QuePaxa: Escaping the Tyranny of Timeouts in Consensus

Pasindu Tennage*
EPFL

Cristina Băsescu*
EPFL

Lefteris Kokoris-Kogias
ISTA Austria

Ewa Syta
Trinity College

Philipp Jovanovic
UCL

Vero Estrada-Galiñanes
EPFL

Bryan Ford
EPFL

Abstract

Leader-based consensus algorithms are fast and efficient under normal conditions, but lack robustness to adverse conditions due to their reliance on timeouts for liveness. We present QuePaxa, the first protocol offering state-of-the-art normal-case efficiency without depending on timeouts. QuePaxa uses a novel randomized asynchronous consensus core to tolerate adverse conditions such as denial-of-service (DoS) attacks, while a one-round-trip fast path preserves the normal-case efficiency of Multi-Paxos or Raft. By allowing simultaneous proposers without destructive interference, and using short hedging delays instead of conservative timeouts to limit redundant effort, QuePaxa permits rapid recovery after leader failure without risking costly view changes due to false timeouts. By treating leader choice and hedging delay as a multi-armed-bandit optimization, QuePaxa achieves responsiveness to prevalent conditions, and can choose the best leader even if the current one has not failed. Experiments with a prototype confirm that QuePaxa achieves normal-case LAN and WAN performance of 584k and 250k cmd/sec in throughput, respectively, comparable to Multi-Paxos. Under conditions such as DoS attacks, misconfigurations, or slow leaders that severely impact existing protocols, we find that QuePaxa remains live with median latency under 380ms in WAN experiments.

1 Introduction

In widely-deployed state machine replication (SMR) protocols [64, 75], a fault-tolerant group of distributed replicas use consensus [16, 42] to agree on a strongly-ordered series of states. Mainstream SMR protocols, however, depend heavily on timeouts in their operation, leading to three related issues we call the *tyranny of timeouts*. First, because most practical consensus algorithms are leader-driven, they rely on partial-synchrony assumptions and timeout-triggered view changes for availability, and may lose liveness under adverse network conditions. Second, because simultaneous leaders interfere destructively and view changes incur high costs, timeouts must be chosen conservatively large to avoid false triggers and maintain performance. Third, timeouts incur the administrative cost of careful manual configuration, and mistakes can lead to poor performance or complete outages. Can practical SMR systems escape the tyranny of timeouts?

We introduce QuePaxa, a novel approach to consensus and SMR that confronts these issues. QuePaxa seeks to ensure robust availability and performance under diverse conditions such as transient slowdowns, targeted denial-of-service attacks, or misconfigurations. In brief, QuePaxa achieves robustness by (1) leveraging randomized asynchronous consensus to guarantee liveness under worst-case conditions; (2) relying on hedging [23, 71] instead of timeouts to gain efficiency comparable to leader-based protocols under normal conditions; and (3) adaptively choosing leaders and hedging schedules to mitigate the costs and risks of manual configuration.

Asynchronous consensus algorithms have long held promise to avoid relying on timeouts for liveness [2, 14, 29], and to tolerate arbitrary network conditions including targeted denial-of-service attacks. Under normal conditions, however, asynchronous algorithms are usually much less efficient than partially-synchronous algorithms. QuePaxa introduces a novel asynchronous crash-stop consensus protocol that randomly prioritizes proposals to circumvent the FLP theorem [27] and guarantees commitment in a few round-trips with high probability. To avoid the traditional efficiency costs of asynchronous consensus, QuePaxa supports a fast path allowing a designated leader to commit in a single round-trip merely by adjusting its priority selection, thereby achieving normal-case efficiency comparable to partially-synchronous protocols.

Timeouts in traditional protocols must be set conservatively large to avoid false triggers, because simultaneous proposers destructively interfere with each other, preventing either from progressing, and unnecessary view changes are costly. Due to QuePaxa's asynchronous core, however, simultaneous proposers not only do not interfere destructively, but can even cooperatively help each other decide faster. In

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10.
https://doi.org/10.1145/3600006.3613150

place of timeouts and view changes, proposers in QuePaxa use *hedging* [23, 71] – a delayed-activation schedule – allowing proposers later in the schedule to exercise "enlightened procrastination" to avoid unnecessary effort (computation and bandwidth consumption) redundant with the work of earlier proposers. Hedging enables QuePaxa to achieve the same $O(n)$ complexity as conventional protocols under stable network conditions. Further, hedging delays may be set aggressively small, minimizing recovery time in case of leader failure. False triggers rarely delay QuePaxa consensus, as our experiments confirm, and they never compromise liveness even if hedging delays are badly misconfigured.

Finally, timeouts traditionally incur the administrative cost of manual tuning, and limit the system's *responsiveness* [69, 93] or dynamic adaptivity to prevailing conditions. Today's popular consensus protocols such as Raft [64] can remain "stuck" with a leader that is slow but not slow enough to trigger a view change, even when faster leaders are available. QuePaxa incorporates delay monitoring and adaptation techniques, inspired by multi-armed bandit theory [78], to adjust its leader choice and hedging schedule dynamically and ensure responsiveness to time-varying conditions.

We evaluated a prototype of QuePaxa in Go [52] against Multi-Paxos [42], Raft [64], EPaxos [55], and Rabia [66]. We tested QuePaxa on Amazon EC2 in both local-area (single-region) and wide-area (multi-region) deployments. We first confirm that QuePaxa delivers 584k cmd/sec in throughput under 5.8ms median latency, comparable to state-of-the-art protocols under normal conditions. QuePaxa also remains live, however, under adversarial network attacks that slow or halt existing protocols. QuePaxa maintains full performance even when its hedging delays are only $1/3$ the underlying round-trip time, whereas timeouts in Multi-Paxos and Raft must be at least $1.8\times$ the round-trip delay to avoid stalls due to false triggers and unnecessary view changes. Finally, we show that QuePaxa automatically identifies and converges to the fastest leader replica, delivering 1.4ms lower median latency among heterogeneous replicas in a data center setting.

This paper makes the following main contributions:

- The first practical asynchronous consensus protocol that supports single-round-trip, $O(n)$ normal-case commitment, merely by adjusting the priorities of proposals.
- The use of hedging instead of timeouts to avoid the high costs of unnecessary view changes, and to minimize recovery time after leader failure.
- Novel adaptation techniques enabling QuePaxa to optimize its leader choice and hedging schedule and maintain responsiveness to prevalent conditions.
- A working prototype and experimental analysis of QuePaxa under both normal and adversarial conditions.
- Correctness proofs (Appendix C) and Promela models verified using the SPIN model checker (Appendix D).

Section 2 summarizes background, and Section 3 presents a high-level overview of QuePaxa. Section 4 details the consensus protocol, while Section 5 covers its use in efficient SMR. Section 6 describes our prototype, Section 7 evaluates it experimentally, and Section 8 outlines related work.

2 Background and Motivation

This section outlines the current state-of-the-art in consensus and SMR and the challenges that motivate QuePaxa.

Leader-based consensus: The most widely-deployed consensus and SMR protocols elect one *leader* at a time to drive consensus [42, 64], because simultaneously-active replicas destructively interfere as we examine later in Section 3.4. If the leader fails, consensus stalls until enough replicas time out and trigger a *view change* to elect a new leader.

These protocols guarantee liveness only under *partial synchrony*: when network delays are small and stable enough for some leader to make progress between timeout-triggered view changes. Adverse network conditions such as periods of high delay or jitter, asymmetric connectivity, or misconfiguration can drastically slow or halt progress [7, 40, 47]. Further, if an adversary in the network can focus a denial-of-service (DoS) attack on the current leader, the attacker can slow or halt all progress until a timeout. If such an attacker can employ traffic analysis to detect view changes and refocus the attack on each new leader, then the attacker can in principle halt progress indefinitely while targeting only one replica at a time [80].

Asynchronous protocols: Asynchronous consensus [2, 14, 53, 63] avoids depending on timeouts, in principle promising robust availability even under arbitrary worst-case network conditions. Asynchronous protocols are typically much less efficient under the normal conditions that leader-based protocols optimize for, however, incurring $O(n^2)$ rather than $O(n)$ communication complexity per decision. Asynchronous protocols are thus rarely deployed in practice. We would ideally like a consensus protocol that could achieve the normal-case efficiency of leader-based protocols, together with the robust worst-case availability guarantees of asynchronous protocols.

Rabia [66] is a randomized crash-fault-tolerant SMR protocol that builds on Ben-Or's binary asynchronous consensus algorithm [14] as a central component. Rabia specializes in low-delay, high-capacity data-center networks, however, making assumptions and design choices that limit its usability in other environments (see Section 7.2 and Appendix E).

Conservative timeouts: Because simultaneous leaders destructively interfere and view changes are costly in leader-based schemes, timeouts must be conservatively large to limit risk of false triggers due to random delays or brief slowdowns. As our experiments later in Section 7.5 confirm, leader-based protocols drastically slow or stop entirely if timeouts are too short with respect to underlying delays. Large timeouts imply a long recovery time if the leader fails, however. We would thus prefer a protocol that could enable short recovery times without risking availability loss from false timeouts.

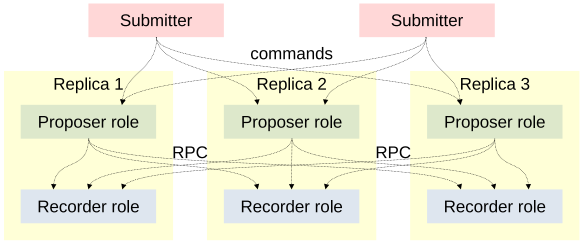


Figure 1. Architecture of QuePaxa. A configuration consists of $2f+1$ replicas tolerating f faults. Each replica plays two roles: proposers actively drive consensus, while recorders passively store and update state in response to commands.

Configuration sensitivity: Timeouts in leader-based protocols must be carefully configured in practice, requiring administrative effort and risking outages if misconfigured. Further, statically-configured timeouts prevent leader-based schemes from adapting dynamically to prevailing conditions, such as when the current leader is slow but not slow enough to trigger a timeout. We would ideally prefer a protocol requiring no manual configuration of delays, and which would adapt automatically to prevailing conditions, such as by choosing a faster leader automatically if one is available.

In brief, QuePaxa seeks to address three main challenges:

- How to achieve robust availability under worst-case conditions together with high normal-case efficiency?
- How to minimize recovery time without risking liveness?
- How to adapt dynamically to current network conditions?

3 QuePaxa Architecture Overview

This section outlines QuePaxa's architecture at a high level, covering system model, assumptions, workflow, and properties enabling QuePaxa to avoid dependence on timeouts.

3.1 QuePaxa system model

Figure 1 illustrates QuePaxa's architecture. Any number of *submitters* send *commands* describing transaction requests to a group of *replicas* collectively responsible for storing state. Submitters could be clients that generate commands directly, but in modern tiered deployments, a submitter is more commonly a front-end proxy that receives commands from actual clients elsewhere on the Internet, often gathering commands into batches for submission to the proposers.

As usual in crash-fault consensus, there are $n \geq 2f + 1$ replicas, of which at most f may fail, by going silent forever (faults are not Byzantine). We assume the set of replicas is well-known and static, but reconfiguration may be supported via standard practices [42, 64]. Submitters send commands to all replicas, so submitted commands cannot starve indefinitely regardless of which proposers commit transactions.[1]

As shown in Figure 1, each replica plays two functional roles internally: (1) a replica's *proposer* role receives commands from submitters and actively drives the process of committing these commands, and (2) a replica's *recorder* role passively maintains consensus state while responding to RPC-style requests from proposers. This active/passive division of roles is analogous to that of Disk Paxos [32].

3.2 Assumptions and threat model

QuePaxa assumes that while the n replicas are trustworthy, communication paths are not. Especially in wide-area networks (WANs), communication may be disrupted in practice by intermittent outages, high delays or jitter, and asymmetric connectivity [7, 47]. Intelligent adversaries in the network may identify consensus protocol flows (*e.g.*, via traffic analysis patterns) and seek to slow or halt progress via targeted denial-of-service (DoS), route hijacking [19], or other attacks.

Formally, QuePaxa assumes that any message sent between correct (non-faulty) nodes is eventually delivered [16], an assumption we fulfill in practice by building atop a reliable transport such as TCP [81]. We consider a *broadcast* to the n replicas to consist of n separate message transmissions in parallel: we do not assume efficient network broadcast.

QuePaxa assumes that the network adversary is content-oblivious [10]. That is, the adversary may manipulate network delays and order packets arbitrarily, but cannot see message content or replica memory. This assumption is realistic in that we can satisfy it in practice simply by encrypting pairwise communication between replicas, *e.g.*, via TLS [73].[2]

We say that the network is *partially synchronous* if there exists some maximum delay Δ, unknown to the consensus protocol, that upper bounds all message transmission delays [24]. The network is *asynchronous* if no such maximum delay exists. Partial synchrony models "normal-case" periods when network delays are reasonably stable, while asynchrony models adverse conditions such as DoS attacks. QuePaxa seeks high performance and efficiency under partial synchrony, while ensuring safety and liveness even under asynchrony.

3.3 QuePaxa workflow overview

Figure 2 illustrates QuePaxa's workflow, which employs the standard state machine replication (SMR) paradigm [75]. A series of *slots* represent successive state transitions forming a totally-ordered history. A submitter broadcasts a command (or a batch of commands) to all the proposers, who in turn propose these commands in the next free slot. The proposers then run a consensus protocol instance in each slot to agree on a unique state transition for that slot. Proposers finally report the commitment of commands to the submitter. Proposers may batch concurrent submitters' commands (or batches)

[1] Submitters must send commands to at least $f + 1$ replicas anyway, in any consensus protocol, since f replicas may fail. A standard optimization to reduce bandwidth and load is for submitters to contact only the current leader first, then other replicas only if the leader does not respond quickly.

[2] Information leakage via side channels could compromise this content-oblivious adversary assumption. Such risks may be mitigated via implementation best practices such as constant-size messages and constant-time code paths, but side channels are beyond the scope of this paper.

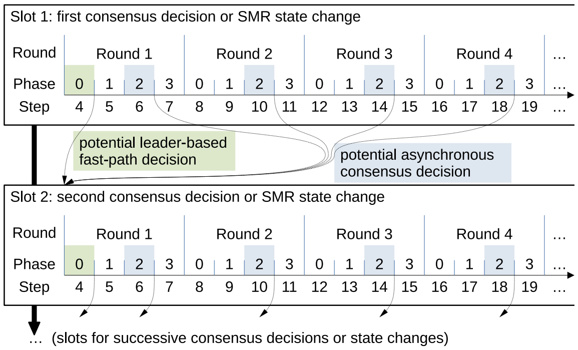

Slot 1: first consensus decision or SMR state change
Round
Round 1
Round 2
Round 3
Round 4
…
Phase
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 …
Step
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 …
potential leader-based fast-path decision
potential asynchronous consensus decision
Slot 2: second consensus decision or SMR state change
Round
Round 1
Round 2
Round 3
Round 4
…
Phase
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 …
Step
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 …
… (slots for successive consensus decisions or state changes)

Figure 2. QuePaxa workflow overview. *Slots* represent successive state changes decided by consensus. A decision takes one or more *rounds* of four *phases* each. *Steps* combine round and phase numbers into a logical clock: $step = 4 \times round + phase$.

together into one slot, or defer them to later slots. Batching is thus possible at both the submitter and proposer stages.

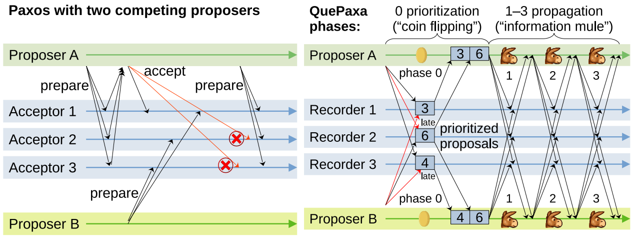
Because asynchronous consensus is deterministically unsolvable in general [27], any single decision attempt may fail and need to be retried. QuePaxa thus decides each slot in a series of attempts or *rounds* numbered from 1. Each round consists of four protocol *phases* 0–3. A *step* number or threshold clock [28] counts phases across rounds, such that $step = 4 \times round + phase$. A step represents progress in completing a threshold of communication, and assumes neither network synchrony nor synchronized clocks. Each step requires one communication round-trip between at least one proposer and a *quorum* or majority of $n - f$ recorders.

The passive recorder role is simple, merely storing a concise, constant-space summary of recent proposals aggregated via simple arithmetic (*e.g.*, integer maximum). We formulate this recorder functionality in a primitive we call an *interval summary register* or ISR, detailed later in Section 4.2.2.

