Formal Specification and Model Checking of Raft Log Replication in Maude

Takanori Ishibashi
School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
Ishikawa, Japan
takanori.ishibashi@jaist.ac.jp

Kazuhiro Ogata
School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
Ishikawa, Japan
ogata@jaist.ac.jp

***Abstract*—Raft is a popular distributed consensus protocol and is used to build highly available and strongly consistent services in the industry. Using Maude, we formally specify the log replication in Raft and conduct model checking to check whether the protocol enjoys the Log Matching Property and the State Machine Safety Property. The Log Matching Property is 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. The State Machine Safety Property is that if any two servers have applied two entries to their state machines at a same index, the two entries must be always the same. Our model checking experiments show that the protocol enjoys the properties under the condition that we limit the length of the server's log and the number of servers.**

***Index Terms*—state machines, invariant properties, rewriting logic, search command**

I. Introduction

Distributed consensus protocols, such as Raft [1], play a critical role in modern computing systems. These systems require high availability and strong consistency. The correctness of the systems is highly dependent on the correctness of Raft, and therefore, Raft provides significant functionality, and a bug in Raft can have tremendous effects; however, it is difficult to implement correctly distributed consensus protocols and to test their correctness. A proof assistant, such as Coq [2], allows us to prove that Raft enjoys the desired properties, but the time required for the investigation would probably be very long.

In this paper, we concentrate on the log replication, which is one of the basic mechanisms in Raft. We formally specify the log replication in Raft using Maude, which is a rewriting logic-based specification/programing language. We model check with Maude that Raft enjoys the Log Matching Property and the State Machine Safety Property, which are the properties that Raft is expected to guarantee. The Log Matching Property is 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. The State Machine Safety Property is that if any two servers have applied two entries to their state machines at a same index, the two entries must be always the same. Our model checking experiment shows that Raft enjoys the two properties. The formal specification of the Raft log replication in Maude is available online[1].

The contribution of the work described in the present paper is to demonstrate how the log replication in Raft is formally specified in Maude and model checked with the Maude and to show that Raft enjoys the two properties under the condition that the length of the server's log and the number of servers are limited. We assume that a server in a Raft cluster conducts unexpected operations, which is different from the log replication in Raft. A server failure can result not only in a simple shutdown, but also in incorrect behavior. It is preferable to be able to handle the latter as well. Our model checking experiments also show that servers except for a server that conducts unexpected operations enjoy the two properties. To our knowledge, no study has examined this.

Diego Ongaro's original description of Raft [3] showed a formal specification of Raft in TLA+ [4] but did not show how to model check invariants of Raft with the TLA model checker. The description reported that using the TLA model checker on the specification was attempted but was abandoned because this approach did not scale well to larger models. The formal verification of the safety properties of the Raft was conducted using the Verdi framework for distributed systems verification [5]. This formal verification in Verdi proves some invariants, and consists of approximately 50,000 lines of Coq [2].

II. Preliminaries

Raft [1] is a distributed consensus protocol. Raft divides a distributed consensus problem into two independent sub-problems: leader election and log replication[2]. In leader election, Raft chooses at most one leader in each logical time called a term. There is one and only one leader in a Raft cluster in regular operations and all the other servers are then followers. In log replication, the leader accepts requests from clients, saves such requests in its log, and forwards them

This work was supported in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant JP24H03370.

DOI reference number: 10.18293/DMSVIVA23-010

[1]https://github.com/11Takanori/raft-maude

[2]Diego Ongaro et al. discussed that Raft divides a distributed consensus problem into three independent sub-problems: leader election, log replication, and safety [1]. In this paper, we consider that Raft divides the problem into two independent sub-problems: leader election and log replication, and safety is a requirement to be satisfied by the two subproblems.

to all the other servers. On receipt of such requests, each server saves them in its log. When the leader receives positive replies for a client request from the majority of servers, it commits (or consents to) the request. Each server has a state machine in which clients' requests are processed. When a follower receives a message saying that a client request has been committed, the follower commits the clients' request up to the client request (inclusive).

