Model Checker for Safety of Raft Leader Election Algorithm

Report submitted in partial fulfillment for
the award of degree of
Bachelor of Technology
in
Computer Science and Engineering

by

Akshit Dudeja
21CS01026

Under the supervision of
Dr. Srinivas Pinisetty

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SCHOOL OF ELECTRICAL AND COMPUTER SCIENCES

INDIAN INSTITUTE OF TECHNOLOGY BHUBANESWAR

Acknowledgement

I would like to express my sincere gratitude to everyone who has contributed to the successful completion of my project. Firstly, I would like to thank my project supervisor, **Dr. Srinivas Pinisetty**, for his invaluable guidance, mentorship, and continuous support throughout the project. His expertise and insights were instrumental in shaping the direction and quality of my work.

I would also like to express my appreciation to all my institution's professors, faculty members, and staff for their support and encouragement. Last but not least, I extend my heartfelt thanks to my family and friends for their unwavering support and encouragement throughout the project. Thank you all for your valuable contributions, guidance, and support.

Contents

Chapter 1

Introduction

Since the advent of the blockchain technology in 2008, it has has been a topic of undivided attention. Being a distributed system, it uses many consensus algorithms that are required to make sure data is consistent. Consensus algorithms refer to mechanisms used in distributed systems, particularly in blockchain and peer-to-peer networks to achieve consensus of a single data value or, more precisely, the history of transactions across nodes. Paxos and Raft are two Consensus algorithms which fall under CFT (Crash Fault Tolerance). CFT consensus algorithms are meant to be resilient to only "crash faults", that is, those situations in which nodes of the network fail and do not process messages. They consider nodes as either functioning correctly or simply failed (crashed) in a benign way without malice. CFT algorithms focus on realizing that, even when some of the nodes have crashed or have become inactive, the system is able to reach the consensus that results among the remaining active nodes. The second family of the consensus algorithm is Byzantine Fault Tolerance (BFT). The consensus algorithms of the BFT deals with the Byzantine faults that are worse than the crash faults, nodes can behave arbitrarily—malfunctioning, acting maliciously, or sending conflicting information to other nodes.

Paxos Algorithm is a consensus algorithm, introduced by Leslie Lamport in the year of 1990. It is actually a family of protocols for solving consensus in a network of unreliable processors. It's one of the easiest and most efficient algorithms to implement in distributed systems. What is Paxos? Paxos is a consensus algorithm. It is a collective agreement on a single value that can be achieved by a collection of machines. Once agreed upon, the value becomes final. Design is fault-tolerant and highly available. Paxos is widely employed in distributed systems such as Google's Chubby lock service, Apache Zookeeper, and the Raft consensus algorithm. The Paxos algorithm is tough to understand and implement the distributed system. Therefore, The Raft consensus algorithm is proposed, which can be considered as efficient like Paxos, and easier to understand and implement.

Raft is the consensus algorithm proposed by Diego Ongaro and John Ousterhout in 2014. This is a consensus algorithm with an intention to make things understandable and easily implementable. Raft is a consensus algorithm which allows a group of computers

to collectively decide on a single value. Once the value is decided, it cannot be changed. The algorithm is designed to be failure-resilient and Raft is used in many distributed systems, such as etcd. In the consensus algorithm of raft, the leaders keep replication of log entries of followers. Leaders send them log entries, and then they respond back. Log entries are committed if it has received acknowledgments from the majority of followers by the leader. The commit message is forwarded to the followers; log entries are applied on the state machines of the followers.

In this project, we will model the Raft consensus algorithm with Promela. Promela is a process meta-language that is used for modelling distributed systems. It is used to model the system and verify the correctness of the system. We will be using the SPIN model checker to verify correctness of the Raft consensus algorithm. As an addition to the solutions that already exist, we will try to implement a new solution which considers n servers instead of the traditional 3. We will use the SPIN model checker to check correctness of the new solution. We will show comparison results of the new solution with existing solutions.

Some of the LTL properties that we will be verifying are:

- **State Machine Safety** : This property states that if a server has applied a log entry to its state machine, then all the servers in the cluster must have applied the same log entry to their state machines.
- **Leader Completeness** : This property states that if a server has committed a log entry, then all the servers in the cluster must have committed the same log entry.
- **Log Matching** : This property states that if two servers have the same log entry at a given index, then the logs of the two servers must be identical up to that index.
- **Leader Append Only** : This property states that if a server has a log entry at a given index, then the leader must have the same log entry at that index.
- **Election Safety** : This property states that at most one leader can be elected in a given term.(Term is basically a monotonically increasing number which here represents the election term, i.e., the number of times an election has been held)

Chapter 2

Background

2.1 Formal Verification

Formal verification is the procedure of using mathematical proof to ensure the correctness of a system or a software program in computer science. It requires forming a formal model of the system and applying mathematical methods to prove that the formal model meets certain properties or specifications. Formal verification is quite common in the application of safety-critical systems where even small errors can have serious repercussions. Formal verification can be done at varying levels of abstraction, from hardware circuits and software programs to whole systems. It is applicable to verify the properties safety (ensuring that nothing bad happens) or liveness (ensuring that something good happens eventually). It is typically carried out manually but automated tools called model checkers can also be utilised. Formal verification is a powerful technique for ensuring the correctness of complex systems. Formal verification gives high assurance that a system works as expected, thus it can help identify bugs and design flaws early in the development process. However, formal verification is time-consuming and requires a certain amount of specialized knowledge and expertise.

2.2 Model Checking

A Model Checker is a verification tool applied in computer science which can automatically check if a system model satisfies a given specification, specified mainly in temporal logic - some examples are Linear Temporal Logic, LTL, or Computation Tree Logic, CTL. The way the Model Checker works is by exploring all possible states of the model in an exhaustive manner to ensure that a system satisfies certain desired properties like safety (nothing bad happens) and liveness (something good eventually happens)

A Model Checker is a verification tool for ensuring the correctness in the presence of simultaneous interactions among the various components of a distributed system across networked environments. Due to concurrency, partial failures, message passing, and syn-

chronization anomalies, distributed systems inherently exhibit complexity that is hard to determine whether any such system behaves correctly in all of its possible scenarios. That is where model checkers come in: they exhaustively simulate and verify all possible states and transitions in the distributed system model to identify deadlocks, race conditions, and violations of consistency and safety properties.

2.3 Distributed Systems

2.3.1 What is a Distributed System?

Distributed system is a network of separate computers that cooperate to accomplish a common objective and provide the impression to the end user that they are one cohesive system. In a distributed system, a number of computers, or nodes, are linked together over a network and exchange messages with one another.

2.3.2 Features of Distributed Systems

- Nodes, which can work individually or together to complete complex tasks, share data and processing.
- The system can withstand the breakdown of one or more nodes without losing data or experiencing service interruptions thanks to fault tolerance and redundancy.

  Because many nodes can operate in parallel, concurrency is achieved, which often increases scalability and performance.

2.3.3 Need of Distributed system

- Scalability: Distributed systems allow applications to scale out horizontally by adding more machines/servers instead of relying on one powerful machine. This is crucial for handling extremely Amounts of data and high volumes of user requests .
- Fault Tolerance and Reliability: In a distributed system, if one node fails, others can take over its tasks or data, ensuring that the system as a whole remains available. This redundancy builds redundancy and minimizes the chances of downtime ensuring that single point of failure does not occur.
- Performance: With the dispersal of jobs across various nodes, distributed systems can process data and requests parallelly, and that can result in significant performance benefits, especially in applications that require high computation or data processing speeds.

2.4 Consesnsus Algorithms

Consensus algorithms are forms of rules or protocols that allow nodes within a blockchain to agree on the required state and thereby reach consensus among its nodes. This agreement is basically on a common state of the network. They are used to guarantee that all nodes in the network reached some form of agreement about which transactions are valid and in what order they get added to the blockchain.

2.4.1 Types of Consensus Algorithms

- Proof of Work (PoW): Miners have to compete in solving complex mathematical puzzles used to validate transactions and establish new blocks. A miner who solves the puzzle first will earn the right to add the block to the blockchain and will also earn some cryptocoins as a reward.
- Proof of Stake (PoS): The validators in the case of PoS are selected for creating new blocks as a function of their coin holding. In this model of PoS, validators get selected at random based on their stake in the network, and they are rewarded through the fees collected on every transaction post-addition.
- Byzantine Fault Tolerance (BFT): BFT algorithms are those which are designed to eventually reach consensus in those distributed systems in which some nodes may be faulty or malicious. They ensure that all honest nodes agree on the state of the network, even when several nodes are behaving incorrectly.
- Practical Byzantine Fault Tolerance (PBFT): PBFT is a kind of consensus protocol, where nodes in a network agree on transaction validity, even when some nodes have become faulty or malignant. It is used in permissioned blockchain networks where all the nodes are known and trusted.
- DAG-based Consensus: Directed Acyclic Graph (DAG) based consensus algorithms use a graph structure to represent transactions and blocks in the blockchain. Nodes in the network vote on the validity of transactions and create a directed acyclic graph to order transactions without the need for mining or proof of work.
- Crash Fault Tolerance (CFT): CFT algorithms are designed to accomplish consensus in distributed systems where nodes can crash but do not act adversarially. They provide assurance that all the honest nodes agree on the The state of the network, even if some nodes crash.

2.4.2 Types of CFT Algorithms

- Paxos: This is a consensus algorithm which allows a network of nodes to agree on a single value even in case of failure or incorrect behavior from some nodes. Paxos

is used with distributed systems ensuring that all the nodes in the system reach a consensus in terms of the state of the network.

- Raft: Raft is an algorithm for consensus, designed to be more intuitive and easier to implement than Paxos. It allows a set of nodes on a network to choose a leader and agree on the state of the network in the form of exchanges messages and keeping a replicated log of transactions.

