(Distributed Rocks!)DB

CS 739 Project - 4

Akshay Parashar
Divyanshu Kumar
Hemal Kumar Patel
Mohit Loganathan

(Dept. of Computer Sciences/CDIS)

Motivation

- RocksDB is a **log structured database** engine.
- It provides a persistent key-value store with keys/values of arbitrary sized byte streams.
- Configurable to high random-read workloads, write-heavy workloads and combination of both.
- Organizes data in **sorted order.**
- Maintains three different constructs:
  - **Memtable**: In-memory data structure
  - **Sstfile**: persistent data store storing keys in sorted order
  - **Logfile**: Persistent checkpoint log for memtable recovery in case of failures
- Limitations:
  - **Not distributed**
  - **No failover**
  - **Not highly Available**

Our Contribution - To solve these limitations!

Goals

- Make a distributed key-value store by plugging any KV store
- We chose to distribute and scale-out RocksDB.
- Final system should :
    - accommodate failures,
    - be distributed,
    - highly available, and
    - can be scaled out

  In addition to supporting all the existing features and guarantees of RocksDB!

Key Assumptions

- Non Byzantine Failures
- Stable Storage - No Disk Failures or Corruption
- No RocksDB/<underlying DB> Corruption
- One node always up in the cluster
    - Backup becomes primary

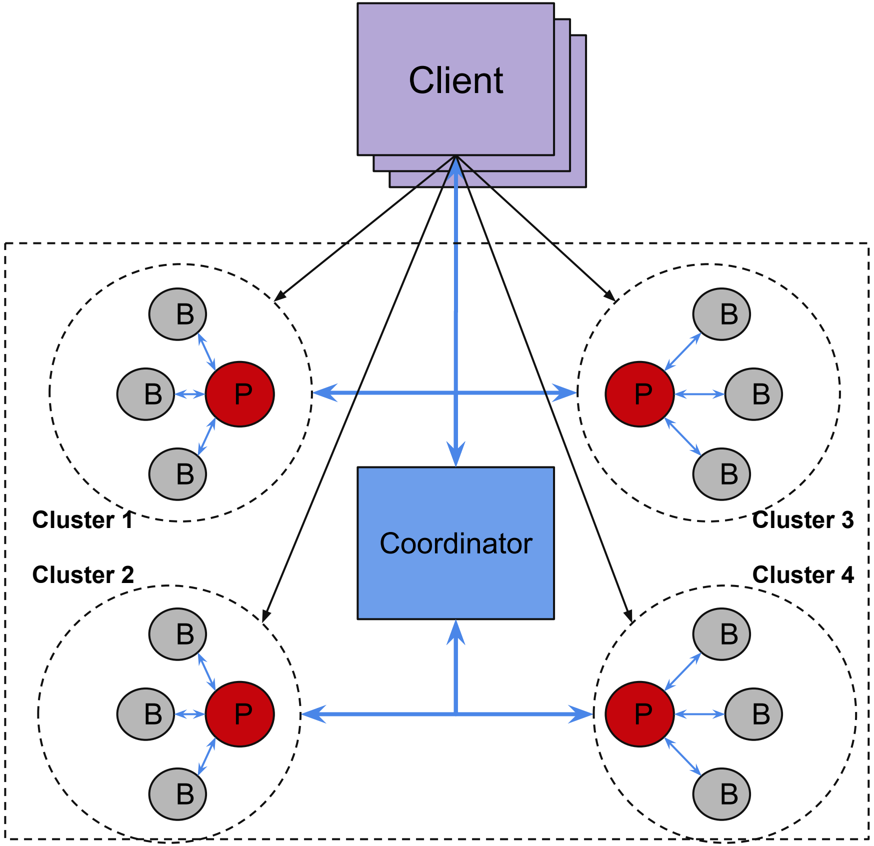
System Architecture and Design Decisions

System Architecture

1. **Centralized Coordinator:**
   Coordinator maintains system state dynamically as nodes enter and leave the system.

2. **Multi-Cluster Primary-Backup:**
   Read/Write on Primary and Reads from backup. rocksDB instance per P/B server.

3. **Crashes**:
   System can handle n - 1 failures per cluster. Reintegration of a node to the system is handled using checkpoints.



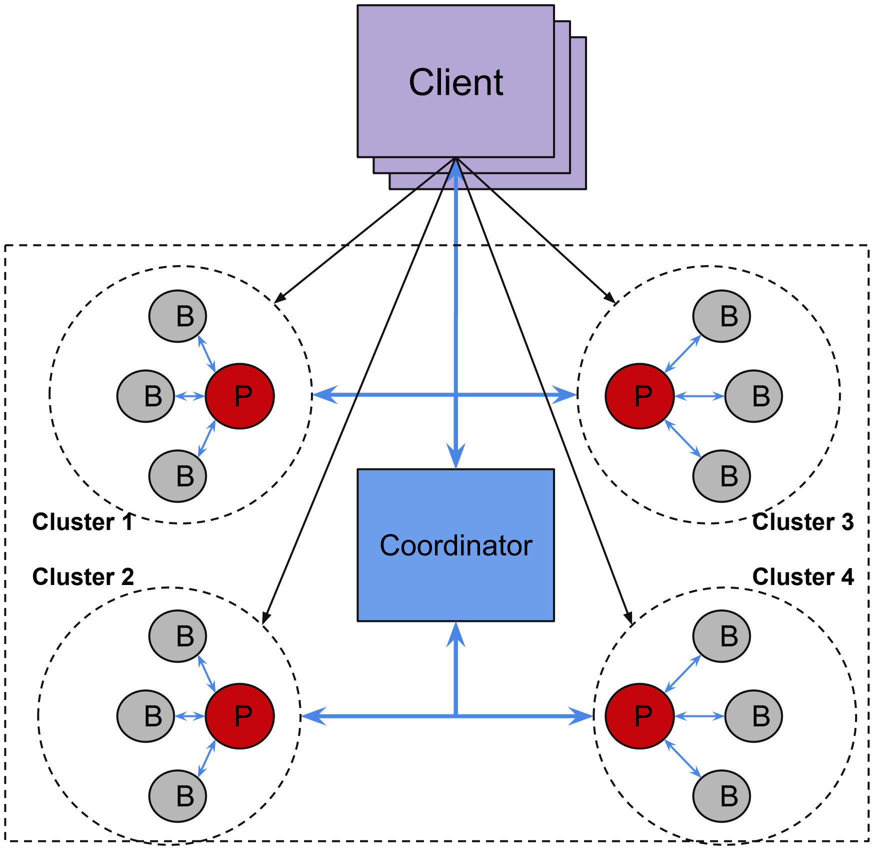
System Architecture

4. **Client Side Caching with server callbacks**:

   Client caches data locally for future use. Primary server is responsible to notify in case of any write on that key.

5. **Sharding of key-range:**

   Keys are distributed equally to all the available clusters to support large range of keys



Design Decisions

- Read heavy workload - Eventual + Optional client caching
    - Configurable Read Consistency - **Strong** and **Eventual**

- Write heavy workload - Fast-acknowledge writes + Sharding
    - Configurable Write Consistency - **Durable** and **Fast-acknowledge (loose durability)**
    - Scalability - Large number of key-value pairs and clients - **Sharding**

- Fast failover - rapid update of **system state** by centralized **coordination** service.

- Large number of nodes in system
    - **Centralized heartbeat** with Coordinator
    - Availability not affected by coordinator crashes

Design Decisions

- **Cache Consistency** - 2 Choices : Client driven staleness check vs Server driven notifications.
- Our choice - Lease with server driven notifications.
    - **Lease** - To reduce state at server.
    - **Notification** - rpc time to check cache staleness from client side is approx. same as a read rpc. Notification reduces rpc traffic for read heavy workload.
    - Client has **streaming rpc for cache invalidation with the primary** to reduce the overhead of new rpc for each invalidation.

