xLSR-Extensible Large Scale Replicated Database

Sourav Suresh, Varun Kaundinya
University of Wisconsin Madison

I. Introduction

This research project focuses on designing a extensible large Scale storage solution leveraging Raft consensus algorithm to build a cloud native replicated database system using LevelDB as the backend datastore. The proposed algorithm is an adaptive version that has flexibility to autoscale the system and also batches requests to improve overall system throughput. Communication between nodes is achieved using gRPC. The main objective of this study is to demonstrate the effectiveness of build a large scale storage systems which work effectively in Cloud Environments where the scalability is one of the huge factor. The implementation details and performance characteristics of the system are discussed in further sections.

II. Design And Implementation

In the rapidly evolving landscape of cloud-based systems, the ever-growing demand for data storage and scalability from the backend poses a significant challenge. To address this challenge, our research introduces the notion of Data Clusters. By creating smaller subclusters, we establish a foundation for horizontal auto-scaling, allowing for the seamless addition or consolidation of new subclusters. To support the management of multiple subclusters, we propose the implementation of an additional layer in the form of a Load Balancer. This Load Balancer provides the advantage of intelligent redirection to the appropriate data clusters, effectively abstracting the selection process while ensuring uniform utilization of the Data Clusters. Furthermore, the Load Balancer offers mechanisms such as automatic scaling based on storage utilization to cope with the increased demand for data writes. These combined mechanisms grant clients an abstracted perspective of unlimited storage, guaranteeing durability and consistency.

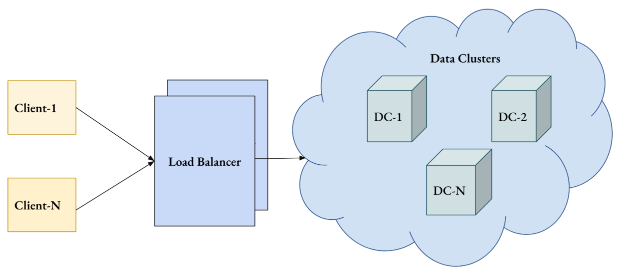

Fig. 1. System Architecture of xLSR DB.

Leveraging Raft as the consensus protocol, our 3 main key components of the system are *Leader Election*, *HeartBeat Entries*, and *Commiter*. With Leader Election, all nodes in cluster go through the election process requesting for the vote with randomized timeouts by incrementing their term. As defined by Raft protocol, the winning criterion to be a leader is to have higher term and higher log entries. Each HeartBeat the leader gives a confirmation about the state to followers along with any new AppendEntries. During this time we introduced a mechanism of *Snapshot* (point in time copy), where new incoming requests go to a buffer and eventually applied back to the main log entry once they get majority consensus. We see 2 main benefits from this approach. 1. It gives flexibility for the leader to accept client requests asynchronously and send them as a batch with the next heartbeat. This allows adaptive buffering/batching of client requests which would help us in scalability. 2. In case there is no majority for the log entries, it ensures no holes are present within them. With our Snapshot mechanism we rollback the difference which had not gotten majority and apply the snapshot back which keeps the log entries clean. The Committer would help us in committing all the requests which had got the majority from the consensus. All the 3 mentioned components would run as background scheduled threads, which makes the system modularized from an implementation perspective. We also increase the term gradually during the leadership phase to mimic the original implementation.

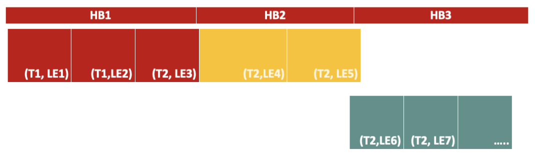

Fig. 2.. LogEntry design, red means committed, yellow means waiting for consensus and green is in snapshot.