- Zab: Zab is a consensus algorithm for Apache ZooKeeper that maintains consistency and ordering in distributed systems. A primary node can propose a sequence of transactions which are then logged on to backup nodes to ensure fault tolerance and reliability.

- Viewstamped Replication: Viewstamped Replication is a consensus algorithm that is used in distributed systems to replicate state across multiple nodes. It allows a primary node to propose a sequence of transactions, and which are then replicated to backup nodes in order to ensure fault tolerance and reliability.

2.5 Paxos Algorithm

Paxos is an agreement protocol that allows a network of nodes to collectively agree on a single value, even if some of the nodes are failing or behaving incorrectly. It was designed originally by Leslie Lamport in the year 1990 and subsequently applied to most distributed systems for the purpose of having all nodes come to agreement as far as state on the network goes.
Paxos algorithm has three main roles: namely, proposers, acceptors, and learners.

- Proposers: A value to be agreed on is proposed by the proposers. The proposers send a proposal message to most of the acceptors and wait for the acceptor's response

- Acceptors: The acceptors accept or decline a proposal from any one of the proposers. They send an acknowledgment to the proposer either promising to accept the proposal or rejecting it

- Learners: The learners will learn the accepted value from the network. They wait for a majority of acceptors to agree on a proposal and then learn the value that has been accepted.

The Paxos algorithm consists of two phases: the prepare phase and the accept phase.

- Prepare Phase: In the prepare phase, a proposer sends a prepare request to a majority of acceptors with a proposal number. The acceptors respond with a promise not to accept any proposal with a lower number.

- Accept Phase: In the accept phase, if it has obtained a promise from the majority of acceptors, it broadcasts an accept request with the proposed value to the acceptors.If no acceptor has promised the accept of a higher-numbered proposal, the acceptors accept

Once a majority of acceptors have accepted a proposal, the value is considered agreed upon, and the learners can learn the value. When a proposer failed or behaved in an incorrect way, there is another proposer who can take over and continue the whole process of reaching consensus.

2.6 Raft Algorithm

Raft is a consensus algorithm known to be more intuitive and easier to implement than the Paxos. This will allow the nodes in a network to select a leader and reach agreement concerning the state of the network by the message exchange and replicating log of all transactions.

The Raft algorithm The algorithm involves three basic roles: leaders, followers and candidates.

- Leaders: Leaders are responsible for managing the replication of the log and co- They coordinate the agreement protocol. They receive heartbeat messages from other followers in order to ensure their position.

- Followers: Followers are nodes that service the request from the client and replicate the log from the leader. They service the request of the leader and other followers to ensure consistency .

- Candidates: Candidates are nodes which are trying to become the leader. They send request votes to other nodes to become leader as well as start a new term.

The Raft algorithm essentially runs in two major phases, which are the leader election and log replication.

- Leader Election: Nodes start out as followers and become candidates if they fail to hear from the leader. Candidates send request votes to other nodes, and node with the highest votes becomes the leader for the next term

- Log Replication: After the election of a leader, it starts sending heartbeat messages to followers to maintain its authority. The leader replicates the log to followers that update their state machines based on the appropriate updates so they remain in the consistent state.

The Raft algorithm is resilient to faults and fails under any conditions. It ensures that the network can continue to function even if nodes fail or use their protocol incorrectly. It is commonly used in distributed systems that promote consistency and order across a network.

Chapter 3

Spin Model Checker

3.1 About

The SPIN Model Checker is a widely used tool in the verification of concurrent and distributed systems. SPIN especially finds an especially strong niche in analyzing systems, where the interactions among different processes are highly complex and include things like communication protocols and synchronization mechanisms. SPIN describes system models using a high-level language called Promela (Process Meta Language). Using Promela gives the user opportunity to define processes, message channels, and shared variables in order to encapsulate a behavior of concurrent systems. Specifications or properties to be established, such as safety (mutual exclusion) and liveness-eventual delivery of message are stated in Linear Temporal Logic LTL or directly as assertions at the model level.

Key Features

- Automated State Exploration: SPIN explores all possible states of the model exhaustively or randomly, identifying potential violations of specified properties.
- Scalability Techniques: It employs techniques like partial-order reduction and on-the-fly verification to handle large state spaces more efficiently.
- Error Traces: When a violation is found, SPIN provides an error trace showing the sequence of steps that led to the failure, aiding in debugging and fixing issues.

3.2 Promela

Promela (Process Meta Language) is a high-level language used to describe concurrent systems. It is the input language for the SPIN model checker. Promela is a simple language that allows the user to define processes, message channels, and shared variables. The language is designed to be easy to read and write, and it is expressive enough to model a wide range of systems.

3.3 Entites

3.3.1 Channels

Channels are used to model communication between processes in a concurrent system. A channel is a first-in-first-out (FIFO) queue that allows processes to send and receive messages. Channels can be used to model synchronous or asynchronous communication, and they can be used to model various communication patterns, such as client-server communication, producer-consumer communication, and message passing.

Channel Message Passing Operators

- **? (Receive)**: The receive operator is used to receive a message from a channel. The operator blocks until a message is available on the channel.
- **! (Send)**: The send operator is used to send a message to a channel. The operator blocks until the message is sent to the channel.

Example

An example of a channel in Promela is shown below:

```
chan c = [10] of { byte }
```

This code defines a channel c that can hold up to 10 messages of type byte. The channel can be used by processes to send and receive messages of type byte.

3.4 Message Passing

Message passing is a communication pattern in which processes communicate by sending and receiving messages. Message passing can be used to model various communication patterns, such as client-server communication, producer-consumer

3.4.1 Example

An example of message passing in Promela is shown below:

```
proctype sender(chan c) {
    byte msg = 42;
    c ! msg;
}
proctype receiver(chan c) {
    byte msg;
```

```
        c ? msg;
    }
```

This code defines two processes, sender and receiver, that communicate using a channel c. The sender process sends a message with the value 42 to the channel c, and the receiver process receives the message from the channel c.

3.4.2 Processes

Processes are the basic building blocks of a concurrent system. A process is a sequential program that can execute concurrently with other processes. Processes can communicate with each other using channels, and they can synchronize using synchronization primitives, such as locks and condition variables. Processes can be used to model various entities in a system, such as threads, tasks, clients, servers, and devices.

3.4.3 Shared Variables

Shared variables are used to model shared state in a concurrent system. Shared variables can be read and written by multiple processes, and they can be used to synchronize the behavior of processes. Shared variables can be used to model various entities in a system, such as shared memory, registers, flags, and counters.

3.5 Verification

SPIN uses a variety of techniques to verify properties of concurrent systems. Some of the key techniques used by SPIN include:

3.5.1 State Exploration

SPIN explores all possible states of the model exhaustively or randomly, identifying potential violations of specified properties. SPIN uses a depth-first search algorithm to explore the state space of the model, visiting each state and checking if it satisfies the specified properties.

3.5.2 Partial-Order Reduction

Partial-order reduction is a technique used to reduce the size of the state space by eliminating redundant states. SPIN uses partial-order reduction to explore only a subset of the possible interleavings of concurrent events, reducing the number of states that need to be explored.

3.6 Linear Temporal Logic

Linear Temporal Logic (LTL) is a formal language used to specify properties o f concurrent systems. LTL allows the user to express temporal properties, such as safety (mutual exclusion) and liveness (eventual delivery of message), using a combination of logical operators and temporal operators. LTL properties can be specified directly in the model using assertions or in a separate LTL formula file.

3.6.1 LTL Operators

LTL provides a set of logical operators and temporal operators that can be used to express properties of concurrent systems. Some of the key LTL operators include:

Unary Operators

- [] **(Always)**: The always operator is used to specify that a property holds in all future states of the system.
- <> **(Eventually)**: The eventually operator is used to specify that a property holds in some future state of the system.
- **! (Negation)**: The negation operator is used to specify that a property does not hold in the current state of the system.

Binary Operators

- **U (Strong Until)**: The strong until operator is used to specify that a property holds until another property becomes true.
- **W (Weak Until)**: The weak until operator is used to specify that a property holds until another property becomes true, but the second property does not have to become true.
- **V (Dual of U)**: The dual of U operator is used to specify that a property does not hold until another property becomes true.
- **&& (Logical And)**: The logical and operator is used to specify that two properties hold simultaneously.
- **||| (Logical Or)**: The logical or operator is used to specify that at least one of two properties holds.
- **→ (Logical Implication)**: The logical implication operator is used to specify that if one property holds, then another property holds.

- $\leftrightarrow$ **(Logical Equivalence)**: The logical equivalence operator is used to specify that two properties are equivalent.

Example

An example of an LTL property is the mutual exclusion property, which states that two processes cannot be in their critical sections at the same time. The mutual exclusion property can be expressed in LTL as follows:

```
[]!(p1_critical && p2_critical)
```

This property states that it is always the case that process 1 is not in its critical section and process 2 is not in its critical section at the same time.

Example 2

Another example of an LTL property is the eventual delivery property, which states that a message sent by a process will eventually be received by another process. The eventual delivery property can be expressed in LTL as follows:

```
[]<>(p1_send -> <>p2_receive)
```

This property states that it is always the case that if process 1 sends a message, then process 2 will eventually receive the message.

3.7 Problems explored for understanding SPIN

3.8 Lift Model

In the lift model there are 5 states of the lift. The lift can be in the following states:

- Idle
- DoorOpening
- DoorOpen
- DoorClosing
- MovingUp
- MovingDown
- Stopped

The lift has 3 boolean variables:

- door_closed
- dir_up
- dir_down

The Lift process has the following transitions:

- MovingUp: The lift is moving up to the target floor.
- MovingDown: The lift is moving down to the target floor.
- Idle: The lift is idle at a floor.
- DoorOpening: The lift is opening the door.
- DoorOpen: The door is open.
- DoorClosing: The door is closing.