A proposer may decide a slot in two ways. Under partial synchrony, a unique designated proposer or *leader* may commit in just one round-trip (round 1 phase 0). This fast path is largely equivalent to fast consensus in Multi-Paxos by an already-prepared leader [42]. If the fast path fails for any reason, including leader failure or network asynchrony, then any proposer can potentially decide the slot in phase 2 of any round. Rounds 2 and higher in each slot are leaderless and fully asynchronous, each round guaranteed to succeed independently with probability at least 1/2. Phase 3 of each round is needed only to prepare for the next round, in case the current round fails to reach consensus.

3.4 From competitive claim-staking to cooperation

Like Paxos, QuePaxa is most efficient (and its fast path most likely to succeed) when only one proposer (the leader) proposes at once. In normal-case operation, replicas in both protocols have an expectation of which proposer "should" propose (first) in a slot. The protocols differ fundamentally, however, in the strength of this expectation and in the consequences of its being violated. Figure 3 illustrates this difference.



(a) Destructive interference in Paxos consensus.

(b) Cooperative concurrency in QuePaxa consensus.

Figure 3. Destructive versus constructive concurrency between multiple proposers in consensus algorithms.

The purpose of Paxos's **prepare** phase is akin to staking a territorial claim on a majority of the proposers, and Paxos's **accept** phase essentially records a successful claim. Two competing proposers interfere destructively, however, in the worst case blocking each other's progress indefinitely. In Fig. 3a, Proposer B's **prepare** phase interrupts Proposer A's attempt to complete (via **accept**) a transaction it had successfully prepared. Proposer A then retries from the **prepare** phase with a higher ballot number, interrupting proposer B – and so on, ad infinitum. This destructive interference motivates traditional view change protocols, in which *only* the leader of a given view can propose – and *must* propose, before the other proposers' timeouts expire – otherwise consensus stalls until further view changes find a live leader. If timeouts are configured too short for the network delay, then exactly this "worst-case" scenario happens and Paxos livelocks forever.

A QuePaxa leader, however, is merely "first among equals" with a special fast-path privilege. Other replicas may also propose, with little risk or penalty apart from redundant effort. Simultaneous proposers do not destructively interfere, and can even help each other complete consensus rounds faster.

As Fig. 3b illustrates, proposers in phase 0 serve a "coin flipping" function, attaching a random priority to the first proposal each recorder sees. In phases 1–3, proposers serve an "information mule" function, propagating information about prioritized proposals among recorders. In both of these functions, it matters little whether just one proposer, or several, perform these steps at once. Consensus rounds complete regardless, each with a constant probability of deciding.

3.5 Escaping the tyranny of timeouts

Given the above operational overview, we can now summarize more precisely how QuePaxa escapes from the three "tyranny of timeouts" issues introduced in Section 1.

Liveness: QuePaxa eliminates reliance on timeouts for liveness. All phases of all rounds progress asynchronously. The first round in each slot is leader-based, allowing single-round-trip fast-path commit under normal conditions. An intelligent

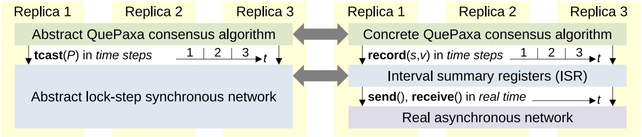

Replica 1
Replica 2
Replica 3
Abstract QuePaxa consensus algorithm
tcast(P) in time steps
1 | 2 | 3
t
Abstract lock-step synchronous network
Replica 1
Replica 2
Replica 3
Concrete QuePaxa consensus algorithm
record(s,v) in time steps
1 | 2 | 3
t
Interval summary registers (ISR)
send(), receive() in real time
t
Real asynchronous network

Figure 4. Protocol layering. Abstract QuePaxa (Algorithm 1) operates atop lock-step threshold synchronous broadcast (**tcast**), which the concrete protocol (Algorithm 4) simulates using interval summary registers or ISRs (Algorithm 2).

network adversary could potentially exploit knowledge of the leader to make this first round consistently fail to decide, but subsequent rounds are leaderless and fully asynchronous, guaranteeing at least $1/2$ success probability each round.

Hedging: By avoiding destructive interference or view changes, QuePaxa permits all proposers to participate in any round, on a *hedging* schedule [23, 71]. Proposers later in the schedule wait for longer, then propose only if they have not seen earlier proposers make progress. Under normal conditions, only the first scheduled proposer usually proposes in each round, yielding $O(n)$ communication cost as in leader-based protocols. Unlike timeouts, hedging delays may be short for fast recovery after a leader failure. Even if set so small that several proposers start before the first one completes, consensus remains live, the only costs being redundant proposer effort and lower fast-path success probability.

Auto-tuning: Because choice of leader and hedging schedule in QuePaxa are optimization parameters not critical to liveness, these choices are a form of multi-armed bandit (MAB) problem [78]. QuePaxa thus leverages simple explore/exploit processes inspired by MAB theory to explore alternatives and auto-tune consensus to exploit learned knowledge. Unlike existing protocols, QuePaxa can thus find a better leader proactively even if the current leader has not failed. QuePaxa also eliminates both the administrative burden of configuring timeouts and the main risks of misconfiguring them.

Having summarized how QuePaxa operates and avoids dependence on timeouts, we now detail its design.

4 QuePaxa Protocol Design

This section details QuePaxa's design, first in terms of a simplified abstract formulation of the core protocol for clarity, followed by a concrete instantiation of that core (see Fig. 4).

4.1 Abstract QuePaxa consensus protocol

For simplicity, we temporarily ignore the challenge of network asynchrony and address *only* the unreliability problem: *i.e.,* that anticipated messages fail to arrive due to replica failures. Figure 4 depicts the architectural layers of this abstract protocol, and their correspondence to the concrete protocol we present later in Section 4.2. We assume for now that the network provides a *threshold synchronous broadcast* or **tcast** primitive described next. We then define QuePaxa's abstract consensus protocol (Algorithm 1) based on **tcast**. In brief, **tcast** provides each replica with the messages from any majority or quorum of replicas at each time step. Each replica also identifies one message received by all live replicas. Consensus atop **tcast** provides each replica with two proposal sets that bound some set of proposals received by any replica. This cross-node subset relationship is essential for replicas to ensure safety. Each replica attaches a random local priority to its proposal, enabling the protocol to terminate with probability 1 in a small constant expected number of rounds.

Algorithm 1: Abstract QuePaxa consensus algorithm

```
Input: $v \leftarrow$ value preferred by this replica
repeat                                        // iterate through rounds
    $p \leftarrow \langle v, \textbf{random}() \rangle$                  // prioritized proposal
    $(P, \_) \leftarrow \textbf{tcast}(\{p\})$               // propagate our proposal
    $(E, P') \leftarrow \textbf{tcast}(P)$                // propagate existent sets
    $(C, U) \leftarrow \textbf{tcast}(P')$               // propagate common sets
    $v \leftarrow \textbf{best}(C).\textbf{value}$                // next candidate value
    if $\textbf{best}(E) = \textbf{best}(U)$ then               // detect consensus
        $\textbf{deliver}(v)$                           // deliver decision
```

4.1.1 Threshold synchronous broadcast (tcast)

We pretend for now that the n replicas run atop an idealized network operating in lock-step synchrony: any message delivery takes exactly one *step*. This idealized network provides a *threshold broadcast* primitive, or **tcast** [28, 29], which we find particularly well-suited to information dissemination in consensus.

At each time step, each live replica i invokes $\textbf{tcast}(P_i)$ with some set of proposals P_i that i wishes to disseminate to the other replicas. After one time step, each replica i's $\textbf{tcast}(P_i)$ call completes and returns a pair of proposal sets (R_i, B_i). The sets R_i and B_i satisfy two key properties that we define below.

The first set R_i returned by **tcast** is the set of all proposals received by replica i in this broadcast step. This R_i includes the inputs from a majority of replicas. That is, there is some set S of replicas such that $|S| > n/2$, and $\forall j \in S, P_j \subseteq R_i$.

The second set B_i returned by **tcast** is some proposal set input (*i.e., P_j* for some j) that **tcast** has successfully broadcast to *all* non-faulty replicas during this broadcast step. That is, the returned B_i is the proposal set input P_j of some replica j, not necessarily the same as i, such that for all replicas k, $P_j \subseteq R_k$. As a result, for all replicas i and j, $B_i \subseteq R_j$.

In summary, **tcast** ensures two key properties: (1) all live replicas receive a majority of replicas' inputs, and (2) at least one replica's input (returned in B) is seen by *all* live replicas.[3]

4.1.2 Building consensus atop tcast

Algorithm 1 presents QuePaxa's abstract consensus protocol core, built atop **tcast**, for a single SMR slot. Each replica conceptually runs an unlimited series of rounds in the slot, delivering a consensus

[3] These two properties could be separated into two distinct communication primitives, but we felt this combined formulation is easier to understand.

decision with some probability in each round. Different replicas may reach a decision earlier or later, in different rounds.

In each round, each replica i first associates its current preferred value v with a random numeric *priority* to form i's proposal p_i. All replicas choose these priorities independently from the same private random distribution. For simplicity, we will assume for now that priorities never tie within a round.[4]

All n replicas then disseminate their prioritized proposals in three successive **tcast** steps. The first **tcast** gives each replica i a proposal set P_i containing the proposals from any majority of replicas. The second **tcast** takes P_i as input, and gives i a proposal set P'_i that is guaranteed to be included in the *existent* sets E_j returned to all other replicas j. Finally, the third **tcast** uses this set P'_i as input, and gives replica i a *common* proposal set C_i and a *universal* proposal set U_i.

The one important goal that these protocol steps achieve is that $\forall i, j, U_i \subseteq C_j \subseteq E_i$. That is, every replica's universal set U_i is a subset of every other replica's common set C_j, which in turn is a subset of any replica's existent set E_i.

Perhaps more intuitively, a proposal p is *existent* from replica i's perspective (*i.e.,* $p \in E_i$) if i knows that p exists: that is, i knows that some replica proposed p in this round. A proposal p is *common* for i ($p \in C_i$) if i knows that *all* replicas know that p exists. A proposal p is *universal* for i ($p \in U_i$) if i knows that *all* replicas know that p is *common*.

Finally, each replica chooses $\mathbf{best}(C_i)$, the highest-priority proposal from i's common set C_i, as i's preferred value as input to the next consensus round. Each replica also checks whether its best known existent proposal $\mathbf{best}(E_i)$ is identical to its best known universal proposal $\mathbf{best}(U_i)$, and if so delivers this proposal's value as the consensus decision.

4.1.3 Consensus protocol correctness

We now briefly sketch arguments for this algorithm's correctness. Detailed correctness proofs may be found in Appendix B.

Theorem (Safety). *Abstract QuePaxa ensures the key safety properties of consensus: validity, integrity, and agreement.*
Proof sketch: If replica i sees that $\mathbf{best}(E) = \mathbf{best}(U)$ and delivers a decision in some round, then *every* replica j must choose the same proposal as its next candidate value $\mathbf{best}(C_j)$. That is, $\forall i, j, \mathbf{best}(E_i) = \mathbf{best}(C_j) = \mathbf{best}(U_i)$, because of the cross-node subset relationship $U_i \subseteq C_j \subseteq E_i$ established above and because priorities are never tied. Since each subsequent round uses only values carried over from the prior round, this decision is the only value available to be decided upon by any replica in this or any subsequent round, thus ensuring agreement. The decided value is carried over from the previous round and, by induction, originates in the first round, which employs only values proposed by replicas, thus ensuring validity. Each replica trivially ensures integrity by maintaining a local decision flag (not shown in Algorithm 1 for simplicity) to decide only once per slot.

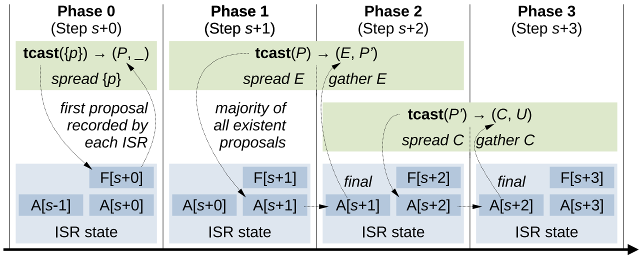


Figure 5. Correspondence between the four-phase concrete protocol in Algorithm 4 and the three **tcast** invocations in the abstract QuePaxa algorithm in Algorithm 1.

Theorem (Liveness). *Abstract QuePaxa terminates with probability 1 in less than two rounds in expectation.*
Proof sketch: Informally, each replica i is guaranteed to decide in a given round if the unique best proposal in that round appears in i's universal set U_i. In this case, that unique best proposal must also appear, and be best, in E_i and in the C_j of every other replica j. All the sets returned from the **tcast** invocations in Algorithm 1 include proposals from a majority of the replicas. Because the network-scheduling adversary is content-oblivious and hence unaware of the priorities attached to proposals (see 3.1), each replica i observes at least $1/2$ probability of the round's unique best proposal appearing in U_i and hence deciding. Each replica therefore decides in under two rounds in expectation and decides eventually with probability 1, thus ensuring the protocol's liveness.

4.2 Concrete QuePaxa consensus protocol

The concrete QuePaxa consensus protocol essentially simulates the abstract protocol described above, implementing it more realistically and efficiency in several ways. As outlined earlier in 3.3, the concrete QuePaxa protocol separates each replica's active and passive roles, handles network asynchrony by threshold logical clocking [28], transmits only constant-space integer summaries rather than proposal sets, and incorporates a Paxos-like fast path to achieve single-round consensus in favorable network conditions with a known leader.

Figure 5 shows an overview of how the three **tcast** operations in the abstract protocol above map to the four phases of the concrete protocol detailed below. Concretely implementing the first **tcast** operation in Algorithm 1 requires only one threshold clock time-step (phase 0), because this step requires only that each replica obtain proposals from some majority of replicas. Concretely implementing the second and third **tcast** operations in Algorithm 1 require two threshold clock steps each, using a *spread/gather* sequence detailed below to propagate at least one replica's **tcast** input to *all* live replicas. We can pipeline these latter two **tcast** operations, however, so

[4] We can ensure a negligible chance of a tie for best by choosing priorities with high entropy (*e.g.*, 256 bits) drawn from a strong (*e.g.*, cryptographic) random number generator. Appendix A discusses alternative approaches to choosing priorities and handling ties if high-entropy priorities are undesirable.

```
Algorithm 2: Interval summary register (ISR)
State: $S$ current logical clock step, initially 0
State: $F[s]$ first value recorded at each step, default nil
State: $A[s]$ aggregate of values in each step, default nil

record $(s, v) \to (s', f', a')$:                  // handle an invocation
    if $s > S$ then                                // advance to a higher step
        $S \leftarrow s$                           // update current step number
        $F[s] \leftarrow v$                        // record first value in this step
    if $s = S$ then                                // aggregate all values
        $A[s] \leftarrow$ aggregate$(A[s], v)$     // seen in this step
    return $(S, F[S], A[S-1])$                     // return a summary
```

as to use only three steps total (phases 1–3). The full concrete protocol thus comprises four phases total per round.

4.2.1 Separating active and passive roles Each replica plays an active *proposer* role, which drives consensus, and a passive *recorder* role, which merely records state. All communication is RPC-style, proposer-to-recorder. Proposers never interact directly with each other, and neither do recorders.

Any proposer can drive consensus, by guiding the recorders through a series of states that simulate an execution of the abstract QuePaxa protocol (Algorithm 1). As in traditional leader-based consensus protocols, it is sufficient, and most efficient, when only one proposer drives consensus in the common case. Thus, we expect most replicas' proposer roles (except the leader) to be idle much of the time normally. If multiple proposers *are* active simultaneously, however, they merely work together to drive this simulation faster (*i.e.,* at the speed of the fastest proposer at each step) rather than interfering destructively as in a Paxos-like protocol.

4.2.2 Logical clocks and interval summary registers As the concrete protocol runs atop an asynchronous network, the recorders use *threshold logical clocks* [28] to simulate the synchronous, lock-step notion of time that Algorithm 1 assumes. Each consensus round consists of four logical time steps. A *step* is a non-negative integer that has no direct correspondence to real time, but advances only when a threshold of communication has been completed in the prior step.

