WPaxos: Ruling the Archipelago with Fast Consensus

Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas
and Tevfik Kosar
University at Buffalo, SUNY
Email: {ailidani,acharapk,demirbas,tkosar}@buffalo.edu

***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 fast phase-2 commits, WPaxos adopts the flexible quorums idea in a novel manner, and appoints phase-2 acceptors to be close to their respective leaders. We implemented WPaxos and evaluated it on WAN deployments across 5 AWS regions. The dynamic partitioning of the object-space and emphasis on zone-local commits allow WPaxos to significantly outperform both partitioned Paxos deployments and leaderless Paxos approaches, while providing the same consistency guarantees.**

1. Introduction

Paxos [1] 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).

Paxos and its variants have been deployed widely, including in Chubby [2] based on Paxos [3], Apache ZooKeeper [4] based on Zab [5], and etcd [6] based on Raft [7]. All of these implementations depend on a centralized primary process(i.e., the leader) to serialize all commands. 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 [8]–[10], distributed filesystems [11]–[13], and social networks [14,15].

In order to eliminate the single leader bottleneck, EPaxos [16] 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. Systems like Google Spanner [8], ZooNet [17], Bizur [18] 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 multi-leader Paxos protocol that provides low-latency and high-throughput consensus across WAN deployments. WPaxos leverages the *flexible quorums* [19] idea to cut WAN communication costs. WPaxos deploys flexible quorums in a novel manner to appoint *multiple concurrent leaders* across the WAN. By strategically selecting the phase-2 acceptors to be close to the leader, WPaxos achieves fast 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 multi-leaders and partitions the object-space among these multi-leaders. Every node in WPaxos acts as a leader for a subset of 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. 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

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

commit update-requests on the object locally until the object is stolen by another leader.

With its multileader protocol, WPaxos achieves the same consistency guarantees as in EPaxos: linearizability is ensured per object, and when deployed with multi-object transactions, strict serializability is ensured across objects. We present safety and liveness properties of WPaxos in Section 3.4. We model WPaxos in TLA+/PlusCal [20] and present the algorithm using the PlusCal specification in Section 3. The consistency properties of WPaxos are verified by model checking this specification, which is available at http://github.com/ailidani/paxi/tree/master/tla.

Since object stealing is an integrated part of phase-1 of Paxos, WPaxos remains simple as a pure Paxos flavor and obviates the need for another service/protocol. There is no need for a configuration service for relocating objects to zones as in Spanner [8] and vertical Paxos [21]. Since the base WPaxos protocol guarantees safety to concurrency, asynchrony, and faults, the performance can be tuned orthogonally and aggressively, as we discuss in Section 4. To improve performance, we present a locality adaptive object stealing extension in Section 4.1. We also provide an embedded solution for transactions all inside the basic Paxos/WPaxos protocol in Section 4.3. This solution obviates the need for integrating a separate two-phase commit for transactions as in Spanner [8].

To quantify the performance benefits from WPaxos, we implemented WPaxos in Go (http://github.com/ailidani/paxi) and performed evaluations on WAN deployments across 5 AWS regions. Our results in Section 6 show that WPaxos outperforms EPaxos, achieving 2.4 times faster average request latency and 3.9 times faster median latency than EPaxos using a $\sim$70% access locality workload. Moreover, for a $\sim$90% access locality workload, WPaxos improves further and achieves 6 times faster average request latency and 59 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. Moreover, WPaxos is able to maintain low-latency responses under a heavy workload: Under 10,000 requests/sec, using a $\sim$70% access locality workload, WPaxos achieves 9 times faster average request latency and 54 times faster median latency than EPaxos. Finally, we evaluate 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 and present a reconfiguration protocol for WPaxos in Section 5. While WPaxos helps most for slashing WAN latencies, it is also suitable for intra-datacenter deployments for its high-availability and throughput benefits.

2. WPaxos

We assume a set of *nodes* communicating through message passing in an asynchronous environment. The nodes are deployed in a set of *zones*, which are the unit of availability isolation. Depending on the deployment, a zone can range from a cluster or datacenter to geographically isolated regions. Zones can be added to or removed from a running system through reconfigurations. We assume at most f nodes may crash in a zone of $N = 2f + 1$ nodes, and at most F zones may become unavailable out of a total Z zones. Each node is identified by a tuple consisting of a zone ID and node ID, i.e. $Nodes \triangleq 1..Z \times 1..N$.

Every node maintains a sequence of instances ordered by an increasing *slot* number. Every instance is committed with a *ballot* number. Each ballot has a unique leader. Similar to Paxos implementation [3], we construct the ballot number as lexicographically ordered pairs of an integer and its leader identifier, s.t. $Ballots \triangleq Nat \times Nodes$. Consequently, ballot numbers are unique and totally ordered, and any node can easily retrieve the id of the leader from a given ballot.

2.1. WPaxos Quorums

WPaxos leverages on the flexible quorums idea [19]. This 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 (Q_1) intersect with phase-2 quorums (Q_2). Flexible-Paxos, i.e., FPaxos, allows trading off Q_1 and Q_2 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 Q_2 at the expense of making the infrequently used Q_1 larger.

Definition 1. A quorum system over the set of nodes is *safe* if the quorums used in phase-1 and phase-2, named Q_1 and Q_2, intersect. That is, $\forall q_1 \in Q_1, q_2 \in Q_2 : q_1 \cap q_2 \neq \emptyset$.

