Model Checking the Safety of Raft Leader Election Algorithm

Qihao Bao, Bixin Li, Tianyuan Hu, and Dongyu Cao
School of Computer Science and Engineering, Southeast University
Nanjing 211189, Jiangsu Province, P.R. China
bx.li@seu.edu.cn

Abstract*—With the wide application of the Raft consensus algorithm in blockchain systems, its safety has attracted more and more attention. However, although some researchers have formally verified the safety of the Raft consensus algorithm in most scenarios, there are still some safety problems with Raft consensus algorithm in some special scenarios, and cause problems now and then. For example, as a core part of the Raft consensus algorithm, the Raft leader election algorithm usually faces some safety problems in following scenarios: if the network communication between some nodes is abnormal, the leader node could be unstable or even cannot be elected, or the log entry cannot be updated, etc. In this paper, we model check the safety of the Raft leader election algorithm throughly using Spin. We use Promela language to model the Raft leader election algorithm and use Linear-time Temporal Logic (LTL) formulae to characterize three safety properties including *stability*, *liveness*, and *uniqueness*. The verification results show that the Raft leader election algorithm does not hold *stability* and *liveness* when some nodes are faulty and node log entries are inconsistent. For these safety problems, we give the suggestions for improving safety by analyzing counter examples.

***Keywords*—blockchain system; Raft consensus algorithm; model checking; Promela; Spin**

I. Introduction

As the underlying technology of Bitcoin, blockchain technology has immediately become the focus of research in computer science since it was introduced in 2008 [1]. With the increasing maturity of blockchain technology, finance, energy, medical, and other fields have also been further developed. As a decentralized and distributed system, blockchain uses consensus algorithms to achieve data consistency and reach an agreement on a proposal among scattered and parallel nodes. In order to meet the application requirements of different scenarios, many consensus algorithms have been proposed.

At present, according to whether the system allows Byzantine nodes, consensus algorithms can be divided into CFT (Crash Fault Tolerance) consensus and BFT (Byzantine Fault Tolerance) consensus [2]. Among these consensus algorithms, the Paxos algorithm has been recognized as the most classic distributed consensus algorithm since it was proposed [3]. However, it is not only difficult to understand, but also difficult to implement the Paxos algorithm in distributed systems. For this purpose, the Raft consensus algorithm is proposed as an algorithm that is as efficient as Paxos and easier to understand and implement. At present, the Raft consensus algorithm is widely used in traditional distributed systems, consortium blockchains, and private blockchains.

This work is supported by the National Key Research and Development Program of China (under no.2019YFE0105500), and partially supported by the Key Research and Development Program of Jiangsu Province (under no.BE2021002-3) and the National Natural Science Foundation of China (under no. 61872078).

With the extensive application of the Raft consensus algorithm in distributed systems, its safety research becomes more and more urgent because the unsafety consensus algorithms may lead to the abnormal operation or even crash of distributed systems. Unfortunately, only a handful of studies have been done on the safety of the Raft consensus algorithm. Ongaro and Ousterhout analyzed the safety of the Raft consensus algorithm and proved the Log Completeness Property with TLA proof system while proposing the Raft consensus algorithm [4]. Woos et al. have proven the Raft safety by theorem proving its unique properties: state machine safety, election safety, log matching, and leader integrity [5]. However, when studying the safety of the Raft leader election algorithm, they only considered the node crash, but not the network abnormality between some nodes. They also did not analyze the stability or liveness of the Raft leader election algorithm.

Based on the above insight, we use a formal method to verify the safety of the Raft leader election algorithm. We use Promela language to model the Raft leader election algorithm and simulate different fault types, such as partial node crash, the network anomaly between partial nodes, and network partition. Then we describe the three safety properties of stability, liveness, and uniqueness into formal language by LTL formulae and verify them with the Spin tool. By analyzing the counterexamples given by the Spin tool, the safety problems that exist in the Raft leader election algorithm are found. Finally, we analyze the existing safety problems and give the suggestions for improving safety.

The main contributions of this paper are summarized as follows:

1) When modeling the Raft leader election algorithm in Promela language, we consider and analyze all kinds of node faults tolerated by the Raft consensus algorithm, including less than half of the nodes crash, the network anomaly between some nodes, and network partition;

2) We study some typical safety properties of the Raft leader election algorithm not only uniqueness, but also stability and liveness. By analyzing the results of formal verification, we find that the Raft leader election algorithm has the safety problems of stability when the network between some nodes is abnormal. When some nodes are faulty and the logs of nodes are inconsistent due to the fault, the Raft leader election algorithm will have the liveness problem.

The rest of this paper is organized as follows. In section II, we introduce the preliminary knowledge of Raft consensus algorithm. In section III, we describe the model of the Raft leader election algorithm based on Promela. In section IV, we use LTL formulae to characterize the safety properties of the Raft leader election algorithm. In section V, we use the Spin tool to verify the model and analyze the verification results. Next, we discuss related works in section VI. Finally, we make a conclusion for our work in section VII.

II. Background Knowledge

A. Raft Consensus Algorithm

The Raft consensus algorithm is a simplified algorithm based on the Paxos algorithm, which enhances intelligibility and is comparable to Paxos in terms of performance, reliability, and usability. Raft simplifies the states and splits the problems on the basis of Paxos, breaking down the previously complex logic into several sub-problems, which can be summarized into the following aspects: leader election, log replication, and safety.

1) Leader Election: In the Raft consensus algorithm, each node is in one of three states at any given time: follower, candidate, leader. Followers respond only to requests from the leader and candidate, but do not send requests. Candidates are formed by followers during the cluster election. If a candidate obtains a majority of votes in the election, it becomes the leader. The leader is elected by all nodes from the candidates and is responsible for the status of the entire cluster and data management. The states of the node and their transitions are shown in Figure 1. Furthermore, Raft divides time into terms of arbitrary length, which are numbered with consecutive integers. Each term begins with an election, in which one or more candidates try to become leaders.

