International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2021, 11(4): 473–503, doi: 10.21655/ijsi.1673-7288.00257

Research Article

Raft with Out-of-Order Executions

Xiaosong Gu (谷晓松)[1], Hengfeng Wei (魏恒峰)[1], Lei Qiao (乔磊)[2], Yu Huang (黄宇)[1]

[1] (State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)
[2] (Beijing Institute of Control Engineering, Beijing 100190, China)
Corresponding author: Hengfeng Wei, hfwei@nju.edu.cn; Lei Qiao, fly2moon@aliyun.com

Abstract PolarFS is a distributed file system developed by Alibaba with ultra-low latency and high availability. It implements a variant of the Raft consensus protocol, called ParallelRaft. ParallelRaft breaks Raft's strict serialization restrictions in the commitment and execution of log entries and enables state machines to commit and execute log entries in an out-of-order way. However, ParallelRaft is not open-sourced. It has only a brief description, lacking formal specification. Moreover, the correctness of ParallelRaft has not been manually proven or formally checked. The purpose of the study is to provide a precise formal specification for ParallelRaft and to prove its correctness. Specifically, the following main contributions are accomplished. First, to clarify the relationship between Raft and ParallelRaft, ParallelRaft-SE (Sequential Execution) is proposed, which allows out-of-order commitment but prohibits out-of-order executions. Also, a refinement mapping from ParallelRaft-SE to Multi-Paxos is established. Second, it is discovered that ParallelRaft, according to its brief description in the literature, neglects the so-called "ghost log entries" phenomenon, which may violate the consistency among state machines. Therefore, based on ParallelRaft-SE, ParallelRaft-CE (Concurrent Execution) is proposed. ParallelRaft-CE avoids the "ghost log entries" phenomenon and ensures the consistency among state machines when executing concurrently by limiting parallelism in the commitment of log entries. The correctness of ParallelRaft-CE is proved manually. Finally, the formal specifications of ParallelRaft-SE and ParallelRaft-CE are provided by TLA+ (TLA stands for temporal logic of actions), and the refinement mapping from ParallelRaft-SE to Multi-Paxos and the correctness of ParallelRaft-CE are verified using the TLC model checker when the number of participants of the protocols is small.

Keywords Raft; ParallelRaft; Multi-Paxos; consensus protocols; TLA+; refinement; model checking

Citation Gu XS, Wei HF, Qiao L, Huang Y. Raft with out-of-order executions, *International Journal of Software and Informatics*, 2021, 11(4): 473–503. http://www.ijsi.org/1673-7288/257.htm

Distributed consensus is the core of distributed computing, which requires multiple participants to agree on a value or a set of values (also called a sequence)[1, 2]. Distributed systems

This is the English version of the Chinese article "支持乱序执行的 Raft 协议. 软件学报, 2021, 32(6): 1748–1778. doi: 10.13328/j.cnki.jos.006248".
Funding items: National Natural Science Foundation of China (61932021, 61772258); Space Advanced Computing and Electronic Information Laboratory of BICE (OBCandETL-2020-04)
Received 2020-08-30; Revised 2020-10-26; Accepted 2021-01-27; IJSI published online 2021-12-23

generally use consensus protocols to provide the required strong consistency, including open-sourced Ceph[3], Spanner from Google[4], MySQL from Oracle[5], PaxosStore from Tencent[6], and PolarDB from Alibaba[7].

Multi-Paxos (Paxos)[8, 9] and Raft[10] are two classical protocols to solve the distributed consensus problem. Both based on the replicated state machine[11] model, they guarantee state consistency across multiple replica nodes through voting and log replication. Each user command goes through commitment and execution phases. If a command receives a vote from a majority of servers, it is called to be committed. Only the committed commands can be executed. For multiple commands, both phases can be performed in a sequential or out-of-order way. In the out-of-order way, the command corresponding to a large log number may be committed or executed before the command corresponding to a small log number. For example, Raft requires sequential commitment and sequential executions, while Multi-Paxos allows out-of-order commitment but prohibits out-of-order executions.

PolarDB uses the distributed file system PolarFS[7]. To improve the system performance, PolarFS implements the ParallelRaft consensus protocol based on Raft that allows out-of-order commitment and out-of-order executions. However, the precise formal specification of ParallelRaft is not provided. Particularly, there is no complete description of its out-of-order execution mechanism. Moreover, ParallelRaft has not yet been proved mathematically or verified formally. This paper aims to provide a precise formal specification for ParallelRaft and prove its correctness with refinement relationships[12] and mathematical arguments. Specifically, the main contributions of this paper are as follows.

- To clarify the relationship between ParallelRaft and Raft, we propose the ParallelRaft-SE (Sequential Execution) protocol based on Raft, which allows out-of-order commitment but prohibits out-of-order executions. ParallelRaft can be viewed as an out-of-order execution version of ParallelRaft-SE. We establish the refinement relationship between ParallelRaft-SE and Multi-Paxos (showing ParallelRaft-SE is an implementation of Multi-Paxos), thus verifying the correctness of ParallelRaft-SE.
- Besides, we find that the out-of-order execution mechanism of ParallelRaft which is described in literature may ignore the "ghost log entries" phenomenon that can break state consistency. Studies show that in the case of a sequential execution (e.g., Raft or Multi-Paxos), "ghost log entries" phenomenon does not break state consistency among replicas. On the contrary, we will prove that it may cause state inconsistency among replicas during an out-of-order execution. Further, it is argued that the solutions to Raft and Multi-Paxos do not solve the consistency problem of state machines in the mode of out-of-order execution.
- The ParallelRaft-CE (Concurrent Execution) protocol is proposed on the basis of ParallelRaft-SE, which supports out-of-order executions. By limiting the parallelism of ParallelRaft-SE in out-of-order commitment, ParallelRaft-CE avoids the "ghost log entries" phenomenon. We prove the correctness of ParallelRaft-CE.
- At last, we provide the formal specification of ParallelRaft-SE and ParallelRaft-CE in the TLA+[13–15] language. For the case where the number of protocol participants is small, we use the model checker TLC[16] to verify the refinement relationship between ParallelRaft-SE and Multi-Paxos and the correctness of ParallelRaft-CE.

Section 1 introduces propaedeutics, including the formal specification language TLA+, the distributed consensus problem, and the Multi-Paxos and Raft consensus protocols. Section 2 gives a description of the ParallelRaft-SE protocol and establishes the refinement relationship between ParallelRaft-SE and Multi-Paxos. Section 3 analyzes the impact of the "ghost log entries" phenomenon on state consistency in the mode of out-of-order executions

and points out the shortcomings of existing solutions. Section 4 introduces the ParallelRaft-CE protocol, which allows out-of-order executions and avoids the "ghost log entries" phenomenon. Section 5 briefly proves the correctness of ParallelRaft-CE. Section 6 verifies the refinement relationship between ParallelRaft-SE and Multi-Paxos and the correctness of ParallelRaft-CE by the model checker TLC. Section 7 discusses related work. Section 8 summarizes the paper and discusses possible work in the future. The complete TLA+ specification and model checking results are available in the GitHub repository[17].

1 Propaedeutics

1.1 Introduction of TLA+

TLA+, developed by Leslie Lamport, is a formal specification language[13] based on the temporal logic of actions[15]. It is particularly suitable for describing concurrent systems and distributed protocols.

A TLA+ specification contains a set of variables, an initial state, and a set of actions. It is usually expressed as $Spec = Init \wedge [Next]_{vars}$, where *vars* is the set of all variables. A state is an assignment of all variables. The predicate *Init* defines the initial state of the system. *Next* indicates the disjunctive form of all actions and defines the transition relationships between states. $[Next]_{vars}$ is true when and only when *Next* is true (an action is true, namely that an action is executed) or the values of all variables remain unchanged. A behavior is a sequence of states. TLA+ uses a variable without an apostrophe to represent its value in the current state and a variable with an apostrophe to indicate the value in the new state. In this way, an action can be described by a formula that contains variables with and without an apostrophe. For example, the action $x' = x + 1$ indicates that the value of the variable x in the new state is increased by 1 compared with that in the old state.

TLA+ supports first-order predicate logic and Zermelo-Fraenkel (ZF) set theory, which can express many types of data[18, 19]. Table 1 summarizes logics and set operators used in this paper. Reference [20] provides the complete list of TLA+ operators.

Table 1 A summary of the TLA+ operators used in this paper

Style	Operator	Meaning
Logic	*CHOOSE* $x \in S : P$	Select the element x satisfying the condition p in the set S (generally used in the case where x is the only one satisfying the condition)
Set	*SUBSET* S	Power set of S
Function	$f(e)$	Apply the function f on the parameter e
	$[x \in S \mapsto e]$	For given $x \in S$, the function $f(x) = e$
	$[f$ *EXCEPT* $![e_1] = e_2]$	$\bar{f} : \bar{f}[e] = \begin{cases} e_2 & \text{if } e = e_1 \\ f[e] & \text{otherwise} \end{cases}$
Record	$e.h$	Record the domain h of e
	$[h_1 \mapsto e_1 \cdots h_n \mapsto e_n]$	Record whose domain h_i is e_i
	$[h_1 : S_1 \cdots h_n : S_n]$	Set composed of records satisfying that domain h_i belongs to S_i
	$[r$ *EXCEPT* $![h] = e]$	Record $\bar{r}$ equals to r except $\bar{e}.h = e$
	$[r$ *EXCEPT* $![h] = e]$, e contains symbol @	@ in e stands for $r.h$
Tuple	$e[i]$	The i-th component of the tuple e
Action Operator	e'	Value of e in the new state
	UNCHANGED e	e remains unchanged: $e' = e$
	$[A]_e$	Action A holds or e remains unchanged
Sequence Operator	$\Box F$	Always holds ($\Box$ means "always")
	$\diamond F$	Holds eventually ($\diamond$ means "eventually")

TLA+ allows mutual reference in the form of modules. Each module can declare constants and variables, define operators, or propose theorems[18, 19]. A module can introduce declarations, definitions, and theorems from other modules by an extend command. The introduced modules can be instantiated. For example, the module M introduces module M_1.

$$IM_1 B \text{ INSTANCE } M_1 \text{ WITH } p_1 \leftarrow e_1, \cdots, p_n \leftarrow e_n,$$

where p_i contains all the constants and variables in module M_1, and e_i is a legitimate expression defined by the constants and variables in M. This statement replaces p_i in M_1 with corresponding e_i. We can access the expression F in module M_1 through $IM_1!F$. When e_i is same with p_j, the implicit substitution rule of TLA+ allows us to omit $p_j \leftarrow e_i$[18, 19].

TLC[16] is a model checker for TLA+. It can traverse all possible system behaviors and check all states to verify whether a system satisfies specific properties. However, some distributed systems contain infinite states. For example, the TLA+ specification usually contains natural numbers, which are infinite. To verify such systems, TLA+ introduces a model. All sets in the model are finite, and thus the number of system states is finite. The problem of combination explosion often occurs in model checking. In response, TLC can use the symmetry of the model to reduce the state space. For example, it is assumed that CONSTANTS *Server* defines the set of all processes in a system. In model checking, we need to instantiate it as a finite set, i.e., $Server = \{S_1, S_2, S_3\}$. If the system specification satisfies given properties under any permutation of S_1, S_2, and S_3 (e.g., S_1 replaces S_2; S_2 replaces S_3; S_3 replaces S_1), we can set *Server* as a symmetry set[13, 21].

In TLA+, refinement relationships[12] are used to describe the logical implication relationships between modules[18]. Refinement relationships are defined by refinement mapping. For example, the refinement mapping ϕ from the specification *ImplSpec* in the module *ImplModule* to the specification *AbsSpec* of the module *AbsModule* makes each variable v in *AbsSpec* correspond to an expression $\bar{v}$, which is defined by the variables in *ImplSpec*. For each state s in *ImplSpec*, the refinement mapping ϕ defines the state $\bar{s}$ of *AbsSpec*, and the value of a variable v in $\bar{s}$ is defined by $\bar{v}$ in s. If σ is the behavior $s_1 \rightarrow s_2 \rightarrow \cdots$ of *ImplSpec*, we define the behavior $\bar{\sigma}$ of *AbsSpec* as $\bar{s}_1 \rightarrow \bar{s}_2 \rightarrow \cdots$. *ImplSpec* implements/refines *AbsSpec* (*ImplSpec* $\Rightarrow$ *AbsSpec*) under the refinement mapping ϕ when and only when for any behavior σ satisfying *ImplSpec*, the behavior $\bar{\sigma}$ satisfies the specification *AbsSpec*[14]. To check the refinement relationship between *ImplSpec* and *AbsSpec* under the refinement mapping ϕ using TLC, we add a definition *AbsSub* $\doteq$ INSTANCE *AbsModule* in the module *ImplModule* and verify the theorem *ImplSpec* $\Rightarrow$ *AbsSub!AbsSpec*[18].