We describe the log replication in a bit more detail. The leader serves the client's request and appends it to its log. To replicate log entries, the leader sends appendEntries request messages to each of the other servers in the Raft cluster. Each server that has received the appendEntries request message responds to the message by sending a appendEntries response message to the leader. The appendEntries request message includes the following:

- term - the leader's term
- leaderId - the leader's ID
- prevLogIndex - the index of the log entry immediately preceding new ones
- prevLogTerm - the term of the log entry immediately preceding new ones
- entries - the log which contains leader's term and the client request messages
- leaderCommit - the leader's commit index

The appendEntries response message includes the following:

- term - the receiver's term
- success - a boolean value that means whether the receiver appends the new log entry to its log or not

If the follower that receives appendEntries request message finds an entry in its log with the same *prevLogIndex* and *prevLogTerm*, and the follower's term is not bigger than the leader's term, then the follower appends the new log entry to its log; otherwise, the follower refuses the new log entry. When the log entry has been replicated on a majority of the servers in the Raft cluster, the leader applies the log entry to its state machine (called commit). If the leader's commitIndex is greater than the commitIndex of the follower, the leader's commitIndex and the follower's index of the last new entry are compared and the follower commits at a smaller one. If the followers' log is inconsistent with the leader's log, the leader decrements nextIndex and retries the appendEntries request message. The leader maintains a nextIndex for each follower, which is the index of the next log entry the leader will send to that follower. In log replication, Raft is expected to guarantee the Log Matching Property and the State Machine Safety Property. These properties are discussed in the introduction. Raft is expected to guarantee that each of all properties is true under all non-Byzantine conditions, including network delays, duplication, partitions, message loss, and reordering.

A state transition system[3] is $\langle S, I, T \rangle$, where S is a set of states, $I \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is a binary relation over states. Each element $(s, s') \subseteq T$ is called a state transition from *s* to *s'* and *T* is called the state transitions. There are multiple possible ways to express states. In this paper, we express a state as a braced associative-commutative collection of name-value pairs, where a name may have parameters. Associative-commutative collections are called soups according to the nomenclature of the Maude community, and name-value pairs are called observable components. That is, a state is expressed as a braced soup of observable components. We use the juxtaposition operator as the constructor of soups. Let oc_0, oc_1, oc_2 be observable components, and then $oc_0\, oc_1\, oc_2$ is the soup of those three observable components. A state is expressed as $\{oc_0\, oc_1\, oc_2\}$. *T* is specified in terms of rewrite rules. A rewrite rule starts with the keyword *rl*, followed by a label enclosed with square brackets and a colon, two pattern connected with $=>$, and ends with a period. A conditional rewrite rule starts with the keyword *crl* and have a condition following the keyword if before a period. The following is a form of a conditional rewrite rule: *crl [lb] : l => r if* ... $/\backslash$ c_i $/\backslash$ ... where lb is a label given to the rule and c_i is part of the condition, which may be an equation $lc_i = rc_i$. The negation of $lc_i = rc_i$ could be written as *(lci =/= rci) = true*, where = *true* could be omitted. If the condition ... $/\backslash$ c_i $/\backslash$ ... holds under some substitution σ, $\sigma(l)$ can be replaced with $\sigma(r)$.

[3]It may be called a state machine but because what are called state machines are used by Raft, we use the terminology "state transition system" in this paper.

III. Formal specification of the log replication

To formalize the log replication in Raft as a state transition system, we use the following observable components.

- *(term[s]: t)* - *s* is a server ID. *t* is a term. This means that the term of a server *s* is *t*. For each server s participating in a Raft cluster, an instance of this observable component is used.
- *(role[s]: r)* - *s* is a server ID. *r* is a role: leader or follower. This means that the role of a server *s* is *r*. For each server participating in a Raft cluster, an instance of this observable component is used.
- *(log[s]: l)* - *s* is a server ID. *l* is a list of log entries. This means that a server *s* has a log *l*. A server *s* appends log entries to the log *l*. For each server a participating in Raft cluster, an instance of this observable component is used.
- *(commitIndex[s]: ci)* - *s* is a server ID. *ci* is an index of highest log entry known to be committed. This means that index *ci* is the highest index at which a server s has committed log entries. For each server s participating in a Raft cluster, an instance of this observable component is used.
- *(nextIndex[s0][s1]: ni)* - *s0* and *s1* are server IDs. *ni* is an index of the next log entry that has been sent to s1 by s0. This means that a server *s0* will next send a server *s1* a log entry at nextIndex *ni*. A leader maintains a nextIndex for each follower, which is the index of the next log entry the leader will send to the follower. For each follower of a leader, an instance of this observable component is used.
- *(matchIndex[s0][s1]: mi)* - *s0* and *s1* are server IDs. *mi* is an index of the highest log entry such that s0 knows

