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

WPaxos is a multileader Paxos protocol that provides low-latency and high-throughput consensus across wide-area network (WAN) deployments. Unlike statically partitioned multiple Paxos deployments, WPaxos perpetually adapts to the changing access locality through object stealing. 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 zone-local phase-2 commits, WPaxos adopts the flexible quorums idea in a novel manner, and appoints phase-2 acceptors to be at the same zone as their respective leaders.

The perpetual dynamic partitioning of the object-space and emphasis on zone-local commits allow WPaxos to significantly outperform leaderless approaches, such as EPaxos, while maintaining the same consistency guarantees. We implemented WPaxos and evaluated it on WAN deployments across 5 AWS regions using the benchmarks introduced in the EPaxos work. Our results show that, for a ∼70% access locality workload, WPaxos achieves 2.4 times faster average request latency and 3.9 times faster median latency than EPaxos due to the reduction in WAN communication. For a ∼90% access locality workload, WPaxos improves further and achieves 6 times faster average request latency and 59 times faster median latency than EPaxos.

1 Introduction

Paxos, introduced in 1989 [17], 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) in the face of concurrent and asynchronous execution, crash/recovery of the nodes, and arbitrary loss of messages. When the conditions improve such that distributed consensus becomes solvable, Paxos satisfies the progress property (i.e., nodes decide on a value as a function of the inputs).

The Paxos algorithm and its variants have been deployed widely, including in Google Chubby [5] based on Paxos [28], Apache ZooKeeper [13] based on Zab [15], and etcd [7] based on Raft [24]. All of these implementations depend on a centralized primary process (i.e., the leader) to serialize all commands/updates. During normal operation, only one node acts as the leader: all client requests are forwarded to that leader, and the leader commits the requests by performing phase-2 of Paxos with the acceptors. Due to this dependence on a single centralized leader, these Paxos implementations 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 datacenters and sites) has gained greater importance, especially for database applications and NewSQL datastores [3, 6, 29], distributed filesystems [9, 22, 26], and social network metadata updates [4, 20].

In order to eliminate the single leader bottleneck, EPaxos [23] proposes a leaderless Paxos protocol where any replica at any zone can propose and commit commands opportunistically, provided that the commands are non-interfering. This opportunistic commit protocol requires an agreement from a fast-quorum of roughly 3/4ths of the acceptors[1], which means that WAN latencies are still incurred. Moreover, if the commands proposed by multiple concurrent opportunistic proposers do interfere, the protocol requires performing a second phase to record the acquired dependencies, and agreement from a majority of the Paxos acceptors is needed.

Another way to eliminate the single leader bottleneck is to use a separate Paxos group deployed at each zone.

[0] An archipelago is a chain, cluster, or collection of islands.

[1] For a deployment of size $2F+1$, fast-quorum is $F+\lfloor\frac{F+1}{2}\rfloor$

Systems like Google Spanner [6], ZooNet [19], Bizur [11] achieve this via a static partitioning of the global object-space to different zones, each responsible for a shard 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.

Contributions. We present *WPaxos*, a novel multileader Paxos protocol that provides low-latency and high-throughput consensus across WAN deployments.

WPaxos adapts the "*flexible quorums*" idea (which was introduced in 2016 summer as part of FPaxos [12]) to cut WAN communication costs. WPaxos uses the flexible quorums idea in a novel manner for deploying *multiple concurrent leaders* across the WAN. By strategically appointing the phase-2 acceptors to be at the same zone as the leader, WPaxos achieves local-area network commit decisions. We present how this is accomplished in Section 2.1.

Unlike the FPaxos protocol which uses a single-leader and does not scale to WAN distances, WPaxos uses multileaders and partitions the object-space among these multileaders. On the other hand, WPaxos differs from the existing static partitioned multiple Paxos deployment solutions, because it implements a dynamic partitioning scheme: leaders coinciding in different zones steal ownership/leadership of an object from each other using phase-1 of Paxos, and then use phase-2 to commit update-requests on the object locally until the object is stolen by another leader. We describe the WPaxos protocol in Sections 2.2, 2.3, and 2.4, and present the algorithm in detail in Section 3.

With its multileader protocol, WPaxos achieves the same consistency guarantees as in EPaxos: linearizability is ensured per object, and strict serializability is ensured across objects. We present safety and liveness properties of WPaxos in Section 3.4. We modeled WPaxos in TLA+ [16] and verified the consistency properties by model checking this TLA+ specification. The TLA+ specification is available at `http://github.com/wpaxos/tla`.

To quantify the performance benefits from WPaxos, we implemented WPaxos and performed evaluations on WAN deployments across 5 AWS regions using the benchmarks introduced in EPaxos [23]. Our results in Section 4 show that WPaxos outperforms EPaxos, achieving 2.4 times faster average request latency and 3.9 times faster median latency than EPaxos using a ∼70% access locality workload. Moreover, for a ∼90% access locality workload, WPaxos improves further and achieves X times faster average request latency and Y times faster median latency than EPaxos. This is because, while the EPaxos opportunistic commit protocol requires about 3/4ths of the Paxos acceptors to agree and incurs one WAN round-trip latency, WPaxos is able to achieve low latency commits using the zone-local phase-2 acceptors.

When we test for increased throughput in WAN deployments using the 70% locality workload, we found that WPaxos is able to maintain low-latency responses long after EPaxos latencies take a big hit. Under 10,000 requests/sec, WPaxos achieves 9 times faster average request latency and 54 times faster median latency than EPaxos. We also evaluated WPaxos with a shifting locality workload and show that WPaxos seamlessly adapts and significantly outperforms static partitioned multiple Paxos deployments.

While achieving low latency and high throughput, WPaxos also achieves seamless 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-election (i.e., object stealing) is 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. While WPaxos helps most for slashing WAN latencies, it is also suitable for deployment entirely inside the same datacenter/cluster for its high-availability and throughput benefits.

2 WPaxos

In this section we present a high-level overview of WPaxos, relegating a detailed presentation of the protocol to Section 3.

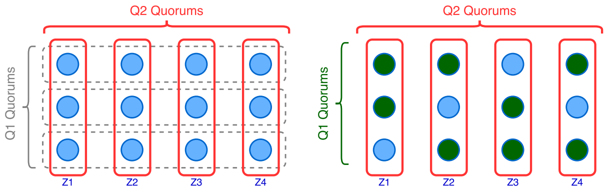
2.1 WPaxos Quorums

WPaxos leverages on the flexible quorums idea [12]. This surprising result shows that we can weaken Paxos' "all quorums should intersect" assertion 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, i.e., FPaxos, 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

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, 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



(a) Grid quorums with $Q1$s in rows and $Q2$s in columns

(b) WPaxos quorum with 2 nodes per region in $Q1$

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

respectively. An attractive property of this grid quorum arrangement is $Q1+Q2$ does not need to exceed 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.