WPaxos adopts the flexible quorum idea to WAN deployments. Our quorum system derives from the grid quorum layout, shown in Figure 1a, in which rows and columns act as Q_1 and Q_2 quorums respectively. An attractive property of this grid quorum arrangement is $Q_1 + Q_2$ does not need to exceed N, the total number of acceptors, in order to guarantee intersection of any Q_1 and Q_2. Let q_1, q_2 denote one specific instance in Q_1 and Q_2. Since $q_1 \in Q_1$ are chosen from rows and $q_2 \in Q_2$ are chosen from columns, any q_1 and q_2 are guaranteed to intersect even when $|q_1 + q_2| < N$.

In WPaxos quorums, each column represents a zone and acts as a unit of availability or geographical partitioning.

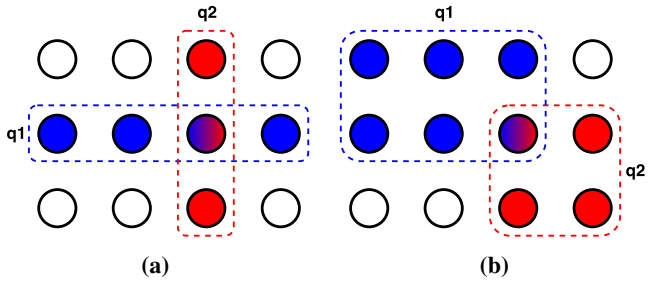


Figure 1: (a) Grid quorums with Q_1 in rows and Q_2 in columns (b) WPaxos quorum where $f = F = 1$

The collection of all zones form a grid. In this setup, we further relax the grid quorum constraints in both Q_1 and Q_2 to achieve a more fault-tolerant and efficient alternative. Instead of using rigid grid columns, WPaxos picks $f + 1$ (majority) nodes in a zone over $2f + 1$ nodes, regardless of their row position, to tolerate f crash failures in every zone. In addition, so as to tolerate F *zone failures* within Z zones, $q_1 \in Q_1$ is selected from $Z - F$ zones, and $q_2 \in Q_2$ from $F + 1$ zones. Figure 1b shows one such example of q_1 and q_2 intersecting at one node. In that deployment, each zone has 3 nodes, and each q_2 includes 2 out of 3 nodes from 2 zones. The q_1 quorum spans 3 out of 4 zones and includes any 2 nodes from each zone. Using a 2 row q_1 rather than 1 row q_1 has negligible effect on the performance, as we show in the evaluation. But then using a larger quorum allows us to better handle failures, as we discuss in Section 5.

Next, we formally define WPaxos quorums in TLA+ [20] and prove that our quorums always intersect.

$$\text{vertical} \triangleq \{q \in \textbf{SUBSET Nodes} : \wedge \forall i, j \in q : i[1] = j[1] \\ \wedge \text{Cardinality}(q) = f + 1\}$$

$$Q_1 \triangleq \{q \in \textbf{SUBSET Nodes} : {}^{2} \\ \wedge \text{Cardinality}(q) = (f + 1) \times (Z - F) \\ \wedge \text{Cardinality}(\{i[1] : i \in q\}) = Z - F \\ \wedge \text{Cardinality}(\{z \in \text{vertical} : z \subseteq q\}) = Z - F\}$$

$$Q_2 \triangleq \{q \in \textbf{SUBSET Nodes} : \\ \wedge \text{Cardinality}(q) = (f + 1) \times (F + 1) \\ \wedge \text{Cardinality}(\{i[1] : i \in q\}) = F + 1 \\ \wedge \text{Cardinality}(\{z \in \text{vertical} : z \subseteq q\}) = F + 1\}$$

Lemma 1. *WPaxos Q_1 and Q_2 quorums satisfy intersection requirement (Definition 1).*

Proof. (1) WPaxos q_1s involve $Z - F$ zones and q_2s involve $F + 1$ zones, since $Z - F + F + 1 = Z + 1 > Z$, there is at least one zone selected by both quorums. (2) Within the common zone, both q_1 and q_2 selects $f + 1$ nodes out of $2f + 1$ nodes, since $2f + 2 > 2f + 1$, there is at least one node in the intersection. □

2. **SUBSET** S is the set of subsets of S

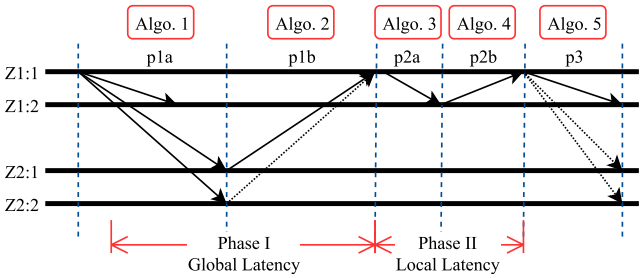


Figure 2: Normal case messaging flow

2.2. Multi-leader

In contrast to FPaxos [19] which uses flexible quorums with a classical single-leader Paxos protocol, WPaxos presents a multi-leader protocol over flexible quorums. Every node in WPaxos can act as a leader for a subset of 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 $q_1 \in Q_1$. Then phase-2 commits the update-requests to the object over $q_2 \in Q_2$, selected from the leader's zone (and nearby zones) for improved locality. The leader can execute phase-2 multiple times until some other node steals the object.

The phase-1 of the protocol starts only the node needs to steal an object from a remote leader or if a client has a request for a brand new object that is not in the system. 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.

