P3 - Replicated Block Store

Yilei Hu, Wiley Corning, Yatharth Bindal

1 Design and Implementation

We implemented our replicated storage system following a Primary / Backup design with two nodes. When both nodes are live, all client operations are handled by the primary server, which forwards all write operations to the backup server. If the primary crashes, the backup will temporarily assume its responsibilities and handle client requests. While either node is offline, the other will retain information about new updates to allow for later resynchronization.

Our project is built with C++, and uses gRPC as the underlying communications layer.

1.1 Replication Strategy

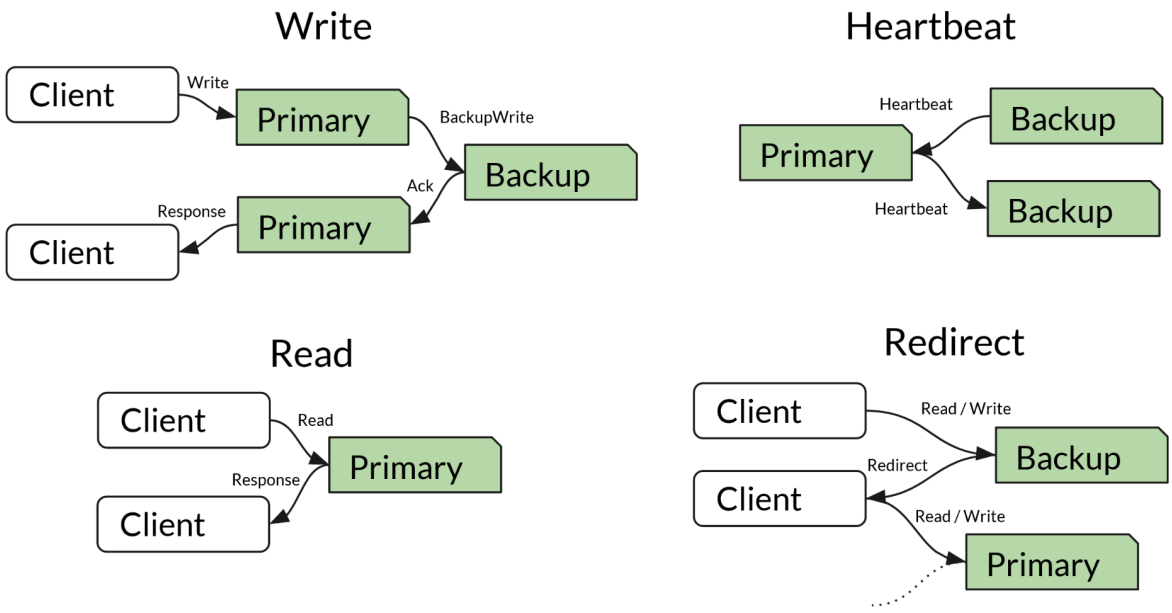


Fig 1. Protocol events when both servers are available.

Our protocol allows clients to read and write 4KB blocks of data starting at arbitrary byte addresses. During normal operation, each write is replicated immediately once it is received: the primary will only return from a write RPC once its update has been applied to both servers.

We assume no disk failures or other sources of Byzantine fault. We assume that at least one server will be healthy at all times; that is, if one server has crashed or is in the process of recovering, its counterpart is expected to remain available to help it recover. We also assume

that no network partitions will occur. If both servers were to remain live but unable to communicate with each other, each would begin acting as an independent source of truth, creating inconsistencies in the state. In this way, our system provides high consistency and availability at the cost of low partition tolerance. Any dropped request between the servers is interpreted as a sign that the receiver has crashed, and that the sender should operate standalone until it helps the receiver to recover.

Given that these assumptions hold, our system maintains several desirable invariants. We guarantee that at most one server will accept Read() and Write() requests at a time; i.e., that ownership of the block store is mutually exclusive. Strong consistency is guaranteed: any read will always return the most recently written data. The system can eventually make a full recovery from any transient faults, as long as the crashed server continues trying to recover.

1.2 Durability

Data is persisted to disk in a single large file. When reading and writing data, the address associated with the request is interpreted as a byte offset from the start of the file. Access to the file is guarded by a mutex, preventing inconsistencies due to data being written on one thread and read or written on another.

