REYKJAVIK UNIVERSITY

DISTRIBUTED CONSENSUS

CONCURRENT AND DISTRIBUTED SYSTEMS – T419-CADP

Róbert Logi Jónsson
robertj22@ru.is

Snorri Bjarkason
snorrib22@ru.is

Veigar Elí Grétarsson
veigarg22@ru.is

Teacher: Dr. Marcel Kyas
Teacher Assistant: Ilham Ahmed Qasse

April 3, 2025

TABLE OF CONTENTS

BREAKDOWN OF RAFT SERVER NODES

PROTOCOL MAPPING & JUSTIFICATION

LEADER ELECTION

The process of electing a new leader in Raft is fundamental to ensuring that the cluster remains available and consistent, even when servers fail. This process begins when a follower transitions into the candidate state after an election timeout. In our implementation, this timeout is monitored by the monitorElectionTimer() function, which continuously checks whether the server has received an AppendEntries RPC (heartbeat) from a valid leader. If no such message is received within the randomized election timeout interval, the follower concludes that the current leader may have failed and begins a new election.

At this point, the server promotes itself to a candidate, increments its currentTerm, votes for itself, and sends out RequestVote RPCs to all other peers in the cluster. This behavior mirrors the election process outlined in Section 5.2 of Ongaro and Ousterhout's Raft paper, where it is explicitly stated that a candidate must gather a majority of votes to be elected. The candidate includes its currentTerm, lastLogIndex, and lastLogTerm in each vote request, so that recipients can make an informed decision about log recency, which is essential to maintaining Raft's safety guarantees.

Incoming vote requests are handled by the handleRequestVote() function. This function only grants a vote if the term in the request is equal to or greater than the receiver's current term and the candidate's log is at least as up-to-date as the receiver's own log. Additionally, a server only grants one vote per term, as stored in the votedFor variable. These checks strictly follow the criteria in the Raft protocol to prevent split-brain situations or electing a leader with outdated logs.

When vote responses are received, they are processed by the handleVoteResponse() function. This function tallies the votes, and once a candidate has secured votes from a majority of servers in the cluster, it transitions to the leader state. Upon becoming the leader, the server initializes two key maps: nextIndex[] and matchIndex[]. These maps are used to track log replication progress for each follower, and they are initialized exactly as described in the paper, with nextIndex set to the leader's last log index plus one, and matchIndex set to zero.

This entire leader election sequence has been implemented in accordance with the Raft protocol. It respects randomized timeouts to reduce the chance of split votes, enforces strict voting rules based on log freshness, ensures that servers only vote once per term, and only promotes a candidate to leader after receiving a majority. By doing so, the implementation guarantees that at most one leader is elected per term and that this leader has the most up-to-date log, thereby maintaining consistency and correctness across the system.

LOG REPLICATION

Once a server successfully becomes the leader through a majority vote, it immediately takes responsibility for maintaining the stability and consistency of the cluster by sending out periodic heartbeat messages. In Raft, these heartbeats are actually empty AppendEntries RPCs that are broadcast to all other servers to assert the leader's authority and prevent new elections. This mechanism is implemented in our code through the sendHeartbeats() function, which runs continuously as long as the server remains the leader. It operates by iterating over all other servers and sending either heartbeats or actual log entries, depending on whether new entries have been added to the leader's log since the last communication with each follower. This

aligns exactly with the heartbeat behavior described in Section 5.2 and 5.3 of Ongaro and Ousterhout's Raft paper.

When a new client command arrives at the leader, it is first appended to the leader's local log via the appendToMemoryLog() function. However, this entry is not considered committed until it has been replicated to a majority of the cluster. To replicate log entries, the leader includes them in AppendEntries RPCs and sends them alongside the current term, leaderId, the prevLogIndex and prevLogTerm, and the leaderCommit index. The previous log index and term are critical to ensuring the follower's log is consistent with the leader's. This exact structure is reflected in our code when constructing the AppendEntriesRequest inside the sendHeartbeats() function.

