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

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

[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
[*] State Key Laboratory of Chemical Safety, China University of Petroleum (East China), Qingdao, China
[†] 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], [31], [36]. 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 allows SMR to continue operating despite the failure of some replicas and the presence of arbitrary network latencies.

Existing SMR implementations predominantly rely on leader-based consensus protocols (e.g., Paxos [22], Raft [29], 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.

The authors gratefully acknowledge these comments and suggestions. The work described in this article was supported by the National Key R&D Program of China under Grant NO. 2023YFB4503100, 2021YFA1000102 and 2021YFA1000103.

Xiaojian Liao is the corresponding author.

A series of works have been proposed to reduce the load on the leader [6], [8], [9], [39]. Some systems [17], [32], [38] 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], [39], employing proxy leaders [34], [37], utilizing Paxos quorum reads [8], and leveraging communication aggregation and piggybacking [9]. Unfortunately, these methods either compromise linearizability [32], [38] 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. Using Pig-Paxos [9] and SD-Paxos [39] 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 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 responsible 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 [29]. 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 comparing 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.

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 *Accepte* 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

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

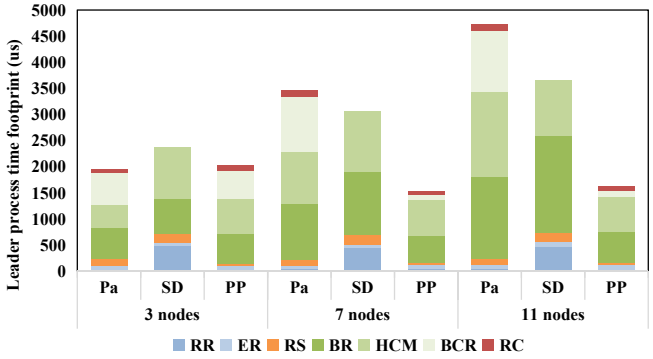

Fig. 1. Latency breakdown of various tasks performed by the leader. **Pa**: Paxos [22], **SD**: SD-Paxos [39], **PP**: Pig-Paxos [9]. Blue indicates the necessary load, which includes RR, BR, and ER. Green represents the additional load, encompassing tasks including BR (partially), HCM, RC and BCR. Orange highlights the leader's core responsibility: establishing a global order for requests.

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.

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 [39]) 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], [34], [37]. SD-Paxos [39] 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.

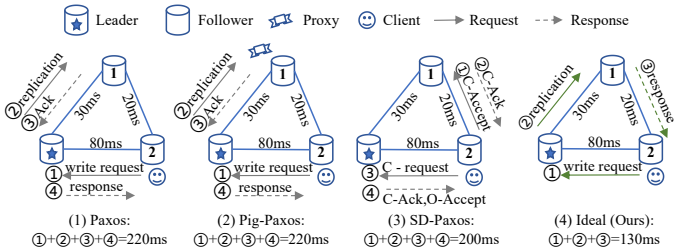

Fig. 3. Comparison of request paths of different protocols in wide-area environments.

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 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.

	RR	RS	BR	HCM	ER	RC	BCR	HRR
Paxos [22], Raft [29]	★	★	★	★	★	★	★	★
SD-Paxos [39], S-Paxos [6]	★	★	⊗*	★	★	⊗	⊗	★
Pig-Paxos [9], CP-Paxos [37], Pando [34]	★	★	★	★	★	★	★	★
WAN-Paxos [2], [3], D-Paxos [28]	★	★	★	★	★	★	★	★
RS-Paxos [27], CRaft [35]	★	★	★	★	★	★	★	★
RL-Paxos (ours)	★	★	★	⊗	★	⊗	⊗	⊗

TABLE I

COMPARISON BETWEEN RL-PAXOS AND OTHER LEADER-BASED PROTOCOLS IN GEO-REPLICATED ENVIRONMENTS. ★ INDICATE THE LEADER'S TASK, WHILE ⊗ DENOTE THE FOLLOWER'S TASK. ⊗* INDICATES THAT ALTHOUGH SD-PAXOS ELIMINATES THE TASK OF BROADCASTING A REQUEST, A DIFFERENT KIND OF MESSAGE, O-ACCEPT, NEEDS TO BE BROADCAST IN THIS PLACE.

