Abstract

We present WPaxos, a multileader wide area network (WAN) Paxos protocol, that achieves low-latency high-throughput consensus across WAN deployments. WPaxos dynamically partitions the object space across multiple concurrent leaders that are deployed strategically using flexible quorums. Per zone and per object-space leadership 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, 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 minitransactions.

We implemented and evaluated WPaxos across WAN deployments using the benchmarks introduced in the EPaxos work. Our results show that WPaxos achieves up to 30 times smaller median request latency than EPaxos due to the reduction in WAN communication.

1 Introduction

Paxos, introduced in 1989 [10], 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 asynchronous and concurrent 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 condition (i.e., nodes decide on suitable decision value as a function of the inputs).

Since Paxos provides an elegant and uniform solution to the distributed consensus problem, it has been adopted widely. There have been many implementations/adaptations of the Paxos protocol, including Google Chubby [4], Apache ZooKeeper [9], and recently Raft [16]. 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 should be forwarded to that leader, and that leader commits the requests by performing the second phase of Paxos with the acceptors. Therefore, 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 clusters and datacenters) has gained greater importance. WAN coordination has become important for database applications and NewSQL datastores [3, 5] and distributed filesystems [6, 14, 17]. WAN coordination is also important for consistent ordering for social network metadata updates.

In order to eliminate the single leader bottleneck, EPaxos [15] proposes a leaderless Paxos protocol where any replica at any site can propose and commit commands opportunistically provided the commands are non-interfering. However, this opportunistic commit protocol requires 3/4ths of the Paxos acceptors to agree, 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; this time agreement is expected from majority of the Paxos acceptors.

Another way to eliminate the single leader bottleneck is to use a separate Paxos group deployed at each zone/site. Google Spanner [5] achieves this via a static partitioning of the global keyspace to smaller keyspaces and assigning each such keyspace to a zone/site. However, such a static partitioning is inflexible and WAN latencies will be incurred persistently to access/update an
object in a keyspace that is mapped to a different site. Finally in order to perform transactions involving objects in different keyspaces a separate 2PC mechanism 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 WAN deployments.

While WPaxos uses multiple concurrent leaders, each responsible for a keyspace, it differs from the existing solutions, because it manages the keyspaces in a dynamic fashion. The concurrent leaders can steal keys from each others’ keyspaces using phase-1 of Paxos. The commit decisions for updates in a keyspace are fast as the phase-2 acceptors are located at the same zone as the zone-leader. Finally transactions across keyspaces (such as a consistent read of multiple objects across various keyspaces) are implemented efficiently in one round-trip-time via the key-stealing mechanism using phase-1 of Paxos, and there is no need for a separate 2PC protocol across zone-leaders.

To achieve communication-efficient WAN coordination, WPaxos adapts the “flexible quorums” idea (which was introduced in 2016 summer as part of EPaxos [8]). WPaxos adopts the flexible quorums rule in a novel manner for deploying multiple concurrent leaders across the WAN strategically. Unlike the single-leader FPaxos protocol, WPaxos uses a multi-leader Paxos protocol. With its multileader protocol, WPaxos achieves the same consistency guarantees as EPaxos: Linearizability is ensured within a keyspace, and serializability and causal-consistency is ensured across keyspaces. We present how these are achieved in Section 2.1, Section 2.2, and Section 3.

We implemented WPaxos and performed evaluations across WAN deployments using the evaluation benchmarks introduced in the EPaxos work. Our results in Section 4 show that WPaxos significantly outperforms EPaxos. WPaxos achieves up to 30 times smaller median request latency than EPaxos. This is because, while the EPaxos opportunistic commit protocol requires 3/4ths of the Paxos acceptors to agree and incurs almost one WAN round-trip latency, WPaxos is able to achieve site-local-latency Paxos commits using the site-local phase-2 acceptors.

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

**2.1 WPaxos Quorums**

WPaxos relies on the flexible quorums. This simple and surprising result states that we can weaken the Paxos requirement that “all quorums intersect” to require that “only quorums from different phases intersect”. In other words, 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 WPaxos quorum idea derives from the grid quorum layout, shown in figure 1a, in which rows and columns act as Q1 and Q2 quorums respectively. Consider all the acceptors at all zones/sites form a grid, then Q2 sets are columns in the acceptor grid, and Q1 sets are rows in the acceptor grid. 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.

