CS 739 Project 3 Report

Heather Jia Cong Ding Song Bian Yifei Yang

Contents

1 Overview

In this project, we implement an only-2-node primary-backup replicated block storage system. We design and implement a mechanism to keep the data synchronized between primary and backup node. We also use this mechanism to deal with the case that only one of the two nodes will crash at a time. In the experimental section, we test the strong consistency of the system by using checksum technique. In addition, we run several workloads to test the performance of system and present our takeaways and observations in this report.

2 Design

2.1 Key Principles

Our key design principles are as follows.

1. If both the primary node and the backup node work well, then we will read data from the primary node directly. As for the writing operation, we will write data to the primary node and then the data will be sent to the backup node to make the data synchronized. The client will only get acknowledgement from the primary if the primary knows that the data is synchronized on the backup node.

2. If the primary node works well and the backup node crashes, then the data will be stored only on primary. When the backup is reconnected, the primary will send all outdated files to the backup for synchronization.

3. If the primary node crashes and the backup node works well, then the backup node will be promoted to the primary node. Next, we perform reads and writes on the new primary (previously backup) node. When the other node rejoins, the new primary will treat the new node as backup and do the synchronization.

2.2 Reliability and Crash Recovery

We consider both primary crash and backup crash in our system.

2.2.1 Backup Crash

When the backup is crashed, the primary has no knowledge of it and does nothing special to handle it. Instead, it will figure this out during the next write comes, when it tries to synchronize the write to the backup. At this moment, the primary will start a thread trying to connect to the backup intermittently, with a user-defined a retry gap. At some point later when the backup is recovered, it will start sending heartbeats to the primary and a request to ask for synchronizing outdated blocks. Before synchronizing, the primary fetches the timestamps of blocks of the backup and compare with the ones of itself so that it can figure out outdated blocks.

2.2.2 Primary Crash

The backup sends heartbeat messages to the primary to ensure it's alive. When the primary is crashed, the backup will know from the unsuccessful heartbeat. It will actively promote itself to the new primary, and start a thread trying to connect to the crashed old primary intermittently. Once the crashed old primary is recovered, it becomes the new backup and the intermittent connect trial will succeed. Then the primary will actively synchronize outdated blocks to the backup, following the same procedure as above.

2.2.3 Transparency to Clients

On an operation, the client always issues two identical requests to both the primary and the backup, and the primary will processes the request regularly while the backup will discard it. When there is one server failed, the other one must be the primary, where it's either the old primary working well or the old backup promoted to the new primary. Thus the request to the primary can be processed regularly, and the client will just need to discard the failed request sent to the crashed server and apply the successful one.

3 Evaluation

We run several benchmarks and conduct measurements to evaluate our final implementation. Our experiments are conducted on CloudLab [1] machines with one primary server, one backup and a maximum of two clients.

3.1 Availability

In this section, we want to show that we only crash one node at a time and a crash is hidden from the user. You can refer to the availability demo availability demo (Link: https://drive.google.com/file/d/1vBmicaXQug9hu9h6eL9Hi0q2vb4HrR0t/view?usp=sharing) for details.

3.2 Strong Consistency

In this section, in order to show the strong consistency of the system, we run the following experiments: one process write the data to multiple blocks in the storage, then two clients read the corresponding files concurrently. You can refer to the consistency demo (Link: https://drive.google.com/file/d/1-SFXcMjzaLt7nE0uGXqwLPLHw6brnNzA/view?usp=sharing) for details.

3.3 Read/Write Latency

3.3.1 Running time of reading operation

We perform the reading operations at random locations multiple times. The experiment (Figure 1) is run under the multi-server (one primary node and one backup) and single-server scenarios with one or two clients reading concurrently. We vary the number of reads to obtain the lines. And the latency is recorded as the vertical axis in terms of seconds.

