BTECH PROJECT- 1

by

AKSHIT DUDEJA
21CS01026

Supervisor Name: Dr. Srinivas Pinisetty

Department of Computer Science and Engineering
School of Electrical and Computer Sciences

Model Checker for Safety of Raft Leader Election Algorithm

August
Spin Model Checker

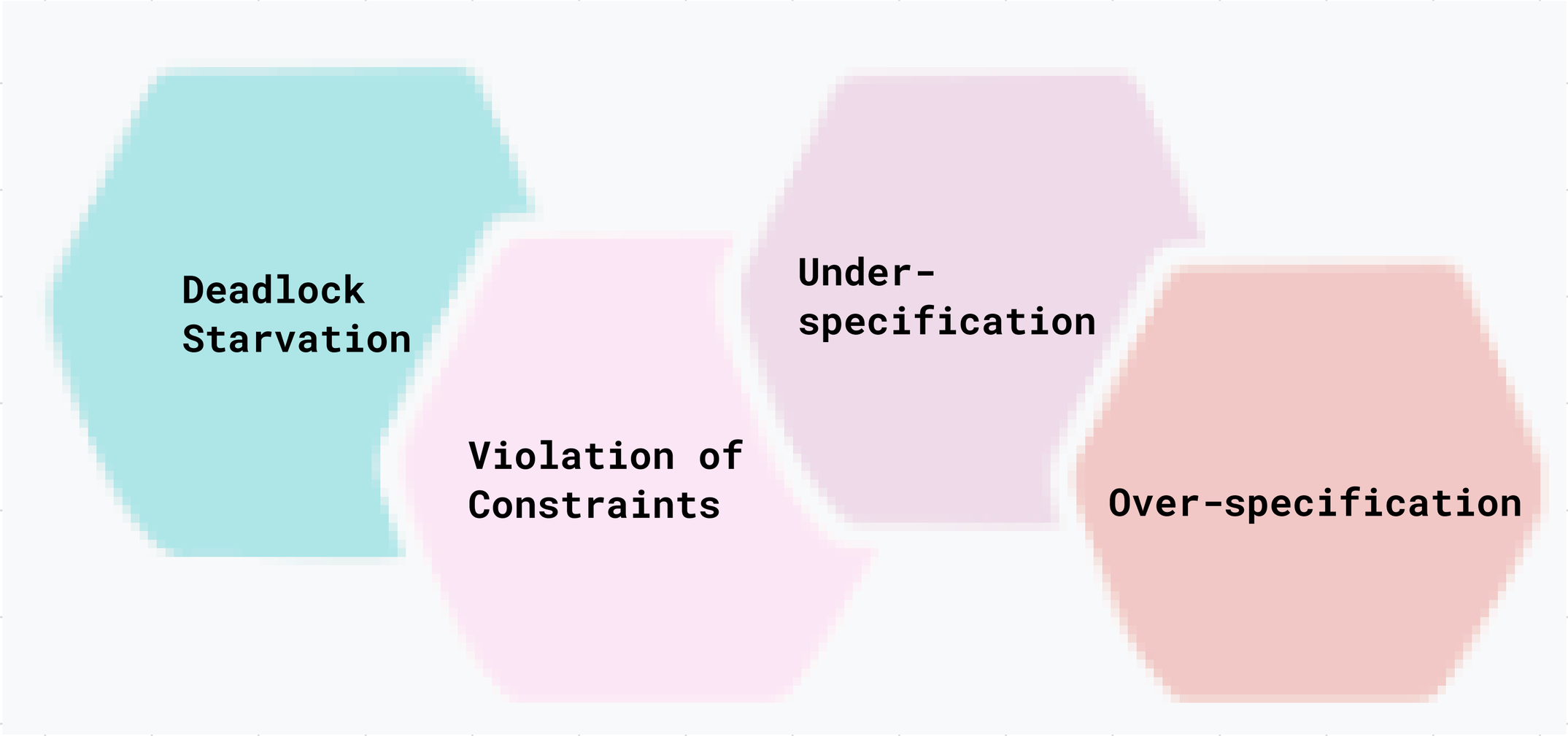
September
Raft Algorithm

October
Examining the existing raft-spin implementation

November
Optimizing the raft-spin model

TABLE OF CONTENT

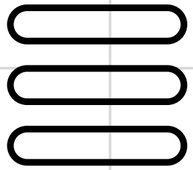
Common Design Flaws



Formal Verification

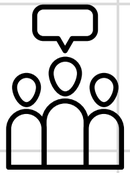
Formal verification is the procedure of using mathematical proof to ensure the correctness of a system or a software program in computer science.

Formal verification can be done at varying levels of abstraction, from hardware circuits and software programs to whole systems.



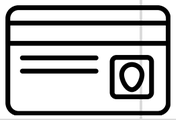
It is applicable to verify the properties safety (ensuring that nothing bad happens) or liveness (ensuring that something good happens eventually) .