2.3. 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 performs phase-1 on some $q_1 \in Q_1$ with a larger ballot. Object stealing is successful if the candidate node can out-ballot the existing leader. This is achieved in just one phase-1 attempt, provided that the local cache is current and a remote leader is not engaged in another phase-1 on the same object.

Once the object is stolen, the old leader cannot 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 q_1 when the key

was stolen, because the intersected node in q_2 will reject any object operations attempted with the old ballot. The object stealing 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 different objects from each other by constantly proposing a higher ballot than the opponent. Using separate ballot numbers for each object alleviates 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: (1) resolving ballot conflict by zone ID and node ID in case the ballot counters are the same, and (2) implementing a random back-off mechanism in case a new dueling iteration starts anyway.

Object stealing is part of core WPaxos protocol. In contrast to the simplicity and agility of object stealing in WPaxos, object relocation in other systems require integration of another service, such as movedir in Spanner [8], or performing multiple reconfiguration or coordination steps as in Vertical Paxos [21]. Vertical Paxos depends on a reliable master service that overseeing configuration changes. Object relocation involves configuration change in the node responsible for processing commands on that object. When a node in a different region attempts to steal the object, it must first contact the reconfiguration master to obtain the current ballot number and next ballot to be used. The new leader then must complete phase-1 of Paxos on the old configuration to learn the previous commands. Upon finishing the phase-1, the new leader can commit any uncommitted slots with its own set of acceptors. At the same time the new leader notifies the master of completing phase-1 with its ballot. Only after the master replies and activates the new configuration, the leader can start serving user requests. This process can be extended to multiple objects, by keeping track of separate ballot numbers for each object. Vertical Paxos requires three separate WAN communications to change the leadership, while WPaxos can do so with just one WAN communication.

3. Algorithm

In the basic algorithm, every node maintains a set of variables and a sequence of commands written into the command *log*. The command log can be committed out of order, but has to be executed against the state machine in the same order without any gap. Every command accesses only one object o. Every node leads its own set of objects in a set called own.

All nodes in WPaxos initialize their state with above variables. We assume no prior knowledge of the ownership of the objects; a user can optionally provide initial object assignments. The highest known ballot numbers for objects are constructed by concatenating counter=0 and the node ID (line 1). The slot numbers start from zero (line 2), and the objects self owned is an empty set (line 3). Inside the log, an instance contains three components, the ballot number b for that slot, the proposed command/value v and a flag c indicates whether the instance is committed (line 4).

process(self $\in$ Nodes) Initialization

variables

1: $ballots = [o \in Objects \mapsto \langle 0, self \rangle]$;
2: $slots = [o \in Objects \mapsto 0]$;
3: $own = \{\}$
4: $log = [o \in Objects \mapsto$
$[s \in Slots \mapsto$
$[b \mapsto 0, v \mapsto \langle\rangle, c \mapsto \text{FALSE}]]]$;

3.1. Phase-1: Prepare

Algorithm 1 Phase-1a

```
1: macro p1a () {
2:     with (o ∈ Objects) {
3:         await o ∉ own;
4:         ballots[o] := ⟨ballots[o][1] + 1, self⟩;
5:         Send([type ↦ "1a",
                 n ↦ self,
                 o ↦ o,
                 b ↦ ballots[o]]); }}
```

WPaxos starts with a client sending requests to one of the nodes. A client typically chooses a node in the local zone to minimize the initial communication costs. The request message includes a command and some object o on which the command needs to be executed. Upon receiving the request, the node checks if the object exists in the set of own, and start phase-1 for any new objects by invoking **p1a()** procedure in Algorithm 1. If the object is already owned by this node, the node can directly start phase-2 of the protocol. In p1a(), a larger ballot number is selected and **"1a"** message is sent to a Q_1 quorum.

Algorithm 2 Phase-1b

```
1: macro p1b () {
2:     with (m ∈ msgs) {
3:         await m.type = "1a";
4:         await m.b ≥ ballots[m.o];
5:         ballots[m.o] := m.b;
6:         if (o ∈ own) own := own \ {m.o};
7:         Send([type ↦ "1b",
                 n ↦ self,
                 o ↦ m.o,
                 b ↦ m.b,
                 s ↦ slots[m.o]]); }}
```

The **p1b()** procedure processes the incoming **"1a"** message sent during phase-1 initiation. A node can accept the sender as the leader for object o only if the sender's ballot $m.b$ is greater or equal to the ballot number it knows of (line 4). If object o is owned by current node, it is removed from set own (line 6). Finally, the **"1b"** message acknowledging the accepted ballot number is send (line 7). The highest slot

associated with o is also attached to the reply message, so that any unresolved commands can be committed by the new leader.

3.2. 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. WPaxos carries out this phase on a Q_2 quorum residing in the closest $F+1$ zones, thus all communication is kept local, greatly reducing the latency.

