Optimizing Consistency in Distributed Data Services: The CP-Raft Protocol for High-Performance and Fault-Tolerant Replication

Haiwen Du, Kai Wang*, *Member, IEEE,* Yulei Wu, *Senior Member, IEEE*, Hongke Zhang, *Fellow, IEEE*

***Abstract*—The data consistency protocol is a core component of distributed data services that provide fault-tolerance and data consistency across distributed data centers and even edge networks. Raft is a popular approach due to its ease of implementation and superior performance. However, Raft adopts a sequential log entry processing strategy, where log entries without dependencies are not allowed to be processed in parallel, limiting system performance in high-concurrency scenarios. To address this challenge, researchers propose Raft-based protocols that support *out-of-order apply* (OOApply), which is called OORaft. Existing OORaft protocols adopt the Paxos-style election and replication process to merge missing entries on leader candidates. It leads to problems such as extra overhead on dependency analysis, availability when the network is partitioned, and incomplete correctness verification.**

This paper proposes a *concise paralleled Raft* protocol called CP-Raft, which is the first OORaft protocol to focus on dependency analysis overhead and to use full TLA$^+$ validation for the leader election process. Specifically, 1) CP-Raft proposes a Raft-aligned three-step election method that significantly simplifies the difficulty of understanding and solves the availability problem when the network is partitioned. 2) CP-Raft applies a leader-side bitmap-based dependency analysis and representation method to break through the performance bottleneck caused by the high overhead of dependency analysis. 3) CP-Raft discusses why existing methods cannot achieve OORaft correctness verification using TLA$^+$ in a limited time and uses phased verification methods to ensure its correctness. Finally, we implement CP-Raft based on an open-source Raft protocol and discuss its potential performance bottlenecks in various scenarios. The experimental results show that CP-Raft can achieve 1.5× transaction per second (TPS) performance of DP-Raft and 2× of ParallelRaft-CE. It also provides better availability than state-of-the-art OORaft protocols.

***Index Terms*—Distributed systems, fault tolerance, consistency protocol, variant raft protocol.**

I. Introduction

In distributed data services, maintaining data consistency across distributed data centers and edge networks is a crucial challenge [1]. These services rely on data consistency protocols that ensure fault tolerance, data integrity, and system reliability despite network failures and partitions. As the cornerstone of modern distributed data services, many well-known distributed file systems, database and blockchain systems, such as Etcd [2], CockroachDB [3], TiKV [4], Hyperledger Fabric [5], etc., use Raft as their distributed data consistency protocol [6]–[8]. The Raft [9] protocol improved the ease of understanding and implementation of the Paxos protocol [10] by combining the leader selection process with log replication, making it the most widely used crash fault-tolerant (CFT) protocol [11], [12].

However, Raft performs state machine replication based on write-ahead logging (WAL), which consists of consecutive indexed log entries, the commit and apply process must follow the index order [13]. This constraint prevents independent log entries from being processed concurrently, even when they could be committed and applied in parallel without affecting the consistency of the system. In high-concurrency environments, this sequential execution creates a bottleneck, preventing the system from fully exploiting available computational resources, such as multiple CPU cores. As a result, despite the adoption of multi-threaded parallel replication, Raft's performance remains significantly constrained.

If Raft were able to commit and apply log entries in parallel and out of index order, its replication performance could be significantly improved. However, it conflicts with Raft's WAL constraints. To address this limitation, leading distributed systems, such as PolarFS, have introduced variant Raft protocols that support out-of-order processing, known as OORaft [14]–[16]. OORaft allows log entries that do not have read or write dependencies to be committed and applied independently, without strictly following index order. To maintain consistency, it reserves gaps at positions where dependent entries have not yet processed, ensuring correct execution while improving concurrency and system throughput.

Existing OORaft protocols and their drawbacks: Firstly, OORaft protocols cannot check the completeness of the log using only two parameters ($lastLogIndex$ and $lastLogTerm$) [17] because of the existence of gaps. Therefore, existing approaches focus on how to make the new leader collect and check all uncommitted entries from all nodes, complicating the protocol design and [14], [16] causing *ghost logs* and *availability* problems under network partitioning.

Secondly, in OORaft protocols, the replication performance is improved by parallelizing the replicating, committing and applying process of log entries without dependencies. All

This work is supported by National Natural Science Foundation of China (NSFC) (Grant number 62272129) and Taishan Scholar Foundation of Shandong Province (Grant number tsqn202408112) (*Corresponding author: Kai Wang)

Haiwen Du and Kai Wang are with the School of Computer Science and Technology, Harbin Institute of Technology, Weihai, China (email: {haiwen.du, dr.wangkai}@hit.edu.cn)

Yulei Wu is with the School of Electrical, Electronic and Mechanical Engineering, Faculty of Science and Engineering, University of Bristol, Bristol, BS8 1UB, UK (e-mail: y.l.wu@bristol.ac.uk)

Hongke Zhang is with the School of Electronic and Information Engineering, Beijing Jiaotong University, Beijing 100044, China(e-mail: hkzhang@bjtu.edu.cn)

of these protocols use dependency analysis methods to find parallelizable processed log entries. However, the overhead incurred by the dependency analysis process and the impact of the dependency representation on system performance are not considered.

Lastly, in terms of correctness verification, most OORaft protocols are not verified using TLA$^+$ [14]. Only ParallelRaft performs an incomplete verification (they run the TLA$^+$ validation process for seven days and then interrupt it otherwise it would take too long time). The reason is that the OORaft protocol has excessive state depth and width in TLA$^+$, in which conventional validation methods take very long to complete. Existing methods do not consider how to optimize the verification process to achieve a complete verification of the OORaft protocol.

From the above analysis, we can conclude that the design of OORaft should focus on the simplicity, parallelism, and correctness of the protocol. To address these challenges, we propose a *concise paralleled Raft* protocol called CP-Raft. Its contributions mainly include:

1) Bug Fixes Through a Concise Election Strategy: CP-Raft introduces a three-phase election strategy that integrates log collection and synchronization with the pre-election and election phases. Compared to Raft, CP-Raft only adds a *log-type* attribute to enable parallelized log replication and election safety, making it the OORaft variant with the fewest additional attributes. It minimizes the modifications to Raft and avoids problems of *availability* and *ghost logs*.

2) Performance Optimization via Efficient Dependency Analysis: CP-Raft is the first OORaft protocol to address the overhead of log entry dependency analysis. It introduces a leader-side, bitmap-based dependency analysis and representation method, reducing analysis time by 80% compared to state-of-the-art approaches. This efficiency significantly enhances parallel log entry processing, thereby improving TPS performance.

3) Comprehensive Verification Using a Phase-Based Approach: CP-Raft proposes a phased TLA$^+$ validation method. It explores the reasons why we cannot directly validate OORaft using TLA$^+$ and performs a complete TLA$^+$ validation for two key issues: *leader election safety* and *ghost logs problem*.

The remainder of this paper is organized as follows: Section II describes the background and motivation for our proposed protocol. Section III presents our detailed design of the log entry and dependency analysis method. Section IV introduces the Raft-aligned election process of the CP-Raft. Section V shows how we verify the bug fixes using the phase-based approach (contributions 1-*ghost-logs* and 3). Section VI outlines the performance experiments (contributions 1-*availability* and 2). Section VII discusses related work, and Section VIII provides the conclusions of our work.

II. Background and Motivation

In this section, we provide insights of existing OORaft protocols and show the motivation behind this paper. Firstly, we introduce why consistency protocol is important in distributed data services. Then, the most concerned processes of the protocols are discussed, including log entry replication, leader election, and protocol verification.

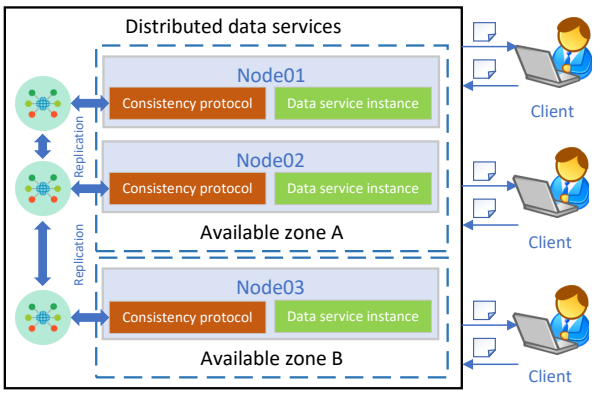


Fig. 1. In distributed data services, the consistency protocol is responsible for synchronizing the data from each node. For data read and write requests, they need to be processed by the consistency protocol before responding to the client, thus ensuring linearizability.

A. Consistency Protocols in Distributed Data Services

Consistency protocols such as Raft and Paxos are foundational in distributed data services to ensure that all nodes reflect the same data state, even in the presence of failures or concurrent operations. In regular distributed data services workflows, all client requests must go through consistency protocol to achieve linearizability, as is shown in Fig. 1.

As a key component of distributed data service, Raft's primary responsibility in a distributed data service is to safely replicate data from the leader to the followers to ensure consistency and fault tolerance. Unlike Multi-Paxos [18], the Raft protocol organically combines the serial log entry replication mechanism with the leader election process, and its index continuity of log entries eliminates gaps recovery process, thus simplifying its implementation and understanding [19].

Therefore, a well-designed consistency protocol should be easy-to-understand, high-performance, and validated by formal verification tools. For better understanding, we summarize related methods and their essential features in Table. I.

B. Parallel Replication Strategies in Raft Protocols

As the key feature related to performance, the efficiency of log entry replication plays a crucial role in protocol design and implementation. The mainstream ways can be concluded in two classes.

