Abstract

We present \textit{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).

\textit{An archipelago is a chain, cluster or collection of islands. https://en.wikipedia.org/wiki/Archipelago}

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 dependance 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 3/4ths of the acceptors\(^1\), 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

\(^1\)For 2\(F\) + 1 cluster, fast-quorum is \(F + \lceil \frac{F+1}{2} \rceil\)
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 (Q1s) intersect with phase-2 quorums (Q2s). 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 Q1s are chosen from rows and Q2s 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 Q2s 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

\(^2\)Our implementation will be made available as an opensource project on https://github.com/ailidani/paxi
Table 1: Terminology used in this work

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

(a) Grid quorums with $Q_1$s in rows and $Q_2$s in columns  
(b) WPaxos quorum with 2 nodes per region in $Q_1$

Figure 1: Grid and WPaxos quorums. (a) Regular grid quorum. (b) WPaxos quorum with one possible $Q_1$ of 2 nodes per region.

Figure 2: Normal case messaging flow of WPaxos

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 $Q_2$. 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 $Q_1$ quorum. Successful completion of phase-1 transitions the protocol into phase-2, which in turn will run on a local $Q_2$ 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 when 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.

(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 OpZ2 is issued in Z2; (b) initial leader is elected in the zone of the first request; (c) heavy traffic OpZ3 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 \( L_n \) detects that the object \( X \) has more requests coming from a remote zone, it will initiate the object handover by communicating with the node \( L_m \), and in its turn \( L_m \) will start the phase-1 protocol to steal the leadership of the object.

### 2.5 Minitransactions

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 minitransactions [1]. Minitransactions 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 minitransaction 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 minitransaction 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 updates the states of its variables when processing the incoming messages, and eventually commits and executes commands.

Our algorithms 1-3, while algorithms 4 and 5 cover phase-2.

#### 3.1 Initialization

<table>
<thead>
<tr>
<th>Node ( \alpha ) Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( \text{function} \ INIT(\alpha) )</td>
</tr>
<tr>
<td>2: ( \beta \leftarrow \emptyset ) ( \triangleright ) No known ballot numbers</td>
</tr>
<tr>
<td>3: ( \Pi \leftarrow \emptyset ) ( \triangleright ) Phase-1 requests initially empty</td>
</tr>
<tr>
<td>4: ( \forall \beta \in \text{peers} : s[\beta] \leftarrow -1 ) ( \triangleright ) Initial slot numbers</td>
</tr>
<tr>
<td>5: ( \forall \beta \in \text{peers} : \Sigma[\beta] \leftarrow \emptyset ) ( \triangleright ) Phase-2 instances initially empty</td>
</tr>
<tr>
<td>6: ( I \leftarrow \emptyset ) ( \triangleright ) No known index</td>
</tr>
<tr>
<td>7: ( H \leftarrow \emptyset ) ( \triangleright ) Empty access history</td>
</tr>
<tr>
<td>8: ( \forall o \in \Omega : I[o] \leftarrow \alpha ) ( \triangleright ) Initial index</td>
</tr>
<tr>
<td>9: ( \forall o \in \Omega : b[o] \leftarrow (1 \cdot \alpha) ) ( \triangleright ) Initial Ballot</td>
</tr>
</tbody>
</table>

The \( \text{INIT}(\alpha) \) 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 \( \kappa \) sending a \( \text{request}, \kappa, \gamma \) message to one of the nodes in the system. Client typically chooses a local zone node to minimize the initial communication costs. The \( \text{request} \) message includes the command \( \gamma \), containing some object \( o \) on which the command needs to be executed. Upon receiving the command from the client, a node \( \alpha \) checks if the
Algorithm 1 Node $\alpha$: client request handler

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

Algorithm 2 Node $\alpha$: prepare message handler

1: function HANDLE((prepareReply, $o$, $b$)) from $\beta$ \\
2:     $\lambda \leftarrow I[\alpha]$ \\
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:         $b'[\alpha] \leftarrow \max\{b[o], b\}$ $\triangleright$ Update ballot of object $o$ \\
7:         SENDTO($\beta$, (prepareReply, $o$, ok $\leftarrow$ true, $b'[\alpha], s', \delta$)) \\
8:     else \\
9:         if $b > b'[\alpha]$ then $\triangleright$ Accept only if higher ballot \\
10:            if $o \in \Pi$ then $\triangleright$ If there is outstanding phase 1 \\
11:                $r \in \Pi[\alpha]$ : Retry request $r$ after random time \\
12:        $\Pi \leftarrow \Pi \setminus o$ \\
13:         $b'[\alpha] \leftarrow b$ \\
14:         $\Pi[\alpha] \leftarrow \beta$ \\
15:         $\Pi[\alpha] \leftarrow \beta$ $\triangleright$ New leader \\
16:         if $f \neq b'[\alpha]$ then $\triangleright$ Reject \\
17:             END \\
18:             SENDTO($\beta$, (prepareReply, $o$, ok $\leftarrow$ false, $(\lambda, b[o])$))

