RL-Paxos: Relieving the Leader's Burden with Efficient Task Offloading in Distributed Consensus

Chenhao Zhang[a,e,f], Jinquan Wang[b], Meng Han[c], Bing Wei[d], Xiaojian Liao[b,*], Limin Xiao[b], and Shanchen Pang[a,e,f]

[a] College of Computer Science and Technology, China University of Petroleum (East China), Qingdao, China
[b] State Key Laboratory of Software Development Environment, Beihang University, Beijing, China
[c] School of Integrated Circuits, Tsinghua University, Beijing, China
[d] School of Cyberspace Security, Hainan University, Hainan, China
[e] State Key Laboratory of Chemical Safety, China University of Petroleum (East China), Qingdao, China
[f] Shandong Key Laboratory of Intelligent Oil & Gas Industrial Software, Qingdao, China
zch13021728086@buaa.edu.cn, liaoxj@buaa.edu.cn

***Abstract*—Existing state machine replication protocols are typically leader-based, with the leader responsible for all operations, leading to a scalability bottleneck. Despite extensive research efforts to address this issue, alleviating the leader's workload remains a significant challenge. In this paper, we provide an in-depth analysis of the leader bottleneck and introduce RL-Paxos, which retains only the core function of ordering (e.g., establishing a global order and performing commutativity checks) for the leader, while offloading other tasks to the followers. We integrate RL-Paxos into the classic Paxos protocol and compare it with state-of-the-art protocols. Extensive experimental evaluations demonstrate that RL-Paxos achieves high throughput and low wide-area latency under conditions of high concurrency, multi-node deployment, mixed workloads, and read-only scenarios.**

***Index Terms*—State machine replication, Paxos, Leader bottleneck, Latency, Performance**

I. Introduction

State machine replication (SMR) is the cornerstone of fault-tolerant distributed systems and services [7], [15], [18], [32], [38]. The consensus protocol realizes the linearization and fault tolerance of the SMR. Linearizability enables users to interact with distributed clusters as if they were standalone nodes, while fault tolerance ensures the system remains operational despite partial node failures or network delays.

Existing SMR implementations predominantly rely on leader-based consensus protocols (e.g., Paxos [22], Raft [30], Zab [20]). The leader simplifies consistency management and eases implementation, which has led to its widespread adoption in practical systems. However, this leader-centric design places a disproportionate workload on the leader, limiting parallel processing and resulting in scalability and performance bottlenecks. Moreover, for geo-distributed clients, this centralized approach introduces significant latency penalties. As a result, improving the performance and scalability of replication systems requires alleviating the leader's workload.

A series of works have been proposed to reduce the load on the leader [6], [8], [9], [41]. Some systems [17], [33], [40] introduce follower reads to offload read requests to follower nodes. Other approaches aim to alleviate the leader bottleneck using various techniques, such as separating sequencing from replication [6], [41], employing proxy leaders [36], [39], utilizing Paxos quorum reads [8], and leveraging communication aggregation and piggybacking [9]. Unfortunately, these methods either compromise linearizability [33], [40] or fail to fully offload the leader's responsibilities (details in §II-B), thereby offering sub-optimal performance and scalability, especially when the leader becomes overloaded. Taking Pig-Paxos [9] and SD-Paxos [41] as examples, this paper identifies three key issues in current approaches.

First, existing methods for offloading leader tasks are not comprehensive enough, and the offloading itself may introduce additional overhead. For example, while SD-Paxos offloads replication to followers, it introduces additional types of consensus messages, which in turn increase the leader's workload by approximately 23.3% (Limitation 1, §II-B).

Second, existing methods have high request latency in geo-replicated environment. For example, Pig-Paxos selects proxy leaders in the region to communicate with followers, which does not change the long request path (Pig-Paxos does not reduce the inherent wide-area latency of Paxos. In fact, it may even increase the request path length due to the rotation of proxy leaders.) of Paxos, but makes fault recovery more complicated. SD-Paxos has a request latency 53.85% higher than ideal due to inefficient task distribution between leaders and followers (Limitation 2, §II-B).

Third, existing leader-based consensus protocols process all requests sequentially, severely limiting the system's ability to execute requests in parallel (Limitation 3, §II-B).

To alleviate the leader's burden and improve the performance of consensus protocols, we introduce RL-Paxos, a new consensus protocol for distributed systems (§III). The core idea of RL-Paxos is to have a lightweight leader re-

We thank the anonymous reviewers for their feedback. This work was supported by the National Key R&D Program of China under Grant NO. 2023YFB4503100, 2021YFA1000102 and 2021YFA1000103; the project ZR2025QC1540, ZR2025QC1546 supported by Shandong Provincial Natural Science Foundation; the National Natural Science Foundation of China (62502540).

* Xiaojian Liao is the corresponding author.

sponsible only for ordering (e.g., establishing a global order, performing commutativity checks). All other tasks, including linearizable reads, receiving messages, replying to clients, and asynchronously notifying other nodes to commit, are fully offloaded to the followers. Followers can independently monitor the consensus process, determine consensus status autonomously, and respond to requests from geographically proximate clients, significantly reducing the burden on the leader.

Additionally, to overcome the bottleneck of serial execution in consensus protocols, RL-Paxos introduces a lightweight out-of-order execution technique. In addition to determining the global order, the leader checks the commutativity of requests within a predefined window, and sends this information to the followers, allowing them to process these requests out of order. While this technique adds some workload to the leader, it enhances overall concurrency and helps avoid prolonged blocking caused by request loss in uncertain WAN (wide-area network) environments.

We integrate RL-Paxos into the framework of the classic Paxos protocol to support SMR in various systems. The implementation retains the core principles of Paxos while modifying the replication phase logic. As a result, the optimization techniques in RL-Paxos are applicable to many centralized Paxos variants, including Multi-Paxos [13] and Raft [30]. Additionally, RL-Paxos preserves the same fault tolerance as the classical Paxos protocol.

We evaluate RL-Paxos in WAN deployments across five regions and compare its performance against the state-of-the-art leader-based protocols such as SD-Paxos and Pig-Paxos (§IV). Our experiments under various conditions demonstrate that RL-Paxos effectively improves scalability and reduces latency in WAN. In particular, under high concurrency conditions (600 clients), RL-Paxos achieves a throughput increase of 21.46% and 33.48% compared to SD-Paxos and Pig-Paxos, along with a reduction in average latency by 16.67% and 25.82%. When the number of nodes is 11, RL-Paxos exhibits a 65.86% increase in average throughput and a 45.11% reduction in latency compared to SD-Paxos. Under mixed load conditions (write ratios=25%), the throughput of RL-Paxos is 2.47 $\times$ and 2.85$\times$ that of SD-Paxos and Pig-Paxos, respectively. For read-only loads, the throughput of RL-Paxos doubles compared to SD-Paxos and Pig-Paxos.

In summary, we make the following contributions:

- We identify throughput bottlenecks of existing leader-based consensus protocols through comparative experiments and analysis.
- We design and implement RL-Paxos, with a suite of techniques to fully offload the leader's burden.
- We conduct evaluations to demonstrate performance improvement over existing consensus protocols.

II. Background and Motivation

A. Paxos Overview

Paxos has been synonymous with distributed consensus [16]. The classic Paxos protocol can be divided into three phases.

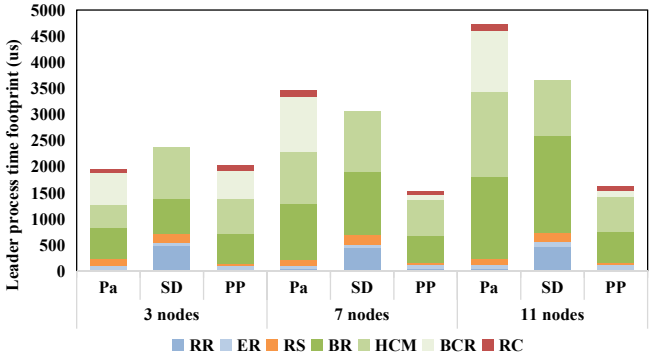

Fig. 1. Latency breakdown of various tasks performed by the leader. **Pa**: Paxos [22], **SD**: SD-Paxos [41], **PP**: Pig-Paxos [9]. Blue indicates the necessary load, which includes RR, BR, and ER. Green represents the additional load, encompassing tasks including BR (partially), HCM, RC and BCR. Orange highlights the leader's core responsibility: establishing a global order for requests.

