P4 Replicated Block Store using Raft

Group 5: Aakarsh Agarwal, Muhammad Asad Khan, Bijan Tabatabai, Salman Munaf

Table of Contents

1. Introduction

We implemented a Replicated Block Store System using Raft consensus protocol. This is an extension of P3. We used Primary/Backup configuration with 2 server nodes in our P3 implementation. However, for scalability and to increase the number of servers in our replication system to handle more failures, we needed some consensus protocol. We chose Raft because it is one of the popular state-of-the-art consensus algorithms for state replication over a distributed system of nodes. Also, it is easier to implement than Paxos and the paper (base for our implementation) is very well explained.

2. Design

2.1 Design Overview

Our system consists of a leader and an arbitrary number of followers. The leader is exclusively in charge of handling read and write requests from the clients, while followers are used only for data replication. The leader handles read requests by reading from its local copy of the datastore. Write requests are handled by the leader appending the request to its persistent log and forwarding the request to the followers so they can do the same. Once the leader receives confirmation that a majority of the servers in the system have persisted the log entry, it commits the request and applies it to its copy of the datastore.

If the leader becomes unavailable for whatever reason, the followers will increment its term number and attempt to become leader. If a follower receives votes from a majority of the servers to become leader, it will take on the leadership role. The leader will periodically send the followers heartbeat messages to indicate its liveness.

2.2 The Raft Log

A server's raft log is a record of all the write requests, committed and uncommitted, a server has received. Each entry in the log contains the follower: the term the request was received, the address in the block store to write to, and the data to write. A log entry is uniquely identified its term and its index in the log.

When the leader receives a write request, it appends the request to the end of its log. It then tells its followers to add the request to its log in the same location using the AppendEntries RPC. If the follower's log does not match the leader's, the leader will roll back the follower's log until it matches and append the new request there. This mechanism helps ensure that if the leader and a follower have the same log entry at an index, all preceding indices will be the same as well.

After a follower has persisted a request to its log, it will notify the leader. When the leader has received such a notification from a majority of the servers, including itself, it will commit the request by applying it to its block store. In subsequent messages to the followers, the leader will send the index of the latest committed long entry, so they can apply the request themselves.

2.3 Leader Election

If a follower has not heard from the leader for some timeout, it will increment its term, note that it has voted for itself for that term, and send RPCs to the other servers asking them to vote for it. If it receives yes votes from the majority of servers, it becomes the leader for the term. The aforementioned timeout is randomized for each server to prevent servers from vying for leadership at the same time.

A working democracy requires well informed voters, so when a server receives an RPC from another server asking for its vote, the receiving server only grants its vote if the sender meets the following criteria. If the requester's term is less than the receiver's, the sender is out of date, so the receiver rejects the request. However, if the requester's term is greater than the receiver, the receiver updates its term. The receiver will only reply with a yes vote if the requester's term is greater than or equal to its own, the receiver has not yet voted for a leader in the requested term, and the requester's log is at least as up to date as the receivers. The last condition is needed to ensure that the new leader will have every committed write request. This is because in order to be committed, a request must be replicated on a majority of servers; therefore, a server with an incomplete log cannot get a majority of yes votes to become leader.

2.4 Persistence

Both the blockstore data and raft log are persisted to the server's disk. To prevent data corruption in the case of a crash when writing to the datastore, the original contents of the to be overwritten data is written to an undo log which is deleted only after the write is completed. If an undo file exists when the server begins, that is an indication that there was a crash during a write, so the server applies the contents of the undo file.

The raft log is persisted in a similar way. However, because the raft log is append only, the process can be simplified. Instead of writing the original contents to the undo log, only the offset to truncate the file to is stored in the undo file.

For the correctness of the protocol, the current term and who the server voted for in that term are also persisted.

3. Implementation

- Our implementation is coded in C++.
- We used gRPC for communication between the nodes.
- Client identities leader from the current leader id sent in Read and Write requests by the server nodes.
- We use .txt file in arguments containing server IPs to identify other nodes' location.
- Blockstore is stored on disk as a single file, same as in P3. For raft log, log entries are stored as an array/vector in a raft.log file. Each log entry index contains term and data to be written.