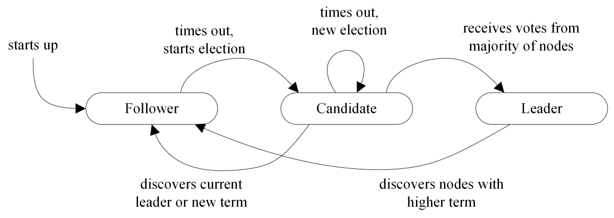

starts up
times out, starts election
times out, new election
receives votes from majority of nodes
Follower
Candidate
Leader
discovers current leader or new term
discovers nodes with higher term

Figure 1. Node states and their transitions

2) Log Replication: Once a leader has been elected, it begins servicing client requests. The client sends the request for updating the log entry to the leader, and the leader sends the messages to other followers after updating it locally. After receiving the messages, the followers update the log entry after determining that the new log entry does not conflict with the local log entry and return the messages to the leader. After the leader receives more than half of the log entry update messages from followers, the leader formally writes the log entry to the state machine and the log entry state becomes "committed". The leader then sends a message to notify followers and the client respectively. Followers will write their logs to the local state machine after receiving the messages, and the state of their entries on followers will change to "committed". At this point the log entry is actually in effect and cannot be rolled back.

3) Safety: To make no irreversible errors or return incorrect content to the client under any circumstances, Raft adds some restrictions on leader election and log replication. Leadership election restriction means that followers only vote for the candidate whose log is more up-to-date than its own. A more up-to-date log means that the last entry in the log has a higher term, or the log is longer with the same term. Log replication restriction means that the leader does not commit log entries with previous terms, but instead commit log entries with previous terms by committing log entries with the current term.

B. Safety of the Raft Leader Election Algorithm

The safety of the consensus algorithms means that in the process of reaching an agreement, the system reaches an unsafe state due to the triggering of some conditions. In this state, the system reports errors or even crashes, resulting in abnormal system operation and major safety accidents.

In the Raft leader election algorithm, safety research also needs to be focused on. For the leadership election, the main safety concerns include:

1) Whether the leader node is stable. The state of the leader node should remain stable while the leader node is in normal communication with most nodes. Otherwise, frequent leader changes will result in unstable log updates.

2) Whether a leader node will be elected over time. Normally, a system will elect a leader node to manage data over time. If no leader node is ever elected, the logs cannot be updated and the distributed system cannot function properly.

3) Whether the elected leader is unique in each term. If the leader node is not unique, then there will be multiple leader nodes to manage the system data at the same time, resulting in follower nodes do not know which leader node to respond to the log update request.

C. Formal Verification

The safety of consensus algorithms is often verified informally. When the algorithms become complex, the non-formal safety verification is prone to error. Only when the algorithm is formally validated can it gain sufficient credibility.

Formal verification is used in hardware and software systems to prove or deny the correctness of the expected algorithm based on the system according to certain formal specifications or properties by using mathematical formal methods [6]. Formal verification verifies the reliability of a program by mathematical logic so that it can be proven that a system does not have a defect or conforms to some properties. A large number of industrial practices show that when developing complex hardware and software systems, people spend more effort on verifying the correctness of the system than on building the system. At present, formal verification can be divided into three categories: equivalence checking, theorem proving, and model checking [6]–[8].

Model checking is the most widely used formal verification method because of its full automation and fast verification speed. In model checking, many auxiliary checking tools are used, and Spin is one of the most popular model checking tools. Using Promela as the input language, the Spin tool can check the logical consistency of specifications in the design of algorithms, and report the occurrence of deadlock, invalid loops, undefined reception and incomplete markup in the system.

III. Formal Modeling Based On Promela

Since Woos et al. have provided a complete and detailed theorem proof on the safety of log replication in the Raft consensus algorithm, this paper focuses on the formal modeling and verification of the safety of leader election.

Through the description of Raft consensus algorithm in Section II, we clearly know the process of the algorithm. To facilitate formal modeling and verification, we first abstract the Raft leader election algorithm and construct the mapping relationship between the Raft leader election algorithm and the Promela model, as shown in Figure 2.

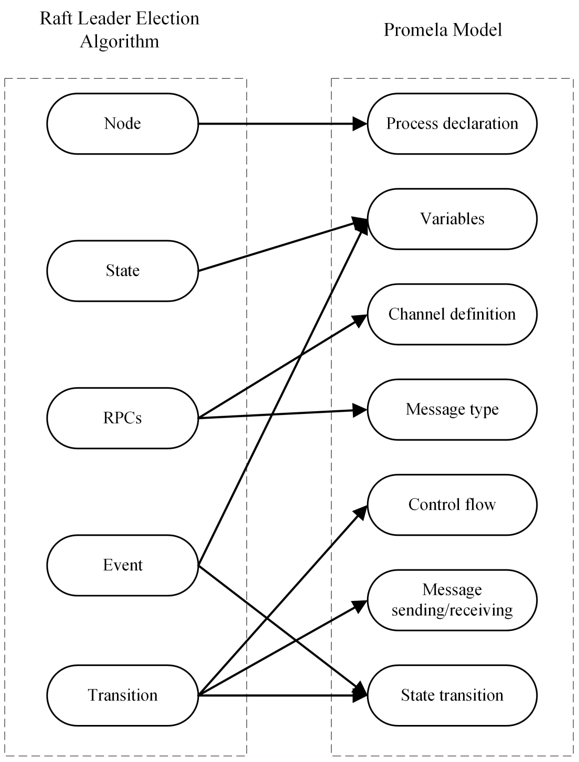


Figure 2. Mapping relationship between the Raft leader election algorithm and Promela model

The Promela model consists of processes, message channels, and variables, which is equivalent to a finite transformation system. The mapping relationship from the Raft leader election algorithm to the Promela model includes mapping from state transition to message definition, from state transition to process control inside process entities, message reception, and so on. The detailed mapping relationship is described as follows:

