

Multileader WAN Paxos: Ruling the Archipelago with Fast Consensus

Ailidani Ailijiang Aleksey Charapko Murat Demirbas Tevfik Kosar
Computer Science and Engineering
University at Buffalo, SUNY

Abstract

We present *WPaxos*, a multileader wide area network (WAN) Paxos protocol, that achieves low-latency high-throughput consensus across WAN deployments. *WPaxos* dynamically partitions the global object-space across multiple concurrent leaders that are deployed strategically using flexible quorums. This partitioning and emphasis on local operations allow our protocol to significantly outperform leaderless approaches, such as *EPaxos*, while maintaining the same consistency guarantees. Unlike statically partitioned multiple Paxos deployments, *WPaxos* adapts dynamically to the changing access locality through adaptive object stealing. The ability to quickly react to changing access locality not only speeds up the protocol, but also enables support for mini-transactions.

We implemented *WPaxos* and evaluated it across WAN deployments using the benchmarks introduced in the *EPaxos* work. Our results show that *WPaxos* achieves up to 10 times faster average request completion than *EPaxos* due to the reduction in WAN communication.

1 Introduction

Paxos, introduced in 1989 [15], provides a formally-proven solution to the fault-tolerant distributed consensus problem. Notably Paxos preserves the safety specification of distributed consensus (i.e., no two nodes decide differently) to the face of concurrent and asynchronous execution of the nodes, crash/recovery of the nodes, and arbitrary message loss. When the conditions improve so that distributed consensus becomes solvable, Paxos also satisfies the progress property (i.e., nodes decide on suitable decision value as a function of the inputs).

Paxos algorithm and its variants have been deployed widely, including for Google Chubby [5] (Paxos [23]), Apache ZooKeeper [11] (Zab [13]), and recently for etcd [7] (Raft [21]). All of these implementations depend on a centralized primary process (a.k.a., leader) to serialize all operations and updates. During normal operation, only one server acts as the leader, all client requests are forwarded to that leader, and that leader commits the requests by performing the second phase of Paxos with the acceptors. Due to this dependence on a single centralized leader, these Paxos implementations only support deployments in local area and cannot deal with write-intensive scenarios across wide area networks (WANs) well. In recent years, however, coordination over wide-area (across zones, such as clusters, sites and datacenters) has gained greater importance. WAN coordination has become essential for database applications and NewSQL datastores [3, 6], distributed filesystems [8, 19, 22], and social network metadata updates [4, 17].

In order to eliminate the single leader bottleneck, *EPaxos* [20] proposes a leaderless Paxos protocol where any replica at any zone can propose and commit commands opportunistically provided the commands are non-interfering. This opportunistic commit protocol requires an agreement from a fast-quorum of roughly $3/4$ ths of the acceptors¹, which means that WAN latencies are still incurred. Moreover, if the commands proposed by multiple concurrent opportunistic leaders do interfere, the protocol requires performing a second phase to record the acquired dependencies requiring agreement from a majority of the Paxos acceptors.

Another way to eliminate the single leader bottleneck is to use a separate Paxos group deployed at each zone. Systems like Google Spanner [6], ZooNet [16], Bizur [9] achieve this via a static partitioning of the global object-space to different zones, each responsible for a shard

⁰An archipelago is a chain, cluster or collection of islands. <https://en.wikipedia.org/wiki/Archipelago>

¹For $2F + 1$ cluster, fast-quorum is $F + \lfloor \frac{F+1}{2} \rfloor$

of the object-space. However, such static partitioning is inflexible and WAN latencies will be incurred persistently to access/update an object mapped to a different zone. Moreover, in order to perform transactions involving objects in different zones, a separate mechanism (such a two-phase commit) would need to be implemented across the corresponding Paxos groups.

Contributions. We present *WPaxos*, a multileader WAN Paxos protocol, that achieves low-latency high-throughput consensus across a WAN deployment.

To achieve communication-efficient WAN coordination, *WPaxos* adapts the “flexible quorums” idea (which was introduced in 2016 summer as part of *FPaxos* [10]). *WPaxos* uses the flexible quorums rule in a novel manner for deploying *multiple concurrent leaders* across the WAN strategically. The commit decisions for updates are fast as *WPaxos* appoints the phase-2 acceptors to be at the same zone as the leader. We present how this is achieved in Section 2.1.

Unlike the *FPaxos* protocol which uses a single-leader and do not scale to WAN distances, *WPaxos* uses multileaders and partitions the object-space among the multiple leaders. On the other hand, *WPaxos* differs from the existing static partitioned multiple Paxos deployment solutions, because it implements a dynamic partitioning scheme: The concurrent leaders *steal* objects from each other using phase-1 of Paxos. This object-stealing mechanism also enables transactions across leaders (such as a consistent read of multiple objects in different partitions) naturally within the Paxos updates, obviating the need for a separate two phase commit protocol across zone-leaders. We describe the *WPaxos* protocol in Section 2.2, Section 2.5, and present the algorithm in Section 3.

With its multileader protocol, *WPaxos* achieves the same consistency guarantees as in *EPaxos*: linearizability is ensured per object, and serializability and causal-consistency are ensured across objects. To quantify the performance benefits from *WPaxos*, we implemented *WPaxos*² and performed evaluations across WAN deployments using the evaluation benchmarks introduced in *EPaxos* [20]. Our results in Section 4 show that *WPaxos* significantly outperforms *EPaxos*, achieving up to 5 times faster average request commit than *EPaxos*. This is because, while the *EPaxos* opportunistic commit protocol requires about 3/4ths of the Paxos acceptors to agree and incurs almost one WAN round-trip latency, *WPaxos* is able to achieve zone-local-latency Paxos commits using the zone-local phase-2 acceptors.

While achieving low-latency and high-throughput, *WPaxos* also achieves incessant high-availability by having multileaders: failure of a leader is handled gracefully as other leaders can serve the requests previously

processed by that leader via the object stealing mechanism. Since leader re-elections are handled through the Paxos protocol, safety is always upheld to the face of node failure/recovery, message loss, and asynchronous concurrent execution. We discuss fault-tolerance properties of *WPaxos* in Section 5. Finally, while *WPaxos* helps most for slashing WAN latencies, it is also possible to deploy *WPaxos* entirely inside the same datacenter across clusters for its high-availability and throughput benefits. *WPaxos* provides throughput benefits by load-balanced parallel deployment of coordinating multileaders across the object space.

2 WPaxos

In this section we present a high level overview of *WPaxos*, and relegate a detailed explanation of the protocol to Section 3. Table 1 summarizes some common terminology used throughout the rest of the paper.

