JAIST Repository
https://dspace.jaist.ac.jp/

Title	【課題研究報告書】分散合意プロトコル Raft の形式仕様とモデル検査
Author(s)	石橋, 孝則
Citation	
Issue Date	2023-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/18758
Rights	
Description	Supervisor: 緒方 和博, 先端科学技術研究科, 修士(情報科学)

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

課題研究報告書

分散合意プロトコル Raft の形式仕様とモデル検査

石橋 孝則

主指導教員 緒方 和博

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

令和 5 年 9 月

Abstract

In this master's research project report, we report on a case study in which Raft is formally specified in Maude and model checking experiments are conducted based on the formal specification. The model checking experiments say that Raft enjoys properties that it is expected to guarantee. Raft is widely known as one of the distributed consensus protocols and is used to build highly available and strongly consistent services. Raft divides a distributed consensus problem into two independent sub-problems: leader election and log replication. In the 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 the log replication, the leader accepts requests from clients, saves such requests in its log, and forwards them 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).

In this master's research project report, we concentrate on the leader election and the log replication, which are basic mechanisms in Raft. We formally specify the leader election and the log replication in Raft using Maude, which is a rewriting logic-based specification/programming language. In the leader election, we model check with Maude that Raft enjoys the Election Safety Property. In the log replication, we model check with Maude that Raft enjoys the Log Matching Property and the State Machine Safety Property. The Election Safety Property is that at most one leader can be elected in each logical time. 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 always be the same. The first property is expressed as an invariant property of the state transition system formalizing the leader election, and the last two properties are expressed as invariant properties of the state transition system formalizing the log replication. Maude is equipped with a linear temporal logic (LTL) model checker and a reachability analyzer (called the search command) as model checking facilities. The search command can be used as an invariant model checker. Because the three properties are invariant properties, we use the search command for the model checking experiments. In the leader election, our model checking experiments show that the protocol enjoys the Election Safety Property under the condition that we limit the logical time

and the number of servers. In the log replication, our model checking experiments show that the protocol enjoys the Log Matching Property and the State Machine Safety Property under the condition that we limit the length of the server's log and the number of servers.

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 the Log Matching Property and the State Machine Safety Property.

keyword: distributed consensus protocols, invariant properties, Maude, model checking, Raft, search command, state transition systems

目 次

第1章　はじめに

1.1　背景

分散合意形成プロトコルである Raft[1] は、高い可用性と強い一貫性を必要とする現代のコンピューティングインフラストラクチャにおいて重要な役割を果たしている。分散データベースである etcd[29] や CockroachDB[30]、YugabyteDB[31]、TiKV[32] は Raft を使用している。これらのデータベースの正しさは Raft の正しさに大きく依存している。Raft は重要な役割を果たしており、Raft のバグは非常に大きな影響をもたらすが、分散合意形成プロトコルを正確に実装し、その正しさをテストすることは難しい。Coq[3] などの証明支援システムを使うことで、Election Safety Property、 Log Matching PropertyState Machine Safety Property などの Raft が保証すべき性質が満たされていることを証明できる。しかし、証明に数ヶ月から数年の時間がかかる可能性がある。

1.2　目的

本稿では、Raft の基本的なメカニズムであるリーダーの選出とログの複製に焦点を当てて、Maude[2] を使ってリーダーの選出とログの複製の形式仕様を作成する。そして、その形式仕様に基づいて、Election Safety Property、Log Matching Property、State Machine Safety Property といった Raft が保証すべき性質が満たされるかどうかをモデル検査により検証する。Election Safety Property とは、各ターム（term）で最大 1 人のリーダーが選出されることである。ここでのタームとは論理時間を意味している。Log Matching Property とは、同じインデックスとタームのログエントリを含む 2 つのログがある場合、そのインデックスまでのすべてのログエントリが等しいことである。State Machine Safety Property とは、任意の 2 台のサーバーが同じインデックスで 2 つのログエントリを状態機械に適用した場合、常に 2 つのログエントリが同一であることである。

1.3　構成

本稿の構成は以下の通りである。

- 第 1 章 - 本稿の背景や目的を説明する。
- 第 2 章 - Raft や他の分散合意形成プロトコルの形式検証の関連研究について報告する。
- 第 3 章 - 本稿で必要となる用語や知識について説明する。Raft のリーダー選出とログの複製の機能やそれらで満たすべき性質、モデル検査、Maude や状態遷移システムについて説明する。
- 第 4 章 - Raft のリーダー選出に関する形式仕様とその形式仕様に基づいたモデル検査を説明する。
- 第 5 章 - Raft のログの複製に関する形式仕様とその形式仕様に基づいたモデル検査を説明する。
- 第 6 章 - 本研究の総括と今後の課題について報告する。

第2章　関連研究

この章では Raft など分散プロトコルの形式検証の先行研究をいくつか報告し、最後に本課題研究の位置づけについて記載する。

2.1　分散プロトコルの形式検証の先行研究

Raft を考案した Diego Ongaro の博士論文 [4] では、TLA+[6] での Raft の形式的仕様が示されていたが、その形式仕様に基づいてモデル検査する方法やモデル検査の結果は示されていなかった。TLA モデルチェッカーをその形式仕様に適用することが試みられたが、このアプローチは大規模なモデルに対してスケーラビリティが低いため、断念したという記述があった。分散動的再構成プロトコルである MongoRaftReconfig の安全性が TLA+で示されている [10]。MongoRaftReconfig とは Raft から派生したレプリケーションプロトコルを持つ分散データベースである MongoDB[33] 用に設計、実装されたものである。ここでの検証は 3198 行の TLA+ のコードから構成されている。

Doug Woos らは Verdi[9] を用いて Raft の状態機械の安全性を証明している [8]。Verdi とは定理証明支援系 Coq[3] で分散システムを形式検証するためのフレームワークである。この Verdi での形式検証では、Raft が期待される Election Safety Property、Log Matching Property、State Machine Safety Property などの不変性が証明されている。その証明は約 50,000 行の Coq のソースコードから構成されている。また、その証明に約 18 か月の期間が費やされた。IronFleet という TLA+[6] を改良したものとホーア論理 [28] を組み合わせた手法を用いて、分散システムの安全性と活性を示されている [17]。この研究では Paxos[13] ベースの状態機械を高レベルの仕様のレイヤと実装のレイヤに分割し、安全性と活性の検証が行われている。また、Adore というモデルを導入し、分散合意形成プロトコルの再構成の機能の安全性が示されている [11]。この研究ではネットワークレベルの通信を隠蔽しながら、状態機械にコミットした情報や選挙で使用する情報などの重要な依存関係や不変性を抽出することが行われている。Verdi や IronFleet、Adore は複雑なシステムに対する検証に成功しているが、大量の手作業が必要とされており、検証にコストがかかる。

Ivy[14] という演繹的検証ツールを用いて、Raft と MultiPaxos[13] に対する検証が行われている [12]。この研究では Raft と MultiPaxos の基本的な機能のみに着目

し、サーバーごとの状態機械に異なるログがコミットされないことを検証している。また、ここではDecidable Decompositionという手法によって、検証対象を決定可能な論理で証明された補題に分割してる。Ivyは状態が無限のシステムの安全性を対話的に検証するためのツールである。Ivyは検証が失敗した際に、帰納法に対する具体的な反例をグラフィカルに表示することができる。Ivyのユーザーはこの反例を確認しながら、一般化できる不変性を対話的に検証する。この手法でも手作業による検証作業が多く残る。

定理証明とは異なり、検証を自動化し、検証のコストを減らす研究もいくつかある。DistAI[19]は分散プロトコルの帰納的不変性を学習するためのデータ駆動型自動化システムである。DistAIは異なるインスタンスのサイズで分散プロトコルをシミュレーションし、サンプルとして状態を記録しておく。そのサンプルからより強い不変性を候補として列挙し、それらをSMTソルバーの入力として用いて、安全性を検証する。安全性を示すことができなかった場合、不変性を弱めて検証を再試行する。また、DuoAI[18]は、SMTクエリのコストを削減し、分散プロトコルを検証するための帰納的不変性を自動的かつ効率的に見つけるためのツールである。Paxosなど分散プロトコルに対してこのツールが用いられ、自動検証が行われている。また、分散プロトコルの安全性の自動検証を行うためのSWISSというシステムがある[20]。ここでは、分散システムの形式仕様と保証したい安全性をインプットとして、帰納的不変性の構築を自動化する方法が示されている。SWISSは安全性を証明するのに十分な帰納的不変性を見つけ、それを検証するために、候補の不変性の空間内で網羅的に探索する。Paxosを含むいくつもの分散プロトコルに対して、自動検証が行われている。Incremental Inference of Inductive Invariants（I4）という分散プロトコルの帰納的不変性を自動的に生成するアプローチがある[21]。このアプローチでは、分散プロトコルに対して、有界な状態のもとでの帰納的不変性を用いて、状態が無限な場合でも成り立つような一般的な帰納的不変性を推論している。検証したいプロトコルに対する有界のモデルを作成し、モデル検査を用いて、帰納的不変性を自動的に検証する。そして、そこで得られた不変性をもとに、状態が無限な場合でも成り立つような帰納的不変性に一般化している。I4を用いることで、いくつかの分散プロトコルの性質を証明する際に、手作業による検証作業を減らすことができている。また、IC3PO[26]という分散プロトコルの検証機によって、Paxosの仕様に対して帰納的不変性を自動的に推論することができている[27]。この研究ではプロトコルの構造的対称性の分析と量化を組み合わせた手法が提案されている。また、Sift[22]という自動証明と分割統治手法を組み合わせたアプローチがある。複雑な分散システムの証明をいくつかのステップに分解し、それぞれのステップの証明を自動化することを可能にしている。ここではRaftやPaxosなど分散プロトコルに対して、自動検証が行われている。DistAIやDuoAI、SWISS、I4、IC3PO、Siftは不変性の発見を自動化し、手作業による検証作業のコストを軽減している。

