PigPaxos: Devouring the communication bottlenecks in distributed consensus

Aleksey Charapko
University at Buffalo
acharapk@buffalo.edu

Ailidani Ailijiang
Microsoft
aiailiji@microsoft.com

Murat Demirbas
University at Buffalo
demirbas@buffalo.edu

Abstract

Paxos family of protocols are employed by many cloud computing services and distributed databases due to their excellent fault-tolerance properties. Unfortunately, current Paxos deployments do not scale for more than a dozen nodes due to the communication bottleneck at the leader. PigPaxos addresses this problem by decoupling the communication from the decision-making at the leader. To this end, PigPaxos revises the communication flow in Paxos to replace direct communication between the leader and followers with a relay/aggregate based message flow. Although aggregation-based approaches have been employed in the context of weak-consistency replication protocols, PigPaxos shows how they can be effectively integrated into the strong consistency distributed consensus protocols.

We implement and evaluate PigPaxos, in comparison to Paxos and EPaxos protocols under various workloads over clusters of size 5 to 25 nodes. We show that PigPaxos can provide more than 3 folds improved throughput over Paxos and EPaxos with little latency deterioration. Our experiments also show that the aggregation has negligible overhead for the latency of PigPaxos as compared to the latency of Paxos. We conjecture that PigPaxos would be useful for implementing geo-replicated distributed databases with tens of replicas distributed over many regions around the globe.

1 Introduction

Paxos family of protocols are immensely useful because of their excellent fault-tolerance properties. Many cloud computing services and distributed databases employ Paxos for state machine replication (SMR). Paxos protocols preserve the safety of consensus problem (no two nodes commit different values for the same slot) even to the face of a fully asynchronous execution, crash faults, message losses, and network partitions. Paxos protocols satisfy liveness of consensus problem (some value is eventually committed for the slot) when the system moves outside the realm of the coordinated attack [16] and FLP [15] impossibility results.

One factor that limits adoption of Paxos, is its throughput bottleneck. As all of the protocol's communication drains through a single node – the leader, this hinders the scalability of Paxos, because adding more nodes increases the load on the leader linearly and reduces its performance [2].

An obvious solution to the single leader bottleneck problem is to use sharding. Sharding is useful for scaling Paxos deployments horizontally. A trivial example of sharding is static sharding by deploying Paxos groups, as done by Spanner and CockroachDB. Many Paxos variants explore in-Paxos sharding techniques, and use multiple nodes for leading different Paxos rounds [3,5,20,21]. Such approaches often degrade the "linearizability of commands" guarantee that Paxos provides with its single leader. For example, EPaxos [20] solves the generalized consensus problem and provides only a partial order of commands, while WPaxos [3] provides linearizability per each predefined conflict domain and never orders commands across different conflict domains.

While sharding is great for horizontal scaling, there are several applications that require *vertically scaling* Paxos to run on a large number of nodes all within the same conflict domain. Few application areas where vertical scaling of consensus is needed are distributed ledger based multi-party blockchains [4], cryptocurrencies [6, 27], and adversarial-commerce [17]. Another domain where vertical scaling of consensus is needed is consistent cloud configuration management [25]. Configuration management is required for gating new product features, conducting experiments (A/B tests), performing application-level traffic control, performing topology setup and load balancing, monitoring and remediation, updating machine learning models (which varies from KBs to GBs), controlling applications' behaviors (related to caching, batching, prefetching etc) [25].

Although there is motivation and need for vertically scaling consensus deployments, current Paxos deployments do not scale for more than a dozen nodes due to the aforementioned communication bottleneck at the leader. Moreover, even for Paxos clusters with a small number of nodes, large messages (such as database replication as in CockroachDB [13] and

Spanner [14]) trigger the communication bottleneck at the leader. It is time we resolve this problem.

Contributions of the paper. To address the communication bottlenecks at single-leader Paxos protocols, we introduce PigPaxos. PigPaxos manages to reduce the leader's communication bottleneck by decoupling the decision-making at the leader from the communication at the leader. To this end, PigPaxos replaces the direct communication between the leader and followers with a relay/aggregate based message flow. In particular, at each round, the leader tasks randomly selected nodes from acceptor groups to relay its message to the rest of the acceptors, and to perform in-network aggregation of acknowledgments back from the acceptors. This dynamically rotating communication tree approach allows the leader to only communicate with a small number of relaying nodes. The random alternation of the relay nodes also shields these nodes from becoming hotspots themselves: the extra traffic load a relay node incurs in one round is offset in consecutive rounds when the node no longer serves as relay. These two properties combined improve the throughput scalability of the system.

Although aggregation-based approaches have been known and employed in the context of weak-consistency replication protocols, PigPaxos shows how they can be effectively applied to and integrated with the strong consistency distributed consensus protocols. The communication piggybacking and aggregation technique used in PigPaxos is applicable to many Paxos implementations [22, 23] and variants [3, 18]. Despite significantly improving scalability and performance, PigPaxos reuses the same correctness proof as Paxos, because it preserves the core protocol, and modifies only the communication implementation to improve scalability and performance of the protocol. Moreover, PigPaxos tolerates up to f node failures in a cluster of size $2f+1$, just like the classical Paxos protocol.

The basic dynamic communication overlay tree scheme used in PigPaxos lends itself to many optimizations. We implement PigPaxos in our Paxi framework [2] and evaluate several of these optimizations. We employ Paxi as a level playground to compare and benchmark Paxos, EPaxos, and PigPaxos in the context of the same in-memory key-value store implementation. We conducted our experiments on AWS EC2 nodes, and tested the protocols from 5 nodes upto 25 nodes in various LAN and WAN topologies.

Our experiments show that PigPaxos is very effective in improving scalability of consensus with respect to increasing number of nodes. For a 25 node deployment, using a workload of random selection of a key from a 1000 item database, EPaxos throughput get saturated at 1000 requests per second, Paxos throughput reaches its limit of around 2000 req/sec, whereas PigPaxos can scale to 7000 req/sec throughput with little latency deterioration.

Our experiments also show that the aggregation has negligible overhead for the latency of PigPaxos as compared to the latency of Paxos. Another interesting result from our evaluation is that PigPaxos provides benefits over Paxos for consensus clusters as small as 5 nodes. In particular, due to aggregation at the relay nodes, PigPaxos throughput scales with respect to the size of the messages. This makes PigPaxos applicable for distributed database systems that use Paxos for replication such as CockroachDB and Spanner. We conjecture that PigPaxos would be useful for implementing geo-replicated distributed databases with tens of replicas distributed over many regions around the globe.

