Fault-Tolerant Distributed Key-Value Store Using Raft

Zhiyuan Chen
Computer Science & Engineering
University of California, San Diego
San Diego, California
zhc074@ucsd.edu

Michael Liang
Electrical & Computer Egineering
University of California, San Diego
San Diego, California
mil118@ucsd.edu

***Abstract*—Our final project presents the design and implementation of a distributed key-value storage system built on the Raft consensus algorithm. Our implementation leverages the safety guarantees provided by the Raft protocol and Rust's memory safety guarantees and concurrency primitives to create a fault tolerant distributed system capable of maintaining a consistent state across multiple nodes despite network partitions or node failures. Through testing and demonstration scenarios, we validate the system's ability to maintain data integrity and availability under failure conditions such as node crashes.**

I. Introduction

Distributed systems enable modern cloud infrastructure to scale horizontally and maintain high availability, however, achieving consensus among distributed nodes still remains one of the fundamental challenges we face today. Traditional consensus algorithms like Paxos [3], while theoretically sound, have proven difficult to understand and implement correctly due to the complex protocol semantics, motivating the development of a more understandable alternative. The Raft consensus algorithm, introduced by Diego Ongaro and John Ousterhout in 2014, addresses this need by prioritizing understandability while maintaining strong technical foundations and performance. This project implements a distributed key-value storage system using the Raft algorithm to provide strong consistency guarantees. The system runs multiple server nodes locally, each capable of handling client requests and participating in the consensus protocol. Our work provides three main contributions.

- We build a key-value storage server that supports GET, PUT, APPEND and DELETE operations.
- We implement the core parts of Raft consensus algorithm including leader election and log replication to make the system fault tolerant.
- We conduct experiments to validate that the system can maintain consistency and availability under server node failures.

II. The Raft Consensus Algorithm

A. Algorithm Overview

In this project, we implemented the Raft consensus algorithm, first introduced in [4]. The algorithm makes the rule that at any given time, each server exists in one of three states: leader, follower, or candidate. In normal operation, exactly one server acts as the leader while all others remain followers.

The algorithm operates in terms of elections. Time is divided into arbitrary-length terms, each beginning with an election. If a candidate wins the election, it serves as leader for the remainder of the term. Terms act as a logical clock, allowing servers to detect obsolete information. Each server maintains the current term number, which increases monotonically over time and is exchanged in all RPCs between servers.

Table I summarizes the state variables maintained by each server in our implementation, distinguished between persistent state (that survives crashes) and volatile state (that can be reinitialized).

B. Leader Election

Servers begin in the follower state and remain passive until an election timeout occurs. A follower that times out assumes no viable leader exists, increments its current term, transitions to candidate state, votes for itself, and sends a RequestVote RPCs to all other servers. A candidate wins an election if it receives votes from a majority of servers. Each server votes for at most one candidate per term, with the restriction that the candidate's log must be at least as up to date as the voter's log. This majority rule ensures that at most one candidate can win an election for a particular term.

Table II shows the structure of the RequestVote RPC used in our implementation.

C. Log Replication

Once elected, the leader accepts client requests containing commands to be executed by the replicated state machines. The leader appends each command to its log as a new entry, then issues AppendEntries RPCs in parallel to replicate the entry. When the entry has been safely replicated to a majority, the leader applies the entry to its state machine and returns the result to the client. If followers crash or network packets are lost, the leader retries AppendEntries RPCs indefinitely. A log entry is committed when the leader has replicated it on a majority of servers. Raft guarantees that committed entries are durable and will eventually be executed by all available state machines.

TABLE I
RAFT STATE VARIABLES IN OUR IMPLEMENTATION

Persistent State (on all servers)	
`current_term`	Latest term server has seen (initialized to 0, increases monotonically)
`voted_for`	CandidateId that received vote in current term (or -1 if none)
`log[]`	Log entries; each entry contains command for state machine and term when entry was received by leader (first index is 1)
Volatile State (on all servers)	
`status`	Current server state: Leader, Follower, or Candidate
`commit_index`	Index of highest log entry known to be committed (initialized to 0, increases monotonically)
`last_applied`	Index of highest log entry applied to state machine (initialized to 0, increases monotonically)
`time_election`	Timestamp of last election timeout reset
`timeout_election_m`	Randomized election timeout duration (400-800ms)
`time_heartbeat`	Timestamp of last heartbeat sent (Leader only)
`kv_store`	HashMap storing the actual key-value pairs
Volatile State (on leaders, reinitialized after election)	
`next_index[]`	For each server, index of the next log entry to send to that server (initialized to leader's last log index + 1)
`match_index[]`	For each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

TABLE II
REQUESTVOTE RPC STRUCTURE

Arguments	
`term`	Candidate's term
`candidate_id`	Candidate requesting vote
`last_log_index`	Index of candidate's last log entry
`last_log_term`	Term of candidate's last log entry
Results	
`term`	Current term, for candidate to update itself
`vote_granted`	True if candidate received vote

Table III shows the structure of the AppendEntries RPC, which serves both for log replication and as a heartbeat mechanism.