PlusCal[16]を拡張したModular PlusCalという言語からTLA+[6]とGo[34]に

変換する研究も行われている [15]。モデル検査のために TLA+を出力し、実行可能な実装として Go のコードを出力している。この論文ではこの仕組みを利用して、Raft ベースのキーバリューストアが構築されている。P#というアクターベースのプログラミング言語がある [23]。P#のプログラムは複数の状態機械から構成され、イベントを送受信することで状態機械が相互に通信する。状態機械には入力されるイベントを処理するためのアクションを登録することができ、イベントが到着すると、そのイベントに対して登録されたアクションが実行される。P#はこの状態機械を利用して、静的にデータ競合を分析することができる。分散システムの形式仕様を C++の実装に変換する Mace[24] というツールがある。Mace は Mace の形式仕様をモデル検査することができる。そのため、モデル検査で検証された形式仕様の基づく C++のコードを得ることができる。

本課題研究で形式仕様の記述とモデル検査のために使用している Maude[2] の形式仕様から、分散実装を生成する研究もある [25]。この研究では、Maude でモデル検査した安全性と活性を満たしたまま、Maude のコードを異なるコンピューター上で動作させることができる実装に変換させている。この変換は 2 種類の分散トランザクションシステムに適用されている。

2.2 本課題研究報告書の位置づけ

本課題研究報告書では、Raft クラスタ内のフォロワーが、Raft のログの複製とは異なる動作を行うケースでもモデル検査をしている。サーバーの障害は、単純なシャットダウンだけでなく、誤った動作にもつながる可能性があるので、Raft としては後者も扱えることが望ましいためである。モデル検査の実験では、異なる動作をするサーバー以外のサーバーが Log Matching Property と State Machine Safety Property の性質を満たすことを示している。この検証を行った先行研究はないと考えている。

また、本課題研究で作成した Raft のリーダーの選出の形式仕様は 200 行程度、Raft のログの複製の形式仕様は 250 行程度の規模である。リーダーの選出のモデル検査では最大のタームとサーバーの数、ログの複製のモデル検査ではサーバーのログの長さとサーバーの数を制限する必要があるが、より少ない労力で Election Safety Property、Log Matching Property、State Machine Safety Property といった Raft が保証すべき性質が満たされることを示すことができている。

第3章　予備知識

3.1　Raft

Raft[1] は分散合意プロトコルである。Raft システム（または Raft クラスタ）は、リーダー、フォロワー、または候補者として機能する複数のサーバーで構成される。Raft は、分散合意問題を分散合意の問題をリーダーの選出（leader election）とログの複製（log replication）の 2 つの独立した副問題に分割する[1]。リーダーの選出では、ターム（term）と呼ばれる論理時間ごとに最大 1 人のリーダーを選出する。通常の操作では、Raft クラスタには 1 人だけのリーダーがおり、他のすべてのサーバーはフォロワーである。ログの複製では、リーダーはクライアントからのリクエストを受け入れ、そのリクエストを自分のログに保存し、他のすべてのサーバーに転送する。各サーバーはそのようなリクエストを受け取ると、それらを自分のログに保存する。リーダーはサーバーの過半数からクライアントリクエストに対する肯定的な返答を受け取ると、そのリクエストを自身の状態機械に適用する（これをコミットと呼ぶ)。各サーバーはクライアントからのリクエストを保持する状態機械を持っている。フォロワーはメッセージを受け取ると、リーダーがコミットしたインデックスまでのログの情報をコミットする。リーダーは定期的に自身が正常に動作していることを示すためのメッセージを他のすべてのサーバーに送信する。これはハートビートメッセージと呼ばれる。フォロワーが特定の時間内にリーダーからのハートビートメッセージを受け取らない場合、そのフォロワーはリーダーが正常に動作していないと判断し、リーダーの選出を開始するために候補者になる。

3.1.1　リーダーの選出

リーダーの選出について説明する。Raft ではサーバーが起動すると、すべてのサーバーはフォロワーとして始まる。サーバーは、リーダーまたは候補者から有効なメッセージを受信している限り、フォロワーのままである。フォロワーが選挙タイムアウト（election timeout）と呼ばれる期間中にメッセージを受信しない場

[1]Diego Ongaro らは、Raft が分散合意問題をリーダーの選出、ログの複製、および安全性という 3 つの独立したサブ問題に分割することを論じている [1]。本稿では、Raft が問題をリーダーの選出とログの複製という 2 つの独立した副問題に分割し、安全性はこれら 2 つの副問題によって満たされるべき要求事項であると考える。

合、リーダーが故障している、もしくは他のサーバーと通信できなくなっていると想定し、新しいリーダーを選出するための選挙を開始する。フォロワーは選挙を開始するために、自身のタームを 1 つ増やし、候補者になる。その後、自分自身に投票し、Raft クラスタ内の他のすべてのサーバーに requestVote リクエストメッセージを送信する。requestVote リクエストメッセージを受信した各サーバーは、候補者に requestVote 応答メッセージを送信することでメッセージに応答する。requestVote リクエストメッセージには、候補者の ID とタームが含まれる。サーバーが候補者のログエントリが最新であることを確認するためには、他のパラメータも必要である。リーダーの選出の章ではログの複製には触れていないため、リーダーの選出の章では requestVote リクエストメッセージで他のパラメータを用いていない。requestVote 応答メッセージには、サーバーのタームと、サーバーが候補者に投票するかどうかを示す voteGranted が含まれる。各サーバーは、タームごとに最大 1 つの候補者に先着順で投票する。requestVote リクエストメッセージに含まれるタームが、サーバーが維持しているタームより古い場合、サーバーは候補者に投票しない。候補者は、同じタームでクラスタ全体のサーバーの過半数から投票を受け取ると、選挙に勝利し、リーダーになる。リーダーは、新しい選挙を開始しないように、すべてのフォロワーにログエントリのない定期的な appendEntries リクエストメッセージ（ハートビートメッセージと呼ばれる）を送信する。appendEntries リクエストメッセージには、リーダーのタームと ID が含まれる。ログの複製でリーダーが他のすべてのサーバーにログエントリを送信する場合、他のパラメータも必要であるが、リーダーの選出の章ではログの複製には触れていないため、リーダーの選出の章では appendEntries リクエストメッセージで他のパラメータを用いていない。リーダーの選出の章では、appendEntries リクエストをハートビートメッセージとしてのみを使用している。appendEntries 応答メッセージには、サーバーのタームと成功が含まれる。この成功は真偽値であり、appendEntries が正常に完了したかどうかを意味する。すべてのサーバーは、メッセージを交換することで、自分のタームが他のサーバーのタームより小さいかどうかを知る。自分のタームが他のサーバーのタームより小さい場合、フォロワーになり、他のサーバーのタームの値で自分のタームを更新する。

リーダーの選出では、タームごとに最大 1 人のリーダーしか選出されないという Election Safety Property が保証されることが期待されている。Election Safety Property はビザンチン故障 [7] を除き、メッセージの喪失、ネットワークの遅延、メッセージの複製、ネットワークの分断、メッセージの並び替えなどが発生しうる状況でも保証されることが期待されている。

3.1.2 ログの複製

ログの複製について説明する。リーダーはクライアントのリクエストを処理し、自分のログに追加する。リーダーはそのログエントリを複製するために、Raft クラ

スタ内の他のすべてのサーバーに appendEntries リクエストメッセージを送信する。appendEntries リクエストメッセージを受信した各サーバーは、リーダーに appendEntries 応答メッセージを送信することでメッセージに応答する。appendEntries リクエストメッセージには、以下の要素が含まれる:

- term - リーダーのターム
- leaderId - リーダーの ID
- prevLogIndex - 新しいログエントリの直前のログエントリのインデックス
- prevLogTerm - 新しいログエントリの直前のログエントリのターム
- entries - リーダーのタームとクライアントのリクエストメッセージが含まれるログ
- leaderCommit - リーダーがコミットしたインデックス

appendEntries 応答メッセージには、以下の要素が含まれる:

- term - 受信したサーバーのターム
- success - 受信者が新しいログエントリを自分のログに追加するかどうかを示す真偽値

appendEntries リクエストメッセージを受信したフォロワーが自身のログに prevLogIndex の値と prevLogTerm の値と等しいログエントリを見つけ、かつ、リーダーのタームが自身のタームがより古くない場合、フォロワーは新しいログエントリを自分のログに追加する。そうでない場合、フォロワーは新しいログエントリを拒否する。リーダーはログエントリが Raft クラスタ内の過半数のサーバーに複製されたことがわかると、そのログエントリを自身の状態機械に適用する（これをコミットと呼ぶ)。リーダーの commitIndex がフォロワーの commitIndex より大きい場合、フォロワーはリーダーの commitIndex と自身の最新のログエントリのインデックスを比較し、小さい方でコミットする。フォロワーのログがリーダーのログに不整合があり、フォロワーは新しいログエントリを拒否した場合、リーダーは nextIndex をデクリメントし、appendEntries リクエストメッセージを再送する。送信すべきログエントリがフォロワーごとに異なる場合があるため、リーダーはフォロワーごとに次に送信するログのインデックスである nextIndex を保持する。

ログの複製では、Log Matching Property と State Machine Safety Property をを保証されることが期待されている。Log Matching Property とは、同じインデックスとタームのログエントリを含む 2 つのログがある場合、そのインデックスまでのすべてのログエントリでログが一致していることである。State Machine Safety