WPaxos further relaxes the grid quorum constraints for $Q1$ to achieve a more fault-tolerant and efficient alternative. Instead of using rigid grid rows for $Q1$s, WPaxos picks nodes from each column regardless of 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 improved fault-tolerance.) In this deployment, each zone has 3 nodes, and each $Q2$ quorum is any 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

In contrast to FPaxos which uses flexible quorums with a classical single-leader Paxos protocol, WPaxos presents a multileader protocol over flexible quorums. 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 object in the system is maintained in its own commit log, allowing for per-object linearizability. A node can lead multiple objects at once, all of which may have different ballot and slot numbers in their corresponding logs.

The WPaxos protocol consists of two phases. The concurrent leaders *steal* ownership/leadership of objects from each other using phase-1 of Paxos executed over $Q1$. Then phase-2 commits the update-requests to the object over the corresponding $Q2$, and can execute multiple times until some other node steals the object.

The phase-1 of the protocol starts only 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. After a node becomes the owner/leader for an object, it repeats phase-2 multiple times on that object for committing commands/updates, incrementing the slot number at each iteration, while the ballot number for the object stays the same.

Figure 2 shows the normal operation of both phases, and also references each operation to the algorithms in Section 3. Table 1 summarizes some common terminology used throughout the paper.

2.3 Immediate Object Stealing

When a node needs to steal an object from another leader in order to carry out a client request, it first 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

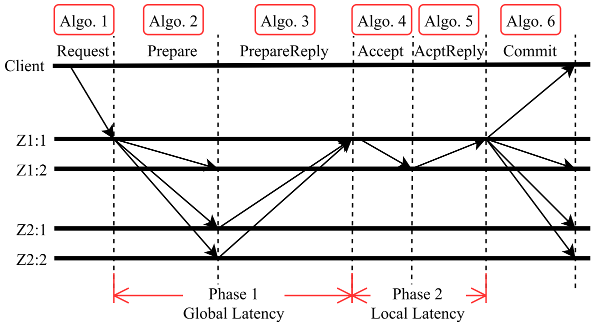
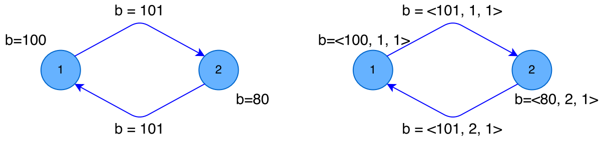


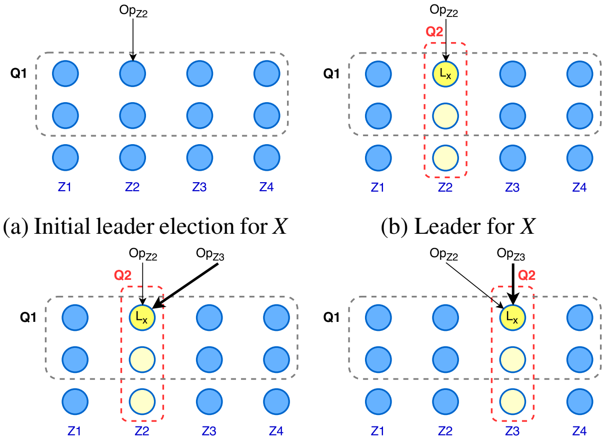
Figure 2: Normal case messaging flow of WPaxos.



(a) Ballot conflict between two nodes

(b) Ballot conflict is resolved by comparing ids

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.



(a) Initial leader election for X

(b) Leader for X

(c) Heavy cross-region traffic

(d) Object is migrated

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.

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. The 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 isolating the effects of object stealing. Keeping per-leader ballot numbers, i.e., keeping a single ballot number for all objects maintained by the leader, would necessitate out-balloting all objects of a remote leader when trying to steal one object. This would then create 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.

Using separate ballot numbers for each object allows 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 mitigate that issue, we use 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, maintaining per-object ballot numbers for one million objects would only require 16 MB of memory: 8 MB for 64-bit keys for objects and 8 MB more for the corresponding ballots.

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, that approach may cause a performance degradation when an object is frequently needed in many 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, and may 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.

3 Algorithm

We assume a set of nodes communicating through message passing in an asynchronous environment. Each WPaxos node is a deterministic state machine that maintains a set of variables and an internal datastore. The protocol updates the states of a node's 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 **instance**s in Σ, identified by an increasing slot number s. At most one command will be decided in any instance.

In the basic algorithm we present here, every command γ accesses only one object, identified by $\gamma.o$. Every node α leads its own set of objects $\mathcal{O}_\alpha$, however, each object has its command log. Object states are replicated to every node in the system, with local $Q2$ relying on phase-2 for replication, while the rest of the nodes learn the states as non-voting learners. Ballot numbers $\mathbf{b}$ for each object's log are constructed by concatenation of counter and the leader-node id $\langle c \bullet \lambda \rangle$. Therefore, any acceptor α can identify the current leader by examining object's ballot number. 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 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 represents 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
$\mathcal{O}_\lambda$	Set of objects led by λ
$\mathbf{H}$	Access history
$\bullet$	Concatenation operator
$\langle \mathbf{type}, \gamma, \mathbf{b}[o], \mathbf{s}[o] \rangle$	General message format

Algorithms 1-6 show the operations of a WPaxos node. Phase-1 of the protocol is described in the algorithms 1-3, while algorithms 4-6 cover phase-2.

3.1 Initialization

Node α Initialization

```
1: function INIT($\mathcal{O}$)
2:     $\mathbf{b}, \mathbf{s}, \Pi, \Sigma, \mathbf{H} \leftarrow \emptyset$
3:     $\forall o \in \mathcal{O} : \mathbf{b}[o] \leftarrow \langle 1 \bullet \alpha \rangle$        ▷ Initial ballot number
4:     $\forall o \in \mathcal{O} : \mathbf{s}[o] \leftarrow 0$        ▷ Initial slot number
```

The INIT($\mathcal{O}$) function describes the state initialization of the nodes. We assume no prior knowledge of the locations of objects or ballots. While WPaxos makes the initial object assignment optional, a user may provide the set of starting objects, allowing the initialization routine to construct ballots (lines 3-4).