Algorithm 3 Phase-2a

```
1: Q1Satisfied(o, b) ≜ ∃q ∈ Q1 : ∀n ∈ q : ∃m ∈ msgs :
                          ∧ m.type = "1b"
                          ∧ m.o = o
                          ∧ m.b = b
                          ∧ m.n = n
2: macro p2a () {
3:     with (m ∈ msgs) {
4:         await m.type = "1b";
5:         await m.b = ⟨ballots[m.o][1], self⟩;
6:         await m.o ∉ own;
7:         if (Q1Satisfied(m.o, m.b)) {
8:             own := own ∪ {m.o};
9:             slots[m.o] := slots[m.o] + 1;
10:            log[m.o][slots[m.o]] := [b ↦ m.b,
                                        v ↦ ⟨slots[m.o], self⟩,
                                        c ↦ FALSE];
11:            Send([type ↦ "2a",
                     n ↦ self,
                     o ↦ m.o,
                     b ↦ m.b,
                     s ↦ slots[m.o],
                     v ↦ ⟨slots[m.o], self⟩]); }}}
```

Procedure **p2a()** in Algorithm 3 collects the "1b" messages for itself (lines 4-6). The node becomes the leader of the object only if Q_1 quorum is satisfied (line 7,8). The new leader then recovers any uncommitted slots with suggested values and starts the accept phase for the pending requests that have accumulated in queue. Phase-2 is launched by increasing the highest slot (line 9), and creates new entry in log (line 10), sending **"2a"** message (line 11).

Algorithm 4 Phase-2b

```
1: macro p2b () {
2:     with (m ∈ msgs) {
3:         await m.type = "2a";
4:         await m.b ≥ ballots[m.o];
5:         ballots[m.o] := m.b;
6:         log[m.o][m.s] := [b ↦ m.b, v ↦ m.v, c ↦ FALSE];
7:         Send([type ↦ "2b",
                 n ↦ self,
                 o ↦ m.o,
                 b ↦ m.b,
                 s ↦ m.s]); }}
```

Once the leader of the object sends out the "2a" message at the beginning of phase-2, the replicas respond to this message as shown in Algorithm 4. The leader node updates its instance at slot $m.s$ only if the message ballot $m.b$ is greater or equal to accepted ballot (line 4-6).

Algorithm 5 Phase-3

```
1: Q2Satisfied(o, b, s) ≜ ∃q ∈ Q2 : ∀n ∈ q : ∃m ∈ msgs :
                            ∧ m.type = "2b"
                            ∧ m.o = o
                            ∧ m.b = b
                            ∧ m.s = s
                            ∧ m.n = n
2: macro p3 () {
3:     with (m ∈ msgs) {
4:         await m.type = "2b";
5:         await m.b = ⟨ballots[m.o][1], self⟩;
6:         await log[m.o][m.s].c ≠ TRUE;
7:         if (Q2Satisfied(m.o, m.b, m.s)) {
8:             log[m.o][m.s].c := TRUE;
9:             Send([type ↦ "3",
                     n ↦ self,
                     o ↦ m.o,
                     b ↦ m.b,
                     s ↦ m.s,
                     v ↦ log[m.o][m.s].v]); }}}
```

3.3. Phase-3: Commit

The leader collects replies from its Q_2 acceptors. The request proposal either gets committed with replies satisfying a Q_2 quorum, or aborted if some acceptors reject the proposal citing a higher ballot number. In case of rejection, the node updates a local ballot and puts the request in this instance back to main request queue to retry later.

3.4. Properties

Non-triviality. For any node n, the set of committed commands is always a sequence σ of proposed commands, i.e. $\exists\sigma : committed[n] = \perp \bullet \sigma$. 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 n, the sequence of committed commands at any time is a prefix of the sequence at any later time, i.e. $\exists\sigma : committed[n] = \gamma \text{ at } t \implies committed[n] = \gamma \bullet \sigma$ at $t + \Delta$. Stability asserts any committed command cannot be overridden later. It is guaranteed and proven by Paxos that any leader with higher ballot number will learn previous values before proposing new slots. WPaxos inherits the same process.

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 [22]. 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.

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

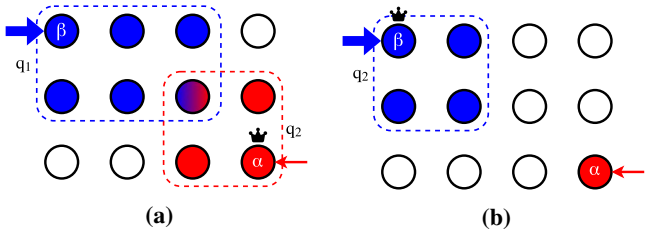


Figure 3: (a) Initial leader α observes heavy cross-region traffic from node β, thus triggers β to start phase-1 on its q_1. (b) β becomes new leader and benefits more on the workload.

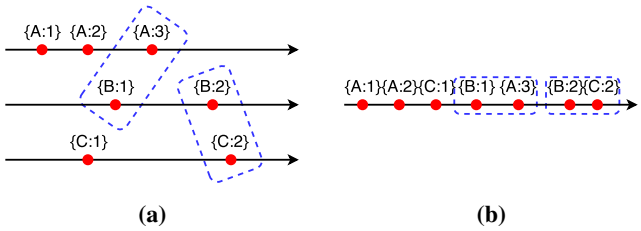


Figure 4: (a) Logs for A, B, C three objects where dashed boxes encompass multi-object transactions. (b) One possible serialization ordered by common object B's slot numbers.

$\gamma \in committed[n]$. The PlusCal code presented in Section 3 specifies what actions each node is allowed to perform, but not when to perform, which affects liveness. The liveness property satisfied by WPaxos algorithm is same to that of ordinary Paxos: as long as there exists $q_1 \in Q_1$ and $q_2 \in Q_2$ are alive, the system will progress.