2.1 WPaxos Quorums

WPaxos relies on flexible quorums [10]. This surprising result showed we can weaken Paxos’s assertion that “all quorums should intersect” to instead “only quorums from different phases should intersect”. That is, majority quorums are not necessary for Paxos, provided that phase-1 quorums ($Q1$ s) intersect with phase-2 quorums ($Q2$ s). Flexible Paxos allows trading off $Q1$ and $Q2$ sizes to improve performance. Assuming failures and resulting leader changes are rare, phase-2 (where the leader tells the acceptors to decide values) is run more often than phase-1 (where a new leader is elected). 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.

WPaxos adopts the flexible quorum idea to WAN deployments for the first time. Our quorum concept derives from the grid quorum layout, shown in Figure 1a, in which rows and columns act as $Q1$ and $Q2$ quorums respectively. An attractive property of this grid quorum arrangement is $Q1 + Q2$ does not need to be greater than N , the total number of acceptors, in order to guarantee intersection of any $Q1$ and $Q2$. Since $Q1$ s are chosen from rows and $Q2$ s are chosen from columns, any $Q1$ and $Q2$ are guaranteed to intersect even when $Q1 + Q2 < N$.

In *WPaxos* quorums, each column represents a zone and acts as a unit of geographical partitioning. The collection of all columns/zones form a grid. In this setup, $Q1$ quorums span across all the zones, while $Q2$ s remain bound to a column, making phase-2 of the protocol operate locally without a need for WAN message exchange. We also relax some of the grid quorum constraints for $Q1$ to get a more fault-tolerant and efficient alternative. Our

²Our implementation will be made available as an opensource project on <https://github.com/ailidani/paxi>

Table 1: Terminology used in this work

Term	Meaning
Zone	Geographical isolation unit, such as datacenter or a region
Node	Maintainer of consensus state, combination of proposer and acceptor roles
Leader	Sequencer of proposals. Maintains a subset of all objects
Ballot	Round of consensus, combination of counter and zone ID and node ID
Slot	Uniquely identifies a sequence of instances proposed by a leader
Phase-1	Prepare phase, protocol to establish a new ballot/leader
Phase-2	Accept phase, normal case

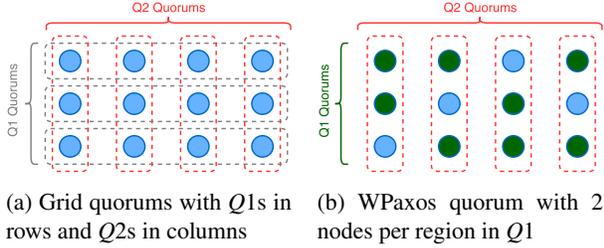


Figure 1: Grid and WPaxos quorums. (a) Regular grid quorum. (b) WPaxos quorum with one possible $Q1$ of 2 nodes per region.

quorums no longer use rigid grid rows for selecting $Q1$ s, instead we pick nodes from each column, no matter their row position.

Figure 1b shows the WPaxos flexible grid deployment used in this paper. (As we discuss in Section 5, it is possible to use alternative deployments for more improved fault-tolerance.) In this deployment each zone has 3 nodes, and each $Q2$ quorum is the 2 of the 3 nodes in a zone. The $Q1$ quorum, consists of 2 flexible rows across zones, that is, it includes any 2 nodes from each zone. Using a 2 row $Q1$ rather than 1 row $Q1$ has negligible effect on the performance, as we show in the evaluation. On the other hand, using a 2 row $Q1$ allows us to better handle node failures within a zone, because the 2-node $Q2$ quorum will intersect the $Q1$ even in the presence of a single node failure. Additionally, this allows for $Q2$ performance improvement, as a single straggler will not penalize the phase-2 progress.

2.2 WPaxos Protocol Overview

WPaxos is a multi-leader protocol flexible quorum, which contrasts with single-leader approach taken in FPaxos. Every node in WPaxos acts as a leader for a subset of all objects in the system. This allows the protocol to process requests for objects under different leaders concurrently. Each leader maintains a single log for all of its objects, making the objects linearizable with respect to each other for the time the objects remain under

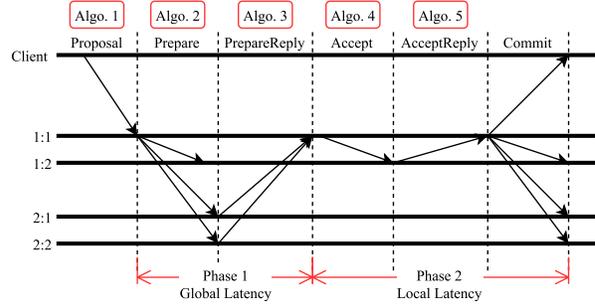


Figure 2: Normal case messaging flow of WPaxos

the same leadership. Unlike a single shared log, objects get their own ballot numbers that do not depend on each other.

Basic WPaxos protocol is broken down into two distinct phases, and each phase operates on a separate quorum. Phase-1 of the protocol, or the object-stealing phase is responsible for moving the ownership of the object between different leaders, while phase-2 replicates the object requests on some $Q2$. Phase-2 can execute multiple times until some other node steals the object.

The phase-1 of the protocol starts if a client has a request for a brand new object that is not in the system or the node needs to steal an object from a remote leader. This phase of the algorithm causes the ballot number to grow for the object involved. It is very similar to regular Paxos phase-1, however, WPaxos performs it on some global $Q1$ quorum. Successful completion of phase-1 transitions the protocol into phase-2, which in turn will run on a local $Q2$ quorum. This stage is used to decide the operations on the particular object. WPaxos repeats phase-2 multiple times, incrementing the slot number on each iteration.

Figure 2 shows the normal operation of both phases, and also references each operation to the algorithms in Section 3.

2.3 Immediate Object Stealing

WPaxos dynamically partitions objects across leaders in various zones, creating use cases when a client needs an object that belongs to a different zone. Our protocol makes this remote operation transparent for the client, allowing the client contact any local node with a remote request instead of reaching out across zones. The node, however, needs to deal with such request in a special manner, because it cannot process the request locally: it needs to steal the object from the current leader in order to carry out the request. Node consults its internal cache to determine the last ballot number used for the object and starts the WPaxos phase-1 on some $Q1$ quorum with a larger ballot. Object stealing will be successful if the local node is able to out-ballot the existing leader. Most of the times this is achieved in just one phase-1 attempt, provided that the local cache is current and the remote leader is not engaged in another phase-1. Once the object is stolen, the old leader will not be able to act on it, since the object is now associated with a higher ballot number than the ballot it had at the old leader. This is true even when the old leader was not in the $Q1$ when the key was stolen, because the intersected node in $Q2$ will reject any object operations attempted with the old ballot. Object stealing procedure may occur when some commands for the objects are still in progress, therefore, a new leader must recover any accepted, but not yet committed commands for the object.