The Lift process has the following properties:

- p1: The lift cannot move up or down if the door is open.
- p2: The door can only be open when the lift is idle or the door is opening.

Code

```
mtype = {Idle, DoorOpening, DoorOpen, DoorClosing, MovingUp,
MovingDown, Stopped}; mtype
state = Idle;
bool door_closed = true; bool dir_up = false; bool dir_down
= false;

int current_floor = 0; int target_floor = 5;

active proctype Lift()
{
    do :: state == MovingUp ->
        if
        :: current_floor < target_floor ->
            current_floor++
        :: current_floor == target_floor ->
            state = Idle;
            dir_up = false;
            dir_down = true;
        fi
    :: state == MovingDown ->
        if
        :: current_floor > 0 ->
            current_floor--
        :: current_floor == 0 ->
            state = Idle;
            dir_up = true;
            dir_down = false;
        fi
    :: state == Idle if
    :: dir_up = true -> state = MovingUp
    :: dir_down = true -> state = MovingDown
    :: door_closed = true -> state = DoorOpening fi
```

```
        :: state == DoorOpening
            if
            :: door_closed = false -> state = DoorOpen
            fi
        :: state == DoorOpen
            if
            :: door_closed = true -> state = DoorClosing
            fi
        :: state == DoorClosing
            if
            :: door_closed = false -> state = DoorOpen
            :: door_closed = true -> state = Idle
            fi
      od;
    }
    // LTL properties ltl p1
      {
          []!(state == MovingUp && state == DoorOpen)
          []!(state == MovingDown && state == DoorOpen)
      }
```

3.9 Traffic Light Model

The traffic light model has 3 states of the traffic light. The traffic light can be in the following states:

- Green
- Amber
- Red

The traffic light has 2 boolean variables:

- **tick:** A boolean variable that toggles between true and false.
- **status:** A variable that represents the status of the traffic light.

The TrafficLight process has the following transitions:

- Go: The traffic light is green.
- Change: The traffic light is amber.
- Stop: The traffic light is red.

The TrafficLight process has the following properties:

- p1: The traffic light is red when the status is stop.

- p2: The traffic light eventually changes from red to green.
- p3: The traffic light eventually cycles through the sequence red, amber, green.

Code

```
mtype = { GREEN, AMBER, RED }; mtype = { GO, CHANGE, STOP };

bool tick = true; mtype status = GO; mtype light = GREEN;
byte ctr = 0;
active proctype TrafficLight()
{
    do
    :: true ->
        if
        :: tick = false;
        :: tick = true;
        fi;

        if
        :: (status == GO) && (ctr == 3) && tick ->
            status = CHANGE;
            ctr = 0;
        :: (status == CHANGE) && (ctr == 1) && tick ->
            status = STOP;
            ctr = 0;
        :: (status == STOP) && (ctr == 3) && tick ->
            status = CHANGE;
            ctr = 0;
        :: else ->
            ctr = (tick -> (ctr + 1) % 4 :  ctr);
        fi;

        if
        :: status == GO ->
            light = GREEN;
        :: status == CHANGE ->
            light = AMBER;
        :: status == STOP ->
            light = RED;
        fi;
    od;
}

// LTL properties
ltl safety { [](status==STOP ->

light==RED) };
ltl liveness { []((light == RED) -> <>(light == GREEN)) };
ltl liveness2 {([]tick) -> []((light == RED) -> <>
(light == GREEN)) };
```

```
ltl sequence { [](( light == RED) U ((light == AMBER) U
(light == GREEN))) }
```

Chapter 4

Raft Algorithm

4.1 About Raft Algorithm

Raft is a consensus algorithm that was primarily designed to be easier to understand and to work with than Paxos. Applying a communication model whereby nodes converse and hold a copy of transactions, it facilitates the process of electing a leader and reaching consensus over the network state. The Raft system was devised by Diego Ongaro and John Ousterhout in 2013, and Now many use it nowadays. It employs this protocol within a distributed system in order to ensure that all the nodes in that distributed system agree regarding the state of the network.

4.2 States in Raft Algorithm

The Raft algorithm consists of three main roles: leaders, followers, and candidates.

- **Leaders:** They manage the replication of the log and actually coordinate the consensus process. They send heartbeat messages to the followers which maintains their authority and lets them know that indeed the network is in sync.
- **Followers:** Followers are nodes that listen to the leader and replicate the log. Respond to requests from the leader and other followers to maintain consistency in the network.
- **Candidates:** Candidates are nodes which are running for election as the leader. They send request votes to other nodes in the network to become the leader. If a candidate wins votes from a simple majority of the nodes, then that candidate becomes the leader of the distributed system.

4.3 Terms in Raft Algorithm

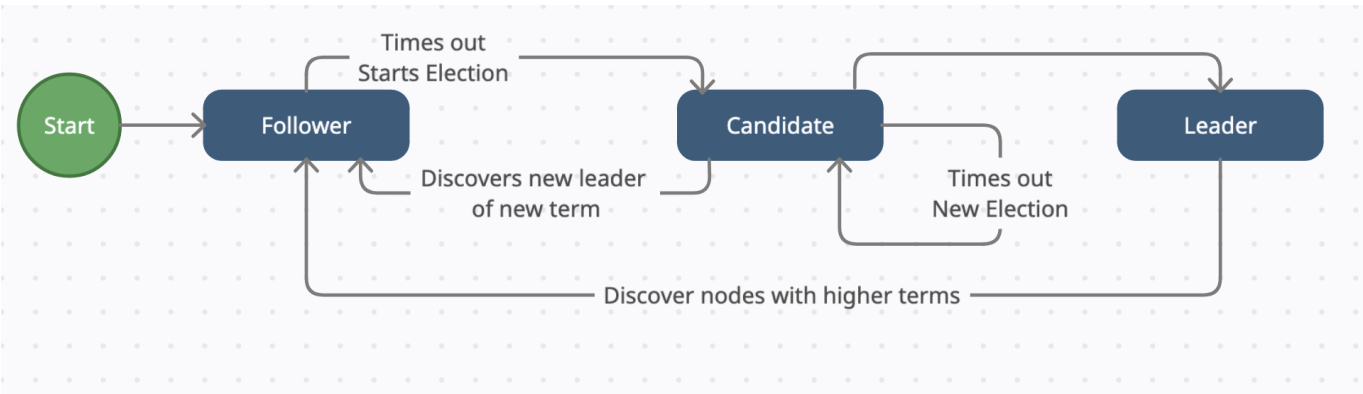
Raft divides time into terms of arbitrary length, numbered with consecutive integers. Each term begins with an election, during which one or more candidates attempt to become a leader. When a candidate receives votes from the majority of nodes, it is considered to be a network leader for that term. If a leader fails or behaves inappropriately, a new term is It thus starts, and a new leader is elected.

4.4 Phases in Raft Algorithm

The Raft algorithm works in two main phases: leader election and log replication.

- Leader Election: In the leader election phase, nodes in the network compete to become the leader. If a node does not hear from the leader for a certain period of time, it becomes a candidate and requests votes from other nodes. If a candidate receives votes from a majority of nodes, it becomes the leader. Followers accept the leader if it has the most up-to-date log. Each term begins with a leader election, and a new leader is elected if the current leader fails or behaves incorrectly.
- Log Replication: Once a leader is elected, it sends heartbeat messages to followers to maintain its authority. The leader receives requests from clients and appends them to its log. It then replicates the log to followers, who apply the changes to their own logs to maintain consistency.

Raft ensures that the network remains available and consistent even if some nodes fail or behave incorrectly.



4.5 Log Replication in Raft Algorithm

After electing a leader, it starts processing client requests. When a client wants to update the log, the leader first updates its version of log entry in local memory. Then sends this

updated entry to every follower. Every follower checks the new entry against its log in order to make sure that this new entry does not conflict with the existing ones in the log. When no conflicts occur, the follower updates its log and reports back to the leader. Once the leader gets acknowledgments from a majority of followers, it considers the entry "committed" and writes it in its state machine. The leader then notifies both the followers and the client that the entry is committed. Upon getting this message, followers also log the entry in their own state machine writing that it is "committed." Once committed, the log entry becomes durable and cannot be rolled back.

4.6 Safety in Raft Algorithm

Raft ensures safety by enforcing rules on both the leader election process and log replication. For leader election,

- A candidate can only be elected as leader if it receives votes from a majority of nodes. This ensures that the leader has the support of the majority of the network and prevents multiple leaders from being elected in the same term.
- Followers only vote for candidates whose logs are at least as up-to-date as their own. This ensures that the leader has the most recent log and prevents outdated entries from being committed.
- A candidate's log is considered more up-to-date if its last entry has a higher term, or if it has a longer log length for the same term. For log replication, the leader commits only those log entries that belong to its current term. However, entries from previous terms can be considered committed if they are followed by entries from the current term. This ensures that incorrect or outdated entries are not permanently applied, maintaining the system's integrity.

4.7 Raft Server Roles and State Transitions

Each server in Raft can be in one of three states: Follower, Candidate, or Leader.

- Follower: Passive role; responds to requests from leaders and candidates.
- Candidate: Tries to become the leader by starting an election
- Leader: Manages the log replication process and handles client requests.

The state transitions among these roles occur based on timeouts, election results, and message exchanges.

4.7.1 Follower

Initial State: When the system starts, each server begins as a follower.
Actions:

- It Follows the leader.
- Responds to requests from the leader.
- If no messages from the leader or candidates are received within a specific election timeout, the follower becomes a candidate.