We distill each recorder's state and behavior into a simple abstraction we call an *interval summary register* or ISR, which may be of interest beyond QuePaxa. Intuitively, an ISR accepts a succession of values each associated a logical time step, and in response to each invocation, returns a concise summary of all the values that were presented to the ISR so far in the current and immediately-prior time steps.

```
Algorithm 3: Specialized constant-space integer ISR
State: $S$ current logical clock step, initially 0
State: $F_c$ first value received in current step $S$, initially 0
State: $A_c$ maximum value seen in this step, initially 0
State: $A_p$ maximum value seen in prior step, initially 0

record $(s, v) \to (s', f', a')$:              // handle an invocation
    if $s = S$ then                            // aggregate all values
        $A_c \leftarrow \max(A_c, v)$          // seen in this step
    else if $s > S$ then                       // advance to a higher step
        if $s = S + 1$ then                    // exactly one step forward
            $A_p \leftarrow A_c$               // current aggregate now prior
        else                                   // skipping one or more step(s)
            $A_p \leftarrow 0$                 // we saw nothing in $s - 1$
        $S \leftarrow s$                       // advance to the new higher step
        $F_c \leftarrow v$                     // record first proposal this step
        $A_c \leftarrow v$                     // initial aggregate for this step
    return $(S, F_c, A_p)$                     // return a summary
```

Algorithm 2 captures the operation of our ISR in a generic, abstract form. The ISR provides only a single operation, **record**, taking two parameters s, v and returning three results s', f', a'. The value v is associated with logical time-step s. The **record** operation first uses s to increase the ISR's internal step counter S as needed to the maximum step ever seen so far, and to record the first value v submitted at each step. The ISR then uses some binary combinator **aggregate**, which we will elaborate later in this section, to summarize all values seen during each step. If the step s associated with v is smaller than the ISR's internal step counter S, this means the provided value v is obsolete and the ISR simply discards it. In any case, the ISR returns its internal step counter S, the first value presented in the current step, and the aggregation of *all* values presented in the immediately-prior step.

This ISR formulation assumes that there is a well-defined "base" value we call **nil**, such that **aggregate**$(v,$ **nil**$) = v$. Also, for clarity, Algorithm 2 is formulated as if the ISR permanently records values from *all* historical time steps. This is obviously unnecessary, since the ISR only ever returns the first and aggregate values for the current and prior step, respectively. An ISR implementation thus requires only constant space if the values presented to it are constant size.[5]

4.2.3 Specialized ISR for concrete QuePaxa In the concrete QuePaxa protocol we must instantiate the generic ISR with a suitable value type, **nil** value, and **aggregate** combinator. Since the abstract consensus algorithm (Algorithm 1) uses proposal sets, a naïve ISR for QuePaxa might use a proposal set as its value type, the empty set $\emptyset$ as its **nil** value, and set union $\cup$ as the **aggregate** combinator.

In practice, since we only ever need the **best** or highest-priority proposal in a set, a more optimized implementation of QuePaxa can use simple binary integers as ISR values, zero as **nil**, and integer maximum for **aggregate**. A realistic

[5] A QuePaxa recorder must also store a slot and step number, of course, which might be unbounded in principle. In practice, however, slot numbers may be limited to fixed-size integers by resetting them at reconfiguration events, and forcing a reconfiguration before slot number overflow. Step numbers may be limited to ≈ 10 bits in practice, because the probability of a slot remaining undecided for more than ≈ 256 rounds is cryptographically negligible.

```
Algorithm 4: Protocol for QuePaxa proposer $i$
  Input: $v$ preferred value of this proposer $i$
  $s \leftarrow 4 \times 1 + 0$                        // start at round 1, phase 0
  $p \leftarrow \langle H, i, v \rangle$                    // initial proposal template
  repeat
      $p_j \leftarrow p$ for all recorders $j$              // prepare proposals
      if $s$ mod $4 = 0$ and ($s > 4$ or $i$ is not leader) then
          $p_j$.priority $\leftarrow$ random$(1..H-1)$ for all $j$
      Send record$(s, p_i)$ in parallel to each recorder $j$
      Await $R \leftarrow$ quorum of replies $(s'_j, f'_j, a'_j)$
      if $s'_j = s$ in all replies received in $R$ then
          if $s$ mod $4 = 0$ then                   // phase 0: propose
              if $f'_j$.priority $= H$ in all replies then
                  return $f'_j$.value from any reply in $R$
              $p \leftarrow$ best$_j$ of $f'_j$ from all replies in $R$
          if $s$ mod $4 = 1$ then                   // phase 1: spread $E$
                                                    // no action required
          if $s$ mod $4 = 2$ then     // phase 2: gather $E$, spread $C$
              if $p =$ best$_j$ of $a'_j$ from all replies in $R$ then
                  return $p$.value                  // report decision
          if $s$ mod $4 = 3$ then                   // phase 3: gather $C$
              $p \leftarrow$ best$_j$ of $a'_j$ from all replies in $R$
          $s \leftarrow s + 1$                          // advance to next step
      else if any reply in $R$ has $s'_j > s$ then
          $s, p \leftarrow s'_j, f'_j$                     // catch up to step $s'_j$
```

ISR for QuePaxa is thus constant space. For completeness, Algorithm 3 presents pseudocode for the concrete, integer-specialized, constant-space ISR needed by the concrete QuePaxa proposer protocol, which we describe next.

4.2.4 Concrete QuePaxa proposer protocol

Algorithm 4 presents pseudocode for the concrete QuePaxa proposer algorithm. The algorithm uses four logical time-steps per consensus round, starting at step $s = 4$ to represent round 1, phase 0. Figure 5 illustrates how these four phases of the concrete protocol correspond to and implement the three **tcast** invocations in Algorithm 1 through interactions with the recorders and their ISR state, as detailed below. Each step incurs one round trip between the proposer and a majority of recorders.

A proposal is logically a ⟨**priority**, **proposer**, **value**⟩ triple. We assume each component is encoded in a fixed-width binary format, then concatenated, so that the ISR-based recorders defined above see a proposal simply as a single binary integer.

Because the ISR aggregates values using integer maximum, and **priority** is the first component of the triple, ISR aggregation selects the highest-priority proposal of those submitted during a step, disambiguated by **proposer** in case of a tie.

Proposal randomization: Phase 0 of each round, where s mod $4 = 0$, implements the prioritization of proposals and the first **tcast** in Algorithm 1. Proposer i chooses a random priority on behalf of each recorder, except in leader-based rounds as discussed later in Section 4.2.5. After sending each proposal p_j to recorder j, proposer i awaits replies of the form (s'_j, f'_j, a'_j) from a majority quorum of recorders. If the step s'_j response from each recorder j in this quorum is equal to the proposer's step number s, the proposer tests for a fast-path decision (Section 4.2.5), then chooses the **best** (highest-priority) proposal from all the f'_j (first-value) responses in the quorum, as i's new proposal p for phase 1 below.

Proposer catch-up: In any phase, if proposer i receives a response (s'_j, f'_j, a'_j) from any recorder j whose $s'_j > s$, this means that proposer i has fallen behind recorder j (and hence behind some other proposer) in logical time. In this case, proposer i simply "catches up" to step s'_j directly, by (a) adopting s'_j as i's new step number s, and (b) taking f'_j as i's proposal template p for this later step.

Spread/gather propagation: Phases 1–3 of Algorithm 4 implement the last two **tcast** invocations in Algorithm 1, in which at least one replica's input is broadcast to *all* live replicas. Algorithm 4 implements these **tcast** operations in two steps each: a *spread* step spreads some proposer's input to a majority of recorders, then a *gather* step gathers knowledge of these spread inputs from a majority of recorders. As Fig. 5 illustrates, Algorithm 4 pipelines the 2-step spread/gather of existent sets (the second **tcast** in Algorithm 1) together with the 2-step spread/gather of common sets (the third **tcast**), so that these operations take only three steps total in Algorithm 4.

For phase 1 of each consensus round, no phase-specific code is required in Algorithm 4. In this phase, proposer i spreads the "best-of-a-quorum" proposal p resulting from phase 0 (corresponding to the **best** of set P in Algorithm 1) to a quorum of recorders. If proposer i succeeds in this venture, then at the end of phase 1, proposer i knows that the *existence* of its proposal p (or a better one) will become known to *all* proposers by the next phase. This proposal p thus corresponds to the **best** of set P' in Algorithm 1, a proposal set guaranteed to appear in all replicas' existent (E) sets in the abstract algorithm. If proposer i fails to spread its proposal p to a quorum of replicas before some replica advances to the next phase, however, then the generic catch-up logic above leaves i with a (possibly different) proposal that was successfully spread by another (faster) proposer. Either way, i's proposal p at the end of phase 1 is now a *common* proposal.

Phase 2 of each round serves three purposes: to gather knowledge of existent (E) proposals, to spread knowledge of common (C) proposals, and to determine if consensus has been reached. Any proposal that was successfully spread (became common) in phase 1 will have been aggregated by a majority of recorders' ISRs in phase 1. Any such proposal will thus be accounted for in the prior-step-aggregate a'_j returned by *some* recorder j that i queries in phase 2. The best of a quorum of these aggregates is thus the best of an existent (E) set in Algorithm 1. Further, the working proposal p at the end of phase 2 corresponds to a universal (U) proposal, because the *knowledge that p is common* was spread to a quorum of recorders during this phase. Thus, proposer i can actually perform consensus detection – representing the test if $\mathbf{best}(E) = \mathbf{best}(U)$ in Algorithm 1 – at the end of this phase. Proposer i returns a decision immediately if this test succeeds.

Phase 3 is required only in consensus rounds that do not succeed in deciding from proposer i's perspective. In this phase, i gathers knowledge of common (C) proposals, exactly as it gathered knowledge of existent proposals in phase 2. At the end of phase 3, i selects the best of the prior-step-aggregates a'_j of the recorders j in its quorum of responses as its initial proposal p for the next consensus round. This p corresponds to the next candidate $\mathbf{best}(C).\mathbf{value}$ computed in Algorithm 1, and defines i's preferred value in the next round.

4.2.5 Fast path: supporting leader-based rounds The concrete QuePaxa protocol can implement either leaderless asynchronous consensus or efficient leader-based consensus. At the start of each round, all proposers must have already agreed on which proposer, if any, is the leader for this round. This agreement may derive from a prior decision, for example.

In leaderless rounds, all proposers choose the priorities of their proposals as random integers between 1 and $H - 1$, where H is the highest possible priority. In this case, no proposer is behaviorally distinguished and QuePaxa acts as an asynchronous consensus protocol in this round.

In leader-based rounds, however, the unique designated leader attaches the highest priority H, reserved for this purpose, to all of its proposals. If the leader's proposal is the first to reach a quorum of recorders in phase 0, then this high-priority proposal naturally dominates the consensus process: *only* the leader's high-priority proposal can ever subsequently be chosen. If the leader obtains such a quorum in phase 0, therefore, the leader can decide at the end of phase 0, after only a single round-trip with the proposers. Under typical network conditions, this fast path enables QuePaxa to commit in a single round-trip, equivalent in efficiency to Multi-Paxos or Raft commitment by an already-prepared leader.

A strong network adversary can always prevent leader-based rounds from succeeding, *e.g.*, by scheduling messages such that the leader's proposal propagates to all proposers' E sets but to none of their U sets. We would thus lose robustness to asynchrony if we always used leader-based rounds. QuePaxa therefore uses a leader only in the *first* round of any slot, then falls back on leaderless rounds if the first round fails to decide. In this way, the leader can generally decide on the fast path in the first round under normal network conditions, while subsequent leaderless rounds provide a robust asynchronous backup path if the first round fails to decide.[6]

This leader-based fast path with asynchronous backup addresses our first main "tyranny of timeouts" challenge, liveness loss under network asynchrony. To address the other two challenges, we next focus on how QuePaxa uses the above consensus protocol for state machine replication (SMR).

5 High performance SMR with Hedging

This section elaborates on how QuePaxa leverages hedging for efficiency and dynamically optimizes the hedging schedule.

5.1 Retroactive versus proactive risk management

Hedging is the practice of launching operations redundantly on different nodes simultaneously, possibly but not necessarily staggered by brief delays, in order to "hedge one's bets" against an instance of the operation taking unexpectedly long [23, 71]. This practice is well-established in large-scale multi-tier query architectures, but to our knowledge QuePaxa is the first work to apply this concept in consensus protocols.

There is a basic difference between a timeout and a hedging delay. A timeout serves to detect a likely failure retroactively, as evidenced by a lack of observed normal-case progress. A timeout typically initiates an abnormal-case recovery process, such as a view change, which interferes with normal-case progress if triggered too early. Hedging initiates *non-interfering* parallel effort, in contrast, proactively limiting risks of long delays. Hedging is safe and often useful even if no failure has occurred. A timeout can never sensibly be configured to zero, as this would leave no time for normal-case progress and would doom the system to an endless failure-recovery loop. A hedging delay of zero not only makes sense but is common, whenever the reduced risk of long delays justifies the costs of simultaneous redundant effort.

5.2 Using hedging instead of timeouts in QuePaxa

Leveraging the fact that multiple proposers may be simultaneously active in any protocol step without destructive interference (Section 3.4), QuePaxa organizes potential proposers into a *hedging schedule* or delayed-activation sequence. The designated leader, if any, is always the first in the schedule with a delay of zero. All other proposers follow in some known order, sorted in non-decreasing order of associated delays. Each proposer in the schedule waits its associated delay before proposing, and does so *only* if it has not by then seen evidence that some other proposer (probably earlier in the schedule) has already driven the relevant step to completion.

While the hedging schedule technically needs to include only $f + 1$ proposers to ensure liveness against f failures, QuePaxa always includes all proposers in the schedule for simplicity. QuePaxa currently just chooses a single *base delay* parameter δ, then assigns the second proposer (after the leader) a hedging delay of δ, assigns the third proposer a hedging delay of 2δ, etc. Other approaches to scheduling are certainly feasible, such as launching the first two proposers simultaneously, or assigning delays to later proposers based

[6] We expect that fast-path optimizations could be pushed further. Using flexible quorums [6, 39], for example, we could reduce the quorum size required in the fast commit path, at the cost of requiring a larger quorum in the following step. We leave such optimizations to future work, however.

on historically-measured timings of all proposers. We leave such scheduling refinements to future work, however.

During periods of synchrony when the maximum round-trip network delay Δ (unknown to the protocol) is less than the current base delay δ, normally only the first proposer in the schedule will activate, the rest remaining passive upon seeing the leader making progress. Even if δ is less than Δ by some constant factor, however (*i.e.,* $\Delta = O(\delta)$), at most a constant number of proposers will activate in each step, ensuring the same $O(n)$ asymptotic communication cost as in conventional leader-based protocols during synchronous periods. Choosing δ too small can cause too many proposers to activate and revert to the $O(n^2)$ worst-case communication cost that applies under asynchronous conditions. But we will next explore how QuePaxa tunes its choices of leader to prevalent conditions whenever network delays are stable.

5.3 Leader tuning in QuePaxa

We usually do not know how each replica will perform as leader until we try it. Even then, observations may be noisy, affected by load and many other factors. Leader selection is thus a form of *multi-armed bandit* problem, a term inspired by a gambling machine ("bandit") whose multiple levers ("arms") each have different, unknown payout chances [78].

QuePaxa adopts a well-known strategy for such problems, first *exploring* or testing alternatives, then *exploiting* or applying learned knowledge. QuePaxa divides SMR slots into fixed-length *epochs* each with a stable leader. In the first $2n+1$ epochs QuePaxa rotates among leaders, round-robin, giving each replica two epochs as leader. After this exploration, QuePaxa exploits these trials by forming and agreeing on a hedging schedule with replicas sorted in descending order of their observed average epoch completion time. QuePaxa then continues monitoring the current leader's performance, recomputing the hedging schedule each epoch, but it no longer proactively explores other leaders unless the current leader's performance falls below that of the next in the schedule.[7]

6 Implementation

We implemented QuePaxa using Go version 1.18 [52], in 4368 lines of code as counted by CLOC [22]. We use the standard Go network library and TCP [81] for reliable point-to-point links between replicas. We used Protobuf encoding [36] with the gRPC [30] plugin for remote procedure call.

Our implementation supports batching in both submitters and proposers, as well as pipelining, as in existing implementations of Rabia [65] and EPaxos [54]. The current prototype does not implement reconfiguration, but could readily be extended to do so by using consensus to agree on new configurations, as per standard existing practices [42, 64]. An open source release of our prototype is available [83].

6.1 Reducing the leader bottleneck in LAN scenarios

In leader-driven consensus, the leader is often a performance bottleneck because it must send n messages and receive up to n messages per commit, even on the fast path. The size of these messages mainly depends on the batch sizes of submitters: while QuePaxa metadata is usually only a few bytes, batches of commands are often kilobytes or even megabytes.

