Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage

Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha
University of Michigan

Abstract— **By replicating data across sites in multiple geographic regions, web services can maximize availability and minimize latency for their users. However, when sacrificing data consistency is not an option, we show that service providers have to today incur significantly higher cost to meet desired latency goals than the lowest cost theoretically feasible. We show that the key to addressing this sub-optimality is to 1) allow for erasure coding, not just replication, of data across data centers, and 2) mitigate the resultant increase in read and write latencies by rethinking how to enable consensus across the wide-area network. Our extensive evaluation mimicking web service deployments on the Azure cloud service shows that we enable near-optimal latency versus cost tradeoffs.**

1 Introduction

Replicating data across data centers is important for a web service to tolerate the unavailability of some data centers [1] and to serve users with low latency [5]. A front-end web server close to a user can serve the user's requests by accessing nearby copies of relevant data (see Figure 1). Even in collaborative services such as Google Docs and ShareLaTeX, accessing a majority of replicas suffices for a front-end to read or update shared data while preserving consistency.

However, it is challenging to keep data spread across the globe strongly consistent as no single design can simultaneously minimize read latency, write latency, and cost.

- To preserve consistency, *any* subset of sites which are accessed to serve a read must overlap with *all* subsets used for writes. Therefore, allowing a front-end to read from nearby data sites forces other front-ends to write to distant data sites, thus increasing write latency.
- Similarly, providing low read latency requires having at least one data site near each front-end, thereby increasing the total number of data sites. This inflates expenses incurred both for storage and for data transfers to synchronize data sites.

Given these tradeoffs, service providers must determine how to meet their desired latency goals at minimum cost. Or, correspondingly, how to minimize read and write latencies given a cost budget? In this paper, we make the following contributions towards addressing these questions.

1. We show that existing solutions for enabling strongly consistent distributed storage are far from optimal in trading off latency versus cost. The cost necessary to satisfy bounds on read and write latencies is often significantly higher than the lowest cost theoretically feasible. For example, across a range of access patterns and latency bounds, the state-of-the-art geo-replication protocol EPaxos [51] imposes on average 30% higher storage cost than is optimal (§5.1.2). This sub-optimality also inflates the minimum latency bounds satisfiable within a cost budget.

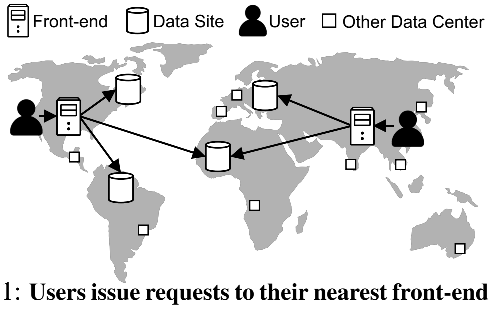


Figure 1: **Users issue requests to their nearest front-end servers which in turn access geo-distributed storage.**

2. We demonstrate the feasibility of achieving near-optimal latency versus cost tradeoffs in strongly consistent geo-distributed storage. In other words, we do not merely improve upon the status quo, but show that there remains little room for improvement over the tradeoffs enabled by Pando, our new approach for consensus across the wide-area network. Pando exploits the property that, from any data center's perspective, some data centers are more proximate than others in a geo-distributed deployment. Therefore, beyond reducing the *number of round-trips* of wide-area communication when executing reads and writes (as has typically been the goal in prior work [49, 42, 51]), it is equally important to reduce the *magnitude of delay incurred on every round-trip*. We apply this principle in two ways.

2a. We show how to erasure-code objects across data sites without reads incurring higher wide-area latencies compared to replicated data. By splitting each object's data and storing one split (instead of one replica) per data site, a service can use its cost budget to spread each object's data across more data centers than is feasible with replication. To leverage this increased geographic spread for minimizing latencies, Pando separates out two typically intertwined aspects of consensus: discovering whether the last write completed, and determining how to resolve any associated uncertainty. Since writes seldom fail in typical web service deployments, we enable a client to read an object by first communicating with a small subset of nearby data sites; only in the rare case when it is uncertain whether the last write completed does the client incur a latency penalty to discover how to resolve the uncertainty.

2b. In the wide-area setting, we show how to reach consensus in two rounds, yet approximate a one-round protocol's latency. Executing writes in two rounds simplifies compatibility with erasure-coded data, and we ensure that this approach has little impact on latency. First, PANDO requires clients to contact a smaller, more proximate subset of data sites in the first round than in the second round. Second, after a client initiates the first round, it delegates initiation of the second round to a more central data center, which receives all responses from the first round. By combining these two measures, messaging delays incurred in the first phase of a write help reduce the latency incurred in the second phase, instead of adding to it.

3. We compare PANDO to state-of-the-art consensus protocols via extensive measurement-driven analyses and in deployments on Azure. In the latency–cost tradeoff space, we find that PANDO reduces by 88% the median gap between achievable tradeoffs and the best theoretically feasible tradeoffs. Moreover, PANDO can cut dollar costs to meet the same latency goals by 46% and lower 95^{th} percentile read latency by up to 62% at the same storage overhead.

2 Setting and Motivation

We begin by describing our target setting, the approach we use for enabling globally consistent reads and writes, and the shortcomings of existing solutions that use this approach.

2.1 System model, goals, and assumptions

We seek to meet the storage needs of globally deployed applications, such as Google Docs [4] and ShareLaTeX [8], in which low latency and high availability are critical, yet weak data consistency (such as eventual or causal) is not an option. In particular, we focus on enabling a geo-distributed object/key-value store which a service's front-end servers read from and write to when serving requests from users. We aim to support GETs and conditional-PUTs on any individual key; we defer support for multi-key transactions to future work. In contrast to PUTs (which blindly overwrite the value for a key), conditional-PUTs attempt to write to a specific version of a key and can succeed only if that version does not already have a committed value. This is essential in services such as Google Docs and ShareLaTeX to ensure that a client cannot overwrite an update that it has not seen.

In enabling such a geo-distributed key-value store, we are guided by the following objectives:

- **Strong consistency:** Ensure all reads and writes on any key are linearizable; i.e., all writes are totally ordered and every read returns the last successful write.
- **Low latency:** Satisfy service provider's SLOs[1] (service-level objectives) for bounds on read and write latencies, so as to ensure a minimum quality-of-service for all users. We focus on the wide-area latency incurred when serving reads or writes, assuming appropriate capacity planning and load balancing to bound queuing delays.
- **Low cost:** Minimize cost (sum of dollar costs for storage, data transfers, storage operations, and compute) necessary to satisfy latency goals. Since cost for storage operations and data transfers grows with more copies stored, in parts of the paper, we use storage overhead (i.e., number of copies stored of every data item) as a proxy for cost. This frees us from making any assumptions about pricing policy or the workload (e.g., read-to-write ratio).
- **Fault-tolerance:** Serve requests on any key as long as fewer than f data centers are unavailable.

We focus on satisfying input latency bounds in the absence of conflicts and failures—both of which occur rarely in practice [11, 2, 48, 29]—but seek to minimize performance degradation when they do occur (§3.5 and §5.1.2). In addition, we build upon state-of-the-art cloud services which offer low latency variance between their data centers [34] and within their intra-data center storage services (e.g., Azure's CosmosDB provides a 10 ms tail read latency SLA [12]).

Note that, in order to satisfy desired latency SLOs at minimum cost (or to minimize latencies given a cost budget), a service cannot select the data sites for an object at random. Instead, as we describe later in Section 4, any service must utilize its knowledge of an object's workload (e.g., locations of the users among whom the object is shared) in doing so.

2.2 Approach

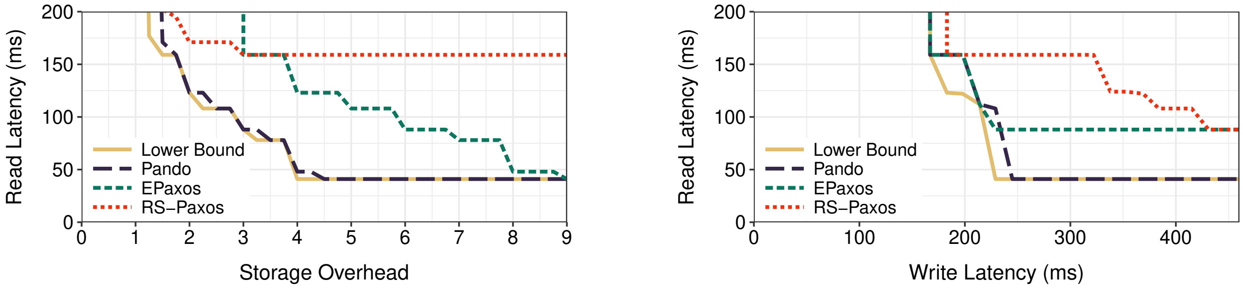
One can ensure linearizability in distributed storage by serializing all writes through a leader and rely on it for reads, e.g., primary-backup [18], chain replication [68], and Raft [57]. A single leader, however, cannot be close to all front-ends across the globe. Front-ends which are distant from the leader will have to suffer high latencies.

To reduce the need to contact a distant leader, one could use read leases [21, 52] and migrate the leader based on the current workload, e.g., choose as the leader the replica closest to the front-end currently issuing reads and writes. However, unless the workload exhibits very high locality, tail latency will be dominated by the latency overheads incurred during leader migration and lease acquisition.

To keep read and write latencies within specified bounds irrespective of the level of locality, we pursue a leaderless approach. Among the leaderless protocols which allow every front-end to read and write data from a subset of nearby data sites (a read or write quorum), we consider those based on Paxos because it enables consensus. Other quorum-based approaches [20] which only enable atomic register semantics (i.e., PUT and GET) are incapable of supporting conditional updates [35]. While there exist many variants of Paxos, in all cases, we can optimize latencies in two ways.

First, instead of executing Paxos, a front-end can read an object by simply fetching the object's data from a read quorum. To enable this, a successful writer asynchronously marks the version it wrote as committed at all data sites. In the common case, when there are no failures or conflicts, a read is complete in one round trip if the highest version seen across a read quorum is marked as committed [44].

[1] Unlike SLAs, violations of SLOs are acceptable, but need to be minimized.



(a) Write latency $\leq$ 300 ms

(b) Storage overhead $\leq 6\times$

Figure 2: **Slices of the three-dimensional tradeoff space where we compare latency estimates for replication-based EPaxos [51], erasure coding-based RS-Paxos [53], and our solution PANDO against a lower bound. Front-ends are in Azure's Australia East, Central India, East Asia, East US, and Korea South data centers, whereas data sites are chosen from all Azure data centers.**

Second, instead of every front-end itself executing reads and writes, we allow for it to relay its operations through a delegate in another data center. The flexibility of utilizing a delegate can be leveraged to reduce latency when, compared to the front-end, that delegate is more centrally placed relative to the data sites of the object being accessed.

2.3 Sub-optimality of existing solutions

The state-of-the-art Paxos variant for geo-replicated data is EPaxos [51], as we show in Section 5. For typical replication factors (i.e., 3 or 5), EPaxos enables any front-end to read/write with one round of wide-area communication with the nearest majority of replicas. If lower read latencies than feasible with $2f + 1$ replicas are desired, then one can use a higher replication factor N, set the size R of read quorums to be $\geq f + 1$ (to ensure overlap with write quorums even in the face of f failures) and set the size W of write quorums to $N - R + 1$ (to preserve consistency).

Figure 2 shows the tradeoffs enabled by EPaxos for an example access pattern. For each read latency bound, these graphs respectively plot the minimum storage overhead and write latency bounds that are satisfiable. As we discuss later in the paper, we compute these bounds by solving protocol-specific mixed integer programs (§4) which take as input the expected access pattern and latency measurements between all pairs of data centers (§5.1). We show two two-dimensional slices of the three-dimensional read latency–write latency–storage overhead tradeoff space.