1) Parallel Replication with Sequential Apply: Parallel replication with sequential apply is adopted by regular Raft implementations. Although Raft follows sequential-commit and sequential-apply policy, the log entries are replicated from the leader to followers in parallel [22]. The leader can copy multiple batches of log entries to followers and does not need to wait for the response of the previous replication request [23]. Therefore, parallel replication can significantly improve the performance of Raft protocol [24]. However, the replication requests arrive at followers out-of-order in practical scenarios due to out-of-order delivery in computer networks

TABLE I
COMPARISON OF RELATED OOAPPLY PROTOCOLS.

Features	OOApply	No ghost logs	Availability	Verification	Dependency resolving	Election phases
Vanilla Raft [9]	×	✓	×	✓	N/A	Only vote
Raft [20]	×	✓	✓	✓	N/A	Pre-vote + vote + configuration
Multi-Paxos [18]	✓	×	×	✓	Dependency ID	Paxos synchronize progress
ParallelRaft [15]	✓	×	×	×	LBA conflicting	Paxos synchronize progress
ParallelRaft-CE [14]	✓	✓	×	Not complete	Dependency ID	Paxos synchronize progress
DP-Raft [16]	✓	×	✓	×	WAR checking	Pre-vote + vote + merge
LCR [21]	✓	✓	✓	×	Non-dependency data only	Pre-vote + vote
CP-Raft	✓	✓	✓	✓	**Bitmap**	**Pre-vote + vote + full-vote**

[25]. Followers cache and hang the out-of-order request until all log entries are persisted before it. However, it occupies more thread resources on the leader and prolongs the response time of replication requests, which leads to overhead on the leader and becomes the bottleneck of system performance.

2) Parallel Replication with Out-of-order Apply: Researchers proposed variant Raft protocols to optimize the efficiency of committing and applying log entries. As the applying order of non-dependency log entries will not break the sequential consistency of the state machine, ParallelRaft [15] is proposed to support the out-of-order committing and applying feature on Raft, i.e., an OORaft protocol. However, although OORaft can provide higher log entry replication efficiency, there are two key issues to be addressed.

The first issue is that ParallelRaft increases the implementation difficulties, which may break the easy-to-understand feature of Raft. The most complex process is the log recovery process when the leader changes. The new leader must merge the out-of-order committed log entries from all the nodes, similar to a multi-Paxos procedure [19]. Existing approaches introduce Paxos-style attributes [14], [16] to achieve log recovery. OORaft needs to implement Paxos procedures, which seriously complicate the implementation of the protocol.

The second is that OORaft only cares about how to analyze dependency, but does not consider the impact of dependency analysis overhead on system performance. Since the application scenarios of ParallelRaft are distributed file systems and redo logs in databases [15], [26], it can resolve dependencies by checking read-write conflicts on logic block address (LBA) of log entries. Therefore, each entry can save a few number-formatted dependency identifiers to support the dependency-resolving method by passing the identifier. However, ParallelRaft is restricted to specific application scenarios in that dependencies can be numeral presented [16]. Besides, the system performance is also sensitive to the dependency-resolving method because it can delay the committing and applying operations. The overhead of inefficiency methods will be greater than the optimization effect [27].

C. Problems in OORaft Election Process

Although OOApply is a promising way to improve the performance of the Raft protocol, the re-designed log recovery process can lead to many problems. This section analyzes the existing protocols and discusses the potential issues to guide the design of the election process.

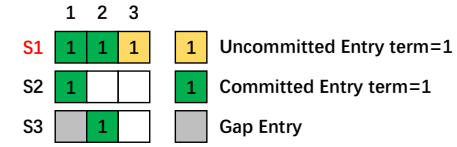


Fig. 2. When the leader node is S1, and 1 to 2 indexed log entries are committed. If S1 fails at this point, both S2 and S3 cannot become the new leader without a log recovery process because none of them have all of the committed entries.

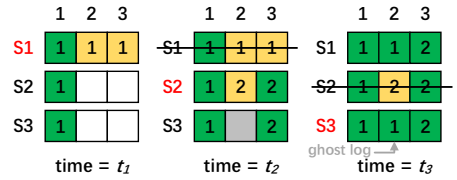


Fig. 3. At t_1, S1 persists log entries $E_{1,1}$ to $E_{1,3}$ and applies $E_{1,1}$. Then, S1 fails at t_2, and S2 is chosen as the new leader. Since S2 cannot recover $E_{1,2}$ and $E_{1,3}$ from S1, it will upgrade the term to 2, start providing service, and append new log entries from $E_{2,2}$. After S2 replicating, committing and applying $E_{2,3}$, clients can read the payload that $E_{2,3}$ carries. Finally, S2 fails at t_3, S1 resumes, and S3 is chosen as the new leader. S3 will recover, commit, and apply the log entries it misses, i.e., $E_{1,2}$ from S1. At this point, $E_{1,2}$ shows again and will be applied to the state machine, causing ghost logs and violating linearizability. If $E_{1,2}$ and $E_{2,3}$ write on the same keys, it will be worse because ghost values overwrite the state machine. This issue is present in both ParallelRaft and DP-Raft protocols.

1) Ghost Logs: As OOApply on log entries generates gaps in the logs on non-leader nodes, Raft log entries must be committed and applied consecutively on each node. Without log entry recovery, we cannot ensure the new leader contains all the committed log entries, as shown in Fig. 2.

Unfortunately, we cannot do it by only collecting missing log entries from other nodes because of 'ghost logs' [28], as shown in Fig. 3. The newly elected leader may revert to log entries considered gaps by the previous leader. To eliminate this problem, ParallelRaft-CE solves this problem by phased log recovery that uses the synchronization number property, i.e., Paxos progress [14]. DP-Raft [16] adds an extra 'merge' phase after voting. When the candidate collects enough votes, it becomes a pre-leader and collects the missing gap entries from followers. However, these methods bring more phases in the log recovery process and increase the complexity of the implementation because they break the concise nature of the Raft voting process. Therefore, ensuring that the log recovery mechanism is integrated into the leader election process to match Raft's election phases is a challenging task.

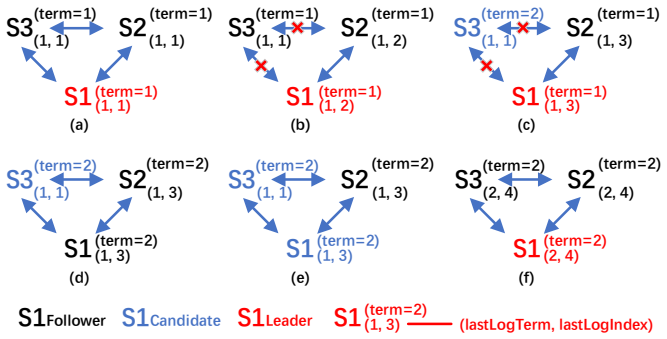


Fig. 4. An Example of availability problem in network partitioning. Suppose we have a 3-node cluster in (a), where the leader is S1. Then, in (b), two nodes, S1 and S2, can normally communicate, while S3 cannot communicate with S1 and S2. After some time, in (3), S3 becomes the candidate, increases its term number to 2, and sends vote requests to S1 and S2. Then, in (d), S3 re-joins the majority partition. Since S3 has a higher term, vote requests from S3 will make S1 and S2 raise their term to match term 2 and change their status to followers. In this period, there is no leader in the cluster, so the cluster is not available to clients. However, S3 cannot become the new leader due to its lower $lastLogIndex_{S3}$ and $lastLogTerm_{S3}$, as shown in (e). Finally, S1 will be chosen as leader again, and the cluster will start to provide service to clients in (f).

2) Availability in Network Partitioning: Network partitioning is a common failure in distributed systems where nodes are partitioned into regions that can only communicate internally [4], [29], [30]. In vanilla Raft and ParallelRaft, if a network partition occurs and a partition contains a majority of nodes, the nodes not in the majority partition will increase their term number and generate voting requests. When it re-joins the majority partition, the higher term number of these voting requests will trigger an election process, as shown in Fig. 4.

To solve this problem, adding a pre-voting phase to Raft has become the leading solution [20]. Well-known open-source database systems such as Etcd [2] and cockroachDB [3] provide open-source implementations of Raft with the pre-voting phase. In this scheme, a node increases its term after a timeout once it collects pre-voting grants from the majority. The grant condition is the same as the voting condition, i.e., its $lastLogIndex$ and $lastLogTerm$ are not less than the pre-candidate. Only when the node collects the majority grants does it increase its term number and start voting. Although this adds another phase to the leader election process, it provides higher availability for the cluster.

Unfortunately, this problem still exists in OORaft protocols. OORaft protocols such as ParallelRaft-CE focus on how to fill the gaps but how to achieve higher availability. They did not consider how to integrate the pre-voting phase of Raft.

D. Correctness Verification Process

For the correctness of Raft protocols, using TLA$^+$ language and TLC tools is the best way to formalize and verify them. TLA$^+$ [31] is a formal specification language proposed by Leslie Lamport in 1999 for designing, modeling, documenting, and verifying programs.

The TLA$^+$ specification can be represented as $Spec = Init \cap [Next]vars$. During the verification process, the TLA$^+$ step executes the entire combination of actions while the developer sets some invariants, i.e., the incorrect states. If all reachable states satisfy the constraints of the invariants, the system is proven correct.