To reduce this bottleneck in data center settings, QuePaxa leverages the property of modern data center LANs that when a node broadcasts a message m to several others, recipients usually receive m at almost the same time, within a millisecond-latency upper bound [46, 66]. After a submitter broadcasts a batch of commands to all replicas in QuePaxa, it sends only a small unique batch ID (*e.g.*, a cryptographic hash) to the consensus layer. The consensus logic then agrees on batch IDs rather than batch contents, reducing the leader's bandwidth burden. When a recorder receives a proposal containing a batch ID, it first checks whether it has received that batch's content, and if so – as is common in a data center LAN – the recorder responds immediately according to algorithm 2. If the recorder has not yet received the batch, it first requests the batch from any proposer before responding. Other protocols such as Rabia [66] and NOPaxos [46] have employed similar optimizations specific to data center networks.

6.2 Model-checked implementation in Promela

In addition to the above full prototype in Go, we implemented the core QuePaxa consensus logic in Promela and exhaustively verified the model's safety using the Spin model checker [37]. See Appendix D for further details. This verification is subject to the usual limitations of model checking, such as the need to constrain the problem to a finite state space and the inability to verify properties such as the probabilistic success of consensus rounds. The verification nevertheless increases our confidence in the basic algorithm's correctness.

7 Experimental evaluation

We evaluated QuePaxa to address the following key questions: (1) Does QuePaxa perform comparably to state-of-the-art consensus algorithms under normal network conditions? (2) Can QuePaxa offer robustness to adversarial network conditions? (3) What is the impact of hedging on liveness and on recovery time? (4) Can QuePaxa converge to the best hedging schedule in a realistic heterogeneous deployment?

We compare QuePaxa's performance against four state-of-the-art SMR algorithms: Multi-Paxos [42], Raft [64], Rabia [66], and EPaxos [55]. Multi-Paxos is a classic leader-based algorithm. Raft is a leader-based algorithm based on viewstamped replication [62]. Rabia uses randomization to simplify SMR, specializing in data-center networks.

EPaxos is a multi-leader protocol that partitions commands across consensus instances in parallel as their dependencies permit. EPaxos's primary goal of improving throughput via parallelism is orthogonal and complementary to QuePaxa's

[7] Refinements inspired by "restless bandits" [90] might periodically re-explore to detect dynamic performance improvements in non-leader replicas.

Algorithm	Implementation		Lines of Code	Note
Multi-Paxos	existing	[54]	2891	8
EPaxos	existing	[54]	4658	
Rabia	existing	[65]	4572	
Multi-Paxos	new	[82]	2743	
Raft	new	[82]	2802	
QuePaxa	new	[83]	4368	

Table 1. Lines of code [22] in the SMR implementations.

primary goal of robustness, and thus represents a less "apples-to-apples" baseline than Multi-Paxos and Raft, but we include it when feasible for a diverse comparison.

When feasible we use the existing Go implementations of Multi-Paxos and EPaxos [54], and of Rabia [65], each with minor enhancements for our experiments [56, 67]. We found that the existing Multi-Paxos/EPaxos code base [54] does not correctly implement leader-failure scenarios, however. Upon a leader timeout, the new leader does not initiate the prepare-promise phase. With five replicas and command execution enabled via the `-exec` flag, after any replica failure, the existing implementation never makes any progress.

Due to this issue, we use this existing EPaxos code base only for normal-case execution in Section 7.2 below. Other experiments use our own publicly-available Paxos and Raft implementations [82], which correctly handle replica failures.

For reference, Table 1 lists the SMR implementations we evaluated, each with line counts measured with CLOC [22].

7.1 Experimental configuration and workloads

We use Amazon EC2 virtual machines [8] of type c4.4xlarge (16 virtual CPUs, 30 GB memory) and c4.2xlarge (8 virtual CPUs, 15 GB memory) for replicas and submitters, respectively. We test both a local-area-network (LAN) configuration where all replicas and submitters are located in the North Virginia AWS region, and in a wide-area-network (WAN) setup where the replicas and submitters are distributed globally across AWS regions Tokyo, Mumbai, Singapore, Ireland, and São Paulo. We use Ubuntu Linux 20.04.5 LTS [87].

Following the evaluation approach of Rabia [65], we use a string-to-string key-value store as the backend application.

Submitters generate traffic with a Poisson distribution in the open-loop model [76]. All algorithms use batching in both submitters and proposers. Multi-Paxos, EPaxos, and QuePaxa support pipelining, while the Raft and Rabia implementations do not. Client requests are 17 bytes (1-byte GET/PUT opcode plus 8-byte keys and values), consistent with request sizes common in production systems and prior research [15, 66].

For Multi-Paxos, Raft, Rabia and QuePaxa we measure end-to-end execution latency, counting the submitter-observed time required both to order and execute commands. EPaxos's execution latency is significantly higher than its commit latency [86], however, due to its dependency tracking and parallelization of commands, a feature orthogonal to this work's focus. For EPaxos, we therefore measure both ordering-with-execution latency, denoted "EPaxos-exec" in the graphs below, and commit-only latency (omitting the time required to execute commands), denoted "EPaxos-commit" below.

We run each experiment for one minute, repeating experiments 3 times. We measure throughput in commands per second (cmd/sec), where a command is one 17-byte request.

7.2 Normal-case performance evaluation

We first evaluate the performance of QuePaxa under normal failure-free conditions, in LAN and WAN settings. We employ pipelining only in the WAN scenario (with a pipeline length of 10), as we did not observe any benefit from pipelining in the LAN case. Figure 6 depicts the results of this experiment.

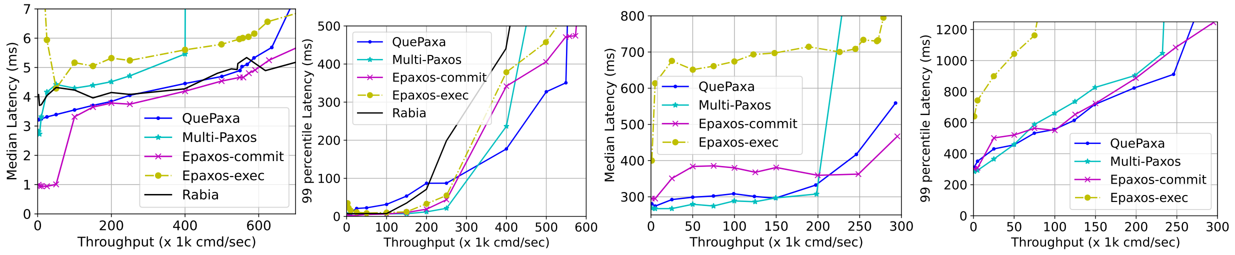
We observe in figure 6a that QuePaxa delivers LAN saturation throughput of 584k cmd/sec under a median latency upper bound of 5.8ms, compared with Multi-Paxos's saturation throughput of 400k under 5.6ms. We attribute QuePaxa's higher LAN throughput to the optimization discussed in Section 6.1, using submitter-driven batch dissemination to reduce critical-path bandwidth. Multi-Paxos carries these batches in the critical path, thereby incurring higher latencies.

We see in figure 6a that EPaxos-commit (without command execution) delivers LAN throughput of 699k cmd/sec under 5.8ms latency, 16.5% higher than QuePaxa's saturation throughput. This higher throughput results from EPaxos's partitioning of commands across consensus instances, a useful optimization that could be combined with QuePaxa but is outside the scope of this work. The EPaxos-commit experiment uses a conflict rate of 2%, thus committing commands in one round-trip 98% of the time. Because our QuePaxa prototype lacks this partitioning and employs only a single leader at once, its performance is naturally bottlenecked by the leader.

EPaxos-commit's 2% conflict rate impacts its 99% LAN tail latency shown in figure 6b, however. In the WAN case shown in Fig. 6c, the median latency of EPaxos-exec with command execution is 400ms higher on average than QuePaxa (in the 50k–200k cmd/sec range). This higher latency results from EPaxos's dependency management, consistent with prior observations [51, 86]. Finally, even the median WAN latency of EPaxos-commit is 60ms higher than QuePaxa in the 0-150k cmd/sec range. This is because a single conflicting command in a batch requires EPaxos to take the two-round-trip slow path, thus impacting not just tail latencies but the latencies of most commands [86].

We observe in Fig. 6a that Rabia's median latency is comparable to QuePaxa's. As Fig. 6b shows, however, Rabia's tail latency is 100ms-300ms higher than QuePaxa's in the throughput region 250k-400k, due to the cost of slot forfeiting in Rabia [66]. Moreover, we observed that under WAN deployment, the throughput of Rabia falls under 10 cmd/sec

[8] The Rabia implementation includes log compaction; the others do not.



(a) Median Latency on LAN **(b)** 99% Latency on LAN **(c)** Median Latency on WAN **(d)** 99% Latency on WAN

Figure 6. Throughput versus latency for normal-case execution, comparing QuePaxa to Rabia, Multi-Paxos and EPaxos.

with latency greater than 2s. This low WAN performance stems from Rabia's assumption that network delay is small compared with the interval between consecutive requests [66, §3.2], a condition that holds in the LAN but not the WAN.

7.3 Scalability

This experiment evaluates the scalability of QuePaxa in a single data center (North Virginia), with an increasing number of replicas. We measure the saturation throughput of each algorithm, under a 5.8ms median latency upper bound, which we chose based on the saturation point we observed in Fig. 6a. Figure 7 depicts these scalability results.

In contrast with blockchain algorithms targeting scalability up to hundreds of nodes [34, 93], crash-fault-tolerant protocols are generally deployed and evaluated at smaller scales of under 15 nodes [41, 49], so we follow this convention.

We compare QuePaxa, Multi-Paxos, and EPaxos in this experiment. We observed that EPaxos is hard-coded to support only up to 5 replicas. With more than 5 replicas, EPaxos crashes with an index-out-of-bounds exception, which stems from a hard-coded array of size 5. We reported this as a bug in the EPaxos code repository [54]. Hence, for EPaxos, we depict only the 3- and 5-replica configurations.

We observe that the throughput of QuePaxa decreases from 584k to 467k cmd/sec as the replica count increases from 3 to 13. QuePaxa uses a quorum-based broadcast for replicating commands. When replication factor increases, the current leader in QuePaxa must exchange an increasing number of messages with non-leader replicas. This load explains the 20% throughput drop with increasing replication factor.

We observe that for all replica configuration sizes, the throughput of QuePaxa is 35% higher than Multi-Paxos, on average. We attribute this gain to QuePaxa's LAN optimization of using client replicas to disseminate the requests, reducing critical-path bandwidth usage. With this optimization disabled, we find that QuePaxa and Multi-Paxos deliver essentially the same throughput in all configuration sizes.

Finally, we observe that EPaxos provides better throughput than QuePaxa and Multi-Paxos. While we do not have the empirical data to showcase the scalability of EPaxos, theoretically, we expect EPaxos to scale better than QuePaxa and Multi-Paxos, because EPaxos partitions over commands and only partially orders commands. In contrast, QuePaxa and Multi-Paxos place all commands in a total order.

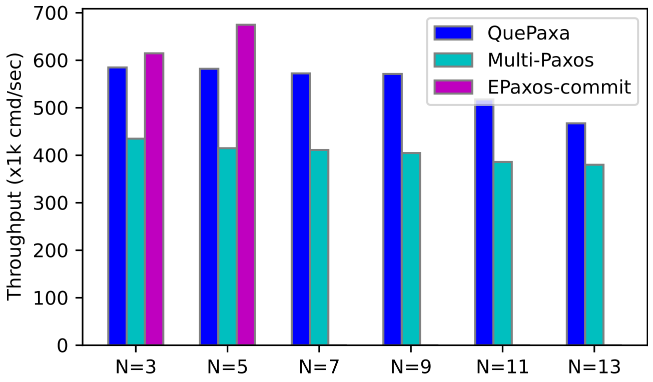


Figure 7. Scalability in a single data-center deployment

7.4 Performance under adversarial network conditions

This experiment evaluates QuePaxa under simulated attack by a network adversary, similar to attacks used in recent consensus robustness work [80, 84, 85]. This adversary controls communication delays of a minority of replicas, with the aim of compromising the liveness and performance of consensus. Our simulated adversary is intended to model realistic network-based attacks, such as a denial-of-service (DoS) attack focused on a few replicas, or the use of BGP hijacking [19] to divert routes and gain direct control over delays between some replicas. Our simulated attacker increases the egress packet latency of a minority of replicas up to 500ms dynamically in 5s time epochs. This experiment runs in the WAN setting with 5 replicas. Figure 8 depicts these results.

We observe that under the simulated attack conditions, QuePaxa sustains a throughput of at least 75k cmd/sec under 380ms median latency. In contrast, the throughput of Multi-Paxos and Raft saturate at 2.5k cmd/sec. We interpret these results as confirming that QuePaxa's asynchronous core provides significant robustness under attack, in contrast with Multi-Paxos and Raft, which stall and make little or no progress when the current leader is under attack.

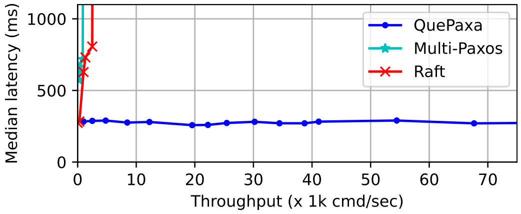


Figure 8. Throughput versus median latency with a network adversary that randomly attacks a minority of replicas at once.

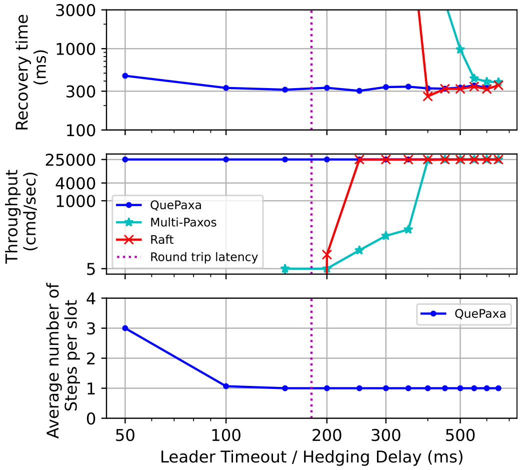


Figure 9. Impact of timeout/hedging delay configuration on recovery time (upper) and throughput (middle).

7.5 Impact of protocol delays on liveness and recovery

This experiment evaluates the impact of configured protocol delays – hedging delays in QuePaxa, and view-change timeouts in legacy protocols – on protocol liveness and recovery time after leader failure. We use five replicas in the WAN setting, among which we measured an average round-trip delay of 180ms. In this experiment, five submitters inject a constant aggregate load of 25k commands per second.

We first evaluate throughput under varying hedging delays (QuePaxa) or leader timeouts (existing protocols). We then investigate the protocols' recovery time after leader failure.

To measure recovery time, we "crash-stop" the leader at time $t = 15$ seconds, and measure the time it subsequently takes for a new leader or alternate proposer to resume making progress. Figure 9 depicts these experimental results.

Protocol Liveness: As Fig. 9 (middle) shows, we find that QuePaxa consistently delivers throughput of 25k cmd/sec, keeping pace with the imposed load regardless of hedging delay. When QuePaxa's hedging delay is less than the average network round-trip-time of 180ms, the non-leader replicas also propose commands. However, each non-leader replica waits a short time before proposing in a slot, while the leader proposes with no delay. Even with competition from other proposers, we observe that the leader still "wins" most slots and commits in one round-trip. Moreover, when non-leader replicas propose concurrently with the leader, we find that the non-leader proposers often help the leader propagate its command, confirming that proposers effectively cooperate in QuePaxa. Even when hedging delay is less than network round-trip time, QuePaxa thus delivers stable performance.

As shown in Fig. 9 (bottom), the main cost of hedging in QuePaxa is increased bandwidth use. When the hedging delay is less than the network round-trip time, more than one proposer submits commands, increasing message overhead.

Multi-Paxos and Raft, in contrast, rapidly lose throughput and ultimately liveness when their view-change timeout is close to or less than the average network round-trip time. In this case, no leader can make progress without interference from false view-change triggers. With a timeout above 330ms, Multi-Paxos and Raft deliver 25k cmd/sec throughput as expected, as the leader can proceed without interruption.

We conclude that QuePaxa maintains liveness and performance with any hedging delay, while Multi-Paxos and Raft depend on their view-change timeout being correctly configured to at least about 1.8 times the network round-trip time.

Leader recovery: As Fig. 9 (upper) shows, we see that for all hedging delays, QuePaxa exhibits a recovery time after leader failure between 303ms and 473ms. QuePaxa exhibits close to its lowest recovery time with a hedging delay around 200ms – barely above the 180ms average RTT. When hedging delay is below the RTT, recovery time increases slightly due to the redundant proposers, but nevertheless remains modest.