On the follower side, incoming AppendEntries RPCs are processed by the handleAppendEntriesRequests() function. As described in the paper, the follower first checks whether the incoming term is at least as high as its current term. If it is not, the request is immediately rejected. If the term is valid, the follower then compares the prevLogIndex and prevLogTerm from the request with its own log. If it does not contain an entry at that index, or if the term at that index does not match, the follower rejects the request. This is a crucial step in enforcing log consistency and is directly aligned with rule 2 and rule 3 from Section 5.3 of the Raft paper.

When a follower detects a conflict, meaning it has a log entry at the given index, but the terms do not match, it deletes that entry and all entries that follow it. This is implemented in our system using the deleteConflictingEntriesFromMemory() function, which trims the in-memory log to remove any entries from the point of conflict onward. This ensures that any diverged or outdated log entries are safely discarded before new ones are appended, as specified in the protocol.

Once the logs are verified to be consistent or have been cleaned up appropriately, the follower appends any new entries from the leader using the appendToMemoryLog() function. These entries are only appended if they do not already exist or if they replace conflicting entries, preserving both safety and idempotency. After entries are appended and if the leaderCommit value is higher than the follower's current commitIndex, the follower updates its commitIndex and applies all committed entries to the state machine by writing them to disk using commitEntriesUpToIndex().

This implementation ensures log replication is safe and deterministic. Entries are only accepted if they extend a matching prefix of the log, and mismatches are corrected by removing conflicting entries. This design enforces Raft's central idea: that logs must be consistent before they can be committed. By following these steps, our code mirrors the Raft protocol's requirement that a log entry is only committed once it is stored on a majority of the servers and only if it extends a log that is already known to be correct. This guarantees the safety property of Raft and ensures that all nodes eventually converge on the same sequence of log entries.

COMMITMENT AND APPLYING LOGS

A critical component of the Raft consensus algorithm is determining exactly when a log entry can be considered committed. This distinction ensures that commands replicated across the cluster are not just stored, but also reliably agreed upon before they affect the system's state. In our implementation, this commitment logic is handled primarily through the tryAdvanceCommitIndex() function. On the leader side, the function periodically scans for log indices that have been replicated on a majority of servers (tracked through the matchIndex[] array) and, crucially, confirms that the log entry at that index was added during the leader's current term. This final check, that the entry's term equals the current term, is a requirement in

Raft's safety model (as defined in Section 5.4 of the paper) and ensures that leaders cannot accidentally commit entries from older terms, even if those entries happen to be widely replicated.

Once the leader determines that an entry is safely committed, it proceeds to apply that entry to the replicated state machine. In this assignment a local file serves as a replacement or a substitute to the state machine, where the command is written in the format defined for each server. This is implemented in the flushCommittedEntriesToFile() function, which takes care of persisting all committed log entries in order up to the current commitIndex. This design ensures durability and proper ordering, as log entries are only written once they meet Raft's commitment criteria.

Followers also participate in this commitment mechanism. Each time they receive an AppendEntriesRequest from the leader, they compare their current commitIndex with the LeaderCommit field. If the leader's commit index is greater, the follower updates its own commit index to the minimum of the leader's commit index and the index of the last new entry it received in that same request. This logic, which we haveve implemented inside the handleAppendEntriesRequests() function and executed through commitEntriesUpToIndex(), directly matches the follower behavior described in Section 5.3 of the paper. It ensures that followers only apply entries that the leader has committed, and that they never get ahead of what the leader considers safe.

By respecting the separation between replication and commitment, and ensuring that committed entries are only ever applied once they are durable and agreed upon by a majority, this implementation upholds the key guarantees of Raft. Log entries are applied in order, only once, and only once the system has reached consensus, preserving both consistency and safety across the cluster.

SAFETY AND TERM MANAGEMENT

Safety in Raft is guaranteed through strict handling of term numbers and well-defined rules about how servers transition between roles. In our implementation, this is enforced by always comparing the term value received in any RPC message with the server's own currentTerm, and immediately reacting if a higher term is observed. This comparison and reaction logic is present in both handleRequestVote() and handleAppendEntriesRequests(), which are responsible for processing Raft's two core message types. If a server receives a message with a higher term than its own, it updates its currentTerm to match and steps down to the follower state if it is currently a candidate or leader. This behavior directly reflects the requirements outlined in Section 5.1 of the Raft paper, where any server must always yield to messages carrying a higher term, ensuring term monotonicity across the system.