State Transitions:

- **Follower → Candidate**: If the election timeout expires without hearing from a leader.
- **Follower → Follower**: If it receives an AppendEntries (heartbeat) from a valid leader.
- **Follower → Follower**: If it receives a RequestVote RPC with a higher term than its own, it updates its term and remains a follower.
- **Follower → Follower**: If it receives a RequestVote RPC with a lower term than its own, it ignores the request.

4.7.2 Candidate

Conditions to become Candidate: If a follower does not receive any messages (like AppendEntries) from a leader within an election timeout.
Actions:

- Increments its current term.(Term is basically a counter that increases with each new election.)
- Votes for itself.
- Sends RequestVote RPCs to other servers.
- Waits to receive votes from a majority of servers.
- If it wins the election (majority votes), it becomes the leader.
- If another server claims a higher term, it steps down to follower.
- If the election timeout elapses without winning, it starts a new election.

State Transitions:

- **Candidate → Leader**: If it receives votes from a majority of the servers.
- **Candidate → Follower**: If it receives an AppendEntries from a valid leader (with a higher or equal term).
- **Candidate → Candidate**: If the election timeout elapses without a majority, it restarts the election with an incremented term.

4.7.3 Leader

Conditions to become Leader: When a candidate wins the election.
Actions:

- Sends periodic AppendEntries (heartbeat) messages to all followers to maintain authority.
- Responds to client requests by appending entries to its log and replicating these to followers.
- If an entry is committed (a majority of followers have appended it), it applies the entry to its state machine.

State Transitions:

- **Leader → Follower**: If it discovers a server with a higher term (in AppendEntries or RequestVote RPC).
- **Leader → Leader**: As long as it has authority (sends heartbeats) and there is no term conflict.

4.8 Process

1. **Start:**
   - All servers start as followers, waiting for a leader to be appointed.
2. **Election Trigger:**
   - If a follower does not receive any messages from a leader within a specific election timeout, it becomes a candidate and starts an election.
3. **Election Process:**
   - Each candidate increases its term, votes for itself, and sends RequestVote messages to other servers.

- If a candidate receives votes from a majority (in this case, 2 out of 3 servers), it becomes the leader.
- If another server claims a higher term or another candidate receives a majority of votes, it reverts to follower or retries the election after another timeout.

4. **Leader Operations:**
   - The elected leader starts sending heartbeats (AppendEntries with no data) to maintain its leadership.
   - When a client request is received, the leader appends the request to its log and sends AppendEntries to followers to replicate the log.
   - Once a majority of followers confirm the log entry, it's marked as committed, and the leader can apply it to its state machine.
5. **Handling Failures:**
   - If a leader crashes, followers will stop receiving heartbeats and transition to candidates after an election timeout.
   - They start a new election, and a new leader is chosen.
6. **Handling Network Partitions:**
   - If the network partition divides the servers such that a minority partition cannot form a quorum, servers in that partition will remain followers or candidates.
   - When the network partition heals, servers with outdated terms will revert to followers after receiving messages from the current leader.

Chapter 5

FORMAL SPIN MODELING FOR RAFT ALGORITHM

5.1 Introduction

A mapping is first required between the raft algorithm and the spin model. The raft algorithm, as known to everyone is a consensus algorithm that is used to derive a single value in agreement among a set of processes. The spin model, on the other hand, is a formal modeling language used in modeling concurrent systems. The spin model is based on the principle of processes and message passing over channels by involving them. processes are the entities that communicate with each other by exchanging messages. Processes are modelled as finite state machines. State of a process is collection of variables. The transitions among states are resulted from the messages that are sent and received by the processes. The spin model checker is a tool for correctness verification of the system. The Spin model checker makes use of the formal verification method of model checking that checks the correctness of the system. Model checking technique is founded upon the state space exploration concept. The state space of the system consists of all the possible states through which the system might pass. The model checker checks the state space of the system to see whether the system satisfies a given property. If the system satisfies the property, then the model checker returns a positive result. If the system does not satisfy the property, then the model checker returns a negative result.

The correctness of the raft algorithm will be verified using the spin model checker. There are several entities within the raft algorithm that have to be mapped onto the Model of spin. Node, state, messages, RPC, events, and transitions are entities of the raft algorithm. The entities of the model of spin are the variables, channels, processes, the types of messages, the state transitions, message sending, receiving, and control flow. The entities of the raft algorithm will be mapped on Let us represent entities of the spin model with the aim of creating a formal model of the raft algorithm. Such a formal model will then

be used to verify the correctness of the raft algorithm with the help of the spin model checker.

5.2 Entities of Raft Algorithm

- **Node:** A node is a process that is in the raft cluster. The nodes communicate with each other so as to reach an agreement over a single value. The nodes are modeled as processes/servers in the spin model.
- **State:** The state of a node is represented by the set of variables that present the state of the node at a certain point in time. The state of a node comprises the current term, the current state, the log, commit index, last applied index, next index, and match index.
- **Remote Procedure Call (RPC):** A message sent from one node to another node in order to request a service. The RPC contains the kind of the request, term, candidate id, last log index, last log term, and commit index the entries, the leader id, the prev log index, the prev log term, the success flag, and the vote flag.
- **Event:** An event refers to a change in the state of a node. The events are invoked by the messages that are exchanged between the nodes. These events are modeled as transitions in the spin model.
- **Transition:** According to the algorithm of the raft, a transition means change in a state of a node in the system The events that are occurring in the system are what causes these transitions.

5.3 Entities of Spin Model

- **Process:** A process is an entity that interacts with other. It will be declared by using proctype in the spin model. The process is modeled as a finite state machine. The terms Server and Process will be interchangebly used to represent this entity. Example: Declaring a process in promela

```
1    proctype server(byte serverId) {
2        // Process code
3    }
```

- **Variable:** A variable is a data item that is used to represent the state of a process. The variables are used to store the current state of the process. Each variable has a type and a value. The variables are modeled as state variables in the spin model.

Example: Declaring a variable in promela

```
mtype State = { LEADER, FOLLOWER, CANDIDATE };
mtype State state[N];
byte currentTerm[N] = 0;
```

- **Channel:** A channel is a communication link between two processes. The channel is used to send and receive messages between the processes. A message is sent using the send operator **!** and received using the receive operator **?**.
  Example: Declaring a channel in promela

```
chan ch = [1] of { mtype };
```

- **Message Type:** A message type is a data structure that is used to represent a message. The message type includes the type of the message, the term, the candidate id, the last log index, the last log term, the commit index, the entries, the leader id, the prev log index, the prev log term, the success flag, and the vote flag.
  Example: Type defination for AppendEntry message in promela

```
typedef AppendEntry {
    byte term;
    byte senderId;
    byte index;
    byte prevLogTerm;
    byte leaderCommit;
}
```

- **State Transition:** A change from one state to another is referred to as state transition. variable of a process. The state transition is caused by an event that It occurs in the system. It models the state transition as a transition in the spinning model.

- **Message Sending and Receiving:** A message is sent from one process to another process through a channel. The message is received by the receiving process through the channel.

- **Control Flow:** The control flow of the system is the sequence number of events that happen in the system. The control flow is represented in the model as a sequence of transitions in the spin model.

  by the control_flow channel.

```
    active proctype ControlFlow() {
        mtype state; // State variable
        state = START; // Initial state
        control_flow!state;
        printf("State: START\n");

        // Transition to PROCESS state
        state = PROCESS;
        control_flow!state; // Send the PROCESS
        printf("State: PROCESS\n");

        // Transition to END state
        state = END;
        control_flow!state; // Send the END state
        printf("State: END\n");
    }
```

5.4 Entity Mapping

- **Node to Process:** The node is mapped to a process in the spin model. The process represents the node in the spin model.
  Mapping a raft node to a process in promela:

```
    proctype server(byte serverId) {
        // Process code
    }
```

- **State to Variable:** The state of a node is mapped to a set of variables in the spin model. The variables represent the state of the node in the spin model.
  Mapping the state of a node to variables in promela:

```
    mtype State = { LEADER, FOLLOWER, CANDIDATE };
    mtype State state[N];
    byte currentTerm[N] = 0;
```

- **Message to Message Type:** The message is mapped to a message

- **Event to State Transition:** The event is mapped to a state transition in the spin model. The state of the process changes when an event occurs. Change of variables and state values correspond to the state transition.

- **Remote Procedure Call (RPC) to Message Type:** The RPC is mapped to a message type in the spin model. Type of the message includes type of the request, term, candidate id, last log index, last log term, commit index entries, leader id, prev log index, prev log term, success flag, and the vote flag.

5.5 Initialisation

The basic structure of the raft algorithm involves

- **Participants:** The participants of the raft algorithm are the nodes that are part of the raft cluster. The nodes communicate with each other to reach an agreement on a single value.
- **Participant States:** The states of the participants include the current term, the current state, the log, the commit index, the last applied index, the next index, and the match index.
- **Participant Term:** The term of a participant is the current term of the participant. The term is used to identify the current leader of the cluster.
- **Log:** The log of a participant is the sequence of entries that are stored by the participant. The log includes the term, the index, and the command of each entry.

5.5.1 Header Declaration

```
#define N 3
#define MAX_TERM 3
#define MAX_LOG 2
#define NONE 111
```

N is the maximum number of servers in the cluster. MAX_TERM is the maximum number of terms in the cluster. MAX_LOG is the maximum number of logs in the cluster. NONE is a constant with a large value.

5.5.2 State Declaration

```
    mtype:State = { LEADER,FOLLOWER,CANDIDATE };
    mtype:State state[N];
    byte currentTerm[N] = 0;
```

The state of a participant is the current state of the participant. The state of a participant can be LEADER, FOLLOWER, or CANDIDATE. The current term is basically the term of the participant. The current term is used to identify the current leader of the cluster. The current term is initialized to 0.