Components in Formal Verification



Equivalence checking uses mathematical modeling techniques to prove that two representations of a design exhibit the same behavior.

Theorem proving is a proof-based approach to formal verification that uses mathematical reasoning to verify that the implemented system meets design requirements.



Security verification is a formal verification approach that uses a framework and a formal specification to prove the correctness of underlying algorithms.

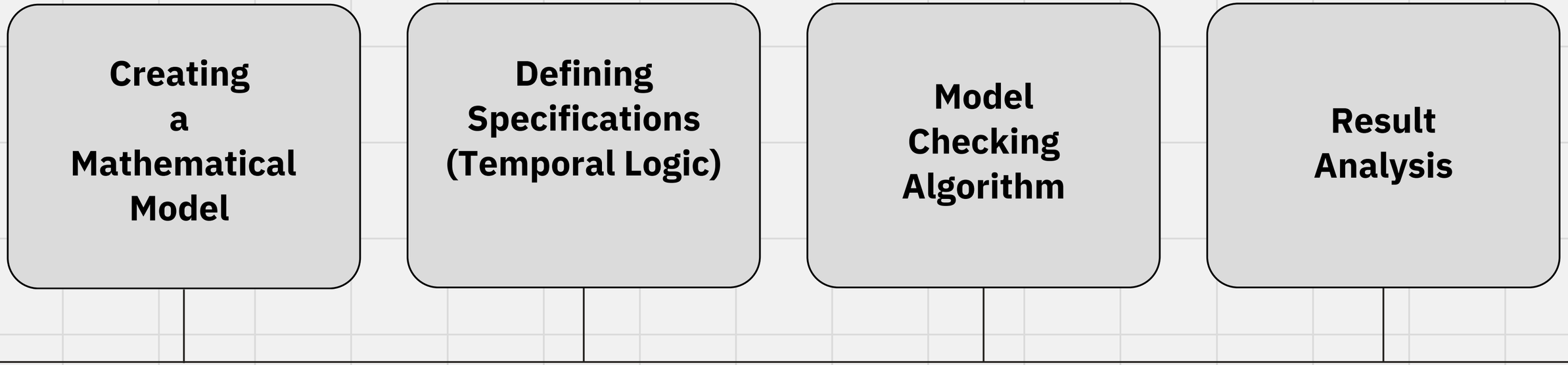
MODEL CHECKING

Model checking is formal verification technique that analyzes a system's behavior with respect to selected properties.

Inputs: Model’s specification and properties to be checked.

Output: A claim that the property is true or a counterexample that falsifies the property.

BASIC STEPS INVOLVED



SPIN MODEL CHECKER

An open-source software tool used to verify the correctness of software models.

Focuses on properties

1. Eventually
2. Liveliness
3. Assertion violations.

It is based on Promela modeling language.