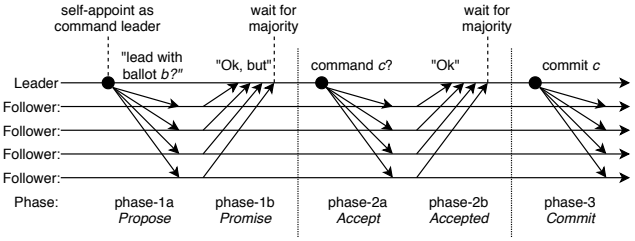


Figure 1: The three phases of Paxos protocol

We confirm our experimental observations with the analytical modeling of protocol bottlenecks, and show the extent to which PigPaxos helps shift these bottlenecks from the leader to the followers. We also provide formulas to estimate relative bottlenecks of different PigPaxos configurations. For instance, we demonstrate that the cluster size has limited and capped impact on follower load, while the number of relay groups significantly affects the leader.

2 Background and Related Work

2.1 Paxos

The Paxos protocol operates in three phases as illustrated in Figure 1. The first phase, often called propose phase, establishes a leader node. In this phase, a node tries to acquire leadership by reaching out to other nodes with some ballot number. The replicas ack the leadership proposal only if they have not seen a higher ballot. When a node collects a majority of acks, it can proceed to the second phase. In phase-2, the accept phase, the leader tells all the followers to accept a command. The command is either a new command of the leader's choosing, or an old command if some nodes replied during phase-1 with an earlier uncommitted command. Once the leader receives a majority of acks from nodes accepting the command, the command is anchored, and the leader proceeds to the commit phase (phase-3). The leader then instructs the followers to finalize the command in the log and execute it against their state machines provided that there is no gap in their logs up to this command's slot.

This basic Paxos protocol is commonly extended to the Multi-Paxos [26] protocol, which allows the same leader to propose commands in subsequent slots (i.e., consensus

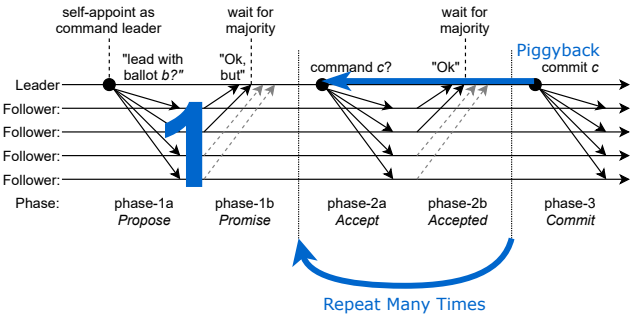


Figure 2: Multi Paxos optimization

instances) as illustrated in Figure 2. The leader election in phase-1 is performed only once, and this cost is avoided in subsequent consensus instances as long as the incumbent leader's ballot number remains the highest the followers have seen. Additionally, phase-3 is piggybacked to the next phase-2, further reducing the communication overheads. In this paper when we mention Paxos protocol, we refer to this Multi-Paxos optimization, and apply the Multi-Paxos for PigPaxos with a stable leader as well.

PigPaxos overhauls the Paxos protocol communication implementation. Paxos safety and liveness proofs are oblivious to how the communication between the leader and followers are implemented – we expand on this in Section 3. Therefore, PigPaxos is an orthogonal technique that can be applied in the context of many Paxos variants for improving throughput.

2.2 Flexible quorums

The flexible quorums result from 2016 [18] weakens Paxos' "all quorums should intersect" assertion to instead "only quorums from different phases should intersect". That is, it shows that majority quorums are not necessary for Paxos, provided that phase-1 quorums (Q1) intersect with phase-2 quorums (Q2).

The flexible quorums result allows trading off Q1 and Q2 sizes to improve Paxos performance (to the detriment of fault-tolerance). Assuming failures and resulting leader changes are rare, phase-2 (where the leader tells the acceptors to decide values) is run often, whereas phase-1 (where a new leader is elected) execution is rare. Thus it is possible to improve performance of Paxos by reducing the size of Q2 at the expense of making the infrequently used Q1 larger. For example, for a 10 node deployment, we can safely allow any set of only 3 nodes to participate in phase 2, provided that we require 8 nodes to participate for phase 1. Note that while the majority quorums (Q1=Q2=6) would be able to mask upto 5 node failures, the Q1=8 configuration can only with stand upto 2 node failures as it needs 8 nodes to be able to perform phase-1.

While using a smaller Q2 is beneficial for reducing latency, it does not reduce the leader bottleneck and does not help improving scalability or throughput. This is because flexible Paxos talks to all nodes and takes the first $|Q2|$ responses for satisfying phase-2 requirement, although the other responses also arrive and overwhelm the leader. While it is possible to use a thrifty optimization [20] and contact only $|Q2|$ nodes for phase-2, in that case a single faulty or sluggish node in Q2 stalls the performance.

In contrast to flexible quorums approach, PigPaxos clears the bottleneck at the leader while also preserving the robustness against failures and resilience to the effects of faults on performance. It is possible to employ PigPaxos approach with the flexible quorums idea, for providing improved scalability and throughput to flexible Paxos [18].

2.3 Multi-leader consensus protocols

Scalability bottlenecks have been a big problem in consensus research and have received a lot of attention. Mencius [5] uses rotating leaders to alleviate the bottleneck at a single leader to improve throughput. For Mencius to work effectively, the workload has to be partitioned across each leader nicely, so that each leader has something to propose in their turn. Stalls in rotating leader proposals create problems with Mencius performance, and delays and failures cause tricky corner cases. In contrast, PigPaxos is simple to implement and avoids these issues. Instead of rotating leaders, PigPaxos rotates the small number of relay nodes the leader interacts with. This clears the communication bottlenecks and improves scalability and throughput, while still keeping the protocol simple and avoiding the complexity and performance complications Mencius suffers from.