4. Extensions

4.1. Locality Adaptive Object Stealing

The basic protocol migrates the object from a remote region to a local region upon the first request, but that causes a performance degradation when an object is frequently accessed across many zones. With locality adaptive object stealing we can delay or deny the object transfer to a zone issuing the request based on an 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 accessing the object from less frequent zones to get their requests forwarded to the remote leader.

Our *majority-zone* migration policy aims to improve the locality of reference by transferring the objects to zones that sending out the highest number of requests for the objects, as shown in Figure 3. Since the current object leader handles all the requests, it has the information about which clients access the object more frequently. If the leader α detects that the object has more requests coming from a remote zone, it will initiate the object handover by communicating with the node β, and in its turn β will start the phase-1 protocol to steal the leadership of that object.

4.2. Replication Set

WPaxos provides flexibility in selecting a replication set. The phase-2 (p2a) message need not be broadcast to the entire system, but only to a subset of Q_2 quorums, denoted as a replication Q_2 or RQ_2. The user has the freedom to choose the replication factor across zones from the minimal required $F + 1$ zones up to the total number of Z zones. Such choice can be seen as a trade off between communication overhead and a more predictable latency, since the replication zone may not always be the fastest to reply. Additionally, if a node outside of the RQ_2 becomes the new leader of the object, that may delay the new phase-2 as the leader need to catch up with the missing logs in previous ballots. One way to minimize the delay is let the RQ_2 reply on phase-2 messages for replication, while the potential leader nodes learn the states as non-voting learners.

4.3. Transactions

Here we present a simple implementation of multi-object transactions that happens entirely within the Paxos protocol, and that obviates the need for integrating a separate two-phase commit for transactions as in Spanner [8]. In this implementation, the node that initiates a transactional operation, first steals all the objects needed for the transaction to itself. This is done in increasing order of object IDs to avoid deadlock and livelock. This phase may require multiple Q_1 accesses. Then the leader commits the transaction in phase-2 via a Q_2 access. The execution order is achieved by collating/serializing the logs together, establishing the order of interfering transactions by comparing the slot numbers of the common objects in the transactions, as shown in Figure 4. This ordering happens naturally as the transactions cannot get executed before the previous slots for all related objects 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 serializable protocol [23,24]. WPaxos transactions ensure strict serializability if the commit notification for any request is sent to client after execution.

To improve the performance, it is possible to implement an object-group abstraction that packs closely-coupled objects in one object-group to use one log/ballot. We relegate optimization and evaluation of multi-object transaction implementation to future work.

5. Fault-tolerance & Reconfiguration

5.1. Fault Tolerance

WPaxos can make progress as long as it can form valid q_1 and q_2 quorums. The flexibility of WPaxos enables the user to deploy the system with quorum configuration tailored to their needs. Some configurations are geared towards performance, while others may prioritize fault tolerance. By default, WPaxos configures the quorums to tolerate

one zone failure and minority node failures per zone, and thus provides similar fault tolerance as Spanner with Paxos groups deployed over three zones.

WPaxos remains partially available when more zones fail than the tolerance threshold it was configured for. In such a case, no valid q_1 quorum may be formed, which halts the object stealing routine, however the operations can proceed for objects owned in the remaining live regions, as long as there are enough zones left to form a q_2 quorum.

5.2. Dynamic Reconfiguration

The ability to reconfigure, i.e., dynamically change the membership of the system, is critical to provide reliability for long periods as it allows crashed nodes to be replaced. WPaxos achieves high throughput by allowing *pipelining* (like Paxos and Raft algorithms) in which new commands may begin phase-2 before any previous instances/slots have been committed. Pipelining architecture brings more complexity to reconfiguration, as there may be another reconfiguration operation in the pipeline which could change the quorum and invalidate a previous proposal. Paxos [3] solves this by limiting the length of the pipeline window to $\alpha > 0$ and only activating the new config C' chosen at slot i until slot $i + \alpha$. Depending on the value of α, this approach either limits throughput or latency of the system. On the other hand, Raft [7] does not impose any limitation of concurrency and proposes two solutions. The first solution is to restrict the reconfiguration operation, i.e. what can be reconfigured. For example, if each operation only adds one node or removes one node, a sequence of these operations can be scheduled to achieve arbitrary changes. The second solution is to change configuration in two phases: a union of both old and new configuration $C + C'$ is proposed in the log first, and committed by the quorums combined. Only after the commit, the leader may propose the new config C'. During the two phases, any election or command proposal should be committed by quorum in both C and C'. To ensure safety during reconfiguration, all these solutions essentially prevent two configurations C and C' to make decision at the same time that leads to divergent system states.

WPaxos adopts the more general two-phase reconfiguration procedure from Raft for arbitrary C's, where $C = \langle Q_1, Q_2 \rangle$, $C' = \langle Q'_1, Q'_2 \rangle$. WPaxos further reduces the two phases into one in certain special cases since adding and removing one zone or one row operations are the most common reconfigurations in the WAN topology. These four operations are equivalent to the Raft's first solution because the combined quorum of $C + C'$ is equivalent to quorum in C'. We show one example of adding new zone of dashed nodes in the Figure 5.

Previous configuration Q_1 involves two zones, whereas the new config Q'_1 involves three zones including the new zone added. The quorums in Q'_1 combines quorums in Q_1 is same as Q'_1. Both Q_2 and Q'_2 remains the same size of two zones. The general quorum intersection assumption and the restrictions $Q'_1 \cup Q_1 = Q'_1$ and $Q'_2 \cup Q_2 = Q'_2$ ensure that old and new configuration cannot make separate decisions and provides same safety property as in S2.