To gauge the optimality of the tradeoffs achievable with EPaxos, we compare it against a lower bound. Given a bound on read latency, the minimum storage overhead necessary and the minimum write latency bound that can be satisfied cannot be lower than those determined by our lower bound. Though the lower bound may be unachievable by any existing consensus protocol, we compute it by solving a mixed integer program which assumes that reads and writes can be executed in a single round and enforces the following properties that *any* quorum-based approach must respect:

- *Tolerate unavailability of $\leq f$ data centers:* All data sites in at least one read and one write quorum must be available in the event that $\leq f$ data centers fail.
- *Prevent data loss:* At least one copy of data must remain in any write quorum when any f data sites are unavailable.
- *Serve reads:* The data sites in any read quorum must collectively contain at least one copy of the object.
- *Preserve strong consistency:* All read–write and write–write quorum pairs must have a non-empty intersection.

Equally important are constraints that we do *not* impose: all read quorums (same for write quorums) need not be of the same size, and an arbitrary fraction of an object's data can be stored at any data site.

Figure 2 shows that EPaxos is sub-optimal in two ways. First, to meet any particular bound on read latency, EPaxos imposes a significant cost overhead; in Figure 2(a), EPaxos requires at least 9 replicas to satisfy the lowest feasible read latency bound (40 ms), whereas the lower bound storage overhead is 4x. Recall that, greater the number of copies of data stored, higher the data transfer costs when reading and writing. Second, given a cost budget, the read latencies achievable with EPaxos are significantly higher than the lower bound; in Figure 2(b), where storage overhead is capped at 6x, we see that the minimum read latency achievable with EPaxos (80 ms) is twice the lower bound (40 ms).

Of course, a lower bound is just that; some of the tradeoffs that it deems feasible may potentially be unachievable. However, for the example in Figure 2 and across a wide range of configurations in Section 5, we show that PANDO comes close to matching the lower bound. We describe how next.

3 Design

The fundamental source of EPaxos's sub-optimality in trading off cost and latency is its reliance on replication. Replication-based approaches inflate the cost necessary to meet read latency goals because spreading an object's data across more sites entails storing an additional *full copy* at each of these sites. To enable latency versus cost tradeoffs that are closer to optimal, the key is to store a *portion* of an object's data at each data site, like in the lower bound.

Therefore, we leverage erasure coding, a data-agnostic approach which enables such flexible data placement while matching replication's fault-tolerance at lower cost [69]. For example, to tolerate $f = 1$ failures, instead of requiring at least $2f + 1 = 3$ replicas, one could use Reed-Solomon coding [60] to partition an object into $k = 2$ splits, generate $r = 2$ parity splits, and store one split each at $k + r = 4$ sites; any k splits suffice to reconstruct the object's data. Compared to replication, this reduces storage overhead to $2\times$, thus also reducing the number of copies of data transferred over the wide-area when reading or writing.

State-of-the-art implementations of erasure coding [9] require only hundreds of nanoseconds to encode or decode kilobyte-sized objects. This latency is negligible compared to wide-area latencies, which range from tens to hundreds of milliseconds. Moreover, the computational costs for encoding and decoding pale in comparison to costs for data transfers and storage operations (§ 5.1.3).

3.1 Impact of erasure coding on wide-area latency

While there exist a number of protocols which preserve linearizability on erasure-coded data [15, 24], they largely focus on supporting PUT/GET semantics. To support conditional updates, we consider how to enable consensus on erasure-coded data with a leaderless approach such as Paxos. We have one of two options.

One approach would be to extend one of several one-round variants of Paxos to work on erasure-coded data. However, most of these protocols require large quorums (e.g., a write would have to be applied to a super-majority [42] or even all [49] data sites), rendering them significantly worse than the lower bound. Whereas, extending EPaxos [51], which requires small quorums despite needing a single round, to be compatible with erasure-coded data is far from trivial given the complex mechanisms that it employs for failure recovery.

Therefore, we build upon the classic two-phase version of Paxos [40] and address associated latency overheads. In either phase, a writer (a front-end or its delegate) communicates with all the data sites of an object and waits for responses from a write quorum. In Phase 1, the writer discovers whether there already is a value for the version it is attempting to write and attempts to elect itself leader for this version. In Phase 2, it sends its write to all data sites. A write to a version succeeds only if, prior to its completion of both phases, no other writer has been elected the leader. If the leader fails during Phase 2 but the write succeeds at a quorum of data sites, subsequent leaders will adopt the existing value and use it as part of their Phase 2, ensuring that the value for any specific version never changes once chosen.

This natural application of Paxos on erasure-coded data, called RS-Paxos [53], is inefficient in three ways.

- **Two rounds of wide-area communication.** Any reduction in read latency achieved by enabling every front-end to read from a more proximate read quorum has twice the adverse effect on write latency. In Figure 2(b), we see that when the read latency bound is stringent (e.g., $\leq$ 100 ms), the minimum write latency bound satisfiable with RS-Paxos is twice that achievable with EPaxos. When the read latency bound is loose (e.g., $\geq$ 150 ms), write latency inflation with RS-Paxos is lower because the data sites are close to each other and front-ends benefit from delegation.

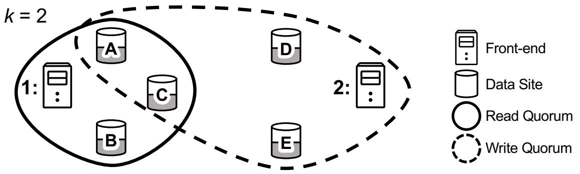


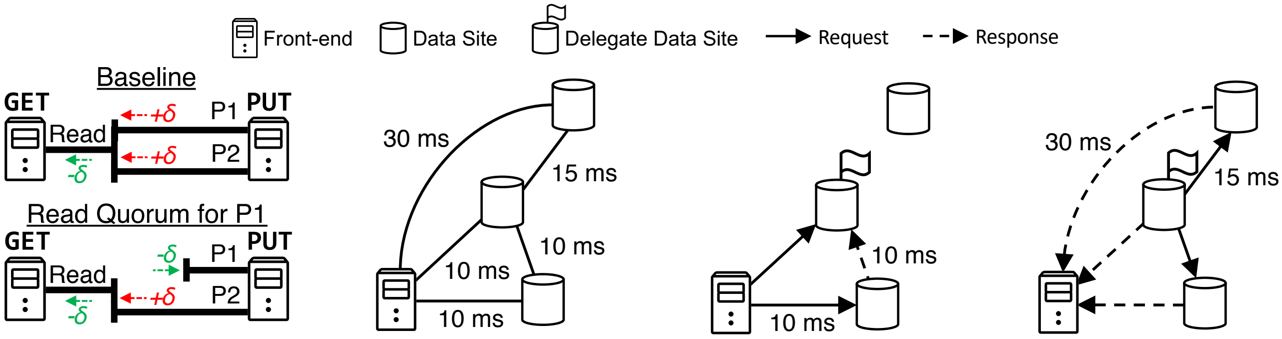
Figure 3: **Example execution of RS-Paxos on an erasure-coded object, whose data is partitioned into $k = 2$ splits. For all readers and writers to be able to reconstruct the last successful write, any write quorum must have an overlap of k or more data sites with every read and every write quorum.**

- **Increased impact of conflicts.** Executing writes in two rounds makes them more prone to performance degradation when conflicts arise. When multiple writes to the same key execute concurrently, none of the writes may succeed within two rounds. Either round of each write may fail at more than a quorum of data sites if other writes complete one of their rounds at those sites.
- **Larger intersections between quorums.** As we see in Figure 2(a), at storage overheads of 4x or more, the minimum read latency bound satisfiable with RS-Paxos is significantly higher than that achievable with EPaxos. This arises because, when an object's data is partitioned into k splits, every read quorum must have an overlap of at least k sites with every write quorum (see Figure 3). Thus, erasure coding's utility in helping spread an object's data across more sites (than feasible with replication for the same storage overhead) is nullified.

3.2 Overview of PANDO

What if these inefficiencies did not exist when executing Paxos on erasure-coded data? To identify the latency versus cost tradeoffs that would be achievable in this case, we consider a hypothetical ideal execution of Paxos on erasure-coded data: one which requires a single round of communication and can make do with an overlap of only one site between read–write and write–write quorum pairs. For the example used in Figure 2, this hypothetical ideal (not shown in the figure) comes close to matching the lower bound.

Encouraged by this promising result, we design PANDO to approximate this ideal execution of Paxos on erasure-coded data. First, we describe how to execute Paxos in two rounds on geo-distributed data, yet come close to matching the messaging delays incurred with one-round protocols. Second, leveraging the rarity of conflicts and failures in typical web service workloads, we describe how to make do with a single data site overlap between quorums in the common case. Finally, we discuss how to minimize performance degradation when conflicts do arise. In our description, we assume an object's data is partitioned into k splits.



(a) Small Phase 1 Quorums (b) Inter-DC Latencies (c) Phase 1 by Front-end (d) Phase 2 by Delegate

Figure 4: **(a) Reusing read quorums in Phase 1 of writes enables reduction in read latency without impacting (Phase 1 + Phase 2) latency for writes. (b) Example deployment with one-way delays between relevant pairs of data centers shown. Phase 1 quorum size is 2 and Phase 2 quorum size is 3. If same (Phase 2) quorum were used in both phases of a write, like in RS-Paxos, write latency would be 120 ms. (c) and (d) By directing Phase 1 responses to a delegate and having it initiate Phase 2, PANDO reduces write latency to 65 ms (20 ms in Phase 1 + 45 ms in Phase 2), close to the 60 ms latency feasible with one-round writes.**

3.3 Mitigating write latency

We reduce the latency overhead of executing Paxos in two rounds by revisiting the idea of delegation (§ 2.2): a front-end sends its write request to a stateless delegate, which executes Paxos and returns the response. When data sites are spread out (to enable low read latencies), two round-trips to a write quorum incurs comparable delay from the front-end versus from the delegate. The round-trip from the front-end to the delegate proves to be an overhead.

To mitigate this overhead, what if 1) transmission of the message from the front-end to the delegate overlaps with Phase 1 of Paxos, and 2) transmission of the response back overlaps with Phase 2? The latency for a front-end to execute the two-phase version of Paxos would then be roughly equivalent to one round-trip between the front-end and the delegate, thus matching the latency feasible with a one-round protocol. We show how to make this feasible in two steps.

3.3.1 Shrinking Phase 1 quorums

First, we revisit the property of classic Paxos that a writer needs responses from the same number of data sites in both phases of Paxos: the size of a write quorum. To ensure that a writer discovers any previously committed value, Paxos only requires that any Phase 1 quorum intersect with every Phase 2 quorum; Phase 1 quorums need not overlap [37]. In PANDO, we take advantage of this freedom to use a smaller quorum in the first phase of Paxos than in the second phase.

We observe that the intersection requirements imposed on Phase 1 and Phase 2 quorums are precisely the properties required of read and write quorums: any read quorum must intersect with every write quorum, whereas no overlap between read quorums is required. Therefore, when executing Phase 1 of Paxos to write to an object, it suffices to get responses from a read quorum, thus allowing improvements in read latency to also benefit leader election. A writer (a front-end or its delegate) needs responses from a write quorum only when executing Phase 2.

Figure 4(a) illustrates the corresponding improvements in write latency. When a quorum of the same size is used in both phases of a write, a reduction of δ in the read latency bound results in a 2δ increase in the minimum satisfiable write latency bound (because of the need for read and write quorums to overlap). In contrast, our reuse of read quorums in the Phase 1 of writes ensures that spreading out data sites to enable lower read latencies has (roughly speaking) no impact on write latency; when read quorums are shrunk to reduce the read latency bound by δ, the increase of δ in Phase 2 latency (to preserve overlap between quorums) is offset by the decrease of δ in Phase 1 latency.

3.3.2 Partially delegating write logic

While our reuse of read quorums in Phase 1 of a write helps reduce write latency, Phase 2 latency remains comparable to a one-round write protocol. Therefore, the total write latency remains significantly higher than that feasible with one-round protocols.

PANDO addresses this problem via *partial* use of delegation. Rather than having a front-end executing a write either do all the work of executing Paxos itself or offload all of this work to a delegate, we offload *some* of it to a delegate.