5.5.3 Channel Declaration

Append Entry Channel

```
1     typedef AppendEntry {
2         byte term;
3         byte senderId;
4         byte index;
5         byte prevLogTerm;
6         byte leaderCommit;
7     }
8     typedef AppendEntryChannels {
9         chan ch[N] = [1] of { AppendEntry };
10    };
11    AppendEntryChannels appendEntryChannels[N];
```

The append entry channel is used to send append entry messages from one participant to another participant. The append entry message includes the term, the sender id, the index, the prev log term, and the leader commit. The append entry channel is modeled as a channel in the spin model.

Append Entry Response Channel

```
1     typedef AppendEntryResponse {
2         byte term;
3         byte senderId;
4         bool success;
5     };
6     typedef AppendEntryResponseChannels {
7         chan ch[N] = [1] of { AppendEntryResponse };
8     };
9     AppendEntryResponseChannels appendEntryResponseChannels[N];
```

The append entry response channel is used to send append entry response messages from one participant to another participant. The append entry response message includes the term, the sender id, and the success flag. The append entry response channel is modeled as a channel in the spin model.

Request Vote Channel

```
typedef RequestVote {
    byte term;
    byte senderId;
    byte lastLogIndex;
    byte lastLogTerm;
}
typedef RequestVoteChannels {
    chan ch[N] = [1] of { RequestVote };
};
RequestVoteChannels requestVoteChannels[N];
```

The request vote channel is sent by one participant to another participant requesting the vote. The request vote includes the term, the sender id, the last log index, and the last log term. The request vote channel is modeled as a channel within the spin model.

Request Vote Response Channel

```
typedef RequestVoteResponse {
    byte term;
    bool voteGranted;
    byte senderId;
}
typedef RequestVoteResponseChannels {
    chan ch[N] = [1] of { RequestVoteResponse };
};
RequestVoteResponseChannels requestVoteResponseChannels[N];
```

The request vote response channel is utilized by one participant to send request vote responses to another. participant to another participant. The request vote response message carries the term, the vote granted flag, and the sender id. The request vote response channel is represented as a channel in the spin model.

5.5.4 Log Declaration

```
typedef Logs {
    byte log[2];
}
Logs logs[N];
byte commitIndex[N] = 0;
```

A participant's log is the sequence of entries that are accumulated by the participant. The log contains the term and the index of each entry. The logs are kept in an array of logs. The commit index of a participant is the index of the highest log entry that is known to be committed. The commit index is used to commit log entries. The commit index is stored in an array of commit indexes.

5.5.5 Helper Functions

```
byte getLogIndex(byte serverId) {
    return logs[serverId].log[0];
}
byte getLogTerm(byte serverId) {
    return logs[serverId].log[1];
}
byte commitIndex[N] = 0;
```

The get log index function is used to get the index of the log entry of a participant. The get log term function is used to get the term of the log entry of a participant. The commit index of a participant is the index of the highest log entry that is known to be committed.

5.5.6 Init Function

```
init {
    byte i = 0;
    do
    :: (i < N) ->
        run server(i);
        i++
    :: else -> break
    od
}
```

The main function initializes the raft algorithm by running the servers. The main function runs the servers in a loop. The servers will run in parallel.

5.6 States and Transitions

5.6.1 Handle Client Request

```
::
(state[serverId] == LEADER && logs[serverId].log[MAX_LOG -
    1] == 0) ->
    logs[serverId].log[MAX_LOG - 1] = currentTerm[serverId]
od;
```

The handle client request transition is triggered when a client request is received by the leader. The transition is enabled when the state of the participant is LEADER and the last log entry is empty.
logs[serverId].log[MAX_LOG - 1] == 0 means that the last log entry is empty.The last log entry is empty because the server has not received any client requests yet.

The transition sets the last log entry to the current term of the participant. The server will append the current term to the last log entry when it receives a client request.

5.6.2 Timeout Transition

This transaction is triggered when the participant times out. This can only happen when the participant is in the FOLLOWER or CANDIDATE state. The state of the participant will be changed to CANDIDATE. Then the current term of the participant will be incremented by one because the participant is starting a new election. The votedFor local variable will be set to the serverId because the participant is voting for itself (This is a part of the election process). The votesGranted array will be initialized to zero. The votesGranted flag of the participant will be set to one. This is basically the start of the election process .

Original Code

```
::
    (state[serverId] ==
       candidate || state[
       serverId] == follower)
       ->
    atomic {
        state[serverId] =
           candidate;
        currentTerm[serverId] =
            currentTerm[
           serverId] + 1;
        end_max_term:
    :: (currentTerm[serverId]
       <= MAX_TERM) -> skip
    fi
    votedFor = serverId;
    votesGranted[0] = 0;
       votesGranted[1] = 0;
       votesGranted[2] = 0;
    votesGranted[serverId] = 1;
    }
```

Optimized Code

```
::
    (state[serverId] ==
       FOLLOWER || state[
       serverId] == CANDIDATE)
       ->
    atomic {
        state[serverId] =
           CANDIDATE;
        currentTerm[serverId] =
            currentTerm[
           serverId] + 1;
        votedFor = serverId;
        i = 0;
        do
        :: (i < N) ->
            votesGranted[i] =
               0;
            i++
        :: else -> break
        od
        votesGranted[serverId]
           = 1;
    }
```

Table 5.1: Comparison of Original and Optimized Code

Note that only the optimized code is shown in the subsequent sections. The original code is shown with the optimized code wherever some new optimization is done.

5.6.3 Restart Election Transition

```
1 ::
2     state[serverId] = FOLLOWER;
```

This transition is triggered when the participant restarts the election. The state of the participant will be changed to FOLLOWER. This is done to restart the election process. The participant will become a follower and wait for the leader to send a heartbeat message.

5.6.4 Become Leader Transition

```
1     ::
2     (state[serverId] == CANDIDATE) ->
3         voteCount = 0;
4         i = 0;
5         do
6         :: (i < N) ->
7             voteCount = voteCount + votesGranted[i];
8             i = i + 1;
9         :: (i >= N) -> break;
10        od;
11        if
12        :: (voteCount >= N / 2 + 1) ->
13            state[serverId] = LEADER;
14        :: else -> skip
15        fi
```

If the state of the server is CANDIDATE, i.e. the server is competing to become the leader of the system, a check is made to see if the server has received votes from more than half of the servers in the system. If the server has received votes from more than half of the servers, the server becomes the leader of the system. The server changes its state to LEADER otherwise the server remains in the CANDIDATE state.

5.6.5 Request Vote

When the state of a server is CANDIDATE then we will be creating a request vote object with

- term as the current term of the server
- lastLogTerm as the term of the last log entry of the server
- lastLogIndex as the index of the last log entry of the server

For the lastLogTerm and lastLogIndex, we will be checking the logs of the server. If the logs of the server are empty, then the lastLogTerm and lastLogIndex will be set to 0. If the logs of the server are not empty, then the lastLogTerm and lastLogIndex will be set to the term and index of the last log entry of the server.

Now the request vote object will be sent to all the servers in the cluster except the current server. The request vote object will be sent to the other servers using the request vote channel.

Original Code:

```
1     :: // request vote
2     (state[serverId] == candidate) ->
3     atomic {
4         requestVote.term = currentTerm[serverId];
5         if
6         :: (logs[serverId].logs[0] == 0) ->
7             requestVote.lastLogTerm = 0;
8             requestVote.lastLogIndex = 0
9         :: (logs[serverId].logs[0] != 0 && logs[serverId].logs
              [1] == 0) ->
10            requestVote.lastLogTerm = logs[serverId].logs[0];
11            requestVote.lastLogIndex = 1
12        :: (logs[serverId].logs[0] != 0 && logs[serverId].logs
              [1] != 0) ->
13            requestVote.lastLogTerm = logs[serverId].logs[1];
14            requestVote.lastLogIndex = 2
15        fi
16
17        if
18        :: (serverId != 0) ->
19            end_rv_0:           requestVoteChannels[serverId].ch
                 [0]!requestVote
20        :: (serverId != 1) ->
21            end_rv_1:           requestVoteChannels[serverId].ch
                 [1]!requestVote
22        :: (serverId != 2) ->
23            end_rv_2:           requestVoteChannels[serverId].ch
                 [2]!requestVote
24        fi
25    }
```

Issues with the original code:

- The code only works when the number of servers is 3.
- The code takes MAX_LOG as 2. This means that the code will only work when the number of logs is 2.

Optimsed Code:

```
1 :: (state[serverId] == CANDIDATE) ->
2     atomic{
3         requestVote.term = currentTerm[serverId];
4         i = 0;
5         do
6         :: (i < MAX_LOG) ->
7             if
8             :: (logs[serverId].log[i] == 0) ->
9                 if
```

```
10                  :: (i == 0) -> requestVote.lastLogTerm = 0;
11                  :: else -> requestVote.lastLogTerm = logs[
                       serverId].log[i - 1];
12                  fi;
13                  requestVote.lastLogIndex = i;
14                  break;// Exit the loop once the last log entry
                       is determined
15              :: (i == MAX_LOG - 1) ->
16      // Set to the last entry if all previous entries are non-
           zero
17                  requestVote.lastLogTerm = logs[serverId].log[i];
18                  requestVote.lastLogIndex = i + 1;
19                  break;
20              fi;
21              i = i + 1;
22          :: (i >= MAX_LOG) -> break;
23          od;
24          for (i : 0 .. N - 1) {// Loop over all server IDs
25              if
26              :: (serverId != i) -> // Exclude the current
                   serverId
27                  requestVoteChannels[serverId].ch[i]!requestVote
28              :: else -> skip// Do nothing if serverId == i
29              fi
30          }
31      }
```

Issued solved by the optimized code:

- The optimized code works for any number of servers.
- The optimized code works for any number of logs.
- The optimized code is more readable and maintainable.