Multi-Paxos and Raft exhibit orders-of-magnitude higher recovery times with timeouts insufficiently above the network's RTT. Below 200ms timeouts, Multi-Paxos and Raft fail to stabilize at all and hence have no recovery time.

In Multi-Paxos, recovery time explodes with 100ms higher timeouts than Raft. This difference results from the Raft implementation using a multi-threaded gRPC design, while the Multi-Paxos code uses a single-threaded event-driven design.

With delays more than 500ms, all protocols converge to a recovery time roughly equal to network round-trip time plus the timeout, as we expect since recovery in this case depends primarily on the combination of network RTT (to run the view change) plus one timeout (to detect the need for it).

We conclude that QuePaxa robustly maintains low recovery times affected only slightly by configured hedging delay. Existing protocols effectively impose a hard lower bound on timeouts, otherwise risking high recovery time or no recovery.

7.6 Automatic convergence to the best leader

This experiment evaluates QuePaxa's auto-tuning mechanism to identify and converge on the best hedging schedule. We ask in particular: can QuePaxa find whichever leader maximizes performance, regardless of initial leader? This experiment uses five replicas in a single AWS region (Oregon), on five heterogeneous EC2 machines (t2.large, t2.2xlarge, c4.large, c4.xlarge, and c4.4xlarge) [8]. These EC2 types have varying computational and memory resources, with t2.large being

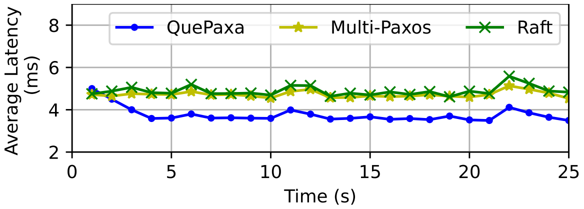


Figure 10. Automatic best-leader discovery in QuePaxa.

the weakest machine and c4.4xlarge being the strongest. For each run, the t2.large machine is the initial leader. We use a constant load of 80k cmd/sec, and measure median command execution latency (Section 5.3). Fig. 10 depicts these results.

We find that Multi-Paxos and Raft maintain a high latency of 5.2 ms, retaining the slow t2.large machine as leader because it never times out. QuePaxa's multi-armed-bandit optimization, in contrast, converges to the best leader after only 4 seconds, thereafter providing a 3.8ms latency. QuePaxa thus achieves 1.4ms lower latency than Raft and Multi-Paxos in this scenario, a significant benefit in a data-center setting.

8 Related Work

The most widespread consensus protocols use a leader to order requests and achieve one-round-trip normal-case commit latency at $O(n)$ cost [42, 62, 64]. Multi-leader variants allow several leaders to propose concurrently, spreading the leadership burden to optimize throughput [43, 49, 55, 84]. Other variants employ overlay networks to reduce bandwidth consumption [18, 50, 85]. Archipelago [9] achieves consensus deterministically without relying on a single leader. All of these protocols lose liveness under asynchrony, however.

Randomized consensus: Many algorithms use randomness to achieve asynchronous consensus [1, 14, 26, 29, 31, 53, 57, 63, 72, 94]. These algorithms are rarely implemented or deployed, however, due to high complexity and poor normal-case efficiency. QuePaxa builds on ideas from QSC [29], but QuePaxa introduces a single-round-trip fast path, $O(n)$ normal-case cost, hedging, and leader-choice optimizations.

Prior hybrid consensus protocols have combined failure detection for synchronous performance with randomization for asynchronous robustness [5, 34, 61, 79, 80, 85]. These protocols still rely on timeouts to recover from failures under synchronous operation, however, and do not achieve QuePaxa's single-round-trip, $O(n)$-cost normal-case efficiency.

Rabia [66] is a randomized crash-fault-tolerant SMR scheme that uses Ben-Or's asynchronous consensus algorithm [14] as a component. Rabia specializes in low-delay, high-capacity data-center networks, however, making assumptions and design choices that limit its usefulness in other contexts. Rabia's fast path incurs three network hops and quadratic message complexity, compared with QuePaxa's two-hop, linear-complexity fast path. Rabia assumes that incoming requests are (correctly) timestamped and that “message delay is small compared to the interval between two consecutive requests” ([66, §3.2]). Experimentally, we found Rabia usable only on low-delay LANs with high network capacity and few replicas (n=3 or 5), as examined in Section 7 and prior reports [85]. Appendix E provides an in-depth comparison with Rabia.

Hedging: Hedging is often employed in online interactive services, which typically operate under strict service-level objectives (SLOs) [11, 12, 23, 33, 68]. QuePaxa is the first consensus protocol to adapt hedging to allow multiple leaders to propose, while minimizing the message overhead. Prior work has explored other ways to make consensus more robust against network performance issues, however [44, 59, 60].

Automated tuning: Most consensus protocols contain many tunable parameters: *e.g.*, leader timeout, batch size, batch time, pipeline length, garbage collection frequency. Couceiro et al. [20] used machine learning to predict the performance of total-order-broadcast protocols. Paolo et al. [74] employed multi-armed-bandit theory to tune batching in consensus protocols. QuePaxa focuses on tuning leader choice and hedging schedule, and hence is complementary to prior work. Multi-armed bandit theory has been used in many domains outside of consensus, of course [3, 21, 45, 48, 91].

Orthogonal goals: As this work focuses on the liveness and performance robustness of consensus, it does not attempt to address many other useful goals: *e.g.*, achieving scalability by partitioning over commands [25, 55] or over state [6, 70], shrinking the quorum required in the fast commit path [6, 39], exploiting WAN locality [6, 58], reducing storage costs via erasure coding [88, 89], reducing the load on leaders by outsourcing work [92], or tolerating Byzantine replica faults [17, 93]. We expect that many techniques from these complementary works could be adapted to QuePaxa, but we leave these interesting challenges to future work.

9 Conclusion

QuePaxa is a novel asynchronous consensus algorithm with the efficiency of partially-synchronous protocols under normal conditions, while being far more robust to challenging conditions. Our evaluation confirms that QuePaxa achieves high common-case performance, robustness to DoS attack, low recovery time, and ability to converge to the best leader.

Acknowledgments

The authors would like to thank Marcos K. Aguilera, Pierluca Borsò, Aleksey Charapko, Rachid Guerraoui, Jovan Komatovic, Derek Leung, Louis-Henri Merino, Shailesh Mishra, Haochen Pan, Rodrigo Rodrigues, Lewis Tseng, and Haoqian Zhang for their helpful feedback on early drafts of this paper.

References

[1] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous Byzantine agreement with expected $O(1)$ rounds, expected communication, and optimal resilience. In *Financial Cryptography and Data Security (FC)*, pages 320–334. Springer, February 2019.

[2] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous Byzantine agreement. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 337–346, July 2019.

[3] Marco Abundo, Valerio Di Valerio, Valeria Cardellini, and Francesco Lo Presti. Bidding strategies in QoS-Aware cloud systems based on N-armed bandit problems. In *2014 IEEE 3rd Symposium on Network Cloud Computing and Applications (ncca 2014)*, pages 38–45. IEEE, February 2014.

[4] Marcos K. Aguilera and Sam Toueg. The correctness proof of Ben-Or's randomized consensus algorithm. *Distributed Computing*, 25:371–381, 2012.

[5] Marcos Kawazoe Aguilera and Sam Toueg. Failure detection and randomization: a hybrid approach to solve consensus. *SIAM Journal of Computing*, 28(3):890–903, 1998.

[6] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. WPaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):211–223, 2019.

[7] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, October 2018.

[8] Amazon. AWS instance types. https://aws.amazon.com/ec2/instance-types/, 2023.

[9] Karolos Antoniadis, Julien Benhaim, Antoine Desjardins, Poroma Elias, Vincent Gramoli, Rachid Guerraoui, Gauthier Voron, and Igor Zablotchi. Leaderless consensus. *Journal of Parallel and Distributed Computing*, 176:95–113, June 2023.

[10] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003.

[11] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[12] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE micro*, 23(2):22–28, 2003.

[13] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, pages 1–15. Springer, 1996.

[14] Michael Ben-Or. Another advantage of free choice (extended abstract) completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

[15] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference USENIX ATC 13*, pages 49–60, June 2013.

[16] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Science & Business Media, 2011.

[17] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[18] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. PigPaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of the 2021 International Conference on Management of Data*, pages 235–247, June 2021.

[19] Shinyoung Cho, Romain Fontugne, Kenjiro Cho, Alberto Dainotti, and Phillipa Gill. BGP hijacking classification. In *Proceedings of the Network Traffic Measurement and Analysis Conference (TMA)*, June 2019.

[20] Maria Couceiro, Paolo Romano, and Luis Rodrigues. A machine learning approach to performance prediction of total order broadcast protocols. In *2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 184–193. IEEE, September 2010.

[21] Penglin Dai, Zihua Hang, Kai Liu, Xiao Wu, Huanlai Xing, Zhaofei Yu, and Victor Chung Sing Lee. Multi-armed bandit learning for computation-intensive services in MEC-empowered vehicular networks. *IEEE Transactions on Vehicular Technology*, 69(7):7821–7834, 2020.

[22] Al Danial. Counting lines of code (CLOC). http://cloc.sourceforge.net/.

[23] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[24] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[25] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for Planet-Scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, April 2020.

[26] Paul Ezhilchelvan, Achour Mostefaoui, and Michel Raynal. Randomized multivalued consensus. In *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISORC 2001*, pages 195–200. IEEE, May 2001.

[27] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[28] Bryan Ford. Threshold logical clocks for asynchronous distributed coordination and consensus. *arXiv preprint arXiv:1907.07010*, 2019.

[29] Bryan Ford, Philipp Jovanovic, and Ewa Syta. Que sera consensus: Simple asynchronous agreement with private coins and threshold logical clocks. *arXiv preprint arXiv:2003.02291*, 2020.

[30] Cloud Native Computing Foundation. A high performance, open source universal RPC framework. https://grpc.io/, 2015.

[31] Roy Friedman, Achour Mostefaoui, and Michel Raynal. Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46–56, 2005.

[32] Eli Gafni and Leslie Lamport. Disk Paxos. In *Distributed Computing: 14th International Conference, DISC 2000*, pages 330–344, 2000.

[33] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, and Esa Hyytia. Reducing latency via redundant requests: Exact analysis. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):347–360, 2015.

[34] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and Ditto: Network-adaptive efficient consensus with asynchronous fallback. In *26th International Conference on Financial Cryptography and Data Security: (FC)*, pages 296–315. Springer, May 2022.

[35] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. 20(1):51–83, January 2007.

[36] Google. Protocol buffers. https://developers.google.com/protocol-buffers/, 2020.

[37] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[38] Gerard J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13:289–307, November 1998.

[39] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible Paxos: Quorum intersection revisited. In *Proceedings of the 20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, December 2016.

[40] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Workshop on Hot Topics*

in Operating Systems (HotOS), May 2017.

[41] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, April 2020.

[42] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4*, 32:51–58, December 2001.

[43] Leslie Lamport. Generalized consensus and Paxos. 2005.

[44] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the Falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, page 279–294. Association for Computing Machinery, 2011.

[45] Feng Li, Dongxiao Yu, Huan Yang, Jiguo Yu, Holger Karl, and Xiuzhen Cheng. Multi-armed-bandit-based spectrum scheduling algorithms in wireless networks: A survey. *IEEE Wireless Communications*, 27(1):24–30, 2020.

[46] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 467–483, November 2016.

[47] Tom Lianza and Chris Snook. Cloudflare outage. https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/, November 2020.

[48] Jingyang Lu, Lun Li, Dan Shen, Genshe Chen, Bin Jia, Erik Blasch, and Khanh Pham. Dynamic multi-arm bandit game based multi-agents spectrum sharing strategy design. In *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, pages 1–6. IEEE, September 2017.

[49] Yanhua Mao, Flavio Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, December 2008.

[50] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 527–536. IEEE, June 2010.

[51] Venkata Swaroop Matte, Aleksey Charapko, and Abutalib Aghayev. Scalable but wasteful: Current state of replication in the cloud. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, pages 42–49, July 2021.

[52] Jeff Meyerson. The Go programming language. *IEEE Software*, 31(5):104–104, 2014.

[53] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42, 2016.

[54] Iulian Moraru, David G Andersen, and Michael Kaminsky. EPaxos go-lang. https://github.com/efficient/epaxos/, 2013.

[55] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372, November 2013.

[56] Iulian Moraru, David G Andersen, Michael Kaminsky, and Pasindu Tennage. EPaxos go-lang – modified for QuePaxa experiments. https://github.com/dedis/quepaxa-ePaxos-open-loop, September 2023.

[57] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 2–9, July 2014.

[58] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. DPaxos: Managing data closer to users for low-latency and mobile applications. In *ACM SIGMOD/PODS Conference on Management of Data*, June 2018.

[59] Harald Ng, Seif Haridi, and Paris Carbone. Omni-Paxos: Breaking the barriers of partial connectivity. In *Eighteenth European Conference on Computer Systems (EuroSys)*, pages 314–330, May 2023.

[60] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. Tolerating slowdowns in replicated state machines using copilots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2020.

[61] Stavros Nikolaou and Robbert Van Renesse. Turtle consensus: Moving target defense for consensus. In *Proceedings of the 16th Annual Middleware Conference*, pages 185–196, December 2015.

[62] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17, january 1988.

[63] Afonso Oliveira, Henrique Moniz, and Rodrigo Rodrigues. Alea-BFT: Practical asynchronous Byzantine fault tolerance. *arXiv preprint arXiv:2202.02071*, February 2022.

[64] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference ATC14)*, pages 305–319, June 2014.

[65] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia. https://github.com/haochenpan/rabia, 2021. Rabia implementation in the Go language (GitHub repository).

[66] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia: Simplifying state-machine replication through randomization. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 472–487, October 2021.

[67] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, Roberto Palmieri, and Pasindu Tennage. Rabia – modified for QuePaxa experiments. https://github.com/dedis/quepaxa-rabia-open-loop, September 2023.

[68] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021.

[69] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, April 2018.

[70] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2016.

[71] Mia Primorac, Katerina J Argyraki, and Edouard Bugnion. When to hedge in interactive services. In *18th USENIX Symposium on Networked Systems Design and Implementation NSDI*, pages 373–387, April 2021.

[72] Michel Raynal. *Fault-tolerant Message-Passing Distributed Systems: an Algorithmic Approach*. Springer, 2018.

[73] Eric Rescorla and Tim Dierks. The transport layer security (TLS) protocol version 1.3, August 2018. RFC 8446.

[74] Paolo Romano and Matteo Leonetti. Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning. In *2012 International Conference on Computing, Networking and Communications (ICNC)*, pages 786–792. IEEE, January 2012.

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

[76] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd USENIX*

Symposium on Networked Systems Design and Implementation (NSDI 06). USENIX, May 2006.

[77] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979.

[78] Aleksandrs Slivkins et al. Introduction to multi-armed bandits. *Foundations and Trends in Machine Learning*, 12(1-2):1–286, 2019.

[79] Alexander Spiegelman. In search for an optimal authenticated Byzantine agreement. In *Proceedings of the 35th International Symposium on Distributed Computing (DISC)*, October 2021.

[80] Alexander Spiegelman and Arik Rinberg. ACE: Abstract consensus encapsulation for liveness boosting of state machine replication. *International Conference on Principles of Distributed Systems, OPODIS*, December 2020.

[81] Transmission control protocol, September 1981. RFC 793.

[82] Pasindu Tennage. Paxos and Raft, September 2023. GitHub repository https://github.com/dedis/paxos-and-raft.

[83] Pasindu Tennage. QuePaxa, September 2023. GitHub repository https://github.com/dedis/quepaxa.

[84] Pasindu Tennage, Cristina Basescu, Eleftherios Kokoris Kogias, Ewa Syta, Philipp Jovanovic, and Bryan Ford. Baxos: Backing off for robust and efficient consensus. *arXiv preprint arXiv:2204.10934*, April 2022.

[85] Pasindu Tennage, Antoine Desjardins, and Eleftherios Kokoris Kogias. Mandator and Sporades: Robust wide-area consensus with efficient request dissemination. *arXiv preprint arXiv:2209.06152*, 2022.

[86] Sarah Tollman, Seo Jin Park, and John K Ousterhout. EPaxos revisited. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 613–632, April 2021.

[87] Ubuntu. Ubuntu Linux. https://releases.ubuntu.com/focal/, 2023.

[88] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. Near-optimal latency versus cost tradeoffs in geo-distributed storage. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, February 2020.

[89] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. CRaft: An Erasure-coding-supported version of Raft for reducing storage cost and network cost. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, February 2020.