3.2 Phase-1: Prepare

Algorithm 1 Node α: client request handler

```
function RECEIVE($\langle$request, $\gamma\rangle$ from $\kappa$)
    $o \leftarrow \gamma.o$        ▷ The object in command $\gamma$
    if $o \notin \mathbf{b}$ then        ▷ Unknown object
        STARTPHASE-1($\gamma$)        ▷ Phase 1
        return
    $\langle c \bullet \lambda \rangle \leftarrow \mathbf{b}[o]$        ▷ Get leader from ballot
    if $\alpha = \lambda$ then        ▷ Leader is self $\alpha$
        if $o \in \Pi$ then        ▷ $\exists$ phase1-request of $o$
            $\Pi[o] \leftarrow \Pi[o] \cup \{\gamma\}$        ▷ Append to current phase-1
        else        ▷ $\alpha$ is the current leader of $o$
            STARTPHASE-2($\gamma$)        ▷ Phase 2
        $\mathbf{H} \leftarrow \mathbf{H} \cup \{o, \kappa\}$        ▷ Save to access history $\mathbf{H}$
        if $\mathbf{H}$ triggers migration event then
            SEND($\beta$, $\langle$migrate, $\gamma.o\rangle$)
    else        ▷ $o$ is owned by other node
        if Immediate object stealing then
            $\mathbf{b}[o] \leftarrow \mathbf{b}[o] + 1$        ▷ Steal with new ballot
            STARTPHASE-1($\gamma$)
        else        ▷ Adaptive object stealing
            SEND($\lambda$, $\langle$request, $\kappa, \gamma\rangle$)        ▷ Forward to node $\lambda$

function STARTPHASE-1($\gamma$)
    $o \leftarrow \gamma.o$
    if $o \in \Pi$ then
        $\Pi[o] \leftarrow \Pi[o] \cup \{\gamma\}$
        return
    $\Pi[o] \leftarrow$ NEWQUORUM(Q1)        ▷ Waiting quorum of phase 1
    BROADCAST($\langle$prepare, $o, \mathbf{b}[o]\rangle$)        ▷ Start phase 1

function STARTPHASE-2($\gamma$)
    $o \leftarrow \gamma.o$
    $\mathbf{s}[o] \leftarrow \mathbf{s}[o] + 1$        ▷ Next available slot
    $\Sigma[o][\mathbf{s}[o]] \leftarrow \langle$instance, $\gamma, \mathbf{b}[o]$, NEWQUORUM($Q2$)$\rangle$
                                        ▷ Create new instance
    MULTICAST($\langle$accept, $\gamma, \mathbf{b}[o], \mathbf{s}[o]\rangle$)        ▷ Start phase 2
```

As shown in Algorithm 1, the WPaxos protocol starts with the client κ sending a $\langle$**request**$, \gamma\rangle$ message to one of the nodes α in the system. A client typically chooses a node in a local zone 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, the node α checks if the object exists in the set of ballots **b**, and starts phase-1 for any new objects by invoking STARTPHASE-1 procedure (lines 3-5).

If the ballot points to the node itself, then it appends the request to current in progress phase-1 if exists, or initiates phase-2 of the protocol in STARTPHASE-2 function (lines 7-11). Phase-2 sends a message to its $Q2$ quorum, and creates a new instance for slot s_α (lines 29-32). However, if the object is found to be managed by some other remote leader λ, depending on the configuration, α will either start immediate object stealing with larger ballot in phase-1 (lines 16-18), or forward the request to λ (line 20). Forwarding may fail when the local hints of the leader is obsolete, in which case node α will broadcast the request.

As part of the locality adaptive object stealing, the leader keeps track of every object's access history (line 12) to determine the most suitable location for the object. Current object leader may decide to relinquish the object ownership based on the locality adaptive rule. In that case, the leader sends out a **migrate** message to the node it determined to be more suitable to lead the object (lines 13-14).

Algorithm 2 Node α: prepare message handler

```
1: function HANDLE(⟨prepare, $o, b_\lambda$⟩) from $\lambda$
2:     for all $s \in \mathbf{s}[o]$ do
3:         if $\Sigma[o][s].b = \mathbf{b}[o] \wedge \Sigma[o][s]$.committed $= false$ then
4:             accepted ← accepted $\cup \langle \mathbf{b}[o], s, \Sigma[o][s].\delta \rangle$
5:     if $b_\lambda > \mathbf{b}[o]$ then
6:         $\mathbf{b}[o] \leftarrow b_\lambda$
7:     SEND($\lambda$, ⟨prepareReply, $o, \mathbf{b}[o]$, accepted⟩)
```

The HANDLE routine of algorithm 2 processes the incoming prepare message sent during phase-1 initiation. The node α can accept the sender node λ as the leader for object o only if ballot b_λ it received is greater than the ballot number α is currently aware of (lines 5-6). Node α collects all uncommitted instances with their slot, ballot and command into the accepted set, and replies node λ with the accepted set so that any unresolved commands can be committed by the new leader (lines 2-4).

Algorithm 3's HANDLE function collects the prepare replies sent by the Algorithm 2 and updates the uncommitted instances with a higher accepted ballot (lines 2-5). The node becomes the leader of object only if the $Q1$ quorum is satisfied after receiving the current message from β (lines 6-7). The new leader can then recover any uncommitted slots with suggested commands (line 8-9), and start the accept phase for the pending requests that have accumulated in Π (lines 10-11). Finally, the object is removed from the phase-1 outstanding set Π (line 12). However, if any reply message has a higher ballot b_β, it means b_β has adopted another leader. As such, it is no longer possible to decide commands using current ballot, so the node simply updates ballot $\mathbf{b}[o]$ to the value it learned (lines 13-14) and retries any pending requests after some random back-off time (lines 16-17).

Algorithm 3 Node α: prepareReply message handler

```
1: function HANDLE(⟨prepareReply, $o, b_\beta$, accepted⟩) from $\beta$
2:     if $b_\beta = \mathbf{b}[o]$ then
3:         for all $\langle b, s, \delta \rangle \in$ accepted do
4:             if $b > \Sigma[o][s].b$ then
5:                 $\Sigma[o][s] \leftarrow \langle b, \delta \rangle$
6:         $\Pi[o].Q1.$ACK($\beta$)                                ▷ Ack by $\beta$
7:         if $Q1.$SATISFIED then
8:             for all $\langle s, \delta \rangle \in \Sigma[o]$ not committed do
9:                 MULTICAST(⟨accept, $\delta, \mathbf{b}[o], s$⟩)
10:            for all $\gamma \in \Pi[o]$ do        ▷ Process all pending requests
11:                HANDLE(⟨request, $\gamma$⟩)
12:            $\Pi \leftarrow \Pi \setminus \{o\}$
13:    else if $b_\beta > \mathbf{b}[o]$ then                          ▷ Preempted
14:        $\mathbf{b}[o] \leftarrow b_\beta$                                    ▷ Update ballot
15:        for all $\gamma \in \Pi[o]$ do
16:            HANDLE(⟨request, $\gamma$⟩)              ▷ Retry pending requests
17:    else return                                    ▷ Ignore old reply msg
```

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 communication is kept local to the zone, greatly reducing the latency.

