



Reaching a Consensus

Comparing RAFT and ZAB



Final Project Presentation
Distributed Systems: CSD438



What is Consensus?

- Consensus = agreement among distributed nodes on a single value or sequence of actions
- Ensures the system behaves like one reliable machine despite crashes or message delays
- Critical for replicated databases, config stores, and distributed coordination

Two Key Properties:

- Safety: No two nodes ever decide different values
- Liveness: The system eventually reaches a decision

Achieved through majority voting (quorums) to maintain overlap between decisions



RAFT: made for clarity and implementability

Server States:

- Leader
- Follower
- Candidate

Key Components

- Leader Election: Random timeouts → candidate → majority votes → leader
- Log Replication: Leader appends entries → replicates to followers → commit on majority
- Safety: No conflicting commits; leader contains all committed entries

Purpose: Replicate a log consistently across servers, even with failures



Implementation

Timing Parameters:

- Heartbeat: 50ms intervals
- Election timeout: 150-300ms (randomized)
- RPC timeout: 100ms
- Reset on: AppendEntries or RequestVote





Election Process:

1. Timeout fired → Follower becomes Candidate
2. Increment term, vote for self, reset timer
3. Send RequestVote RPCs in parallel to all peers
4. Vote granted if: candidate's log \geq up-to-date as voter's log
5. Majority received → Become Leader, start heartbeats
6. Split vote → Timeout and restart with new random interval

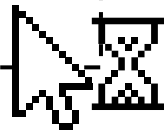


Log Replication:

- Leader appends to own log
- Sends AppendEntries to all followers
- Tracks match_idx for each follower
- Commits when replicated on majority

State Management:

- Persistent: current_term, voted_for, logs[]
- Volatile (all): commit_idx, last_applied
- Volatile (leader): next_idx[], match_idx[]
- State machine: In-memory key-value store





ZAB: Zookeeper Atomic Broadcast

System Model:

- One Leader, multiple Followers
- All writes by Leader
- ZXID prefixes
- Recovery and Broadcast Phases

Guarantees

- Total order: All servers deliver transactions in same sequence
- Prefix consistency: No unacknowledged transactions committed
- Leader stability: Leader never commits from previous epoch

Ensures consistent, totally ordered broadcast of updates



Implementation

State, ZXID Management

- States: LOOKING, FOLLOWING, LEADING
- ZXID: $(\text{epoch} \ll 32) \mid \text{counter}$
- Epochs: `accepted_epoch`, `current_epoch`
- Transactions: `(zxid, key, value)` tuples
- Tracking: `last_zxid`, `last_committed`





Transaction Broadcasting:

Propose Phase:

- Leader increments counter in current epoch
- Creates new ZXID = (epoch << 32) | counter
- Applies to local state machine (optimistic)
- Broadcasts ProposeTransaction to all followers

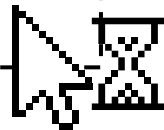


Commit Phase:

- Followers append to transaction log and ACK
- Leader broadcasts CommitTransaction
- All nodes apply to state machine

Recovery Phase:

- Enter LOOKING state on timeout
- Exchange FLE messages with peers
- Update vote if better proposal seen
- Majority agreement → leader elected
- New epoch established





State and ZXID:

- States: LOOKING, FOLLOWING, LEADING
- ZXID: (epoch << 32) | counter
- Epochs: accepted_epoch, current_epoch
- Transactions: (zxid, key, value) tuples
- Tracking: last_zxid, last_committed

Leader Election

- Vote format: (node_id, epoch, zxid, state)
- Comparison: Higher epoch wins
- Tie-break 1: Higher zxid wins
- Tie-break 2: Higher node_id wins
- Timeout: 150-300ms randomized



Tech Stack

Why Not
Alternatives?:

- Not Rust/C++:
Development
speed > memory
safety
- Not Go: (we
don't know it)

GRPC

- Mirrors real-world distributed systems
- Production-ready library support
- No complicated serialization delays
- Built over TCP/IP
- Efficient and reliable

Docker Orchestration: Containerized deployment for cluster management



Each Node Includes

- Consensus Engine: Runs RAFT or ZAB Logic
- Persistent Log: Durable log maintenance
- Comms: GRPC for RPC communication
- Timer Subsystem: Elections and Heartbeats
- Client Interface: SetVal/GetVal
- State Machine: KV Store





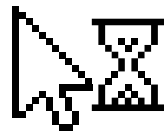
Benchmark Client

- Automated load generation
- Configurable R/W ratio
- Latency and throughput tracking
- Metric Reporting
- Fault Injection