Entities in Promela

Process

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

Variable

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

Channel

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

State Transition

```
:: (state == A) -> state = B
```

TRAFFIC LIGHT EXAMPLE IN PROMELA

INITIALIZATION

```
mtype = { GREEN,AMBER,RED };
mtype = { GO,G2S,S2G,STOP };

bool tick = false;
mtype status = GO;
mtype light = GREEN;
byte ctr = 0;
```

PROCTYPE

```
active proctype TrafficLight() {
  do
  ::
    if
    :: tick = false;
    :: tick = true;
    fi;

    if
    :: (status == GO) && (ctr == 3) && tick -> status = G2S;ctr = 0;
    :: (status == G2S) && (ctr == 1) && tick -> status = STOP;ctr = 0;
    :: (status == S2G) && (ctr == 1) && tick -> status = GO;ctr = 0;
    :: (status == STOP) && (ctr == 3) && tick -> status = S2G;ctr = 0;
    :: else -> ctr = (tick -> (ctr + 1) % 4 :  ctr);
    fi;
    if
    :: status == GO -> light = GREEN;
    :: status == G2S -> light = AMBER;
    :: status == S2G -> light = AMBER;
    :: status == STOP -> light = RED;
    fi;
  od;
}
```

LINEAR TEMPORAL LOGIC

```
ltl liveness { [](light == RED) -> <> (light == GREEN)) };
ltl sequence { [] <> tick -> [](light == RED) U ((light ==
AMBER) U (light == GREEN))) };
ltl always { [](light == GREEN) -> (status == GO)) };
```

```
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
// Safety Property: The lift will not move while the doors are open.
ltl p1 { [] (state == DoorOpen -> (state == Idle)) }

// Progress Property: The lift will eventually open its doors at the target floor after reaching it.
ltl p2 { [](current_floor == target_floor -> <> state == DoorOpen) }

// Strong Until Property: The lift continues to move up until it reaches the target floor.
ltl p3 { [](state == MovingUp U current_floor == target_floor) }

// Weak Until Property: The lift remains idle until it is directed to move up or down, even if no direction is set.
ltl p4 { [](state == Idle W (dir_up || dir_down)) }
```

MODEL CHECKER FOR A LIFT

Example of critical section problem using SPIN:-

- **Two process/servers running started together.**

What does this check?

- **“Asserts” if only one process is in critical section at a given point of time.**

