Replicated Database using Raft

Group 14

Madhushree Nijagal	mnijagal@wisc.edu
Priyavarshini Murugan	pmurugan@wisc.edu
Tanisha Hegde	tghegde@wisc.edu

Design and Architecture

The design of our replicated database consists of 3 servers and 1 client. Each server has a copy of the database and a logfile. The servers reach to a consensus using the Raft Protocol. Every server uses a serverfile that has the IP address of other servers. Our current implementation allows you to start a node as a follower or a leader.

```
./server <ip address>:<port> [is_primary(yes)]
```

LevelDb is used to store Key-value pairs and supports the `Get(key)` and `Put(key,value)` pairs. Our system uses gRPC for communication between servers as well as client-server interaction and is implemented in C++.

Logfile structure: The binary logfile structure is shown in the table.

Current Term	Voted For	Log Entry		
		Term	Key	Value
64 Bytes	22 bytes: IP+port	64 Bytes	4096 bytes	4096 bytes

Communication:

AppendEntriesRPC – The leader sends appendEntryRPCs to followers that includes the current term, Leader Id, follower log index, follower log term, and log entries if there is a mismatch. The appendEntries RPC are termed empty when the entries field is empty. New client requests results in entries being sent to each follower. Entries are truncated by the follower when there is a mismatch in the index.

RequestVoteRPC – The candidate requests votes from every node using separate threads that use a shared pointer which is updated on receiving a vote from the nodes. A quorum is formed when `[number_of_servers/2 + 1]` yes votes are received. The candidate retries requesting votes to nodes every 1ms.

gRPC client requests – The client requires a server file which has the information of all servers that it can try contacting. A single `Get(key)` and `Put(key,value)` request will support 4kB data for key and value. On reaching a follower, if a leader is elected it directly uses the `leader_id` else it will try all the nodes infinitely. Before exiting, the client stores the current leader in a `curr_server.txt` file that could be used if the same client tries a new request. The client can be invoked by `./client server.txt`

Design Flow

The dataflow resembles a typical raft protocol. The AppendEntriesRPC and addressing client requests is taken care by two different threads. Client received an acknowledgement only after the DB operation is successfully completed.

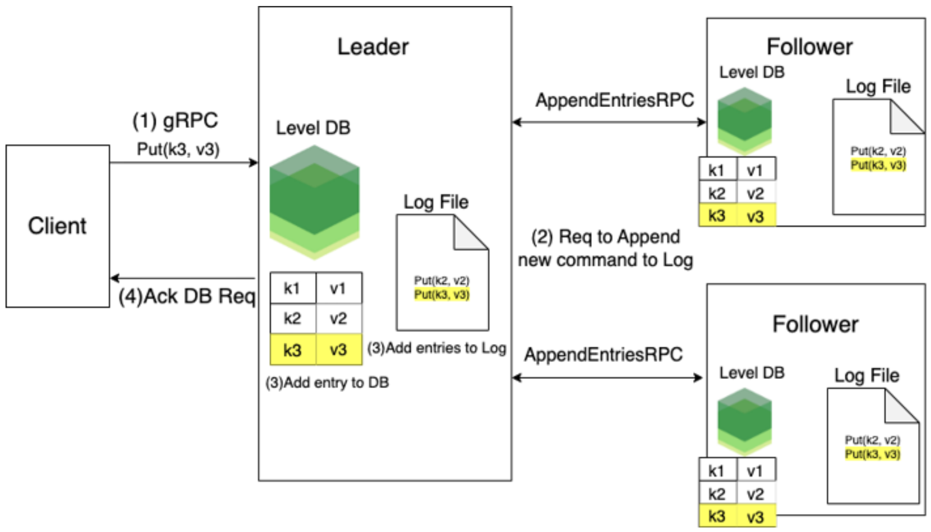


Fig 1: (1) gRPC request from client (2) Req to Append Logs (3) Leader commits the logs to logfile and database (4) Send Ack to client

