High-Throughput Multi-Leader Paxos Consensus with Insanely Scalable SMR

Gabriel Momm Buzzi[1], **Odorico Machado Mendizabal**[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)
Florianópolis, Santa Catarina, Brazil

`gabriel.m.buzzi@grad.ufsc.br, odorico.mendizabal@ufsc.br`

Abstract. *Distributed consensus protocols are essential building blocks for the development of distributed systems. They facilitate critical functionalities such as coordination in mutual exclusion and election algorithms, ensuring total order delivery in broadcast communication, and enabling active replication in approaches like State Machine Replication (SMR). Paxos has emerged as the most prominent distributed consensus algorithm, inspiring numerous optimizations over the past decades. In particular, some enhancements focus on improving Paxos's throughput and scalability by addressing its single-leader bottleneck through multi-leader and leaderless variants. This paper examines key optimizations related to quorum sizes, reconfiguration, message exchange overhead, and the decentralization of the leader role, with an emphasis on multi-leader and leaderless approaches. Additionally, we develop a high-throughput multi-leader Paxos implementation using the ISS (Insanely Scalable SMR) framework. We benchmark this implementation against other multi-leader and leaderless protocols, including WPaxos and EPaxos. Experimental results demonstrate that the proposed implementation achieves nearly twice the throughput of WPaxos and EPaxos, albeit with increased latency due to ISS bucket rotation overheads.*

1. Introduction

Distributed consensus protocols are fundamental building blocks for implementing distributed services. They enable a group of processes to safely agree on a value, even in the presence of failures, while guaranteeing termination, validity, integrity, and agreement in all decisions [Cachin et al. 2014]. These protocols give support to functionalities such as coordination in mutual exclusion and election algorithms, ensuring total order delivery in broadcast communication, and enabling active replication through State Machine Replication (SMR). However, ensuring protocol integrity becomes increasingly challenging as the number of nodes grows. Larger networks require more messages to be sent and processed per decision, leading to reduced throughput and higher latency. Additionally, widely used protocols, such as Paxos [Lamport 1998], Raft [Ongaro and Ousterhout 2014], and PBFT [Castro and Liskov 1999], rely on a single leader process. This leader handles all value proposals and bears a disproportionate share of the processing load, often becoming a bottleneck that limits performance.

Many algorithms have tried to solve this bottleneck problem, be it by changing how messages are sent [Jalili et al. 2010, Charapko et al. 2021], by trading availability for better performance [Howard et al. 2016] or by allowing multiples simultaneous leaders

[Mao et al. 2008, Ailijiang et al. 2020, Moraru et al. 2013]. Among these, multi-leader protocols stand out as a promising approach for improving scalability. By distributing the load of message processing across multiple leaders, these protocols eliminate bottlenecks and enhance throughput without compromising resilience.

This paper examines key optimizations related to quorum sizes, reconfiguration, message exchange overhead, and the decentralization of the leader role, with an emphasis on multi-leader and leaderless approaches. In addition, we implement a high-throughput multi-leader Paxos protocol using the Insanely Scalable SMR (ISS) framework [Stathakopoulou et al. 2022]. ISS enables single-leader consensus protocols to scale with the number of nodes by safely parallelizing their execution, transforming them into multi-leader protocols. We benchmark our Paxos variant against other multi-leader and leaderless protocols, including WPaxos and EPaxos. All implementations are built on the Paxi library [Ailijiang et al. 2019], ensuring a fair performance comparison.

The performance analysis shows that the proposed implementation outperforms other protocols, achieving nearly twice the throughput of WPaxos and EPaxos. However, this comes at the cost of increased latency due to ISS bucket rotations. Additionally, by varying the number of replicated servers, our strategy demonstrates good scalability, as expected from the ISS framework.

The rest of the paper is organized as follows. Section 2 overviews the Paxos protocol and some variants aiming for enhanced throughput and scalability. Section 3 describes well-known Multi-leader and leaderless Paxos proposals. Section 4 introduces the ISS. Section 5 and 6 presents our Multi-leader version of Paxos using ISS and its performance evaluation. Finally, Section 7 concludes the paper.

2. Paxos and its variations

The Paxos consensus protocol [Lamport 1998], proposed by Leslie Lamport in 1998, enables a group of processes in a partially synchronous distributed system to decide on a single value from a set of proposed values, even in the presence of process failures and message loss. Each process in Paxos assumes at least one of three roles: *Proposer*: proposes values, ensuring one is eventually decided; *Acceptor*: stores accepted proposals, serving as the protocol's memory; *Learner*: Learns the decided value. Paxos operates in two communication phases, repeated across rounds as needed.

Phase 1 – Leader Election: A proposer sends a $\mathbf{P1A}$ message with a ballot number to all acceptors, requesting them to ignore lower ballot numbers. Each acceptor compares the received ballot number with the highest known ballot number. If the received number is lower, the message is ignored. Otherwise, the acceptor responds with a $\mathbf{P1B}$ message containing: The highest accepted ballot number, the last accepted value, and the ballot in which the value was accepted. Once a proposer receives a majority of $\mathbf{P1B}$ responses, it is elected as the leader.

