# Replicated Database using RAFT

Sourav Suresh, Varun Kaundinya, Nathan Lidukhover
University of Wisconsin Madison

## I. Introduction

This research project focuses on designing a variant of the Raft consensus algorithm to build a replicated database system using LevelDB as the backend datastore. The proposed algorithm is an adaptive version that batches requests to improve system throughput. Communication between nodes is achieved using gRPC. The main objective of this study is to demonstrate the effectiveness of the modified Raft algorithm in achieving a strongly consistent replicated database system that can handle server failures. The implementation details and performance characteristics of the system are discussed in further sections.

## II. Design And Implementation

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.

As mentioned earlier, one of the advantages with taking Snapshots was easier rollback. This is achieved by maintaining the

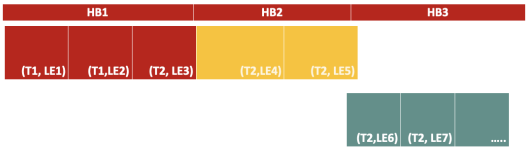


Fig. 1. LogEntry design, red means committed, yellow means waiting for consensus and green is in snapshot.

*lastAppliedIndex* and *lastLogIndex*, where entries till *lastAppliedIndex* (red entires in Fig. 1) ensures log entries have got majority and are ready to be committed. The entries between *lastAppliedIndex* and *lastLogIndex* (yellow entries in Fig. 1) 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. 1). 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.

## III. Consistency Testing

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.

## IV. Performance

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

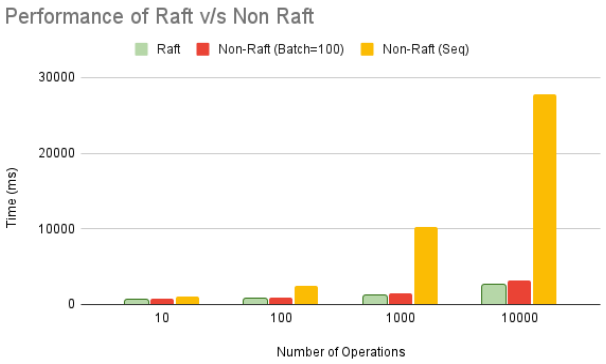


Fig. 2. Performance of proposed Raft based Replicated DB with Non-Raft (operations directly on LevelDB) with and without batching.

With randomized election timeouts, we observed on an average of *2420 ms* for the initial leader to come up in the cluster considering the initial thread setup delay of *1000ms*. Also we observed it takes around *558.6 ms* 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. 2. With Non-Raft, we directly run the operations on LevelDB to capture the performance metrics. The batching version of Non-Raft would send the requests with a defined window size, whereas sequential would just go with window size=1. With Raft based implementation, we run the same set of operations on our proposed consensus algorithm. Fig. 1 shows the comparison graph of all 3 scenarios. 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.

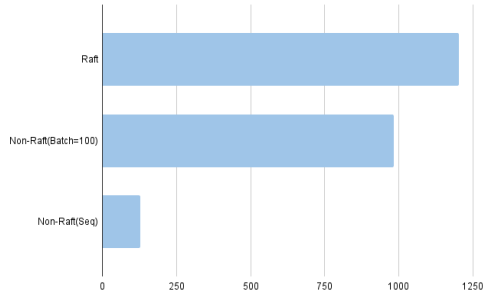


Fig. 3. Throughput with proposed adaptive batching consensus algo with Non-Raft with constant batching and sequential operations.

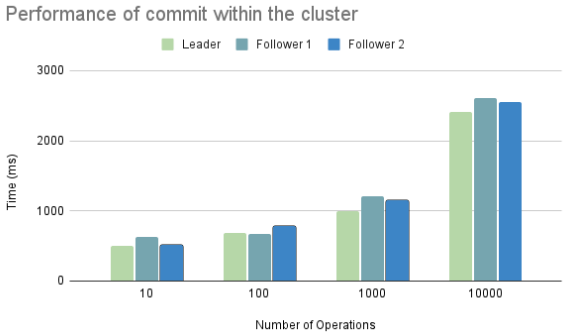


Fig. 4. Performance of commits within the cluster (Leader & Follower nodes)

From Fig. 3., 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 Fig. 4., it shows how much time it takes for all the nodes to commit the writes to the LevelDB datastore. Fig. 5. explains how much time the new Follower takes to synchronize with the existing committed data of the leader.

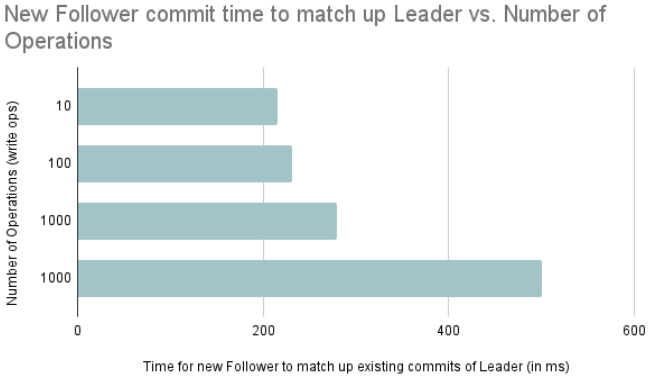


Fig. 5. Performance of commits with new follower when added newly to the cluster

## 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 when we horizontally scale the followers. 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, and helps in decoupling the cluster.