4. Testing

We tested our implementation for correctness using test cases ranging from happy flow to corner cases including crash at possible points in the system. We tested our code using 5 server nodes, but it can be scaled out with N number of nodes in the system. All server nodes get their ids (0-4 in our case) from common arguments.

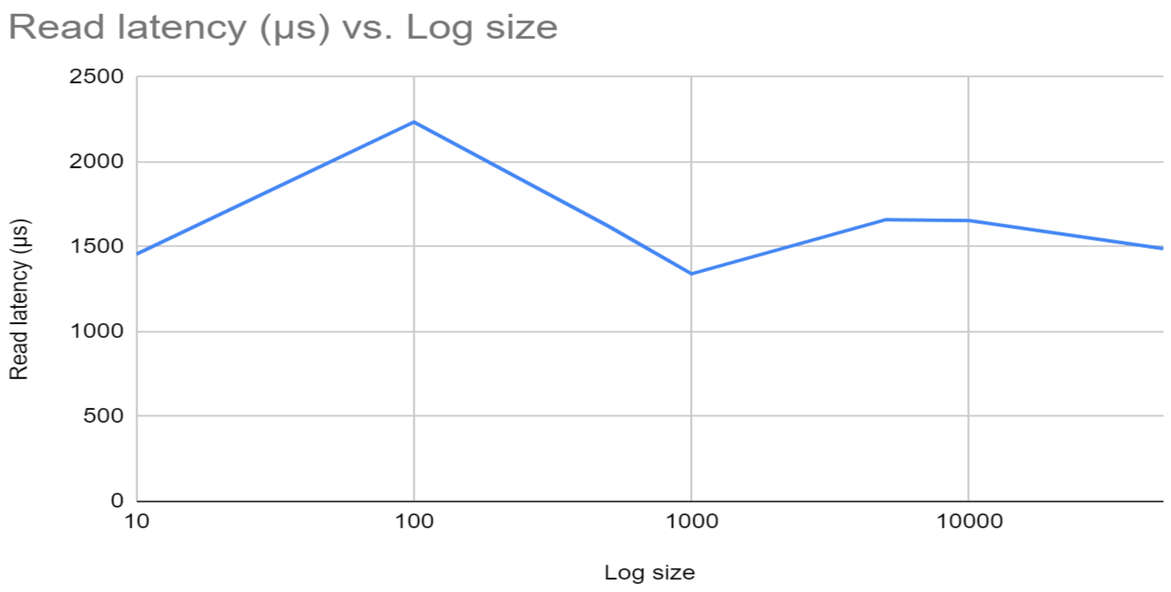
1. **Happy flow (with no anomalies or unexpected situations)** - Once the majority of servers are alive, one of the nodes becomes the leader. This particular node stays as the leader and the current term for this node does not change in case of no crash or no network failures.
2. **Availability (Leader is crashed randomly at some point of time)** - After the current leader crashes, other nodes become candidates (at different times because of randomized timer) and start leader election. Failover time is just above 5 seconds (election timeout = 5 seconds + random few milliseconds) and new leader is elected which starts serving client requests successfully.
3. **Consistency (Leader is crashed after modifying its own log but before replicating on followers) -** We test consistency by ensuring that if replication is unsuccessful on majority of followers and leader crashes, log of this node has to be corrected in future when it is alive again. Future/new leader will send the updated log information to this node.Its incorrect log entries will be deleted and new log entries will be appended.
4. **Partition tolerance** - We tested this by putting arbitrary delays in heartbeat RPCs between the leader and random sets of nodes. We observed that the system is unstable in that case and the leader node keeps changing. This happens because of the delay in heartbeat (which mimics network partition failure) which causes the affected nodes to start election. This is expected behavior because our system ensures availability and consistency, and follows CAP theorem.

