CS 739 Project 4

Avinash Kumar Ishaan Kohli Sandeep Singh Vyom Tyal

1 Introduction

In this project report, we aim to describe how we implemented and evaluated a Sharded Primary Backup Block Store (**SPBBS**). SPBBS has a simple design. It consists of multiple shards, where each shard has a primary and a backup server, where all requests go to primary, until there is a failure and data is strongly replicated to backup. We shard the data based upon the modulo of it's block number(derived by address) with the total shards. In this report, we first shed light on the design of the system and then perform experiments, explaining our evaluations. Before we start, it is important for the reader to understand the main additions from the previous project, so that the report does not feel lengthy. In very brief, we list our extended contributions.

1. We have added a new master server which stores the state of all the shards that are part of the system.
2. We have modified the client to dynamically get the shard list from the master. The client, then decides:
   (a) The shard to which the block request should go to.
   (b) If the request is unaligned, route it to multiple shards.
3. We have added the support of addition of a single shard to the system. This includes reshuffling of data across the shards. Requests are not accepted while reshuffling takes place.
4. We have performed some specific evaluation to measure the degradation of performance of writes when the primary and backup of shard are distributed over WAN, one of our goals.
5. In addition, we measure the performance of shuffling of data and different workloads on the shard system. These are not on WAN!

2 Design

The design is similar to a traditional client-server/request-response distributed system, with addition of a master. We provide a client library, which can be imported by a client requiring to store blocks in SPBBS. The client talks to the master to get a list of active shards, and then dynamically creates separate clients for each shard pair. The blocks can be either read or written by the client according to it's choice. A user is only exposed to read and write, but a server can perform more operations on the other servers. We have two gRPC servers, one for the master and one for the actual store. We chose to use a primary backup pair/pod for a single shard. This was meant to have a replication factor of two for each block. The primary/backup pair are proposed to be kept in separate data centers over the wide network to ensure strong fault tolerance. Figure 1 shows the architecture.

The master implements the following RPCs:-

1. **GetState**: It is used to get the state of the Master Server.
2. **ActiveConnections**: It is used to return a list of active shards which have successfully contacted the master using the *AckMaster* RPC
3. **AckMaster**: It is a message sent by a storage server of a given shard to tell the master about its active presence in the system. This call triggers a re balance of data across the shards if a new shard is detected.

The storage server implements the following RPCS:-

1. **HeartBeat**: It is used to check if a storage server is up or not. The response includes valid status, the name of the mirror(primary or backup) and the states of the replicator.
2. **Read**: It is used to read a block from SPBBS from a given address which may be 4KB aligned or unaligned.
3. **Write**: It is used to write a block to SPBBS from a given address which may be 4KB aligned or unaligned. This write covers writes to both primary and backup servers.
4. **ReplicateBlock**: It is used to send a block received by one server to another server, so that it can be replicated. Success of this call does not ensure that

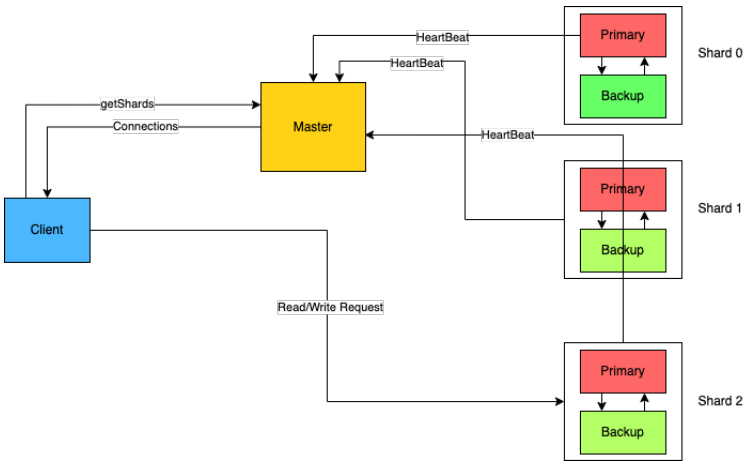

Figure 1: High Level Architecture

the other server has made this block durable, and the block is written temporarily only. A subsequent CommitBlock makes it durable and visible to the client.