WPaxos maintains separate ballot numbers for all objects, making sure that object stealing is not negatively affecting other objects. Our original design kept a single ballot number for all objects maintained by the leader, thus stealing the object required a node to out-ballot all objects of a remote leader. This created a leader dueling problem in which two nodes try to steal objects from each other by constantly proposing with higher ballot than the opponent, as shown in figure 3a.

Separate ballot numbers for different objects allow us to reduce ballot contention, although it can still happen when two leaders are trying to take over the same object currently owned by a third leader. To finally mitigate the issue we have placed two additional safeguards: resolving ballot conflict by zone ID and node ID in case the ballot counters are the same (figure 3b), and implementing a random back-off mechanism in case a new dueling iteration starts anyway. The overheads of maintaining per-object ballots are negligible and far outweigh the performance penalty incurred by having per-leader ballots. For instance, one million objects would only require 16 Mb of memory to store ballots: 8 Mb for a 64-bit key and 8 Mb more for actual ballots.

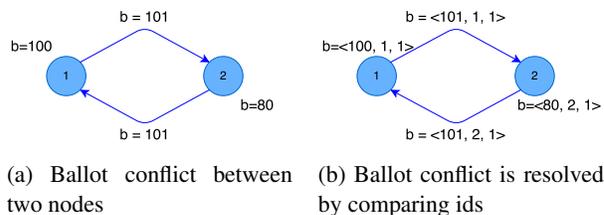


Figure 3: Two nodes compete on the ballot number: (a) prepare with the same ballot number, causing phase-1 to restart for both; (b) ballots are ordered by zone ID and node ID when counters are the same, one node wins.

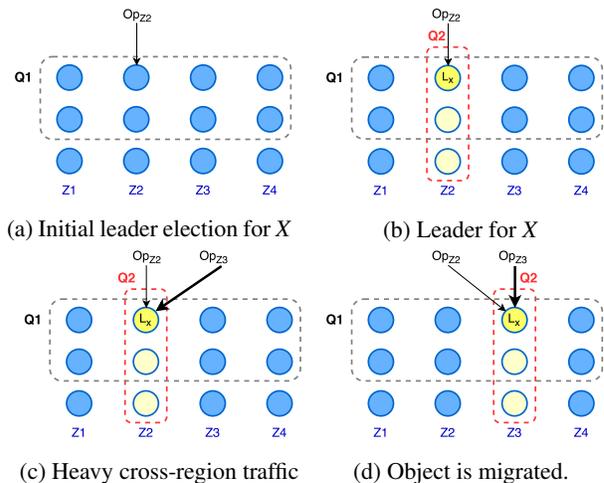


Figure 4: Leader election and adaptive object stealing: (a) WPaxos starts the operation with no prior leader for the object X when operation Op_{z2} is issued in $Z2$; (b) initial leader is elected in the zone of the first request; (c) heavy traffic Op_{z3} from $Z3$ must do WAN communication; (d) object X is stolen to $Z3$.

2.4 Locality Adaptive Object Stealing

The basic protocol migrates the object from a remote region to a local region upon the first request. Unfortunately, immediate approach may cause a performance degradation once the object is frequently needed in more than one zone and incurs WAN latency penalty of traveling back-and-forth between zones.

With locality adaptive object stealing we can delay or deny the object transfer to a zone issuing the request based on WPaxos object migration policy. The intuition behind this approach is to move objects to a zone whose clients will benefit the most from not having to communicate over WAN, while allowing clients from less frequent zones to send their requests over WAN to the remote leaders. In this adaptive mode clients still communicate with the local nodes, however the nodes may not steal the objects right away, instead choose to forward

the requests to the remote leaders.

Our *majority-zone* migration policy aims to improve the locality of reference by transferring the objects to zones sending out the highest number of requests for the objects, as shown in Figure 4. Since the current object leader handles all the requests, it has the information about which clients access the object more frequently. If the leader L_o detects that the object X has more requests coming from a remote zone, it will initiate the object handover by communicating with the node L_n , and in its turn L_n will start the phase-1 protocol to steal the leadership of the object.

2.5 Mitransactions

Linearizability guarantees provided to the objects under the same leader combined with the ability for a node to steal the objects enable WPaxos to support Sinfonia-style mitransactions [1]. Mitransactions are achieved by moving all objects involved in the transaction to a single leader, before processing the transaction.

This transaction process starts with a client sending a mitransaction request to a node it already believes to have the most of the required objects. In its turn, the leader will steal the missing objects and once all objects are collected, the mitransaction can proceed as a single command. This approach, however, is not without a penalty for overall system performance, since stealing objects disrupts the locality balancing.

3 Algorithm

Each WPaxos node is a deterministic state machine that maintains a set of variables and an internal datastore. We assume a set of nodes communicating through message passing in an asynchronous environment. The protocol updates the states of its variables when processing the incoming messages, and eventually commits and executes a sequence of commands Σ against the datastore. For every leader there is an unbounded sequence of **instances** in Σ , identified by an increasing slot number s . At most one command will be decided in any instance. We assume the commands are defined to access one or multiple objects. In this section, we present the basic algorithm that each command γ only access one object, identified by $\gamma.o$. Every node α leads its own set of objects \mathcal{O}_α and provides linearizability for all objects in that set. Nodes also replicates a complete state of all other leaders in the system. Each node α keeps an index \mathbf{I} , mapping each object o to the leader of o known to α , i.e. $\mathbf{I}[o] = \lambda$. When a node tries to acquire the leadership of a new object, it adds the object and all following corresponding requests into the set Π until the phase-1 of protocol completes. Nodes also maintain a set of ballot numbers \mathbf{b} for

all objects, and keep a history \mathbf{H} of all accesses for objects to be used for the locality-adaptive object-stealing. A summary of WPaxos notation is given as follows:

λ, α, β	Nodes
	λ usually represent command leader
κ	Client
γ, δ	Commands
\mathbf{b}	Set of ballot numbers
\mathbf{s}	Set of slot numbers
Π	Set of phase-1 requests
Σ	Sequence of instances
$\mathbf{I}[o]$	Index, mapping objects to leaders
\mathcal{O}_λ	Set of objects led by λ
\mathbf{H}	Access history
\bullet	Concatenation operator
$\langle \mathbf{type}, \gamma, \mathbf{b}[o], s \rangle$	General message format.

Algorithms 1-5 show the operations of a WPaxos node. Phase-1 of the protocol is described in the algorithms 1-3, while algorithms 4 and 5 cover phase-2.

