(Distributed Rocks!)DB

Akshay Parashar Divyanshu Kumar Hemal Kumar Patel Mohit Loganathan

1 Introduction

This report presents the design and rationale of the system "(Distributed Rocks!)DB" that we have implemented in our CS 739 class.

RocksDB is a storage engine library of key-value store interface where keys and values are arbitrary byte streams. RocksDB organizes all data in sorted order and the common operations are Get(key) and Put(key, val).

The three basic constructs of RocksDB are memtable, sstfile and logfile. The memtable is an in-memory data structure - new writes are inserted into the memtable and are optionally written to the logfile (aka. Write Ahead Log(WAL)). The logfile is a sequentially-written file on storage. When the memtable fills up, it is flushed to a sstfile on storage and the corresponding logfile can be safely deleted. The data in an sstfile is sorted to facilitate easy lookup of keys.

The main disadvantages with rocksDB is that it is not distributed! The goal of our project is to use different techniques learnt in the course to make RocksDB highly available, distributed, and fault tolerant system.

To solve the above limitations of vanilla RocksDB we decided to implement the Primary-Backup design with sharding and coordination while providing strong and relaxed consistency semantics for the replicated data. There is a centralized coordinator node which maintains the system state dynamically as nodes enter and leaves the system. Crash consistency and recovery have been an important consideration in our design as we assume that clients and servers can fail anytime (both primary and secondary, but atleast one server per cluster is alive at any time - n-1 failures are tolerated).

We have also added measures to ensure the data in between the writes is not corrupted. Moreover, after primary crash, any of the potential backup servers can switch role as primary and serve future reads and writes. To improve performance further we implemented caching and ensured cache consistency using server driver invalidations. Lastly, we evaluate our implementation and present the benefits and overhead of our design choices. We emphasize on some of these design choices and their effects on final performance.

Key design: RocksDB, Primary-Backup Architecture, Sharding, Caching, Cache Consistency, Server-driven cache invalidations, Coordination, leased caches, Strong and Relaxed Consistency, Lease, Write-ahead log, Checkpointing.

2 System Design

2.1 Assumptions

- **Environmental Assumptions:** We assume that servers are fail-stop (non-byzantine faults only), stable storage (no disk failures or corruption), no rocksDB/underlying DB corruption, stable network (no packet drop), one node always up per cluster (n-1 faults at maximum per cluster/shard).

- **Nature of workload** : We assume that there can be both Read heavy and Write heavy workloads. To optimize for read heavy workload, we chose to provide optional caching at the client side to avoid unnecessary network rpc calls. For write heavy workloads, we have implemented sharding in the system to ease load on servers. Keys are distributed equally to all shard clusters.

- **Configurable consistency semantics**: We provide client configurable consistency semantics. For reads, client can choose the data to be stored in:

  - **Strongly consistent reads:** Here, the data is stored with strong consistency semantics. Even if the data is replicated, single copy view is provided to the clients.
  - **Eventually consistent reads:** Here, the data is read from any of the available backup servers. Data might be stale as the latest value is not propagated to this server yet.
  - **Durable writes:** Here, it is guaranteed that the data is replicated on Primary-Backup servers before sending acknowledgment to the client. Once ack'ed, data will not be lost.
  - **Fast-Acknowledge writes:** These provide loose durability semantics. In this case, a write is acknowledged after making the write durable on the primary. Replication to backup servers happens asynchronously in the background. Situation can arise where primary crashes before completing replication and in that case data is lost.

  The choice of which consistency to opt for falls on the client and its data requirements. A client can choose the

desired consistency level on a per-request basis. Example: social media platform can have fast acknowledgement writes for like count on a post. It does not hurt the system if the write is lost.

- **Cache Invalidation (client vs server driven)**: We chose not to go with client driven cache invalidation checks (pull based) to reduce the load on primary and the associated network traffic. We have also added lease to the caching to reduce server state and overhead during writes. Thus, server is responsible for notifying the clients in case of any write to a cached object which is currently leased.

- The system is designed with the assumption that there will be maximum of N-1 **server crashes at any point of time** (Primary or any Backup).

2.2 Centralized Coordinator

The main purpose of the coordination service is to act as a central hub for communication between servers and assignment of roles. The coordinator maintains a coherent system state dynamically as the server nodes enter and leave the system. On startup, all the servers registers to the coordinator and the coordinator is responsible to assign a cluster and role to the server. Server remembers its role and also fetches the complete system state (addresses of all the servers in the cluster and primary server identity). This system state is useful for multiple purposes like reintegration with primary, replication to backup server etc.

The coordinator maintains a complete up-to-date system state by maintaining heartbeats with all the nodes (P/B). Instead of peer-to-peer heartbeats, centralized heartbeat mechanism with a coordinator results in less number of heartbeats and it becomes easier to maintain a globally consistent view.

Clients also interact with the coordinator. A client, on startup or any time later, can contact the coordinator server to fetch the system state. Coordinator replies back with the primary and backup server addresses per key-range (shards) to be contacted by the client for its read/write operations. In case of any rpc failure/timeout, client refreshes the system state from the coordinator and retries on the newly fetched server addresses.

One might argue that coordinator can be a single point of failure as the system will be unavailable in case of coordinator crash. We can have replication of coordinator service as well but that is not yet implemented in the project as there are many well know coordination service available to use. We could have used Zookeeper in our project but we wanted to build the coordination service from scratch. We believe designing for fault tolerance in coordination service will have similar challenges as our previous project on Replicated Block Store.

2.3 Primary Backup design

We choose the Primary Backup system design where Primary exposes read() and write() APIs to the client. We also allow read() from the backup in case of eventual consistency reads. Reads with strong consistency are redirected to primary and reads with eventual consistency can be performed at any of the backup servers. Writes() are always served by the primary server. This functionality of eventual reads gives an option to client to get the results faster and improves the load balancing of the system by removing the problem of primary being the bottleneck.

Figure 1 shows the system design of the distributed RocksDB system. There is a centarilized coordinator and each shard has a primary-backup design and keys are distributed equally among these shards to improve the scalability of the system.

2.4 Client side leased caching with server callbacks

We have provided optional caching to clients. Clients can specify on a read() upcall that they want to cache this data. The cache consistency is maintained by the server promising to notify the client of invalidation of the cache whenever a write call comes on the same object. Further, to reduce state on server and latency, we choose to make the caches leased. Therefore, a cache automatically expires after a configurable amount of time.