WPaxos relaxes some of the grid quorum constraints to get a more fault-tolerant and better performing alternative. Our quorums no longer use rigid rows for selecting Q1, instead we pick nodes from each column, no matter their row position. Q2s are still constrained to the same column as in the traditional grid layout. In our protocol, each Q2 is tightly packed within the same datacenter in order to reduce the communication overheads, making

\footnote{Our implementation will be made available as an opensource project on https://github.com/ailidani/paxi}
phase-2 of the protocol operate locally with no need for WAN message exchange.

A typical WPaxos Q1 quorum will consist of 2 nodes in each region, while Q2 quorum is 2 nodes within the region. We call such quorum flexible two rows as Q1 has doubled from single row. Increasing the size of Q1 has negligible effect on the performance, as we show in the evaluation, but allows us to better handle node failures within the region, as 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 struggler will not penalize the phase-2 progress.

2.2 WPaxos Protocol Overview

WPaxos takes advantage of flexible quorums, but unlike FPaxos it no longer has a single leader. Multiple leaders are elected as needed through the Q1 quorums; WPaxos can have as many distinct leaders as there are nodes in the system. Each leader is then responsible for a subset of all keys, referred as the dynamic keyspace. Independent and non-overlapping keyspaces allow WPaxos to process many keys in parallel under different leaders and on different Q2 quorums.

Basic WPaxos protocol is broken down in two distinct phases that each phase operating on its own quorum. Phase-1 of the protocol, or a key-stealing phase is responsible for moving the ownership of the object between dynamic keyspaces, while phase-2 replicates the object requests on some Q2. Phase-2 can be repeated multiple times until the object migration is needed again. Figure 2 shows the normal operation in both phases, and also references each operation to the algorithms in Section 3. Each replica is identified by two numbers: a site ID and node ID.

The phase-1 of the protocol starts in two cases: a client has a request for a brand new object that is not in WPaxos or replicas needs to steal an object from some remote dynamic keyspace. This phase of the algorithm is very similar to regular Paxos phase-1, except it is being operated on some global Q1 quorum.

All objects under the same dynamic keyspace are linearizable with respect to each other for the time the objects stay under the same leadership. The leader keeps the same ballot number for all of its items, and consequently phase-1 causes the ballot number to increase: the new leader for the object must prepare with a ballot number higher than its current number and higher than a ballot number of a leader currently holding the object.

Successful completion of phase-1 transitions the protocol into phase-2, which in turn will run on a local Q2 quorum. This stage is used to decide the operations on the particular object. WPaxos repeats phase-2 multiple times, incrementing the slot number on each iteration.

2.3 Immediate Object Stealing

Since objects in WPaxos are dynamically partitioned to different keyspaces in various regions, there will be use cases when a client needs to operate on some remote object. WPaxos protocol makes this operation transparent for the client, as client only contacts one of the local replicas with such remote request, instead of reaching out across datacenters. WPaxos replica, however, needs to deal with such request in a special manner, because it cannot process it locally: it needs to steal the object from the current leader in order to carry out the request. Replica consults its internal cache to determine the ballot number used for the object and starts the WPaxos phase-1 on some Q1 quorum with a larger ballot. Key stealing will be successful if the local replica is able to out-ballot the existing leader, and normally this is achieved in just one phase-1 round, given 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 of the old leader. This is true even when the old leader was not in the Q1 when the key was stolen, because the intersected replica in Q2 will reject the object operations with the new ballot.

In some cases, however, object stealing can be slowed
down by competing leaders trying to outbid each other. When a local replica is trying to acquire the ownership of the object, it proposes with a ballot number higher than the remotes, but the remote replica may be doing the same in order to get hold of some other object from the local replica, as illustrated in figure 3a. This condition puts the two leaders into a loop trying to out-ballot each other, potentially stalling the system for many phase-1 cycles. We mitigate the problem by making the ballot number a tuple of counter and replica ID so that conflict is resolved by replica ID in case the counters are the same (figure 3b).

2.4 Locality Adaptive Object Stealing

The basic protocol will migrate the object from a remote region to a local region upon the first request. However, such simple policy works well only when the records exhibit very good locality of reference. Unfortunately, this approach may cause a severe performance degradation once the object is frequently needed in more than one region and as the object is traveling back-and-forth between the regions, it undergoes costly migration phase, degrading the latency and throughput as perceived by the objects clients.

