Replicated Blob Store

Ajay Joshi (ajsj7598@cs.wisc.edu) Partho Sarthi (sarthi@cs.wisc.edu)
Sandeep Singh (sandeeps@cs.wisc.edu) Rachit Tibdewal (rachit@cs.wisc.edu)

1 Design Rationales

For this assignment, we implemented a Replicated Blob Store system using Primary/Backup replication. We used a two phase protocol for consistency and log files for crash recovery. In this section, we describe the major design decisions that were taken.

1.1 Replication Strategy

Server Implementation: We used Primary/Backup model for replication (shown in Figure 1). Any server which boots up first tries to ping another server and becomes 'Primary' upon failure to the ping request sent to another server. Server which boots up and successfully pings another server becomes the 'Backup'.
Client Library: We developed a client library that connects to the primary machine. If the Primary is down, it retries the request to the Backup.Backup checks if Primary is dead and transitions its role to Primary. This is hidden from the Client Application.

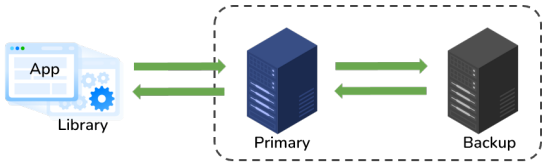

Figure 1: Primary/Backup Replication

1.2 Storage Architecture

We maintain a file per 4KB block of data. This implies that for every aligned write/read, we access 1 file, and each unaligned read/write spans over 2 files. We also maintain a single log file on a server with structure as shown in 3, which is used to append transaction records atomically.

1.3 API

(1) WRITE(ADDRESS, DATA): This included a two phase protocol (shown in Figure 4). In each phase, a local operation was done on the primary, followed by a remote operation. The first phase was PREPARE(). In this, a PREPARELOCAL() method reads the old into *tmp* files (two *tmp* files if the address is unaligned). This is followed by PREPAREREMOTE() which executes the same operation on Backup. Now, in the COMMIT() phase, a COMMITLOCAL() method writes logs entry and renames the tmp files to actual files. COMMITREMOTE() repeats the same on Backup.
(2) READ(ADDRESS): A read request is served by the primary only. Similar to WRITE(), it opens two files if the address is unaligned.

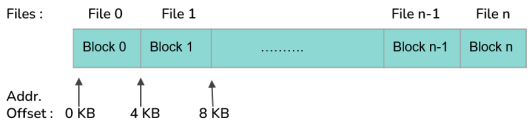

Figure 2: Storage Mechanism

Figure 3: Log structure

1.4 Crash Recovery

We infer that a server has crashed if it does not respond to a ping from the other server. Our protocol for crash recovery is shown in 5 and is as follows:
Protocol: The recovering node starts by default in the Backup state. It then pings the other server to check if it is up. If not it becomes the primary according to the crash detection strategy mentioned above and thus doesn't have to recover.

1. Ping other server. If no response received, assume Primary role and no recovery is required.
2. If the other node is alive, read the entire log file and send all log entries to the Primary through a Recovery RPC call.
3. Primary merges received backup logs with its own logs by identifying the common prefix in both sets of log entries.

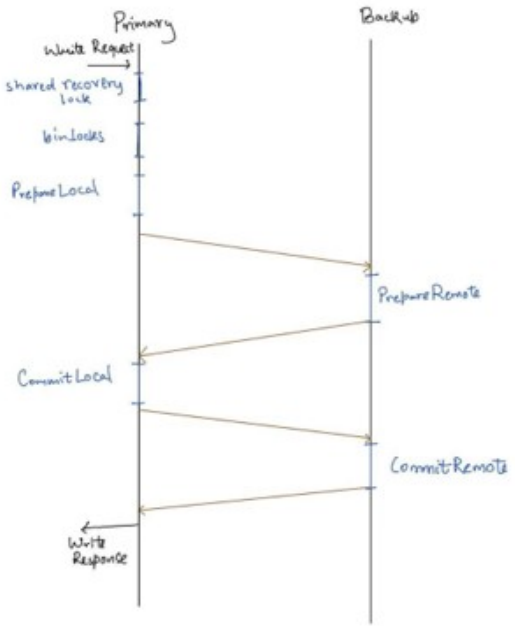

Figure 4: Prepare Commit Phase during write()

4. Primary sends its unmatched logs to the backup and deletes the prefix of matching log entries. It also sends file data corresponding to each log entry. eg: a Write(3) has a log entry (123456, 0, 1, 1) with 123456 being the transaction ID, and (0,1) being the blocks spanned by the write, and 1 being the commit status. Accompanying this log entry, the data for blocks 0 and 1 are sent by the Primary.

5. Backup replays each log entry using the entry as well the corresponding data with the below mentioned replay mechanism

Replay Mechanism:

1. Copy block data into temporary file(s).

2. Atomically append entry to the log file

3. Atomically rename temporary files to the actual destination file(s).

Recovery Isolation: Another important aspect that is required for a correct recovery is that the Primary should not process any requests during the Recovery process and no requests should be processed by the Backup till the recovery process is complete. This is required because if any request processed by the Primary may not be processed by the backup during log replay, causing states to diverge. States of primary and backup will also diverge if backup receives requests during recovery as logs due to request processing will affect log replay.