that s1 has its replication of the log entry. This means that index *mi* is the highest index at which server *s0* has sent server *s1* a log entry. For each follower of a leader, an instance of this observable component is used.
- *(servers: ss)* - *ss* is a soup of server IDs. This maintains the IDs of all servers participating in a Raft cluster. One instance is only used and *ss* never changes.
- *(clientRequests: c)* - *c* is the messages that a client sends to a Raft cluster. One instance is only used. When a client sends a Raft cluster a message, the message is deleted from clientRequests.
- *(network: n)* - *n* is a soup of messages. This expresses the network with which the servers participating in a Raft cluster exchange messages. One instance is only used. A message that has been put into *n* is never deleted, which expresses that a message may be duplicated. Although a message is never deleted from the network, it would be possible for a server to never receive a message addressed to the server, which expresses that a message may be lost.

When there are three servers *s0*, *s1* and *s2* that participate in a Raft cluster, the initial state defined *init* is as follows:

```
{(term[s0]: 1) (term[s1]: 1) (term[s2]: 1)
 (role[s0]: leader) (role[s1]: follower)
 (role[s2]: follower)
 (log[s0]: empty) (log[s1]: empty) (log[s2]: empty)
 (commitIndex[s0]: 0) (commitIndex[s1]: 0)
 (commitIndex[s2]: 0)
 (nextIndex[s0][s1]: 1) (nextIndex[s0][s2]: 1)
 (matchIndex[s0][s1]: 0) (matchIndex[s0][s2]: 0)
 (servers: (s0 s1 s2)) (clientRequests: (cr0 cr1))
 (network: empty)} .
```

In the initial state, each value is as follows:
- the term of each sever is 1
- the role of the server *s0* is a leader
- the role of the server *s1* and the server *s2* is a follower
- the list of log entries that has been appended by each server is empty (meaning that each server has not yet appended any log entries to its log)
- the index that each server has committed is 0 (meaning that each server has not yet committed any log entries)
- the index of the next log entry that the server *s0* will send to the server *s1* and the server *s2* is 1
- the highest index at which the server *s0* has sent the server *s1* and the server *s2* a log entry is 0 (meaning that server *s0* has note yet known to be replicated on server *s1* and server *s2*)
- the soup of the servers that participate in a Raft cluster is *s0 s1 s2* because we suppose that the three servers participate in the Raft cluster
- the soup of the client request messages is *cr0 cr1* because we suppose that the client sends two messages to the Raft cluster
- the network is empty (meaning that no message has been put into the network)

The log replication in Raft is specified as three rewrite rules for each server: appendEntries, handleAppendEntriesRequest, and handleAppendEntriesResponse. The rewrite rules use the following Maude variables:

- *OCs* is a variable of observable components soups
- *S0*, *S1*, and *S2* are variables of server IDs
- *Ss* is a variable of server ID soups
- *T*, *U*, and *PLT* are variables of terms
- *R* and *R0* are variables of server roles
- *CI* and *LCI* are variables of commitIndex
- *MI*, *MI1*, and *MI2* are variables of matchIndex
- *NI*, *NI1*, and *NI2* are variables of nextIndex
- *PLI* is a variable of index of log entry immediately preceding new ones
- *PLT* is a variable of term of log entry immediately preceding new ones
- *Ls* and *L* are variables of log entries soups
- *CR* is a variable of client request messages
- *CRs* is a variable of client request message soups
- *NW* is a variable of message soups
- *AEReq* is a variable of AppendEntries request messages