Canopus [24] is a multi-leader consensus protocol aimed at wide area deployments. Canopus trades off latency and fault-tolerance for throughput. The protocol heavily relies on batching and requires a lot of communication steps to finish a single consensus round. It divides the nodes into a fixed number of groups, called super-leaves. Canopus assumes synchronized rounds, and a network-based reliable broadcast primitive at the super-leaf level. On the first cycle, each node within the super-leaf exchanges the list of proposed commands with other super-leaf peers using the reliable broadcast primitive. Every node then orders these commands in the same deterministic order, creating a virtual node that combines the information of the entire super-leaf. In consecutive cycles, the virtual nodes exchange their commands with each other at increasingly higher levels until a single super-node emerges as the root of the tree. At that point every physical node has all commands in the same order and consensus has been reached across all super-leaves. Canopus suffers from fault-tolerance problems, as it cannot tolerate a network partition or failure of any single super-leaf. Since the nodes that constitute a super-leaf need to located within the same rack (for using the reliable broadcast primitive), Canopus is especially prone to super-leaf failures and loss of unavailability. In contrast to Canopus, PigPaxos can tolerate failures of up to

half of the nodes in the system. Moreover, PigPaxos achieves high-throughput without an observable increase in latency in most cases, as we show in Section 5.7.

In 2017, we introduced WPaxos [3], a WAN-optimized multi-leader Paxos protocol. WPaxos selects leaders (and maintains logs) to be per-object, and employs an object stealing protocol, with adaptive stealing improvements to match the workload access locality [10]. More specifically, multiple concurrent leaders coinciding in different zones steal ownership of objects from each other using phase-1 of Paxos, and then use phase-2 to commit update-requests on these objects locally until they are stolen by other leaders. To achieve fast phase-2 commits, WPaxos leverages the flexible quorums result [18] and appoints phase-2 quorum Q2 to be close to their respective leaders. The multi-leader, multi-quorum setup in WPaxos helps with both WAN latency and throughput due to the smaller and geographically localized quorums. In contrast to WPaxos that uses multiple leaders and multiple conflict domains, PigPaxos improves throughput and scalability using a single leader within a single conflict domain. However, PigPaxos optimization is still applicable within the framework of WPaxos. In addition to their Q2 used for committing, WPaxos leaders can employ PigPaxos for implementing full replication to a large number of nodes and learning when these additional nodes commit as well. This technique could be useful for implementing bounded-staleness [1] at large-scale geo-replicated database deployments.

Fast Paxos [19] and EPaxos [20] removes the requirement of partitioning of conflict domains between nodes prescribed in the previous multi-leader approaches, and instead try to opportunistically commit any command at any node using an opportunistic leaders approach. Any node in EPaxos becomes an opportunistic leader for a command and tries to commit it by running a phase-2 of Paxos in a super-majority quorum system. If some other node in the super-majority quorum is also working on a conflicting command, then the protocol requires performing a second phase to record the acquired dependencies, and agreement from a majority of the Paxos acceptors to establish order on the conflicting commands is needed. EPaxos suffers from increased number of conflicting commands for heavy workloads, and its performance plummets. In Section 5.7, we give performance comparison of EPaxos and PigPaxos. In contrast to EPaxos, PigPaxos provides linearizability of all operations and clears communication bottlenecks and achieves high throughput within the framework of a simple single leader Paxos implementation.

3 PigPaxos

3.1 System Model

Similar to Paxos, PigPaxos considers a crash failure model in which nodes may silently crash and later recover. Such model also allows for network partitions, since partitioned nodes behave similarly to crashed ones when viewed by the nodes on the other side of a partition. PigPaxos preserves safety but not progress under a fully asynchronous model. It achieves progress in a partially synchronous environment where failure detectors stabilize enough to allow a single leader to emerge.

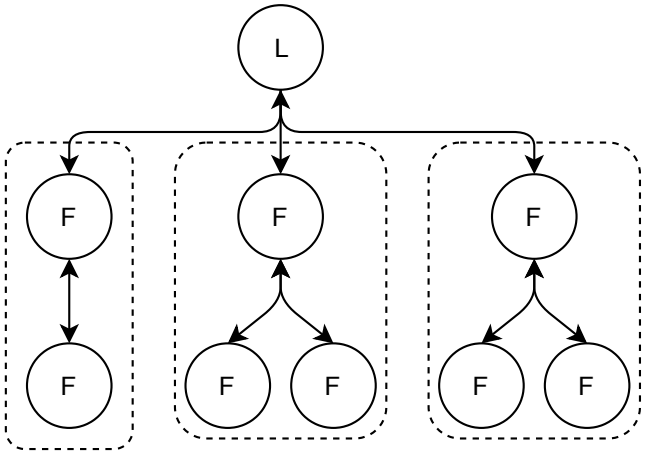


Figure 3: PigPaxos communication flow. All nodes are divided into groups, and the leader communicates only with a random node from each group. The nodes contacted by the leader relay the messages to their peers within the group and aggregate their responses to reply back to the leader.

3.2 Communication Flow

Communication in Paxos exhibits two distinct patterns: a fan-out communication occurs when a leader sends messages out to the followers, and a fan-in behavior occurs when these followers reply back to the leader. Both patterns are present in phase-1 and phase-2 of the protocol, whereas phase-3 contains only the fan-out communication. While the communication in Paxos is direct between the leader and the followers, we observe that it is possible to employ intermediate nodes or proxies in these paths to help relay and aggregate the messages.

Leveraging this observation, PigPaxos adopts a communication tree pattern to reduce the message exchange bottleneck at the leader. Instead of sending the fan-out messages to all of the followers, the leader transmits these to a small set of relay nodes, which propagate these messages to the remaining followers. The relay nodes also act as aggregators for the fan-in communication of the followers' responses, and pass the combined results to the leader.

Figure 3 illustrates how a dynamic communication tree is overlaid over the cluster of nodes. Initially all nodes are statically divided into a number of distinct relay groups. This grouping may happen with the help of a hash function or may be pre-defined in terms of the cluster's topology. For example, in geo-distributed setup, we may pre-define the groups based

on the regions or datacenters in which nodes are located. It is possible to extend the communication flow beyond a single layer of relays groups by breaking the groups into nested subgroups and relay nodes in those subgroups. Such nesting creates a multi-level communication tree pattern, with messages propagating from the leader through multiple layers of relay nodes down to the followers at the tree leaves. However, as we discuss in Section 6.3, such nesting is unwarranted and will not improve the performance in most cases. Therefore, for the sake of simplicity, we present PigPaxos with just a single layer of relay groups.