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. Although SD-Paxos offloads the leader's replication function to a follower, the follower itself becomes a centralized node, requiring multiple rounds of communication with both the followers and the leader.

In the ideal path envisioned by RL-Paxos, the leader ensures the ordering and forwards the sorted request to the closet 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 fluctuations (see Figures 13 and 14 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 aims to retain implementation simplicity and avoid the complex runtime arbitration required by protocols such as EPaxos [26] or CAESAR [4].

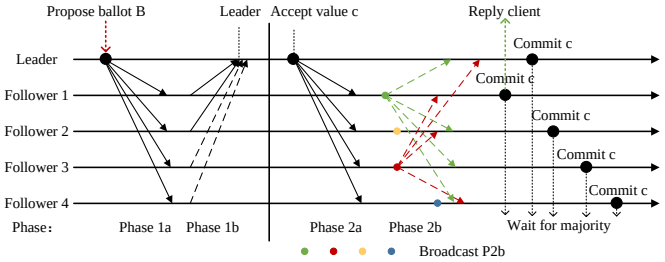

Fig. 4. RL-Paxos communication flow. The leader is only responsible for ordering, and all the followers exchange their own consensus status information, reach consensus and respond to the client on their own.

To support scalable replication under varying network conditions, RL-Paxos implements a Leaderless Commit Relay and an Enhanced Commit Path with Topology-Aware Coordination. Under normal load, 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], Committed phase in Raft [29]). 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. This approach not only significantly reduces the leader's 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 reduces wide-area request latency (thereby addressing Limitation 2 in §II-B). By enabling geographically closer followers to execute requests first, the strategy ensures faster responses to the client. §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

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 reply to client
Commutative	A request is no conflicted with requests in CCW

TABLE II
STATUS AND PARAMETERS DESCRIPTION

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.

Furthermore, RL-Paxos incorporates a lightweight out-of-order execution mechanism to increase concurrency (§III-D), therefore solving the Limitation 3 in §II-B. The key idea is to let the leader perform commutativity checks, as the leader holds the entire log entries and can easily determine whether a request is commutative with prior requests. Unlike decentralized consensus approaches (e.g., EPaxos) that require multiple WAN RTT to establish dependencies for out-of-order execution, the leader-based out-of-order execution of RL-Paxos is feasible and lightweight (experimental results in Table IV). This mechanism allows followers to independently determine whether a request can be executed out of order without requiring additional coordination.

B. Update Phase

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.

Our algorithm goes directly from the replication stage, we omit the leader election stage, because we can elect the leader in many ways [22], [29], which is not our focus and will not affect the correctness of the algorithm. The leader simplifies the management of replication logs. For example, the leader can decide where to place new request in the log without consulting other servers, and manage data transfers in a simple way.

1) Leaderless Commit Relay: To offload as much of the consensus burden as possible from the leader, We propose a Leaderless Commit Relay strategy, which offloads the consensus burden from the leader to follower nodes. After the leader broadcasts the P2a message to followers, followers no longer reply to the leader, but selectively propagates its acceptance either to all replicas or to the commit coordinator (a designated relay node §II-B), depending on current network conditions. Once a follower has gathered a quorum of P2b messages indicating acceptance, it considers the request committed and

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 Persistance 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 Persistance the e according to e.Slot;
7 Broadcast P2b(F.ID, e) to all nodes;

becomes a temporary commit coordinator, responsible for replying to the client when the request is executed. This mechanism differs from traditional Paxos, where the leader alone finalizes the commit and responds. In contrast, RL-Paxos allows any follower that first observes a quorum to complete the commit and handle the client response. If no follower assumes this role within a timeout period, the system triggers a fallback mechanism: other followers may re-issue their P2b messages or initiate a full broadcast 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.