Interactive REPL

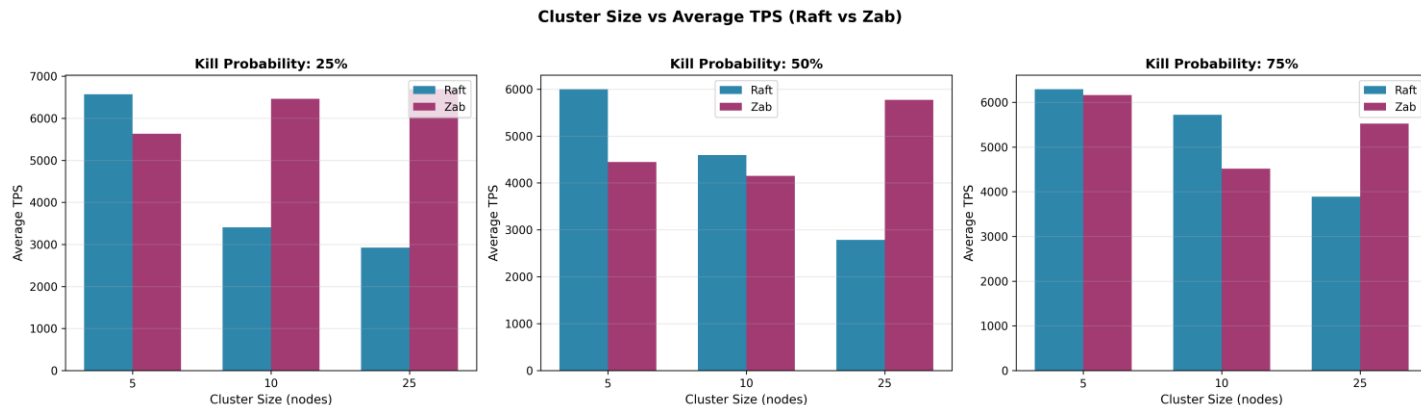
- Manual testing
- GetVal/SetVal commands
- Node Control
- Debugging and Inspection
- Status Monitoring

Request Flow: Random Node -> Check if leader -> Redirect if not -> Execute operation





Benchmark Results



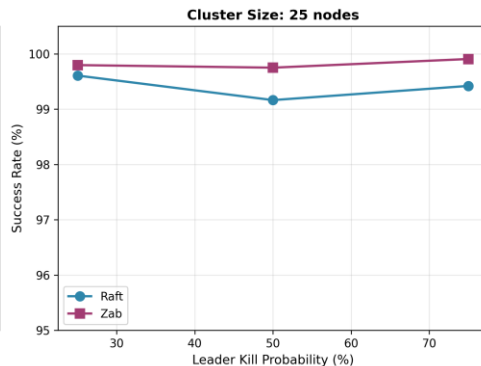
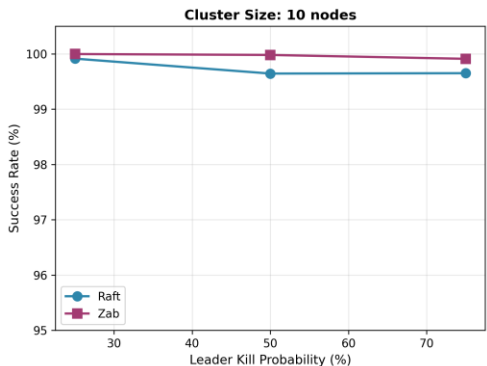
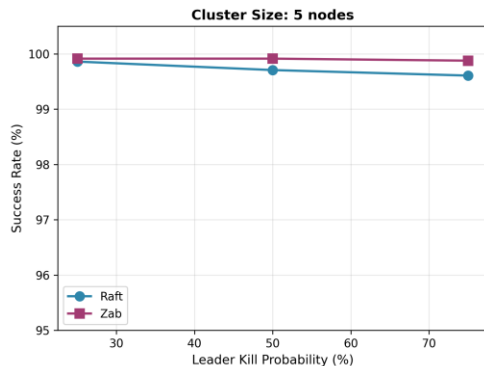
Raft performance degrades with larger clusters while ZAB remains decently consistent





Benchmark Results

Leader Kill Probability vs Success Rate (Raft vs Zab)

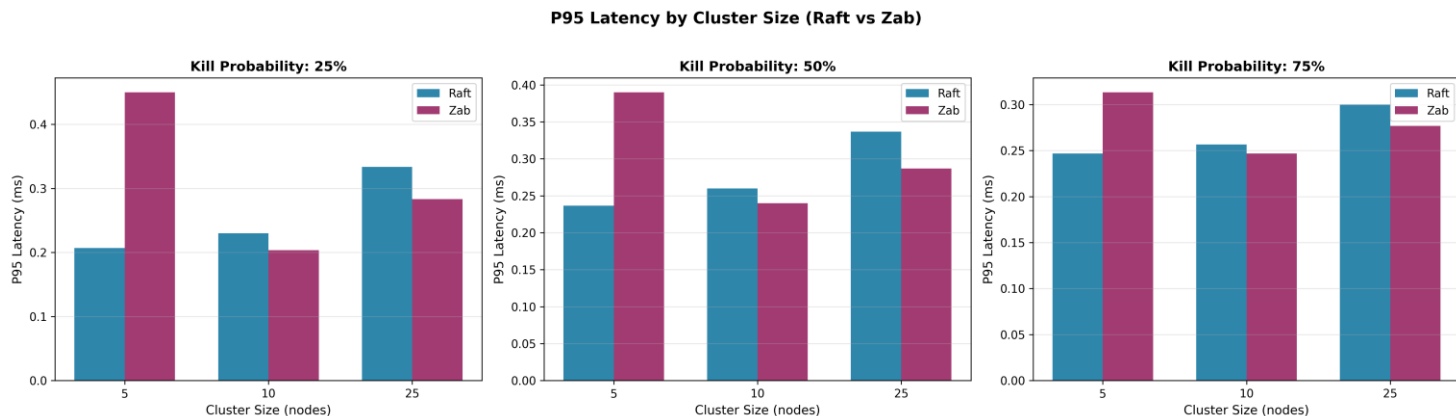


Both algorithms remain similarly performant with varying degrees of leader crashes





Benchmark Results



Latencies tend to increase as cluster sizes increase for each algorithm. Increase in latency is largely consistent for Raft. ZAB tends to perform better on medium sized clusters.