Another critical aspect tied to safety is vote management during elections. When a server observes a higher term, it not only updates currentTerm and changes roles, but also resets its votedFor field. This reset is crucial to ensure that votes are not incorrectly carried over between terms, which could otherwise result in multiple candidates receiving votes in the same election round. This logic maintains the protocol's core invariant: that in any term, at most one leader can be elected.

The code also carefully manages state transitions to prevent inconsistencies or illegal behavior. For example, if a candidate receives an AppendEntries RPC from a valid leader with a term equal to or greater than its own, it will immediately abandon its candidacy and revert to the follower role. This prevents the situation where a server believes it is a candidate or leader while the cluster has already converged under another leader, a key scenario that Raft is designed to avoid. In practice, this logic is handled within handleAppendEntriesRequests() by

checking the term and triggering the transition back to follower when appropriate. At the same time, it resets the election timer to reflect the fact that a valid leader is present and communicating, thus preventing unnecessary elections from being triggered.

Together, these rules enforce Raft's safety guarantee that the cluster cannot diverge or elect more than one leader per term. They also ensure that the replicated state machine on each node can only apply commands that are part of a safely agreed-upon log. By consistently enforcing term progression, resetting votes properly, and ensuring timely role transitions, the implementation preserves the core safety properties of Raft while avoiding edge cases that could otherwise lead to split-brain scenarios or inconsistent state across nodes.

CLIENT REQUESTS AND LOG FLOW

Client commands in this implementation are managed through the handleCommandFromClient() function, which serves as the entry point for any request issued by a client. This function is designed to align with the core Raft requirement that only the leader may accept and log new commands. As such, when a command is received, the function first checks the current state of the server. If the server is a follower or has temporarily failed, it does not attempt to process the request directly. Instead, it forwards the command to the known leader using a CommandName message. This behavior ensures that clients do not unknowingly send state-changing requests to the wrong node and reflects the redirection pattern outlined in Section 5.3 of Ongaro and Ousterhout's paper.

If the server is indeed the leader, it proceeds to handle the command by appending it to its in-memory log using the appendToMemoryLog() function. This function records the command along with the current term and the next available index. Importantly, as stated earlier, this alone does not commit the entry. Replication and commitment occur later via AppendEntries RPCs and the associated commit logic. However, the initial reception and logging of the command is entirely managed by handleCommandFromClient(), making it a key point where client intent is transformed into Raft log structure.

A notable simplification in this implementation is that no confirmation or response is ever sent back to the client, regardless of whether the log entry is ultimately committed (simplification in the assignment). This means that while commands follow the correct flow through Raft's machinery, clients remain unaware of their success or failure. Although this does not violate Raft's safety model, it does omit the usual feedback mechanism that many Raft-based systems rely on to notify clients once their command has been committed and applied to the state machine. In this case, the focus is on correctness and internal replication behavior rather than client-side guarantees.

By routing all client commands through the handleCommandFromClient() function and ensuring that only the leader initiates log changes, the implementation remains faithful to Raft's consistency model. It preserves the single-leader assumption and guarantees that all state-changing operations go through the proper Raft pipeline, even though user-facing acknowledgment is not part of the current design.

REQUEST-RESPONSE TRACKING

The mechanism for tracking requests and ensuring that each receives a corresponding response relies on careful coordination between log replication, state management, and acknowledgment messages. When a client submits a command to a server, the system guarantees that this request will eventually be either committed and applied or rejected, depending on the server's role and the state of the cluster. The implementation ensures that a response is only sent once the safety and consistency conditions of the Raft protocol are met.

When a client issues a command, it may contact any server in the cluster. If the receiving server is a follower or in failed state, it cannot directly process the request. In this case, it forwards the request to the current leader. This forwarding is handled via a CommandName message that is marshaled and sent to the known leaderID. If the server receiving the command is itself the leader, it immediately appends the command as a new log entry to its in-memory log, increments the lastLogIndex, and updates its nextIndex and matchIndex values for itself. If the server is in candidate state it will drop the message, therefore not forwarding it to the follower.