Phase 2 – Value Proposal: The leader sends a $\mathbf{P2A}$ message to all acceptors, including its ballot number and the value it proposes. Acceptors compare the ballot number with the highest known one, ignoring the message if the ballot number is lower. Otherwise, they accept the value and respond to all learners with a $\mathbf{P2B}$ message, containing the ballot number and the accepted value. A learner considers a value decided when it receives a majority of $\mathbf{P2B}$ messages for the same ballot and value.

Paxos ensures that once a value v is decided, it cannot be changed. This safety is achieved by the leader's selection process during Phase 1. If a value has been accepted, the $\mathrm{P1B}$ response contains the accepted value and its associated ballot. The leader then either re-proposes the value with the highest ballot number or proposes a new value if none has been accepted. Since quorums from different phases always intersect, Paxos guarantees that a decided value is preserved across rounds.

Multi-Paxos – Extending Paxos for Multiple Decisions: The original Paxos is limited to deciding a single value. However, applications like reliable broadcast and replication require deciding an unbounded sequence of values. *Multi-Paxos* [Lamport 2001] addresses this by running multiple instances of Paxos, where each instance decides one value in the sequence. A common optimization in Multi-Paxos involves executing *Phase 1* only once across all instances, as long as the leader remains active. This reduces the number of messages exchanged and allows the leader to propose multiple values simultaneously, significantly improving throughput.

Building upon Paxos and Multi-Paxos, numerous variants have been developed over the years to improve performance and resilience, or reduce the costs with redundancy. They include enabling reconfiguration, reducing latency, and enhancing scalability. A review of Paxos variants tailored for Portuguese speakers can be found in [Regis and Mendizabal 2022], though it is not a comprehensive survey.

2.1. Reconfiguration

The need for reconfiguration in consensus protocols comes from the fact that, in a practical environment, the set of participating processes does not remain static, with faulty processes being removed and new ones being added to increase resilience. As such, many strategies to allow reconfiguration have been proposed, for example:

Cheap Paxos [Lamport and Massa 2004]: This approach introduces special reconfiguration commands to modify the list of participating processes. The leader proposes these commands like any other value. Once decided, the command updates the configuration, adding or removing nodes in the list. While simple, this strategy imposes a limitation: a given quorum, for instance the i^{th}, cannot be determined until the decision for the previous instance, *i.e.*, the $i-1$, is finalized, introducing delays in decision-making for subsequent instances.

Vertical Paxos [Lamport et al. 2009]: Vertical Paxos offloads reconfiguration decisions to a configuration master outside the consensus protocol. When a node is suspected of failure or progress halts, a participant contacts the configuration master, which provides a new configuration, leader protocol, and ballot. The new leader then executes *Phase 1*, attempting to establish leadership. In this variant, each configuration corresponds to a unique ballot, allowing simultaneous proposals across multiple instances. This ensures safety while supporting efficient reconfiguration.

2.2. Message Exchange

The performance of consensus protocols is heavily influenced by how messages are sent and processed. Key factors include the number of communication rounds, message overhead, and distribution of processing loads among nodes. Notable optimizations are discussed as follows.

Fast Paxos [Lamport 2006]: In Multi-Paxos, a single leader introduces additional latency due to message forwarding. For instance, as shown in Figure 1(a), a node sends a request to the leader (steps 1 and 2), which then proposes it to acceptors (step 3). Fast Paxos eliminates this step by introducing fast rounds. In a fast round, the leader sends a special $\mathbf{P2A}$ message ($\mathbf{any}$), allowing acceptors to directly accept proposals from proposers. Figure 1(b) illustrates this: proposers send values directly to acceptors (step 2), who respond to the leader (step 3). If a quorum is achieved, the leader finalizes the decision; otherwise, a slow round is initiated, skipping *Phase 1* by reusing $\mathbf{P2B}$ responses. To ensure integrity, quorums in a fast round have increased restraints, which makes it so either $\left\lceil \frac{3N_a}{4} \right\rceil$ of the acceptors are required for fast quorums or that all quorums have an increased size of $\left\lceil \frac{2N_a}{3} \right\rceil$.

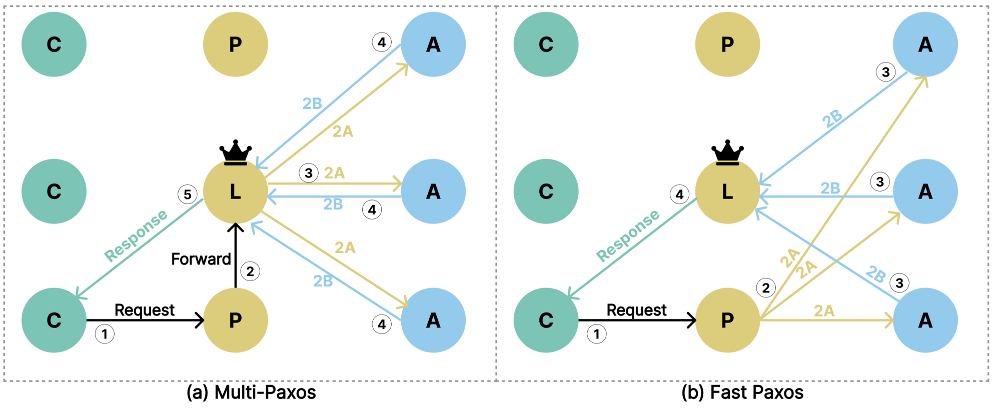


Figure 1. Example of *Phase 2* execution for Multi-Paxos and Fast Paxos.