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 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, and may directly respond to the client. This removes the need for

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* $\geq$ *Majority* **then**
    e.Status $\leftarrow$ Committed;
    Call function **Execute**(e);
  else
    Persistance 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
  $e_l \leftarrow F$'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 $\leftarrow$ e_l.Quorum $\cup$ e.Quorum
        if *e_l.Quorum' size* $\geq$ *Majority* **then**
          e_l.Status $\leftarrow$ Committed;
          Call function **Execute**(e_l);

Algorithm 4: Execution rules for servers.

Function `Exectue`(*e: Entry*)**:**
  if *Server.Execute* $<$ *e.Slot* $\leq$ *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 reply to client;
      e.Status $\leftarrow$ Executed;
    return;
  if *Server.Execute* = *e.Slot* **then**
    if *no entry in this server* **then**
      return;
    if *e.Status* = *Executed* **then**
      Server.Execute $\leftarrow$ Server.Execute + 1;
      return;
    Executes the request and reply to client;
    e.Status $\leftarrow$ Executed;
    Server.Execute $\leftarrow$ Server.Execute + 1;
  return;

a centralized Phase 3 commit from the leader.

Upon receiving the Accept message (P2a), each follower propagates its local consensus state through a P2b message. Depending on network conditions, the follower may either broadcast this message to all replicas or route it to a designated commit coordinator (as detailed in §III-B1). This message contains the slot, current status, and quorum set.

Upon receiving a P2b message, a replica performs one of three updates based on its local log state: If the local slot is empty, it adopts the message content directly; If already marked as Executed, it discards the message as stale; If marked Accepted or Committed, it performs a quorum merge and updates to the higher-status value if applicable.

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. The commit coordinator is automatically selected based on the client's region identifier embedded in each request. Each replica periodically maintains WAN bandwidth estimates to other nodes in the cluster. When the system detects insufficient inter-region bandwidth, it adapts by routing P2b messages to a region-specific relay node, rather than broadcasting to all followers. This commit coordinator—typically the replica with the most favorable cross-region connectivity—collects a quorum of P2b messages and subsequently broadcasts a commit notification. This strategy significantly reduces WAN traffic while preserving the protocol's consistency guarantees.

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.

These steps represent a significant enhancement in RL-Paxos, wherein followers, upon receiving a leader message, no longer respond directly to the leader but instead broadcast their state across the cluster, facilitating state merging among nodes. Upon achieving a majority, each follower autonomously enters the commit phase without requiring notification from the leader, subsequently responding to clients. This approach substantially reduces the leader's load and minimizes response latency, as the leader only needs to forward sequenced messages, with the remaining tasks distributed among followers. Furthermore, followers in closer geographical proximity to clients actively identify the replication progress of requests, leading to earlier client responses and reduced latency.

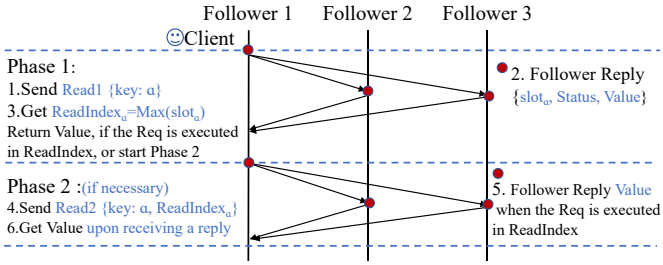

Fig. 5. Schematic of the read process of RL-Paxos. The client sends a read request to the majority of nodes. The first round of reading obtains the ReadIndex, and the second round (if necessary) waits for the ReadIndex location request to be executed. Due to the unique mechanism of RL-Paxos, most requests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

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 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.

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 typically require all replicas to execute committed log entries in the exact same order to preserve consistency. As a result, execution must be strictly serialized, often resembling single-threaded pipelines. This restriction becomes a major performance bottleneck, especially in wide-area deployments where network latency and message delay variability lead to increased tail latency and limited throughput.

In response, many recent protocols have attempted to relax strict sequential execution using commutativity-based techniques. Protocols such as EPaxos [26], CAESAR [4], Fast Paxos [24], Generalized Paxos [23] exploit the semantic independence between requests 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 WAN environments. Moreover, due to the high latency and asynchrony in WANs, the window in which requests appear to conflict increases, leading to frequent fallbacks, dependency reconstruction, or global coordination rounds. These recovery mechanisms incur significant overhead, undermining the advantages of commutativity in such settings.