1.2 Distributed consensus

Distributed consensus requires multiple replica server nodes to maintain a consistent state. Each server node can be modeled as a replicated state machine that performs state switch by executing user commands.

State machines are generally replicated by the mechanism of replicating logs. Each server keeps one copy of a log. A log consists of sequentially numbered (usually with natural numbers) log entries. Each log entry stores one command from users. The server reads the next command that has achieved consensus from the log in sequence, executes it on a state machine, and returns the result to users. As the server executes commands in a numbered order, we call this sequential execution. Traditional distributed consensus protocols (such as Multi-Paxos and Raft) adopt sequential executions. Under this condition, we assume that the replicated state machines on the replica servers have the same initial state. Then the state consistency among servers can be ensured as long as logs are consistent. Thus, the distributed consensus problem

can be translated to guaranteeing consistency among logs on different replica server nodes. Specifically, the following properties should be satisfied.

- Nontriviality: Only the commands issued by users can be agreed on.
- Consistency: Each location can only achieve consensus on at most one command.

The server ensures the log matching property by running consensus protocols. In this paper, we consider the consensus protocol of asynchronous messaging systems, whose failure models are listed below.

- The server may fail by stop, but no Byzantine failures occur[8].
- Messages may be delayed, arrive out of order, be lost or duplicated, but the content of messages will not be tampered with.

1.3 Paxos protocol

The Paxos protocol is the classical protocol for solving distributed consensus problems, which allows a group of servers to agree on a single value (i.e., a single log entry). It is the basis for Multi-Paxos. Paxos defines three roles: proposer, acceptor, and learner. Proposers propose values; acceptors choose values; and learners learn the values that have been chosen. Paxos consists of two phases, and each phase contains two subphases[8].

- Prepare phase (also called Phase1)
  - Subphase Phase1a: The proposer chooses a globally unique number b (usually a natural number) and sends a prepare request numbered b to all acceptors.
  - Subphase Phase1b: The acceptor receives a prepare request numbered b. If it has previously received a prepare request with a number greater than b, it ignores the request with the number of b. Otherwise, the acceptor replies to the proposer with the proposal having the largest number among the proposals it has accepted (including the number and value).
- Accept phase (also called Phase2)
  - Subphase Phase2a: The proposer receives responses to its prepare request numbered b from a majority of acceptors. If the responses do not include any proposals, the proposer can select any value (which is generally the user command that this proposer receives). Otherwise, the proposer selects the value of the highest-numbered response. If the proposer selects the value v, the proposer sends the accept request $< b, v >$ to all acceptors.
  - Subphase Phase2b: An acceptor receives an accept request numbered b. If the acceptor does not receive a request with a number greater than b, it accepts this request. Otherwise, the acceptor ignores this accept request.

1.4 Multi-Paxos protocol

Multi-Paxos runs a separate Paxos instance for each log entry, thus supporting replica servers to reach consensus on logs (sequences of log entries). Since Paxos instances are parallel and independent, Multi-Paxos allows committing user commands in an out-of-order way. In other words, it allows replica servers to reach consensus on the user command corresponding to the log entry with a larger number, without waiting to reach consensus on previous log entries.

In practice, scholars often improve the performance of Multi-Paxos by batching messages of phase1. In this case, the system contains a recovery phase, in which phase1 messages from different instances are subjected to centralized processing.

The module *MultiPaxos* describes the Multi-Paxos protocol. It includes three constants.

- *Acceptors*: the set of all acceptors.
- *Value*: the set of all possible proposal values.
- *Nil*: a special symbol, which does not belong to *Value*.

We use natural numbers to denote the numbers of possible proposals and the number of each instance, i.e., *BallotsNat*, *InstancesNat*. *Quorums* defines a special kind of quorum system: each of its elements is a set consisting of more than half of acceptors.

The specification of Multi-Paxos includes six variables.

- *ballot*: $ballot[a]$ denotes the largest proposal number recorded by the acceptor a and the smallest proposal number that a can accept.
- *vote*: $vote[a][i][b]$ indicates the proposal value accepted by the acceptor a for the instance numbered i when the proposal number of a is b. If a does not accept any value for the instance numbered i when the proposal number is b, then $vote[a][i][b] = Nil$.
- *leaderVote*: $leaderVote[b][i]$ indicates a two-tuples consisting of the proposal number b and the value that the proposer proposes for the instance numbered i when the proposal number is b. In the initial state, for each proposal number b and instance number i, $leaderVote[b][i] =< -1, Nil >$.
- $1amsgs$, $1bmsgs$, and $2amsgs$: different types of message sets.

```
---------------------------- MODULE MultiPaxos ---------------------------------
EXTENDS Integers, FiniteSets
CONSTANTS Acceptors, Nil, Value
Ballots==Nat
Instances==Nat
Quorums=={Q\in SUBSET Acceptors: Cardinality(Q)>Cardinality(Acceptors)\2}
Max(s)==CHOOSE x\in s:\for all y\in s:x\geq y
VARIABLES ballot, vote, leaderVote, 1amsgs, 1bmsgs, 2amsgs
```

The main actions defined by the protocol include the followings:

- $Phase1a(b)$: It corresponds to $Phase1a$ of Paxos. The proposer selects the proposal number b and sends the prepare request to other nodes with a proposal number of b.
- $Phase1b(a, b)$: It corresponds to $Phase1b$ of Paxos. Node a receives the vote request numbered b. If $b > ballot[a]$, a sets $ballot[a]$ to b. For each instance number i, a replies to proposers with the value and the number of the highest-numbered proposal it has accepted.

```
Phase1a(b)==
  /\1amsgs'=1amsgs\cup {⟨⟨b⟩⟩}
  /\UNCHANGED ⟨⟨ballot,vote,leaderVote,1bmsgs,2amsgs⟩⟩

MaxAcceptorVote(a,i)==
  LET maxBallot==Max({b\in Ballots:vote[a][i][b]#Nil}\cup {-1})
    v==IF maxBallot>-1 THEN vote[a][i][maxBallot] ELSE Nil
  IN ⟨⟨maxBallot,v⟩⟩

Phase1b(a,b)==
  /\ballot[a]<b
  /\⟨⟨b⟩⟩\in 1amsgs
  /\ballot'=[ballot EXCEPT ![a]=b]
  /\1bmsgs'=1bmsgs\cup
    {⟨⟨b,{⟨⟨i,MaxAcceptorVote(a,i)⟩⟩:i\in Instances},a⟩⟩}
  /\UNCHANGED ⟨⟨vote,leaderVote,1amsgs,2amsgs⟩⟩

IncreaseBallot(a,b)==
  /\ballot[a]<b
  /\ballot'=[ballot EXCEPT ![a]=b]
  /\UNCHANGED ⟨⟨vote,leaderVote,1amsgs,1bmsgs,2amsgs⟩⟩
```

- $Merge(b)$: It corresponds to $Phase2a$ of Paxos together with action $Propose(b, i)$, $Phase2a(b, i)$. In the *merge* action, the proposer updates its log depending on the received vote responses, but does not initiate an accept request. When the proposer who initiates the prepare request numbered b receives a reply from a majority, it selects proposal values

according to responses (with the method described in Phase2a of Paxos) for each instance number i and saves them in *leaderVote*$[b][i]$. It should be noted that *leaderVote*$[b][i]$ can be updated only if it has not been modified previously (*leaderVote*$[b][i] =< -1, Nil >$). In Multi-Paxos, each proposal number can only have one proposer to initiate the vote, and only one proposal value can be proposed for each instance number.

- *Propose*(b, i): The proposer proposes a proposal value for the instance number i. If *leaderVote*$[b][i] =< -1, Nil >$, the proposer chooses a legitimate proposal value v to update *leaderVote*$[b][i] =< b, v >$ and initiates a request. Otherwise, the proposer sends the *leaderVote*$[b][i]$ directly to acceptors.
- *Phase*$2a(b, i)$: The proposer initiates the accept request in accordance with the proposal value confirmed by *Merge*(b) and *Propose*(b, i) for the instance number i.
- *Vote*(a, b, i): It corresponds to *Phase*$2b$ of Paxos. The acceptor a receives the accept request with a proposal number of b and an instance number i. If b is not smaller than the proposal number of a ($b >$ *ballot*$[a]$), a accepts the accept request and updates *vote*$[a][i][b]$ to the corresponding proposal value.

```
1bMsgs(b,Q)=={m\in 1bmsgs:m[3]\in Q/\m[1]=b}

MaxVote(b,i,Q)==
  LET entries==UNION {m[2]:m\in 1bMsgs(b,Q)}
    ientries=={e\in entries:e[1]=i}
    maxBal==Max({e[2][1]:e\in ientries})
  IN CHOOSE v\in Value\cup {Nil}:\E e\in ientries:
    /\e[2][1]=maxBal/\e[2][2]=v

lastInstance(b,Q)==LET entries==UNION {m[2]:m\in 1bMsgs(b,Q)}
                       valid=={e\in entries:e[2][1]/=-1}
                   IN IF valid={·} THEN -1 ELSE Max({e[1]:e\in valid})
Merge(b)==/\\E Q\in Quorums:
          /\\A a\in Q:\E m\in 1bMsgs(b,Q):m[3]=a
          /\leaderVote'=[leaderVote EXCEPT ![b]=[i\in Instances ↦
            IF (/\i\in 0..lastInstance(b,Q)
              /\leaderVote[b][i][1]=-1)
            THEN ⟨⟨b,MaxVote(b,i,Q)⟩⟩
            ELSE leaderVote[b][i]]]
          /\UNCHANGED ⟨⟨vote,ballot,1amsgs,1bmsgs,2amsgs⟩⟩

Propose(b,i)==/\leaderVote[b][i][1]=-1
              /\\E Q\in Quorums:
              /\\A a\in Q:\E m\in 1bMsgs(b,Q):m[3]=a
              /\LET maxV==MaxVote(b,i,Q)
                safe==IF maxV/=Nil THEN {maxV} ELSE
                  Value \cup {Nil}
                IN \E v\in safe: leaderVote'=[leaderVote EXCEPT ![b][i]=⟨⟨b,v⟩⟩]
              /\UNCHANGED ⟨⟨vote,ballot,1amsgs,1bmsgs,2amsgs⟩⟩

Phase2a(b,i)==
  /\leaderVote[b][i][1]=b
  /\2amsgs'=2amsgs\cup {⟨⟨b,i,leaderVote[b][i]⟩⟩}
  /\UNCHANGED ⟨⟨ballot,vote,leaderVote,1amsgs,1bmsgs⟩⟩

Vote(a,b,i)==
  /\ballot[a]\leq b
  /\ballot'=[ballot EXCEPT ![a]=b]
  /\\E m\in 2amsgs:
    /\m[2]=i/\m[1]=b
    /\vote'=[vote EXCEPT ![a][i][b]=m[3][2]]
  /\UNCHANGED ⟨⟨leaderVote,1amsgs,1bmsgs,2amsgs⟩⟩
```

Next defines the substate relationship, and *Spec* defines the complete behavior specification.

```
Next==
  \/\E a\in Acceptors,b\in Ballots:IncreaseBallot(a,b)
  \/\E b\in Ballots:Phase1a(b)
  \/\E a\in Acceptors,b\in Ballots:Phase1b(a,b)
  \/\E b\in Ballots:Merge(b)
  \/\E b\in Ballots,i\in Instances:Propose(b,i)
  \/\E b\in Ballots,i\in Instances:Phase2a(b,i)
  \/\E a\in Acceptors,b\in Ballots,i\in Instances:Vote(a,b,i)
Spec==Init/\[·][Next]_⟨⟨leaderVote,ballot,vote,1amsgs,1bmsgs,2amsgs⟩⟩
```

1.5 Raft protocol

Raft is a more understandable distributed consensus protocol[10] that enhances serializability among log entries and simplifies the protocol design. Each node in Raft maintains an incremental variable called term. A term is essentially a logical clock jointly maintained by nodes. Through terms, nodes can discover outdated messages. Specifically, nodes should carry their current terms when sending messages. If the term value carried by a message that a node receives is smaller than the current term value of the node, the node rejects this message. Otherwise, the node updates its term value. When a node adds a new log entry to the log, it saves its current term value in the log entry as well, which becomes the term of this log entry.