TABLE III
APPENDENTRIES RPC STRUCTURE

Arguments	
`term`	Leader's term
`leader_id`	So follower can redirect clients
`prev_log_index`	Index of log entry immediately preceding new ones
`prev_log_term`	Term of prev_log_index entry
`entries[]`	Log entries to store (empty for heartbeat)
`leader_commit`	Leader's commit_index
Results	
`term`	Current term, for leader to update itself
`success`	True if follower contained entry matching prev_log_index and prev_log_term

D. Safety Properties

The Election Safety property guarantees that at most one leader can be elected in a given term. The Leader Append-Only property ensures that a leader never overwrites or deletes entries in its log. Most importantly, the State Machine Safety property guarantees that if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. This is achieved through restrictions requiring that a candidate's log must contain all committed entries before it can be elected leader.

III. IMPLEMENTATION IN RUST

A. Rationale for Choosing Rust

Our rationale for implementing the project in Rust was due to its unique combination of low-level control and high-level safety guarantees [2]. The language's ownership system eliminates entire classes of bugs at compile time, including data races, null pointer dereferences, and use-after-free errors, all of which are common pitfalls in concurrent systems programming.

Rust's type system enables fearless concurrency. The compiler enforces that shared mutable state cannot exist without proper synchronization, preventing race conditions that often result in issues with many distributed systems implementations. This guarantee is particularly valuable when implementing the state transitions and concurrent message passing required by Raft. If the code compiles, developers can have high confidence that fundamental concurrency bugs have been eliminated.

Furthermore, the language provides performance comparable to C and C++ while maintaining memory safety without a garbage collector. The existing crates in Rust offer highly useful components for distributed systems development. The Tokio runtime [5] provides a robust asynchronous environment that enables efficient concurrency, allowing our implementation to handle multiple simultaneous RPCs and timer events. The Tonic and Prost crates [1] offer RPC integration with the protobuf definitions, which simplify the communication between nodes while maintaining type safety and performance. Additionally, Rust's expressiveness through its type system

and pattern matching allows for natural representation of state machines and complex protocol logic.

B. Technical Implementation Details

In this section, we describe our design choices. We structure each Raft node as an independent process that communicates with peers through remote procedure calls. We leveraged Rust's `async`/`await` syntax with the Tokio runtime to handle concurrent network operations efficiently. The asynchronous model allows each node to manage multiple ongoing operations, including handling client requests, processing peer messages, and managing election and heartbeat timers, without blocking. The core state machine is implemented using Rust's `enum` types, which naturally express the three possible node states: `Leader`, `Follower`, and `Candidate`. Pattern matching on these enums ensures exhaustive handling of all possible states and transitions.

After a Raft server node is started, it will spawn a `ticker` thread and an `apply_command` thread simultaneously. The `ticker` thread checks heartbeat messages periodically. If the node hasn't received a heartbeat within the heartbeat timeout, it will become a candidate and start leader election. After a node wins the election, the `ticker` thread in the leader will instead broadcast heartbeat messages to all followers. We choose the interval between heartbeats to be 100 ms and the heartbeat timeout to be 400-800 ms to avoid overwhelming the network traffic and also avoid spurious elections.

Other than sending heartbeats to all followers, the leader node will spawn two more threads, which are `broadcast_new_command` and `check_commit`. In `broadcast_new_command`, the leader will check the next index of log entry each follower anticipates and send any newer log entries to the follower through `append_entries` RPC requests. The `check_commit` thread will examine the log entries in the leader server. If a log entry has been replicated to more than half of the total servers, the log entry will be committed. The leader will send the index of committed entries to followers, so followers can also check and commit their log entries. Then the `apply_command` thread in each server node will apply the command in committed log entries and update its KV storage accordingly.