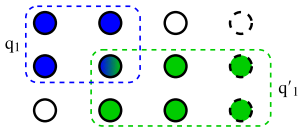


Figure 5: Add one zone

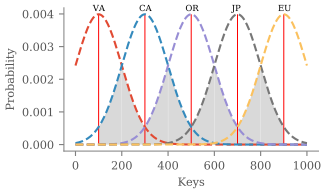


Figure 6: Locality Workload

6. Evaluation

We developed a general framework, called *Paxi* to conduct our evaluation. The framework allows us to compare WPaxos, EPaxos, and other Paxos protocols in the same controlled environment under identical workloads. We implemented Paxi along with WPaxos and EPaxos in Go version 1.9 and released it as an open-source project on GitHub at https://github.com/ailidani/paxi. The framework provides extended abstractions to be shared between all Paxos variants, including location-aware configuration, network communication, client library with RESTful API, and a quorum management module (which accommodates majority quorum, fast quorum, 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, and transparently supports TCP, UDP and simulated connection with Go channels. Additionally, our Paxi framework incorporates mechanisms to facilitate the startup of the system by sharing the initial parameters through the configuration management tool.

6.1. Setup

We evaluated WPaxos using the key-value store abstraction provided by our Paxi framework. We used AWS EC2 [25] nodes to deploy WPaxos across 5 different regions: Virginia (VA), California (CA), Oregon (OR), Tokyo (JP), and Ireland (EU). In our experiments, we used 3 medium instances 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 generating a request on each object. 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 pulled from such set of likely objects. 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 2. ***Locality*** L is the complement of the overlapping coefficient (OVL)[3] among workload distributions:

3. The overlapping coefficient (OVL) is a measurement of similarity between two probability distributions, refers to the shadowed area under two probability density functions simultaneously [26].

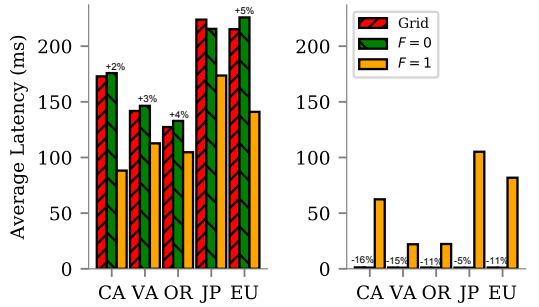


Figure 7: Average latency for phase-1 (left) and phase-2 (right) in different quorum systems

$L = 1 - \widehat{OVL}$.

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.

6.2. Quorum Latencies

In this first set of experiments, we compare the latency of Q_1 and Q_2 accesses in three types of quorums: grid quorum and WPaxos quorum with $F = 0$ and 1. The grid quorum uses a single node per zone/region for Q_1, requiring all nodes in one zone/region to form Q_2. WPaxos quorum uses majority nodes in a zone consequently requires one fewer node in Q_2 than Grid to tolerant one node failure. When F=0, Q_1 uses all 5 zones and Q_2 remains in the same region. With F=1, Q_1 uses 4 zones which reduce phase-1 latency simnifically, but Q_2 requires 2 zones/regions thus exhibits WAN latency. In each region we simultaneously generated a fixed number (1000) of phase-1 and phase-2 requests, and measured the latency for each phase. Figure 7 shows the average latency in phase-1 (left) and phase-2 (right), and the differences between WPaxos F=0 quorum with Grid quorum in data labels above the bar.

Quorum size of Q_1 in Grid is half of that for WPaxos F=0, but both experience average latency of about one round trip to the farthest peer region, since the communication happens in parallel. Within a zone, however, F=0 can tolerate one straggler node, reducing the latency for the most frequently used Q_2 quorum type. Due to lack of space, we use F=0 quorum in following experiments to show the better performance of WPaxos, as similar performance may also achieved with F=1 when two zones are deployed in different AWS availability zones in the same region.

6.3. Latency

We compare the commit latency between WPaxos and EPaxos with three sets of workloads, random (Figure 8a), ∼70% locality (Figure 8b), and ∼90% locality (Figure 8c). 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 nodes versions.

