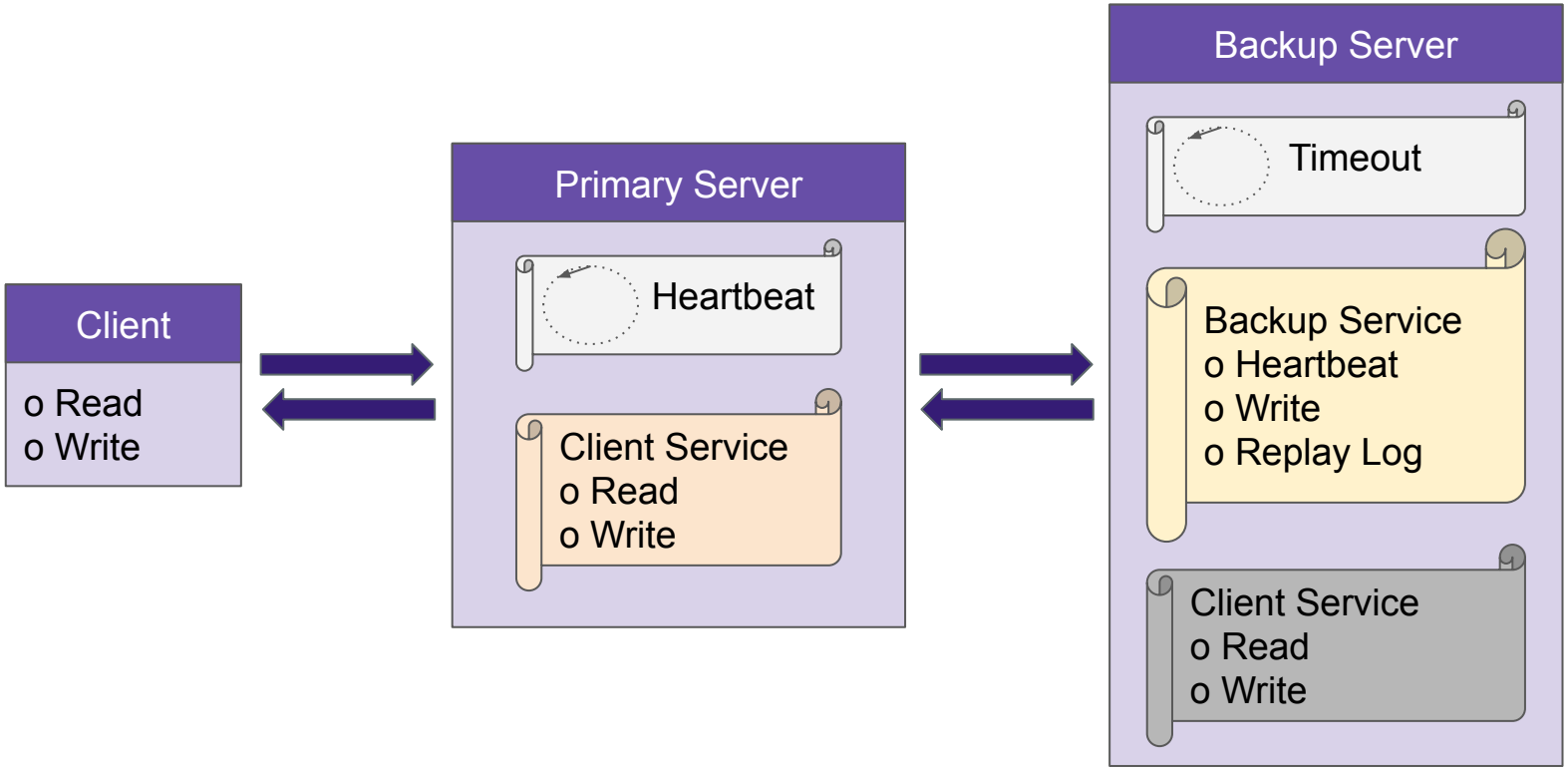
Replicated Block Store

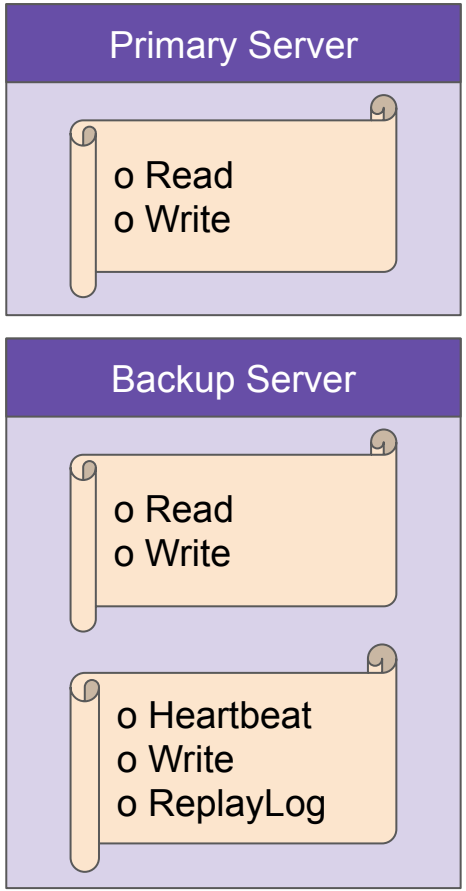
CS739
Spring 2022
Ethan Brown, Marvin Tan
Sam Kottler, Todd Hayes

Architecture » Primary-Backup



Architecture

- Primary-Backup Replication
  - 2-nodes
  - Only primary handles client requests
  - Backup will respond if client tries to communicate to point client to primary
- Used C\C++ with gRPC
  - Two RPC services
- Multi-threaded
  - Separate heartbeat thread
  - Concurrent reads and writes