In the first phase (the *Prepare* phase or leader selection phase), a node (known as the proposer), tries to acquire leadership by sending a preparation request to the other nodes. The request includes a proposal number, and replicas respond with a promise not to accept any proposals numbered lower than the one provided. If the proposer receives a majority of promises, it can proceed to the next phase.

In the second phase (the *Accept* phase or copying phase), the leader tells all followers to accept the request, which can be a new request chosen by the leader, or a request that has been approved but not submitted. A consensus is reached if a majority of nodes accept the proposal.

In the third phase (the *Learn* phase), the leader submits the request, which needs to be learned by all nodes in the system. The leader tells all followers the consensus result to learn, and then performs a state transfer operation.

The basic Paxos protocol is often extended to Multi-Paxos [22], which optimizes the algorithm to handle multiple consensus instances without repeating the entire process for each request. Multi-Paxos introduces a leader that can continuously propose values, reducing communication overhead and improving efficiency, making it more practical for real-world distributed systems. In this paper, references to Paxos imply the Multi-Paxos optimization.

Tasks of the leader can be divided into the following parts:

- **Receiving Request (RR)**: Receive client requests
- **Request Sorting (RS)**: Sort client requests and broadcast message to the other n followers
- **Broadcasting Request (BR)**: Send the sorted message to followers
- **Handling Consensus Message (HCM)**: Receive and process reply messages from all followers
- **Executing Request (ER)**: Commit the consensus requests in order, and then execute them sequentially
- **Responding Client (RC)**: After successfully executing the request, respond to the client

- **Broadcasting Consensus result (BCR)**: Asynchronous broadcast consensus result (e.g. Phase-3 in Paxos)
- **Handling Read Requests**[1] **(HRR)**: Receives and responds to client read requests.

B. Limitations of Existing Leader-based Distributed Consensus

The leader inevitably becomes the performance bottleneck of distributed systems, especially in remote replication scenarios, as all read and write operations must pass through the leader. To address the leader bottleneck, many protocols for remote replication have been proposed (Table I). In this section, we begin by analyzing Paxos variants, as Paxos is one of the most widely used consensus protocols, with nearly all other leader-based consensus protocols derived from it. We conduct the experiments in a 11-node geo-replicated environment. Detailed experimental setup is presented in §IV. Table I summarizes the compared systems and the corresponding tasks performed by their leaders. SD-Paxos, Pig-Paxos, and their variants offload a portion of tasks from the leader, positioning them as our closest competitors. In contrast, other systems, such as WAN-Paxos, D-Paxos, RS-Paxos, and CRaft, do not reduce the leader's load, resulting in significant overhead when the leader becomes overloaded, thereby limiting their scalability. This study focuses on the closest and most advanced competitors, represented by three systems: vanilla Paxos, SD-Paxos (highlighting sort and replication separation), and Pig-Paxos (featuring the Proxy Leader design).

Limitation 1: *Excessive workloads on the leader and inefficient offloading. In existing leader-based protocols, leaders spend 66% of their time on non-essential tasks.*

Earlier in §II-A, we show the tasks of the leader in vanilla Paxos protocol. Fig.1 shows the time required to process each task. The original Paxos protocol, serving as the baseline, spent over 90% of the total processing time on sending, receiving, and processing messages from other nodes in the cluster (represented by the green portion in the figure). These tasks, which are non-critical tasks during the consensus execution, are identified as overhead. Worse still, the overhead increases sharply as the cluster size grows, consuming significant leader resources and ultimately causing the leader to become a system bottleneck.

A group of works (e.g., Pig-Paxos [9] and SD-Paxos [41]) optimizes Paxos by offloading a portion of tasks to followers. Pig-Paxos uses Proxy Leader to reduce the load on the original leader. Instead of broadcasting requests to all followers, the original leader only needs to redirect the request to the Proxy Leader, hence significantly reducing the communication overhead with communication aggregation [9], [36], [39]. SD-Paxos [41] and S-Paxos [6], leverages the separation of sorting and replication to alleviate the overhead of the leader. Both kinds of works significantly reduce the resource consumption of the leader (Fig. 1). For example, the leader of Pig-Paxos maintains almost constant resource consumption as the cluster scales, reducing the additional load to 61%. However, their offloading is inadequate; we observe that there is still a large portion of non-critical tasks (green parts in Fig. 1) in the consensus execution.

[1]In this paper, we only consider linearizable reads for a given request.

Client request process	Pig-Paxos	SD-Paxos	
Request Receiving	33	441	Necessary task
Request Sorting	50	183	
Request Execution	76	66	
Request Broadcast	523	503+712*	
Receive Consensus Message	410	932	Additional task PP: 61% SD: 75%
Handling Consensus Message	290	224	
Response Client	92	0	
Broadcast Consensus result	530	0	
Total request execution time	2004	3061	

Fig. 2. Latency (in microseconds) breakdown of Pig-Paxos and SD-Paxos under a 7-node setup. *: although SD-Paxos eliminates the task of broadcasting requests, its leader needs to broadcast another message and send additional messages.

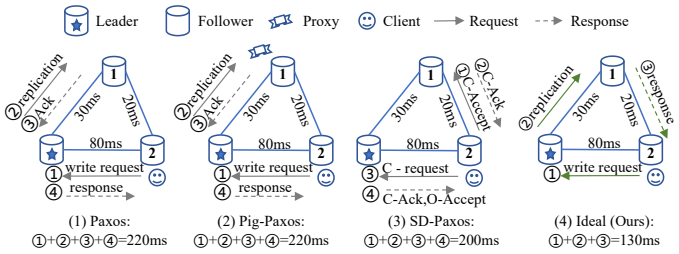

Fig. 3. Comparison of request paths of different protocols in wide-area environments.

What's worse, the offloading of SD-Paxos introduces additional side-effects. Compared to the Paxos, which only requires processing two consensus message types (*P2a, P2b*), the leader of SD-Paxos must handle more messages (*O-Accept, O-ACK, C-Accept, C-ACK, C-Commit, O-Commit*) associated with sort and replication, increasing the burden on the leader. We observe that SD-Paxos spends 1.4× more time in Request Broadcast due to the extra messages (Fig. 2).

Limitation 2: *High request latency in geo-replicated environment. The overall latency increases significantly when clients and the leader are located in different regions, primarily due to inefficient tasks assignment between leaders and followers.*

In Paxos, the data path (Fig. 3) typically consists of four steps: ①the client first sends data to the leader node, ②the leader copies the sorted request to all follower nodes, ③and after receiving a successful response from the majority of nodes, ④the leader sends a completion message to the client. In the example shown in Fig. 3, it requires about 220 ms for Paxos to complete a request.

In Pig-Paxos, the proxy leader is located in the same region as the followers, so the intra-region latency is much smaller than the inter-region latency and can be considered negligible. While the proxy leader collects responses from the followers

TABLE I

COMPARISON BETWEEN RL-PAXOS AND REPRESENTATIVE LEADER-BASED PROTOCOLS IN GEO-REPLICATED ENVIRONMENTS. ★ DENOTES THE LEADER'S TASK, WHILE ⊗ REPRESENTS THE FOLLOWER'S TASK. ⊗* INDICATES THAT ALTHOUGH SD-PAXOS ELIMINATES THE TASK OF BROADCASTING A REQUEST, A DIFFERENT KIND OF MESSAGE (O-ACCEPT) STILL NEEDS TO BE BROADCAST.

Protocol	RR	RS	BR	HCM	ER	RC	BCR	HRR
Paxos [22], Raft [30]	★	★	★	★	★	★	★	★
SD-Paxos [41], S-Paxos [6]	★	★	⊗*	★	★	⊗	⊗	★
Pig-Paxos [9], CP-Paxos [39], Pando [36]	★	★	★	★	★	★	★	★
WAN-Paxos [2], [3], D-Paxos [29]	★	★	★	★	★	★	★	★
RS-Paxos [28], CRaft [37]	★	★	★	★	★	★	★	★
RL-Paxos (ours)	★	★	★	⊗	★	⊗	⊗	⊗

within its region, it must relay these responses to the original leader within a specified time frame, allowing the original leader to determine whether a majority has been reached. As a result, the request latency of Pig-Paxos is typically longer than that of standard Paxos.

In SD-Paxos, the leader shifts the replication task to follower 2, and then the replication process is performed among the followers in one Paxos round (①+②). In another Paxos round, as the ordering is guaranteed by the leader, the follower 2 needs to communicate with the leader (③+④). In summary, SD-Paxos takes nearly 200 ms to finish a request.