As mentioned earlier, one of the advantages with taking Snapshots was easier rollback. This is achieved by maintaining the *lastAppliedIndex* and *lastLogIndex*, where entries till *lastAppliedIndex* (red entires in Fig. 2) ensures log entries have got majority and are ready to be committed. The entries between *lastAppliedIndex* and *lastLogIndex* (yellow entries in Fig. 2) are the ones which are waiting for the majority. Any new log entries from the client while the majority process is going on would end up in Snapshot (green entries in Fig. 2). This gives us flexibility to decide if the difference would be accepted/ rejected based on consensus majority and later on the Snapshot is applied to it. We also ensured when the leader is not getting majority for certain instances of difference, we added a mechanism for the leader to step down and thereby stop accepting any new requests from clients which avoids false commitments until a new leader is elected.

The Load Balancer serves as a crucial component in the system by accepting client requests and efficiently directing them to the appropriate data cluster. To ensure the liveliness of data clusters, the Load Balancer employs a daemon process that collects utilization information from all clusters. By summing up this data and comparing it against a predetermined threshold, the system can automatically scale and add new data clusters when necessary. Furthermore, the system supports versioning, retaining updates within a specified limit. The Load Balancer persistently stores metadata of incoming requests, enabling it to map objects to their respective versions and data clusters. When processing incoming read requests, the Load Balancer refers to the metadata to determine the appropriate redirection destination. Reads occur synchronously within the data cluster and are not ordered by the RAFT protocol. On the other hand, for write requests, the Load Balancer retrieves the latest version, increments it, and redirects the request through the RAFT protocol to the cluster with the lowest utilization. We added adaptive queue based mechanism as mentioned in Fig. 3. which dynamically process the request for the coressponding data cluster. Once we get acceptance from individual data cluster, we now persist this meatadata information with RAFT based consensus. To optimize Load Balancer operations, the system incorporates application-level caching, which enhances their speed. Additionally, the Load Balancer runs a compaction thread on a scheduled basis, sending delete requests to data clusters containing previous versions. This process effectively frees up resources within the data clusters, which can then be utilized by the Load Balancer for request redirection.

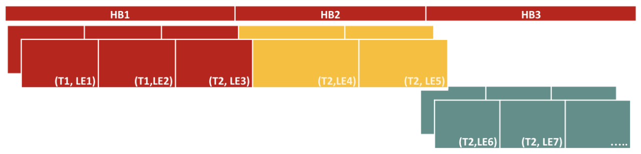

Fig. 3. Queue design, red means ready for persistance, yellow means waiting for data cluster acceptance and green is in snapshot(incoming requests).

III. Testing

Consisteny Tests: The consistency tests at a high level involved validation of the system being fault tolerant with nodes going down, systems having network delays or network partition. We started with basic consistency checks on whether the data is written to the system via leader reaches to all followers. Further, we tried to bring down leader, some follower nodes to verify if the system is still consistent by electing a new leader or getting majority with other available nodes and continue capturing the incoming client requests. Later on, we moved to bit more complicated scenarios to simulate partition/ network delay cases. We evaluated a scenario where one follower was isolated from the cluster and it tried to vote for himself by incrementing term without getting majority, this might lead to a scenario where the isolated follower might outbound the leader's term and when it gets connected back to cluster, we verified this follower irrespective of higher term than existing leader would not make any inconsistencies with leadership. Also, we evaluated scenarios when there is network partition between leader and followers. Followers since they don't receive heartbeat from the leader, they try to step up as a candidate and start the election process. Eventually when the cluster was consistent with the addition of new log entries, the older leader comes back and should not lead to inconsistencies ie., the existing followers should reject any requests by this old leader and eventually this old leader should step down and follow the new leader in the cluster. This also validates a scenario when there are 2 leaders at any given point in the cluster (new & old leader in this case), should not lead to inconsistencies in the system. With the entire system, we performed consistency checks where we wrote the data at different rates and read the written data and checks for the consistency. We also test **per object sequential consistency,** i.e Multiple writes and read, We always get the final written version of the object.

Sanity Tests: We initially check for **utilization autoscaling** i.e based on cummalative storage utilization and threshod, scaling gets triggered and writes gets directed to the lowest utilized subcluster. Then we checked for **leadership failures** across load balancer or data cluster should lead to no acceptance of client request or processing from Load balancer or subcluster and finally we killed of some **follower nodes** which didn't affect the happy path performance. Unless it lead to a majority crisis.