In Figure 4 we demonstrate the communication flow of PigPaxos for Phase-1, Phase-2, and Phase-3. Note that, using Multi-Paxos optimization with a stable leader, PigPaxos would only perform Phase-2 for each consensus instance, with Phase-3 messages piggybacked to Phase-2. We break-down and describe the communication flow consisting of a fan-out and fan-in as follows.

1. When the leader starts a fan-out communication, it picks a random node from each group as a relay node for the message. The random rotation of relay nodes is important for load-balancing the communication burden to all followers and avoiding hotspots. As relay nodes alternate, the extra load a relay node suffers in one round is offset in the consecutive rounds when that node ceases to be a relay.

2. Upon receiving a message from the leader, a relay node processes this message as a regular follower and resends the message to the remaining nodes in its relay group.

3. Upon receiving the messages, the followers start the fan-in pattern by responding back to the relay node as if they were responding directly to the leader.

4. The relay nodes wait for the followers' responses and piggyback them together into a single message. By default, the relays wait for all followers in the group to respond. Since such unbounded wait may create performance problems when some follower becomes sluggish or fail, PigPaxos employs a tight timeout at the relay nodes. If some followers do not reply within the timeout, a relay stops waiting and replies to the leader with all responses collected so far.

3.3 Paxos to PigPaxos mapping

PigPaxos generalizes the communication implementation of the Paxos protocol. Paxos has $N-1$ groups, where each group has one element and the groups do not intersect with each other. In contrast in PigPaxos there are p groups where $p \in \{1..N-1\}$, and the cardinality of the union of p groups is $N-1$. This formulation allows the subgroups to intersect with each other, which may have advantages in terms of adding redundant channels to reach some nodes. However, for simplicity's sake, we preclude intersecting groups in the rest of the paper.

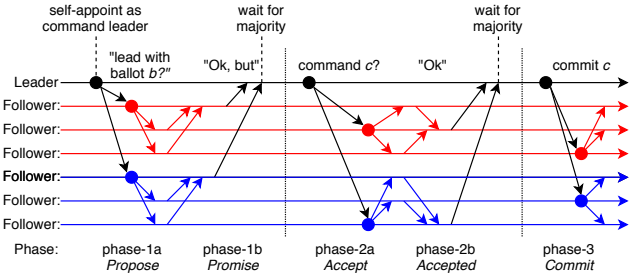


Figure 4: The three phases of PigPaxos with a single level communication relay.

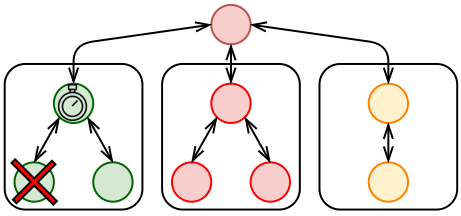
We note that the safety and liveness proofs of Paxos do not depend on the communication implementation between the leader and follower nodes. In Paxos, maintaining correctness in spite of failures is guaranteed by quorum size and the information exchanged within the quorums, and the proofs are oblivious to how communication of this information is achieved. Therefore, PigPaxos preserves the safety and liveness properties of Paxos, as it only modifies the communication implementation. For reasoning about liveness, the message flow pattern and the use of relay nodes requires special attention, as failure of a relay node has disproportional impact compared to the failure of a regular follower. Since PigPaxos uses random selection of relay/aggregator nodes at each round, it circumvents this problem and retains liveness.

3.4 Fault Tolerance

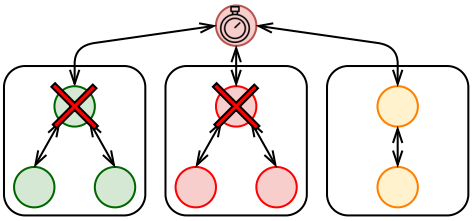
PigPaxos tolerates up to f node failures in a cluster of size $2f+1$, just like the classical Paxos protocol. To present the fault-tolerance of PigPaxos in a simpler manner, we distinguish two common failure cases: a follower failure and a relay node failure.

The protocol deals with follower failure by enacting a tight timeout on the relay node – the relay node does not wait for responses indefinitely and will reply to the leader either upon collection all children responses or reaching a timeout. The relay timeout procedure allows partial set of responses to reach the leader in hopes that the majority quorum is still satisfied. We illustrate this type of failure in Figure 5a. In many cases the leader remains unaffected by the timeout in a minority of relay groups, as the majority quorum may be reached by votes from other relays groups. [1]

[1] If a multi-level grouping structure is used, the timeouts imposed on each level of the tree will be different to account for the depth of the relay-nodes. The relays closer to the leaves have shorter timeout, and relays at higher levels get progressively larger timeouts in order to include the timeouts of the children.



(a) Follower failure causes the relay node to time-out before responding to the leader.



(b) Relay node failures may break message delivery, causing leader to timeout and retrying with different relays.

Figure 5: Failure scenarios at follower and relay nodes.

Relay node failures are more severe and may require a different treatment if large enough number of relays are faulty. When a relay fails, no messages propagate to all the followers in the relay group, creating for a possibility of a minority crashes to completely halt the communication. The random selection of relay nodes allows for such communication collapse to be transient. Similar to the relay nodes, the leader too has a timeout. Upon reaching this timeout with no majority votes collected, the leader restarts the communication with a new random set of relay nodes. Figure 5b demonstrates this failure scenario.

The timeouts employed do not affect safety of the protocol, as they only hinder message delivery and do not affect decision making at the leader node. The possibility of duplicate message delivery also does not pose a problem, because messages are protected by the leader's ballot number and can be identified as duplicates versus coming from a different leader. As a result, if some new leader is in better position to communicate with the cluster, it can get elected with higher ballot and proceed, while resend attempts by the old leader will fail as having an obsolete ballot.

4 Additional Optimizations

4.1 Dynamic Relay Groups

At each round, the PigPaxos leader randomly selects a node from each relay group to carry out relay and aggregation for the group on its behalf. In the basic scheme described in Section 3, we treated the groups as if they are static and predefined. It is possible to change the configuration of relay groups on-the-fly to improve the performance or to react quickly to some crashes in some of the groups. Consider a scenario where a particular configuration of relay groups starts to experience degradation in performance. A leader may reshuffle the nodes to create a different configuration of relay groups to improve performance. It is also possible for the leader to specify a new relay group for each round of the protocol. This may be achieved by either including new group membership information in every message send by the leader, or by generating a seed with which relay nodes can compute their groups for a given round.