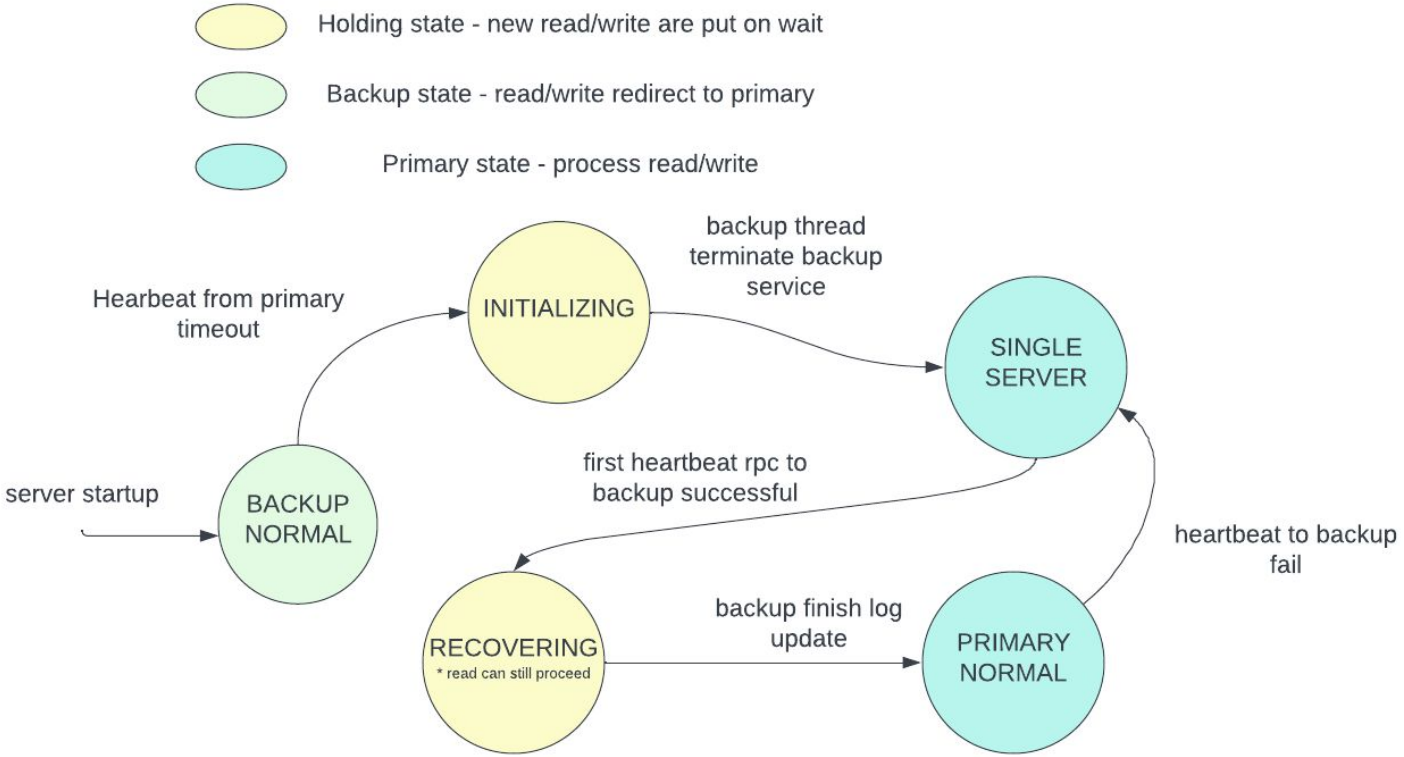
Client Library

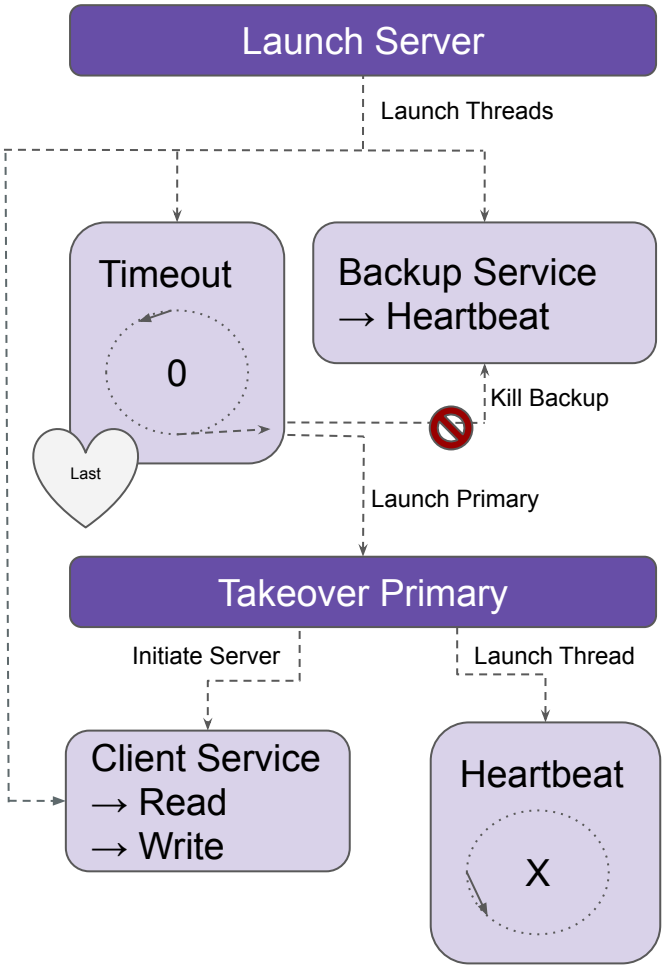
- read/write/init
- Hide failures by blocking until it can reach one of the servers
- Anticipate multiple clients
  - Multiple machines
  - Multiple client processes per machine
  - Tests with both
- Attempted to create test with FUSE/losetup
  - Would create something that looks like any other block device
  - Could be partitioned / formatted with filesystem / etc
  - Code is mostly written, but ran out of time to actually get it working

Server

- A server primary/backup pair export a single 256GB volume
  - Each 4K block is stored in separate file
  - Files aren't created until written to
- Reads and writes are 4K
  - Not necessarily aligned
- Primary decides ordering for strong consistency
- Primary stateful only when backup is down
  - We're assuming at most one server will crash at a time

State Transition



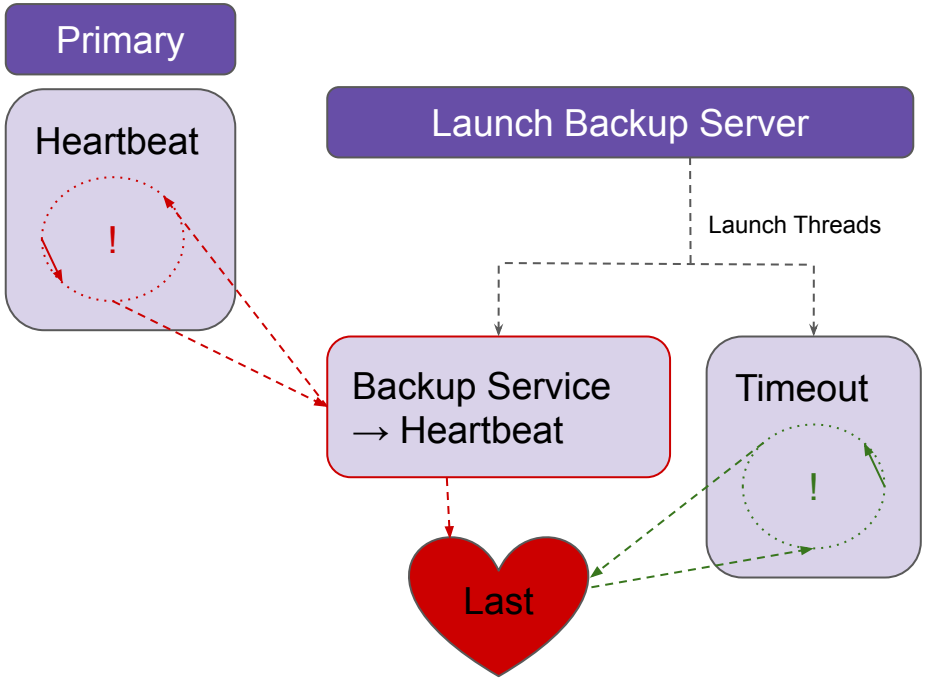


Primary Detection

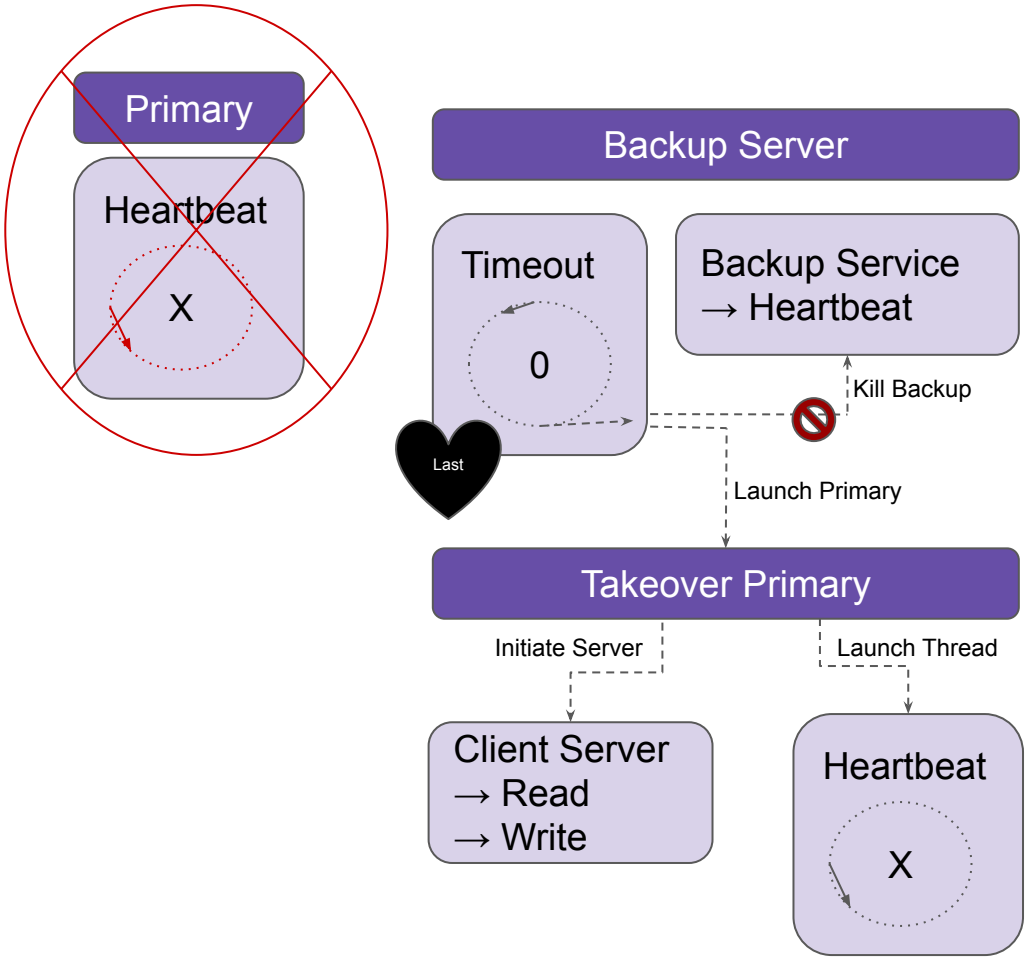
- Launch Server
- Server comes up as backup server
- If no heartbeat detected within initialization timeout (9s), initiate primary takeover
    - Kill backup service and timeout thread
    - Activate client service
    - Initiate heartbeat thread

Normal Operations

- Primary heartbeat thread contacts heartbeat() gRPC at set intervals (1s)
- lastHeartBeat updated on backup service for any grpc calls
- Timeout thread wakes up at set intervals (2s). If lastHeartBeat updated while asleep, goes back to bed