Nodes in Raft have three roles, i.e., leader, follower, and candidate. In the initial state, all nodes are followers. The Raft protocol consists of two main parts: electing a leader and the leader synchronizing logs to followers. Under normal circumstances, the leader periodically sends heartbeat signals to other nodes to maintain authority. When a follower does not receive a heartbeat signal for a period of time, it is transformed to a candidate. The candidate first increases its own term and sends the term to all nodes to initiate the election. After receiving the election request, a node will compare its own term with the term carried by the election request. If its own term is larger, or if it has already voted for another candidate during the term, it rejects this election request. The candidate receiving votes from the majority becomes the new leader. The majority voting mechanism of Raft ensures election safety[10]: There is at most one leader in a term. To ensure the completeness of the new leader, namely that its log should contain all the log entries that have been committed, Raft introduces the following rule. A node will reject this vote request if the log of the candidate is older than its own log[10]. We can determine the oldness of logs by comparing the number and the term (which are called *lastIndex* and *lastTerm*) of the highest-numbered log entry of nodes[10].

The leader synchronizes log entries to followers according to the log number order, and the followers should accept log entries of the leader in the number order. A follower cannot accept a log entry with a larger number until it receives the log entry with a smaller number. The leader and followers maintain the next acceptable number by an ack mechanism. Unlike Multi-Paxos, where instances send and receive log entries independently, Raft establishes a fully sequential relationship of all log entries by numbering. By such a restriction, logs of nodes in Raft do not have holes, and the log matching property between nodes is guaranteed[10]. If the logs of two nodes have the same log entries at the same location, then the log entries at all previous locations must also be the same. According to log matching property, if a log entry has been committed, then all the log entries with smaller numbers in the log have also been committed.

2 ParallelRaft-SE Protocol and Refinement

Raft requires sequential commitment and sequential executions of user commands and this makes it unsuitable for highly concurrent systems[7]. Thus, Alibaba proposed ParallelRaft based on Raft, which allows out-of-order commitment and out-of-order executions of user

commands[7]. To clarify the relationship between ParallelRaft and Raft, we propose ParallelRaft-SE. ParallelRaft-SE allows out-of-order commitment but prohibits out-of-order executions of user commands. Thus, ParallelRaft-SE can be viewed as a sequential execution version of ParallelRaft. To verify the correctness of ParallelRaft-SE, we establish the refinement relationship between ParallelRaft-SE and Multi-Paxos.

2.1 ParallelRaft-SE protocol

ParallelRaft-SE follows roles (leader, follower, and candidate), the concept of term, and the leader election mechanism in Raft. The ParallelRaft-SE protocol consists of three main parts: the leader synchronizing logs to followers, leader election, and log recovery.

In ParallelRaft-SE, the leader can send multiple log entries to followers concurrently. Followers accept and acknowledge the log entries immediately after receiving them without waiting for log entries with smaller numbers. Thus, ParallelRaft-SE supports out-of-order acceptance and commitment of logs.

The election mechanism of ParallelRaft-SE is basically the same as that of Raft, and they both need to ensure the election safety. The difference is that since ParallelRaft-SE does not have the serializability as Raft, there may be holes in the logs of nodes. Thus, during election, ParallelRaft-SE cannot guarantee the completeness of the new leader by comparing the oldness of logs. Thus, ParallelRaft-SE adds a log recovery phase.

The basic idea of log recovery in ParallelRaft-SE is that the new leader collects logs from other nodes and runs the Paxos protocol, so as to recover those log entries that may have been committed but are not stored by the new leader.

After recovery, the new leader satisfies completeness and can then synchronize log entries to followers.

The specification of ParallelRaft-SE uses the following constants and variables (only the ones that are newly introduced compared with Multi-Paxos are presented).

- *Server* is the set of all nodes participating in the consensus.
- *Follower*, *candidate*, and *leader* are three different states of servers.
- $r1$*amsgs*, $r1$*bmsgs*, $r2$*amsgs*, $r2$*bmsgs*, $r3$*amsgs*, and *negMsgs* are sets of different types of messages.
- *currentTerm*$[i]$ is the maximum term recorded by node i.
- *currentState*$[i]$ is the state of the node i, which is one of *Follower*, *Candidate*, and *Leader* at any time.
- *vote*$[i][n][t]$ indicates the log entry with the number of n that is accepted by the node i in the term t. The *vote* is introduced to realize the refinement from ParallelRaft-SE to Multi-Paxos, which is not required in practical protocols.
- *leaderLog* records the log of the new leader in each term. *leaderLog* is also used to establish the refinement from ParallelRaft-SE to Multi-Paxos, which is not required in practical protocols. *leaderLog*$[t][n]$ indicates the log entry numbered n in the log of the leader in the term t. Each log entry is a three-tuples in the form of $< t', v, b >$, where t' indicates the term of the log entry; v is the proposal value; b is a Boolean value. When and only when the log entry is committed, b is true.
- $\log[i][n]$ indicates the log entry numbered n in the log of the node i.

```
--------------------------- MODULE ParallelRaft-SE ---------------------------
EXTENDS Integers,FiniteSets,Sequences,TLC
CONSTANTS Server,Follower,Candidate,Leader,Nil,Value

Quorums=={i\in SUBSET(Server):Cardinality(i)*2>Cardinality(Server)}
Index==Nat
Term==Nat
```

```
VARIABLE r1amsgs,r1bmsgs,r2amsgs,r2bmsgs,r3amsgs,negMsgs,currentTerm,
         currentState,vote,leaderLog,log
serverVars==⟨⟨currentTerm,currentState⟩⟩
vars==⟨⟨r1amsgs,r1bmsgs,r2amsgs,r2bmsgs,r3amsgs,negMsgs,log,serverVars,
       leaderLog,vote⟩⟩
```

The major actions include the followings:

- *Timeout*(i): If a follower or candidate i times out as it has not received a message from the leader, it increases its own term ($currentTerm'[i] = currentTerm[i] + 1$) and changes itself to a candidate ($currentState'[i] = Candidate$).
- *RequestVote*(i): A candidate i sends its own term to other nodes to initiate an election request.
- *HandleRequestVote*(i): A node i receives the election request m. If the term carried by m is greater than that of i ($m[1] > currentTerm[i]$), i upgrades its term, accepts this election request, and sends its log to candidates. Otherwise, i rejects this election request.

```
Timeout(i)==
  /\currentState[i]\in {Follower,Candidate}
  /\currentTerm'=[currentTerm EXCEPT ![i]=currentTerm[i]+1]
  /\currentState'=[currentState EXCEPT ![i]=Candidate]
  /\UNCHANGED ⟨⟨r1amsgs,r1bmsgs,log,r2amsgs,r2bmsgs,r3amsgs,negMsgs,
               leaderLog,vote⟩⟩

RequestVote(i)==
  /\currentState[i]=Candidate
  /\r1amsgs'=r1amsgs\cup {⟨⟨currentTerm[i],i⟩⟩}
  /\UNCHANGED ⟨⟨serverVars,r1bmsgs,log,r2amsgs,r2bmsgs,r3amsgs,negMsgs,
               leaderLog,vote⟩⟩

HandleRequestVoteRequest(i)==
/\\E m\in r1amsgs:
  LET j==m[2]
    grant==m[1]>currentTerm[i]
    entries=={⟨⟨n,log[i][n]⟩⟩:n\in Index}
  IN
    \//\grant
      /\UpdateTerm(i,m[1])
      /\r1bmsgs'=r1bmsgs\cup {⟨⟨m[1],entries,i,j⟩⟩}
      /\UNCHANGED negMsgs
    \//\\neg grant
      /\negMsgs'=negMsgs\cup {⟨⟨currentTerm[i],j⟩⟩}
      /\UNCHANGED ⟨⟨currentState,currentTerm,r1bmsgs⟩⟩
/\UNCHANGED ⟨⟨log,r1amsgs,r2amsgs,r2bmsgs,r3amsgs,vote,leaderLog⟩⟩
```

- *BecomeLeader*(i): The candidate i that receives the majority of votes becomes the leader and restores the log entries that may be missing depending on received logs. For each log number n, log entries numbered n that it receives are examined. The log entry with the largest term is selected, and the term of this log entry is modified to the term of i. To establish the refinement relationship between ParallelRaft-SE and Multi-Paxos, *leaderLog* is modified in the recovery process instead of $\log[i]$ being modified directly. After that, i can send the *RequestSync* request to itself to update the log.
- *RequestSync*(i): The leader i synchronizes log entries to other nodes.

```
Merge(entries,term)==
  LET committed=={e\in entries:e[3]=TRUE}
      chosen==
        CASE committed={·} CHOOSE x\in entries:
                        \A y\in entries:x[1]\geq y[1]
        [·] committed/={·} CHOOSE x\in committed:TRUE
      safe==chosen[2]
```

```
  IN ⟨⟨term,safe,chosen[3]⟩⟩

BecomeLeader(i)==
  /\currentState[i]=Candidate
  /\\E Q\in Quorums:
    LET voteGranted=={m\in r1bmsgs:m[4]=i/\m[3]\in Q
                      /\m[1]=currentTerm[i]}
      allLog==UNION {m[2]:m\in voteGranted}
      valid=={e\in allLog:e[2][1]/=-1}
      end==IF valid={·} THEN -1 ELSE Max({e[1]:e\in valid})
    IN
    /\\A q\in Q:\E m\in voteGranted:m[3]=q
    /\leaderLog'=[leaderLog EXCEPT ![currentTerm[i]]=
                 [n\in Index ↦ IF n\in 0..end THEN
                  Merge({l[2]:l\in {t\in allLog:t[1]=n}},currentTerm[i])
                    ELSE ⟨⟨-1,Nil,FALSE⟩⟩]]
  /\currentState'=[currentState EXCEPT ![i]=Leader]
  /\UNCHANGED ⟨⟨currentTerm,r1amsgs,r2amsgs,r1bmsgs,r2bmsgs,r3amsgs,
              negMsgs,log,vote⟩⟩

RequestSync(i)==
  /\currentState[i]=Leader
  /\LET sync=={n\in Index:leaderLog[currentTerm[i]][n][1]/=-1}
    IN
    \E n\in sync:r2amsgs'=r2amsgs\cup
      {⟨⟨currentTerm[i],n,leaderLog[currentTerm[i]][n],i⟩⟩}
  /\UNCHANGED ⟨⟨serverVars,log,r1amsgs,r1bmsgs,r2bmsgs,
              r3amsgs,negMsgs,leaderLog,vote⟩⟩
```

- *HandleRequestSyncRequest*(i): After i receives the *RequestSync* request m, if the term carried by m is not smaller than the term of i ($m[1] > \mathit{currentTerm}[i]$), i receives this request and upgrades its own term. At the same time, i updates $\log[i]$ and *vote*, and replies with acknowledgement.
- *CommitEntry*(i): After receiving an acknowledgement from a majority of nodes for a log entry, the leader i marks the log entry as committed.
- *RequestCommit*(i): The leader sends the committed log entries to other nodes.

```
HandleRequestSyncRequest(i)==
  /\\E m\in r2amsgs:
    LET j==m[4]
      grant==m[1]\geq currentTerm[i]
    IN
    /\\//\m[1]>currentTerm[i]
        /\UpdateTerm(i,m[1])
      \//\m[1]\leq currentTerm[i]
        /\UNCHANGED ⟨⟨currentTerm,currentState⟩⟩
    /\\//\grant
        /\log'=[log EXCEPT ![i][m[2]]=m[3]]
        /\vote'=[vote EXCEPT ![i][m[2]][m[1]]=m[3][2]]
        /\r2bmsgs'=r2bmsgs\cup {⟨⟨m[1],m[2],i,j⟩⟩}
        /\UNCHANGED negMsgs
      \//\\neg grant
        /\negMsgs'=negMsgs\cup {⟨⟨currentTerm[i],j⟩⟩}
        /\UNCHANGED ⟨⟨vote,r2bmsgs,log⟩⟩
  /\UNCHANGED ⟨⟨r1amsgs,r1bmsgs,r2amsgs,r3amsgs,leaderLog⟩⟩

CommitEntry(i)==
  /\\E index\in Index,Q\in Quorums:
    LET syncSuccess=={m\in r2bmsgs:
                      m[4]=i/\m[3]\in Q
                      /\m[1]=currentTerm[i]/\m[2]=index}
    IN
    /\currentState[i]=Leader
    /\\A q\in Q:\E m \in syncSuccess:m[3]=q
    /\leaderLog'=[leaderLog EXCEPT ![currentTerm[i]][index][3]=TRUE]
  /\UNCHANGED ⟨⟨serverVars,log,r1amsgs,r1bmsgs,r2amsgs,r2bmsgs,
  r3amsgs,negMsgs,vote⟩⟩
```

```
RequestCommit(i)==
  /\currentState[i]=Leader
  /\LET committed=={n\in Index:leaderLog[currentTerm[i]][n][3]=TRUE} IN
    \E n\in committed:r3amsgs'=r3amsgs\cup {⟨⟨currentTerm[i],n,i⟩⟩}
  /\UNCHANGED ⟨⟨serverVars,log,r1amsgs,r1bmsgs,r2amsgs,r2bmsgs,
              negMsgs,leaderLog,vote⟩⟩
```