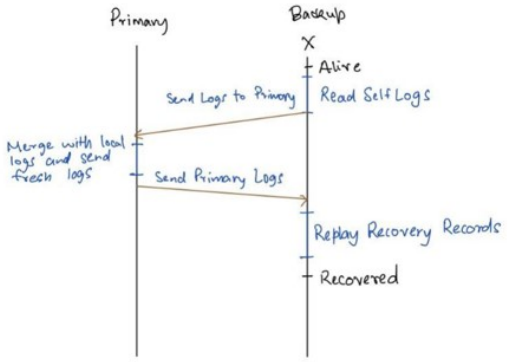

Figure 5: Crash Recovery

2 Correctness

For testing correctness with respect to availability and consistency, we introduced deterministic crashes in the address itself. We used special address that contained special crash instructions. For example, a negative address of *-30004300* denotes an address with crash instruction. The first digit denotes the crash code and remaining digits are the usual logical address. Table 1 shows the various crashing sites that were tested and their corresponding crash code.

2.1 Availability

To showcase that the system is always available (assuming only one machine crashes at a time), we maintain a client side library. Upon failure of a request, the client library retries to the backup server. Thus, the application layer experiences only delays without any knowledge of crashes.

Crash Type	Code
Backup Crash After Primary Prepare	1
Primary Crash After Local Prepare	2
Primary Crash After Local Commit	3
Backup Crash After Primary Commit	4
Primary Crash Before Read	5

Table 1: Various crashing sites and their corresponding codes

2.2 Strong Consistency

In order to test for consistency, we spawned multiple clients and created a workload of interleaving writes

and read of aligned and unaligned requests. We maintained a global hash table which is updated by each client upon successful acknowledgement of a write request. The hash table contained *key* as *address* and *value* was *hash_fn(data)*. Each address offset entry has the updated hash of data written by the last writer(client). Read request's response is verified by comparing the hash of received data and expected data (from hash table)

3 Evaluation

We performed the evaluation in CloudLab machines. We used servers running Intel(R) Xeon(R) CPU E5-2630 with 32 cores and 126 GB RAM. We tested our experiments with Primary and Backup running on different machines and the Client Application on a third machine.

3.1 Read/Write Latency

Experiments for read/write latency is shown in Figure 6. (a) During Backup crash, we observed the write latency to increase. However, once the Primary detected that Backup has crashed, the latency decreases significantly. This was expected as the RPC between Primary and Backup was no more occurring. We did not observe any significant differences in read latency. This was also expected since all the read requests were served by the Primary and this crashing should have no effect. (b) During Primary crash, we observed a large increase in write latency since all the requests had to be retried to the Backup. On receiving these requests, Backup confirms by sending a *ping()* to Primary. Once confirmed that Primary is dead, Backup will server these requests. Hence, due to this the latency increased. A similar pattern was observed during read requests.

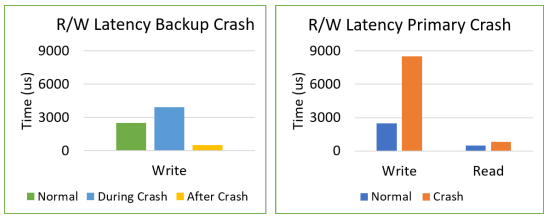

Figure 6: Read/Write Latency with crashes

3.2 Recovery Time

Here, the experimental setup is as follows: Primary and Backup both process 50k requests initially, after which the backup server process is killed. The primary then receives 10k and 50k requests from the client after which the backup process is restarted. The times for different stages in the backup recovery are shown in Figure 7. Time for each phase(Log Read, Log merge, Log Replay) increases as the number of log records replayed increases, which is the expected behavior.

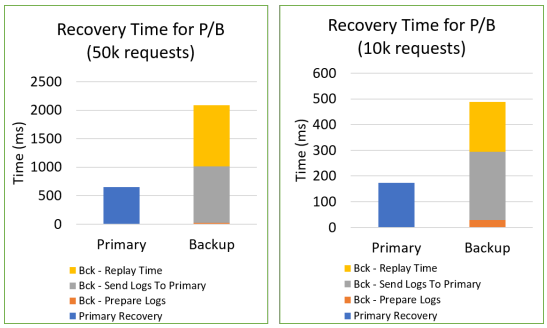

Figure 7: Recovery Time

3.3 Aligned and Unaligned Comparison

Access to 4KB-aligned offsets is faster than unaligned offsets. This is because for aligned offsets, we only access 1 file, whereas for unaligned ones we have to access 2 files causing higher latency and lower throughput, as indicated by 8.

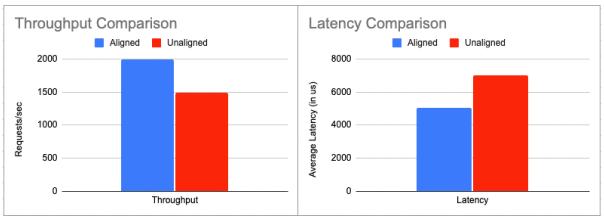

Figure 8: Aligned vs Unaligned Throughput and Latency comparison

4 Conclusion

In this assignment, we implemented a Replicated Blob Store system using Primary/Backup replication. A two phase protocol was designed for consistency and log files for crash recovery. For correctness testing, we introduced deterministic crashes in the address space. We also implemented a hashing technique to compare the data for consistency tests. Finally, we evaluated our system on various parameters such as throughput and latency (with or without crash) and measured the recovery time.

Demo Video: Link