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 the paper [**?**] we have written 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. In our paper [**?**], we show that PANDO achieves a near-optimal trade-off.

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.
- **Data centers are the unit of failure.** We ignore failures within a data center and ignore the cost of providing durability within a data center.
- **Reads and writes are executed at a known set of data centers.** We assume that clients issue requests through 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 [3, 2, 4, 7], follow the sun [9]). Our focus is on what the lowest achievable tail latencies are.
- **Wide-area network latency dominates and has low variance.** We assume that latency between data centers is high enough that the difference between median

and tail latency in the intra-data center storage (< 5 ms in existing SSD-based systems [1]) is insignificant and that the network itself has low variance.

- **Data is incompressible.** Data compression is an orthogonal tool that can be used to reduce the size of data stored locally at a data site or can be used at a front-end before it issues a write. Therefore, we assume that data cannot be or has already been compressed.
- **Writes are not commutative.** We do not assume that write operations on an object commute and therefore must resolve any conflicts during write execution.

Although the latency–cost trade-off achieved by an optimal approach may not be bound by limitations of existing protocols, there are still constraints that limit what can be achieved. While an optimal approach may not use existing data distribution techniques such as replication or erasure coding, it still must fetch (and distribute) the equivalent of one copy's worth of data. 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 (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, it is possible for a write to succeed after contacting just Data Sites A and B followed by a read that contacts only C and D. The read operation may return stale data which would violate linearizability. Similarly, write quorums must overlap to resolve conflicts as they occur, since it is not possible (in general) to resolve them later on.

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 $2f + 1$.

$A.ppn$	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	Value 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.

These constraints are fundamental and we have used them to compute a lower bound on the optimal read-write-storage trade-off. 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. We are not implying that these are the only quorums that can be used, but that, ignoring tail latency effects, a front-end will have *a* best read or write quorum to use. By enforcing the above constraints on just these quorums, we can compute a lower bound on the optimal trade-off.

3 The PANDO write protocol

PANDO bears a strong resemblance to Classic Paxos [6], but diverges in a few important ways. First, like Flexible Paxos [5], PANDO uses separate Phase 1 and Phase 2 quorums. Second, like RS-Paxos [8], PANDO erasure codes data which requires acceptors to store a few additional fields. Lastly, PANDO separates the notion of conflict detection and recovery by offering a *fast path* for Phase 1 when no conflicts occur. This last optimization enables PANDO to use smaller quorum intersections than a straightforward application of erasure coding.

As described in this paper, the PANDO write protocol allows a set of nodes to agree on the contents of a single value. Support for multiple values can be achieved by using PANDO to decide the the value of each entry of a distributed log as is typically done with Paxos. In order to maintain consistency, we require that the quorums used with the PANDO write protocol have the following properties:

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.

In addition, if Phase 1a quorums are relied upon for reads, they must contain a minimum of k splits, or else the reader will have to contact additional nodes to recover data. Lastly, PANDO provides availability so long as both a Phase 1b and a Phase 2 quorum of nodes 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 $\mathrm{Prepare}(p)$ messages to all acceptors.

Acceptor A, upon receiving $\mathrm{Prepare}(p)$ message from Proposer P:

3. If $p > A.ppn$ then set $A.ppn \leftarrow p$ and reply $\mathrm{Promise}(A.apn, A.vid, A.vlen, A.vsplit)$.
4. Else reply NACK.

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

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

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

6. Iterate over all $\mathrm{Promise}$ responses by their apn sorted in decreasing order.
   (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. Broadcast $\mathrm{Propose}(p, vid_{v'}, vlen_{v'}, \mathrm{Split}(v', A))$ messages to all acceptors.

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

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

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

16. 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 set by [**?**] and [5], 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. Without loss of generality, we will prove Lemma 1 for the smallest p' such that $p' > p$. If $v' = v$ then we trivially have the desired property. Therefore, assume $v' \neq v$.

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 receieves $\mathrm{Prepare}(p')$ 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 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

Check this and revise comments.

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*
1 intersection
$\land \forall\, QA \in Quorum1a :$
$\forall\, Q2 \in Quorum2 :$
$Cardinality(QA \cap Q2) = 1$
K intersection
$\land \forall\, QB \in Quorum1b :$
$\forall\, Q2 \in Quorum2 :$
$Cardinality(QB \cap Q2) = K$

VARIABLES *msgs*, The set of messages that have been sent
maxPBal, maxPBal[a] is the highest promised proposal number at acceptor a
maxABal, maxABal[a] is the highest accepted proposal number at acceptor a
maxVal, maxVal[a] is the value for maxABal[a] at acceptor a
committed committed[v] marks whether value v is committed (used for reads)

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

Type invariant.