RL-Paxos proposes a lightweight alternative that enables out-of-order execution with minimal coordination overhead, designed specifically for geo-distributed systems. By shifting commutativity detection to the leader and using a fixed-size commutativity check window (CCW), RL-Paxos avoids runtime arbitration, dependency graphs, or multi-quorum coordination. CCW serves as the execution boundary for out-of-order processing, preventing excessive log sparsity and constraining the complexity introduced by reordering within a manageable and well-defined range. For the rules for Commutativity checks, we follow the definitions of previous works [24], [26], [30]. This allows replicas to execute operations concurrently based on simple local rules, while preserving safety and correctness.

1) Out-of-order Execution: 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 or Raft [28], 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 [26], 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.

IV. Evaluation

In this section, we present the experimental setup and compare the experimental results of RL-Paxos with SD-Paxos.

Experimental setup. To evaluate our protocol, we implemented RL-Paxos in an open-source key-value store, Paxi [1]. Paxi is a fair testbed for prototyping, evaluating, and benchmarking replication protocols, since the only component that changes during testing is the replication protocol. This platform allows us to compare the RL-Paxos and SD-Paxos protocols under the same environment and workloads. Paxi also provides similar benchmarking capabilities to YCSB [10].

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 CPU 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 300 concurrent clients, the system reaches a bandwidth saturation point, leading to increased delay and packet loss.

To accommodate varying network conditions, RL-Paxos dynamically adjusts its Phase 2 commit mechanism. Under light or moderate load, followers exchange P2b messages in an all-to-all manner to expedite commit without leader involvement. However, when bandwidth saturation is detected (e.g., increased message loss or commit latency), the protocol automatically transitions to a Topology-Aware Commit Path, in which each follower routes its P2b message to a geographically proximate commit coordinator. This relay node aggregates quorum responses and broadcasts a compact commit notification, significantly reducing message fan-out and WAN bandwidth usage.

The workload is generated within the cluster by nodes, each node operating multiple client threads within a singular process. The client sends requests in a closed loop and does not start a new operation until the previous one is completed. This means that client operation latency is the time between a client making a request and receiving a successful response. In order to increase the throughput of the system, we used up to 600 clients, which better simulated the scenario of a large number of concurrent clients interacting with the system in the real world. Each experiment was run for 1 minute, and the length of time had no effect on the results.

	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).

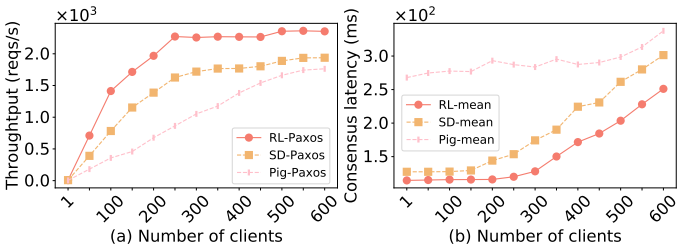

Fig. 6. Comparison of performance with varying numbers of clients.

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 RL-Paxos and SD-Paxos increases significantly as the the number of concurrent clients goes higher, and the maximum throughput (600 clients) of RL-Paxos is 21.46% and 33.48% higher than that of SD-Paxos and Pig-Paxos, respectively. Especially under low load, the throughput of RL-Paxos can increase almost linearly.

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 [39]). 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 and Pig-Paxos as the number of concurrent clients increases. In all cases, RL-Paxos consistently outperforms SD-Paxos and Pig-Paxos in terms of response time. Specifically, the average request latency of RL-Paxos is reduced by

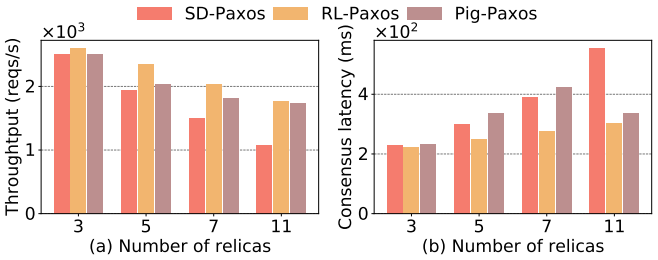

Fig. 7. Comparison of system performance with different number of replicas.