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

Our majority-region migration policy aims to improve the locality of reference by transferring the objects to regions sending out the highest number of requests for these 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 \( L_o \) with a dynamic keyspace \( d \) detects that the object \( X \in d \) has more requests coming from a remote region, it will initiate the object handover by communicating with the server \( L_n \) that will start the phase-1 protocol to steal the leadership of the object.

2.5 Minitransactions

Linearizability guarantees provided to objects within the same keyspace combined with the ability for a replica to steal objects enable WPaxos to support sinfonia-style minitransactions [1]. Minitransactions are achieved through aggregating all objects involved in the transaction under the leadership of a single replica.

This transaction process starts with a client sending a minitransaction request to a replica it already believes to have the most objects needed in the transaction. In its turn, the replica will steal the missing objects and once all objects are collected in the same keyspace, the minitransaction can proceed as a single command on all of the objects. This approach, however, is not without a penalty for overall system performance, since stealing objects disrupts the locality balancing.
3 Algorithm

WPaxos is a WAN coordination protocol that designed to run on nodes located in different datacenters. Each WPaxos replica is a deterministic state machine that maintains a set of variables and an internal data-store. The protocol updates the states of the variables when processing the incoming messages, and eventually commits and executes a sequence of commands against the data store. For every dynamic keyspace \( \mathcal{K} \) there is an unbounded sequence of instances identified by an increasing slot number \( s \). A keyspace \( \mathcal{K} \) provides linearizability for all objects it holds and every replica \( \alpha \) leads its own keyspace \( \mathcal{K}_\alpha \). Replicas also have a cache of a complete state of all other keyspaces in the system. Each replica \( \alpha \) keeps an index \( I \), mapping each object \((key)\) to the keyspace leader that \( \alpha \) knows of, i.e. \( I[key] = \mathcal{K}_\lambda \). When replica tries to acquire a leadership of a new object, it adds the object and all following corresponding proposals into the set \( W \) until the phase-1 of protocol completes. A map of ballot numbers \( b \) for all peers is maintained at every replica to ensure correctness: only strictly greater ballot number \( b \) than all involved replicas’ local records is accepted. Finally, each replica keeps a history of access for objects, and a policy to enable dynamic adaptation for keyspaces. A summary of WPaxos notation is given as following:

- \( \lambda, \alpha, \beta \): Replicas
- \( \kappa \): Client
- \( \gamma, \delta \): Commands
- \( b_{\alpha} \): Ballot number of replica \( \alpha \)
- \( s_{\alpha} \): Slot number of replica \( \alpha \)
- \( \mathcal{K}_\lambda \): Keyspace leading by replica \( \lambda \)
- \( I[key] \): Index, mapping objects to keyspace leader
- \( W[key] \): Waiting set of phase-1 requests
- \( H \): Access history
- \( \sigma \): Sequence of commands
- \( \cdot \): Concatenation
- \( \{type, \gamma, \lambda, b_{\lambda}\} \): General message format.

In original Paxos algorithm, acceptors only maintain single values for the accepted ballot number and the next slot number. WPaxos, however, extends both variables to a vector. As a result, it is crucial for the correctness of the algorithm that all operations on these vectors are atomic. The same restriction also applies to the index \( I \).

Algorithms 1-6 show the operations of a WPaxos replica. Phase-1 of the protocol is described in the algorithms 1-3, while algorithms 4 and 5 cover phase-2. Algorithm 6 illustrates the operation performed on the keyspace \( \mathcal{K} \) during phase-2.

3.1 Phase-0: Initialize

Replica \( \alpha \) Initialization

```
1: function \text{INIT}(\{keys\})
2: \( b_{\alpha} \leftarrow (0 \cdot \alpha) \) \text{ } \triangleright \text{ Construct initial ballot}
3: \( \forall \beta \in \text{peers} : b[\beta] \leftarrow 0 \) \text{ } \triangleright \text{ Set peers ballot}
4: \( \forall \beta \in \text{peers} : s[\beta] \leftarrow -1 \) \text{ } \triangleright \text{ Set slot numbers}
5: \( K_\alpha \leftarrow \text{keys} \) \text{ } \triangleright \text{ Add initial keys to local keyspace}
6: \( \forall k \in \text{\{keys\}} : I[k] \leftarrow \alpha \) \text{ } \triangleright \text{ Initial index}
7: \( W \leftarrow \emptyset \) \text{ } \triangleright \text{ Empty phase-1 waiting set}
8: \( H \leftarrow \emptyset \) \text{ } \triangleright \text{ Empty access history}
```