3.1 Initialization

Node α Initialization

1: function INIT(\mathcal{O})	
2: $\mathbf{b} \leftarrow \emptyset$	\triangleright No known ballot numbers
3: $\Pi \leftarrow \emptyset$	\triangleright Phase-1 requests initially empty
4: $\forall \beta \in \text{peers} : \mathbf{s}[\beta] \leftarrow -1$	\triangleright Initial slot numbers
5: $\forall \beta \in \text{peers} : \Sigma[\beta] \leftarrow \emptyset$	\triangleright Phase-2 instances initially empty
6: $\mathbf{I} \leftarrow \emptyset$	\triangleright No known index
7: $\mathbf{H} \leftarrow \emptyset$	\triangleright Empty access history
8: $\forall o \in \mathcal{O} : \mathbf{I}[o] \leftarrow \alpha$	\triangleright Initial index
9: $\forall o \in \mathcal{O} : \mathbf{b}[o] \leftarrow \langle 1 \bullet \alpha \rangle$	\triangleright Initial Ballot

The INIT(\mathcal{O}) function describes the state initialization of any node before it becomes active. We assume no prior knowledge of ballots or the locations of objects. However, WPaxos makes the initial object assignment optional, a user may provide the set of starting objects, allowing the initialization routine to construct ballots and indices (line 8-9).

3.2 Phase-1: Prepare

Algorithm 1 describes the initial stage of processing every new request received by the node from the client. WPaxos protocol starts with the client κ sending a $\langle \mathbf{request}, \kappa, \gamma \rangle$ message to one of the nodes in the system. Client typically chooses a local zone node to minimize the initial communication costs. The **request** message includes the command γ , containing some object $\gamma.o$ on which the command needs to be executed. Upon receiving the command from the client, a node α checks if the

Algorithm 1 Node α : client request handler

```

1: function RECEIVE( $\langle$ request,  $\gamma$  $\rangle$  from  $\kappa$ 
2:    $o \leftarrow \gamma.o$   $\triangleright$  The object in command  $\gamma$ 
3:   if  $o \notin \mathbf{I}$  then  $\triangleright$  Unknown object
4:     STARTPHASE-1( $\gamma$ )  $\triangleright$  Phase 1
5:     return
6:    $\lambda \leftarrow \mathbf{I}[o]$ 
7:   if  $\alpha = \lambda$  then  $\triangleright$   $\alpha$  is leader of  $o$ 
8:     if  $o \in \Pi$  then  $\triangleright$  Request for  $o$  exists
9:        $\Pi[o] \leftarrow \Pi[o] \cup \{\gamma\}$   $\triangleright$  Append to current phase-1
10:    else  $\triangleright$   $\alpha$  is the current leader of  $o$ 
11:      STARTPHASE-2( $\gamma$ )  $\triangleright$  Phase 2
12:     $\mathbf{H} \leftarrow \mathbf{H} \cup \{o, \kappa\}$   $\triangleright$  Save to access history  $\mathbf{H}$ 
13:    if  $\mathbf{H}$  triggers migration event then
14:      SENDTO( $\beta$ ,  $\langle$ migrate,  $\gamma.o$  $\rangle$ )
15:    else  $\triangleright$   $o$  is owned by other node
16:      if Immediate object stealing then
17:         $\mathbf{b}[o] \leftarrow \mathbf{b}[o] + 1$   $\triangleright$  Steal with new ballot
18:        STARTPHASE-1( $\gamma$ )
19:      else  $\triangleright$  Adaptive object stealing
20:        SENDTO( $\lambda$ ,  $\langle$ request,  $\kappa, \gamma$  $\rangle$ )  $\triangleright$  Forward to node  $\lambda$ 

21: function STARTPHASE-1( $\gamma$ )
22:    $o \leftarrow \gamma.o$ 
23:    $\mathbf{I}[o] \leftarrow \alpha$ 
24:   if  $o \in \Pi$  then
25:      $\Pi[o] \leftarrow \Pi[o] \cup \{\gamma\}$ 
26:     return
27:    $\Pi[o] \leftarrow \text{NEWQUORUM}(Q1)$   $\triangleright$  Waiting quorum of phase 1
28:   BROADCAST( $\langle$ prepare,  $o, \mathbf{b}[o]$  $\rangle$ )  $\triangleright$  Start phase 1

29: function STARTPHASE-2( $\gamma$ )
30:    $o \leftarrow \gamma.o$ 
31:    $s_\alpha \leftarrow s_\alpha + 1$   $\triangleright$  Next available slot
32:    $\Sigma[\alpha][s_\alpha] \leftarrow \langle$ instance,  $\gamma, \mathbf{b}[o], \text{NEWQUORUM}(Q2)$  $\rangle$ 
 $\triangleright$  Create new instance
33:   MULTICAST( $\langle$ accept,  $\gamma, \mathbf{b}[o], s[\alpha]$  $\rangle$ )  $\triangleright$  Start phase 2

```

object exists in the index \mathbf{I} , and starts phase-1 for any missing objects by invoking STARTPHASE-1 procedure (lines 3-5). If the object is known to belong to the node, then it initiates phase-2 of the protocol in STARTPHASE-2 function which sends a message to its $Q2$ quorum, and creates a new instance for slot s_α (lines 30-33). However, if the object is found to be managed by some other remote leader λ , depending on the configuration, α will either forward the request to λ (line 20), or start immediate object stealing with larger ballot in phase-1 (lines 16-18).

Our protocol keeps track of the past accesses in order to facilitate the locality adaptive leader stealing. The leader keeps track of every object's access history (line 12) to determine the best possible position for the object. Current object leader may decide to relinquish its object ownership based on the locality adaptive object stealing rule in place. In that case, the leader sends out a **migrate** message to the node it determined to be more suitable to lead the object (line 13-14).