When handling a write or backup-write request, data is always written to the disk before continuing with the protocol. Each request is therefore permanently committed before we report success. If the caller does not receive this confirmation, then it should retry its request. Writes in this protocol are idempotent, so an at-least-once semantics is acceptable.

It is possible that a server may crash after it has written data to disk but before it is able to report success to the caller. If this happens when the primary is forwarding a request to the backup, or when one server is helping another to recover, then there is no loss of data: the server that sent the request will resend it during every subsequent recovery attempt, so it will eventually be committed on both. However, if the primary crashes between writing to its disk and contacting the backup, then the relevant section of the primary's file may be inconsistent with the backup once the primary recovers. The inconsistency will be resolved if the client retries its request, in which case either a standalone backup server or a recovered primary server will have the opportunity to commit the write properly. If we were to iterate further on this system, we would consider adding a simple journaling mechanism (writing each update to a temp file until it is replicated) which would fully solve this issue.

1.3 Crash Recovery Protocol

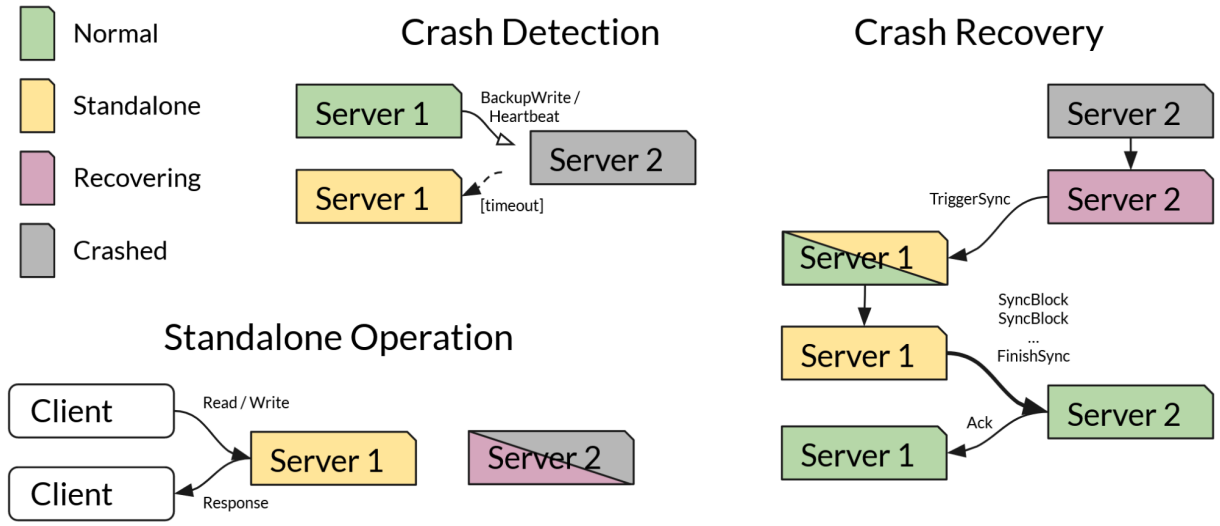


Fig 2. Protocol events when one node has experienced a fault.

Each server operates as a simple state machine with three states: *Normal*, *Standalone*, and *Recovering*. The server's state determines its behavior when responding to client requests and applying updates.

Servers in the *Normal* state function as described in Section 1.1. A *Normal* primary server communicates with the backup via backup-write requests, which are sent whenever a new write request is received from a client. A *Normal* backup continually pings its primary with heartbeat messages to verify that the primary is still available.

If either of these server-to-server requests fails, the request's sender moves to the *Standalone* state. The primary and backup each have identical behavior in this state, and will accept read and write requests from clients. When a *Standalone* server processes a write request, it first persists the data to disk; then, instead of forwarding the write to the other server, it adds the address of the write to its in-memory *dirty* set.

Servers may also enter the *Standalone* state if they receive a recovery request when in the *Normal* state; this indicates that the other server has crashed, restarted, and began recovery before the crash could be detected.

The primary server is initialized in the *Standalone* state when it is first launched. When the backup server is launched, and when either server is restarted after a crash, they are initialized in the *Recovering* state. Recovery is a multi-stage process, and can be safely restarted at any point if messages are dropped or the recovering server crashes again.