[90] P. Whittle. Restless bandits: Activity allocation in a changing world. *Journal of Applied Probability*, 25(A):287–298, 1988.

[91] Wenchao Xia, Tony QS Quek, Kun Guo, Wanli Wen, Howard H Yang, and Hongbo Zhu. Multi-armed bandit-based client scheduling for federated learning. *IEEE Transactions on Wireless Communications*, 19(11):7108–7123, 2020.

[92] Zichen Xu, Christopher Stewart, and Jiacheng Huang. Elastic, geo-distributed RAFT. In *Proceedings of the International Symposium on Quality of Service*. Association for Computing Machinery, 2019.

[93] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, July 2019.

[94] Jialin Zhang and Wei Chen. Bounded cost algorithms for multivalued consensus using binary consensus instances. *Information Processing Letters*, 109(17):1005–1009, 2009.

Appendices

The following appendices contain material supplemental to the main paper above. The authors feel that this material may be useful in understanding certain subtleties of QuePaxa and its relation to prior work in more detail. Readers are advised, however, that the material in these appendices has not received the same level of peer review as the main paper.

A Approaches for choosing random priorities

The simple liveness analysis in Section 4.1.3 assumes that each replica chooses high-entropy priorities using private (local) randomness, so that there is a negligible chance that two or more replicas *tie* by choosing the same highest priority in a round. There are many potential ways to choose random priorities for QuePaxa, however, with different tradeoffs. Appendix A.1 below outlines three alternative ways to handle ties while using private coins, then Appendix A.2 describes how we adjust the per-round decision success probability calculations to account for any non-negligible chance of ties. Finally, Appendix A.3 briefly explores common-coin approaches to handling priorities, which trivially avoid ties but can be less performant when a minority of nodes have failed.

A.1 Choosing priorities via private coins

There are three readily-apparent alternative ways to handle ties: avoid them, break them, or detect and neutralize them. We summarize each of these approaches in turn.

Avoiding ties: The simple analysis in Section 4.1.3 applies verbatim with no adjustment if we simply choose priorities from a sufficiently-high-entropy (*e.g.*, cryptographic) random number distribution, thereby making the chance of a tie (cryptographically) negligible.

This approach keeps both the algorithm and its correctness proof simple, at the cost of requiring priorities to be fairly large, *i.e.,* hundreds of bits. Both the space cost of these large priorities, and the computational cost of generating them from a high-entropy cryptographic random source, may be an issue especially in high-performance implementations of QuePaxa.

Breaking ties: An almost-equally-simple approach is to prevent ties from occurring at all, by choosing the least-significant bits of priorities non-randomly so as to disambiguate them uniquely within a given round.

For example, we can first choose a low-entropy random binary integer, then append to it a binary encoding of the replica number, ensuring that the resulting priority associated with any replica's proposal is unique within a given round. Our main prototype implementation discussed in Section 6 and evaluated in Section 7 uses this approach.

An alternative tiebreaking approach is to choose a low-entropy random binary integer then append a binary encoding of the proposal value itself. Random ties associated with different proposals are then disambiguated by the proposal value, favoring proposals whose encodings represent larger integers. Random ties associated with identical proposals are not broken in this case but are harmless. Our Promela models of QuePaxa employ this approach.

Algorithm 5: Abstract QuePaxa with tie detection

```
Input: $v \leftarrow$ value preferred by this replica
repeat                                                  // iterate through rounds
    $p \leftarrow \langle v, \textbf{random}() \rangle$                          // prioritized proposal
    $(P, \_) \leftarrow \textbf{tcast}(\{p\})$                    // propagate our proposal
    $(E, P') \leftarrow \textbf{tcast}(P)$                     // propagate existent sets
    $(C, U) \leftarrow \textbf{tcast}(P')$                    // propagate common sets
    $v \leftarrow \textbf{anyBest}(C).\textbf{value}$                  // next candidate value
    if $\textbf{uniqueBest}(E) = \textbf{anyBest}(U)$ then        // detect and
        $\textbf{deliver}(v)$                                   // deliver decision
```

While the tiebreaking approach keeps the algorithm simple while allowing priorities to be small, we must adjust the liveness analysis of Section 4.1.3 – and in particular the per-round success probability – since not all of the bits of the resulting priorities are random and hence independent of the network adversary's scheduling decisions. Appendix A.2 below discusses the adjustments necessary in this case.

Detecting ties: A third approach to handling ties is to detect them and avoid deciding when a tie for best proposal has occurred. This approach slightly increases the algorithm's complexity, but may be useful in implementations wishing to keep priorities as small as possible without adding even a few bits for tiebreaking. This tradeoff may be worthwhile in hardware-supported implementations, for example.

Algorithm 5 shows Algorithm 1 adapted to detect ties. The one difference is that Algorithm 5 uses two separate best-proposal-selection functions, **anyBest** and **uniqueBest**, instead of the single **best** function used in Algorithm 1. The function **anyBest**(P) selects *any* proposal from set P that has maximum priority among all those in P, breaking ties arbitrarily. Thus, **anyBest**(P) always returns some valid proposal provided that P is non-empty, which is always the case in Algorithm 5. The function **uniqueBest**(P) returns the highest-priority proposal from P provided there is *only* one highest-priority proposal, and otherwise (on a tie) returns a distinguished **nil** value distinct from any valid proposal.

In Algorithm 5, the next candidate value $v \leftarrow$ **anyBest**(C) always comes from some valid proposal even if there is a tie. If there is a tie for best in any node's common (C) set, and hence ambiguity about which best proposal a node might choose, then all of these tied proposals must also appear in *every* node's existent (E) set. In this case, **uniqueBest**(E) will always return **nil** while **anyBest**(U) always returns a non-**nil** proposal, causing consensus detection to fail on every node and hence safely deferring to the next round.

To implement tie detection in the concrete QuePaxa protocol, using constant-size summaries in place of the proposal sets above, we modify the interval summary register (ISR) definition (Section 4.2.2) to accept and record tuples of the form $\langle$**priority**, **value**, **conflict**$\rangle$ instead of simple binary integers. The **conflict** element is simply a boolean flag that is always **false** in any individual (non-aggregated) proposal. The ISR's summarization operator **aggregate** combines two proposal tuples by choosing the one with the highest **priority** element, provided their priorities are different. Upon encountering two distinct proposals tied for the same priority, **aggregate** returns a tuple whose **priority** is that priority, whose **value** comes from either of the tied proposals arbitrarily (as in **anyBest** above), and whose **conflict** flag is **true** to indicate the conflict.

After this modification to the recorders' ISR logic, the only modification required to the concrete proposer logic (Algorithm 4) is to the decision condition in the $s \bmod 4 = 2$ case. After checking that p is identical to the best of all replies in R, we must also check that the **conflict** flag in p is false, and decide only if p is non-conflicted.

As with the tiebreaking approach, this detection approach also requires an adjustment to the decision success probability calculations as described next.

A.2 Decision success probability adjustment:

Whenever the probability of a tie for best proposal is non-negligible, we must adjust the simplistic decision success probability calculations in Section 4.1.3 to account for ties.

A straightforward approach is to use the union bound as a conservative upper bound on the chance of either of two "bad events" occurring: (a) a tie for highest priority, and (b) the otherwise-unique highest-priority proposal *not* appearing in the universal set (U_i) of a given replica i in question. If the chance of a priority tie is p_t, for example, then the chance of a given replica i not deciding in a round is upper bounded by $p_t + f/n$. Thus, the chance of i deciding is at least $1 - p_t - f/n$. With this approach, the algorithm retains a reasonably-high constant probability of success provided that priorities have at least around $\log_2 n$ bits of entropy.

Single-bit randomness: In certain implementations of QuePaxa it may be desirable to keep the size of random priorities to an absolute minimum, *e.g.*, just one random bit. We can accomplish this goal, while maintaining a reasonable (though not ideal) constant success priority per round, by choosing that random bit from a suitably-biased random distribution.

Suppose for example that each replica chooses priority 2 with probability $1/n$ and chooses priority 1 otherwise, *i.e.,* with probability $1 - 1/n$. (Priority 0 is unused and priority 3 is reserved for the fast-path leader, if any.) Then the desirable event of a given replica i deciding occurs if exactly one replica chooses priority 2 and that replica's proposal appears in i's universal set U_i. The event of a particular replica j choosing the *one and only* proposal of priority 2 occurs with probability $\frac{1}{n}(1 - \frac{1}{n})^{n-1}$. Further, there are at least $n - f > n/2$ such replicas in U_i leading to this desirable outcome. This yields an overall success probability of at least $\frac{1}{2}(1 - \frac{1}{n})^{n-1}$ per round, which is always greater than $\frac{1}{2e} \approx 0.18$ for all $n \geq 1$.

Leader-based rounds: Note that when operating under asynchronous network conditions, the above worst-case success probabilities are guaranteed only in leaderless, fully-asynchronous rounds (*i.e.,* round 2 or greater in any SMR decision slot). In leader-based rounds (the first round of a slot), a strong network adversary with full power to schedule message delivery can prevent this first round from ever succeeding. Thus, the worst-case success probability for this first leader-based round is zero, and the protocol relies on subsequent leaderless rounds to ensure liveness under asynchrony.

A.3 Choosing priorities via common coins

Most asynchronous consensus protocols in practice depend on common (shared), rather than private (local) random coins. It is therefore slightly counter to conventional wisdom that QuePaxa is simpler and more efficient using private coins. In brief, this is because QuePaxa uses randomness to choose proposal priorities, and not to choose either fallback agreement values or unique leaders as in most prior asynchronous protocols. This section briefly discusses the main precedents and reasons for using common coins in consensus protocols and how these precedents relate to QuePaxa. Finally, we briefly discuss common coin implementation issues.

Precedent: randomized fallback values Ben-Or's classic algorithm for asynchronous binary consensus [14] uses randomness to choose actual values for the replicas to agree on, as a "fallback" if they don't agree on their initial inputs. This algorithm can use either private or common coins [5].

Ben-Or's original private-coin version [14] tolerates a strong, non-content-oblivious adversary, but requires a number of rounds exponential in the number of replicas, and is therefore impractical. In essence, the replicas must wait for a rare "lucky" round in which their private coin tosses are sufficiently imbalanced, towards either mostly ones or mostly zeros. The law of large numbers makes such "lucky" (imbalanced) rounds exponentially rare in the number of replicas.

Using a common coin instead of private coins to choose fallback agreement values [5] eliminates the exponential-time problem, because all replicas agree by design on the fallback value for each round. The adversary must then be content-oblivious (like QuePaxa), however. A non-content-oblivious adversary can break the protocol by running just one replica far enough to reveal the (one and only) common coin value for the round, then scheduling the rest of the replicas based on this knowledge [4]. Rabia [66] uses this weaker, but practical, common-coin version of Ben-Or as a building block.

This approach of randomly choosing fallback consensus values does not readily generalize from binary to multi-valued consensus, unfortunately, because with more than one bit of "value space" it becomes nontrivial to ensure *validity*: the important property that the value the replicas agree on is a value that *some* replica actually proposed. Naïvely extending Ben-Or to multi-valued consensus using a common coin, for example, would yield a protocol that often randomly chooses a fallback value that no replica actually proposed. A slightly-less-naïve band-aid solution might be to check whether a randomly-chosen fallback value is valid (*i.e.,* to check whether some replica actually proposed it). We might plausibly perform such a check using binary consensus. This approach, however, in turn raises the spectre of an exponential-time search through a large agreement-value space.

Precedent: randomized leader election Because of the above limitations, most recent asynchronous consensus protocols use common coins not for choosing fallback agreement values but for choosing leaders: *i.e.,* a particular unique replica responsible for performing a critical (typically final) step. For example, ACE [80] runs n concurrent instances of a partially-synchronous consensus protocol, then uses a common coin to choose one of those n executions to "use" – crucially, only *after* any network adversary has had to schedule and run a majority of those concurrent instances to completion. VABA [2] uses a common coin in similar fashion as a final step.

QuePaxa could use common coins similarly, by assigning one unique replica's proposal in a round a designated higher priority (*e.g.*, 2), while assigning all other replicas' proposals some identical lower priority (*e.g.*, 1). The result is a protocol nearly equivalent to the one described above using a single biased bit of private randomness per replica. By design, the common-coin version eliminates the undesirable chance that either two or more, or zero, replicas choose the high priority. The resulting protocol may therefore be close-to-optimally efficient, provided that no replicas have failed.

The cost of using common coins for leader election, however, manifests when some (up to f) of the replicas have failed. If exactly f replicas have failed, for example, then the common coin used for leader election will choose a failed leader in each round with probability f/n. These "unlucky" rounds will always fail to reach consensus, because all of the proposals of non-failed nodes have the same undifferentiated priority (1), *i.e.,* always yielding a tie for highest priority. Asynchronous common-coin protocols like ACE [80] and VABA [2] similarly fail to decide in such "unlucky" rounds, and must simply retry until a lucky round materializes. When using private coins to choose priorities, in contrast, QuePaxa can reach consensus in *every* round even in this "f failures" scenario, because the $n-f$ live nodes always produce *some* highest-priority proposal (unless a tie for best occurs, of course).

A more sophisticated common-coin QuePaxa could mitigate the "unlucky round" effect by using k common random "leader elecctions" per round to choose k-bit priorities, such that each bit $b \in \{0, \dots, k-1\}$ is set in node i's proposal priority if node i is the unique "winner" of random election b. If the "first-choice" winner of bit $k-1$ is one of the f offline nodes, then the "next-choice" winner of bit $k-2$ might still enable the round to succeed if that winner is online, and so on. This multi-bit common-coin protocol is likely just as efficient as the private-coin protocol: perhaps slightly more efficient, since it eliminates the possibility of ties. This common-coin version of QuePaxa seems more complex, however, and perhaps not worth the trouble provided the private-coin protocol has enough bits to limit the frequency of random ties.

Implementing common coins Relying on common coins in asynchronous Byzantine consensus protocols tends to introduce significant challenges and complexity, but is much simpler in asynchronous crash-stop protocols like QuePaxa.

Byzantine consensus protocols usually require common coins to be built using techniques such as Shamir secret sharing [77], because it is usually important for the values of the common coins to be hidden from all participants (especially the Byzantine replicas) until a threshold number of replicas has reached a certain stage. This secret-sharing dependency in turn requires some form of distributed key generation (DKG) for setup [28, 35]. This effective dependency on DKG further creates a "chicken-and-egg" problem for Byzantine consensus, in that solving DKG requires solving consensus as part of the process, to agree on a particular choice of random secrets.

Implementing common coins in crash-stop protocols like QuePaxa is easy, in contrast. We can choose a secret random seed known to all replicas at group setup time. We then use a keyed hash function [13] with the seed as key, and round number as input, to produce fresh common coins for each round. Since crash-fault protocols assume none of the replicas are compromised and the adversary is only in the network, we can simply trust the replicas not to leak the common seed or resulting common coins to the network adversary. In summary, implementing common-coin variants of QuePaxa is feasible and straightforward, although the benefits appear small and may not be worth the marginal complexity cost.

B Abstract QuePaxa Correctness Proofs

In this section, we prove the correctness of abstract QuePaxa described in Algorithm 1. These proofs expand the correctness sketches provided in Section 4.1.3.

Definition B.1 (Proposal)**.** We define a *proposal* as a tuple $\langle \pi_i, i, v_i \rangle$, where i denotes the proposer replica's identifier, v_i denotes the proposed value and π_i represents the priority.

This definition adds the replica identifier i to the proposal tuple used in Algorithm 1, to simplify formal reasoning, and in practice for tiebreaking in case there is a non-negligible chance of ties (see Appendix A for details). By this definition, proposals from different replicas are always distinct even if their priorities and values are identical.

In the formal reasoning below, however, we will assume for simplicity that priorities never tie. That is, for any two independently-chosen random priorities π and π', the probability that $\pi = \pi'$ is zero. This is automatically the case if priorities are *real* numbers chosen uniformly at random from the interval between 0 and 1, for example. The example of using real numbers is only a formal mathematical abstraction; for practical discussion, refer to Appendix A on priorities.

Definition B.2 (Time advancement)**.** In the abstract network model that Algorithm 1 builds on, time advances in consecutive integer *steps*. At each time step, each live replica i invokes the *threshold broadcast* primitive, or **tcast**, exactly once. This invocation of **tcast** completes in exactly one time step, unless replica i fails during that time step. Invocation of **tcast** is the *only* way in which time advances in this abstract network model, so all live replicas by definition run **tcast** simultaneously in each time step.

