. This shows that our single FD approach is beginning to become bottleneck with an increasing number of clients as FD needs to be seeked to a particular address before write operation.
4. With 1000 clients, we see extremely high latency for 99%, which again points towards some clients significantly lagging behind in ops for random workloads. This enforces inference in #2(single FD is a bottleneck). Secondly, for WS(write sequential) workload median is less than mean and that shows increasing tail latency.

3.2.2: Performance difference between 4K-aligned-address vs. unaligned-address requests

We use a single large file for persisting the data, therefore we did not expect to find any performance difference between aligned-address vs unaligned-address requests. We validated our hypothesis using an experiment outlined in figure 7. We compared aligned and non-aligned writes for 1 client doing 1000 ops.

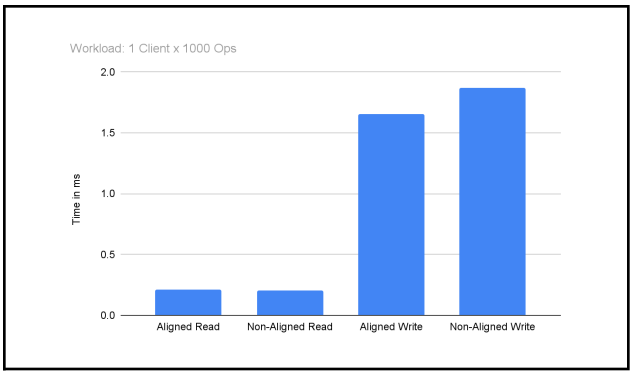

Figure 7: Read/Write latency (in ms) for aligned-address and unaligned-address requests

3.2.3: Recovery Time and Timeline

3.2.3A: Timeline for primary and backup crash

In figure 8, we show a timeline showing both primary and backup crashes. We ran a small workload compromising just writes(1000) when one node is down. Thus, only one is available and acting as the primary. This resulted in the accumulation of pending writes in a queue that is to be sent whenever backup comes alive. As shown below, as soon as backup comes alive and pings primary, it goes into the catch-up phase. After catch-up, the backup node becomes active to receive writes. In the case of primary failure, the backup node goes into the switch phase after three failed heartbeats(100 ms each) and also assumes the role of primary after these 3 heartbeats. Additionally, we show that the failed node starts back as backup only, essential for the correctness of our system.

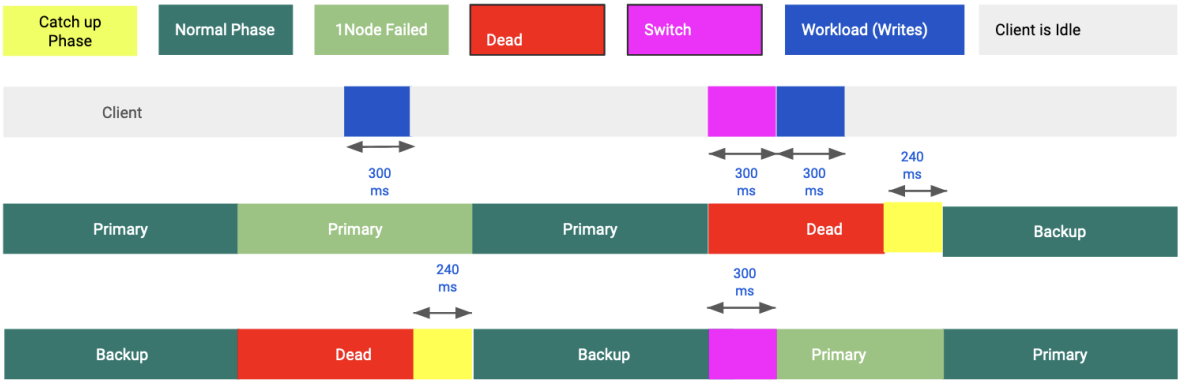

Figure 8: A small workload showing recovery time for both primary and backup crashes.

3.2.3B: Performance of Recovery Mechanism

We measured the performance of our crash recovery mechanism. The failed node on re-joining the system retrieves all the pending writes from the primary node and replays them to persist in its own perstance. In this experiment, we increased the number of pending writes by 10x, and plotted the total time taken to recover on a logarithmic scale. The result is in accordance with our expectation, the total time scales linearly when the number of writes is increased exponentially.

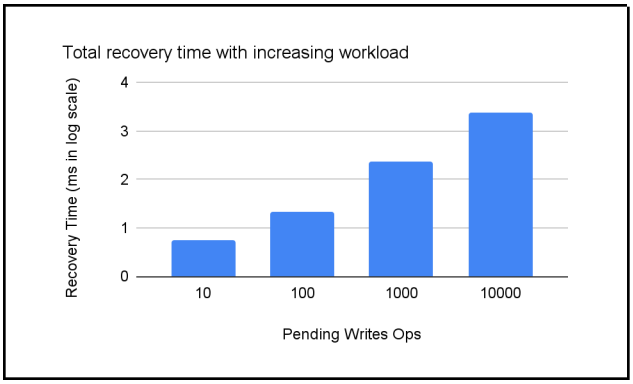

Figure 9: Graphs showing total Recovery time taken by the crashed server in catch-up phase. The x-axis shows increased pending writes, y-axis plots the total time to recover on a logarithmic scale.

3.3 Source Code

- Main code: Less LoC (simplicity).
- System Code is about 500 lines of code (including proto files)
- Main Server and Client code is about 200 lines.
    - Server Code is inside server.cc : 179
    - Interactive Client is inside client.cc : 17
- Testing code is about
    - 50 lines of python code
    - 300 lines of c code
        - Correctness Testing is inside correct.cc : 17 which uses localFile.h : 21(for local file comparison), crash_points.h : 16 (for Crash related locations) and client_rpc.cpp : 97 for enable and disable fsync per write RPC sent.
        - Measurement code is inside measure_client.cc that supports random and sequential works across variable # of threads.

References:

1. FoundationDB: A Distributed Unbundled Transactional Key Value Store
2. Replicated Data Consistency Explained Through Baseball
3. Existential Consistency: Measuring and Understanding Consistency at Facebook

Conclusion:

We developed a strongly consistent, available system that is partition in-tolerant. We observed performance degradations caused by choices made for a strongly consistent system whose performance of aligned and unaligned ops is similar. We learned to prove correctness by generating deterministic workloads and also showed the resilience of the system under random crashes.