The flow goes like this : Whenever any application requests a cached read, it is first checked in local storage and if the cache exists within the lease timing. If it is not present, the file is requested from the server with caching option enabled. The server adds the client to the read list of that key. In case of any future writes, server checks if there are any clients within the lease timings for that key. It then invalidates the client caches via callbacks. This leased caching with server callbacks improves system performance and reduces server state.

2.5 Last writer wins

Last writer wins (rather last write being processed on server wins). Server enables a lock on key being written to ensure coherent view for the system. Thus, if there are multiple clients writing to the same key, the requests are guarded by locks and the last writer wins in this case. We chose this design as opposed to making the writes be submitted in a queue and be executed serially as when writes are non-conflicting (to different keys), the performance is improved drastically while keeping with the strong consistency requirement. Moreover, the order of write operations are decided by the primary server as all the writes (durable and fast acknowledgement) are handled by primary server.

2.6 Parallel Replication and Multi-Producer Multi-Consumer Queue

We initially implemented the replication to backup servers on a write call serially. This design was not scalable with increasing number of servers and clients. We then changed this to parallel replication and it gave much better results than before since now the latency of a write was not limited by the sum of latencies of replication to each server but instead it's limited by the slowest of it all.

Even after parallelizing the replication, we realized that the way we have implemented it could lead to system instability and scope for crashes. We were creating new thread for each replication. Thus, say in case of 3 backup servers, a write will result in 3 additional threads for replicating the data on the 3 servers. When serving many clients with multiple concurrent write requests this results in an unbounded growth in the number of threads. This could have resulted in unstable system and sudden crashes.

We therefore implemented a multi-producer multi-consumer queue on primary where replication requests can be submitted in this queue and a fixed number of worker threads replicates on the backup nodes by picking a request one at a time from the queue. This implementation presented some unique challenges for a stable implementation, and we enjoyed the learnings out of it.

2.7 Eventual reads from backup

Clients are exposed to optional feature of eventual consistency reads. These reads are redirected to one of the backup servers which are less loaded than the primary and therefore should result in lesser latency to serve the read request. Therefore, a client is incentivized to do this kind of read operation for faster results and if the requirement for the data to be latest is not strict. This helps to reduce the network load on Primary server (strongly consistent reads). All of the backup servers converge to a consistent view of the system at some point of time in future.

2.8 Read()

Client can read from both Primary(strong consistent reads) and any of Backups (eventual consistent reads). Eventually consistent reads enables efficient load balancing among both the servers and better use of replication. Primary server has the responsibility to do cache invalidation callbacks to notify client caches for any cache invalidation. We designed the backup server read requests to not handle caching requests to avoid callbacks from backups - to keep it simple. Figure 2 shows non-cached client reads from backup.

The strong consistent read() API starts with client requesting data block from the client library. The client first checks if the data is cached locally and within the lease limits. If

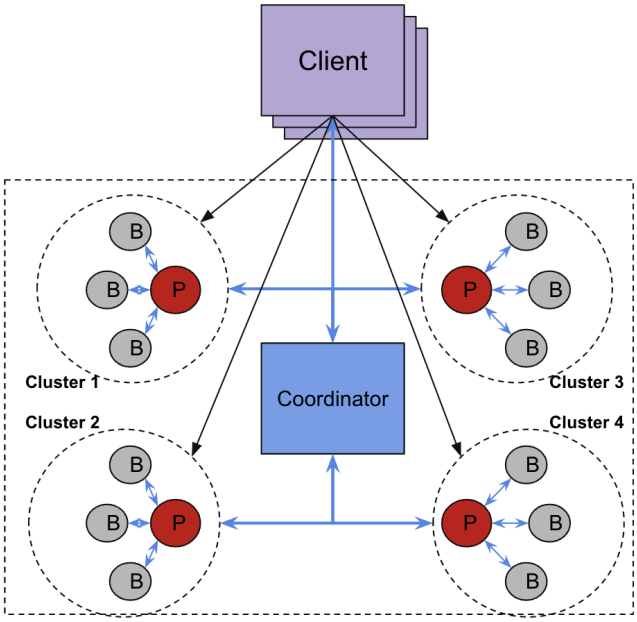

Figure 1: System Design

yes, it is returned to the application. Else, it is forwarded to the Primary server. The primary server reads the data block from local disk and registers the client for callback notifications. These notifications are used to notify the client of any change in the data block that was cached by it. The eventually consistent read() API for backup server does not provide the callback functionality. It is the job of the client to not cache these requests and use read-only mode.

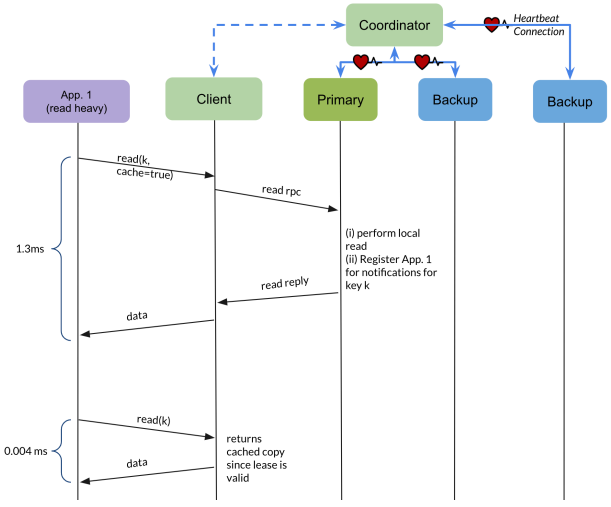

Figure 2: Flow of read() call

2.9 Write()

Client can only write to Primary server and not to the Backup server. In normal scenarios, write request involves 2 rpc calls - One from client to primary and the other from primary to backup server for replication.

In case of durable writes, the primary server does not acknowledge the client before replication is completed. Thus, once the client receives acknowledgement for its writes, it can be sure that data is stored durably on distributed servers. In case of fast acknowledgement writes, the primary server acknowledges the client before replication. It performs replication asynchronously on a parallel thread while sending ack to the client. Note that there might be cases where these writes can be lost forever (primary server crashing before replication is performed while the client is acknowledged of write success). This is acceptable since the client is aware of the loose durability semantics that comes with fast-acknowledged writes. Figure 3 shows the flow of Durable write and Figure 4 shows the flow of Fast-Acknowledgment write.