The RaftState structure encapsulates all critical state for the consensus protocol:

```
struct RaftState {
    status: ServerType,
    current_term: i32,
    voted_for: i32,
    log: Vec<PbEntry>,
    commit_index: i32,
    last_applied: i32,
    next_index: Vec<i32>,
    match_index: Vec<i32>,
    timeout_election_ms: i64,
    time_election: Instant,
    time_heartbeat: Instant,
    kv_store: HashMap<String, String>,
}
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum ServerType {
    Leader,
    Follower,
    Candidate,
}
```

Listing 1. RaftState structure definition

The ticker function demonstrates how pattern matching enables clean state-based logic:

```
match st.status {
    ServerType::Leader => {
        let elapsed = st.time_heartbeat
            .elapsed().as_millis() as i64;
        if elapsed > TIMEOUT_HEARTBEAT_MS {
            st.time_heartbeat = Instant::now();
            do_heartbeat = true;
        }
        do_broadcast = true;
    }
    _ => {
        let elapsed = st.time_election
            .elapsed().as_millis() as i64;
        if elapsed > st.timeout_election_ms {
            st.current_term += 1;
            st.status = ServerType::Candidate;
            st.time_election = Instant::now();
            start_election = true;
        }
    }
}
```

Listing 2. State-based ticker logic with pattern matching

The RequestVote RPC handler demonstrates the integration of Tonic for type-safe gRPC communication:

```
#[tonic::async_trait]
impl Raft for RaftService {
    async fn request_vote(
        &self,
        request: Request<RequestVoteArgs>,
    ) -> Result<Response<RequestVoteReply>, Status>
        {
        let args = request.into_inner();
        let mut st = self.node.state.lock().await;

        if args.term > st.current_term {
            st.current_term = args.term;
            st.voted_for = -1;
            st.status = ServerType::Follower;
        }
        let last_log_index = st.log.len() as i32 -
            1;
        let last_log_term =
            st.log[last_log_index as usize].term;
        let mut vote_granted = false;
        if (st.voted_for == -1
            || st.voted_for == args.candidate_id)
            && (args.last_log_term > last_log_term
            || (args.last_log_term == last_log_term
            && args.last_log_index >= last_log_index
                ))
        {
            st.voted_for = args.candidate_id;
            st.time_election = Instant::now();
            vote_granted = true;
        }
        Ok(Response::new(RequestVoteReply {
            term: st.current_term,
            vote_granted,
        }))
    }
}
```

Listing 3. RequestVote RPC handler implementation

IV. Experiments and Demonstrations

A. Experimental Setup

We deployed a cluster of five Raft nodes running locally on different ports to simulate a distributed environment. The system exposes a simple key-value store interface supporting PUT, GET, and DELETE operations, with all write operations flowing through the Raft consensus protocol. To validate fault tolerance capabilities, we designed experimental scenarios targeting different failure modes and measured both correctness properties and system behavior under various conditions.