Another relay group optimization is to allow groups to overlap. Although this decreases efficiency due to increasing the number of messages sent and received, having alternate paths to reach nodes will improve the reliability and performance consistency in environments with link volatility and failures.

4.2 Partial Response Collection

In the basic PigPaxos protocol, the relay nodes wait to hear from all nodes in their groups before they aggregate and forward these messages to the leader.[2] If some nodes fail to respond, the relay nodes time-out and send whatever messages were collected to the leader. This may cause slowdowns when some nodes in the groups are sluggish or crashed. One way to mitigate the problem is to divide the response process into stages, and to perform threshold based responses to the leader, instead of waiting responses from all nodes in the group. For example, the relay node of group i of size n_i may only wait for g_i nodes to reply before sending its first aggregated reply back. It is still important that the leader collects at least a majority votes across all relay groups, so the g_i must be chosen accordingly: $\sum_{i=1}^{R} g_i >= \lfloor \frac{N}{2} \rfloor + 1$, where R is number of relay groups, and N is total number of nodes. This will improve the per-round latency of PigPaxos as well as reduce momentary slowdowns due to some sluggish or failed nodes.

4.3 Improving Reads

Reading from Paxos-backed state machines typically happens in one of three different ways: reading from any node, reading from the leader node, or reading by the virtue of serializing the read operation in the log (the approach we take in the basic PigPaxos protocol). Unfortunately, all these methods have signficant drawbacks. Reading from any replica, while fast, compromises the consistency guarantee, since the state at a replica may be stale [12]. Reading from a "stable" leader requires additional mechanism, such as leader leases, to prevent "split-brain" when multiple leader candidates emerge. Finally, performing the read by serializing the read operation

[2] Of course, if a relay node receives a reject message (either at Phase-1 or Phase-2) it does not wait for aggregation and send this rejection to the leader immediately.

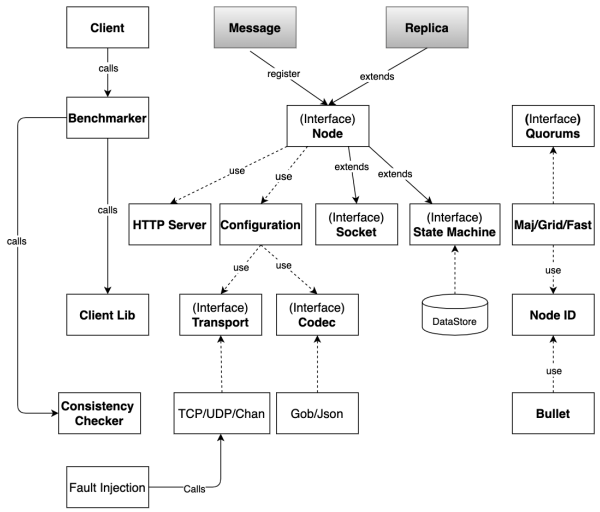


Figure 6: **Paxi modules usage where user implements Messages and Replica type (shaded)**

on the log incurs costs since it requires a full Paxos round to complete.

Our recent work on Paxos Quorum Reads (PQR) [11] provides an alternative method which provides strong consistency guarantees and avoids communication with the leader node as well as the need to perform lease management. Although PQR is a client-based solution, its communication pattern is similar to that of regular phase-2 of Paxos, therefore PQR can be adopted to use the PigPaxos relay groups idea with the client serving as the root of the tree. This however, requires the client to know all nodes in the cluster in order for it to construct random communication trees for each read operation, which may not be practical in some situations. In such cases, any PigPaxos replica can act as a proxy for the PQR and drive the read operation on behalf of the client contacting it.

5 Evaluation

5.1 Implementation

We implemented PigPaxos in Go using Paxi [2]. Paxi is an open-source project https://github.com/ailidani/paxi that we developed for prototyping, evaluating, and benchmarking consensus and replication protocols. As shown in Figure 6, Paxi readily provides most functionality that any coordination protocol needs, including network communication, state machine of a key-value store, client API and multiple types of quorum systems.

Our implementation of PigPaxos in the Paxi framework required almost no changes to the core Paxos code, and focused only on the message passing layer and relay group orchestration. The entire protocol was implemented in just 1200 lines of code.

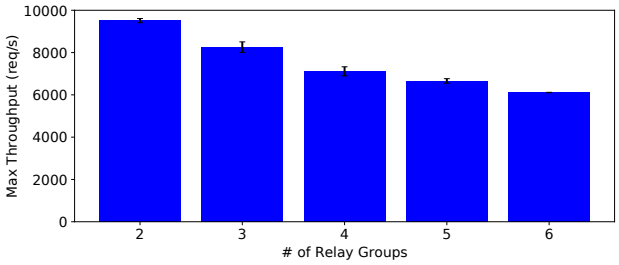


Figure 7: Number of relay groups on 25-node PigPaxos with single relay layer.

5.2 Testbed

We conducted our experiments on AWS EC2 m5a nodes. All Paxos and PigPaxos nodes used m5a.large instances with 2vCPUs. To ensure that the client performance does not impact the results, we used the larger m5a.xlarge instances with 4 vCPUs for the clients. The Paxi benchmark provides an in-memory key-value store backed by the protocol under evaluation. In all experiments, we used 1000 distinct 8 byte keys, with uniform probability of picking the key for each request. Unless otherwise stated, we used small 8 byte payloads for the value in every key value-pair. In most of our experiments we use a workload with even distribution of read and writes operations.

5.3 Number of Relay Groups

The performance of PigPaxos is controlled mainly by the number of relay groups used for message dissemination and aggregation. A basic strategy for determining the number of relay groups in a scale-free manner is to use $\sqrt{N}$ as the number of groups. For example, $N = 25$ implies using 5 groups each with 5 nodes. This makes the leader to communicate with 5 relay nodes and the relay nodes to talk to 4 of their peers, a seemingly load-balanced setup. However, when we evaluate the maximum throughput of 25 nodes PigPaxos cluster with varying number of leader relay groups, we find that this strategy performs badly as shown in Figure 7.