5. **CommitBlock**: It makes the temporary write done by ReplicateBlock on the other server durable, and visible to the client.

6. **CheckConsistency**: It is a helper RPC to check if both the clients and the server have the same data at a given point of time.

7. **TriggerShuffle**: It is a helper RPC to check if both the clients and the server have the same data at a given point of time.

We will now first explain the master, then the client and finally briefly describe about all the components of the storage server.

2.1 Master

One of the main motivation of having a master in our design was because of our aim to add shards dynamically to our system. The master only stores information of shards as primary-backup pairs that sends it an **AckMaster** message. For storing the information, we simply use hashmap, and then persist in a file on the server. The idea behind persisting is that we need to know the shard id of the server. The shard id is just an integer assigned to a primary backup pair. Since we use a modulo based scheme to send requests to the correct the shard, we need the shard id. We use a counter to assign the shard id: if a shard, which is not already on the master pings it, we simply assign the counter value and increment it. Persistence makes sure that if the master process goes down, we can simply load the old shard ids from the file when it comes back up and the data still goes on the correct shard. Apart from this, the master also uses a boolean flag to indicate whether the shards can receive request or not. This can be used by the client to decide whether the shards are ready to accept the requests or not. An example case is when we stall requests during reshuffling. Finally, on addition of a new shard, the master triggers a reshuffle to all the existing servers, and turn the flag to false during this period until the reshuffling is completed. The reshuffling protocol is described more in the below sections.

2.2 Client

The client accepts the address of the master server. It pings the master to get the shard list and the master server state. Depending upon the address, the client computes the modulo and then redirects the request to the correct shard id. The client data structure to maintain is similar to the master. If we had 3 shards, and a request comes to the block 4096, we first compute the block number by dividing by the block size(4KB), which is 1 in this case. 1 modulo 3 gives us 1, which means that the block 4096 would go to shard 1. Similarly, 8096 address will go to shard 2 and so on.

2.2.1 Unaligned Operations

Unaligned operations in our system now span multiple shards. Previously, with just a single shard, such operations spanned two blocks or two files on the same server. Since consecutive blocks now can't be on the same server, we have to make two requests for all reads/writes. The client determines this, and if the request is unaligned, issues two asynchronous request to the relevant shards. This was a decision we made to improve latency

of such requests, as now we just make 1 round trip and not 2 round trips. The latency is then dependent on the slower server

2.3 Storage Server

Since our project is an extension of HABS, which was the previous course project, we will only mention details about the server for basic understanding not deep dive into why the decisions were made for previous features. We will however explain any new additions.

The server process has multiple components for managing the life cycle of incoming requests. Briefly, we have a gRPC server, which handles incoming requests from clients. Depending upon the request, it uses the Block Manager to read or write the blocks. The Block Manager is an abstraction of how the data is stored on disk, and knows how to read, write and commit a block to disk. The replicator is used to manage the replication of blocks to the other server. It has a gRPC client is inside this component, which performs heartbeats to the other server. The replicator only comes into the picture when the other mirror is dead, the given write block is queued to the replicator for future processing. Figure 2 shows a block diagram showing the described components.

Since we have to communicate to the master as soon a shard is up, a shard sends a **AckMaster** message to the master so that master can include the new shard in it's connection list, everytime it boots up.

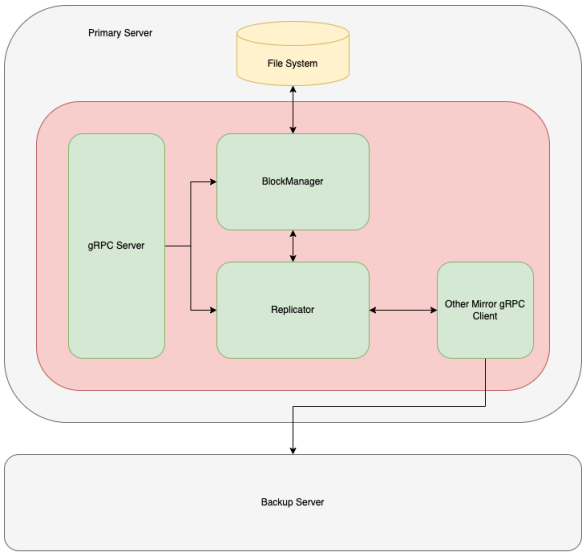

Figure 2: High Level Storage Server Design

Now we will describe each of the components of the system. Note, that we don't explicitly write about the read protocol, because it is simply just using the block manager to read at the given address, and would be explained in the block manager section.