The \text{INIT} function describes the state initialization of any replica before it becomes active. The ballot number of a replica is tuple consisting of a counter and replica ID. Initially the ballot has a counter set to 0 (line 2), and all peers’ ballot numbers are set to 0 (line 3). Replica only completes. A map of ballot numbers \( b \) for all peers is represented as the last position of a proposed instance, which is initialized to -1 for every peer (line 4). The initial keys are optional and can be assigned through a user defined mapping or start up parameters. Any existing initial keys are assigned to local keyspace (line 5) and index (line 6) accordingly. The waiting set \( W \) in phase-1 and access history statistics \( H \) are empty sets at this point (line 7-8).

3.2 Phase-1: Prepare

Algorithm 1 Replica \( \alpha \): client proposal handler

```
1: function \text{RECEIVE}(\text{proposal} \gamma, \kappa) \text{ } \triangleright \text{ Unknown key}
2: if \( \gamma.key \notin I \) then \( \triangleright \alpha \text{ is leader of } \gamma.key \)
3: \( \text{SCOUT}(\gamma) \) \text{ } \triangleright \text{ Phase 1}
4: \text{return}
5: \( \lambda \leftarrow I[\gamma.key] \)
6: if \( \alpha = \lambda \) then \( \triangleright \alpha \text{ is leader of } \gamma.key \)
7: \( q_2 \leftarrow \text{NEWQUORUM}(Q2) \)
8: \( q_2\text{.ACK}(\alpha) \) \text{ } \triangleright \text{ Start phase 2}
9: \( s_{\alpha}++ \) \text{ } \triangleright \text{ Increase slot number}
10: \( K_\alpha \cdot \log(s_{\alpha}) = \{\text{instance} \gamma.b_{\alpha}q_2\} \)
11: \( \text{MULTICAST}(\text{accept, } \gamma, \alpha, s_{\alpha}, b_{\alpha}) \) \text{ } \triangleright \text{ Save to access history } H \)
12: if \( H \) triggers migration event then
13: \( H \leftarrow H \cup \{\gamma, \kappa\} \) \text{ } \triangleright \text{ Broadcast to } \gamma, \kappa \)
14: \( \text{SEND}(\lambda, \{\text{proposal, } \kappa, \gamma\}) \) \text{ } \triangleright \text{ Forward to replica } \lambda \)
15: else
16: if Immediate key migration then
17: \( b_\alpha \leftarrow (\text{max}(b_{\alpha}, b_{\kappa}) + 1) \cdot \alpha \)
18: \( \text{SCOUT}(\gamma) \)
19: else \( \triangleright \text{ Adaptive key migration} \)
20: \( \text{SEND}(\lambda, \{\text{proposal, } \kappa, \gamma\}) \) \text{ } \triangleright \text{ Forward to replica } \lambda \)
```

21: function \text{SCOUT}(\gamma)
22: if \( \gamma.key \notin W \) then
23: \( W \leftarrow W \cup \{\gamma\} \)
24: \text{return}
25: \( \text{SEND}(\lambda, \{\text{prepare, } \gamma, \alpha, b_{\alpha}\}) \) \text{ } \triangleright \text{ Start phase 1}
26: \( \text{BROADCAST}(\text{prepare, } \gamma, \alpha, b_{\alpha}) \) \text{ } \triangleright \text{ Start phase 1}
Algorithm 1 describes the initial stage of each new request coming to the system from the client. A client sends \((\text{prepare}, \kappa, \gamma)\) message to the replica, where command \(\gamma = \langle p, \text{key} \rangle\) defines the operation applies to the object key. At this point replica checks if such key exists in its index, and starts phase-1 for any unindexed keys by \(\text{BROADCAST} \langle \text{prepare}, \text{key}, \alpha, b_\alpha \rangle\) message (line 2-4). If the key is known to belong to replica \(\alpha\), then the replica initiates phase-2 of the protocol by \(\text{MULTICAST} \langle \text{accept}, \alpha, b_\alpha, s_\alpha \rangle\) message to its \(Q_2\) quorum, and a new instance is created for slot \(s_\alpha\) (line 6-11).

However, if the key is found to be managed by some remote replica \(\lambda\), \(\alpha\) will either forward the request to \(\lambda\) (line 20), or start immediate key-migration with larger ballot in phase-1, depends on system configuration (line 16-18).