Issues with the optimized code:
The optimized code does'nt use *end_rv_index* labels (labels are basically static identifiers) to send the request vote message tothe other servers.This is because the optimized code uses a loop to send the request vote message to the other servers. But in Promela, labels are static identifiers attached to specific points in your code. They cannot be dynamically created or parameterized based on variables like loop indices.
This can cause furthur modifications in ltl part of the code which we will be discussing in the later sections.

5.6.6 Handle Request Vote

In this segment we will be handling the request vote message. The components of this segment are:

- **Listening for RequestVote messages:** The server will listen for the request vote messages from the other servers in the cluster. The server will receive the request vote

messages using the request vote channel. The function checks all the channels to see if there is a message in the channel. If no message is recieved in any channel then we are breaking out of this state.

- **Ensure the server is not voting for itself:** If a message is received in some channel, assert is used to check if the serverId is not equal to the serverId of the message. This is done to ensure that the server is not voting for itself. The server will receive the request vote message from the channel
- **Term Comparison and State Update:**
  - The server compares the term in the requestVote message with its current term (currentTerm[serverId]).
  - If the requestVote term is greater than the server's current term, the server updates its currentTerm and changes its state to FOLLOWER. It also resets the votedFor to NONE (indicating it hasn't voted for anyone yet).
  - If the requestVote term is less than or equal to the current term, the server skips the update and does nothing.
- **Log Entry Handling:**Firstly, lastLogTerm and lastLogIndex needs to be determined:
  - For each log entry (from i = 0 to i = MAX_LOG-1), it checks whether the log entry is 0.
  - If a log entry at index i is 0:
    - If i == 0 (the first log entry), it sets lastLogTerm = 0 since there is no prior log.
    - For any other index (i ¿ 0), it sets lastLogTerm to the previous log entry's term (logs[serverId].log[i - 1]).
    - The lastLogIndex is set to i, and the loop breaks.
  - If i == MAX_LOG - 1, it indicates the last log entry. In this case:
    - It sets lastLogTerm = logs[serverId].log[i] (the term of the last log entry).
    - lastLogIndex is set to i + 1 to indicate the index after the last log entry.
    - The loop breaks.
- **Log Consistency Check (logOk):**After determining the lastLogTerm and lastLogIndex, the code checks whether the log in the requestVote message is consistent:
  - It compares requestVote.lastLogTerm with lastLogTerm to check if the candidate's log is more recent.
  - If the lastLogTerm is equal, it checks whether requestVote.lastLogIndex is greater than or equal to the lastLogIndex to ensure the logs are consistent.
- **. Vote Granting Decision:** The server grants the vote only if:

  - The requestVote.term matches the server's current term (currentTerm[serverId]).
  - The log consistency check (logOk) passes.
  - The server has not already voted for another candidate (votedFor == NONE) or it has voted for the current candidate (votedFor == i).

- **Response and Vote Assignment:**
  - The server creates a requestVoteResponse that includes:
    - The vote decision (voteGranted).
    - The current term (term).
  - If the vote is granted (requestVoteResponse.voteGranted), the server records its vote by setting votedFor = i. Otherwise, it skips this step.
  - The vote response is sent to the corresponding requestVoteResponseChannels[i].ch[serverId].

Original Code:

```
::
(state[serverId] == candidate) ->
atomic {
    requestVote.term = currentTerm[serverId];
    if
    :: (logs[serverId].logs[0] == 0) ->
        requestVote.lastLogTerm = 0;
        requestVote.lastLogIndex = 0
    :: (logs[serverId].logs[0] != 0 && logs[serverId].logs[1] ==
        0) ->
        requestVote.lastLogTerm = logs[serverId].logs[0];
        requestVote.lastLogIndex = 1
    :: (logs[serverId].logs[0] != 0 && logs[serverId].logs[1] !=
        0) ->
        requestVote.lastLogTerm = logs[serverId].logs[1];
        requestVote.lastLogIndex = 2
    fi

    if
    :: (serverId != 0) ->
        end_rv_0:           requestVoteChannels[serverId].ch[0]!
            requestVote
    :: (serverId != 1) ->
        end_rv_1:           requestVoteChannels[serverId].ch[1]!
            requestVote
    :: (serverId != 2) ->
        end_rv_2:           requestVoteChannels[serverId].ch[2]!
            requestVote
    fi
}
```

Optimized Code:

```
::
    atomic {
        i = 0;
        do
        :: (i < N) ->
            if
            :: (len(requestVoteChannels[i].ch[serverId]) > 0) ->
                assert(i != serverId);
                requestVoteChannels[i].ch[serverId]?requestVote;

                // Valid channel with valid Vote found
                if
                :: (requestVote.term > currentTerm[serverId]) ->
                    currentTerm[serverId] = requestVote.term;
                    state[serverId] = FOLLOWER;
                    votedFor = NONE;
                :: (requestVote.term <= currentTerm[serverId])
                   ->
                    skip
                fi

                i = 0;
                do
                :: (i < MAX_LOG) ->
                    if
                    :: (logs[serverId].log[i] == 0) ->
                        if
                        :: (i == 0) -> lastLogTerm = 0;
                        :: else -> lastLogTerm = logs[serverId].
                           log[i - 1];
                        fi;
                        lastLogIndex = i;
                        break;
                    :: (i == MAX_LOG - 1) ->
                        lastLogTerm = logs[serverId].log[i];
                        lastLogIndex = i + 1;
                        break;
                    fi;
                    i = i + 1;
                :: (i >= MAX_LOG) -> break;
                od;

                logOk = requestVote.lastLogTerm > lastLogTerm ||
                    requestVote.lastLogTerm == lastLogTerm &&
                   requestVote.lastLogIndex >= lastLogIndex;
                requestVoteResponse.voteGranted = (requestVote.
                   term == currentTerm[serverId]) && logOk && (
                   votedFor == NONE || votedFor == i);

                requestVoteResponse.term = currentTerm[serverId
                   ];
```

```
45                  if
46                  :: requestVoteResponse.voteGranted -> votedFor =
                        i
47                  :: !requestVoteResponse.voteGranted -> skip
48                  fi
49                  end_requestVoteResponse:
                        requestVoteResponseChannels[i].ch[serverId]!
                        requestVoteResponse
50                  break;
51              :: else ->
52                  i = i + 1
53              fi
54          :: else ->
55              break;
56          od;
57          break
58      }
```

5.6.7 Handle RequestVoteResponse

Just like the RequstVote function we will be determining if there is some messge in RequestVoteResponseChannel corresponding to some server If there isn't any response in any channel then we are breaking out of this state.
If there is a response in some channel then:

- **Processing the Response:** If a valid RequestVoteResponse is received, the server reads the message (requestVoteResponseChannels[i].ch[serverId]?requestVoteResponse), which contains:
  - The term of the sender (requestVoteResponse.term).
  - Whether the vote was granted (requestVoteResponse.voteGranted).
- **Updating Term and State:**
  - If the term in the response is greater than the server's current term (currentTerm[serverId]), the server updates its current term and changes its state to FOLLOWER. It also resets the votedFor to NONE (indicating it hasn't voted for anyone yet).
  - If the term in the response is less than or equal to the current term, the server skips the update and does nothing.
- **Counting the Vote:**If the requestVoteResponse.term matches the server's current term and the vote was granted (requestVoteResponse.voteGranted), it sets votesGranted[i] = 1 to indicate that this server has voted in favor of the current candidate.

Original Code:

```
::
    atomic {
        i = 0;
        do
        :: (i < N) ->
            if
            :: (len(requestVoteResponseChannels[i].ch[serverId])
                > 0) ->
                requestVoteResponseChannels[i].ch[serverId]?
                    requestVoteResponse;
                if
                :: (requestVoteResponse.term > currentTerm[
                    serverId]) ->
                    currentTerm[serverId] = requestVoteResponse.
                        term;
                    state[serverId] = FOLLOWER;
                    votedFor = NONE;
                :: (requestVoteResponse.term == currentTerm[
                    serverId] && requestVoteResponse.voteGranted)
                    ->
                    votesGranted[i] = 1
                :: !(requestVoteResponse.term > currentTerm[
                    serverId]) && !(requestVoteResponse.term ==
                    currentTerm[serverId] && requestVoteResponse.
                    voteGranted) ->
                    skip
                fi
                break;
            :: else ->
                i = i + 1
            fi
        :: else ->
            break;
        od;
        break
    }
```

Optimized Code:

```
:: // handle RequestVoteResponse
atomic {
    i = 0;
    do
    :: (i < N) ->
        if
        :: (len(requestVoteResponseChannels[i].ch[serverId]) >
           0) ->
            assert(i != serverId);
            requestVoteResponseChannels[i].ch[serverId]?
               requestVoteResponse;

// Valid channel with valid Vote found
            if
            :: (requestVoteResponse.term > currentTerm[serverId
               ]) -> // update terms
                currentTerm[serverId] = requestVoteResponse.term
                    ;
                state[serverId] = FOLLOWER;
                votedFor = NONE
            :: (requestVoteResponse.term == currentTerm[serverId
               ] && requestVoteResponse.voteGranted) ->
                votesGranted[i] = 1
            :: !(requestVoteResponse.term > currentTerm[serverId
               ]) && !(requestVoteResponse.term == currentTerm[
               serverId] && requestVoteResponse.voteGranted) ->
                skip
            fi
        :: else ->
            i = i + 1
        fi
    :: else ->
        break;
    od;
    break
}
```

5.6.8 Handle Append Entries

When the leader has been choosen, then it's the duty of the leader to send the updates through the append entries. The components of this segment are:

- **Setting the Append Entry:** The leader sets the appendEntry object with the following values:
  - The term of the leader (currentTerm[serverId]).
  - The leader's commit index (commitIndex[serverId]).
  - The index of the log entry to be appended (appendEntry.index).
  - The term of the previous log entry (appendEntry.prevLogTerm).