- **Mapping from node to process declaration.** In the Promela model, there are two types of processes, namely the normal process and the initial process, where the initial process is similar to the main() function in C language. Each node in the distributed system corresponds to a normal process, and the operation of each node is implemented through the execution of statements in the process body.
- **Mapping from state to variables.** In the Raft leader election algorithm, each node is in one of three states at any given time: follower, candidate, and leader. These three states are defined with corresponding enumeration variables.
- **Mapping from RPCs to channel definition and message type.** Nodes in the Raft consensus algorithm communicate using remote procedure calls (RPCs). In the Promela model, processes communicate with each other through message channels. The number of messages that each channel can hold and the data type of the messages need to be defined together with the channel definition.
- **Mapping from event to variables and message sending/receiving.** The state of the node is changed by different events, such as node timeout, receiving a message that the leader node has won, and so on. Corresponding to the Promela model, event triggering is simulated through changing the values of variables and sending and receiving messages.
- **Mapping from transition to process body.** The process body in the Promela model mainly includes three parts: control flow, message sending/receiving and state transition. Control flow corresponds to state transitions in the Raft leader election algorithm, which are guided by different control conditions. Message sending/receiving is the communication part between processes, corresponding to state activities and transition conditions, and related to the above message type and channel definition. Each process needs to use a specific channel to receive and send messages. State transition is a jump after the message is received or sent in the process, which corresponds to the transfer in the Raft leader election algorithm. There are different transfer conditions according to different transitions. If the conditions are met, the next step will be executed.

A. Preparing Work

The leader election process mainly includes four basic types: participant, term, state of participant, and log on the node. The participant and its state only need to be distinguishable, and the term and log can be compared.

Since there are no string variables in Promela language, and to simplify the modeling process, we define the basic types directly with natural numbers to meet the actual requirements of the Raft leader election algorithm.

Therefore, we use enumeration types to define the identity of the participant and the corresponding state. Specific definitions are as follows:

/*
mtype = {follower, candidate, leader};
mtype = {N1, N2, N3, N4, N5};
*/

Since the term and log are only used for comparison in the leader election, we define them as int types in local variables. Larger numbers of term and log indicate higher term and more up-to-date log, respectively. The voting situation is defined by bool type, indicating agree to vote and reject vote.

B. Message Abstraction

In the Promela model, all communication between processes is achieved through channels, so it is important to abstract the message. By analyzing leadership election, it is clear that the process is divided into vote requests and response requests.

The voting request means that a follower turns into a candidate after timing out and sends a voting request message to other nodes. The voting request message contains the node identity, the node state, the term that the node is currently in, and the log on the node.

The response request is that the node responds to the vote after receiving the vote request message. The response request message includes the node identity, the node state, and consent vote. In the Raft leader election algorithm, a node will refuse to respond to a vote request if it receives a vote request from a node whose term is lower or whose log is less up-to-date, or if the node has already voted in the current term. However, in order to prevent message overflow in the channel, we set the model to respond to the voting request regardless of whether we agree with the vote or not. When the node rejects a vote, it will send a reject vote message.

Specific definitions of channels are as follows:

/*
chan RPC[10] = [10] of {mtype, mtype, int, int};
chan REPLY[10] = [10] of {mtype, mtype, bool};
*/

The RPC channel is the voting request channel. Each RPC channel contains four message types, which respectively represent the node identity, the node state, the term that the node is currently in, and the log entry on the node. The REPLY channel is a voting feedback channel. Each REPLY channel contains three message types, which respectively represent the node identity, the node state, and consent vote, and whether to agree to vote.

In addition, if the node fails to receive messages from the leader due to communication failure, it can still know the current tenure through other nodes that can be contacted. Therefore, we define an additional message channel to inform the current term. When the node knows that the leader is elected, it sends the current term to other nodes that can be contacted, which is shown as follows.

/*
chan term_now[20] = [10] of {int};
*/

C. Algorithm Modeling

In the previous work, the underlying types and message mechanisms involved in the algorithm were abstracted and defined in Promela language. Based on this, we will fully model how nodes elect a leader in the Raft leader election algorithm.

There is a randomized election timeout on each node. When the node times out, it will start an election. But because Promela does not provide any time features, the built-in timeout function is just a modeling feature that provides a way out of the suspended state. The suspended state does not correspond to the real timer definition. Therefore, we model the timeout as follows.

```
do
:: (time_out==0) ->option1;
:: (time_out!=0) ->option2;
od;
```

Listing 1. The code for the timeout model

When $time\_out$ is 0, the node will initiate a voting request. When $time\_out$ is not 0, the node will observe whether there is a vote request message in the channel and count down until the node initiates a vote request when $time\_out$ is 0.

During waiting for that timeout to elapse, the node will loop through the channels associated with it to see if there is a vote request message. If the node receives a vote request message, it will evaluate the vote message accordingly, including whether the term is higher than its own and log entry is more up-to-date, and determines whether it has already voted. If one condition is not satisfied, the node will refuse to vote. If the node receives the message that one node is elected as the leader during this period, it stops waiting for timeout to elapse and informs other nodes of its term.

```
::  (time_out!=0)->
    vote_if = 0;
    atomic{
    do
    ::  RPC[0]?candidate,N2,term_current,index_current;
        if
        ::  (term_current >= term&&index_current >= index
            &&vote_if == 0) ->
            Reply[0]!follower,N1,1;
            vote_if = 1;
        ::  else-> Reply[0]!follower,N1,0;
        fi;
    ::  RPC[0]?leader,N2,term,index ->
        term_now[0]!term;
        break;
    ::  empty(RPC[0])->break;
    od;
    }
    time_out--;
```

Listing 2. Part of code that the node does not time out

In order to avoid the impact of the uncertainty of the relative speed in the parallel process and reduce the complexity of the whole process, we add the keyword "atomic" as a prefix to the whole process, indicating that the statement sequence will be executed as an indivisible whole. In addition, the Empty function is used to check if there are any unreceived messages in the channel, lest the node receive outdated messages due to asynchronous communication. Part of the code is as shown in Listing 2.

```
:: (time_out==0)->
    term++;
    if
    :: term > term_current ->
        leaders = 0;
    :: else ->skip;
    fi;
    vote=0;
    RPC[0]!candidate,N1,term,index;
    vote++;
    atomic{
    if
    :: nempty(Reply[0])->
        do
        :: Reply[0]?follower,N2(1)
            ->vote++;
        :: Reply[0]?follower,N2(0)
            ->skip;
        ...
        :: empty(Reply[0])&&empty(RPC[0])->
            if
            :: vote>=3 ->
                leaders ++;
                isLeader = 1;
                leader[0] = 1;
                RPC[0]!leader,N1,term,index;
                term_now[0]!term;
                ...
                break;
            :: else -> time_out = 1; break;
            fi;
        od;
    fi;}
```