Figures 4(c–d) show how this works in PANDO. A front-end initiates Phase 1 of Paxos by sending requests to data sites of the object it is writing to, asking them to send their responses to a chosen delegate. In parallel, the front-end sends the value it wants to write directly to the delegate. Once the delegate receives enough responses (i.e., the size of a read quorum), it will either inform the front-end that Phase 1 failed (the rare case) or initiate Phase 2 (the common case), sending the value to be written to all data sites for the object. Those data sites in turn send their responses directly back to the front-end, which considers the write complete once it receives responses from a write quorum.

Note that partial delegation preserves Paxos's fault tolerance guarantees. To see why, consider the case where a end-user's client sends the same request to two front-ends—perhaps due to suspecting that the first front-end has failed—and both front-ends execute the request. Paxos guarantees that at most one of these writes will succeed. Similarly, with partial delegation, in the rare case when the front-end suspects that the delegate is unavailable, it can simply re-execute both phases on its own. Paxos will resolve any conflicts and at most one of the two writes (one executed via the delegate and the other executed by the front-end) will succeed.

Thanks to the heterogeneity of latencies across different pairs of data centers, the use of small Phase 1 quorums combined with the delegation of Phase 2 eliminates most of the latency overhead of two-phase writes. In Figure 4(b-d), the two techniques reduce write latency down from 120 ms with classic Paxos to 65 ms with PANDO, only 5 ms higher than what can be achieved with a one-round protocol. The remaining overhead results from the fact that there still has to be some point of convergence between the two phases.

3.4 Enabling smaller quorums

The techniques we have described thus far lower the minimum write latency SLO that is satisfiable given an SLO for read latency. However, as we have seen in Figure 2(a), erasure coding inflates the minimum read latency SLO achievable given a cost budget (e.g., a bound on storage overhead). As discussed earlier in Section 3.1, this is due to the need for larger intersections between quorums when data is erasure-coded, as compared to when replicated.

Recall that the need for an intersection of k data sites between any pair of read and write quorums exists so that any read on an object will be able to reconstruct the last value written; at least k splits written during the last successful write will be part of any read quorum. Thus, linearizability is preserved even in the worst case when a write completes at the minimum number of data sites necessary to be successful: a write quorum. However, since concurrent writes are uncommon [48, 29] and data sites are rarely unavailable in typical cloud deployments [11, 2], most writes will be applied to all data sites. Therefore, in the common case, all data sites in any read quorum will reflect the latest write.

In PANDO, we leverage this distinction between the common case and the worst case to optimize read latency (and equivalently any write's Phase 1 latency, given that PANDO uses the same quorum size in both cases) as follows.

Read from smaller quorum in the common case. After issuing read requests to all data sites, a reader initially waits for responses from a subset which is 1) at least of size k and 2) has an intersection of at least one site with every write quorum; we refer to this as a Phase 1a quorum. In the common case, all k splits have the same version and at least one of them is marked committed; the read is complete in this case. An overlap of only one site with every write quorum suffices for the reader to discover the latest version of the object; at least one of the splits received so far by the reader will be one written by the last successful write to this object.

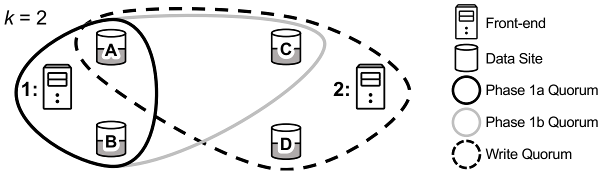


Figure 5: **For an object partitioned into $k = 2$ splits, PANDO requires an overlap of only one site between any Phase 1a and Phase 2 quorum. Responses from the larger Phase 1b quorum are needed only in the case of failure or conflict.**

Read from larger quorum if failure or conflict. At this juncture, if the last successful write has not yet been applied to all data sites, the reader may only know the latest version of the object but not the value of that version. To reconstruct that value, the reader must wait for responses from more data sites until the subset it has heard from has an overlap of k sites or more with every write quorum; this is a Phase 1b quorum. As a result, a reader must incur the latency penalty of waiting for responses from farther data sites only if the last successful write was executed when either some data sites were unavailable or a conflicting write was in progress.

In the example in Figure 5, Front-end 1 can complete reading based on responses from sites A and B in the common case since two splits suffice to reconstruct the object. If the last write was from Front-end 2 and this write completed only at a subset of sites, there are two cases to consider:

- If Front-end 2's write has been applied to a write quorum (say A, C, and D), then the response from site A will help Front-end 1 discover the existence of this write. Front-end 1 needs an additional response from C in this case to be able to reconstruct the value written by Front-end 2.
- If Front-end 2's write has been applied to less than a write quorum (say, A and D), then Front-end 1 may be unable to find k splits for this version even from a Phase 1b quorum (A, B, and C). In this case, that value could not have been committed to *any* Phase 2 quorum. Therefore, the reader falls back to the previous version. PANDO garbage collects the value for a version only once a value has been committed for the next version (§4). The overhead of storing multiple versions of a key will be short-lived in our target setting where failures and write conflicts are rare.

Phase 1a and 1b quorums can also be used as described above during the first round of a write. The only difference in the case of writes is that responses from data sites can be potentially directed to a delegate at a different data center than the one which initiates Phase 1.

To preserve correctness of both reads and writes, the minimum size of Phase 1a quorums must be $\max(k, f+1)$, and Phase 1b and Phase 2 quorums must contain at least $f + k$ data sites. These quorum sizes are inter-dependent because any Phase 1a quorum must have a non-empty overlap with every Phase 2 quorum and any Phase 1b or Phase 2 quorum must have an overlap of at least k sites with every Phase 2 quorum. For each of the three quorum types, all quorums of that type are of the same size and any subset of data sites of that size represent a valid quorum of that type.

Note that, if further reductions in common-case read latency are desired, one could use timed read leases as follows [21, 52]. Instead of using the normal read path, a front-end that holds a lease for a key could cache the value or fetch it from k nearby data sites to avoid the latency of communicating with a complete Phase 1a quorum. However, this approach would not benefit tail latency for reads and may increase latency for writes.

3.5 Reducing impact of conflicting writes

Lastly, we discuss how Pando mitigates performance degradation when conflicts arise. As mentioned before (§2), since conflicts rarely occur in practice [48, 29], we allow for violations of input latency bounds when multiple writes to a key execute concurrently. However, we ensure that the latency of concurrent writes is not arbitrarily degraded.

Our high-level idea is to select one of every key's data sites as the leader and to make use of this leader *only when conflicts arise*. Pando's leaderless approach helps satisfy lower latency bounds by eliminating the need for any front-end to contact a potentially distant leader. However, when a front-end's attempted write fails and it is uncertain whether a value has already been committed for this version, the front-end forwards its write to the leader. In contrast to the front-end retrying the write on its own, relying on the leader can ensure that the write completes within at most two rounds.

To make this work, we ensure that any write executed by a key's leader always supersedes writes to that key being attempted in parallel by front-ends. For this, we exploit the fact that front-ends always retry writes via the leader, i.e., any front-end will attempt to directly execute a write at most once. Therefore, when executing Paxos, we permit any front-end to use proposal numbers of the form (0, front-end's ID) but only allow the leader to set the first component to values greater than or equal to 1, so that its writes take precedence at every data site.

Note that, since we consider it okay to violate the write latency bound in the rare cases when conflicts occur, we do not require the leader to be close to any specific front-end. Therefore, leader election can happen in the background (using any of a number of approaches [23, 14]) whenever the current one fails. If conflicting writes are attempted precisely when the leader is unavailable, these writes will block until a new leader is elected. Like prior work [51, 40], Pando cannot bound worst-case write latency when conflicts *and* data center failures occur simultaneously.

A proof of Pando's correctness and a TLA+ specification are in Appendices A and B.

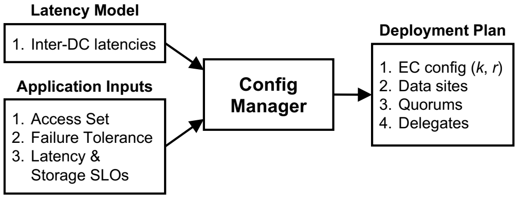


Figure 6: **Selecting a deployment plan with ConfigManager.**

4 Implementation

To empirically compare the manner in which different consensus approaches trade off read latency against write latency and cost, we implemented a key-value store which optimizes the selection of data sites for an object based on knowledge of how the object will be accessed.

ConfigManager. Central to this key-value store is the ConfigManager, which sits off the data path (thus not blocking reads and writes) and identifies *deployment plans*, one per access pattern. As shown in Figure 6, a deployment plan determines the number of splits k that the key's value is partitioned into, the number of redundant splits r, and the $k + r$ data sites at which these splits are stored; $k = 1$ corresponds to replication, and Reed-Solomon coding [60] is used when $k > 1$. The deployment plan also specifies the sizes of different quorum types and the choice of delegates (if any).

To make this determination, in addition to the application's latency, cost, and fault-tolerance goals, ConfigManager relies on the application to specify every key's *access set*: data centers from which front-ends are expected to issue requests for the key. An application can determine an object's access set based on its knowledge of the set of users who will access that object, e.g., in Google Docs, the access set for a document is the set of data centers from which the service will serve users sharing the document. When uncertain (e.g., for a public document), the access set can be specified as comprising all data centers hosting its front-ends; this uncertainty will translate to higher latencies and cost.

The ConfigManager selects deployment plans by solving a mixed integer program, which accounts for the particular consensus approach being used. For example, Pando's ConfigManager selects a delegate and preferred quorums per front-end, using RTT measurements to predict latencies incurred. Given bounds on any two dimensions of the tradeoff space, the ConfigManager can optimize the third (e.g. minimize max read latency across front-ends given write latency and storage cost SLOs). Given the stability of latencies observed between data centers in the cloud both in prior work [34] and in our measurements,[2] and since our current implementation assumes an object's access set is unchanged after it is created, we defer reconfiguration of an object's data sites [19] to future work.

[2] In six months of latency measurements between all pairs of Azure data centers, we observe less than 6% change in median latency from month to month for any data center pair and less than 10% difference between 90^{th} percentile and median latency within each month for most pairs.

Executing reads and writes. Unlike typical applications of Paxos, our use of erasure coding prevents servers from processing the contents of Paxos logs. Instead of separating application and Paxos state, we maintain one Paxos log for every key and aggressively prune old log entries. In order to execute a write request, a Proxy VM initiates Phase 1 of Paxos and waits for the delegate to run Phase 2. If the operation times out, the Proxy VM assumes the delegate has failed and executes both phases itself. Once Phase 2 successfully completes, the Proxy VM notifies the client and asynchronously informs learners so that they may commit their local state and garbage collect old log entries. The read path is simpler: a Proxy VM fetches the associated Paxos state and reconstructs the latest value before returning to the client. If the latest state happens to be uncommitted, then the Proxy VM issues a write-back to guarantee consistency.

5 Evaluation

We evaluate PANDO in two parts. First, in a measurement-based analysis, we estimate PANDO's benefits over prior solutions for enabling strongly consistent distributed storage. We quantify these benefits not only with respect to latency and cost separately, but also the extent to which PANDO helps bridge the gap to the lower bound in the latency–cost tradeoff space (§2). Second, we deploy our prototype key-value store and compare latency and throughput characteristics under microbenchmarks and an application workload. The primary takeaways from our evaluation are:

- Compared to the union of the best available replication- and erasure coding-based approaches, PANDO reduces the median gap to the lower bound by 88% in the read latency–write latency–storage overhead tradeoff space.
- Compared to EPaxos, given bounds on any two of storage overhead, read latency, and write latency, PANDO can improve read latency by 12–31% and reduce dollar costs (for storage, compute, and data transfers) by 6–46%, while degrading write latency by at most 3%.
- In a geo-distributed deployment on Azure, PANDO offers 18–62% lower read latencies than EPaxos and can reduce 95th percentile latency for two GitLab operations by 19–60% over EPaxos and RS-Paxos.

5.1 Measurement-based analysis

Setup. Our analysis uses network latencies between all pairs of 25 Microsoft Azure data centers. We categorize access sets (the subset of data centers from which an object is accessed) into four types: North America (NA), North America & Europe (NA-EU), North America & Asia (NA-AS), and Global (GL). For NA and NA-EU, we use 200 access sets chosen randomly. For NA-AS and GL, we first filter front-end data centers so that they are at least 20 ms apart, and then sample 200 random access sets. In all cases, we consider all 25 Azure data centers as potential data sites.