The write operation starts with client sending write rpc request to the Primary server. The primary server first locks the key so that there should be no concurrent writes to the same keys. Secondly, it notifies all the clients (if any) registered to this particular data block for cache invalidation since the data is now modified. The primary then writes the data locally (in local rocksDB instance). The next step checks availability of backup servers. If any backup server is online, it sends a write rpc request to the backup server (parallely). The primary also logs the write request of the key in the checkpointing log. This will be used later to be sent to the backup server in future reintegration phase. Once all the above steps are performed, lock is released from the current key and finally an ACK to the client is sent signifying that the write is successful at the primary. In absence of this ACK, the client retries the operation with exponential backoff timeout mechanism.

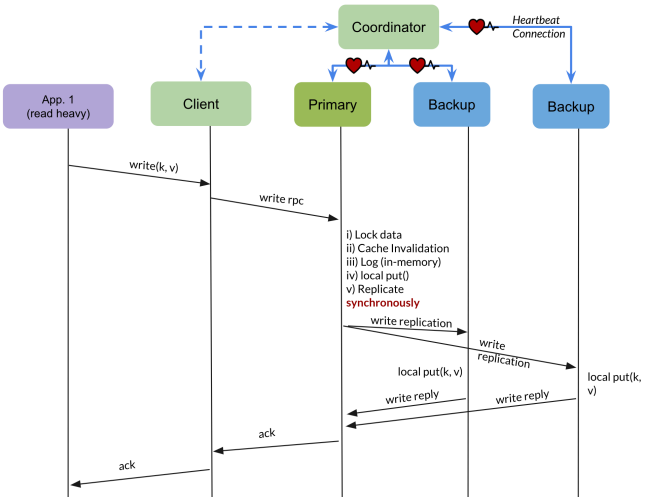

Figure 3: Flow of durable write() call

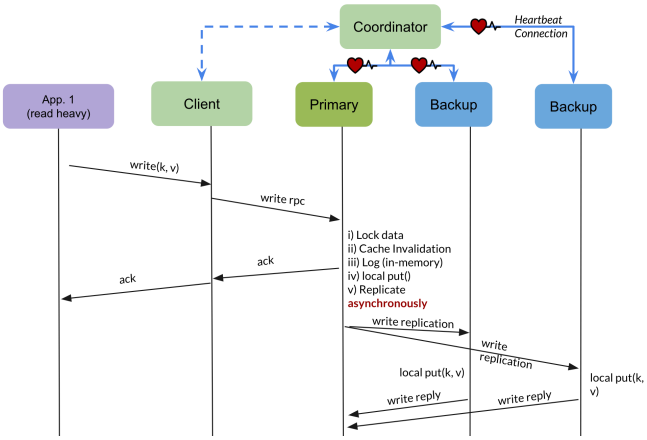

Figure 4: Flow of Fast-Acknowledgement write() call

2.10 Heartbeat

Heartbeats are used to detect failures of nodes in the system. For implementing heartbeats we had 2 options - peer-to-peer heartbeats vs centralized heartbeats. In peer-to-peer heartbeats, each primary-backup pair will have a heartbeat between them. In contrast, centralized heartbeats use a central server to maintain heartbeats with all the nodes. We chose centralized heartbeats with coordinator as the central entity maintaining heartbeats with all the nodes in the system.

When a node joins the system and the coordinator assigns it a role (Primary or Backup), the coordinator also parallely establishes a heartbeat with the node before making the node public to the rest of the system. If a node fails, this heartbeat is broken and the coordinator immediately notices this change and updates it's system state. On the next updateSystemState() call with any server or client node, the coordinator sends this change along and the respective node notices the change and updates their system accordingly. In case of a primary crash, the coordinator also elects a new primary. When the newly elected primary node, serving as backup, receives this change, it immediately switched to the role of primary and starts accepting read and write requests.

In case of rpc timeouts at client, the client requests to get an updated system state from the coordinator. Therefore, if any node crashes while serving a request from any client it only times-out once and on the next updateSystemState() call it gets the updated addresses of each role and therefore the requests succeeds at the second retry.

As for implementation, the heartbeat connection is managed on a separate thread and does not interfere with the normal operation. We create a gRPC streaming connection and keep it open. In case of any node crash, the connection is

broken and the coordinator immediately gets to know about the crash.

2.11 Crash consistency

Our implementation takes inspiration from numerous protocols like **NFS-v4 for server driven callbacks, chubby (leases) and two phase commits**. Figure 5 and Figure 6 demonstrates that our read/write protocols are crash consistent.

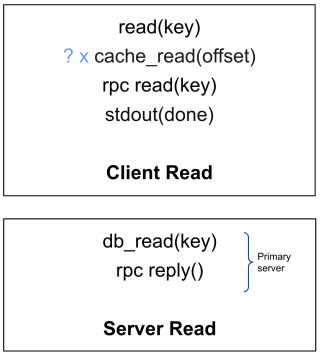

Figure 5: Crash consistent update protocol for reads

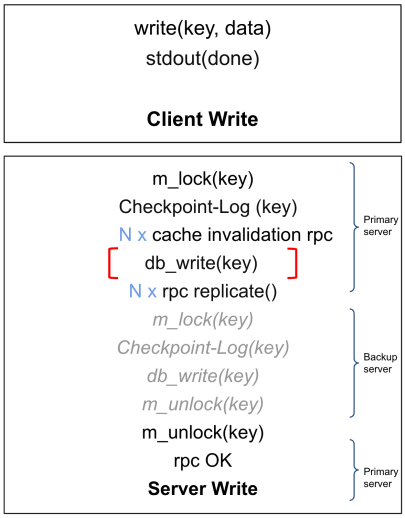

Figure 6: Crash consistent update protocol for writes

For crashes, our system is mostly crash consistent except in the case of Fast-acknowledged writes. For Fast-Acknowledged writes(FAW), the client is aware about the loose durability consistency semantics and therefore the loss of data written using Fast-Acknowledge consistency in some crash scenarios is not unacceptable. The data is only lost in case of FAW when the primary crashes after sending ack of write to client and before replicating the data on backups.

In all other scenarios, the data is not lost. It is either made durable without ack or not made durable without ack. In both the cases since the ack is not sent, the client is configured to retry the write operation on the updated primary node and therefore data is not lost. The data is also not corrupted and the whole write operation can be considered to be atomic as in the absence of ack, the client library will retry the operation.

