Addendum to "Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage"

1 Introduction

Geo-distributed storage systems that provide strong consistency are fundamentally limited in their ability to trade off latency for cost. This technical report supplements our paper [2] on this topic in three ways. First, we describe a set of constraints that limit which read or write latencies can be achieved given a storage budget, and use them to compute a lower bound on the optimal three-way trade-off between read latency, write latency, and storage costs (§2). Second, we provide a proof of correctness for the PANDO consensus protocol (§3). Finally, we provide a TLA+ specification (§4) that we have used to model check PANDO's consistency guarantees under a variety of settings.

2 Lower bound on the latency–cost tradeoff

We begin by describing the setting and assumptions under which we optimize latency and cost:

- **Strong Consistency.** All reads and writes must be linearizable; i.e., all writes are totally ordered and every read returns the last successful write.
- **Reads and writes are executed at a known set of data centers.** We assume that read/write requests are issued by a small number of "front-end" data centers running application code. We refer to the set of front-end data centers that access an object as the object's *access set* and the set of data centers storing the data for that object as its *data sites*.
- **No common-case optimizations.** We do not consider common-case optimizations that improve only throughput or median latency (e.g., read leases [4, 3, 5, 8], follow the sun [10]). Our focus is on the lowest achievable latency bounds.
- **Wide-area network latency dominates and has low variance.** We assume that the latency variance in intra-data center storage (< 5 ms in existing SSD-based systems [1]) is insignificant compared to the latency between data centers and that the network itself has low variance.

Although an approach that optimally trades off latency and cost may use neither replication nor erasure coding to spread data across sites, it still must fetch (distribute)

the equivalent of one copy's worth of data in every read (write). Furthermore, even though there may exist consensus protocols that have yet to be developed, any approach that guarantees linearizability must be able to *"detect updates"* in order to react to them when reading or writing data. Following this line of reasoning, we have assembled a list of properties that have to be maintained by any approach that ensures per-key linearizability (optimal or otherwise):

1. In the event that arbitrary f data sites fail, the remaining data sites *in every write quorum* must contain the equivalent of one copy worth of data.

   Justification: If not met, f failures can lead to unavailability or data loss.

2. Every read quorum must contain at least $f + 1$ data sites.

   Justification: If a read quorum has $\leq f$ data sites, any write that is completed while all data sites in this quorum are unavailable will not be read when a front-end subsequently reads using this quorum. Hence, it is not possible to guarantee linearizability.

3. Any read quorum must contain at least one copy worth of data in total.

   Justification: For a front-end to be able to reconstruct an object's data when it reads the object, it must receive at least one full copy of the object's data.

4. All read–write and write–write quorum pairs must have at least one data site in their intersection.

   Justification: Without this, after a write succeeds after contacting the sites in a write quorum, a subsequent read can complete by contacting a non-overlapping read quorum. The read operation many return stale data, violating linearizability. Similarly, if write quorums do not overlap, two different writes (with different values) for the same key can both succeed, again violating linearizability.

5. There must be at least $2f + 1$ data sites overall.

   Justification: Based on properties (1) and (2), each quorum must contain at least $f + 1$ data sites. If f sites fail, we have to have a read and write quorum available to continue servicing requests. Therefore, after f failures, we need an additional $f + 1$ sites remaining, leading to a minimum total of $2f + 1$.

These constraints are fundamental and we have used them to compute a lower bound on the optimal read latency–write latency–storage overhead tradeoffs. As mentioned earlier, we assume that the set of front-end data centers is known which allows us to optimize data site selection. We do this for the lower bound by selecting one read quorum and one write quorum to use per front-end. Every front-end is free to use any read or write quorum, but in the absence of latency variance and failures, a front-end will incur the lowest read and write latencies when using the quorums chosen for it.