$Messages \triangleq$
$[type : \{\text{“prepare”}\}, bal : Ballots]$
$\cup$ $[type : \{\text{“promise”}\}, bal : Ballots, maxABal : Ballots \cup \{-1\},$
$maxVal : Values \cup \{None\}, acc : Acceptors]$
$\cup$ $[type : \{\text{“propose”}\}, bal : Ballots, phase : \{\text{“a”}, \text{“b”}\},$
$val : Values \cup \{None\}, op : \{\text{“R”}, \text{“W”}\}]$
$\cup$ $[type : \{\text{“accept”}\}, bal : Ballots, val : Values, acc : Acceptors]$

$TypeOK \triangleq$
$\land\, msgs \in$ SUBSET $Messages$
$\land\, maxABal \in [Acceptors \to Ballots \cup \{-1\}]$
$\land\, maxPBal \in [Acceptors \to Ballots \cup \{-1\}]$
$\land\, maxVal \in [Acceptors \to Values \cup \{None\}]$
$\land\, committed \in [Values \to \{\text{FALSE}, \text{TRUE}\}]$
$\land \forall\, a \in Acceptors : maxPBal[a] \geq maxABal[a]$

Initial state.

$$
\begin{array}{lll}
Init \triangleq & \land\, msgs & = \{\} \\
& \land\, maxPBal & = [a \in Acceptors \mapsto -1] \\
& \land\, maxABal & = [a \in Acceptors \mapsto -1] \\
& \land\, maxVal & = [a \in Acceptors \mapsto None] \\
& \land\, committed & = [v \in Values \mapsto \text{FALSE}]
\end{array}
$$

Send message m.

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

WRITE PATH

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

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

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.

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

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.

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

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.

$$
\begin{array}{l}
ProposeB(b) \triangleq \\
\quad \land \neg \exists\, m \in msgs : (m.type = \text{“propose”}) \land (m.bal = b) \\
\qquad\qquad\qquad\qquad \land (m.phase = \text{“a”} \lor m.phase = \text{“b”}) \\
\quad \land \exists\, v \in Values : \\
\qquad \land \exists\, Q \in Quorum1b : \\
\qquad\quad \text{LET } Q1Msgs \triangleq \{m \in msgs : \land m.type = \text{“promise”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land m.bal \;\; = b \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land m.acc \in Q\} \\
\qquad\qquad\quad Q1Vals \triangleq \{m \in Q1Msgs : m.maxVal = v\} \\
\qquad\quad \text{IN} \\
\qquad\qquad \text{Check for promises from all acc in Q.} \\
\qquad\qquad \land \forall\, a \in Q : \exists\, m \in Q1Msgs : m.acc = a \\
\qquad\qquad \text{Use previous val if K exist.} \\
\qquad\qquad \land Cardinality(Q1Vals) \geq K \\
\qquad\qquad \land \exists\, m \in Q1Vals : \\
\qquad\qquad\quad \land \forall\, mm \in Q1Msgs : m.bal \geq mm.bal \\
\quad \land Send([type \mapsto \text{“propose”},\ bal \mapsto b,\ val \mapsto v, \\
\qquad\qquad op \mapsto \text{“W”},\ phase \mapsto \text{“b”}]) \\
\quad \land \text{UNCHANGED } \langle maxPBal,\ maxABal,\ maxVal,\ committed \rangle
\end{array}
$$

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.

$$
\begin{array}{l}
Accept(a) \triangleq \\
\quad \land \exists\, m \in msgs : \\
\qquad \land m.type = \text{“propose”} \\
\qquad \land m.bal \geq maxPBal[a] \\
\qquad \land m.op = \text{“W”} \\
\qquad \land maxABal' = [maxABal \text{ EXCEPT } ![a] = m.bal] \\
\qquad \land maxPBal' = [maxPBal \text{ EXCEPT } ![a] = m.bal] \\
\qquad \land maxVal' \;\; = [maxVal \text{ EXCEPT } ![a] = m.val] \\
\qquad \land Send([type \mapsto \text{“accept”},\ bal \mapsto m.bal,\ acc \mapsto a,\ val \mapsto m.val]) \\
\quad \land \text{UNCHANGED } \langle committed \rangle
\end{array}
$$

Phase 3: If the proposer receives Quorum2 of acknowledgements, the value is successfully chosen and the commit is broadcasted.

$$
\begin{array}{l}
Phase3(b) \triangleq \\
\quad \land \exists\, v \in Values : \\
\qquad \land \exists\, Q \in Quorum2 : \\
\qquad\quad \text{LET } Q2msgs \triangleq \{m \in msgs : \land m.type = \text{“accept”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land m.bal \;\; = b \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land m.acc \in Q\} \\
\qquad\quad \text{IN}
\end{array}
$$

Check for promises from all acc in Q.