Algorithm 4 Node α: accept message handler

```
1: function HANDLE(⟨accept, $\gamma, b_\lambda, s$⟩) from $\lambda$
2:     $o \leftarrow \gamma.o$
3:     if $b_\lambda = \mathbf{b}[o]$ then
4:         $\Sigma[o][s] \leftarrow$ ⟨instance, $\gamma, b_\lambda$, committed ← false⟩
5:     SEND($\lambda$, ⟨acceptReply, $o, \mathbf{b}[o], s$⟩)
```

Once the leader of the object sends out the accept message at the beginning of the phase-2, the acceptors respond to this message as shown in Algorithm 4. Node α updates its instance $\Sigma[o][s]$ at slot s only if the message ballot b_λ is same as accepted ballot $\mathbf{b}[o]$ (lines 3-4).

The node collects replies from its $Q2$ acceptors in Algorithm 5. The request proposal either gets committed when a sufficient number of successful replies are received (lines 2-6), or aborted if some acceptors reject the proposal citing a higher ballot b_β (lines 7-11). In case of

Algorithm 5 Node α: acceptReply message handler

```
1:  function HANDLE($\langle$acceptReply$, o, b_\beta, s\rangle$) from $\beta$
2:      if $b_\beta = \mathbf{b}[o]$ then
3:          $\Sigma[o][s].Q2.$ACK$(\beta)$                    ▷ Ack by $\beta$
4:          if $Q2.$SATISFIED then
5:              $\Sigma[o][s].$committed $\leftarrow$ true
6:              BROADCAST($\langle$commit$, \Sigma[o][s].\gamma, \mathbf{b}[o], s\rangle$)
7:      else if $b_\beta > \mathbf{b}[o]$ then
8:          $\mathbf{b}[o] \leftarrow b_\beta$
9:          if $\Sigma[o][s] \neq \perp$ then
10:             Put $\Sigma[o][s].\gamma$ back to main request queue
11:             $\Sigma[o][s] \leftarrow \perp$
12:     else return                                ▷ Ignore old reply msg
```

rejection, the node updates a local ballot and puts the request in this instance back to main request queue to retry later (lines 8-11).

Algorithm 6 Node α: commit message handler

```
1: function HANDLE($\langle$commit$, \gamma, b_\lambda, s\rangle$) from $\lambda$
2:     $o \leftarrow \gamma.o$
3:     if $b_\lambda > \mathbf{b}[o]$ then
4:         $\mathbf{b}[o] \leftarrow b_\lambda$
5:     if $\Sigma[o][s] = \perp$ then
6:         $\Sigma[o][s] \leftarrow \langle$instance$, \gamma, b_\lambda,$ committed $\leftarrow$ true$\rangle$
7:     else
8:         $\Sigma[o][s].$commit $\leftarrow$ true
```

Finally, Algorithm 6 shows the receipt of a $\langle$**commit**$\rangle$ message for slot s by a node α. Since α may not be included in the $Q2$ of the accept phase, it needs to update the local ballot (lines 3-4) and create the instance in $\Sigma[o][s]$ if it is absent (lines 5-6).

3.4 Properties

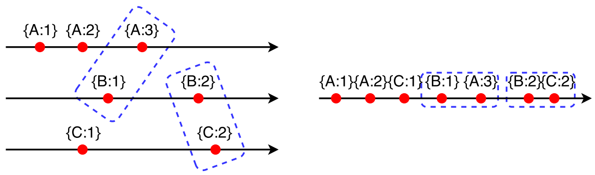
WPaxos protocol provides similar guarantees and properties offered by other Paxos variants, such as EPaxos and Generalized Paxos.

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 sequence of committed commands at any time is a prefix of the sequence 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 slot of any object, no two leaders can commit different values. This property asserts that object stealing and failure recovery procedures do not override any previously accepted or committed values. We verified this consistency property by model checking a TLA+ specification of WPaxos algorithm.

WPaxos consistency guarantees are on par with other protocols, such as EPaxos, that solve the generalized consensus problem, first introduced in 2005 [18]. Generalized consensus relaxes the consensus requirement by allowing non-interfering commands to be processed concurrently. Generalized consensus no longer enforces a totally ordered set of commands. Instead only conflicting commands need to be ordered with respect to each other, making the command log a partially ordered set. WPaxos maintains separate logs for every object and provides per-object linearizability. Moreover, for multi-object commands WPaxos can solve generalized consensus and provide strict serializability. This is achieved by collating/serializing the logs together, establishing the order of interfering commands by comparing the slot numbers of the objects in the commands, as shown in Figure 5. This ordering happens naturally as the commands cannot get executed before the previous slots for all objects in the command are executed. The serializability we achieve through the logs collation along with the per-object linearizability of all objects in the system make WPaxos a strictly serializable protocol [1, 10]. We relegate implementation and evaluation of multi-object commands to future work.



(a) Logs for 3 objects A, B, and C. Dashed box encompasses multi-object commands.

(b) Object logs are serialized. Multi-object commands are ordered based on the slot numbers.

Figure 5: Possible log serialization for objects A, B, and C.

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

4 Evaluation

We developed a general framework, called *Paxi* to conduct our evaluation. The framework allows us to compare WPaxos, EPaxos, and several other Paxos protocols in the same controlled environment under identical workloads. We implemented Paxi along with WPaxos and EPaxos in Go version 1.8 and released it as an open-source project on GitHub at `https://github.com/wpaxos/paxi`. The framework provides extended abstractions to be shared between all Paxos variants, including location-aware configuration, network communication, client library, and a quorum management module (which accommodates majority quorum, fast quo-

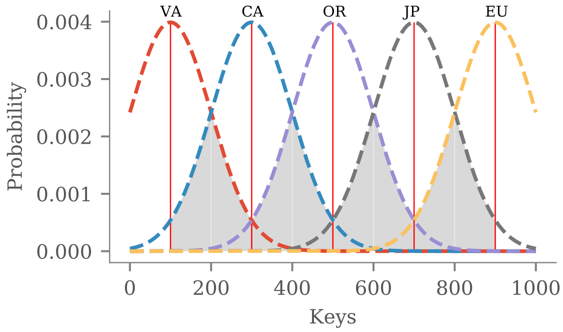


Figure 6: Workload with locality in each region.

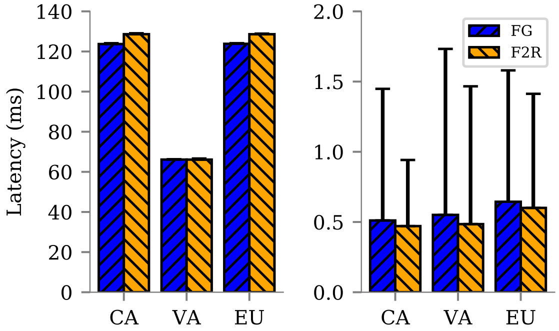


Figure 7: Median and 99%ile latencies for phase-1 (left) and phase-2 (right).

rum, grid quorum and flexible quorum). Paxi's networking layer encapsulates a message passing model and exposes basic interfaces for a variety of message exchange patterns, ranging from direct messaging between nodes to broadcasting for the entire cluster. Additionally, our Paxi framework incorporates mechanisms to facilitate the startup of the system by sharing the initial parameters through the configuration management tool.