Property とは、同じインデックスで 2 つのログエントリを状態機械に適用した任意の 2 台のサーバーが、常に 2 つのログエントリが同じであることである。Log Matching Property と State Machine Safety Property も Election Safety Property と同様にビザンチン故障 [7] を除き、メッセージの喪失、ネットワークの遅延、メッセージの複製、ネットワークの分断、メッセージの並び替えなどが発生しうる状況でも保証されることが期待されている。

3.2 モデル検査

モデル検査 [5] は、システムが特定の性質を満たしているかを自動的に検証する手法の 1 つである。モデル検査では検証対象のシステムを形式仕様として記述したものに対して、システムに期待する性質が満たされているかどうかを網羅的に調べることができる。モデル検査は、テストよりも徹底的な検証であるとみなすことができ、自動化されているという点では、定理証明よりも実用的な手法であると考えることができる。モデル検査の主な目的は、システム設計と開発の過程でエラーを見つけることである。モデル検査でエラーが見つからない場合、高い信頼性が得られる。

3.3 Maude と状態遷移システム

Maude は書換え論理に基づく仕様およびプログラミング言語である [2]。本稿では Maude を用いて、形式仕様の記述とモデル検査を行う。状態遷移システム[2]は $\langle S, I, T \rangle$ で表現される。S は状態の集合、$I \subseteq S$ は初期状態の集合、$T \subseteq S \times S$ は状態間の 2 項関係である。各要素 $(s, s') \subseteq T$ は、s から s' への状態遷移と呼ばれる。状態を表現するためには、複数の方法がある。本稿では、状態を結合法則と交換法則を満たした名前と値のペアのコレクションを中括弧で囲んだものとして表現している。名前にはパラメータが付けられる場合もある。Maude コミュニティの用語として、結合法則と交換法則を満たしたコレクションは、スープ（soup）と呼ばれ、名前と値のペアは観測可能成分（observable component）と呼ばれる。つまり、状態は観測可能成分のスープとして表現される。スープのコンストラクタとして、並置演算子を使用する。$oc_0\ oc_1\ oc_2$ を観測可能成分とし、$oc_0\ oc_1\ oc_2$ はそれらの 3 つの観測可能成分のスープである。状態は $\{oc_0\ oc_1\ oc_2\}$ として表現される。書き換え規則は、キーワード *rl* で始まり、角括弧で囲まれたラベルとコロンが続き、=> で 2 つのパターンを接続し、ピリオドで終える。条件付き書き換え規則は、キーワード *crl* で始まり、ピリオドの前にキーワード if に続く条件がある。条件付き書き換え規則は *crl [lb] : l => r if ... /\ c_i /\ ...* といった形式である。

[2]状態機械と表現されることが多いが、Raft の用語で状態機械という表現が用いられているため、本稿では状態遷移システムと表現する。

lb は規則に与えられたラベルであり、c_i は条件の一部であり、$lc_i = rc_i$ という等式が成り立つ場合がある。$lc_i = rc_i$ の否定は $(lci =/= rci) = true$ と書くことができる。また、$= true$ は省略することができる。条件$... \ /\backslash \ c_i \ /\backslash \ ...$ がある置換 σ の下で成立する場合、$\sigma(l)$ は $\sigma(r)$ で置き換えることができる。

第4章　リーダーの選出

本章ではRaftのリーダーの選出の形式仕様とモデル検査について説明している。モデル検査の結果、RaftがElection Safety Propertyの性質が満たしていることを示している。

4.1　リーダーの選出の形式仕様

Raftのリーダーの選出を状態遷移システムとして形式化するために、以下の観測可能成分を用いる。

- *(term[s]: t)* - *s* はサーバーIDであり、*t* はタームである。これは、サーバーsのタームがtであることを意味する。Raftクラスタに参加する各サーバー*s*に対して、この観測可能成分のインスタンスが使用される。

- *(role[s]: r)* - *s* はサーバーIDであり、rは役割である。役割にはリーダー、候補者、フォロワーがある。これは、サーバーsの役割がrであることを意味する。Raftクラスタに参加する各サーバーに対して、この観測可能成分のインスタンスが使用される。

- *(votedFor[s]: sv)* - *s* および *sv* はサーバーIDである。これは、サーバーsがサーバー*vs*に投票したことを意味する。Raftクラスタに参加する各サーバーに対して、この観測可能成分のインスタンスが使用される。

- *(votedBy[s]: ss)* - *s* はサーバーIDであり、*ss* はサーバーIDの集合である。これは、*s* が *ss* 内のサーバーによって投票されたことを意味する。Raftクラスタに参加する各サーバーに対して、この観測可能成分のインスタンスが使用される。

- *(servers: ss)* - *ss* はサーバーIDの集合である。これは、Raftクラスタに参加するすべてのサーバーのIDを維持する。インスタンスは1つだけ使用され、*ss* は変更されない。

- *(network: n)* - *n* はメッセージの集合である。これは、Raftクラスタに参加するサーバーがメッセージを交換するネットワークを表す。インスタンスは1つだけ使用される。一度 *n* に入れられたメッセージは削除されない。こ

れはメッセージが複製される可能性があることを表現している。メッセージはネットワークから削除されないが、サーバーが宛先のサーバーに送られたメッセージを受信しないこともある。これはメッセージが失われる可能性があることを表す。

Raft クラスタに参加するサーバーがサーバー *s0*、サーバー *s1*、サーバー *s2* の 3 台ある場合、定義された初期状態 *init* は以下のようになる。

```
{(term[s0]: 0) (term[s1]: 0) (term[s2]: 0)
 (role[s0]: follower) (role[s1]: follower) (role[s2]: follower)
 (votedBy[s0]: null) (votedBy[s1]: null) (votedBy[s2]: null)
 (votedFor[s0]: empty) (votedFor[s1]: empty) (votedFor[s2]: empty)
 (servers: (s0 s1 s2)) (network: empty)} .
```

初期状態では、観測可能成分は以下のようになる。

- 各サーバーのタームは 0 である
- 各サーバーの役割はフォロワーである
- 各サーバーが投票したサーバーは null である（各サーバーはまだどのサーバーにも投票していないことを意味する）
- 各サーバーに投票したサーバーの集合は空である（各サーバーはまだどのサーバーからも投票されていないことを意味する）
- Raft クラスタに参加するサーバーはサーバー ID が s0 s1 s2 のサーバーから構成されている（3 つのサーバーが Raft クラスタに参加すると仮定している）
- ネットワークは空である（ネットワークにメッセージが入っていないことを意味する）

Raft におけるリーダーの選出を形式仕様として表現するために、各サーバーに対して *timeout*、*requestVote*、*appendEntriesWithEmptyLog*、*handleRequestVoteRequest*、*handleRequestVoteResponse*、*handleAppendEntriesRequest*、*handleAppendEntries-Response* といった 7 つの書き換え規則を用いる。これらの書き換え規則では以下の Muade の変数を用いる。

- *OCs* - 観測可能成分の集合の変数
- *S0*、*S1* - サーバー ID の変数
- *Ss* - サーバー ID 集合の変数
- *T* - タームの変数

- *R* - サーバー役割の変数
- *VB* - サーバー ID の変数
- *VF0*、*VF1* - サーバー ID 集合の変数
- *NW* - メッセージ集合の変数
- *RVReq* - RquestVote メッセージの変数
- *AEReq* - AppendEntries メッセージの変数

書き換え規則 *timeout* を以下のように定義する。

```
crl [timeout] :
{(term[S0]: T) (role[S0]: R) (votedBy[S0]: VB)
 (votedFor[S0]: VF0) OCs} =>
{(term[S0]: T + 1) (role[S0]: candidate)
 (votedBy[S0]: S0) (votedFor[S0]: S0) OCs}
if (R == follower or R == candidate) and T < maxTerm .
```

ここでの *maxTerm* は到達可能な状態空間を制限するためのもで、最大のタームを表す自然数である。この書き換え規則は、サーバー *S0* がフォロワーまたは候補者であり、サーバー *S0* の現在のターム *T* が *maxTerm* より小さい場合、サーバー *S0* は候補者となり、ターム *T* がインクリメントされることを示している。実際の Raft クラスタでは、150ms などの特定の時間がタイムアウトとして使用されるが、本稿の形式仕様では特定の時間をタイムアウトとして使用せず、S0 に対していつでもタイムアウトが発生するようにする。この場合でも、リーダーの選出が Election Safety Property を満たすことに影響を与えない。

書き換え規則 *requestVote* を以下のように定義する。

```
crl [requestVote] :
{(term[S0]: T) (role[S0]: candidate) (servers: Ss)
 (network: NW) OCs} =>
{(term[S0]: T) (role[S0]: candidate) (servers: Ss)
(network: (NW mkRequestVoteRequests(
  S0, requestVoteRequest(T, S0), Ss - S0))) OCs} if T <= maxTerm .
```

この書き換え規則は、サーバー *S0* の現在のターム *T* が *maxTerm* 以下の場合、サーバー *S0* が他のすべてのサーバーに対して requestVote リクエストメッセージを送っていることを表している。*requestVoteRequest(T, S0)* は、ターム *T* で *S0* がリクエストした requestVote リクエストメッセージである。*Ss - S0* は、サーバー ID のスープ *Ss* からサーバー ID *S0* を削除して得られるサーバー ID のスープである。

mkRequestVoteRequests(S0, RVReq, Ss') は、requestVote リクエストメッセージの本文が *RVReq* であり、*Ss'* 内のすべてのサーバー ID 宛ての requestVote リクエストメッセージを作成することを表している。

mkRequestVoteRequests を以下のように定義する。

```
op mkRequestVoteRequests :
  ServerID RequestVoteRequest Soup{ServerID} -> Soup{Msg} .
eq mkRequestVoteRequests(S0, RVReq, empty) = empty .
eq mkRequestVoteRequests(S0, RVReq, S1 Ss) =
  msg(S0, S1, RVReq) mkRequestVoteRequests(S0, RVReq, Ss) .
```

msg(S0, S1, RVReq) は、メッセージの本文が *RVReq* であり、サーバー *S0* からサーバー *S1* へ送信されるメッセージである。*Ss* 内の各サーバー ID *S1* に対して、*mkRequestVoteRequests(S0, RVReq, Ss)* は *msg(S0, S1, RVReq)* を作成する。requestVote リクエストメッセージを作成する *mkRequestVoteRequests* に加えて、appendEntries リクエストメッセージを作成する *mkAppendEntriesRequest* も定義して用いている。