For non-OORaft protocols, the authors of Raft open-sourced the TLA$^+$ logic of the first version of Raft [32] (i.e., Raft without pre-voting phase). Subsequently, CockroachDB released the Raft protocol with a pre-voting phase implemented by TLA$^+$ on GitHub [33]. However, Raft's verification process requires many actions, so the number of states generated by the TLC verification tool far exceeds the computer's processing ability. For example, in a Raft protocol with the pre-voting phase, we need to build a state graph with a depth of 30 in the verification process with only two rounds of election and 1 log entry committing. With our available computing power, it takes more than 24 hours to compute a state graph of depth 30. To verify the OORaft protocol, we should verify at least two rounds of election and three entries committing. Verifying a state graph with more than 40 depths may take years with the computational capacity of existing hardware. Therefore, decomposing the OORaft protocol into more straightforward phases is a feasible way to prove its correctness [34].

TABLE II
TERMINOLOGY USED IN THIS PAPER

Terms	Description
Si	The node with id i
$Entry_x$	The log entry with index x
$term_i$	The current term of Si
$Term(Entry_x)$	The term of $Entry_x$
$lastLogIndex_i$	The maximum index of the log entries in Si
$lastLogTerm_i$	The term of the log entry with maximum index in Si
$List(GE)_i$	The index list of gap entries in Si

III. Log Entry Design and Dependency Analysis Method of CP-Raft

This section details our design of the CP-Raft protocol in log entry properties, election process, and verification method. Before the introduction, we define the terms in Table II.

A. Design in the Types of Log Entries

Since the gaps need to be reserved in the OORaft log, we design new types of log entries in CP-Raft to distinguish them from regular entries. CP-Raft provides four types of entries: regular entry (RE), configuration entry (CE), gap entry (GE), and confirmed gap entry (CGE). The attributes in the entry are the payload, the term, and the gap index list (gap index list only available for CE). They are detailed as follows:

RE: contains a valid payload. RE is replicated, committed, and executed out of order according to the CP-Raft protocol.

CE: must be the first log entry of a valid term and contain the index of all GEs in the last valid term. All GEs in the last valid term are confirmed when a CE is committed.

GE: only saves the term for the entry without payloads, i.e., the gaps in the log. REs can overwrite GEs by log replication, log synchronization, and log collection requests.

CGE: contains no payload, only the term of the entry. It is generated by the CE applying process, i.e., the GE

TABLE III
ENTRY TYPES IN RAFT AND OORAFT PROTOCOLS

Protocols	RE	CE	GE	CGE	Merge policy
Raft	✓	✓	×	×	N.A.
ParallelRaft	✓	×	×	×	Directly write on gaps
ParallelRaft-CE	✓	×	✓	×	Compare timestamp
DP-Raft	✓	×	✓	×	Compare committing status
LCR	✓	✓	✓	×	Non-conflicting indexes
CP-Raft	✓	✓	✓	✓	Non-overwritten CGE

confirmation process. Unlike a GE, it cannot be overwritten (i.e., filled as a gap) and is treated as a RE to be replicated.

Since the recovery and gap confirmation mechanism must be completed before receiving entries from a new term, out-of-order replication cannot work across different terms. Therefore, the follower node must ensure that it has persisted all non-GEs from the last valid term before it persists a CE with a newer term index. In other words, the follower node must have received a CE in term x before it is allowed to persist entries in term x.

To provide a better view of our protocol, we compare CP-Raft's log entry types with other OORaft protocols in Table III. Since the names of the entry types in different designs are not the same, when they represent the same content, we use the names in CP-Raft uniformly.

B. Log Entries Dependency Analysis

CP-Raft executes a dependency-checking process for dependencies between entries before log entry replication. It provides an interface for applications to implement customized dependency-checking logic. Then, the checking result is represented as a bitmap that saves the entries' dependency relationships. Bitmap can also improve the efficiency of the dependency analysis process because followers do not need to resolve the payload of the entries or use payload-related identifiers to check the dependency.

To prevent excessive storage and processing overhead, the dependency checking range should be limited in a fixed range. We call it *look_back_size*. For $Entry_n$, the dependency bitmap B_n stores all the entry indexes in the range of *n-look_back_size* to *n-1* that should be committed and applied before $Entry_n$. The calculation process of B_n can be represented as equation (1).

$$B_n = (b_{n,n-1}, b_{n,n-2}, \ldots, b_{n,n-look\_back\_size})$$
$$b_{i,j} = \begin{cases} 1, & \text{if } Entry_i \text{ depends on } Entry_j \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Then, we can persist $Entry_n$ on the leader and replicate it to the followers. The follower S_k checks if all entries in the bitmap are persisted by checking $B_n \wedge (I_k << \epsilon)$, where I_k is the index map of the follower S_k and ϵ is the difference between $n-1$ and the maximum index in I_k. If $B_n \wedge (I_k << \epsilon) = B_n$, $Entry_n$ can be processed out of order. Otherwise, $Entry_n$ is added to the cache to wait for the arrival of missing entries. The length of I_k is the same as the cache size. We provide an example in Fig. 5 that helps understand the replication process of CP-Raft.

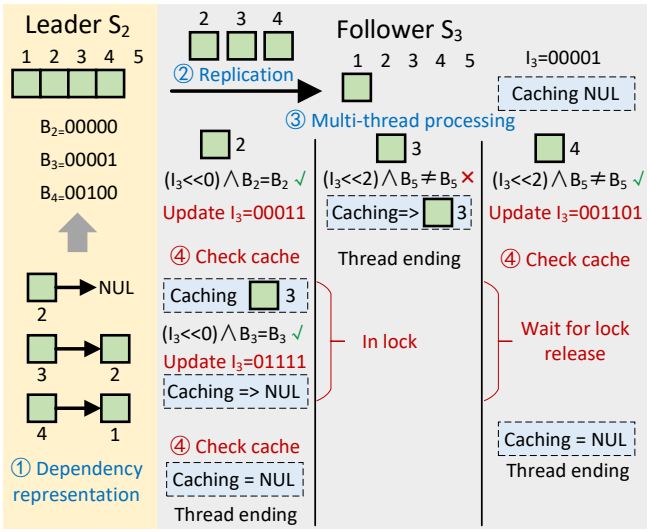


Fig. 5. A sample of log replication process of CP-Raft.

TABLE IV
DEPENDENCY ANALYSIS COST OF BASELINE METHODS

Protocols	Representation	Calculation	Complexity
Raft	I_k: bitmap B_n: bitmap	logical and	$O(1)$
DP-Raft	I_k: index list B_n: index list (len=DAR)	loop check	$O(DAR)$
ParallelRaft	I_k: LBA list B_n: LBA list (len=N)	loop check	$O(N * DAR)$ default $N = 2$

From the example, we can find that the dependency analysis process is executed not only when S_k receives an entry from the leader but also when I_k changes. If I_k changes, all the entries in the cache need to be checked because the entries they depend on may arrive. In addition, this process is executed in locks. These reasons make the performance sensitive to the dependency analysis efficiency. For example, if followers cache M entries (usually close to the CPU thread number) and each round of dependency analysis complexity is $O(t)$, the analysis cost is $O(Mt)$ per entry. When the system runs on high TPS (tens K TPS), the cost will be huge. We show the difference between CP-Raft, DP-Raft, and ParallelRaft in Table. IV. The dependency analysis range (DAR) is also usually close to the CPU thread number for optimal parallel execution, minimizes lock contention, balances throughput and latency, and improves cache efficiency. Therefore, the analysis efficiency of CP-Raft is tens to hundreds of times faster than other protocols.

IV. ELECTION PROCESS OF CP-RAFT

To support OOCommit and OOApply, the consistency of gaps during the leader election process is critical to its correctness and execution efficiency. The newly elected leader must ensure two conditions to avoid the 'ghost logs' problem: 1) it contains all committed log entries; 2) the confirmed gaps should no longer be committed again on any nodes. In the CP-Raft design, we do not use the phased synchronization scheme like ParallelRaft-CE, but instead, we use CEs and utilize the

TABLE V
COMPARISON OF ELECTION PHASES FOR DIFFERENT PROTOCOLS

Seq.	1	2	3	4
Raft	Pre-vote	Vote	—	Service
Multi-Paxos	Vote	Recovery	—	Service
ParallelRaft-CE	Vote	Request synchronization	Update synchronization ID	Request synchronization
DP-Raft	Log collection	Log merging	—	Service
CP-Raft	Log collection	Log synchronization	Gap Confirm	Service

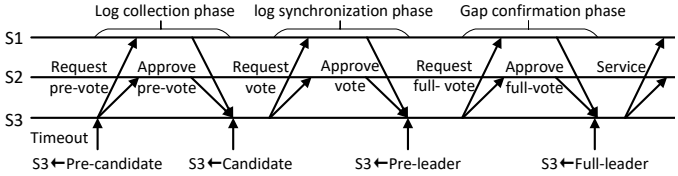


Fig. 6. The election message flow of CP-Raft.

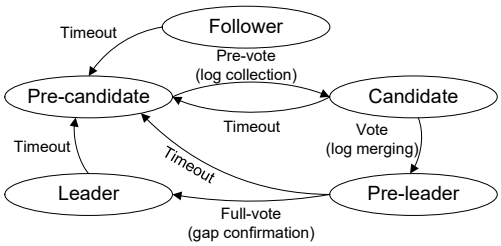


Fig. 7. The state transferring relation of CP-Raft.