4.1 Setup

We evaluated WPaxos using the key-value store abstraction provided by our Paxi framework. We used Amazon AWS EC2 [2] to deploy WPaxos across 5 different regions: Virginia (VA), California (CA), Oregon (OR), Japan (JP), and Ireland (EU). In our experiments, we used 3 medium instances with 2 vCPUs and 4 GB of RAM at each AWS region to host WPaxos.

In order to simulate workloads with tunable access locality patterns we used a normal distribution to control the probability of performing a request on each object from a set of all objects. As shown in the Figure 6, we used a pool of 1000 common objects, with the probability function of each region denoting how likely an object is to be selected at a particular zone. Each region has a set of objects it is more likely to access. We define **locality** as the percentage of the requests or commands pulled from such set of likely objects.

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 in Figure 6.

Definition 4.1. ***Locality*** L is the complement of the overlapping coefficient (OVL)[2] 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 $L = \Phi_1(\hat{x}) - \Phi_2(\hat{x})$. At the two ends of the spectrum, locality equals to 0 if two overlapping distributions are congruent, and locality equals to 1 if two distributions do not intersect.

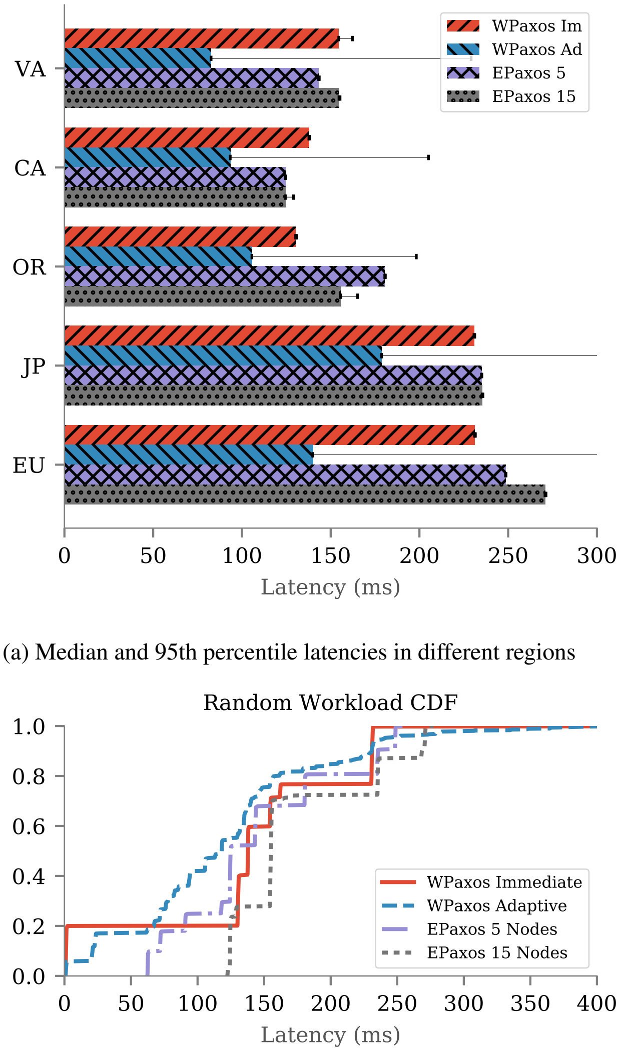
4.2 Quorum Latencies

Choosing proper quorums in a flexible quorum setup is important for the overall performance of the system. It is possible to tune the performance and fault tolerance by adjusting the $Q1$ and $Q2$ selections.

In this first set of experiments, we compare the latency of $Q1$ and $Q2$ accesses in two types of quorums: flexible grid (FG) and flexible 2 rows (F2R). Flexible grid quorum uses a single node per zone/region for $Q1$, requiring all nodes in the zone/region to form a $Q2$. Flexible 2 row configuration is default for WPaxos and uses any 2 rows per zone for $Q1$ and consequently requires one fewer node in $Q2$ than FG. This means that $Q2$ quorum in a region with 3 nodes consists of any 2 nodes and can tolerate one node failure. For this experiment we used a 3 region WPaxos deployment. In each region we simultaneously generated a fixed number of phase-1 and phase-2 requests, and measured the 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

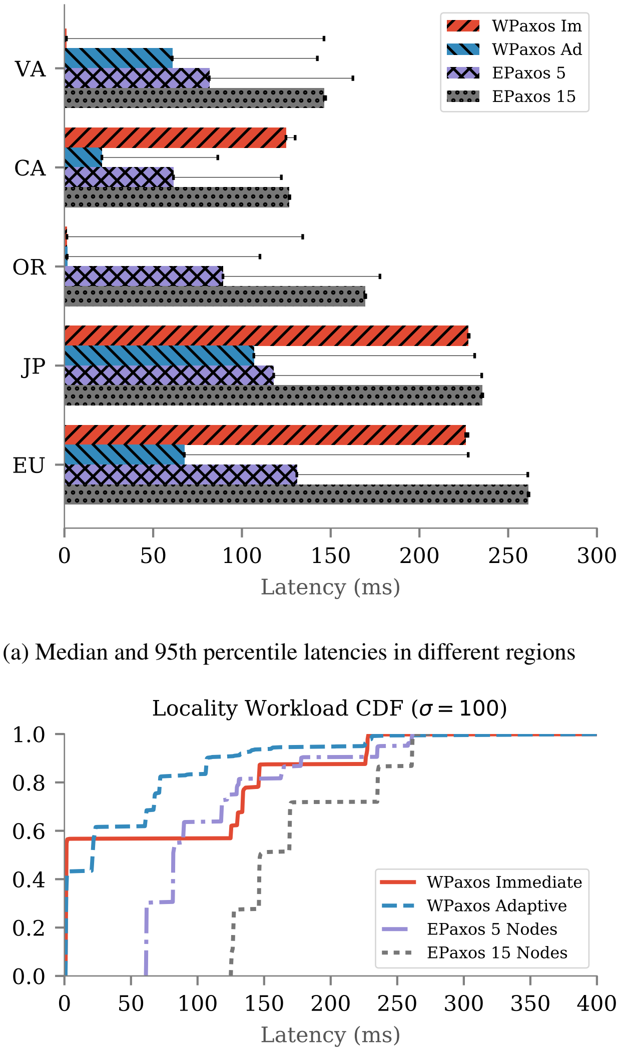
[2] The overlapping coefficient (OVL) is a measurement of similarity between two probability distributions, refers to the shadowing area under two probability density functions simultaneously [14].