To make sure that the log entry is safely replicated, the leader initiates the log replication process by sending AppendEntries RPCs to all followers (in a heartbeat). These messages include the new entry and metadata such as the previous log index and term, allowing followers to detect inconsistencies and align their logs with the leader's. Followers validate the AppendEntries request and append new entries if they do not conflict with existing ones. If they accept the new entry, they send back a response indicating success.

The leader tracks these responses using the matchIndex and nextIndex maps. For each follower, the matchIndex records the highest log index that the leader knows has been replicated on that follower. After each successful AppendEntriesResponse, the leader updates this index. Once a a majority of the servers have acknowledged replication of a given entry, the leader can safely advance its commitIndex. This is done by checking whether there exists an index N such that $N > commitIndex$, a majority of $matchIndex[i] \geq N$, and $log[N].term == currentTerm$. This condition makes sure that only entries from the current leader's term are committed, preserving the Leader Completeness property of Raft.

Once the commitIndex advances, the leader applies the corresponding log entries to the state machine. In our implementation, the state machine is represented by a file named using the format server-host-port.log, which serves as a substitute for an actual state machine. It is only after the entries are applied, by writing them to this file, that the leader considers the client request successfully processed.

On follower nodes, a similar mechanism applies for applying entries to the state machine. When a follower receives an AppendEntries RPC with a leaderCommit value greater than its own commitIndex, it updates its commitIndex and proceeds to apply all entries up to that index. This makes sure that all committed entries are eventually applied on all servers, maintaining a consistent state across the cluster.

In this design, every client request is tied to a specific log entry that must be committed and applied before a response is given. By relying on majority agreement for commitment and by applying entries in order, the implementation makes sure that every request is either acknowledged with a successful application or not applied at all, thereby ensuring correctness, consistency, and liveness. This mechanism is essential for upholding Raft’s safety guarantees and providing clients with reliable, consistent responses in the presence of failures, network delays, or leader changes.

HEARTBEAT & ELECTION MECHANISM

In our implementation the heartbeat mechanism plays a central role in maintaining the leadership status of the current leader and ensuring the overall liveness of the system. This mechanism is implemented by having the leader periodically broadcast AppendEntries RPCs to all other servers in the cluster, regardless of whether there are new log entries to replicate. These heartbeat messages function as signals of the leader’s continued activity and are essential for suppressing unnecessary elections by informing followers that a valid leader is still operational.

The leader sends heartbeats at a fixed interval, defined by the constant heartbeatInterval, which is significantly shorter than the minimum possible election timeout used by followers. This guarantees that under normal network and processing conditions, a follower will receive heartbeats well before its election timer expires. The periodic sending of heartbeats is handled by the sendHeartbeats function, which runs in its own goroutine and iterates through all follower servers, constructing and transmitting AppendEntries RPCs. These RPCs include the leader's current term, its identifier, the index and term of the log entry preceding new entries (or the last entry if no new entries are being sent), an array of new log entries (which may be empty), and the leader's commitIndex. Even if there are no new entries to replicate, an empty AppendEntries message suffices to function as a heartbeat.

On the follower side, each server maintains an election timer initialized with a randomized duration between a defined minimum and maximum value (specified by electionTimeIntervalMin and electionTimeIntervalRandMax). This randomness helps avoid split votes in case multiple followers initiate elections simultaneously. Each time a follower receives a valid AppendEntries message from a leader whose term is equal to or greater than the follower's current term, the follower resets its election timer, remaining in the follower state and thereby acknowledging the leader's legitimacy.

If however a follower's election timer expires without receiving a valid heartbeat (which could happen due to a leader crash, network partition, or prolonged communication delay), the follower transitions to the candidate state and initiates a new election. This transition includes incrementing its current term, voting for itself, resetting its election timer, and broadcasting RequestVote RPCs to all other servers. A new leader is elected once a candidate receives a majority of votes. This design make sure that the system can recover quickly from leader failures while maintaining strong safety guarantees.

Through the careful coordination of regular heartbeat emissions by the leader and responsive election timer resets by followers, our implementation gurantees that a functioning leader maintains its authority and that new elections are triggered only when necessary. This mechanism adheres strictly to the Raft protocol's specification, preserving its safety, liveness, and fault-tolerance properties in both stable and failure-prone environments.