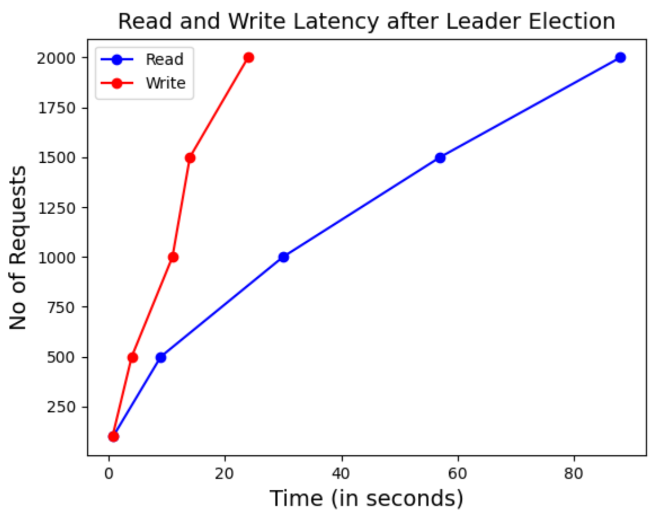
Testcases

	node1	**node2**	**node3**	**Leader**	**Log results**	**Client**
1	Follower (assume leader after elect ion)	Follower	Follower	Election decides who becomes the leader	Leader(n1) logs replicated in followers (n2,n3)	Directed to the leader
2	Leader	Follower	Follower	n1 continues as leader	Leader(n1) logs replicated in followers (n2,n3)	Directed to the leader
2.a	Leader (crash)	Follower (assume leader after elect ion)	Follower	Election start between n2 and n3 and next leader is chosen	Leader (n2) logs replicated in followers (n3)	Retries until leader is elected
2.a. 1	Leader (crash an d restart as leader AFTER	Follower (assume next lead er)	Follower	[1] Election starts between n2 and n3 and timeout decides the next leader	[1] Leader(n2) logs replicated in followers (n3) [2] n1's logs	Requests are redirect ed from n1 to n2

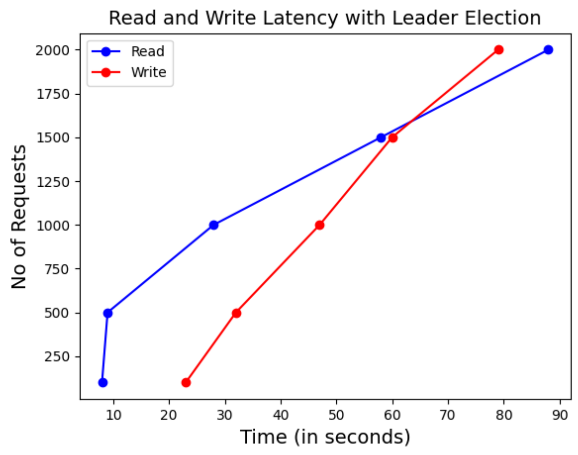
	n2 is chosen as the curre nt leader)			[2] n1 becomes follower even if restarted as leader.	made consistent with the leader(n2)	
2.a. 2	Leader (crash an d restart as leader BEFORE election)	Follower	Follower	n1 continues as leader	Leader(n1) logs replicated in followers (n2,n3)	Requests are still sent to n1
2.b	Leader	Follower	Follower(crash)	n1 continues as leader	Leader(n1) logs replicated in followers (n2)	Requests are still sent to n1

Statistics and Performance

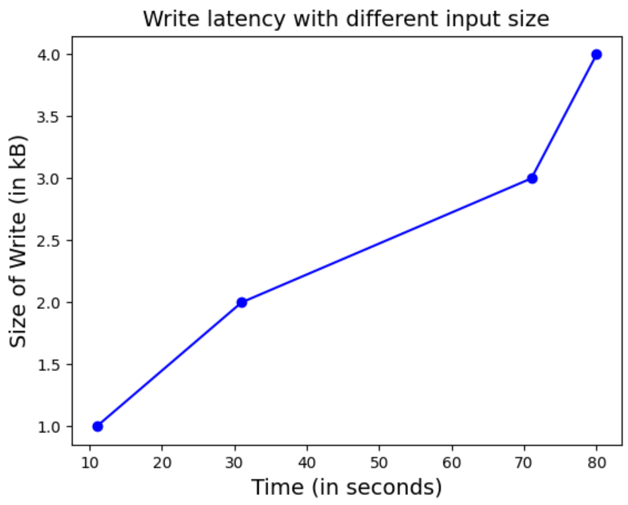
We measured the Read and Write operation time for requests made by the client when the leader is already elected. We first do the Write operation followed by the Read operation. We measured the time taken for different number of requests sent from the client in increasing order from 100-2000. We noticed that the Write operation performed better than Read (as expected) because Read iterates through the database during key search. The graph is as shown below:



Secondly, we measured the Read and Write request made by the client during election. We measured the time taken for different number of requests sent from the client in increasing order from 100-2000. The time taken by the Read operation is consistent with the previous graph readings where the leader was already elected. The noteworthy measurements are of the write operation where there is a slight delay in operation because of the leader election which has a timeout of 5s + Random (0, 100ms). The read latency increases compared to write latency after 1500 requests. The graph is as shown below:



Finally, we measured the time taken to complete the write operation for different values of input size, we found a steady increase in the time as the size of the input was varied between 1kB-4kB of data.



Apart from these performance test cases, we have tested our codes with 3 servers on the same node as well as on different nodes. We observed no significant difference in time.