(a) Median and 95th percentile latencies in different regions

(b) Cumulative distribution of global latencies

Figure 8: Random workload.



(a) Median and 95th percentile latencies in different regions

(b) Cumulative distribution of global latencies

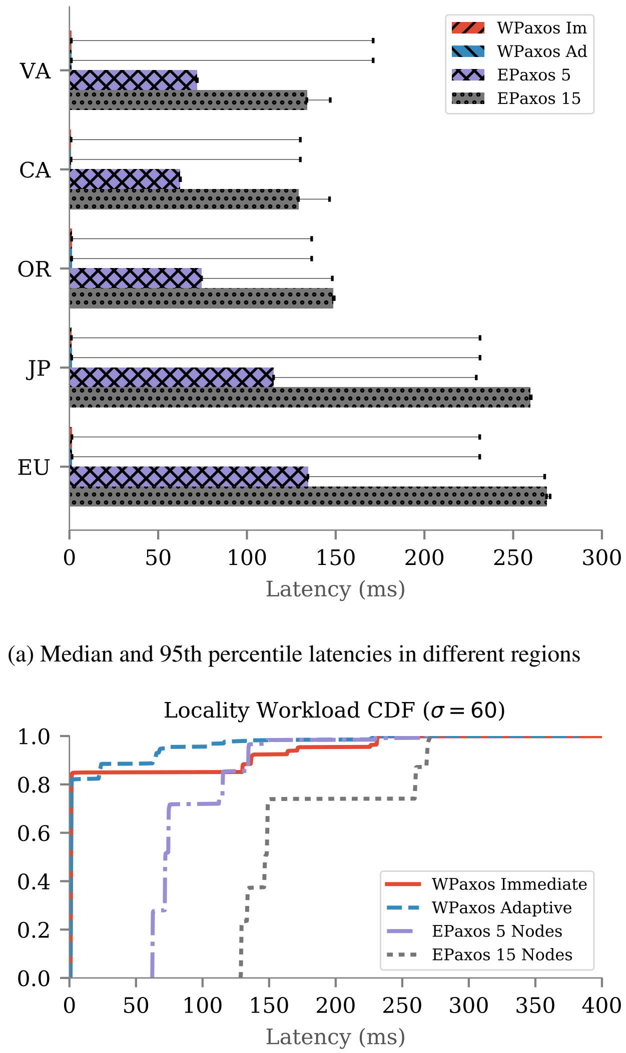
Figure 9: Locality (70%) workload.

one round trip to the farthest peer, since the communication happens in parallel and both FG and F2R are equally affected by the WAN latency. Within a zone, however, F2R can tolerate one straggler node, reducing both median and 99th percentile latency for the most frequently used quorum type.

4.3 Latency

We compare the commit latency between WPaxos and EPaxos with three sets of workloads, random (Figure 8), ∼70% locality (Figure 9), and ∼90% locality (Figure 10). In a duration of 5 minutes, clients in each region generate requests concurrently. WPaxos is deployed with both Immediate and Adaptive versions and EPaxos is deployed with 5 and 15 node versions.

Figure 8a compares the median (color bar) and 95th percentile (error bar) latency of random workload in 5 regions. Each region experiences different latency due to their asymmetrical location in the geographical topology. WPaxos with immediate object stealing shows a similar performance with EPaxos because the random workload causes many phase-1 invocations which require wide area RTT. However, WPaxos with adaptive mode outperforms EPaxos in every region. Figure 8b shows the distributions of aggregated latencies. Even though WPaxos immediate mode enables around 20% of local commit latency, WPaxos adaptive mode smoothens the latency by avoiding unnecessary leader changes and



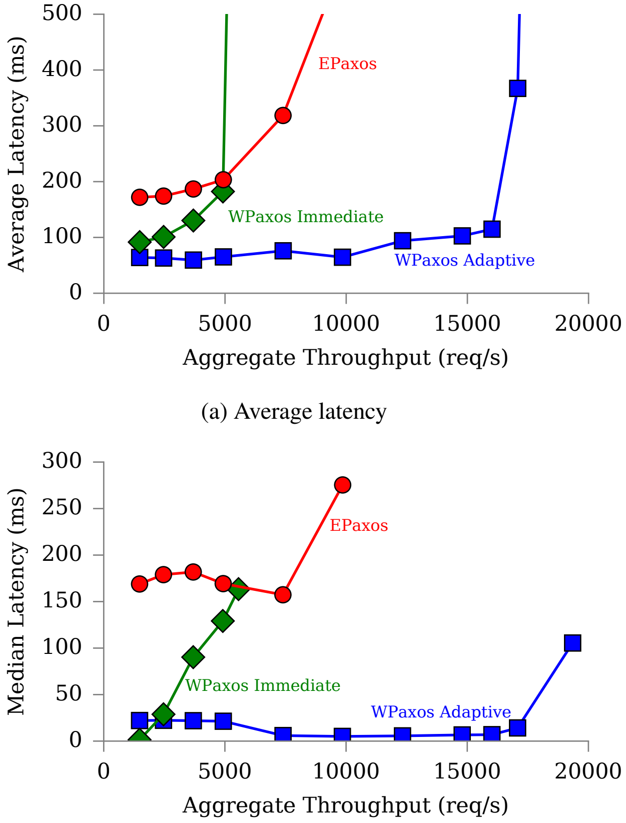
(a) Median and 95th percentile latencies in different regions

(b) Cumulative distribution of global latencies

Figure 10: Locality (90%) workload.

improves average latency.

EPaxos always has to pay the price of WAN communication, while WPaxos tries to keep operations locally as much as possible. Figure 9a shows that, under ∼70% locality workload, regions located in geographic center improve their median latencies. Regions JP and EU suffer from WPaxos immediate object stealing because their Q1 latencies are longer as they remain more towards the edge of the topology. WPaxos adaptive alleviates and smoothens these effects. With EPaxos 5 nodes deployment, the median latency is about 1 RTT between the regions and its second closest neighbor because the fast quorum size is 3. In EPaxos 15 nodes, the fast quorum size increases to 11, which increases the latency and reduce chance of conflict free commits. From an aggregated perspective, the cumulative distribution in Figure 9b indicates about half of the requests are commit in local-area latency in WPaxos. The global average latency of WPaxos Adaptive and EPaxos 5 nodes are 45.3ms and 107.3ms respectively.



(a) Average latency

(b) Median latency

Figure 11: Request latency as the throughput increases.

In Figure 10a we increase the locality to ∼90%. EPaxos shows similar pattern as previous experiments, whereas WPaxos achieves local median latency in all regions. In Figure 10b, 80% of all requests are able to commit with local quorum in WPaxos. The average latency of WPaxos Adaptive and EPaxos 5 nodes are 14ms and 86.8ms respectively, and the median latencies are 1.21ms and 71.98ms.