Design - From Server's View

- Read():
    - **Strongly consistent :** served by Primary server.

    - **Eventually consistent** : served by any of the backup servers.

- Write():
    - **Durable** : Blocking call until parallely replicated on all backups.

    - **Fast-Acknowledge** : Clients are Ack'd as soon as write is done on primary locally. Replication is done asynchronously in background.
        - ***Loose durability*** semantics.
        - Data can be lost in case of primary crashing before replication.

Design - From Server's View

- **Replication** to backups:
  - Parallel replication to replicas

  - Multi-producer multi-consumer design with multiple worker threads.
    - Improves scalability and system load stability as compared to thread creation on-demand for writes.

- **Checkpoint log :**
  - Updates on all servers are logged separately

  - Logging is done **in-memory**

  - **Checkpoint** : In-memory logs are later persisted in a batched manner and primary informs all backups also to checkpoint.

  - Synchronous checkpointing between Primary and Backups needed for **crash recovery.**

APIs in Action

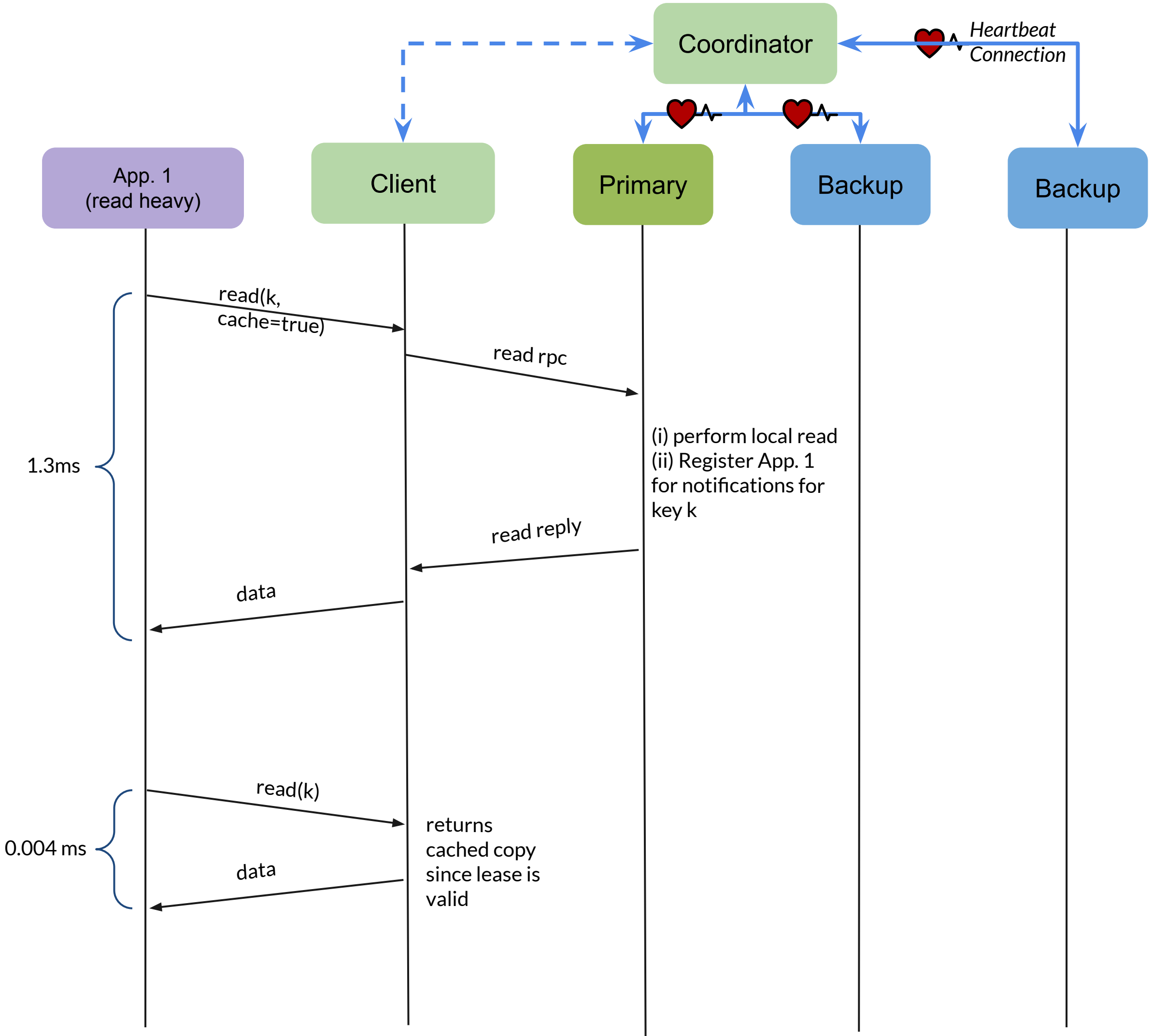
- Read
- Write (Durable Mode)
- Write (Fast-Ack Mode)

read()

Client can read from primary(optional caching) or backup.

Primary registers a client for notification if caching is requested.

Backup Server can be used for reads with eventual consistency.