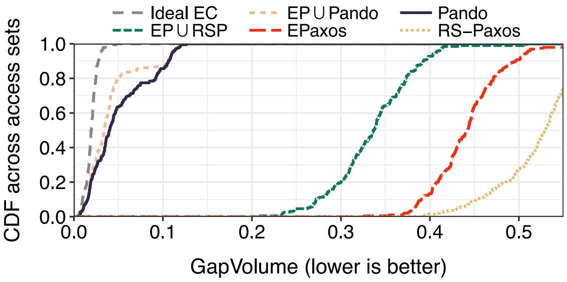


Figure 7: **For NA-AS access sets, comparison of GapVolume with PANDO to EPaxos and RS-Paxos individually and their union (EP ∪ RSP). In addition, we evaluate EP ∪ PANDO (the union of EPaxos and PANDO) and Ideal EC (a hypothetical Paxos variant that supports erasure coding, one-round writes, and 1-split intersection across quorums).**

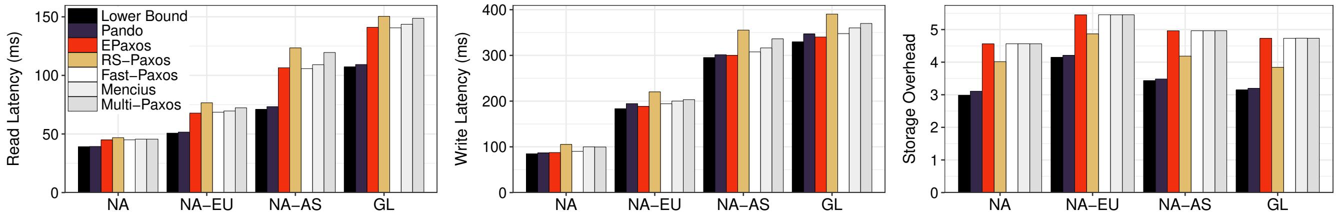
We compare PANDO to four replication-based approaches (EPaxos [51], Fast Paxos [42], Mencius [49], and Multi-Paxos [40]) and the only prior approach which can enable conditional updates on erasure-coded data (RS-Paxos [53]). We refer to the union of EPaxos and RS-Paxos (i.e., use either approach to satisfy the desired SLOs) as EP ∪ RSP.

Metrics. Our analysis looks at three types of metrics: 1) read and write latency (in either case, we estimate the max latency seen by any front-end in the access set) and storage overhead (size of the data stored divided by size of user data); 2) *GapVolume*, a metric which captures the gap in the three-dimensional read latency–write latency–storage overhead tradeoff space between the lower bound (described in §2) and the approach in question; and 3) total dollar cost as the sum of compute, storage, data transfer, and operation costs necessary to support reads and writes.

5.1.1 Impact on Achievable Tradeoffs

We use GapVolume to evaluate how close each approach is to the lower bound (§2.3). For any access set, we compute GapVolume with a specific consensus approach as the gap in the (read, write, storage) tradeoff space between the surfaces represented by the lower bound and by tradeoffs achievable with this consensus approach. We normalize this gap relative to the volume of the entire theoretically feasible tradeoff space, i.e., the portion of the tradeoff space above the lower bound surface. For every access set, we cap read and write latencies at values that are achievable with all approaches, and we limit storage overhead to a maximum of 7 as higher values are unlikely to be tenable in practice.

Proximity to lower bound. Figure 7 shows that PANDO significantly reduces GapVolume compared to EPaxos and RS-Paxos for access sets of type NA-AS. We do not show results for other replication-based approaches because they are subsumed by EPaxos, i.e., every combination of SLOs that is achievable with Mencius, Fast Paxos, and Multi-Paxos is also achievable with EPaxos. PANDO lowers median GapVolume to 4%, compared to 53% with RS-Paxos and 44% with EPaxos. Even with EP ∪ RSP (i.e., use two



(a) Read Latency (b) Write Latency (c) Storage Overhead

Figure 9: **Average performance across different metrics. Lower is better in all plots. For each metric, we pick SLO combinations for the other two metrics that are achievable with all approaches. For each such SLO pair, we estimate the minimum value of the metric achievable with each approach. We then take the geometric mean across all access sets and SLO pairs.**

GapVolume	NA	NA-EU	NA-AS	GL
PANDO	0.06	0.07	0.04	0.07
EP ∪ RSP	0.37	0.40	0.34	0.34
EPaxos	0.44	0.48	0.44	0.49
RS-Paxos	0.52	0.59	0.53	0.48

Table 1: **GapVolume for median access set of various types. Lower values are better; imply closer to the lower bound.**

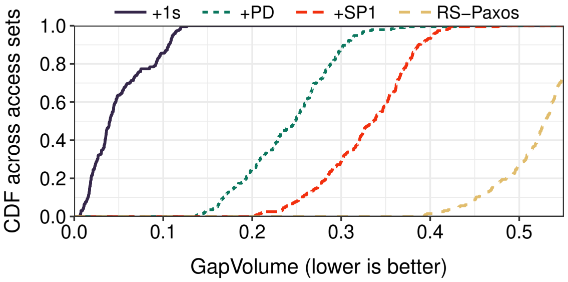


Figure 8: **For access sets of type NA-AS, contributions of each of PANDO's techniques in reducing GapVolume. SP1 = small Phase 1, PD = partial delegation, 1s = 1-split overlap.**

significantly different designs to realize different tradeoffs), median GapVolume remains at 34%. Table 1 shows similar benefits for NA, NA-EU, and GL access sets.

Moreover, EP ∪ PANDO (i.e., SLO combinations achievable with any of EPaxos or PANDO) is only marginally closer to the lower bound (i.e., has lower GapVolume) than PANDO, and that too only for some access sets. The few SLO combinations that EPaxos can achieve but not PANDO all have low write latency SLOs, in which case no choice of delegate can help PANDO overcome the overheads of two-round writes.

Utility of individual techniques. Figure 8 shows that each of the techniques used in PANDO contribute to the GapVolume reductions. For the median access set, using small Phase 1 quorums reduces GapVolume over RS-Paxos by 36%, adding partial delegation reduces GapVolume by a further 16%, and finally incorporating 1-split intersection reduces GapVolume by an additional 39%. When examining the improvements for each access set, we observe that both small Phase 1 quorums and 1-split intersection help across all access sets by reducing quorum size requirements. Similarly, we find that partial delegation typically improves GapVolume, indicating that some data sites are often closer to Phase 1 and Phase 2 quorums than the front-end.

Obstacles to matching the lower bound. From the gap between PANDO and Ideal EC in Figure 7, we surmise that most of the remaining gap between PANDO and the lower bound could be closed if one-round writes on erasure-coded data were feasible. Addressing any potential sub-optimality thereafter likely requires realizing the lower bound's flexibility with regards to varying the fraction of an object's data across sites (e.g., by using a different erasure coding strategy than Reed-Solomon coding) and varying quorum sizes across front-ends.

5.1.2 Latency and Storage Improvements

Figure 9 examines improvements in each of read latency, write latency, and storage overhead independently. To do this for read latency, we first identify all (write, storage) SLO pairs that are achievable by all candidate approaches. For each such pair, we then estimate the lowest read latency bound that is satisfiable with each approach. We take the geometric mean [31] across all feasible (write, storage) SLO pairs for all access sets to compare PANDO's performance relative to other approaches. We perform similar computations for write latency and storage overhead.

We find that PANDO achieves 12–31% lower read latency, 0–3% higher write latency, and 22–32% lower storage overhead than EPaxos across all types of access sets. Although PANDO executes writes in two phases, the use of small Phase 1 quorums plus partial delegation provides similar write latency as EPaxos. In all cases, EPaxos outperforms Fast Paxos, Mencius, and Multi-Paxos. Compared to RS-Paxos, PANDO reduces read latency by 15–40%, write latency by 11–17%, and storage overhead by 13–22%.

Latency under failures. Figure 10 compares the read latency bounds satisfiable with PANDO and EPaxos when any one data center is unavailable. During failures, a front-end may need to contact more distant data sites in order to read or write data. In this case, for the median access set, we observe that PANDO supports a read latency bound which is 110 ms lower than EPaxos. Since erasure coding spreads data more widely than replication for the same storage overhead, there are more nearby sites to fall back on when a failure occurs.

However, erasure coding is not universally helpful in failure scenarios. Upon detecting the loss of its write delegate, a PANDO front-end will locally identify a new one that min-

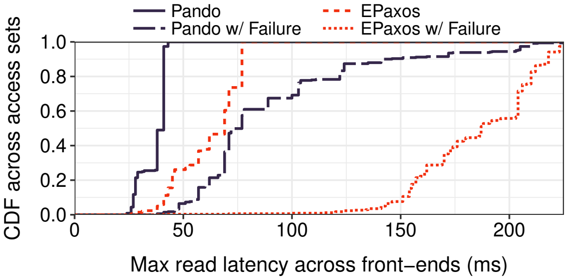


Figure 10: **For access sets of type NA-AS, impact of data center failures on read latency for PANDO and EPaxos (300 ms write SLO, 5× overhead storage SLO).**

imizes latency at the front-end. Still, across NA-AS access sets, median write latency with PANDO is 10% higher than EPaxos when any one data site is unavailable, despite the two approaches having similar latency in the failure-free case. In addition, under permanently data loss, bringing up a replacement data site requires decoding the data of k separate sites instead of fetching the same volume of data from one replica.

5.1.3 Cost

Beyond storage, public cloud providers also charge users for wide-area data transfers, PUT/GET requests to storage, and for virtual machines used to execute RPCs and encode/decode data. These overheads have driven production systems to adopt two key optimizations. First, replication-based systems execute reads by fetching version numbers from remote sites, not data. Second, Paxos-based systems issue writes only to a quorum (or for PANDO, a superset of a write quorum that intersects with all Phase 1a quorums likely to be used). Taking these optimizations into consideration, we now account for these other sources of cost and evaluate PANDO's utility in reducing total cost.

We considered 200 access sets of type NA-AS and set latency SLOs that both RS-Paxos and EPaxos are capable of meeting: 100 ms for read latency and 375 ms for write latency. We derived the CPU cost of Proxy VMs by measuring the throughput achieved in deployments of our prototype system. Using pricing data from Azure CosmosDB [3], we estimated the cost necessary to store 10 TB of data and issue 600M requests, averaged across all access sets; these parameters are based on a popular web service's workload [7] and a poll of typical MySQL deployment sizes [10].

Across several values for mean object size and read-to-write ratio, Figure 11 shows that PANDO reduces overall costs by 6–46% over EPaxos and 35–40% over RS-Paxos. When objects are large, PANDO's cost savings primarily stem from the reduction in the data transferred over the wide-area network. Note that even though EPaxos uses replication, it still requires reading remote data when a copy is not stored at the front-end data center. Whereas, when objects are small, storage fees dominate and PANDO reduces cost primarily due to the lower storage overhead that it imposes. Though erasure coding increases the number of requests to storage compared to replication, ConfigManager opts to erasure-code data only when the corresponding decrease in storage and data transfer costs help reduce overall cost. Unlike write requests, which have to first write metadata to storage before transferring and writing the data itself, read requests only issue storage operations to fetch data. This leads to greater cost reductions for read-dominated workloads.