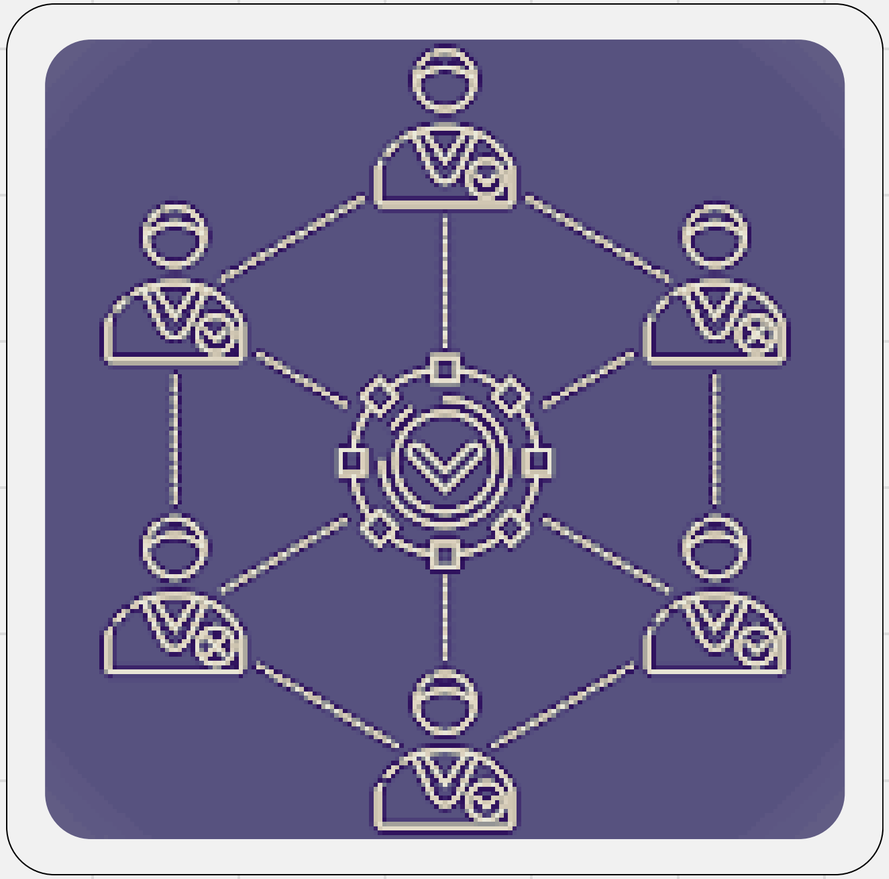
```
    do
    :: // Entry section
        flag[_pid] = true;
        turn = _pid;
// Wait until the other process is not interested or it's our turn
        (flag[1 - _pid] == false || turn == 1 - _pid);

// Critical Section
        cnt++;
        crit: assert(cnt == 1);// Assert that only one process is in the critical section
        cnt--;

        flag[_pid] = false;

// Exit section
    od
}

init {
// Initialize flags and turn
    flag[0] = false;
    flag[1] = false;
    turn = 0;

// Start processes
    run user();
    run user();
}
```

DISTRIBUTED SYSTEMS

WHAT ARE DISTRIBUTED SYSTEMS?

- NETWORK OF INTERCONNECTED SYSTEMS
- EXCHANGE MESSAGES WITH ONE ANOTHER
- NODES CORPORATE TO ACHIEVE A SINGLE OBJECTIVE
- APPEARS AS A SINGLE SYSTEM TO OUTSIDE WORLD

- **NEED OF DISTRIBUTED SYSTEMS HORIZONTAL SCALABILITY FAULT** TOLERANCE AND RELIABILITY PERFORMANCE



CONSENSUS ALGORITHMS

Rules / Protocols that allow a system to reach aggreament.

Benefits

- Decentralization
- Transparency
- Efficiency
- Security

TYPES OF CONSENSUS ALGORITHMS

- Proof of Work (PoW)
- Proof of Stake(PoS)
- Byzantine Fault Tolerance (BFT)
- Practical Byzantine Fault Tolerance
- DAG based Consensus
- Crash Fault Taulrence

Crash Fault Tolerant Algorithms

Zab

It is used by Apache Zookeeper to maintain ordering and consistency.