PROGRESS & DEADLOCK FREEDOM

PROGRESS GUARANTEES

Making sure that the system eventually makes progress is one of the core promises of the Raft protocol. It guarantees that as long as a majority of servers are alive and can communicate with each other, the system will not stall indefinitely and will continue processing client commands. This guarantee is realized in our implementation through careful use of election timeouts, randomized timers, and regular heartbeat broadcasts. Together, these mechanisms ensure that a leader will eventually be elected, stay in power as long as it is healthy, and that log entries will eventually be committed and applied to the system state.

A major factor enabling this liveness is the use of randomized election timeouts, defined by the constants electionTimeIntervalMin and electionTimeIntervalRandMax, which are set to 150ms and 150ms respectively in our code. The result is that each server will start a new election after a randomized delay between 150ms and 300ms, if no valid leader heartbeats are received in that time. This randomness is essential, it significantly reduces the chance that multiple servers become candidates at the same moment and split the vote, which would prevent any of them from winning. Instead, one server is likely to timeout slightly earlier than the others and begin an election unchallenged, increasing the chance of a quick, clean election result.

The electionTimeoutMin and electionTimeoutRandMax parameters control the total duration the system is willing to wait for a successful election to complete. If an election does not conclude within this window, the candidate automatically reverts to follower, resets its state, and prepares for the next election cycle. This mechanism ensures that the system doesn't hang if an election fails or becomes indecisive, it simply retries after a safe delay, eventually electing a leader once enough nodes are healthy.

Once a server is elected leader, it takes responsibility for maintaining progress in the cluster by regularly asserting its authority through heartbeat messages, which are implemented as empty AppendEntriesRequest RPCs. These are sent at fixed intervals defined by heartbeatInterval, set to 75ms in our implementation. The leader sends heartbeats to every follower at this interval to prevent them from starting new elections. Each heartbeat also serves as a carrier for new log entries if there are any to replicate. Because these heartbeats are sent more frequently than the minimum election timeout, they ensure that followers will never mistake an active leader for a failed one, as long as the network is functioning.

Critically, each time a follower receives a heartbeat, it resets its election timer using resetElectionTimer(), restarting the randomized countdown. This creates a feedback loop that continually confirms the leader's presence and stabilizes the cluster. However, if a leader crashes or becomes disconnected, the followers will stop hearing from it. As the election timers expire independently (due to randomization), one follower will likely start a new election before the others, ensuring that the cluster continues forward by electing a new leader.

Additionally the monitorElectionTimer() function ties all of this together by continuously checking if the election timer has expired and, if so, triggering startElection() to begin the leader election process.

Finally, client requests also contribute to progress via handleCommandFromClient(). Although our implementation does not currently include explicit client responses, commands submitted to the leader are logged, replicated, and eventually committed by reaching the majority of servers. Once committed, they are applied to the state machine, represented here by a local file. As long as a leader exists and a majority is functional, these entries will propagate and take effect, preserving the client's intent.

DEADLOCK FREEDOM

Our Raft implementation was carefully designed with concurrency in mind, ensuring that all shared state is safely accessed while also avoiding deadlocks. To guarantee this, we combined strict lock acquisition ordering, minimal lock nesting, disciplined use of goroutines, and bounded non-blocking channel communication.

Locking Strategy and Deadlock Prevention

We use two explicit mutexes throughout the codebase: mutex and logMutex, each serving a distinct purpose. The primary mutex, mutex, protects the internal Raft state, including fields such as currentTerm, state, votedFor, commitIndex, and related variables that must remain consistent across election cycles, role changes, and message handling. On the other hand, logMutex is dedicated to serializing access to the replicated log, specifically, logEntries, lastApplied, and any interaction with the on-disk log file. This separation allows us to isolate responsibilities and prevent unnecessary contention between operations that deal with volatile state versus persistent state.

Critically, we never acquire these two mutexes in an inconsistent or reversed order. Across the entire codebase, when both locks are needed within the same execution path, they are always acquired in the same order: first mutex, then logMutex. We never acquire logMutex in isolation

if there's any chance we may later acquire mutex, and we also avoid holding logMutex across function calls that might acquire mutex. This fixed acquisition order eliminates circular wait conditions, a necessary requirement to prove the absence of deadlocks.

