Oblivious Paxos: Privacy-Preserving Consensus Over Secret-Shares (Extended Version)

Fadhil I. Kurnia
University of Massachusetts Amherst
United States
fikurnia@cs.umass.edu

Arun Venkataramani
University of Massachusetts Amherst
United States
arun@cs.umass.edu

ABSTRACT

State-of-the-art consensus protocols like Paxos reveal the values being agreed upon to all nodes, but some deployment scenarios involving a subset of nodes outsourced to public cloud providers motivate hiding the value. In this work, we present the *primary-backup secret-shared state machine* (PBSSM) architecture and an underlying consensus protocol *Oblivious Paxos* (OPaxos) that enable strong consistency, high availability, privacy, and fast common-case performance. OPaxos enables privacy-preserving consensus by allowing acceptors to safely agree on a secret-shared value without untrusted acceptors knowing the value. We also present *Fast Oblivious Paxos* (Fast-OPaxos), which enables consensus over secret-shares in three one-way delays under low concurrency settings. Our prototype-driven microbenchmarks and smarthome case study show that OPaxos induces a negligible latency overhead of at most 0.1 ms compared to Paxos while maintaining more than 85% of Paxos' capacity for small requests, and can provide lower latency and higher capacity compared to Paxos for large request sizes.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; • **Security and privacy**;

KEYWORDS

Distributed Consensus, Privacy, Secret-Sharing

ACM Reference Format:
Fadhil I. Kurnia and Arun Venkataramani. 2023. Oblivious Paxos: Privacy-Preserving Consensus Over Secret-Shares (Extended Version). In *University of Massachusetts Amherst - Computer Science Technical Report - UM-CS-2023-001, Amherst, MA, USA*. 33 pages.

UMass Amherst - CS Technical Report UM-CS-2023-001, 2023, MA, USA

1 INTRODUCTION

Consensus is a fundamental building block for highly available distributed systems with strong consistency, however state-of-the-art consensus protocols like Paxos [32] and others suffer from a critical privacy drawback: the agreement protocol reveals the value being agreed upon to all replica servers. Traditional approaches such as the replicated state machine (RSM) [55] need replicas to be able to view the underlying application state in order to execute requests. But exposing the request values and state is problematic in deployment scenarios managing sensitive information and outsourcing infrastructure to untrusted cloud providers that may be *honest but curious*.

Building general consensus-based services managing sensitive information on untrusted public clouds is challenging. Purist approaches to this end include secure multiparty computation (SMPC) [15] and fully homomorphic encryption (FHE) [17] that are powerful in their generality, but despite much progress on both fronts in recent years, remain prohibitively costly for general services today. Secure computing hardware, e.g., Intel's SGX, combined with oblivious RAM techniques [18] is another option in the design space with lower overhead but stronger trust assumptions and weaker privacy guarantees. None of these offer *information-theoretic privacy* [57], i.e., resilience against an adversary with unbounded computation power, our overarching design goal, while ensuring strong consistency, high availability, and fast common-case performance for general services, a quartet of design goals that is fundamentally challenging.

Hybrid clouds, a widely used infrastructure setup today, offer a distinct opportunity to come closer to the above ideal quartet of design goals. Hybrid cloud deployments involve a subset of trusted servers on premises while relying on untrusted cloud servers for availability despite failures. The availability of trusted servers in the common case makes it easier to maintain fast common-case performance provided we can smoothly failover to untrusted servers alone while continuing to preserve as much of the applications' functionality and as close to information-theoretic privacy as is practically feasible. Our position is that this point in the design space offers better tradeoffs in practice to the alternative of not having information-theoretic privacy at all and/or

being limited to services for which the overhead of SMPC, FHE, or secure computing infrastructure is acceptable.

To that end, we present a novel architecture, *primary-backup secret-shared state machine* (PBSSM), for building highly available services with strong consistency requirements that ensures information-theoretic privacy of request values as well as underlying state in the common-case when at least one trusted server is available. In the event of failure of all trusted servers, PBSSM gracefully fails over to one of two modes: (1) a client-driven mode suitable for applications with client-partitioned state (e.g., key-value store operations with keys identifying clients) wherein a trusted client can play the role of a trusted primary; or (2) more general operations relying on SMPC with commensurate overhead and *real-world ideal-world* privacy (weaker than information-theoretic privacy as detailed in §3.1).

The key technical innovation that drives PBSSM is *Oblivious Paxos* (OPaxos), a privacy-preserving consensus protocol that provides information-theoretic privacy by integrating secret-sharing into the Paxos family of consensus protocols while preserving its traditional safety and liveness properties. OPaxos uses (t, n) threshold secret-sharing that generates n secret-shares from a single secret value in a manner that enables us to reconstruct the secret with just t shares, for a configurable $t \in [1, n]$. The protocol requires at least t acceptors as the intersection between quorums in the two Paxos phases as opposed to the traditional non-zero intersection requirement (e.g., using majority quorums for both phases). While previous works [47, 63, 70] have leveraged t-intersection of acceptor quorums for performance reasons, to the best of our knowledge, this work is the first to ensure the privacy requirement, one that introduces subtle design differences–yet important to ensure agreement safety–while also enabling unique performance optimization opportunities (detailed in §4). OPaxos can also be used outside the PBSSM context in existing distributed storage systems [7, 19, 31, 38, 40, 59] that do use secret sharing for privacy against untrusted servers but rely on an external coordination service (entailing latency overhead) to ensure consistency under failures or concurrency by offering a general-purpose consensus protocol with in-built privacy.

Given our focus on hybrid cloud scenarios, it is natural to consider the seemingly more straightforward alternative of having the trusted consensus leader simply encrypt request values when at least one trusted server is available. However, encryption carries two fundamental drawbacks: (1) it does not offer information-theoretic privacy, instead relying on assumptions on an adversary's computational resources, and in practice induces vulnerability to key theft and/or increased key management complexity to thwart them; (2) it poses an inconvenient cost-availability trade-off as the system must rely on an external trusted key escrow

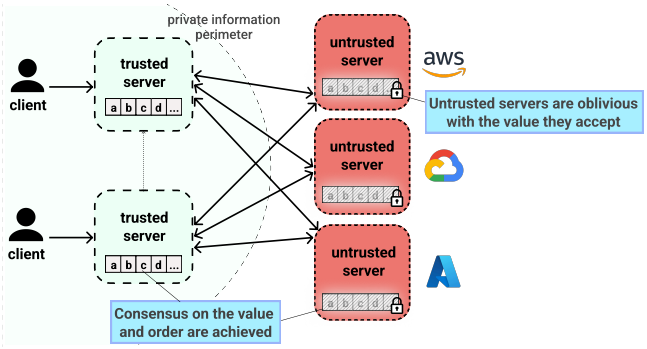


Figure 1: System and threat model: Clients' data must remain private from the untrusted servers placed at different cloud providers.

service as otherwise (permanent) failure of internal trusted nodes possessing the key can cause (permanent) data irrecoverability. Finally, as detailed in §3.1 and our experimental evaluation, secret-sharing allows us to naturally leverage recent advances in SMPC (compared to FHE) for more general services when no trusted server is available with overhead comparable to simple encryption.

We build further upon the OPaxos design to develop *Fast Oblivious Paxos* (Fast-OPaxos), an optimization similar in spirit to Fast-Paxos vis a vis classic Paxos. Fast-OPaxos enables a client to receive a commit confirmation in three one-way delays while being leader-oblivious, one less than in the case of leader-aware traditional Paxos. Fast-OPaxos design improves both latency and throughput with a single trusted proposing client or in general when conflicts are rare

We have implemented an open-source prototype for OPaxos and Fast-OPaxos, and a prototype key-value store embodying the PBSSM approach on top of OPaxos. Through prototype-driven experiments, we show that compared to Paxos, the secret-sharing overhead in OPaxos entails only a modest latency and capacity overhead, in part because of the reduced size of secret shares as well as hardware support such as SIMD and AES-NI in modern CPUs enabling fast cryptographic operations. For small request values, our OPaxos prototype induces a latency overhead of less than 0.1 ms while providing over 85% of non privacy-preserving Paxos' capacity; while for large requests, OPaxos can *improve* both latency and capacity.

We further demonstrate the end-to-end benefits of OPaxos via a case study of a modern smart-home system with trusted servers residing within the home and the traditional smart-home cloud service distributed across multiple (non-colluding) cloud providers, thereby providing high availability and linearizable consistency while limiting the perimeter of private information leakage to within the trusted home zone.

In summary, our primary contribution is a novel architecture, primary-backup secret-shared state machine, to build

fault-tolerant distributed services with strong consistency while ensuring information-theoretic privacy, and comprises the following technical sub-contributions:

(1) Design and implementation of OPaxos (§4) with a rigorous formal proof of safety as well as model checking (§4.4), and a Fast-OPaxos optimization (§4.5);
(2) Prototype-driven evaluation of OPaxos (§7) showing modest latency and capacity overhead in general and improved latency and capacity for large requests;
(3) Case study evaluation of a smart-home system based on OPaxos (§8.2) in a multi-cloud deployment setting.

2 PROBLEM MODEL AND BACKGROUND

This section describes our system, threat and failure model.

2.1 System, Threat, and Failure Models

System model. Our target scenario, as illustrated in Figure 1, is a highly available general service (or state machine) to manage private client data that is provided by a collection of n servers some of which are *trusted* and others *untrusted*. The untrusted servers must not know anything about the managed client data or client requests that act upon that state. The trusted subset of servers implement service execution, i.e., computation and state management in response to client requests, while the untrusted servers may be relied upon to ensure availability in the face of temporary or permanent server failures. The system must ensure strong linearizability consistency[1]. An example scenario is a smart-home management service wherein the trusted nodes reside within the home limiting the perimeter of sensitive information leakage to the home; another is a distributed password management or a client account management system in a hybrid cloud setup wherein the trusted nodes are on-premises while the untrusted nodes are on third-party clouds. Lack of strong consistency in such systems can compromise safety, e.g., they can compromise security in a smart-home if the surveillance devices or smart locks do not reflect the most recent configured routines.

Threat and failure model. The trusted servers and clients mutually trust each other. The untrusted servers are assumed to be honest but curious, i.e., they are expected to execute the protocol correctly but may try to glean any information they can from received protocol messages. The network environment is asynchronous and any node may experience a temporary crash or a permanent failure that may render any state on it irrecoverable. The non-byzantine crash failure model is consistent with the honest-but-curious threat model for untrusted servers, nevertheless untrusted servers may collude with each other provided less than t untrusted servers collude, where t is an a priori known limit. Our threat model precludes side-channel attacks based on message size, timing, etc. Both encryption-based and information-theoretically private approaches in general are susceptible to such side-channel attacks and require additional mechanisms extensively studied by others to thwart them [14, 41, 42] that are outside the scope of this work.

[1]This basic model also generalizes to services with weaker consistency semantics that can be satisfied using consensus as a building block.

2.2 Background Primers

OPaxos builds upon Paxos so as to offer information-theoretic privacy among other design goals. We include brief primers of Paxos [33] and information-theoretic privacy [57].

Paxos is an asynchronous distributed consensus protocol enabling a fixed set of nodes to propose values and agree upon one of those proposals as the chosen decision. Paxos proceeds in increasing, ordered rounds (also called ballots) many-to-one mapped to proposing nodes. A proposing node attempts to complete two phases in its current round: (i) a *prepare* phase wherein a *proposer* attempts to get a *prepare quorum* (or Q1) of *acceptors* to affirm its round number by promising not to accept values in lower rounds and report their respective accepted values if any in the highest lower round; and subsequently (ii) an *accept* phase in which it actually proposes its value and seeks to get an *accept quorum* (or Q2) of *acceptors* to accept that value while respecting their respective promises in the prepare phase. Paxos ensures the safety property that only a single proposal can be chosen as the decision by restricting the proposable value in the second phase based on the values if any reported in the first phase. Paxos requires that the intersection of any Q1 quorum and Q2 quorum is nonempty (e.g., both majorities).

Information-theoretic privacy guarantees that no information about protected data is revealed to adversaries even with unlimited computational resources and time. Information-theoretic privacy-preserving schemes, of which secret-sharing is a well-known example, do not rely on a key and ensure that any adversary with less than the threshold number of shares would find all potential secret values equiprobable, thereby learning no information about the secret [57]. In comparison, encryption-based approaches necessarily make limiting assumptions about the adversary's computation power, e.g., hardness of factoring the product of large primes, to ensure privacy [13]. Furthermore, in practice, encryption-based approaches are also vulnerable to key theft and/or entail additional key management complexity to thwart them, and must rely on a key escrow mechanism to protect against accidental key loss that can potentially render encrypted data permanently unavailable.

3 OPAXOS SYSTEM ARCHITECTURE

OPaxos targets the following design goals, the combination of which is both novel and nontrivial to achieve

Mode	Application class	Privacy property	Mechanism	Example scenarios
Trusted	General State Machine	Information-Theoretic	PBSSM (§3.2)	Warehouse system (§7.4), Smart-home system (§8.2)
Untrusted	Client-Partitioned State Machine	Information-Theoretic	CPSSM (§5.1)	Private key-value store (§7.5)
	General State Machine	Ideal World/ Real World	SMPC (§5.1)	Private data analytics [38, 49, 66]

Table 1: An OPaxos-based system's operational modes.

(1) **Common-case performance**: Low OPaxos overhead when at least one trusted server is up.
(2) **Seamless failover**: Seamless high availability despite server failures including failure of all trusted servers.
(3) **Information-theoretic privacy**: Ensuring that untrusted servers, even with unbounded resources, do not learn anything about the client state or requests.
(4) **Strong consistency**: Support for general applications with strong state consistency constraints.

Strictly achieving all of the above goals with approaches known today is very challenging, if at all possible, so OPaxos' goal is to come as close to them as possible. To better understand the challenge, let's consider a few natural alternatives.

3.1 State-of-the-art Alternatives Analysis

A straightforward approach is encryption wherein the trusted servers can be viewed as a trusted proxy commonly used in privacy-preserving data stores that encrypts client requests before agreeing upon their order in coordination with the untrusted backups. Encryption-based approaches however suffer from several drawbacks. First, they do not afford information-theoretic privacy by definition making them vulnerable to key compromise, theft, or trapdoors. Second, they necessitate additional infrastructure in the form of a key escrow service as otherwise permanent failure (say disaster-induced) of trusted servers or otherwise loss of the encryption key can result in permanent data unavailability. More importantly, they poorly meet the seamless failover design goal as in the absence of any trusted server, untrusted servers need to be able to make progress with encrypted copies of state. Supporting general state machine services in this mode necessitates fully homomorphic encryption (FHE), but the set of applications for which FHE techniques are low-overhead enough to be practical today is rather limited.

In comparison to FHE, advances in secure multiparty computation (SMPC) in recent years show significant speedup [15] and have rapidly expanded the class of computations that can be performed with overheads low enough to be usable in practice, e.g., systems such as SECRECY [38], Obscure [19], Senate [49], Conclave [66], and others have expanded the scope of computation from simple operations like boolean and arithmetic operations to richer functions like those in typical database query languages (such as `select`, `count`, `limit`, `join`, etc.) as well as to optimize the query processing plan for sophisticated database queries all while maintaining privacy of the underlying state from untrusted servers.

However, the privacy afforded by both SMPC- as well as FHE-based approaches have a critical fundamental limitation, namely they do not afford request privacy, only state privacy, in a state machine. In either approach, the function being computed (locally in FHE and distributed with interactive rounds in SMPC) is by design public to the untrusted servers. Furthermore, SMPC in general also exposes the result of the computation (although for specific functions, it may be possible to orchestrate the computation so that individual untrusted servers only produce secret shares of the result that can be re-assembled by a trusted end-client). Although state-of-the-art SMPC techniques internally employ secret sharing, they also do not provide information-theoretic privacy, rather they enable ideal-world/real-world privacy [15] that guarantees that untrusted servers don't learn anything more about client data compared to what they would have learned by submitting the request to a trusted server and obtaining the result of the computation from it.

In keeping with its stated design goals, the overhead limitations of general private computation techniques, and the availability of trusted servers in the common case in many real-world scenarios, OPaxos adopts a pragmatic approach that, in the common case of trusted server availability, enables information-theoretic privacy (including request privacy) with overhead comparable to traditional consensus-based fault-tolerance. When no trusted servers are available, OPaxos continues to provide information-theoretic privacy for a class of services referred to as *client-partitioned* state machines (detailed in §5.1). For general state machines under no trusted server availability, OPaxos' secret-sharing based approach enables it to seamlessly fall back on SMPC-based alternatives and their (weaker) ideal-world privacy guarantee and overhead limitations. Our position is that this design pushes the envelope closest to the OPaxos' targeted design goals. The next section explains this design.

3.2 OPaxos/PBSSM High-level Design

OPaxos adopts a *primary-backup secret-shared state machine* (PBSSM) design wherein the trusted subset of servers (possibly singular) store application state in plaintext and untrusted backup servers store corresponding secret shares. Under graceful conditions when at least one trusted server is available, a trusted server is designated as the consensus leader (or *primary*) with the following key differences from a traditional RSM design in that: 1) the primary first executes the (tentative) next client request before agreement

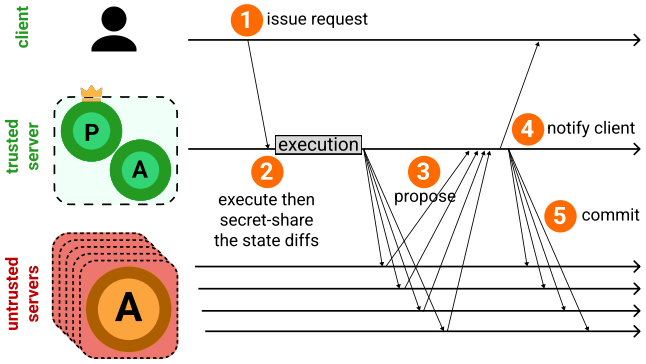


Figure 2: PBSSM execution in the default trusted mode.

over its order; and 2) the primary uses the *state diffs* or secret shares thereof resulting from the execution as its proposal for agreement over the next slot number; 3) upon agreement termination, untrusted backups apply secret shares of the agreed upon state diffs while trusted backup servers apply plaintext diffs to their copy of the state, and the primary replies back to the request-issuing client. Figure 2 illustrates this high-level PBSSM design.