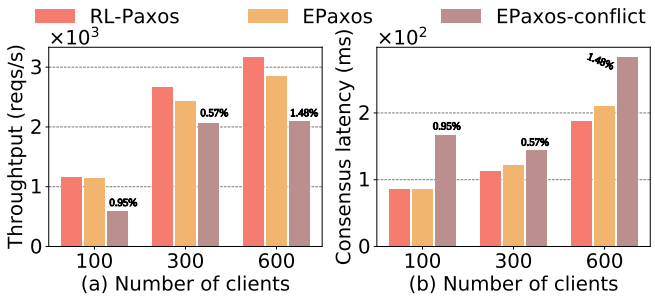

Fig. 8. Comparison with EPaxos.

10.09%–21.92% compared to SD-Paxos and 25.63%–57.22% compared to Pig-Paxos. 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: In this part, we used 600 concurrent clients to test their maximum throughput and request latency under different number of servers. The default number of relay groups in Pig-Paxos is 2, that is, two proxy leaders. Fig.7 shows that, in all cases, RL-Paxos consistently outperforms SD-Paxos and Pig-Paxos in both throughput and request latency. When the number of replicas was increased to 11, the throughput of RL-Paxos was 65.85% higher than that of SD-Paxos and matched that of Pig-Paxos. In terms of request latency, when the number of replicas is 5 or 7, the latency of RL-Paxos is reduced by 16.67%-45.18% and 9.93%-25.64% compared with SD-Paxos and Pig-Paxos respectively. This indicates that RL-Paxos had better server scalability. This is because RL-Paxos sufficiently reduces the load on the leader and optimizes the path of the request. Pig-Paxos introduces proxy leaders to communicate with a large number of followers in place of the leader, so there is a clear throughput advantage with a large number of servers (n=11), and the advantage of these proxy leaders is not obvious with a small number of servers (n ¡11). In contrast to the aforementioned protocols, SD-Paxos adds an additional Paxos round due to the separation of replication properties, thus increasing the complexity and time of coordination when the number of nodes increases, because each decision must wait for a sufficient number of nodes to confirm.

3) Comparison against EPaxos: This section compares RL-Paxos with EPaxos, the state-of-art decentralized protocol. In our setup, both the RL-Paxos and Epaxos clients are located in the SD region, with RL-Paxos colocated with the leader. We evaluate the throughput and latency of both protocols under varying levels of concurrency. For the conflict rate in Epaxos, we employed the method proposed in [33] to assess the actual conflict rate. The yellow region represents the optimal performance of the Epaxos protocol, where requests are processed through the fast path with no conflicts. The brown region indicates the performance under conflict conditions, where the conflicted client is placed in the GD region to observe the impact on performance.

The experimental results demonstrate that RL-Paxos consistently outperforms the EPaxos protocol in terms of throughput and latency across different concurrency levels. At 600 concurrent clients, RL-Paxos shows an 11.44% improvement in throughput and a 10.18% reduction in latency compared to Epaxos. This is attributed to the fact that RL-Paxos requests require majority confirmation, while even under the fast path, Epaxos demands confirmation from $2f$ of nodes, resulting in comparatively lower performance. When conflicts occur, Epaxos' performance declines significantly, with throughput dropping by 33.68% and latency increasing by 50.82% at 600 clients and a conflict rate of 1.5%. This degradation is primarily due to the additional wide-area round-trips required by Epaxos to resolve conflicts, which leads to a latency increase, with the collision operation potentially being over 4x times slower than that of a leader-based protocol [33].

