P3 : Replicated Block Store

High Level System Design

We have used a primary backup replication strategy for servers. We ensure strong consistency in reads/writes. We use a load balancer for health check and service discovery.

Client : Clients issue read/ write requests to servers. Reads are processed by primary and backup but writes are always directed to the primary server. Crashes are transparent to clients by retrying attempts with service discovery.

Load Balancer : This server is assumed to be always available. Its primary purpose is to keep track of server states and send this information to the client for read/write. It periodically sends heartbeats to servers and updates the primary/ backup status.

Primary/ Backup Server : The primary server processes write requests and also sends these writes to the backup server to ensure consistency.

Data Design

Our data on the servers stays in different block states to ensure consistency.

DISK : This is the state of the block when it is written to primary or when it is committed to file at the backup.

LOCKED : This is the state of the block when it is written in memory at backup. This is required to stall the reads at backup if the block is in a locked state to ensure consistency.

MEMORY : Block is written in Memory state when backup is recovering and reintegration is in progress.

Write Protocol:

Steps of write() call:

1. Client issues write() to primary.
2. Primary server writes the block in memory buffer
3. Primary sends the writeRequest to the secondary server if its alive.
4. Backup writes the block to its memory and sends acknowledgement.
5. Primary does pwrite to file
6. Primary sends ack to client and sends commit message to backup asking it to do a pwrite to its local file asynchronously.
7. Backup calls pwrite to file and sends ack to primary

Strong consistency

We ensure strong consistency at all times. If the client reads data from the backup server after receiving acknowledgement from primary after 6 and before 7 step above, the reads are stalled till the data is committed at the backup.

If the primary fails after step 5 above, the backup server becomes primary and switches the state of all its blocks from LOCKED state to DISK state. Hence, even when primary comes back up, previous data will be present at the new primary and hence data will always be fully consistent.

Crashes and Reintegration

The crashes are fully transparent. Load balancer detects crashes by sending heartbeats to primary and backup servers. On detecting a crash, it informs the client as well as the other server of the dead state. Update server and lease info wherever required.

When a dead server comes back alive, before responding to the load balancer, it starts the reintegration procedure where it sends a reintegration request to the primary. The primary on receiving this request, sends all its blocks in the DISK state to the backup server. At the same time, it writes all the new writes in MEMORY state in the buffer to differentiate the records which have already been sent to backup. This is followed by a second phase of reintegration where all MEMORY blocks are sent to the backup and new writes are stalled at primary. After completion off second phase of reintegration, the backup responds to the load balancer and updates its status to enable receiving read requests from clients.

Performance and testing

We tested with zipf and uniform distribution workloads with multiple client threads. Results of latency and bandwidth are present in the presentation.