The rewrite rule *appendEntries* is defined as follows:

```
rl [appendEntries] :
{(term[S0]: T) (role[S0]: leader)
 (commitIndex[S0]: CI) (log[S0]: Ls)
 (servers: Ss) (clientRequests: (CR CRs))
 (network: NW) OCs} =>
{(term[S0]: T) (role[S0]: leader)
 (commitIndex[S0]: CI)
 (log[S0]: Ls[length(Ls) + 1] := log(T, CR))
 (servers: Ss) (clientRequests: CRs)
 (network: (NW mkAppendEntriesRequests(
   S0, appendEntriesRequest(T, S0,
   length(Ls), term(Ls[length(Ls)]),
   log(T, CR), CI), Ss - S0))) OCs} .
```

The rewrite rule says that when a server *S0* is a leader and there exists a client request *CR*, *S0* puts an appendEntries request message in the network addressed to all the other servers. *appendEntriesRequest(T, S0, length(Ls), term(Ls[length(Ls)]), log(T, CR), CI)* is the body of the appendEntries request message. The first argument is *S0*'s term. The second argument is the leader's server ID. The third argument is the index of the log entry immediately preceding new ones (called a prevLogIndex). The fourth argument is the term of the log entry immediately preceding new ones (called a prevLogTerm). The fifth argument is the log which contains *S0*'s term and the client request messages. The sixth argument is *S0*'s commit index. $Ss - S0$ is the soup of server IDs obtained by deleting the server ID *S0* from the soup *Ss* of server IDs. *mkAppendEntriesRequests(S0, AEReq, Ss')* makes the appendEntries request message whose body is *AEReq* and that is addressed to all server IDs in *Ss'*.

The rewrite rule *handleAppendEntriesRequest* is defined as follows:

```
crl [handleAppendEntriesRequest] :
{(term[S0]: T) (role[S0]: R)
 (commitIndex[S0]: CI) (log[S0]: Ls)
 (network: (msg(S1, S0,
   appendEntriesRequest(U, S1, PLI, PLT,
    log(U, CR), LCI)) NW)) OCs} =>
{(term[S0]: if U > T then U else T fi)
 (role[S0]: if U >= T then follower else R fi)
 (commitIndex[S0]:
  if LCI > CI then min(LCI, length(L))
              else CI fi) (log[S0]: L)
 (network: (msg(S0, S1, appendEntriesResponse(
  (if U > T then U else T fi), B,
```

```
   appendEntriesRequest(U, S1, PLI, PLT,
    log(U, CR), LCI)))
  msg(S1, S0, appendEntriesRequest(
      U, S1, PLI, PLT, log(U, CR), LCI)) NW)) OCs}
if length(Ls) < 3
/\ B := U >= T and ((Ls[PLI] =/= null
        and term(Ls[PLI]) == PLT) or (PLI == 0))
/\ L := if B and Ls[PLI + 1] == null
        then Ls[PLI + 1] := log(U, CR)
        else (if (CI < PLI)
              and (length(Ls) =/= PLI
              or term(Ls[PLI]) =/= PLT)
              then Ls[: PLI] else Ls fi) fi .
```

When there exists *msg(S1, S0, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI))* in the network and the length of the server *S0*'s log is less than 3, the following are conducted. If the server *S0*'s current term *T* is less than or equal to *U* ($U >= T$) and the server *S0* has an entry at prevLogIndex whose term matches prevLogTerm or the leader has not yet appended any log entries to its log ((Ls[PLI] =/= null and term(Ls[PLI]) == PLT) or (PLI == 0)), then *B* is true. If *B* is true and the server *S0* has not yet had a log entry at the index $PLI + 1$ (*B and Ls[PLI + 1] == null*), then the sever *S0* appends the log entry to the log. If the commitIndex of the server *S0* is less than prevLogIndex ($CI < PLI$) and an existing log entry of the server S0 conflicts with a new one (*length(Ls) =/= PLI or term(Ls[PLI]) =/= PLT*), then *S0* deletes the existing entry and all that follow it. If the commitIndex of the leader is greater than the commitIndex of the server *S0*, then the commitIndex of the leader and the index of last new entry are compared and the commitIndex of the server *S0* is updated by the smaller one. If $U > T$, then the server *S0*'s current term becomes *U* and *msg(S0, S1, appendEntriesResponse(U, B, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI))* is put into the network; otherwise the server *S0*'s current term remains the same and *msg(S0, S1, appendEntriesResponse(T, B, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI))* is put into the network. If $U >= T$, then the server S0 becomes a follower. Note that *msg(S1, S0, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI)* is not deleted from the network. This is because a message may be duplicated.