Let us suppose Alice is a recovering server and Bob is her counterpart. First, Alice generates a random integer ID for her current sync request. She sends this to Bob in a TriggerSync RPC. Bob acknowledges this request, and then (on a new thread) sends each of the blocks listed in the *dirty* set back to Alice, each labeled with the sync ID. Bob finishes the sequence by sending a FinishSync RPC, which contains the sync ID and the number of blocks sent in this sequence. If the sync ID is current and Alice has received the correct number of blocks, she changes state to *Normal* and confirms the end of recovery, at which point Bob goes to *Normal* and clears his *dirty* set.

If any of Bob's RPCs fails during this sequence, he will stop the sequence immediately, inferring that Alice must have crashed and will need to restart synchronization. On Alice's end, if no new sync messages are received within a given time window, Alice will generate a new sync ID and send a new TriggerSync. Any messages subsequently received with the old sync ID will be discarded. This prevents a scenario in which Alice crashes and restarts during recovery, and then accepts a stale FinishSync erroneously.

While a server is recovering, it will refuse all client requests (thus bouncing the client to the other server). A *Standalone* server, on the other hand, may service new writes while in the process of synchronizing; a shared mutex is used to ensure that no updates are applied using standalone logic after synchronization is complete. The availability of the overall system is therefore maintained during recovery. Other locks are used to prevent data races relating to the state value and recovery sync ID.

2 Testing and Measurement

In our setup, we used separate Google Cloud VMs to host our client and primary and backup servers. Each VM in our setup had a two-core processor and 4 GB of memory, and ran Ubuntu 20.04.4.

2.1 Correctness

To validate our replication and crash-recovery implementation, we created a mechanism for triggering crashes at specific points in the server code. The server scans incoming messages for certain specific addresses; if one of the trigger addresses is found, it may prime the server for a future crash or, if it is already primed, crash the process by dereferencing a null pointer.

We implemented crashes in five specific locations:

- Between storing data on the primary's disk and sending it to the backup
- Asynchronously, after finishing a write on the primary
- Between storing data on the backup's disk and exiting the backup-write RPC
- Asynchronously, after finishing a backup-write on the backup
- Just before the end of the recovery process

The use of a two-stage priming mechanism, where one message changes the server state so that the next message will crash it, allowed us to run test sequences in which all of the client's messages are eventually committed after retry despite at least one of them causing a crash. These sequences demonstrate that the crash and recovery are effectively hidden from the user.

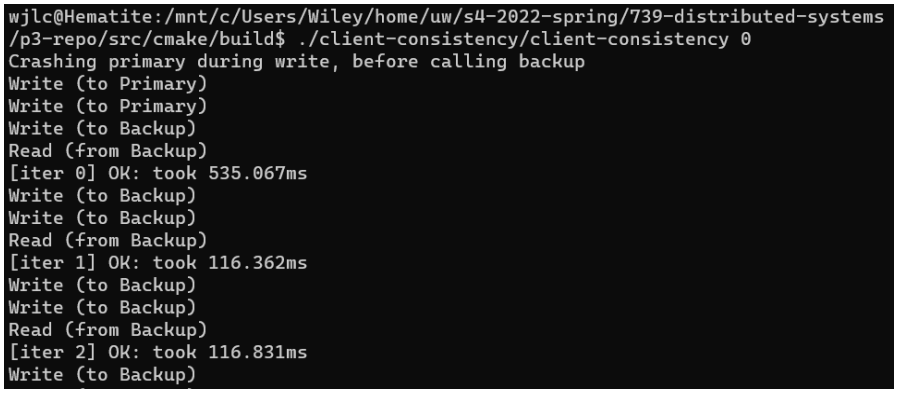
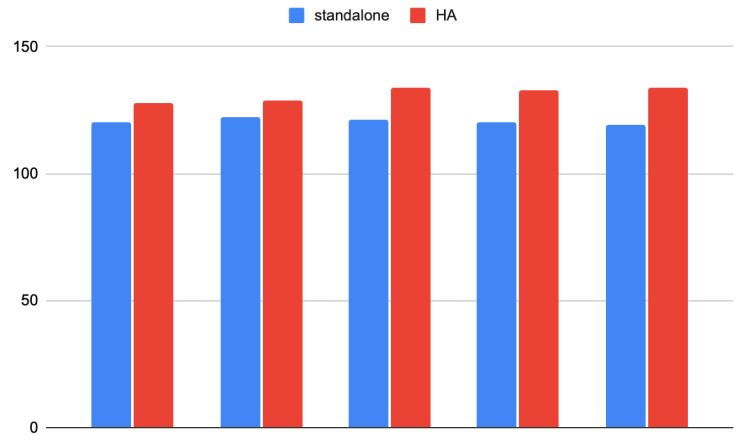