IV. Performance

As requested during the presentation we came up with few more performance metrics, these are described below.

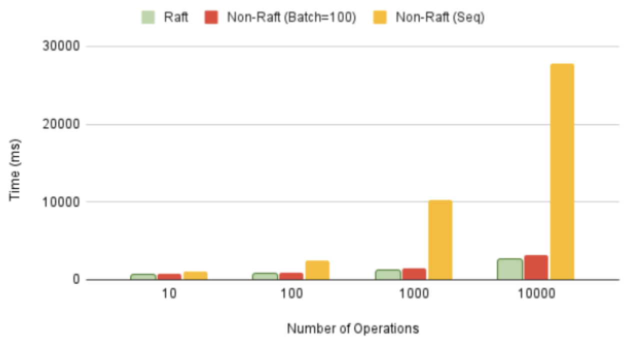

Fig. 4. Performance of proposed Raft based Replicated DB with Non-Raft (operations directly on LevelDB) with and without batching.

With randomized election timeouts, we observed on an average of *1.4 µs* for the initial leader to come up in the clusters. Also we observed it takes around 0*.5 µs* on average during re-election. We evaluated our performance benchmark by comparing with proposed Raft based implementation, Non-Raft sequential, and Non-Raft batching with Fig. 4. We can see a significant difference between Sequential and Batching the requests, and also we see Non-Raft with batching and proposed Raft based consensus implementation gives comparable performance. From Fig. 5., we can observe the advantage of adaptive batching which is one of the benefits of snapshotting that can provide better throughputs than fixed size batching. The intuition behind this is for every heartbeat, we apply the batch proportional to the number of requests received between 2 heartbeats which gives us flexibility compared to having fixed size batching which leads to more rpc requests when the number of client requests exceeds batch size. With these results, we now leverage Raft with batching as the consensus protocol.

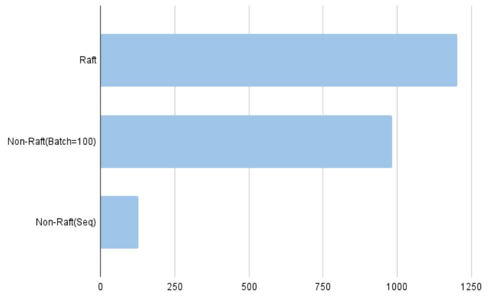

Fig. 5. Throughput with proposed adaptive batching consensus algo with Non-Raft with constant batching and sequential operations.

With Fig. 6. & 7. we see comparable results with write and reads. With our asynchronous model, we expect write performance to be good and the same can be observed from results. Since, for every read we go to subsequent data cluster redirected by Load Balancer synchronously, we see comparable performance. This is the cost we are paying for consistency (similar to how we observed with non-nilext operations in [2]).