A few remarks are in order for the choice of the above design, why it works, and its tradeoffs. The primary-backup design is well aligned with our threat model and assumption of common-case trusted server availability. Unlike a traditional RSM, untrusted OPaxos servers only store secret shares of application state and therefore cannot directly execute requests to compute state transformations locally, however trusted servers with plaintext state can simply execute requests and compute and transmit state diffs to the backups, so OPaxos' leader election accordingly prioritizes trusted servers, if any are available, over untrusted ones. A primary-backup approach also has the desirable dual side-effects of reducing the replication cost of execution and making the state machine deterministic as only a single trusted primary executes a request; all non-primaries, including trusted ones, apply state diffs transmitted by that primary. The justification for performing execution before agreement is more subtle and has to do with ensuring state convergence safety with non-deterministic state machines (as detailed in §5.2).

OPaxos' design enables it to seamlessly fail over from a primary-backup to a decentralized secret-shared state machine with untrusted backups alone when no trusted server is available. This mode of execution, referred to as the *untrusted* mode, for general state machines relies on secure multiparty computation, but simple applications such as key-value stores can make do with a trusted client dealer without SMPC techniques. The SMPC mode fundamentally entails the necessary limitations of requiring state machine determinism as well as lack of request privacy, neither of which limits either of the (trusted) PBSSM mode or client-dealer-driven untrusted modes. We describe the untrusted mode

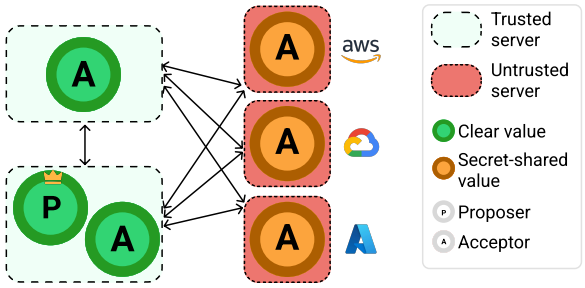


Figure 3: OPaxos' proposers and acceptors placement in the trusted and untrusted servers. Some acceptors are placed in trusted server.

operation in more detail in §5.1 and summarize the high-level operational modes in Table 1.

4 OPAXOS CONSENSUS PROTOCOL

We describe the roles, event-action protocol, quorum constraints, and formal proofs of OPaxos' consensus protocol.

4.1 Overview

As in Paxos, OPaxos has three actors: *proposers*, *acceptors*, and *learners*. A *proposer* proposes a secret-shared value to the *acceptors*, all the *acceptors* try to agree on the value based on their respective secret-shares, and the *learners* are eventually informed of the shares of the agreed upon secret value. For simplicity, we combine the *learner* and *acceptor* roles and just refer to them as *acceptor*. Since proposers perform secret-sharing, they are placed in the trusted servers allowed to know the secret value as illustrated in Figure 3. Acceptors may be placed on either trusted or untrusted servers, but untrusted acceptors must not know the secret value.

OPaxos uses (t, n) threshold secret-sharing wherein the proposer transforms a secret value into n secret-shares distributed to all the n acceptors with at least t shares required to reconstruct the secret, thereby making it resistant to $(t-1)$-collusion. A key technical challenge is to integrate threshold secret-sharing into a quorum-based consensus protocol.

Why Challenging. To appreciate the challenge, consider a strawman protocol where a proposer simply issues secret shares of the proposed value instead of the proposed value itself in a protocol otherwise identical to Paxos. This protocol has at least two problems. The first problem is the recovery of values that may have already been decided in lower round. In Paxos, if a proposer as part of the promise messages in the first phase receives even a single accepted value in a lower round, it loses the flexibility to propose an arbitrary value, however in the strawman protocol, the proposer may only retrieve a single secret share in the first phase, which is insufficient to reconstruct any lower ballot value. The second problem is Paxos safety's reliance on proposers being truthful thereby preventing them from proposing different values in

Variable Name	Description
Acceptor	
bmax	The highest ballot-number this acceptor has seen.
bacc	The ballot number in which this acceptor accepts the secret-share.
bori	The original-ballot; the first ballot number used to propose the accepted secret-share. Used as the secret value's identifier.
ssval	The accepted secret-share.
committed	A flag indicating whether ssval is committed or not
Proposer	
bcur	The current ballot number for this proposer.

Table 2: Variables in OPaxos' acceptors and proposers.

the same round, so proposing different secret shares violates that literal assumption, and furthermore, a proposer in this strawman protocol has no way to know if two secret shares received from two different acceptors were generated from the same secret value because the secret shares by design reveal no information about the original secret.

To address the first problem, OPaxos adapts the quorum constraint to ensure t acceptors intersection (not just nonzero intersection). To address the second problem of attaching additional information with a secret share that allows a node to determine if two shares came from the same secret without revealing any additional information about the secret, OPaxos relies on the *original ballot* or the ballot in which a secret share was first proposed. We explain these and other subtle adaptations including precise pseudocode next.

4.2 Consensus Protocol Phases

OPaxos, like Paxos, has two safety-critical phases initiated by a proposer in some round designated to it: *prepare* and *propose* [2]. In the prepare phase, a proposer seeks to acquire promises from some |Q1| quorum of acceptors not to accept any value in any lower round and retrieve secret shares if any that they accepted in their respective highest lower round. In the propose phase, the proposer seeks to acquire a |Q2| quorum of acceptances from acceptors for shares of the value if any retrieved from the lower round acceptances reported in the promise messages in the prepare phase, else for shares of an arbitrary value. A value is decided if a proposer acquires the designated quorums in both phases in the same round, otherwise it may retry with a higher round designated to it.

A round, also known as ballot number, is represented as a two-tuple consisting of a counter and the identity of the proposing server. Ballots define a totally ordered space wherein a ballot b_1 is greater than ballot b_2 if b_1's counter is greater than b_2's counter or if both counters are equal and b_1's identity is lexicographically "greater" than that of b_2.

Phase 1: Prepare-Promise. In the prepare phase (Phase-1) of OPaxos shown in Figure 4, a proposer sends a PREPARE(bcur)

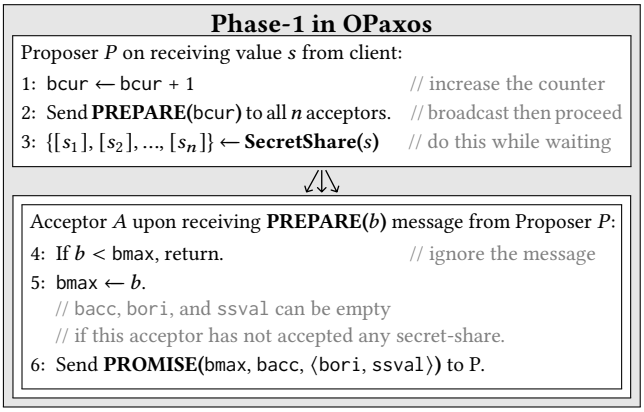


Figure 4: Phase-1: acceptors sending promises.

message to acceptors where bcur is its current ballot number. While the proposer waits for promises from acceptors, it can in parallel generate the secret-shares of value s that it wishes to propose. In all pseudocode, we use $[s_i]$ (with square brackets) to denote the i-th secret-share of s.

An acceptor maintains a total of *three* ballot numbers: (1) bmax, the highest ballot it has seen; (2) bacc, also referred to as the *accepted ballot* that is highest ballot in which it has accepted some value; (3) bori, also known as the *original ballot* that is ballot of the proposer that originally proposed the value (re-)proposed and accepted with ballot bacc. Traditional Paxos only maintains two ballot numbers at acceptors that are analogous to bmax and bacc, but OPaxos relies on bori as a connecting identifier of the secret shares of the same secret value without revealing any secret information.

Phase 2a: Recovery and Propose. The proposer waits to receive |Q1| PROMISE messages from acceptors in response to its PREPARE message. As in Paxos, if none of the PROMISEs report any secret shares accepted with a lower accepted ballot (bacc), the proposer is free to propose any value. However, if the PROMISEs did report shares accepted in lower ballots, the value recovery process is different, as shown in Figure 5, and described next.

From the |Q1| promises, the proposer needs to find a promise with the highest accepted ballot, denoted as S. The proposer then tries to find t promises, including S, whose original-ballot is the same as the original-ballot in S. If t such shares exist (line 7), the proposer needs to re-generate more shares from those t shares (line 7a) and reuse bo as the original ballot (line 7c). Else, the proposer is free to propose any value (line 8a) with the original ballot bori set to its own ballot bcur (line 8b).

Phase 2b: Accept. Upon receiving a proposal, if the proposer's ballot number is less than the highest ballot bmax the acceptor has seen so far, then the proposal is ignored (line 10), otherwise the acceptor accepts the proposal, and

[2] Some descriptions of Paxos refer to the *propose* phase as the *accept* phase

Phase-2a in OPaxos

Proposer P after receiving **PROMISE(**b, ba, $\langle$bo, $[y]\rangle$**)** messages from at least |Q1| acceptors with b = bcur:

Let R be the set containing all the promises received by P.
Let S be a promise with the highest accepted ballot ba in R.
Let G be a set of at least t promises whose original-ballot bo is equal to S's original-ballot: S.bo.

// $G \leftarrow \{$(ba, $\langle$bo, $[s'_1]\rangle$),
// (ba$_1$, $\langle$bo, $[s'_2]\rangle$),
// (ba$_2$, $\langle$bo, $[s'_3]\rangle$), ...$\}$

7: If G exist:
  a. $\{[s'_1], [s'_2], ..., [s'_n]\} \leftarrow$ **GenMissShare(**$[s'_1], [s'_2], ..., [s'_t]$**)**
  b. SS $\leftarrow \{[s'_1], [s'_2], [s'_3], ..., [s'_n]\}$
  c. bori $\leftarrow$ S.bo
8: Else:
  a. SS $\leftarrow \{[s_1], [s_2], [s_3], ..., [s_n]\}$ // from line 3
  b. bori $\leftarrow$ bcur
9: Send proposals containing secret-shares SS and original-ballot bori. For each $[s_i]$ in SS:
  a. Send **PROPOSE(**bcur,$\langle$ bori, $[s_i]$ $\rangle$**)** to the i-th acceptor.

Figure 5: Phase-2a in OPaxos: proposer recovers and re-proposes the previously accepted secret value s' or proposes any secret value s sent by a client (in Phase-1).

Phase-2b and Commit in OPaxos

Acceptor A_i upon receiving **PROPOSE(**b, $\langle$bo, $[s_i]\rangle$**)** message from a proposer P.

10: If $b <$ bmax, return. // ignore the message
11: bmax $\leftarrow b$
12: bacc $\leftarrow b$
13: bori $\leftarrow$ bo
14: ssval $\leftarrow [s_i]$
15: Send **ACCEPT(**bacc**)** to proposer P

↘↓↙

Proposer P after receiving **ACCEPT(**b**)** messages from at least |Q2| acceptors with b = bcur:

16: Respond to the client.
17: For each $[x_i]$ in SS: // SS and bori are
  a. Send **COMMIT(**bcur,$\langle$ bori, $[s_i]$ $\rangle$**)** // from line 9 to the i-th acceptor A_i.

Figure 6: Phase-2b in OPaxos: proposer learns whether proposed secret-shares are accepted by a Q2 quorum.

updates the highest ballot bmax, the accepted ballot bacc, the original ballot bori, and the secret-share ssval in durable storage (line 11-14).

Commit Phase. If the proposer receives |Q2| ACCEPT responses from acceptors for its proposed shares, it considers the corresponding secret value as decided and issues a COMMIT message to that effect to all acceptors (line 17).

4.3 Quorum Constraints

Unlike Paxos that can use simple majority quorums or any quorum sizes |Q1| and |Q2| respectively for the two phases that ensure a nonempty intersection, OPaxos with (t, n) threshold secret-sharing requires the two quorums to intersect in at least t acceptors:

$$|\text{Q1} \cap \text{Q2}| \geq t \tag{1}$$

A conservative quorum configuration used in OPaxos to satisfy that requirement is $|\text{Q1}| = |\text{Q2}| = \lceil \frac{n+t}{2} \rceil$. For example, if we have 4 acceptors with a (2,4) threshold secret-sharing scheme, the quorum size is 3 (incidentally same as majority quorums in Paxos with 4 acceptors), a configuration that can handle one failed acceptor. Figure 7 shows several possible configurations of acceptors in OPaxos compared to Paxos.

OPaxos enables flexible quorum sizes [24] satisfying the t-intersection constraint that trade off availability for common-case performance, e.g., by making Q2 smaller and Q1 bigger, we get higher capacity and slightly lower request latency under graceful conditions, however availability is still limited by the Q1 quorum. Our implementation defaults to the ceiling/floor quorums in (2) and (3) below very slightly favoring common-case performance without hurting fault tolerance.

$$|\text{Q1}| = \left\lceil \frac{n+t}{2} \right\rceil \tag{2}$$

$$|\text{Q2}| = \left\lfloor \frac{n+t}{2} \right\rfloor \tag{3}$$

4.4 Safety, Liveness, and State Integrity

The safety and liveness properties ensured by OPaxos are identical to those of Paxos with generalized quorums (as opposed to simple majority quorums), as outlined below.

Theorem (Agreement and Validity Safety). *OPaxos ensures that at most a single value that some proposer has proposed is chosen as the decision (i.e., committed in line 17.a of Phase-2b in Figure 6).*

The formal proof of this theorem is deferred to Appendix A. Here we outline key differences in the proof of safety compared to Paxos. Recall that Paxos' agreement safety relies on a crucial invariant, namely, a proposer can propose a value v with ballot n iff there exists a Q1 quorum of acceptors such that either (1) no acceptor in that quorum has accepted any proposal with ballot less than n; or (2) v is the value of the proposal with the highest ballot less than n accepted by any acceptor in that quorum. Let us call a value-ballot two-tuple (v, k) as *proposable* if it satisfies either part of that disjunctive condition in the invariant.

OPaxos generalizes the definition of *proposable(v,k)* so as to preserve a similar-in-spirit invariant crucial to proving agreement safety, and depends on (in general) distinct values of original ballot and accepted ballot for a proposal as well as the secret sharing threshold:

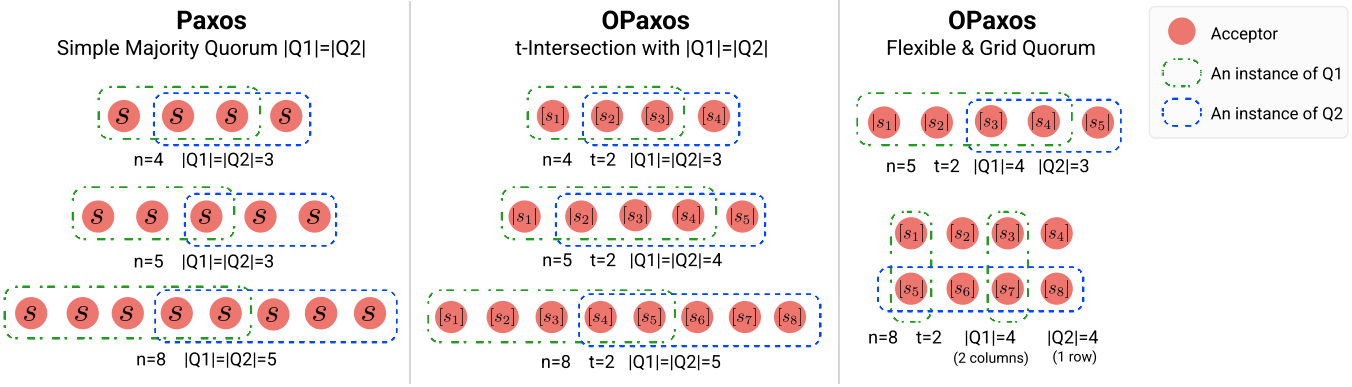


Figure 7: Example configuration of acceptor quorum set Q1 and Q2 in Paxos and OPaxos. Left: simple majority quorum, $|Q1| = |Q2| = \lceil \frac{n+1}{2} \rceil$; Middle: majority quorum with t-intersection, $|Q1| = |Q2| = \lceil \frac{n+t}{2} \rceil$; Right: flexible quorum with t-intersection, also grid quorum.

Definition (*proposable*). *proposable*(v,k) is defined as true iff there exists a Q1 quorum of acceptors such that either

i) at least t acceptors in that quorum accepted shares of v with the same original ballot in their respective highest accepted ballots less than k, and one of those t acceptors has the highest accepted ballot less than k across all acceptors in that quorum; or
ii) less than t acceptors in that quorum have accepted value shares with the same accepted ballot as the highest accepted ballot less than k across all acceptors in that quorum.

The formal proof shows that if a value v has been decided in some ballot k, *proposable*(x,j) is false for any $x \neq v$ and $j > k$, which combined with the invariant helps complete the proof. The proof relies on the t-intersection property of the Q1 and Q2 quorums in order to ensure that a decided value will always be reconstructable by a new ballot coordinator.

In Appendix H, we also formally describe the OPaxos-driven PBSSM protocol and prove that it preserves *primary integrity*, a critical property needed to prevent state corruption or divergence despite leader changes in any primary-backup replication system [27].

Liveness. OPaxos preserves Paxos' liveness property ensuring progress when at least max(|Q1|, |Q2|) acceptors are up and can communicate in a timely manner. Termination can not be guaranteed because of the FLP impossibility result, but the protocol will terminate when a single coordinator remains uncontested and long-lived enough to complete both phases of the protocol in the same round. Furthermore, because the threshold $t \leq \min(|Q1|, |Q2|)$, a decided value will always be reconstructable during periods of liveness.

4.5 Fast Oblivious Paxos

We have additionally developed Fast Oblivious Paxos (Fast-OPaxos), which similar in spirit to Fast Paxos [35] is an optimization that reduces end-to-end delay and also places less work on the bottleneck server, thereby also making it well suited to leaderless Paxos extensions [34, 45]. A detailed description of the threshold-based quorum constraints and formal proofs of safety and liveness are in Appendix D.