Failover



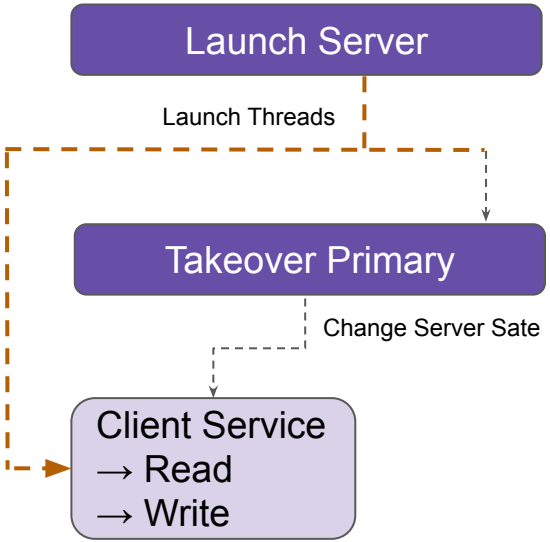
- Identical to primary detection
- If heartbeat fails, backup transitions to primary
- Never transition back to backup outside of failure

Client-Server Launch Approaches

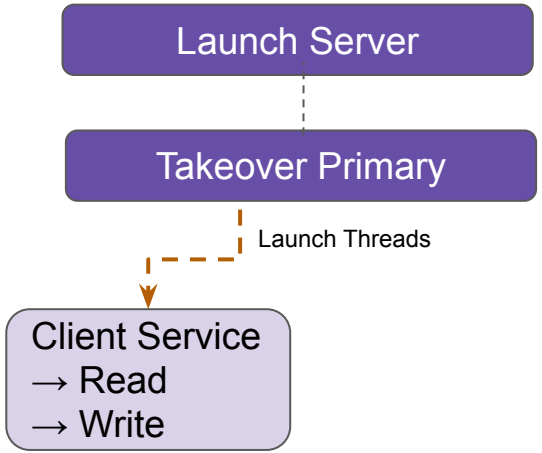
Considered 2 Approaches

- Early
  - Launch with backup
- Late
  - Launch with primary

Early Launch



Late Launch



Client Response to Primary Failure

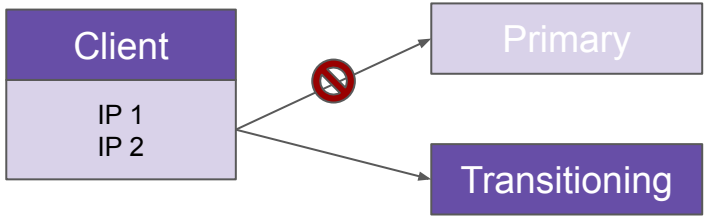
Client Aware

- Client knows IP for both servers, on failure try alternative
  - Works with early and late launch

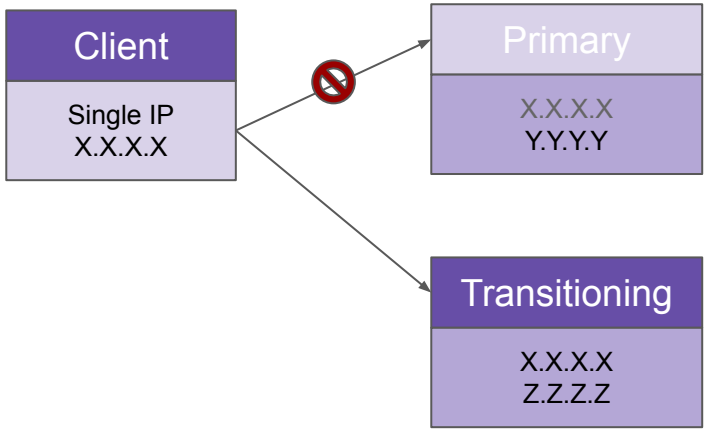
Client Transparent

- Client knows single IP, on promotion to primary, server listens to that address
  - Only work with late launch, must know IP to listen on first

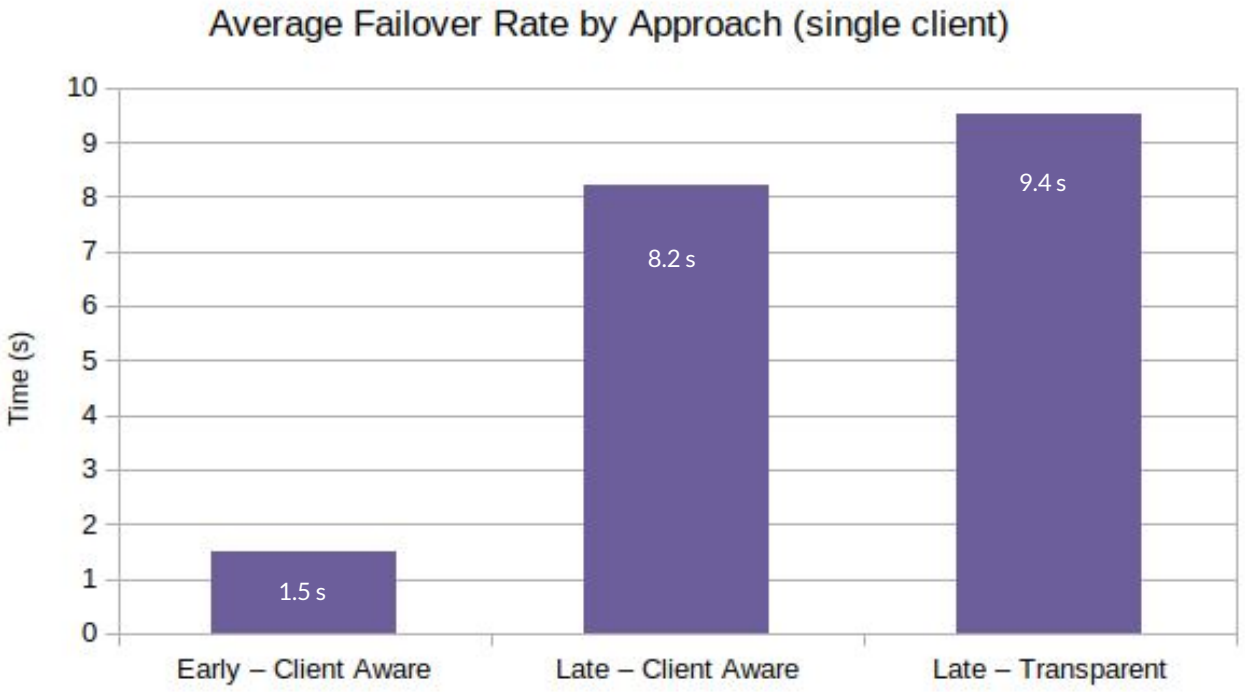
Client Aware



Client Transparent



Client Response to Primary Failure



Early launch of client service resulted in significantly faster failover. This approach only worked with client aware failover, and is what we utilized in our final prototype

The late client-service launch appears to be the source of the slowdown

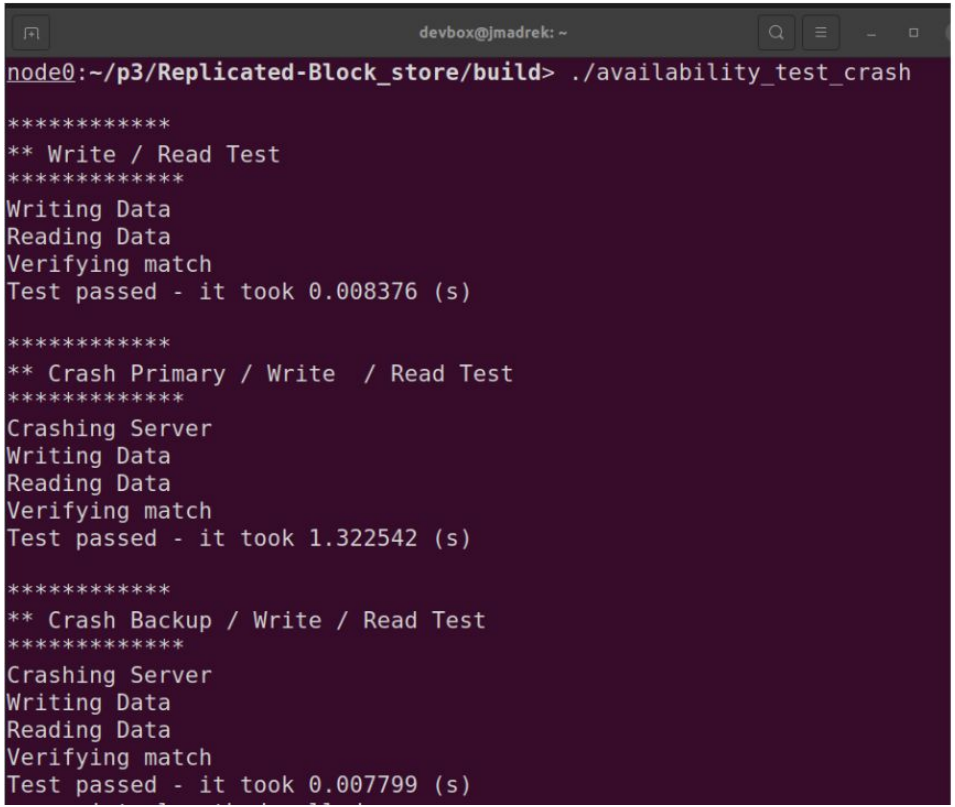
Availability Demo

- Modified version of consistency demo
- Runs 3 test scenerios
  - No crash
  - Primary crash
  - Backup crash
- Performs Write \ read \ match data