2.3.1 Block Storage (Block Manager)

Our choice was to map each block to a file in the local file system. Each file represents a 4KB space in the complete block storage, and the start and end address of these blocks are in multiples of 4KB. We found this system be better than implementing storage depending on a single file. The block manager is local to one server, and both primary and backup server have their independent block managers, unaware of their existence. It is only a means to store blocks on the local file system. Finally, the block manager is decoupled with other components, and we can easily change the storage scheme if we have to, without changing higher level implementation.

Block Metadata: Since now any arbitrary has the possibility to reach all shards depending upon the number of shards, we can't just store the block as is. We need to maintain the number of blocks and which blocks are stored on the given shard. Hence, we persist the address numbers on a file and the total number of blocks.

Directory Structure: For this, we use a simple scheme based on the block address. A single level directory structure is created, where the block store root path has multiple directories and each directory have a fixed number of blocks in them. The directories structure are numbered from 0 to the maximum directories we want to support. We also call it the **BLOCK_DIVIDER_SIZE**. By taking the modulo of the block number with the **BLOCK_DIVIDER_SIZE**, we find the directory the block should get placed in and then save the block in it, named by it's block number. Figure 3 shows the directory structure diagrammatically. The grey boxes represents directories, and the blue boxes represents files.

Temporary Writes, Commits and Locks: As we now know where the blocks end up in the file system, we move ahead by explaining what interface the block manager provides to perform durable IO on a given address. Firstly, the vanilla write method performs a temporary write in the local file system. We just append a **-tmp** to the block store root path, and use the existing scheme to store blocks inside it. This is to make sure that the block has first successfully reached the local file system, before overwriting the previous block data. Reads don't read from the temp block and always go to the actual block path. Secondly, we provide

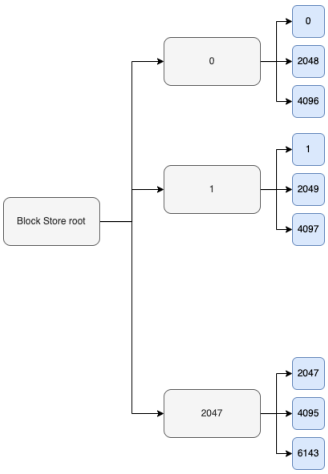

Figure 3: Block Directory Structure

a commit method, which ensures that the last written block becomes durable in the file system. This simply renames the temporary block path to the actual path. Post this operation, the read method will return the newly committed block. Lastly, we provide methods to lock and unlock operations on a particular block address. These are not internally used by the block manager, but are just provided as means to be used by components utilizing the block manager, in conjunction with the read and write operations on the block.

2.3.2 Replicator

The replicator was created to manage failures of one of the server in SPBBS. It consist of a queue and a hash map, which store addresses, which are not yet replicated to the other server. It provides a method to add an address to the queue, which can be called by higher level function, which may not invoked until there is a failure. It also has health state of the server, which is determined by the existence of the other failure, and build up of the queue size. When it detects that the other server is up, it replicates the block to the other server until it becomes empty. The addition and removal from the queue is made atomic using locks to avoid race conditions. We don't buffer write requests. State change and invocation of recovery takes place inside a background thread. Next we explain the states of the replicator and how recovery is achieved. **Replicator State**: Replicator state is maintained to know the health of a server. It can help indicate whether an alive server can accept read/write requests. They are described below.

1. ***UNHEALTHY***: This is the first state the replicator goes into before it can move to any state.

2. ***HEALTHY***: This state indicates an alive server is okay to serve requests.

3. ***SINGLE_REPLICA_AHEAD***: This indicates that the other mirror is dead, and all blocks are updated, but have a single replica only.

4. ***REINTEGRATION_AHEAD***: This state indicates that the server is actively replicating the blocks in the replicator queue to the other server.

5. ***REINTEGRATION_BEHIND***: This state indicates that the server is up, but there are still some blocks which are not replicated from the other server, which is actively trying to replicate them to it.