4.4 Throughput

We experiment on scalability of WPaxos with respect to the number of requests it processes by driving a steady workload at each zone. Instead of the *medium* instances we used in our other experiments, we used a cluster of 15 *large* EC2 instances with 2 vCPUs and 8 GB RAM each to host WPaxos deployments. EPaxos is hosted at the same instances, but with only one EPaxos node per zone. We opted out of using EPaxos with 15 nodes, because our preliminary experiments showed significantly higher latencies with such a large EPaxos deployment. (A 15 node EPaxos cluster needs very large fast quorum, and additionally having 3 nodes at each region results in the unnecessary contention for the objects that belong to the region.)

Clients on a separate VM in each region generate the workload by issuing a specific number of requests each second. We limit WPaxos deployments to a single leader per zone to be better comparable to EPaxos. We gradually increase the load on the systems by issuing more requests and measure the latency at each of the throughput levels. Figure 11 shows the latencies as the aggregate throughput increases.

At low load, we observe both immediate and adaptive WPaxos significantly outperform EPaxos as expected. With relatively small number of requests coming through the system, WPaxos has low contention for object stealing, and can perform many operations locally within a region. On the contrary, even without the contention, EPaxos needs to pay WAN cost to reach to a fast quorum, making its requests take longer. As the number of requests increases and contention rises, performance of both EPaxos and WPaxos with immediate object stealing deteriorates.

Immediate WPaxos suffers from leaders competing for objects with neighboring regions, degrading its performance faster than EPaxos. Figure 11b illustrating median request latencies shows this deterioration more clearly. This behavior in WPaxos with immediate object stealing is caused by dueling leaders: as two nodes in neighboring zones try to acquire ownership of the same object, each restarts phase-1 of the protocol before the other leader has a chance to finish its phase-2. When studying performance of WPaxos under 90% locality, we observed much greater scalability of immediate object stealing mode due to greatly reduced contention for the objects between neighboring zones.

On the other hand, WPaxos in adaptive object stealing mode scales much better and shows almost no degradation until it starts to reach the CPU and networking limits of individual instances. Adaptive WPaxos median latency actually decreases under the medium workloads, while EPaxos shows gradual latency increases. At the workload of 10000 req/s adaptive WPaxos outperforms EPaxos 9 times in terms of average latency and 54 times in terms of median latency.

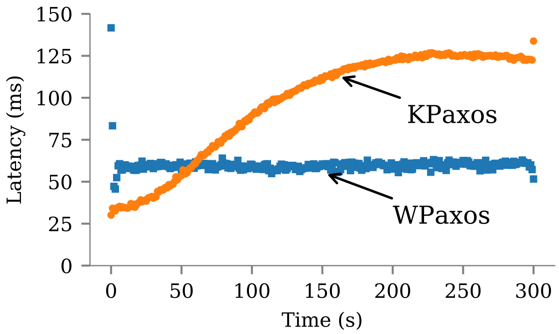


Figure 12: The average latency in each second.

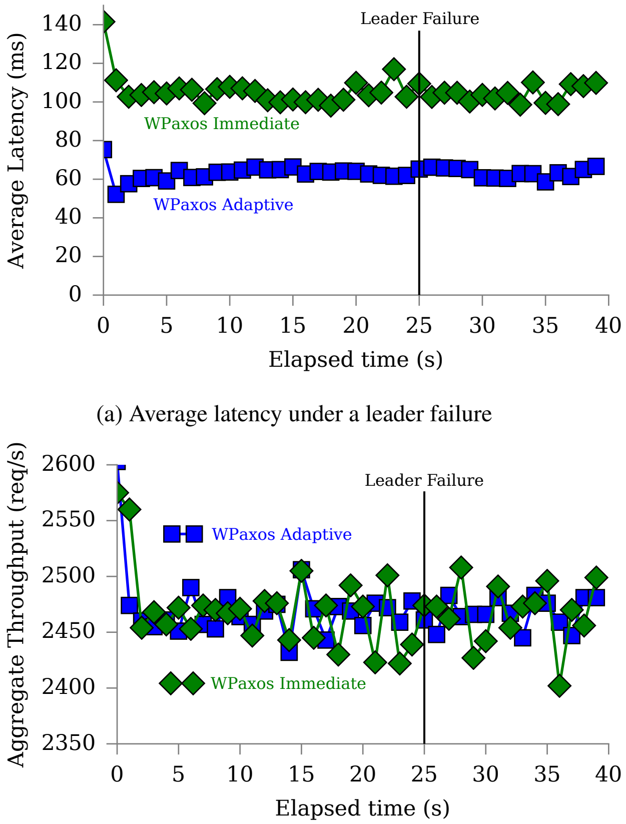
4.5 Shifting Locality Workload

Many applications in the WAN setting may experience workloads with shifting access patterns. Diurnal patterns are common for large scale applications that process human input [8, 30]. For instance, social networks may experience drifting access patterns as activity of people changes depending on the time of the day. WPaxos is capable to adapt to such changes in the workload access patterns while still maintaining the benefits of locality. While statically partitioned solutions perform well when there are many local requests accessing each partition, they require a priori knowledge of the best possible partitioning. Moreover, statically partitioned Paxos systems cannot adjust to changes in the access pattern, thus their performance degrade when the locality shifts.

In Figure 12, we illustrate the effects of shifting locality in the workload. Statically key-partitioned Paxos (KPaxos) starts in the optimal state with most of the requests done on the local objects. When the access locality is gradually shifted by changing the mean of the locality distributions at a rate of 2 objects/sec, the access pattern shifts further from optimal for statically partitioned Paxos, and its latency increases. WPaxos, on the other hand, does not suffer from the shifts in the locality. The adaptive algorithm slowly migrates the objects to regions with more demand, providing stable and predictable performance under shifting access locality.

5 Fault-tolerance

WPaxos is able to make progress as long as it can form 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 to form $Q2$ quorums at that zone and form $Q1$ quorums passing through that zone. It is possible to further increase the fault tolerance guarantees by configuring the number of nodes in each zone and sizes of $Q1$ and $Q2$ quorums.



(a) Average latency under a leader failure

(b) Throughput under a leader failure

Figure 13: Leader failure in one zone has no measurable impact on performance.

A leader recovery is handled naturally by the object stealing procedure. Upon a leader failure, all of its objects will become unreachable through that leader, forcing the system to start the object stealing phase. A failed node does not prevent the new leader from forming a $Q1$ quorum needed for acquiring an object, thus the new leader can proceed and get the leadership of any object. The object stealing procedure also recovers accepted but not committed instances in the previous leaders log for the object and the same procedure is carried out when recovering from a failure.