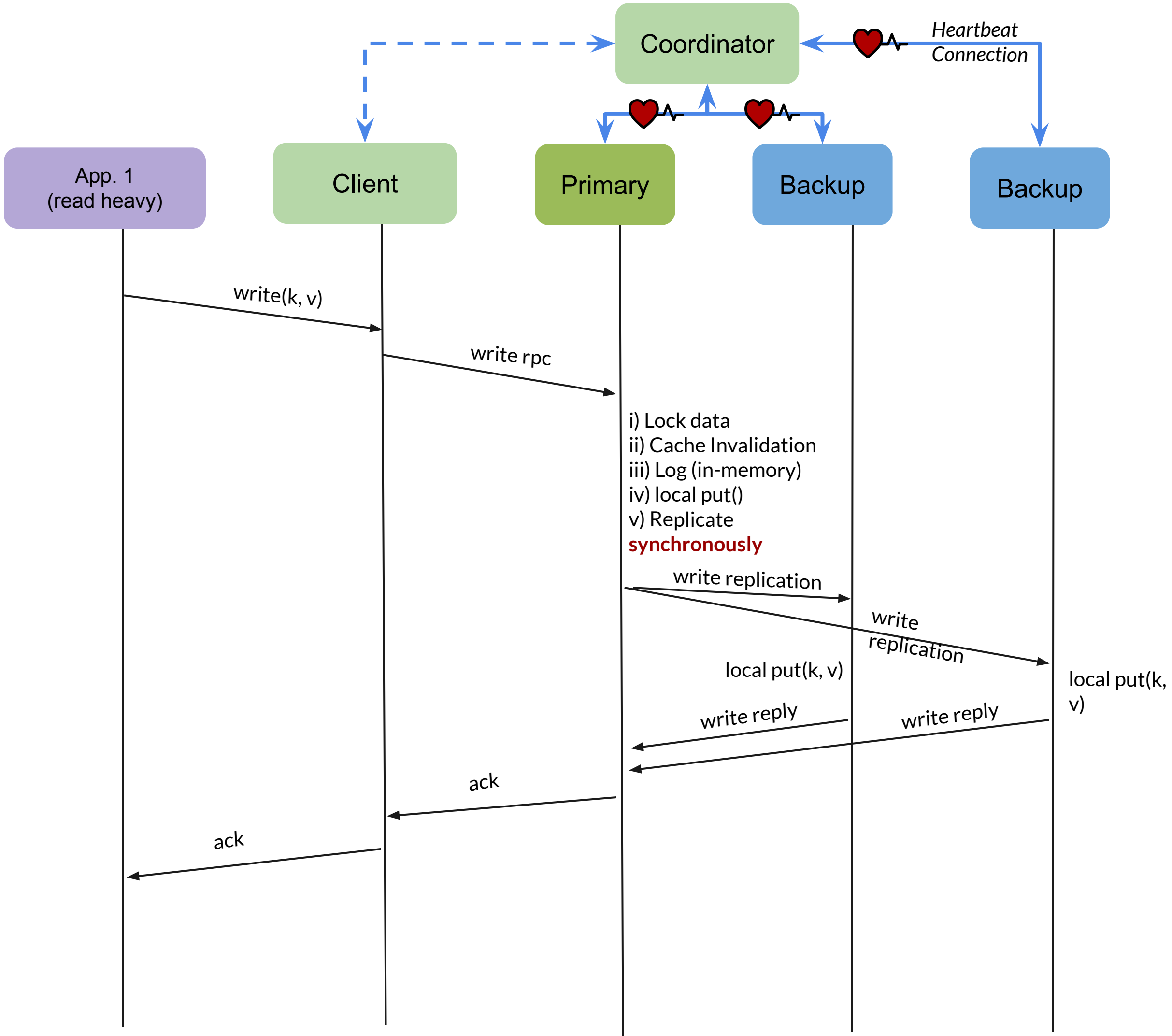


Durable write()

Only **Primary** handles write().
Steps:

- Locks data
- Cache invalidation
- logs the current data in memory for crash recovery
- perform local put()
- replicate request on backup in parallel
- **Wait for replication replies**
- send acknowledgement to client

Backup is only in listen mode for write() from primary and doesn't accept write() from client.

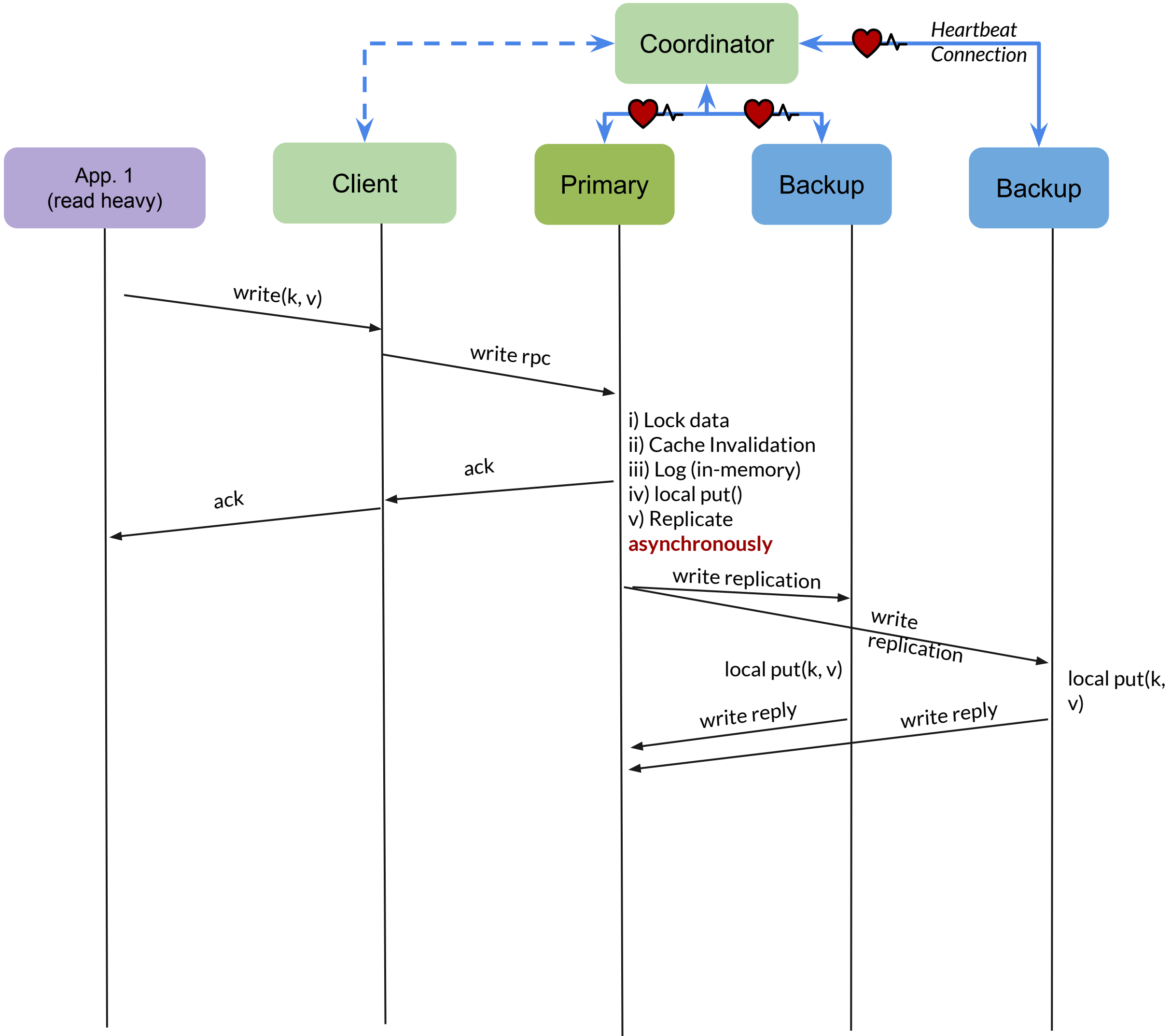


Fast Acknowledge write()

Steps:

- Locks data
- Cache invalidation
- logs the current data in memory for crash recovery
- perform local put()
- **Asynchronously** replicate request on backup in parallel
- send acknowledgement to client
- No need to wait for replication replies before ack'ing the client.

Backup is only in listen mode for write() from primary and doesn't accept write() from client.



Crash is Common, Unfortunately :(

Design - Crash

1. Node contacts coordinator first to get information about primary to sync state.

2. Primary drains current operations and blocks further.

3. It sends the missing writes (found out by comparing checkpoints) to the backup nodes with a streaming rpc call.

4. The backup node also parallely registers itself with the coordinator allowing for it to receive updates after re-integration is complete.