5 OPERATIONAL ODDS AND ENDS

5.1 Untrusted Mode Operation

The description thus far assumed that at least one trusted server is available to act as the PBSSM primary. We next describe the operation of an OPaxos-based system when no trusted server is available, and further separate it into two application sub-categories: 1) general state machines; 2) client-partitioned state machines, explained in turn below. The former can support arbitrary services but entails more overhead while the latter is more suitable for services whose underlying state is logically partitioned into small units mapped to corresponding end-clients that manage that state, for example, a key-value store wherein each key (or bag of keys) is mapped to a client that owns and manages the key-value pair(s). In both untrusted mode categories, trusted clients are assumed to perform secret-sharing.

General. Supporting general state machine services in OPaxos' untrusted mode requires secure multiparty computation. Unlike the PBSSM mode wherein a trusted server plays three roles: *primary*, *consensus leader*, and *secret dealer*, in the untrusted mode, a client plays the role of the secret dealer and an untrusted server acts as the consensus leader, and there is no primary as untrusted servers operate as a decentralized secret-shared state machine. Being based on SMPC, this mode fundamentally cannot support *request privacy*, i.e., untrusted nodes can and need to see the contents of client requests in plaintext in order to execute the request in a distributed manner, however the underlying application state is still secret-shared and therefore remains private from the

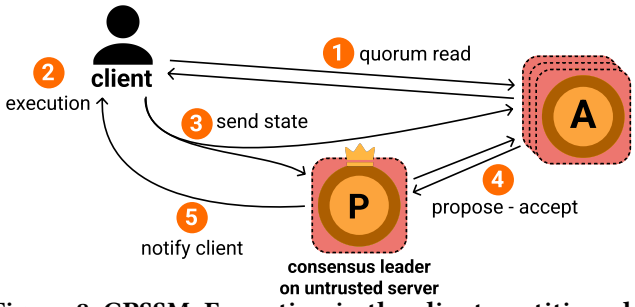


Figure 8: CPSSM: Execution in the client-partitioned secret-shared state machine.

untrusted servers as per the ideal-world/real-world model that in general is weaker than information-theoretic privacy. Specifically, SMPC in general only guarantees that untrusted servers learn no more information than they would by observing the input requests and the result of executing the requests, whereas information-theoretic privacy in OPaxos' PBSSM mode guarantees that untrusted servers can obtain no information by observing secret-shared requests as well as underlying application state.

Client-Partitioned. Supporting client-partitioned state machine services is much simpler, lower overhead, and enables information-theoretic privacy even in the untrusted mode as follows. To "execute" a request, a client dealer fetches the state partition it manages, e.g., a key-value pair, locally computes the result and re-distributes secret shares of any state modifications back to the untrusted servers (shown in Figure 8). It is straightforward to ensure request privacy by performing both read and write operations in a manner so as to appear identical, e.g., by always updating a nonce in the key-value pair even for reads to make them indistinguishable from writes. Furthermore, it is straightforward to hide the access pattern by using Oblivious RAM [18] techniques at the application level with a commensurate overhead cost.

5.2 Why Execution Before Agreement

In the trusted PBSSM mode, conducting execution before agreement or the other way round does not make a significant difference to overall request latency as either option allows the primary (a backup) to complete request handling in two (three) one-way network delays, however OPaxos' design choice prevents the possibility of state divergence when a trusted primary fails and another trusted server takes over as the consensus leader and request executing primary. The traditional sequence of agreement followed by execution decouples an already decided request from its underlying state transformation, which allows a new primary to issue state diffs different from those already applied by one or more backups and issued by the previous primary, so preventing state divergence because of nondeterministic request execution would require backups to support an *undo* operation to roll back and re-apply state diffs and for the protocol to guarantee detection of such potential divergence.

In contrast, with execution before agreement, undos are unnecessary for correctness at backups. A primary that crashes and recovers can simply roll forward from the most recent checkpoint only up until the highest cumulatively agreed upon slot number and follow the new primary's lead on subsequent state diffs. In the rare event of a new primary taking over after mistakenly suspecting the old primary as having failed (say because of asynchrony), a rollback at the old primary may still be necessary in which case OPaxos simply rolls forward the old primary from the most recent checkpoint (effectively emulating a crash), however the likelihood of such false positives can be engineered to be low with long-lived primaries and does not require additional protocol mechanisms to guarantee state convergence safety.

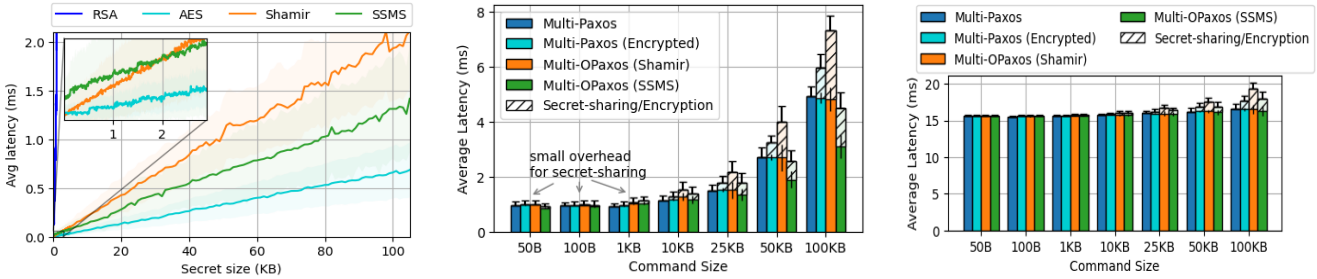
In the untrusted mode, a corner case with client-driven execution is when only a subset of servers have received secret shares of the updated state and the orchestrating client dealer fails midway. Fortunately, as with the PBSSM mode, performing execution before agreement can alleviate this scenario as follows: for each request, a client first collects a threshold number of secret shares from a Q1 quorum of backups for its partition, verifies that they have the same version number, locally executes the request, and then proposes secret shares of the corresponding updated partition as the next proposal whose order needs to be agreed upon by the untrusted servers. Successful agreement automatically ensures availability of a threshold number of secret shares of the updated partition whenever a majority is available (as a majority outnumbers the OPaxos' threshold by design).

6 IMPLEMENTATION CONSIDERATIONS

In this section, we describe several implementation considerations applicable to OPaxos as well as Fast-OPaxos.

Tracking state diffs in trusted mode. Our prototype implementation of PBSSM uses Linux's `strace` [39] to capture state diffs while executing each command, enabling us to support general state machines with modest overhead. For each executed command, PBSSM persistently modifies the state in the file system using file IO system calls (e.g `write` and `pwrite`). Thus, we can capture any file changes, secret-share them, and broadcast them to the backups before sending the execution results to the user. Simpler applications like key-value stores can directly implement OPaxos secret-sharing at the application layer without the overhead of `strace`.

Preventing duplicate shares. Because proposers independently reconstruct the Shamir polynomial and all n shares



(a) Secret-sharing and encryption latency **(b) Latency on LAN** **(c) Latency on WAN**

Figure 9: a) Microbenchmark for secret-sharing and encryption latency; b) Latency overhead of secret-sharing is negligible for small secret values, for larger values SSMS reduces the latency. c) The wide-area latency dominates, making the overhead of secret-sharing small.

using just t shares, it is possible for two different acceptors to end up receiving duplicate shares unless all trusted proposers have an a priori agreed upon deterministic way to permute shares across acceptors. Simplistic schemes like assigning the first (lowest x) share to the first acceptor or other simple mapping functions can reveal some information about the underlying polynomial, so the best practice is for trusted proposers to use an a priori known function to generate a random permutation of shares to assign to acceptors.

Faster randomization source. We learned from our implementation and prototype evaluation experience that the randomization used to generate secret-shares of a value can induce a significant overhead, which is consistent with prior work [58] in other secure systems contexts. In OPaxos, we use multiple workers for the secret-sharing process in different CPU cores, but in Linux, all those workers share the same random generator `/dev/urandom` which becomes the bottleneck. After observing this issue, we reverted to a cryptographically secure pseudorandom number generator (CSPRNG) based on the AES counter mode for which modern CPUs widely provide hardware support, e.g., AES-NI for cryptographic computation, that greatly increases the throughput of secret generation. We extended the open-source Shamir implementation of Hashicorp Vault [22] by using CSPRNG as the random source, leveraging SIMD for Galois arithmetic operation, and with more cache-friendly polynomial generation. Our open-source implementation [29], therefore does secret-sharing faster than the original.

Smaller secret-share size. OPaxos works with any threshold-secret sharing schemes, including SSMS [28] that enables secret-shares' size reduction. SSMS uses erasure-coding that reduces the shares' size by $1/t$ without revealing the value, even in a partial form. We use an open-source reed-solomon library [52] written in `Go` for implementing SSMS. This SSMS optimization is critical to improve the latency and capacity performance for large client request sizes.

Other optimizations. We note a few other optimizations though we have not implemented these. First, we can use grid quorums in OPaxos where the first quorum Q1 is strictly in the form of columns and the second quorum Q2 is in the form of rows, thereby further reducing the size of each quorum but at the cost of availability, as exemplified in Figure 7 (right), where with a total of 8 acceptors and $t = 2$, we only need 4 acceptors in both quorums; two columns for Q1 and one row for Q2. However, if say acceptors that store $[x_3]$ and $[x_5]$ fail simultaneously, the protocol cannot make progress as we cannot have a complete row for the first phase. Second, in some runs, we can have a smaller Q1, for example, when all the incoming promises are empty. Specifically, in Phase-1, when the proposer gets $(|Q1| - t + 1)$ empty promises, the proposer can directly start Phase-2. When that happens, waiting for the remaining $(t-1)$ promises will not change the fact that there is no recoverable value. This optimization is similar to that in PANDO [63] and can further be generalized in OPaxos, as described in Appendix B.

7 EVALUATION

Our prototype for OPaxos is implemented on top of Paxi [2], an open-source research framework for evaluating consensus protocols. We wrote the protocols using Go in around 9300 LoC, and it can be accessed at [30].

In this section, we evaluate the overhead of providing the privacy-preserving property in OPaxos (§7.1, §7.2), compared to Paxos; show how our OPaxos implementation handles node failures (§7.3); and demonstrate the latency for an emulated PBSSM (§7.4). Finally, we emulate a simple key-value store deployed across different cloud providers with no trusted server (§7.5).

The implemented prototype of Paxos and OPaxos do the typical consensus over a log of requests or state diffs (multi-decree). We call these multi-decree consensus as Multi-OPaxos and Multi-Paxos, but in this report we also refer them as OPaxos and Paxos for brevity. We use two secret-sharing schemes for OPaxos: Shamir and SSMS, and compare them

with ordinary Paxos also encrypted Paxos where the value is encrypted before being proposed.

Default evaluation setup. We evaluate OPaxos with five `m510` machines in the CloudLab cluster [10]. Each node has 2.0 Ghz Intel Xeon CPUs, 64GB RAM, and is interconnected via 1 Gbps network links. The clients run on a separate machine, and by default use 50 bytes values. TCP is used for client-leader communication as well as among the consensus instances. We use two trusted servers, $t = 2$, and the majority with t-intersection to determine the quorum sizes: for $n = 5$ and $t = 2$ the quorum sizes are $|Q1| = 4$ and $|Q2| = 3$. Beside this default setup, in some experiments, we vary the value sizes, emulate state-machine computation with a standard workload, and emulate WAN latency using Linux's `tc`.

7.1 Latency Overhead in OPaxos

We measured the latency overhead under low load, i.e., with a single outstanding request at any time. The *latency* is measured from the client side as the time since the client sent the request until it receives the response. For each protocol, a single client sends 100K requests to the leader. We plot the average latency with error bars showing standard deviation.

Secret-sharing and encryption microbenchmark. We first measure the latency of threshold secret-sharing schemes we use (Shamir [56] & SSMS [28]), and also the latency of standard encryption of RSA & AES. Using one of the machine as described in the evaluation setup, we measured the latency for (5,2) threshold secret-sharing: the time needed for generating $n = 5$ shares with $t = 2$, and the latency of the RSA & AES library in Go. As shown in Figure 9a, symmetric encryption such as AES has the lowest latency, followed by Shamir and SSMS. The latency of asymmetric encryption as RSA is too high for consensus usage, so we only consider AES for encrypted Paxos. We can see that with up to 44KB secret value, the secret-sharing and encryption latency is still below 1ms. For small secret values, the secret-sharing latency for Shamir is lower than SSMS. This trend will come in handy when we later measure OPaxos' latency overhead.

End-to-end latency overhead. With a stable OPaxos leader, every received value from the clients needs to be secret-shared, which induces additional latency compared to Paxos. However, as shown in the secret-sharing microbenchmark, for small secret value we expect the latency overhead to be small. The result is shown in Figure 9b wherein for small values up to 1KB, the secret-sharing latency overhead is less than 0.1ms. In real multi-cloud deployment, that latency overhead is negligible as the inter-node latency will typically be much higher, as we also show next.

Consistent with the microbenchmark in Figure 9a, we also see that with larger values the latency overhead in

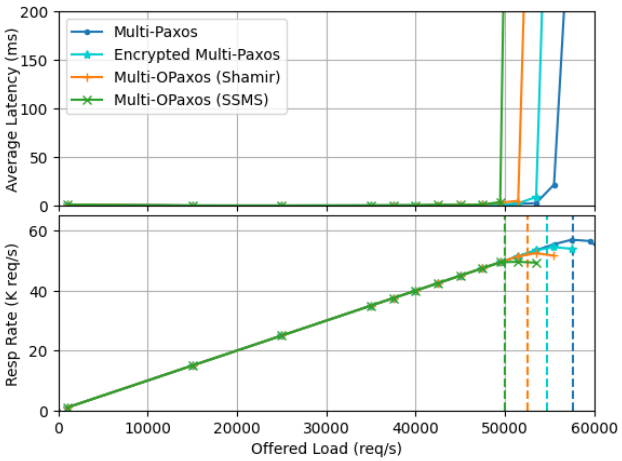


Figure 10: Secret-sharing overhead reduces the capacity of OPaxos compared to Paxos, as measured with 50 bytes values.

OPaxos also increases. For OPaxos with SSMS, the benefit of secret-shares' size reduction can be seen in the measurement with large values (10-100KB). With those large values, using Shamir is slower than SSMS; this is also consistent with the result in Figure 9a. The smaller overhead of using SSMS with large values is the result of both faster secret-sharing and less data transmission from the leader to acceptors. With 50-100KB values we can even see that OPaxos with SSMS offers lower or similar latency than the non-encrypted Paxos.

Latency on wide area network. The smaller overhead of using secret-sharing in OPaxos becomes more apparent when we run the measurement in a WAN measurement. We emulate WAN deployment by having 5ms RTT in the client-leader link and 10ms RTT in the inter-node networks. As shown in Figure 9c, the wide area latency dominates, making the overhead of secret-sharing negligible.

In Appendix C.1, we also analyze the impact of quorum sizes and show reduction in average latency with smaller Q2 quorum sizes (and thus larger Q1 quorums).

7.2 Capacity Overhead

In this evaluation, we measure the capacity overhead of OPaxos compared to Paxos. The secret-sharing process adds more work for OPaxos, thus OPaxos is expected to provide lower capacity than Paxos. We estimate the *capacity* of the system as the load when the difference between response-rate and the load grows more than 1%, wherein *load* is the rate of 50 bytes requests sent by the clients to the leader, and *response-rate* is the rate of responses received by the clients. We use 10 parallel clients to overload the system. We vary the load in Paxos and OPaxos, and for each load the clients send the requests for 30 seconds with inter-request times

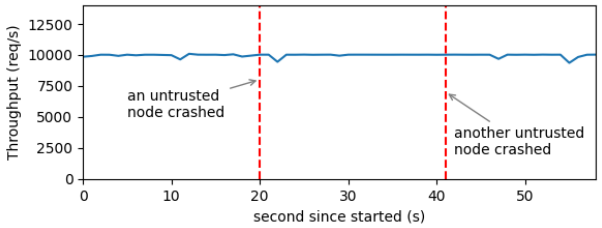


Figure 11: Failures of some untrusted servers.

that are Poisson- distributed. We additionally also record the average latency for every load.

Figure 10 shows the load-latency measurement with the default evaluation setup. When the load is near the system capacity, we can see the response rate flattening and the latency going up drastically. In our measurement, Paxos' capacity is around 57.7K req/s; encrypted Paxos capacity is around 54.7K req/s; while the capacity of OPaxos using Shamir and SSMS are 52.5K req/s and 51.0K req/s respectively; making OPaxos' capacity with Shamir and SSMS to be 91.3% and 86.9% of Paxos' capacity. These results are close to our expectation. Measured with the same setup in a single machine, we observed the secret-sharing throughput of Shamir and SSMS are around 683.5 K req/s and 536.3 K req/s, respectively; thus, we expect our implemented OPaxos to offer more than 85% of Paxos' capacity. We also show when OPaxos with SSMS can offer higher capacity compared to Paxos in Appendix C.2.

7.3 Performance Under Server Failures

Next, we show that our implementation of OPaxos is able to handle some node failures while making progress. However, compared to Paxos, the t-intersection requirement in OPaxos reduces the availability. Using the default setup with $n = 5, t = 2, |Q1| = 4$, and $|Q2| = 3$, we emulated node failures by dropping all incoming messages to some nodes.

We make the client continuously send requests to the leader with a constant load of 10,000 reqs/s, while we capture the throughput for every 100ms from the client. As shown in the Figure 11, with $n = 5$ nodes and $t = 2$, OPaxos can handle two failed untrusted nodes under a stable leader. However, the remaining three untrusted nodes after the 40s mark are not enough for leader election (Phase-1) as $f = 1$.

7.4 PBSSM Primary-Backup Computation

We try to run an arbitrary application in a primary-backup fashion using OPaxos and Paxos. We employ a primary that execute a TPC-C workload [61]. For each transaction the primary executes the queries on an sqlite database while capturing the state diffs. Then, the primary run consensus over the state diffs. With OPaxos, the primary secret-share the state diffs while in the encrypted Paxos, the primary encrypts the state diffs. The result is shown in Figure 12.