3 The PANDO write protocol

PANDO bears a strong resemblance to Classic Paxos [7], but diverges in a few important ways. First, like Flexible Paxos [6], PANDO uses separate Phase 1 and Phase 2 quorums. Second, like RS-Paxos [9], PANDO allows for data to be erasure coded which requires acceptors to store additional metadata. Lastly, PANDO separates detection of potentially incomplete writes and associated recovery by offering a *fast path* for Phase 1 when no uncertainty exists. This last optimization enables PANDO to use smaller quorum intersections than a straightforward execution of Paxos on erasure-coded data.

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. We achieve this without sacrificing fault tolerance as follows. Each proposer is assigned proposer 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 a write from the proposer and a write from the (proposer, delegate) write are writes from two separate nodes. We already prove (§3.1) 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 no value has been chosen). We maintain linearizability with this approach because the task of resolving uncertainty and conflicts is done using the write path.

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.

$A.ppn$	Promised proposal number stored at acceptor A
$A.apn$	Accepted proposal number 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, used to remove padding from coding
$\text{Split}(v, A)$	(Computed on proposers) The erasure-coded split associated with acceptor A

Figure 1: Summary of notation.

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 Promise messages from a Phase 1a quorum:

5. If the values in all Promise responses are NULL, then skip to Phase 2 with $v' \leftarrow v$.

Proposer P, upon receiving 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 1). 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 $\mathrm{Propose}(p, vid, vlen, vsplit)$ **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 $\mathrm{Accept}(p)$

Proposer P, upon receiving $\mathrm{Accept}(p)$ **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.

3.1 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 quorum of acceptors that all agree on the identity of the value and store splits that correspond 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 as long as RPCs complete before they time out, and all acceptors in a 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 eventually choose a value as long as RPCs complete before they time out, and all acceptors in a Phase 1b and Phase 2 quorum are available.*

Proof. Let t refer to the (maximum) network and execution latency for an RPC. Since PANDO consists of 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 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 [6], 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 $\mathrm{Prepare}(p')$ before $\mathrm{Propose}(p, \ldots)$.

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

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

  The last promised proposal number at A is q such that $p \leq q < p'$ ($q > p'$ would be a contradiction since $\mathrm{Prepare}(p')$ 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 $\mathrm{Promise}(q, vid_v, \ldots)$.

At this point, the proposer has received at least one $\mathrm{Promise}$ 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 $\mathrm{Promise}$ 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 $\mathrm{Prepare}(p')$ would have failed.
- Case 3: $p < q' < p'$. This implies that a $\mathrm{Propose}(q', v'')$ 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. □

4 TLA+ specification for PANDO reads and writes

MODULE *Pando*

EXTENDS *Integers*, *TLC*, *FiniteSets*

CONSTANTS *Acceptors*, *Ballots*, *Values*,
*Quorum*1*a*, *Quorum*1*b*, *Quorum*2, *K*

ASSUME *QuorumAssumption* $\triangleq$
$\land$ *Quorum*1*a* $\subseteq$ SUBSET *Acceptors*
$\land$ *Quorum*1*b* $\subseteq$ SUBSET *Acceptors*
$\land$ *Quorum*2 $\subseteq$ SUBSET *Acceptors*
Overlap of 1
$\land \forall\, QA \in Quorum1a :$
$\forall\, Q2 \in Quorum2 :$
$Cardinality(QA \cap Q2) \geq 1$
Overlap of K
$\land \forall\, QB \in Quorum1b :$
$\forall\, Q2 \in Quorum2 :$
$Cardinality(QB \cap Q2) \geq K$

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

$vars \triangleq \langle msgs,\ maxPBal,\ maxABal,\ maxVal,\ chosen,\ readLog \rangle$
$None \triangleq$ CHOOSE $v : v \notin Values$

Type invariants.