Paxos

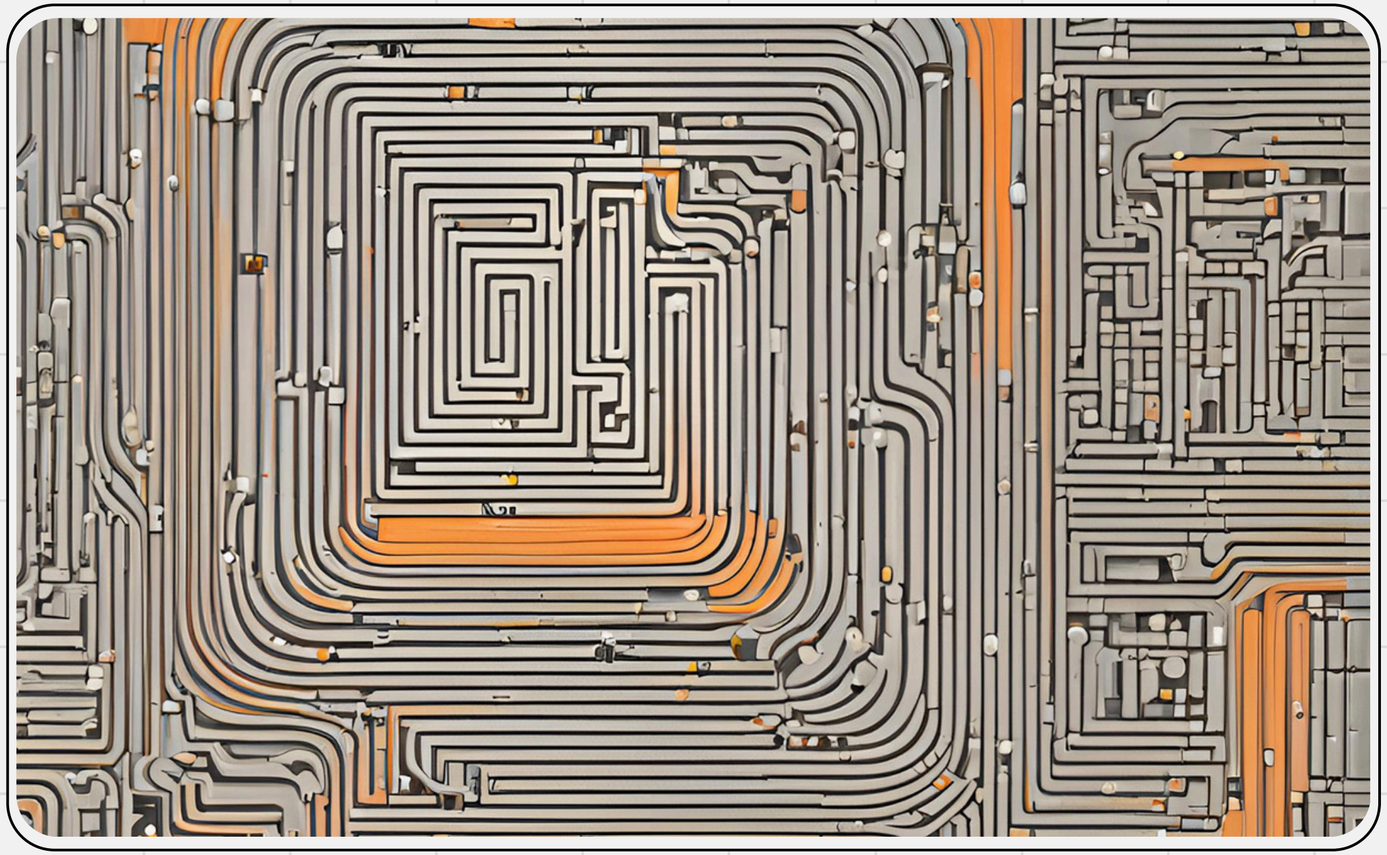
It allows a network of nodes to agree on a single value even in case of failure or incorrect behavior from some nodes.

Raft

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.

RAFT ALGRORITHM

Raft 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.



Raft Roles

LEADER

Coordinates the consensus process

Manage replication of logs

Sends heart beat messages to all other nodes in the system

CANDIDATE

Nodes which are running for election as the leader

They sends request votes to other nodes in the network to become the leader.

Becomes leader if got majority votes.

FOLLOWER

Listen to the leader and replicate the log.

Times out if no heart beat recieved from the leader

TERM

- 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 starts.

LEADER ELECTION PHASE

STEP 1

Nodes in the network compete to become the leader.

STEP 2

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.