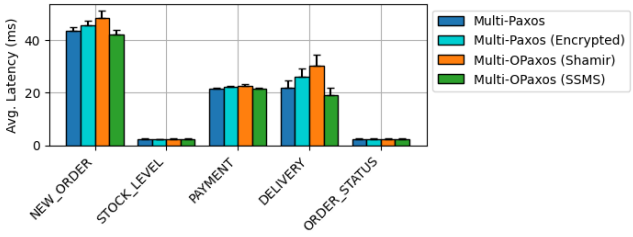


Figure 12: OPaxos and Paxos under TPCC Workload

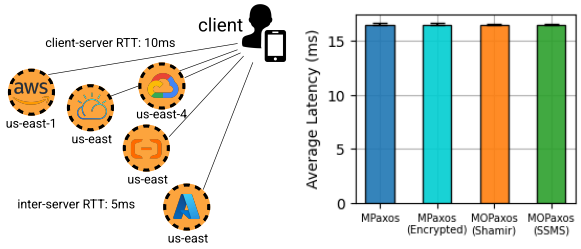


Figure 13: Key-Value Store without trusted leader. Left: emulated WAN setup. Right: client-perceived latency.

The latency depends on the transaction type, in general we can see that OPaxos with Shamir imposes latency overhead while OPaxos with SSMS can offers lower latency. This result is consistent with our previous latency measurements.

7.5 Key-Value Store On Untrusted Mode

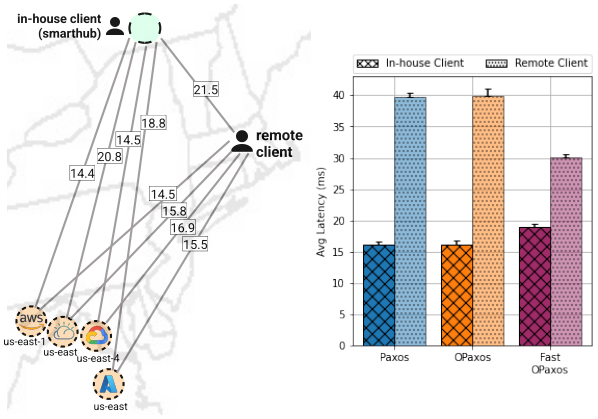
We also run a simple key-value store (KVS) with no trusted nodes available. The KVS is deployed in an emulated multi cloud providers as shown in the left of Figure 13. The client acts as secret-sharing dealer and directly broadcast the secret shares to all the nodes. One of the untrusted node acts as a leader to assign slot number to the client's proposed secret value. In the encrypted Paxos, the client only needs to send the encrypted value to the leader, not to all the nodes. As in previous measurement, we record the perceived latency from the client side under low load. The result in Figure 13 shows that the latency overhead of OPaxos is negligible.

8 SMART-HOME SYSTEM CASE STUDY

In this section, we describe a case study evaluation of OPaxos' usability in smart-home systems.

8.1 Current Smart-home System Issues

The prevailing architecture for smart-home systems today suffers from two key drawbacks: lack of privacy and lack of strong consistency guarantees, addressing which forms a key motivation for OPaxos. Privacy is a concerning issue because smart-home systems store sensitive personal data on third-party clouds making them vulnerable to breaches, as has also been observed by many prior works [4, 44, 53, 60]. To address this privacy concern, local-first smart-home systems



(a) Smart-home system setup (b) Smart-home latency

Figure 14: Smart-home system setup and the ping latencies, also client perceived latencies using smart-home actions dataset.

are being developed, for example Home-Assistant [23] and Hubitat [26] that prioritize local execution in the hub, even for features that traditionally rely on the cloud such as voice recognition [54]. However, they trade fault-tolerance for privacy by storing all the data locally in the hub making the data irrecoverable if the local storage is damaged.

Strong consistency is important because smart-home systems physically interact with users and physical systems thereby impacting their safety. For example, when there are concurrent updates to lock/unlock a smart-lock, or concurrent updates to routines managing surveillance devices, all of the backups must reflect the most recent configuration update and device state as any inconsistency could compromise physical safety. Melissaris et al. [43] shows this consistency issue in popular hubs like SmartThings and Vera where the events to unlock/lock a door can be *reordered* at multiple locations: the wireless protocol, the multi-threaded smart-home application, and in the storage layer, resulting in the door being erroneously left in a state that the user does not expect. The problem becomes more concerning when we have multiple backups that need to be synchronized. Saeida et al. [3] also highlights the importance of fault-tolerant event ordering in smart-home systems for safety.

We envision an OPaxos-based smart-home system wherein trusted home hubs/servers act as proposers and untrusted server across cloud providers act as backups, thereby ensuring strong consistency, high availability, and privacy.

8.2 Real World Workload Measurement

As a case study, we measure how OPaxos and Fast-OPaxos might perform, in terms of latency, in a real multi-cloud deployment under real smart-home systems access pattern. We instantiated virtual machine on AWS, GCP, Azure, and IBM Cloud, then measured the ping latency. We use the ping measurement results to simulate the system in our prototype by injecting delay when some nodes send messages to others; see Figure 14a for the smart-home system setup. The system has two types of clients: in-house and remote client. The in-house client resides in the house near the trusted node: the local smarthub, and the remote client is far from the house. The untrusted nodes that act as acceptors, for backup purposes, reside in different cloud providers.

We used the Mon(IoT)r [53] and PingPong [62] smart-home datasets that contain timestamped packet-captured files corresponding to actions the client performs on smart-home devices. We convert the datasets into tracefiles of read/write commands for our implemented KV store, e.g., an action to turn on a smart-device translates to a write command with value `{"state":"on"}` and device ID as key.

We got comparable latency measurement results for both datasets, which is expected since both datasets contain low-load, small-request actions typical for a smart-home system; it is rare to have concurrent clients in a smart-home system. As can be seen in Figure 14b, the overhead of doing secret-sharing in OPaxos is negligible compared to higher WAN latency. The in-house client's perceived latency for Paxos and OPaxos are 16.1ms and 16.2ms respectively. The latency for the in-house client using Fast-OPaxos is higher since a larger quorum is needed, and there is not much latency savings because the in-house client is near the leader. However, for the remote client, we can see that Fast-OPaxos offers lower latency compared to Paxos and OPaxos: the remote client experiences around 24% lower latency compared to either.

As a proof-of-concept, we have also integrated OPaxos into a popular smart-home system, Home-Assistant (HA), that supports fully-local operation. Internally, HA uses `sqlite` to store all updates. Users can manually backup the event updates [65, 67], but there might be critical updates that are missing after the last backup. There is a cloud-based backup plugin available [6], but it only backs up on a single provider and requires third-party account management. Our OPaxos-HA integration implements a simple listener for HA events provides consistent distributed backup storage while confining the perimeter of sensitive information leakage to the home, a novel combination of features that to our knowledge does not exist in any smart-home system. The purpose of this minimal proof-of-concept was to validate feasibility of integration; a more complete integration would require also implementing checkpoint/recovery mechanisms so as to maintain consistency between device state and database state that we have not implemented (but we do not foresee problems implementing them).

9 RELATED WORK

To our knowledge, this work is the first to develop a provably safe crash-fault-tolerant consensus protocol that operates

	Paxos[32]	Fast-Paxos[35]	RS-Paxos[47]	PANDO[63]	OPaxos	Fast-OPaxos
Privacy-preserving (confidentiality)	No	No	No	No	Yes	Yes
Min. intersection between Q1 and Q2	1	1	t	t	t	t
Client's Latency (message delay)	4	3 or 2*	4	6 or 4**	4	3 or 2*
Num. messages to/from the leader* **	$3n-1$	$2n$	$3n-1$	write: $2n+2$, read: $2n$	$3n-1$	$2n$
Value identifier	the value itself	the value itself	unspecified	rand. number/ hash	original-ballot	original-ballot

* Fast-Paxos and Fast-OPaxos reach consensus in one round-trip for a non-executable update when *all acceptors* directly notify the client, not the coordinator.
** PANDO does two-rounds write with delegation and one-round read under no conflict; one round-trip more is needed for client to contact *a frontend.*
*** The leader is an elected proposer in Paxos, RS-Paxos, and OPaxos; it is the coordinator in Fast-Paxos and Fast-OPaxos; it is a frontend in PANDO.

Table 3: Paxos variants as well as other consensus protocols with $t > 1$ intersection between the two phases.

over secret shares, however it builds upon a significant body of closely related work on consensus as well as privacy.

Paxos variants employing similar techniques. Using quorums with $t > 1$ intersection is not new, for example, RS-Paxos [47] and PANDO [63] rely on $t > 1$ intersection. However, their main objective is storage or latency reduction, which is achieved using erasure-coding. Likewise, CRaft [70] integrates erasure-coding into Raft [48], and adaptively replicates values either with full replication or erasure-coding depending on the number of available replicas, which enables it to achieve better availability. In contrast to these, our work primarily addresses privacy issues using secret-sharing, a goal not addressed by those works and one that introduces subtle yet important differences in the protocol design as well as implementation. Furthermore, we also develop Fast-OPaxos, an optimization to reduce the end-to-end delay of consensus to at most three one-way delays. Table 3 provides the context to position our work as compared to closely related prior work.

Secret-Sharing Systems. In the industry, many secret management systems, like Hashicorp Vault [21] and Mozilla SOPS [46], use secret-sharing to secure the master key held by multiple users. Proposed research systems, like Sieve [68], DepSKY [7], Ghostor [25] also use secret-sharing for similar purpose. However, they typically assume no concurrency or use expensive locking to prevent the inconsistency. Our proposed protocols enable concurrent proposers to broadcast secret-shares of different values while providing consistency through the agreement safety property.

Several recent works [5, 8, 64] have studied how to combine secret-sharing into Byzantine Fault Tolerant (BFT) consensus. These works address a harder technical problem than the one addressed herein by providing stronger guarantees compared to our work that targets a (benign) crash-prone failure model. However, like any BFT protocol, they induce a high quadratic message complexity, and also require a higher replication factor. They necessitate a more complex secret-sharing scheme so as to ensure the verifiability property, which requires quadratic message exchange or an external PKI. Our position is that, given industry norms and regulatory constraints, an honest-but-curious threat model resistant to $t-1$-collusion as in OPaxos is a point in the design space that offers a more practically desirable tradeoff between security guarantees and resource cost and complexity.

Other Privacy-Preserving Systems. Many systems protect user privacy in different ways, including in smart home system settings, such as: hiding the access pattern [9, 12, 20, 69], working only on encrypted data [16, 37, 50, 51], or doing private operation using secure multi-party computation (SMPC) [1, 11, 19, 38]. These approaches are complementary to OPaxos or too heavy handed for the scenarios of our interest. SMPC is a powerful approach enabling a state machine to be distributed across untrusted nodes but incurs high overhead; for instance, a recent work shows the need of 6-24 additional communication rounds for simple equality, inequality, and boolean addition operation [38]. Computing directly on encrypted data, in addition to the drawbacks of encryption-based approaches previously noted, either incurs high overhead or limits application generality.

10 CONCLUSION

We present OPaxos and Fast-OPaxos, a novel family of fault-tolerant privacy-preserving consensus protocols that allow a set of trusted and untrusted nodes to agree on secret-shared values while keeping the agreed upon values information theoretically hidden from the untrusted nodes, a system setup that is valuable for hybrid cloud deployments with honest-but-curious cloud nodes. We include rigorous formal proofs of correctness and TLA+ model checking of OPaxos. Our prototype-driven microbenchmarks show that OPaxos induces only a modest latency and capacity overhead in general, and for large requests, can even improve latency and capacity compared to traditional Paxos. Our case study evaluation shows that our approach can improve privacy in smart home systems from third-party cloud providers while maintaining high availability and strong consistency.

Our open-source implementation of OPaxos and Fast-OPaxos can be accessed at: https://opaxos.github.io.

ACKNOWLEDGEMENT

We thank the anonymous SoCC reviewers and our shepherd, Jelle Hellings, for their helpful comments and feedback. This work was supported by the National Science Foundation under grant numbers 1719386, 1717132, and 1900866.

REFERENCES

[1] Ittai Abraham, Benny Pinkas, and Avishay Yanai. 2020. Blinder–Scalable, Robust Anonymous Committed Broadcast. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1233–1252.

[2] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. 2019. Dissecting the Performance of Strongly-Consistent Replication Protocols. In *Proceedings of the 2019 International Conference on Management of Data*. 1696–1710.

[3] Masoud Saeida Ardekani, Rayman Preet Singh, Nitin Agrawal, Douglas B Terry, and Riza O Suminto. 2017. Rivulet: a fault-tolerant platform for smart-home applications. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 41–54.

[4] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If This Then What? Controlling Flows in IoT Apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1102–1119.

[5] Soumya Basu, Alin Tomescu, Ittai Abraham, Dahlia Malkhi, Michael K Reiter, and Emin Gün Sirer. 2019. Efficient verifiable secret sharing with share recovery in BFT protocols. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. 2387–2402.

[6] Stephen Beechen. 2022. Home Assistant Google Drive Backup. https://habackup.io/

[7] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *ACM Trans. Storage* 9, 4, Article 12 (nov 2013).

[8] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. 2008. DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. 163–176.

[9] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Lannan Luo. 2022. PFirewall: Semantics-Aware Customizable Data Flow Control for Smart Home Privacy Protection. In *The Network and Distributed System Security Symposium (NDSS) 2021*.

[10] CloudLab. 2023. https://www.cloudlab.us/.

[11] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 259–282.

[12] Trisha Datta, Noah Apthorpe, and Nick Feamster. 2018. A Developer-Friendly Library for Smart Home IoT Privacy-Preserving Traffic Obfuscation. In *Proceedings of the 2018 Workshop on IoT Security and Privacy* (Budapest, Hungary) *(IoT S&P '18)*. Association for Computing Machinery, New York, NY, USA, 43–48.

[13] Whitfield Diffie and Martin E Hellman. 1976. New Directions in Cryptography. *IEEE Transactions On Information Theory* 22, 6 (1976).

[14] Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014. Formal verification of software countermeasures against side-channel attacks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–24.

[15] David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. 2018. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security* 2, 2-3 (2018), 70–246.

[16] Luca Ferretti, Michele Colajanni, and Mirco Marchetti. 2012. Supporting Security and Consistency for Cloud Database. In *Proceedings of the 4th International Conference on Cyberspace Safety and Security* (Melbourne, Australia) *(CSS'12)*. Springer-Verlag, Berlin, Heidelberg, 179–193.

[17] Craig Gentry. 2010. Computing arbitrary functions of encrypted data. *Commun. ACM* 53, 3 (2010), 97–105.

[18] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 182–194.

[19] Peeyush Gupta, Yin Li, Sharad Mehrotra, Nisha Panwar, Shantanu Sharma, and Sumaya Almanee. 2019. Obscure: Information-Theoretic Oblivious and Verifiable Aggregation Queries. *Proc. VLDB Endow.* 12, 9 (May 2019), 1030–1043.

[20] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. 2016. Scalable and Private Media Consumption with Popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 91–107.

[21] Hashicorp. 2023. Vault by HashiCorp. https://vaultproject.io/.

[22] Hashicorp. 2023. Vault's Shamir Implementation. https://github.com/hashicorp/vault/tree/main/shamir.

[23] Home Assistant. 2023. https://home-assistant.io/.

[24] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2017. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems*.

[25] Yuncong Hu, Sam Kumar, and Raluca Ada Popa. 2020. Ghostor: Toward a Secure Data-Sharing System from Decentralized Trust. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 851–877.

[26] Hubitat. 2023. https://hubitat.com/.

[27] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 245–256.

[28] Hugo Krawczyk. 1993. Secret sharing made short. In *Annual international cryptology conference*. Springer, 136–146.

[29] Fadhil I. Kurnia. 2022. Go-Shamir: Fast Shamir Secret Sharing in Pure Go. https://github.com/fadhilkurnia/shamir.

[30] Fadhil I. Kurnia. 2023. Oblivious Paxos and Fast Oblivious Paxos Prototype. https://opaxos.github.io/.

[31] S. Lakshmanan, M. Ahamad, and H. Venkateswaran. 2003. Responsive Security for Stored Data. *IEEE Transactions on Parallel and Distributed Systems* 14, 9 (2003), 818–828.

[32] Leslie Lamport. 1998. The Part-time Parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

[33] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[34] Leslie Lamport. 2005. Generalized Consensus and Paxos. (2005).

[35] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19, 2 (2006), 79–103.

[36] Leslie Lamport. 2023. The TLA+ Home Page. https://lamport.azurewebsites.net/tla/tla.html.

[37] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. 2004. Secure Untrusted Data Repository (SUNDR). In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, San Francisco, CA.

[38] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. SECRECY: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation*.

[39] Linux strace. 2023. https://linux.die.net/man/1/strace.

[40] Thomas Loruenser, Andreas Happe, and Daniel Slamanig. 2015. ARCHISTAR: towards secure and robust cloud based data sharing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 371–378.

[41] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2021. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–37.

[42] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH computer architecture news* 40, 3 (2012), 118–129.

[43] Themis Melissaris, Kelly Shaw, and Margaret Martonosi. 2019. OKAPI: in support of application correctness in smart home environments. In *2019 fourth international conference on fog and mobile edge computing (FMEC)*. IEEE, 173–180.

[44] Hooman Mohajeri Moghaddam, Gunes Acar, Ben Burgess, Arunesh Mathur, Danny Yuxing Huang, Nick Feamster, Edward W. Felten, Prateek Mittal, and Arvind Narayanan. 2019. Watching You Watch: The Tracking Ecosystem of Over-the-Top TV Streaming Devices. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 131–147.

[45] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372.

[46] Mozilla. 2023. SOPS: Secrets OPerationS. https://github.com/mozilla/sops.

[47] Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. 2014. When paxos meets erasure code: Reduce network and storage cost in state machine replication. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 61–72.

[48] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319.

[49] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. 2021. Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics.. In *USENIX Security Symposium*. 2129–2146.

[50] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 85–100.

[51] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. 2014. Building Web Applications on Top of Encrypted Data Using Mylar. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 157–172.

[52] Klaus Post. 2023. Reed-Solomon. https://github.com/klauspost/reedsolomon.

[53] Jingjing Ren, Daniel J. Dubois, David Choffnes, Anna Maria Mandalari, Roman Kolcun, and Hamed Haddadi. 2019. Information Exposure From Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach. In *Proceedings of the Internet Measurement Conference* (Amsterdam, Netherlands) *(IMC '19)*. Association for Computing Machinery, New York, NY, USA, 267–279.