This lock discipline is applied clearly in multiple core functions. In handleAppendEntriesRequests(), we begin by acquiring mutex to validate incoming RPCs, update term and role, and process leader state transitions. If the follower determines that new entries should be committed, it calls commitEntriesUpToIndex(), which acquires logMutex to persist the committed entries to disk. Since mutex is already held at this point, and logMutex is acquired strictly after it, the locking order remains consistent.

Similarly, in the leader path, the function tryAdvanceCommitIndex() is invoked while holding mutex. This function computes the highest log index safely replicated to a majority and updates commitIndex. If it finds that progress has been made, it calls flushCommittedEntriesToFile() to write the newly committed entries to the state machine file. That function acquires logMutex internally, again, following the same mutex then logMutex locking sequence.

We are also extremely careful to never release mutex while still holding logMutex. Both commitEntriesUpToIndex() and flushCommittedEntriesToFile() fully encapsulate their use of logMutex; the lock is acquired at the start of the function and released before returning. This guarantees that any function further up the call stack that holds mutex remains in full control of concurrency while the file write occurs. At no point is logMutex allowed to escape its local context or remain held across calls that might attempt to reacquire mutex.

To avoid lock inversion altogether, we've also made sure that any function that operates on both shared Raft state and log data, including lastApplied, commitIndex, and logEntries, either acquires both locks in the correct order or delegates work to internal functions like commitEntriesUpToIndex() and flushCommittedEntriesToFile(), which manage logMutex themselves.

There are no places in our code where logMutex is acquired first, and then mutex is subsequently locked, a pattern that could lead to circular waits if another goroutine held mutex and tried to acquire logMutex. We explicitly designed the control flow to avoid this scenario. Even functions that only deal with logs (like printLogEntries()) avoid touching mutex, and those that access both (like handleCommandFromClient() or handleAppendEntriesRequests()) do so using the defined acquisition strategy.

This consistent lock ordering, tight containment of critical sections, and function-level responsibility boundaries ensure that our system remains deadlock-free, even under concurrent client load, rapid leadership transitions, and heavy log replication activity.

Safe Channel Communication

Channel communication in our Raft implementation is deliberately structured to be safe, non-blocking, and free from deadlocks or indefinite stalls. Channels are used to signal critical events such as election resets and leader detection, but we always make sure they're used in a way that avoids blocking goroutines or causing resource contention.

We use three primary channels in our system:

- updateElectionTimerChan – to signal that the election timer should be reset
- leaderElectedChan – to inform that a candidate has won the election
- leaderDetectElectChan – to notify that a valid leader has been detected

All three channels are bounded, meaning they have a fixed buffer capacity (typically 1), and are only used for lightweight, single-purpose signaling. By design, they do not carry any large payloads or complex data, their sole role is to trigger state transitions or internal updates.

One of the most critical examples of safe channel usage is with updateElectionTimerChan. Whenever an event occurs that requires the election timer to be reset (e.g. receiving a heartbeat from a valid leader or granting a vote), the following non-blocking select block is used:

```
select {
case updateElectionTimerChan <- 1:
default:
}
```

This pattern ensures that if the channel already has a pending signal (i.e., if the receiver hasn't drained it yet), we don’t block the sender. This is crucial for avoiding goroutines getting stuck — a potential source of deadlock or thread starvation in other designs. We use this exact structure in resetElectionTimer() to guarantee that signals can always be fired without depending on the receiver's readiness.

Similar care is taken with leaderElectedChan and leaderDetectElectChan, both of which are buffered channels with size 1, and are also used with select blocks or guarded with logic like:

```
select {
case leaderElectedChan <- true:
default:
}
```

This allows the system to avoid re-sending signals when the channel is already full. In practice, this makes sense, we don’t need to send multiple redundant signals if one is already pending. Instead, the channel acts as a trigger, once the receiver is ready to process it, it drains the signal and can receive the next one.