$$
\begin{array}{ll}
Messages \triangleq & \\
 & [type : \{\text{“prepare”}\},\ bal : Ballots] \\
\cup & [type : \{\text{“promise”}\},\ bal : Ballots,\ maxABal : Ballots \cup \{-1\}, \\
 & \ maxVal : Values \cup \{None\},\ acc : Acceptors, \\
 & \ chosen\ : Values \cup \{None\}] \\
\cup & [type : \{\text{“propose”}\},\ bal : Ballots,\ val : Values \cup \{None\}, \\
 & \ op : \{\text{“R”},\ \text{“W”}\}] \\
\cup & [type : \{\text{“accept”}\},\ bal : Ballots,\ val : Values,\ acc : Acceptors, \\
 & \ op : \{\text{“R”},\ \text{“W”}\}] \\
\cup & [type : \{\text{“learn”}\},\ bal : Ballots,\ val : Values]
\end{array}
$$

$$
\begin{array}{lll}
TypeOK \triangleq & \wedge\ msgs & \in \text{SUBSET } Messages \\
 & \wedge\ maxABal & \in [Acceptors \to Ballots \cup \{-1\}] \\
 & \wedge\ maxPBal & \in [Acceptors \to Ballots \cup \{-1\}] \\
 & \wedge\ maxVal & \in [Acceptors \to Values \cup \{None\}] \\
 & \wedge\ chosen & \in [Acceptors \to Values \cup \{None\}] \\
 & \wedge\ readLog & \in [Ballots \to Values \cup \{None\}] \\
 & \wedge\ \forall\ a & \in Acceptors : maxPBal[a] \geq maxABal[a]
\end{array}
$$

Initial state.

$$
\begin{array}{lll}
Init \triangleq & \wedge\ msgs & = \{\} \\
 & \wedge\ maxPBal & = [a \in Acceptors \mapsto -1] \\
 & \wedge\ maxABal & = [a \in Acceptors \mapsto -1] \\
 & \wedge\ maxVal & = [a \in Acceptors \mapsto None] \\
 & \wedge\ chosen & = [a \in Acceptors \mapsto None] \\
 & \wedge\ readLog & = [b \in Ballots \mapsto None]
\end{array}
$$

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.

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

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.

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

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.

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

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.

$$
\begin{array}{l}
ProposeB(b) \triangleq \\
\quad \land \neg \exists\, m \in msgs : (m.type = \text{“propose”}) \land (m.bal = b) \\
\quad \land \exists\, Q \in Quorum1b : \\
\qquad \text{LET } Q1Msgs \triangleq \{m \in msgs :\ \land\ m.type = \text{“promise”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ m.bal\ \ = b \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ m.acc\ \ \in Q\} \\
\qquad\qquad Q1Vals \triangleq [v \in Values \cup \{None\} \mapsto \\
\qquad\qquad\qquad\qquad\qquad \{m \in Q1Msgs : m.maxVal = v\}] \\
\qquad \text{IN} \\
\qquad\quad \text{Check that all acceptors from } Q \text{ responded} \\
\qquad\quad \land \forall\, a \in Q : \exists\, m \in Q1Msgs : m.acc = a \\
\qquad\quad \land \exists\, v \in Values :
\end{array}
$$

```
        ∧  No recoverable value, use anything
           ∨ ∀ vv ∈ Values : Cardinality(Q1Vals[vv]) < K
           Check if v is recoverable and of highest ballot
           ∨  Use previous value if K splits exist
              ∧ Cardinality(Q1Vals[v]) ≥ K
              ∧ ∃ m ∈ Q1Vals[v] :
                    Ensure no other recoverable value has a higher ballot
                    ∧ ∀ mm ∈ Q1Msgs :
                         ∨ m.bal ≥ mm.bal
                         ∨ Cardinality(Q1Vals[mm.maxVal]) < K
     ∧ Send([type ↦ "propose", bal ↦ b, val ↦ v, op ↦ "W"])
∧ UNCHANGED ⟨maxPBal, maxABal, maxVal, chosen, readLog⟩
```