  - The leader's server ID (appendEntry.senderId).

- **Determining the Index:** The leader determines the index of the log entry to be appended by comparing its logs with the logs of the other servers. The leader iterates over the logs of the other servers to find the index of the log entry that is different from its own logs.

- **Sending the Append Entry:** After identifying the first mismatch (if any), the leader sends the appendEntry message to server i using **appendEntryChannels[serverId].ch[i]!appendEntry**.

Original Code:

```
:: (state[serverId] == leader) ->
atomic {
    if
    :: (serverId != 0) -> i = 0
    :: (serverId != 1) -> i = 1
    :: (serverId != 2) -> i = 2
    fi

    appendEntry.term = currentTerm[serverId];
    appendEntry.leaderCommit = commitIndex[serverId];
    if
    :: (logs[serverId].logs[0] != logs[i].logs[0]) ->
        appendEntry.index = 0
    :: (logs[serverId].logs[1] != 0 && logs[serverId].logs[0] ==
        logs[i].logs[0] && logs[serverId].logs[1] != logs[i].
       logs[1]) ->
        appendEntry.index = 1
        appendEntry.prevLogTerm = logs[i].logs[0]
    :: appendEntry.index = NIL
    fi
    end_ae:          appendEntryChannels[serverId].ch[i]!
       appendEntry
}
```

Optimised Code:

```
        :: // Append Entries
        (state[serverId] == LEADER) ->
        atomic{
            appendEntry.term = currentTerm[serverId];
            appendEntry.senderId = serverId;
            appendEntry.index = 0;

            do
            :: (i < N) ->
                if
                :: (i != serverId) ->
                    atomic{
                        int j = 0;
```

```
14                              do
15                              :: (j < MAX_LOG) ->
16                                  if
17                                  :: (logs[serverId].log[j] != logs[i
                                        ].log[j]) ->
18                                      appendEntry.index = j;
19                                      appendEntry.prevLogTerm = logs[i
                                            ].log[j - 1];
20                                      break;
21                                  fi;
22                                  j = j + 1;
23                              :: (j >= MAX_LOG) -> break;
24                              od;
25
26                          }
27                          end_appendEntry:        appendEntryChannels[
                                serverId].ch[i]!appendEntry
28                      fi;
29                      i++;
30                  :: else -> break
31                  od
32          }
```

5.6.9 Handle AppendEntry

Original Code:

```
1  || appendEntryChannels[1].ch[serverId]?[appendEntry] ||
       appendEntryChannels[2].ch[serverId]?[appendEntry])
2  -> atomic { // calculate the id of the sender if ::
       appendEntryChannels[0].ch[serverId]?appendEntry
3  -> i = 0 :: appendEntryChannels[1].ch[serverId]?appendEntry -> i
       = 1 :: appendEntryChannels[2].ch[serverId]?appendEntry
4  -> i = 2 fi assert(i != serverId);
5
6  // update terms if :: (appendEntry.term > currentTerm[serverId])
       -> currentTerm[serverId]
7  = appendEntry.term; state[serverId] = follower; votedFor = NIL
       :: (appendEntry.term
8  <= currentTerm[serverId]) -> skip fi assert(appendEntry.term <=
       currentTerm[serverId]);
9
10 // return to follower state
11
12 if :: (appendEntry.term == currentTerm[serverId] && state[
       serverId] == candidate)
13 -> state[serverId] = follower; votedFor = NIL :: (appendEntry.
       term != currentTerm[serverId]
14 || state[serverId] != candidate) -> skip fi assert(!(appendEntry
       .term == currentTerm[serverId])
15 || (state[serverId] == follower));
16
```

```
logOk = appendEntry.index == 0 || (appendEntry.index == 1 &&
    appendEntry.prevLogTerm
== logs[serverId].logs[0]); appendEntryResponse.term =
    currentTerm[serverId]; if
:: (appendEntry.term < currentTerm[i] || appendEntry.term ==
    currentTerm[serverId]
&& state[serverId] == follower && !logOk) -> // reject request
appendEntryResponse.success = 0; end_aer_rej:
    appendEntryResponseChannels[serverId].ch[i]!
    appendEntryResponse
:: (appendEntry.term == currentTerm[serverId] && state[serverId]
    == follower &&
logOk) -> appendEntryResponse.success = 1;

logs[serverId].logs[appendEntry.index] = appendEntry.term;
commitIndex[serverId] = appendEntry.leaderCommit;

end_aer_acc: appendEntryResponseChannels[serverId].ch[i]!
    appendEntryResponse fi }
```

Optimised Code:

```
        :: // handle AppendEntry
        atomic{
            i = 0;
            do
            :: (i < N) ->
                if
                :: (len(appendEntryChannels[i].ch[serverId]) >
                    0) ->
                    assert(i != serverId);
                    appendEntryChannels[i].ch[serverId]?
                        appendEntry;
// update terms
                    if
                    :: (appendEntry.term > currentTerm[serverId
                        ]) ->
                        currentTerm[serverId] = appendEntry.term
                            ;
                        state[serverId] = FOLLOWER;
                        votedFor = NONE;
                    :: (appendEntry.term <= currentTerm[serverId
                        ]) ->
                        skip
                    fi
                    assert(appendEntry.term <= currentTerm[
                        serverId]);
// Return to FOLLOWER state
                    if
                    :: (appendEntry.term == currentTerm[serverId
                        ] && state[serverId] == CANDIDATE) ->
                        state[serverId] = FOLLOWER;
                        votedFor = NONE;
```

```
25                      :: (appendEntry.term != currentTerm[serverId
                            ] || state[serverId] != CANDIDATE) ->
26                          skip
27                      fi
28                      assert(!(appendEntry.term == currentTerm[
                            serverId]) || (state[serverId] ==
                            FOLLOWER));
29                      logOk = appendEntry.index == 0 || (
                            appendEntry.index == 1 && appendEntry.
                            prevLogTerm == logs[serverId].log[0]);
30                      if
31                      :: (appendEntry.term < currentTerm[i] ||
                            appendEntry.term == currentTerm[serverId]
                             && state[serverId] == FOLLOWER && !logOk
                            ) -> // reject request
32                          appendEntryResponse.success = 0;
33                          end_aer_rej:
                                appendEntryResponseChannels[serverId
                                ].ch[i]!appendEntryResponse
34                      :: (appendEntry.term == currentTerm[serverId
                            ] && state[serverId] == FOLLOWER && logOk
                            ) ->
35                          appendEntryResponse.success = 1;
36                          logs[serverId].log[appendEntry.index] =
                                appendEntry.term;
37                          commitIndex[serverId] = appendEntry.
                                leaderCommit;
38                          end_aer_acc:
                                appendEntryResponseChannels[serverId
                                ].ch[i]!appendEntryResponse
39                      fi
40                  :: else ->
41                      i = i + 1
42                  fi
43              :: else ->
44                  break;
45              od;
46              break;
47
48          }
```

5.6.10 Handle AppendEntryResponse

Original Code:

```
1
2 :: // handle AppendEntryResponse
3 (appendEntryResponseChannels[0].ch[serverId]?[
      appendEntryResponse] || appendEntryResponseChannels[1].ch[
      serverId]?[appendEntryResponse] ||
      appendEntryResponseChannels[2].ch[serverId]?[
      appendEntryResponse]) ->
```

```
4  atomic {
5  // calculate the id of the sender
6      if
7      :: appendEntryResponseChannels[0].ch[serverId]?
          appendEntryResponse -> i = 0
8      :: appendEntryResponseChannels[1].ch[serverId]?
          appendEntryResponse -> i = 1
9      :: appendEntryResponseChannels[2].ch[serverId]?
          appendEntryResponse -> i = 2
10     fi
11     assert(i != serverId);
12
13     if
14     :: (appendEntryResponse.term > currentTerm[serverId]) -> //
          update terms
15         currentTerm[serverId] = appendEntryResponse.term;
16         state[serverId] = follower;
17         votedFor = NIL
18     :: (appendEntryResponse.term < currentTerm[serverId]) ->
19         skip
20     :: (appendEntryResponse.term == currentTerm[serverId] &&
          appendEntryResponse.success && state[serverId] == leader)
          ->
21 // advance commit index
22 // as we only have 3 servers
23 // one success AppendEntry means committed
24
25         end_commitIndex:    if// end if commitIndex reaches the
              limit
26     :: (commitIndex[serverId] == 0 && logs[i].logs[0] == logs[
          serverId].logs[0]) ->
27         commitIndex[serverId] = 1
28     :: (commitIndex[serverId] == 1 && !(logs[serverId].logs[1]
          != 0 && logs[i].logs[1] == logs[serverId].logs[1])) ->
29         skip;// actually this case won't be reached
30     fi
31 :: (appendEntryResponse.term == currentTerm[serverId] && !(
     appendEntryResponse.success && state[serverId] == leader)) ->
32     skip
33 fi
34 }
```

Optimised Code:

```
1  :: // Handle AppendEntryResponse
2  atomic{
3      i = 0;
4      do
5      :: (i < N) ->
6          if
7          :: (len(appendEntryResponseChannels[i].ch[serverId]) >
              0) ->
8              assert(i != serverId);
```

```
9                appendEntryResponseChannels[i].ch[serverId]?
                     appendEntryResponse;
10               if
11               :: (appendEntryResponse.term > currentTerm[serverId
                     ]) -> // update terms
12                   currentTerm[serverId] = appendEntryResponse.term
                         ;
13                   state[serverId] = FOLLOWER;
14                   votedFor = NONE
15               :: (appendEntryResponse.term < currentTerm[serverId
                     ]) ->
16                   skip
17               :: (appendEntryResponse.term == currentTerm[serverId
                     ] && appendEntryResponse.success && state[
                     serverId] == LEADER) ->
18                   if
19                   :: (commitIndex[serverId] == 0 && logs[i].log[0]
                         == logs[serverId].log[0]) ->
20                       commitIndex[serverId] = 1
21                   :: (commitIndex[serverId] == 1 && !(logs[
                         serverId].log[1] != 0 && logs[i].log[1] ==
                         logs[serverId].log[1])) ->
22                       skip
23                   fi
24               :: (appendEntryResponse.term == currentTerm[serverId
                     ] && !(appendEntryResponse.success && state[
                     serverId] == LEADER)) ->
25                   skip
26               fi
27           :: else ->
28               i = i + 1
29           fi
30       :: else ->
31           break
32       od;
33       break
34
35   }
```

5.7 Safety Checks

In Raft consensus algorithm, the safety properties are checked to ensure that the system behaves correctly. The safety properties are checked using Linear Temporal Logic (LTL) expressions. The safety properties that are checked in the Raft consensus algorithm are:

- **Election Safety:** There can be at most one leader in a given term.
- **Election Liveliness:** A leader will eventually be elected in a given term.
- **Log Matching:** If two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.