Definition B.3 (Threshold broadcast)**.** At each time step, each live replica i invokes **tcast**(P_i) with some set of proposals P_i that i wishes to disseminate to the other replicas. After one step, if replica i is still live, **tcast** returns to i a pair of proposal sets (R_i, B_i). The **tcast** primitive satisfies two properties, presented informally in Section 4.1.1:

- **Property T1.** R_i includes the inputs from a majority of replicas, *i.e.,* $|R_i| > n/2$, and $\forall P_j \in R_i$, $\exists$ replica j s.t. j proposed P_j in that step.
- **Property T2.** B_i is the proposal set input of some replica (*i.e.,* P_j for some j) that **tcast** has successfully broadcast to all non-faulty replicas during this step. In other words, $\exists$ replica j s.t. $B_i = P_j$, j invoked **tcast**(P_j) in this time step, and $\forall$ live replicas k, $B_i \subseteq R_k$, where (B_k, R_k) is the pair returned from **tcast** on replica k.

Lemma B.4 (Set cardinalities)**.** *In every consensus round, the sets P_i, P_i', E_i, C_i, and U_i that are computed in Algorithm 1 each have cardinality greater than $n/2$.*

Proof. By induction over consensus rounds. In the base case, the initial preferred value v_i of each replica i is a well-defined input to the algorithm.

In each round, assuming that v_i is well-defined, the replica's proposal p_i is likewise well-defined, so the input to the first **tcast** invocation is a well-defined proposal set of cardinality one. By **tcast** property T1, the received proposal set P returned from this **tcast** invocation contains the **tcast** inputs to a majority of replicas in this step. Each of these **tcast** inputs has cardinality one and contains a distinct proposal, due both to the inclusion of replica identifier i and to the assumption that priorities never tie, as discussed above. The set P returned from the first **tcast** thus has cardinality greater than $n/2$.

The proposal sets R_i and B_i returned by any **tcast** invocation on replica i are at least as large as the input P_j to **tcast** on *some* replica j in the same time step. As a result, the fact that $\forall$ replicas $j, |P_j| > n/2$ in Algorithm 1 implies that $\forall$ replicas $i, |E_i| > n/2$ and $|P_i'| > n/2$. This in turn implies that $\forall i, |C_i| > n/2$ and $|U_i| > n/2$, As a result, the statement $v \leftarrow$ **best**(C_i).**value** chooses a next candidate value from a non-empty common set C_i, thus ensuring that v is well-defined in the next consensus round as well. □

Lemma B.5 (Set relationship)**.** *The QuePaxa protocol achieves the property $\forall$ live replicas $i, j, U_i \subseteq C_j \subseteq E_i$ (Section 4.1.2).*

Proof. The QuePaxa protocol executes three **tcast** steps. The first **tcast** gives replica i a proposal set P_i containing the proposals of any majority of replicas. The second **tcast** takes P_i as input, and gives replica i a proposal set P_i' that, by **tcast** property T2, is guaranteed to appear in the *existent* sets E_j returned to any replica j. Finally, the third **tcast** uses P_i' as input, and returns to i a *common* proposal set C_i and an *universal* proposal set U_i. By **tcast** property T2, U_i is guaranteed to appear in the sets C_j returned to all replicas j (hence $U_i \subseteq C_j$, $\forall$ replicas i, j). By **tcast** property T1, $\forall P_j'' \subseteq C_i$, $\exists$ replica j that proposed $P_j' = P_j''$. Since the input parameter P_j' used in the third **tcast** call at replica j is guaranteed to be in the E_k set received by any replica k in the second **tcast** call, we have $C_i \subseteq E_k$, $\forall$ replicas i, k. Hence we have $\forall$ replicas $i, j, U_i \subseteq C_j \subseteq E_i$.

Furthermore, we can derive that $\forall$ replicas i, j, if **best**(U_i) = **best**(E_i), then **best**(U_i) = **best**(C_j) = **best**(E_i), due to the assumption above that independent priorities never tie. □

Lemma B.6 (Validity)**.** *Any value decided by QuePaxa is a value proposed by some replica.*

Proof. Algorithm 1 has a single decision condition: **best**(E) = **best**(U). When this condition is satisfied, QuePaxa delivers **best**(C).**value** as the consensus decision to the upper layer. We show that the set C contains only proposed values, hence any decided value is a proposed value.

By Lemma B.5, $C \subseteq E$ in round r. We show that $\forall p \in E$, p is a value proposed by some replica. From the second **tcast** call of round r, by **tcast** property T1, there is a set S containing a majority of replicas such that $\forall j \in S$, $P_j \subseteq E$. Each replica j sets the input argument $P = P_j$ for the second **tcast** to be equal to the P set returned in the first **tcast** call of the same round r. This P set consists of the values p received from a majority of the replicas as input to the first **tcast** call, which each replica i sets as $p = \langle \pi_i, i, v_i \rangle$. If $r = 0$, then v_i is proposed by some replica, hence the validity holds. If $r > 0$, then we can inductively show that p is derived from a value that was initially proposed by some replica in round 0. Hence, the validity property holds. □

Lemma B.7 (Agreement)**.** *No two replicas decide differently.*

Proof. Assume by contradiction that two replicas i and j decide different values $v_i \neq v_j$ for the same slot. Without loss of generality, assume that i decides first v_i in round r_i, and j decides second the value v_j in round r_j, s.t $r_j \geq r_i$.

Case 1: $r_j = r_i$ (both replicas decide in the same round and phase).

Replica i decides $v_i = \textbf{best}(C_i).\textbf{value}$, if $\textbf{best}(U_i) = \textbf{best}(E_i)$. By Lemma B.5, we have $U_i \subseteq C_i \subseteq E_i$, and we can derive that $\textbf{best}(U_i) = \textbf{best}(E_i) = \textbf{best}(C_i) = \langle \pi_i, i, v_i \rangle$. Using the same lemma for replica j's decision, we have $U_i \subseteq C_j \subseteq E_i$, and we can also derive $\textbf{best}(U_i) = \textbf{best}(E_i) = \textbf{best}(C_j) = \langle \pi_j, j, v_j \rangle$. $\textbf{best}(U_i) = \langle \pi_i, i, v_i \rangle = \langle \pi_j, j, v_j \rangle$ is a contradiction, given our assumption that i and j are different replicas and priorities are unique per proposal. Hence i and j must be the same replica and $v_i = v_j$.

Case 2: $r_j > r_i$ (replica j decides in a later round than replica i).

When replica i decides v_i, by Lemma B.5 ($U_i \subseteq C_k \subseteq E_i$), any replica k advancing to round r_i+1 chooses $\textbf{best}(C_k).\textbf{value} = v_i$ and proposes $\langle \pi_k, k, v_i \rangle$ in the first **tcast** of round $r_i + 1$. Since all the sets in the three **tcast** steps are derived only from proposals containing v_i as value, we can inductively show that in any round $r_k > r_i$, any replica k chooses v_i in its proposal. Thus all the sets in any round $r > r_i$ contain only proposals with v_i as value, and thus the only decision possible is $\textbf{best}(C).\textbf{value} = v_i$. This contradicts our assumption that some replica j decides $v_j \neq v_i$ in round $r_j > r_i$. □

Lemma B.8 (Integrity). *No replica decides twice.*

Proof. This property may be trivially satisfied by using a boolean flag decided per replica, initialized to false. A replica decides only if its decided flag is false, and sets it to true once it has decided. Due to its implementation triviality, we have not included the flag in the abstract QuePaxa algorithm. □

Lemma B.9 (Liveness). *Each non-faulty replica eventually decides with probability 1. The expected termination time per slot is less than two asynchronous (leaderless) rounds.*

Proof. Replica i decides in round r if the set U returned from i's third **tcast** call in round r contains the round's unique globally-best prioritized proposal v. Then i will necessarily choose v and deliver it.

We compute the probability that the desirable event ($v \in U$) occurs. As stated in Section 3.2, we assume a content-oblivious network whose choices of sets does not depend on the content of messages or their priority values. Together with Lemma B.4, this means that replica i's U set contains more than $n/2$ proposals, each of which is equally likely to be one of the n proposed values in round r. Thus, $\Pr[v \in U] > 1/2$.

Hence we see that with probability greater than 1/2, each replica decides in a given round. We conclude that each replica decides with probability 1 eventually, and delivers the decided value to the upper layer in less than two asynchronous rounds in expectation. □

Note that the above analysis ignores the first round in an SMR slot if that first round is leader-based, since a leader-based round has a worst-case success probability of zero under asynchronous network conditions.

C Concrete QuePaxa Correctness Proofs

We prove the correctness of the concrete QuePaxa protocol (Algorithm 4 and Algorithm 2) by showing that concrete QuePaxa correctly simulates abstract QuePaxa (Algorithm 1). We first show that concrete QuePaxa computes functionally-equivalent state as the abstract QuePaxa in each step. We then show that each proposer in concrete QuePaxa eventually reaches the next step until it decides. These two statements imply that the correctness proofs for abstract QuePaxa (Appendix B) also apply to concrete QuePaxa.

Definitions. Throughout this section we use the proposal definitions and **tcast** properties defined in Appendix B.

Definition C.1 (Recorder reply). We define as *recorder reply* a tuple $r = \langle s, f, a, j \rangle$, where s is the current step of the recorder's ISR, f is the first proposal the ISR received in step s, a is the ISR aggregate in step $s - 1$, and j is the recorder identity.

Lemma C.2 (Recorder reply step). *In concrete QuePaxa, a proposer i invoking* **record**(s, p) *at recorder j cannot receive a reply $r_j = \langle s'_j, f'_j, a'_j, j \rangle$ from j such that $s'_j < s$.*

Proof. Suppose proposer i receives a reply from recorder j with $s'_j < s$. This means i is ahead of recorder j that sent the respective reply. We show by contradiction that $s'_j \nless s$.

Assuming that $s'_j < s$, then recorder j's ISR is in step $s'_j < s$ when it replies to i. However, when j receives i's invocation **record**(s, p), it first updates its internal step to s. Because ISR steps only increase, it cannot be that at a later time, j's step decreases to $s'_j < s$. This contradicts our assumption. □

Lemma C.3 (Proposer catch-up correctness). *In concrete QuePaxa, a proposer catching up to any step is initialized to a correct $\langle s, p \rangle$ state. In other words, there exists a valid execution of concrete QuePaxa where the proposer does not execute the catch-up procedure and reaches the same $\langle s, p \rangle$ state.*

Proof. Proposer i in step s executes the catch-up procedure when i observes that it lags behind one or more recorders by inspecting the reply set R. Let recorder j trigger the catch-up procedure at proposer i when it replies $r_j = \langle s'_j, f'_j, a'_j, j \rangle \in R$ with $s'_j > s$. According to the catch-up procedure, the proposer state becomes $\langle s'_j, f'_j \rangle$.

Recorder j initialized its ISR to s'_j and its $F[s'_j]$ to f'_j when receiving from a proposer the proposal f'_j tagged with step s'_j. Let this be proposer k. W.l.o.g., assume that proposer k reached state $\langle s = s'_j, p = f'_j \rangle$ without executing any catch-up procedure, which we can prove by induction. Indeed, ∃ at least one proposer in any step that does not execute a catch-up procedure, *e.g.*, the first proposer that is first to execute that step; if all proposers were to catch-up in a step, this means that $\forall$ proposer $x, \exists$ proposer y s.t. $x.\text{step} < y.\text{step}$, which is impossible.

Thus i catches up to the same state as proposer k, and proposer k did not execute any catch-up statement. Moreover, by our assumptions on the network adversary and the communication model, the message delivery schedule at any proposer is agnostic to the proposer's identity. We conclude that there exists a valid execution for proposer i, *e.g.*, the same execution and message delivery as for proposal k, which enables i to reach the state s'_j, f'_j. □

Lemma C.4 (Phase 0 of concrete QuePaxa)**.** *Each concrete QuePaxa proposer computes in phase 0 of each round proposal* $p = \mathbf{best}(P)$*, where* $P = \{f' \mid \langle s', f', a', j\rangle \in R\}$*, and where R is the reply set. In abstract QuePaxa,* $p = \mathbf{best}(P)$*, where P is the output set P of the first* **tcast** *call of the round.*

Proof. In abstract QuePaxa, consider a replica i that proposes p and broadcasts this proposal through the first **tcast** call. By **tcast** property T1, at the end of the first **tcast** call, the output set P of replica i contains the proposals of a majority of replicas. Because we assume that priorities do not tie, there exists a **best** proposal of set P, which we denote by **best** (P).

In concrete QuePaxa, in phase 0 and step s (with $s \bmod 4 = 0$), consider proposer i proposing p_j to recorder j, where $p_j = \langle \pi_j, i, v\rangle$ (pi_j is the per-recorder priority, all other proposal fields do not change per recorder). The proposer submits (s, p_j) by sending $\mathbf{record}(s, p_j)$. The proposer then waits for a quorum (majority) of replies $r_j = \langle s'_j, f'_j, a'_j, j\rangle$ from recorders, which forms its reply set R.

Each live recorder j receives such proposals (s_i, p_i) from potentially multiple proposers i. Consider a recorder with ISR current step S. When $s_i > S$, the recorder advances $S = s_i$ and stores $F[S] = p_i$, *i.e.,* F[S] is the first proposal the recorder received at step S. The recorder replies with $\langle S, F[S], A[S-1], j\rangle$, where $A[S-1]$ is an aggregate that proposer do not use in phase 0.

Case 1: All replies $r_j \in R$ have the step $s'_j = s$, the proposer's current step. In this case, f'_j from all replies in R represent a majority of proposals from the same step, and $p \leftarrow \mathbf{best}(f'_j), \forall r_j \in R$ represents the best of a majority of proposals. Thus, p is the same as $\mathbf{best}(P)$ in abstract QuePaxa.

Case 2: At least a proposal $r_j \in R$ has the step $s'_j \neq s$. We know from Lemma C.2 that $s'_j \not< s$. When $s'_j > s$, then the proposer has fallen behind some recorder and executes the catch-up sequence. Lemma C.3 proves that the proposer catches up correctly to the beginning of step s'_j and resumes execution in step s'_j. □

Lemma C.5 (Spread/gather relationship to **tcast** property T2)**.** *In concrete QuePaxa, consider a proposer i that successfully spreads proposal p in step s, and all live proposers executing gather in step* $s+1$*. This spread/gather sequence implements* **tcast** *property T2 in abstract QuePaxa,* i.e., $p \in B$.

Proof. We first show that in step s at least a majority of recorders accumulate proposal p. Then we show that in step $s+1$ all proposers gather in the set R all proposals accumulated at a majority of recorders in step s. Thus, in step $s+1$ all recorders gather p, which corresponds to $p \in B$ in **tcast**.

Let proposer i invoke $\mathbf{record}(s, p)$ at all recorders, then waits to collect in R a majority of recorder replies, where $r_j = \langle s'_j, f'_j, a'_j, j\rangle$ is the reply from recorder j. There are three cases based on the current steps of replying recorders.

Case 1: All replies have the recorder ISR step equal the proposer's step s, *i.e.,* $\forall r_j = (s'_j, f'_j, a'_j, j) \in R$, $s'_j = s$. Let $A = \{$ recorder $j | j$ sent reply $r_j \in R\}$. Because $|R| > n/2$, we have that $|A| > n/2$. Moreover, $\forall$ recorders $j \in A$, the ISR of j aggregated in $A_j[s'_j]$ all proposals it received in step s, including proposer i's value p.

Case 2: At least one reply is from step $s'_j > s$ (by Lemma C.2, $s_j \not< s$). Such a recorder j is ahead of the proposer and by ISR construction discards proposals from earlier steps, such as p. Thus, recorder j does not accumulate p and, in fact, does not change their ISR state based on p. In this case, proposer i cannot be sure that a majority of recorders aggregated p. Thus proposer i is unsuccessful in spreading p, executes the catch-up sequence (Lemma C.3) and resumes execution from step s'_j.

Now consider a proposer k in step $s+1$ executing a gather operation by invoking $\mathbf{record}(s, *)$, where $*$ is a wildcard for any proposal that does not affect the gather operation. Each contacted recorder l in step $s_l < s+1$ advances its step to $s_l = s{+}1$. Then recorder l replies with $r_l = (s'_l, f'_l, A'_l[s_l{-}1], l)$, and proposer k gathers a majority of such replies in set R. Similar to case 1 above, if all replies have $s'_l = s+1$, then proposer k gathers in R the $A'_l[s]$ sets of a majority of recorders, Let $B = \{$ recorder $l | l$ sent reply $r_l \in R\}$. Because $|R| > n/2$, we have that $|B| > n/2$. If proposer k finds itself in case 2 above, it simply catches up to a higher step.

