Assignment 2: Distributed log server

Markus Wilhelmsen, Tor Opdahl

March 2025

1 Introduction

The task of this assignment was to implement a consistent distributed log server that clients can connect to in order to store their log entries.
This solution to the assignment focuses on Raft, and its algorithm to achieve consensus between the nodes on the network. This report also assumes the reader already knows the details of Raft([1]), and will therefore not cover any background.

2 Design

2.1 Node Configuration

Each node has 2 threads running in parallel. One thread for running the server, and the other for managing the server. This essentially means that the thread "running" the servers job is to receive new requests from the network and client, whilst the other, who is managing the servers job is to monitor the network and send out necessary requests between the other nodes. This relation will be referred to as "server" and "manager", this relation together makes the Node.

2.2 Manager Thread

This threads responsibility changes based on its role in the network. If it is a leader, it will have other tasks and responsibilities than if it where a follower.

2.2.1 Leader

If the manager threads role is a leader, its responsibilities are; sending out "heartbeats" and send out replicas of its own log to other nodes on the network.

2.2.2 Follower

The managers responsibility when it is a follower is to monitor the timer related to last successful communication with leader. If this timer reaches its threshold,

the manager starts a new leader election, setting itself as candidate for this election.

2.3 Server Thread

The responsibilities of the server are also different based on the role the node has in the network.

2.3.1 Leader

When a servers role is leader, its responsibilities are to receive logs from the client and logs forwarded from its followers and store these in its own local log. Its also the leaders responsibility to detect that it has been replaced by another node.

2.3.2 Follower

The servers responsibilities as a follower is to receive heartbeats from the leader node, and respond accordingly to these requests, receive and forward logs sent from the client, vote accordingly during elections, and receive log updates from the leader.

3 Implementation

3.1 Manager Thread

3.1.1 State Management and Election

All nodes are initialized with different timers that decide when it will trigger a reelection if the leader fails to respond. When an election has started, the node going as candidate updates its term, votes for itself, and sends out requests to the other nodes stating that there is a new election. For a node to receive a vote from another node it must have:

- **Rule 1:** Term of last log must be equal to or grater.
- **Rule 2:** If rule 1 is equal, number of logs must be equal to or greater.

The node asking for votes first goes through its list from start to finish, and when it has collected enough votes, it stops sending out requests asking for more. It then states that it won the election by going though the list of nodes on the network in reverse order. This way the implementation minimizes the chance of nodes triggering new elections at the same time.
If a node does not get enough votes, it goes back to its follower role, and waits for the another node to start an election.
A node that won, will then move to the "leader loop", where all it does is just send out a heartbeat every 0.2 seconds. This loops condition is "while still leader", as the other thread is the one discovering when it is no longer the leader,

and changes role back to follower. It then goes back to the regular follower loop, where all it does is monitoring its timer.

3.1.2 Heartbeat Mechanism

A heartbeat the leader sends out consists of:

- Its own ID
- Its current term
- Number of Logs in its local log
- Term of last log in local log

The receiving node will either ignore the request if the leader is outdated, waiting for its or another nodes timer to expire, or check it its own log is outdated, if it is, the follower will respond with 209 instead of 200, indicating it wants an update.
The leader since it needs to constantly send requests to the other node, has a timeout on these outgoing requests to be 50 milliseconds, prioritizing volume of messages instead of consistent communication. If a node does not respond within this window, the leader continues as if everything is ok. The implementation relies on the assumption that if it is not crashed, it will respond most of the time on every heartbeat, so that if it is crashed and does not respond, the leader wastes minimal time.

3.1.3 Log Replication

Here the leader utilizes its matrix that has overview of the next and match index of all other nodes on the network, to know what logs it needs to send to the node that requested an update on its logs. Utilizing these indexes in the matrix tells the leader exactly what logs it needs to send to the other nodes for them to have the exact same log after the transaction is completed. After a successful update, the leader can now update the match and next index for this node in its matrix.

3.2 Server Thread

3.2.1 Request Handling

The server thread handles the following HTTP endpoints (only one endpoint for PUT, rest is POST):

- **PUT:** Forward to leader, or store incoming log if leader.
- **Heartbeat:** Process leaders state with term and log state, ignore if leader is outdated, ask for update if own log is outdated. Resolves old-leader conflicts if there are multiple nodes who are leaders simultaneously.

- **basicElection:** Handles incoming request for an election. If the node that sent the request is up to date, and follows the two rules, the vote will be granted.
- **newLeader:** Handles leader change when a new leader has been elected.
- **appendEntries:** Handles replication of the logs. Appends incoming logs from the leader.

All these different endpoints make up the request handling of the server thread.

3.2.2 Log Storage

The request sent to this endpoint contains all logs, to the best of the leaders knowledge, that is different between the leader and follower. This endpoint takes this information and stores it in its local logs.

3.2.3 Forwarding Logic

All followers will forward a PUT request from the client to the leader if the leader is available and the network has a leader. If the follower fails to forward, it will give a 503 respons to the server, as the current implementation has not implemented a buffer into each follower to store logs that the leader never received.

3.3 Thread Coordination

3.3.1 Shared State

The server-thread can access all necessary information that the manager thread uses to make its decisions. This is because the server class that the server thread uses is passed the RaftNode object in its initialization. The manager thread is reliant on the server thread updating the object when new information arrives.

3.3.2 Lock Management

Every time information is changed in the shared RaftNode object, locks are employed to avoid conflicts. When requests are sent, no locks are held, rather using the snapshot technique so that information stays consistent during broadcast and messaging, in addition to avoid deadlocks.

4 Testing

The logs are always the same in all nodes when the test is done. And almost always gets a success, when running scenario 0-2. The only thing is that it sometimes does not include all the logs sent by the client in scenario 3-4. So it will sometimes finish missing 1-3 logs. All testing has been done running with 5 nodes.

5 Discussion

The biggest reason for picking Raft as the consensus algorithm for this log server implementation is how easy the algorithm is to understand. It became the obvious choice after watching the video related to this paper ([1]). It portrayed the algorithm in such a good way, and sparked a lot of ideas on how to solve the assignment.
The implementation successfully maintains log consistency across the network even during node crashes and recoveries. The biggest challenge was handling leader transitions and ensuring log consistency during crashes. The current design achieves consistency at the cost of occasionally missing some logs during extreme failure scenarios (when multiple nodes crash simultaneously).

6 Conclusion

This implementation demonstrates that Raft provides a robust foundation for building consistent distributed systems. By separating server and manager responsibilities into different threads, the system achieves good separation of concerns while maintaining consistency guarantees. The implementation successfully replicates logs across all nodes in most scenarios, with only occasional log loss during extreme failure conditions.

References

[1] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, 2014. USENIX Association.