2.12 Crash Recovery

We placed a strong emphasis to make the system fault tolerant and recover quickly after failures.

2.12.1 Reintegration Phase

Our system uses checkpoints to recover a crashed node or bring a new node upto speed. The normal operation of checkpointing is explained in Figure 7.

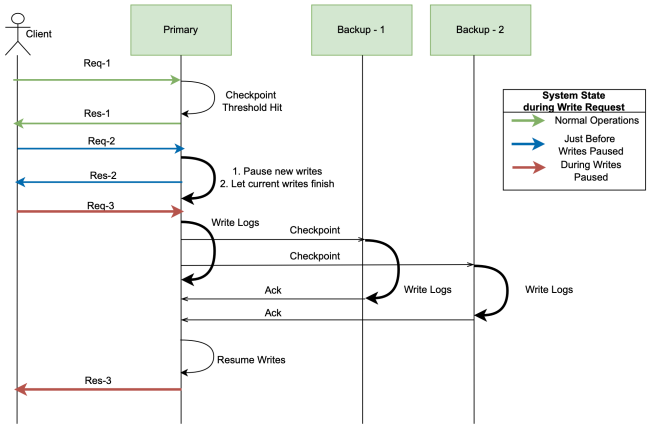

Figure 7: Flow of Checkpoint operation

Reintegration phase starts on server startup. After a server starts, it first contacts the coordinator to get the system state from which it primarily needs the primary address. If there is no primary, then the server gets assigned as the primary node and can assume no recovery is needed as it's the first node being configured in the cluster. In case there is a primary, the server gets assigned the role of backup. It then contacts the primary with a streaming rpc_recover() call. In this call, it passes the last checkpoint durably stored in the server's local storage.

The primary on receving the rpc_recover() call checks the checkpoint log number with it's own latest checkpoint. If there is a difference in the checkpoints, it accumulates all

the key-value pairs written from all the checkpoints since the recovered node's last checkpoint and sends it in a batched mode to the new backup node.

Till now, the write update operations were not blocked and the sync with backup was going parallely but we still need to pass the writes which are in-memory and not yet checkpointed. Therefore, the primary blocks any new writes from happening but completes all ongoing writes. After all writes are completed, it then does another checkpoint to flush the in-memory state of the primary to a checkpoint.

The primary then compares to see how many checkpoints have been written since the time it prepared and sent the batched writes to the new backup node (In most cases, it's the most recently (forcely) written checkpoint but in some cases of large number of missed writes, there could be multiple checkpoints written parallely when the primary was sending the old writes to the new backup node). It then again batches all the key-value pairs corresponding to the keys in these missed checkpoints and sends them again in batched mode to the new backup node.

The primary then does a final "RecoveryReply-Type::TXNS_DONE" reply on this streaming rpc call to signal to the backup node that all missed updates have been sent and now it should expect to receive new replicated write requests. The primary then continues the blocked write requests and continues operating like normally. It's system state has also changed with the addition of this new backup node and now it replicates each write request to this node as well.

As represented in Figure 8 , Primary sends all the logged writes to backup server during this phase. It is important to block writes on primary for a short duration of time, because we have to transition (w.r.t. to replication) to the new system view where there is a new backup node while also making sure all existing write operations are also relayed.

Some optimizations we did in the recovery phase are covered here for completeness's sake. is to only send a set of updated keys and not all the write update requests. For ex, if a key 15 was written 10 times when the new backup node didn't exits, it is only send the value of key 15 once and not 10 times.

2.12.2 Backup Server Crash

In case of backup server crash, it is unavailable for eventually consistent reads and the replicated writes from the primary server. The coordinator notices the broken heartbeat with this crashed node and updates its system state which is replicated in the next updateSystemState() call with each node. Whenever the backup node is started again, it syncs its last known checkpoint with the coordinator with the reintegration process described above.

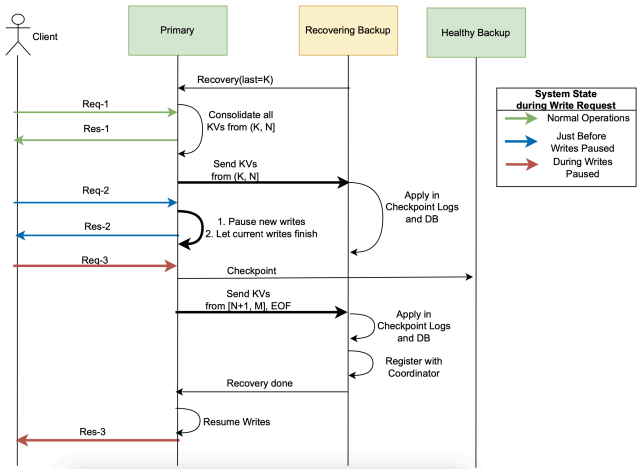

Figure 8: Reintegration Process

2.12.3 Primary Crash

Primary server can crash during two different scenarios of read() and write() API calls. These are explained in detailed as below:

A primary can crash at any point in the progress of an operation and our system handles it. Coordinator notices this crash and elects a new primary and updates its system state. A client and all backup nodes gets the identity of the new primary in the next updateSystemState() broadcast. In-progress read and write requests to the crashed node eventually time-out at the client side and it refreshes its system state from the coordinator. Normal operation continues after this and system only notices a minor hiccup due to the primary crash. No errors are propogated to the application since the client library automatically handles the time-out of rpc with a retry with the new primary.