Flexible Paxos [Howard et al. 2016]: Traditional Paxos requires majority quorums for all phases, requiring at least $(N_a/2) + 1$ response messages for N_a acceptors. Flexible Paxos relaxes this requirement, requiring only that *Phase 1* and *Phase 2* quorums intersect. This allows for smaller *Phase 2* quorums ($\mathbf{Q2}$) by increasing the size of *Phase 1* quorums ($\mathbf{Q1}$), which are rarely used. Besides this trade-off, it is also possible to create specific quorum structures that guarantee this interception, such as creating a grid of acceptors, such that its rows are quorums for *Phase 1* and its columns are quorums for *Phase 2*. For example, as shown in Figure 2, a grid quorum structure can optimize quorum sizes, trading fault tolerance for scalability.

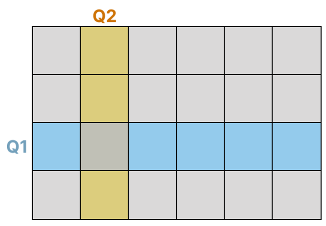


Figure 2. Example of a grid quorum.

2.3. Load Distribution

It is also possible to improve the throughput and scalability of a protocol by better distributing the processing load of messages across all nodes. In standard Paxos, the leader processes a disproportionate number of messages, often becoming a bottleneck as the system scales [Ailijiang et al. 2020, Moraru et al. 2013, Mao et al. 2008, Stathakopoulou et al. 2022]. Strategies like Ring Paxos and Pig Paxos distribute this load:

Ring Paxos [Jalili et al. 2010]: It builds a logical ring configuration, composed of a sequence of nodes (a quorum of acceptors) ending with the leader. Responses ($\mathbf{P1B}$ and $\mathbf{P2B}$) follow this ring, with each node forwarding and aggregating responses until the leader receives them. This reduces direct communication to the leader but may increase latency, as seen in Figure 3(a). The first acceptor in the ring, instead of directly sending its response to a leader, forwards it to the next acceptor in the ring configuration. This acceptor node then combines the response with its own and forwards it to the next acceptor, with this step being repeated until all the responses are forwarded to the leader, who learns if the value was decided.

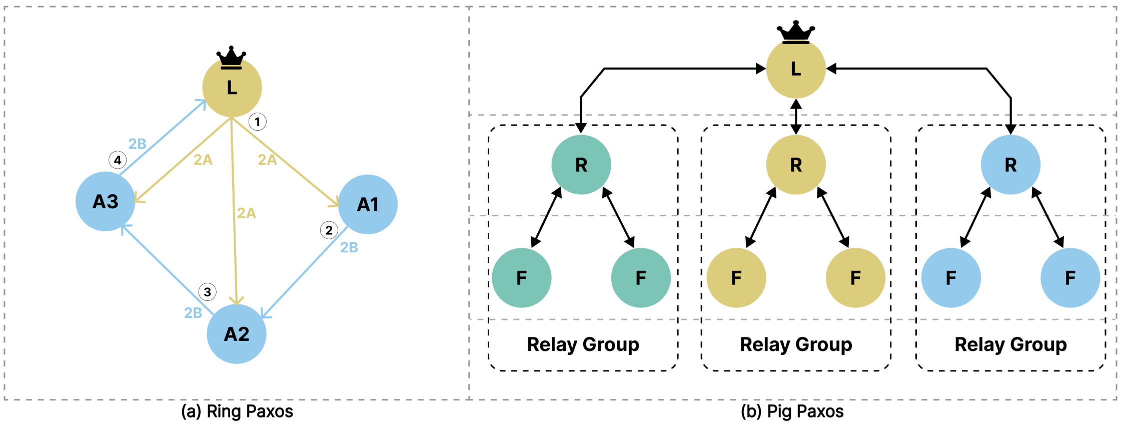


Figure 3. Messaging structure of Ring Paxos and Pig Paxos.

Pig Paxos [Charapko et al. 2021]: Adopts a hierarchical configuration where acceptors are divided into relay groups. When a proposer sends a $\mathbf{P1A}$ or $\mathbf{P2A}$ message instead of broadcasting it to all acceptors, it is sent to one randomly selected member of each relay group, which acts as a relay node. The relay nodes forward messages within their groups and aggregate responses to send back to the leader, as shown in Figure 3(b). This reduces the leader's message load but increases latency due to additional forwarding steps and introduces potential single points of failure at the relay group level.

3. Multi-Leader and leaderless protocols

Multi-leader protocols allow multiple processes to propose values concurrently, enabling all processes to act as leaders or eliminating the need for a designated leader. These approaches enhance protocol scalability and mitigates the leader bottleneck problem. Adding new nodes not only increases the overall message throughput but also expands the pool of nodes available for processing.

Next, we describe the most popular multi-leader and leaderless Paxos strategies.