Listing 3. Part of code that the node times out

When the node times out, it will increment the current term by 1 and send a vote request message to each of the other nodes separately. After traversing all REPLY channels, if the number of votes received exceeds half of the number of the nodes in the cluster, the node is elected as the leader and sends the elected message to other nodes. If the number of votes is less than half, it means the election is lost and the node restarts the timeout. Part of the code is as shown in Listing 3.

```
do
:: term_now[0]?term_current;
    if
    :: term_current > term ->
        term = term_current;
        goto leader_election;
    :: else -> skip;
    fi;
...
od;
```

Listing 4. Part of the code that the nodes inform each other of the current term

When a node wins the election, all nodes end the timeout, and the node will inform the node that can be contacted of the current term. So we add a loop to the model in which the node restarts the election timeout if it learns from other nodes that the current term is larger. Part of the code is as shown in Listing 4, where *leader_election* is the label of the voting process.

Through the above analysis, we ensure and believe in the consistency between the formal modeling of leader election of Raft and the reality of leader election of Raft.

IV. Characterization of Safety Properties

In the Spin tool, the safety properties of the Raft leader election algorithm are characterized using the LTL formulae. For the Raft leader election algorithm, we mainly characterize the following safety properties:

- **Property 1** (Stability). When the leader communicates with the most nodes normally, the leader state of the node remains stable.
- **Property 2** (Liveness). After a period of time, one leader must be elected.
- **Property 3** (Uniqueness) There is at most one leader per term.

The above properties can be expressed with the help of atomic predicates. In the Promela model, we describe the properties by defining some variables.

/*
int leaders = 0; bool isLeader = 0; int connect[];
bool leader[];
*/

"leaders" indicates the number of leaders. "isLeader" indicates whether there is a leader in the system. If a leader exists, isLeader = 1. "connect[]" and "leader[]" indicate the number of network connections of each node and whether the node is elected as the leader.

For property 1, when "connect[i]" $\geqslant$ 3 and "leader[i] " = 1, "leader[i]" will always be 1. For property 2, "isLeader" will equal 1 eventually. For property 3, "leaders" always do not exceed 1.

The above safety properties are translated into the LTL formulae as follows:

- **Property 1.** $(connect[i] \geqslant 3 \ \&\& \ leader[i] == 1) - > [](leader[i] == 1)$.
- **Property 2.** $<> (isLeader == 1)$.
- **Property 3.** $[](leaders \leqslant 1)$.

V. Verification

A. Experiment Settings

In this paper, we used Spin 6.4.9 Version to formally verify the safety of the Raft leader election algorithm in Windows 10 system.

As described in the Raft paper by Ongaro and Ousterhout [4], 5 nodes allow the system to tolerate 2 failures. But Ongaro and Ousterhout did not provide a clear explanation for other node faults, which means that a node may not crash, but it can be disconnected from some nodes in the cluster. In this paper, we set a total of three fault types: 1) Network anomaly: one node cannot communicate with a certain node normally, but can communicate with other nodes normally; 2) Network partition: the whole cluster is divided into multiple groups, the nodes in each group can communicate normally, but no node between groups can communicate normally; 3) Node crash: the node cannot communicate with all nodes in the cluster. In the Promela model, different nodes communicate through channels, so we simulate the fault types of nodes by removing the corresponding channels between nodes.

After modeling the Raft leader election algorithm in Promela language and characterizing the safety properties in LTL formulae, the Spin tool can be used for automated formal verification. The verification results of the Spin tool will show whether the model design is correct or meets the safety properties described. When the design contains errors or does not meet safety properties, the Spin tool gives an error description. With the run instructions of the Spin tool, error trajectory can be generated to obtain the false counterexamples. The defects or problems in the design can be analyzed according to the false counterexamples.

B. *Verification Result for Property 1*

The Spin tool gives an error result when we set the node fault to Figure 3. In the cluster, normal communication is maintained between all nodes except N1 and N5. N1 cannot send messages to N5 and cannot receive messages from N5. The corresponding verification result is shown in Figure 4.

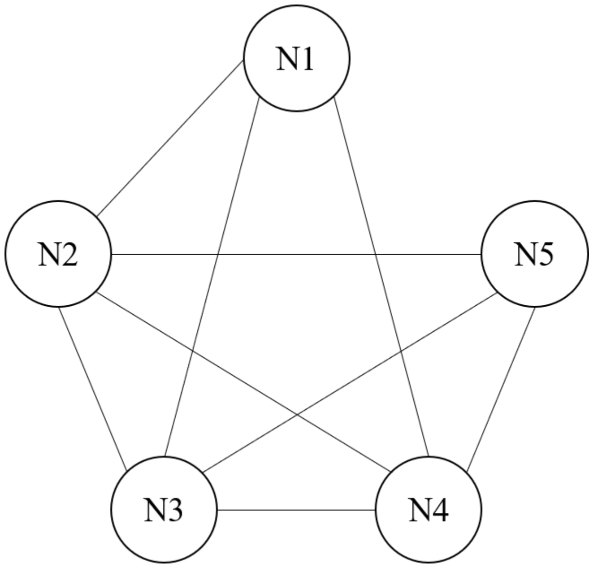
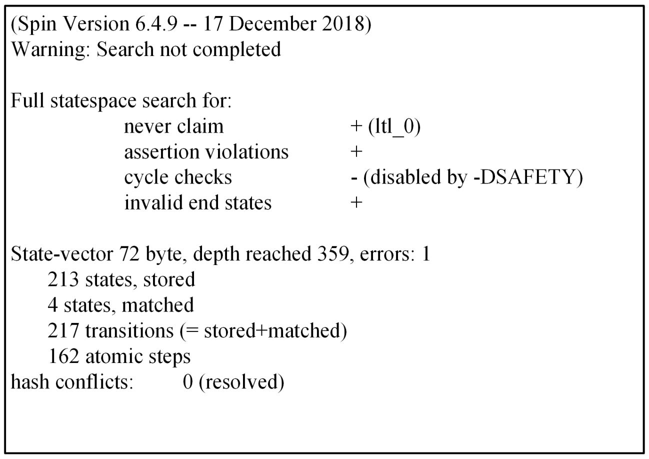