B. Performance under Mixed Loads

In this section, we explore the performance of RL-Paxos, SD-Paxos and Pig-Paxos 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 and Pig-Paxos need to send read requests to the leader, while RL-Paxos reads via our linearizable read on followers (RFL for short) . In SD-Paxos and Pig-Paxos protocols, 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 illustrates that RL-Paxos consistently exhibits superior throughput compared to SD-Paxos and Pig-Paxos across varying read ratios. Both 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. Notably, under full read load conditions (write proportion = 0%), RL-Paxos demonstrates approximately a 1.0x throughput improvement over SD-Paxos and Pig-Paxos. The reason is

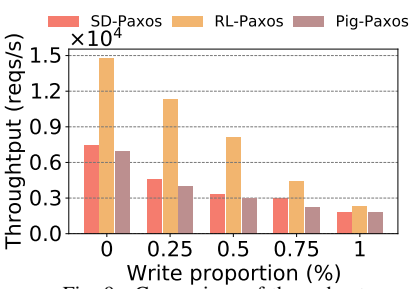

Fig. 9. Comparison of throughput under different write ratios.

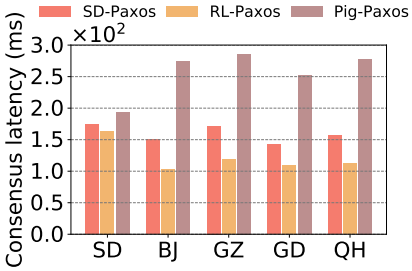

Fig. 10. Average response latencyin each region

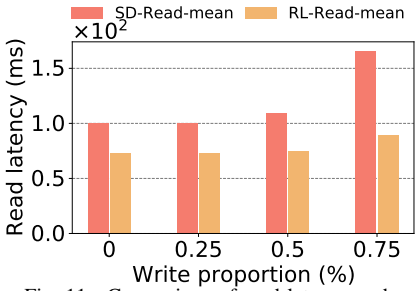

Fig. 11. Comparison of read latency under different write ratios.

SD-Paxos and Pig-Paxos's throughput improvement remains constrained by the single-leader architecture. 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, reducing wide-area read latency compared to the overhead incurred when accessing a remote leader, as evidenced in *Experiment IV-D of the study*.

C. WAN Latency Experiment

In this section, we show the average request latency for clients operating locally in different regions under medium load conditions (300 concurrent clients per region). Fig.10 illustrates the average request latency observed by clients in different regions when sending requests to local nodes. RL-Paxos consistently exhibits significantly lower request latency compared to SD-Paxos and Pig-Paxos. Compared with SD-Paxos, RL-Paxos reduces request latency by 5.94%, 30.79%, 31.13%, 23.52%, and 28.58% in the SD, GD, GZ, BJ, and QH regions, respectively. Compared with Pig-Paxos, RL-Paxos reduces request latency by 15.45%, 62.09%, 58.45%, 56.69%, and 59.64% in the SD GD, GZ, BJ, and QH regions, respectively. RL-Paxos achieves this improvement by decentralizing leadership, allowing all replicas to participate in the consensus process, with each node self-monitoring 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 and Pig-Paxos are hindered by reliance on a remote leader for consensus, resulting in longer client wait times before execution. Notably, latency differences between SD-Paxos and RL-Paxos are minimal in the SD region, where SD acts as the local leader, thereby rendering both protocols equivalent in performance under these conditions.

D. Read Latency Experiment

In this section, we compare the read performance of RFL (Read from followers) with SD-Paxos employing leader optimization (referred to as SD-RL). Fig.11 illustrates that across different write ratios, RFL consistently exhibits significantly lower read latency compared to SD-RL. Under low to medium write ratios (write proportion ≤ 0.5), the read latency of RFL and SD-RL remains relatively stable, with RFL reducing read latency by approximately 27.13% to 31.76%. However, at a write ratio of 0.75, both SD-RL and RFL experience a notable increase in read latency, reaching 165.79ms and 89.58ms, respectively, with RFL reducing latency by 45.97%. This increase is attributed to heightened read and write conflicts as write ratios increase. To ensure linear consistency in reading, both SD-RL and RFL must await the completion of the consensus process for updated objects, significantly delaying reads. RFL is specifically optimized for such scenarios by performing quorum reads from geographically proximate majority nodes, enabling it to return reads to clients upon object execution, even if preceding objects are still pending. In contrast, SD-RL must delay reads until all preceding objects have completed execution. Under full read load conditions (write proportion = 0), RFL achieves a 27.36% reduction in read latency. Notably, RFL not only guarantees linearly consistent reads but also minimizes access costs to remote leaders, thereby substantially reducing wide-area read latency.