- *HandleRequestCommit*(i): After a node i receives a *RequestCommit* request m from the leader, if the term carried by m is not smaller than the term of i, i marks the corresponding log entry as committed.
- *ClientRequest*(i): After receiving a user command v, the leader i adds v as a new log entry to the log.

```
HandleRequestCommitRequest(i)==
  /\\E m\in r3amsgs:
    LET grant==currentTerm[i]\leq m[1]
        j==m[3]
    IN
    /\\//\m[1]>currentTerm[i]
        /\UpdateTerm(i,m[1])
      \//\m[1]\leq currentTerm[i]
        /\UNCHANGED ⟨⟨currentTerm,currentState⟩⟩
    /\\//\grant
        /\log[i][m[2]][1]=m[1]
        /\log'=[log EXCEPT ![i][m[2]][3]=TRUE]
        /\UNCHANGED negMsgs
      \//\\neg grant
        /\negMsgs'=negMsgs\cup {⟨⟨currentTerm[i],j⟩⟩}
        /\UNCHANGED log
  /\UNCHANGED ⟨⟨serverVars,r1amsgs,r1bmsgs,r2amsgs,r2bmsgs,
              r3amsgs,leaderLog,vote⟩⟩

ClientRequest(i)==
  LET ind=={b\in Index:leaderLog[currentTerm[i]][b][1]/=-1}
    nextIndex==IF ind={·}
      THEN 0
      ELSE Max(ind)+1
  IN
  /\currentState[i]=Leader
  /\\E v\in Value:leaderLog'=[leaderLog EXCEPT ![currentTerm[i]][nextIndex]=
                        ⟨⟨currentTerm[i],v,FALSE⟩⟩]
  /\UNCHANGED ⟨⟨serverVars,log,r1amsgs,r1bmsgs,r2amsgs,r2bmsgs,r3amsgs,
              negMsgs,vote⟩⟩
```

Next defines the substate relationship. *Spec* defines the complete behavior specification.

```
Next==\/\E i\in Server:Timeout(i)
  \/\E i\in Server:RequestVote(i)
  \/\E i\in Server:HandleRequestVoteRequest(i)
  \/\E i\in Server:BecomeLeader(i)
  \/\E i\in Server:CommitEntry(i)
  \/\E i\in Server:ClientRequest(i)
  \/\E i,j\in Server:RequestCommit(i)
  \/\E i\in Server:HandleRequestCommitRequest(i)
  \/\E i,j\in Server:RequestSync(i)
  \/\E i\in Server:HandleRequestSyncRequest(i)

Spec==Init/\[·][Next]_vars
```

2.2 Refinement from ParallelRaft-SE to Multi-Paxos

ParallelRaft-SE supports out-of-order commitment and sequential executions of user commands, which is the same as Multi-Paxos. Moreover, the log recovery phase of ParallelRaft-

SE essentially uses Paxos to reconfirm the log entries that may be missing, while Multi-Paxos initiates proposals through Paxos. In fact, we can establish the refinement relationship between ParallelRaft-SE and Multi-Paxos, thus proving the correctness of ParallelRaft-SE. This refinement relationship is based on the following similarities between ParallelRaft-SE and Multi-Paxos.

- *RequestVote* corresponds to *Phase*1a. The term in ParallelRaft-SE corresponds to the proposal number of Multi-Paxos.
- *HandleRequestVote* corresponds to *Phase*1b. The two both need to compare terms/proposal numbers to decide whether to agree an election/accept a proposal request, and the two both need to include their own logs in responses.
- *BecomeLeader* corresponds to *Merge*. The leader/proposer that receives replies from the majority runs Paxos to recover log entries that may be missing.
- *RequestSync* corresponds to *Phase*2a. In ParallelRaft-SE, the leader synchronizes log entries to followers after completing log recovery. In Multi-Paxos, the proposer selects the proposal value by running Paxos and synchronizes it to acceptors.
- *HandleRequestSync* corresponds to *Vote*. In ParallelRaft-SE, followers update their logs after receiving the synchronization request. In Multi-Paxos, acceptors accept the proposal and record it locally after receiving the proposal from the proposer.
- *ClientRequest* corresponds to *Propose*. In ParallelRaft-SE, the leader accepts the user command and adds it to the end of the log as a log entry. In Multi-Paxos, if the proposer does not receive a proposal value for a certain number, it can propose any legitimate value. Thus, it can add a legitimate log entry to the end of the log.

The ParallelRaft-SE specification provides the refinement mapping between constants and variables in ParallelRaft-SE and Multi-Paxos.

```
Acceptors==Server
Ballots==Term
Instances==Index
ballot==currentTerm
leaderVote==[i\in Ballots ↦ [j\in Index ↦ ⟨⟨leaderLog[i][j][1],leaderLog[i][j]
     [2]⟩⟩]]
1amsgs=={⟨⟨m[1]⟩⟩:m\in r1amsgs}
1bmsgs=={⟨⟨m[1],{⟨⟨e[1],⟨⟨e[2][1],e[2][2]⟩⟩⟩⟩:e\in m[2]},m[3]⟩⟩:m\in r1bmsgs}
2amsgs=={⟨⟨m[1],m[2],⟨⟨m[3][1],m[3][2]⟩⟩⟩⟩:m\in r2amsgs}

Spec==Init/\[·][Next]_vars

MP==INSTANCE MultiPaxos

THEOREM Refinement==Spec MP!Spec
```

3 Model of Out-of-Order Executions and "Ghost Log Entries"

Compared to Raft, ParallelRaft-SE supports out-of-order commitment. However, it still requires sequential executions of user commands and is thus not suitable for highly concurrent systems. In application scenarios of PolarFS, ParallelRaft needs to support out-of-order executions. In other words, it allows state machines to execute committed log entries with a larger number first without having to wait for log entries with a smaller number to be committed or executed[7]. To satisfy state consistency despite out-of-order executions, it is necessary to ensure that commands executed out of order are conflict-free. Thus, this section first describes the out-of-order execution model used by ParallelRaft. In analyzing the correctness of the ParallelRaft protocol, we find that the description of ParallelRaft in Reference [7] ignores the "ghost log entries" phenomenon that may violate state consistency. This section analyzes the

challenges that this phenomenon poses to the out-of-order execution mechanism.

3.1 Out-of-order execution model

The out-of-order execution model of ParallelRaft provides rules for determining conflicts between user commands[7]. In the application scenarios of PolarFS, each user command includes the Logic Block Address (LBA) of the data accessed by this command. Commands with non-overlapping LBAs do not have conflicts and can be executed in an out-of-order way. In contrast, commands with overlapping LBAs need to be executed sequentially according to the log number.

In this model, we should first check if a command conflicts with a command numbered smaller before this command is executed. Since log entries are accepted out of order (and may not have been committed yet), there may be holes in the log. To determine whether the current command is executable despite that there are holes, ParallelRaft requires each log entry to record the LBAs of K (a parameter to be determined) log entries before it, called the Look Behind buFfer (LBF). Thus, it is possible to determine whether there is a conflict between any two log entries/commands as long as there is no hole with a length greater than K in the log. Otherwise, the log entries after the "hole" need to wait.

3.2 "Ghost log entries" phenomenon

After log recovery, there is no hole in the log of the leader in ParallelRaft, so it can calculate LBF for each log entry. Afterward, the leader can send LBF to followers along with log entries. Proper conflict determination requires that the LBF of each log entry accurately records the LBAs of K log entries before it. However, the "ghost log entries" phenomenon may cause the LBF calculated by the leader to be different from the actual one. Figure 2 depicts the "ghost log entries" phenomenon, which consists of three phases (Each square in the figure represents a log entry, which records the term of the log entry and the user command it carries. The log entries are numbered from 1).

(1) As shown in Figure 1(a), s_1 is the leader in phase 1. s_1 adds four log entries numbered from 1 to 4 to the log. The log entries numbered 1 and 2 are accepted by s_2 and s_3, and these two log entries are committed. The log entries numbered 3 and 4 have not been accepted by s_2 and s_3. s_1 fails.

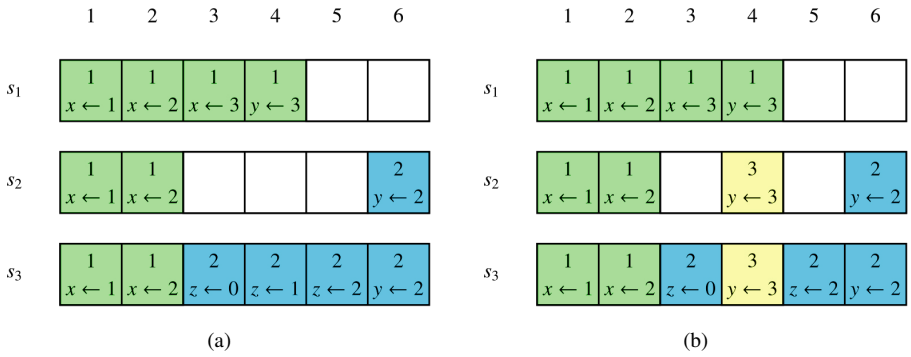

Figure 1 The "ghost log entries" phenonmenon violates consistency

(2) In phase 2, s_3 becomes the leader. s_3 does not receive uncommitted entries from s_1 during the recovery process (it should be noted that s_3 only needs to collect log entries from the majority of nodes). Thus, after the recovery is completed, the log of s_3 does not contain the uncommitted log entries of s_1. As shown in Figure 1(a), s_3 adds new log entries numbered from 3 to 6 to the log afterward and sends the log entry numbered

6 to s_2. This log entry is committed. As this entry is a modification of y ($y \leftarrow 2$), there is no conflict with previous log entries. As a result, s_3 executes $y \leftarrow 2$ (out-of-order execution). After that, s_3 also fails and the system elects a new leader s_2.

(3) As shown in Figure 1(b), s_3 receives an uncommitted log entry (numbered 4) from s_1 (s_1 is normal currently) during the recovery process in phase 3 and sends it to s_3. After this entry is committed, s_3 needs to execute it. It is also a modification of the variable y ($y \leftarrow 3$), which should be executed first according to the rules of out-of-order execution. However, s_3 has executed $y \leftarrow 2$, so an inconsistent execution order is caused.

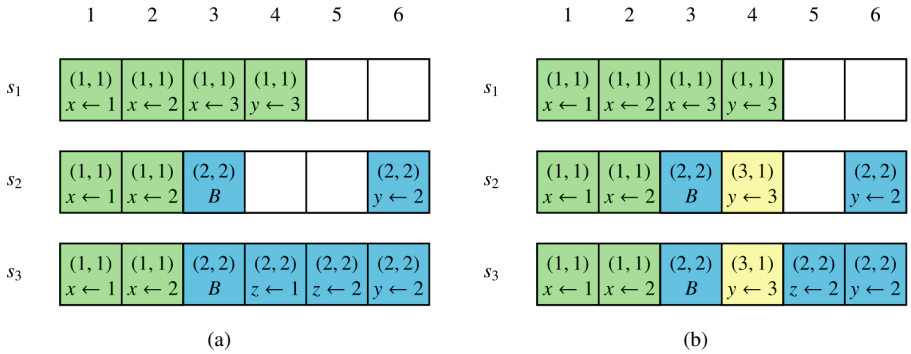

Figure 2 Passive way to address the "ghost log entries" phenomenon

The uncommitted log entries of s_1 in phase 1 are not in the log of the leader s_3 in phase 2, but then appear in the log of the leader s_2 in phase 3. We call this "ghost log entries" phenomenon. This makes the LBF calculated by s_3 not consistent with the actual one, which may lead to a wrong conflict determination. This further affects the correctness of ParallelRaft-SE (ParallelRaft) in the out-of-order execution. The key to avoiding the "ghost log entries" phenomenon is that during the log recovery process, the leader s_3 in phase 2 needs to specify the state of uncommitted log entries in s_1. They are either recommitted or deleted and would not be committed by a leader later. The "ghost log entries" phenomenon can also occur in the case of sequential executions (e.g., Multi-Paxos and Raft), but does not affect the correctness of the protocols. In sequential executions, the "ghost log entries" phenomenon is easily solved. Then we will apply the Multi-Paxos solution to ParallelRaft-SE and briefly discuss the solution in Raft to the "ghost log entries" phenomenon. We will show that these solutions cannot address the "ghost log entries" phenomenon in the mode of out-of-order executions.