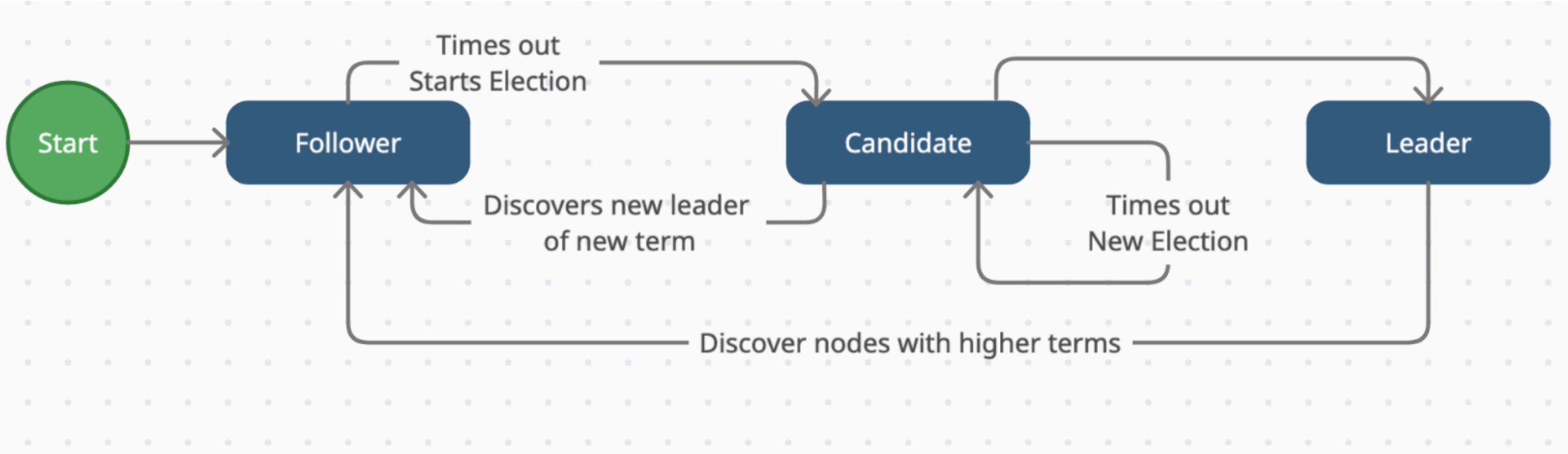
STEP 3

If a candidate receives votes from a majority of nodes, it becomes the leader.

STEP 4

Followers accept the leader if it has the most up-to-date log.

State Diagram and Transitions



Source:- https://ieeexplore.ieee.org/abstract/document/10062357

Transitions

FROM\TO	LEADER	CANDIDATE	FOLLOWER
LEADER	If it discovers a server with a higher term (AppendEntries or RequestVote RPC).	If it receives votes from a majority of the servers.	-
CANDIDATE	-	If the election timeout elapses without a majority, it restarts the election with an incremented term	If the election timeout expires without hearing from a leader.
FOLLOWER	If it discovers a server with a higher term (in AppendEntries or RequestVote RPC).	If it receives an AppendEntries from a valid leader (with a higher or equal term).	If it receives an AppendEntries (heartbeat) from a valid leader.

LOG REPLICATION PHASE

1.
Once a leader is elected, it sends heartbeat messages to all the followers.

2.
The leader receives requests from clients and appends them to its log.

3.
It then replicates the log to followers, who apply the changes to their own logs to maintain consistency

Failure Scenarios

LEADER CRASHED

If a leader crashes, followers will stop receiving heartbeats and transition to candidates after an election timeout.

A new election, and a new leader is chosen.

NETWORK PARTITIONS

If a network partition divides such that minority partition can’t form a quorum, all servers will remain candidates and followers.

When the network partition heals, servers with outdated terms will revert to followers after receiving messages from the current leader.

ORIGINAL IMPLEMENTATION

- Made as a course project of Software Formal Verification Tsinghua University.
- Formally verified Raft specification in SPIN.