書き換え規則 *appendEntriesWithEmptyLog* を以下のように定義する。

```
crl [appendEntriesWithEmptyLog] :
{(term[S0]: T) (role[S0]: leader) (servers: Ss)
 (network: NW) OCs} =>
{(term[S0]: T) (role[S0]: leader) (servers: Ss)
(network: (NW mkAppendEntriesRequests(
  S0, appendEntriesRequest(T, S0), Ss - S0))) OCs}
if T <= maxTerm .
```

この書き換え規則は、サーバー *S0* の現在のターム *T* が *maxTerm* 以下の場合、サーバー *S0* が他のすべてのサーバーに対して appendEntriesRequest メッセージを送信することを表している。*appendEntriesRequest(T, S0)* は、ターム *T* で *S0* によってリクエストされた appendEntriesRequest メッセージの本文である。*mkAppendEntriesRequests(S0, AEReq, Ss')* は、メッセージの本文が *AEReq* であり、*Ss'* 内のすべてのサーバー ID に宛てられた appendEntriesRequest メッセージを作成する。

mkAppendEntriesRequests を以下のように定義する。

```
op mkAppendEntriesRequests :
  ServerID AppendEntriesRequest Soup{ServerID} -> Soup{Msg} .
eq mkAppendEntriesRequests(S0, AEReq, empty) = empty .
eq mkAppendEntriesRequests(S0, AEReq, S1 Ss) =
  msg(S0, S1, AEReq) mkAppendEntriesRequests(S0, AEReq, Ss) .
```

この章では、Raft のリーダーの選出にのみ焦点を当てており、appendEntriesRequest メッセージには実際のログエントリが含まれていない。Raft のリーダーの選出において appendEntriesRequest メッセージを使用する必要がある理由は、リーダーが他のサーバーが新しいリーダーを選出するための選挙を始めることを防ぐためである。サーバーは現在のリーダーから appendEntriesRequest メッセージを受信し、リーダーのタームが古くない場合、そのサーバーはリーダーの存在を認識し、フォロワーになる。

書き換え規則 *handleRequestVoteRequest* を以下のように定義する。

```
crl [handleRequestVoteRequest] :
{(term[S0]: T) (role[S0]: R) (votedBy[S0]: VB) (votedFor[S0]: VF0)
 (network: (msg(S1, S0, requestVoteRequest(U, S1)) NW)) OCs} =>
{(term[S0]: if U > T then U else T fi)
 (role[S0]: if U > T then follower else R fi)
 (votedBy[S0]:
   if ((VB == null or VB == S1) and U >= T) then S1 else VB fi)
 (votedFor[S0]: if U > T then empty else VF0 fi)
 (network:
  if ((VB == null or VB == S1) and U >= T)
  then (msg(S1, S0, requestVoteRequest(U, S1)) NW
        msg(S0, S1, requestVoteResponse(U, true)))
  else (msg(S1, S0, requestVoteRequest(U, S1)) NW
        msg(S0, S1, requestVoteResponse(
            (if  U > T then U else T fi), false))) fi) OCs}
if T <= maxTerm .
```

ネットワーク内に *msg(S1, S0, requestVoteRequest(U, S1))* が存在し、サーバー *S0* の現在のタームが *maxTerm* 以下の場合、以下の処理が行われる。サーバー *S0* の現在のターム T が U 以下（$U >= T$）であり、*S0* がまだどのサーバーにも投票していないか、*S0* に投票している場合（$VB == null$ または $VB == S0$）、*msg(S0,S1,requestVoteResponse(U, true))* がネットワークに追加され、サーバー *S0* がサーバー *S1* に投票することを意味する。上の条件と合致しない場合、*msg(S0,S1,requestVoteResponse(T2, false))* がネットワークに追加され、サーバー *S0* はサーバー *S1* に投票しない（サーバー *S0* がすでに投票したサーバーを変更しない）ことを意味する。ここで *T2* は、$U > T$ の場合は U、それ以外の場合は T である。$U > T$ の場合、サーバー *S0* の現在のタームは U になり、サーバー *S0* はフォロワーになり、サーバー *S0* に投票したサーバーのスープも空になる。*msg(S1, S0, requestVoteRequest(U, S1))* をネットワークから削除しない。これはメッセージの重複が発生する可能性があるためである。

書き換え規則 *handleRequestVoteResponse* を以下のように定義する。

```
crl [handleRequestVoteResponse] :
{(term[S0]: T) (role[S0]: R) (votedBy[S0]: VB) (votedFor[S0]: VF0)
(network: (msg(S1, S0, requestVoteResponse(U, B)) NW)) OCs} =>
{(term[S0]: if U > T then U else T fi)
 (role[S0]: if U > T then follower else
   (if length(VF1) >= majority then leader else R fi) fi)
 (votedBy[S0]: if U > T then null else VB fi)
 (votedFor[S0]: if U > T then empty else VF1 fi)
 (network: (msg(S1, S0, requestVoteResponse(U, B)) NW)) OCs}
if T <= maxterm /\ VF1 := if T == U and B == true
 and R == candidate then (VF0 S1) else VF0 fi .
```

ネットワーク内に *msg(S1, S0, requestVoteResponse(U, B))* が存在し、サーバー *S0* の現在のタームが *maxTerm* 以下の場合、以下の処理が行われる。サーバー *S0* の現在のターム *T* が *U* に等しく、*B* が真であり、*S0* が候補者である場合（すなわち、サーバー *S1* が *S0* に投票したサーバーのスープに追加される場合)、*VF1* を *VF0 S1* とする。ここで、*VF0* はこれまでS0に投票したサーバーのスープである。上の条件と合致しない場合、*VF1* を *VF0* とする（すなわち、サーバー *S1* はスープに追加されない)。*T* が *U* 以上であり、*VF1* 内のサーバー数がRaftクラスタに参加するサーバーの過半数である場合、サーバー *S0* はリーダーになる。*VF1* 内のサーバー数が過半数でない場合、*S0* の役割は変わらない。*T* が *U* 未満の場合、サーバー *S0* のタームは *U* になり、サーバー *S0* はフォロワーになり、サーバー *S0* によって投票されたサーバーは *null*（ダミーサーバーID）になり、サーバー *S0* に投票したサーバーのスープは空になる。*msg(S1, S0, requestVoteResponse(U, B))* をネットワークから削除しない。これはメッセージの重複が発生する可能性があるためである。

書き換え規則 *handleAppendEntriesRequest* を以下のように定義する。

```
crl [handleAppendEntriesRequest] :
{(term[S0]: T) (role[S0]: R) (votedBy[S0]: VB)
 (votedFor[S0]: VF0)
 (network: (msg(S1, S0, appendEntriesRequest(U, S1)) NW)) OCs} =>
{(term[S0]: if U > T then U else T fi)
 (role[S0]: if U >= T then follower else R fi)
 (votedBy[S0]: if U > T then null else VB fi)
 (votedFor[S0]: if U > T then empty else VF0 fi)
 (network:
  if U > T
  then (msg(S1, S0, appendEntriesRequest(U, S1)) NW
        msg(S0, S1, appendEntriesResponse(U, false)))
```

```
    else (msg(S1, S0, appendEntriesRequest(U, S1)) NW
          msg(S0, S1, appendEntriesResponse(T, true))) fi) OCs}
  if T <= maxTerm .
```

ネットワーク内に *msg(S1, S0, appendEntriesRequest(U, S1))* が存在し、サーバー *S0* の現在のタームが *maxTerm* 以下の場合、以下の処理が行われる。サーバー *S0* の現在のタームを T とする。*msg(S0, S1, appendEntriesResponse(U, B))* がネットワークに追加される。ここで、B は $U > T$ の場合は true、それ以外の場合は false となる。$U > T$ の場合、サーバー *S0* のタームは U になり、サーバー *S0* に投票されたサーバーは null（ダミーサーバー ID）になり、サーバー *S0* に投票したサーバーのスープは空になる。$U >= T$ の場合、サーバー *S0* はフォロワーになる。つまり、リーダーのターム（appendEntriesRequest に含まれる）がサーバーの現在のターム以上であれば、サーバーはリーダーを正当なものと認識し、再びフォロワーに戻る。*msg(S1, S0, appendEntriesRequest(U, S1))* はネットワークから削除しない。これはメッセージの重複が発生する可能性があるためである。

書き換え規則 *handleAppendEntriesResponse* を以下のように定義する。

```
crl [handleAppendEntriesResponse] :
{(term[S0]: T) (role[S0]: R) (votedBy[S0]: VB)
 (votedFor[S0]: VF0)
 (network: (msg(S1, S0, appendEntriesResponse(U, B)) NW)) OCs} =>
{(term[S0]: if U > T then U else T fi)
 (role[S0]: if U > T then follower else R fi)
 (votedBy[S0]: if U > T then null else VB fi)
 (votedFor[S0]: if U > T then empty else VF0 fi)
 (network: (msg(S1, S0, appendEntriesResponse(U, B)) NW)) OCs}
if T <= maxTerm .
```