We find that the best throughput is achieved with the smallest aggregation factor, using only two relay groups. The reason is that this setup minimizes the communication bottleneck at the leader the most, and shifts large portion of the message exchange overheads to the followers. The relay nodes do not get overwhelmed when talking to 12 other nodes in their relay groups because the relay nodes are rotated at every round. As relay nodes randomly rotate, the extra load in one round is offset in consecutive rounds when the node no longer serves

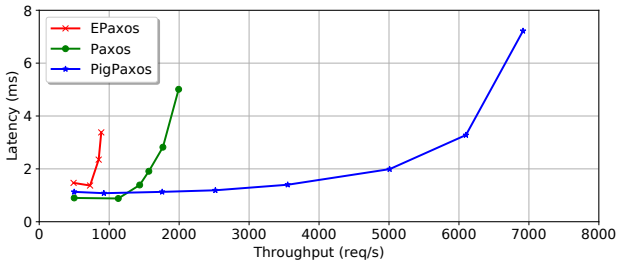


Figure 8: Latency and Throughput on 25-node cluster with 3 relay nodes.

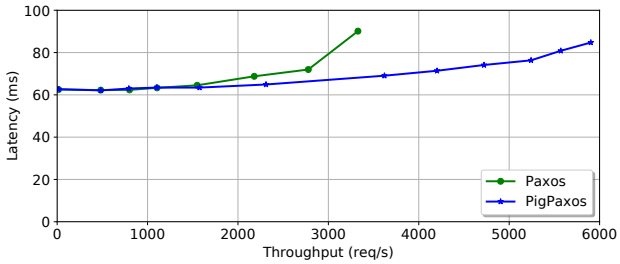


Figure 9: Latency and Throughput on 15-node WAN cluster in Virginia, California and Oregon.

as relay. We discuss and analyze this observation further in Section 6.1.

5.4 Latency and Throughput

In the presence of relay nodes, there is an overhead latency added to the consensus. To study the impact relay nodes have on PigPaxos latency, we compare PigPaxos latency with that of a corresponding traditional Paxos deployment. We also compare with Egalitarian Paxos (EPaxos) [20], since it presents an alternative way of reducing the leader bottleneck –eliminating dedicated leaders altogether.

Figure 8 illustrates the latency and throughput performance of these three protocols in a 25-node cluster. Our clients are configured to communicate with Paxos and PigPaxos leader for all operations, and with a random node in EPaxos for each operation. Due to both the larger super-majority quorum system and the high conflict rate (with only a 1000 items picked at random), EPaxos performance suffers from conflict resolution phase draining the resources of every node. On the other hand, Paxos is limited by a single leader exchanging messages with all followers. Initially, Paxos has a lower request latency, but its throughput gets saturated quickly and reaches its limit of around 2000 requests per second. While PigPaxos pays the initial price of having 30% higher latency, we see that it scales to much higher throughput with very little latency deterioration after that.

PigPaxos provides great throughput improvements over traditional Paxos in wide area deployments as well. In Figure 9 we illustrate PigPaxos in 15 nodes configuration spread over Virginia, California and Oregon regions. Each region represents a separate PigPaxos relay group, with the leader node located in Virginia region. In this setup, the latency is dominated by cross-region distance, and as such the difference between Paxos and PigPaxos is not observable at low loads. Similarly to the local area experiments, PigPaxos maintains low latency for much higher levels of throughput.

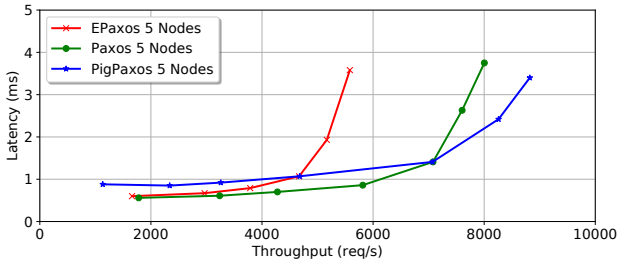


Figure 10: Latency and Throughput on 5-node cluster with 2 relay nodes.

5.5 Small Clusters

We find that the benefits from PigPaxos extend to small clusters as well. In Figure 10 we illustrate a 5 node cluster running Paxos and PigPaxos with 2 relay groups. The difference in latency between the two protocols is more pronounced in this setup, as Paxos can maintain its low-latency performance for longer. However, even in this cluster PigPaxos scales to higher throughput than Paxos since it communicates with fever nodes. PigPaxos talks to just two nodes, which is exactly how many followers Paxos needs to contact for majority quorum (including one self-vote), but Paxos still sends four messages in each round. Similar to our large cluster experiment, EPaxos exhibits too many conflicts in a workload with just 1000 items to scale well.

A slightly larger cluster of 9 nodes allows PigPaxos to use different number of relay groups as shown in Figure 11. As before, PigPaxos scales better than Paxos in both 2 and 3 relay groups configurations. However, with bigger cluster size, Paxos latency advantage over PigPaxos diminishes even at lower throughput levels.

5.6 Payload Size

Payload size may have an impact on the communication performance of the system. Large messages require both more resources for serialization and more network capacity for transmission. To study how different payload size impacts the

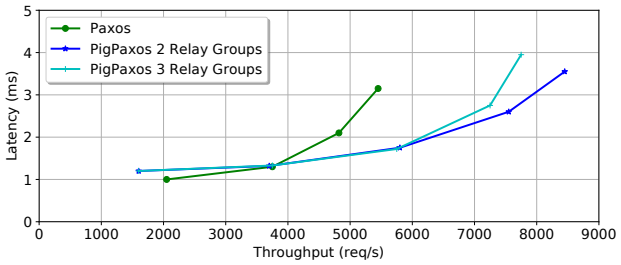
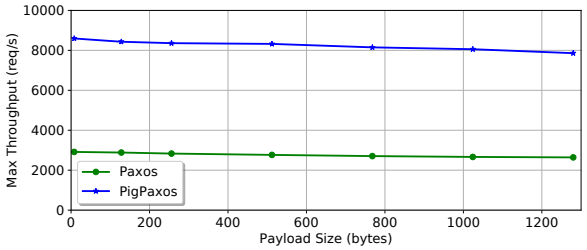
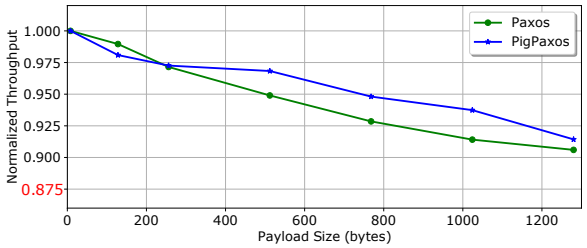


Figure 11: Latency and Throughput on 9-node cluster with 2 and 3 relay groups.