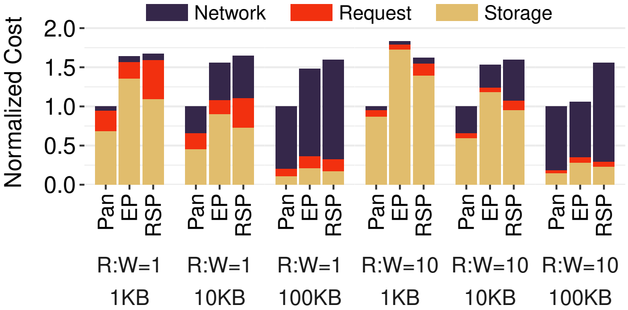


Figure 11: **Comparison of cost for a month in NA-AS to store 10 TB of data and execute 600M requests/month. In all cases, the costs of Proxy VMs (not shown) were negligible, and read and write latency SLOs were set to 100 ms and 375 ms.**

5.2 Prototype deployment

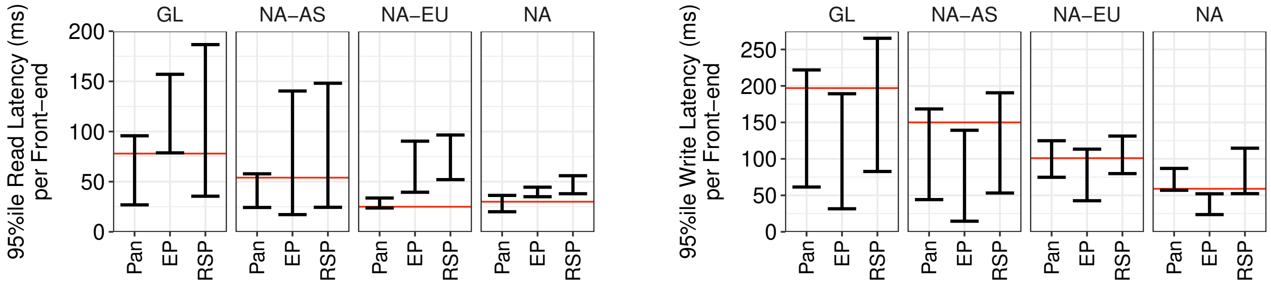
Next, via deployments on Azure, we experimentally compare PANDO versus EPaxos and RS-Paxos. We use our implementations of PANDO and RS-Paxos and the open-source implementation of EPaxos [50]. This experimental comparison helps account for factors missing from our analysis, such as latency variance and contention between requests. We consider one access set of each of our 4 types:

- NA: Central US, East US, North Central US, West US
- NA-EU: Canada East, Central US, North Europe, West Europe
- NA-AS: Central US, Japan West, Korea South
- GL: Australia East, North Europe, SE Asia, West US

Informed by prior studies of production web service workloads [29, 22], we read and write objects between 1–100 KB in size. Unless stated otherwise, we use A1v2 (1 CPU, 2 GB memory) virtual machines and issue requests using YCSB [28]—a key-value store benchmark.

5.2.1 Microbenchmarks

Tail latency across front-ends. Figure 12 shows 95th percentile read and write latencies for each of the four access sets when running a low contention (zipfian coefficient = 0.1) workload with 1 KB values and a read:write ratio of 1. In all cases, when using PANDO, we observe that the slowest front-end performs similarly to the read latency SLO deemed feasible by ConfigManager. This confirms the low latency variance in the CosmosDB instance at each data center and on the network paths between them. While all approaches achieve sub-55 ms read latency in NA, only PANDO can provide sub-100 ms latency in all regions. In GL, NA-AS, and



(a) Read Latency when Write SLO = 300, 300, 150, and 100 ms for GL, NA-AS, NA-EU, and NA, respectively

(b) Write Latency when Read SLO = 200, 150, 125, and 75 ms for GL, NA-AS, NA-EU, and NA, respectively

Figure 12: **Latency comparison with a low contention workload under a storage SLO of 3× overhead. Red lines represent the lowest latency SLO that ConfigManager identifies as feasible with PANDO. With every approach, in each access set, we measure 95th %ile latency at every front-end and plot the min and max of this value across front-ends. Pan = Pando, EP = EPaxos, and RSP = RS-Paxos.**

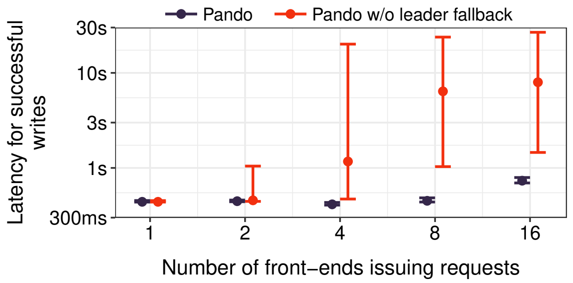


Figure 13: **Write latency comparison under contention using a fully leaderless approach and the leader-based fallback (§3.5). 5th percentile, median, and 95th percentile across 1000 writes are shown. Note logscale on y-axis.**

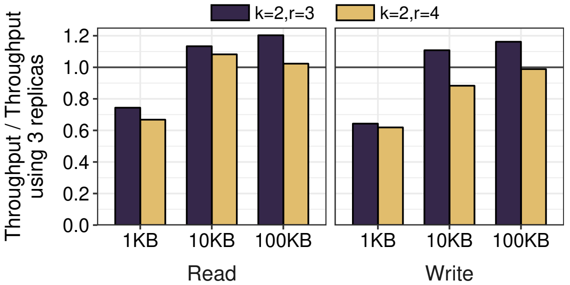


Figure 14: **Per-machine throughput of different erasure coding configurations compared to using 3 replicas.**

NA-EU, PANDO improves read latency for the slowest front-end by 39–62% compared to EPaxos. PANDO falls short of the write latency offered by EPaxos but comes close.

Latency under high conflict rates. Although our focus is on workloads with few write conflicts, we seek to bound performance degradation when conflicts occur. To evaluate this, we setup front-ends in 16 Azure data centers spread across five continents. We mimic conflicts by synchronizing a subset of front-ends (assuming low clock skew) to issue writes on the same key and version simultaneously. We show latency for successful conditional writes since other writes will learn the committed value and terminate shortly afterward.

Figure 13 shows that PANDO is effective at bounding latency for writes in the presence of conflicts. Without a leader-based fallback, writes in PANDO may need to be tried many times before succeeding, resulting in unbounded latency growth, e.g., with four concurrent writers, we observe more than 15 proposals for particular (key, version) pairs. In contrast, falling back to a leader ensures that a write succeeds within two write attempts.

Read and write throughput. While erasure coding can decrease bandwidth usage compared to replication, it requires additional computation in the form of coding/decoding and messaging overhead. We quantify the inflection point at which CPU overheads dominate by deploying PANDO in a single data center and measuring the achievable throughput with all data in memory. Each server, which stored 1 split or 1 replica, had two Xeon Silver 4114 processors and 192 GB of memory. All servers were connected over a 10 Gbps network with full bisection bandwidth. Across multiple value sizes, we measured the per-server throughput of filling the system with over 20 GB of data and reading it back.

Figure 14 compares the per-machine throughput achieved with 3 replicas to two erasure coding configurations, one with the same storage overhead and another with lower storage overhead. When objects are 10 KB or larger, we find that bandwidth is the primary bottleneck. Because it has identical bandwidth demands as replication, the $(k = 2, r = 4)$ configuration achieves similar read throughput and 0.9–1× the write throughput of replication for objects larger than 10 KB. Whereas, due to its lower bandwidth consumption, the $(k = 2, r = 3)$ configuration offers 1.1-1.2× the throughput of replication for 10 KB–100 KB sized objects. All configurations are CPU-bound with value sizes of 1 KB or smaller. Since replication requires exchanging fewer messages per request than erasure coding, it has lower CPU overhead and can thus achieve higher throughput.

5.2.2 Application Workload

Lastly, we evaluate the utility of PANDO on a geo-distributed deployment of GitLab [6], a software development application that provides source code management, issue tracking, and continuous integration.

Operations and setup. We evaluate the performance of two GitLab operations: listing issues targeting a devel-

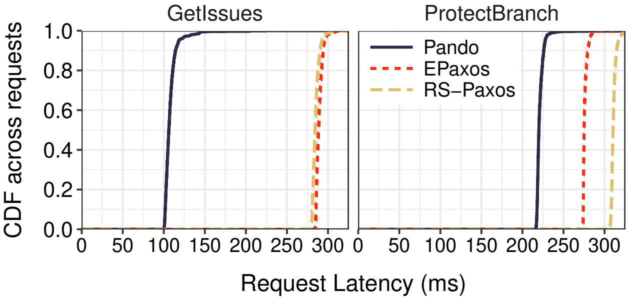


Figure 15: **Latencies for GitLab requests in Central US.**

opment milestone (GetIssues) and (un-)protecting a branch from changes (ProtectBranch). GetIssues fetches a list of issues for the requested milestone and then fetches 20 issues in parallel to display on a page. ProtectBranch reads the current branch metadata then updates its protection status.

We deployed front-ends and storage backends in the NA-AS access set on A2v2 (2 CPU, 4 GB memory) virtual machines, and preloaded the system with 100 projects, each with 20 branches, 10 milestones, and 100 issues. We used a 3× bound on storage overhead and set the write latency SLO to 175 ms. Every front-end executed 1000 GetIssues and ProtectBranch requests in an open loop and selected items using a uniform key distribution.

Performance. Figure 15 shows the latency distribution observed for both operations by the front-end in Central US. PANDO reduces 95th percentile GetIssues latency by over 59% compared to both EPaxos and RS-Paxos. Because ProtectBranch consists of a write in addition to a read operation, it incurs higher latency compared to GetIssues, which consists solely of read operations. Despite this, PANDO is able to lower 95th percentile ProtectBranch latency by 19% over EPaxos and 28% over RS-Paxos.

6 Related work

Geo-distributed storage. While some prior geo-distributed storage systems [46, 45, 47, 66] weaken consistency semantics to minimize latencies and unavailability, PANDO follows others [29, 64, 71, 21] in serving the needs of applications that cannot make do with weak consistency. Compared to efforts focused solely on minimizing latency with any specific replication factor [51, 49, 42], PANDO aims to also minimize the cost necessary to meet latency goals. Unlike systems [13, 70] which focus only on judiciously placing data to minimize cost, we also leverage erasure coding and rethink how to enable consensus on erasure-coded data.

Partial delegation in PANDO is akin to the chaining of RPCs [63] to eliminate wide-area delays. We show that combining this technique with the use of smaller quorums in Phase 1 of Paxos helps a two-round execution approximate the latencies achievable with one-round protocols in a geo-distributed setting.

Erasure-coded storage. Erasure coding has been widely used for protecting data from failures [69], most notably in RAID [58]. While PANDO leverages Reed Solomon codes [60] for storage across data centers, other codes have been used to correct errors in DRAM [33], transmit data over networks [62], and efficiently reconstruct data in cloud storage [67, 38, 61]. In contrast to the typical use of erasure coding for immutable and/or cold data [54, 32, 59, 55, 27], PANDO supports the storage of hot, mutable objects.

Previous protocols [15, 24] that support strong consistency with erasure-coded data provide only atomic register semantics or require two rounds of communication [53]. We show how to enable consensus on geo-distributed erasure-coded data without sacrificing latency. Some systems [26, 27] support strong consistency by erasure coding data but replicating metadata. We chose to not pursue this route to avoid the complexity of keeping the two in sync, as well as to minimize latency and metadata overhead.

Paxos variants. Many variants of Paxos [40] have been proposed over the years [53, 37, 43, 41], including several [51, 42, 49] which enable low latency geo-distributed storage. Compared to Paxos variants that reduce the number of wide-area round trips [51, 49, 42], PANDO lowers latency by reducing the magnitude of delay in each round trip.

Flexible Paxos [37] was the first to observe that Paxos only requires overlap between every Phase 1–Phase 2 quorum pair, and others [16, 56] have leveraged this observation since. All of these approaches make *Phase 2 quorums smaller*, so as to improve throughput and common case latency in settings with high spatial locality. In PANDO, we instead *reduce the size of Phase 1 quorums and reuse these quorums for reads*, thereby enabling previously unachievable tradeoffs between read and write latency bounds in a workload-agnostic manner.

Compression. Data compression is often used to lower the cost of storing data [36, 65, 25] or transferring it over a network [30]. In contrast to erasure coding, the effectiveness of compression depends on both the choice of compression algorithm used as well as the input data [17]. Compression and erasure coding are complementary as data can be compressed and then erasure-coded or vice-versa.

7 Conclusion

Today, geo-distributed storage systems take for granted that data must be replicated across data centers. In this paper, we showed that it is possible to leverage erasure coding to significantly reduce costs while successfully mitigating the associated overheads in wide-area latency incurred for preserving consistency. The key is to rethink how consensus is achieved across the wide-area. Importantly, we showed that the latency versus cost tradeoffs achievable with our approach for enabling consensus, PANDO, are close to optimal.