Results

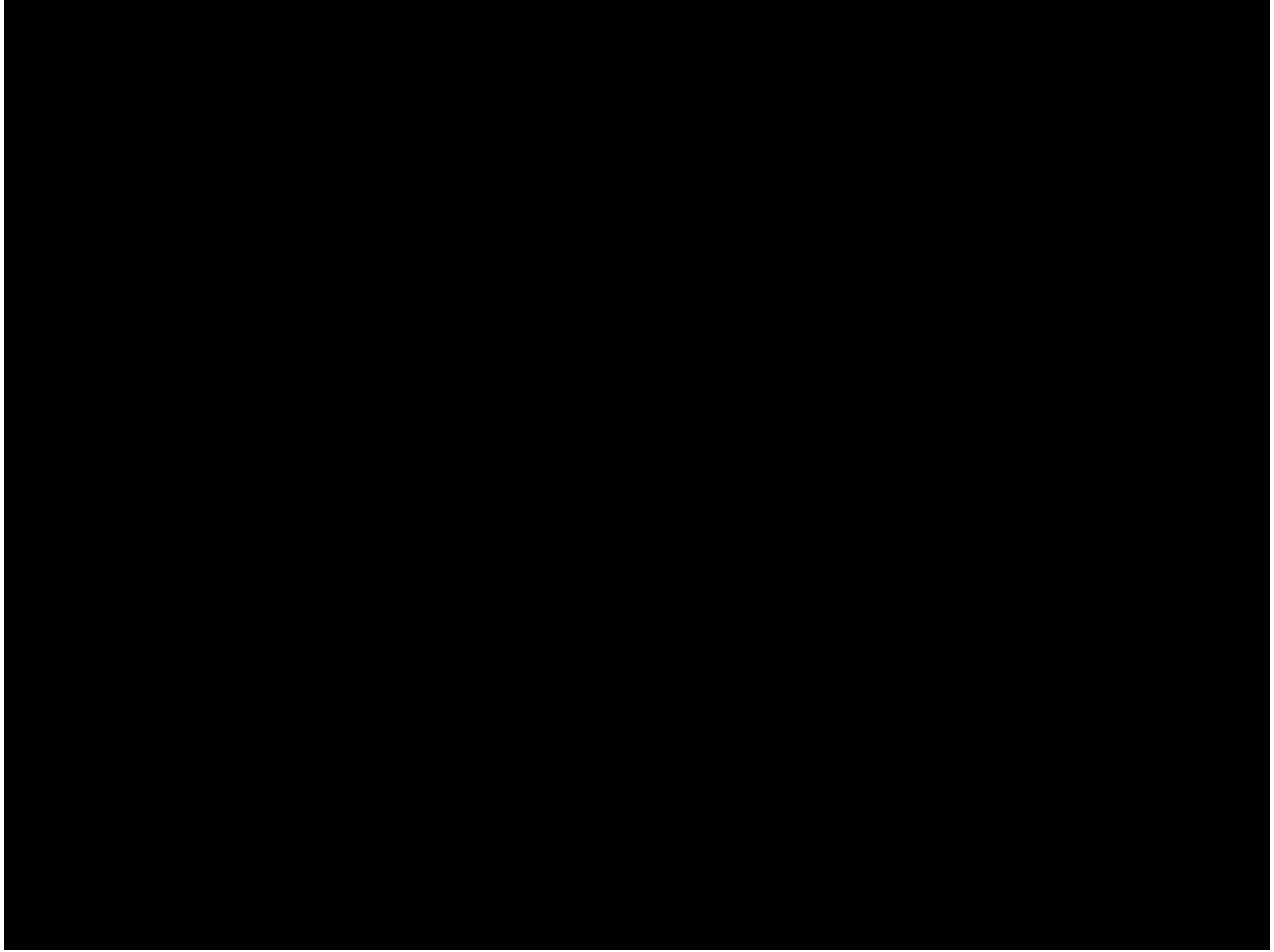
- No errors despite dead servers

Type	No Crash	**Crash Primary**	Crash Backup
Runtime	0.00837 s	**1.32254 s**	0.00779 s



Terminals -

Client	Server 1 (starts as primary)
	Server 2 (starts as backup)



Primary-Backup Read/Write Protocol

Upon receiving write request, check backup status:

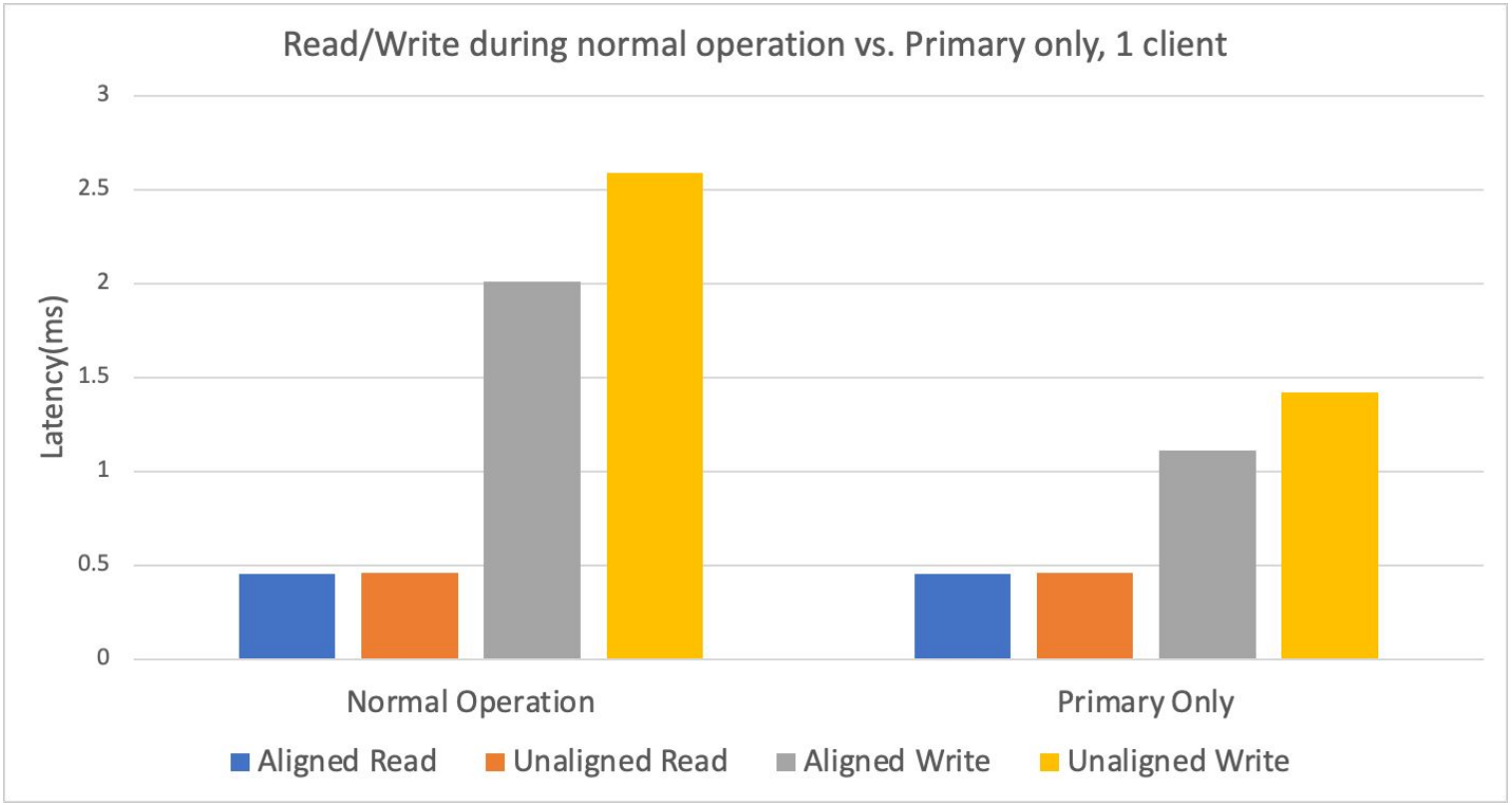
- Case 1: Healthy
  - Write to backup synchronously
  - Write local
- Case 2: Single-server
  - Write to log
  - Write local
- Case 3: In crash recovery
  - Wait. When backup finishes recovery, do case 1