(a) Maximum throughput at various payload sizes.



(b) Normalized throughput at various payload sizes. Note that the throughput scale starts at 0.86 of maximum throughput, and that neither protocol dipped below 0.9 of its top performance.

Figure 12: Performance of PigPaxos at various payload sizes, varying from 8 to 1280 bytes.

performance of PigPaxos and Paxos, we experiment with 25 node clusters, where PigPaxos uses 3 relay groups. To gauge the performance and scalability with respect to payload size we measured the maximum throughput on each system under a write-only workload generated by 150 clients running on 3 VMs.

Figure 12a shows the maximum throughput of PigPaxos and Paxos at payload sizes varying from 8 to 1280 bytes. While both protocols show drastically different level of performance, they exhibit similar relative level of degradation as payload size increases. Figure 12b illustrates the throughput normalized to the maximum observed value.

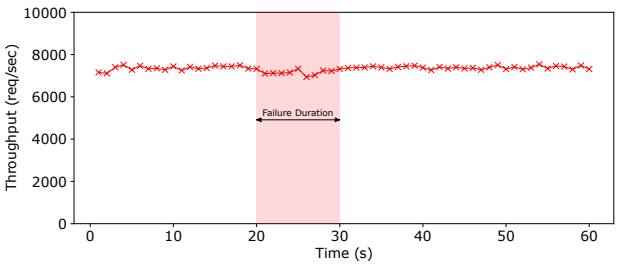


Figure 13: Maximum throughput on 25 node cluster with 3 relay groups under a single node failure. Throughput is sampled over 1-second intervals

5.7 Fault Tolerance

Since PigPaxos preserves the control and decision-making logic in Paxos, it is easy to see that PigPaxos can make progress as long as majority nodes are still up and running. However, it is also important to maintain adequate level of performance even under failures, and with the use of relay groups and relay nodes, it is worth investigating if this property is satisfied as well. Figure 13 illustrate how maximum performance of PigPaxos is impacted when one of the 3 relay groups in a 25 node cluster is faulty with one or more nodes not responding in the group. In this experiment we did not use any additional optimizations beyond the group timeout to illustrate the worst possible case of a fault relay group. The timeout was set to 50ms, more than 40 times over the normal operation latency.

A failure in a single relay group causes the relay group to timeout. However, the two other relay groups still constitute the majority and can respond to the leader quickly, resulting in a very little change in performance. We find that the maximum throughout over the duration of the fault declined by only 3%, mainly due to the leader having to wait a little longer. The little extra wait is because in some cases the faulty group would have been the first to reply if it did not have a failure, giving a slight boost to overall system performance.

6 Discussion

6.1 Number of Relay Groups and Leader Bottleneck

PigPaxos relies on the relay groups to deliver messages to every participating node. As we have observed in Figure 7, the number of relay groups in a deployment impacts the performance significantly. With just two relay groups, PigPaxos achieved its maximum throughput, significantly outperforming other configurations. To explain this result, we refer to the performance modeling ideas in Ailijiang et.al. [2] and adopt and extend that modeling idea to cover PigPaxos. In particular, we adopt the model to look at the number of ex-

changed messages M in every node of the PigPaxos protocol as a proxy for the load on the node. A leader needs to receive one message from the client and eventually send a reply, in addition, the leader communicates with r number of relay groups. A total number of messages M_l handled by the leader is then expressed as:

$$M_l = 2r + 2 \tag{1}$$

The message load on the followers depends on the role a follower is performing in the protocol for a given round – relay followers are responsible for vastly more messages than the regular follower nodes. As a relay, node needs to receive one incoming message from the leader, send one message back to the leader, and handle a round trip commination with every remaining follower in the relay group. As a non-relay follower, the node only processes one round trip communication with the relay node. On the other hand, the random alternation of the relay nodes also shields the relays from becoming hotspots themselves: the extra traffic load a relay node incurs in one round is offset in consecutive rounds when the node no longer serves as relay. Therefore, on average, we have the following message load on each follower:

$$M_f = 2(\frac{r}{N-1})(\frac{N-r-1}{r}) + 2 \tag{2}$$

$$= \frac{2(N-r-1)}{N-1} + 2 \tag{3}$$

where N is the total number of nodes in the system, and $\frac{r}{N-1}$ is the probability of a node being chosen as a relay for the round.

The number of message exchanges at the nodes draw a similar picture to our empirical results, indicating that the leader bottleneck grows as the number of relay groups increases. The reason for this is that the leader still handles more communication and data processing than the "average" follower node in the 25-node configuration, as we show in Table 1. We see that the overhead is smallest for the 2 relay groups configuration and grows significantly as the number of relay groups increases. The most significant observation here is that the 2 relay group configuration still creates a large leader bottleneck. We have confirmed this difference by measuring the CPU utilization, and found a growing difference in CPU utilization between leader and follower nodes as the number of relay groups increases. Our analytical reasoning matches the empirical results in Section 5.3.

6.2 PigPaxos in Small Clusters

PigPaxos scalability benefits apply not only to large clusters, but also to smaller deployments. As we illustrate in Figure 11, a 9 node PigPaxos cluster improves the throughput over classical Multi-Paxos by as much as 57% with very little degradation in latency. Using the analytical approach described earlier, we illustrate the relative message-induced bottleneck of Paxos and PigPaxos leaders in 9 nodes in Table 2.

# of Relay Groups (r)	Messages at Leader ($M-l$)	Messages at Follower (M_f)	Leader Overhead
2	6	3.83	56%
3	8	3.75	113%
4	10	3.67	172%
5	12	3.58	234%
6	14	3.50	300%
24 (Paxos)	50	2	2400%

Table 1: Message load at leader and followers for different number of relay groups in 25 node cluster.

# of Relay Groups (r)	Messages at Leader ($M-l$)	Messages at Follower (M_f)	Leader Overhead
2	6	3.5	71%
3	8	3.25	146%
4	10	3	233%
8 (Paxos)	18	2	800%

Table 2: Message load at leader and followers for different number of relay groups in 9 node cluster.

Moreover, the benefit of PigPaxos extends down to 7 and even 5 node clusters (Figure 10). In the latter case PigPaxos has very little flexibility for its configuration, as going down to just a single relay group presents a performance problem under even a single failure. Two relay group configuration, however, can withstand the single node crash without a dramatic impact on latency or throughput, since a leader and another functioning relay group deliver the majority quorum.