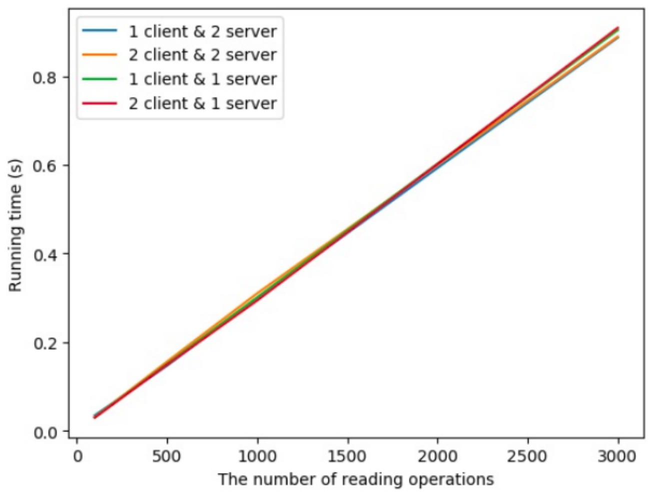

Figure 1: Read latency

3.3.2 Running time of writing operation

We perform the writing operations at random locations multiple times. The experiment (Figure 2) is run under the multi-server (one primary node and one backup) and single-server scenarios with one or two clients writing concurrently. We vary the number of writes to obtain the lines. And the latency is recorded as the vertical axis in terms of seconds. Writing to a single server is faster than writing to both the primary and backup due to the write sync-up on backup. Writing by only one client is faster than two clients writing concurrently due to the lock contention.

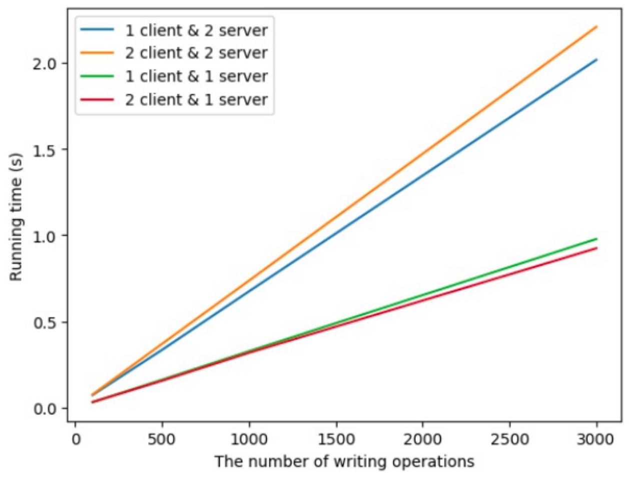

Figure 2: Write latency

3.4 Latency of read and write if a crash happens

We perform the reading and writing operations at random locations 10000 times. We repeatedly perform one write followed by one read. The experiment (Figure 3) is run under the multi-server (right-hand-side graph) and single-server (left-hand-side graph) scenarios with two clients running this experiment concurrently. The blue bar is a complete run without any server crashes. The red bar indicates that there is a server crash happen in the middle of the experiment. The latency of the entire experiment is recorded as the vertical axis in terms of seconds. After a backup crash, the rest of the writes do not need to sync with the backup and therefore speeds up the operations.

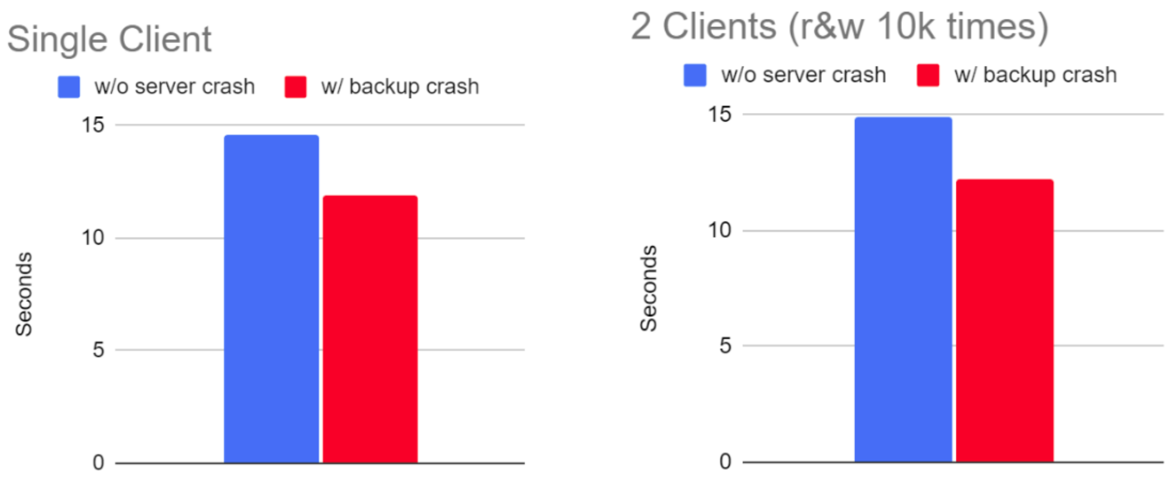

Figure 3: Read and write with crash

3.5 Recovery time

We measure the latency of different phases during a crash, including failover, synchronization, and reintegration. The failover time is the period between the old primary fails and the old backup is promoted to the new primary. Due to the fact that the clocks of the two servers are hard to synchronize, we measure this in a different way. We capture the crash signal of the old primary (e.g. SIGINT for CTRL_C), and before the program exits, the old primary sends a message denoting it's going to die to the old backup, and we just regard the start time of failover as the time when the old backup receives this message. The synchronization time is the time spent on that specific RPC call, and the reintegration time is the period between the crashed server starts and it completes all setups, e.g. set up connection to the primary.