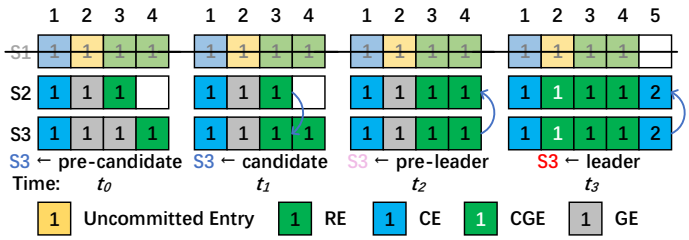


Fig. 8. An example of how CP-Raft achieves log recovery in the election process. Since term 1 starts from index 1, $Entry_1$ is CE. $Entry_2$ only persisted in the old leader and is not replicated to S2 and S3. $Entry_3$ is committed and replicated to S2 but not S3. $Entry_4$ is committed and replicated to S3 but not S2. S3 becomes pre-candidate at t_0 and tells other nodes its missing entries ($Entry_2$ and $Entry_3$) by *pre-vote* request. At t_1, S3 gets $Entry_3$ and knows S2 also does not have $Entry_2$ and $Entry_4$ in the response of *pre-vote* from S2. Then, S3 becomes a candidate and stops processing other *pre-vote* requests because it collects more than 1/2 *pre-vote* grants. At t_2, S3 tries to replicate $Entry_4$ to S2 and tells S2 $Entry_2$ is not persisted by it by *vote* request. After receiving the grant in the response of *vote* from S2, S3 becomes pre-leader. Finally, S3 generates a CE, which takes the message that $Entry_2$ is confirmed as CGE as $Entry_5$ and replicates it to S2. When $Entry_5$ is committed, $Entry_2$ is confirmed as CGE (will not be recovered by any node), and S3 becomes the leader of term 2.

nature of Raft to solve this problem. Compared with Raft, the only additional property is $entryType$. To better present the difference between the phases of CPRaft, ParallelRaft-CE, Paxos, and Raft, we compare them in Table V.

The CP-Raft election process is divided into three phases: log collection, log synchronization, and gap confirmation. Of these, the log collection phase includes the pre-voting phase, the log synchronization phase corresponds to and includes a voting phase, and the gap confirmation phase corresponds to and includes a process of committing configuration log entries to Raft. The state transition relationship is shown in Fig. 7. The election phases include:

1) *Timeout*: Si has not received messages from the old leader, it becomes *pre-candidate*.

2) *Log collection phase (pre-vote)*: Si collects missing but committed entries from other nodes.

3) *Become candidate*: Si collects enough grants from the majority nodes and learns the entries that other nodes miss. It becomes *Candidate*.

4) *log synchronization phase (vote)*: Si synchronizes entries that other nodes miss.

5) *Become leader*: Si collects enough grants from the majority nodes. It generates a CE that saves the confirmed gaps in the last term. It becomes *Leader*.

6) *Gap confirmation phase (full-vote)*: Si replicate the CE to other nodes.

7) *Become full-leader*: The CE is committed on majority nodes and the GEs become CGEs in the last term on majority. They will never be overwritten, which solves the *ghost log* bug. Si becomes *Full-leader* and start service.

To better understand how the election process runs, we provide the message flow of the election in Fig. 6 and provide an example in Fig. 8. In the following sections, we will detail what will happen in *Log collection phase*, *log synchronization phase*, and *Gap confirmation phase*.

A. Log collection phase

A node Si becomes a pre-candidate when it triggers a timeout because it has not received a request from the leader node for a long time. It will broadcast to other nodes $lastLogTerm_i$, $lastLogIndex_i$ and $List(GE)_i$. When the other node Sj confirms that the pre-candidate has higher $lastLogTerm_i$ and $lastLogIndex_i$, it returns the entries in $List(GE)_i$ and $List(GE)_j$. It grants the request after Sj contains no missing log entries in $List(GE)_i$.

B. Log synchronization phase

Si must have recovered all the committed log entries when it collects more than half of the grants in the log collection phase. At this point, it transfers its state to the candidate and operates $term_i$++. Subsequently, it synchronizes the log entries to other nodes. Take Sj as an example; Si synchronizes the log entries to Sj based on the $List(GE)_j$ during the log collection phase. For an index corresponding to GE on both

Si and Sj, Si still synchronizes it. When Sj confirms that all its GE are confirmed or filled, Sj grants the request from Si.

For node Sm that did not communicate with Si during the log collection phase, Si will synchronize from $Entry_1$ by default. If Sm's maximum consecutive log entry index is not zero, it will return its maximum consecutive log index to Si, causing Si to start synchronizing from that index. If the term of $Entry_{lastLogIndex_m}$ does not match the one in the leader, Sm will discard it and use the one from the leader. This process is the same as Raft's log replication process, which uses reverse-order, entry-by-entry truncation to synchronize log entries with different terms between leader and followers.

C. Gap confirmation phase

When Si has collected more than half of the voting grants, it has collected and synchronized all of the committed log entries to the majority. At this point, it will become the pre-leader. However, the pre-leader can still not provide services because the majority has not confirmed the CGEs. In this case, the Si adds the index of GEs in the CE and replicates CE to the majority node. Due to CE having a dependency on all the entries in the last valid term, CE must be committed and applied sequentially (in fact, since the Si cannot provide services, i.e., it cannot perform RE replication, thereby cannot execute OOCommits or OOApply). When the majority commits the CE, the Si becomes the leader, modifies the state of GE to CGE, and starts to provide services to the client.

From the detailed design, we can conclude that only GE can be overwritten when there is an entry with a different entry type, the same term, and the same index. In CP-Raft, for non-GE entries with the same term and index on different nodes, they must be identical, following Raft's consistency rules. It benefits from the design of the term in GE, the introduction of CGE, and the $List(GE)$ in CE. Besides, the 3-phases of the election process also match the design of Raft. They simplify CP-Raft implementation and enable us to make as few changes as possible from popular open-sourced Raft implementations.

V. Protocol Verification Method of CP-Raft

In this section, we detail how to use TLA$^+$ to verify the correctness of CP-Raft. Firstly, we explain why existing methods fail to completely verify the correctness of OORaft protocols. Then, we formalize key lemmas of CP-Raft and use a phase-based method to verify them. Finally, we show and discuss the results.

A. Why TLA$^+$ cannot directly verify OORaft

When we use TLA$^+$ to verify Raft protocols, the number of system parameters and the states in the election and log replication process generate an enormous combination of states. The TLA$^+$ code, behaviors and interactions of the CP-Raft can be found at online repository[1].

The TLC tool must check all the reachable and independent states to verify the protocol entirely. Take a three-node cluster as an example; the actions and states from the 'the initial cluster' to 'the candidate is elected' are shown in Fig. 9 (Even without the consideration of $Restart$, $DuplicateRequest$, and $DropRequest$ actions).

Fig. 9. In the verification process, some actions must be sequentially executed. For example, when we execute $RequestPreVote(S1, S2)$, $Timeout(S1)$ must be executed before it. Therefore, we use different colors to denote different execution sequence flows. Besides, actions in different flows do not need to follow the sequence. Finally, we can get a set of combinations of actions, which are the states we need to check.

Therefore, we can calculate the number of independent states $States_C$ from the initial cluster state to the candidate is $A_{15}^{15}/\left(A_5^5\right)^3$.

Similarly, the states are generated from 'the candidate is elected' to 'the pre-leader is elected' is $State_{PL}$. The states from 'the pre-leader is elected' to the leader is elected $State_L$. $State_{PL} = State_L = A_{12}^{12}/\left(A_4^4\right)^3$ because $Timeout$ is no more needed. Therefore, a full verification of a selection process would require a state graph of at least 39 depths. The number of independent states $State_L * State_{PL} * State_C$ is more than 900 trillion. However, for our experimental environment (7 million states per minute), completing this verification would take close to 250 years.

To verify the TLA$^+$ code with a complete configuration, we design a phased verification approach to reduce the number of states of the experiments. It allows some of the critical verification steps of CP-Raft to be completed quickly. This approach improves the efficiency of verification efforts and helps ensure the correctness and reliability of the system or protocol under scrutiny.

B. Formalization of key lemmas in CP-Raft protocol

We define phases, initial states, and invariants for each lemma to show how we complete the verification process.

1) Leader completeness in elections: the new leader node stores all the committed log entries after the election.

Since the leader's REs must be sequential during the active term, their status has the following possibilities: sequential committed, uncommitted, and out-of-order committed. Therefore, we initialize a three-node cluster that covers all these cases, as an example in Fig. 10. Then, we trigger a leader election to verify the leader's completeness.

Phase 1: from 'old leader fails' state to 'new candidate is elected' state.

$S3$ can be selected as the new leader node after $S1$ stops working. Therefore, we define the invariant for finishing $S3$'s log collection phase (becoming a candidate) as:

$$\begin{aligned} &\wedge \exists index \in \{1,2,4,5\} : log\left[s3\right]\left[index\right].value = GE \\ &\wedge state[s3] = Candidate \end{aligned} \tag{2}$$

[1]https://github.com/Magnomic/CP-Raft

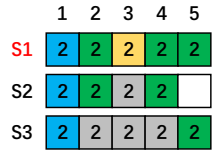


Fig. 10. In the initial state, $S1$ is the leader of term 2. The status of $Entry_2$, $Entry_3$, and $Entry_4$ are sequential committed, uncommitted, and out-of-order committed. If $S1$ stops working, the new leader should commit $Entry_2$ and $Entry_4$, while $Entry_3$ is confirmed as a gap.

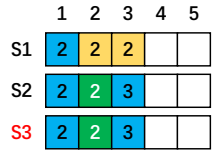


Fig. 11. In the initial state, $S3$ is the leader of term 3. $Entry_2$ is confirmed as CGE in $S3$, where the corresponding information is saved in *empty_index_list* of $Entry_3$ on $S3$. If $S3$ stops working and $S1$ is elected as the new leader, $Entry_2$(RE) on $S1$ will conflict with $Entry_2$(CGE) on $S2$ and $S3$. $S1$ should replace $Entry_2$(RE) with $Entry_2$(CGE). Otherwise, $Entry_2$(RE) will be committed and applied by $S1$, which causes a 'ghost log' problem.