$\land \forall\, a \in Q : \exists\, m \in Q2msgs : (m.acc = a) \land (m.val = v)$

$\land\ committed' = [committed \text{ EXCEPT } ![v] = \text{TRUE}]$

$\land \text{ UNCHANGED } \langle msgs,\ maxABal,\ maxPBal,\ maxVal \rangle$

READ PATH

Read Path : Check if any value (with the highest ballot) returned from Quorum1b was committed. If so, return that value. Otherwise, perform a write-back.

$ReadPath(b) \triangleq$

$\land \neg\exists\, m \in msgs : (m.type = \text{“propose”}) \land (m.bal = b)$

$\land \exists\, Q \in Quorum1b :$

LET $RMsgs \triangleq \{m \in msgs : \land m.type = \text{“promise”}$
$\land m.bal = b$
$\land m.acc \in Q\}$

IN

Check for promises from all acc in Q.

$\land \forall\, a \in Q : \exists\, m \in RMsgs : m.acc = a$

Can't read anything if no previous value exists.

$\land \forall\, m \in RMsgs : m.maxABal \neq -1$

$\land$

Complete if committed value found.

$\lor \exists\, m \in RMsgs :$

$\land committed[m.maxVal]$

Ensure that the value has the highest ballot.

$\land \forall\, mm \in RMsgs : m.bal \geq mm.bal$

Check if value can be recovered (K splits exist).

$\land\ Cardinality(\{mm \in RMsgs : mm.maxVal = m.maxVal\}) \geq K$

Send this message so that subsequent phases don't occur.

$\land\ Send([type \mapsto \text{“propose”},\ bal \mapsto b,\ phase \mapsto \text{“a”},$
$val \mapsto m.maxVal,\ op \mapsto \text{“R”}])$

Otherwise, do nothing else on this ballot.

$\lor\ Send([type \mapsto \text{“propose”},\ bal \mapsto b,\ phase \mapsto \text{“a”},$
$val \mapsto None,\ op \mapsto \text{“R”}])$

$\land \text{ UNCHANGED } \langle maxPBal,\ maxABal,\ maxVal,\ committed \rangle$

Next state.

$Next \triangleq \lor \exists\, b \in Ballots : \lor Prepare(b)$
$\lor ProposeA(b)$
$\lor ProposeB(b)$
$\lor Phase3(b)$
$\lor ReadPath(b)$
$\lor \exists\, a \in Acceptors : Promise(a) \lor Accept(a)$

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

CorrectlyCommitted checks that a committed value is accepted by a write quorum.

$$
\begin{array}{l}
CorrectlyCommitted \triangleq \\
\quad \forall\, v \in Values : committed[v] \Rightarrow \\
\quad\quad \exists\, Q \in Quorum2 : \\
\quad\quad\quad \forall\, a \in Q : \\
\quad\quad\quad\quad \exists\, m \in msgs : \ \wedge\, m.type = \text{“accept”} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \wedge\, m.acc = a \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \wedge\, m.val = v
\end{array}
$$

OnlyOneCommitted checks that only a single value is ever committed.

$$
\begin{array}{l}
OnlyOneCommitted \triangleq \\
\quad \forall\, v,\, vv \in Values : (committed[v] \wedge committed[vv]) \Rightarrow (v = vv)
\end{array}
$$

Learning invariant

$$
LearnInv \triangleq CorrectlyCommitted \wedge OnlyOneCommitted
$$

Read Path invariant

$$
\begin{array}{l}
ReadInv \triangleq \\
\quad \text{Skip check if no reads occured.} \\
\quad \vee \neg \exists\, m \in msgs : (m.type = \text{“propose”}) \wedge (m.op = \text{“R”}) \\
\quad \text{Read should only take fast path if values are committed.} \\
\quad \vee \forall\, m \in msgs : \\
\quad\quad (m.type = \text{“propose”} \wedge m.op = \text{“R”} \wedge m.val \neq None) \Rightarrow \\
\quad\quad\quad \wedge\, m.type = \text{“propose”} \\
\quad\quad\quad \wedge\, m.op = \text{“R”} \\
\quad\quad\quad \wedge\, committed[m.val] \\
\quad\quad\quad \text{Make sure K splits exist in Quorum1b when the read was performed.} \\
\quad\quad\quad \wedge\, \exists\, Q \in Quorum1b : \\
\quad\quad\quad\quad \text{LET } RMsgs \triangleq \{mm \in msgs : \ \wedge\, mm.type = \text{“promise”} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \wedge\, mm.bal\ \ = m.bal \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \wedge\, mm.acc \in Q\} \\
\quad\quad\quad\quad \text{IN} \\
\quad\quad\quad\quad\quad \text{Check for promises from all acc in Q.} \\
\quad\quad\quad\quad\quad \wedge\, \forall\, a \in Q : \exists\, mm \in RMsgs : mm.acc\ \ = a \\
\quad\quad\quad\quad\quad \wedge\, Cardinality(\{mm \in RMsgs : mm.maxVal = m.val\}) \geq K
\end{array}
$$