Acknowledgments. This work was supported in part by the NSF under grants CNS-1563095, CNS-1563849, CNS-1617773, and CNS-1900665.

References

[1] 100% uptime anybody? http://www.riskythinking.com/articles/article8.php.

[2] And the cloud provider with the best uptime in 2015 is ... http://www.networkworld.com/article/3020235/cloud-computing/and-the-cloud-provider-with-the-best-uptime-in-2015-is.html.

[3] Azure Cosmos DB - globally distributed database service — Microsoft Azure. https://azure.microsoft.com/en-us/services/cosmos-db/.

[4] Google Docs. https://docs.google.com.

[5] Latency is everywhere and it costs you sales - how to crush it. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[6] The only single product for the complete devops lifecycle - GitLab. https://about.gitlab.com/.

[7] Quizlet.com audience insights - Quantcast. https://www.quantcast.com/quizlet.com#trafficCard.

[8] ShareLaTeX. https://sharelatex.com.

[9] Storage - Intel ISA-L — Intel software. https://software.intel.com/en-us/storage/ISA-L.

[10] What is the largest amount of data do you store in MySQL? - Percona database performance blog. https://www.percona.com/blog/2012/11/09/what-is-the-largest-amount-of-data-do-you-store-in-mysql/.

[11] Which cloud providers had the best uptime last year? http://www.networkworld.com/article/2866950/cloud-computing/which-cloud-providers-had-the-best-uptime-last-year.html.

[12] SLA for Azure Cosmos DB. https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/v1_3/, 2019.

[13] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, 2010.

[14] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *International Symposium on Distributed Computing*, pages 108–122. Springer, 2001.

[15] M. K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *DSN*, 2005.

[16] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar. Multileader WAN paxos: Ruling the archipelago with fast consensus. *CoRR*, 2017.

[17] J. Alakuijala, E. Kliuchnikov, Z. Szabadka, and L. Vandevenne. Comparison of brotli, deflate, zopfli, lzma, lzham and bzip2 compression algorithms. *Google Inc.*, 2015.

[18] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *ICSE*, 1976.

[19] M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *OSDI*, 2014.

[20] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[21] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[22] 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. In *USENIX ATC*, 2013.

[23] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[24] V. R. Cadambe, N. Lynch, M. Médard, and P. Musial. A coded shared atomic memory algorithm for message passing architectures. *Distributed Computing*, 30(1):49–73, 2017.

[25] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[26] H. Chen, H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan, and B. Zang. Efficient and available in-memory kv-store with hybrid erasure coding and replication. *ACM Transactions on Storage (TOS)*, 13(3):25, 2017.

[27] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure coding objects across global data centers. In *USENIX ATC*, 2017.

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[29] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[30] R. Fielding and J. Reschke. RFC 7230: Hypertext transfer protocol (HTTP/1.1): Message syntax and routing. *Internet Engineering Task Force (IETF)*, 2014.

[31] P. J. Fleming and J. J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *CACM*, 1986.

[32] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *NSDI*, 2005.

[33] R. W. Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[34] O. Haq, M. Raja, and F. R. Dogar. Measuring and improving the reliability of wide-area cloud paths. In *WWW*, 2017.

[35] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), 1991.

[36] D. R. Horn, K. Elkabany, C. Lesniewski-Lass, and K. Winstein. The design, implementation, and deployment of a system to transparently compress hundreds of petabytes of image files for a file-storage service. In *NSDI*, 2017.

[37] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum intersection revisited. In *OPODIS*, 2016.

[38] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure storage. In *USENIX ATC*, 2012.

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

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

[41] L. Lamport. Generalized consensus and paxos. 2005.

[42] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[43] L. Lamport and M. Massa. Cheap paxos. In *DSN*, 2004.

[44] B. Lampson. The abcd's of paxos. In *PODC*, 2001.

[45] C. Li, D. Porto, A. Clement, J. Gehrke, N. M. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, 2012.

[46] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[47] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, 2013.

[48] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: Measuring and understanding consistency at Facebook. In *SOSP*, 2015.

[49] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. In *OSDI*, 2008.

[50] I. Moraru. Epaxos. `https://github.com/efficient/epaxos`, 2014. commit 791b115.

[51] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *SOSP*, 2013.

[52] I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *SoCC*, 2014.

[53] S. Mu, K. Chen, Y. Wu, and W. Zheng. When Paxos meets erasure code: Reduce network and storage cost in state machine replication. In *HPDC*, 2014.

[54] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's warm BLOB storage system. In *OSDI*, 2014.

[55] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. OceanStore: An architecture for global-scale persistent storage. In *OSDI*, 2014.

[56] F. Nawab, D. Agrawal, and A. El Abbadi. DPaxos: Managing data closer to users for low-latency and mobile applications. In *SIGMOD*, 2018.

[57] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[58] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, 1988.

[59] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *OSDI*, 2016.

[60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[61] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. 2013.

[62] A. Shokrollahi. Raptor codes. *IEEE/ACM Transactions on Networking (TON)*, 14(SI):2551–2567, 2006.

[63] Y. J. Song, M. K. Aguilera, R. Kotla, and D. Malkhi. RPC chains: Efficient client-server communication in geodistributed systems. In *NSDI*, 2009.

[64] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[65] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, 2005.

[66] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.

[67] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, S. Hussain, and S. Nandi. Clay codes: Moulding MDS codes to yield an MSR code. In

FAST, 2018.

[68] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[69] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS*, 2002.

[70] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, 2013.

[71] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, 2013.

A The PANDO write protocol: specification and proof of correctness

In this section, we focus on how PANDO achieves consensus on a *single value* and prove that it matches the guarantees provided by Paxos. Other functionality used in our paper is layered on top of this base as follows:

- **Mutating values.** As with Multi-Paxos, we build a distributed log of values and run PANDO on each entry of the log. We only ever attempt a write for version i if we know that $i-1$ has already been chosen. This invariant ensures that the log is contiguous, and that all but possibly the latest version have been decided.
- **Partial delegation of writes.** One of the key optimizations used in PANDO is to execute Phase 1 and Phase 2 on different nodes (§3.3.2). We achieve this without sacrificing fault tolerance as follows. Each proposer is assigned an id (used for Lamport clocks), but we additionally assign a proposer id to each (proposer, delegate) pair. When executing a write using partial delegation, we simply direct responses accordingly, and have the proposer inform the delegate about which value to propose (unless one was recovered, in which case the delegate has to inform the proposer about the change). In case the delegate fails, a proposer can always choose to execute a write operation normally, and because it uses a different proposer id in this case, it will look as though the write from the proposer and the write from the (proposer, delegate) pair are writes from two separate nodes. We already prove (§A.1) that PANDO maintains consistency in this case.
- **One round reads.** As with other consensus protocols, we support (common-case) one-round reads by adding a third, asynchronous phase to writes that broadcasts which value was chosen and caches this information at each acceptor. Upon executing a read at a Phase 1a quorum, we check to see if any acceptor knows whether a value has already been chosen. If we find such a value, we try to reconstruct it and fall back on the larger Phase 1b quorum in case there are not enough splits present in the Phase 1a quorum. Otherwise, we follow the write path, but propose a value only if we were able to recover one (else none have been chosen). We maintain linearizability with this approach because the task of resolving uncertainty is done via the write path.
- **Fallback to leader.** In PANDO, front-ends directly execute writes unless a conflict is observed, in which case they defer the request to a leader (§3.5). From the perspective of the consensus protocol, the leader is just another proposer, so no consistency issues may arise even if multiple leaders exist. However, PANDO prevents non-leader front-ends from attempting writes more than once which can lead to unavailability if the leader fails. It is up to the leader election mechanism to quickly elect a new leader when the the current one fails.

$A.ppn$	Promised proposal no. stored at acceptor A
$A.apn$	Accepted proposal no. stored at acceptor A
$A.vid$	Accepted value id stored at acceptor A
$A.vlen$	Accepted value length stored at acceptor A
$A.vsplit$	Accepted value split stored at acceptor A
vid_v	Unique id for v, typically a hash or random number
$vlen_v$	Length of v (to remove padding)
$\text{Split}(v, A)$	(Computed on proposers) The erasure-coded split associated with acceptor A

Figure 16: Summary of notation.

PANDO's consistency and liveness properties rely on certain quorum constraints being met. We describe the constraints below under the assumption that data is partitioned into k splits (Constraint 3 needed only if Phase 1a quorums are used for reads):

1. The intersection of any Phase 1a and Phase 2 quorums contains at least 1 split.
2. The intersection of any Phase 1b and Phase 2 quorums contains at least k splits.
3. A Phase 1a quorum must contain at least k splits.
4. After f nodes fail, at least one Phase 1b and Phase 2 quorum must consist of nodes that are available.

Below is pseudocode for the PANDO write protocol.

Phase 1 (Prepare-Promise)

```
Proposer $P$ initiates a write for value $v$:
  1. Select a unique proposal number $p$ (typically done using Lamport clocks).
  2. Broadcast $\text{Prepare}(p)$ messages to all acceptors.

Acceptor $A$, upon receiving $\text{Prepare}(p)$ message from Proposer $P$:
  3. If $p > A.ppn$ then set $A.ppn \leftarrow p$ and reply $\text{Promise}(A.apn, A.vid, A.vlen, A.vsplit)$.
  4. Else reply NACK.

Proposer $P$, upon receiving $\text{Promise}$ messages from a Phase 1a quorum:
  5. If the values in all $\text{Promise}$ responses are NULL, then skip to Phase 2 with $v' \leftarrow v$.

Proposer $P$, upon receiving $\text{Promise}$ messages from a Phase 1b quorum:
```

6. Iterate over all Promise responses sorted in decreasing order of their apn.
    (a) If there are at least k splits for value w associated with apn, recover the value w (using the associated $vlen$ and $vsplit$s) and continue to Phase 2 with $v' \leftarrow w$.
7. If no value was recovered, continue to Phase 2 with $v' \leftarrow v$.

Phase 2 (Propose-Accept)

Proposer P, initiating Phase 2 to write value v' with proposal number p:

8. If no value was recovered in Phase 1, set $vid_{v'} = hash(v)$ (or some other unique number, see Figure 16). If a value was recovered, use the existing $vid_{v'}$.
9. Broadcast $\text{Propose}(p, vid_{v'}, vlen_{v'}, \text{Split}(v', A))$ messages to all acceptors.

Acceptor A, upon receiving $\text{Propose}(p, vid, vlen, vsplit)$ **from a Proposer P:**

10. If $p < A.ppn$ reply NACK
11. $A.ppn \leftarrow p$
12. $A.apn \leftarrow p$
13. $A.vid \leftarrow vid$
14. $A.vlen \leftarrow vlen$
15. $A.vsplit \leftarrow vsplit$
16. Reply $\text{Accept}(p)$

Proposer P, upon receiving $\text{Accept}(p)$ **messages from a Phase 2 quorum:**

17. P now knows that v' was chosen, and can check whether the chosen value v' differs from the initial value v or not.

A.1 Proof of correctness

Definitions. We let $\mathcal{A}$ refer to the set of all acceptors and use $\mathcal{Q}_a$, $\mathcal{Q}_b$, and $\mathcal{Q}_2$ refer to the sets of Phase 1a, Phase 1b, and Phase 2 quorums, respectively. Using this notation, we restate our quorum assumptions:

$$Q \subseteq \mathcal{A} \qquad \forall Q \in \mathcal{Q}_a \cup \mathcal{Q}_b \cup \mathcal{Q}_2 \tag{1}$$

$$|Q_a \cap Q_2| \geq 1 \qquad \forall Q_a \in \mathcal{Q}_a, Q_2 \in \mathcal{Q}_2 \tag{2}$$

$$|Q_b \cap Q_2| \geq k \qquad \forall Q_b \in \mathcal{Q}_b, Q_2 \in \mathcal{Q}_2 \tag{3}$$

Definition 1. *A value is* ***chosen*** *if there exists a Phase 2 quorum of acceptors that all agree on the identity of the value and store splits corresponding to that value.*

We now show that the PANDO write protocol provides the same guarantees as Paxos:

- **Nontriviality.** Any chosen value must have been proposed by a proposer.
- **Liveness.** A value will eventually be chosen provided that RPCs complete before timing out and all acceptors in at least one Phase 1b and Phase 2 quorum are available.
- **Consistency.** At most one value can be chosen.
- **Stability.** Once a value is chosen, no other value may be chosen.