Phase 2: from 'new candidate is elected' state to 'new leader is elected' state.

We define the invariant from the S3 log synchronization phase to finishing the gap confirmation phase (becoming a leader) as:

$$\begin{aligned} &\wedge \vee \exists index \in \{1,2,4,5\} : log\,[s3]\,[index]\,.value = GE \\ &\quad \vee \exists server \in \{s2, s3\} : log\,[server]\,[3]\,.value \neq CGE \\ &\wedge state[s3] = Leader \end{aligned} \tag{3}$$

2) Ghost log security: The obsolete RE identified as CGE by the leader will not be recovered.

If an entry is confirmed as CGE, the new leader will add its index to the *empty_index_list* of the new term's CE in CP-Raft. Here, we initialize a three-node cluster to verify if the CGE entries can be recovered again as a 'ghost log.' We show the initial state of the case in Fig. 11.

Phase 1: from 'initial state' to '$S1$ receives $Entry_3$(CE).'

$S1$ is qualified to become a candidate once it receives the $Entry_3$(CE). The reason is that it violates the pre-order continuity (the term of $Entry_3$(CE) on $S3$ is greater than the term of $Entry_3$(RE) on $S1$). When $S1$ receives the $AppendEntries$ request from $S3$, it removes the $Entry_3$(RE) and then saves $Entry_3$(CE) from the request. Therefore, we verify that $S1$ cannot be elected as $Candidate$ when $AppendEntries$ is disabled. The invariant in this phase is defined as:

$$\begin{aligned} &\wedge \exists server \in \{S2, S3\} : \\ &\quad \exists m \in InvalidMessage(messages) : \\ &\qquad (m.mtype = RequestPreVoteResponse \\ &\qquad \wedge m.msource = server \\ &\qquad \wedge m.mdest = S1 \\ &\qquad \wedge mvoteGranted = TRUE) \\ &\wedge state[s1] = Candidate \end{aligned} \tag{4}$$

TABLE VI
RUNTIME ENVIRONMENT OF EXPERIMENTS

Component	Description
CPU	2 x Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
Cores	36
Threads	72
Memory	504 GB
HDD Storage	Seagate HDD with 4 TB capacity and 7200-RPM
SSD Storage	SAMSUNG SSD 970 Evo Plus 500 GB
Switch	H3C Mini S8G-U, 16-Gb/s switching

TABLE VII
VERIFICATION RESULTS OF LEMMA: LEADER COMPLETENESS IN ELECTIONS

Phase ID	State num	Different state num	Time costs	Max depth
Phase 1	86,541,870	13,973,698	7 min 30 s	38
Phase 2	50,417	10,691	5 s	62

TABLE VIII
VERIFICATION RESULTS OF LEMMA: GHOST LOG SECURITY

Phase ID	State num	Different state num	Time costs	Max depth
Phase 1	61,403	7,999	9 s	29
Phase 2	32,240,914	5,462,251	3 min 10 s	54

Phase 2: from $S1$ receives $Entry_3$(CE) to $S1$ is elected as leader.

$S1$ can get grants from $S2$ when it replace $Entry_3$(RE) with $Entry_3$(CE). However, $Entry_2$(RE) will not be replaced by $Entry_2$(CGE) because they have the same term. When $S1$ receives $Entry_3$(CE), it will read the $confirmedGaps$ attribute from the entry and replace $Entry_2$(RE) with $Entry_2$(CGE). The information is saved in the $confirmedGaps$ attribute of $Entry_3$(CE). Therefore, when $S1$ becomes the leader, $Entry_2$(RE) will not show in any node. We define the invariant in this phase as:

$$\begin{aligned} &\wedge \exists server \in \{S1, S2, S3\} : log[server][2] \neq CGE \\ &\wedge state[s1] = Leader \end{aligned} \tag{5}$$

These lemmas ensure that the committed entries do not meet the ghost log issue in the leader node.

C. Experimental results

In this section, we use the TLC tool to validate the correctness of the CP-Raft protocol. The corresponding code is open-sourced at our online repository. Experiments (including performance evaluation) are performed on servers with hardware environment in Table VI. The experiment is divided into three parts. Firstly, we validate 'Leader completeness in elections' and 'Ghost log security' using the phased verification method. The results are shown in Table VII and Table VIII. For 'Leader completeness in elections', the verification process takes 7 minutes and 35 seconds, covering over 90 million states with a depth of 100. For 'Ghost log security', it takes 3 minutes and 19 seconds, covering more than 30 million states with a depth of 83. Therefore, we can conclude that the phase-based method can achieve the verification process in 12 minutes.

TABLE IX
RESULTS OF OVERALL VERIFICATION

Phase ID	State num	Different state num	Time costs	Max depth
Overall	105,137,797,840	20,456,383,708	240 h	37

Then, we conduct an 'overall verification' process that tries to validate all the reachable states of CP-Raft. The results is shown in Table IX. As discussed, OORaft protocols (including CP-Raft) cannot be fully validated by TLA$^+$. However, the experiments on the overall verification process are still included in our paper as supplemental proof. It will not make our validation incomplete because CP-Raft did not change the property of *Election Safety*, *Leader Append-Only*, and *Log Matching*. We run the overall verification for 240 hours, where more than 100 billion states are verified and the maximum validated state depth is 37. The invariant is inherited from Raft, i.e., neither leaders nor entry misplaces.

From the experimental results, we can prove the correctness of the critical election process of CP-Raft within 10 minutes, which benefits from our phased validation strategy.

VI. PERFORMANCE EVALUATION

We implement CP-Raft based on BRaft [35], which is the most stared C++ stand-alone Raft implementation on GitHub. The state machine is a distributed file on which the client makes read and write requests. We use this state machine because we can easily adapt them to simulate random or sequential R/W, R/W on different device types, and R/W dependencies.

To accurately model the dependency between log entries, we use a Zipfian distribution and change distribution parameter s, which can capture the heavy-tailed nature of data access patterns observed in many practical workloads. The Zipfian distribution ensures that a small subset of file are modified significantly more frequently, simulating realistic dependencies that arise in high-contention environments. By incorporating this distribution into our simulation, we can better evaluate the impact of dependency constraints on Raft's performance and assess the effectiveness of optimizations designed to mitigate replication bottlenecks.

All experiments are carried out on a 9-node cluster with the same hardware environment specified in Table VI. Two nodes function as clients, while the remaining nodes run the baseline protocols. To fairly compare performance differences between protocols, we disable the batching (i.e., only one log entry is replicated in every AppendEntries request) and pipeline features for CP-Raft and all the baselines as state-of-the-art papers did [21], [36]. We use open-looped clients to evaluate the performance of protocols because they provide a controlled and realistic workload generation mechanism, independent of system feedback. All network requests between servers are RPC calls based on the bRPC framework. The baselines we use are Raft and popular variant Raft protocols: Raft-cache (cache enabled), DP-Raft, ParallelRaft, ParallelRaft-CE, and LCR. We introduce the baselines as follows:

Raft-cache [20]: Compared with Raft, Raft-cache will cache out-of-order arrived replication requests but follow sequential committing and applying policy. It can achieve better TPS and response latency performance than Raft because the leader does not need to resend out-of-order arrived log entry replication requests.

ParallelRaft [15]: ParallelRaft achieves out-of-order committing and applying by introducing log entries recovery process. ParallelRaft uses *look behind buffer* to resolve the dependency of log entries, which acts as a bridge built over a possible hole in the log.

ParallelRaft-CE [14]: Unlike ParallelRaft, ParallelRaft-CE uses Paxos-style approaches to merge entries in the election process. It solves the problem of the 'ghost log' in ParallelRaft and uses TLA$^+$ to verify it. Since ParallelRaft-CE did not propose a new log replication or dependency resolving method, the TPS performance is the same as ParallelRaft.

DP-Raft [16]: The dependency-resolving process of entries in DP-Raft is finished before they are replicated to followers. DP-Raft's entry keeps an index list of the entities on which they have dependencies. It designs a 'condition-based entries merging policy' in the election process, which is more efficient than ParallelRaft.

LCR [21]: LCR allows entries that only write on new keys to be out-of-order committed and applied. Different from the methods above, these entries are replicated by followers. The leader needs to check the validity of these replication requests and use confirmation signals to commit and apply them. As LCR does not need to check dependencies between entries, there is no extra dependency resolving cost on followers.

We do not compare our protocol with optimized non-OORaft protocols such as multi-leader (e.g., multi-raft [4],) entry incremental coding, and some batching and pipelining replication methods. The reason is that the features of OORaft are compatible with these methods, i.e., OORaft can also benefit from these methods. In the following sections, we show the design of experiments and provide the experimental results, which support the contributions we listed in Section I.

A. Experiments on TPS Performance

In this section, we explain why our dependency analysis method can improve the replication efficiency during the Raft-based state machine replication process.

1) Dependency analysis cost: Of the baseline methods, the one that makes the most difference is the dependency analysis method. As the dependency analysis process is in the critical path of log replication, the system's overall performance is sensitive to its overhead. To evaluate the impact of this overhead on TPS performance, we implement the dependency analysis methods proposed by baseline methods. Further, we measure the time cost of these methods and simulate more workloads for better understanding. The experimental results are shown in Fig. 12.