Upon receiving read request, primary server reads data and responds alone

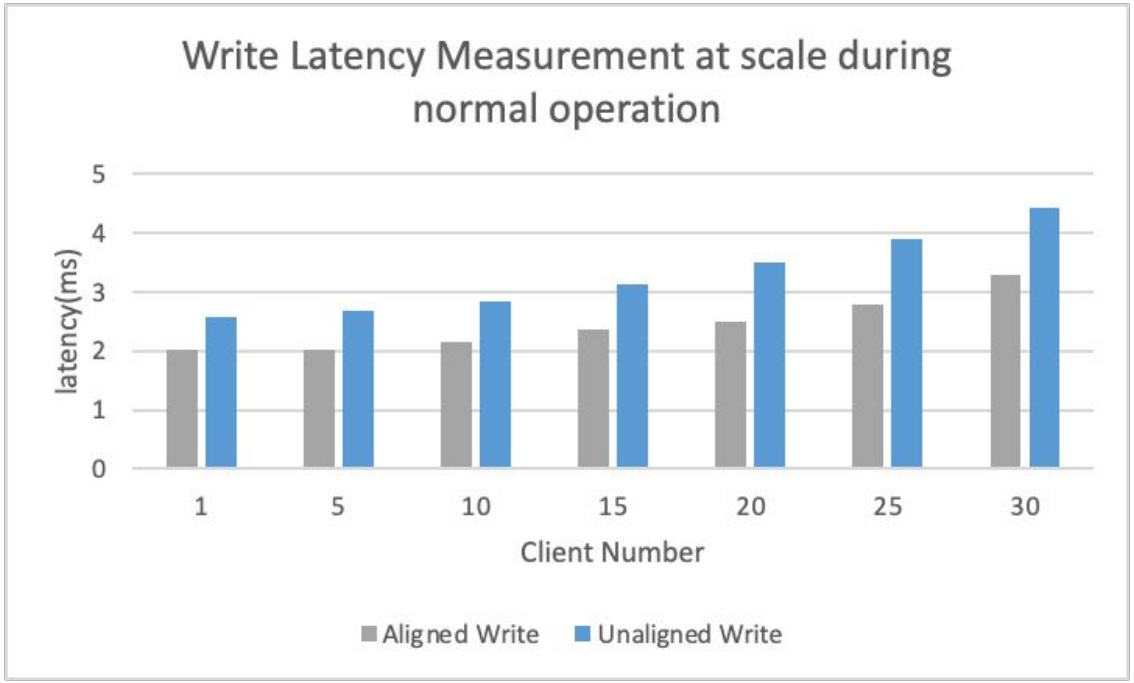
Primary-Backup Read/Write

- Primary acquires read/write locks, backup does not
  - Synchronous primary writes and locking guarantees backup write order
- Per-block 4K-byte files representing the volume
- .tmp file writes/swaps for mid-write consistency
- Operations performed on 1 file for aligned, 2 files for unaligned