The rewrite rule *handleAppendEntriesResponse* is defined as follows:

```
crl [handleAppendEntriesResponse] :
{(term[S0]: T) (role[S0]: R)
 (commitIndex[S0]: CI)
 (nextIndex[S0][S1]: NI1) (matchIndex[S0][S1]: MI1)
 (matchIndex[S0][S2]: MI2) (log[S0]: Ls)
 (network: (msg(S1, S0,
  appendEntriesResponse(U, B, AEReq)) NW)) OCs} =>
{(term[S0]: if U > T then U else T fi) (role[S0]: R0)
 (commitIndex[S0]:
  if replicatedCount(N, (MI MI2)) >= majority
  and T == term(Ls[N]) and R0 == leader
  and N > CI then N else CI fi)
 (nextIndex[S0][S1]: NI) (matchIndex[S0][S1]: MI)
 (matchIndex[S0][S2]: MI2) (log[S0]: Ls)
 (network: if (not B and U <= T and R0 == leader)
  then (msg(S0, S1, appendEntriesRequest(
          T, S0, PI, term(Ls[PI]), Ls[NI],
          leaderCommit(AEReq)))
        msg(S1, S0, appendEntriesResponse(
          U, B, AEReq)) NW)
  else (msg(S1, S0, appendEntriesResponse(
          U, B, AEReq)) NW) fi) OCs}
if MI := if B then max(prevLogIndex(AEReq) + 1, MI1)
         else MI1 fi
/\ NI := if B then MI + 1 else max(sd(NI1, 1), 1) fi
/\ PI := sd(NI, 1) /\ N := prevLogIndex(AEReq) + 1
/\ R0 := if U > T then follower else R fi .
```

When there exists *msg(S1, S0, appendEntriesResponse(U, B, AEReq))* in the network, the following are conducted. If *B* that is included in *appendEntriesResponse* is true, *S0*'s matchIndex for *S1* is updated with the greater of *prevLogIndex(AEReq) + 1* and *MI1*, where *prevLogIndex(AEReq) + 1* is the prevLogIndex (included in its appendEntriesRequest) plus 1 and *MI1* is server *S0*'s matchIndex for server *S1*, and server *S0*'s nextIndex for server S1 is updated with server *S0*'s matchIndex for server *S1* plus 1; otherwise, *S0*'s nextIndex for *S1* is updated with the greater of the *S0*'s nextIndex for *S1* minus 1 and 1 (the minimum value for nextIndex is 1). It is wrong to simply increment the matchIndex and the nextIndex because an *appendEntriesResponse* may be duplicated and the server *S0* may receive the appendEntriesResponse multiple times. Let *N* be the previous index plus 1. If the server *S0*'s matchIndex for server *S1* or the server *S0*'s matchIndex for server *S2* is greater than or equal to *N*, the term of server *S0*'s log at index *N* is server *S0*'s current term, the role of server *S0* is a leader and *N* is greater than server *S0*'s commitIndex, then server *S0*'s commitIndex is updated by *N*. If *B* is false, *U* is less than or equal to *T* and the role of server *S0* is a leader, then *msg(S0, S1, appendEntriesRequest(T, S0, PI, term(Ls[PI]), Ls[NI], leaderCommit(AEReq)))* is put into the network, meaning that server *S0* sends appendEntriesRequest message with the log entry at the previous nextIndex because of log inconsistency between server *S0* and server *S1*. If $U > T$, then the *S0*'s term becomes *U*, *S0* becomes a follower. Note that *msg(S1, S0, appendEntriesResponse(U, B, AEReq))* is not deleted from the network due to as mentioned beforehand.