In line 12, the current leader will save the access event to local history \(H\), and trigger key migration if the amount of remote site accesses exceed some threshold. Such threshold can be defined by any adaptive migration policy, such as the percentage of requests in the past period of time or the number of consecutive accesses from the same site.

Algorithm 2 Replica \(\alpha\): prepare message handler

```
1: function HANDLE(\langle \text{prepare}, \text{key}, \beta, b_\beta \rangle)
2:   α ← I[\text{key}]
3:   if \(\beta = \lambda\) then
4:     \(b_\lambda \leftarrow \max(b_\lambda, b_\beta)\) \hspace{1em} \triangleright \text{Update ballot of replica } \lambda
5:     SENDTO(\beta, \langle \text{prepareReply}, \text{ok} \leftarrow \text{true, key, } \lambda, b_\lambda \rangle)
6:   else \hspace{1em} \triangleright \text{New leader}
7:     if \(b > b_\lambda\) then \hspace{1em} \triangleright \text{Accept only if higher ballot}
8:       \(K_\lambda \leftarrow K_\lambda \setminus \text{key}\) \hspace{1em} \triangleright \text{Remove key from old keyspace}
9:       \(b_\lambda \leftarrow \max(b_\lambda, b_\beta)\)
10:      r.SENDTO(\beta, \langle \text{prepareReply}, \text{ok} \leftarrow \text{true, key, } \beta, b_\beta \rangle)
11:     else \hspace{1em} \triangleright \text{Reject}
12:      r.SENDTO(\beta, \langle \text{prepareReply}, \text{ok} \leftarrow \text{false, key, } \lambda, b_\lambda \rangle)
```

The HANDLE routine of algorithm 2 is responsible for processing the incoming \(\langle \text{prepare}, \text{key}, \beta, b_\beta \rangle\) messages send during the phase-1 initiation. The prepare message will be accepted in one of two cases: the sender of the prepare is the same node as in replicas index \(I_0\) (line 3-5) or senders ballot number \(b_\beta\) is greater than the ballot number \(b_\lambda\) we have in our cache (line 7-10). Otherwise, it will be rejected with local record of \(\lambda\) (line 12).

Algorithm 3 Replica \(\alpha\): prepareReply message handler

```
1: function HANDLE(\langle \text{prepareReply}, \text{ok, key, } \beta, b_\beta \rangle)
2: q1 ← W[\text{key}] \hspace{1em} \triangleright \text{Get quorum 1}
3: if \(q1 = ⊥\) then
4:   return \hspace{1em} \triangleright \text{Ignore old reply msg}
5: if \(\text{ok}\) then
6:   q1.ACK(\beta)
7: if q1.SATISFIED then
8:   \(q1 \leftarrow \perp\)
9:   \(K_\alpha \leftarrow K_\alpha \cup \{\text{key}\}\) \hspace{1em} \triangleright \text{Add key to local keyspace}
10: \text{HANDLE(}\{p : \forall p \in W[\text{key}]\}\) \hspace{1em} \triangleright \text{Process all pending proposals}
11: \text{else} \hspace{1em} \triangleright \text{Reject message}
12: \(K_\beta \leftarrow K_\beta \cup \{\text{key}\}\) \hspace{1em} \triangleright \text{Updates keyspace}
13: \(b_\delta \leftarrow \max(b_\beta, b_\delta)\) \hspace{1em} \triangleright \text{Update ballot}
14: \(I[\text{key}] \leftarrow \beta\) \hspace{1em} \triangleright \text{Update index}
15: \(q1 \leftarrow \perp\) \hspace{1em} \triangleright \text{End phase 1 of key}
16: \text{Retry } \{p : \forall p \in W[\text{key}]\}
17: W ← W \setminus \{\text{key}\}
```

Algorithm 3’s HANDLE function collects the prepare replies sent by the algorithm 2, and checks if the Q1 quorum is satisfied, at which point the key is removed from the waiting set \(W\) and added to the keyspace \(K_\alpha\) (line 6-9). The related pending proposals in waiting set will be processed to start phase-2 (line 10). If the phase-1 is rejected, the local caches for keyspace state and index are updated with new information (line 12-17), and the pending proposals in such phase-1 are retried by pushing them back to the main request queue (line 16).

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 key. The replication can be repeated many times until the object needs to change its keyspace and migrate to a different leader. WPaxos carries out this phase on the \(Q_2\) quorum residing in a single datacenter, thus all inter-replica communications are kept local, greatly reducing the latency of handling the proposal requests from the client.