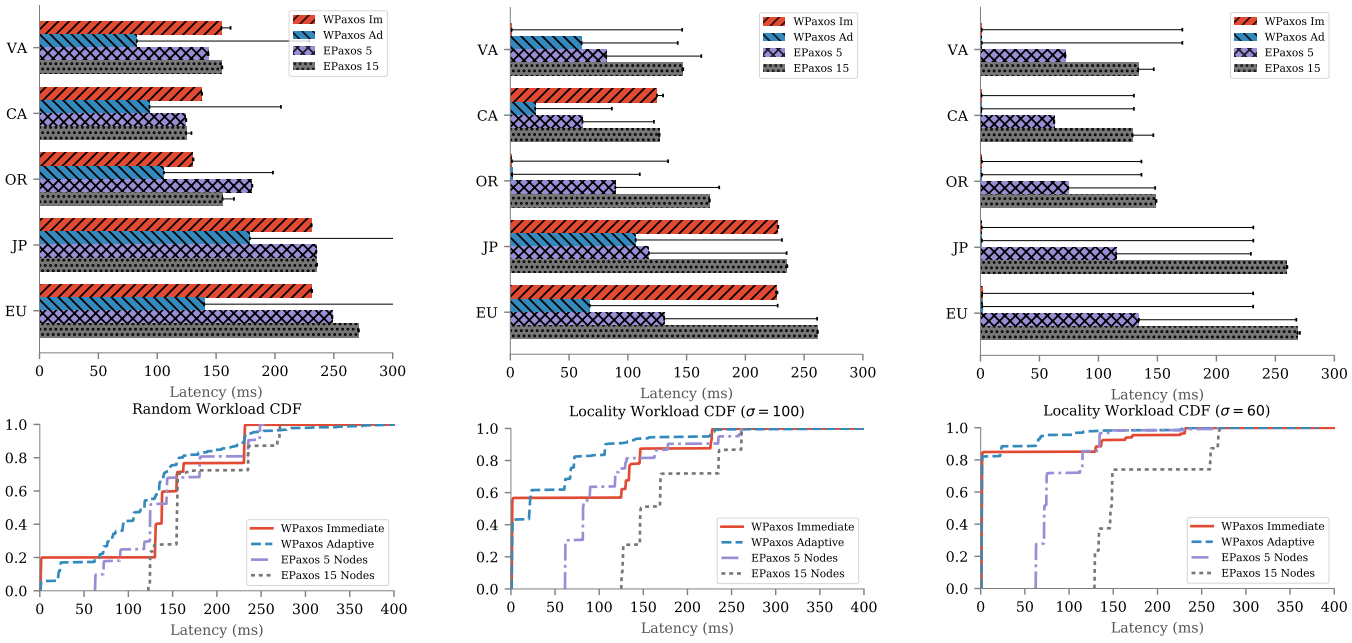
Figure 8a compares the median (color bar) and 95th percentile (error bar) latency of random workload in 5 regions. Each region experiences different latency due to their asymmetrical location in the geographical topology. WPaxos with immediate object stealing shows a similar performance with EPaxos because the random workload causes many phase-1 invocations which require wide area RTT. However, WPaxos with adaptive mode outperforms EPaxos in every region. Figure 8a also 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 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 8b 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 8b 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.

In Figure 8c we increase the locality to ∼90%. EPaxos shows similar pattern as previous experiments, whereas WPaxos achieves local median latency in all regions. In Figure 8c, 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.

6.4. 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 a cluster of 15 *large* EC2 nodes to host WPaxos deployments. EPaxos is hosted at the same nodes, 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. 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 9 shows the latencies as the aggregate throughput increases.



(a) Uniformly random workload **(b)** Locality (70%) workload **(c)** Locality (90%) workload

Figure 8: Median/95th percentile latencies in different regions and their global CDF

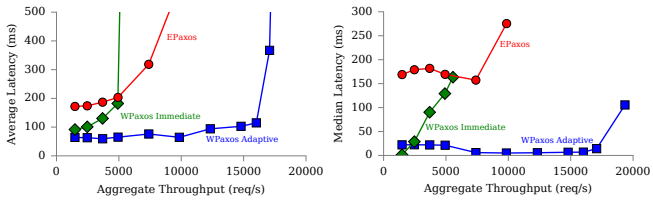


Figure 9: Request latency as the throughput increases.

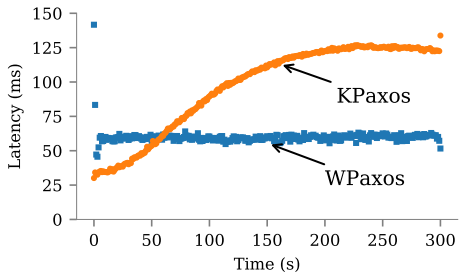


Figure 10: The average latency in each second.

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. 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 9 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 greater scalability of immediate object stealing mode due to reduced contention for the objects between neighboring zones.

On the other hand, WPaxos in adaptive object stealing mode scales 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.

6.5. Shifting Locality Workload

Many applications in the WAN setting may experience workloads with shifting access patterns such as diurnal patterns [27,28]. Figure 10 illustrates the effects of shifting locality in the workload on WPaxos and statically key-partitioned Paxos (KPaxos). 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 re-

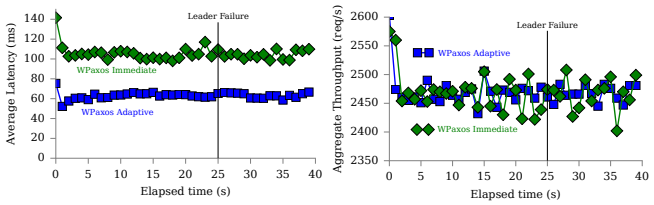


Figure 11: Leader failure in one zone has no measurable impact on performance.

gions with more demand, providing stable and predictable performance under shifting access locality.

6.6. Fault Tolerance

Figure 11 illustrates that there is negligible impact on performance due to a single leader/node failure. In this experiment, we run a steady locality biased workload through the WPaxos deployment. At 25th second mark we kill the leader replica in OR region, 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 also insignificant, and the recovery is spread in time, as recovery for each individual object happens only when that object is needed. The throughput is unaffected.

7. Related Work

Several attempts have been made for improving the scalability of Paxos. Mencius [29] 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 [16] 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 [30], 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 commands, but is supported for multiple-object updates. M^{2}Paxos does not address WAN deployments and is subject to WAN latencies for commit operations since it uses majority quorums.