The HANDLE routine of algorithm 2 processes the in-

Algorithm 2 Node α : prepare message handler

```

1: function HANDLE( $\langle$ prepare,  $o, b$  $\rangle$ ) from  $\beta$ 
2:    $\lambda \leftarrow \mathbf{I}[o]$ 
3:    $s' \leftarrow \max(\{i : o \in \Sigma[\lambda][i]\})$   $\triangleright$  Get the largest slot for  $o$ 
4:    $\delta \leftarrow \Sigma[\lambda][s']$   $\triangleright$  Get instance command
5:   if  $\beta = \lambda$  then  $\triangleright$  Old leader
6:      $\mathbf{b}[o] \leftarrow \max(\mathbf{b}[o], b)$   $\triangleright$  Update ballot of object  $o$ 
7:     SENDTO( $\beta$ ,  $\langle$ prepareReply,  $o, ok \leftarrow true, \mathbf{b}[o], (s', \delta)$  $\rangle$ )
8:   else  $\triangleright$  New leader
9:     if  $b > \mathbf{b}[o]$  then  $\triangleright$  Accept only if higher ballot
10:      if  $o \in \Pi$  then  $\triangleright$  If there is outstanding phase 1
11:         $\forall r \in \Pi[o] : \text{Retry request } r \text{ after random time}$ 
12:         $\Pi \leftarrow \Pi \setminus o$ 
13:       $\mathbf{b}[o] \leftarrow b$ 
14:       $\mathbf{I}[o] \leftarrow \beta$ 
15:      SENDTO( $\beta$ ,  $\langle$ prepareReply,  $o, ok \leftarrow$ 
 $true, \mathbf{b}[o], (s', \delta)$  $\rangle$ )
16:    else  $\triangleright$  Reject
17:      SENDTO( $\beta$ ,  $\langle$ prepareReply,  $o, ok \leftarrow false, (\lambda, \mathbf{b}[o])$  $\rangle$ )

```

coming prepare message sent during phase-1 initiation. The node α can accept the sender node β as the leader for object o in one of two cases: there is no leader change and the sender is the same node as kept in the local index (lines 5-7) or sender's ballot number b of o is greater than the ballot number α is aware of (lines 9-15). In the second case, node α also checks if it currently involves in phase-1 for the same object (line 10). It cancels and schedule retries of those pending requests for a random back-off time (lines 11-12). Node α replies the accepted leader with the largest slot number and its accepted command, such that any unresolved commands can be recovered by the new leader (lines 3,4,7,15). Otherwise, the node rejects β as the leader λ for object o , and sends back the ballot number that has caused the rejection (line 17).

Algorithm 3 Node α : prepareReply message handler

```

1: function HANDLE( $\langle$ prepareReply,  $o, ok, (s, \delta), (\lambda, b)$  $\rangle$ ) from  $\beta$ 
2:   if  $o \notin \Pi \vee b < \mathbf{b}[o]$  then
3:     return  $\triangleright$  Ignore old reply msg
4:   if  $ok$  then  $\triangleright$  Acked
5:      $\Pi[o].Q1.ACK(\beta)$ 
6:      $\Pi[o] \leftarrow \Pi[o] \cup (s, \delta)$ 
7:     if  $Q1.SATISFIED$  then
8:       Recover  $\text{MAX}((s', \delta') \in \Pi[o])$ 
9:       HANDLE( $\{r : \forall r \in \Pi[o]\}$ )
 $\triangleright$  Process all pending requests
10:     $\Pi \leftarrow \Pi \setminus o$ 
11:   else  $\triangleright$  Handle reject message
12:      $\mathbf{I}[o] \leftarrow \lambda$   $\triangleright$  Update index
13:      $\mathbf{b}[o] \leftarrow \max(\mathbf{b}[o], b)$   $\triangleright$  Update ballot
14:     Retry  $\{r : \forall r \in \Pi[o]\}$  after random time
15:      $\Pi \leftarrow \Pi \setminus o$ 

```

Algorithm 3's HANDLE function collects the prepare replies sent by the algorithm 2 (lines 4-6), and checks if the $Q1$ quorum is satisfied, at which point the new leader select the largest slot to depend on, and recover

any uncommitted slots with suggested commands (lines 7-8), then start accept phase for the pending requests that have accumulated in Π (line 9). Finally, the object is removed from the phase-1 outstanding set Π (line 10). If the phase-1 is rejected, the local caches for remote object and index are updated with new information (lines 11-13), and the pending requests in such phase-1 are retried by scheduling a random back-off time to push them back to the main request queue (line 14).

3.3 Phase-2: Accept

Phase-2 of the protocol starts after the completion of phase-1 or when it is determined that no phase-1 is required for a given object. The accept phase can be repeated many times until some remote leader steals the object. WPaxos carries out this phase on a $Q2$ quorum residing in a single zone, thus all inter-node communications are kept local to the zone, greatly reducing the latency.

Algorithm 4 Node α : accept message handler

```

1: function HANDLE( $\langle$ accept,  $\gamma, b, s$  $\rangle$ ) from  $\beta$ 
2:   if  $\gamma.o \notin \mathbf{I}$  then
3:      $\mathbf{I}[\gamma.o] \leftarrow \beta$ 
4:      $\lambda \leftarrow \mathbf{I}[\gamma.o]$ 
5:     if  $\beta = \lambda \vee b \geq \mathbf{b}[o]$  then ▷ Known leader or new ballot
6:        $\mathbf{I}[o] \leftarrow \beta$ 
7:        $\mathbf{b}[o] \leftarrow \max(\mathbf{b}[o], b)$ 
8:       if  $\Sigma[\lambda][s] = \perp$  then  $\Sigma[\lambda][s] \leftarrow \langle$ instance,  $\gamma, b$  $\rangle$ 
▷ Create instance if not exists
9:       if  $b \geq \Sigma[\lambda][s].b$  then
10:        SENDTO( $\lambda, \langle$ acceptReply,  $ok \leftarrow true, o, \lambda, b, s, \rangle$ )
11:       else
12:        SENDTO( $\lambda, \langle$ acceptReply,  $ok$  ←
false,  $o, \lambda, \Sigma[\lambda][s].b, s, \rangle$ )
13:       else ▷ Old ballot
14:        SENDTO( $\lambda, \langle$ acceptReply,  $ok \leftarrow false, o, \lambda, \mathbf{b}[o], s$  $\rangle$ )

```

Once the leader sends out the accept message at the beginning of the phase-2, acceptors must properly respond to this message. Algorithm 4 shows how acceptors handle the \langle accept \rangle message. In the normal case, node will accept the message if it is a known leader with the same or higher ballot number (lines 5-10). However, if there exists a different leader or the proposed ballot number is smaller than the instance of the same slot s , node will reject with existing ballot from the instance (lines 11-14).

Leader collects the replies from its $Q2$ acceptors in Algorithm 5. The request proposal either gets committed when a sufficient number of successful replies are received (lines 6-8), or aborted if some acceptors reject the proposal (lines 9-15). In case of rejection, leader also updates its cache with new object and index information it has received from the rejecting acceptors.