3.3 Solution of Multi-Paxos

Multi-Paxos uses a passive solution. The proposer saves the generation moment of a log entry when it is added (i.e., the proposal number of the proposer generating this log entry). Nodes can detect the "ghost log entries" phenomenon by the generation moment of log entries and ignore them. In view of this, the new proposer writes a special no-op log entry, called a barrier, in the log after the recovery process. Only after the barrier log entry has been committed can the proposer add a new log entry to the log. Thus, the barrier marks the end of the log recovery and the starting point for the new proposer to add log entries. If the generation moment of a log entry afterward is smaller than that of the barrier, it is considered a "ghost log entry".

As shown in Figure 2(a), the two-tuple of each log entry records the proposal number (which may be changed later) and the generation moment of the log entry, respectively. s_1 is the proposer of the proposal numbered 1, which adds the log entries with log numbers 1–4 to the log. Their proposal numbers and generation moments are all 1. The log entries numbered

3 and 4 are not committed. After that, s_1 fails and s_3 becomes the proposer of the proposal numbered 2. After s_3 receives the log entries of s_2 and completes the recovery process, it writes a barrier (which is numbered 3 and marked with B in the figure) in the log, whose proposal number and generation moment are both 2. s_3 first sends the barrier to s_2 and commits it. Then, s_3 responses to user commands and adds log entries numbered 4–6 to the log. The log entry numbered 6 is accepted (and thus committed) by s_2. s_3 fails after executing this log entry.

s_2 becomes the proposer of the proposal numbered 3. As shown in Figure 2(b), s_2 finds the log entry numbered 4 ($y \leftarrow 3$) from the log of s_1 and adds it to its log in the recovery process. The number of this proposal is 3, but its generation moment is 1 as it is written by s_1 in the log. The barrier numbered 3 in the log of s_2 has the generation moment of 2. Thus, s_2 can determine that the log entry numbered 4 is a "ghost log entry." s_2 deletes this entry from the log, thus solving the "ghost log entries" problem.

3.4 Solution in Raft

Raft uses a proactive solution[22]. The limitation in the leader election mechanism is used, making the node with "ghost log entries" unable to become a new leader. Thus, the new leader elected in Raft adds a barrier to the log. Only after the barrier is committed can the new leader respond to user requests. According to the election rules of Raft, a node without a barrier cannot be a leader (it cannot get the majority of votes due to its old log). On the other hand, the node that receives the barrier deletes all log entries after the barrier in the synchronization phase of log entries. As a result, there are no "ghost log entries."

As shown in Figure 3(a), s_1 is the leader of the term 1, and it writes four log entries numbered 1–4 in the log. The log entries numbered 3 and 4 have not been committed yet, and s_2 becomes the leader. As shown in Figure 3(b), after s_2 becomes the leader of the term 2, it first writes a barrier to the log and commits it. Then s_2 can respond to user requests. Even if s_2 fails afterward, according to the leader election rules of Raft, s_1 needs to delete the redundant log entries (numbered 3 and 4) and add the barrier first if it wants to be the leader. Thus, the log entries numbered 3 and 4 will not appear in the log of the new leader.

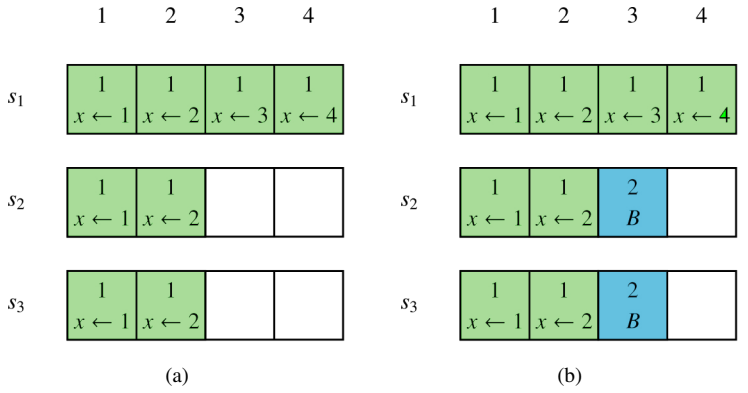

Figure 3 Proactive way to address the "ghost log entries" phenomenon

3.5 Challenge in out-of-order executions

Applying the solution of Multi-Paxos to the "ghost log entries" phenomenon directly to ParallelRaft-SE cannot guarantee the correctness of ParallelRaft-SE (or ParallelRaft) in out-of-order executions. In other words, the "ghost log entries" phenomenon is not a necessary

condition for an inconsistent state in the case of out-of-order executions. For example, the log of the leader i contains uncommitted log entries. When the leader changes, i receives the log entries of the new leader out of order. i is unaware that the uncommitted log entries in the log are out of date and thus mistakenly believes that there is no conflict, leading to inconsistent behavior.

On the other hand, Raft uses serializability to eliminate the "ghost log entries" phenomenon during the election process. However, ParallelRaft-SE supports out-of-order commitment, and there may be holes in the log. This makes it impossible to apply the solution of Raft directly to ParallelRaft-SE as well.

4 ParallelRaft-CE Protocol and Specification

To get rid of the "ghost log entries" phenomenon in out-of-order executions, we propose the ParallelRaft-CE protocol based on ParallelRaft-SE. ParallelRaft-CE avoids the "ghost log entries" phenomenon and ensures the consistency of state machines in out-of-order executions by limiting the parallelism of ParallelRaft-SE in the out-of-order commitment phase. ParallelRaft-CE mainly includes three parts: the log synchronization mechanism, the leader election mechanism, and the log recovery mechanism.

We first provide some key properties of ParallelRaft-CE. The concepts of sync number and candidate of the leader will be introduced in the following protocol description. The next section will briefly demonstrate these key properties and the correctness of ParallelRaft-CE.

(1) The term of a node increases monotonically with the sync number.
(2) Election safety: At most one leader is elected in one term.
(3) Let the term of a leader candidate be t and the sync number be s. Then $s < t$ and there are no log entries with a term greater than s in the system.
(4) Leader completeness: If the term of the leader is t, its log includes all committed log entries whose term is smaller than t.
(5) Consistency: For any two nodes, if their sync numbers are both greater than a term number t, their logs contain the same log entries with a term of t.

4.1 Log synchronization mechanism

In ParallelRaft-CE, log entries are sent and accepted out of order partially. The log entries with a same term can be sent and accepted concurrently, while log entries with different terms need to be sent and received sequentially. Thus, each node maintains a sync number, which indicates that the current node only accepts log entries with a term equal to the sync number. When a follower receives a log entry from the leader, it checks the term of the log entry. If the term of the log entry is the same as the sync number of the follower, the follower accepts this log entry and replies with an acknowledge message. Otherwise, the follower rejects this log entry and sends its own sync number to the leader. The leader also maintains the sync number of each follower. After receiving a sync number from a follower, the leader upgrades the sync number of the follower it records. After a follower acknowledges all log entries in the current term, the leader notifies the follower to set the sync number to the next term. This log synchronization mechanism limits the parallelism of the commitment phase.

4.2 Leader election mechanism

The leader election mechanism of ParallelRaft-CE is similar to that of Raft. The difference is that ParallelRaft-CE determines the generation moment of log entries by the sync number. A larger sync number of a node indicates that its log is newer. When a candidate initiates an election, it sends its own sync number to other nodes as well. The receiving node agrees this election request only when its sync number is not greater than the sync number of the candidate.

4.3 Log recovery mechanism

In ParallelRaft-CE, the log entries with the same term can be sent and accepted concurrently. Thus, there may be "holes" in the log and the new leader needs to recover the log entries of the previous term. The goal of recovery is to upgrade the sync number of the leader to its term. At this point, the system has reached a consensus on all log entries with terms smaller than the term of the leader. The leader that has not yet completed the log recovery is called a leader candidate. After the recovery, the states of all log entries with terms smaller than the term of the leader are determined: Log entries that have been committed (or executed) are still in the log of the new leader. The previously uncommitted log entries are either committed or discarded and will not be committed again.

Let the term of the leader candidate l be t and the sync number be s ($s < t$ according to the property 3). According to the property 5, the system has reached a consensus on log entries with terms smaller than s. As shown by property 3, there are no log entries with terms exceeding s in the system. Thus, l only needs to recover log entries with the term of s. The method of recovering log entries is the same as that in ParallelRaft-SE (or Multi-Paxos). l collects log entries from the majority of nodes and selects the latest log entry. As the term of a log entry is unchanged, ParallelRaft-CE adds a new variable for each log entry, date, which is equivalent to the proposal number of the log entry in Paxos. For the log entries with the same number, the one with a bigger date is newer.

l needs to synchronize the recovered log entries to followers in three phases. Let the sync number of the follower that l records be n. If $n > s$, the synchronization enters directly the phase 3. If $n = s$, the synchronization goes directly to phase 2. Otherwise, the synchronization arrives at phase 1 first.

Phase 1 ($n < s$): l first sends log entries with a term of n to the follower. When these entries are all acknowledged by the follower, l notifies the follower to upgrade to the next sync number k that satisfies the following conditions:

(1) $k > n$;
(2) The log of l contains log entries with a term of k;
(3) k is a minimum value that satisfies conditions 1 and 2.

After receiving the request of upgrading the sync number, the follower deletes all unacknowledged log entries with a term of n in the log and upgrades the sync number to k. Then, the leader candidate sends the log entries with a term of k to the follower. The above steps are repeated until the sync number of the follower is upgraded to s. The synchronization enters phase 2.

Phase 2 ($n = s$): l continues to synchronize log entries to the follower until all log entries with a term of s are acknowledged by the follower. The synchronization goes to phase 3.

Phase 3 ($n > s$): l waits until all log entries with a term of s are committed by the same majority, which is denoted as Q. Then l upgrades its sync number to its own term t (which is safe according to property 3). After l receives acknowledge from a majority of nodes (all from Q), the recovery process ends and l officially becomes the leader.

4.4 TLA+ specification of ParallelRaft-CE

ParallelRaft-CE uses the constant *LeaderCandidate* to represent the leader candidate. The variables include the followings:

- *messages* indicates the set of messages sent by nodes.
- *currentTerm*$[i]$ indicates the maximum term recorded by the node i.
- *currentState*$[i]$ indicates the state of the node i, which is one of *Leader*, *LeaderCandidate*, *Follower*, and *Candidate*.

- *votedFor* indicates that each node can only vote for one candidate in a term. *votedFor*$[i]$ indicates the candidate that i votes for in the current term (*currentTerm*$[i]$). If i does not vote for any node, *votedFor*$[i] =$ *Nil*.
- *sync*$[i]$ indicates the sync number of the node i.
- *end*$[i][t]$ indicates the maximum number of acceptable log entries recorded by the node i with a term of t. ParallelRaft-CE uses the voting mechanism of Paxos to confirm the maximum number of each term.
- $\log[i]$ represents the log of the node i. $\log[i][k]$ indicates the log entry numbered k in the log of i. Each log entry is a four-tuples of $< t, d, v, b >$, where t is the term of a log entry, which cannot be modified; d is the date, which is used to determine the generation moment of a log entry; v is the proposal value; b is a Boolean value, which is true when and only when a log entry is committed.
- *syncTrack* indicates the sync number of a node in a state of *Leader* or *LeaderCandidate* for recording other nodes. *syncTrack*$[i][a]$ indicates the sync number of the node a recorded by the node i.
- *elections* and *halfElections* are historical variables, which record the election information.

```
---------------------------- MODULE ParallelRaft-CE ----------------------------
CONSTANTS Server,Follower,Candidate,Leader,LeaderCandidate,Nil,Value,
          RequestVoteRequest,RequestVoteResponse,
          RequestCommitRequest,RequestCommitResponse,
          RequestSyncRequest,RequestSyncResponse,
          UpdateSyncRequest,UpdateSyncResponse

Quorums=={i\in SUBSET (Server):Cardinality(i)*2>Cardinality(Server)}

VARIABLE messages,currentTerm,currentState,votedFor,sync,end,log,syncTrack

serverVars==⟨⟨currentTerm,currentState,votedFor,sync,end⟩⟩

VARIABLE halfElections,elections
electionVars==⟨⟨halfElections,elections⟩⟩

vars==⟨⟨messages,log,serverVars,syncTrack,electionVars⟩⟩
```

The major actions of ParallelRaft-CE include the followings:

- *UpdateTerm*(i): s receives a message m. If $m.mterm >$ *currentTerm*$[i]$, i upgrades the term (*currentTerm*$'[i] = m.mterm$) and changes into a follower (*currentState*$'[i] =$ *Follower*). The voted record is emptied (*votedFor*$'[i] =$ *Nil*) as i has not voted for any node in the new term yet. In ParallelRaft-CE, a node needs to check the term for any request it receives. When the term of the message is larger than that of the node, the node executes *UpdateTerm* and then responds to the request. When the message has the same term as the node, the node processes the request directly. When the term of the message is smaller than that of the node, the node rejects any request and replies to the sender with its own term.
- *RequestVote*(i): i initiates the election. Unlike ParallelRaft-SE (Multi-Paxos), i needs to send its sync number (*sync*$[i]$) to other nodes.
- *HandleRequestVoteRequest*(i): The node i that receives the election request m compares the terms (*currentTerm*$[i]$ and $m.mterm$) and the sync numbers (*sync*$[i]$ and $m.msync$). When $m.mterm =$ *currentTerm*$[i]$ and *sync*$[i] \geq m.msync$ and i has not agreed to the election request of other nodes in this term, i agrees to the election request and sends the log entries with a term of $m.msync$ in its log and the maximum number (*end*$[i][m.msync]$) of log entries with a term of $m.msync$ that i can accept to the election initiator.

```
UpdateTerm(i)==
  /\\E m\in messages:
    /\m.mterm>currentTerm[i]
    /\\/m.mdest=i
      \/m.mdest=Nil
    /\currentTerm'=[currentTerm EXCEPT ![i]=m.mterm]
    /\currentState'=[currentState EXCEPT ![i]=Follower]
    /\votedFor'=[votedFor EXCEPT ![i]=Nil]
  /\UNCHANGED ⟨⟨messages,sync,log,syncTrack,electionVars,end⟩⟩

RequestVote(i)==
  /\currentState[i]=Candidate
  /\Send([mtype ↦ RequestVoteRequest,
          mterm ↦ currentTerm[i],
          msync ↦ sync[i],
          msource ↦ i,
          mdest ↦ Nil])
  /\UNCHANGED ⟨⟨serverVars,syncTrack,log,electionVars⟩⟩

HandleRequestVoteRequest(i)==
  /\\E m\in messages:
    LET j==m.msource
        syncOK==/\m.msync\geq sync[i]
        grant==/\syncOK
            /\votedFor[i]\in {Nil,j}
            /\currentTerm[i]=m.mterm
    IN
      /\m.mterm\leq currentTerm[i]
      /\m.mtype=RequestVoteRequest
      /\\/grant/\votedFor'=[votedFor EXCEPT ![i]=j]
        \/\neg grant/\UNCHANGED votedFor
      /\Send([mtype ↦ RequestVoteResponse,
              mterm ↦ currentTerm[i],
              mvoteGranted ↦ grant,
              mlog ↦ LET C=={n\in Index:log[i][n].term=sync[i]}
                  IN {⟨⟨n,log[i][n]⟩⟩:n\in C},
              mend ↦ end[i][m.msync],
              msource ↦ i,
              mdest ↦ j])
  /\UNCHANGED ⟨⟨currentTerm,currentState,sync,log,syncTrack,
              electionVars,end⟩⟩
```

- *BecomeLeaderCandidate*(i): If a candidate i receives a message of agreeing to the election from the node of a majority (*voteGranted*), i becomes a leader candidate. i will recover the log entries whose terms are equal to its sync number (*sync*$[i]$). Thus, i should first determine the maximum number of log entries with a term of *sync*$[i]$ that it can accept and save it in *end*$[i][$*sync*$[i]]$. The method is described as herein. i selects the latest result in the m.*mend* of the received messages. Then for each acceptable number, i selects the latest one among the received log entries with the corresponding number and writes it in the log. For each node p, i initiates *syncTrack*$[i][p]$ = *sync*$[i]$.

```
Merge(entries,term,date)==
  IF entries={·} THEN [term ↦ term,
                       date ↦ date,
                       value ↦ Nil,
                       committed ↦ FALSE]
  ELSE
    LET
      committed=={e\in entries:e.committed=TRUE}
      chosen==
        CASE committed={·} CHOOSE x\in entries:
                  \A y\in entries:x.date\geq y.date
        [·] committed/={·} CHOOSE x\in committed:TRUE
    IN
      [term ↦ chosen.term,
```

```
        date ↦ date,
        value ↦ chosen.value,
        committed ↦ chosen.committed]

BecomeLeaderCandidate(i)==
  /\currentState[i]=Candidate
  /\\E P\in Quorums:
    LET voteGranted=={m\in messages:/\m.mtype=RequestVoteResponse
                    /\m.mdest=i
                    /\m.msource\in P
                    /\m.mterm=currentTerm[i]
                    /\m.mvoteGranted=TRUE}
    allLog==UNION {m.mlog:m\in voteGranted}
    endLine==LET allPoint=={m.mend:m\in voteResponded}
            IN e==CHOOSE e1\in allPoint:
                      (\A e2\in allPoint:e1[1]\geq e2[1])
    toRecover=={n\in 0..endLine:log[i][n].committed=FALSE}
    toSync=={⟨⟨n,Merge({l[2]:l\in {t\in allLog:t[1]=n}},sync[i],currentTerm[i])⟩⟩:
     n\in toRecover}
    IN
    /\\A p\in P:\E m\in voteGranted:m.msource=p
    /\log'=[log EXCEPT ![i]=[n\in Index ↦ IF n\in toRecover THEN
                                              (CHOOSE e\in toSync:e[1]=n)[2]
                                                  ELSE log[i][n]]]
    /\end'=[end EXCEPT ![i][sync[i]]=⟨⟨currentTerm[i],end⟩⟩]
  /\currentState'=[currentState EXCEPT ![i]=LeaderCandidate]
  /\syncTrack'=[syncTrack EXCEPT ![i]=[j\in Server ↦ sync[i]]]
  /\UNCHANGED ⟨⟨messages,currentTerm,votedFor,sync,elections⟩⟩
```

- *RequestSync*(i): When i is a leader or a leader candidate, i sends log entries with a term of *syncTrack*$[i][p]$ to a node p. These log entries can be sent and accepted out of order.
- *HandleRequestSyncRequest*(i): The node i receives a synchronization request m. When $m.msync < sync[i]$ or $m.msync > sync[i]$, i rejects the request and replies with the *RequestSyncResponse* message, so as to inform the sender of $sync[i]$. When $m.msync = sync[i]$, i agrees to the request, replies with an acknowledgement, and modifies $end[i]$ and $\log[i]$ according to $m.mend$ (the maximum number of acceptable log entries with a term of $m.msync$) and $m.mentries$ (log entries with a term of $m.msync$). The modification of the log is as follows. The log entries with a number greater than $m.mend$ in the log are deleted, and the log entries with a number not larger than $m.mend$ are replaced with the corresponding entries in $m.mentries$.

```
RequestSync(i)==
  /\currentState[i]\in {LeaderCandidate,Leader}
  /\\E s\in 0..sync[i]:
    LET start==Min({n\in Index:log[i][n].term=s})
        end==Max({n\in Index:log[i][n].term=s})
    IN
      /\Send([mtype ↦ RequestSyncRequest,
              mterm ↦ currentTerm[i],msync ↦ s,
              mstart ↦ start,mend ↦ end,
              mentries ↦ IF start=-1 THEN Nil ELSE
                                  [n\in start..end ↦ log[i][n]],
              msource ↦ i,mdest ↦ Nil])
  /\UNCHANGED ⟨⟨serverVars,logVars,electionVars,syncTrack⟩⟩

HandleRequestSyncRequest(i)==
  /\\E m\in messages:
    LET j==m.msource
      grant==/\m.mterm=currentTerm[i]
             /\m.msync=sync[i]
    IN
      /\m.mtype=RequestSyncRequest
      /\m.mterm\leq currentTerm[i]
      /\j/=i
      /\\//\grant
```

```
        /\log'=[log EXCEPT ![i]=[n\in Index ↦
          IF n<m.mstart THEN log[i][n]
          ELSE IF n\in m.mstart..m.mend
            THEN m.mentries[n]
          ELSE [term ↦ -1,date ↦ -1,
                value ↦ Nil,committed ↦ FALSE]]]
      /\endPoint'=[endPoint EXCEPT ![i][sync[i]]=⟨currentTerm[i],m.mend⟩]
        \//\\neg grant
          /\UNCHANGED ⟨log,endPoint⟩
      /\Send([mtype ↦ RequestSyncResponse,mterm ↦ currentTerm[i],
              msyncGranted ↦ grant,msync ↦ sync[i],
              mstart ↦ m.mstart,mend ↦ m.mend,
              msource ↦ i,mdest ↦ j])
  /\UNCHANGED ⟨currentTerm,currentState,sync,votedFor
```

- *HandleRequestSyncResponse*(i): i receives a *RequestSyncResponse* message m from the node p when i is a leader or a leader candidate. If m is an acknowledge message ($m.msyncGranted$ = TRUE), and the sync number of p is smaller than that of i ($m.msync < sync[i]$), i sends an *UpdateSyncRequest* request to p to notify p to upgrade to the next sync number. The next sync number is elected in the same way as described above. If m is a rejection message, then i modifies $syncTrack[i][p]$ ($syncTrack'[i][p] = m.msync$).
- *UpdateSync*(i): When i is a leader candidate, if there exists a node majority that finishes the synchronization for log entries with a term of $sync[i]$, and i receives their acknowledgements (i.e., the *RequestSyncResponse* message), then i sends an *UpdateSyncRequest* message to these nodes to notify them to upgrade the term with a sync number of i ($currentTerm[i]$).

```
HandleRequestSyncResponse(i)==
  /\\E m\in messages:
    LET j==m.msource IN
    /\m.mtype=RequestSyncResponse
    /\m.mdest=i
    /\currentTerm[i]=m.mterm
    /\currentState[i]\in {Leader,LeaderCandidate}
    /\syncTrack'=[syncTrack EXCEPT ![i][j]=m.msync]
    /\\//\m.msyncGranted
      /\m.msync<sync[i]
      /\Send([mtype ↦ UpdateSyncRequest,
              mterm ↦ currentTerm[i],
              msync ↦ Min({sync[i]} \union {k\in Nat:k>m.msync/\
                    Cardinality({n\in Index:log[i][n].term=k})>0}),
              msource ↦ i,
              mdest ↦ {j}])
      \//\\neg m.msyncGranted
        /\UNCHANGED messages
  /\UNCHANGED ⟨serverVars,log,electionVars⟩

UpdateSync(i)==
  /\currentState[i]=LeaderCandidate
  /\\E Q\in Quorums:
    LET syncUpdated=={m\in messages:/\m.mtype=RequestSyncResponse
                                    /\m.mterm=currentTerm[i]
                                    /\m.msyncGranted=TRUE
                                    /\m.msync=sync[i]
                                    /\m.msource\in Q
                                    /\m.mdest=i}
    IN
    /\\A q\in Q:(\E m\in syncUpdated:m.msource=q)\/q=i
    /\Send([mtype ↦ UpdateSyncRequest,
            mterm ↦ currentTerm[i],
            msync ↦ currentTerm[i],
            msource ↦ i,
            mdest ↦ Q])
  /\UNCHANGED ⟨serverVars,log,syncTrack,electionVars⟩
```

- *HandleUpdateSyncRequest*(i): i upgrades its sync number ($sync'[i] = m.msync$) after receiving the *UpdateSyncRequest* request m and marks all the log entries in the log as committed. Then i replies with acknowledgements to notify the sender of the upgraded sync number. The leader (or leader candidate) that receives the acknowledgement from i modifies the record of the sync number of i (*syncTrack*) according to the reply.
- *HandleUpdateSyncResponse*(i): After the leader or leader candidate i receives the *UpdateSyncResponse*, it updates the record of the sync number of the sender.

```
HandleUpdateSyncRequest(i)==
  \E m\in messages:
    LET grant==/\currentTerm[i]=m.mterm
               /\m.msync>sync[i]
      j==m.msource
    IN
    /\m.mtype=UpdateSyncRequest
    /\i \in m.mdest
    /\m.mterm\leq currentTerm[i]
    /\\//\grant
      /\sync'=[sync EXCEPT ![i]=m.msync]
      /\log'=[log EXCEPT ![i]=[n\in Index ↦
        IF log[i][n].term=sync[i] THEN
          log[i][n].committed=TRUE
        ELSE log[i][n]]]
      \//\\neg grant
      /\UNCHANGED ⟨⟨log,sync⟩⟩
    /\Send([mtype ↦ UpdateSyncResponse,mterm ↦ currentTerm[i],
            mupdateSyncGranted ↦ grant,msync ↦ sync'[i],
            msource ↦ i,mdest ↦ j])
  /\UNCHANGED ⟨⟨currentTerm,currentState,votedFor,end,syncTrack,electionVars⟩⟩

HandleUpdateSyncResponse(i)==
  /\\E m\in messages:
  LET j==m.msource IN
  /\m.mtype=UpdateSyncResponse
  /\m.mdest=i
  /\currentTerm[i]=m.mterm
  /\currentState[i]\in {Leader,LeaderCandidate}
  /\\//\m.mupdateSyncGranted
    /\syncTrack'=[syncTrack EXCEPT ![i][j]=m.msync]
  \//\\neg m.mupdateSyncGranted
    /\UNCHANGED syncTrack
/\UNCHANGED ⟨⟨messages,serverVars,log,electionVars⟩⟩
```