Mencius: One of the most intuitive strategies for implementing multi-leader protocols is exemplified by *Mencius* [Mao et al. 2008]. In Mencius, the sequence of values to be decided is partitioned in a round-robin fashion, assigning a different process as the leader for each partition. Multiple instances of consensus can be decided in parallel by different partitions. However, variations in the speeds of different leaders can result in gaps in the log. For instance, in Figure 4 only the command x_0 is executable, despite other commands having been decided. This occurs because commands must be executed in order, and the second partition may not have decided a value for position 1. To address these gaps, Mencius introduces the concept of *skip messages*. When a leader detects that it is lagging behind others, it sends a message **skip** to indicate that no value will be decided for a particular instance. This mechanism ensures continuity in execution order. Additionally, skip messages are optimized by being decided with a single P2A message. This optimization maintains integrity since Mencius restricts node behavior such that only the initial leader of each partition can propose a value other than **skip**.

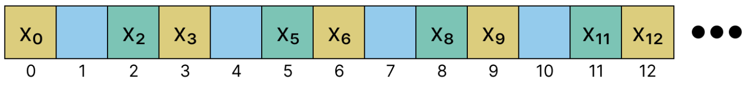


Figure 4. Example of a Mencius of with gaps (different colors represent different partitions).

Wan Paxos (WPaxos): Wan Paxos (WPaxos) [Ailijiang et al. 2020] is a multi-leader protocol designed to reduce latency and enhance scalability in key-value databases operating over Wide-Area Networks (WANs). WPaxos achieves this by parallelizing protocol execution, creating a separate instance of Multi-Paxos for each key in the key-value store. Each key is assigned an independent leader, allowing decisions to be made independently. WPaxos further optimizes WAN request latency by using flexible grid quorums, which form *Phase 2* quorums with geographically close nodes. Additionally, an adaptive steal strategy is employed during *Phase 1* to assign leaders closer to regions where specific keys are frequently accessed. While WPaxos offers excellent scalability, its applicability is limited to systems capable of fully partitioning data, as it does not support operations that span multiple partitions.

Egalitatian Paxos (EPaxos): Egalitatian Paxos (EPaxos) [Moraru et al. 2013] takes a different approach by deciding on a set of dependencies and a sequence number for each value rather than directly determining an ordered sequence of values. These dependencies indicate which commands have already been accepted and which conflict with the execution of the new command. A dependency graph is then constructed to establish a partial order in which commands can be executed, ensuring consistency across all nodes. EPaxos operates as a leaderless protocol where all nodes are equal, and decisions can be reached via a *fast path*, similar to Fast Paxos [Lamport 2006]. If a quorum of acceptors agrees on the same dependencies, the command is decided immediately. However, if conflicting proposals are received in different orders by different acceptors, additional message exchanges are required to resolve conflicts in what is termed the *slow path*. Consequently, the performance of EPaxos is closely tied to the frequency of conflicting commands. Higher contention leads to more commands being resolved via the slow path, increasing the average number of messages required per decision.

4. Insanely Scalable SMR

Insanely Scalable SMR (ISS) [Stathakopoulou et al. 2022] is a framework designed to enhance the throughput and scalability of single-leader consensus protocols by parallelizing their execution, transforming them into multi-leader protocols. ISS achieves this by employing consensus protocols that implement *Sequenced Broadcast* (SB), which decides on a predetermined number, n, of commands before halting. Each instance of SB has an initial leader, which is the sole process permitted to propose non-empty values.

Similar to Mencius, ISS partitions the sequence of values to be decided (the log) into s non-overlapping segments using a simple round-robin algorithm. Each partition is assigned its own instance of SB, complete with a dedicated leader. This setup allows commands to be decided in parallel across partitions without interference. However, this approach introduces new consistency challenges. For example, client commands might be decided out of order if they are sent to different instances. As illustrated in Figure 5, commands $C1.1$ and $C1.2$ are decided by instance 0, but if command $C1.3$ is sent to a different instance, it could be decided before $C1.2$, violating the expected order.

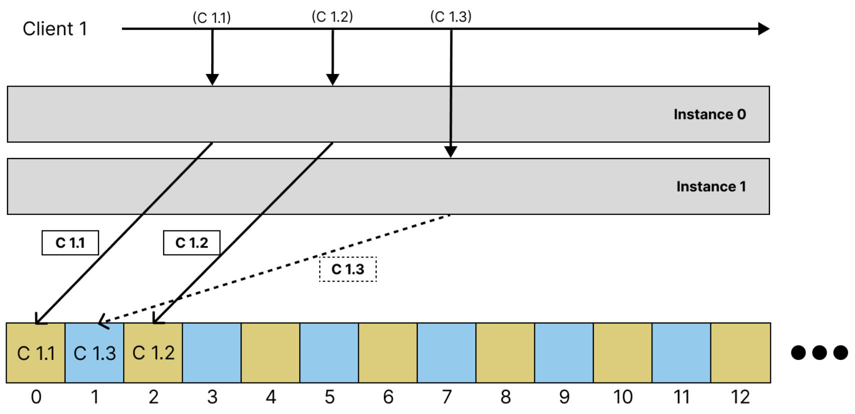


Figure 5. Example of a inconsistent order for client requests.

To address such issues, ISS introduces the concept of *buckets*. A bucket is a collection of commands associated with a subset of clients. By assigning each bucket to a specific segment, and ensuring that only the leader of that segment can propose values within the bucket, ISS prevents inconsistencies and ensures the correct ordering of client requests. This mechanism maintains the integrity of the execution order while preserving the benefits of parallelism.