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) ≜
  ∧ ∃ m ∈ msgs :
    ∧ m.type = "propose"
    ∧ m.bal ≥ maxPBal[a]
    ∧ maxABal' = [maxABal EXCEPT ![a] = m.bal]
    ∧ maxPBal' = [maxPBal EXCEPT ![a] = m.bal]
    ∧ maxVal'  = [maxVal  EXCEPT ![a] = m.val]
    ∧ Send([type ↦ "accept", bal ↦ m.bal, acc ↦ a, val ↦ m.val,
            op ↦ m.op])
  ∧ UNCHANGED ⟨chosen, readLog⟩
```

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

```
ProposerEnd(b) ≜
  ∧ ∃ v ∈ Values :
    ∧ ∃ Q ∈ Quorum2 :
      LET Q2msgs ≜ {m ∈ msgs : ∧ m.type = "accept"
                               ∧ m.bal  = b
                               ∧ m.val  = v
                               ∧ m.acc ∈ Q}
      IN
        Check for accept messages from all members of Q
        ∧ ∀ a ∈ Q : ∃ m ∈ Q2msgs : m.acc = a
        If this was in response to a read, log the result
        ∧  Read: log the result
           ∨ ∧ ∃ m ∈ Q2msgs : m.op = "R"
             ∧ readLog' = [readLog EXCEPT ![b] = v]
           Write: don't log the result
           ∨ (∀ m ∈ Q2msgs : m.op = "W" ∧ UNCHANGED ⟨readLog⟩)
    ∧ Send([type ↦ "learn", bal ↦ b, val ↦ v])
  ∧ UNCHANGED ⟨maxABal, maxPBal, maxVal, chosen⟩
```

```
Learn: A proposer has announced that value v is chosen.
Learn(a) ≜
  ∧  ∃ m ∈ msgs :
    ∧ m.type = "learn"
     Process accept before learn, needed for ReadInv, not the protocol
    ∧ maxABal[a] ≥ m.bal
    ∧ chosen' = [chosen EXCEPT ![a] = m.val]
  ∧ UNCHANGED ⟨msgs, maxPBal, maxABal, maxVal, readLog⟩

Count how many splits of v we have received.
CountSplitsOf(resps, v) ≜ Cardinality({m ∈ 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) ≜
  ∧ ¬∃ m ∈ msgs : (m.type = "propose") ∧ (m.bal = b)
  ∧
     Fastest path: Phase 1a quorum has k splits and the value is chosen
    ∨ ∧ ∃ Q ∈ Quorum1a :
        LET RMsgs ≜ {m ∈ msgs : ∧ m.type = "promise"
                                ∧ m.bal  = b
                                ∧ m.acc ∈ Q}
        IN   Check that all acceptors from Q responded
             ∧ ∀ a ∈ Q : ∃ m ∈ RMsgs : m.acc = a
             Check that we have k splits of a chosen value
             ∧ ∃ m ∈ RMsgs :
                 ∧ m.chosen ≠ None
                 ∧ CountSplitsOf(RMsgs, m.chosen) ≥ K
                 ∧ readLog' = [readLog EXCEPT ![b] = m.chosen]
      ∧ UNCHANGED ⟨msgs, maxPBal, maxABal, maxVal, chosen⟩
     Fast path: Phase 1b quorum has k splits and the value is chosen
    ∨ ∧ ∃ Q ∈ Quorum1b :
        LET RMsgs ≜ {m ∈ msgs : ∧ m.type = "promise"
                                ∧ m.bal  = b
                                ∧ m.acc ∈ Q}
        IN   Check that all acceptors from Q responded
             ∧ ∀ a ∈ Q : ∃ m ∈ RMsgs : m.acc = a
             Check that we have k splits of a chosen value
             ∧ ∃ m ∈ RMsgs :
                 ∧ m.chosen ≠ None
                 ∧ CountSplitsOf(RMsgs, m.chosen) ≥ K
                 ∧ readLog' = [readLog EXCEPT ![b] = m.chosen]
      ∧ UNCHANGED ⟨msgs, maxPBal, maxABal, maxVal, chosen⟩
     Slow path: Phase 1b recovery and write back
    ∨ ∧ ∃ Q ∈ Quorum1b :
```