Algorithm 4 Replica \(\alpha\): accept message handler

```
1: function HANDLE(\langle \text{accept}, \gamma, \beta, b_\beta, s_\beta \rangle)
2: if \(\gamma \notin I\) then
3: \(I[\text{key}] \leftarrow \beta\)
4: \(b_\beta \leftarrow \max(b_\beta, b_\delta)\)
5: \(\lambda \leftarrow I[\text{key}]
6: if \(\beta = \lambda\) then \hspace{1em} \triangleright \text{Known leader}
7: \(ok, b, δ \leftarrow K_\lambda.ACEPT(\langle γ, b, s_\delta \rangle)\)
8: \(\text{SENDTO}(\lambda, \langle \text{acceptReply}, \text{ok, } \alpha, \lambda, b, δ \rangle)\)
9: \text{else} \hspace{1em} \triangleright \text{New leader}
10: if \(b > b_\lambda\) then \hspace{1em} \triangleright \text{Accept higher ballot}
11: \(ok, b, δ \leftarrow K_\beta.ACEPT(\langle γ, b, s_\delta \rangle)\)
12: \(\text{SENDTO}(\beta, \langle \text{acceptReply}, \text{ok, } \alpha, \lambda, b, δ \rangle)\)
13: \text{else} \hspace{1em} \triangleright \text{Reject}
14: \(r.SENDTO(\lambda, \langle \text{acceptReply}, \text{ok} \leftarrow \text{false, } \alpha, \lambda, b_\lambda \rangle)\)
```

The \text{acceptReply} routine of algorithm 2 is responsible for processing the incoming \(\langle \text{accept}, \gamma, \beta, b_\beta, s_\beta \rangle\) messages send during the phase-2 initiation. The prepare message will be accepted in one of two cases: the sender of the prepare is the same node as in replicas index \(I_0\) (line 3-5) or senders ballot number \(b_\beta\) is greater than the ballot number \(b_\lambda\) we have in our cache (line 7-10). Otherwise, it will be rejected with local record of \(\lambda\) (line 12).

Once the leader multicasts the accept message at the beginning of the phase-2, replicas must properly respond to the message. Algorithm 4 shows how replicas handle the \(\langle\text{accept}, \gamma, \beta, b, s, \delta\rangle\) message. In the normal case, replica will accept the message if it has no conflict with the proposed and known ballot number \(b\) (line 6-8). However, sometimes a node may be aware of a different leader for the key in which case the replica will either reject the request if it already adopted a same or higher ballot number for the keyspace (line 13-14) or accept if a sender has overtaken the ballot (line 10-12). The ACCEPT function of each keyspace (line 7,11) shares the logic of a normal acceptor in Multi-Paxos [11] where the instance of slot \(s\) is compared. We present keyspace’s pseudocode in Algorithm 6.

**Algorithm 5** Replica \(\alpha\): acceptReply message handler

1. function HANDLE((acceptReply, ok, \(\gamma, \beta, b, s, \delta\))
2.     if \(\beta = \alpha \land b < \text{ok}\) then
3.         committed, \(\delta \leftarrow K_{\alpha}.\text{ACK}(\gamma, \beta, b, s, \delta)\)
4.     if committed then
5.         \(\text{BROADCAST(commit, } \alpha, b, s, \delta)\)
6.     else
7.         \(\text{return false, } K_{\alpha}, b, K[s], \delta\)
8.         \(b_{1} \leftarrow \max(b_{1}, b)\)
9.         \(\delta \leftarrow b_{1}\)
10.        \(\text{return false, } \alpha, K[s], b\)
11.        \(\text{return false, } \alpha, K[s], b\), \(\bot\)

Leader replica collects the replies from its Q2 acceptors in Algorithm 5. The request proposal either gets committed when a sufficient number of successful replies is received (2-5), or aborted if some nodes reject the proposal (6-11). In case of rejection, leader replica also updates its cache with new keyspace and index information it has received from the rejecting replicas.