[54] Rhasspy. 2023. Rhasspy Voice Assistant. http://rhasspy.org/.

[55] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[56] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[57] Claude E Shannon. 1949. Communication theory of secrecy systems. *The Bell system technical journal* 28, 4 (1949), 656–715.

[58] Roman Shor, Gala Yadgar, Wentao Huang, Eitan Yaakobi, and Jehoshua Bruck. 2018. How to Best Share a Big Secret. In *Proceedings of the 11th ACM International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '18)*. Association for Computing Machinery, New York, NY, USA, 76–88.

[59] Arun Subbiah and Douglas M. Blough. 2005. An Approach for Fault Tolerant and Secure Data Storage in Collaborative Work Environments. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability* (Fairfax, VA, USA) *(StorageSS '05)*. Association for Computing Machinery, New York, NY, USA, 84–93.

[60] Madiha Tabassum, Tomasz Kosinski, and Heather Richter Lipford. 2019. "I don't own the data": End User Perceptions of Smart Home Device Data Practices and Risks. In *Fifteenth symposium on usable privacy and security (SOUPS 2019)*. 435–450.

[61] TPC. 2023. TPC-C: On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc/

[62] Rahmadi Trimananda, Janus Varmarken, Athina Markopoulou, and Brian Demsky. 2020. Packet-Level Signatures for Smart Home Devices. *Proceedings of the 2020 Network and Distributed System Security (NDSS) Symposium* (February 2020).

[63] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V Madhyastha. 2020. Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 157–180.

[64] Robin Vassantlal, Eduardo Alchieri, Bernardo Ferreira, and Alysson Bessani. 2022. COBRA: Dynamic Proactive Secret Sharing for Confidential BFT Services. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1528–1528.

[65] Marcel Van Der Veldt. 2019. Hassio Add-ons: Auto backup.

[66] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–18.

[67] Vorion. 2020. Create Automated Backups Every Day.

[68] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. 2016. Sieve: Cryptographically enforced access control for user data in untrusted clouds. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 611–626.

[69] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. 2017. Splinter: Practical Private Queries on Public Data. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 299–313.

[70] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. 2020. CRaft: An Erasure-coding-supported Version of Raft for Reducing Storage Cost and Network Cost. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 297–308.

A PROOF OF OPAXOS SAFETY AND LIVENESS

As in the body of the paper, we use the terms *round* and *accepted ballot* synonymously.

Definition A.1. *decided*(v,k): A value v is said to be decided in round k, denoted by the predicate *decided*(v,k), when a Q2 quorum of acceptors have all accepted a share of v with the same original ballot (denoted `bori` in the protocol) and the same accepted ballot k (denoted `bacc` in the protocol).

Theorem A.1. If a value v is decided in any round, no value other than v is proposed in any higher round.

In order to prove Theorem A.1, we first prove the protocol Invariant A.1 and Lemma A.1 below.

Definition A.2. *proposable*(v,k) is defined as true iff there exists a Q1 quorum of acceptors such that either

i) at least t acceptors in that quorum accepted shares of v with the same original ballot in their respective highest accepted ballots less than k, and one of those t acceptors has the highest accepted ballot less than k across all acceptors in that quorum; or
ii) less than t acceptors in that quorum have accepted value shares with the same accepted ballot as the highest accepted ballot less than k across all acceptors in that quorum.

Invariant A.1. If a proposer proposes a value v in some round k, then *proposable*(v,k) is true.

Proof. Suppose some proposer C is just about to propose v in round k. By protocol line 5, some Q1 quorum of acceptors, say Q, previously affirmed C's PREPARE(k) message.

We claim that at the instant the last (i.e., latest in time) acceptor in Q affirmed C's PREPARE(k) message, *proposable*(v,k) was true. This claim directly follows from the observation that no acceptor can accept any value in any round less than k after affirming C's PREPARE(k) message (by the round ascendancy promise in line 4 & 10) and the value recovery procedure (lines 7a-c). We next show that no protocol action can falsify *proposable*(v,k) at any future time after the last acceptor in Q affirmed C's PREPARE(k).

1) *An acceptor not in Q accepts a share of any value*: *proposable*(v,k) is trivially unaffected as Q still helps satisfy *proposable*(v,k).
2) *An acceptor in Q accepts a share of any value*: This acceptance must have necessarily happened in some round greater than or equal to k (again by the round ascendancy promise), thus *proposable*(v,k) is true.

Any non-acceptance actions trivially preserve *proposable*(v,k). Hence Invariant A.1. ∎

Lemma A.1. Any time after a value v is decided in round k, *proposable*(x,j) is false for any $x \neq v$ and $j > k$.

Proof. A value v is decided in round k, i.e., the predicate *decided*(v,k) becomes true, at the instant when the |Q2|'th acceptor accepts a value share of v with the same original ballot and with the accepted ballot k. *decided*(v,k) remains forever true after that instant and is false any time before.

We show that *decided*(v,k) implies *proposable*(x,j) is false for any $x \neq v$ and $j > k$ as follows. *decided*(v,k) implies that there is a Q2 quorum, say Q, of acceptors that accepted shares with the same original ballot, say m, and accepted ballot k. At the instant *decided*(v,k) became true, the second sub-condition of *proposable*(v,k) of the invariant can not be true, i.e., it is not possible for a Q1 quorum to exist such that it has no subset of t acceptors that have all accepted value shares with the same original ballot, which is because the quorum intersection property $|Q1 \cap Q2| \geq t$ ensures the existence of such a t-sized subset. Thus, after *decided*(v,k) becomes true, *proposable*(x,j) by definition can be true only if its first sub-condition holds true.

proposable(x,$k+1$) is not true at the instant *decided*(v,k) became true and can never become true thereafter because the highest accepted ballot less than $k+1$ is k and we know that at least |Q2| acceptors accepted shares of v with original ballot m and accepted ballot k, so the quorum intersection property guarantees that at least t of those acceptors are in any Q1 quorum. Thus, no proposer can ever propose a share of any value $x \neq v$ in round $k+1$. Furthermore, a proposal in round $k+1$ if any must be for a share of v with original ballot m (via protocol lines 7b-c).

Similarly, *proposable*(x,$k+2$) for $x \neq v$ is false at the instant *decided*(v,k) became true and any time thereafter because no value shares other than those of v could have been proposed in round $k+1$ as argued immediately above, therefore the value share accepted with the highest accepted ballot less than $k+2$ across acceptors in any Q1 quorum is a share of v, not x. By induction, it follows that *proposable*(x,j) for any accepted ballot $j > k$ and value $x \neq v$ remains false forever beyond the instant when *decided*(v,k) became true. Hence Lemma A.1. ∎

Proof of Theorem A.1. Theorem A.1 directly follows from the protocol invariant A.1 and Lemma A.1. ∎

The agreement safety property, namely that at most a single value can get decided (across all rounds), follows from Theorem A.1 and the observation that shares of at most a single value can be proposed in any round. The validity safety property, namely that only a proposed value can be decided, trivially follows from inspection of the protocol and Definition A.1.

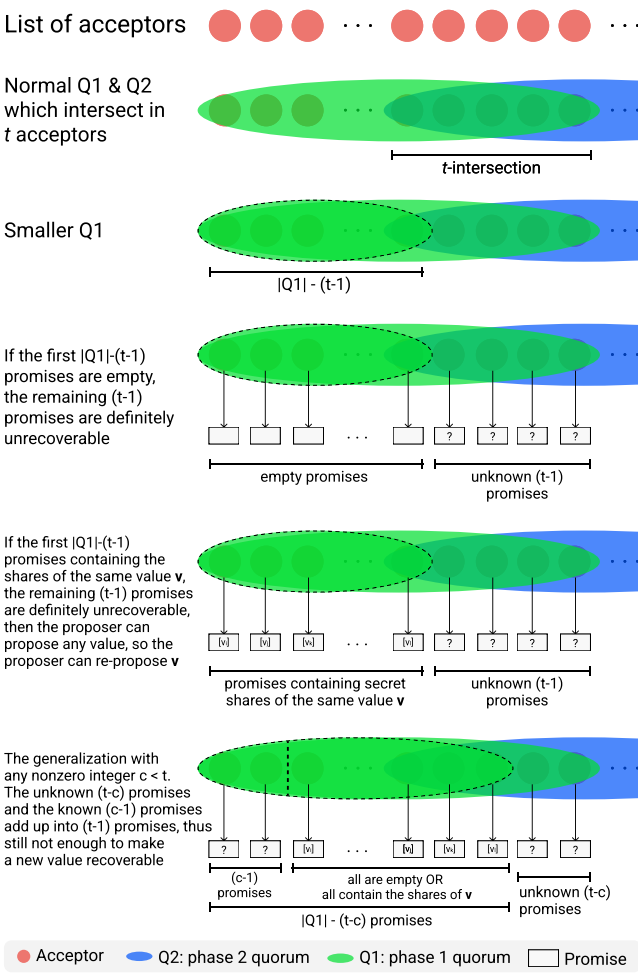


Figure 15: Smaller quorum size for Phase-1 can be used when it is sure that waiting for more promises will not lead to a new recoverable value since not enough shares will be present ($< t$).

OPaxos preserves the classic Paxos liveness property ensuring progress when at least max(|Q1|, |Q2|) acceptors are up and can communicate in a timely manner. Termination can not be guaranteed because of the FLP impossibility result, but the protocol will terminate when a single coordinator remains uncontested and long-lived enough to complete both phases of the protocol in the same round.

B OPAXOS' SMALLER PHASE-1 QUORUM

We have shown in the paper that OPaxos with $|Q1 \cap Q2| \geq t$ works and correct. Generally, we suggest using $|Q1| \geq |Q2|$ which make the quorum for Phase-1 larger compared to Paxos, especially when $t \geq 2$. That is suggested, considering the existence of a stable leader that makes Phase-1 rarely executed. For example, with $n = 5$ acceptors and $t = 2$, we suggest using $|Q1| = 4$ and $|Q2| = 3$.

We also have mentioned that we can have smaller Q1, that is when the first $(|Q1| - (t-1))$ promises are empty, as used in PANDO as well. Using the previous example with $n = 5$ acceptors, $t = 2$, $|Q1| = 4$, and $|Q2| = 3$; when a proposer receives 3 empty promises in Phase-1, the proposer can directly start Phase-2. That is, the proposer does not need to wait for the 4th promise. Even if the 4th promise is not empty (it has a secret-share), the secret value represented by that secret-share is certainly not chosen because a chosen value need to has secret-shares accepted in at least 3 acceptors. The fact that the first 3 promises are empty means there are only 2 acceptors left that can accept the secret-share, definitely not enough to make a secret value chosen.

Additionally, we can also have a smaller Q1 when the first $(|Q1| - (t-1))$ promises all contain the secret-shares of the same recoverable secret value. Let v denote that recoverable secret value. Again, we are going to explain this using the previous example with $n = 5$ acceptors, $t = 2$, $|Q1| = 4$, and $|Q2| = 3$. When a proposer receives the first 3 promises, and all of them contain the secret-shares of v, then the proposer can directly execute Phase-2 by re-proposing the secret-shares of v. That is, the proposer does not need to wait for the 4th promise. Even if the 4th promise contains the secret-share of another value w and has higher ballot number than all the ballot number in the first 3 promises, w is still not recoverable using those 4 promises: the first 3 promises have the secret-share of v and only the 4th promise has the secret-share of w. Thus, w is definitely not a chosen value and the proposer can propose any value, but the proposer is *forced* to re-propose v in this optimization. The proposer *must* re-propose v since in another case, the 4th promise might actually contain the secret-share of v and has an even higher accepted ballot than all the ballots in the first 3 promises. In this case, OPaxos' recovery rule instructs to re-propose v.

In summary, there are two conditions when we can use $(|Q1|-(t-1))$ as the size for the first quorum: either (i) all the promises are empty, or (ii) all the promises contain the secret-shares of the same recoverable value. If the $(|Q1| - (t-1))$ promises contain secret-share of multiple values, or only some of them are empty, then the proposer must use the ordinary size for Q1, thus the proposer need to wait for more promises.

We can further generalize the smaller quorum size as $(|Q1| - (t-c))$, where c is an integer and $0 < c < t$. The proposer only need to wait for $(|Q1| - (t-c))$ promises when all, but $(c-1)$, promises are empty or contain the secret-shares of the same recoverable secret value. If all but $(c-1)$ promises are empty, then the proposer can propose secret-shares of any secret value. If all but $(c-1)$ promises contain the secret-shares of v, then the proposer is *forced* to re-propose v's secret-shares. Please see Figure 15.

C ADDITIONAL OPAXOS EVALUATIONS

In this section, we evaluate OPaxos' performance with different Q1 and Q2 quorum sizes and different value sizes.

C.1 Latency With Different Quorum Sizes

We measured the latency of OPaxos by varying the quorum size. In this measurement, we use majority quorum for Paxos and vary the quorum sizes for OPaxos. With smaller quorum size for Phase-2 (and larger quorum size for Phase-1), the leader waits for fewer acceptors to learn a chosen secret value. Therefore, the average latency with smaller Phase-2 quorum is also smaller, as can be seen in Figure 16. Using Shamir, OPaxos with $|Q1| = 5$ and $|Q2| = 2$ has lower average latency compared to Paxos with majority quorum, however that configuration for OPaxos is not fault-tolerant as any proposer needs all acceptors to be alive for Phase-1.

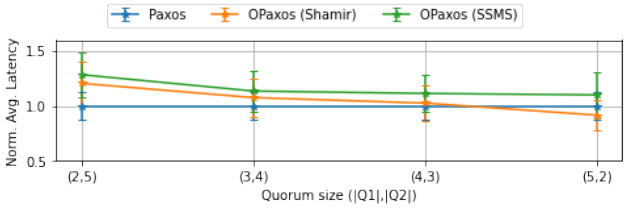


Figure 16: The latency of OPaxos with varying quorum sizes and $t = 2$ compared to Paxos with majority quorum as the baseline. With a stable leader, the latency tends to decrease for smaller Q2 and larger Q1, but it also becomes less fault-tolerant.

C.2 Capacity With Large Values

We present a scenario where OPaxos can offer higher capacity compared to Paxos. From the latency measurement, previously shown in §7.1, we already see that OPaxos can have lower latency compared to Paxos for consensus over large values. That is achieved by using SSMS that reduces the overall shares' size using erasure-coding. We measured the load-latency relationship for OPaxos and Paxos with various large value sizes (0.5-4KB). The result is shown in Figure 17. Consistent with the low load latency measurement (§7.1), for large values the capacity of OPaxos with SSMS is *higher* than Paxos.

D FAST OBLIVIOUS PAXOS

In this section, we show how the OPaxos design can be extended to *Fast Oblivious Paxos* (Fast-OPaxos), similar in spirit to Fast-Paxos [35], an optimization that reduces end-to-end delay and also places less work on the bottleneck server, thereby also making it well suited to leaderless Paxos extensions [34, 45].

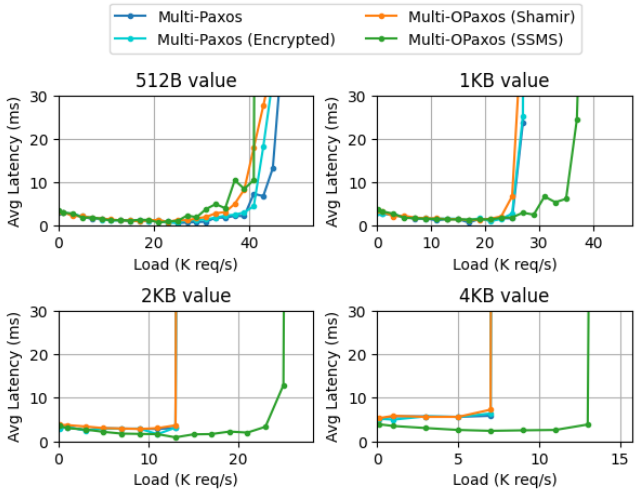


Figure 17: Capacity under large values: using SSMS to reduce the shares' size can increase OPaxos' capacity.

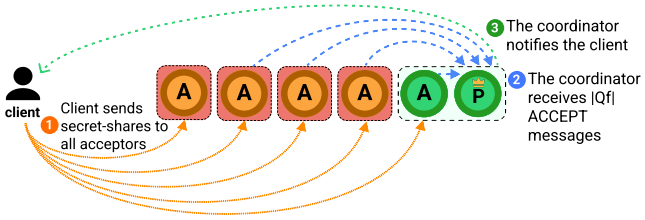


Figure 18: Fast path in Fast-OPaxos

Fast-Paxos primer. Fast-Paxos leverages two key observations about Paxos: (1) the quorum sizes may be different for different rounds provided the quorum intersection property is always obeyed; (2) acceptors may accept values from different proposers in the same round, including directly from clients, provided that round's coordinator commits a decision only if the same value is accepted in its round by a sufficiently large Phase-2 quorum of acceptors.

Accordingly, Fast-Paxos introduces the following adaptations. First, there are two types of rounds, *fast* and *classic*, that have different Phase-2 quorum sizes. Second, a *coordinator proposer* (as distinguished from a *client proposer*) in the common case of a fast round, after acquiring a Q1 quorum in the first phase, may skip the second phase and instead issue a special ANY value to acceptors authorizing them to accept the first value they receive directly from any client proposer. Third, each acceptor that overwrote its ANY placeholder with a client proposed value reports it to the coordinator for that round, and if a coordinator receives a Qf quorum of acceptances of the same value in its round, it commits that value as the decision, else it reverts to the slow path wherein it determines the value to propose based on the Qf reports and, like in classic Paxos, attempts to get a Q2 quorum of acceptors to accept it in order to commit it as the decision.

Protocol overview. Fast-OPaxos extends OPaxos with a high-level structure similar to Fast-Paxos, but with important differences in (1) quorum constraints; and (2) the procedure to recover values from lower rounds, which we explain next.