Multi-Threaded Recovery: The queue is burned off by removing the top most address from it and the hashmap, reading from locally using the block manager and sending a **ReplicateBlock** call followed by a **CommitBlock** call to replicate for all the blocks to the queue until the queue becomes empty. If any address can't be replicated, we simply place it back in the queue and stop the reintegration, as the server might be dead again. For faster recovery times, we choose a multi-threaded version of queue processing. We spawn threads equal to the number of CPUs present in the machine, and each of the thread processes part of the queue. We call this the consumer function, and the producer is anyone who adds an address in the queue.

2.3.3 Write Protocol

One of the main focus for SPBBS was to continue offering strong write consistency. Since we followed a primary/backup approach [1], we needed to make sure that a block was written to the local file system of both the primary and backup before the client can know that the writ was successful. After reading the above sections, the reader might already infer that the replicator and block manager help in fulfilling one or more of the requirement(s). The write follows 2-Phase Commit semantics, utilizing the block manager and replicator. The protocol makes sure that the data is on file system of both servers(Prepare Phase), before we commit it to the actual block location(Commit Phase).

When the primary receives a write request, it starts the write phase by first sending a ReplicateBlock to the backup, and after the call succeeds, we call the local block manager to do a temporary write. In case the backup is dead, we skip the ReplicateBlock and simply send the address to the replicator to buffer them for recovery when backup comes back up. Apart from that, if the backup is alive and still rejects ReplicateBlock, we reject the whole write, because this could be because of different kind of issues such as disk failures, no storage space or file system faults, which we did not handle. If

the ReplicateBlock and the local write both succeed, we assume that the block has reached both server's file system/disk and start the commit phase. The primary sends a CommitBlock to the backup server and only if it also succeeds, we call commit on local block manager. After the commit phase ends, we send a success to the client. We evaluate different crash points in the subsequent sections.

Finally, we want to mention that since this protocol makes multiple local and remote calls, and each write request is executed in it's own thread, multiple writes to the same block are prone to race conditions leading to inconsistency between primary backup. This is only a problem in case of concurrent writes to the same address/block. Hence, we use address level locks to synchronize writes for each block address, using the lock interface exposed by the block manager.

2.4 Reshuffling of Blocks

The idea of reshuffling the blocks allows us to balance the load across shards. It is extremely important for us to understand the dependency of the number of blocks on shuffling duration. Based on our design of request routing and address distribution protocol, shuffling time is directly proportional to the load on the shards. In the upcoming subsections, we will try to understand how the shuffling action is triggered and who is responsible for it? We will further discuss about the work done by each shard and finally, we will cover some failure scenarios.

2.4.1 How and Who?

Whenever a new shard comes into the picture, it will request the master to add it to the list of active connections. This list is a persistent META Data kept on the master. Once the master receives the request from the new shard, it will add it to the list and increase the total shard count. After this, it will generate a TriggerShuffle request for all the active shards.

2.4.2 Work done by each shard

As discussed in the above section, a TriggerShuffle request for all the active shards is issued when a new shard is added. Once the primary server of the given shard receives the request, it will traverse the committed block list and start sending the write via the client library(Figure 4). Sending the write request via the client library will enable the request to be routed to the correct shard. This commit protocol of this write request remains similar to the regular write request.

The shuffling of blocks requires us to read the given logical block address and write a different logical block address. Since the values of these addresses are not unique, we have to put some mechanism to avoid data corruption during the process. Our block manager provides this mechanism. The block manager maintains a variable called curr_index, which gets toggled on every TriggerShuffle request. This enables us to read and write the logical address simultaneously without the fear of data corruption.

2.4.3 Requirements

We will now discuss some of the requirements of our system to execute the shuffling protocol. Firstly, once a shuffling is triggered, we expect that either the primary doesn't fail or the primary failure is transient. If there is a failure of the primary, our system will be at a halt, and no further progress will occur. Secondly, our system handles master failure only in cases when either the TriggerShuffle request is not sent to any shards or sent to all the shards. We do not handle partial scenarios. Finally, read and write requests are stalled during the shuffling protocol.

3 Evaluation

This section will cover the experiments we have conducted to evaluate consistency, performance, and recovery protocol. Most of the performance/consistency metrics that we found from the experiments still hold true in our system, only exception being in case we deploy our setup across WAN. In addition, we evaluate our shuffling performance.