Our video demonstrations for availability and consistency are uploaded at
https://drive.google.com/drive/folders/1B0ihsLa8zrKLqdmCx9wijvf6FAO93V8w?usp=sharing

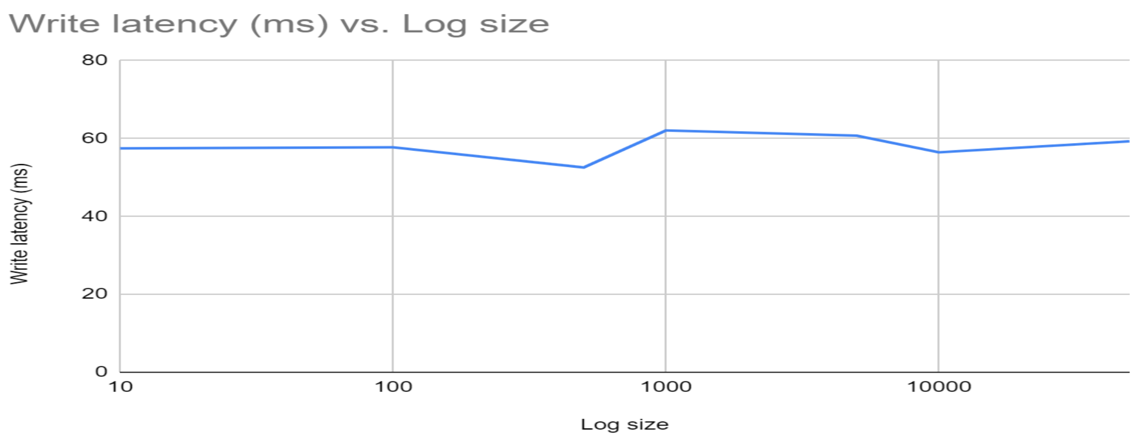
5. Performance Evaluation

Hardware:
For latency measurements, we used 6 Cloudlab C220g5 nodes connected on a 1Gbps network link. We ran 5 servers on 5 nodes. The client was run on the 6th node

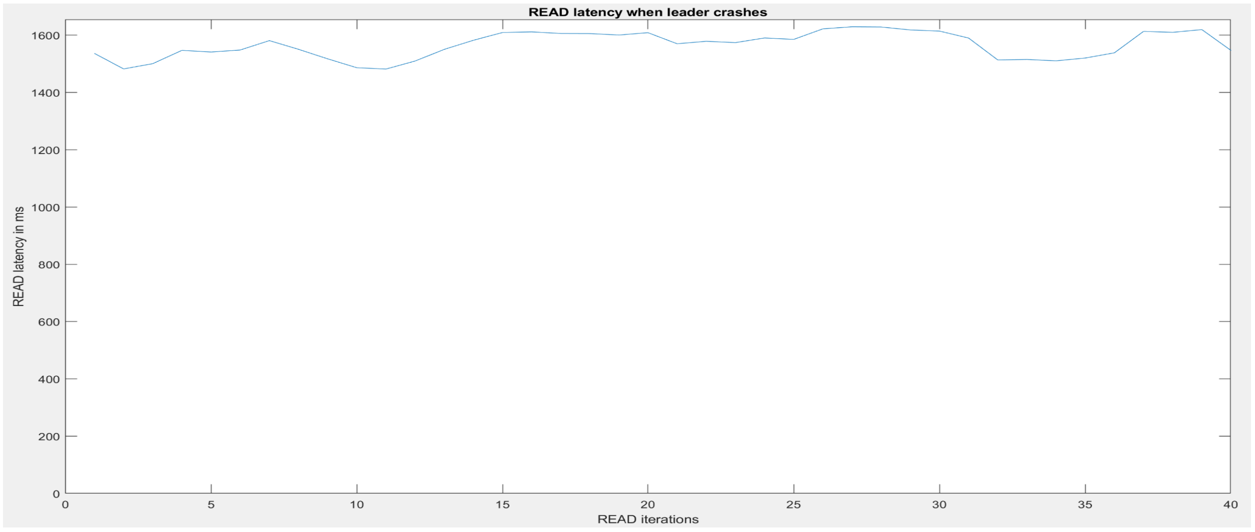
READ Latency:
We sent 1000 read and write requests to 5 different servers. The average read latency was 1826.222 microseconds.
In the case of 3 servers, we found our average read latency to be 1806.112 microseconds.