Figure 13a illustrates that there is negligible impact on performance due to a single leader/node failure. In this experiment, we ran a steady locality biased workload of about 2500 req/s through the WPaxos deployment. At 25 second mark we took the leader replica in OR region offline, however this has virtually no impact on the system performance. For immediate WPaxos, leader failure means all clients in that zone can simply communicate to another available local replica, and that replica will start phase-1 to acquire the objects of the dead node when a request for the object comes. Adaptive WPaxos will act in a similar way, except it will not start phase-1 if the object has already been acquired by another region. The effects of a single leader failure are not instant either, and the full recovery spreads in time, as recovery for each individual object happens only when that object is needed. The throughput of the system is not affected either, as shown in Figure 13b.

A zone failure will make forming the $Q1$ impossible in WPaxos, halting all object movement in the system. However, leaders in all unaffected zones can continue to process requests on the objects they already own. Immediate WPaxos suffers more, as it will not be able to perform remote requests on unaffected objects. On the other hand, adaptive WPaxos can still serve all request (remote or local) for all unaffected objects. In both WPaxos modes, objects in the affected regions will be unavailable for the duration of zone recovery.

Network partitioning can affect the system in a similar manner as a zone failure. If a region is partitioned from the rest of the system, no object movement is possible, however both partitions will be able to make some progress in WPaxos under the adaptive object stealing rules.

6 Related Work

Several attempts have been made for improving the scalability of Paxos. Table 2 summarizes properties of some of the significant ones.

Mencius [21] proposes to reduce the bottlenecks of a single leader by incorporating multiple rotating leaders. Mencius tries to achieve better load balancing by partitioning consensus sequence/slot numbers among multiple servers, and aims to distribute the strain over the network bandwidth and CPU. However, Mencius does not address reducing the WAN latency of consensus.

Other Paxos variants go for a leaderless approach. EPaxos [23] 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, the 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 requires performing a second phase to record the acquired dependencies requiring agreement from a majority of the Paxos acceptors.

A recent Paxos variant, called M^{2}Paxos [25], takes advantage of multileaders: each node leads a subset of all objects while forwarding the requests for objects it does not own to other nodes. Each leader runs phase-2 of Paxos on its objects using classical majority quorums. Object stealing is not used for single-object

Table 2: Protocol comparison

	WPaxos	EPaxos	M^2Paxos	Bizur	ZooNet	FPaxos
WAN-optimized	✓	✓	✗	✗	✓	✗
Multi-leader	✓	✗(leaderless)	✓	✓(static)	✓	✗
Concurrent	✓	✓	✓	✓(across buckets)	✓	✗
Object-stealing	✓	✗	✓(partial)	✓	✗	✗
Locality adaptive	✓	✗	✓(partial)	✗	✗	✗
Quorums	flexible	fast & classical	classical	classical	classical	flexible
Log	per-object	per-node	per-object	register	per-zone	single
Consistency	generalized	generalized	generalized	per-bucket	reads	strong

commands, but is supported for multiple-object update clients. M^2Paxos does not address WAN deployments and is subject to WAN latencies for commit operations since it uses majority quorums.

Bizur [11] also uses multileaders to process independent keys from its internal key-value store in parallel. However, it does not account for the data-locality nor is able to migrate the keys between the leaders. Bizur elects a leader for each bucket, and the leader becomes responsible for handling all requests and replicating the objects mapped to the bucket. The buckets are static, with no procedure to move the key from one bucket to another: such an operation will require not only expensive reconfiguration phase, but also change in the key mapping function.

ZooNet [19] is a client approach at improving the performance of WAN coordination. By deploying multiple ZooKeeper services in different regions with observers in every other region, it tries to achieve fast reads at the expense of slow writes and data-staleness. In other words, the system operates 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. ZooNet does not support load adaptive object ownership migration.

Supercloud [27] takes a different approach to handling diurnal patterns in the workload. Instead of making end systems adjustable to access patterns, Supercloud provides the solution to move non-adjustable components to the places of most frequent use by using live-VM migration. One key advantage of Supercloud is its ability to migrate not only standalone systems that reside in a single VM, but also interconnected distributed applications without breaking them.

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, WPaxos adapts dynamically to the changing access locality through adaptive object stealing. The object stealing mechanism also enables support for multi-object transactional updates: move all the needed objects to the same leader, before processing the multi-object update command at that leader. In future work, we will investigate the feasibility of this straightforward implementation and explore optimizations. We will also investigate more sophisticated object stealing strategies that better cater to the needs of the clients and that proactively move objects to zones with high demand.

References

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

[2] Amazon Inc. Elastic Compute Cloud, Nov. 2008.

[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/`.

[8] Gmach, D., Rolia, J., Cherkasova, L., and Kemper, A. Workload analysis and demand prediction of enterprise data center applications. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on* (2007), IEEE, pp. 171–180.

[9] GRIMSHAW, A., MORGAN, M., AND KALYANARAMAN, A. Gffs – the XSEDE global federated file system. *Parallel Processing Letters 23*, 02 (2013), 1340005.

[10] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[11] HOCH, E. N., BEN-YEHUDA, Y., LEWIS, N., AND VIGDER, A. Bizur: A Key-value Consensus Algorithm for Scalable File-systems.

[12] HOWARD, H., MALKHI, D., AND SPIEGELMAN, A. Flexible Paxos: Quorum intersection revisited.

[13] HUNT, P., KONAR, M., JUNQUEIRA, F., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC* (2010), vol. 10.

[14] 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.

[15] 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.

[16] LAMPORT, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 872–923.

[17] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS) 16*, 2 (1998), 133–169.

[18] LAMPORT, L. Generalized consensus and paxos. Tech. rep., Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[19] 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).

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] PELUSO, S., TURCU, A., PALMIERI, R., LOSA, G., AND RAVINDRAN, B. Making fast consensus generally faster. *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016* (2016), 156–167.

[26] 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.

[27] SHEN, Z., JIA, Q., SELA, G.-E., RAINERO, B., SONG, W., VAN RENESSE, R., AND WEATHERSPOON, H. Follow the sun through the clouds: Application migration for geographically shifting workloads. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (2016), ACM, pp. 141–154.

[28] VAN RENESSE, R., AND ALTINBUKEN, D. Paxos made moderately complex. *ACM Computing Surveys (CSUR) 47*, 3 (2015), 42.

[29] XIE, C., SU, C., KAPRITSOS, M., WANG, Y., YAGHMAZADEH, N., ALVISI, L., AND MAHAJAN, P. Salt: Combining acid and base in a distributed database. In *OSDI* (2014), vol. 14, pp. 495–509.

[30] ZHANG, G., CHIU, L., AND LIU, L. Adaptive data migration in multi-tiered storage based cloud environment. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on* (2010), IEEE, pp. 148–155.