Algorithm 5 Node α : acceptReply message handler

```

1: function HANDLE( $\langle$ acceptReply,  $ok, o, \lambda, b, s$  $\rangle$ ) from  $\beta$ 
2:   if  $ok$  then
3:     if  $b < \Sigma[\alpha][s].b \vee \Sigma[\alpha][s]$  is committed then
4:       return ▷ Ignore old reply
5:        $\Sigma[\alpha][s].ACK(\beta)$ 
6:       if  $\Sigma[\alpha][s].Q2.SATISFIED$  then
7:          $\Sigma[\alpha][s] \leftarrow$  committed
8:         BROADCAST( $\langle$ commit,  $\alpha, b[o], s, \gamma$  $\rangle$ )
9:     else
10:      if  $b > \mathbf{b}[o]$  then
11:         $\mathbf{I}[o] = \lambda$ 
12:         $\mathbf{b}[o] = b$ 
13:      if  $\Sigma[\alpha][s] \neq \perp$  is not committed then
14:        Put  $\Sigma[\alpha][s].\delta$  back to main request queue
15:       $\Sigma[\alpha][s] \leftarrow \perp$ 

```

3.4 Properties

WPaxos provides similar guarantees offered by other Paxos variants (EPaxos, Generalized Paxos) as well as some unique properties to its clients.

Non-triviality. For any node α , the set of committed commands is always a sequence σ of proposed commands, i.e. $\exists \sigma : committed[\alpha] = \perp \bullet \sigma$. Non-triviality is straightforward since nodes only start phase-1 or phase-2 for commands proposed by clients, in Algorithm 1.

Stability. For any node α , the set of committed commands at any time is a prefix of the set at any later time, i.e. $\exists \sigma : committed[\alpha] = \gamma$ at any $t \implies committed[\alpha] = \gamma \bullet \sigma$ at $t + \Delta$.

Consistency. For any leader α , if command γ is committed at instance $\mathcal{O}_{\alpha}.i$ by some node, no other node can have a different command committed for the same instance.

Liveness. A proposed command γ will eventually be committed by all non-faulty nodes, i.e. $\diamond \forall \alpha : \gamma \in committed[\alpha]$.

In the next revision, we will provide a modeling of WPaxos in TLA+ [14] and model-checking of these properties against the model.

4 Evaluation

We implemented WPaxos on top of Paxi, our reusable framework for evaluating Paxos-style consensus protocol. This framework allowed us to compare WPaxos and EPaxos in the same controlled environment under identical workloads. We conducted our experiments on a testbed consisting of AWS [2] EC2 medium Linux instances³ located at four AWS regions, namely: California (CA), Virginia (VA), Ireland (EU), and Japan (JP). Each AWS region corresponds to a single WPaxos zone.

³Two 64-bit virtual cores and 4GB memory.

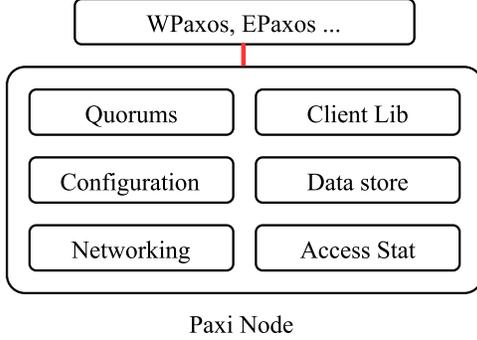


Figure 5: Schematic representation of Paxi framework and WPaxos

4.1 Implementation

We implemented a general framework, called *Paxi*, to accommodate for various styles of Paxos algorithms. WPaxos protocol was placed on top of Paxi by adding the inter-node message definition and message handling procedures. We also adopted the original EPaxos code to work with our framework. Both the framework and WPaxos protocol were built in Go version 1.8 and they will be available as an opensource project on GitHub repository <https://github.com/ailidani/paxi>.

Paxi provides extended abstractions to be shared between all Paxos variants, including location-aware configuration, network communication, client library and four types of quorum systems (majority quorum, fast quorum, grid quorum and flexible quorum), as shown in Figure 5. Networking layer encapsulates message passing model and exposes basic interfaces including broadcast, multicast, intra-zone, inter-zone and peer-to-peer messaging for cross-node traffics. Similar to EPaxos, Paxi incorporates the mechanisms to facilitate the startup for the system and share initial parameters through the configuration management. Paxi framework can accommodate a greater variety of quorums through its quorum management module. Both nodes and clients use the API provided by the framework.

4.2 Workload

Paxi provides a replicated key-value store as the state machine on top of the protocols under evaluation. The client library of Paxi’s key-value store has both synchronous and asynchronous version of update (put) and query (get) operations. We used different types of put operations in our evaluation to simulate practical and realistic workloads. Our experimental workloads exercise two primary parameters: conflict and locality.

Definition 4.1. *Conflict* c is the proportion of commands operated on the objects shared across zones.

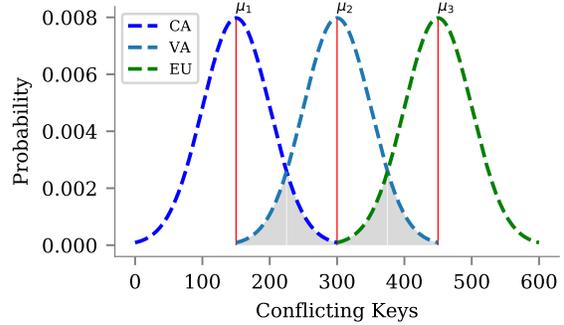


Figure 6: Workload with locality in each region

The workload with conflicting objects exhibits no locality if the objects are selected uniformly random at each zone. We introduce locality to our evaluation by drawing the conflicting keys from a Normal distribution $\mathcal{N}(\mu, \sigma^2)$, where μ can be varied for different zones to control the locality, and σ is shared between zones. The locality can be visualized as the non-overlapping area under the probability density functions, as illustrated in Figure 6.

Definition 4.2. *Locality* l is the complement of the overlapping coefficient (OVL)⁴ among workload distributions: $l = 1 - \widehat{OVL}$.

Let $\Phi(\frac{x-\mu}{\sigma})$ denote the cumulative distribution function (CDF) of any normal distribution with mean μ and deviation σ , and \hat{x} as the x-coordinate of the point intersected by two distributions, locality is given by $l = \Phi_1(\hat{x}) - \Phi_2(\hat{x})$. It is worth mentioning the two special cases when there is no single intersecting point: locality equals to 0 if two overlapping distributions are congruent, or equals to 1 if two distributions do not intersect.

In our experiments we vary the conflict and locality parameters to test WPaxos under different scenarios. We run each experiment for 5 minutes, given 500 keys that are shared across regions and 500 designated keys local to each region. We perform the experiments with three nodes in each of the three regions.

4.3 Quorum Tests

In the first set of experiments, we compare the latency of both $Q1$ and $Q2$ in two types of quorums, Flexible Grid (FG) and Flexible 2 Rows (F2R). Flexible grid quorum uses a single node per region for $Q1$, while F2R is our chosen WPaxos quorum approach as described in Section 2. Clients in each region simultaneously generate

⁴The overlapping coefficient (OVL) is a measurement of similarity between two probability distributions, refers to the shadowing area under two probability density functions simultaneously [12].