Figure 3. Fault settings when the system violates Property 1

The Spin tool gives the error trajectory, which we have collated and simplified as shown in Table 1. N1 is the first to time out, thus sending a vote request message to N2, N3, and N4. Since N2, N3, and N4 are in the follower state and do not vote for other nodes, they all agree to vote for N1 and send the messages. After receiving the message from N2, N3, and N4, N1 is elected as the leader because the number of votes reaches 4 (including its own), which is the majority of the number of nodes in the cluster. After N1 informs N2, N3, and N4 of the election, N2, N3, and N4 also inform the nodes they can contact about the current term. However, due



```
(Spin Version 6.4.9 -- 17 December 2018)
Warning: Search not completed

Full statespace search for:
            never claim             + (ltl_0)
            assertion violations    +
            cycle checks            - (disabled by -DSAFETY)
            invalid end states      +

State-vector 72 byte, depth reached 359, errors: 1
     213 states, stored
     4 states, matched
     217 transitions (= stored+matched)
     162 atomic steps
hash conflicts:       0 (resolved)
```

Figure 4. Verfication result for Property 1 when the fault is set as in Figure 3

to communication fault between N5 and N1, N5 can only learn the term from other nodes without knowing the message that N1 has been elected the leader. When N5 times out, N5 increments the term and sends a vote request message to N2, N3, and N4. Just like N1 is elected as the leader, N5 will be elected as the leader by a majority of votes. Then N1 learns from other nodes that the current term is larger than its own, and is downgraded to the state of follower. However, since we do not know that the leader exists in the cluster, N1 will restart its election timeout and initiate a new vote request. Thus, the leader switches between N1 and N5 constantly, violating property 1.

C. *Verification Result for Property 2*

The Spin tool gives an error result when we set the node fault to Figure 5. In the cluster, normal communication is maintained between all nodes except N1 and N5. N2 has a communication fault with N4 and N5, and N4 and N5 also have a communication fault with each other. In addition, the log entries on N2 and N3 are less up-to-date than those on other nodes, which are highlighted in red. The corresponding verification result is shown in Figure 6.

The Spin tool gives the error trajectory, which we have collated and simplified as shown in Table 2. N1 times out but because it crashes, it does not send a vote request message to other nodes and will not receive a reply message from other nodes. N2 times out and sends a voting request message to N3. After receiving the message, N3 returns a message agreeing to vote to N2. However, due to the communication fault, N2 can only get the votes of N3 and herself, which is less than half of the number of the nodes in the cluster, so N2 cannot be elected as the leader. When N3 times out, it sends a vote request message to N2, N4, and N5. But since its log entry is less up-to-date than the log entry on N4 and N5, N3 will only get N2's vote, and N4 and N5 will refuse to vote for it. Similar to N2, when N4 and N5 time out, they can only get N3 and their own votes, so they cannot be elected as leaders due to insufficient votes. In summary, none of the nodes in the

Table 1. Error Trajectory of Verfication Result for Property 1

Process Statement		RPC[0]	RPC[1]	RPC[2]	RPC[3]	RPC[4]	RPC[5]	REPLY[0]	REPLY[1]	REPLY[2]	REPLY[3]	REPLY[4]	REPLY[5]	term_now[0]	term_now[1]
1 node_1	time_out != 0;														
1 node_1	time_out = time_out-1;														
1 node_1	time_out == 0;														
1 node_1	term = term+1;														
1 node_1	vote = vote+1;														
1 node_1	RPC[0]! candidate, N1, 1, 0;	[candidate, N1, 1, 0]													
1 node_1	RPC[1]! candidate, N1, 1, 0;	[candidate, N1, 1, 0]	[candidate, N1, 1, 0]												
1 node_1	RPC[2]! candidate, N1, 1, 0;	[candidate, N1, 1, 0]	[candidate, N1, 1, 0]	[candidate, N1, 1, 0]											
2 node_2	RPC[0]? candidate, N1, 1, 0;		[candidate, N1, 1, 0]	[candidate, N1, 1, 0]											
2 node_2	REPLY[0]! follower, N2, 1;		[candidate, N1, 1, 0]	[candidate, N1, 1, 0]				[follower, N2, 1]							
3 node_3	RPC[1]? candidate, N1, 1, 0;			[candidate, N1, 1, 0]				[follower, N2, 1]							
3 node_3	REPLY[1]! follower, N3, 1;			[candidate, N1, 1, 0]				[follower, N2, 1]	[follower, N3, 1]						
4 node_4	RPC[2]? candidate, N1, 1, 0;							[follower, N2, 1]	[follower, N3, 1]						
4 node_4	REPLY[2]! follower, N4, 1;							[follower, N2, 1]	[follower, N3, 1]	[follower, N4, 1]					
1 node_1	REPLY[0]? follower, N2, 1;								[follower, N3, 1]	[follower, N4, 1]					
1 node_1	vote = vote+1;								[follower, N3, 1]	[follower, N4, 1]					
1 node_1	REPLY[1]? follower, N3, 1;									[follower, N4, 1]					
1 node_1	vote = vote+1;									[follower, N4, 1]					
1 node_1	REPLY[2]? follower, N4, 1;														
1 node_1	vote = vote+1;														
1 node_1	vote>= 3;														
1 node_1	leaders = leaders+1;														
1 node_1	isLeader = 1;														
1 node_1	leader[0] = 1;														
1 node_1	RPC[0]! leader, N1, 1, 0;	[leader, N1, 1, 0]													
1 node_1	RPC[1]! leader, N1, 1, 0;	[leader, N1, 1, 0]	[leader, N1, 1, 0]												
1 node_1	RPC[2]! leader, N1, 1, 0;	[leader, N1, 1, 0]	[leader, N1, 1, 0]	[leader, N1, 1, 0]											
2 node_2	RPC[0]? leader, N1, 1, 0;		[leader, N1, 1, 0]	[leader, N1, 1, 0]											
3 node_3	RPC[1]? leader, N1, 1, 0;			[leader, N1, 1, 0]											
4 node_4	RPC[2]? leader, N1, 1, 0;														
4 node_4	term_now[0]! 1;													[1]	
5 node_5	term_now[0]? 1;														
5 node_5	time_out != 0;														
5 node_5	time_out = time_out-1;														
5 node_5	time_out == 0;														
5 node_5	term = term+1;														
5 node_5	vote = vote+1;														
5 node_5	RPC[3]! candidate, N5, 2, 0;				[candidate, N5, 2, 0]										
5 node_5	RPC[4]! candidate, N5, 2, 0;				[candidate, N5, 2, 0]	[candidate, N5, 2, 0]									
5 node_5	RPC[5]! candidate, N5, 2, 0;				[candidate, N5, 2, 0]	[candidate, N5, 2, 0]	[candidate, N5, 2, 0]								
2 node_2	RPC[3]? candidate, N5, 2, 0;					[candidate, N5, 2, 0]	[candidate, N5, 2, 0]								
2 node_2	REPLY[3]! follower, N2, 1;					[candidate, N5, 2, 0]	[candidate, N5, 2, 0]				[follower, N2, 1]				
3 node_3	RPC[4]? candidate, N5, 2, 0;						[candidate, N5, 2, 0]				[follower, N2, 1]				
3 node_3	REPLY[4]! follower, N3, 1;						[candidate, N5, 2, 0]				[follower, N2, 1]	[follower, N3, 1]			
4 node_4	RPC[5]? candidate, N5, 2, 0;										[follower, N2, 1]	[follower, N3, 1]			
4 node_4	REPLY[5]! follower, N4, 1;										[follower, N2, 1]	[follower, N3, 1]	[follower, N4, 1]		
5 node_5	REPLY[3]? follower, N2, 1;											[follower, N3, 1]	[follower, N4, 1]		
5 node_5	vote = vote+1;											[follower, N3, 1]	[follower, N4, 1]		
5 node_5	REPLY[4]? follower, N3, 1;												[follower, N4, 1]		
5 node_5	vote = vote+1;												[follower, N4, 1]		
5 node_5	REPLY[5]? follower, N3, 1;														
5 node_5	vote = vote+1;														
5 node_5	vote >=3;														
5 node_5	leaders = leaders+1;														
5 node_5	isLeader = 1;														
5 node_5	leader[4] = 1;														
5 node_5	RPC[3]! leader, N5, 2, 0;				[leader, N5, 2, 0]										
5 node_5	RPC[4]! leader, N5, 2, 0;				[leader, N5, 2, 0]	[leader, N5, 2, 0]									
5 node_5	RPC[5]! leader, N5, 2, 0;				[leader, N5, 2, 0]	[leader, N5, 2, 0]	[leader, N5, 2, 0]								
2 node_2	RPC[3]? leader, N5, 2, 0;					[leader, N5, 2, 0]	[leader, N5, 2, 0]								
3 node_3	RPC[4]? leader, N5, 2, 0;						[leader, N5, 2, 0]								
4 node_4	RPC[5]? leader, N5, 2, 0;														
2 node_2	term_now[1]! 2;														[2]
1 node_1	term_now[1] ?2;														
1 node_1	time_out != 0;														
1 node_1	time_out = time_out-1;														
1 node_1	time_out == 0;														

Table 2. Error Trajectory of Verification Result for Property 2

Process Statement		RPC[6]	RPC[7]	RPC[8]	REPLY[6]	REPLY[7]	REPLY[8]
1 node_1	time_out != 0;						
1 node_1	time_out = time_out-1;						
1 node_1	time_out == 0;						
1 node_1	term = term+1;						
1 node_1	vote = vote+1;						
2 node_2	time_out != 0;						
2 node_2	time_out = time_out-1;						
2 node_2	time_out == 0;						
2 node_2	term = term+1;						
2 node_2	vote = vote+1;						
2 node_2	RPC[6]! candidate, N2, 1, 0;	[candidate, N2, 1, 0]					
3 node_3	RPC[6]? candidate, N2, 1, 0;						
3 node_3	REPLY[6]! follower, N3, 1;				[follower, N3, 1]		
2 node_2	REPLY[6]? follower, N3, 1;						
2 node_2	vote = vote+1;						
3 node_3	time_out != 0;						
3 node_3	time_out = time_out-1;						
3 node_3	time_out == 0;						
3 node_3	term = term+1;						
3 node_3	vote = vote+1;						
3 node_3	RPC[6]! candidate, N3, 2, 0;	[candidate, N3, 2, 0]					
3 node_3	RPC[7]! candidate, N3, 2, 0;	[candidate, N3, 2, 0]	[candidate, N3, 2, 0]				
3 node_3	RPC[8]! candidate, N3, 2, 0;	[candidate, N3, 2, 0]	[candidate, N3, 2, 0]	[candidate, N3, 2, 0]			
2 node_2	RPC[6]? candidate, N3, 2, 0;		[candidate, N3, 2, 0]	[candidate, N3, 2, 0]			
2 node_2	REPLY[6]! follower, N2, 1;		[candidate, N3, 2, 0]	[candidate, N3, 2, 0]	[follower, N2, 1]		
4 node_4	RPC[7]? candidate, N3, 2, 0;			[candidate, N3, 2, 0]	[follower, N2, 1]		
4 node_4	REPLY[7]! follower, N4, 0;			[candidate, N3, 2, 0]	[follower, N2, 1]	[follower, N4, 0]	
5 node_5	RPC[8]? candidate, N3, 2, 0;				[follower, N2, 1]	[follower, N4, 0]	
5 node_5	REPLY[8]! follower, N5, 0;				[follower, N2, 1]	[follower, N4, 0]	[follower, N5, 0]
3 node_3	REPLY[6]? follower, N2, 1;					[follower, N4, 0]	[follower, N5, 0]
3 node_3	vote=vote+1;					[follower, N4, 0]	[follower, N5, 0]
3 node_3	REPLY[7]? follower, N4, 0;						[follower, N5, 0]
3 node_3	REPLY[8]? follower, N5, 0;						
4 node_4	time_out != 0;						
4 node_4	time_out = time_out-1;						
4 node_4	time_out == 0;						
4 node_4	term = term+1;						
4 node_4	vote = vote+1;						
4 node_4	RPC[7]! candidate, N4, 3, 1;		[candidate, N4, 3, 1]				
3 node_3	RPC[7]? candidate, N4, 3, 1;						
3 node_3	REPLY[7]! follower, N3, 1;					[follower, N3, 1]	
4 node_4	REPLY[7]? follower, N3, 1;						
4 node_4	vote = vote+1;						
5 node_5	time_out != 0;						
5 node_5	time_out = time_out-1;						
5 node_5	time_out == 0;						
5 node_5	term = term+1;						
5 node_5	vote = vote+1;						
5 node_5	RPC[8]! candidate, N5, 3, 1;			[candidate, N5, 3, 1]			
3 node_3	RPC[8]? candidate, N5, 3, 1;						
3 node_3	REPLY[8]! follower, N3, 1;						[follower, N3, 1]
5 node_5	REPLY[8]? follower, N3, 1;						
5 node_5	vote = vote+1;						