3.1 Test Framework

Since testing and evaluating a distributed system is always a challenge, we built this framework to make our experiments easy to implement. There are two major simulation components. The first is the write traffic simulator, and the second is the fault injection simulator.
We have provided two APIs that simulated write traffic on the same address and different addresses in the write traffic simulation. These APIs are well supported for concurrent writes. For Fault injection simulation, we have an API that injects crash on server sided under different states of server in a deterministic manner.

3.2 Experimental Setup

We have set up our primary and backup servers on the clouldlab machine and then spawned clients from different machines using python, bash scripts and c++ programs. Below is the configuration of our machines:

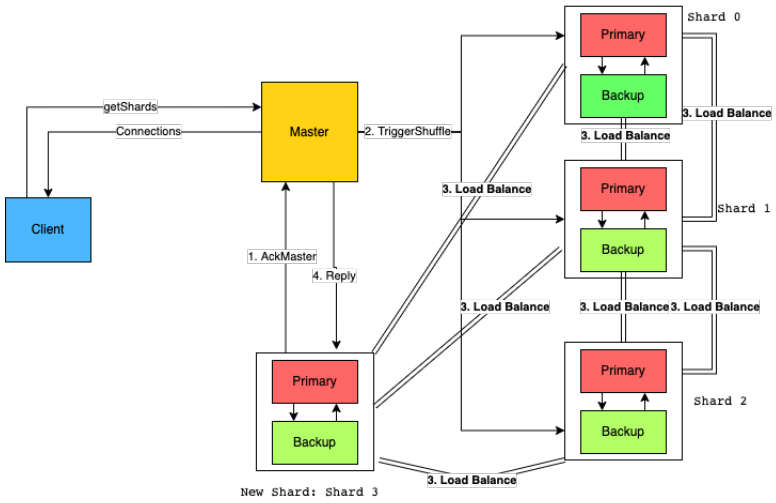

Figure 4: Shuffling Protocol

CPU Name : AMD EPYC 7302P @ 3GHz
CPUs : 16
OS : Ubuntu **Memory** : 128 GB

For evaluating normal cases, we use 9 nodes in the same cluster(Wisc) and same link to simulate up to 4 shards plus 1 master. For the WAN experiments, we use 3 extra nodes for the backup server, in a different cluster(Utah).

The client library takes care of the routing of requests to correct shard. Each block is random data and verified using a checksum during the test. Times are measured in client libraries using the Chronos library.

3.3 Consistency

Our consistency measurements were about the same as in our last project. The only difference was in case of unaligned operations. Since the operation is orchestrated on the client side, it is possible for concurrent writes on the nearby blocks to overwrite or get overwritten. We summarize the consistency below.

Test Scenarios	Consistency Status
Diff addr (Aligned)	Strong
Diff addr (Unaligned)	Weak
Same addr (Aligned)	Strong
Same addr (Unaligned)	Weak
Single shard crash	Strong

To evaluate consistency, we are using SHA-256[2] to compute the hash for a given block address or set of block addresses. We believe in non-failure case consistency validation of concurrent writes from multiple clients on the same address is one of the primary indicators of strong consistency.

3.4 Performance

3.4.1 Sequential Reads/Write Performance

In this experiment , while running the experiment we created a single client which sent concurrent write/read requests to our system wherein we varied the number of active shards in our system amongst 2, 3 and 4. Our client kept on sending the requests and we calculated the latency of each read/write done on all the shards (along with the total latency for the full batch of reads/writes) . From these latency's, we found the write bandwidth to be around 1.61 MB /sec with a single shard which is similar to our p3 values but we can see the actual jump in the bandwidth when we run more shards in which we see a bandwidth increasing by a factor of around 20-25 percent for the writers because of the increase in the servers on which the writes are distributed. Similarly for the read case we see our bandwidth increasing by a factor of around 30 percent and the reason is same as mentioned above. Figure 6 shows the result.