Yet, we find that the request latency in both Pig-Paxos and SD-Paxos remains far from ideal, primarily due to inefficient task allocation between leaders and followers. For example, in Pig-Paxos, the Proxy Leader frequently communicates across domains with the original leader, thus inheriting the shortcomings of the original Paxos. In SD-Paxos, separating sorting and replication adds extra message types and requires both processes to independently go through the Paxos protocol, resulting in multiple communication rounds. RL-Paxos optimizes this by offloading non-sorting tasks to followers, avoiding extra message types and allowing followers to independently monitor the consensus processthus eliminates multiple voting rounds, improving communication efficiency and reducing latency compared to SD-Paxos and Pig-Paxos.

In the ideal path envisioned by RL-Paxos, the leader ensures the ordering and forwards the sorted request to the closest follower (①+②). The follower then performs replication and exchanges status among the other followers. Once a follower learns that the request has been received by the majority of nodes in the cluster, it responds directly to the client (③). The ideal path only takes nearly 130 ms.

Limitation 3: *Non-scalable request execution. Existing leader-based consensus protocols employ strict serial request execution, which incurs high tail latency in WAN.*

Leader-centric Paxos variants typically rely on the leader executing requests serially. For instance, Raft is designed to be highly serialized for simplicity and understandability. As illustrated in Table I, nearly all existing leader-based replication protocols in wide-area environments are serial-executed protocols. However, message out-of-order arrival and packet loss in WAN are much more common than in local area networks (LAN). As a result, these serial-executed protocols experience significant tail delays in the presence of network

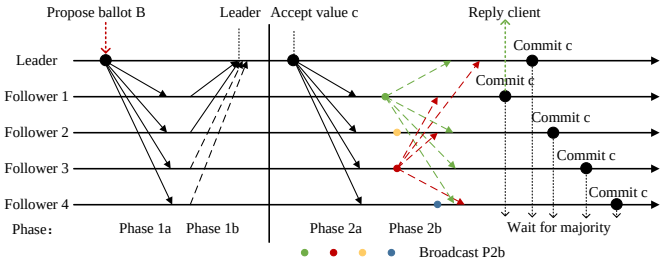

Fig. 4. RL-Paxos communication flow. The leader is only responsible for ordering, and all the followers exchange their own consensus status information, reach consensus and respond to the client on their own.

fluctuations (see Figures 10 and 11 for more details). In addition, modern hardware has expanded to chips with dozens of cores [14], and these single-threaded protocols clearly cannot benefit from massive hardware-level parallelism.

III. RL-PAXOS DESIGN AND IMPLEMENTATION

A. RL-Paxos Overview

We propose RL-Paxos to address the limitations identified in existing work in §II-B. The core design of RL-Paxos involves decoupling the ordering function from the consensus process , with the leader focusing solely on tasks related to ordering such as establishing the global order and performing commutativity checks. Additionally, RL-Paxos (Fig.4) aims to retain implementation simplicity and avoid the complex runtime arbitration required by protocols such as EPaxos [27] or CAESAR [4].

To support scalable replication under varying network conditions, RL-Paxos implements a *Leaderless Commit Relay* and an *Enhanced Commit Path with Topology-Aware Coordination*. In the normal case, followers broadcast P2b messages among each other to quickly form quorums. Furthermore, followers autonomously monitor the replication progress of their local requests and independently reach the committed state, without requiring the leader's reminder phase (e.g., P3 phase in Paxos [22], committedIndex in Raft [30]). The first follower to reach consensus can promptly respond to the client. However, when bandwidth saturation is detected, RL-Paxos dynamically switches to a Topology-Aware Commit Path, where each follower routes its P2b to a proximity-selected coordinator node, which then relays the final commit decision. These approaches not only significantly reduce the leader's

TABLE II
DESCRIPTION OF SYSTEM PARAMETERS AND REQUEST STATUS DEFINITIONS IN RL-PAXOS.

Parameters	**Description**
execute	Node's next entry to be executed
CCW	Range of commutativity checks, predefined constants
Status	**Description**
Accepted	A request is persisted by the node and waits to commit
Committed	A request is persisted by majority and waits to execute
Executed	A request is executed and replies to the client
Commutative	A request has no conflict with requests in CCW

load, such as receiving and processing consensus messages, broadcasting consensus results, and responding to the client (hence addressing Limitation 1 in §II-B), but also reduce wide-area request latency (thereby addressing Limitation 2 in §II-B). §III-B presents the implementation of RL-Paxos's core design.

The ordering-only leader design in RL-Paxos facilitates linearizable reads at followers (RFL), detailed in §III-C. In a WAN, when the leader is far from the followers, the latter have a higher likelihood of reaching consensus before the leader. Consequently, the most up-to-date data may reside with the followers rather than the leader. This allows the client to easily access the latest data through a single round of majority reads. In most cases, our RFL can retrieve the latest data with just one wide-area round-trip. In scenarios where RL-Paxos cannot fetch the most recent data in a single WAN RTT, it reverts to the traditional Paxos read process, using a two-round read to ensure linearizability. This approach aligns with our goal of offloading read tasks to the followers.

In terms of parallelism, RL-Paxos introduces an out-of-order execution mechanism (§III-D) that allows multiple requests to be processed concurrently under certain conditions. The leader performs commutativity checks and, once requests are deemed commutative, they are executed in parallel by followers without additional coordination. This mechanism increases concurrency and throughput, addressing Limitation 3 in §II-B. Unlike decentralized consensus approaches (e.g., EPaxos) that require multiple WAN RTTs for establishing dependencies, the leader-based out-of-order execution in RL-Paxos is lightweight (experimental results in Table IV), enabling parallel execution while maintaining consistency.

B. Update Phase

Our algorithm goes directly from the replication stage. We omit the leader election stage, because we can elect the leader in many ways [22], [30], which is not our focus and will not affect the correctness of the algorithm.

The update stage in RL-Paxos incorporates two key mechanisms: leaderless commit relay and enhanced commit path with topology-aware coordination, both designed to improve scalability and bandwidth efficiency in wide-area environments.

1) Leaderless Commit Relay: To offload the consensus burden from the leader and improve commit scalability, RL-Paxos introduces a *Leaderless Commit Relay* strategy. This

Algorithm 1: Rules for leader.

Input: Request σ which leader P received.
1 $C \leftarrow$ check result of σ's commutative within the CCW;
2 $Q \leftarrow [P.\text{ID}]$;
3 $e \leftarrow$ Entry[Slot, σ, Accepted, C, Q];
4 Persist the e;
5 **while** *True* **do**
6 Broadcast P2a(e) to all followers;
`// Check e's async responses.`
7 **while** *True* **do**
8 **if** *check* e *send successful* **then**
9 e.Status $\leftarrow$ Committed;
10 Call function **Execute**(e);
11 **if** e *has timeout* **then**
12 **goto** line 5;

Algorithm 2: Rules for followers which receive the P2a(e).

Input: Request P2a(e) which follower F received.
1 Push F.ID into e.Quorum;
2 **if** e.*Quorum' size* $\geq$ *Majority* **then**
3 e.Status $\leftarrow$ Committed;
4 Call function **Execute**(e);
5 **else**
6 Persist the e according to e.Slot;
7 **if** *the cluster bandwidth is sufficient* **then**
8 Broadcast P2b(F.ID, e) to all nodes;
9 **else**
10 Send P2b(F.ID, e) to the Commit Coordinator

mechanism differs from traditional Paxos, where the leader alone finalizes the commit and responds. In RL-Paxos, after broadcasting the P2a message, the leader delegates quorum formation to followers. Each follower selectively propagates its acceptance (P2b)—either to all replicas or to a designated commit coordinator (§III-B2)—depending on current network conditions. Once a follower gathers a quorum of P2b messages, it marks the request as *Committed* and directly replies to the client, eliminating the need for the leader-initiated learning phase (P3 in §II-A). If no follower reaches a quorum within a timeout, the leader re-broadcasts the P2a messages or initiates a full-state synchronization to ensure progress and fault tolerance. This design ensures commit progress with minimal latency, avoids central bottlenecks, and allows followers close to the client to respond quickly. The detailed algorithm process is as follows:

As shown in Algorithm.1, 2, 3: In RL-Paxos, once the leader receives an external request from a follower, it performs two tasks: (1) assigns a globally unique slot for ordering, and (2) evaluates commutativity with prior requests based on a bounded window (see §III-D). The leader then appends the request to its local log and marks the entry as *Accepted*.