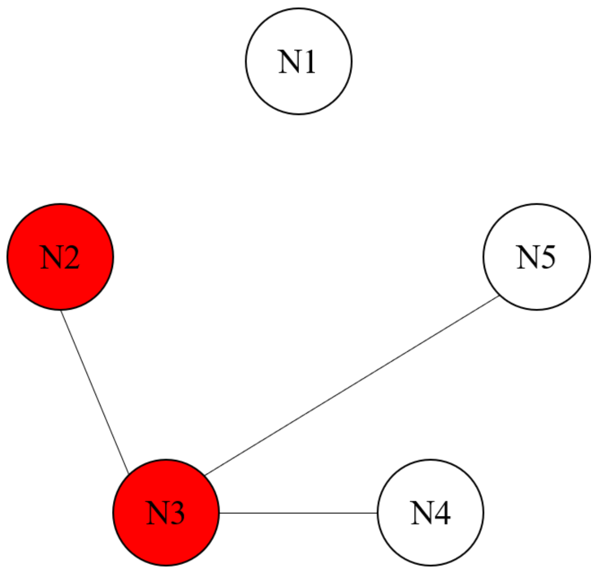
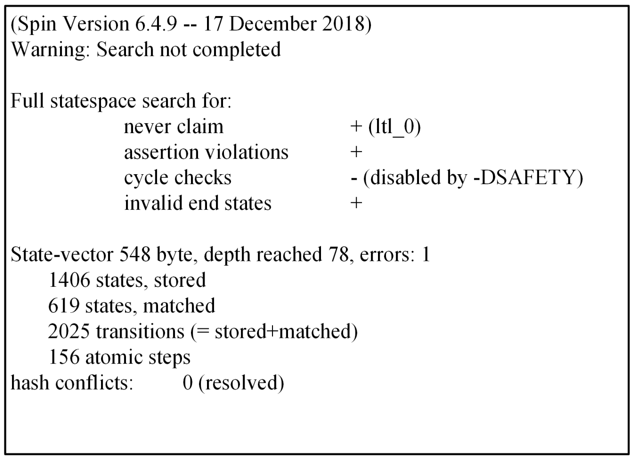


Figure 5. Fault settings when the system violates Property 2



```
(Spin Version 6.4.9 -- 17 December 2018)
Warning: Search not completed

Full statespace search for:
        never claim             + (ltl_0)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 548 byte, depth reached 78, errors: 1
    1406 states, stored
    619 states, matched
    2025 transitions (= stored+matched)
    156 atomic steps
hash conflicts:      0 (resolved)
```

Figure 6. Verification result for Property 2 when the fault is set as in Figure 5

current cluster will be elected as the leader, violating property 2.

D. Verification Result for Property 3

By setting the above three fault types, the Spin tool gives no error results when verifying Property 3. This indicates that in the case of any node faults, there will be no more than one leader node in each term in the Raft leader election algorithm, which is also consistent with the conclusion of the other two papers [4], [5].

E. Discussion

Based on the above verification results, we find that even if only two nodes have network anomalies, the Raft leader election algorithm will not hold stability, that is, the leader node will constantly switch between the two nodes with network anomalies. Through further analysis, it is found that once this type of fault exists in the cluster, the frequent switching of the leader node will continue unless the other node times out before the faulty node and starts the election. Alternatively, the leader node can complete the log entry update before the election is initiated by other candidate nodes. Then the other follower nodes will not vote for the new candidate node because the log entry on the candidate node is less up-to-date than that on other follower nodes.

In addition, when the nodes in the cluster are faulty and the log entry updates on the nodes are inconsistent due to the faults, the Raft leader election algorithm will not hold liveness, that is, the leader node cannot be elected over time. Through analysis, we find that there must be such a node in the system, it can maintain normal communication with more than half of the nodes in the cluster, and the log entry on this node is the most up-to-date.

In conclusion, to ensure the safety of the Raft leader election algorithm, we make the following suggestions.

1) In case of network communication faults in the cluster, there must be a node to maintain normal communication with all nodes in the cluster, to ensure the stable operation of the system.
2) In order to ensure that the cluster can elect a leader, there must be a node in the cluster with the most up-to-date logs and normal connection with most nodes, to ensure the normal operation of the system.

Unfortunately, in a distributed system, each node can only observe information relevant to itself, that is, each node cannot determine the network condition inside the other nodes. In the blockchain system, because of the decentralized setting, no one can know the health of all the nodes in the current system. Therefore, when node faults affect the safety of the Raft consensus algorithm, we can only hope that the nodes can recover from the faults as early as possible.

VI. Related Work

In this section, we review the works on model checking the distributed consensus algorithms for safety, as well as the safety and security research of the Raft consensus algorithm.