Bizur [18] 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 [17] 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. That is, the object-space is 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 [31] takes a different approach to handling diurnal patterns in the workload. Instead of making end systems adjustable to access patterns, Supercloud moves non-adjustable components to the places of most frequent use by using live-VM migration.

We presented WanKeeper [32] for WAN coordination. WanKeeper uses a token broker architecture where the centralized broker gets to observe access patterns and improves locality of update operations by migrating tokens when appropriate. To achieve scalability in WAN deployments, WanKeeper uses hierarchical composition of brokers, and employs the concept of token migration to give sites locality of access and autonomy. WanKeeper provides local reads at sites, and when locality of access is present, it also enables local writes at the sites. WPaxos provides a simple flexible protocol and can be used as building primitive for other protocols, services, and applications.

8. 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 other WAN Paxos solutions, while maintaining the same consistency guarantees. Since the object stealing is an integrated part of phase-1 of Paxos, WPaxos remains simple as a pure Paxos flavor and obviates the need for another service/protocol for relocating objects to zones. Since the base WPaxos protocol guarantees safety to concurrency, asynchrony, and faults, the performance can be tuned orthogonally and aggressively. In future work, we will investigate smarter object stealing that can proactively move objects to zones with high demand. We will also investigate implementing transactions more efficiently leveraging WPaxos optimizations.

References

[1] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.

[2] M. Burrows, "The chubby lock service for loosely-coupled distributed systems." in *OSDI*. USENIX Association, 2006, pp. 335–350.

[3] R. Van Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Computing Surveys (CSUR)*, vol. 47, no. 3, p. 42, 2015.

[4] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX ATC*, vol. 10, 2010.

[5] F. Junqueira, B. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Dependable Systems & Networks (DSN)*. IEEE, 2011, pp. 245–256.

[6] "A distributed, reliable key-value store for the most critical data of a distributed system," https://coreos.com/etcd/.

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

[8] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman *et al.*, "Spanner: Google's globally-distributed database," *Proceedings of OSDI*, 2012.

[9] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson *et al.*, "Megastore: Providing scalable, highly available storage for interactive services," *CIDR*, pp. 223–234, 2011.

[10] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan, "Salt: Combining acid and base in a distributed database." in *OSDI*, vol. 14, 2014, pp. 495–509.

[11] D. Quintero, M. Barzaghi, R. Brewster, W. H. Kim, S. Normann, P. Queiroz *et al.*, *Implementing the IBM General Parallel File System (GPFS) in a Cross Platform Environment*. IBM Redbooks, 2011.

[12] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières, "Replication, history, and grafting in the ori file system," in *Proceedings of SOSP*, ser. SOSP '13, New York, NY,, 2013, pp. 151–166.

[13] A. Grimshaw, M. Morgan, and A. Kalyanaraman, "Gffs – the XSEDE global federated file system," *Parallel Processing Letters*, vol. 23, no. 02, p. 1340005, 2013.

[14] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's distributed data store for the social graph," *Usenix Atc'13*, pp. 49–60, 2013.

[15] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *SOSP*, 2011, pp. 401–416.

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

[17] K. Lev-Ari, E. Bortnikov, I. Keidar, and A. Shraer, "Modular composition of coordination services," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[18] E. N. Hoch, Y. Ben-Yehuda, N. Lewis, and A. Vigder, "Bizur: A Key-value Consensus Algorithm for Scalable File-systems," 2017. [Online]. Available: http://arxiv.org/abs/1702.04242

[19] H. Howard, D. Malkhi, and A. Spiegelman, "Flexible Paxos: Quorum intersection revisited," 2016. [Online]. Available: http://arxiv.org/abs/1608.06696

[20] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872–923, May 1994.

[21] L. Lamport, D. Malkhi, and L. Zhou, "Vertical paxos and primary-backup replication," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 2009, pp. 312–313.

[22] L. Lamport, "Generalized consensus and paxos," Technical Report MSR-TR-2005-33, Microsoft Research, Tech. Rep., 2005.

[23] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990. [Online]. Available: http://doi.acm.org.gate.lib.buffalo.edu/10.1145/78969.78972

[24] M. K. Aguilera and D. B. Terry, "The many faces of consistency." *IEEE Data Eng. Bull.*, vol. 39, no. 1, pp. 3–13, 2016.

[25] Amazon Inc., "Elastic Compute Cloud," Nov. 2008.

[26] H. F. Inman and E. L. Bradley Jr, "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*, vol. 18, no. 10, pp. 3851–3874, 1989.

[27] G. Zhang, L. Chiu, and L. Liu, "Adaptive data migration in multi-tiered storage based cloud environment," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE, 2010, pp. 148–155.

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

[29] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building Efficient Replicated State Machines for WANs," *Proceedings of the Symposium on Operating System Design and Implementation*, pp. 369–384, 2008. [Online]. Available: http://www.usenix.org/event/osdi08/tech/full{_}papers/mao/mao{_}html/

[30] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran, "Making fast consensus generally faster," *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016*, pp. 156–167, 2016.

[31] Z. Shen, Q. Jia, G.-E. Sela, B. Rainero, W. Song, R. van Renesse, and H. Weatherspoon, "Follow the sun through the clouds: Application migration for geographically shifting workloads," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 141–154.

[32] Wankeeper project. [Online]. Available: https://github.com/ailidani/wankeeper