- **Leader Completeness:** If a log entry is committed in a given term, then all the servers in that term must have committed that entry.
- **State Machine Safety:** If a server has applied a log entry at a given index to its state machine, then no other server will apply a different log entry for the same index.
- **Leader Append Only:** A leader never overwrites or deletes entries in its log; it only appends new entries.

5.7.1 Election Safety

The Election Safety property states that there can be at most one leader in a given term. This property is checked using the following LTL expression:

Original Code:

```
ltl electionSafety { [] (state[0] != leader || state[1] !=
    leader || state[2] != leader) }
```

Optimised Code:

```
ltl electionSafety{ always (leaders <= 1) }
```

I have removed the hard coded values for checking the leader state of each server and used a variable leaders to keep track of the number of leaders in the system.

This variable is updated whenever a server becomes a leader or loses its leader

5.7.2 Election Liveliness

The Election Liveliness property states that a leader will eventually be elected in a given term. This property is checked using the following LTL expression:

Original Code:

```
ltl electionLiveness{ <> (state[0] == LEADER || state[1] ==
    LEADER || state[2] == LEADER) }
```

Optimised Code:

```
ltl electionLiveness{ eventually (leaders > 0) }
```

As described in the previous section, I have used the leaders variable to keep track of the number of leaders in the system. The electionLiveness property is checked using the leaders variable. Now I will be checking if the value of leaders is eventually greater than 0.

5.7.3 Log Matching

The Log Matching property states that if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.

Original Code:

```
ltl logMatching {
    always (
        ((logs[0].logs[1] != 0 && logs[0].logs[1] == logs[1].
            logs[1])
        implies (logs[0].logs[0] == logs[1].logs[0]))
        && ((logs[0].logs[1] != 0 && logs[0].logs[1] == logs[2].
            logs[1])
        implies (logs[0].logs[0] == logs[2].logs[0]))
        && ((logs[1].logs[1] != 0 && logs[1].logs[1] == logs[2].
            logs[1])
        implies (logs[1].logs[0] == logs[2].logs[0]))
    )
}
```

Optimised Code: To be optimized later

5.7.4 Leader Completeness

The Leader Completeness property states that if a log entry is committed in a given term, then all the servers in that term must have committed that entry.

Original Code:

```
ltl leaderCompleteness {
    always (
        (
            (commitIndex[0] == 1) implies
                always (
                ((state[1] == leader) implies (logs[0].logs[0]
                    == logs[1].logs[0]))
                && ((state[2] == leader) implies (logs[0].logs
                    [0] == logs[2].logs[0]))
                )
                ) && (
                (commitIndex[1] == 1) implies
                always (
                ((state[0] == leader) implies (logs[1].logs[0]
                    == logs[0].logs[0]))
                && ((state[2] == leader) implies (logs[1].logs
                    [0] == logs[2].logs[0]))
                )
                ) && (
                (commitIndex[2] == 1) implies
                always (
                ((state[0] == leader) implies (logs[2].logs[0]
                    == logs[0].logs[0]))
```

```
19                  && ((state[1] == leader) implies (logs[2].logs
                        [0] == logs[1].logs[0]))
20              )
21          )
22      )
23  }
```

Optimised Code: To be optimized later

5.7.5 State Machine Safety

The State Machine Safety property states that if a server has applied a log entry at a given index to its state machine, then no other server will apply a different log entry for the same index.

Original Code:

```
1  ltl stateMachineSafety {
2      always (
3          ((commitIndex[0] == 1 && commitIndex[1] == 1) implies (
               logs[0].logs[0] == logs[1].logs[0]))
4          && ((commitIndex[0] == 1 && commitIndex[2] == 1) implies
                (logs[0].logs[0] == logs[2].logs[0]))
5          && ((commitIndex[1] == 1 && commitIndex[2] == 1) implies
                (logs[1].logs[0] == logs[2].logs[0]))
6      )
7  }
```

Chapter 6

Results and Future Work

6.1 Results

Currently the system is at a stage where model can run for more than 3 leaders and 3 log entries. The system is able to handle the leader election with most of the scenarios covered in case of failures using timeouts. The system is able to handle the leader election with most of the scenarios covered in case of failures using timeouts.

The spin simulation is able to handle all the properties stated and is able to search in the state space for the properties.

```
ltl electionSafety: [] (! ((((((state[0]==3)) && ((state[1]==3))) && ((currentTerm[0]==currentTerm[1]))) || ((((state[0]==3)) && ((state[2]==3))) && ((currentTer
m[0]==currentTerm[2])))) || ((((state[1]==3)) && ((state[2]==3))) && ((currentTerm[1]==currentTerm[2])))))
Depth=     466 States=    1e+06 Transitions= 2.69e+06 Memory=   572.577 t=     2.37 R=   4e+05
Depth=     466 States=    2e+06 Transitions= 5.85e+06 Memory=  1016.425 t=     5.33 R=   4e+05
Depth=     466 States=    3e+06 Transitions= 9.04e+06 Memory=  1460.370 t=     8.15 R=   4e+05
Depth=     476 States=    4e+06 Transitions= 1.17e+07 Memory=  1868.769 t=     10.5 R=   4e+05
Depth=     507 States=    5e+06 Transitions= 1.47e+07 Memory=  2287.030 t=     13.3 R=   4e+05
Depth=     520 States=    6e+06 Transitions= 1.78e+07 Memory=  2713.593 t=     16.6 R=   4e+05
Depth=     520 States=    7e+06 Transitions= 2.08e+07 Memory=  3138.593 t=     20.2 R=   3e+05
Depth=     520 States=    8e+06 Transitions= 2.38e+07 Memory=  3562.421 t=     24.5 R=   3e+05
Depth=     520 States=    9e+06 Transitions= 2.67e+07 Memory=  4003.339 t=     29.1 R=   3e+05
Depth=     520 States=    1e+07 Transitions= 2.97e+07 Memory=  4447.284 t=     34.7 R=   3e+05
Depth=     520 States=  1.1e+07 Transitions= 3.28e+07 Memory=  4891.132 t=     41.7 R=   3e+05
Depth=     520 States=  1.2e+07 Transitions= 3.58e+07 Memory=  5322.089 t=     60.8 R=   2e+05
Depth=     520 States=  1.3e+07 Transitions= 3.84e+07 Memory=  5725.116 t=     82.3 R=   2e+05
Depth=     529 States=  1.4e+07 Transitions= 4.14e+07 Memory=  6151.288 t=      115 R=   1e+05
^CInterrupted

(Spin Version 6.5.2 -- 6 December 2019)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             + (electionSafety)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 488 byte, depth reached 529, errors: 0
 14894453 states, stored
 29326130 states, matched
 44220583 transitions (= stored+matched)
2.1753285e+08 atomic steps
hash conflicts:   8284148 (resolved)

Stats on memory usage (in Megabytes):
 7329.500       equivalent memory usage for states (stored*(State-vector + overhead))
 6421.203       actual memory usage for states (compression: 87.61%)
                state-vector as stored = 424 byte + 28 byte overhead
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
   17.004       memory lost to fragmentation
 6532.733       total actual memory usage



pan: elapsed time 209 seconds
pan: rate 71190.388 states/second
```

In the results, it can be seen that the system is able to handle the ltl properties without any error and the depth reached is 529. The system is able to handle the

- States = 1.4e+07

- Transitions = 4.14e+07
- Memory = 6151.288
- Errors = 0
- Rate = 71190.388 states/second
- Matched Transitions = 44220583

6.2 Future Work

The future work includes the following:

- Handling the following ltl properties for n servers and m log entries:
  - State Machine Safety
  - Leader Completeness
  - Log Matching
- Optimising the existing code to use more global shared variables instead of local variables.
- Replacing the existing channels with bi-directional channels.
- Figuring out a way to use single channel for voting and single for entries b/w a set of servers.

Bibliography

[1] Qihao Bao, Bixin Li, Tianyuan Hu, and Dongyu Cao, "Model Checking the Safety of Raft Leader Election Algorithm"
https://ieeexplore.ieee.org/abstract/document/10062357

[2] Raft and Paxos Consensus Algorithms for Distributed Systems
https://medium.com/@mani.saksham12/raft-and-paxos-consensus-algorithms-for-distributed-

[3] Raft Consensus(Github)
https://github.com/debajyotidasgupta/raft-consensus

[4] Raft Spin(Github)
https://github.com/namasikanam/raft-spin

[5] Spin Manual
https://spinroot.com/spin/Man/Manual.html