Theorem 1. *(Nontriviality)* PANDO *will only choose values that have been proposed.*

Proof. By definition, a value can only be chosen if it is present at a Phase 2 quorum of acceptors. Values are only stored at acceptors in response to Propose messages initiated by proposers. □

Theorem 2. *(Liveness)* PANDO *will choose a a value provided that RPCs complete before timing out and all acceptors in at least one Phase 1b and Phase 2 quorum are available.*

Proof. Let t refer to the (maximum) network and execution latency for an RPC. Since PANDO has two rounds of execution, a write can complete within $2t$ as long as a requested is uncontended. If all proposers retry RPCs using randomized exponential backoff, a time window of length $\geq 2t$ will eventually open where only a Proposer P is executing. Since no other proposer is sending any RPCs during this time, both Phase 1 and Phase 2 will succeed for Proposer P. □

Following the precedence of [37], we will show that PANDO provides both consistency and stability by proving that it provides a stronger guarantee.

Lemma 1. *If a value v is chosen with proposal number p, then for any proposal with proposal number $p' > p$ and value v', $v' = v$.*

Proof. Recall that PANDO proposers use globally unique proposal numbers (Line 1); this makes it impossible for two different proposals to share a proposal number p. Therefore, if two proposals are both chosen, they must have different proposal numbers. If $v' = v$ then we trivially have the desired property. Therefore, assume $v' \neq v$.

Without loss of generality, we will consider the smallest p' such that $p' > p$ and $v' \neq v$ (*minimality assumption*). We will show that this case always results in a contradiction: either the Prepare messages for p' will fail (and thus no Propose messages will ever be sent) or the proposer will adopt and re-propose value v.

Let $Q_{2,p}$ be the Phase 2 quorum used for proposal number p, and $Q_{a,p'}$ be the Phase 1a quorum used for p'. By Quorum Property 2, we know that $|Q_{2,p} \cap Q_{a,p'}|$ is non-empty. We will now look at the possible ordering of events at each acceptor A in the intersection of these two quorums ($Q_{2,p}$ and $Q_{a,p'}$):

- Case 1: A receives $\text{Prepare}(p')$ before $\text{Propose}(p,\ldots)$.

  The highest proposal number at A would be $p' > p$ by the time $\text{Propose}(p,\ldots)$ was processed, and so A would reject $\text{Propose}(p,\ldots)$. However, we know that this is not the case since $A \in Q_{2,p}$, so this is a contradiction.

- Case 2: A receives $\text{Propose}(p,\ldots)$ before $\text{Prepare}(p')$.

  The last promised proposal number at A is q such that $p \leq q < p'$ ($q > p'$ would be a contradiction since $\text{Prepare}(p')$ would fail even though $A \in Q_{a,p'}$). By our minimality assumption, we know that all proposals z such that $p \leq z < p'$ fail or re-propose v. Therefore, the acceptor A responds with $\text{Promise}(q, vid_v, \ldots)$.

At this point, the proposer has received at least one Promise message with a non-empty value. Therefore, it does not take the Phase 1 fast path and waits until it has heard from a Phase 1b quorum (denoted $Q_{b,p'}$). Using the same logic as above, the proposer for p' will receive a minimum of k Promise messages each referencing value v since there are k acceptors in $Q_{b,p'} \cap Q_{2,p}$ (Quorum Property 3). Since the proposer has a minimum of k responses for v, it can reconstruct value v. Let q denote the highest proposal number among all k responses.

Besides those in $Q_{b,p'} \cap Q_{2,p}$, other acceptors in $Q_{b,p'}$ may return values that differ from v. We consider the proposal number q' for each of these accepted values:

- Case 1: $q' < q$. The proposer for p' will ignore the value for q' since it uses the highest proposal number for which it has k splits.

- Case 2: $p' < q'$. Not possible since $\text{Prepare}(p')$ would have failed.

- Case 3: $p < q' < p'$. This implies that a $\text{Propose}(q', v'')$ was issued where $v'' \neq v$. This violates our minimality assumption.

Therefore, the proposer will adopt value v since it can reconstruct it (the proposer has k splits from the acceptors in $Q_{b,p'} \cap Q_{2,p}$ alone) and the highest returned proposal number references it. This contradicts our assumption that $v' \neq v$. □

Theorem 3. *(Consistency)* PANDO *will choose at most one value.*

Proof. Assume that two different proposals with proposal numbers p and q are chosen. Since proposers use globally unique proposal numbers, $p \neq q$. This implies that one of the proposal numbers is greater than the other, assume that $q > p$. By Lemma 1, the two proposals write the same value. □

Theorem 4. *(Stability) Once a value is chosen by* PANDO, *no other value may be chosen.*

Proof. The proposal numbers used for any two chosen proposals will not be equal. Thus, with the additional assumption that acceptors store their state in durable storage, this follows immediately from Lemma 1. □

B TLA+ specification for PANDO reads and writes

In addition to our proof of correctness for PANDO's write path, we have model checked PANDO's correctness using TLA+ [39]. The purpose of this exercise was to *mechanically verify* PANDO's safety guarantees under a number of scenarios.

We checked the following invariants: consistency and stability for writes, that any value marked chosen at an acceptor was indeed chosen, and that successful reads only ever returned chosen values. The configurations modeled used 2–3 proposers (and readers) that could write (read) 2–3 values to (from) 4–6 acceptors when splitting the data into 2–4 splits. We set up 2–3 quorums of each type (Phase 1a, Phase 1b, and Phase 2).

The TLA+ model checker considers all possible histories including those with message reordering and arbitrary (or infinite) delay in delivering messages. When run on the specification for PANDO (below) and the configurations listed earlier, no invariant violations were found.

```
------------------------------------ MODULE Pando ------------------------------------
EXTENDS Integers, TLC, FiniteSets

CONSTANTS Acceptors, Ballots, Values,
          Quorum1a, Quorum1b, Quorum2, K

ASSUME QuorumAssumption ≜
          ∧ Quorum1a ⊆ SUBSET Acceptors
          ∧ Quorum1b ⊆ SUBSET Acceptors
          ∧ Quorum2  ⊆ SUBSET Acceptors
          Overlap of 1
          ∧ ∀ QA ∈ Quorum1a :
            ∀ Q2 ∈ Quorum2 :
              Cardinality(QA ∩ Q2) ≥ 1
          Overlap of K
          ∧ ∀ QB ∈ Quorum1b :
            ∀ Q2 ∈ Quorum2 :
              Cardinality(QB ∩ Q2) ≥ K

VARIABLES msgs,        The set of messages that have been sent
          maxPBal,     maxPBal[a] is the highest promised ballot (proposal number) at acceptor a
          maxABal,     maxABal[a] is the highest accepted ballot (proposal number) at acceptor a
          maxVal,      maxVal[a] is the value for maxABal[a] at acceptor a
          chosen,      chosen[a] is the value that acceptor a heard was chosen (or else is None)
          readLog      readLog[b] is the value that was read during ballot b

vars  ≜ ⟨msgs, maxPBal, maxABal, maxVal, chosen, readLog⟩
None  ≜ CHOOSE v : v ∉ Values

Type invariants.
Messages ≜
          [type : {"prepare"}, bal : Ballots]
    ∪     [type : {"promise"}, bal : Ballots, maxABal : Ballots ∪ { − 1},
           maxVal : Values ∪ {None}, acc : Acceptors,
           chosen  : Values ∪ {None}]
    ∪     [type : {"propose"}, bal : Ballots, val : Values ∪ {None},
           op : {"R", "W"}]
    ∪     [type : {"accept"}, bal : Ballots, val : Values, acc : Acceptors,
           op : {"R", "W"}]
    ∪     [type : {"learn"}, bal : Ballots, val : Values]

TypeOK ≜ ∧ msgs    ∈ SUBSET Messages
         ∧ maxABal ∈ [Acceptors → Ballots ∪ { − 1}]
         ∧ maxPBal ∈ [Acceptors → Ballots ∪ { − 1}]
         ∧ maxVal  ∈ [Acceptors → Values ∪ {None}]
```

```
        $\land\ chosen \quad \in [Acceptors \rightarrow Values \cup \{None\}]$
        $\land\ readLog \ \in [Ballots \rightarrow Values \cup \{None\}]$
        $\land\ \forall a \qquad \in Acceptors : maxPBal[a] \geq maxABal[a]$
```

Initial state.

```
$Init \triangleq \land\ msgs \qquad = \{\}$
          $\land\ maxPBal = [a \in Acceptors \mapsto -1]$
          $\land\ maxABal = [a \in Acceptors \mapsto -1]$
          $\land\ maxVal \quad = [a \in Acceptors \mapsto None]$
          $\land\ chosen \quad = [a \in Acceptors \mapsto None]$
          $\land\ readLog \ = [b \in Ballots \mapsto None]$
```

Send message m.

```
$Send(m) \triangleq msgs' = msgs \cup \{m\}$
```

Prepare: The proposer chooses a ballot id and broadcasts prepare requests to all acceptors.
All writes start here.

```
$Prepare(b) \triangleq \land\ \neg\exists\, m \in msgs : (m.type = \text{“prepare”}) \land (m.bal = b)$
                 $\land\ Send([type \mapsto \text{“prepare”},\ bal \mapsto b])$
                 $\land\ \text{UNCHANGED}\ \langle maxPBal,\ maxABal,\ maxVal,\ chosen,\ readLog\rangle$
```

Promise: If an acceptor receives a prepare request with ballot id greater than that of any prepare request which it has already responded to, then it responds to the request with a promise. The promise reply contains the proposal (if any) with the highest ballot id that it has accepted.

```
$Promise(a) \triangleq$
  $\exists\, m \in msgs :$
    $\land\ m.type = \text{“prepare”}$
    $\land\ m.bal > maxPBal[a]$
    $\land\ Send([type \mapsto \text{“promise”},\ acc \mapsto a,\ bal \mapsto m.bal,$
              $maxABal \mapsto maxABal[a],\ maxVal \mapsto maxVal[a],$
              $chosen \mapsto chosen[a]])$
    $\land\ maxPBal' = [maxPBal\ \text{EXCEPT}\ ![a] = m.bal]$
    $\land\ \text{UNCHANGED}\ \langle maxABal,\ maxVal,\ chosen,\ readLog\rangle$
```

Propose (fast path): The proposer waits until it collects promises from a Phase 1a quorum of acceptors. If no previous value is found, then the proposer can skip to Phase 2 with its own value.

```
$ProposeA(b) \triangleq$
  $\land\ \neg\exists\, m \in msgs : (m.type = \text{“propose”}) \land (m.bal = b)$
  $\land\ \exists\, v \in Values\ :$
    $\land\ \exists\, Q \in Quorum1a :$
      $\text{LET}\ Q1Msgs \triangleq \{m \in msgs : \land\ m.type = \text{“promise”}$
                                          $\land\ m.bal \quad = b$
                                          $\land\ m.acc \quad \in Q\}$
      $\text{IN}$
        Check for promises from all acceptors in Q
        $\land\ \forall\, a \in Q : \exists\, m \in Q1Msgs : m.acc = a$
        Make sure no previous vals have been returned in promises
        $\land\ \forall\, m \in Q1Msgs : m.maxABal = -1$
    $\land\ Send([type \mapsto \text{“propose”},\ bal \mapsto b,\ val \mapsto v,\ op \mapsto \text{“W”}])$
  $\land\ \text{UNCHANGED}\ \langle maxPBal,\ maxABal,\ maxVal,\ chosen,\ readLog\rangle$
```