**Algorithm 6** Keyspace \(K\)

1. function ACCEPT(\(\gamma, b, s, \delta\))
2.     if \(b < K[s] . b\) then
3.         \(\text{return false, } K[s], b, K[s], \delta\)
4.         \(K[s] . b \leftarrow \max(K[s] . b, b)\)
5.         \(\text{if } K[s] . \delta \neq \bot\) then
6.         \(\text{return true, } K[s], b, K[s], \delta\)
7.         \(K[s] \leftarrow \gamma\)
8.         \(\text{return true, } K[s], b, \bot\)
9. function ACK(\(\gamma, \beta, b, s, \delta\))
10.     if \(K[s] . \text{committed} \land b < K[s] . b\) then
11.         \(\text{return false, } \bot\)
12.         \(K[s] . q_{2} . \text{ACK}(\beta)\)
13.         \(\text{if } K[s] . q_{2} . \text{Satisfied then}\)
14.         \(K[s] . \text{committed} \leftarrow \text{true}\)
15.         \(\text{return true, } K[s], \delta\)
16.         \(\text{return false, } \bot\)

Algorithm 6 presents the simple operations processed within a keyspace \(K\). The replicas can act on the keyspace in different ways, depending on the role played by replica in \(K\). This algorithm shares similar logic with normal Multi-Paxos algorithm.

### 3.4 Properties

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

- **Non-triviality** For any replica \(\alpha\), the set of committed commands is always a sequence \(\sigma\) of proposed commands, i.e. \(\exists \sigma : \text{committed}[\alpha] = \bot \land \sigma\).
- **Stability** For any replica \(\alpha\), the set of committed commands at any time is a prefix of the set at any later time, i.e. \(\exists \sigma : \text{committed}[\alpha] = \gamma\) at any \(t \implies \text{committed}[\alpha] = \gamma \land \sigma\) at \(t + \Delta\).
- **Consistency** For any keyspace \(K_{\alpha}\), if command \(\gamma\) is committed at instance \(K_{\alpha,i}\) by some replica, \(\gamma\) is or will be committed by every replica at the same instance.
- **Liveness** A proposed command \(\gamma\) will eventually be committed by all non-faulty replicas, i.e. \(\forall \alpha : \gamma \in \text{committed}[\alpha]\).

The formal proofs of above properties will be given in TL\(\alpha\)+ in future version of this work. In this paper, we outline the intuitive propositions that reveals more detailed properties of WPaxos protocol.

**Non-triviality** is straightforward since replica only start phase-1 or phase-2 for commands proposed by clients, in Algorithm 1.

**Definition 3.1.** Two commands \(\gamma\) and \(\delta\) interfere if there exists a sequence of commands \(\sigma\) s.t. \(\sigma \land \gamma \land \delta \neq \sigma \land \delta \land \gamma\).

The definition of interference can also be interpreted as interfering commands are commands targeting the same object.

**Lemma 3.1.** There are no interfering commands committed in two different keyspace at the same time.

**Lemma 3.2.** If interfering commands \(\gamma\) and \(\delta\) are committed, they will be executed in the same order by every replica.

### 4 Evaluation

We have implemented WPaxos on top of 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 have conducted our experiments in the system deployed on AWS [2] EC2 medium Linux instances\(^2\) located in three AWS regions, namely: Virginia

---

\(^2\)Two 64-bit virtual cores and 4GB memory.
Figure 5: Schematic representation of PAXI framework and WPaxos (VA), California (CA) and Ireland (EU). WPaxos implementation is discussed in section 4.1, we explain our experimental workloads in 4.2 followed by the evaluation results.

4.1 Implementation

Inspired by E Paxos’s experience, we have implemented a general framework, called PAXI, to accommodate various styles of Paxos algorithms. We have learned that E Paxos’s framework lacks the generality needed for our use as it does not provide features like site-aware node and grid or flexible quorums. Moreover, E Paxos leaves the bulk of networking to be implemented separately by all the different algorithms. We built WPaxos on top of the PAXI, and adopted original E Paxos code to work with our framework.

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-site, inter-site and peer-to-peer messaging for cross-replica traffic. Similarly to E Paxos, PAXI incorporates the mechanisms to facilitate the startup for the cluster and share initial parameters through the configuration management. The quorum management module built on top of the configuration allows for greater flexibility with quorums. Both replicas and clients use the API provided by the PAXI modules.

WPaxos protocol was placed on top of PAXI by adding the inter-replica message definition and message handling procedures. PAXI and WPaxos protocol were built in Go version 1.8 and they will be available as an opensource project on GitHub page https://github.com/ailidani/paxi.