5. Updates are unblocked and normal operation continues after re-integration is done.

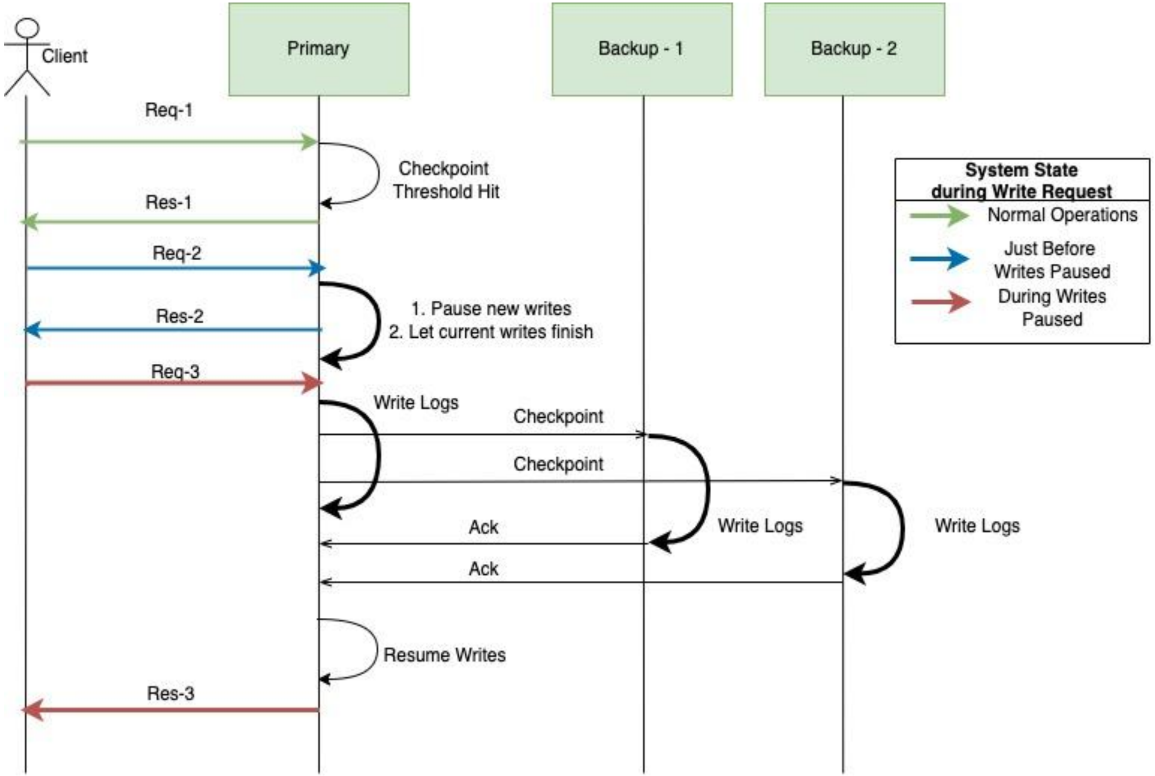
Checkpoint (CP)

Operations:

- Global write lock is acquired during CP
- New writes are suspended during CP op, and ongoing writes will continue as it is
- After CP is done on all replica, global write lock is released

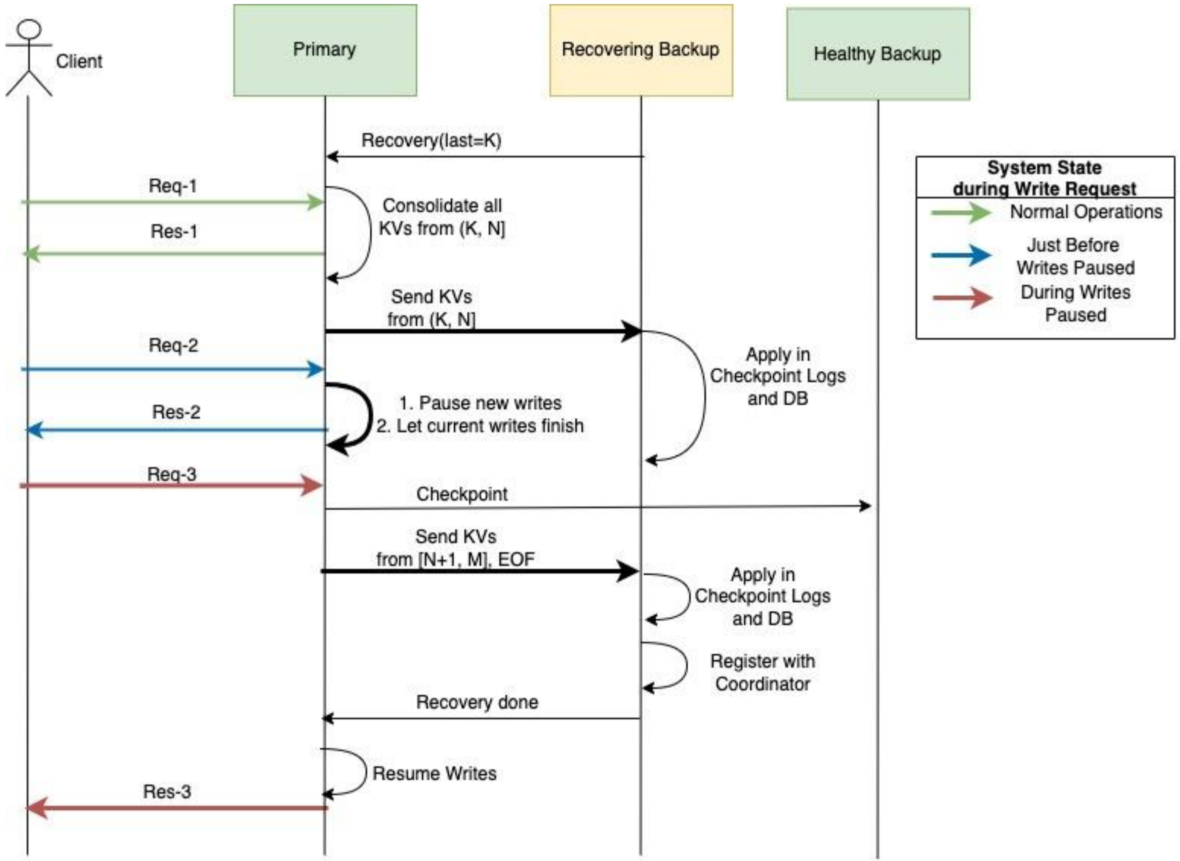
Optimizations:

- Local CP log contains just keys
- CP RPC is lightweight and does not contain any txn details



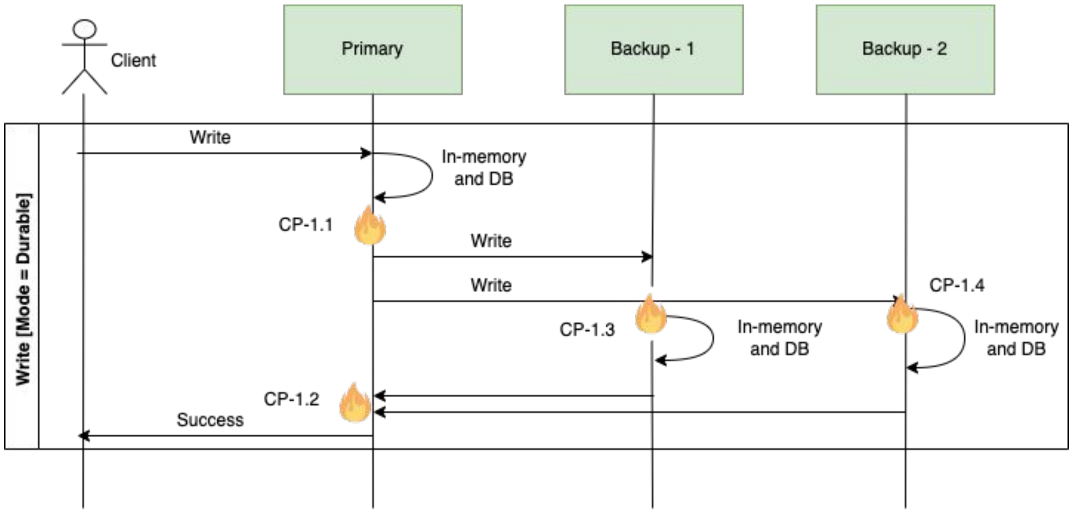
Recovery

- Server takes new requests only when it is recovered fully
- Recovering server asks Primary to get all pending writes since last CP
- **Writes will continue to function when Primary is consolidating and sending all txns**
- Same CP write lock protocol is used
- Primary makes one more CP and then later sends all txn to recovering server
- Recovering server applies those txns and registers itself
- Global write lock is released