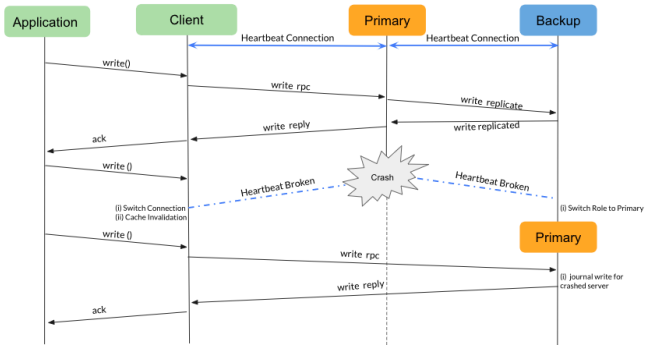

Figure 9: Primary server crash during write API

3 Performance Evaluation and Testing

3.1 Testing Methodology

In this section, we describe the testing setup and methodology in detail. For each different test case, the method of proof of correctness is different. We will go through them one by one

- **Latency measurement**: We implemented helper functions to print mean, min, max, median and p99-p100 latency of read/write operations performed by the client. Each client library spawns multiple applications on separate thread and at the end, it aggregates all of these numbers and prints the above statistics. For more reference, we can refer to client.cc part of the code.
- **Data Consistency (during and after crashes)**: We tested the data consistency in face of failures by performing the following steps:
  - The client first writes a (key,value) pair in normal case without failure to P/B configuration. It then performs read on the same key from both primary and backup servers () and verifies that the data is same.
  - After this, we kill one of the backup nodes and perform above read-write operations on the same key while backup was offline.
  - The backup is now restarted and it automatically performs reintegrate routine with the primary and fetches the unseen updates.
  - Now, the client again performs a read from the recovered backup node for the same key and the client verifies if the data is same as it wrote when the node was online

  The demo of above methodology is demonstrated in detail in the video.
- **Data corruption**: RocksDB by default provides the data corruption semantics via configurable checksum feature in its db. We felt there was no need to perform checksum on our end as it is already provided by most of the underlying db over which our method can be plugged into.

3.2 Experimental Setup

We used the CloudLab machines to measure the performance of our distributed file system. System Configuration:

- *Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz*
  Memory : 164941188 kB
  CPU MHz: 1197.416,
  CPU max MHz: 3300.0000,
  CPU min MHz: 1200.0000,
  Virtualization: VT-x,
  L1d cache: 32K,
  L1i cache: 32K,
  L2 cache: 256K,
  L3 cache: 25600K

We first present the performance numbers of vanilla distributed rocksDB (i.e. without optimizations like caching enabled) implementation and compare it with non-distributed rocksDB. In case of baseline non-distributed rocksDB, there is only one node with rocksDB executing on the server with clients interacting via gRPC. We build upon the optimizations (as explained in system design section) added and their improvements:

Throughput (number of operation/sec)

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

Figure 10: Baseline vs Proposed throughput

3.3 Vanilla Performance and baseline comparison

First, we represent the read and write latency of vanilla Primary Backup implementation of the rocksDB system as shown in Figure 10.

Setup: we used 1 shard of 1 primary and 2 backup servers with caching disabled in the client side. The client performs strongly consistent reads and durable writes to the system.

Observations: We are able to achieve read latency of 1.12 ms and write latency of 2.8 ms. The baseline latency of non-distributed rocksDB has read latency of 1.11 ms and write latency of 1.3ms. We also observe that the goodput of the system is same as throughput for upto 2000 clients after which there are cases when the read/write rpc calls might fail.

3.4 Scalability

In this section, we first showcase how the system scales for increasing number of clients. The main aim of this experiment is to test the system limits for parallel incoming client requests by the server and compare our system with that of baseline non-distributed RocksDB.

Setup: Each shard/cluster consists of 1 Primary and 2 Backup nodes. Caching is disabled from the client side and client performs a mix of strongly consistent reads and writes to the server.

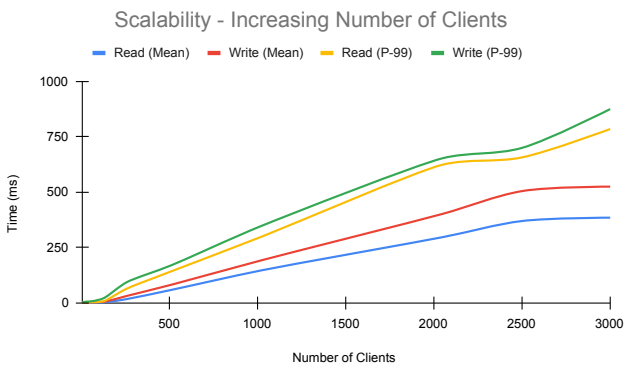

Figure 11: Scalability: Increasing number of clients

- **Increasing clients:** Figure 11 showcases the trend of our system with increasing number of client requests. Each client performs 100 write and 200 read operations on random keys and the number of parallel threads (applications) are controlled via application logic.

  Observations: We notice that the latency of operations increases linearly up to 2500 clients after which the system starts to saturate as the p99 latency starts to shoot up. Potential reason for this observation is that server is facing contention from the incoming client requests. We also observe that p99 (capturing what most clients observe in worst case latency) is affected more with the increasing number of clients than mean due to high system load.

- **Increasing clusters:** Figure 12 and Figure 13 showcases the effect of increasing the number of clusters on read and write performance. We chose to show p99 latency as it is a better than mean latency and production level systems measure p99 for stress testing of their systems.

  Observations: The latency of operations decreases with the increase in number of clusters. This is expected since the load is balanced among these shards/clusters and there is better distribution of load among the servers.

3.5 Overall Comparison: baseline vs proposed system

To find out the overall comparison of the vanilla baseline non-distributed implementation vs our system, we perform the above scalability experiment on both baseline and proposed system. Aim here is to find out what are the downsides of having a fault tolerant and distributed rocksDB system on latency. We notice some interesting observations as presented in Figure 14, Figure 15 and Figure 16.