Unlike traditional Paxos, RL-Paxos decouples the leader

Algorithm 3: Rules for followers which receive the P2b(id, e).

```
Input: Request P2b(id, e) which follower F received.
if no entry in the e.Slot of F's log then
    Push F.ID into e.Quorum;
    if e.Quorum' size ≥ Majority then
        e.Status ← Committed;
        Call function Execute(e);
    else
        Persist the e according to e.Slot;
        if the cluster bandwidth is sufficient then
            Broadcast P2b(F.ID, e) to all nodes;
        else
            Send P2b(F.ID, e) to the Commit
             Coordinator
else
    e_l ← F's log[e.Slot];
    switch e_l.Status do
        case Executed do
            Return
        case Committed do
            Call function Execute(e_l)
        case Accepted do
            if e.Status is Executed or Committed then
                Update e_l status based on e;
            else
                e_l.Quorum ← e_l.Quorum ∪ e.Quorum
                 if e_l.Quorum' size ≥ Majority then
                    e_l.Status ← Committed;
                    Call function Execute(e_l);
```

Algorithm 4: Execution rules for servers.

```
Function Execute(e: Entry):
    if Server.Execute < e.Slot ≤ Server.Execute +
     len(CCW) then
        if no entry in this server then
            return;
        if e.Commutative = True and e.Status =
         Committed then
            Executes the request and replies to client;
            e.Status ← Executed;
        return;
    if Server.Execute = e.Slot then
        if no entry in this server then
            return;
        if e.Status = Executed then
            Server.Execute ← Server.Execute + 1;
            return;
        Executes the request and replies to client;
        e.Status ← Executed;
        Server.Execute ← Server.Execute + 1;
    return;
```

from the quorum formation process. Upon receiving the Accept message (P2a), each follower locally logs the request and updates the quorum field by adding its own node ID. If the quorum size reaches a majority, the follower autonomously marks the request as *Committed*, transitions to execution phase (§III-D), and may directly respond to the client. Otherwise, the follower may either broadcast its acceptance (P2b) to all replicas or route it to a designated commit coordinator (as detailed in §III-B2)—depending on current network conditions. This removes the centralized Phase 3 commit from the leader.

Upon receiving a P2b message, a follower first checks whether a corresponding log entry exists for the given slot. If no local entry is found, the message is treated equivalently to receiving a leader-issued *P2a*, and the follower follows the same procedure described in Algorithm.2.

Otherwise, the follower updates the existing entry based on its local state: if already marked as *Executed*, it discards the message as stale; if *Committed*, it triggers execution; and if *Committed*, the follower merges the quorum information and upgrades to a higher status when a majority is reached.

2) Enhanced Commit Path with Topology-Aware Coordination: To address potential bandwidth bottlenecks caused by the all-to-all broadcast of P2b messages among followers in RL-Paxos, we introduce a topology-aware optimization. Instead of broadcasting P2b messages to all followers, each node selectively routes its P2b message to a designated follower node, referred to as the commit coordinator. Each replica periodically maintains WAN bandwidth estimates to other nodes in the cluster, and the commit coordinator is automatically selected based on the client's region identifier embedded in each request. When the system detects insufficient inter-region bandwidth, it adapts by routing P2b messages to a region-specific relay node. This commit coordinator—typically the replica with the most favorable cross-region connectivity—finalizes quorum formation and ensures commit progress.

Once the coordinator gathers a quorum of P2b messages, it enters the commit phase and optionally broadcasts a compact commit message (analogous to Phase 3) to inform other replicas. This design retains RL-Paxos's decentralized commit mechanism while significantly reducing inter-node messaging overhead, particularly in WAN settings. When the bandwidth between WAN sites is limited or under high concurrency, this optimization helps maintain performance and stability.

In the event of coordinator unavailability or timeout, the protocol gracefully falls back to the original all-to-all gossip-style broadcast, preserving correctness and liveness.

Together, these mechanisms represent a significant enhancement in RL-Paxos, enabling leaderless commit progress that substantially reduces the leader's workload, improves scalability, and lowers request latency in wide-area environments.

C. Follower-Based Linearizable Read

To reduce leader load and support low-latency reads in geo-distributed environments, RL-Paxos enables linearizable reads from follower nodes. Unlike Paxos and Raft, where

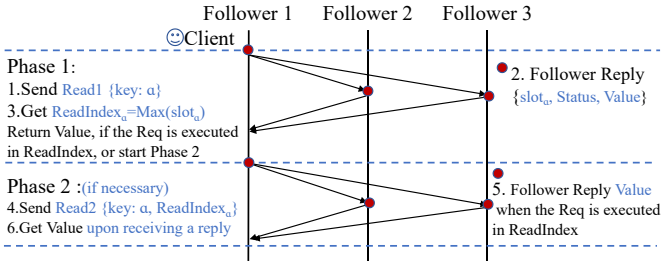

Fig. 5. Schematic of the read process of RL-Paxos. The client sends a read request to the majority of nodes. The first round of reading obtains the ReadIndex, and the second round (if necessary) waits for the ReadIndex location request to be executed. Due to the unique mechanism of RL-Paxos, most requests can be completed with a single read.

clients read the latest value directly from the leader, RL-Paxos allows clients to query any follower, while still ensuring linearizability via a one- or two-phase read protocol. The detailed process is shown in Fig. 5:

1) Fast Read Phase: When a client initiates a read (denoted read1), it sends the request to a majority of followers and collects responses that include the read value, the associated slot, and the consensus status. Upon collecting replies, the client identifies the response with the highest slot—referred to as the ReadIndex—and discards responses from lower slots.

If the response at the ReadIndex is marked as *Executed*, the client deems the value safe and returns it immediately. This path completes the read in a single WAN round-trip, offering low-latency access in most cases. If no executed entry is found, the client proceeds to a secondary read phase.

2) Fallback Read Phase: To ensure strong consistency when fast reads fail, the client triggers a second-stage read (denoted read2) to randomly selected followers. This request includes the previously determined ReadIndex, allowing replicas to align their logs to the expected consistency point.

Upon receiving read2, the follower compares the ReadIndex with its local execution pointer:

- If ReadIndex $\leq$ execute, the value is returned directly;
- If ReadIndex $>$ execute, the follower triggers a log alignment process: it re-broadcasts the relevant P2b message to help advance consensus and execution for the target slot across the majority.

This mechanism enables anti-entropy reconciliation and ensures that the ReadIndex becomes *Executed* at all relevant replicas. The follower then completes the read by returning the consistent value to the client.

D. Efficient Out-of-Order Execution with Minimal Coordination

Traditional consensus protocols require replicas to execute log entries in the same order to maintain consistency, leading to strict serialization and performance bottlenecks, especially in wide-area networks (WANs) due to high latency and message delays.

Recent protocols like EPaxos [27], CAESAR [4], Fast Paxos [24], and Generalized Paxos [23] relax this by exploiting commutativity to allow parallel execution of non-conflicting operations. However, these methods often rely on complex runtime dependency tracking, additional voting rounds or fast quorum thresholds that exceed majority (e.g., $\lceil (3N)/4 \rceil$ in Fast Paxos, $2f$ in EPaxos), which can be costly or impractical in WANs. Moreover, increased conflict windows in WANs lead to frequent fallbacks and global coordination rounds, undermining parallel execution benefits.

RL-Paxos offers a lightweight solution for geo-distributed systems, enabling out-of-order execution with minimal coordination. By shifting commutativity detection to the leader and using a fixed-size commutativity check window (CCW), RL-Paxos avoids runtime arbitration, dependency graphs, and multi-quorum coordination. The CCW defines the execution boundary, preventing excessive log sparsity and limiting the complexity of reordering. Commutativity checks follow the rules from previous works [24], [27], [31], allowing replicas to execute operations concurrently based on simple local rules while ensuring safety and correctness. The detailed execution stage process is as follows:

1) Execution Phase: Algorithm.4 describes the same execution rule followed by all nodes, without distinguishing between the leader and follower. As shown in Algorithm.4, each replica in RL-Paxos maintains a key local variable execute, which points to the next log slot expected to be executed in sequential order. Similar to traditional Paxos [22] or Raft [30], entries matching the execute pointer are processed by the sequential execution engine.

RL-Paxos divides the execution logic into two parallel paths:

1) Sequential Execution Path: If a committed request's slot equals the execute pointer, the replica processes it in order and advances the pointer.
2) Out-of-Order Execution Path: If a committed request's slot is within a predefined commutativity check window (CCW) and is marked as commutative by the leader, it can be executed immediately—even if earlier slots remain unexecuted.

To avoid reordering side-effects, the execute pointer is only updated by in-order execution. This ensures that dependent or non-commutative operations are never skipped. The out-of-order execution logic merely supplements throughput by parallelizing safe operations.