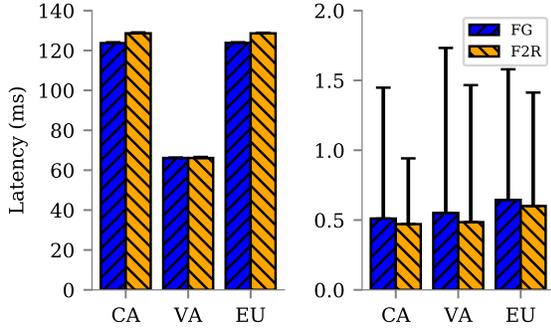


Figure 7: Median and 99%ile latency for phase-1 (left) and phase-2 (right)

the same number of phase-1 and phase-2 requests, and measure the commit latency for each phase. Figure 7 shows the median and 99th percentile latency in phase-1 (left) and phase-2 (right).

Quorum size of $Q1$ in FG is a half of that for F2R, but both experience a similar median latency of about one round trip to the farthest peer, since the communication happens in parallel and both FG and F2R are affected by WAN communication. Within a zone, however, F2R can tolerate one straggler node, reducing both median and 99th percentile latency.

4.4 Impact of Leader Switching

Objects in WPaxos can change their geographical location through object stealing procedures. The simplest routine attempts to steal the object from the remote zone upon the very first request, however this is not an ideal case for many realistic workloads in which objects exhibit locality, and yet need to be accessed from different zones. Our adaptive object stealing procedure utilizes the request frequency metric to control which zone can steal the object and when. The results of the adaptive stealing experiments will be added in the future revisions of this work. All the experiments below are performed with the immediate object stealing scheme.

4.5 Latency

We compare the commit latency between WPaxos and EPaxos with two set of workloads.

Figure 8 compares the median (color bar) and 99th percentile (error bar) latency with different conflicts in three regions. EPaxos always have to pay the price of WAN communication, while WPaxos tries to keep as many operations locally. With small conflicts $c \leq 50\%$, the median latency of EPaxos is about 1 RTT between the region and its closest neighboring region. WPaxos

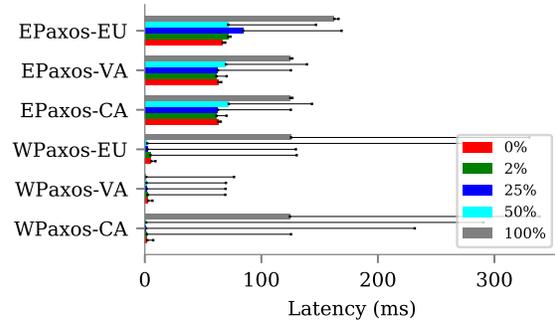


Figure 8: Median (color bar) and 99%ile (error bar) latency for EPaxos and WPaxos

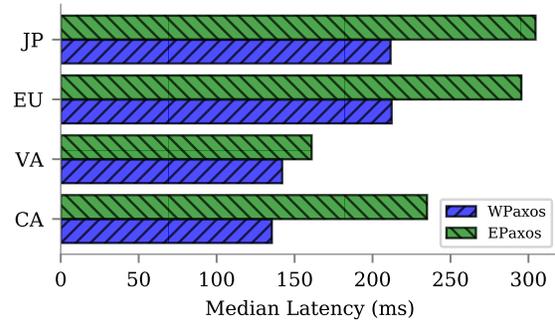


Figure 9: 100% Conflict median latency of 4 regions for EPaxos and WPaxos

reduces median latency to local commit time. Under full conflict ($c = 100\%$), both EPaxos and WPaxos degrade to full WAN RTT, as EPaxos no longer able to commit most commands in fast quorum, and WPaxos is forced to do frequent object-stealing. WPaxos, however can achieve good median latency in VA, which is a geographically central region in our topology. This is because the performance penalty for stealing an object to this region is significantly lower, allowing VA to process more requests, some of which will be local due to the previously stolen objects.

We repeat the 100% conflict experiment with 4 regions, adding Tokyo (JP) in Figure 9. In the new topology, EPaxos's fast quorum size expands to 3 regions instead of 2, hence the minimum commit latency increases to the second smallest RTT. The median latency of EPaxos, however, reflects more normal Paxos rounds under high conflicts.

Figure 11 shows the average latency of workload with locality derived from Figure 6, where conflict $c = 100\%$, $\sigma = 50$, $\mu = 150, 300, 450$ respectively, and locality $l = 86.6\%$. EPaxos replica in VA region experiences the

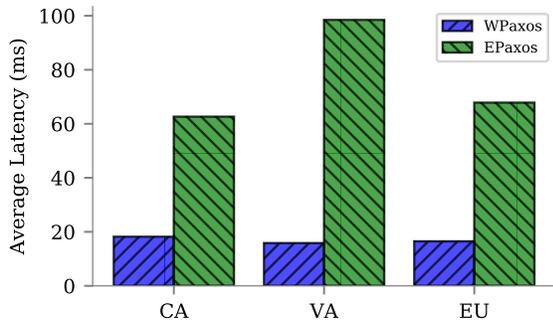


Figure 10: Average latency of locality workload for EPaxos and WPaxos

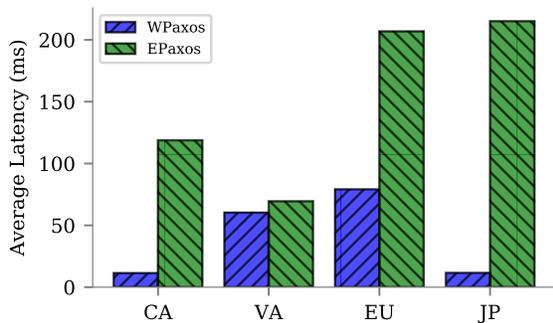


Figure 11: Average latency of locality workload of 4 regions for EPaxos and WPaxos

highest average commit latency because its distribution has overlaps with two other regions, whereas in WPaxos, this disadvantage is canceled out by favorable location that enable VA region to steal more objects. The average latency in WPaxos is one third to one fifth (depends on the region) compared to that of EPaxos.

4.6 Throughput

Similar to latency, we compare throughput of WPaxos and EPaxos. In theory, both system can process multiple non-conflicting objects independently. Throughput experiments will be added in the future revisions of this work.

5 Fault-tolerance

In WPaxos, progress is still possible as long as it can form a valid $Q1$ and $Q2$ quorums. Our default deployment scheme uses 3 nodes per zone, thus it can mask failure of a single node per zone and can still form $Q2$ quorum at that zone and $Q1$ quorums passing through

that zone in an unaffected manner. It is possible to fine-tune the per-zone fault tolerance by changing the number of nodes in each zone and sizes of $Q1$ and $Q2$ quorums.