- *BecomeLeader*(i): After the leader candidate i executes *UpdateSync*, if the sync numbers of nodes in a majority Q are upgraded to *currentTerm*$[i]$ and i receives their acknowledgements ($\forall q \in Q : syncTrack[i][q] = currentTerm[i]$), then i becomes a leader and commits all log entries in the log.
- *ClientRequest*(i): When i is a leader, it can respond to user requests and insert them to the log.

```
BecomeLeader(i)==
  /\currentState[i]=LeaderCandidate
  /\\E Q\in Quorums:\A q\in Q:(q=i\/syncTrack[i][q]=currentTerm[i])
                      /\elections'=elections\union
                      {[eterm ↦ currentTerm[i],
                        esync ↦ sync[i],
                        eleader ↦ i,
                        evotes ↦ Q,
                        evoterLog ↦ {log[k]:k\in Q},
                        elog ↦ log[i]]}
  /\sync'=[sync EXCEPT ![i]=currentTerm[i]]
  /\currentState'=[currentState EXCEPT ![i]=Leader]
  /\log'=[log EXCEPT ![i]=[n\in Index ↦
                                IF log[i][n].term=sync[i] THEN
```

```
                                    log[i][n].committed ↦ TRUE]
                                  ELSE log[i][n]]]
  /\UNCHANGED ⟨⟨messages,currentTerm,votedFor,end,syncTrack,halfElections⟩⟩

ClientRequest(i,v)==
  LET nextIndex==logTail(log[i])+1
    entry==[term ↦ currentTerm[i],
            date ↦ currentTerm[i],
            value ↦ v,
            committed ↦ FALSE]
  IN
    /\currentState[i]=Leader
    /\log'=[log EXCEPT ![i][nextIndex]=entry]
    /\UNCHANGED ⟨⟨messages,serverVars,electionVars,syncTrack⟩⟩
```

Next defines the substate relationship. *Spec* defines the complete behavior specification.

```
Next==\/\E i\in Server:Restart(i)
  \/\E i\in Server:Timeout(i)
  \/\E i\in Server:UpdateTerm(i)
  \/\E i\in Server:RequestVote(i)
  \/\E i\in Server:HandleRequestVoteRequest(i)
  \/\E i\in Server:BecomeLeaderCandidate(i)
  \/\E i\in Server:BecomeLeader(i)
  \/\E i\in Server,v\in Value:ClientRequest(i,v)
  \/\E i,j\in Server:RequestSync(i)
  \/\E i\in Server:HandleRequestSyncRequest(i)
  \/\E i\in Server:HandleRequestSyncResponse(i)
  \/\E i,j\in Server:UpdateSync(i)
  \/\E i\in Server:HandleUpdateSyncRequest(i)
  \/\E i\in Server:HandleUpdateSyncResponse(i)

Spec==Init/\[·][Next]_vars
```

5 Proof of Correctness of ParallelRaft-CE

This section briefly demonstrates the correctness of the ParallelRaft-CE protocol. The correctness of ParallelRaft-CE consists of two parts: the consistency of the consensus protocol and the absence of the "ghost log entries" phenomenon in ParallelRaft-CE. The security of the replicated state machines can be obtained: There is no conflict between nodes in the order of executing each log entry, and the state machines have the consistent state.

Property 1. The term and sync number of a node increase monotonically.

According to the specification, it is easy to find that the term and sync number of a node are increasing monotonically.

According to the specification, a node will only modify log entries whose terms are equal to its sync numbers, and will not add, delete, or modify log entries with terms smaller than its sync number.

Property 2 (Election security)**.** At most one leader is elected in a term.

In ParallelRaft-CE, there are two stages for a follower to become a leader. A follower first becomes a leader candidate by election, and then the leader candidate becomes a leader. The leader candidate is selected in the same way as the leader is selected in Raft. According to the security of leader election in Raft, at most one leader candidate is elected for each term in ParallelRaft-CE.

A leader candidate may successfully become a leader, or it may not become a leader due to failure or an election initiated by a node with a larger term. In this case, the system proceeds to the next term. Therefore, at most one leader is elected in a term.

Log entries with the same term and the same number must contain the same command. According to the election security, a leader can obtain any number by adding one log entry at most. This property is also shown in Raft.

Property 3. Let the term of the leader candidate be t and the sync number be s. Then $s < t$ and there is no log entry with a term greater than s in the system.

First, the sync number of any node is not greater than its term. Since the node needs to increase the term (the sync number remains unchanged) when initiating an election and the term and sync number remain unchanged during the election process, the sync number of the leader candidate is smaller than its term. Let the sync number of leader candidate i be s and the term be t $(s < t)$.

For any $s < k < t$, there is no leader with a term of k. The proof by contradiction is adopted here. It is assumed that there is a leader j with a term of k. According to the specification, there is a node majority Q_1 voting for i. j becomes a leader from a leader candidate, and thus there is a node majority Q_2 upgrading its sync number to k. According to the property of the majority, $Q_1 \cap Q_2 \neq \varnothing$. Let $r \in Q_1 \cap Q_2$. If r votes for i first, then r upgrades its term to t. As $t > k$, r will reject the request of upgrading the sync number of j. This contradicts with $r \in Q_2$. If r upgrades the sync number to k first, then since $k > s$, r rejects the election request from i (note that sync numbers are compared in voting for election), which contradicts with $r \in Q_1$. Thus, there is no such k.

For $k \geq t$, there is no leader with a term of k. If there exists a leader with a term of k in the system, then k receives votes from the node majority Q in the election, so the term of nodes in Q is k at least. Since nodes have an increasing term, the nodes in Q when i initiates an election either have a term greater than t or have already voted for other nodes with a term of t and therefore will not vote for i. This contradicts with the fact that i becomes a leader candidate with a term of t.

In summary, there has not been a leader with a term greater than s in the system. Thus, there is no log entry with a term greater than s.

Property 4 (Leader completeness)**.** If the term of the leader is t, then its log includes all log entries with terms smaller than t that have been committed.

In ParallelRaft-CE, when the leader with a term of t_1 fails, the new leader candidate (with a term of t_2) reconfirms the log entries with a term of t_1. During the reconfirmation process, the states of all log entries in the previous term are determined: committed or discarded. After the reconfirmation process, the leader candidate becomes the new leader. The reconfirmation is essentially to execute the Paxos protocol for each location, so that consensus is guaranteed and the log entries that consensus has been reached on are not lost. According to the specification, when reconfirmation is completed, a node majority upgrades the sync number to t_2. In addition, since the sync number of the node increases, nodes with sync numbers smaller than t_2 cannot become a leader candidate through election. Therefore, the log entries with a term of t_1 that consensus has been achieved on will not be changed.

Property 5 (Consistency)**.** For any two nodes, if their sync numbers are both greater than a term t, their logs contain the same log entries with a term of t.

The leader of ParallelRaft-CE synchronizes the log entries according to the sync number of the follower. This, together with the correctness of the recovery phase (Property 3 and Property 4), guarantees the log matching property of nodes.

There is no "ghost log entries" phenomenon in ParallelRaft-CE.

In ParallelRaft-CE, the new leader candidate reconfirms the log entries of the previous leader. After the reconfirmation, the sync number of the leader candidate is upgraded to its

term. Since the sync number of the node increases, the election mechanism ensures that a node with a small sync number cannot become a leader. Therefore, the system does not recommit the log entries of the previous term.

6 Model Checking and Simulation Testing

This section uses the model checker TLC to verify the correctness of the ParallelRaft-CE protocol in the model checking mode and the simulation mode and to verify the refinement relationship between ParallelRaft-SE and Multi-Paxos.

6.1 Experimental setup

In all experiments, we resize the participant set *Proposer* (*Server*), the proposer set *Value*, and the proposal number set *Ballot* (*Term*), and set the first two as symmetric sets[13] to improve the validation efficiency of TLC. We use 10 threads for our experiments, and the experimental results are shown in the followings.

The statistical results of the model checking mode include the diameter of the traversed (traversed in the BFS way) system state diagram, the number of traversed states, the number of discovered different states, and the time spent for checking (in hh:mm:ss). In verifying the refinement from ParallelRaft-SE to Multi-Paxos, we set the experiment to artificially stop when the number of different states examined by TLC exceeds 100 million. When verifying that ParallelRaft-CE satisfies consistency, we set the experiment to artificially stop when the number of different states examined by TLC exceeds 200 million.

In the simulation mode, we use random seeds and set the maximum depth to 50. The results are mainly the number of states that have been traversed. Each set of experiments was run for six hours.

The experiments were performed on a 2.40 GHz 10-core CPU with 64 GB of RAM, and TLC 1.7.0 is adopted.

6.2 Verification results of model checking

6.2.1 *Refinement from ParallelRaft-SE to Multi-Paxos*

Table 2 presents the verification results of the refinement from ParallelRaft-SE to Multi-Paxos in the model checking mode and different configurations.

Table 2 Model checking results of verifying the refinement from ParallelRaft-SE to Multi-Paxos

TLC model (# of states, # of proposal values, rounds of voting)	Diameter of state diagram	# of states	# of different states	Checking time (hh:mm:ss)
(3, 2, 2)	22	1,183,766,512	104,836,664	10:05:25
(3, 2, 3)	23	899,846,293	102,806,000	09:54:49
(3, 3, 2)	22	950,017,774	100,064,549	12:49:10
(3, 3, 3)	22	828,085,252	100,020,363	12:40:07
(4, 2, 2)	24	1,045,345,827	100,093,827	12:52:38
(4, 2, 3)	22	14,830,223,804	100,044,759	09:42:07
(4, 3, 2)	24	1,150,819,236	100,101,884	23:21:09
(4, 3, 3)	21	1,452,458,565	100,054,689	20:04:08

The refinement relationship is given in the TLA+ specification of ParallelRaft-SE. Experimental data show that the number of participants and the number of proposal values have a great impact on the experimental scale and the checking time, while the number of voting rounds has a small impact on the experimental scale.

6.2.2 *ParallelRaft-CE satisfying consistency*

Table 3 presents the results of using the model checking mode to verify that ParallelRaft-CE satisfies consistency in different configurations.

Table 3 Model checking results of verifying that ParallelRaft-CE satisfies consistency

TLC model (# of states, # of proposal values, rounds of voting)	Diameter of state diagram	# of states	# of different states	Checking time (hh:mm:ss)
(2, 2, 2)	26	1,274,709,286	201,161,468	01:51:02
(2, 2, 3)	24	1,104,433,959	202,169,934	01:31:11
(2, 3, 2)	26	1,218,607,009	200,208,445	02:27:13
(2, 3, 3)	24	1,108,870,367	200,916,171	02:22:13
(3, 2, 2)	22	1,370,489,381	200,340,341	03:02:17
(3, 2, 3)	19	1,221,006,004	200,913,649	02:40:23
(3, 3, 2)	22	1,369,865,234	200,265,437	05:44:01
(3, 3, 3)	19	1,236,091,812	200,041,489	05:02:33

The consistency of ParallelRaft-CE is defined formally as follows:

```
AllEntries(i)=={⟨⟨n,log[i][n]⟩⟩:n\in Index}
Consistency==\A i,j\in Server:
  {e \in AllEntries(i):e[2].term\geq 0/\e[2].term<Min({sync[i],sync[j]})}=
  {e\in AllEntries(j):e[2].term\geq 0/\e[2].term<Min({sync[i],sync[j]})}
```

In the above, *AllEntries*(i) indicates all log entries of the node i (two-tuples consisting of log number and log). *Consistency* defines the consistency of ParallelRaft-CE. Specifically, if the sync numbers of any two nodes are both greater than t, then their logs include the same log entries with a term of t.