B. Server Failure and Recovery

Our primary demonstration involves deliberate server failure during active client operations. Starting with a stable three-node cluster processing client write requests, we terminate one server process abruptly to simulate a crash failure. The remaining nodes detect the leader's absence through heartbeat timeouts within 800 milliseconds. One of the remaining nodes times out first, initiates a new election by incrementing its term and requesting votes from peers. Leader elections consistently complete within 200-500 milliseconds, demonstrating Raft's ability to quickly restore availability. Once a new leader is elected, it begins accepting client requests and ensures that any uncommitted entries from the previous leader are properly handled according to Raft's safety rules. We verified that all client operations acknowledged before the leader's failure remained visible after the election, confirming Raft's fundamental safety guarantee.

```
Started servers: PID0=96604 PID1=96605 PID2=96606
Client sends some command
[2025-12-11T22:08:06Z INFO  raft_rs::raft_node] In term [0], server [0] becomes a candidate
[2025-12-11T22:08:06Z INFO  raft_rs::raft_node] Server [1] votes for candidate [0] in term [1]
[2025-12-11T22:08:06Z INFO  raft_rs::raft_node] Server [2] votes for candidate [0] in term [1]
[2025-12-11T22:08:06Z INFO  raft_rs::raft_node] In term [1], server [0] wins election and becomes the leader
Client 1: leader is http://127.0.0.1:6000
[2025-12-11T22:08:09Z INFO  raft_rs::raft_node] Server [0] applies command: index [1], kind=PUT, key=0, value=0
[2025-12-11T22:08:09Z INFO  raft_rs::raft_node] Server [1] applies command: index [1], kind=PUT, key=0, value=0
[2025-12-11T22:08:09Z INFO  raft_rs::raft_node] Server [2] applies command: index [1], kind=PUT, key=0, value=0
[2025-12-11T22:08:10Z INFO  raft_rs::raft_node] Server [0] applies command: index [2], kind=PUT, key=1, value=100
[2025-12-11T22:08:10Z INFO  raft_rs::raft_node] Server [1] applies command: index [2], kind=PUT, key=1, value=100
[2025-12-11T22:08:10Z INFO  raft_rs::raft_node] Server [2] applies command: index [2], kind=PUT, key=1, value=100
[2025-12-11T22:08:11Z INFO  raft_rs::raft_node] Server [0] applies command: index [3], kind=PUT, key=2, value=200
[2025-12-11T22:08:11Z INFO  raft_rs::raft_node] Server [2] applies command: index [3], kind=PUT, key=2, value=200
[2025-12-11T22:08:11Z INFO  raft_rs::raft_node] Server [1] applies command: index [3], kind=PUT, key=2, value=200
Kill server 2 (PID 96606)
./test_failure.sh: line 19: 96606 Terminated: 15          ./target/debug/server 2
Client sends more command
Client 2: leader is http://127.0.0.1:6000
Client receives value: 100
Restart server 2
[2025-12-11T22:08:19Z INFO  raft_rs::raft_node] Server [0] applies command: index [4], kind=APPEND, key=1, value= 2947
[2025-12-11T22:08:20Z INFO  raft_rs::raft_node] Server [2] applies command: index [1], kind=PUT, key=0, value=0
[2025-12-11T22:08:20Z INFO  raft_rs::raft_node] Server [2] applies command: index [2], kind=PUT, key=1, value=100
[2025-12-11T22:08:20Z INFO  raft_rs::raft_node] Server [2] applies command: index [3], kind=PUT, key=2, value=200
[2025-12-11T22:08:20Z INFO  raft_rs::raft_node] Server [2] applies command: index [4], kind=APPEND, key=1, value= 2947
[2025-12-11T22:08:20Z INFO  raft_rs::raft_node] Server [1] applies command: index [4], kind=APPEND, key=1, value= 2947
[2025-12-11T22:08:26Z INFO  raft_rs::raft_node] Server #0 has entry: Key 0, Value 0
[2025-12-11T22:08:26Z INFO  raft_rs::raft_node] Server #0 has entry: Key 2, Value 200
[2025-12-11T22:08:26Z INFO  raft_rs::raft_node] Server #0 has entry: Key 1, Value 100 2947
[2025-12-11T22:08:26Z INFO  raft_rs::raft_node] Server #1 has entry: Key 1, Value 100 2947
[2025-12-11T22:08:26Z INFO  raft_rs::raft_node] Server #1 has entry: Key 2, Value 200
[2025-12-11T22:08:26Z INFO  raft_rs::raft_node] Server #1 has entry: Key 0, Value 0
[2025-12-11T22:08:39Z INFO  raft_rs::raft_node] Server #2 has entry: Key 1, Value 100 2947
[2025-12-11T22:08:39Z INFO  raft_rs::raft_node] Server #2 has entry: Key 2, Value 200
[2025-12-11T22:08:39Z INFO  raft_rs::raft_node] Server #2 has entry: Key 0, Value 0
Kill all servers and exit
./test_failure.sh: line 34: 96604 Terminated: 15          ./target/debug/server 0
./test_failure.sh: line 35: 96819 Terminated: 15          ./target/debug/server 2
```

Fig. 1. System logs demonstrating leader election, command replication, and automatic state synchronization after a server failure and restart.

Figure 1 shows the output of our experiment. In this experiment, we start three Rafts servers. After a client sends some commands, we kill one of the servers. Then the client sends more commands and we reconnect the server. We dump the entries in each server's KV storage at the end of the test. We can find that the system keeps providing service to the client in the whole process, and all three servers have the same log entries at last. This test demonstrates the fault tolerance property of the system.

V. Conclusion and Future Work

This project demonstrates the implementation of a fault-tolerant distributed key-value store using the Raft consensus algorithm in Rust. The system provides strong consistency guarantees while maintaining high availability in the face of node failures and network partitions. Our experimental results validate Raft's theoretical properties in practice, showing rapid leader election, correct log replication, and guaranteed durability of committed operations.

For future improvement, we plan to implement log compaction and batched logging. The log compaction technique takes a Snapshot of the system state at a certain timestamp and all logs before the timestamp can be aborted. This technique saves disk space and allows the system to operate for extended periods. Batched logging allows the server to wrap multiple commands in a single log entry. In this way the overhead for sending RPC requests can be reduced and the system throughput can be improved.

References

[1] Contributors, T. Tonic: A native grpc client & server implementation with async/await support. https://docs.rs/tonic/latest/tonic/, 2025. Accessed: 2025-12-11.

[2] Klabnik, S., and Nichols, C. *The Rust Programming Language*, 2 ed. 2023.

[3] Lamport, L. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[4] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), pp. 305–319.

[5] The Tokio Project. Tokio: A runtime for writing reliable asynchronous applications with rust. https://tokio.rs/, 2024. Accessed: 2024-12-10.