6.3 Number of Relay Layers

While we limited our discussion to a single layer of relay nodes, in the most generic form, PigPaxos extends to multiple layers of relay nodes. Multi-layered PigPaxos may allow lower layer relays to offload some of the communication burden from the upper layers. However, this only makes sense if the there is no bottleneck on the upper levels of the communication flow. As we have shown for 25 node cluster, the leader remains the bottleneck even with just two relay nodes. Under such conditions, offloading the communication burden from the followers will not result in any significant improvement, since the bottleneck remains unchanged.

Moreover, we find that even in the most extreme configurations with arbitrary number of nodes, a leader remains a bottleneck. The leader's message load M_l is a linear function of the number of relay groups r, as show in formula 1. In order to minimize the leader message load we need to minimize the number of relay groups, which cannot be smaller than 1. As a result, the smallest message load on the leader is $M_l = 4$. At the same time, with $r = 1$ and $N \to \infty$, the load on the followers $\frac{2(N-r-1)}{N-1} + 2$ asymptotically increases to 4, as $\lim_{N\to\infty} \frac{2(N-2)}{N-1} + 2 = 4$.

Even in the most extreme cases, the average/amortized load on follower nodes only matches that of the leader, indicating that the leader still remains a bottleneck regardless of the cluster size and the number of relay nodes. Further adding to

the leader's load is heavier message processing, because the leader needs to tally up the votes and decide if the quorum has been met. Since the bottleneck cannot be shifted entirely to the followers with any N, adding more layers to offload the communication work from the followers will not result in a performance gain. However, our simple model does not consider other variables that may play larger role for very big clusters or very big messages. For example, in a large cluster, the cost of maintaining many connections may start to add up and affect relays, leaving the possibility for multi-layer communication trees to be beneficial at very large scale.

6.4 Bandwidth Across Regions in WAN

The PigPaxos communication model enables a more efficient bandwidth utilization for achieving wide area network (WAN) consensus on a large cluster. This efficiency comes from the ability to assign all nodes in the region to a single relay group, and making PigPaxos leader send only one message per region across WAN. With many cloud providers charging for WAN traffic, a PigPaxos system can provide significant savings to applications requiring geo-redundancy and strong consistency. For example, a 3 region deployment with 3 nodes in each region will have only 2 messages going across WAN for each write operation. Paxos-backed system, in contrast, will send 6 separate messages, representing 3 times the WAN traffic.

7 Concluding Remarks

To address the communication bottlenecks at single-leader Paxos protocols, we presented PigPaxos. PigPaxos manages to reduce the leader's communication bottleneck by decoupling the decision-making at the leader from the communication at the leader. Although aggregation approaches have been known and employed in the context of weak-consistency replication protocols, PigPaxos shows how they can be effectively applied to and integrated with the strong consistency distributed consensus protocols.

Our evaluations show that PigPaxos can provide more than 3 folds improvement in throughput over Paxos and EPaxos with little latency deterioration. Our experiments also show that the aggregation used in PigPaxos has negligible overhead for the latency as compared to the latency of Paxos. The communication piggybacking and aggregation technique used in PigPaxos is readily applicable to many Paxos implementations [22, 23] and variants [3, 18], and we conjecture that PigPaxos would be useful for implementing geo-replicated distributed databases with tens of replicas distributed over many regions around the globe.

An immediate future work is to design and evaluate further optimization opportunities in the communication tree of PigPaxos, and to scale the protocol to hundreds of nodes. This is likely to open many new challenges. When scaled to two-orders of magnitude, it can be argued that the transformation is not incremental/evolutionary and would necessitate new supplementary approaches and services for coping with this scale. For example, in the failure detector module in Paxos, the threshold for a node to become a leader candidate should be modified to be conservatively high, because contesting the leader is very costly at the scale of hundred participants. Therefore, instead of static setting of failure detector thresholds, new approaches are needed to set the thresholds to be proportional to the size of N, such that only a couple nodes at a given time period will step up to become a leader candidate.

Another future work is to apply and integrate our approach for achieving scalability to modern PBFT [9] descendant byzantine fault-tolerant blockchain protocols, such as Tendermint [7], Casper [8], and LibraBFT [6, 27].

Acknowledgments

This project is in part sponsored by the National Science Foundation (NSF) under award number CNS-1527629 and XPS-1533870.

References

[1] M. K. Aguilera and D. B. Terry. The many faces of consistency. *IEEE Data Eng. Bull.*, 39(1):3–13, 2016.

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

[3] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar. Wpaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):211–223, 2019.

[4] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, and et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[5] C.-S. Barcelona. Mencius: building efficient replicated state machines for wans. *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, 2008.

[6] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino. State machine replication in the libra blockchain, 2019.

[7] E. Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[8] V. Buterin and V. Griffith. Casper the friendly finality gadget. 2017.

[9] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association.

[10] A. Charapko, A. Ailijiang, and M. Demirbas. Adapting to access locality via live data migration in globally distributed datastores. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3321–3330. IEEE, 2018.

[11] A. Charapko, A. Ailijiang, and M. Demirbas. Linearizable quorum reads in paxos. In *11th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[12] A. Charapko, A. Ailijiang, M. Demirbas, and S. Kulkarni. Retroscope: Retrospective monitoring of distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 30(11):2582–2594, Nov 2019.

[13] Cockroach Labs. Cockroachdb: The sql database for global cloud services. https://www.cockroachlabs.com/docs/stable/architecture/overview.html, 2018.

[14] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. *Proceedings of OSDI*, 2012.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1982.

[16] J. N. Gray. Notes on data base operating systems. pages 393–481, 1978.

[17] M. Herlihy, B. Liskov, and L. Shrira. Cross-chain deals and adversarial commerce. *Proceedings of the VLDB Endowment*, 13(2):100–113, 2019.

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

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

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

[21] F. Nawab, D. Agrawal, and A. 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. ACM, 2018.

[22] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[23] Red Hat. etcd. a distributed, reliable key-value store for the most critical data of a distributed system. https://coreos.com/etcd/, 2019.

[24] S. Rizvi, B. Wong, and S. Keshav. Canopus: A scalable and massively parallel consensus protocol. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 426–438, New York, NY, USA, 2017. Association for Computing Machinery.

[25] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 328–343, New York, NY, USA, 2015. Association for Computing Machinery.

[26] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42, 2015.

[27] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery.