Replicated Database

Akshay Raghavan
raghavan9@wisc.edu

Girish Jonnavithula
girish.jonnavithula@wisc.edu

Rishideep Reddy
rrallbandi@wisc.edu

Overview:

The replicated database is implemented using a replicated state machine approach with the help of Raft consensus algorithm. The underlying database used is LevelDB. We use 2n+1 servers, each with a database server to intercept client requests and a raft component to implement a distributed replicated log. The raft component establishes a consensus on the order and contents of the replicated log by

1. Electing a leader among all the servers on start-up and on leader failure.
2. Leader replicating log entries in all followers even under follower crashes and network partitions.

Design:

Each server has a database component that intercepts client requests and a raft component that accepts entries from the database component to replicate. The raft component by itself contains multiple parts:

1. AppendEntries: Contains a GRPC client and server thread to interface with other servers. An AppendEntries threadpool tries to append log entries on followers until success and reduces the overhead of thread lifecycle management. This RPC is also responsible for catching up any slow or crashed follower.
2. Heartbeat: An empty append entries RPC is sent periodically to reset a watchdog timer on the followers.
3. Leader Election: A follower becomes a candidate and requests votes to become a leader when the watchdog timer expires. A RequestVotes threadpool is used for the election and the new leader, if elected, starts appending new entries and sending heartbeats.
4. A user interface is created for better user experience, analysis of the algorithm and extracting metrics out of the system.

Experiments:

We obtain metric values by utilizing UI endpoints. We input a request rate, which generates random data and stores it in the database. Finally, we receive the total operation time once it's completed.
A few observations:

1. From our experiments of sending GET requests to any server versus sending all the requests only to the leader, we observed a significant performance disparity. Hence we chose performance over consistency.
2. The response time for the first 10 PUT requests remained constant while increasing linearly for higher numbers of PUT requests. This was because of the threadpool being the bottleneck with a size of 10.
3. The increase in time for 1000 PUT requests with cluster size was not substantial. This is attributed to the fact that all our testbed servers ran on localhost.
4. Due to increased odds of multiple simultaneous candidates, increasing the cluster size saw an increase in the time taken to elect a new leader.

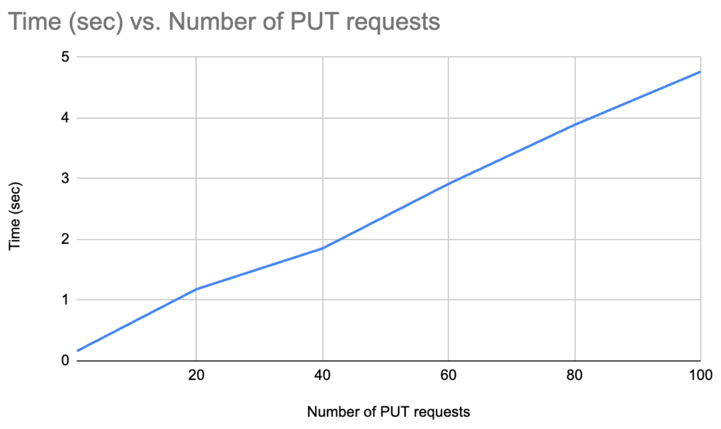
Please find all the graphs in the presentation.

Concerns from our presentation:

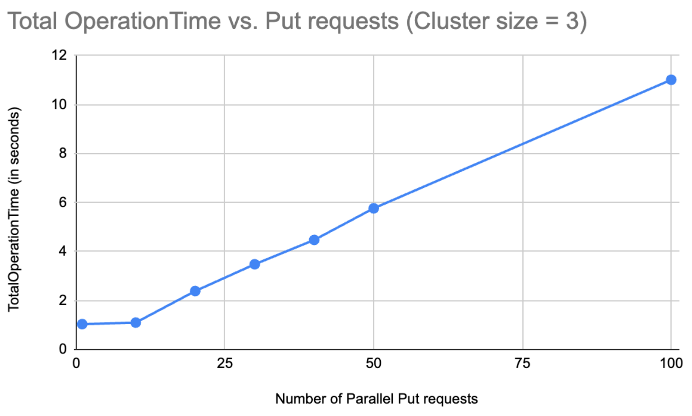
Prof. Remzi pointed out that the throughput of our algorithm is slow and he suspected that the web server might be the issue. We later realized that the high sleep times across our codebase might be the culprit. We fixed it by

1. Using a heartbeat of 1 sec instead of 5 seconds used earlier
2. Decreasing the poll time of the database server to check the commit index to 0.15 seconds instead of 1 second.
3. Decreasing the grpc retry timeout to 0.5 seconds instead of 1 second.

Here is the updated PUT throughput graph.



The original graph for reference:



[Presentation Link]