ISSUES

1. Scalability:
   - Only takes into account 3 servers.
   - Only considers 3 terms and 3 logs stored at a time.

2. Hardcodes most of the transitions
3. Never ending simulation Redundancy
4. Redundant Channels(2 for election and 2 for logs)

CHANNELS DEFINITION

- **Entiries**

Append Entry Channels

Append Entry Response Channels

- **Voting**

Request Vote Channels

Request Vote Response Channels

TRANSITIONS HANDLED

Spin Transtions triggered:

1. Timeout for starting election (Candidate|Follower)
2. Timeout for restarting election
3. Become Leader(Candidate)
4. Request Vote(Candidate)
5. Request Vote Response(Follower/Candidate/Leader)
6. Handle Request Vote Response(Candidate)
7. Handle Client Request(Leader)
8. Append Entry(Leader)
9. Handle Append Entry (Follower)
10. Handle Append Entry Response(Leader)

Improvement Example

Original

```
:: // handle RequestVote
(requestVoteChannels[0].ch[serverId]?[requestVote] || requestVoteChannels[1].ch[serverId]?[requestVote] || requestVoteChannels[2]
ch[serverId]?[requestVote]) ->
atomic {
// calculate the id of the sender
    if
    :: requestVoteChannels[0].ch[serverId]?requestVote -> i = 0
    :: requestVoteChannels[1].ch[serverId]?requestVote -> i = 1
    :: requestVoteChannels[2].ch[serverId]?requestVote -> i = 2
    fi
    assert(i != serverId);
// update terms
    if
    :: (requestVote.term > currentTerm[serverId]) ->
        currentTerm[serverId] = requestVote.term;
        state[serverId] = follower;
        votedFor = NIL
    :: (requestVote.term <= currentTerm[serverId]) ->
        skip
    fi

    if
    :: (logs[serverId].logs[0] == 0) ->
        lastLogTerm = 0;
        lastLogIndex = 0
    :: (logs[serverId].logs[0] != 0 && logs[serverId].logs[1] == 0) ->
        lastLogTerm = logs[serverId].logs[0];
        lastLogIndex = 1
    :: (logs[serverId].logs[0] != 0 && logs[serverId].logs[1] != 0) ->
        lastLogTerm = logs[serverId].logs[1];
        lastLogIndex = 2
    fi

    logOk = requestVote.lastLogTerm > lastLogTerm || requestVote.lastLogTerm == lastLogTerm && requestVote.lastLogIndex >=
    lastLogIndex;
    requestVoteResponse.voteGranted = requestVote.term == currentTerm[serverId] && logOk && (votedFor == NIL || votedFor == i);

    requestVoteResponse.term = currentTerm[serverId];
    if
    :: requestVoteResponse.voteGranted -> votedFor = i
    :: !requestVoteResponse.voteGranted -> skip
    fi
    end_requestVoteResponse:        requestVoteResponse_ch[serverId].ch[i]!requestVoteResponse
}
```

Modified

```
:: 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];
                if
                :: requestVoteResponse.voteGranted -> votedFor = i
                :: !requestVoteResponse.voteGranted -> skip
                fi
                end_requestVoteResponse:        requestVoteResponseChannels[i].ch[serverId]!requestVoteResponse
                break;
            :: else -> i = i + 1
            fi
        :: else -> break;
        od;
        break
    }
```

LTL PROPERTIES CHECKED

ELECTION LIVELINESS

ELECTION SAFETY

LOG MATCHING

LEADER APPEND ONLY

STATE MACHINE SAFETY

Election Safety

Original

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

Try 1 (The most obvious one could think)

```
ltl electionSafety {
    always (
    forall i,j : (0 <= i && i < N && 0 <= j && j < N && i != j) ->
    (state[i] == LEADER && state[j] == LEADER) -> currentTerm[i] == currentTerm[j]
    )
}
```

Final Improvement

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

Result

```
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
```

FUTURE WORK

- Improving State Machine Safety to n servers
- Improving Leader Completeness to n servers
- Using of global states where ever possible
- Using single channel for voting and single for entries.
- Figuring out a way to writing bi-directional channel

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

THANK YOU!