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

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

$$
ChosenIn(v,\, b) \triangleq \exists\, Q \in Quorum2 : \forall\, a \in Q : VotedForIn(a,\, v,\, b)
$$

Chosen checks that a given value is agreed upon by any phase 2 quorum.

$$Chosen(v) \triangleq \exists\, b \in Ballots : ChosenIn(v,\, b)$$

Consistency predicate to make sure only a single value is ever chosen.

$$Consistency \triangleq \forall\, v1,\, v2 \in Values : Chosen(v1) \land Chosen(v2) \Rightarrow (v1 = v2)$$

WontVoteIn is a predicate that implies that a has not voted and never will vote in ballot b.

$$\begin{aligned} WontVoteIn(a,\, b) \triangleq\ & \land \forall\, v \in Values : \neg\, VotedForIn(a,\, v,\, b) \\ & \land\ maxPBal[a] > b \end{aligned}$$

SafeAt implies that no value other than perhaps v has been or ever will be chosen in any ballot numbered less than b.

$$\begin{aligned} &SafeAt(v,\, b) \triangleq \\ &\quad \forall\, c \in 0\,..\,(b-1) : \\ &\qquad \lor \exists\, Q \in Quorum1a : \\ &\qquad\quad \forall\, a \in Q : VotedForIn(a,\, v,\, c) \lor WontVoteIn(a,\, c) \\ &\qquad \lor \exists\, Q \in Quorum1b : \\ &\qquad\quad \forall\, a \in Q : VotedForIn(a,\, v,\, c) \lor WontVoteIn(a,\, c) \end{aligned}$$

The predicate NoFutureProposal ensures that no future proposal has a different value.

$$\begin{aligned} &NoFutureProposal(v,\, b) \triangleq \\ &\quad \forall\, vv \in Values : \\ &\qquad \forall\, bb \in Ballots : \\ &\qquad\quad (bb > b \land ProposedValue(vv,\, bb)) \Rightarrow v = vv \end{aligned}$$

Message invariant.

$$MsgInv \triangleq$$

Check that the system is consistent.

$$\land\ Consistency$$

Check that if a value is chosen, no future value can be proposed

$$\begin{aligned} &\land \forall\, v \in Values : \\ &\quad \forall\, b \in Ballots : ChosenIn(v,\, b) \Rightarrow NoFutureProposal(v,\, b) \end{aligned}$$

Check all messages.

$$\begin{aligned} &\land \forall\, m \in msgs : \\ &\quad \land\ (m.type = \text{“promise”}) \Rightarrow \\ &\qquad \land\ m.bal \leq maxPBal[m.acc] \\ &\qquad \land\ \lor\ \land\ m.maxVal \in Values \\ &\qquad\qquad\quad \land\ m.maxABal \in Ballots \\ &\qquad\qquad\quad \land\ VotedForIn(m.acc,\, m.maxVal,\, m.maxABal) \\ &\qquad\qquad \lor\ \land\ m.maxVal = None \\ &\qquad\qquad\quad \land\ m.maxABal = -1 \\ &\qquad \land \forall\, c \in (m.maxABal + 1)\,..\,(m.bal - 1) : \\ &\qquad\quad \neg \exists\, v \in Values : \\ &\qquad\qquad\quad VotedForIn(m.acc,\, v,\, c) \\ &\quad \land\ (m.type = \text{“propose”}) \Rightarrow \end{aligned}$$

$\lor\ m.op = \text{“R”}$
$\lor\ \land\ SafeAt(m.val,\ m.bal)$
$\land\ \forall\ ma \in msgs :$
$(ma.type = \text{“propose”}) \land (ma.bal = m.bal) \land (ma.phase = m.phase)$
$\Rightarrow (ma = m)$
$\land\ (m.type = \text{“accept”}) \Rightarrow$
$\land\ \exists\ ma \in msgs :\ \land\ ma.type = \text{“propose”}$
$\land\ ma.bal = m.bal$
$\land\ ma.val = m.val$
$\land\ ma.op = \text{“W”}$
$\land\ m.bal \leq maxABal[m.acc]$

Acceptor invariant.

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

References

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

[2] 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.

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

[4] 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.

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

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

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

[8] 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.

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