The ParallelRaft-CE protocol is more complex and includes many states. Thus, we set the experiment to stop when the number of different states reaches 200 million.

6.3 Verification results of simulation mode

6.3.1 *Refinement from ParallelRaft-SE to Multi-Paxos*

Table 4 shows the experimental results of using the simulation mode to verify the refinement from ParallelRaft-SE to Multi-Paxos under different configurations. After estimation, the diameter of the state diagram is around 30 on a small experimental scale (three participants and two rounds of voting). The diameter of the state diagram is about 50 on a large experimental scale (four participants and three rounds of voting). In the experiment, the detection depth in the simulation mode is set to 50.

Table 4 Simulation results of the refinement from ParallelRaft-SE to Multi-Paxos

TLC model (# of states, # of proposal values, rounds of voting)	# of states	Checking time (hh:mm:ss)
(3, 2, 2)	3,203,570,873	06:00:01
(3, 2, 3)	2,723,736,314	06:00:00
(3, 3, 2)	3,137,465,261	06:00:00
(3, 3, 3)	2,670,655,875	06:00:01
(4, 2, 2)	2,911,602,481	06:00:00
(4, 2, 3)	2,414,873,632	06:00:00
(4, 3, 2)	2,976,053,370	06:00:00
(4, 3, 3)	2,298,047,612	06:00:01

6.3.2 *ParallelRaft-CE satisfying consistency*

Table 5 shows the experimental results of using the simulation mode to verify that ParallelRaft-CE satisfies consistency under different configurations. The maximum depth is also set to 50 in the experiment.

Table 5 Simulation results of the consistency of ParallelRaft-CE

TLC model (# of states, # of proposal values, rounds of voting)	# of states	Checking time (hh:mm:ss)
(2, 2, 2)	4,521,671,984	06:00:17
(2, 2, 3)	4,209,786,847	06:00:20
(2, 3, 2)	4,480,532,710	06:01:05
(2, 3, 3)	4,091,547,805	06:00:12
(3, 2, 2)	3,181,834,406	06:02:14
(3, 2, 3)	2,722,329,411	06:00:10
(3, 3, 2)	3,116,899,023	06:00:45
(3, 3, 3)	2,690,629,148	06:04:19

7 Related Work

The classical distributed consensus protocol Multi-Paxos (Paxos)[8, 9] has derived several variants[23, 24], such as Disk Paxos[25], Cheap Paxos[26], Fast Paxos[27], Generalized Paxos[28], Stoppable Paxos[29], Vertical Paxos[30], Byzantine Paxos[31], EPaxos[32], and TPaxos[18]. Multi-Paxos supports out-of-order commitment and sequential executions of user commands. Raft[10] can also be seen as a variant of Multi-Paxos (Paxos)[33], which prohibits out-of-order commitment and simplifies the protocol design by sequential commitment and sequential executions of user commands. In this paper, we focus on the ParallelRaft protocol[7] used by the distributed file system PolarFS. Realizing out-of-order commitment and out-of-order executions based on Raft, it is more suitable for highly concurrent systems. To clarify the relationship between ParallelRaft and Raft, we propose ParallelRaft-SE. It is a variant of Multi-Paxos that supports out-of-order commitment and sequential executions of user commands.

The performance of consensus protocols can be improved utilizing the exchangeability between conflict-free commands. In Generalized Paxos[28], if commands are conflict-free, users can bypass the leader and broadcast the commands directly to all nodes, reducing communication overhead and improving performance. Tribble[34], a distributed replication framework proposed by Microsoft, uses multi-threading techniques to concurrently execute conflict-free commands while ensuring state consistency. ParallelRaft[7] makes conflict judgments based on the LBAs of commands, and conflict-free commands can be executed out of order. We analyze the out-of-order execution mechanism of ParallelRaft and find that it ignores the "ghost log entries" phenomenon that may violate state consistency according to the description in Reference [7]. Therefore, we propose ParallelRaft-CE, which avoids the "ghost log entries" phenomenon by limiting the parallelism in the out-of-order commitment phase.

Describing distributed protocols using formal specification languages and verifying expected properties using model checkers can be effective in increasing confidence in the reliability of the protocols[18]. Lamport used TLA+[13, 15] to describe protocols such as Paxos[35], Fast Paxos[27], and Byzantine Paxos[31] and used the model checker TLC to verify the correctness of these protocols (on a restricted scale). Ongaro *et al*. proposed a variant of Paxos, Raft, and gave a TLA+ specification for Raft[22]. In this paper, we provide TLA+ specifications for ParallelRaft-SE and ParallelRaft-CE and verify their correctness (on a restricted scale).

Refinement techniques[12] help to understand the relationships between various protocols. In studying Paxos, Lamport proposed Consensus, an abstract description of the consensus

problem, provided Voting, a centralized solution to the distributed consensus problem[35], and constructed refinement relationships from Paxos to Voting and from Voting to Consensus[35]. Yi *et al*. proposed a variant of Voting, EagerVoting, in studying TPaxos of the Tencent's PaxosStore system and constructed refinement relationships from TPaxos to EagerVoting and from EagerVoting to Consensus[18]. In this paper, we construct a refinement relationship from ParallelRaft-SE to Multi-Paxos and prove the correctness of ParallelRaft-SE.

8 Summary and Future Work

This paper aimed to provide a rigorous formal specification for the ParallelRaft protocol[7] (which supports out-of-order commitment and out-of-order executions of user commands) used in the file system PolarFS and prove its correctness. First, to clarify the relationship between ParallelRaft and Raft, we proposed the ParallRaft-SE protocol that allows out-of-order commitment and sequential executions and establish a refinement relationship between ParallelRaft-SE and Multi-Paxos. Second, we found that ParallelRaft ignores the "ghost log entries" phenomenon that may violate state consistency according to its description and proposed the ParallelRaft-CE protocol. By limiting the parallelism of ParallelRaft-SE in the out-of-order commitment phase, ParallelRaft-CE avoids the "ghost log entries" phenomenon. Finally, we provided TLA+ specifications for ParallelRaft-SE and ParallelRaft-CE and verified the refinement relationship between ParallelRaft-SE and Multi-Paxos and the correctness of ParallelRaft-CE using the model checker TLC in the case of a small number of protocol participants.

Currently, we are using TLA+ proof system (TLAPS)[14, 16] to develop a mechanized correctness proof method for ParallelRaft-CE (and Raft). In addition, we will make a review on known distributed consensus protocols in terms of sequential/out-of-order commitment and sequential/out-of-order executions and study the relationships between them using formal methods.

References

[1] Fischer MJ, Lynch NA, Paterson MS. Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM), 1985, 32(2): 374–382. [doi: https://doi.org/10.1145/3149.214121]

[2] Herlihy M. Wait-Free synchronization. ACM Trans. on Programming Languages and Systems (TOPLAS), 1991, 13(1): 124–149. [doi: https://doi.org/10.1145/114005.102808]

[3] Weil SA. Ceph: Reliable, scalable, and high-performance distributed storage [Ph.D. Thesis]. Santa Cruz: University of California, 2007.

[4] Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman JJ, Ghemawat S, Gubarev A, Heiser C, Hochschild P, Hsieh W. Spanner: Google's globally distributed database. ACM Trans. on Computer Systems (TOCS), 2013, 31(3): 1–22.

[5] https://www.mysql.com/.

[6] Zheng J, Lin Q, Xu J,*et al*. PaxosStore: High-availability storage made practical in WeChat. Proc. of the VLDB Endowment, 2017, 10(12): 1730–1741.

[7] Cao W, Liu Z, Wang P, *et al*. PolarFS: An ultra-low latency and failure resilient distributed file system for shared storage cloud database. Proc. of the VLDB Endowment, 2018, 11(12): 1849–1862.

[8] Lamport L. Paxos made simple. ACM Sigact News, 2001, 32(4): 18–25.

[9] Lamport L. The part-time parliament. ACM Trans. on Computer Systems (TOCS), 1998, 16(2): 133–169.

[10] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. Proc. of the 2014 USENIX Annual Technical Conf. (USENIXATC 2014). 2014. 305–319.

[11] Schneider, Fred B. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys, 1990, 22(4): 299–319.

[12] Abadi M, Lamport L. The existence of refinement mappings. Theoretical Computer Science, 1991, 82(2): 253–284.

[13] Lamport L. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Boston: Addison-Wesley Longman Publishing Co., Inc., 2002.

[14] Lamport L. The TLA+ hyperbook. 2015. http://research.microsoft.com/enus/um/people/lamport/tla/hyperbook.html

[15] Lamport L. The temporal logic of actions. ACM Trans. on Programming Languages and Systems (TOPLAS), 1994, 16(3): 872–923.

[16] Yu Y, Manolios P, Lamport L. Model checking TLA+ specifications. Proc. of the Advanced Research Working Conf. on Correct Hardware Design and Verification Methods. Berlin, Heidelberg: Springer-Verlag, 1999. 54–66.

[17] https://github.com/HappyCS-Gu/Parallel-Raft-tla.

[18] Yi XC, Wei HF, Huang Y, *et al*. TPaxos consensus protocol in PaxosStore: Derivation, specification, and refinement. Ruan Jian Xue Bao/Journal of Software, 2020, 31(8): 2336–2361. http://www.jos.org.cn/1000-9825/5964.htm [doi: 10.13328/j.cnki.jos.005964]

[19] Ji Y, Wei HF, Huang Y, Lü J. Specifying and verifying CRDT protocols using TLA+. Ruan Jian Xue Bao/Journal of Software, 2020, 31(5): 1332–1352 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5956.htm [doi: 10.13328/j.cnki.jos.005956]

[20] Lamport L. Summary of TLA+. http://lamport.azurewebsites.net/tla/summary-standalone.pdf

[21] Yuan D, Luo Y, Zhuang X, *et al*. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. Proc. of the 11th USENIX Symp. on Operating Systems Design and Implementation (OSDI). 2014. 249–265.

[22] Ongaro D. Consensus: Bridging theory and practice [Ph.D. Thesis]. Stanford University, 2014.

[23] Lampson B. The ABCD's of Paxos. Proc. of the 20th Annual ACM Symp. on Principles of Distributed Computing (PODC), Vol.1. 2001. 13.

[24] van Renesse R, Altinbuken D. Paxos made moderately complex. ACM Computing Surveys (CSUR), 2015, 47(3): 1–36.

[25] Gafni E, Lamport L. Disk Paxos. Distributed Computing, 2003, 16(1): 1–20.

[26] Lamport L, Massa M. Cheap Paxos. Proc. of the Int'l Conf. on Dependable Systems and Networks. 2004. 307–314.

[27] Lamport L. Fast Paxos. Distributed Computing, 2006, 19(2): 79–103.

[28] Lamport L. Generalized consensus and Paxos [Technical Report]. Microsoft Research, 2005.

[29] Lamport L, Malkhi D, Zhou L. Stoppable Paxos [Technical Report]. Microsoft Research, 2008.

[30] Lamport L, Malkhi D, Zhou L. Vertical Paxos and primary-backup replication. Proc. of the 28th ACM Symp. on Principles of Distributed Computing. 2009. 312–313.

[31] Lamport L. Byzantizing Paxos by refinement. Proc. of the Int'l Symp. on Distributed Computing. 2011. 211–224.

[32] Moraru I, Andersen DG, Kaminsky M. There is more consensus in egalitarian parliaments. Proc. of the 24th ACM Symp. on Operating Systems Principles. 358–372.

[33] Wang Z, Zhao C, Mu S, *et al*. On the parallels between Paxos and Raft, and how to port optimizations. Proc. of the 2019 ACM Symp. on Principles of Distributed Computing. 2019. 445–454.

[34] Guo Z, Hong C, Yang M, *et al*. Paxos made parallel [Technical Report]. Microsoft Research Asia, 2012. 118.

[35] Lamport L, Merz S, Doligez D. A TLA+ specification of Paxos and its refinement. 2019. https://github.com/tlaplus/Examples/tree/master/specifications/Paxos

[36] Chaudhuri K, Doligez D, Lamport L, *et al*. TLA+ proof system. Proc. of the LPAR Workshops, CEUR Workshop. 2008. 17–37.

Xiaosong Gu, bachelor, CCF student member. His research interests include distributed data consistency, distributed systems, and formal methods.

Lei Qiao, Ph.D., researcher, CCF professional member. His research interests include embedded operating system design and verification for spacecraft.

Hengfeng Wei, Ph.D., CCF professional member. His research interests include distributed data consistency and formal methods.

Yu Huang, Ph.D., professor, doctoral supervisor, CCF professional member. His research interests include distributed algorithms, distributed systems, and networked software systems.