Experimental results indicate that the time overhead of dependency analysis significantly impacts performance. This is because dependency analysis occurs at two critical protocol stages. When the follower is processing $Entry_n$, the first stage

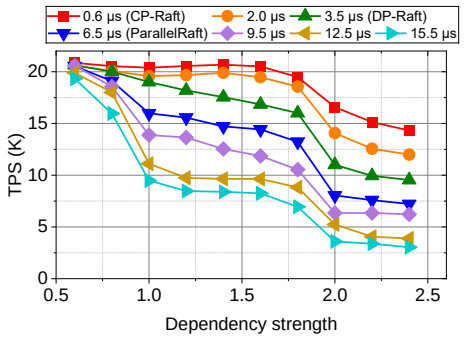


Fig. 12. As the time consumed by the dependency analysis rises, the TPS performance of the system decreases. In addition, we calculated the overhead incurred by the baseline approach. CP-Raft takes about 0.6 microseconds, DP-Raft takes about 3.5 microseconds, and ParallelRaft-CE (when N=2) takes about 6.5 microseconds when maximum DAR is 64.

determines whether it can be committed and executed out of order. If so, the follower performs the corresponding operation and updates its local index bitmap. In the second stage, all cached entries must be re-evaluated to check if they can now be processed following the bitmap update. The eligible entries are then committed and executed, triggering further bitmap updates. In our experiment environment, when the cluster runs at peak TPS performance, the average dependency analysis time cost per CP-Raft entry is 0.6 μs, while 3.5 μs for DP-Raft and 6.5 μs for ParallelRaft.

Dependency strength is controlled by the Zipfian distribution parameter s, which ranges from 0.6 to 2.4. When dependencies between log entries are weak (i.e., s is small), methods with higher computational overhead exhibit performance similar to those with lower overhead. However, as s increases, the performance gap becomes more pronounced. This is because, even if the dependency analysis time cost per entry is low (several microseconds), stronger dependencies result in more entries being cached (because more entries cannot be out-of-order processed) and frequent executions of the second stage. This amplifies the overall dependency checking overhead, leading to a decline in TPS performance.

2) Response latency: To understand the scalability and performance trade-offs of the protocol, we test the TPS-latency relationship. It helps identify the protocol's operational limits and its ability to maintain low-latency responses under high-concurrency workloads. In the experiment, we continually increase the number of clients to simulate the different workloads on 3, 5 and 7 node clusters. The experimental results are shown in Fig. 13.

When the cluster runs on low workloads, the both TPS and response latency are low, and the differences between baselines are small. In this condition, the entry processing ability of followers is higher than the arrival rate of entries from the leader. Therefore, even when the out-of-order processing method is applied, both the response latency and the TPS performance are not much improved. However, as we increase the client number, the sequential processing ability cannot cover the arrival rate of log entries. As a result, more requests will be cached by followers and accumulate until they fill up the cache space. At this point, the cluster reaches its maximum processing capacity. As we continue to increase the number of clients, replication requests that exceed the processing capacity are rejected by the follower and retransmitted until they are persisted to the follower. This results in the response latency of client requests increasing as the load increases after the TPS reaches its peak.

As the size of the cluster increases, the performance of TPS tends to decline due to the increased coordination overhead on the leader, as verified by existing works [37], [38]. With a larger cluster size, the log entry arrival rate of each follower decreases, and the number of entries that can be processed out of order is further reduced. However, CP-Raft outperforms all baselines, benefiting from its higher dependency analysis efficiency. Its lower response time enables more frequent leader replication, mitigating some of the performance degradation caused by scaling.

3) Dependency strength: Dependency Strength (DS) represents the probability of dependency between two log entries. To simulate state-machine modifications, such as file-write offsets, we design an operation generator following a Zipfian distribution. If two operations overlap in their modifications, a dependency is established. The parameter s of the Zipfian distribution ranges from 0.1 to 2.8. The experimental results are presented in Fig. 14.

The results show that TPS performance declines significantly as DS increases. When DS is extremely high, e.g., $s > 2.0$, more than 70% transactions operate on the same object. In this condition, OORaft protocols perform worse than vanilla Raft. Higher DS severely limits the number of entries that can be processed in parallel and reduces the follower's transaction processing efficiency. Given that transaction dependencies remain constant, refining the granularity of transactions per entry becomes crucial. However, accurately disentangling dependencies is challenging, especially when an entry contains multiple transactions. We will explore this issue further in Section VII.

4) Dependency analysis range (DAR): When out-of-order committing and applying are enabled, entries can be processed as long as their dependent entries are persisted. It allows followers to process these replication requests in parallel without queueing them sequentially. However, the DAR cannot be infinite because it generates an additional storage footprint in entries. Intuitively, the larger the DAR, the larger the number of transactions that the followers can process in parallel. Therefore, we experimented with the relationship between TPS performance and DAR of clusters under different latency environments and node sizes. The experimental results are shown in Fig. 15.

The results indicate that increasing the DAR significantly improves TPS performance under moderate skew workloads ($s = 0.8$), as a larger DAR enables followers to identify more log entries that can be processed out of order. However, under heavy skew workloads ($s = 1.6$), TPS initially increases but then declines as DAR continues to rise, particularly in DP-Raft and ParallelRaft. This decline occurs because higher DAR increases the computational cost of dependency analysis, as more entries must be checked in each analysis round.

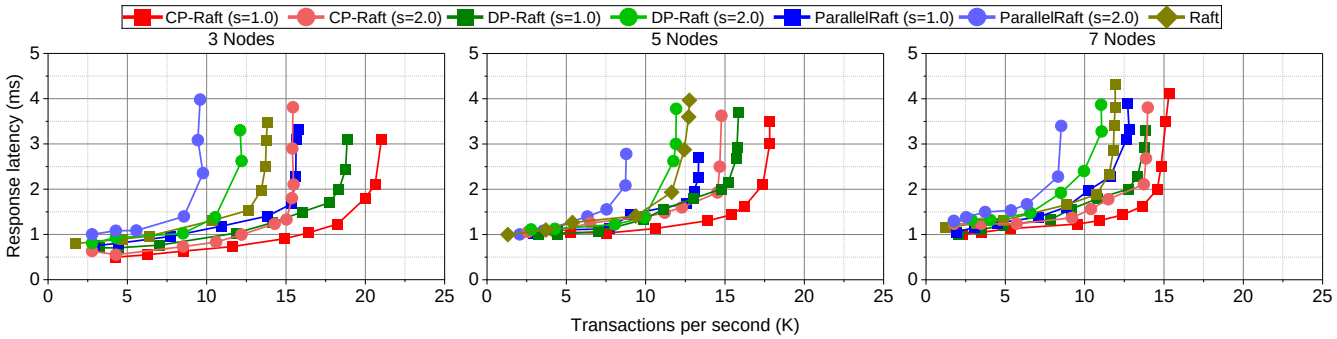


Fig. 13. We use two workloads to test the performance of baselines in different cluster size, which are $s = 1.0$ and $s = 2.0$. We keep increasing the number of clients until the TPS of cluster is not improving. The DAR parameter setting of DP-Raft and ParallelRaft are 64.

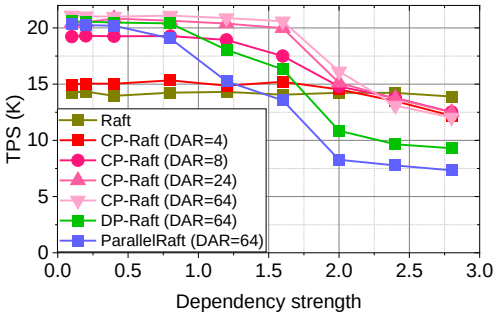


Fig. 14. We tested TPS performance of baselines under workloads with different dependeny strength. The DAR of DP-Raft and ParallelRaft is 64.

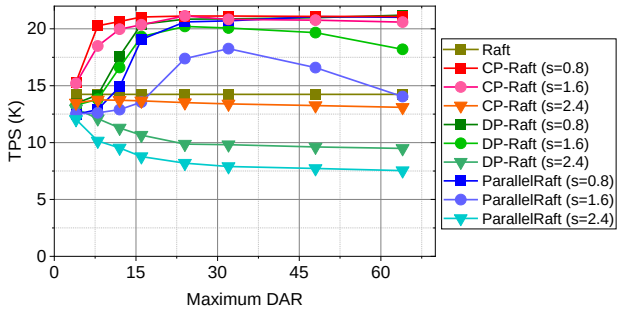


Fig. 15. We try different DAR to show the performance of baselines under different workloads. Specifically, we consider three representative workloads: $s = 0.8$ (moderate skew, typical of OLTP systems), $s = 1.6$ (heavy skew, common in e-commerce platforms), and $s = 2.4$ (extreme skew, characteristic of hotspot key-value stores). These scenarios help assess how different skew levels impact protocol efficiency and scalability.

When the performance gains from a broader analysis range are outweighed by the overhead of dependency analysis, TPS performance decreases.

For extreme skew workloads ($s = 2.4$), OORaft fails to provide performance improvements and instead reduces TPS as DAR increases. In these cases, most log entries must be processed sequentially, and a higher DAR only introduces additional overhead without yielding significant benefits. This experiment highlights the importance of careful DAR selection and explains why ParallelRaft recommends a small N parameter ($N = 2$). Notably, CP-Raft consistently outperforms other OORaft baselines due to its lower dependency analysis overhead, demonstrating its efficiency in handling dependencies.

5) Overall comparisons with baselines: To provide a comprehensive performance comparison with the baseline methods, we run these methods under different parameter settings and workloads. The results are shown in Fig. 16.

Across different workloads, CP-Raft consistently outperforms other OORaft protocols. Even under extremely skewed conditions where most entries must be executed sequentially ($s = 2.4$), CP-Raft maintains competitive performance without severe degradation compared to Raft. In contrast, ParallelRaft and DP-Raft suffer a 30%-50% performance drop under heavily skewed workloads. Furthermore, with larger DAR values, ParallelRaft and DP-Raft exhibit significant sensitivity to workload variations, experiencing 60% and 50% performance losses compared to their peak performance. In comparison, CP-Raft demonstrates only a 25% performance decline, highlighting its greater stability and adaptability to different workload patterns.

Regarding DAR parameter selection, CP-Raft exhibits the least sensitivity to changes in DAR among all baseline methods. Whether DAR is set to 32 or 64, its performance remains relatively stable. In contrast, ParallelRaft and DP-Raft show a stronger correlation between performance and DAR variations. While selecting a smaller DAR can help mitigate performance degradation under highly skewed workloads ($s \geq 2.0$), it also results in poorer performance under normal workloads ($s \leq 1.6$). This trade-off makes DAR selection more challenging for these baseline methods.

The experimental results show that CP-Raft consistently outperforms existing OORaft protocols on diverse workloads under different DAR parameters. Even in extremely skewed workloads, CP-Raft can still exhibit comparable performance to Raft. For example, when $s \geq 2.0$ with properly tuned parameters, CP-Raft achieves up to 1.5 × the performance of DP-Raft and 2 × that of ParallelRaft. Given the variability of runtime dependencies in data-intensive applications, a protocol that maintains stable performance across different workload conditions is crucial for preventing system performance degradation and ensuring reliability.

B. Experiments on Availability

To better understand the availability issues discussed in Section II.C and Table I, we simulate server failures and

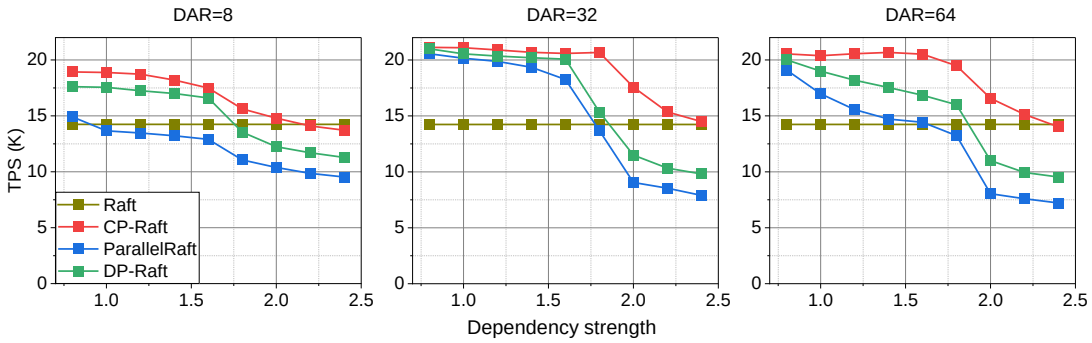


Fig. 16. We compare the TPS performance of CP-Raft with baseline methods on a 3-node cluster. The workloads are simulated by using different Zipfian parameter settings, which range from 0.8 to 2.4. We also try different dependency analysis parameter settings including DAR=8, DAR=32, and DAR=64.

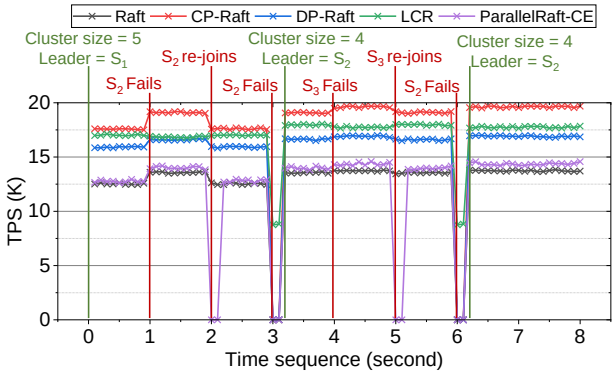


Fig. 17. The availability experiment result. When TPS drops to 0, the cluster stops providing service and is unavailable. Configuration parameters of the cluster is DAR=64, $s = 1.0$.

analyze their impact on TPS performance as an indicator of cluster availability. At $t = 0$, the cluster consists of five nodes, with S1 as the leader. At $t = 1$, follower node S2 fails and temporarily leaves the cluster, rejoining at $t = 2$. At $t = 3$, the leader node S1 fails and does not rejoin, triggering a leader election where S2 becomes the new leader, reducing the cluster size to four. At $t = 4$, follower node S3 fails but rejoins at $t = 5$. At $t = 6$, the leader node S2 fails, reducing the cluster size to three, with S3 elected as the new leader. The experimental results in Fig. 17 show performance fluctuations caused by leader transitions and node failures, highlighting the impact of availability on system stability.

From the experiment, we verify that ParallelRaft-CE meets the availability problem when failed follower nodes re-join the cluster. It makes the cluster face an increased risk of unavailability. Let every node have the same failure probability. The unavailable time of the ParallelRaft cluster is the node-size times of CP-Raft, which matches the analysis in Fig. 4. We also notice that the TPS performance of Raft, CP-Raft, DP-Raft, and ParallelRaft-CE increases when the alive node number decreases. The reason is the same as the analysis in the 'Response latency' section and Fig. 13. However, LCR does not show this trend because more active nodes enable more 'Future entries' to be processed in parallel.

For the election time, all of the OORaft protocols have similar costs. In detail, the first pre-candidate node shows OORaft needs 20 ms on average to elect a new leader (the cost of timeout is not counted because it is not included in the election process.) The time cost is higher than the Raft protocol, which is 16.5 ms on average. The reason is that OORaft protocols need to merge and check gap entries. Although OORaft protocols do not need more election phases, they still generate overhead on resolving and checking processes.

VII. Discussions

A. TPS performance bottlenecks of OORaft

Experimental results and analysis indicate that the TPS performance of Raft and OORaft is influenced by various factors, including cluster size, DAR, dependency analysis methods, and workload characteristics. In this section, we discuss the primary performance bottlenecks:

1) **Dependency analysis complexity**. Many OORaft and Paxos-based protocols propose different dependency representations and analysis methods based on replication policies. However, existing research lacks a thorough investigation of the trade-off between dependency computation complexity and replication efficiency. For example, in ParallelRaft, each entry's logical block address (LBA) range must be checked against all reachable entries (as missing sequential entries may break the dependency chain). DP-Raft, on the other hand, maintains an N-length index list of conflicting entries, requiring recurrent checks upon each entry's arrival. Some methods, like LCR, avoid dependency analysis by processing only non-transactional entries out of order, while others, such as EPaxos, do not provide a detailed dependency analysis strategy. Experimental results confirm that dependency analysis complexity directly impacts TPS performance, as the locking mechanism introduced during this process increases contention, reduces parallel processing efficiency, and degrades performance.

2) **Dependency analysis range**. A higher DAR enables the discovery of more out-of-order processable entries, thereby enhancing TPS performance. However, this also increases dependency analysis overhead, as a larger range requires more storage for dependency identifiers and leads to greater processing complexity—especially in methods using payload-related dependency representations. Therefore, choosing an optimal DAR value is crucial for maximizing protocol performance. For instance, ParallelRaft recommends a default setting of

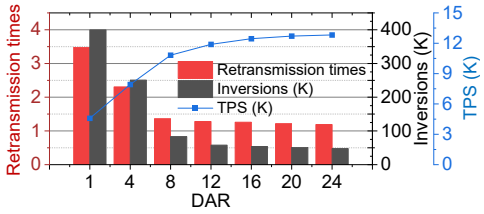


Fig. 18. The experiment runs at a 5-node cluster. DAR ranges from 1 to 24. With the increase of DAR, re-transmission times and inversions significantly decrease. Besides, they show a negative correlation with TPS performance.

$N = 2$. Advanced adaptive parameter tuning mechanisms could further optimize DAR selection based on workload conditions.

3) **Dependency strength of the workload**. Dependency strength is influenced not only by transaction dependencies but also by how transactions are packed into log entries. Packing more transactions into a single entry increases the likelihood of dependencies across entries, reducing parallelism. Conversely, smaller entries lead to increased replication frequency, imposing higher communication overhead. Thus, achieving an optimal balance between DS and the number of transactions per entry is essential for maximizing performance. Workload-aware transaction packing strategies hold promise as an effective optimization technique for OORaft.

4) **Re-replications on the follower node**. In prior studies, re-replication of entries is an overlooked factor that limits processing efficiency. When an entry is replicated to a follower with a full cache, it is rejected and must be re-replicated by the leader. This disrupts the sequential arrival of entries, further impacting TPS performance. For example, if a follower rejects $Entry_{10}$ due to insufficient cache space while receiving $Entry_{11}$ to $Entry_{20}$ in parallel, it may later process $Entry_{11}$ before Entry 10 is re-replicated, leading to replication disorder. Fig. 18 shows the relationship between re-transmission times, entries arrival sequence and TPS performance. Minimizing re-replications and optimizing cache utilization are critical to improving performance. Techniques like incremental encoding that can increase cache space usage efficiency are promising optimizations for OORaft protocols.

B. Problems in the election process of OORaft

Easy-to-understand is a key feature of Raft that makes it the most popular consistency protocol. Therefore, OORaft should also follow this design concept as a variant of Raft protocols. Existing approaches ignore this design concept and focus on using Paxos progress and entry parameters to achieve gap entry recovery correctly. This makes the protocol lose both the ease of understanding of Raft and the flexibility of Paxos.

Also, the existing research design did not address availability and ghost logs issues. The reason is that after introducing the Paxos-based log entries merging strategy, they did not notice that Paxos-progress could not solve these issues. Therefore, the design of future protocols should ensure that these two issues are addressed if Paxos progress is used.