ネットワーク内に *msg(S1, S0, appendEntriesResponse(U, B))* が存在し、サーバー *S0* の現在のタームが *maxTerm* 以下の場合、以下の処理が行われる。サーバー *S0* の現在のタームを T とする。$U > T$ の場合、サーバー *S0* のタームは U になり、サーバー *S0* はフォロワーになり、サーバー *S0* に投票されたサーバーは *null*（ダミーサーバー ID）になり、*S0* に投票したサーバーのスープは空になる。*msg(S1, S0, appendEntriesResponse(U, B))* はネットワークから削除しない。これはメッセージの重複が発生する可能性があるためである。

4.2　リーダーの選出のモデル検査

実験環境として、2.8GHz 16 コアプロセッサと 1.5 TB のメモリを搭載したコンピューターに SUSE Linux Enterprise Server 15 SP1 をインストールしたものを使

用している。このコンピューターは、JAIST の情報社会基盤研究センターによって管理されており、1 週間連続でコンピューターを使用することが許可されている。モデル検査の実験が 1 週間で終了しない場合、ジョブは強制終了される。タームが 2 以下でサーバー数が 3 の条件のもとで、いくつかのモデル検査の実験を行った。

まず、以下の Maude のコマンドを使用して、リーダーが正しく選出されることを確認した。

```
search [1] in RAFT : init =>* {(role[S0]: leader) OCs} .
```

Maude によって返される search コマンドの結果は以下の通りである。

```
Solution 1 (state 75)
states: 76  rewrites: 1451 in 0ms cpu
(1ms real) (~ rewrites/second)
OCs --> servers: (s0 s1 s2) network:
(msg(s0, s1, requestVoteRequest(1, s0))
 msg(s0, s2, requestVoteRequest(1, s0))
 msg(s1, s0, requestVoteResponse(1, true)))
(term[s0]: 1) (term[s1]: 1) (term[s2]: 0)
(role[s1]: follower) (role[s2]: follower)
(votedFor[s0]: s0) (votedFor[s1]: s0) (votedFor[s2]: null)
(votedBy[s0]: s0 s1) (votedBy[s1]: empty) votedBy[s2]: empty
S0 --> s0
```

これは初期状態から、あるサーバーがリーダーとして選出された状態に至るパスが存在することを示している。ただし、この結果はリーダーが最終的に選出されることを意味しているわけではない。リーダーが選出されないパスも存在する。

以下の Maude のコマンドを使用して、Raft が Election Safety Property を満たしているかどうかを確認する。

```
search [1] in RAFT : init =>*
{(term[S0]: T) (term[S1]: T) (role[S0]: leader)
 (role[S1]: leader) OCs} .
```

Maude によって返される search コマンドの結果は以下の通りである。

```
No solution.
states: 2810044  rewrites: 958548787 in 5601856ms cpu
(5602405ms real) (171112 rewrites/second)
```

これは Maude はどのタームにおいても、異なる 2 つのサーバーがリーダーである状態を見つけられなかったことを示している。このことによって、タームが 2 以

下でサーバー数が 3 の条件のもとで、Raft が Election Safety Property を満たしていると結論づけることができる。

タームが 3 以下でサーバー数が 3 の条件のもと、およびタームが 2 以下でサーバー数が 4 の条件のもとで、Raft が Election Safety Property を満たしているかどうかを確認するために、モデル検査の実験を行った。しかし、モデル検査の実験の完了に 1 週間以上かかり、モデル検査を実行しているジョブは強制終了された。そのため、タームが 3 以下でサーバー数が 3 の条件のもと、およびタームが 2 以下でサーバー数が 4 の条件のもとで Raft が Election Safety Property を満たすことは示せていない。

第5章　ログの複製

本章では Raft のログの複製の形式仕様とモデル検査について説明している。モデル検査の結果、Raft が Log Matching Property と State Machine Safety Property の性質が満たしていることを示している。Raft クラスタ内のフォロワーが、Raft のログの複製とは異なる動作を行うケースでもモデル検査をしている。サーバーの障害は、単純なシャットダウンだけでなく、誤った動作にもつながる可能性があるので、Raft としては後者も扱えることが望ましい。モデル検査の実験では、異なるログの複製を行うサーバー以外のサーバーが Log Matching Property と State Machine Safety Property の性質を満たすことを示している。

5.1　ログの複製の形式仕様

Raft のログの複製を状態遷移システムとして形式化するために、以下の観測可能成分を使用する。

- *(term[s]: t)* - s はサーバー ID、t はタームである。これは、サーバー s のタームが t であることを意味する。Raft クラスターに参加する各サーバー s に対して、この観測可能成分のインスタンスが使用される。

- *(role[s]: r)* - s はサーバー ID、r は役割であり、リーダーまたはフォロワーである。これは、サーバー s の役割が r であることを意味する。Raft クラスターに参加する各サーバーに対して、この観測可能成分のインスタンスが使用される。

- *(log[s]: l)* - s はサーバー ID、l はログエントリの配列である。これは、サーバー s がログ l を持っていることを意味する。サーバー s は、ログ l にログエントリを追加する。Raft クラスターに参加する各サーバーに対して、この観測可能成分のインスタンスが使用される。

- *(commitIndex[s]: ci)* - s はサーバー ID、ci はコミットされた最大のログエントリのインデックスである。これは、インデックス ci が、サーバー s がコミットしたログエントリの最大のインデックスであることを意味する。Raft クラスターに参加する各サーバー s に対して、この観測可能成分のインスタンスが使用される。

- *(nextIndex[s0][s1]: ni)* - *s0* および *s1* はサーバー ID、*ni* は、s0 が s1 に送信した次のログエントリのインデックスである。これは、サーバー *s0* が次にサーバー *s1* にログエントリを送信する際はインデックスが *ni* であるログエントリを送信することを意味する。リーダーは、それぞれのフォロワーに対して次に送信するログエントリのインデックスである nextIndex を保持する。リーダーの各フォロワーに対して、この観測可能成分のインスタンスが使用される。

- *(matchIndex[s0][s1]: mi)* - *s0* および *s1* はサーバー ID であり、*mi* は、s0 が s1 がログエントリの複製を持っていることを認識している最大のログエントリのインデックスである。これは、インデックス *mi* が、サーバー *s0* がサーバー *s1* にログエントリを送信した最大のインデックスであることを意味する。リーダーの各フォロワーに対して、この観測可能成分のインスタンスが使用される。

- *(servers: ss)* - *ss* はサーバー ID の集合である。これは、Raft クラスターに参加するすべてのサーバーの ID を維持する。1 つのインスタンスのみが使用され、*ss* は変更されない。

- *(clientRequests: c)* - *c* は、クライアントが Raft クラスターに送信するメッセージである。1 つのインスタンスのみが使用される。クライアントが Raft クラスターにメッセージを送信すると、そのメッセージは clientRequests から削除される。

- *(network: n)* - *n* はメッセージの集合である。これは、Raft クラスターに参加するサーバーがメッセージを交換するネットワークを表現する。1 つのインスタンスのみが使用される。一度 *n* に入れられたメッセージは削除されない。これは、メッセージが複製される可能性があることを表現している。メッセージはネットワークから削除されないが、サーバーがそのサーバー宛のメッセージを受信しないことがありうる。これは、メッセージが失われる可能性があることを表現している。

Raft クラスタに参加するサーバーがサーバー *s0*、サーバー *s1*、サーバー *s2* の 3 台ある場合、定義された初期状態 *init* は以下のようになる。

```
{(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)} .
```

初期状態では、観測可能成分は以下のようになる。

- 各サーバーの term は 1
- サーバー *s0* の役割はリーダー
- サーバー *s1* およびサーバー *s2* の役割はフォロワー
- 各サーバーによって追加されたログエントリの配列は空（つまり、各サーバーはまだログにログエントリを追加していない）
- 各サーバーがコミットしたインデックスは 0（つまり、各サーバーはまだログエントリをコミットしていない）
- サーバー *s0* がサーバー *s1* およびサーバー *s2* に送信する次のログエントリのインデックスは 1
- サーバー *s0* がサーバー *s1* およびサーバー *s2* にログエントリを送信した最大のインデックスは 0（つまり、サーバー *s0* はまだサーバー *s1* およびサーバー *s2* にログが複製されていることを知らない）
- Raft クラスターに参加するサーバーの集合は *s0 s1 s2* であり、3 つのサーバーが Raft クラスターに参加すると仮定している
- クライアントリクエストメッセージの集合は *cr0 cr1* であり、クライアントが Raft クラスターに 2 つのメッセージを送信すると仮定している
- ネットワークは空（つまり、ネットワークにはまだメッセージが入っていない）

Raft におけるログの複製を形式仕様として表現するために、各サーバーに対して *appendEntries*、*handleAppendEntriesRequest*、*handleAppendEntriesResponse* といった 3 つの書き換え規則を用いる。これらの書き換え規則では以下の Maude 変数を用いる。

- *OCs* - 観測可能成分の集合の変数
- *S0*、*S1*、*S2* - サーバー ID の変数
- *Ss* はサーバー ID の集合の変数
- *T*、*U*、*PLT* - term の変数

- *R*、*R0* - サーバーの役割の変数
- *CI*、*LCI* - commitIndex の変数
- *MI*、*MI1*、*MI2* - matchIndex の変数
- *NI*、*NI1*、*NI2* - nextIndex の変数
- *PLI* - 新しいログエントリの直前のインデックスの変数
- *PLT* - 新しいログエントリの直前の term の変数
- *Ls*、*L* - ログエントリの集合の変数
- *CR* - クライアントのリクエストメッセージの変数
- *CRs* - クライアントのリクエストメッセージの集合の変数
- *NW* - メッセージの集合の変数
- *AEReq* - AppendEntries リクエストメッセージの変数

書き換え規則 *appendEntries* を以下のように定義する。

```
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} .
```