A. Model Checking the Distributed Consensus Algorithms for Safety

In order to solve the problem that the large state space makes model checking infeasible, Tsuchiya and Schiper reduced the verification problem to a small model checking problem involving only a single stage of the algorithm execution, thus achieving consistency and termination verification of the Last-voting algorithm [9]. Noguchi restricted the use of model checking to a single round of the algorithm, solving the problem of infinite state space by using a finite-state model that is very close to the behavior of any single round. In case studies, they verified the Mostefaoui-Raynal algorithm and the Chandra-Toueg algorithm using this approach [10]. Based on the above research, we simplify the Raft leader election algorithm by simulating the whole process of the algorithm as a voting process between nodes to avoid the problem of state space explosion.

B. Safety Research of the Raft Consensus Algorithm

In order to improve the understandability of Paxos, Ongaro and Ousterhout proposed Raft consensus algorithm and analyzed the safety of the algorithm [4]. They also mechanically proved Log Completeness Property with the TLA proof system. However, this proof relies on invariants that have not been mechanically checked. Woos et al. proposed the first formal verification for state machine safety of the Raft consensus algorithm [5]. However, they did not conduct a detailed analysis of node fault types, nor did they study the stability and liveness of the Raft leader election algorithm. Yu et al. increased the leadership transfer function to the Raft consensus algorithm to prevent cluster unavailability due to leader shutdown or removal, and verified its correctness using TLC Model Checker [11]. However, in terms of the Raft consensus algorithm, they only analyzed that the re-election of the leader may lead to cluster unavailability, but do not prove it from a formal perspective.

C. Security Research of the Raft Consensus Algorithm

In addition, many researchers have modified the Raft consensus algorithm to improve its safety and security. Hammer et al. found that bursty DDoS attacks and intermittent overload in network demands can trigger confusion when the Raft consensus algorithm is implemented in SDN controllers [12]. Then they proposed BabbleResistantRaft, an algorithm that can keep them safe, active, and stable in the face of these types of attacks and network conditions. Zhou and Ying proposed an improved Byzantine fault-tolerant algorithm based on the Raft consensus algorithm, which can resist the threat of Byzantine nodes by introducing the concept of node trust values [13]. Wang et al. proposed an improved Raft consensus algorithm called "hhRaft" to cope with the high real-time and high adversarial blockchain environment [14]. The algorithm optimizes the Raft consensus process by introducing a new Monitor role to improve Byzantine fault resistance. These studies improved the mechanisms of the Raft consensus algorithm to improve its ability to resist attacks. However, all these studies evaluated the security of the Raft consensus algorithm from a quantitative perspective, and did not prove it through formal methods, resulting in a lack of credibility in the results.

VII. Conclusion

Throughout the modeling and verification process, we formally verify the *stability*, *liveness*, and *uniqueness* of the Raft leader election algorithm using Promela language in the Spin tool. With the verification results, we find that the Raft leader election algorithm does not hold *stability* and *liveness* when some nodes are faulty and node log entries are inconsistent. Through this study, we establish and firmly believe that progress can be made using formal methods in the distributed domain.

In real distributed system, node failure often occurs at any time and suddenly. A node may be unable to send messages due to a network anomaly after receiving messages from other nodes. Or a node cannot receive the messages from other nodes after sending the messages. In the future, we can further optimize the model of the Raft leader election algorithm so that the model can dynamically simulate node faults. At the same time, because the Raft consensus algorithm has many variants, including the Tangaroa algorithm [15] and the ScalableBFT algorithm [16] that can tolerate Byzantine nodes, we will further study these algorithm variants, and make an overall modeling and verification of the whole Raft consensus algorithm family, to enhance the confidence of the algorithms in practical application.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[2] Q. Xia, W. Dou, K. Guo, G. Liang, C. Zuo, and F. Zhang, "Survey on blockchain consensus protocol," *Journal of Software*, vol. 32, no. 2, pp. 277–299, 2021.

[3] L. Leslie, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

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

[5] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson, "Planning for change in a formal verification of the raft consensus protocol," in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pp. 154–165, ACM, 2016.

[6] O. Hasan and S. Tahar, "Formal verification methods," in *Encyclopedia of Information Science and Technology, Third Edition*, pp. 7162–7170, IGI Global, 2015.

[7] W. Zhenzhen, "Survey of model checking," *Computer Science*, vol. 40, no. Z6, pp. 1–14, 2013.

[8] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[9] T. Tsuchiya and A. Schiper, "Verification of consensus algorithms using satisfiability solving," *Distributed Computing*, vol. 23, no. 5, pp. 341–358, 2011.

[10] T. Noguchi, T. Tsuchiya, and T. Kikuno, "Safety verification of asynchronous consensus algorithms with model checking," in *2012 IEEE 18th Pacific Rim International Symposium on Dependable Computing*, pp. 80–88, IEEE, 2012.

[11] G. Yu, L. Hua, L. Yuanping, L. Bowei, W. Xianrong, and R. Hongwei, "Using tla+ to specify leader election of raft algorithm with consideration of leadership transfer in multiple controllers," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 219–226, IEEE, 2019.

[12] R. Hanmer, S. Liu, L. Jagadeesan, and M. R. Rahman, "Death by babble: Security and fault tolerance of distributed consensus in high-availability softwarized networks," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, pp. 266–270, IEEE, 2019.

[13] S. Zhou and B. Ying, "Vg-raft: An improved byzantine fault tolerant algorithm based on raft algorithm," in *2021 IEEE 21st International Conference on Communication Technology (ICCT)*, pp. 882–886, IEEE, 2021.

[14] Y. Wang, S. Li, L. Xu, and L. Xu, "Improved raft consensus algorithm in high real-time and highly adversarial environment," in *International Conference on Web Information Systems and Applications*, pp. 718–726, Springer, 2021.

[15] C. Copeland and H. Zhong, "Tangaroa: a byzantine fault tolerant raft." https://www.scs.stanford.edu/14auGcs244b/labs/projects/copeland_zhong.pdf, 2016.

[16] W. Martino, "The first scalable, high performance private blockchain." http://kadena.io/docs/Kadena-ConsensusWhitePaper-Aug2016.pdf, 2016.