3.4.2 Same Address vs Different Address

In this performance experiment our single client sends the 10000 read / write requests in a similar fashion as mentioned in the previous case. The only difference is that the system is set to run with only 2 active shards but we now wanted to see the effect of varying addresses for these reads/writes on the system. Therefore we sent 10000 writes and reads first on evenly balanced sequential addresses and then on a single address in one of the

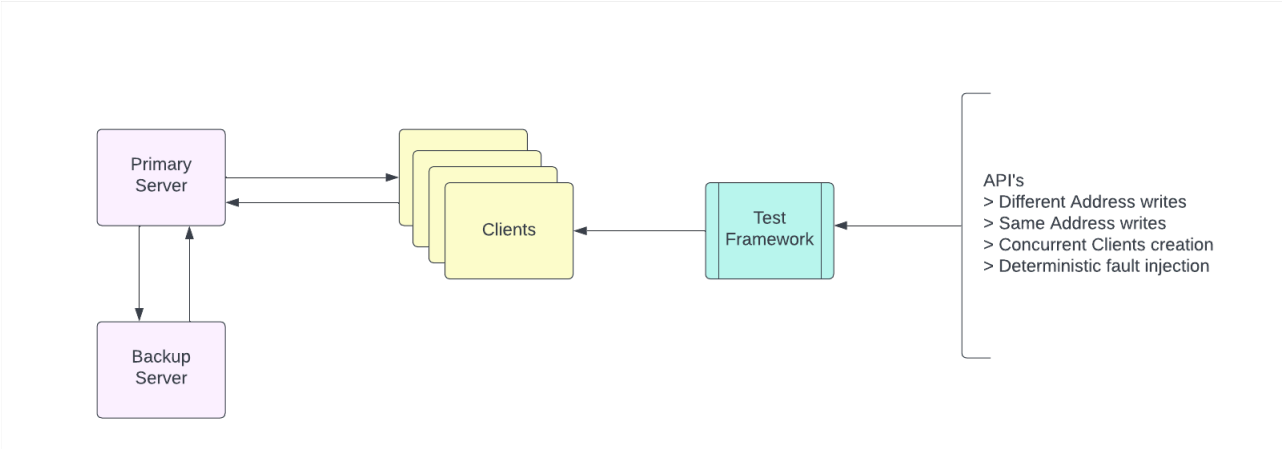

Figure 5: Test Framework

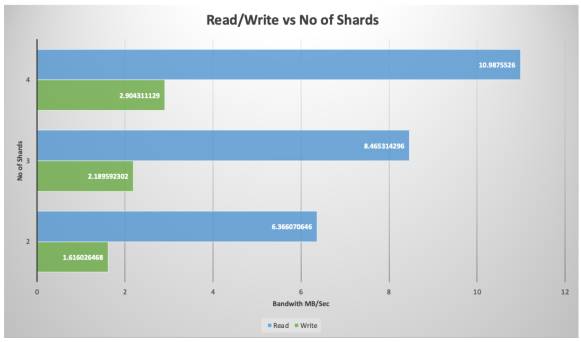

Figure 6: Sequential Reads/Write Performance

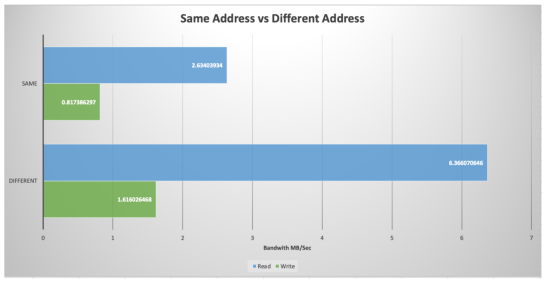

Figure 7: Same Address vs Different Address

shards. Our intuition was that a single address write and read will create a hotspot and we wanted to decipher the amount of decrease in the bandwidths for this case. Same address writes can't be paralleled but our system as they will only got to a single (or a single pair) of shards and therefore they have to be served sequentially. As expected we saw a decrease of around half in the write bandwidth and nearly 66 percent for the read bandwidth.

3.4.3 Concurrent requests

In this case, we now finally vary the number of clients which in turn varies the number of parallel read/write requests to our system and we wanted to infer if there is any difference in the variations of the write latencies for the 2 and 3 active shard case. All the clients are always sending the same number of requests individually. Because of the network tunnel being flooded with the write requests by the multiple clients, we see a near Linear increase in the write latencies as we increase the number of parallel clients. We were not expecting much of a difference in the latencies (averaged out) for the 2 and 3 shard case, which is also seen in the graph although the time latencies for 2 shard case is always slightly higher than the 3 shard case. This shows that for more number of clients, the number of active shards won't have much affect on the write latencies once the full network bandwidth is passed.