E. CPU Usage Experiment

In this section, we employ a balanced write workload, wherein 50% of the operations are writes. Both protocols issue writes from the same client to the same replica group; however, RL-Paxos utilizes RFL for reads, while SD-Paxos reads from the leader. We measure the CPU utilization of the replica using the cpustat request, reporting it as a percentage of a core, with 100% indicating full utilization of a single core. Data is collected at 1-second intervals, focusing on the leader node (SD node) and a replica node, specifically the BJ node.

Within the Paxos protocol, the leader carries the majority of the workload, with replicas merely providing passive responses, resulting in a significantly unbalanced load across nodes. Fig.12 demonstrates that under high concurrency, the load disparity between the leader and replicas can be substantial, often by an order of magnitude. Fig.12 illustrates the efficacy of our approach in mitigating the leader's load.

As shown by the experimental results, the leader serves as the protocol's bottleneck. The leader's CPU utilization escalates with an increasing number of clients. When the number of concurrent clients reaches 600, the CPU utilization of the SD-Paxos leader approaches maximum capacity, whereas the cluster's replicas remain minimally loaded. At

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

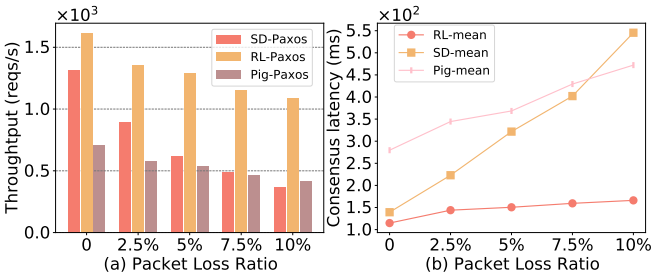

Fig. 13. Comparison of performance under different packet loss.

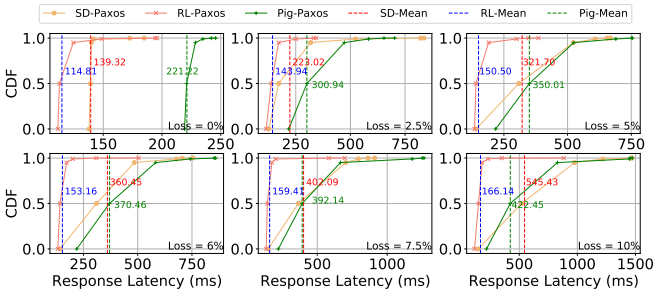

Fig. 14. Comparison of long-tail latency under different packet loss ratios.

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
COMMUTATIVITY CHECK TIME (IN μS)

this juncture, the BJ replica's CPU utilization is approximately 73%, noting that each node in the cluster is equipped with two cores. This indicates that SD-Paxos moderately alleviates the leader's load. Conversely, RL-Paxos effectively distributes the load across all replicas, substantially reducing the leader's burden. Consequently, the leader's CPU usage in RL-Paxos increases at a slower rate, which is pivotal to achieving higher throughput compared to SD-Paxos.

F. 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 this section, we use the *tc* request 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.13 illustrates the pronounced decline in throughput and latency of SD-Paxos and Pig-Paxos as the packet loss rate increases. Specifically, at a packet loss rate of 2.5%, SD-Paxos experiences a throughput reduction of 32.1% and a latency increase of 60.1%. Similarly, Pig-Paxos sees an 18.92% reduction in throughput and a 36.03% increase in latency. When the packet loss rate escalates to 10%, SD-Paxos and Pig-Paxos' throughput further diminishes. When the packet loss rate rises to 10%, both SD-Paxos and Pig-Paxos exhibit a further decrease in throughput. 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-Paxos maintains robust stability and exhibits a notable capacity to adapt to the inherent uncertainties of wide-area networks due to its out-of-order execution mechanism.

Fig.14 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 delay and average delay. 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 95th and 99th response latency, respectively. 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 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.

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 and EPaxos 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 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] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.