However, this strategy introduces a limitation: if a leader fails, the system becomes unavailable to all clients whose requests are mapped to that leader's buckets, as only the initial leader is permitted to propose non-empty values. To address this issue, ISS utilizes the concept of *epochs*. An epoch is a contiguous partition of the log of size e, consisting of multiple segments, as illustrated in Figure 6, where different colors represent individual segments. These epochs are executed sequentially, *i.e.*, epoch $i + 1$ begins only after all values from epoch i have been decided. This sequential execution allows ISS to use the transition period between epochs, during which all nodes are aware of the commands decided thus far, to perform deterministic reconfiguration operations. During these transitions, ISS can rotate bucket assignments between leaders to mitigate unavailability caused by leader failure. Additionally, reconfiguration operations, such as adjusting leader assignments based on a failure detector or processing explicit reconfiguration commands, can be executed during this period. This mechanism ensures the system remains robust and minimizes the impact of leader failures while maintaining consistency and scalability.

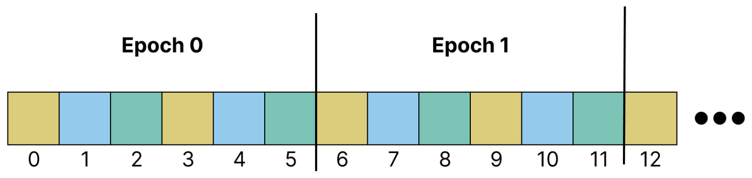


Figure 6. Example of ISS' epoch structure (each color represents a different segment).

5. Multi-leader Paxos with ISS

This section presents the implementation of a multi-leader Paxos protocol using ISS. The ISS framework was implemented in Go,[1] with the utilization of the Paxi library [Ailijiang et al. 2019][2] for reliable communication, quorum management for *phases* 1 and 2, and workload generator. A replicated key-value store available in Paxi was executed on top of the implemented service for evaluation purposes.

5.1. ISS framework design

The ISS framework employs *buckets*, implemented as doubly linked lists, to store client requests alongside a boolean value indicating whether the request has been decided. A key-value dictionary maps client commands to bucket items, enabling quick access. A mutex synchronizes the access to each bucket.

To facilitate easy retrieval of requests by segments, a *bucket group class* was developed. Each bucket group maintains a list of buckets assigned to a specific segment and implements the **getRequest** method, which retrieves commands from the associated buckets. If no values are available, the method can either immediately return an empty (no-op) command or wait for a specified duration, during which it checks for new requests. If no new request is added within the wait period, it returns a no-op command.

The *coordinator* forms the core of the ISS and is responsible for (i) managing epochs from initialization to termination, (ii) processing client requests, (iii) forwarding messages to the appropriate segments, and (iv) updating the log and delivering commands. To execute these functions safely and efficiently, the coordinator runs four parallel co-routines:

- **handleRequest**: Processes client requests and assigns them to the appropriate bucket using a hash function;
- **runEpoch**: Manages epoch initialization and forwards messages to the correct segments. This co-routine ensures that messages are not misdelivered to segments of different epochs by allowing only one thread to advance the epoch. It also handles messages destined to not yet initialized epochs by forwarding them to the bufferManager for later retrieval. During initialization, buckets are assigned to

[1]Implementation available: `https://github.com/g-buzzi/iss-paxos`
[2]Paxi library: `https://github.com/ailidani/paxi`

segments, and all segment instances are initialized for the current epoch. Epoch transitions are triggered by signals from **logManager** when all values in the current epoch are decided;

- **bufferManager**: Buffers messages for future epochs and forwards them to **runEpoch** once a new epoch begins. Messages from past epochs are ignored, while those destined for the current epoch are forwarded;
- **logManager**: Updates the log when signals are received from segment instances. It verifies the readiness of subsequent log positions, delivering values in sequence. If the current epoch's final value is delivered, it signals **runEpoch** to transition to the next epoch.

5.2. Adaptations for Paxos

Some adaptations were necessary to execute Paxos as a *Sequenced Broadcast (SB)* instance.

- *Limited Proposal Scope*: Each SB instance is restricted to deciding a fixed number of requests (s) based on the segment size. Only the initial leader of each instance can propose non-empty values;
- *Active Proposal Behavior*: Initial leaders no longer passively wait for client requests. Instead, a co-routine actively invokes the bucket group's **getRequest** method to propose values;
- *Concurrency Handling*: To address concurrency issues, a worker co-routine was introduced to parallelize the execution of methods for sending and receiving messages.

A simple failure detector ensures protocol termination. It uses a timer reset whenever the protocol progresses toward deciding a value. If the timer expires, the node initiates phase 1 of the Paxos algorithm to become the leader and decide any undecided values. If a value was already proposed for an instance, that value is re-proposed; otherwise, an **skip** message is sent.

As in Mencius [Mao et al. 2008], **skip** messages are optimized to require only a single **P2A** message for their decision. This optimization maintains correctness since: The initial leader sent the **skip**, and no other value can be decided because only the initial leader can propose a non-**skip** value; and a subsequent proposer sent the **skip** after becoming the leader, meaning the initial leader is no longer active and cannot propose non-**skip** values.

6. Experimental Evaluation

To assess the performance of the implemented algorithm, we conducted experiments using the Paxi framework, as it allows for a fair comparison among different Paxos implementations. For comparative analysis, experiments were also executed running the existing Paxi implementations of WPaxos and EPaxos, each representing a multi-leader and leaderless variant of the Paxos protocol. We performed experiments with 3 and 6 server replicas.