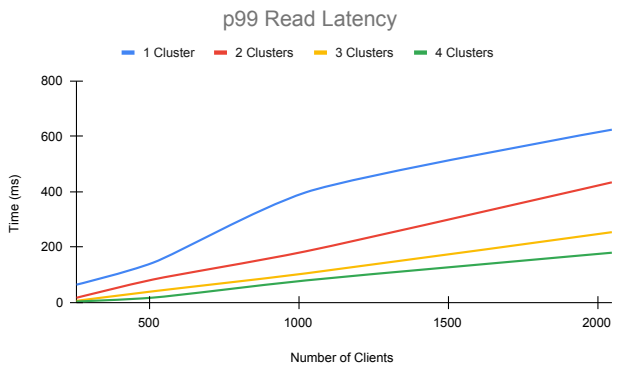

Figure 12: Scalability: Increasing number of Shards/clusters

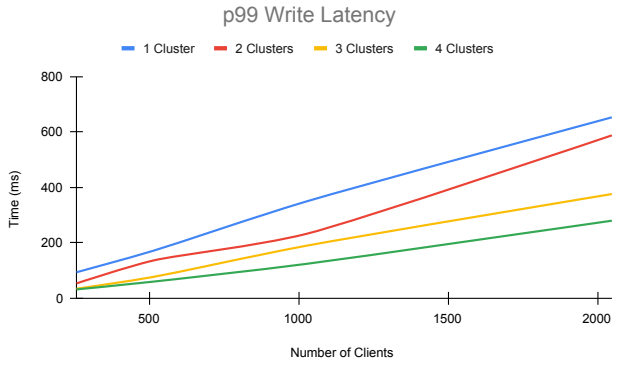

Figure 13: Scalability: Increasing number of Shards/clusters

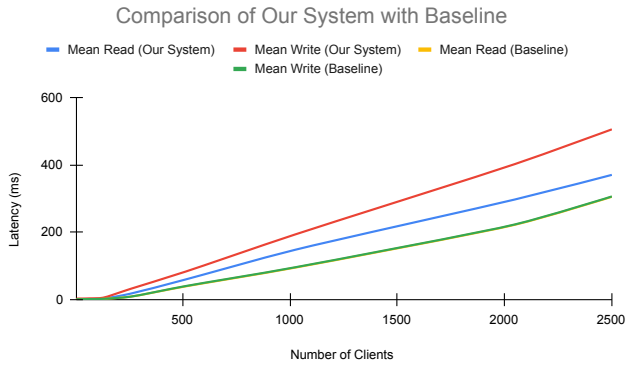

Figure 14: Comparison of Proposed (without sharding) vs baseline

- Figure 14 showcases experimentation when we compare the baseline numbers with a single cluster distributed

rocksDB implementation.

Observations: Firstly, we notice that read and write times of the baseline system (green and yellow color lines) are same. This is due to the fact that both of these operations require only 1 RTT (from client to the server) and involves no replication (as baseline system is having no replication). Secondly, we notice that the write latency of our system is more than the baseline system. This is expected as replication comes with the cost of increased write latency. Thirdly, we surprisingly notice that the read time of our proposed solution is higher than the baseline. We believe, this is due to the fact that the primary is having high load in the proposed system with the added overhead of replication task. This indirectly increases the read latency (high contention due to increased number of replication threads on primary).

- Figure 15 and Figure 16 showcases experimentation when we compare the baseline numbers with a multi-cluster distributed rocksDB implementation. We see the latency of increasing the number of shards and compare it with the baseline implementation of rocksDB.

  Observations: We notice that our system with sharding performs better than baseline non-distributed RocksDB for both read and writes. This showcases the better distribution of load among servers and hence more scalability than the baseline system. This experimentation proves that we can achieve fault tolerance and distributed flavour of RocksDB by efficiently distributing load over different shards and having better latency at the same time.

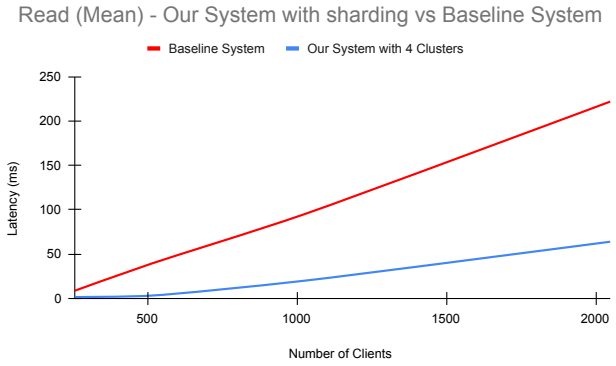

Figure 15: Comparison of Proposed (with sharding) vs baseline

4 Optimizations

In this section, we present and evaluate the effects of the optimizations we implemented on the system (throughput and latency). These optimizations follow the design assumptions and reasoning as explained in the assumptions section.

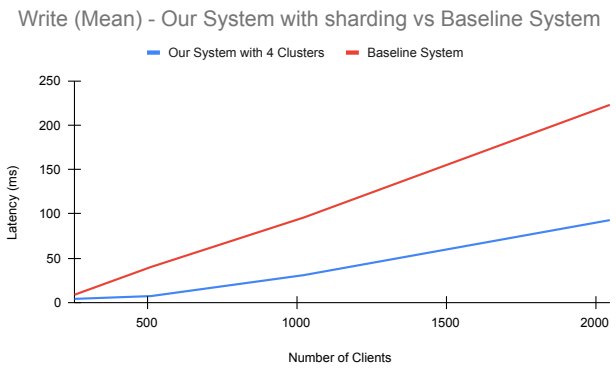

Figure 16: Comparison of Proposed (with sharding) vs baseline

4.1 Durable vs Fast Acknowledgement Writes

Setup: System consists of 1 shard with 1 primary and 2 backup servers. Caching is disabled from the client side and the writes are performed in both durable and fast-ack mode. Aim of this experiment is to see the latency benefits of both of these modes of write() API.

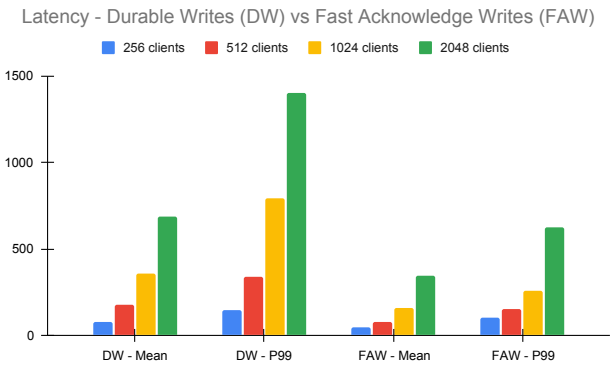

Figure 17: Latency: Durable vs Fast-Acknowledgement writes

Observations: In Figure 17 We see that fast-ack writes scales well with the increasing number of clients as expected. This is due to the fact that the server state is reduced (as we are not blocking the client writes on the server during replication). Clients are ack'ed as soon as the primary server performs local write and replication happens parallely in background. On the other hand durable writes increase system load due to overhead of blocking replication calls. We also notice that the p99 of durable writes is affected the worst possibly due the high system load.

4.2 Strongly vs Eventual Consistent Reads

Setup: System consists of 1 shard with 1 primary and 2 backup servers. Caching is disabled from the client side and the writes are performed in both durable and fast-ack mode. Aim of this experiment is to see the latency benefits of both of these modes of read() API.