C. Correctness of OORaft

We proved that traditional approaches cannot fully verify the correctness of the OORaft protocol using the traditional TLA$^+$ method. Therefore, phased methods are the feasible way to achieve the verification task. It saves verification time and ensures the fullness of the critical process's verification.

However, it also has certain limitations. First of all, although it verifies critical issues, it cannot verify the status of the entire agreement. This method can only verify the protocols with little modifications. When the protocols have changed a lot, the complete verification process is still necessary. Secondly, we need to make more efforts to design use cases of key issues.

VIII. Conclusion

This paper proposes a CP-Raft protocol with a Raft-aligned three-step election method, significantly simplifying the understanding and implementation difficulty. CP-Raft uses a leader-side bitmap-based dependency analysis method to improve parallel replication efficiency. Further, we discuss why existing methods cannot achieve OORaft correctness verification using TLA$^+$ in a limited time and use phased methods to verify CP-Raft. Finally, we implement CP-Raft based on an open-sourced Raft protocol and discuss its potential performance bottlenecks in various scenarios. Experimental results demonstrate that CP-Raft: (1) resolves the availability issues observed in ParallelRaft-CE and mitigates the ghost-log problem in DP-Raft; (2) achieves up to 50% higher transaction throughput (TPS) than DP-Raft and 100% higher than ParallelRaft-CE under high dependency strength (DS) workloads; and (3) completes critical feature verification within 12 minutes using the proposed phase-based TLA$^+$ method.

References

[1] Y. Jia, G. Xu, C. W. Sung, and S. Mostafa, "Adaptive erasure coded data maintenance for consensus in distributed networks," in *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2021, pp. 345–346.
[2] ETCD, "Etcd," https://github.com/etcd-io/etcd, accessed: 2024-05-30.
[3] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss *et al.*, "Cockroachdb: The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1493–1509.
[4] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang *et al.*, "Tidb: a raft-based htap database," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3072–3084, 2020.
[5] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.
[6] H. Dai, Y. Wang, K. B. Kent, L. Zeng, and C. Xu, "The state of the art of metadata managements in large-scale distributed file systems—scalability, performance and availability," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3850–3869, 2022.
[7] F. Al-Doghman, N. Moustafa, I. Khalil, N. Sohrabi, Z. Tari, and A. Y. Zomaya, "Ai-enabled secure microservices in edge computing: Opportunities and challenges," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1485–1504, 2022.
[8] J. Leng, M. Zhou, J. L. Zhao, Y. Huang, and Y. Bian, "Blockchain security: A survey of techniques and research directions," *IEEE Transactions on Services Computing*, vol. 15, no. 4, pp. 2490–2510, 2020.
[9] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.

[10] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[11] M. Dabbagh, K.-K. R. Choo, A. Beheshti, M. Tahir, and N. S. Safa, "A survey of empirical performance evaluation of permissioned blockchain platforms: Challenges and opportunities," *computers & security*, vol. 100, p. 102078, 2021.

[12] Z. Wang, C. Zhao, S. Mu, H. Chen, and J. Li, "On the parallels between paxos and raft, and how to port optimizations," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 445–454.

[13] H. Li, Z. Liu, and Y. Li, "An improved raft consensus algorithm based on asynchronous batch processing," in *International Conference on Wireless Communications, Networking and Applications*. Springer, 2021, pp. 426–436.

[14] X. Gu, H. Wei, L. Qiao, and Y. Huang, "Raft with out-of-order executions." *Int. J. Softw. Informatics*, vol. 11, no. 4, pp. 473–503, 2021.

[15] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma, "Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1849–1862, 2018.

[16] Z. Zhang, H. Hu, Y. Yu, W. Qian, and K. Shu, "Dependency preserved raft for transactions," in *Database Systems for Advanced Applications: 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24–27, 2020, Proceedings, Part I 25*. Springer, 2020, pp. 228–245.

[17] J. Xu, W. Wang, Y. Zeng, Z. Yan, and H. Li, "Raft-plus: Improving raft by multi-policy based leader election with unprejudiced sorting," *Symmetry*, vol. 14, no. 6, p. 1122, 2022.

[18] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, 2007, pp. 398–407.

[19] H. Howard and R. Mortier, "Paxos vs raft: Have we reached consensus on distributed consensus?" in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, 2020, pp. 1–9.

[20] D. Ongaro, *Consensus: Bridging theory and practice*. Stanford University, 2014.

[21] H. Du, D. Zhu, Y. Sun, and Z. Tian, "Leader confirmation replication for millisecond consensus in private chains," *IEEE Internet of Things Journal*, vol. 9, no. 11, pp. 7944–7958, 2021.

[22] S. M. Pedersen, "A practical analysis of the gorums framework: A case study on replicated services with raft," Master's thesis, University of Stavanger, Norway, 2017.

[23] D. Wang, P. Cai, W. Qian, A. Zhou, T. Pang, and J. Jiang, "Fast log replication in highly available data store," in *Web and Big Data: First International Joint Conference, APWeb-WAIM 2017, Beijing, China, July 7–9, 2017, Proceedings, Part II 1*. Springer, 2017, pp. 245–259.

[24] T. Jiang, X. Huang, S. Song, C. Wang, J. Wang, R. Li, and J. Sun, "Non-blocking raft for high throughput iot data," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 1140–1152.

[25] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[26] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan *et al.*, "{POLARDB} meets computational storage: Efficiently support analytical workloads in {Cloud-Native} relational database," in *18th USENIX conference on file and storage technologies (FAST 20)*, 2020, pp. 29–41.

[27] J. WANG, B.-h. LI, J.-j. WU, and X.-y. SONG, "Distributed consistency protocol supporting log submission out-of-order," *Journal of ZheJiang University (Engineering Science)*, vol. 57, no. 2, pp. 320–329, 2023.

[28] D. Ren, J. Tu, and W. Xie, "An improved raft protocol combined with cauchy reed-solomon codes," in *2022 5th International Conference on Artificial Intelligence and Big Data (ICAIBD)*. IEEE, 2022, pp. 563–568.

[29] H. Howard, "Arc: analysis of raft consensus," University of Cambridge, Computer Laboratory, Tech. Rep., 2014.

[30] J. Hu and K. Liu, "Raft consensus mechanism and the applications," in *Journal of physics: conference series*, vol. 1544, no. 1. IOP Publishing, 2020, p. 012079.

[31] Y. Yu, P. Manolios, and L. Lamport, "Model checking tla+ specifications," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 1999, pp. 54–66.

[32] D. Ongaro, "Raft.tla," https://github.com/ongardie/raft.tla, accessed: 2024-05-30.

[33] S. Irfan, "Raft.tla," https://github.com/irfansharif/raft.tla, accessed: 2024-05-30.

[34] M. A. Kuppe, L. Lamport, and D. Ricketts, "The tla+ toolbox," *arXiv preprint arXiv:1912.10633*, 2019.

[35] Baidu, "Braft," https://github.com/baidu/braft.git, accessed: 2025-04-01.

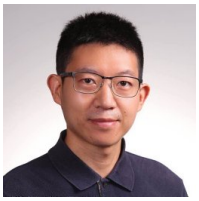
[36] P. Fouto, N. Preguiça, and J. Leitão, "High throughput replication with integrated membership management," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 575–592.

[37] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for wans," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 369–384.

[38] A. Charapko, A. Ailijiang, and M. Demirbas, "Pigpaxos: Devouring the communication bottlenecks in distributed consensus," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 235–247.

Haiwen Du is a Lecturer with the School of Computer Science and Technology, Harbin Institute of Technology, Weihai, China. He received the B.S., M.S. and Ph.D. degrees from the Harbin Institute of Technology, Harbin, China. His research interests include distributed data storage systems and massive data management.

Kai Wang (Member, IEEE) is currently a professor with the School of Computer Science and Technology, Harbin Institute of Technology (HIT), Weihai. He received the B.S. and Ph.D. degrees from Beijing Jiaotong University. He has published more than 40 papers on IEEE TITS, IEEE TCE, ACM TOIT, ACM TIST, etc. His current research interests include applied machine learning for network attack detection and information forensics. He is a Member of the IEEE and ACM, and a Senior Member of the China Computer Federation (CCF).



Yulei Wu (Senior Member, IEEE) is an Associate Professor with the Faculty of Engineering and the Bristol Digital Futures Institute at University of Bristol, UK. He received his Ph.D. degree in Computing and Mathematics and B.Sc. (1st Class Hons.) degree in Computer Science from the University of Bradford, United Kingdom. His research mainly focuses on network digital twins, native AI networks and systems, edge AI, and trustworthy AI. He has published over 10 authored/edited monograph books, and over 150 peer-reviewed research papers in prestigious international journals and conferences. Dr. Wu serves as an Associate Editor of IEEE TNSM and IEEE TNSE, as well as an Editorial Board Member of Computer Networks, Future Generation Computer Systems, and Nature Scientific Reports at Nature Portfolio. He is chairing an IEEE Special Interest Group (SIG) on Ethical AI for Future Networks and Digital Infrastructure. He is a Senior Member of the ACM.

Hongke Zhang (Fellow, IEEE) is currently a Professor with the School of Electronic and Information Engineering, Beijing Jiaotong University, Beijing, China, where he currently directs the National Engineering Center of China on Mobile Specialized Network. He received the Ph.D. degree in communication and information system from the University of Electronic Science and Technology of China, Chengdu, China, in 1992. His current research interests include architecture and protocol design for the future Internet and specialized networks. He currently serves as an Associate Editor for the IEEE TNSM and IEEE Internet of Things Journal. He is an Academician of China Engineering Academy.