3.4.4 Unaligned vs Aligned Operation comparison

In this experiment, we tested the impact of performance in case of unaligned request, where we need to send two requests. We overall noted a increase in latency in this case for both reads and writes. Since the two requests happen in a single round trip, the latency did not increase by a huge margin.

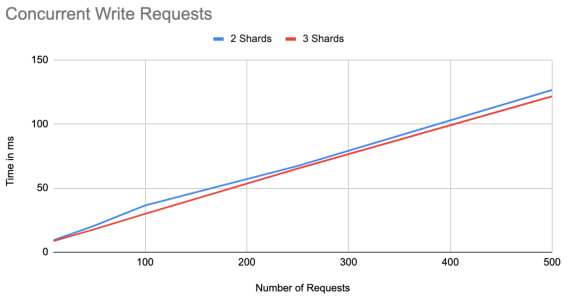

Figure 8: Concurrent requests

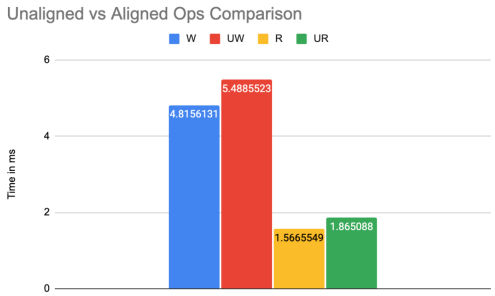

Figure 9: Unaligned vs Aligned Operation comparison

3.4.5 Comparison of Latency with WAN vs Same cluster

In this experiment, we compared the write latency when a shard is spread across WAN and when it is in same network link. We noted that the performance gets degraded by 20 times, which is because we ensure strong consistency and replicating it to the other server is expensive. We did not test reads because they only go to one server, and expected them to be the same as before

3.4.6 Recovery from failures in WAN

In this case, we simulated a queue on two shards, one spread across the wide area and the other spread across the same network link. We saw an increase in recovery time in case of the shard in WAN. This was expected because of the added latency. Apart from that, we observe that the latency is not as bad as the write latency, possibly because of parallel processing of queue.

4 Conclusion

During the course of this project, we extended our previous block store system by adding support of sharding

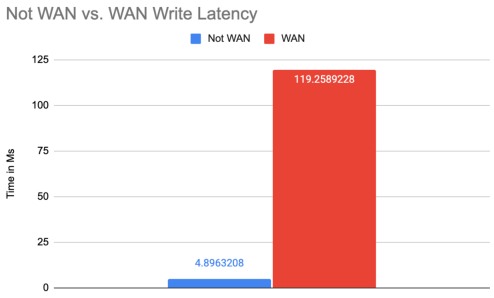

Figure 10: Comparison of Latency with WAN vs Same cluster

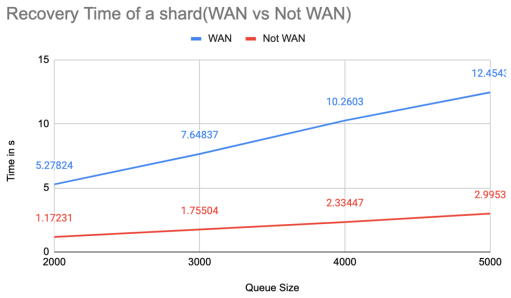

Figure 11: Recovery from failures in WAN

data in a primary backup server pair and then enabled dynamic scale up of shards in the system.

We learned a lot during this project, specially while designing the shuffling protocol. We also compared different designs and decided to go with the one we implemented based upon fruitful discussions.

Finally, we evaluated our system, and drew inferences from it.

References

[1] Anupam Bhide, Elmootazbellah N. Elnozahy, and Stephen P. Morgan. A highly available network file server. In *In Proceedings of the Winter USENIX Conference*, page pages, 1991.

[2] Olivier Gay. C++ sha256 function. URL: http://www.zedwood.com/article/cpp-sha256-function.