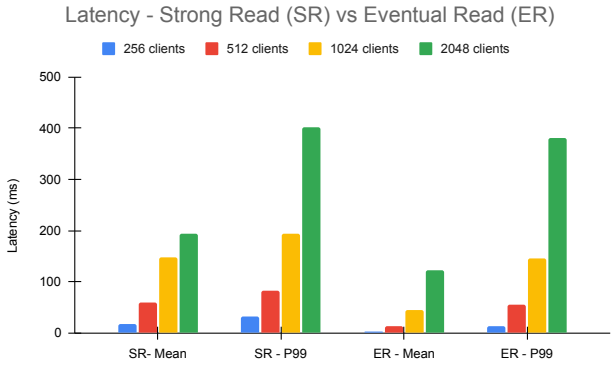

Figure 18: Latency: Strong vs Eventual reads

Observations: As shown in Figure 18, we observe that strong reads (SR) are 2x the latency of eventual reads (ER). Reason for this is that primary is the source of bottleneck for SR (as all strong consistent reads are catered by primary server only). We also notice that with increasing number of clients, the difference between p99 of SR and ER reduces as the number of backup servers are fixed and we are increasing the load by increasing number of clients (i.e. it is the saturation point for this P/B config). With these many ops per second, it doesn't matter where the request is sent since both the primary and the backups are overloaded. We conclude that ER increases the system throughput and reduces latency but at the cost of loose consistency semantics (as data is not guaranteed to be consistent).

4.3 Latency: with and without client caching

Setup: The concept of client side caching is introduced keeping read-heavy workloads into consideration. Hence, the setup for this experimentation includes client with more reads than writes. We use client with reade/write in ratio of 10:1 ratio. We use 1 shard with 1 primary and 2 backup servers with client side caching enabled.

Observations: Figure 19 showcases the effects of caching on read latency. As expected the mean of read drastically reduces due to caching. P99 of the diagram represents the cache miss as these reads are performed from the server. Moreover, due to less server load, the P99 of caching case is better than the non-cached case p99. All of these observations showcases the benefits of client caching.

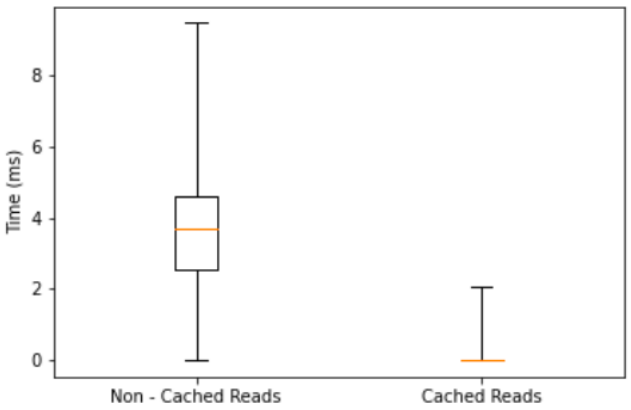

Figure 19: Read Latency: With vs Without Caching

Figure 20 showcases the effects of caching on write latency. Generally, one would expect the write latency to worsen due to additional client cache invalidations to be sent by the primary server for cached keys. But looking at the amortized cost, we get surprisingly better results - we see that write latency is also improved in the caching case as compared to the non-cache case. The reason being system is less loaded due to the cached reads being catered by the client themselves (in addition to the reduced network traffic) and hence the server load is drastically reduced.

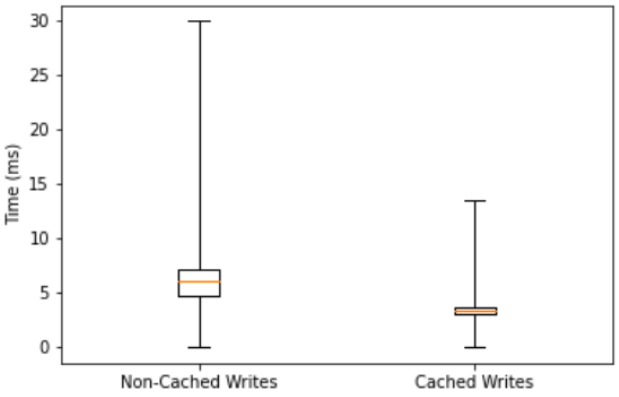

Figure 20: Write Latency: With vs Without Caching

5 Server Load

In this section, we experiment and see the load on server with and without different proposed optimizations and use-cases. The aim of this experiment is to see the system state from the server's point of view (as opposed to client's point of view who wants low latency).

5.1 Leased client caches with server-driven invalidations

Setup: The concept of client side caching is introduced keeping read-heavy workloads into consideration. Hence, the setup for this experimentation includes client with more reads than writes. We use 100 concurrent clients reading and writing in ratio of 10:1 ratio. We use 1 shard with 1 primary and 2 backup servers with client side caching enabled. Server load is calculated as number of operations/second served by the server.

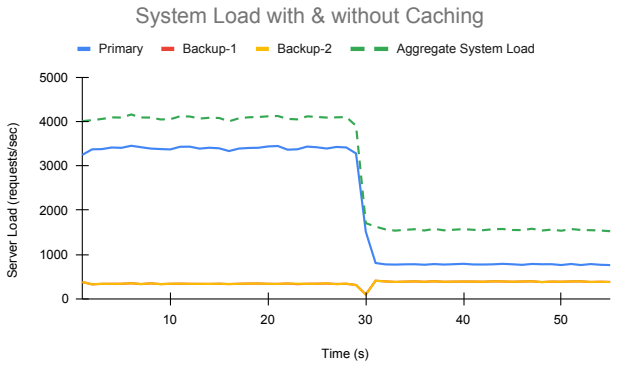

Figure 21: Server load: With vs without Caching

Experimentation: We use 2 scenarios of clients. One with caching disabled (left side of the diagram) and one with caching enabled (right side of the diagram). Once the clients with caching disabled are done with execution, clients with caching enabled are started (marked after the drop in server requests). We plot the aggregate and individual server load in the timeline as shown in Figure 21.

Observations: Firstly, we notice that the aggregate system load (throughput) increases with caching as there is less client-server interaction and hence less server load. Primary server load decreases in the right side of figure (clients with caching). Secondly, the time to process the same workload (marked by the width of each case on timeline) reduces in the caching case. Finally, one interesting observation is that the load on backup server increases with the caching case. We believe this is due to the fact of primary serving more writes which results in more number of replications per second which, therefore, implies increased backup load.