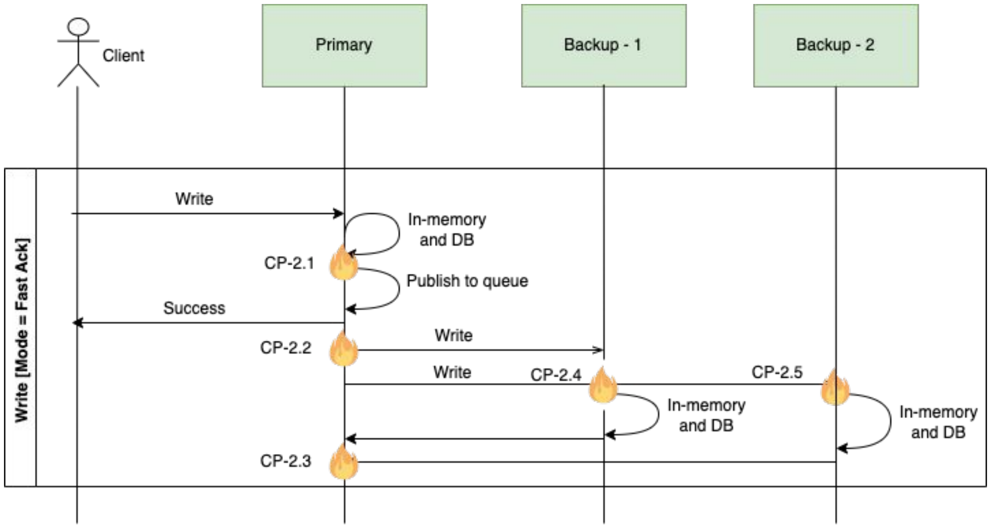
System Behaviour During Crash Points (Durable Writes)

Crash Point	Server	System	Client
1.1	No Op	New Primary, System **Consistent (in-built client retires using client-lib)**	Retry
1.2	recovery	New Primary, System Consistent	Retry
1.3	recovery	System Consistent	No Op
1.4			



System Behaviour During Crash Points (Fask-Ack Writes)

Crash Point	Server	System	Client
2.1	No Op	New Primary, System **Consistent** **(in-built client retires using client-lib)**	Retry
2.2		New Primary, **System Inconsistent**	**Write is lost**
2.3		New Primary, System Consistent	No Op
2.4	recovery	System Consistent	No Op
2.5			



System Evaluation

Experimentation and System configuration

For every experiment, each server is run on a separate node on cloudlab.

- Cloud provider: Cloudlab
- Architecture: x86_64
- CPU(s): 40
- OS: ubuntu 18 LTS

We evaluate three categories:

1. Performance
2. Scalability
3. Optimizations

1. Vanilla Performance (Get/Put throughput)

- Experimental setup
    - 1 shard - P + 2 Backup nodes
    - Caching disabled from client
    - Number of clients = 1
    - Strong consistent reads
    - Durable writes

- Observations
    - A single client reaches upto ~900 read() and 350 write() operations per second
    - Latency
        - read() = 1.12 ms
        - write() = 2.8 ms (2RTTs)
    - Baseline latency (gRPC + rocksDB)
        - read() = 1.11 ms
        - write() = 1.3 ms (1RTT)
    - **Goodput** is same as throughput with upto 2000 concurrent clients

Throughput (number of operation/sec)

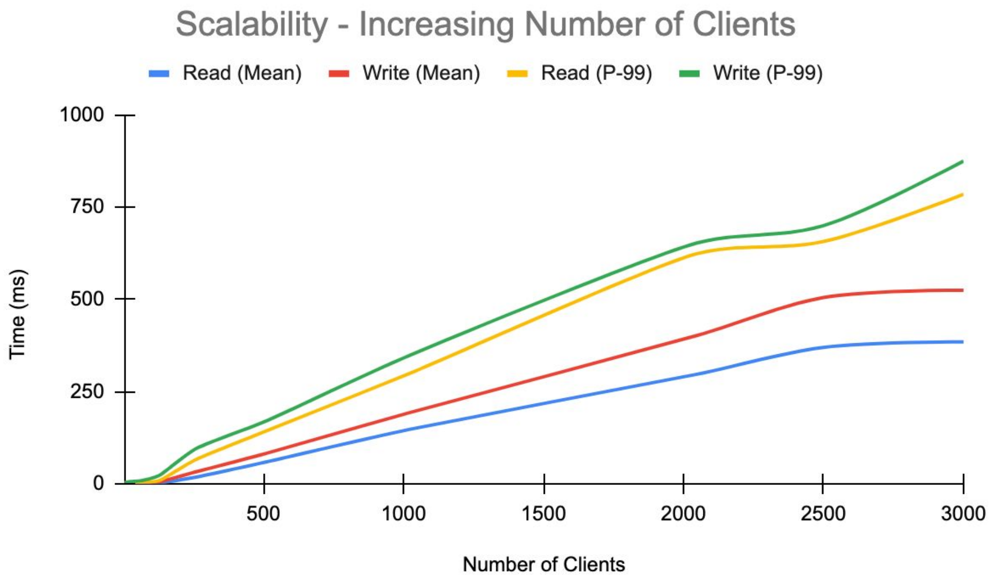
System	Read()	Write()
Baseline (w/o replication)	909	769
Our System	893	356

2. Scalability(Read/Write): Without Sharding

- Effect of increasing number of clients on performance.

- Setup
  - 1 shard - Primary + 2 Backup nodes
  - Caching disabled from client
  - Strong consistent reads
  - Durable writes

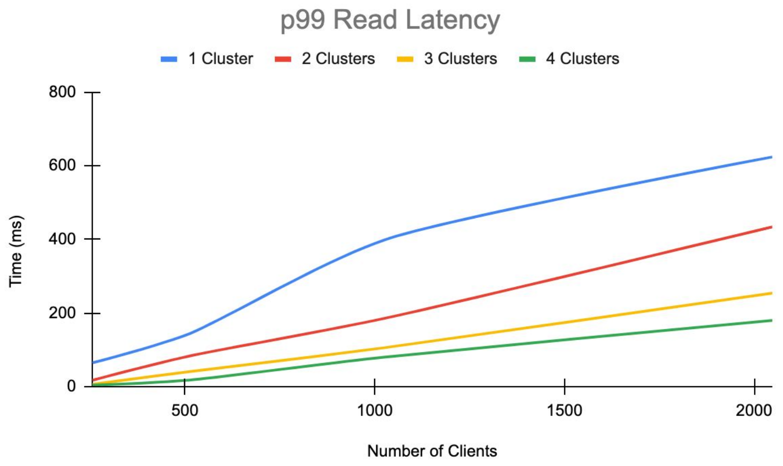
- Observations
  - The latency of operations linearly increases upto ~2500 clients.
  - p99 (capturing what most clients observe in worst case) is affected more with increasing number of clients than mean due to high system load

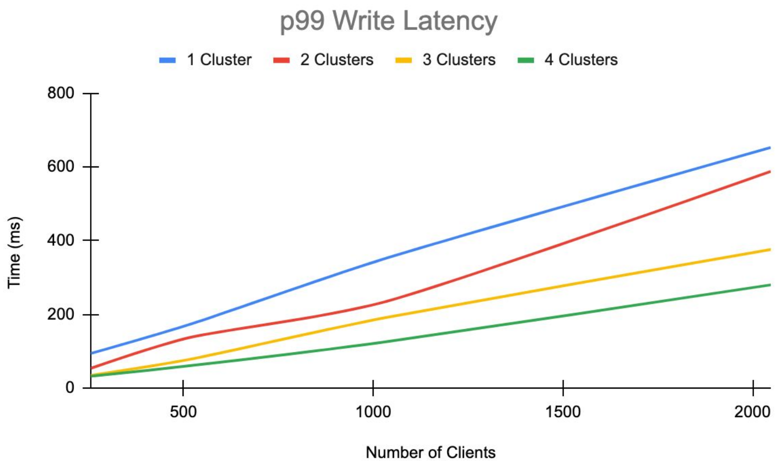


2. Scalability: With Sharding

- Setup
  - Each shard - Primary + 2 Backup nodes
  - Caching disabled from client
  - Strong consistent reads and writes