D.1 Quorum Constraint

Fast-OPaxos requires the three quorums, the fast quorum Qf, the prepare quorum Q1, and the propose quorum Q2 to satisfy the OPaxos quorum constraint $|Q1 \cap Q2| \geq t$ and additionally the following two constraints:

$$|Q1 \cap Qf| \geq t \tag{4}$$
$$|Q1 \cap Qf_a \cap Qf_b| > 0 \tag{5}$$

The first constraint helps ensure that a coordinator proposer can always obtain enough shares to reconstruct a client-proposed secret value decided in the fast path by way of acquiring a Qf quorum. The second constraint is identical to that in Fast-Paxos and helps ensure that, in the event of conflicting client proposals in the same round, shares of two different values can not both acquire $\lceil \frac{|Q1|}{2} \rceil$ acceptances, the criterion used in the protocol to select from multiple reconstructable values.

Fast quorum with n acceptors. The quorum constraints above can be used to derive the default fast quorum instantiation in our Fast-OPaxos implementation. From Eq. (4) we get:

$$\begin{aligned} |Q1| + |Qf| - n &\geq t \\ |Qf| &\geq n + t - |Q1| \end{aligned} \tag{6}$$

Then, from Eq. (5) we get:

$$\begin{aligned} |Q1| + (2|Qf| - n) - n &> 0 \\ |Qf| &> n - \frac{|Q1|}{2} \end{aligned} \tag{7}$$

Since $t > 0$ and $|Q1| > 0$, Eq. (7) provides a tighter lower bound than Eq. (6). Using Eq. (7) and substituting $|Q1|$ with OPaxos' default Q1 quorum size in Eq. (2), we get:

$$\begin{aligned} |Qf| &\geq n - \frac{\frac{n+t}{2}}{2} \\ |Qf| &\geq \frac{3n - t}{4} \end{aligned} \tag{8}$$

Finally, assuming that $|Qf| \geq |Q1|$ as in Fast-Paxos, we get our final fast quorum size as

$$|Qf| = \max\left(\left\lceil \frac{3n - t}{4} \right\rceil, |Q1|\right) \tag{9}$$

D.2 Fast-Path Execution

Preparation. The fast round starts before the client sends secret-shares to acceptors. A proposer, with fast ballot-number, does the preparation as depicted in Figure 19. The proposer collects |Q1| empty promises from the acceptors then sends

Phase-1 of Fast-OPaxos

Proposer P on starting Phase-1:
1: bcur ← bcur + 1
2: Send **PREPARE(**bcur**)** to all n acceptors.

Acceptor A upon receiving **PREPARE(**b**)** messagefrom Proposer P:
3: If $b <$ bmax, return. // ignore the message
4: bmax ← b.
5: If ssval = Any, drop the special Any value:
a. bacc ← null.
b. bori ← null.
c. ssval ← null.
// bacc, bori, and ssval can be empty (null)
6: Send **PROMISE(**bmax, bacc, ⟨bori, ssval⟩**)** to P.

Figure 19: Fast round preparation in Fast OPaxos

Phase-2a of Fast-OPaxos

Proposer P after receiving **PROMISE(**b, ba, ⟨bo, [y]⟩**)** messages from at least |Q1| acceptors with b = bcur, it becomes the leader, then:
7: If there is a non-empty promise, then start the Recovery (§D.3).
8: Else, propose the special Any value:
a. Send **PROPOSE(**bcur, Any**)** to all n acceptors.

Acceptor A upon receiving **PROPOSE(**b, Any**)** message from the Proposer P which already becomes a leader:
9: If $b <$ bmax, return. // ignore the message
10: bmax ← b.
11: bacc ← b.
12: bori ← null.
13: ssval ← Any.
14: $L \leftarrow P$ // store the current coordinator leader

Figure 20: The leader broadcasts the special Any value

the special Any value to all acceptors. The acceptors that received the special Any value can start accepting a secret-share from the client (Figure 21). Then, the acceptors execute the Phase-2b as described in Figure 22.

Client proposer. After the preparation, a client can directly propose a value to all servers without first contacting the coordinator. The client proposer has its own per-request ballot number that acts as the original-ballot for the secret-shared value (bcur in line C3). As shown in Figure 22, the coordinator leader L waits for |Qf| of acceptances from the acceptors before it can send the decision to the client. When there is no concurrent client proposer, all the |Qf| acceptances received by the coordinator are likely to contain the proposal from the same client (as identified by the original-ballot) making the fast-path successful. Otherwise, when not all the |Qf| acceptances contain the same secret-shared value, the coordinator must fall back to the slow path by running Phase-1 again.

Client actions in Fast-OPaxos

Client C on proposing a secret value s:

C.1: bcur ← bcur + 1

C.2: SS ← {[s_1], [s_2], ..., [s_n]} ← **SecretShare(**s**)**

C.3: For each [s_i] in SS:

a. Send **FAST-PROPOSE(**bcur, [s_i]**)** to the i-th acceptor A_i.

C.4: Wait for notification from the leader.

// Optionally, if the acceptors also send acceptance info
// to the client, the client can broadcast commit
// to all the acceptors after receiving |Qf| acceptances.

Figure 21: Client actions in Fast-OPaxos

Phase-2b of Fast-OPaxos

Acceptor A_i upon receiving **FAST-PROPOSE(**⟨bo, [s_i]⟩**)** message from a client C.

15: If ssval ≠ ANY, return. // ignore the message

16: bori ← bo

17: ssval ← [s_i]

18: Send **ACCEPT(**bacc, ⟨bori, ssval⟩ **)** to the coordinator leader L

Leader L, after receiving **ACCEPT(**b, ⟨bo, [y_i]⟩**)** messages from at least |Qf| acceptors with b = bcur:

19: If all the |Qf| acceptances contain the same original-ballot bo, the secret value is chosen:

a. {[y_1], [y_2], ..., [y_n]} ← **GenMissShare(**[y_1], [y_2], ..., [$y_{|\mathbf{Qf}|}$]**)**

b. SS ← {[y_1], [y_2], ..., [y_n]}

c. Notify the client who propose the value.

d. Send **COMMIT(**bcur,⟨bo, [y_i]⟩**)** for each acceptor A_i.

20: Else, the fast path is failed. Fallback to the slow path, by restarting Phase-1.

Figure 22: The leader waits for |Qf| ACCEPT messages

Similar to Fast-Paxos, the coordinator can skip Phase-1 in the slow path if the coordinator is also the proposer of the subsequent ballot number.

D.3 Slow-Path: Collision Recovery

When a coordinator proposer receives a Q1 quorum of PROMISE messages, if it is able to reconstruct any secret value from the |Q1| PROMISE messages in the prepare phase, it automatically enters the slow path and attempts to obtain a Q2 quorum for secret shares of *some* (non-ANY) value. The reconstruction procedure is similar to OPaxos in that the proposer looks for at least t shares that have been accepted with the same original ballot and one of those shares was accepted with the highest accepted ballot across all acceptors in that Q1 quorum. A key difference in Fast-OPaxos however is that the proposer may be able to reconstruct more than one secret value proposed by conflicting client proposers. If so, it selects the value whose secret shares have been accepted by the greatest number of acceptors in that Q1 quorum. If it can reconstruct exactly one value, it proposes that value. Else, it can either propose any non-ANY value in the slow path or propose the ANY value setting up the fast path.

Recovery in Fast-OPaxos

Upon starting the recovery process, after the proposer P receives |Q1| promises, **PROMISE(**b, ba, ⟨bo, [s_i]⟩**)**:

Let Q be the set of non empty promises from those |Q1| promises.
Let k be the highest accepted-ballot ba in Q.
Let V be the set of secret-shares in Q whose original-ballot bo is equal to the original-ballot of *any* promises whose k is the accepted ballot.

// V ← {(k, ⟨bo_1, [s'_1]⟩), (k', ⟨bo_1, [s'_2]⟩), ... ,
// (k, ⟨bo_2, [s''_1]⟩), (k'', ⟨bo_2, [s''_2]⟩), ... ,
// (k, ⟨bo_3, [s'''_1]⟩), (k''', ⟨bo_3, [s'''_2]⟩), ... ,
// ... }

R.1: If there is only one recoverable value s^* in V whose original-ballot is bo, then repropose s^*:

a. {[s^*_1], [s^*_2], ..., [s^*_n]} ← **GenMissShare(**[s^*_1], [s^*_2], ..., [s^*_t]**)**

b. SS ← {[s^*_1], [s^*_2], [s^*_3], ..., [s^*_n]}

c. bori ← bo

R.2: Else, if there are multiple recoverable values in V, and there is a single secret value s^* whose secret-shares form a majority –among the |Q1| promises– and bo is the original-ballot, then repropose s^*:

a. {[s^*_1], [s^*_2], ..., [s^*_n]} ← **GenMissShare(**[s^*_1], [s^*_2], ..., [s^*_t]**)**

b. SS ← {[s^*_1], [s^*_2], [s^*_3], ..., [s^*_n]}

c. bori ← bo

R.3: Else, propose any value s which initially P want to propose:

a. SS ← {[s_1], [s_2], ..., [s_n]} // shares generated previously

b. bori ← bcur // P's current ballot

// Optionally, P can also propose the special ANY value
// to start a new fast round.

R.4: For each [s_i] in SS:

a. Send **PROPOSE(**bcur,⟨ bori, [s_i] ⟩**)** to all the i-th acceptor.

Figure 23: Recovery process in Fast-OPaxos

The formal proof of Fast-OPaxos' agreement safety and liveness properties is in Section E.

E PROOF OF FAST-OPAXOS SAFETY AND LIVENESS

Below, we use the term *leader* (instead of *coordinator* or *proposer*) in order to distinguish it from *client* proposers.

Definition E.1. A value v is said to be decided either when a quorum Qf of acceptors have all accepted v in the same fast round (via protocol lines 16-17) or a quorum Q2 of acceptors have all accepted v in the same slow round (via OPaxos protocol line 14). We denote this event by *decided*(v,k) where k is the fast or slow round as applicable.

A value is said to be *fast-committed* by a leader (via protocol line 19d) when it receives |Qf| acceptances (via line

19) from acceptors in response to a FAST-PROPOSE from a client, all with the same original ballot and accepted ballot.

Theorem E.1. If a value v is decided in some round, no value other than v is proposed or fast-committed by any leader in any higher round.

Note that unlike Theorem A.1 in classic OPaxos, the theorem above includes "*or fast-committed*" because in Fast-OPaxos, a leader may decide a value proposed by a client without having proposed it itself.

In order to prove Theorem E.1, we introduce the following two lemmas. Unlike OPaxos, an invariant-based proof does not appear to make the proof shorter or easier to read and verify here, hence the more direct formal proof structure.

Lemma E.1. After a value v is decided in some round, a leader can not issue an Any value proposal in any higher round.

Proof. Suppose a value v is decided in some round k. We first show that it is not possible for some leader C to issue an ANY value proposal in round $k+1$, and then complete the proof by induction on rounds.

C can issue an Any value proposal in round $k+1$ only if it acquires a Q1 quorum of PROMISE messages containing accepted secret shares from which no accepted value in any round less than k+1 is recoverable (by line 8). By assumption, v is decided in round k, therefore either a Qf quorum of acceptors accepted v in fast round k or a Q2 quorum of acceptors accepted v in slow round k. Since $|Q1 \cap Qf|$ as well as $|Q1 \cap Q2|$ are at least t, there are at least t acceptors in that Q1 quorum that accepted matching shares of v and furthermore, those acceptors must have done so before affirming C's round $k+1$ PREPARE$(k+1)$ message (as that is an implicit promise not to accept any values in lower rounds). Thus, C will reconstruct at least one (non-Any) value as part of its PROMISE messages and will therefore not propose an Any value message in round $k+1$.

Suppose no leader proposes an Any value in any rounds in $[k+1, k+n]$, both inclusive, for some $n > 0$. It follows that no acceptor in round k's Qf or Q2 quorum could have overwritten its accepted value v in round k with the Any value in any round in $[k+1, k+n]$. Thus, a round $k+n+1$ leader will necessarily receive a (non-Any) value as part of its PROMISE messages via the same argument as the base case above, therefore it will not propose an Any value in round $k+n+1$. Hence proved. ∎

Corollary to Lemma E.1 : *After a value v is decided in some round, no acceptor can accept a (fast round) client proposal for any value in a higher round.*

Proof. Suppose value v is decided in round k. In any higher round $j > k$, an acceptor can accept a client proposal only if it previously received an Any value proposal from the leader for round j, which is ruled out by Lemma E.1. ∎

Lemma E.2. If a value v is decided in some round k and a leader proposes a value x in some higher (slow) round i, then either $x = v$ or there exists a slow round j in (k, i) in which some leader proposed x.

Proof. A leader can propose $x \neq v$ only if it recovered that value x via PROMISE messages in some round j higher than k and lower than i (by the recovery protocol line R.1 or R.2), which means that x was accepted by at least t acceptors in round j.

Round j can not be a fast round because of the corollary to Lemma E.1. Thus, round j must be a slow round, i.e., some leader must have proposed x. Hence proved. ∎

Lemma E.3. If a value v is decided in some round k and a leader proposes a value x in some higher (slow) round i, then $x = v$.

Proof. Consider the lowest round j in (k, i) where some leader C proposed x, whose well-definedness is ensured by Lemma E.3 above. C must have received a Q1 quorum of acceptances in its PROMISE messages such that the value reported as accepted in the highest lower round and accepted by more acceptors in that round than any other value (protocol line R.2) is x.

For v to be decided in round k, there exists either a quorum Qf of acceptors that previously, i.e., before receiving C's PREPARE(k) message, accepted v from some client or there exists a Q2 quorum of acceptors that previously accepted v from some leader's ACCEPT message. Note that if one of those acceptances had not previously happened, they can not happen subsequent to C acquiring a Q1 quorum in round j (because of the round ascendancy promise). In either of those two cases – a Qf quorum acceptance of a client-proposed value or a Q2 quorum acceptance of a leader-proposed value – leader C would have either reconstructed value v as the only value in its PROMISE messages or there are two cases to consider:

Case 1: There is another value x that was also accepted by some acceptor in round k (necessarily from a client in the fast round k); or

Case 2: There is another value x that was accepted in a round strictly in between k and j (also necessarily from a client in a fast round).

We can rule out the latter case using Lemma E.1 as follows: a fast-round acceptance by some acceptor of x proposed by some client is possible in a round in (k, j) only if that acceptor had previously received an Any value proposal from the leader for that round, which is not possible as per the corollary to Lemma E.1. Thus, Case 1 is the only possibility as we consider next.

For v to have been decided in round k, it must have been accepted by a Qf quorum of acceptors. Thus, x could have been accepted by at most $N - |\text{Qf}|$ acceptors. By design assumptions, we have for any two Qf quorums Y and Z and a Q1 quorum Q, the intersection of Y, Z, and Q is nonempty. That is, $|Y \cap Q| > N - |\text{Qf}|$ where N is the total number of acceptors. Thus, if value v was accepted by a Qf quorum Y and the leader for round j acquired a Q1 quorum Q in its PROMISE messages, the number of affirmations $|Y \cap Q|$ reporting the value v will necessarily outnumber the maximum number $N - |\text{Qf}|$ of acceptors that could have accepted x in round j. Therefore, by the recovery protocol line R.2, choosing the recovered value as the value accepted by the greatest number of acceptors in the highest lower round, the leader C must necessarily choose v over x. Hence proved. ■

Proof of Theorem E.1. Lemma E.3 rules out any value other v being proposed by a leader in a higher round, so we only need to now argue that no other value can be fast-committed by way of acquiring a Qf quorum in a higher round from a client proposal. The latter follows from the corollary to Lemma E.1 that precludes any acceptor from accepting any client proposal in any higher round. Hence Theorem E.1. ■

The proof of agreement safety directly follows from Theorem E.1 because either a proposal or fast-commit by a leader of shares of a value is necessary for that value to be committed as a decision, thus any value that is decided in any round must be the same as the value that is decided in the lowest round in which a value got decided. As in OPaxos, validity follows from the protocol that can only decide proposed values.

Fast-OPaxos's liveness property is similar to that of Fast Paxos ensuring fast-path (slow-path) progress during periods when at least $\max(|\text{Q1}|, |\text{Q2}|, |\text{Qf}|)$ ($\max(|\text{Q1}|, |\text{Q2}|)$)acceptors are up and can communicate in a timely manner with termination possible only when a single leader remains uncontested and long-lived enough to complete the two steps of a fast-commit or the two phases of the slow path.

F TLA+ SPECIFICATION OF OPAXOS

Beside the formal proof in Appendix A, we also write OPaxos's specification in TLA+[36], a language for model checking concurrent or distributed systems. The specification checks for the correctness property: when there is a secret value v with original-ballot bo *decided* in a round with ballot-number b, then there should be no other value (identified by other original-ballot) proposed with higher ballot-number. The model-checking process can help us to identify any violation in our protocol. The model checker runs by checking various possibilities based on the provided specification, especially for a small number of possible states. We also publish our TLA+ code at [30], along with OPaxos implementation.

We did the model-checking on the `rs630` machine with all of its 20 CPU cores (20 model-checking workers) in the CloudLab. We did the model-checking by having 1-4 concurrent proposers, 4-6 acceptors, threshold $t = 2$, and 2-3 possible values. At the end of the model-checking process, we did not find any correctness violation.