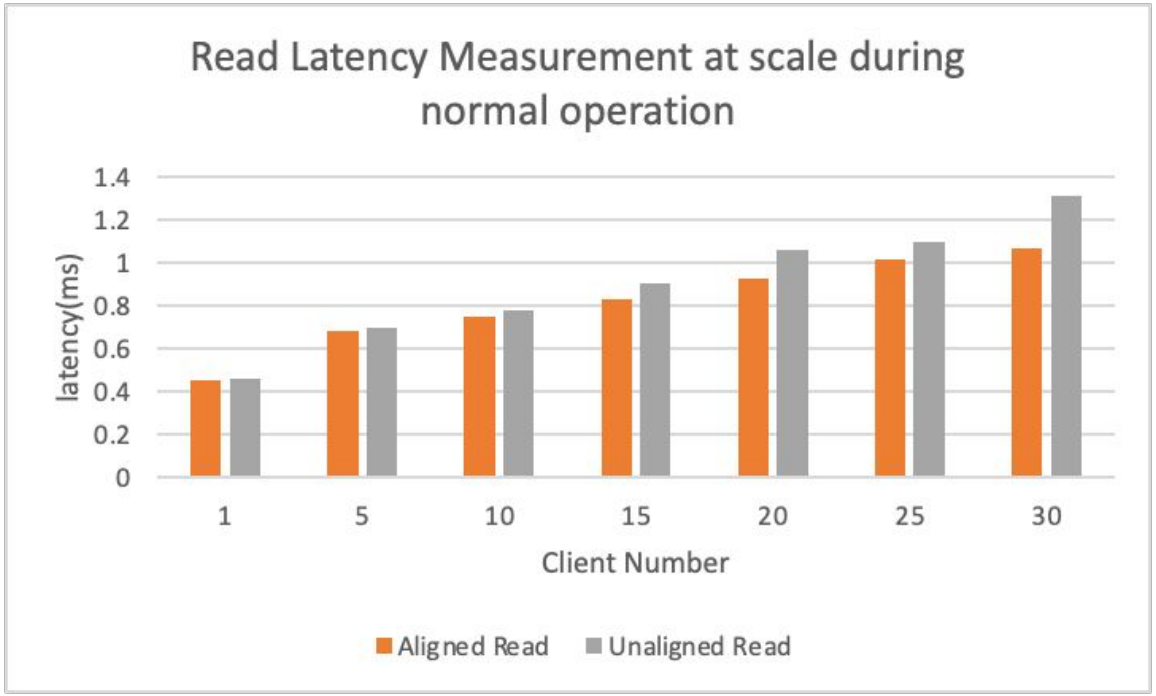
Average Read/Write Latency



Average Write latency at scale



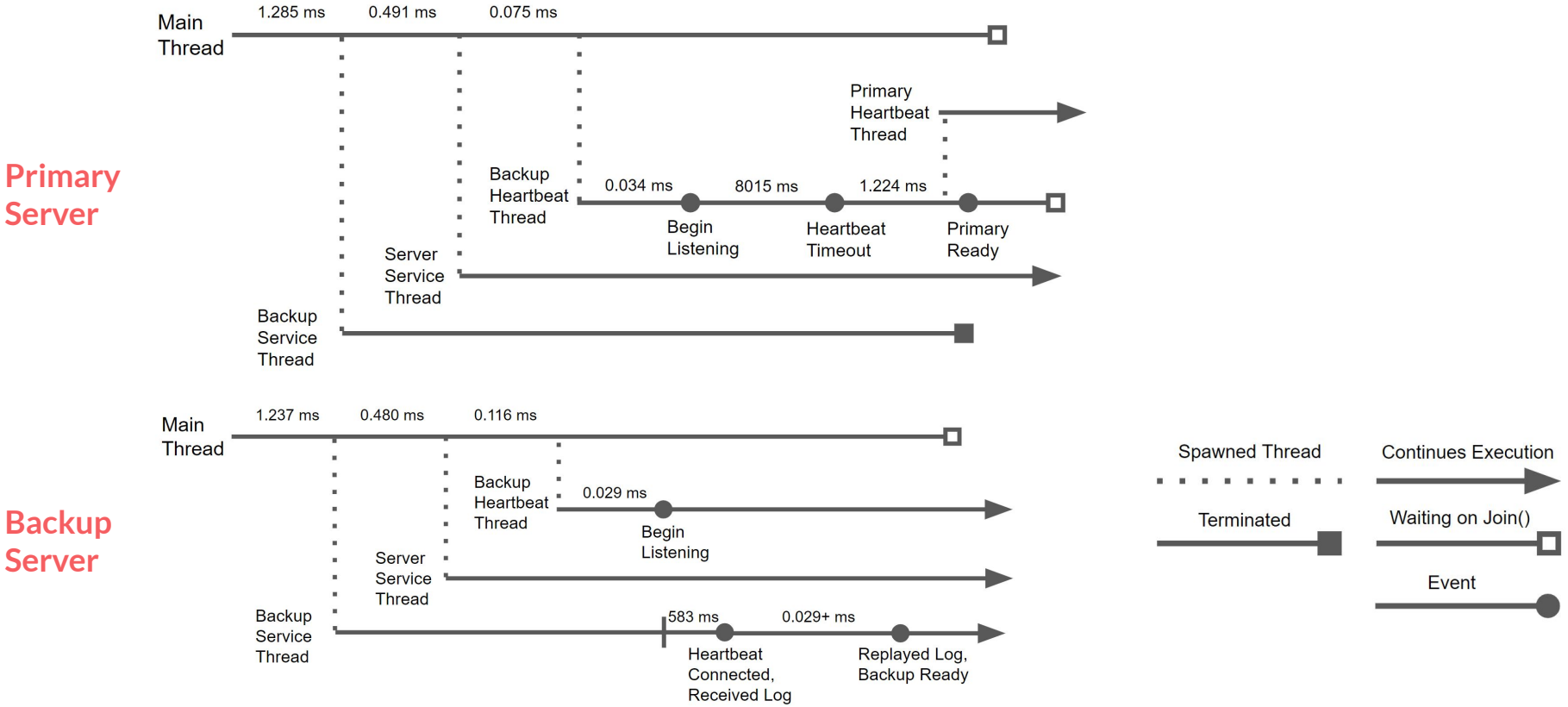
Average Read latency at scale



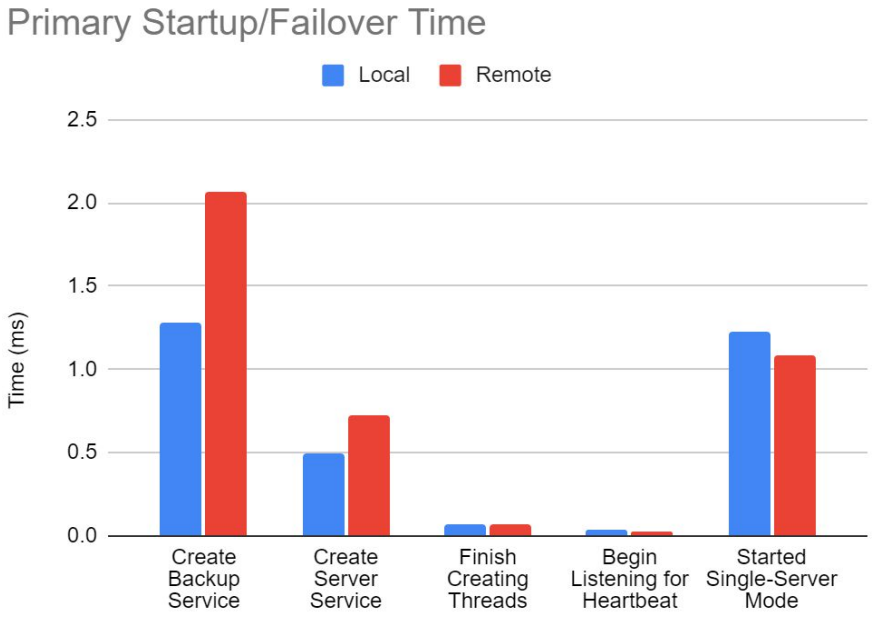
Logging and Crash Recovery

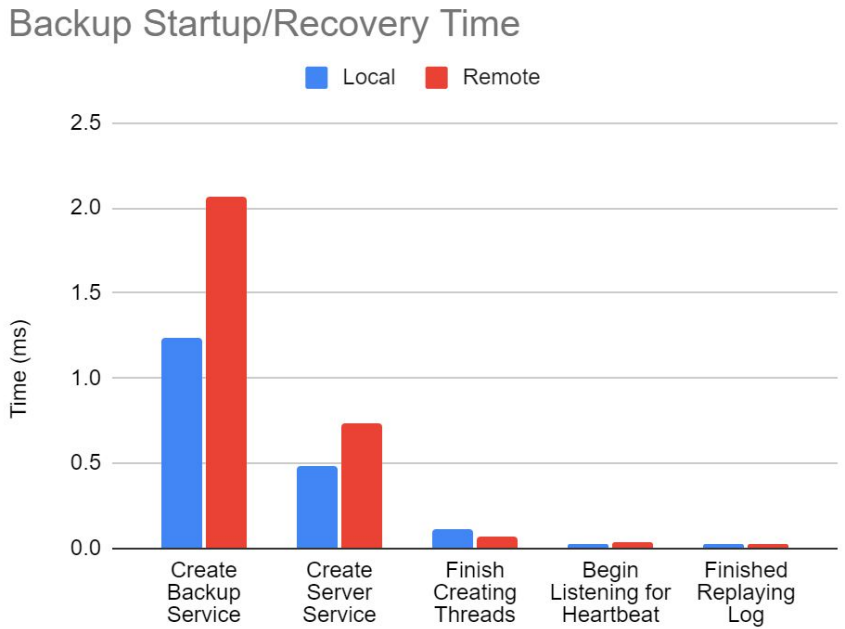
- During single-server mode, primary logs dirty offsets in non-repeating list
  - Optimization: Repeated writes to the same offset result in only one log entry to replay
- During backup recovery, the primary reads data from disk for each log offset and sends it to the backup
  - Guarantees most recent write is committed to backup for consistency
- Assume no primary crashes during replay or single-server mode, log stored in memory
- Primary only clears log after receiving OK from backup indicating committed data
  - Partially committed log data will be overwritten again upon recovery if the backup crashes during replay, ensuring consistency

Startup/Recovery Time



Startup/Recovery Time

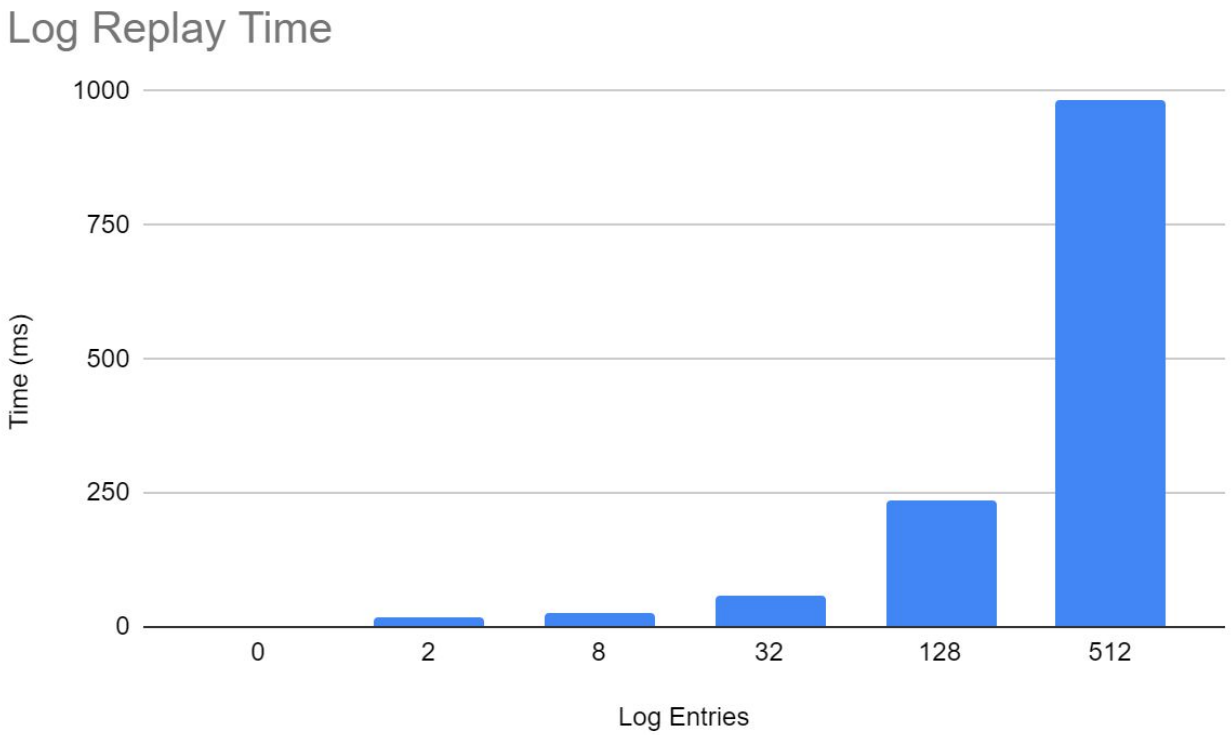




- 8015 ms (8 sec) to timeout after listening for heartbeat on primary
- 583 ms to connect after listening for heartbeat on backup

Startup/Recovery Time

- Approx. linear time to commit logged writes to backup



Locks

Per-block reader/writer locks

- Up to two per request (only on primary)
- Acquired in ascending order to prevent deadlocks

Recovery lock

- Single reader/writer lock to prevent writes during recovery
- Acquire shared lock before writing and exclusive lock before sending log to backup

Wait to start read/write on primary until done initializing

- Use a condition variable that is notified when done initializing

Crash Triggering

Dynamic crash codes:

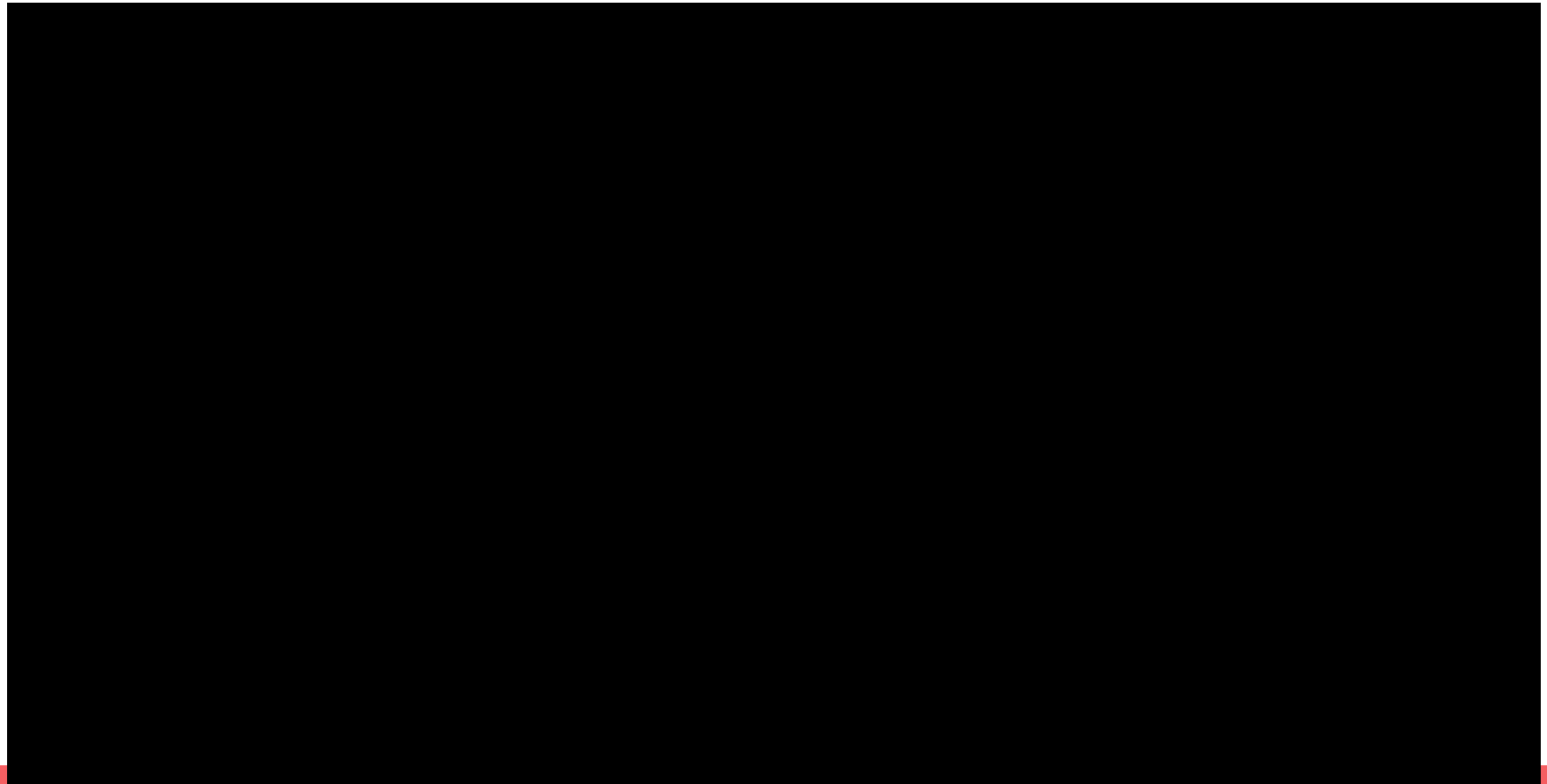
```
//Client:
char code[] = {'C', 'R', 'A', 'S', 'H', 1, 0, 0};
write(write_buf, *(long*) code);

//Server:
long offset = request->offset();
check_offset((char*)&offset, 0);
```

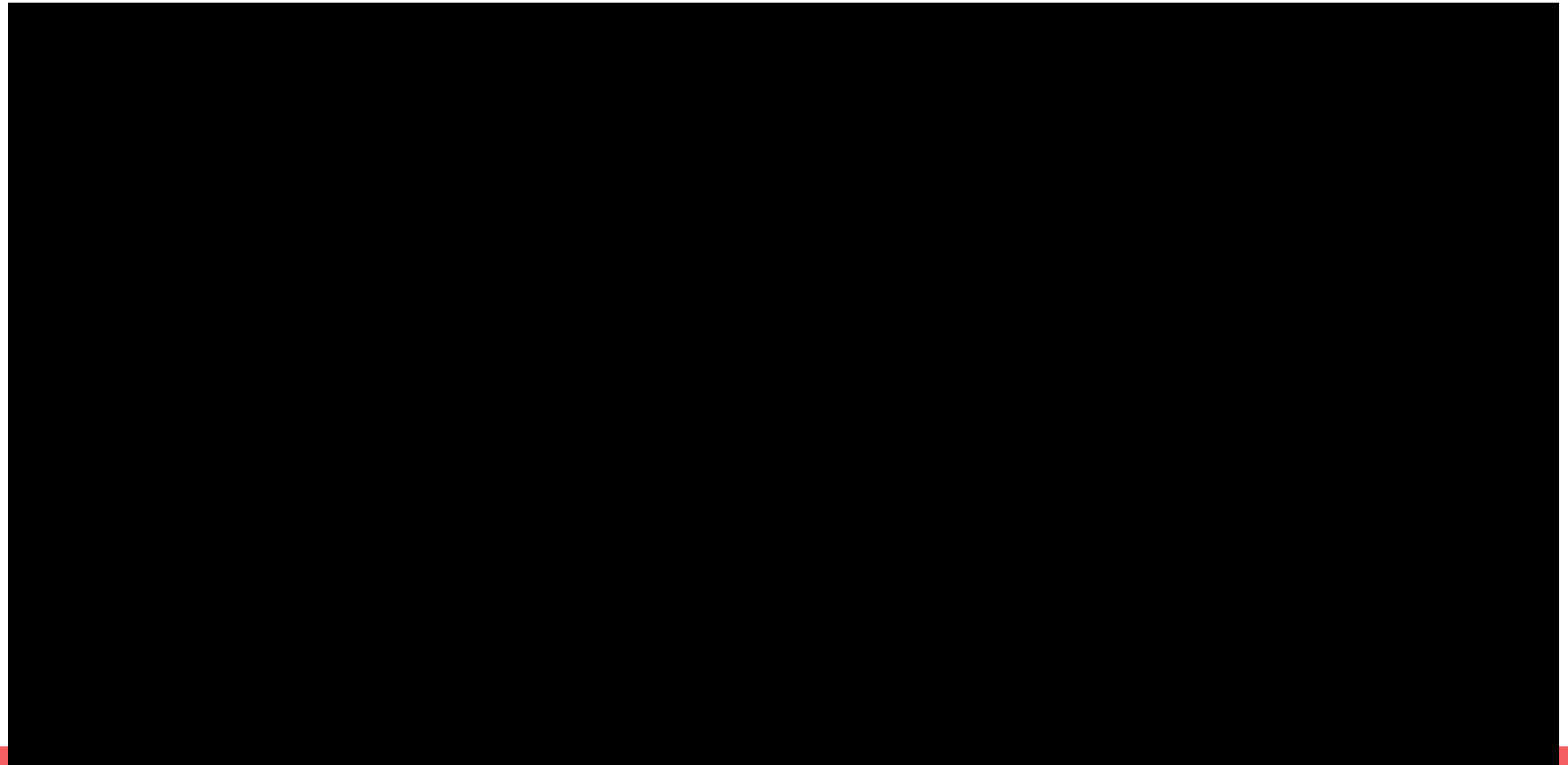
Add request offset to log.

Can crash primary and backup in read/write/recovery/etc from client.