この書き換え規則は、サーバー *S0* がリーダーであり、クライアントリクエスト *CR* が存在する場合、*S0* は他のすべてのサーバーに対して appendEntries リクエストメッセージを送信することを示している。*appendEntriesRequest(T, S0, length(Ls), term(Ls[length(Ls)]), log(T, CR), CI)* は、appendEntries リクエストメッセージの本文である。第 1 引数は *S0* のタームである。第 2 引数はリーダーのサーバー ID である。第 3 引数は新しいログエントリの直前のインデックス（prevLogIndex と呼ばれる）である。第 4 引数は新しいログエントリの直前の term（prevLogTerm と呼ばれる）である。第 5 引数は *S0* の term とクライアントのリクエストメッセージを含むログである。第 6 引数は *S0* の commit index である。$Ss - S0$ は、サーバー ID の集合 *Ss* からサーバー ID *S0* を削除した結果得られるサーバー ID の集合で

ある。*mkAppendEntriesRequests(S0, AEReq, Ss')* は、メッセージの本文が *AEReq* であり、*Ss'* 内のすべてのサーバーに対して appendEntries リクエストメッセージを作成する。

書き換え規則 *handleAppendEntriesRequest* を以下のように定義する。

```
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 .
```

ネットワーク内に *msg(S1, S0, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI))* が存在し、サーバー *S0* のログの長さが 3 未満の場合、以下が実行される。サーバー *S0* の現在の term T が U 以下である場合（$U >= T$）、およびサーバー *S0* が prevLogIndex でタームが prevLogTerm と一致するログエントリを持っている場合、またはリーダーがまだログエントリをログに追加していない場合（(Ls[PLI] =/= null and term(Ls[PLI]) == PLT) or (PLI == 0)）、B は真である。B が真であり、サーバー *S0* がまだインデックス $PLI + 1$ にログエントリを持っていない場合（*B and Ls[PLI + 1] == null*）、サーバー *S0* はログエントリをログに追加する。サーバー *S0* の commitIndex が prevLogIndex よりも小さく（$CI < PLI$）、サーバー *S0* の既存のログエントリが新しいものと競合している場

合（*length(Ls) =/= PLI or term(Ls[PLI]) =/= PLT*）、*S0* は既存のログエントリとそれに続くすべてのログエントリを削除する。リーダーの commitIndex がサーバー *S0* の commitIndex よりも大きい場合、リーダーの commitIndex と最新のログエントリのインデックスが比較され、サーバー *S0* の commitIndex は小さい方で更新される。$U > T$ の場合、サーバー *S0* の現在のタームは U となり、*msg(S0, S1, appendEntriesResponse(U, B, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI))* がネットワークに追加される。それ以外の場合、サーバー *S0* の現在のタームは同じままであり、*msg(S0, S1, appendEntriesResponse(T, B, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI))* がネットワークに追加される。$U >= T$ の場合、サーバー *S0* はフォロワーになる。*msg(S1, S0, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI))* をネットワークから削除しない。これはメッセージの重複が発生する可能性があるためである。

書き換え規則 *handleAppendEntriesResponse* を以下のように定義する。

```
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 .
```

ネットワークに *msg(S1, S0, appendEntriesResponse(U, B, AEReq))* が存在する場合、以下の処理が行われる。*appendEntriesResponse* に含まれる *B* が真である場合、*S0* の *S1* に対する matchIndex は、*prevLogIndex(AEReq) + 1*（その appendEntriesRequest に含まれる prevLogIndex に 1 を加えたもの）と *MI1*（サーバー *S0* のサーバー *S1* に対する matchIndex）のうち大きい方で更新され、サーバー *S0* のサーバー *S1* に対する nextIndex は、サーバー *S0* のサーバー *S1* に対する matchIndex に 1 を加えたもので更新される。それ以外の場合、*S0* の *S1* に対する nextIndex は、*S0* の *S1* に対する nextIndex から 1 を引いたものと 1（nextIndex の最小値は 1）のうち大きい方で更新される。*appendEntriesResponse* が重複している可能性があり、サーバー *S0* が複数回 appendEntriesResponse を受信する可能性があるため、単純に matchIndex と nextIndex をインクリメントするのは誤りである。*N* を前のインデックスに 1 を加えたものとする。サーバー *S0* のサーバー *S1* に対する matchIndex またはサーバー *S0* のサーバー *S2* に対する matchIndex が *N* 以上である場合、サーバー *S0* のログのインデックス *N* でのタームはサーバー *S0* の現在のタームであり、サーバー *S0* の役割がリーダーであり、*N* がサーバー *S0* の commitIndex より大きい場合、サーバー *S0* の commitIndex は *N* で更新される。*B* が偽であり、*U* が *T* 以下であり、サーバー *S0* の役割がリーダーである場合、*msg(S0, S1, appendEntriesRequest(T, S0, PI, term(Ls[PI]), Ls[NI], leaderCommit(AEReq)))* がネットワークに追加される。これはサーバー *S0* とサーバー *S1* の間でログに不整合があるために、サーバー *S0* が、1 つ前の nextIndex のログエントリで appendEntriesRequest メッセージを送信することを意味する。$U > T$ である場合、*S0* の term は *U* になり、*S0* はフォロワーになる。*msg(S1, S0, appendEntriesResponse(U, B, AEReq))* をネットワークから削除しない。これはメッセージの重複が発生する可能性があるためである。

5.2 ログの複製のモデル検査

実験環境は、リーダーの選出の実験と同様である。サーバーのログの長さが 3 未満で、サーバーの数が 3 であるという条件のもとで、いくつかのモデル検査の実験を行った。

5.2.1 サーバーが期待どおりの動作をすることを前提としたモデル検査

まず、以下の Maude のコマンドを使用して、ログが正しく複製されていることを確認した。

```
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 によって返される search コマンドの結果は以下の通りである。

```
Solution 1 (state 823)
states: 824  rewrites: 134117 in 50ms cpu
(54ms real) (2682340 rewrites/second)
OCs --> servers: (s0 s1 s2) clientRequests: empty network:
(msg(s0, s1, appendEntriesRequest(1, s0, 0, 0, log(1, cr0), 0))
 msg(s0, s1, appendEntriesRequest(1, s0, 1, 1, log(1, cr1), 1))
 msg(s0, s2, appendEntriesRequest(1, s0, 0, 0, log(1, cr0), 0))
 msg(s0, s2, appendEntriesRequest(1, s0, 1, 1, log(1, cr1), 1))
 msg(s1, s0, appendEntriesResponse(1, true,
  appendEntriesRequest(1, s0, 0, 0, log(1, cr0), 0)))
 msg(s1, s0, appendEntriesResponse(1, true,
  appendEntriesRequest(1, s0, 1, 1, log(1, cr1), 1)))
 msg(s2, s0, appendEntriesResponse(1, true,
  appendEntriesRequest(1, s0, 0, 0, log(1, cr0), 0)))
 msg(s2, s0, appendEntriesResponse(1, true,
  appendEntriesRequest(1, s0, 1, 1, log(1, cr1), 1))))
(term[s0]: 1) (term[s1]: 1) (term[s2]: 1)
(role[s1]:follower) role[s2]: follower
S0 --> s0
L0:Logs --> logWithIndex(1, log(1, cr0)),
            logWithIndex(2, log(1, cr1))
S1 --> s1
L1:Logs --> logWithIndex(1, log(1, cr0)),
            logWithIndex(2, log(1, cr1))
S2 --> s2
L2:Logs --> logWithIndex(1, log(1, cr0)),
            logWithIndex(2, log(1, cr1))
```

Maude の search コマンドによって、初期状態からすべてのサーバーが 2 つのログエントリを持ち、リーダーはインデックス 2 でコミットし、他のサーバーはインデックス 1 でコミットし、リーダーの他のサーバーに対する matchIndex が 2 であり、リーダーの他のサーバーに対する nextIndex が 3 である状態に至るパスが存在することが示された。これは、2 つのログエントリが正しく複製されていることを意味する。ただし、この結果はログエントリが最終的に正しく複製されていることを示しているわけではない。ログエントリが最終的に正しく複製されていないパスは存在する。

以下の Maude のコマンドを使用して、Raft が Log Matching Property を満たしているかどうかを確認する。

```
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)]))) .
```

Maude によって返される search コマンドの結果は以下の通りである。

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

これは Maude が同じインデックスとタームを持つログエントリを含む 2 つのログがある場合、そのインデックスまでのすべてのログエントリにおいて異なるログエントリを含むような状態を見つけなかったことを示している。したがって、サーバーのログの長さが 3 未満で、サーバーの数が 3 である条件のもとで、Raft は Log Matching Property を満たしていると結論づけることができる。

以下の Maude のコマンドを使用して、Raft が State Machine Safety Property を満たすかどうかを確認する。

```
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]))).
```

Maude によって返される search コマンドの結果は以下の通りである。

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

これはMaudeがあるサーバーが特定のインデックスでログエントリをコミットし、他のサーバーが同じインデックスで異なるログエントリをコミットするような状態を見つけなかったことを示している。したがって、サーバーのログの長さが3未満で、サーバーの数が3である条件のもとで、Raftは Machine Safety Property を満たすと結論づけることができる。

5.2.2 サーバーが期待しない動作をすることを前提としたモデル検査

Raftクラスタ内でフォロワーが期待しない動作を行うことを想定する。前の説では *SERVER-ID* モジュールを以下のように定義していた。

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

この説では *SERVER-ID* モジュールを以下のように定義する。

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

ServerID と *BadServerID* がソートで、ソート *BadServerID* はソート *ServerID* のサブソートである。また、*s0*、*s1*、および *s2* は定数、*s0* と *s1* のソートはServerIDであり、*s2* のソートは *BadServerID* である。

以下の書き換え規則を追加する。