MODULE *OPaxos*

EXTENDS *Integers*, *FiniteSets*, *TLC*
CONSTANTS *Acceptors*, *Value*, *MaxBallot*, T, $NumQ1$, $NumQ2$

$Ballot \triangleq 1 \,..\, MaxBallot$
$Quorum1 \triangleq \{subset \in \text{SUBSET } Acceptors : Cardinality(subset) = NumQ1\}$
$Quorum2 \triangleq \{subset \in \text{SUBSET } Acceptors : Cardinality(subset) = NumQ2\}$

The intersection between any $Phase1$ quorum $Q1$ and $Phase2$ quorum $Q2$ should intersect in at least T acceptors, *i.e.*, $|Q1| \cup |Q2| \geq T$
ASSUME $QuorumAssumption \triangleq (\forall Q1 \in Quorum1 : \forall Q2 \in Quorum2 : Cardinality(Q1 \cap Q2) \geq T)$

VARIABLE
- $maxBal$, — $maxBal[a]$: (bmax) the highest ballot seen by acceptor a
- $accBal$, — $accBal[a]$: (bacc) the ballot of the accepted share in acceptor a
- $accSS$, — $accSS[a]$: ($ssval$) the accepted secret-share in acceptor a
- $accSSOri$, — $accSSOri[a]$: (bori) the original-ballot of the accepted secret-share in acceptor a
- $msgs$ — Set of messages sent by proposers and acceptors, *i.e.*, message history

Message is a set of all possible messages sent by proposers and acceptors

$Message \triangleq$
- $[type : \{\text{"1a"}\},\ bal : Ballot]$ — $PREPARE(b)$
- $\cup$ $[type : \{\text{"1b"}\},\ acc : Acceptors,\ bal : Ballot,\ accbal : Ballot \cup \{-1\},\ oribal : Ballot \cup \{-1\},\ ssval : Value \cup \{\text{""}\}]$ — $PROMISE(b,\ ba,\ (bo,\ s))$
- $\cup$ $[type : \{\text{"2a"}\},\ bal : Ballot,\ oribal : Ballot,\ ssval : Value]$ — $PROPOSE(b,\ (bo,\ s))$
- $\cup$ $[type : \{\text{"2b"}\},\ acc : Acceptors,\ bal : Ballot,\ oribal : Ballot,\ ssval : Value]$ — $ACCEPT(b)$

TypeOK ensures the correctness of the variables. Empty ballot is indicated with -1. Empty secret-share is "".

$$\begin{aligned} TypeOK \triangleq\ & \wedge maxBal \in [Acceptors \to Ballot \cup \{-1\}] \\ & \wedge accBal \in [Acceptors \to Ballot \cup \{-1\}] \\ & \wedge accSS \in [Acceptors \to Value \cup \{\text{“”}\}] \\ & \wedge accSSOri \in [Acceptors \to Ballot \cup \{-1\}] \\ & \wedge msgs \subseteq Message \end{aligned}$$

Init defines the initial states in the acceptors and empty message history

$$\begin{aligned} Init \triangleq\ & \wedge maxBal = [a \in Acceptors \mapsto -1] \\ & \wedge accBal = [a \in Acceptors \mapsto -1] \\ & \wedge accSS = [a \in Acceptors \mapsto \text{“”}] \\ & \wedge accSSOri = [a \in Acceptors \mapsto -1] \\ & \wedge msgs = \{\} \end{aligned}$$

Sending a message is simply adding the message to the *msgs* set, *i.e.*, adding message to the message history

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

$$\begin{aligned} & Phase1a(b) \triangleq \\ & \quad \wedge \quad Send([type \mapsto \text{“1a”},\ bal \mapsto b]) \\ & \quad \wedge \quad \text{UNCHANGED } \langle maxBal,\ accBal,\ accSS,\ accSSOri \rangle \end{aligned}$$

$$\begin{aligned} Phase1b(a) \triangleq\ & \exists\, m \in msgs : \\ & \wedge m.type = \text{“1a”} \\ & \wedge m.bal > maxBal[a] \\ & \wedge maxBal' = [maxBal \text{ EXCEPT } ![a] = m.bal] \\ & \wedge Send([type \mapsto \text{“1b”}, \\ & \qquad acc \mapsto a, \\ & \qquad bal \mapsto m.bal, \\ & \qquad accbal \mapsto accBal[a], \\ & \qquad ssval \mapsto accSS[a], \\ & \qquad oribal \mapsto accSSOri[a]]) \\ & \wedge \text{UNCHANGED } \langle accBal,\ accSS,\ accSSOri \rangle \end{aligned}$$

*Phase*2*a*, handling $|Q1|$ *PROMISE* messages

$Phase2a(b,\, v) \triangleq$
$\land \quad \neg\exists\, m \in msgs \quad : \land m.type = \text{“2a”}$
$\qquad\qquad\qquad\qquad\quad \land m.bal = b$
there is no '2a' messages (ACCEPT) with ballot b yet

$\land \quad \exists\, Q \in Quorum1 :$
$\quad \text{LET } Q1b \triangleq \{m \in msgs : \land m.type = \text{“1b”}$
$\qquad\qquad\qquad\qquad\qquad\quad \land m.acc \in Q$
$\qquad\qquad\qquad\qquad\qquad\quad \land m.bal = b\}$
PROMISE()es with ballot b

$\qquad Q1bss \triangleq \{m \in Q1b : m.accbal \neq -1\}$
previously accepted secret-shares

$\qquad Q1bmax \triangleq \{m \in Q1bss :$
$\qquad\qquad\qquad \forall\, mm \in Q1bss :$
$\qquad\qquad\qquad\quad m.accbal \geq mm.accbal\}$
Set of shares with the highest ballot

number of shares that have the same original ballot (*ID*) as the share with the highest accepted ballot number

$\qquad numss \triangleq \text{IF } Q1bmax = \{\} \text{ THEN } 0$
$\qquad\qquad\qquad\quad \text{ELSE } Cardinality($
$\qquad\qquad\qquad\qquad \{m \in Q1bss :$
$\qquad\qquad\qquad\qquad\quad m.oribal = (\text{CHOOSE } x \in Q1bmax : \text{TRUE}).oribal\})$

$\quad \text{IN} \quad \land \forall\, a \in Q : \exists\, m \in Q1b : m.acc = a$

there is a share with the highest accepted ballot and p can reconstruct it, p need to repropose it with the same original ballot

$\qquad \land \lor \land numss \geq T$
$\qquad\qquad\quad \land \exists\, share \in Q1bmax :$
$\qquad\qquad\qquad\quad \land share.ssval = v$
$\qquad\qquad\qquad\quad \land Send([type \mapsto \text{“2a”},$
$\qquad\qquad\qquad\qquad\qquad\quad bal \mapsto b,$
$\qquad\qquad\qquad\qquad\qquad\quad ssval \mapsto share.ssval,$
$\qquad\qquad\qquad\qquad\qquad\quad oribal \mapsto share.oribal])$

there is no shares accepted yet or not enough shares ($< T$) available to reconstruct value with the highest ballot number

$\qquad\qquad \lor \land numss < T$
$\qquad\qquad\quad \land Send([type \mapsto \text{“2a”},$
$\qquad\qquad\qquad\qquad\quad bal \mapsto b,$
$\qquad\qquad\qquad\qquad\quad ssval \mapsto v,$
$\qquad\qquad\qquad\qquad\quad oribal \mapsto b])$

$\land \quad \text{UNCHANGED } \langle maxBal,\, accBal,\, accSS,\, accSSOri\rangle$

*Phase*2*b*, handling PROPOSE messages

$Phase2b(a) \triangleq \land \exists\, m \in msgs : \land m.type = \text{“2a”}$
$\qquad\qquad\qquad\qquad\qquad\quad \land m.bal \geq maxBal[a]$
$\qquad\qquad\qquad\qquad\qquad\quad \land maxBal' = [maxBal \text{ EXCEPT } ![a] = m.bal]$
$\qquad\qquad\qquad\qquad\qquad\quad \land accBal' \;= [accBal \text{ EXCEPT } ![a] \;= m.bal]$
$\qquad\qquad\qquad\qquad\qquad\quad \land accSS' = [accSS \text{ EXCEPT } ![a] = m.ssval]$
$\qquad\qquad\qquad\qquad\qquad\quad \land accSSOri' = [accSSOri \text{ EXCEPT } ![a] = m.oribal]$
$\qquad\qquad\qquad\qquad\qquad\quad \land Send([type \mapsto \text{“2b”},$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad acc \mapsto a,$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad bal \mapsto m.bal,$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad ssval \mapsto m.ssval,$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad oribal \mapsto m.oribal])$

$$Next \triangleq \lor \exists\, b \in Ballot : \lor Phase1a(b) \lor \exists\, v \in Value : Phase2a(b,\, v) \lor \exists\, a \in Acceptors : Phase1b(a) \lor Phase2b(a)$$

Definition when a secret-shared value v with original ballot bo is chosen. That is when a $Q2$ of acceptors already accepted v with the same ballot b, it is indicated by the sending of $ACCEPT(b)$.

$$Chosen(v,\, bo,\, b) \triangleq \exists\, Q \in Quorum2 : \forall\, a \in Q : \exists\, m \in msgs : \land\, m.type = \text{“2b”} \land\, m.acc = a \land\, m.bal = b \land\, m.ssval = v \land\, m.oribal = bo$$

If a secret value v with original ballot ob is chosen with ballot b there should be no “2a” messages (PROPOSE) with larger or equal ballot number proposing different (original ballot, value) pair other than (bo,v)

$$Correctness \triangleq \forall\, v \in Value : \forall\, ob \in Ballot : \forall\, b \in Ballot : Chosen(v,\, ob,\, b) \Rightarrow \neg\exists\, m \in msgs : \land\, m.type = \text{“2a”} \land\, m.bal \geq b \land\, m.ssval \neq v \land\, m.oribal \neq ob$$

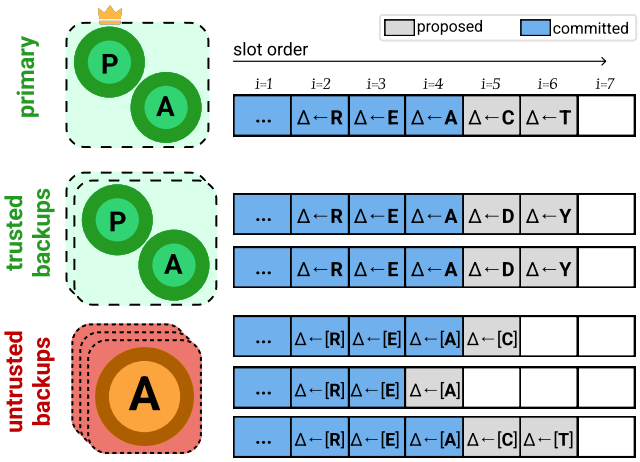


Figure 24: Primary and backups placement in PBSSM

G PBSSM WITH OPAXOS

In this section we describe how primary-backup secret-shared state machine (PBSSM) extends the idea of OPaxos to provide consistent primary-backup replication.

G.1 System Model

PBSSM, as in any primary-backup approach, has nodes that act as *primary* or *backup*. Both the primary and backup respectively keep a log containing slots of state diffs in clear and secret-shared form. For a single slot, the terms *primary* and *backup* are respectively synonymous with the terms *proposer* and *acceptor* as used previously in the description of (single-slot) OPaxos. The primary always resides in a trusted server while backups can reside in either trusted or untrusted servers. The untrusted servers can only know the secret-shared state and state diffs. The placement of primary and backup in PBSSM is shown in Figure 24. Under normal operation, a primary executes incoming requests from clients and captures the state diffs, then the primary uses OPaxos to reach agreement on the order of the state diffs with backups before returning a response to the client.

A primary has a unique ballot number and maintains its leadership using periodic heartbeats. When a trusted backup gets no heartbeat for a long enough time, it times out and starts a leader election process by running a similar process as Phase-1 of OPaxos. A backup that starts a leader election is called as a *primary candidate* and acts as the OPaxos' proposer with its new unique ballot number. The primary candidate becomes a primary once it receives |Q1| promises and completes a synchronization process.

Despite the possibility of primary changes and nondeterministic request execution, PBSSM must ensure the *primary integrity* property, i.e., the state diff corresponding to the $(i+1)$'st request must be computed by executing request $(i+1)$ upon the state obtained by the cumulative execution of every request until i by some primary. As an example, consider the replicated system in Figure 24 that runs an application that appends letters to a string so as to form a valid English sentence. A primary that recovers log suffixes in Figure 24 cannot subsequently propose state diff 'C' and 'Y' respectively in the slots $i = 5$ and $i = 6$ as state diff 'Y' is generated after applying state diff 'D', not 'C'. Proposing suffix ending with "CY" would make a sentence prefix of "REACY" that is not a valid English word (unlike "READ" or "REACT"). A similar primary integrity property and a detailed example of how it can be violated in MultiPaxos is shown in prior work on a primary-backup protocol: ZAB [27].

Variable Name	Description
Primary Node	
`bcur`	The current ballot number consists of a pair of counter and node ID
`log[]`	A 1-indexed log in primary nodes containing a sequence of state diffs, each slot `log[i]` consists of - `val` : the state diff in slot index i - `bacc` : the ballot number in which this node accepts the state diff - `bori` : the ballot number in which the state diff is first proposed - `committed` : the flag indicating whether the state diff is committed or not - `num_acc` : number of nodes accepted `val` - `resp` : the stored response for client Assignment to a slot in the pseudocode uses a tuple with elements described above, in the written order (e.g., `log[i]` $\leftarrow \{v, b, b', false\}$).
`next_slot`	The index of the first empty slot in the log into which this primary can propose a state diff value. It is initialized with 1.
Backup Node	
`bmax`	The highest ballot number this node has seen so far.
`sslog[]`	A log containing a sequence of secret-shared state diffs, corresponds to the `log` variable in primary.
`promises{}`	Set of log suffixes from promises received during a leader election process.
All Node	
`cur_role`	A constant of either 'primary', 'backup', or 'primary_candidate' indicating the current role of this node.
`cur_primary`	ID of the current primary believed by this node.
`commit_head`	The index of the last contiguously committed slot in the `log` (or `sslog`). It is initialized with 0.

Table 4: Variables in PBSSM primaries and backups.

The next sections cover the PBSSM protocol under normal run and leader election phases. We explain both phases, accompanied by the maintained variables in Table-4, and emphasize important constraints to preserve the primary-integrity property.

G.2 Normal Run

In the normal run as shown with the pseudocode in Figure 25, a primary handles all client requests and records the state

diffs after executing each request (line 1). The primary keeps the response (resp) for the client request and then initiates consensus over the order of the secret-shared state diffs in the backups. The primary keeps the next_slot variable to identify which slot the proposer need to propose the obtained state diff. A primary has its current ballot (bcur) that it sends along with any of its **PROPOSAL** messages (line 6a); the ballot identifies the primary (or *epoch* in ZAB terminology). The backup uses the primary's ballot to ignore all proposals from previous primaries that have lower ballot (line 7), just like an acceptor in OPaxos. The normal run resembles Phase-2 of OPaxos with two important differences: in-order acceptances by backups and in-order commits issued by primaries.

The first property, *in-order acceptances* consists of two constraints on an acceptor's acceptance actions: (i) acceptance in slot order (ASO), and (ii) acceptance from monotonically increasing primaries (AMIP). ASO means that the backups need to process all incoming secret-shared state diff proposals in slot order, i.e., for a backup to accept any state diff in slot index i, the backup needs to have previously accepted a proposal for slot $(i-1)$, so it must wait if it receives proposals not in slot order (possibly from different primaries). Note that this enforcement does not exist in Multi-Paxos where a proposal for say slot $(i+99)$ can be accepted even when some or all of the slots $i..(i+98)$ have no accepted proposal yet. ASO in PBSSM prevents gaps in any backup's slot-ordered log. AMIP means that the backups can only accept proposals from primaries that monotonically increase with the slot number. Thus, when a backup receives the first proposal for slot i from a new primary, the backup discards acceptances if any for slots greater than i for proposals issued by lower-ballot primaries (line 8). Failing to discard acceptances for subsequent slots not respecting MIP can result in violation of the primary integrity property.

Second, the primary needs to commit the proposals also in the correct slot order. The primary keeps the commit_head variable, which indicates the last slot in which this primary knows a committed state diff, and also keeps the number of acceptances for each slot (line 12a). When a primary with ballot b receives enough **ACCEPT(**b,i**)** messages for slot i, the primary does not directly commit the state diff in slot i as it needs to first ensure that the state diff in slot $(i-1)$ is committed (line 13a). When sending the **COMMIT** message, the primary could send a clear, not secret-shared, state diff to the trusted backups, so when one of those backup later becomes a primary, it possess the last committed state.

During a normal run, a primary maintains two versions of the underlying state: the committed state and proposed state. The committed state is obtained by applying all the prefix of committed state diffs, while the proposed state is obtained by applying the prefix of proposed state diffs. The primary need to keep two running state as the proposed state can still be changed by another primary with a higher ballot number. Only keeping a single running state would necessitate an undo operation to rollback the proposed state diffs that have been applied but did not eventually get committed. An equivalent alternative to maintaining the committed state that avoids undos is to maintain periodic checkpoints and a log of committed state diffs for subsequent slots.

Normal Run in PBSSM

Primary P on receiving a request req from client:

1: diff, resp ← Execute(req) // get state diff & response
2: SS ← {[s_1], [s_2], ..., [s_n]} ← **SecretShare(**diff**)**
3: i ← next_slot
4: next_slot ← next_slot + 1
5: Accept the state diff SS:
  a. log[i] ← {diff, bcur, bcur, *false*, 1, resp}
6: For each [s_j] in SS:
  a. Send **PROPOSE(**bcur,⟨i,bcur,[s_j]⟩**)** to the j-th server.

Backup B when receiving a **PROPOSE(**b,⟨i,bo,[s_j]⟩**)** message from Primary P:

7: If $b <$ bmax, return. // ignore the message
8: If $b >$ bmax, update the current primary:
  a. bmax ← b, cur_primary ← P
  b. If $b \neq$ bcur: cur_role ← 'backup'
  c. RemoveSuffix(sslog, $i+1$)
9: If $i > 1$ and sslog[$i-1$] = null: // in-order accept
  a. Wait the proposal for sslog[$i-1$] // blocking
10: Accept the secret-shared state diff [s_j]:
  a. sslog[i] ← {[s_j], bmax, bo, *false*}
  b. Send **ACCEPT(**b,i**)** to P

Primary P on receiving an **ACCEPT(**b,j**)** message:

11: If $b \neq$ bcur, return. // ignore the message
12: Increase the acceptance for log[j]:
  a. log[j].num_acc ← log[j].num_acc + 1
13: For i ← (commit_head + 1) to i ← (next_slot − 1):
  a. If log[i].num_acc ≥ |Q2| and
    (i = 1 or log[$i-1$].committed), // in-order commit
    mark log[i] as committed:
    1. log[i].committed ← *true*
    2. Send response log[i].resp to client // resp from line 1
    3. commit_head ← commit_head + 1
    4. Broadcast commit for slot log[i]:
      i. For each k-th server, send **COMMIT(**b,⟨i,bo,[s_k]⟩**)**
        // secret-shared state diff [s_k] is from line 2.
        // Alternatively, send a clear, non secret-shared, state diff
        // for trusted backups.