A zone failure will make forming the $Q1$ impossible, thus halting all object movement in the system, however, object of leaders in unaffected zones can continue process requests, albeit with no locality adaptive properties. Objects in the affected zone will be unavailable for the duration of zone recovery.

A leader recovery is handled naturally by the object stealing procedure. Upon a leader failure, all of its objects will become unreachable at that leader, forcing the system to start object stealing phase. A failed node does not prevent the new leader from forming a $Q1$ quorum needed for object stealing, thus the new leader can proceed and acquire the leadership of an object. Normal object stealing procedure also calls for a recovery of accepted but not committed instances for the object in the failed leader log and the same procedure is carried out even when the original leader has failed.

6 Related Work

Several attempts have been made for addressing consensus scalability. Certain systems, such as Mencius [18] try to reduce the bottlenecks of a single leader by incorporating multiple rotating leaders. Mencius tries to eliminate the single entry-point requirement of Paxos and achieve better load balancing by partitioning consensus sequence numbers (consensus requests/instances) among multiple servers. This load balancing helps distribute the network bandwidth and CPU load better. However, Mencius does not address reducing the WAN latency of consensus.

Other Paxos variants go for a leaderless approach. EPaxos [20] is leaderless in the sense that any node can opportunistically become a leader for an operation. At first EPaxos tries the request on a fast quorum and if the operation was performed on a non-conflicting object, fast quorum will decide on the operation and replicate it across the system. However, if fast quorum detects a conflict (i.e., another node trying to decide another operation for the same object), EPaxos will default to the standard Paxos procedure.

Bizur [9] aims to process independent keys from its internal key-value store in parallel with the help of multiple leaders. However, it does not account for the data-locality nor is able to migrate the keys between the leaders. Bizur maps each key into a bucket with a hash function and replicates these buckets within the cluster, allowing different buckets to proceed independently. The buckets are rather static, with no quick procedure to move the key from one bucket to another, since such operation will require not only expensive reconfiguration phase, but also change in the key mapping function.

Bizur elects a leader for each bucket, and the leader becomes responsible for handling all requests and replicating the bucket in the cluster. The system can scale up by moving the buckets to new servers, however such scalability is assumed to be in the same datacenter.

ZooNet [16] is a client approach at improving the performance of WAN coordination. It tries to achieve fast reads at the expense of some data-staleness and slow writes by deploying multiple ZooKeeper services in different regions with observers in every other region. As such, the system operates just like ZooKeeper with the object-space statically partitioned across regions. ZooNet provides a client API for consistent reads by injecting sync requests when reading from remote regions.

7 Concluding Remarks

WPaxos achieves fast wide-area coordination by dynamically partitioning the objects across multiple leaders that are deployed strategically using flexible quorums. Such partitioning and emphasis on local operations allow our protocol to significantly outperform leaderless approaches, such as EPaxos, while maintaining the same consistency guarantees. Unlike statically partitioned Paxos used in Google’s Spanner and other systems, WPaxos adapts dynamically to the changing access locality through adaptive object stealing. The ability to quickly react to variations in access locality not only speeds up the protocol, but also enables support for minitransactions. Future work is to develop more sophisticated object stealing strategies, that not only pay attention to the actual object requests, but also for the object demand, since the demand may not perfectly match the number of requests made.

References

- [1] AGUILERA, M., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 159–174.
- [2] <http://aws.amazon.com>.
- [3] BAKER, J., BOND, C., CORBETT, J., FURMAN, J., KHORLIN, A., LARSON, J., ET AL. Megastore: Providing scalable, highly available storage for interactive services. *CIDR* (2011), 223–234.
- [4] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook’s distributed data store for the social graph. *Usenix Atc ’13* (2013), 49–60.
- [5] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *OSDI* (2006), USENIX Association, pp. 335–350.
- [6] CORBETT, J., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., ET AL. Spanner: Google’s globally-distributed database. *Proceedings of OSDI* (2012).
- [7] A distributed, reliable key-value store for the most critical data of a distributed system. <https://coreos.com/etcd/>, 2017.
- [8] GRIMSHAW, A., MORGAN, M., AND KALYANARAMAN, A. Gffs – the XSEDE global federated file system. *Parallel Processing Letters* 23, 02 (2013), 1340005.
- [9] HOCH, E. N., BEN-YEHUDA, Y., LEWIS, N., AND VIGDER, A. Bizur: A Key-value Consensus Algorithm for Scalable File-systems.
- [10] HOWARD, H., MALKHI, D., AND SPIEGELMAN, A. Flexible Paxos: Quorum intersection revisited.
- [11] HUNT, P., KONAR, M., JUNQUEIRA, F., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC* (2010), vol. 10.
- [12] INMAN, H. F., AND BRADLEY JR, E. L. The overlapping coefficient as a measure of agreement between probability distributions and point estimation of the overlap of two normal densities. *Communications in Statistics-Theory and Methods* 18, 10 (1989), 3851–3874.
- [13] JUNQUEIRA, F., REED, B., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems & Networks (DSN)* (2011), IEEE, pp. 245–256.
- [14] LAMPORT, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 16, 3 (May 1994), 872–923.
- [15] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (1998), 133–169.
- [16] LEV-ARI, K., BORTNIKOV, E., KEIDAR, I., AND SHRAER, A. Modular composition of coordination services. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016).
- [17] LLOYD, W., FREEDMAN, M., KAMINSKY, M., AND ANDERSEN, D. Don’t settle for eventual: Scalable causal consistency for wide-area storage with cops. In *SOSP* (2011), pp. 401–416.
- [18] MAO, Y., JUNQUEIRA, F. P., AND MARZULLO, K. Mencius: Building Efficient Replicated State Machines for WANs. *Proceedings of the Symposium on Operating System Design and Implementation* (2008), 369–384.
- [19] MASHTIZADEH, A. J., BITTAU, A., HUANG, Y. F., AND MAZIÈRES, D. Replication, history, and grafting in the ori file system. In *Proceedings of SOSP* (New York, NY, 2013), SOSP ’13, pp. 151–166.
- [20] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 358–372.
- [21] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), pp. 305–319.
- [22] QUINTERO, D., BARZAGHI, M., BREWSTER, R., KIM, W. H., NORMANN, S., QUEIROZ, P., ET AL. *Implementing the IBM General Parallel File System (GPFS) in a Cross Platform Environment*. IBM Redbooks, 2011.
- [23] VAN RENESSE, R., AND ALTINBUKEN, D. Paxos made moderately complex. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 42.