Compared to EPaxos [27], Fast Paxos [24] or CAESAR [4], RL-Paxos's approach is considerably simpler: it requires no global dependency graph, no per-request coordination beyond the leader's initial commutativity marking, and no fallback arbitration. This design achieves execution parallelism at minimal cost, making it particularly effective in WAN deployments where high latency and bandwidth constraints preclude complex coordination.

E. Fault Tolerance and Handling Failures

Similar to Paxos, RL-Paxos follows a crash failure model where nodes can fail silently. RL-Paxos retains Paxos' core principles while modifying the replication phase, allowing it

to tolerate up to f node failures in a cluster of size $2f+1$. However, *Leaderless Commit Relay* introduces new failure modes, as relay nodes may retry, delaying message propagation. We distinguish three common types of communication failures: leader failure, follower failure, and relay node failure.

Leader Failure: When the leader fails, the protocol pauses until the leader recovers or a new leader is elected. If the leader recovers and resumes sending heartbeats, the system detects this and synchronizes the latest consensus state from followers. If the leader doesn't recover within the timeout, a new leader is elected using Paxos' election process. The new leader then resumes protocol execution, handling pending and new client requests, ensuring system availability and consistency.

Follower Failure: In case of a follower failure, RL-Paxos' Leaderless Commit Relay mechanism allows each node to independently monitor consensus progress. When a P2b message is received, each node updates its state, and as long as enough nodes ($\geq f+1$) participate, the protocol continues.

Relay Node Failure: If the selected relay node fails (i.e., after a follower sends a P2b message and doesn't receive the P3 message from the coordinator in time), the system switches to the default all-to-all broadcast mode. The followers will independently broadcast the P2b message to each other. This enables the follower nodes to reach a commit state on their own, ensuring that consensus continues without interruption and preventing data propagation from being stalled.

F. Correctness

RL-Paxos guarantees safty and liveness of command execution. Here we give a sketched proof, mainly to intuitively explain why RL-Paxos is correct. We have implemented our core design based on Paxos. The complete formal proof on the correctness of RL-Paxos is in the technical report [34], here we have also formalized a description of the algorithm in TLA+ [34], which has been model-checked with TLC model-checker.

Safety. *At most one command can be executed by different replicas in the same slot of the global log.*

Proof sketch: In RL-Paxos, the consensus protocol ensures that each command can be chosen by at most one replica within the same slot. This is achieved through the Paxos-based leader-election mechanism, where only the leader can propose commands in each slot. Commands are assigned to global log slots according to the consensus process, which guarantees that only one command can be selected for a given slot at any point in time. The absence of conflicting proposals in the same slot is ensured by the protocol's commitment rules, which prevent multiple replicas from executing different commands in the same slot, thus maintaining consistency across replicas.

Liveness. *Any value proposed by a replica will eventually be learned and executed under eventual message delivery and majority availability.*

Proof sketch: RL-Paxos optimizes the communication mechanism for scalability and performance while preserving the core Paxos protocol. As a result, RL-Paxos inherits the same liveness properties as Paxos. Each replica continuously retransmits messages to other replicas until receiving the required acknowledgments. Specifically, like classic Paxos, RL-Paxos can tolerate up to f node failures in a $2f+1$ replica cluster. The protocol guarantees that once a command is proposed and accepted by a quorum, it will eventually be executed, even in the presence of node failures, ensuring continuous system progress.

	SD	GD	GZ	BJ	QH
SD	0	80.2	99.8	66.1	86.9
GD		0	69.7	36.1	56.3
GZ			0	44.9	74.2
BJ				0	41.4
QH					0

TABLE III
THE RTTS BETWEEN THE FIVE CENTERS (MS).

IV. EVALUATION

Experimental setup. We implemented RL-Paxos in the open-source key-value store Paxi [1], which provides a fair testbed for prototyping and benchmarking replication protocols, since only the protocol component changes during testing. Paxi also offers benchmarking capabilities similar to YCSB [10]. This unified platform enables us to compare RL-Paxos with representative protocols under the same environment and workloads. In particular, we include the following baselines:

1) **SD-Paxos [41]**: the most advanced competitor for reducing leader load, as discussed in our motivation.
2) **Pig-Paxos [9]**: a variant that alleviates leader bottlenecks through Proxy Leader.
3) **Raft [30]**: a widely used leader-based protocol in practice, emphasizing simplicity and understandability.

In addition, We also include comparisons with decentralized protocols such as EPaxos [27] and CAESAR [4], as they represent state-of-the-art designs for parallel and WAN-aware out-of-order execution, providing a broader perspective on RL-Paxos's advantages beyond leader-based approaches.

We performed evaluations in real-world deployments of five wide-area centers, each of which represented a region. The five wide-area centers we use are located in Shandong (SD), Guangdong (GD), Guizhou (GZ), Beijing (BJ), and Qinghai (QH). Round-trip time (RTT) delays between each pair of centers are shown in Table III. The machine used runs Linux and has 2 CPUs and 4 GB of memory. In Paxos, the client only communicates with the Leader, while in RL-Paxos, the client only sends requests to its geographically nearest replica. Each request either writes or reads a 8-byte value, and unless otherwise stated, these requests are write requests. The inter-node bandwidth measured between experimental sites is approximately 940 Mbit/s. While this is sufficient to support follower-to-follower messaging under moderate load, we observe that at $250-300$ concurrent clients, the system reaches a bandwidth saturation point, leading to increased delay and packet loss.

The workload is generated within the cluster by nodes, each node operating multiple client threads within a singular

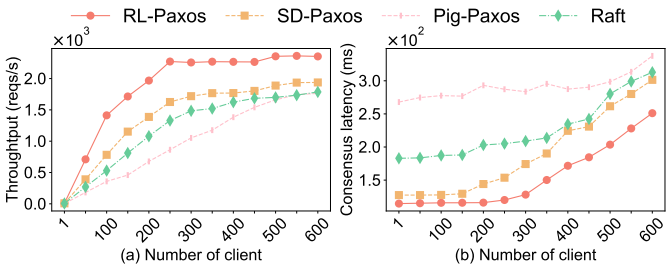

Fig. 6. Comparison of throughput (a) and consensus latency (b) of RL-Paxos against existing consensus protocols under increasing client concurrency.

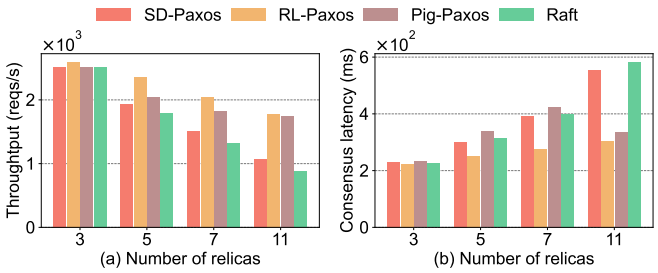

Fig. 7. Comparison of throughput (a) and consensus latency (b) of RL-Paxos against existing consensus protocols under different replica configurations.

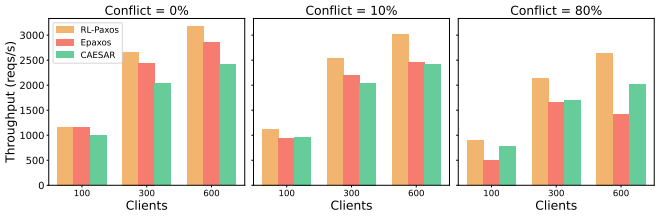

Fig. 8. Throughput comparison of RL-Paxos, EPaxos, and CAESAR under different conflict rates and client concurrency levels.

process. The client sends requests in a closed loop and does not start a new operation until the previous one is completed. Client operation latency is defined as the time between a client making a request and receiving a successful response. Throughput is measured as the number of requests successfully processed per second. Each experiment was run for 1 minute, and the length of time had no effect on the results.

A. Scalability Experiment

In this section, we will show the throughput and request latency of the protocol under different number of servers and concurrent clients. The leader is located in SD region, and the client is located in GZ region.

1) Client Scalability: As we can see in Fig.6, the throughput of all protocols increase significantly as the number of concurrent clients goes higher. RL-Paxos consistently outperforms the other protocols across all loads, and its throughput scales almost linearly under low concurrency. At peak load (600 clients), RL-Paxos achieves 21.46%, 33.48%, and 32.75% higher throughput than SD-Paxos, Pig-Paxos, and Raft, respectively.