```
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) .
```

サーバーIDのソートが *BadServerID* であり、ネットワーク内に *msg(S1, S2:BadServerID, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI))* が存在し、サーバー *S0* のログの長さが3未満である場合、以下の処理が行われる。サーバー *S2* は、クライアントのリクエスト *crb* およびターム U を含むログエントリをログに追加する。ただし、*crb* は正常ではない値であり、クライアントが送信した値ではない。サーバー *S2* のcommitIndexは、サーバー *S2* のログの長さによって更新される。もし $U > T$ であれば、サーバー *S0* の現在のタームは U になり、*msg(S2:BadServerID, S1, appendEntriesResponse(U, true, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI))* がネットワークに追加される。それ以外の場合、サーバー *S0* の現在のタームは変わらず、*msg(S2:BadServerID, S1, appendEntriesResponse(T, true, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI))* がネットワークに追加される。もし $U >= T$ であれば、サーバー *S0* はフォロワーになる。*msg(S1, S2:BadServerID, appendEntriesRequest(U, S1, PLI, PLT, log(U, CR), LCI)* をネットワークから削除しない。これはメッセージの重複が発生する可能性があるためである。

この書き換え規則は期待しない動作を示している。ログエントリの複製方法とコミット方法は、Raftにおけるログの複製とは異なる。ソートが *BadServerID* であるサーバー *s2* は、書き換え規則 *handleAppendEntriesRequest* または書き換え規則 *badHandleAppendEntriesRequest* を実行する。これは、ソート *BadServerID* がソート *ServerID* のサブソートであるためである。つまり、サーバー *s2* は正常な操作と期待しない操作の両方を実行する。

以下のMaudeのコマンドを使用して、変更された仕様でログが正しく複製されることを確認した。

```
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 によって返される search コマンドの結果は以下の通りである。

```
Solution 1 (state 4772)
states: 4773  rewrites: 718392 in 430ms cpu
(426ms real) (1670679   rewrites/second)
OCs --> servers: (s0 s1 s2) clientRequests: empty network:
(msg(s0, s1, appendEntriesRequest(1, s0, 0, 0, log(1, cr0), 0))
 msg(s0, s1, appendEntriesRequest(1, s0, 1, 1, log(1, cr1), 1))
 msg(s0, s2, appendEntriesRequest(1, s0, 0, 0, log(1, cr0), 0))
 msg(s0, s2, appendEntriesRequest(1, s0, 1, 1, log(1, cr1), 1))
 msg(s1, s0, appendEntriesResponse(1, true,
  appendEntriesRequest(1, s0, 0, 0, log(1, cr0), 0)))
 msg(s1, s0, appendEntriesResponse(1, true,
  appendEntriesRequest(1, s0, 1, 1, log(1, cr1), 1)))
 msg(s2, s0, appendEntriesResponse(1, true,
  appendEntriesRequest(1, s0, 0, 0, log(1, cr0), 0)))
 msg(s2, s0, appendEntriesResponse(1, true,
  appendEntriesRequest(1, s0, 1, 1, log(1, cr1), 1))))
 (term[s0]: 1) (term[s1]: 1) (term[s2]: 1)
 (role[s1]: follower) role[s2]: follower
S0 --> s0
L0:Logs --> logWithIndex(1, log(1, cr0)),
            logWithIndex(2, log(1, cr1))
S1 --> s1
L1:Logs --> logWithIndex(1, log(1, cr0)),
            logWithIndex(2, log(1, cr1))
S2 --> s2
L2:Logs --> logWithIndex(1, log(1, cr0)),
            logWithIndex(2, log(1, cr1))
```

Maude の search コマンドによって、すべてのサーバーが 2 つのログエントリを持ち、リーダーがインデックス 2 でコミットし、他のサーバーがインデックス 1 でコミットし、リーダーの他のサーバーに対する matchIndex が 2 であり、リーダーの他のサーバーに対する nextIndex が 3 である状態に至る初期状態からのパスが存在することが示している。これは 2 つのログエントリが正しく複製されていること

を意味する。ただし、結果はログエントリが最終的に正しく複製されていることを示していない。2 つのログエントリが正しく複製されていないパスが存在する。

次の Maude コマンドを使用して、サーバー *s0* およびサーバー *s1* に対して Raft が Log Matching Property を満たすかどうかを Maude で確認する。

```
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)]))) .
```

Maude によって返される search コマンドの結果は以下の通りである。

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

これは Maude がサーバー *s0* とサーバー *s1* の両方が同じインデックスとタームを持つログエントリを持ち、サーバー *s0* のログとサーバー *s1* のログが、そのインデックスまでのすべてのログエントリで異なるログエントリを含む状態を見つけられなかったことを示している。したがって、サーバーのログの長さが 3 未満でサーバー数が 3 である条件で、期待しない動作を行わないサーバーは Raft が Log Matching Property を満たしていると結論付けることができる。

以下の Maude のコマンドを使用してサーバー *s0* およびサーバー *s1* に対して Raft が State Machine Safety Property を満たすかどうかを Maude で確認する。

```
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]))).
```

Maude によって返される search コマンドの結果は以下の通りである。

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

これは Maude がサーバー *s0* またはサーバー *s1* があるインデックスでログエントリをコミットし、反対のサーバーが同じインデックスの異なるログエントリをコミットする状態を見つけられなかったことを示している。したがって、サーバーのログの長さが 3 未満でサーバー数が 3 である条件のもとで、期待しない動作を行わないサーバーで Raft が State Machine Safety Property を満たしていると結論付けることができる。

サーバーのログの長さが 4 未満でサーバー数が 3 である条件と、サーバーのログの長さが 3 未満でサーバー数が 4 である条件のもとで、Raft が Log Matching Property を満たすかどうか、および State Machine Safety Property を満たすかどうかを確認するためのモデル検査の実験を行った。しかし、モデル検査の実験は 1 週間以上かかり、モデル検査を実行しているジョブは強制終了された。そのため、サーバーのログの長さが 4 未満でサーバー数が 3 である条件とサーバーのログの長さが 3 未満でサーバー数が 4 である条件のもとで、Raft が Log Matching Property と State Machine Safety Property を満たすことは示せていない。

第6章　おわりに

本章では、6.1 節で Raft のモデル検査の内容、モデル検査の実験を通して得た結論を報告する。6.2 節で Raft のモデル検査の実験から得られた課題を報告する。

6.1　本研究の成果と結論

Raft のリーダーの選出とログの複製の形式仕様を作成し、モデル検査を行った。形式仕様の記述とモデル検査は Maude を用いている。Maude とは書換え論理に基づく仕様およびプログラミング言語である。リーダーの選出の形式仕様は 200 行程度、ログの複製の形式仕様は 250 行程度の規模である。モデル検査を実行する実験環境として、2.8GHz 16 コアプロセッサと 1.5 TB のメモリを搭載したコンピューターに SUSE Linux Enterprise Server 15 SP1 をインストールしたものを使用している。このコンピューターは、JAIST の情報社会基盤研究センターによって管理されており、1 週間連続でコンピューターを使用することが許可されている。リーダーの選出とログの複製ごとに説明する。

6.1.1　リーダーの選出

Raft におけるリーダーの選出では、タームと呼ばれる論理時間ごとに最大 1 人のリーダーを選出する。通常の操作では、Raft クラスタには 1 人だけのリーダーがおり、他のすべてのサーバーはフォロワーである。リーダーは定期的に自身が正常に動作していることを示すために、ハートビートメッセージを他のすべてのサーバーに送信する。フォロワーが特定の時間内にリーダーからのハートビートメッセージを受け取らない場合、そのフォロワーはリーダーが正常に動作していないと判断し、リーダーの選出を開始するために候補者になる。

このような Raft におけるリーダーの選出に対して Maude で形式仕様を作成し、その形式仕様に基づいてモデル検査を行った。リーダーの選出のモデル検査の実験では、タームが 2 以下でサーバー数が 3 の条件のもとで、以下の 2 つを示した。

- サーバーがリーダーとして選出されること
- Election Safety Property が満たされていること

Election Safety Property とは、各タームで最大 1 人のリーダーしか選出されないという性質である。

タームが 3 以下でサーバー数が 3 の条件のもと、およびタームが 2 以下でサーバー数が 4 の条件のもとで、Raft が Election Safety Property を満たしているかどうかを確認するために、モデル検査の実験を行った。しかし、モデル検査の実験の完了に 1 週間以上かかり、モデル検査を実行しているジョブは強制終了された。そのため、タームが 3 以下でサーバー数が 3 の条件のもと、およびタームが 2 以下でサーバー数が 4 の条件のもとで Raft が Election Safety Property を満たすことを示せていない。

6.1.2 ログの複製

ログの複製では、リーダーはクライアントからのリクエストを受け入れ、そのリクエストを自分のログに保存し、他のすべてのサーバーに転送する。各サーバーはそのようなリクエストを受け取ると、それらを自分のログに保存する。リーダーはサーバーの過半数からクライアントリクエストに対する肯定的な返答を受け取ると、そのリクエストを自身の状態機械に適用する。各サーバーはクライアントからのリクエストを保持する状態機械を持っている。フォロワーはメッセージを受け取ると、リーダーがコミットしたインデックスまでのログの情報を状態機械に適用する。

このような Raft におけるログの複製に対して Maude で形式仕様を作成し、その形式仕様に基づいてモデル検査を行った。ログの複製のモデル検査の実験では、サーバーのログの長さが 3 未満でサーバー数が 3 である条件のもとで、以下の 3 つを示した。

- ログが正しく複製されること
- Log Matching Property が満たされていること
- State Machine Safety Property が満たされていること

Log Matching Property とは、同じインデックスとタームのログエントリを含む 2 つのログの場合、そのインデックスまでのすべてのログエントリで同一であるという性質である。State Machine Safety Property とは、任意の 2 つのサーバーが同じインデックスで 2 つのログエントリをそれぞれの状態機械に適用した場合、2 つのログエントリは常に同じでなければならないという性質である。