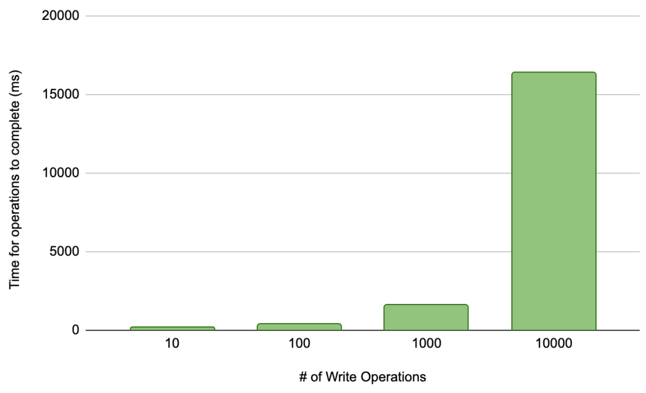

Fig. 6. Performance of writes with proposed design

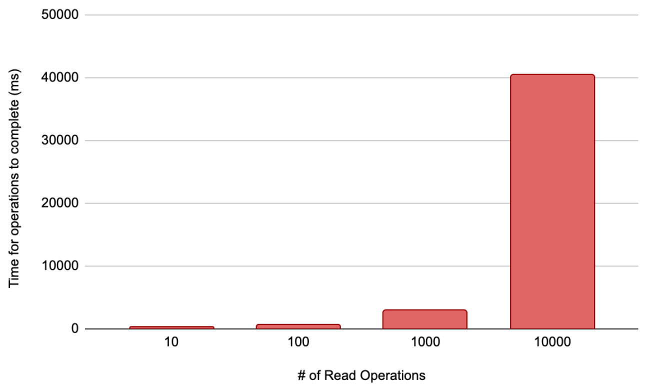

Fig. 7. Performance of writes with proposed design

From Fig. 8. and Fig. 9. we clearly observe that Recent Updates has a significant greater performance (High throughput & lower latency) compared to actual writes. This is due to application level caching where writes are automatically versioned through cache. As mentioned above, we clearly observe our write performance is better over reads due to asynchronous nature. Thereby we can claim our system is more suitable for applications which are write/ update heavy).

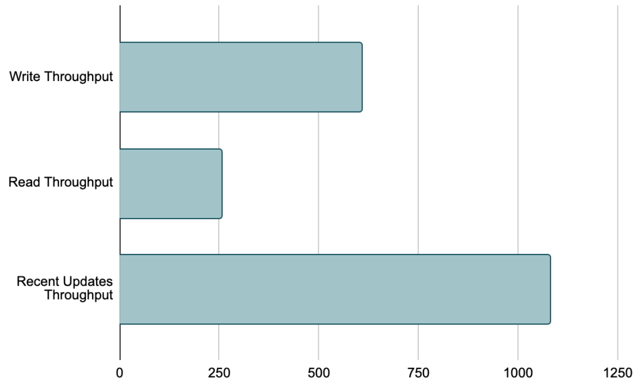

Fig. 8. Throughput comparison between writes, reads & updates.

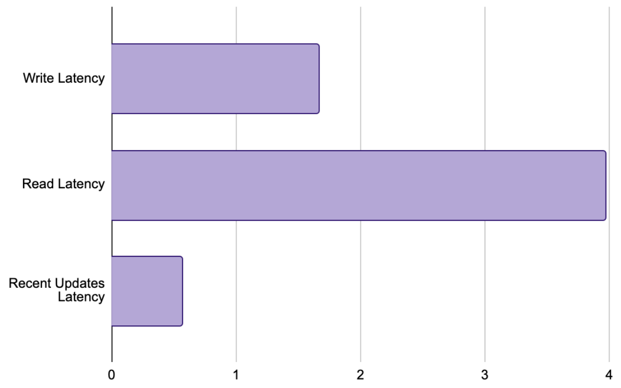

Fig. 9. Latency (in ms) comparison between writes, reads & updates.

V. Future Work

One thing which was apparent was having a single Leader to accept all client requests and also manage all the followers is a bottleneck and a redesign of the system needs to be done to handle the scale. We have addressed this problem at the data cluster level but at the load balancer level sharding should be implemented to distribute the load. Our design is entirely Async, it would be interesting to have a performance comparison between async and sync versions of our Database. We believe the sync version will provide stronger guarantees consistency. Also, we see scope in checkpointing the old log entries to avoid OOM issues. Another solution from an implementation standpoint, is to deliver the heartbeats and voteRequest via a push-based notification scheme like SNS. Doing this reduces threading overheads, nd helps in decoupling the cluster and finally we could have intelligent placement strategy that uses locality and access pattern alongside utlization strategy to direct the next placement.

V. References

[1] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC'14). USENIX Association, USA, 305–320.

[2] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Exploiting Nil-Externality for Fast Replicated Storage. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21). Association for Computing Machinery, New York, NY, USA, 440–456. https://doi.org/10.1145/3477132.3483543

[3] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. 1996. Bayou: replicated database services for world-wide applications. In Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications (EW 7). Association for Computing Machinery, New York, NY, USA, 275–280. https://doi.org/10.1145/504450.504497