The experiments were conducted on Emulab [White et al. 2002]. Each node is configured with an Intel Xeon E5-2630 v3 processor, featuring 8 cores with hyper-threading, providing 16 virtual cores. The nodes had 65GB of RAM and a 1Gbps network interconnection. The OS used was Ubuntu 20.04 (64-bit), and the Go version was 1.22.1.

Each node was assigned either as a client or a server. Six nodes acted as clients, using Paxi's workload generator to produce requests and collect statistics. Each execution takes 60 seconds, and servers store 12,000 distinct keys. Client nodes gradually increase concurrency by increasing the number of concurrent clients, reaching at most 256 concurrent threads, all producing read operations to eliminate potential collisions in the EPaxos algorithm. To simulate a realistic geographical access pattern, each client determined the likelihood of accessing a key using a normal distribution characterized by parameters σ and μ. A fixed σ value was used across all clients, while the μ value varied for each client, making it so that each client accesses different subsets of keys with varying frequency.

Table 1 presents the experiment parameters for segment size and the waiting time before a leader proposes a no-op value when no client request is received. Larger segment sizes are expected to improve throughput by enabling more concurrent proposals and reducing the overhead associated with epoch initialization. Since fewer epochs are needed to decide the same number of values, the cost of initializing new epochs decreases. Regarding the waiting time for client requests, this strategy was anticipated to reduce the number of **skip** messages, potentially lowering latency by minimizing unnecessary network usage. Additionally, unless otherwise stated, **skip** messages were not optimized using the approach described in Section 5.2.

Table 1. ISS parameters.

Name	Segment Size	Request waiting time (ms)
ISS_32s_0t	32	0
ISS_32s_30t	32	30
ISS_64s_0t	64	0
ISS_64s_30t	64	30
ISS_128s_0t	128	0
ISS_128s_30t	128	30

6.1. Performance analysis: running 3 replicas

For the experiments with three replicas, Figure 7 shows that the throughput of the implemented framework was nearly double that of WPaxos and EPaxos. This performance gain is likely due to the parallelism inherent in the ISS framework, which allows multiple nodes to propose values simultaneously and enhances system resource utilization. Regarding the impact of ISS parameters, a slight increase in throughput was observed when using larger segments. As for the waiting time before proposing a no-op value, it did not significantly affect throughput. This is because, under high-load conditions, it is rare for any replica node to encounter an empty bucket.

As shown in Figure 8(a), WPaxos and EPaxos achieved better latency, with median values close to 1 ms when processing 5,400 requests/s. ISS exhibited higher latencies, ranging from 14 to 60 ms. This increase is due to two factors: in addition to the time required for a value to be decided, clients may also have to wait for the node handling their request to become the leader of the segment responsible for accessing the corresponding bucket. Results indicate that larger segment sizes, which lead to longer bucket rotation times, also increase latency, corroborating this observation. When requests were not delayed, latency remained consistent throughout execution. This means that proposing **skip** messages had the same impact on latency as proposing actual values. However,

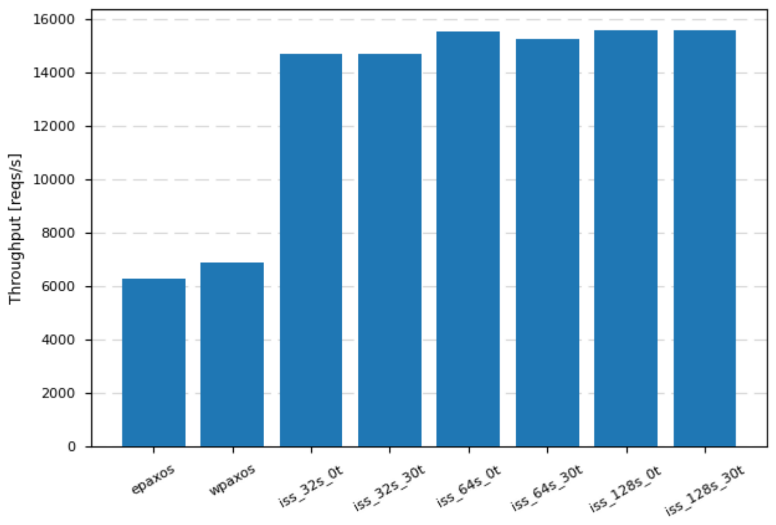
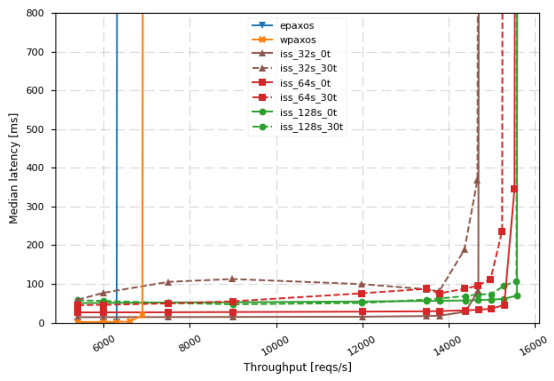
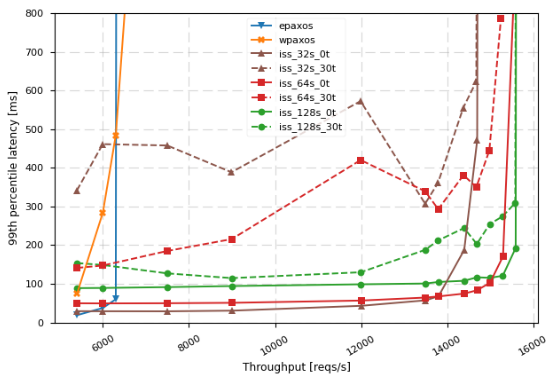


Figure 7. Maximum throughput for 3 replicas.

when replicas are configured to wait 30 ms before proposing, latency increases due to the extended epoch durations caused by the waiting period.