さらに、Raft のログの複製とはログの追加方法やコミットの方法が異なる動作を行うことがあるサーバーが存在すると仮定した場合の形式仕様を作成し、モデル検査を行った。サーバーの障害は、単純なシャットダウンだけでなく、誤った動作にもつながる可能性があるので、Raft としては後者も扱えることが望ましいため、このような実験を行った。異なる動作をするサーバーは受け取った値とは異

なる値をログに追加し、それをすぐにコミットするという動きをする。このような場合においても、Raft のログの複製とは異なる動作を行わないサーバーは Log Matching Property と State Machine Safety Property が満たされていることをモデル検査で示した。この検証を行っている先行研究はないと考えている。

サーバーのログの長さが 4 未満でサーバー数が 3 である条件と、サーバーのログの長さが 3 未満でサーバー数が 4 である条件のもとで、Raft が Log Matching Property を満たすかどうか、および State Machine Safety Property を満たすかどうかを確認するためのモデル検査の実験を行った。しかし、モデル検査の実験は 1 週間以上かかり、モデル検査を実行しているジョブは強制終了された。そのため、サーバーのログの長さが 4 未満でサーバー数が 3 である条件とサーバーのログの長さが 3 未満でサーバー数が 4 である条件のもとで、Raft が Log Matching Property と State Machine Safety Property を満たすことを示せていない。

6.2 今後の課題

状態爆発し、モデル検査が完了しないところで課題が残っている。

- Raft におけるリーダーの選出とログの複製を組み合わせた形式仕様に対するモデル検査で Election Safety Property や Log Matching Property、State Machine Safety Property が満たされていることを示すことができない。本稿のリーダーの選出の形式仕様では、クライアントからログを受け取らず、どのサーバーもログが増えていない。また、ログの複製の形式仕様では、選挙が開始されず、リーダーが変更したり、タームが増加したりしていない。リーダーの選出とログの複製を組み合わせた形式仕様を作成し、その形式仕様に基づいてモデル検査を行った。しかし、モデル検査の実験は 1 週間以上かかり、モデル検査を実行しているジョブは強制終了された。モデル検査に要する時間を見積もるための実験を行ったところ、数十年以上かかる可能性があることがわかった。リーダーの選出とログの複製を組み合わせたものを検証することが今後の課題として残っている。

- 前節で記載したように、リーダーの選出のモデル検査でタームの最大値が 2、サーバーの台数が 3 台といった制限がある条件のもとでしか Election Safety Property が満たされることを示すことができていない。また、ログの複製のモデル検査ではログの最大の長さが 2 とサーバーの台数が 3 台という制限がある条件のもとでのでしか Log Matching Property と State Machine Safety Property が満たされているを示すことができない。サーバーの台数やタームの最大値、ログの長さを増やした条件のもとでもモデル検査し、Raft が期待される Election Safety Property や Log Matching Property、State Machine Safety Property などの性質が満たされることを示すことが今後の課題として残っている。

上記の課題を解決するためには何らかの方法で状態数を削減する必要がある。

Raft にはリーダーの選出とログの複製以外にも、実運用を考慮して、以下のような機能がある。

- クラスターのメンバーシップ交換
- ログの圧縮

クラスターのメンバーシップ交換とは Raft クラスターを停止せず、クラスターに新しくサーバーを追加したり、サーバーを削除したりする仕組みのことである。サーバーの設定を動的に安全に更新したり、クラスターのサイズを変更したりする際にこの機能が利用される。ログの圧縮とはある時点の状態機械のスナップショットを作成する機能である。システムが長期間稼働するにつれてログが肥大化していき、ストレージの容量を大きく消費してしまう問題を解決するためにこの機能が利用される。これらの機能に対しても形式仕様を作成し、モデル検査で Raft が期待される Election Safety Property や Log Matching Property、State Machine Safety Property などの性質が満たされることを示すことが今後の課題として残っている。

また、本課題研究では Raft クラスターとクライアント間の通信では問題が発生しないことを前提としていた。Raft クラスターとクライアント間の通信で問題が発生する状況を想定した形式仕様を作成し、モデル検査で Raft が期待される性質を満たすことを示めせていない。

付録 A ソースコード

本稿の実験に用いた Maude のコードは、https://github.com/11Takanori/raft-maude にアップロードしている。

謝辞

本研究に取り組むにあたり、主指導教員としてご指導くださった緒方和博教授に心からの感謝の意を表します。先生の熱意に満ちた指導により、本課題研究報告書を形にすることができました。先生のお力添えがなければ、私はこの研究を完遂することができませんでした。平石邦彦教授、青木利晃教授、石井大輔准教授にも感謝いたします。中間審査で貴重なお時間を割いていただき、多くの示唆に富む指摘とアドバイスを頂戴しました。

参考文献

[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] M. Clavel, et al., Ed., All About Maude, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4350.

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

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

[5] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2000. Model checking. MIT Press, Cambridge, MA, USA.

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

[7] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. 4, 3 (July 1982), 382-401. https://doi.org/10.1145/357172.357176

[8] 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

[9] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In Proceedings of the

36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15). Association for Computing Machinery, New York, NY, USA, 357-368. https://doi.org/10.1145/2737924.2737958

[10] William Schultz, Ian Dardik, and Stavros Tripakis. 2022. Formal verification of a distributed dynamic reconfiguration protocol. In Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2022). Association for Computing Machinery, New York, NY, USA, 143-152. https://doi.org/10.1145/3497775.3503688

[11] Wolf Honoré, Ji-Yong Shin, Jieung Kim, and Zhong Shao. 2022. Adore: atomic distributed objects with certified reconfiguration. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022). Association for Computing Machinery, New York, NY, USA, 379-394. https://doi.org/10.1145/3519939.3523444

[12] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for decidability of deductive verification with applications to distributed systems. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018). Association for Computing Machinery, New York, NY, USA, 662-677. https://doi.org/10.1145/3192366.3192414

[13] Leslie Lamport. 1998. The Part-Time Parliament. ACM Trans. Comput. Syst. 16, 2 (1998), 133ś169. https://doi.org/10.1145/279227.279229

[14] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16). Association for Computing Machinery, New York, NY, USA, 614-630. https://doi.org/10.1145/2908080.2908118

[15] Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, and Ivan Beschastnikh. 2023. Compiling Distributed System Models with PGo. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 159-175. https://doi.org/10.1145/3575693.3575695

[16] Leslie Lamport. 2009. The PlusCal Algorithm Language. Theoretical Aspects of Computing-ICTAC 2009, Martin Leucker and Carroll Morgan editors. Lecture Notes in Computer Science, number 5684, 36-60.

(Jan 2009). https://www.microsoft.com/ en-us/research/publication/pluscal-algorithm-language/

[17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15). Association for Computing Machinery, New York, NY, USA, 1-17. https://doi.org/10.1145/2815400.2815428

[18] ianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2022. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In USENIX Symposium on Operating Systems Design and Implementation (OSDI).

[19] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In USENIX Symposium on Operating Systems Design and Implementation (OSDI).

[20] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. 2021. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI).

[21] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: incremental inference of inductive invariants for verification of distributed protocols. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19). Association for Computing Machinery, New York, NY, USA, 370-384. https://doi.org/10.1145/3341301.3359651

[22] Haojun Ma, Hammad Ahmad, Aman Goel, Eli Goldweber, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci. 2022. Sift: Using Refinement-guided Automation to Verify Complex Distributed Systems. In USENIX Annual Technical Conference (ATC).

[23] Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. 2015. Asynchronous programming, analysis and testing with state machines. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15). Association for Computing Machinery, New York, NY, USA, 154-164. https://doi.org/10.1145/2737924.2737996

[24] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. 2007. Mace: language support for building distributed systems. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07). Association for Computing Machinery, New York, NY, USA, 179-188. https://doi.org/10.1145/1250734.1250755

[25] Si Liu, Atul Sandur, José Meseguer, Peter Csaba Ölveczky, and Qi Wang. 2020. Generating Correct-by-Construction Distributed Implementations from Formal Maude Designs. In NASA Formal Methods: 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, Proceedings. Springer-Verlag, Berlin, Heidelberg, 22-40. https://doi.org/10.1007/978-3-030-55754-6_2

[26] Aman Goel and Karem Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In Proceedings of the 13th NASA Formal Methods Symposium (NFM '21), pages 131-150, May 2021.

[27] Aman Goel and Karem A Sakallah. Towards an automatic proof of Lamport's Paxos. In Proceedings of the 21st Conference on Formal Methods in Computer Aided Design (FMCAD '21), pages 112-122, October 2021.

[28] Floyd, R. 1967. Assigning Meanings to Programs. In Proceedings of Symposia in Applied Mathematics.

[29] etcd. [n.d.]. https://etcd.io

[30] CockroachDB. [n.d.]. https://github.com/cockroachdb/cockroach

[31] YugabyteDB. [n.d.]. https://github.com/yugabyte/yugabyte-db.

[32] TiKV. [n.d.]. https://github.com/pingcap/tidb.

[33] MongoDB. [n.d.]. https://github.com/mongodb/mongo

[34] Go. [n.d.]. https://go.dev

発表済み論文

- Takanori Ishibashi, Kazuhiro Ogata. 2023. Formal Specification and Model Checking of Raft Leader Election in Maude. In ICSCA 2023: 2023 12th International Conference on Software and Computer Applications, ACM.
- Takanori Ishibashi, Kazuhiro Ogata. 2023. Formal Specification and Model Checking of Raft Log Replication in Maude. In DMSVIVA 2023: 29th International DMS Conference on Visualization and Visual Languages, KSI Research Inc.