Propose (slow path): The proposer waits for promises from a Phase 1b quorum of acceptors. If no value is found accepted, then the proposer can pick its own value for the next phase. If any accepted coded split is found in one of the promises, the proposer detects whether there are at least K splits (for the particular value) in these promises. Next, the proposer picks up the recoverable value with the highest ballot, and uses it for next phase.

```
$ProposeB(b) \triangleq$
  $\land\ \neg\exists\, m \in msgs : (m.type = \text{“propose”}) \land (m.bal = b)$
  $\land\ \exists\, Q \in Quorum1b :$
    $\text{LET}\ Q1Msgs \triangleq \{m \in msgs : \land\ m.type = \text{“promise”}$
```

```
                              $\land m.bal \ = b$
                              $\land m.acc \ \in Q\}$
        $Q1Vals \ \triangleq \ [v \ \in Values \cup \{None\} \mapsto$
                     $\{m \in Q1Msgs : m.maxVal = v\}]$
   IN
     Check that all acceptors from $Q$ responded
     $\land \forall a \in Q : \exists m \in Q1Msgs : m.acc = a$
     $\land \exists v \in Values :$
        $\land$ No recoverable value, use anything
           $\lor \forall vv \in Values : Cardinality(Q1Vals[vv]) < K$
           Check if $v$ is recoverable and of highest ballot
           $\lor$ Use previous value if $K$ splits exist
              $\land Cardinality(Q1Vals[v]) \geq K$
              $\land \exists m \in Q1Vals[v] :$
                   Ensure no other recoverable value has a higher ballot
                 $\land \forall mm \in Q1Msgs :$
                    $\lor m.bal \geq mm.bal$
                    $\lor Cardinality(Q1Vals[mm.maxVal]) < K$
        $\land Send([type \mapsto \text{"propose"}, bal \mapsto b, val \mapsto v, op \mapsto \text{"W"}])$
  $\land$ UNCHANGED $\langle maxPBal, maxABal, maxVal, chosen, readLog \rangle$
```

Phase 2: If an acceptor receives an accept request with ballot i, it accepts the proposal unless it has already responded to a prepare request having a ballot greater than it does.

```
$Accept(a) \ \triangleq$
  $\land \exists m \in msgs :$
    $\land m.type = \text{"propose"}$
    $\land m.bal \geq maxPBal[a]$
    $\land maxABal' = [maxABal$ EXCEPT $![a] = m.bal]$
    $\land maxPBal' = [maxPBal$ EXCEPT $![a] = m.bal]$
    $\land maxVal' \ \ = [maxVal$ EXCEPT $![a] \ = m.val]$
    $\land Send([type \mapsto \text{"accept"}, bal \mapsto m.bal, acc \mapsto a, val \mapsto m.val,$
              $op \mapsto m.op])$
  $\land$ UNCHANGED $\langle chosen, readLog \rangle$
```

ProposerEnd: If the proposer receives acknowledgements from a Phase 2 quorum, then it knows that the value was chosen and broadcasts this.

```
$ProposerEnd(b) \ \triangleq$
  $\land \exists v \in Values :$
    $\land \exists Q \in Quorum2 :$
      LET $Q2msgs \ \triangleq \ \{m \in msgs : \land m.type = \text{"accept"}$
                                $\land m.bal \ \ = b$
                                $\land m.val \ \ = v$
                                $\land m.acc \in Q\}$
       IN
         Check for accept messages from all members of $Q$
         $\land \forall a \in Q : \exists m \in Q2msgs : m.acc = a$
         If this was in response to a read, log the result
         $\land$ Read: log the result
            $\lor \land \exists m \in Q2msgs : m.op = \text{"R"}$
              $\land readLog' = [readLog$ EXCEPT $![b] = v]$
            Write: don't log the result
            $\lor (\forall m \in Q2msgs : m.op = \text{"W"} \land$ UNCHANGED $\langle readLog \rangle)$
    $\land Send([type \mapsto \text{"learn"}, bal \mapsto b, val \mapsto v])$
  $\land$ UNCHANGED $\langle maxABal, maxPBal, maxVal, chosen \rangle$
```

Learn: A proposer has announced that value v is chosen.

```
$Learn(a) \triangleq$
  $\land \; \exists\, m \in msgs :$
    $\land\; m.type = $ “learn”
    Process accept before learn, needed for ReadInv, not the protocol
    $\land\; maxABal[a] \geq m.bal$
    $\land\; chosen' = [chosen \text{ EXCEPT } ![a] = m.val]$
  $\land$ UNCHANGED $\langle msgs, maxPBal, maxABal, maxVal, readLog \rangle$
```

Count how many splits of v we have received.

```
$CountSplitsOf(resps, v) \triangleq Cardinality(\{m \in resps : m.maxVal = v\})$
```

FastRead: Check if any value returned from a Phase 1a quorum was chosen. If we have enough splits to reconstruct that value, then return immediately. If not, wait for Phase 1b quorum. If we have a value that was marked chosen, return. Otherwise, perform a write-back.

```
$FastRead(b) \triangleq$
  $\land\; \neg\exists\, m \in msgs : (m.type = $ “propose”$) \land (m.bal = b)$
  $\land$
    Fastest path: Phase 1a quorum has $k$ splits and the value is chosen
    $\lor\; \land\; \exists\, Q \in Quorum1a :$
          LET $RMsgs \triangleq \{m \in msgs :$ $\land\; m.type = $ “promise”
                                                $\land\; m.bal \;\; = b$
                                                $\land\; m.acc \in Q\}$
          IN    Check that all acceptors from $Q$ responded
                $\land\; \forall\, a \in Q : \exists\, m \in RMsgs : m.acc = a$
                Check that we have $k$ splits of a chosen value
                $\land\; \exists\, m \in RMsgs :$
                      $\land\; m.chosen \neq None$
                      $\land\; CountSplitsOf(RMsgs, m.chosen) \geq K$
                      $\land\; readLog' = [readLog \text{ EXCEPT } ![b] = m.chosen]$
       $\land$ UNCHANGED $\langle msgs, maxPBal, maxABal, maxVal, chosen \rangle$
    Fast path: Phase 1b quorum has $k$ splits and the value is chosen
    $\lor\; \land\; \exists\, Q \in Quorum1b :$
          LET $RMsgs \triangleq \{m \in msgs :$ $\land\; m.type = $ “promise”
                                                $\land\; m.bal \;\; = b$
                                                $\land\; m.acc \in Q\}$
          IN    Check that all acceptors from $Q$ responded
                $\land\; \forall\, a \in Q : \exists\, m \in RMsgs : m.acc = a$
                Check that we have $k$ splits of a chosen value
                $\land\; \exists\, m \in RMsgs :$
                      $\land\; m.chosen \neq None$
                      $\land\; CountSplitsOf(RMsgs, m.chosen) \geq K$
                      $\land\; readLog' = [readLog \text{ EXCEPT } ![b] = m.chosen]$
        $\land$ UNCHANGED $\langle msgs, maxPBal, maxABal, maxVal, chosen \rangle$
      Slow path: Phase 1b recovery and write back
      $\lor\; \land\; \exists\, Q \in Quorum1b :$
            LET $Q1Msgs \triangleq \{m \in msgs :$ $\land\; m.type = $ “promise”
                                                    $\land\; m.bal \;\; = b$
                                                    $\land\; m.acc \;\; \in Q\}$
                $Q1Vals \;\; \triangleq [v \;\; \in Values \cup \{None\} \mapsto$
                                $\{m \in Q1Msgs : m.maxVal = v\}]$
            IN
              Check that all acceptors from $Q$ responded
              $\land\; \forall\, a \in Q : \exists\, m \in Q1Msgs : m.acc = a$
              $\land\; \exists\, v \in Values :$
                    Check if $v$ is recoverable and of highest ballot
                    Use previous value if $K$ splits exist
```

```
                  $\land\ Cardinality(Q1Vals[v]) \geq K$
                  $\land\ \exists\, m \in Q1Vals[v]:$
                        Ensure no other recoverable value has a higher ballot
                        $\land\ \forall\, mm \in Q1Msgs:$
                              $\lor\ m.bal \geq mm.bal$
                              $\lor\ Cardinality(Q1Vals[mm.maxVal]) < K$
                  $readLog$ will be updated in ProposerEnd
                  $\land\ Send([type \mapsto \text{“propose”},\ bal \mapsto b,\ val \mapsto v,$
                              $op \mapsto \text{“R”}])$
          $\land$ UNCHANGED $\langle maxPBal,\ maxABal,\ maxVal,\ chosen,\ readLog\rangle$
        No value recovered: Return $None$
        $\lor\ \land\ readLog' = [readLog$ EXCEPT $![b] = None]$
          $\land$ UNCHANGED $\langle msgs,\ maxPBal,\ maxABal,\ maxVal,\ chosen\rangle$
```

Next state.

```
$Next \triangleq$  $\lor\ \exists\, b \in Ballots$      $:\ \lor\ Prepare(b)$
                                        $\lor\ ProposeA(b)$
                                        $\lor\ ProposeB(b)$
                                        $\lor\ ProposerEnd(b)$
                                        $\lor\ FastRead(b)$
                $\lor\ \exists\, a \in Acceptors$  $:\ Promise(a) \lor Accept(a) \lor Learn(a)$

$Spec \triangleq Init \land \Box[Next]_{vars}$
```

Invariant helpers.

```
$AllChosenWereAcceptedByPhase2 \triangleq$
  $\forall\, a \in Acceptors:$
     $\lor\ chosen[a] = None$
     $\lor\ \exists\, Q \in Quorum2:$
         $\forall\, a2 \in Q:$
            $\exists\, m \in msgs:$  $\land\ m.type = \text{“accept”}$
                              $\land\ m.acc\ \ = a2$
                              $\land\ m.val\ \ = chosen[a]$

$OnlyOneChosen \triangleq$
    $\forall\, a,\ aa \in Acceptors:$
        $(chosen[a] \neq None \land chosen[aa] \neq None) \implies (chosen[a] = chosen[aa])$

$VotedForIn(a,\ v,\ b) \triangleq \exists\, m \in msgs:$  $\land\ m.type = \text{“accept”}$
                                          $\land\ m.val\ \ = v$
                                          $\land\ m.bal\ \ = b$
                                          $\land\ m.acc\ \ = a$

$ProposedValue(v,\ b) \triangleq \exists\, m \in msgs:$  $\land\ m.type = \text{“propose”}$
                                          $\land\ m.val\ \ = v$
                                          $\land\ m.bal\ \ = b$
                                          $\land\ m.op\ \ \ = \text{“W”}$

$NoOtherFutureProposal(v,\ b) \triangleq$
  $\forall\, vv \in Values:$
     $\forall\, bb\ \ \in Ballots:$
        $(bb > b \land ProposedValue(vv,\ bb)) \implies v = vv$

$ChosenIn(v,\ b)\ \ \triangleq \exists\, Q \in Quorum2: \forall\, a \in Q: VotedForIn(a,\ v,\ b)$
$ChosenBy(v,\ b)\ \triangleq \exists\, b2 \in Ballots: (b2\ \ \leq b \land ChosenIn(v,\ b2))$
$Chosen(v)\ \triangleq \exists\, b \in Ballots: ChosenIn(v,\ b)$
```

Invariants.

$LearnInv \triangleq AllChosenWereAcceptedByPhase2 \land OnlyOneChosen$
$ReadInv \triangleq \forall\, b \in Ballots : readLog[b] = None \lor ChosenBy(readLog[b],\, b)$
$ConsistencyInv \triangleq \forall\, v1,\, v2 \in Values : Chosen(v1) \land Chosen(v2) \implies (v1 = v2)$

$$\begin{aligned}
&StabilityInv \triangleq \\
&\quad \forall\, v \in Values : \forall\, b \in Ballots : ChosenIn(v,\, b) \implies NoOtherFutureProposal(v,\, b)
\end{aligned}$$

$$\begin{aligned}
&AcceptorInv \triangleq \\
&\quad \forall\, a \in Acceptors : \\
&\qquad \land (maxVal[a] \;= None) \equiv (maxABal[a] = \,-1) \\
&\qquad \land maxABal[a] \leq maxPBal[a] \\
&\qquad \land (maxABal[a] \geq 0) \implies VotedForIn(a,\, maxVal[a],\, maxABal[a]) \\
&\qquad \land \forall\, c \in Ballots : \\
&\qquad\quad c > maxABal[a] \implies \neg\exists\, v \in Values : VotedForIn(a,\, v,\, c)
\end{aligned}$$