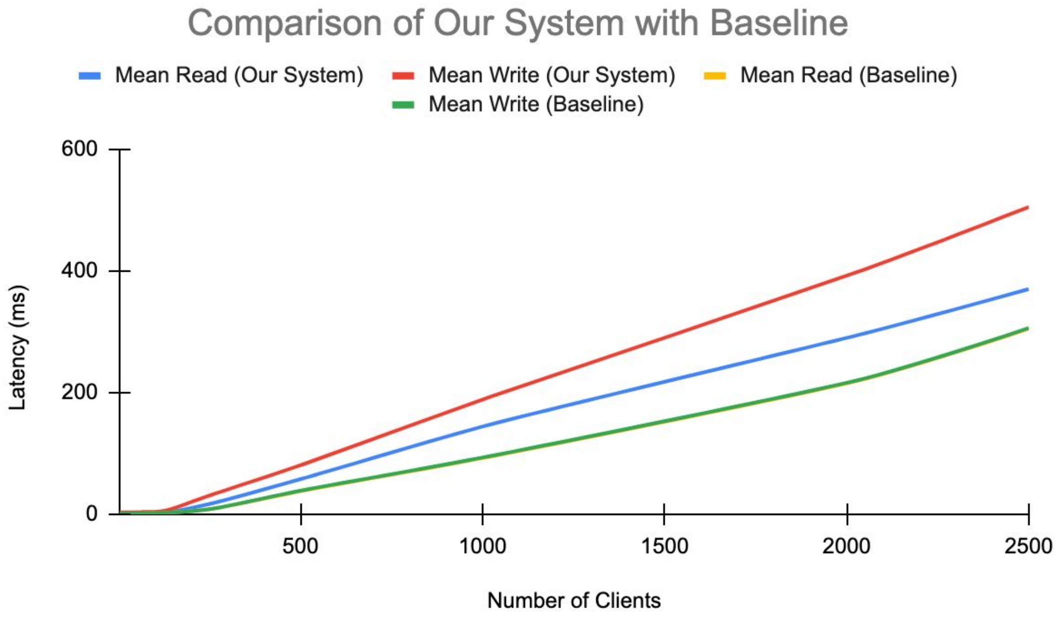
- Observations
  - The latency of operations decreases with increasing number of shards.
  - This shows better load distribution with distributing key-range.





3. Comparison with Baseline - without sharding

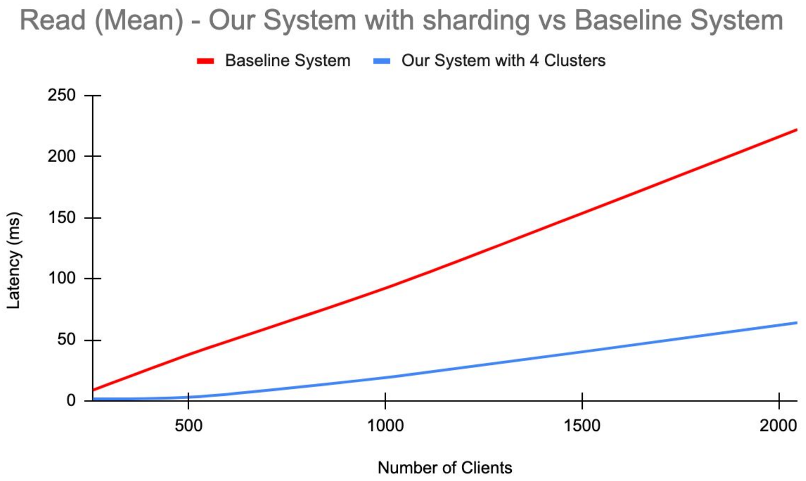
- Baseline: i.e. gRPC + vanilla RocksDB
- Our system setup
  - Each shard - Primary + 2 Backup nodes
  - Caching disabled from client
  - Strong consistent reads and writes
- Observations:
  - Read and Write time of baseline system are same due to no replication (both are 1 RTTs).
  - Write of our system is high due to replication.
  - **Read time of our system is also affected** due to server load on primary (replication task).

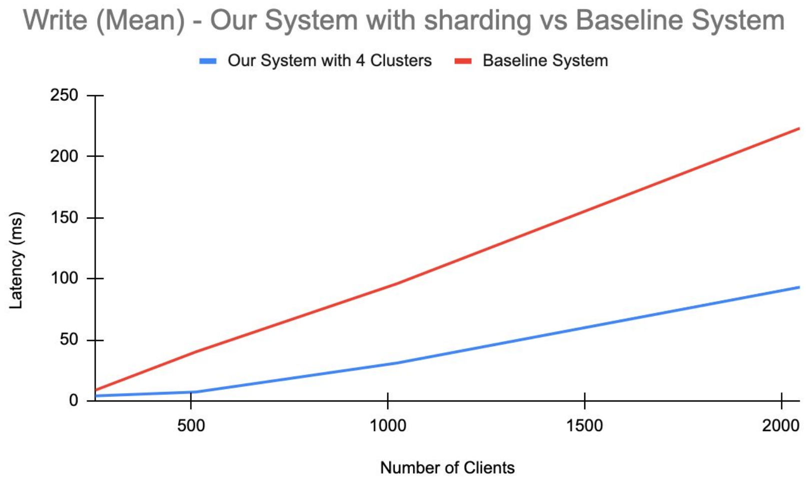


3. Comparison with Baseline - with sharding

- Setup:
  - Each shard - Primary + 2 Backup nodes
  - Caching disabled from client
  - Strong consistent reads and writes

- Observations:
  - Our system performs **better for both read and writes with sharding** even after incurring replication overhead.
  - This shows better distribution of work and therefore more **scalable** than the baseline system.

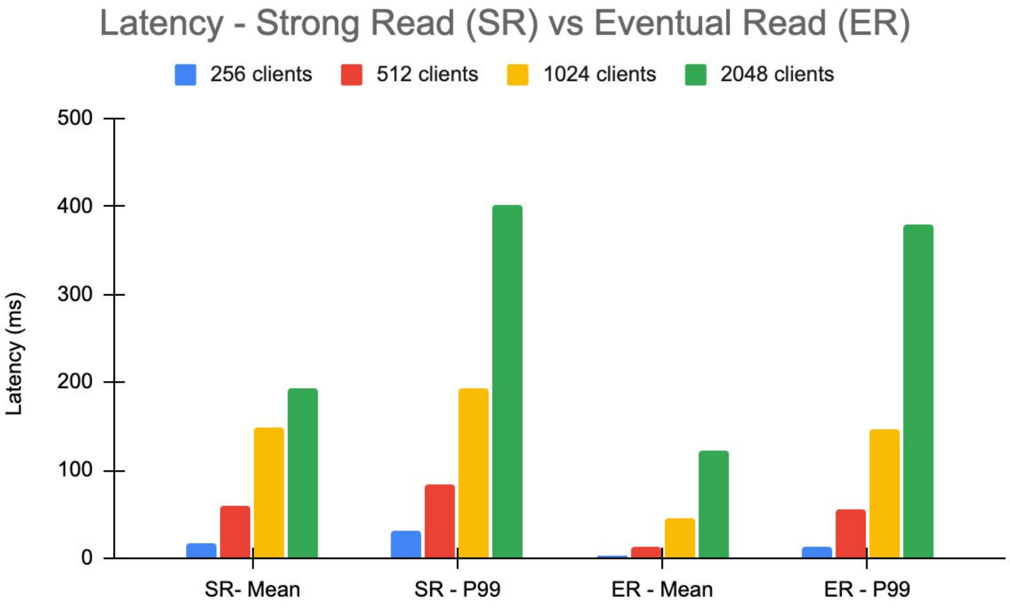




4. Optimizations:
Strong vs Eventual Consistency Reads

- Setup
  - 1 shard : Primary + 2 Backup nodes
  - Caching disabled from client.