Figure 25: Normal run in PBSSM when a primary is proposing secret-shared state diffs after local execution

Start of a Leader Election in PBSSM

Backup B on hearing no heartbeat from any primary (timeout):

L.1: `cur_role` ← 'primary_candidate'
L.2: `bcur` ← `bcur` + 1 // increase the counter
L.3: `promises` ← {} // set as an empty set
L.4: `start_idx` ← `commit_head` + 1
L.5: Reconstruct state diff in `log[]` up to slot index `commit_head`
L.6: Broadcast **PREPARE(**`bcur`,`start_idx`**)** to all servers

Any server when receiving a **PREPARE(**b,`start_idx`**)** message from a primary candidate C:

L.7: If $b \leq$ `bmax`, return. // ignore the message
// promise to ignore lower ballot by updating `bmax`
L.8: `bmax` ← b
L.9: SSLog← **Suffix(**`sslog`, `start_idx`**)**
L.10: Send **PROMISE(**b,SSLog**)** to the primary candidate C.

Figure 26: Start of a leader election in PBSSM when a trusted backup is trying to be a new primary

G.3 Leader Election

When a primary is suspected as having failed, a trusted backup node can become a *primary candidate* and start the leader election process by broadcasting a **PREPARE** message that contains its current ballot number chosen so as to be higher than the previous known primary. The prepare message also contains the starting slot index from which the candidate wants to start its leadership (line L6). The start of this leader election process is shown in Figure 26.

Any node that receives the candidate's prepare message responds with a **PROMISE** message if the candidate's ballot number is higher than any ballot number that node has seen before (in either a PREPARE or ACCEPT message). This promise resembles Phase-1 of single decree OPaxos, however the promise in PBSSM applies to proposals for all slots in the log, not just a single slot. While sending the promise message, the responding node also includes the log suffix starting from the asked start index (line L9). Each slot in the log suffix contains four fields: the accepted (secret-shared) state diff (`val`), the accepted ballot (`bacc`), the original ballot (`bori`), and the `committed` flag, which later will be used by the primary candidate for committing recovered proposals when the candidate successfully becomes a primary with its current ballot. The candidate stores the log suffixes received in the **PROMISE** messages in the set `promises`.

Once a candidate receives a Q1 quorum of promises, the newly elected primary run a synchronization process that recovers state diffs from the previous primaries. This synchronization at the end of a leader election process is shown in Figure 27. For each slot from the start index onwards, the candidate attempts to reconstruct lower-ballot state diff

End of a Leader Election in PBSSM

Backup B, who is a 'primary_candidate' with `bcur` = b, upon receiving **PROMISE(**b,SSLog**)** messages from |Q1| other servers:

Let `promises` be the set of log suffixes SSLog from |Q1| servers.
Let L be an indexed log prefix with three components: value (`val`), original-ballot (`bori`), and `commit` flag.

L.11: L ← `[]` // initialize an empty log prefix
L.12: With `promises`, reconstruct state diffs from previous primaries,
Let `start_idx` be the first slot index among all suffixes in `promises`.
Let `max_idx` be the last slot index among all suffixes in `promises`.

For i ← (`start_idx`) to i ← (`max_idx`):
Let V_i be all slots with index i in `promises`.
Let s be a slot with the highest accepted-ballot in V_i
Let bori be the original-ballot of s.
Let S be a set of slots in V_i whose original-ballot is bori.
Let `shares` be a set of all secret-shared state diffs in S.
Let `commit` be true iff there is a slot in S with `committed` flag set as *true*.

a. If commit:
1. `diff` ← Reconstruct(`shares`)
2. $L[i]$ ← {`diff`, bori, *true*}
3. `next_slot` ← $i + 1$
4. `commit_head` ← i // advance the commit index

b. Else, not commit:
1. If reconstructible and MIP,
|`shares`| ≥ t and s.`bacc` ≥ $L[i-1]$.`bacc` :
// do not forget to handle if ($i - 1$) < `start_idx`
a. `diff` ← Reconstruct(`shares`)
b. $L[i]$ ← {`diff`, bori, *false*}
c. `next_slot` ← $i + 1$
2. Else, not reconstructible or not MIP:
Stop the For loop in line L.12.

L.13: `cur_role` ← 'primary'
L.14: Secret-share state diffs in the recovered log prefix L,
{[L_1], [L_2], ..., [L_n]} ← **SecretShare(**L**)**
L.15: Send **BULK-PROPOSE(**`bcur`,[L_j]**)** for all the j-th server.

Any server when receiving a **BULK-PROPOSE(**b,[L]**)** message from a newly elected primary P:

L.16: If $b <$ `bmax`, return. // ignore the message
L.17: If $b \neq$ `bcur`: `cur_role` ← 'backup'
L.18: `bmax` ← b, `cur_primary` ← P
L.19: Atomically accept all prefix slots [$i..j$] in L:
a. `sslog`[$i..j$] ← {L_{ss}[$i..j$].`val`, b, L_{ss}[$i..j$].`bori`, L_{ss}[$i..j$].`commit`}
b. RemoveSuffix(`sslog`, $j + 1$)
c. Send bulk acceptance **ACCEPT(**b,[$i..j$]**)** to P.

Figure 27: End of leader election when a trusted backup becomes a primary and synchronize the log prefix

proposals by using the same procedure as OPaxos while additionally ensuring that reconstructed proposals for increasing

slots have monotonically increasing primaries (MIP) that proposed them, and stopping at the first slot that violates MIP. A committed state diff is always reconstructible as there are guaranteed to be least t shares in the |Q2| promises. For non-committed state diffs, similar to OPaxos, the new primary uses the original-ballot to identify secret-shares of the same state diff and to decide whether enough shares are available for reconstruction (line L12.b.1). The recovery process iterates from the first slot in V and stops at the earliest *irrecoverable* slot in V, i.e., a slot whose state diff is not reconstructible or violates MIP (line L12.b.2). The new primary uses the recovered suffix as the prefix of its leadership by broadcasting the **BULK-PROPOSE** message to all nodes. The new primary is allowed to propose new state diffs starting from the first irrecoverable slot.

Nodes that receive the **BULK-PROPOSE(**$b, [L]$**)** message accepts the bulk proposal if the ballot b in the message is higher than any primary's ballot (`bmax`) this node has seen so far (line L16). The acceptance of the bulk proposal must be *atomic*, i.e., a node must either accept all the slots in the prefix $[L]$ or none at all (line L19). Failure to handle the bulk proposal atomically can violate primary integrity.

G.4 Comparison with ZAB

Overall, the PBSSM protocol is substantively similar to ZAB [27] but with some high-level and technical differences. At a high level, PBSSM is structured as a multi-slot consensus protocol employing OPaxos per slot, which closely resembles how Multi-Paxos generalizes single-slot Paxos (also known as the Synod protocol), but with some additional ordering requirements so as to ensure primary integrity. In comparison, Zab is structured as a single integrated protocol that appears to be significantly different from MultiPaxos.

At a high-level, Zab has three phases: discovery, synchronization, and broadcast, whereas PBSSM like Paxos has two: prepare and propose. In the discovery phase, Zab has a CEPOCH message that has no analogue in PBSSM and does not affect safety, only liveness, even in Zab. Zab's NEWPOCH message is analogous to PBSSM's PREPARE message and Zab's PROPOSE message is analogous to PBSSM's PROPOSE.

In contrast to Zab that has the synchronization phase where a primary must wait to commit all proposals recovered from lower primaries via a three-step propose-accept-commit sequence, PBSSM simply issues a **BULK-PROPOSE** that looks like any other proposal in its own epoch with the only difference that acceptors must accept the bulk proposal atomically. However, unlike Zab, PBSSM does not require the primary to wait for the bulk proposal to be committed and can proceed directly to issuing subsequent proposals in its epoch in a pipelined manner.

The above difference does not impact either agreement safety or primary integrity as we show in our formal proof in the following section. However, it makes a difference in the strength of the primary integrity property that Zab ensures compared to the primary integrity property that PBSSM needs and ensures. Zab's primary integrity property ensures that a primary can not issue any proposal in its epoch until it has delivered (i.e., applied state diffs corresponding to) all committed proposals from lower primary epochs. PBSSM's primary integrity property is slightly weaker and only ensures that in the committed sequence of state diffs, the committed state diff for any slot is computed by some primary after it has cumulatively applied the state diffs of all slots until the immediately preceding one (H.2). We think that Zab and its main application, Zookeeper, also do not need the stronger primary integrity property and can make do with PBSSM's slightly weaker primary integrity property.

H PBSSM CORRECTNESS PROOF

In this section, we proof that PBSSM maintains primary integrity that guarantee consistent state diffs despite primary changes (refer to Theorem H.2 for the precise definition). We are using the following definitions in the proof.

(1) A *proposal* is a tuple $\{s, v, \texttt{bacc}, \texttt{bori}\}$ consisting respectively of the slot number, state diff value, accepted ballot, and original ballot.
(2) A *proposer* is identified by a ballot and ballots are strictly ordered, so a *lower* or *higher* proposer respectively refers to one with a lower or higher ballot.
(3) The proposer of a proposal p refers to $p.\texttt{bacc}$.
(4) The original proposer of a proposal p refers to $p.\texttt{bori}$.
(5) A *prepare* is a tuple $\{s, \texttt{bacc}\}$ where `bacc` is the ballot identifying the proposer issuing the prepare and s is its tentative start slot (that may subsequently advance upon receipt of promise messages).
(6) A slot is said to be committed by a proposer P if P is the lowest proposer that received a Q2 quorum of acceptances for its proposal for that slot. Note that *decided(*v, k*)* as defined in OPaxos (Definition A.1) does not necessarily preclude *decided(*v, j*)* for $j < k$ from being true.

We begin our proof by describing five invariants enforced by the PBSSM protocol as can be observed in the pseudocode.

Invariant H.1 (Acceptor-Promise)**.** If an acceptor accepts a prepare from a proposer, the acceptor can not accept any proposal from a lower proposer (PBSSM protocol line 7).

Note that there is no mention of slot numbers in Invariant H.1, so the constraint is stricter than the corresponding constraint for Multi-Paxos that is for a single slot.

Invariant H.2 (Acceptor Slot Order / ASO)**.**

(1) An acceptor can not accept any proposal (from any proposer) for slot $(i+1)$ unless it has previously accepted a proposal for slot i.
(2) If a proposer issues proposals for both slots i and $(i+1)$ in its epoch, an acceptor accepts a proposal for slot $(i+1)$ from that proposer only if the acceptor has previously accepted proposal for slot i.

Invariant H.2 is enforced via the PBSSM protocol in line 9.

Invariant H.3 (Acceptor MIP / AMIP)**.**

(1) If an acceptor accepts a proposal $\{s, v, \texttt{bacc}, \texttt{bori}\}$, the acceptor can not subsequently accept any proposal from a proposer less than `bacc` for any slot (PBSSM protocol line 9).
(2) If an acceptor accepts a proposal for slot i, the acceptor must discard all previously accepted proposals for slots greater than i from lower proposers (PBSSM protocol line 8b).

Note that Invariant H.3.2 is unnecessary to state in Synod (single-decree) Paxos that only has a single slot, so an acceptor accepting a proposal implicitly overwrites values proposed by lower proposers, however PBSSM extends the overwriting to all higher slots. It is okay (but not necessary) to not discard an accepted slot provided it maintains primary integrity (defined in Theorem H.2), but whether or not it is maintained in general is not easy to know without maintaining information about the previous slot's original proposer.

Invariant H.4 (Proposer Ascendancy)**.** For any proposal p, the ballot of p is larger than or equal to the original-ballot of p, i.e., $p.\texttt{bacc} \geq p.\texttt{bori}$ (PBSSM protocol lines 6a and L16).

Invariant H.5 (Bulk Recovery)**.** A proposer proposes all proposals for the recovered prefix of proposals from lower proposers in bulk and an acceptor accepts them atomically, all or not at all (PBSSM protocol lines L16 and L19).

Theorem H.1 (Agreement Safety)**.** At most a single value is decided for each slot.

Proof. Informally, agreement safety of PBSSM follows from agreement safety of Paxos because it preserves the invariants of Paxos essential for safety for each slot:

(1) An acceptor receiving a prepare promises to not accept a proposal from a lower proposer (for every slot);
(2) A proposer picks the value for a slot if any recovered from its Q1 quorum responses as the value accepted with the highest (lower) ballot by any acceptor in that quorum;
(3) A proposer proposes at most one value for a slot.

There is an important difference from Synod Paxos that needs further justification: an acceptor in PBSSM can "forget" previously accepted proposals in order to enforce Invariant H.3.2 (Acceptor-MIP.2). Acceptor amnesia in Synod Paxos can compromise agreement safety. However, it does not compromise agreement safety in PBSSM because a proposer can be thought of acquiring proposing rights for all possible slots in parallel, and the protocol's Bulk Recovery (Invariant H.5) ensures that no decided value is discarded.

Formally, agreement safety for each slot follows because OPaxos' Proposable Invariant A.1 and Lemma A.1 hold for each slot independently. It is straightforward to verify that discarded proposals by an acceptor in order to maintain Acceptor MIP do not affect Invariant A.1. ∎

Lemma H.1 (Strictly Increasing Commit Time or SICT)**.** Slot $(i+1)$ can not get committed before slot i is committed.

Proof. Each acceptor accepts a value for slot $(i+1)$ only if the acceptor has already accepted some value for slot i (ASO.1 in Invariant H.2.1).

Case 1: The values for both slots i and $(i+1)$ are committed by the same proposer, say P. In order for P to commit values for both slots i and $(i+1)$, the proposer P must have previously proposed values for those slots. Furthermore, P must have proposed a value for i before $(i+1)$. Any acceptor that accepts P's proposed value for $(i+1)$ must have previously accepted P's proposed value for i (ASO.2 in Invariant H.2.2), thus slot i must have been committed before $(i+1)$.

Case 2: The values for i and $(i+1)$ are committed by different proposers, say respectively by P and Q. Any acceptor that accepted Q's proposal for slot $(i+1)$ must have previously accepted some proposal for slot i (ASO.1 in Invariant H.2.1).

Case 2.1: If Q proposed a value for slot i, proposer Q must have done so before proposing any value for slot $(i+1)$, and any acceptor in the Q2 quorum accepting Q's proposal for slot $(i+1)$ must have previously accepted Q's proposal for i (ASO.2 in Invariant H.2.2). Given that slot i was committed by a different proposer $P \neq Q$, let us consider two further sub-cases: $P > Q$ and $P < Q$.

Case 2.1.1, $P > Q$: This case is not possible because if $(i+1)$ is committed by Q, it must have also received a Q2 quorum for i, violating this case's premise that P is the lowest proposer that received a Q2 quorum for i.

Case 2.1.2, $P < Q$: In this case, Q must have retrieved P's committed value (because of the OPaxos' Proposable Lemma A.1) and re-proposed the value, preserving the claim.

Case 2.2: Q started from $(i+1)$. If so, proposer Q gathered a value already committed for slot i even before Q proposed any value (recovered or originated) at all, preserving the claim.

■

Theorem H.2 (Primary Integrity)**.** The value committed for slot $(i+1)$ is generated by the cumulative output of applying all commits up to slot i.

Proof. This claim is trivially true for all slots originated by the same proposer assuming synchronization-commit, i.e., the proposer commits lower epoch proposals before originating any of its own proposals. But we can prove the claim via induction without that synchronization-commit assumption as follows.

Inductive assumption: Suppose the claimed property holds until some slot i. This assumption is well defined as slot i is committed strictly before slot $(i+1)$ (via SICT in Lemma H.1).

Inductive step: If slot $(i+1)$'s commit is originated by the same proposer as slot i, i.e., $\text{bori}_{1+1} = \text{bacc}_i$, the claim is preserved. If $(i+1)$'s commit is originated by a different proposer $\text{bori}_{i+1} \neq \text{bacc}_i$, we have the following cases:

Case 1, bori_{i+1} *started its epoch at slot* $(i+1)$*:* If so, $\text{bori}_i + 1$ found slot i to be committed already by $(\text{bacc}_i, \text{bori}_i)$, so it will originate its first proposal (with no recovered proposals) for slot $(i+1)$ as the state diff obtained by executing the corresponding request on the cumulatively committed state up to slot i, which preserves the primary integrity invariant for this inductive step.

Case 2: bori_{i+1} *started its epoch at slot* i *or lower:* If so, any acceptor that accepted bori_{i+1}'s proposal for $(i+1)$ would have also accepted its proposal for i (ASO.2 in Invariant H.2.2), and any acceptor that subsequently overwrote slot i with a different proposal would have also overwritten or discarded slot $(i+1)$ (Acceptor-MIP.2 and Bulk Recovery), so any proposer bacc_{i+1} would either recover bori_{i+1}'s proposals for both i and $(i+1)$ or neither. By case premise, bacc_i commits a proposal with original proposal bori_i for slot i. Thus the only way for bori_{i+1} proposal for slot $(i+1)$ to not be overwritten or discarded and to be committed instead is if $\text{bori}_{i+1} = \text{bori}_i$, i.e., primary integrity is preserved until slot $(i+1)$ completing the inductive step proof. ■

The above two theorems respectively on agreement safety and primary integrity complete the essential formal safety guarantees ensured by PBSSM. By the nature of its design, PBSSM additionally happens to ensure that the committed slot sequence of proposals reflects monotonically increasing primaries as shown below.

Lemma H.2 (MIP)**.** Committed slots are in Monotonically Increasing Proposer (MIP) order.

Proof. Suppose all slots up to i are committed and in MIP order, an assumption that is well defined by SICT proven just above (Lemma H.1). Let P be the proposer that committed slot i. If P also commits $(i+1)$, MIP is preserved. Let $Q \neq P$ be some other proposer that committed $(i+1)$.

Case 1, $Q < P$*:* By definition, P is the lowest proposer that committed a value for slot i. If Q committed slot $(i+1)$, Q proposed a value for slot $(i+1)$ that was accepted by a Q2 quorum.

Case 1.1, Q proposed a value for slot i. This case is not possible because then that value would have also been accepted by the same Q2 quorum as slot $(i+1)$, which violates the premise that i was committed by P.

Case 1.2, Slot $(i+1)$ *is the first slot for which Q proposed a value.* That means that Q must have seen slot i as committed by P. By protocol line L7, Q will therefore not propose any values because it will not get a Q1 quorum.

Case 2, $Q > P$*:* MIP Lemma is preserved. ■