(a) median latency



(b) 99^{th} percentile latency

Figure 8. Latency impact for 3 replicas.

Instead of sending requests to a designated server, clients randomly selected a server node for each request. This approach was intended to partition client nodes into buckets accurately. However, it introduced random variance in the 99^{th} percentile latency, as shown in Figure 8(b). To address this issue, in subsequent experiments, ISS client behavior was modified to align with the other algorithms, where each client exclusively sent requests to a single server.

6.2. Performance analysis: running 6 replicas

As the number of replicated servers increased, all evaluated algorithms exhibited a reduction in throughput, as shown in Figure 9. The most important relative decrease occurred in EPaxos, with a 14% drop, followed by ISS, with an average reduction of $\approx 10\%$, and WPaxos, losing only 4% of its maximum throughput. This suggests that WPaxos may offer better scalability than our Multi-leader Paxos based on ISS.

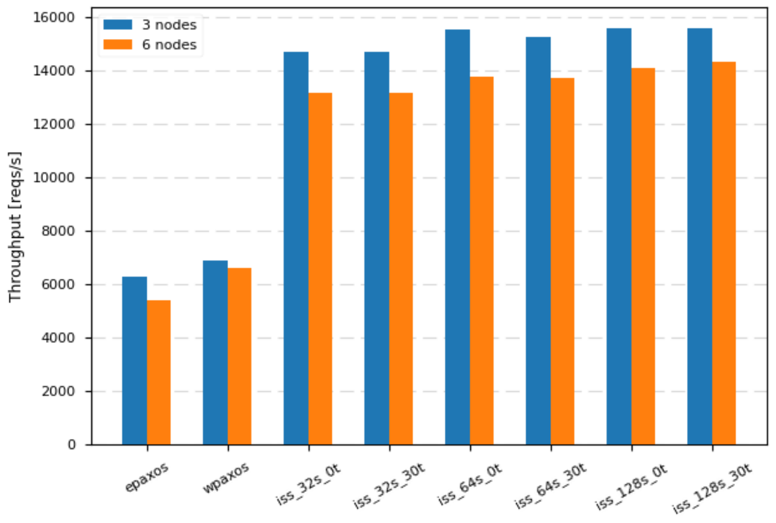
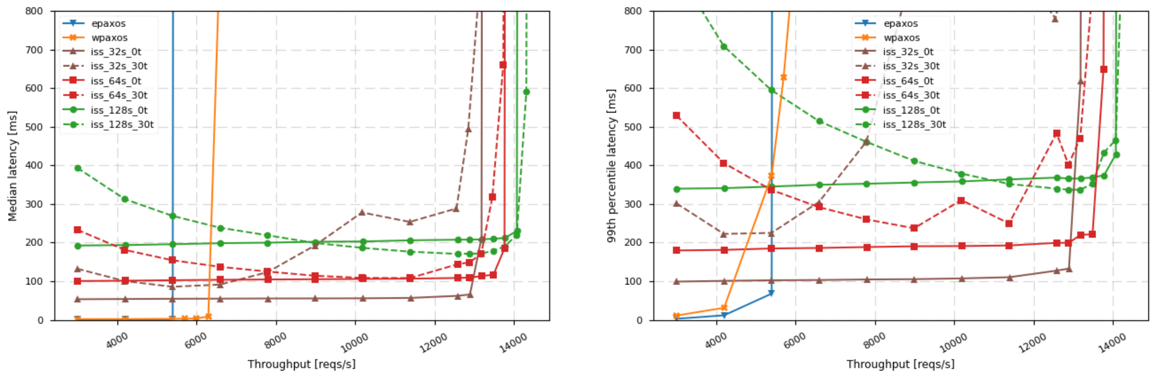


Figure 9. Maximum throughput comparison between 3 and 6 replicas.