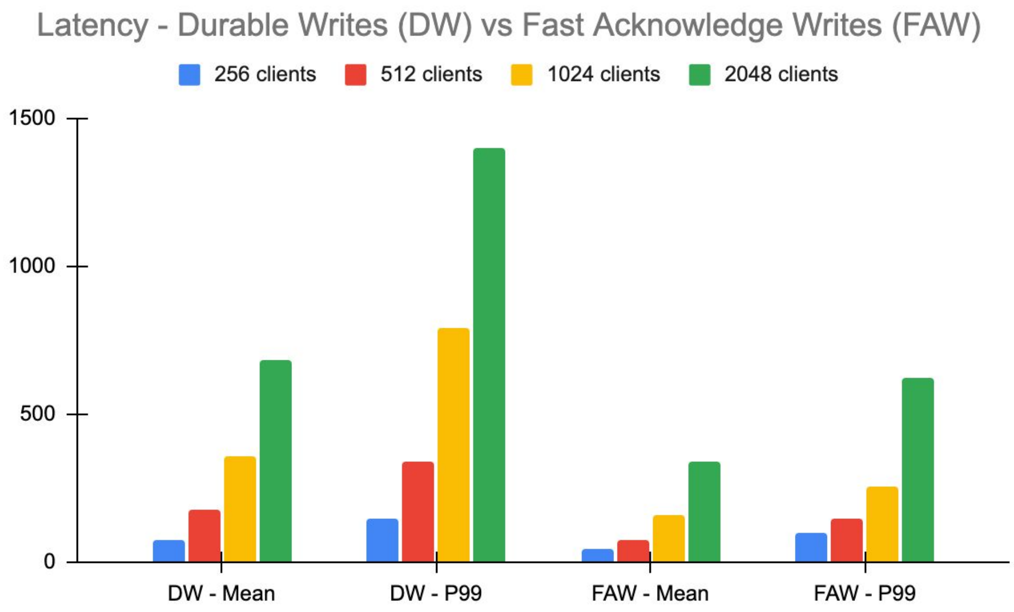
- Observations
  - SR mean is ~2x ER mean reason being **primary is the bottleneck** for strongly consistent reads
  - **With increasing clients, diff between P99 of SR & ER reduces** as number of backups are fixed and we are increasing clients - saturation point for this P/B config.
  - Eventual reads increases system throughput and **reduces latency**



5. Optimizations:
Durable vs Fast-Ack Writes

- Setup
  - 1 shard : Primary + 2 Backup nodes (caching = false)

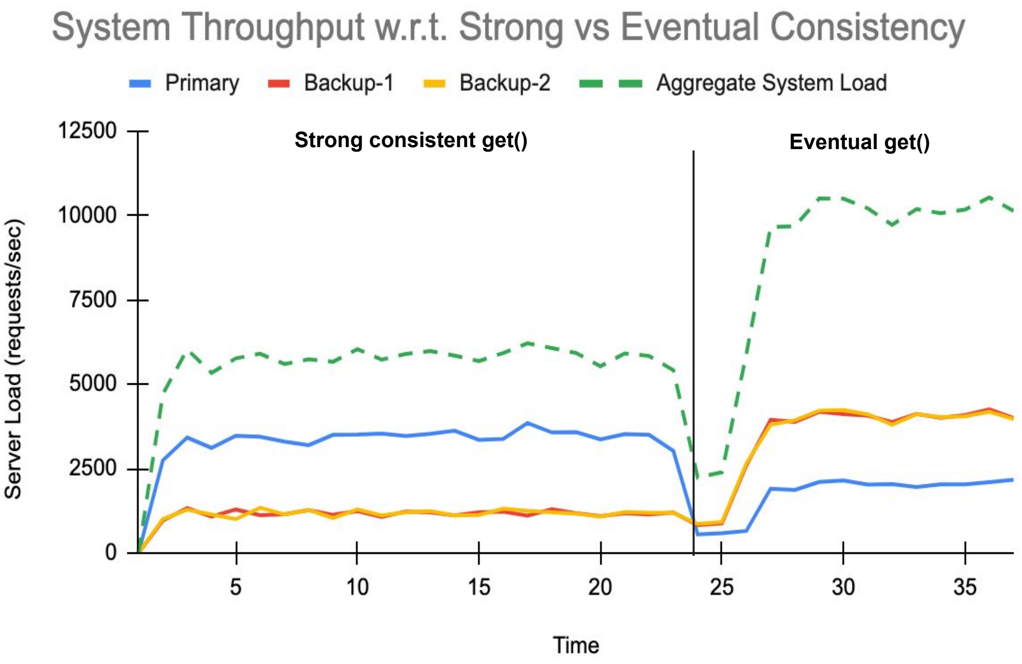
- Observations
  - Fast ack writes **scales well** with increasing number of clients as expected
  - Durable writes increases system load due to overhead of **blocking replication calls**
  - P99 in durable writes are affected the worst possibly due to high system load.



6. Server Load

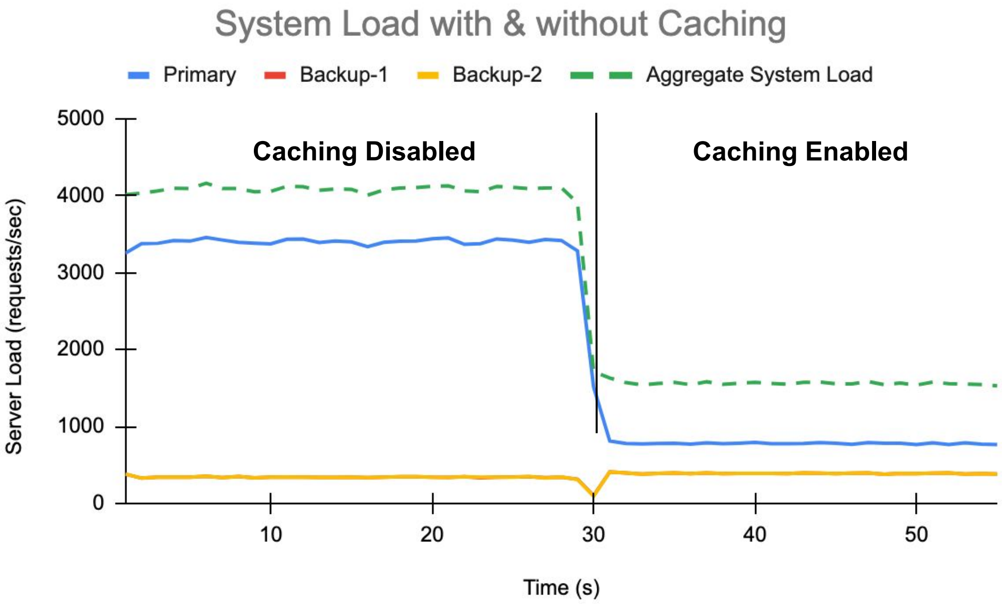
- Setup:
  - 1 Primary + 2 Backup servers
  - 500 clients performing read and writes in ratio of 1:1

- Observations:
  - **Aggregate throughput increases** with eventual consistency due to better load distribution.
  - **Time** to process same workload also **reduces** with eventual consistency reads