To optimize throughput, our RL-Paxos leader retains only the core functionality of ordering, which fully releases the potential of the leader. SD-Paxos, though, is also able to spread most of the leader's load across all replicas, achieving higher throughput improvements (see original experimental results [41]). However, because the leader of SD-Paxos processes more *O-instance* messages than the leader of other protocols, and the *O-Accept* message needs to wait for the remote replica to trigger, the request path naturally increases by half a round trip, forcing the client request to wait for a long time before execution. Compared with SD-Paxos, RL-Paxos can handle higher client load. Due to the additional processing introduced by the proxy leader, Pig-Paxos is less efficient in handling requests under high concurrency compared to the other two protocols.

Fig.6(b) illustrates the response time of RL-Paxos, SD-Paxos, Pig-Paxos and Raft as the number of concurrent clients increases. In all cases, RL-Paxos consistently outperforms SD-Paxos, Pig-Paxos and Raft in terms of response time. Specifically, the average request latency of RL-Paxos is reduced by 10.09%–21.92% compared to SD-Paxos and 25.63%–57.22% compared to Pig-Paxos. Compared to Raft, RL-Paxos achieves up to a 42.87% reduction in response latency. This improvement can be attributed to RL-Paxos's unique design, which allows all replicas to act as leaders. The first replica to reach consensus responds directly to the client, significantly reducing response time. In contrast, SD-Paxos struggles to meet client demands, causing requests to queue for extended periods before execution. Additionally, the use of a proxy leader in Pig-Paxos may increase the response latency.

2) Server Scalability: We tested the maximum throughput and request latency of RL-Paxos, SD-Paxos, Pig-Paxos and Raft with 600 concurrent clients under different server counts. Fig.7 shows that RL-Paxos consistently outperforms the other protocols in both throughput and latency. With 11 replicas, RL-Paxos' throughput is 65.85% higher than SD-Paxos and matches Pig-Paxos. In terms of latency, RL-Paxos reduces latency by 16.67%-45.18% compared to SD-Paxos and 9.93%-25.64% compared to Pig-Paxos, demonstrating better scalability. Compared to Raft, RL-Paxos achieves a 101.58% throughput improvement and a 47.91% latency reduction. This is because RL-Paxos reduces leader load and optimizes request paths. Pig-Paxos uses proxy leaders to handle many followers, showing a throughput advantage with a larger number of servers ($n = 11$), but this advantage is less noticeable with fewer servers ($n < 11$). SD-Paxos adds complexity with an extra Paxos round, increasing coordination time as the number of nodes grows. Raft serves as a baseline and lacks corresponding offloading mechanisms, resulting in higher latency and limited scalability.

3) Comparison against EPaxos and CAESAR: This section compares RL-Paxos with EPaxos and CAESAR, two state-of-the-art decentralized consensus protocols. In our setup, all protocols are deployed in the same environment, where clients

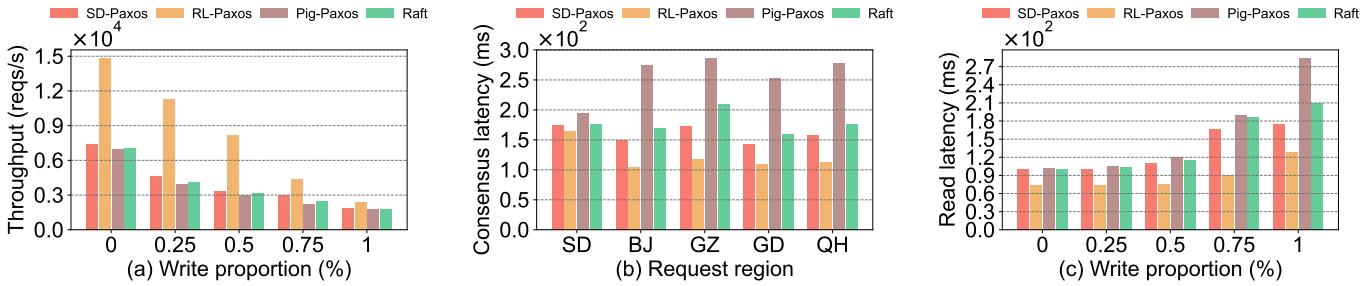

Fig. 9. (a) Comparison of throughput under different write ratios; (b) Average response latency for clients in different regions; (c) Comparison of read latency under different write ratios.

are located in the SD region and RL-Paxos is colocated with its leader. For the conflict rate in EPaxos and CAESAR, we refer to the method proposed in [35] to assess the actual conflict rate. The conflicted client is placed in the GD region to observe the impact on performance. We evaluate throughput as the number of concurrent clients increases (100, 300, 600) under three conflict settings (0%, 10%, 80%).

The experimental results demonstrate that RL-Paxos consistently outperforms both EPaxos and CAESAR across all scenarios, particularly excelling at high client loads. At 0% conflict, RL-Paxos achieves the highest throughput, with EPaxos slightly behind and CAESAR lagging further. As the conflict increases, RL-Paxos maintains its lead, with EPaxos suffering a noticeable drop in performance, particularly at 80% conflict. CAESAR, designed to optimize EPaxos in high-conflict situations, performs better than EPaxos in those scenarios but still falls short of RL-Paxos' performance.

B. Performance under Mixed Loads

In this section, we explore the performance of RL-Paxos, SD-Paxos, Pig-Paxos and Raft under mixed loads with different write request ratios (0%, 25%, 50%, 75%, 100%). Requests for both protocols are issued by the same client node located in the GZ center. To ensure linearly consistent reads, SD-Paxos, Pig-Paxos and Raft need to send read requests to the leader, while RL-Paxos reads via our linearizable read on followers (RFL for short). In Pig-Paxos and Raft, reads are served directly by the leader. In the SD-Paxos protocol, we use the optimized read where read requests are returned directly by the leader, because the read operation does not change the system state, so it does not need to be recorded in the log. The leader in SD-Paxos knows when to read in the local log because it can see all the updates on the object to be read.

Fig.9(a) illustrates that RL-Paxos consistently exhibits superior throughput compared to SD-Paxos, Pig-Paxos and Raft across varying read ratios. All protocols experience notable throughput increases as the proportion of read operations escalates. At a write ratio of 25%, the throughput of RL-Paxos is 2.47× and 2.85× that of SD-Paxos and Pig-Paxos, respectively, and 2.75× that of Raft. Notably, under full read load conditions (write proportion = 0%), RL-Paxos delivers approximately 2.0× higher throughput than SD-Paxos, Pig-Paxos, and Raft. The reason is SD-Paxos and Pig-Paxos's throughput improvement remains constrained by the single-leader architecture. As a highly serialized protocol, Raft does not optimize the leader's load, and thus shows similarly poor performance. In contrast, RL-Paxos mitigates single-leader load issues by evenly distributing load among follower nodes, thereby fully releasing followers' potential and significantly enhancing read performance, particularly under read-only scenarios. Moreover, RL-Paxos's use of RFL reads ensures linear consistency locally, significantly reducing wide-area read latency compared to the overhead incurred when accessing a remote leader, as evidenced in *Experiment Fig.9(c)*. These results demonstrate that RL-Paxos consistently achieves better performance across varying read-write ratios, highlighting its overall advantage under mixed workloads.

C. WAN Latency Experiment

Fig.9(b) shows the average request latency for clients in different regions. RL-Paxos consistently outperforms SD-Paxos, Pig-Paxos, and Raft. Compared to SD-Paxos, RL-Paxos reduces latency by 5.94%, 30.79%, 31.13%, 23.52%, and 28.58% in the SD, GD, GZ, BJ, and QH regions, respectively. Compared to Pig-Paxos, RL-Paxos reduces latency by 15.45%, 62.09%, 58.45%, 56.69%, and 59.64% in the same regions. RL-Paxos also reduces latency by 6.42%, 31.33%, 43.42%, 38.91%, and 36.5% compared to Raft. RL-Paxos achieves this improvement by leaderless commit relay, allowing all replicas to participate in the consensus process. Each node autonomously the replication progress of local requests and quickly reaching consensus across geographic distances. A replica geographically close to the client can be the first to reach consensus and respond to the client, reducing the client response time. In contrast, SD-Paxos, Pig-Paxos and Raft are hindered by reliance on a remote leader for consensus, resulting in longer client wait times before execution. Notably, in the SD region—where SD acts as the local leader—the request paths of SD-Paxos, Raft, and RL-Paxos are equivalent, resulting in nearly identical latency.

D. Experiment with WAN Packet Loss

In WAN, message loss and out-of-order arrival are more common than in a single center. In such scenarios, serial execution of state machines inevitably experiences extended periods of congestion, leading to significant tail latency. In

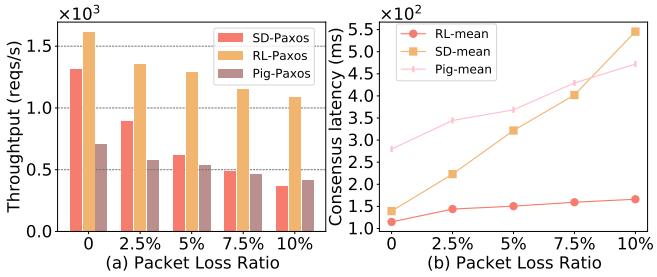

Fig. 10. Evaluation of throughput (a) and consensus latency (b) under varying packet loss ratios.

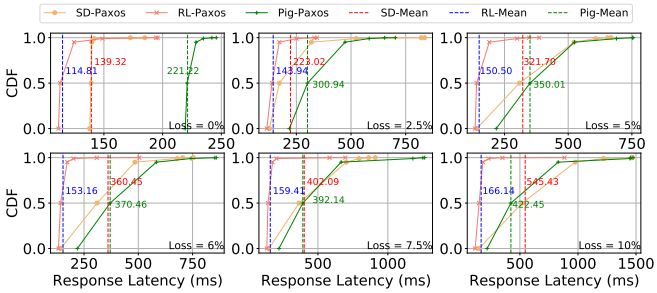

Fig. 11. Comparison of long-tail latency under different packet loss ratios.

this section, we use the *tc* command to add the packet loss rate to the nodes of the system to simulate the uncertainty in the WAN. The leaders are also in the SD region, the clients are in the GZ region, and the concurrency is 200.

Fig.10 shows that throughput and latency of SD-Paxos and Pig-Paxos decline significantly with increasing packet loss. At 2.5% loss, SD-Paxos experiences a 32.1% throughput reduction and 60.1% latency increase, while Pig-Paxos sees an 18.92% throughput drop and 36.03% latency increase. At 10% packet loss, both SD-Paxos and Pig-Paxos experience a further decrease in throughput. In contrast, RL-Paxos shows a more moderate throughput decrease of 32.5% and a 44.7% latency increase, achieving 2.98x higher throughput than SD-Paxos and 2.62x higher than Pig-Paxos, demonstrating better stability and adaptability to WAN uncertainties.

In contrast, RL-Paxos demonstrates a relatively moderate throughput reduction of 32.5%, with an average latency increase of only 44.7%. Under the same conditions, RL-Paxos achieves throughput that is 2.98x higher than SD-Paxos and 2.62x higher than Pig-Paxos. These experimental results demonstrate that RL-Paxosmaintains robust stability and exhibits a notable capacity toadapt to the inherent uncertainties of wide-area networks dueto its out-of-order execution mechanism.

Fig.11 illustrates the distribution of response latency for the above method under different packet loss rates. It is evident that RL-Paxos consistently outperforms SD-Paxos and Pig-Paxos in terms of both tail and average latencies. Notably, the latency performance of SD-Paxos deteriorates sharply with increasing packet loss rates. Specifically, at a packet loss rate of 10%, RL-Paxos achieves reductions of 76.1% and 81.2% in 999th and 99th response latency, respectively, compared to SD-Paxos. Moreover, RL-Paxos demonstrates a substantial reduction in average system response times across all tested conditions (as indicated by the vertical lines in the figure). In summary, the results affirm that RL-Paxos not only enhances parallel performance but also effectively adapts to WAN uncertainties by allowing out-of-order execution.

TABLE IV
EVALUATION OF COMMUTATIVITY CHECK OVERHEAD UNDER DIFFERENT CHECK RANGES. (μS)

Commutativity check range	Total leader processing time	Commutativity check time	Proportion of check overhead
2	1602	1	0.0062%
5	1603	2	0.12%
10	1613	5	0.30%
15	1635	8	0.48%

Table IV presents the total processing time for a leader to handle a request, the time taken for commutativity checks, and the associated overhead for various commutativity check ranges (2, 5, 10, 15). The experimental results indicate that the leader's total processing time remains relatively stable at approximately 1600 microseconds, while the time required for commutativity checks increases from 1 microsecond to 8 microseconds. Despite this increase, the overall overhead remains minimal, rising from 0.0062% to 0.48%. These findings suggest that the overhead introduced by our commutativity checking mechanism is negligible. In our experiments, the commutativity check range is 5, as it offers a satisfactory performance improvement. Setting the range too large, however, may negatively impact system performance due to the introduction of numerous log holes in the system logs.

E. Read Latency Experiment

In this experimentwe compare the read performance of RFL (Read from Followers), SD-Paxos with leader optimization (SD-RL), Pig-Paxos, and Raft. The read and write clients of all protocols are located on the same node. The read operations of Sd-Paxos, Pig-paxos, and Raft are all sent to the leader node, while the read operations of RL-Paxos are completed by the follower node.

Fig.**??** shows that under a fully read workload (write ratio = 1), RFL reduces latency by approximately 27.14% compared to other protocols. As the write ratio increases (write ratio = 0.5), RFL achieves at least a 31.76% latency reduction relative to other protocols. Under a high write ratio (write ratio = 0.75), RFL further reduces latency by about 45.97%, 51.61%, and 49.05% compared to SD-Paxos, Pig-Paxos, and Raft, respectively. The increase in latency across all protocols is primarily due to the growing read-write conflicts as the write ratio rises, forcing read operations to wait for pending writes to complete. However, the performance advantage of RFL becomes more pronounced under such conditions. The leaderless commit relay mechanism in RL-Paxos allows write requests to complete quickly among follower nodes, while RFL optimizes read performance by performing quorum reads from nearby

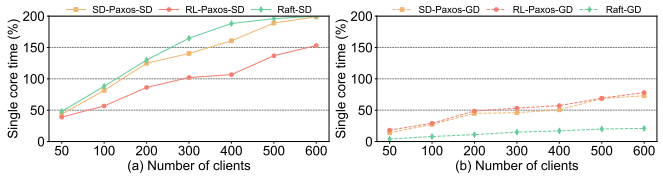

Fig. 12. Comparison of CPU usage across different client concurrency levels.

replicas. In contrast, SD-Paxos waits for all preceding objects to complete before serving reads. Raft and Pig-Paxos exhibit higher latency because they rely on a single leader to complete consensus, leading to slower read responses. Note that a write ratio of 1 represents a fully write workload, and thus the measured latency corresponds to write requests.

F. CPU Usage Experiment

In this experiment (Fig.12), we measure CPU utilization under a balanced write workload (50% writes), with RL-Paxos using RFL for reads, while SD-Paxos and Raft read from the leader.

As shown in the left graph, Raft places the highest load on the leader, resulting in the highest CPU utilization, while replicas experience minimal load. SD-Paxos follows, with its leader under significant load, though less than Raft. In contrast, RL-Paxos distributes the load more evenly across replicas, significantly reducing the leader's CPU utilization. The right graph further confirms this, with Raft showing the highest leader CPU usage, while RL-Paxos keeps it low by offloading work to replicas. SD-Paxos shows moderate leader utilization, with replicas maintaining low CPU usage.

G. Ablation Study

To quantify the contribution of each optimization in RL-Paxos, we conducted an ablation study. The three key mechanisms are: ① **Relay (R)**: *the Leaderless Commit Relay*, which reduces leader-side bottlenecks. ② **TACP (T)**: *the Topology-Aware Commit Path*, which reduces cross-region communication overhead. ③ **OOO (O)**: *Out-of-Order Execution with Minimal Coordination*, which allows parallel execution of commutative operations within a bounded window.

Table V summarizes throughput and latency results. The findings highlight that: All three optimizations are complementary. The full configuration (R+T+O) consistently achieves the best throughput and lowest latency across all scenarios. Relay provides clear benefits under high concurrency: removing Relay causes the leader to become a bottleneck, reducing throughput and increasing tail latency. TACP is most effective in wide-area deployments: without it, cross-region fanout increases significantly, leading to reduced throughput under bandwidth-limited conditions. OOO is particularly impactful under lossy networks: disabling it forces sequential execution, which degrades throughput and exacerbates tail latency.

H. Performance Under Failures

To evaluate the fault tolerance of RL-Paxos, we conducted micro-benchmarks under follower, leader, and coordinator

TABLE V
RESULTS OF THE ABLATION STUDY. R, T, AND O DENOTE **RELAY**, **TOPOLOGY-AWARE COMMIT PATH**, AND **OUT-OF-ORDER EXECUTION (OOO)**, RESPECTIVELY. THE TABLE REPORTS THROUGHPUT (OPS/S) AND P95 LATENCY (MS) UNDER DIFFERENT COMPONENT COMBINATIONS ACROSS THREE WAN SCENARIOS.

Scenario	Method	Thr. ↑	Lat. ↓	P95 ↓	P99 ↓
0% Loss 600 Clients	R†	1197.68	679.55	889.93	1287.15
	R+O	1437.22	577.01	730.15	901.16
	R+T	2049.23	350.35	430.32	1083.16
	R+T+O	**2389.56**	**251.09**	**272.41**	**440.72**
5% Loss 300 Clients	R	760.51	303.82	443.87	822.48
	R+O	980.52	231.56	340.61	519.15
	R+T	1121.13	201.41	305.16	712.15
	R+T+O	**1288.28**	**150.50**	**190.28**	**299.42**

† Topology-Aware Commit Path (T) is an adaptive mode that activates only when Relay Commit (R) is enabled; hence there is no standalone configuration for "T" or "T+O."

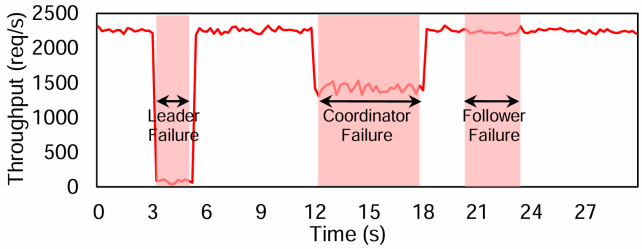

Fig. 13. Performance of RL-Paxos under different types of failures.

failures with 300 concurrent clients in a 5-replica WAN deployment. Fig.13 summarizes throughput under different types of failures:

Leader Failure (3-5 seconds): When the leader fails, throughput drops significantly, approaching zero, because the system must pause while waiting for the leader to recover. Once the leader recovers, it synchronizes the latest consensus state from the follower nodes, particularly pending requests and logs, ensuring that the recovered leader processes requests in a consistent state. After recovery, the leader resumes the protocol, and throughput stabilizes quickly, returning to normal operation. Coordinator Failure (12-18 seconds): The failure of the coordinator causes a noticeable temporary drop in throughput, approximately 38.3%. This occurs due to network congestion in the WAN link under high concurrency, causing the system to switch to an all-to-all broadcast mode, as described in the fault tolerance mechanism. In this mode, the system continues achieving consensus without a dedicated coordinator. While throughput is temporarily affected, the system does not stall completely. Once the coordinator recovers, the heartbeat mechanism allows the topology-aware enhanced commit path to resume, and throughput returns to normal.

Follower Failure (21-23 seconds): When a follower fails, the protocol continues with the remaining healthy nodes. As long as enough nodes ($\geq$ f+1) are involved, the protocol runs normally. Throughput decreases slightly (around 3%) but has negligible impact.

V. Related work

Minimizing the leader's load is the key problem we aim to address. Our work is related to work devoted to overcoming leader bottlenecks, and we surveyed the literature on these efforts. Table I summarizes the characteristics of the methods we investigated. Additionally, we discuss decentralized protocols and multi-leader protocols relevant to our research. These efforts are not included in Table I as they do not pertain to centralized protocols.

Mencius and EPaxos. Mencius [26] and EPaxos [27] explore multi-leader and leaderless designs, respectively. Mencius rotates proposal generation among multiple nodes to distribute the workload and enhance system performance, though lagging leaders can impede execution efficiency. In contrast, EPaxos eliminates fixed leaders, utilizing conflict detection and dependency tracking to execute requests in a consistent order, thereby improving parallelism and throughput. If a conflict arises, EPaxos reverts to Paxos to re-establish order, but conflicting requests can significantly degrade performance under heavy workloads. RL-Paxos innovatively shifts leader responsibility to followers, alleviating leader bottlenecks and improving scalability and performance. This approach maintains simplicity and ease of implementation, avoiding the complexities associated with rotating leader mechanisms or conflict detection and handling mechanisms.

Hardware assisted consensus. Efforts [5], [11], [12], [19], [21], [25] to improve performance at the hardware level, such as No-Paxos [25], which uses programmable switches to move the sorting process to the network layer, and FLAIR [21], which employs programmable switches to accelerate read performance. Additionally, [19] utilizes field-programmable gate arrays (FPGAs) to achieve consensus. However, these approaches depend on specific hardware and are currently challenging to deploy in wide-area data centers.

In summary, none of the methods in Table I effectively minimize the leader's load, prompting us to design RL-Paxos to reduce leader responsibilities and thereby fully unlock protocol performance. Table I and Fig.1 also present SD-Paxos and Pig-Paxos are the most advanced methods to reduce leader load at present, and this paper takes them as the benchmark for comparison.

VI. Conclusion

In this paper, we present RL-Paxos, a novel approach that offloads most tasks to the followers while retaining only the core function of ordering for the leader. By integrating RL-Paxos into the classic Paxos protocol, we demonstrate its ability to effectively mitigate the leader bottleneck. Experimental results show that RL-Paxos delivers high throughput and low wide-area latency, even under high concurrency, multi-node deployments, and mixed workloads.

References

[1] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. Dissecting the performance of strongly-consistent replication protocols. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1696–1710, 2019.

[2] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Wpaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):211–223, 2019.

[3] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Bekir O Turkkan, and Tevfik Kosar. Efficient distributed coordination at wan-scale. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1575–1585. IEEE, 2017.

[4] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 49–60, 2017.

[5] Giacomo Belocchi, Valeria Cardellini, Aniello Cammarano, and Giuseppe Bianchi. Paxos in the nic: Hardware acceleration of distributed consensus protocols. In *2020 16th International Conference on the Design of Reliable Communication Networks DRCN 2020*, pages 1–6. IEEE, 2020.

[6] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 111–120. IEEE, 2012.

[7] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.

[8] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Linearizable quorum reads in paxos. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[9] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Pigpaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of the 2021 International Conference on Management of Data*, pages 235–247, 2021.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[11] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. Network hardware-accelerated consensus. *arXiv preprint arXiv:1605.05619*, 2016.

[12] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–7, 2015.

[13] Hao Du and David J St Hilaire. Multi-paxos: An implementation and evaluation. *Department of Computer Science and Engineering, University of Washington, Tech. Rep. UW-CSE-09-09-02*, page 10, 2009.

[14] Vasilis Gavrielatos, Antonios Katsarakis, and Vijay Nagarajan. Odyssey: The impact of modern hardware on strongly-consistent replication protocols. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 245–260, 2021.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[16] Heidi Howard and Richard 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*, pages 1–9, 2020.

[17] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[18] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[19] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 425–438, 2016.

[20] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.

[21] Ibrahim Kettaneh, Ahmed Alquraan, Hatem Takruri, Ali José Mashtizadeh, and Samer Al-Kiswany. Accelerating reads with in-network

consistency-aware load balancing. *IEEE/ACM Transactions on Networking*, 30(3):954–968, 2021.

[22] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[23] Leslie Lamport. Generalized consensus and paxos. 2005.

[24] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, 2006.

[25] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say {NO} to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 467–483, 2016.

[26] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. 2008.

[27] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372, 2013.

[28] Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. When paxos meets erasure code: Reduce network and storage cost in state machine replication. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 61–72, 2014.

[29] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1221–1236, 2018.

[30] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[31] Seo Jin Park and John Ousterhout. Exploiting commutativity for practical fast replication. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 47–64, 2019.

[32] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

[33] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 1493–1509, 2020.

[34] Chenhao Zhang. The tla+ proof of rl-paxos. https://github.com/stupidzzz/proof-of-RL-Paxos, 2025.

[35] Sarah Tollman, Seo Jin Park, and John Ousterhout. {EPaxos} revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 613–632, 2021.

[36] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V Madhyastha. {Near-Optimal} latency versus cost tradeoffs in {Geo-Distributed} storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 157–180, 2020.

[37] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. {CRaft}: An erasure-coding-supported version of raft for reducing storage cost and network cost. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 297–308, 2020.

[38] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[39] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated state machines with compartmentalization. *Proceedings of the VLDB Endowment*, 14(11):2203–2215, 2021.

[40] Yugabyte Inc. Yugabytedb. the distributed sql database for mission critical applications., 2024. Accessed: 2024-07-20.

[41] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. Sdpaxos: Building efficient semi-decentralized geo-replicated state machines. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 68–81, 2018.