IV. Model checking the log replication

The experimental environment used is SUSE Linux Enterprise Server 15 SP1 installed on a computer with a 2.8GHz 16 core processor and 1.5 TB memory. The computer is maintained by Research Center for Advanced Computing Infrastructure, JAIST. We are allowed to keep on using the computer up to one week in a row for each job, a model checking experiment for us. If a model checking experiment is not completed in one week, the job is killed. Under condition that the length of server's log is less than 3 and the number of servers is 3, we conducted some model checking experiments.

A. Model checking under assumptions that no server conducts unexpected operations

We first confirmed that logs are replicated correctly in our specification by using the following Maude command:

```
search [1] in RAFT : init =>*
{(role[S0:ServerID]: leader)
 (log[S0:ServerID]: L0:Logs)
 (log[S1:ServerID]: L1:Logs)
 (log[S2:ServerID]: L2:Logs)
 (commitIndex[S0]: 2)
 (commitIndex[S1]: 1) (commitIndex[S2]: 1)
 (matchIndex[S0][S1]: 2) (matchIndex[S0][S2]: 2)
```

```
 (nextIndex[S0][S1]: 3) (nextIndex[S0][S2]: 3) OCs}
such that length(L0:Logs) == 2
  and length(L1:Logs) == 2 and length(L2:Logs) == 2 .
```

Maude finds a solution for the search command. The solution says that there exists a path from the initial state leading to a state in which all servers have two log entries, the leader commits at index 2, the other servers commit at index 1, the leader's matchIndex for the other servers is 2, and the leader's nextIndex for the other servers is 3. This means that two log entries are replicated correctly. Note that the result dose not say that log entries will be eventually replaced correctly. There is a path along which two log entries have not been replicated.

We use Maude to check that Raft satisfies the Log Matching Property or not by using the following search command:

```
search [1] in RAFT : init =>*
{(log[S0:ServerID]: L0:Logs)
 (log[S1:ServerID]: L1:Logs)
 (matchIndex[S0][S1]: I:Nat) OCs}
such that L0:Logs[I:Nat] =/= null
 and L1:Logs[I:Nat] =/= null
 and (term(L0:Logs[I:Nat])
  == term(L1:Logs[I:Nat]))
 and ((term(L0:Logs[sd(I:Nat, 1)])
  =/= term(L1:Logs[sd(I:Nat, 1)]))
 or (value(L0:Logs[sd(I:Nat, 1)])
  =/= value(L1:Logs[sd(I:Nat, 1)]))) .
```

The result returned by Maude for the search command is as follows:

```
No solution. states: 2805  rewrites: 1049267 in 488ms
cpu (486ms real) (2150137 rewrites/second)
```

That is to say, Maude did not find any state in which two logs contain a log entry with the same index and term, and the logs contain a different log entry in all entries up through the index. Consequently, we can conclude that Raft enjoys the Log Matching Property under the condition that the length of server's log is less than 3 and the number of servers is 3.

We use Maude to check that Raft satisfies the State Machine Safety Property or not by using the following search command:

```
search [1] in RAFT : init =>*
{(log[S0:ServerID]: L0:Logs)
 (log[S1:ServerID]: L1:Logs)
 (commitIndex[S0]: I:Nat)
 (commitIndex[S1]: I:Nat) OCs}
such that ((term(L0:Logs[I:Nat])
  =/= term(L1:Logs[I:Nat]))
 or (value(L0:Logs[I:Nat])
  =/= value(L1:Logs[I:Nat]))).
```

The result returned by Maude for the search command is as follows:

```
No solution. states: 2805  rewrites: 993397 in 448ms
cpu (451ms real) (2217404 rewrites/second)
```

That is to say, Maude did not find any state in which a server has applied a log entry at a given index to its state machine, and other servers apply a different log entry for the same index. Consequently, we can conclude that Raft enjoys the State Machine Safety Property under the condition that the length of server's log is less than 3 and the number of servers is 3.