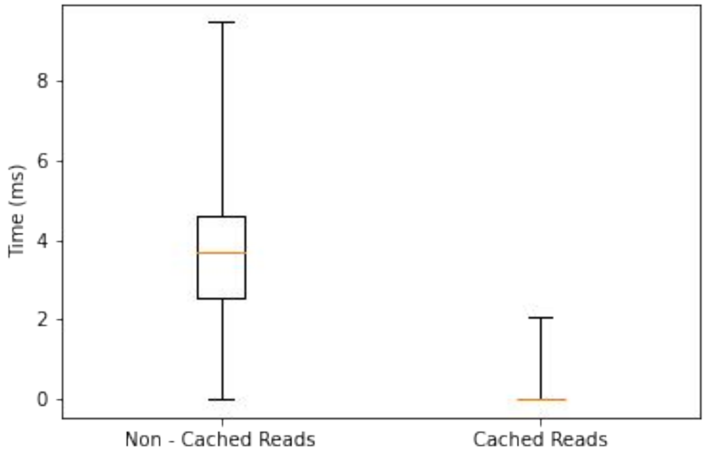


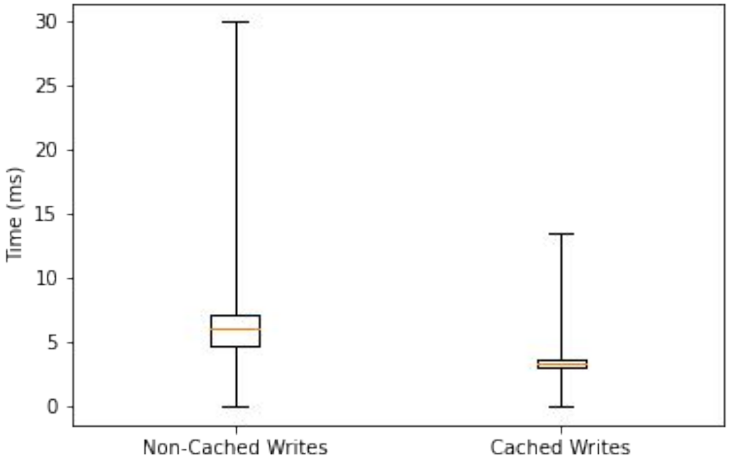
7. Optimizations:
Caching - effects at server

- Setup:
  - 1 Primary + 2 Backup servers
  - 100 clients performing read and writes in ratio of 10:1
- Observations:
  - **Aggregate** throughput **increases** with caching as there is less client-server interaction
  - Primary server **load decreases**
  - **Time to process** same **workload** reduces in caching case
  - The **Backup load** (indirectly) **increased** with caching, as Primary can server more writes => more replications per second.



8. Optimizations:
Caching-effects at client





Read:

- Due to caching, mean drastically reduces
- P99 represents cache miss
- P99 (is also less due to less server load due to caching)

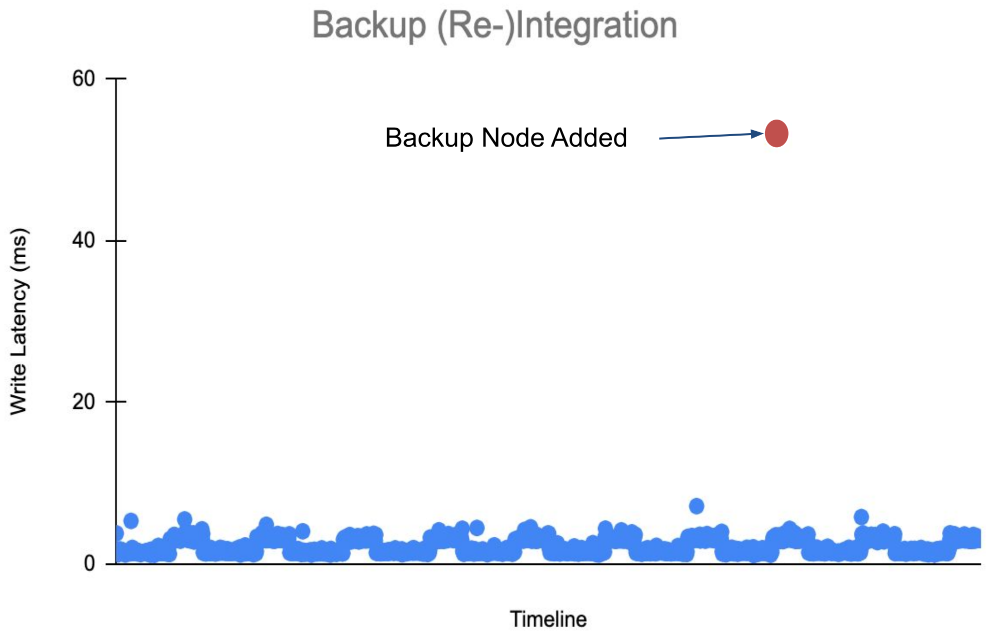
Writes:

- We expected write latency to worsen due to cache invalidation overhead
- <u>**Surprisingly**</u>, it improved. We think this is because system has less load due to cached reads being served from client.

9. Crash Recovery

- Setup
    - 1 shard: Primary + 2 Backups

- Observations - Adding 1 backup
    - **Waves** - correspond to normal operation and checkpoint sync between P-B servers .
    - The **increased write latency** (red dot) for some time is due to primary blocking updates during backup re-integration.



Consistency Protocol

write(key, data)
stdout(done)

Client Write

read(key)
? x cache_read(offset)
rpc read(key)
stdout(done)

Client Read

m_lock(key)
Checkpoint-Log (key)
N x cache invalidation rpc
[db_write(key)]
N x rpc replicate()

Primary server

m_lock(key)
Checkpoint-Log(key)
db_write(key)
m_unlock(key)

Backup server

m_unlock(key)
rpc OK

Primary server

Server Write

db_read(key)
rpc reply()

Primary server

Server Read

Consistency Protocol

rpc get_primary()

rpc sync_with_primary()

m_lock(system_state)

m_lock(write)

N x *rpc stream state diff*

m_unlock(write)

m_unlock(system_state)

rpc OK

Backup server

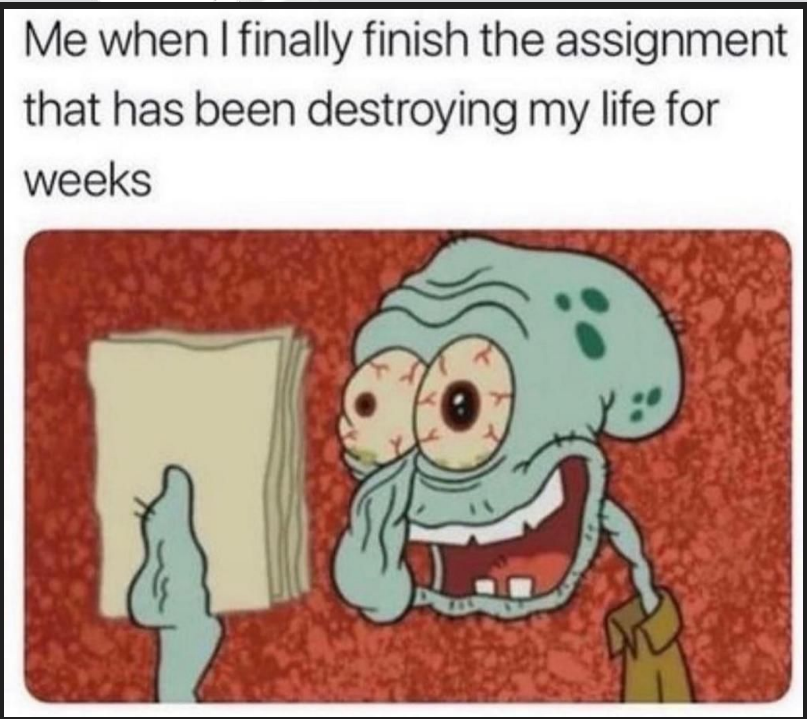
Primary server

Backup server

Recovery

Demo Time!

Thank You!


Me when I finally finish the assignment that has been destroying my life for weeks