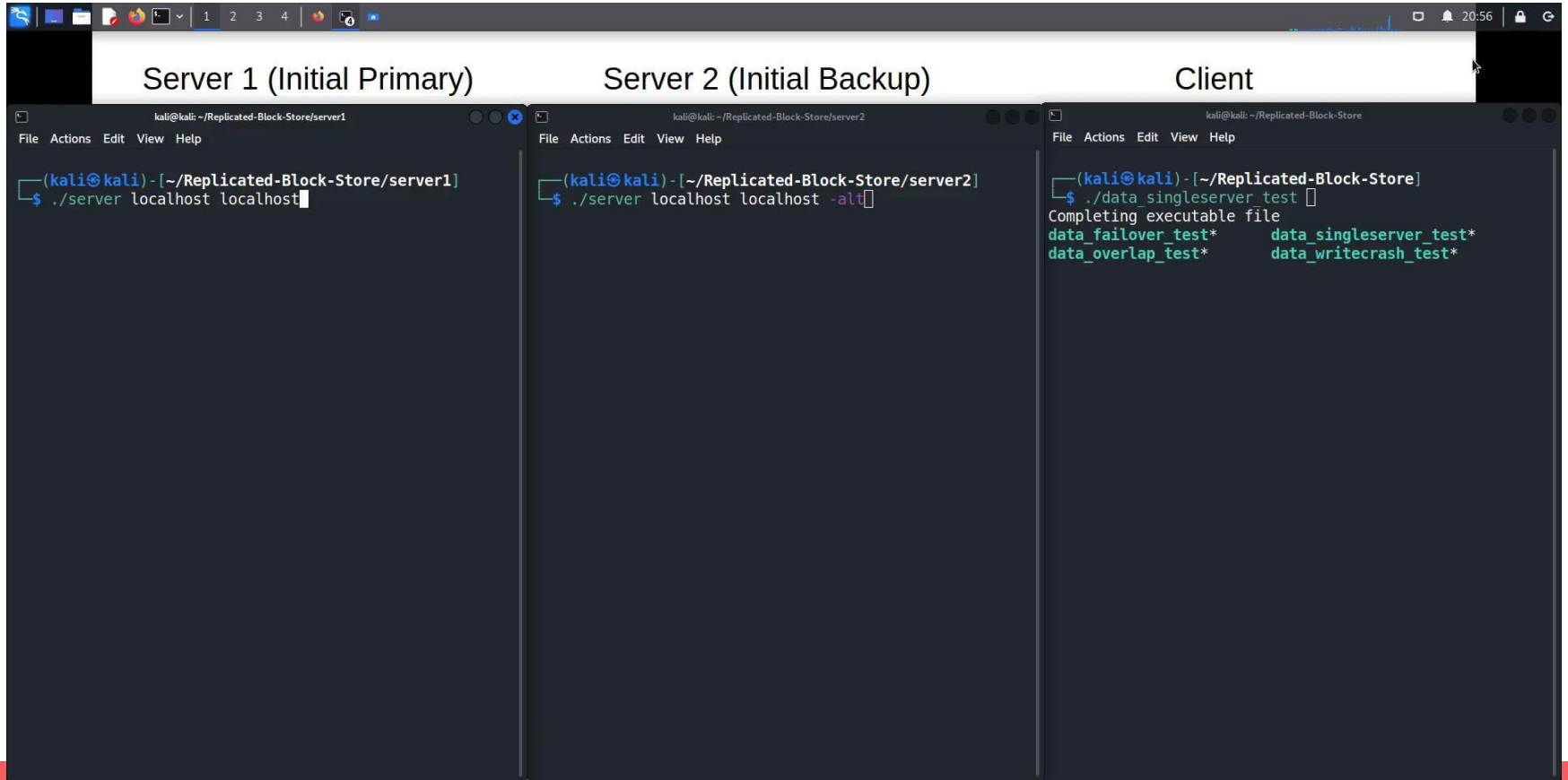
Normal Operation Consistency



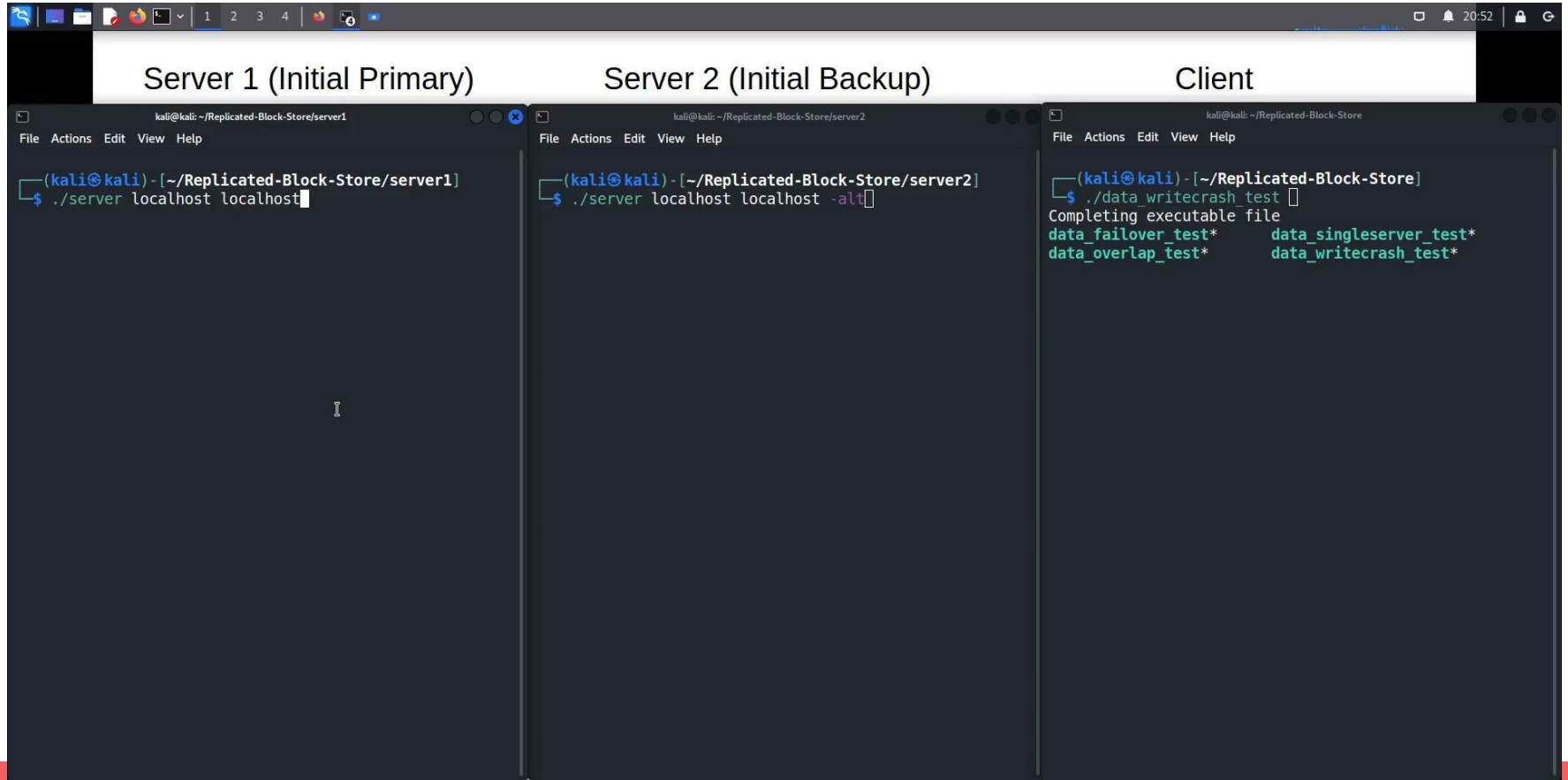
Failover Consistency



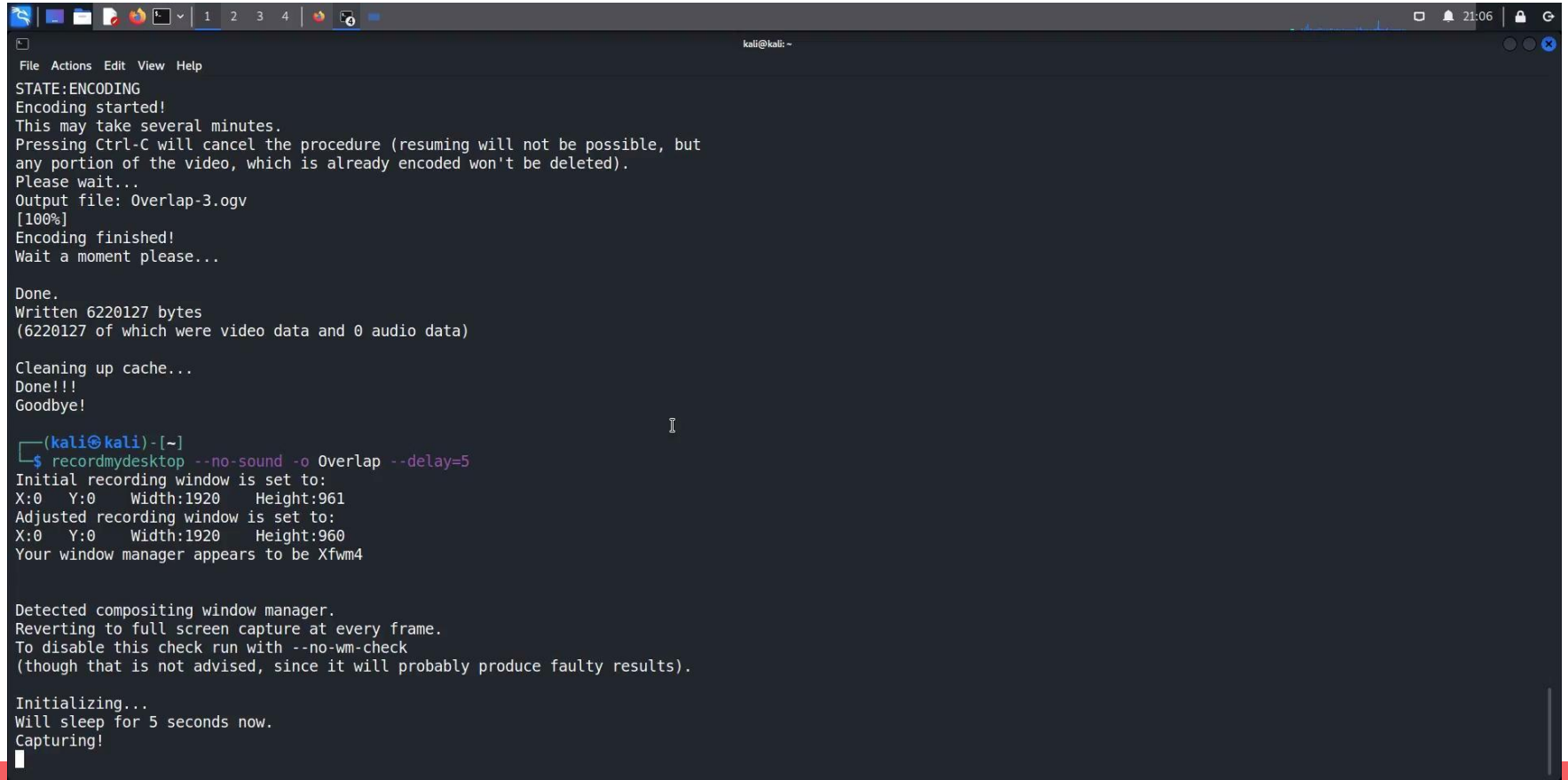
Single-Server and Recovery Consistency



Mid-Write Consistency



Log Replay Consistency



FIN