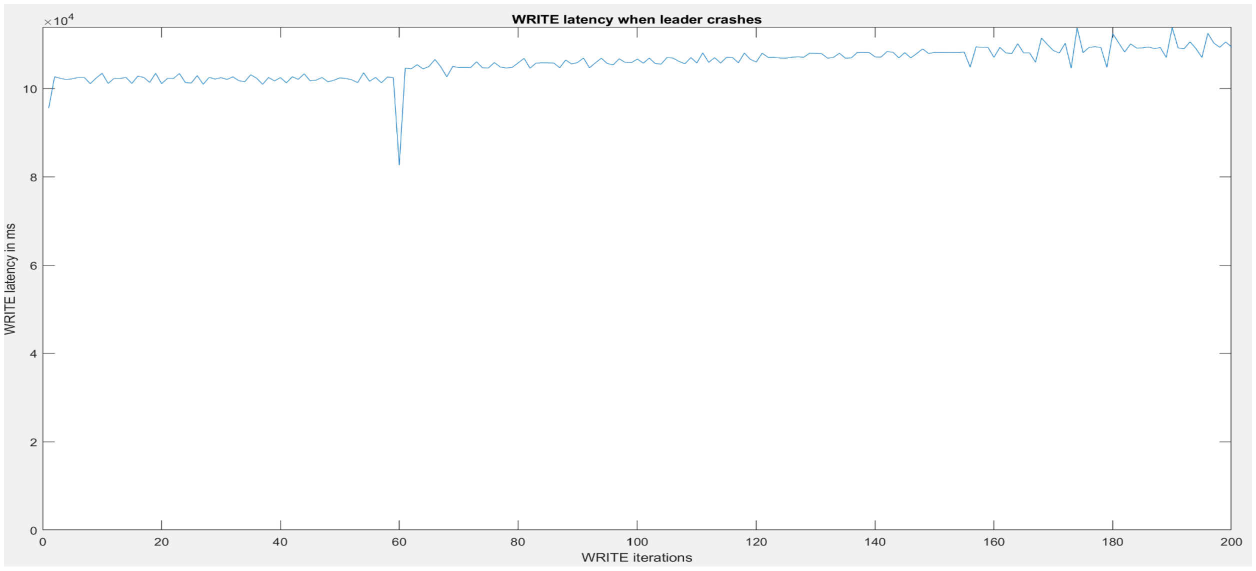
WRITE latency:
The average WRITE latency on 5 servers for 1000 requests was 54.987 milliseconds.
In the case of 3 servers, the WRITE latency was 52.811 milliseconds.



READ latency with crashes:
The average READ latency in case of follower crashes remains the same.
The average READ latency in case of leader crashes increases as election timeout happens and a new leader is elected
The failover time is on average 5.71s in case of 3 servers.
For READ latency readings in case of crash, we sent 1000 requests and then averaged it out to remove distorted readings. The following is the graph we got. As we can see, the peak point of latency is reached when the leader crashes.



WRITE latency with crashes:
The average WRITE latency in case of follower crashes remains the same.
The average WRITE latency in case of leader crashes increases as election timeout happens and a new leader is elected.



We are curious as to why our WRITE latency keeps on gradually increasing after the crash. We are investigating this issue.

Recovery Timeline:
This is the time it takes the leader to transfer the log to the follower after the follower comes back again. We ran multiple experiments with different sizes of logs and found the average time to be around 900 ms. This reading needs more rigorous experimentation.

6. Conclusion

We successfully implemented Raft consensus protocol for block store replication and ensured its correctness according to the paper we studied in the class. We tested and evaluated our Raft implementation on a distributed system of 3 and 5 nodes. We learnt about the challenges of implementing a consensus protocol in a distributed setting including RPC failures, computational complexity of leader election handling and persistence, ensuring availability and consistency simultaneously, and leader crash handling.