We vary the number of outdated blocks and measure these three kinds of latency. The results are shown in Figure 4. First, we observe that the failover time is the major cost during recovery, which is mainly spent on setting a thread up for intermittently trying to connect to the crashed server. Theoretically this latency should remain stable. Our result shows relative fluctuation probably because we only measured for once, which can be improved by measuring multiple times and computing the average. The synchronization time is as expected, which grows linearly with the number of outdated blocks. The reintegration time is stable because the major part of latency is to establish the connection to the primary.

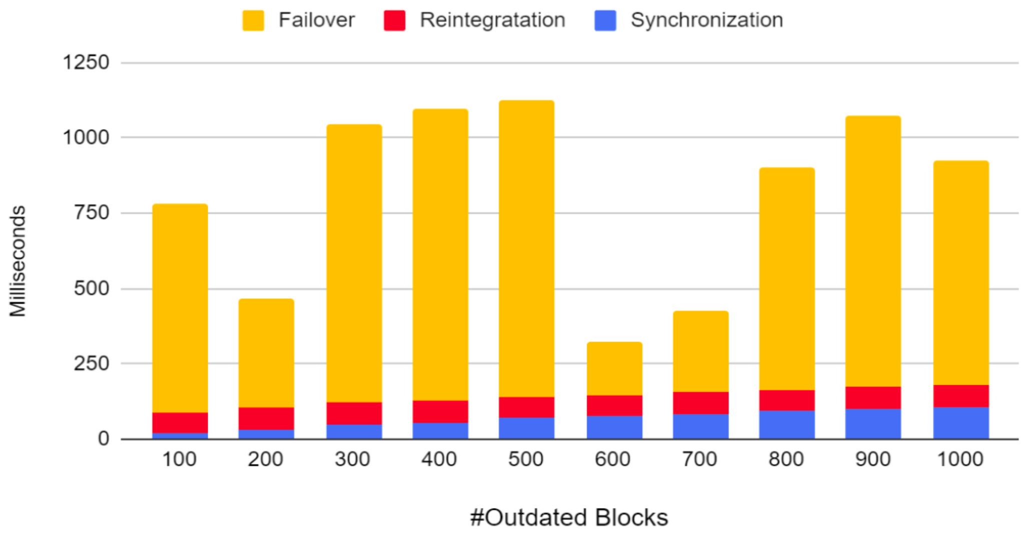

Figure 4: Recovery time

3.6 Aligned and unaligned address

We perform the writing operations at 4K-aligned addresses and unaligned addresses each for 3000 times. The unaligned addresses are always the middle of a block. This experiment (Figure 5) runs under the single server and primary-back scenarios with one or two clients. The aligned-address experiments take less time due to the interference with fewer blocks.

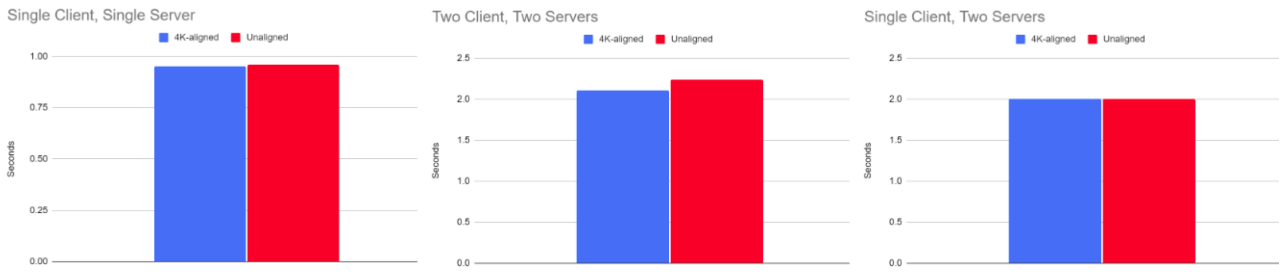

Figure 5: Aligned and unaligned address

4 Conclusion

We design and implement a 2-node primary-backup block store that support strong consistency, and availability in case of one node crashes. We verify the correctness on one node crashing and evaluate the system performance both with and without crashing and recovery time as well.

References

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