B. Model checking under assumptions that a server conducts unexpected operations

We assume that a follower conducts unexpected operations in a Raft cluster. We define *SERVER-ID* module in the previous subsection as follows:

```
fmod SERVER-ID is
 sort ServerID .
 ops s0 s1 s2 : -> ServerID [ctor] .
endfm
```

In this subsection, we modify *SERVER-ID* module as follows:

```
fmod SERVER-ID is
 sorts ServerID BadServerID .
 subsort BadServerID < ServerID .
 ops s0 s1 : -> ServerID [ctor] .
 op s2 : -> BadServerID [ctor] .
endfm
```

This means *ServerID* and *BadServerID* are sorts, sort *BadServerID* is a subsort of sort *ServerID*, *s0*, *s1* and *s2* are constants, *s0*'s sort and *s1*'s sort are ServerID, and *s2*'s sort is *BadServerID*.

We add the following rewrite rule.

```
crl [badHandleAppendEntriesRequest] :
{(term[S2:BadServerID]: T) (role[S2:BadServerID]: R)
 (commitIndex[S2:BadServerID]: CI)
 (log[S2:BadServerID]: Ls)
 (network: (msg(S1, S2:BadServerID,
   appendEntriesRequest(U, S1, PLI, PLT,
   log(U, CR), LCI)) NW)) OCs} =>
{(term[S2:BadServerID]: if U > T then U else T fi)
 (role[S2:BadServerID]:
   if U >= T then follower else R fi)
 (commitIndex[S2:BadServerID]: length(L))
 (log[S2:BadServerID]: L)
 (network: (msg(S2:BadServerID, S1,
   appendEntriesResponse(
    (if U > T then U else T fi), true,
    appendEntriesRequest(U, S1, PLI, PLT,
     log(U, CR), LCI)))
  msg(S1, S2:BadServerID, appendEntriesRequest(
    U, S1, PLI, PLT, log(U, CR), LCI)) NW)) OCs}
if length(Ls) < 3 /\ L := Ls[PLI + 1] := log(U, crb) .
```

When the sort of sever ID is *BadServerID*, there exists *msg(S1, S2:BadServerID, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI))* in the network and the length of the server *S0*'s log is less than 3, the following are conducted. The sever *S2* appends the log entry including client request *crb* and term *U* to the log. *crb* is the unexpected value and it is not the value that client sent. The commitIndex of server *S2* is updated by the length of server *S2*'s log. If $U > T$, then the server *S0*'s current term becomes *U* and *msg(S2:BadServerID, S1, appendEntriesResponse(U, true, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI))* is put into the network; otherwise the server *S0*'s current term remains the same and *msg(S2:BadServerID, S1, appendEntriesResponse(T, true, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI))* is put into the network. If $U >= T$, then the server *S0* becomes a follower. Note that *msg(S1, S2:BadServerID, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI)* is not deleted from the network due to as already mentioned.

This rewrite rule shows an unexpected operation. How to replicate a log entry and how to commit are different from the log replication in Raft. The server *s2* whose sort is

BadServerID conducts the rewrite rule *handleAppendEntries-Request* or the rewrite rule *badHandleAppendEntriesRequest*. This is because sort *BadServerID* is a subsort of sort *ServerID*. That is to say, the server *s2* conducts both normal operations and unexpected operations.

We confirmed again that logs are replicated correctly in our modified specification by using the following search command:

```
search [1] in RAFT : init =>*
{(role[S0:ServerID]: leader)
 (log[S0:ServerID]: L0:Logs)
 (log[S1:ServerID]: L1:Logs)
 (log[S2:ServerID]: L2:Logs)
 (commitIndex[S0]: 2)
 (commitIndex[S1]: 1) (commitIndex[S2]: 1)
 (matchIndex[S0][S1]: 2) (matchIndex[S0][S2]: 2)
 (nextIndex[S0][S1]: 3) (nextIndex[S0][S2]: 3) OCs}
such that length(L0:Logs) == 2
  and length(L1:Logs) == 2 and length(L2:Logs) == 2 .
```

Maude finds a solution for the search command. The solution says that there exists a path from the initial state leading to a state in which all servers have two log entries, the leader commits at index 2, the other servers commits at index 1, the leader's matchIndex for the other servers is 2, and the leader's nextIndex for the other servers is 3. This means that two log entries are replicated correctly. Note that the result dose not say that log entries will be eventually replaced. There is a path along which two log entries have not been replicated.