Fig. 3. The first message in this sequence crashes the client; subsequent messages are seamlessly processed by the backup in Standalone mode

2.2 Performance

Write/Read performance

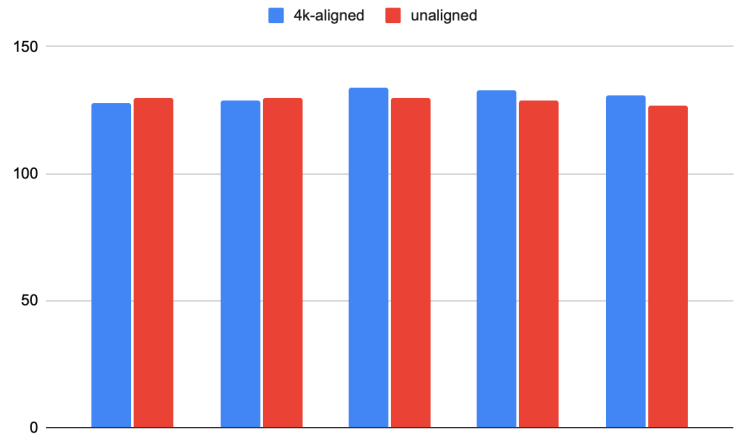
Time(ms) spent on writing different 4KB blocks to random addresses and reading them back:



Each 4KB block is a randomly generated string. Blue bar is in standalone mode and red bar is in HA mode. Writes and reads took ~5ms longer in replicated mode because of data transfer.

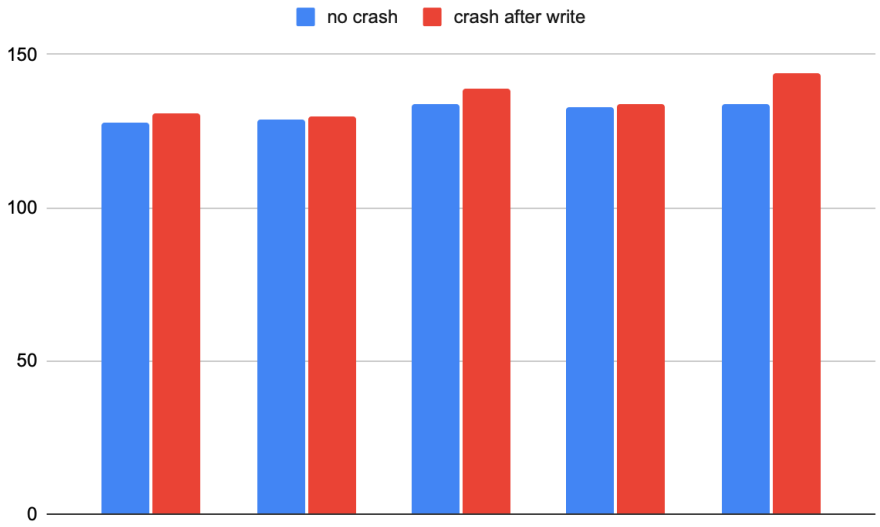
4KB aligned v.s. Unaligned addresses

Time(ms) spent on writing 4KB blocks to 4KB aligned and unaligned addresses and reading them back:



This is in replicated mode. Blue bar is 4k aligned and the red bar unaligned. As expected, there's no significant difference in performance because essentially we are writing to an offset within a file.

No crash v.s. Crash after write:



This is in HA mode. Blue bar is a normal write/read pair, red bar is crashing the primary after writing to it and then the client issuing the read to the primary. The returned block is from backup, which should match with the original block. In that way, we proved the availability, the crash is hidden from the client. As expected, it took longer because the client issued the read to the primary first but got a bad response then issued the read to the back up and got the block, which added a round-trip time.