```
    LET Q1Msgs ≜ {m ∈ msgs : ∧ m.type = "promise"
                             ∧ m.bal  = b
                             ∧ m.acc  ∈ Q}
        Q1Vals ≜ [v ∈ Values ∪ {None} ↦
                    {m ∈ Q1Msgs : m.maxVal = v}]
    IN
      Check that all acceptors from Q responded
      ∧ ∀ a ∈ Q : ∃ m ∈ Q1Msgs : m.acc = a
      ∧ ∃ v ∈ Values :
          Check if v is recoverable and of highest ballot
          Use previous value if K splits exist
          ∧ Cardinality(Q1Vals[v]) ≥ K
          ∧ ∃ m ∈ Q1Vals[v] :
              Ensure no other recoverable value has a higher ballot
              ∧ ∀ mm ∈ Q1Msgs :
                  ∨ m.bal ≥ mm.bal
                  ∨ Cardinality(Q1Vals[mm.maxVal]) < K
          readLog will be updated in ProposerEnd
          ∧ Send([type ↦ "propose", bal ↦ b, val ↦ v,
                  op ↦ "R"])
      ∧ UNCHANGED ⟨maxPBal, maxABal, maxVal, chosen, readLog⟩
    No value recovered: Return None
    ∨ ∧ readLog' = [readLog EXCEPT ![b] = None]
      ∧ UNCHANGED ⟨msgs, maxPBal, maxABal, maxVal, chosen⟩
```

Next state.

```
Next ≜ ∨ ∃ b ∈ Ballots    : ∨ Prepare(b)
                            ∨ ProposeA(b)
                            ∨ ProposeB(b)
                            ∨ ProposerEnd(b)
                            ∨ FastRead(b)
       ∨ ∃ a ∈ Acceptors  : Promise(a) ∨ Accept(a) ∨ Learn(a)

Spec ≜ Init ∧ □[Next]_vars
```

Invariant helpers.

```
AllChosenWereAcceptedByPhase2 ≜
  ∀ a ∈ Acceptors :
    ∨ chosen[a] = None
    ∨ ∃ Q ∈ Quorum2 :
        ∀ a2 ∈ Q :
          ∃ m ∈ msgs : ∧ m.type = "accept"
                       ∧ m.acc  = a2
                       ∧ m.val  = chosen[a]

OnlyOneChosen ≜
```

$$\forall\, a,\, aa \in Acceptors :$$
$$\quad (chosen[a] \neq None \land chosen[aa] \neq None) \Rightarrow (chosen[a] = chosen[aa])$$

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

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

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

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

Invariants.

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

$$StabilityInv \triangleq$$
$$\forall\, v \in Values : \forall\, b \in Ballots : ChosenIn(v,\, b) \Rightarrow NoOtherFutureProposal(v,\, b)$$

$$AcceptorInv \triangleq$$
$$\forall\, a \in Acceptors :$$
$$\quad \land\, (maxVal[a] = None) \equiv (maxABal[a] = -1)$$
$$\quad \land\, maxABal[a] \leq maxPBal[a]$$
$$\quad \land\, (maxABal[a] \geq 0) \Rightarrow VotedForIn(a,\, maxVal[a],\, maxABal[a])$$
$$\quad \land\, \forall\, c \in Ballots :$$
$$\quad\quad c > maxABal[a] \Rightarrow \neg\exists\, v \in Values : VotedForIn(a,\, v,\, c)$$

References

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

[2] Anonymous. Near-optimal latency versus cost tradeoffs in geo-distributed storage.

[3] 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.

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

[5] 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.

[6] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum Intersection Revisited. In *OPODIS*, 2016.

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

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

[9] 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.

[10] Z. Shen, Q. Jia, G.-E. Sela, B. Rainero, W. Song, R. van Renesse, and H. Weatherspoon. Follow the sun through the clouds: Application migration for geographically shifting workloads. In *SoCC*, 2016.