We use Maude to check that Raft satisfies the Log Matching Property for the server *s0* and the server *s1* or not by using the following Maude command:

```
search [1] in RAFT : init =>*
{(log[s0]: L0:Logs) (log[s1]: L1:Logs)
 (matchIndex[s0][s1]: I:Nat) OCs}
such that L0:Logs[I:Nat] =/= null
 and L1:Logs[I:Nat] =/= null
 and (term(L0:Logs[I:Nat])
  == term(L1:Logs[I:Nat]))
 and ((term(L0:Logs[sd(I:Nat, 1)])
  =/= term(L1:Logs[sd(I:Nat, 1)]))
 or (value(L0:Logs[sd(I:Nat, 1)])
  =/= value(L1:Logs[sd(I:Nat, 1)]))) .
```

The result returned by Maude for the search command is as follows:

```
No solution. states: 24987
rewrites: 8900588 in 6364ms cpu
(6389ms real) (1398583 rewrites/second)
```

That is to say, Maude did not find any state in which the server *s0* and the server *s1* have a log entry with the same index and term, and the server *s0*'s log and the server *s1*'s log contain a different log entry in all entries up through the index. Consequently, we can conclude that Raft enjoys the Log Matching Property in servers which do not conduct unexpected operations under the condition that the length of server's log is less than 3 and the number of servers is 3.

We use Maude to check that Raft satisfies the State Machine Safety Property for the server *s0* and the server *s1* or not by using the following search command:

```
search [1] in RAFT : init =>*
{(log[s0]: L0:Logs) (log[s1]: L1:Logs)
 (commitIndex[s0]: I:Nat)
 (commitIndex[s1]: I:Nat) OCs}
such that ((term(L0:Logs[I:Nat])
  =/= term(L1:Logs[I:Nat]))
 or (value(L0:Logs[I:Nat])
  =/= value(L1:Logs[I:Nat]))).
```

The result returned by Maude for the search command is as follows:

```
No solution. states: 24987
rewrites: 8380677 in 5412ms cpu
(5417ms real) (1548536 rewrites/second)
```

That is to say, Maude did not find any state in which the server *s0* or the server *s1* has applied a log entry at a given index to its state machine, and the opposite server apply a different log entry for the same index. Consequently, we can conclude that Raft enjoys the State Machine Safety Property in servers which do not conduct unexpected operations under the condition that the length of server's log is less than 3 and the number of servers is 3.

Under the condition that the length of the server's logs is less than 4 and the number of servers is 3 and the condition that the length of the server's logs is less than 3 and the number of servers is 4, we conducted model checking experiments to check whether Raft enjoys the Log Matching Property or not, and the State Machine Safety Property or not. However, the model checking experiments took over a week to complete and the job running model checking was killed.

V. Conclusion

We reported that the log replication in Raft is formally specified in Maude and model checking experiments are conducted based on the formal specification. Our model checking experiments have said that logs are replicated correctly and Raft enjoys the Log Matching Property 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 and the State Machine Safety Property that if any two servers have applied two entries to their state machines at a same index, the two entries must be always the same. Under assumptions that a server conducts unexpected operations, which are different from the log replication in Raft, our model checking experiments have said that Raft enjoys the Log Matching Property and the State Machine Safety Property in servers that do not conduct unexpected operations.

References

[1] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC'14). USENIX Association, USA, 305-320. https://dl.acm.org/doi/10.5555/2643634.2643666

[2] Yves Bertot and Pierre Castéran. 2013. Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media.

[3] Diego Ongaro. 2014. Consensus: Bridging Theory and Practice. Ph.D. Dissertation. Stanford University.

[4] Lamport, L. Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2002.

[5] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. 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 (CPP 2016). Association for Computing Machinery, New York, NY, USA, 154-165. https://doi.org/10.1145/2854065.2854081