For example, in handleAppendEntriesRequests(), when a candidate receives a valid AppendEntries RPC from a leader, it steps down to follower and signals the detection of a valid leader using this exact mechanism:

```
select {
case leaderDetectElectChan<- true:
default:
}
leaderDetectedElectChan <- true
```

This logic is even more robust because it avoids a race where the channel might already contain an unread signal. By draining it before sending, we guarantee there's always room, while still avoiding blocking.

Additionally, no channel in our system is ever used as a form of data pipe between goroutines, they're strictly for coordination. This simplicity in usage keeps our synchronization model predictable and easy to reason about.

Goroutines and Isolation

We use goroutines extensively to handle background tasks in a clean, modular, and concurrency-safe manner. Each long-running task, such as monitorElectionTimer() and sendHeartbeats(), is encapsulated in its own goroutine, scoped to the server's internal state and lifecycle. This architecture ensures that concurrent responsibilities like heartbeat transmission, election monitoring, and client request handling (the requestes in form of log, print, resume, suspend) do not block one another or interfere with the main message processing loop.

These background goroutines are tightly integrated with the Raft state machine. For instance, the sendHeartbeats() function, which maintains leadership by periodically sending AppendEntries RPCs, continuously checks whether the server is still in the Leader role by acquiring mutex. If it detects that the role has changed, the goroutine exits immediately. This prevents outdated behavior or redundant network traffic from a node that is no longer in control. Similarly, monitorElectionTimer() runs in its own loop, monitoring timers for possible leader absence, and only initiates a new election when the node is a Follower and the election timeout has expired.

Importantly, we ensure that these goroutines never hold locks across blocking channel operations or long delays. For example, in monitorElectionTimer(), the main control loop selects over two cases: a timeout on timer.C and a signal on updateElectionTimerChan. This selection is performed without holding any locks, which prevents deadlock scenarios where a goroutine might wait indefinitely for input while blocking others from progressing.

In terms of locking behavior, all goroutines respect the same lock discipline used throughout the codebase. mutex is acquired to protect shared Raft state during transitions, but it is not held longer than necessary. We avoid long-held or I/O-related use of logMutex in background goroutines such as sendHeartbeats() or monitorElectionTimer(). However, in some cases like sendHeartbeats(), we acquire logMutex briefly to safely read in-memory log entries (e.g., via getLogEntriesFrom() or getLogEntryByIndex()). These accesses are read-only, tightly scoped, and free of file I/O, preserving isolation and avoiding lock contention or deadlocks.

Additionally, no goroutine in our system is structured to acquire one lock and then await a signal or message from another part of the system. This helps eliminate deadlock risks stemming from circular waiting or complex interleaving. Each goroutine makes decisions based on the current state, proceeds with clearly scoped locking and exits cleanly when its purpose no longer applies.

Finally, all of our goroutines are self-contained, predictable, and role-aware. They exit naturally when their associated role (e.g. leader or candidate) no longer holds, ensuring that only the necessary behavior is active at any point in time.

BREAKDOWN OF RAFT CLIENT NODES

PROGRESS & DEADLOCK FREEDOM

The client does not deadlock because it is implemented as a sequential, interactive process that operates in a loop, reading user input, validating it, and sending messages to the server. At no point does the client wait indefinitely for a response from the server or another process. It does not rely on synchronous communication, acknowledgements, or blocking reads that would require input from another component in the system. Instead, the client simply takes the user's input, verifies it with a regular expression, and sends it to the specified Raft server using UDP, which is a connectionless protocol that does not wait for acknowledgements.

Moreover the client's design makes sure that even in the case of network failure or a non-responsive server, the client remains responsive to the user. Since sending messages via UDP does not block the client's execution, a failed transmission does not halt the program. If an error occurs during message preparation or transmission, the client simply skips that iteration and returns to the input loop, allowing the user to continue entering commands. The presence of a clear exit condition, triggered by typing the word "exit", also guarantees that the client can always terminate cleanly upon user request, avoiding any infinite waiting state. This simplicity in the client logic and the use of non-blocking communication makes sure that deadlocks are inherently avoided.

REFERENCES

Ongaro, D., & Ousterhout, J. (2014). *In Search of an Understandable Consensus Algorithm (Extended Version)*. Stanford University. Retrieved from https://raft.github.io/raft.pdf