object exists in the index $I$, 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 $s_\alpha$ (lines 30-33). However, if the object is found to be managed by some other remote leader $\lambda$, depending on the configuration, $\alpha$ will either forward the request to $\lambda$ (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 incoming prepare message sent during phase-1 initiation. The node $\alpha$ can accept the sender node $\beta$ 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 $\alpha$ is aware of (lines 9-15). In the second case, node $\alpha$ 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 $\alpha$ 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 $\beta$ as the leader $\lambda$ for object $o$, and sends back the ballot number that has caused the rejection (line 17).

Algorithm 3 Node $\alpha$: prepareReply message handler

1: function HANDLE((prepareReply, $o$, $\alpha$, $\delta$, $(\lambda, b[o])$)) from $\beta$ \\
2:     if $o \notin \Pi \setminus b < b[o]$ then $\triangleright$ Ignore old reply msg \\
3:         return \\
4:     if $\alpha$ $\triangleright$ Acked \\
5:         $\Pi[\alpha] \leftarrow Q1.ACK(\beta)$ \\
6:         $\Pi[\alpha] \leftarrow \Pi[\alpha] \cup (s, \delta)$ \\
7:         if $Q1.SATISFIED$ then \\
8:             $\Pi[\alpha] \leftarrow \Pi[\alpha]$ \\
9:             HANDLE($\{r : \forall r \in \Pi[\alpha]\}$) $\triangleright$ Process all pending requests \\
10:        $\Pi \leftarrow \Pi \setminus o$ \\
11:     else $\triangleright$ Handle reject message \\
12:        $\Pi[\alpha] \leftarrow \lambda$ $\triangleright$ Update index \\
13:        $b[o] \leftarrow \max\{b[o], b\}$ $\triangleright$ Update ballot \\
14:        Retry $\{r : \forall r \in \Pi[\alpha]\}$ 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 $Q^2$ quorum residing in a single zone, thus all inter-node communications are kept local to the zone, greatly reducing the latency.

#### Algorithm 4: accept message handler

```
function HANDLE((accept, γ, b, s)) from β
if γ ∉ I then
    Π[γ]b ← β
else if β = λ ∨ b ≥ Σ[β]b then
    Π[β]b ← β
else if Σ[β]b = ⊥ then
    Σ[β]b ← instance, γ, b
else
    Π[β]b ← max(β, Σ[β]b)
if b > Σ[β]b then
    Π[β]b ← b
sendTo(β, (acceptReply, ok ← true, o, λ, b, s))
else
    sendTo(β, (acceptReply, ok ← false, o, λ, Σ[β]b, s))
```

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 (accept) 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 $Q^2$ 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: acceptReply message handler

```
function HANDLE((acceptReply, ok, o, λ, b, s)) from β
if ok then
    if b < Σ[β]|b|, Σ[β]|s| is committed then
        return ▷ Ignore old reply
    Σ[β]|s|, Λ(b, o, s, γ)
else
    if b > b[o] then
        Π[o] = λ
    b[o] = b
if Σ[β]|s| ⊥ ∧ is not committed then
    Π[β]|s|, back to main request queue
```

### 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. $∀σ : committed[α] = ⊥ • σ$. 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. $∀σ : committed[α] = γ$ at any $t \implies committed[α] = γ • σ$ at $t + Δ$.

**Consistency.** For any leader $α$, if command $γ$ is committed at instance $O_α$, 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. $∀γ : γ \in committed[α]$.

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 instances located at four AWS regions, namely: California (CA), Virginia (VA), Ireland (EU), and Japan (JP). Each AWS region corresponds to a single WPaxos zone.

---

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

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 $\mu$ can be varied for different zones to control the locality, and $\sigma$ 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: $l = 1 - \overline{OVL}$.

Let $\Phi((x - \mu) / \sigma)$ denote the cumulative distribution function (CDF) of any normal distribution with mean $\mu$ and deviation $\sigma$, 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 $Q_1$ and $Q_2$ in two types of quorums, Flexible Grid (FG) and Flexible 2 Rows (F2R). Flexible grid quorum uses a single node per region for $Q_1$, while F2R is our chosen WPaxos quorum approach as described in Section 2. Clients in each region simultaneously generate

---

$^4$The overlapping coefficient (OVL) is a measurement of similarity between two probability distributions, refers to the shadowing area under two probability density functions simultaneously [12].
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 $Q_1$ 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 struggler 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 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
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 $Q_1$ and $Q_2$ 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 $Q_2$ quorum at that zone and $Q_1$ 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 $Q_1$ and $Q_2$ quorums.

A zone failure will make forming the $Q_1$ 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 $Q_1$ 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