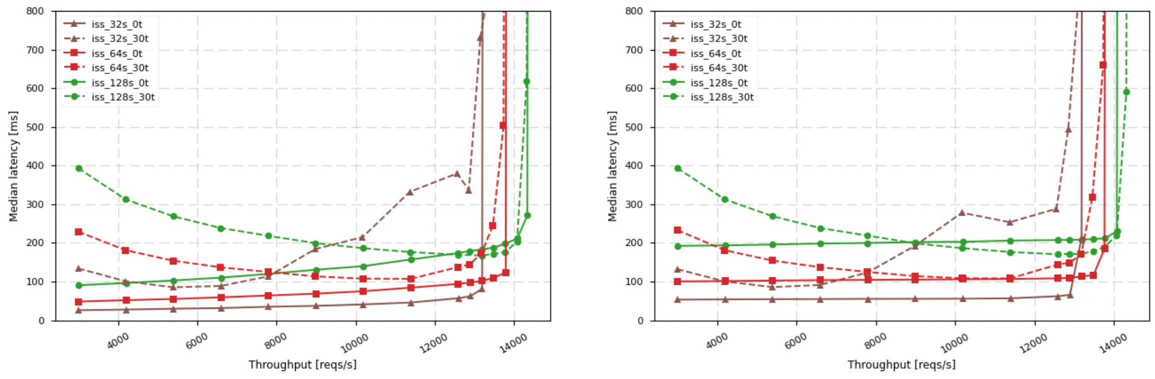
Regarding median latency (see Figure 10(a)), EPaxos and WPaxos exhibited relatively minor increases, remaining below 2ms while processing 5,400 requests/s. In contrast, ISS's latency nearly quadrupled. This increase can be attributed to the higher number of messages required for each decision and to the larger epoch sizes, which extend client waiting times for bucket rotations. With more balanced client loads, the behavior of nodes that wait for requests becomes more apparent. In both median and 99^{th} percentile latency measurements, low-throughput scenarios resulted in higher latency for waiting nodes than their non-waiting counterparts, as bucket rotations took longer. However, as throughput increased, the latency of waiting variants became comparable to the non-waiting ones. This is because waiting occurred less frequently at higher loads and was more likely to result in an actual request being selected when it did occur.



(a) median latency (b) 99^{th} percentile latency

Figure 10. Latency impact for 6 replicas.

By optimizing **skip** messages, allowing them to be decided with a single **P2A** message, we see in Figures 11(a) and (b) that while ISS configurations exhibit similar behavior in high-throughput scenarios, the optimization reduces latency in lower-throughput conditions. This is because **skip** messages are rarely sent near the protocol's saturation point, limiting their impact at higher loads. More specifically, at 5,400 requests/s, where **skip** messages are more frequent, latency reduced by $\approx 50\%$. As throughput increases, latency steadily rises as fewer **skip** messages are sent, eventually converging to values close to the non-optimized version when **skip** messages become negligible.



(a) with optimization (b) without optimization

Figure 11. Latency impact for 6 replicas with and without skip optimization.

7. Conclusion

This work proposes a multi-leader Paxos protocol using the ISS framework to enhance parallelism in consensus decisions. It reviews key performance-optimized Paxos variants from the literature and discusses strategies for improving efficiency. We outline the necessary adaptations for running Paxos on the ISS framework and evaluate its performance against high-performance Paxos variants, WPaxos and EPaxos. All implementations were developed using the Paxi library [Ailijiang et al. 2019], which provides essential components for Paxos and replication protocols, ensuring a fair comparison of throughput, latency, and scalability. Performance analysis shows that the proposed implementation outperforms other protocols, achieving nearly twice the throughput of WPaxos and EPaxos, albeit with increased latency. This latency increase results from the higher number of messages required per decision and larger epoch sizes, which extend client waiting times for bucket rotations. Additionally, as the number of replicated servers increases, our approach maintains good scalability, as expected from the ISS framework. In future work, we will evaluate the performance of our Paxos implementation and competing protocols in geo-replicated scenarios, where ISS is expected to perform particularly well.

References

Ailijiang, A., Charapko, A., and Demirbas, M. (2019). Dissecting the performance of strongly-consistent replication protocols. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1696–1710.

Ailijiang, A., Charapko, A., Demirbas, M., and Kosar, T. (2020). WPaxos: Wide Area Network Flexible Consensus. *IEEE Trans. Parallel Distrib. Syst.*, 31(1):211–223.

Cachin, C., Guerraoui, R., and Rodrigues, L. (2014). *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition.

Castro, M. and Liskov, B. (1999). Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, page 173–186.

Charapko, A., Ailijiang, A., and Demirbas, M. (2021). Pigpaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of the 2021 International Conference on Management of Data*, page 235–247.

Howard, H., Malkhi, D., and Spiegelman, A. (2016). Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*.

Jalili, P., Primi, M., Schiper, N., and Pedone, F. (2010). Ring paxos: A high-throughput atomic broadcast protocol. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 527–536.

Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.

Lamport, L. (2001). Paxos made simple. *Sigact News - SIGACT*, 32.

Lamport, L. (2006). Fast paxos. *Distrib. Comput.*, 19(2):79–103.

Lamport, L., Malkhi, D., and Zhou, L. (2009). Vertical paxos and primary-backup replication. Technical Report MSR-TR-2009-63, Microsoft Research.

Lamport, L. and Massa, M. (2004). Cheap paxos. In *International Conference on Dependable Systems and Networks (DSN 2004)*.

Mao, Y., Junqueira, F. P., and Marzullo, K. (2008). Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, page 369–384.

Moraru, I., Andersen, D. G., and Kaminsky, M. (2013). There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, page 358–372.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, page 305–320.

Regis, S. and Mendizabal, O. (2022). Análise comparativa do algoritmo paxos e suas variações. In *Proceedings of the 23rd Workshop on Testing and Fault Tolerance*, pages 71–84.

Stathakopoulou, C., Pavlovic, M., and Vukolić, M. (2022). State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*, page 17–33.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review*, 36:255–270.