5.2 Server load: Strong vs Eventual Reads

Setup: In this experiment, we compare the individual and aggregate server load with strong consistency vs eventual consistency reads. System contains one shard with 1 Primary and 2 backup servers and caching is disabled from client side. We execute 500 concurrent clients performing mix of read/write operations in ratio of 1:1.

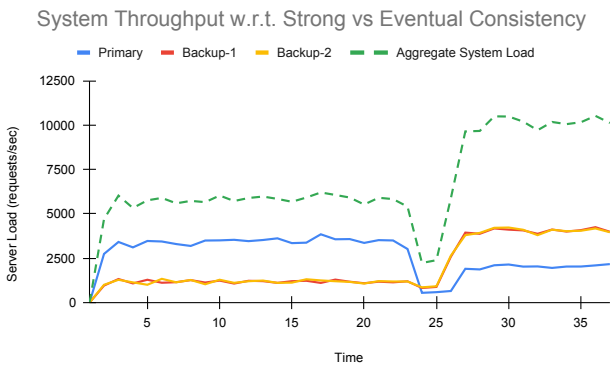

Figure 22: Server load: Strong vs Eventual Reads

Experimentation: First, we execute clients with strongly consistent read workloads (left side the figure 22 - before the dip in curve). Once 500 clients are done executing, we start 500 clients with eventual reads (right side of the dip in curve) and contrast the server loads in both cases.

Observations: With reference to Figure 22, we notice that the aggregate throughput increases with eventual consistency due to better load distribution on servers. Moreover, the time to process the same workload reduces in case of eventually consistent reads as the servers are less loaded and can serve more requests/second

6 Conclusion

In this report, we show that our distributed key-value store built on top of a production KV store - RocksDB, is highly fault tolerant while not loosing on the performance benefits of the original system.

The primary reasons for this is our original design decision to scale-out by using the concepts of Sharding, Primary-Backup architecture, Centralized coordinator and checkpoints. We heavily drew upon and used the concepts of concurrency to process things in parallel when possible, for ex. a multi-producer multi-consumer queue we implemented to process replication in parallel, checkpointing and recovering in parallel to serving incoming requests. In addition, paying attention to varying demands of user workloads and providing application centric features like Caching with lease, faster Reads (Eventually consistent reads) and faster Writes (Fast-Acknowledge write) we not only reduce the observed latency at user side but also increase the system throughput and decrease the load at the same time.

Our system is also fault tolerant and it recovers in case of both primary and backup server failures. Moreover, it provides crash and data consistency guarantees in the face of failures while providing high performance and scalability. We present several insights derived from our analysis of this sys-

tem. We back the design decisions explained in the system design by the performance statistics collected in numerous experiments.

Overall, it was a really fun project and we thank the professors and TAs (Jung, Vinay and Surabhi) for their support, time and the opportunity :)

7 Project Demos

Kindly refer to the below hyperlinks for detailed demonstration of the proposed system

- **End-to-End functionality : Video**
- **Sharding : Video**
- **Data consistency : Video**
- **Crash Demo : Video**

8 Implementation details: Issues faced, How and where?

In this project, we used C++ for implementation and third party tools like open sourced rocksDB, gdb tool for segmentation fault fixing. In this section, we describe interesting bugs (locking issues, memory management issues, data structures, some interesting bugs) and implementation details of some problems we used in our project.

- **Coordination Service:** Multiple servers and clients requests current system state with coordination service in parallel. Also, updates can happen in parallel. Thus locking and synchronization was essential on the coordination service.
  - Locking and synchronization: coordination service is responsible to maintain a coherent view of system state (currently online servers and their roles) and broadcasting them to the clients and live servers. We faced lot of issues in the starting due to locking issues as multiple nodes (clients and servers) can call getSystemState() rpc while the server is updating the system state. So, we had to use exclusive locks on the system state to avoid the above issue. The changes are at many places but one reference of the code is coordinator.h line: 53.
  - In case of failures, the coordinator elects a new primary. Initially, we faced a bug where we forgot to remove the backup which was elected as new primary from the backups set. This caused configuration issues for us. This was later identified and fixed at coordinator.h line : 82
  - Heartbeat: The coordination server maintains heartbeat with each node. Initially, we implemented each registration of a node to spawn a parallel thread and joining on that. We had to add adhoc method on the server side to wait for 1 second assuming that registration is done even without any ack from the coordinator service. We solved this issue by detaching heartbeat threads from registration routine and making sure to clear these threads on coordination service shut-down coordinator.h line : 199
- **Server library:** ...
  - System instability due to on-demand creation of threads at backups. Initially, we had implemented on-demand threads for the primary server side replication task. It worked fine for 30-50 client applications but system throttled and we faced very bad read/write latency in case of multiple concurrent applications. This was fixed by implementation of multiple-producer multuple-consumer thread pool. We create a pool of threads which caters requests from a common queue one-by-one. This made our system scalable and run with 3500 clients without any issues. Implementation to this can be referred to the thread pool implementation commit : 2028957b1
  - deadlock: We faced an interesting issue where our rpc write was acquiring a lock on system state to get a list of backups for replication routine. After this it acquires a shared lock on the system state so that an updateSystemState() from coordinator cannot change the backup set while we are reading the backups set. Issue happened because in checkpoints, we acquire an exclusive lock on systemState and after this we were acquiring locks on writes to disable further writes while checkpointing is being done. This cyclic dependency of locks caused deadlock. We resolved this by creating an ordering on the lock acquisition as shown in deadlock prevention commit : 291d3a3e0
  - Random segmentation faults with multiple clients: We are using many hashmaps in the server memory to store relevant information regarding serverState. Concurrent access on these hashmaps are not thread safe (surprisingly!). Issue being hashmaps are rebalanced sometimes in background and any operation during that time will cause seg faults. We used gdb tool for finding this issue of seg fault and fixed it by protecting these hashmaps using locks. Refer to commit for more information on this issue: unordered map concurrent access issue commit : 291d3a3e0a

- **Client library:** current implementation supports multiple concurrent applications: We designed our client library code which spawns multiple concurrent application threads that interacts with this client library. Each application calculates its own set of latencies and statistics which are aggregated by the client library and printed. Below are the set of implementation challenges and how we solved them:
  - Dynamic updation of system state with coordination service: Whenever any of the server is killed and rpc from the client library fails, client automatically fetches the new system state (live server addresses) from the coordinator service. It then redirects all of the application threads' rpc requests to the new server. Refer to this part of client.cc line: 437.