4.2 Workload

PAXI provides a replicated key-value store as the state machine on top of the protocols under the 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 more practical and realistic workloads. Our experimental workloads reflect two primary parameters: conflict and locality. Conflict is expressed by a percentage of commands from multiple clients in different sites targeting the same set of keys, while the rest of the commands target the keys designated for each site. Locality is the measure of how often a particular region will be accessing a conflicting key.

We run each experiment for 5 minutes with a total of 1000 keys, with 500 shared, conflicting keys and 500 designated keys. The workload runs for a fixed period of time instead of fixed number of requests to avoid the situation in which one site may finish the workload much earlier when it owns the majority of keys, allowing another site to uncontestably steal the keys, resulting in the illusion of excellent performance. We perform the experiments with three replicas 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 standard WPaxos quorum approach. Clients in each site simultaneously generate the same number of phase-1 and phase-2 requests, and measure the commit latency for each phase. Figure 6 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 site, 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 an object stealing procedures. The most straightforward stealing routine will attempt to take the object from the remote datacenter 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 sites. Our adoptive object stealing procedure utilizes the request frequency metric
to control which site can steal the object and when. In this experiment we vary conflict ratio to reveal the impact of conflict on commit latency. We illustrate how two object stealing strategies behave under different degrees of object conflict.

The results of these experiments will be added in the future revisions of this work.

4.5 Latency

We compare the commit latency between WPaxos and EPaxos. Figure 7 shows the median (color bar) and 99th percentile (error bar) latency in three sites. EPaxos always have to pay the price of WAN communication, while WPaxos tries to keep as many operations locally. The ability of WPaxos to do so greatly depends on the locality of the conflicting keys. As a result, WPaxos reduce median latency by 96% at all conflict ratios. Even with 100% conflicts, WPaxos is able to reduce the median latency by exploring the intermediate locality among the shared keys. The 99th percentile latency of high conflicts (≥ 50%) indicates a small fraction of the slow commits, where EPaxos has to do 2RTT in a slow path, WPaxos is able to commit in one RTT.

4.6 Throughput

Similarly 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

The fault tolerance of the coordination service depends on when the failures happened and what server have been affected. The progress is still possible as long as we can form a valid $Q_1$ and $Q_2$ quorums. In our case, we need at least one running server in each region to form a $Q_1$ quorum and at least one fully operational region to make for a $Q_2$ quorum.

We show arrangement of flexible quorums that mask failure of one $Q_2$ acceptor. This is achieved by increasing $Q_1$ set to include two $Q_2$ nodes from each site.

Failures in $Q_2$ quorum cause the system to find another quorum to decide on the operations. Our system does not need to go through the first phase of Paxos to elect a new leader in case of such failure, as we can simple use acceptors from a different regions $Q_2$ quorum. During such operation, the leader no longer acts as an acceptor and acts only as the proposer, since it is not part of the active $Q_2$. Once the failed nodes participation in the system is restored and it joined all of its $Q_1$s, we can move the operation back to the local region. Theoretical performance of the system under such failure should be comparable to regular Multi-Paxos approach with the same number of nodes, as we will be paying a penalty of one WAN RTT to reach the acceptors in a different region.

$Q_1$ nodes failures have very little impact in the most common cases. Given that the leader election phase has succeeded and failed node is not the leader, the system can make progress as normal until a new leader is required. If failed node is the leader or the failure happened during the phase 1 of Paxos, we need to restart leader election on another available $Q_1$.

6 Related Work

Several attempts have been made for addressing consensus scalability. Certain systems, such as Mencius [13] 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 [15] is leaderless in a sense that any replica can 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 key, fast quorum will decide on the operation and replicate it across the system. However, if fast quorum detects a conflict in terms of another replica trying to decide on the operation for the same key, EPaxos will default to standard Paxos procedure. Similarly with Mencius, EPaxos reduces the bottleneck of a single leader, but it also provides for better WAN performance if deployed in such setting due to the fast quorum being smaller than the full quorum, thus not needing the replies from the furthest replicas.

Bizur [7] continues the trend of multileader consensus systems and it is similar in spirit to WPaxos. It 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 [12] 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 key-space partitioned across regions. ZooNet provides consistent reads by injecting sync requests when reading from remote region.

7 Concluding Remarks

WPaxos achieves fast wide area coordination by dynamically partitioning the objects across multiple leaders that are deployed strategically using flexible quorums. Per-zone and per-keyspace leadership 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 changing access locality not only speeds up the protocol, but also enables support for mini-transactions. 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