The first proposer executing gather in step $s+1$ makes the recorder-aggregated sets $A_j[s]$ in step s final. Consider the first proposer k in step $s+1$ executing a gather operation. Each contacted recorder j ISR first advances its step to $s+1$, and stops accumulating values for step s, *i.e.,* $A_j[s]$ is final. Indeed, no recorder can be in step $s{+}1$ upon receiving proposer i's message because recorders only advance their step when a proposer in a higher step contacts them, and i is the first proposer to execute gather in step $s{+}1$, thus no other proposer besides i contacted the recorders in step $s+1$.

Now consider proposer l that fell behind in step s attempting to spread a proposal. l waits for a majority of replies from recorders, and by quorum intersection receives at least one reply r_j with $s_j = s+1$. As a result, proposer l learns that its proposal was not spread to a majority of recorders, and executes the catch-up procedure. Thus, a proposer executing gather in step $s + 1$ cannot miss any proposals that were successfully spread in step s.

We conclude that A and B intersect in at least one recorder, thus all proposers executing step $s + 1$ successfully (either directly or via catch-up) gather proposal p. Hence proposal p in concrete QuePaxa fulfills $p \in B$ in **tcast**. □

Lemma C.6 (Gather relationship to **tcast** property T1)**.** *In concrete QuePaxa, proposer i executing gather in step* $s + 1$ *after spread in step s compute* $p = \mathbf{best}(\{A[s]_j \mid \langle j, s + 1, f'_j, A[s]_j\rangle \in R_i\})$*, where* R_i *is the reply set. In abstract QuePaxa,* $p = \mathbf{best}(R)$*, where R is the output set of a* **tcast** *operation performed at step s.*

Proof. From Lemma C.5 we infer that, $\forall$ proposer i that successfully spread p_i in step s, all proposers k have $p_i \in \{A[s]_j \mid \langle j, s + 1, f'_j, A[s]_j\rangle \in R_i\}$ by the end of step $s + 1$. Thus $\{A[s]_j \mid \langle j, s + 1, f'_j, A[s]_j\rangle \in R_i\} = R$, where R is the first output parameter of **tcast** in step s, and $\mathbf{best}(\{A[s]_j \mid \langle j, s + 1, f'_j, A[s]_j\rangle \in R_i\}) = best(R)$. □

Lemma C.7. *Concrete QuePaxa computes in phases 1 and 2 of each round: (i) proposal* $a = \mathbf{best}(E)$*, where* $E = \{a'_j \mid \langle j, s'_j, f'_j, a'_j\rangle \in R\}$ *and R is the reply set of phase 2, which in abstract QuePaxa is* $a = \mathbf{best}(E)$*, where E is the first output of the second* **tcast** *call of the round, and of the second* **tcast** *call of the round; and (ii) proposal* $p = \mathbf{best}(P')$*, where* P' *is the second output set of the second* **tcast** *call.*

Proof. In concrete QuePaxa phase 1, each proposer i in step $s \bmod 4 = 1$ selects proposal p = **best** of f' from all replies in set R, and spreads it to all recorders. By Lemma C.4, $p = \mathbf{best}(P)$, where P is the first output set of the first **tcast** call in abstract QuePaxa. By Lemma C.5, $p = \mathbf{best}(P')$ in abstract QuePaxa.

Using Lemma C.6, all proposers completing phase 2 in step $s + 1 \bmod 4 = 2$ gather **best** of $a'_j = A_j[s + 1], \forall j \in R$, which corresponds to the $\mathbf{best}(E)$ in abstract QuePaxa. □

Lemma C.8. *Concrete QuePaxa computes in phases 2 and 3 of each round: (i) proposal* $a = \mathbf{best}(C)$*, where* $C = \{a'_j \mid \langle j, s'_j, f'_j, a'_j\rangle \in R\}$ *and R is the reply set of phase 3, which in abstract QuePaxa is* $a = \mathbf{best}(C)$*, where C is the first output of the third* **tcast** *call of the round; and (ii) proposal* $p = \mathbf{best}(U)$*, where U is the second output set of the second* **tcast** *call.*

Proof. In concrete QuePaxa phase 2, by Lemma C.7, each proposer i in step $s \bmod 4 = 2$ proposes $p = \mathbf{best}(P')$ in abstract QuePaxa. Using Lemma C.5, by the end of phase 3, $p = \mathbf{best}(U)$ in abstract QuePaxa. Using Lemma C.6, all proposers completing phase 3 in step $s + 1 \bmod 4 = 3$ gather **best** of $a' = A_j[s + 1], \forall j \in R$, which corresponds to the **best** (C) proposal in abstract QuePaxa. □

Lemma C.9 (Asynchronous decision path)**.** *The asynchronous path decision in concrete QuePaxa, returning* $p.\mathbf{value}$ *in phase 2, when* $p = \mathbf{best}\{a'_j | \langle j, s'_j, f'_j, a'_j\rangle \in R\}$*, where R is the reply set, represents the same decision as* $\mathbf{best}(C).\mathbf{value}$ *when* $\mathbf{best}(E) = \mathbf{best}(U)$ *after the third* **tcast** *call of abstract QuePaxa.*

Proof. Using Lemma C.7, at the end of phase 2 in concrete QuePaxa, $p = \mathbf{best}\{a'_j|\langle j, s'_j, f'_j, a'_j\rangle \in R\} = \mathbf{best}(E)$ in abstract QuePaxa. We now need to show that $p = \mathbf{best}(U)$ in abstract QuePaxa.

Using the same same Lemma C.7, $p = \mathbf{best}(P')$ in abstract QuePaxa. Moreover, using Lemma C.5, a proposer checks the condition p = **best** of a' from all replies in the reply set R, at the end of phase 2. At the end of phase 2, using similar reasoning as in Lemma C.5, p is accumulated in the ISR registers of a majority of recorders. Thus, although p is not yet a universal proposal, already at the end of phase 2 it is inevitable that $p \in U$ by the end of phase 3, as shown in Lemma C.8. Because p was selected to be the best proposal the proposer knew of at the end of phase 1, the proposer knows at the end of phase 2 that $p = \mathbf{best}(U)$ that the proposer knows of. Thus $p = \mathbf{best}(U) = \mathbf{best}(E)$ and the asynchronous decision condition in concrete QuePaxa is the same as the decision condition in abstract QuePaxa.

When $\mathbf{best}(E) = \mathbf{best}(U)$, then $\mathbf{best}(C) = \mathbf{best}(E) = \mathbf{best}(U)$ (Lemma B.5). We conclude that, when the decision condition is met, $p.\mathbf{value}$ in concrete QuePaxa equals $\mathbf{best}(C).\mathbf{value}$ in abstract QuePaxa. □

Lemma C.10 (Fast-path decision)**.** *The fast-path decision condition in step 0 in leader-based concrete QuePaxa is correct.*

Proof. In concrete QuePaxa, proposers propose in phase 0 (step $s \bmod 4 = 0$) of a round. Leader-based concrete QuePaxa uses a single deterministic leader only in the first step, which is step 4, of the execution. All subsequent steps $s \bmod 4 = 0$, with $s > 4$, do not have a designated leader. Each proposers assigns a priority to its proposal, and the leader's proposal is the only proposal receiving the highest priority H in step 0. The fast-path decision condition states that, if a proposer executing step $s = 4$ gathers in its reply set $\forall r_j = \langle s'_j, f'_j, a'_j, j\rangle \in R$, $f'_j.\mathbf{priority} = H$ and $|R| > n/2$, then the proposer decides $f'_j.\mathbf{value}$. We show that this decision condition is equivalent with Lemma C.9.

Let the leader be proposer l that proposes p_l with $p_l.\mathbf{priority} = H$, and let proposer i satisfy the fast-path condition and decide v_i in step $s_i = 4$. Because proposer i has $\forall r_j = \langle s'_j, f'_j, a'_j, j\rangle \in R_i$ in step 4, $f'_j.\mathbf{priority} = H$ and $|R_i| > n/2$, more than $n/2$ recorders store in their ISR registers a proposal $F[4]$ with $F[4].\mathbf{priority} = H$. Because only the leader's proposal has priority H, it must be that these recorders ISR registers store the leader's proposal p_l in $F[4]$.

We claim that each ISR writes $F[4]$ only once. Indeed, the first proposal p tagged with $s = 4$ that reaches a recorder's ISR triggers $F[4] \leftarrow s$ because the ISR's step $S < s$, then advances $S \longleftarrow s$. As ISR steps never decrease, the ISR never overwrites $F[4]$. Hence, the leader's proposal p_l is recorded in $F[4]$ at more than $n/2$ recorders, and it is never overwritten.

$\forall$ proposer j executing step 4, proposer j computes a reply set R_j s.t. $|R_j| > n/2$. R_i and R_j must intersect in at least one recorder, hence $\exists$ recorder k s.t. reply $r_k = \langle s'_k, f'_k, a'_k, k\rangle \in R_j$ has f'_k.**priority** $= H$. There are two cases.

Case 1: Proposer j decides some p.**value** on the fast path. In this case, it must be that p.**priority** $= H$, and because only p_l has p_l.**priority** $= H$, j decides p_l.**value**.

Case 2: Proposer j does not decide on the fast path. Then j chooses p s.t. p is **best** in R_j. Because the leader's proposal $p_l \in R_j$, j chooses and spreads p_l in the next step. All proposers not deciding in step 4 choose and spread p_l for the next step $s + 1 \bmod 4 = 1$. In other words, p_l is the only proposal broadcast. We show that j decides p_l.**value** in the same round.

By Lemma C.7, Lemma C.7 and **tcast** properties T1 and T2, all output sets E, C, U must contain p_l. Thus, by Lemma C.9, all live proposers decide p_l.**value**. □

Lemma C.11 (Liveness)**.** *In concrete QuePaxa, each live proposer eventually decides with probability 1.*

Proof. We first show that $\forall$ live proposer j, j always advances to a higher step unless it decides. Let proposer j execute step s. $\forall$ steps s, j waits for the replies of a majority of recorders, which it gathers in set R. Because a majority of recorders never crash, j eventually receives a majority of replies. There are two cases possible.

Case 1: All replies $r \in R$ are tagged with step s. Then j increments its step counter $s \longleftarrow s + 1$ or stops execution by deciding.

Case 2: At least one reply in $r' \in R$ is tagged with step $s' \neq s$. We know from Lemma C.2 that $s' \nless s$. When $s' > s$, then the proposer has fallen behind some recorder and executes the catch-up sequence, which simply enables j to advance to step s' and adopt the proposal $r'.f'$.

In both these cases, j advances to a higher step until it decides. We now show that every live proposer j eventually decides with probability 1. Proposer j either decides through the fast-path in the first step (step 4) – which under asynchronous conditions reduces to probability 0 – or through the asynchronous path in any step s with $s \bmod 4 = 2$. By Lemma C.9, concrete QuePaxa's asynchronous decision path represents the same decision condition as abstract QuePaxa. Because in abstract QuePaxa every live replica eventually decides with probability 1 Lemma B.9, the same holds for concrete QuePaxa. □

D Promela Models for SPIN Model Checker

To validate QuePaxa's safety, we implemented two Promela models of its basic consensus logic, supporting exhaustive verification of the state space using the SPIN model checker. The first model (Appendix D.1) is simpler and cheaper to verify, while the second model (Appendix D.2) is more realistic but much more costly to verify.

Both implementations enable us to verify only the algorithm's safety, since decision success in any round is probabilistic and the SPIN model checker is not designed to check statistical properties. Nevertheless, the model checker does verify that the models never deadlock or get "stuck" without completing the designated number of logical time steps, and that all of the code is covered (no code is unreachable).

As with any use of model checking, our verification was constrained by the need to limit the problem to a finite state space. As our baseline configuration we focus on two proposers and three recorders, in executions of two full rounds (logical time steps 4–11), with 1-bit random priorities (not counting 0 or H) and 2-bit proposal values. We confirmed manually that any of these parameters may be increased in moderation while leaving the model verifiably safe.

D.1 Promela model with inline recorder logic

Our first Promela model of QuePaxa is designed to minimize state space explosion by modeling proposer/recorder communication as shared-memory interactions rather than by passing messages explicitly between processes. In this model, only the active proposer roles are processes, while the recorder roles are represented by **atomic** code manually inlined into the proposer logic at appropriate points. While drastically simplified, this model still requires the model checker to explore all possible interleavings of the proposer processes (two in the baseline configuation) throughout executions of their non-atomic code. If the algorithm had any safety violations reachable in the finite state space explored, therefore, we would expect the model checker to find them.

We found it feasible to verify this model exhaustively in under three minutes on a 2021 MacBook Pro laptop with 64GB of RAM.

D.2 Promela model with separate recorder processes

Our second and more realistic Promela model of QuePaxa models recorders more realistically as separate processes, and uses message passing for proposer/recorder interaction. Because of the larger number of processes (five instead of two in the baseline configuration) and their more complex interaction, the state space of this model explodes *much* more rapidly than that of the simpler one above.

We found it necessary to use bitstate verification [38] to limit memory consumption in checking this model even for just two rounds. In particular, we verified this model in about 24 hours, on the same 2021 MacBook Pro laptop as the one above.

Because bitstate verification uses a hash table to check whether states have already been explored, false positives due to hash collisions can result in some paths being left unexplored. Thus, our verification of this model was not necessarily fully exhaustive. Nevertheless, since the core consensus logic it represents is identical to that of the simpler model above that was exhaustively verified, we consider it unlikely that bitstate verification missed any safety failures in this case.

E Detailed comparison with Rabia

As another recent randomized consensus protocol addressing crash-stop failures, Rabia [66] represents a particularly important comparison point for QuePaxa. While similar in certain respects, Rabia and QuePaxa embody fundamental differences in goals, assumptions, and design.

Goals If we take widely-deployed partially-synchronous consensus algorithms like Paxos and Raft as a baseline, then Rabia and QuePaxa essentially pursue *opposite* goals on a spectrum of tradeoffs between simplicity and adaptivity. Rabia pursues simplicity through randomization as its primary goal, but specializes in low-latency data-center networks. QuePaxa, in contrast, pursues adaptability to diverse network conditions including LAN, WAN, grey failures [7, 40, 47], and adversarial conditions such as network-based DoS attacks – at some cost in protocol complexity.

Safety and liveness assumptions To ensure safety and liveness, QuePaxa and Rabia both make the standard assumptions that $n \geq 2f + 1$, that messages between correct replicas are eventually delivered, and that replicas are fail-stop (non-Byzantine). Unlike QuePaxa, Rabia requires incoming requests to be timestamped, and relies on correct timestamping for liveness, as we examine further below.

Fast-path assumptions Both Rabia and QuePaxa rely on further assumptions holding in order to achieve their ideal fast-path performance.

QuePaxa's fast path achieves consensus in two network hops (one round-trip), with linear communication cost. QuePaxa can achieve this fast-path performance and efficiency when network delays are stable enough that its hedging schedule normally activates only $O(1)$ proposers per round.

Rabia's fast path operates in three network hops, and always exhibits quadratic communication cost: all replicas broadcast to all others in every round. For its fath path to succeed, Rabia assumes that under the normal data center conditions it targets, "message delay is small compared to the interval between two consecutive requests" [66, §3.2].

Design Both QuePaxa and Rabia are randomized consensus algorithms, but they differ drastically in the way they use randomness to circumvent the FLP impossibility result [27].

As a key component, Rabia builds on Ben-Or's classic algorithm for asynchronous binary consensus [14], made practical using common coins as discussed in Appendix A.3. Binary consensus alone is insufficient to achieve multi-valued consensus or state machine replication (SMR), however. The challenge of "bridging the gap" from binary to multi-valued consensus adds substantial complexity to prior asynchronous consensus protocols such as HoneyBadger [53].

Rabia avoids this complexity by relying on request timestamps and the data center delay assumption above. In essence, Rabia assumes that the low-delay data-center network normally creates a natural multi-valued consensus among replicas as to the "oldest[-timestamped] pending request" [66, §3.2]. The replicas then use Ben-Or binary consensus merely to confirm whether or not this network-based multi-valued consensus has occurred in a particular SMR decision slot.

QuePaxa's design differs fundamentally by using randomness to choose proposal priorities, and not to choose either fallback agreement values as in Ben-Or binary consensus or to choose random leaders as in many other prior protocols. In particular, QuePaxa need not use common coins in order to be practical, and normally does not, as detailed in Appendix A.

E.1 Implications of Rabia's two key assumptions

Rabia's timestamping and data-center delay assumptions have a few subtle but important implications, examined next.

Implications of request timestamping While the Rabia paper does not state exactly how or where timestamps are assigned, practical considerations suggest that they are assigned either by clients themselves or trusted front-end proxies. If the replicas themselves assigned timestamps on a request's arrival, then the same request arriving at different replicas would typically receive different timestamps, causing effective request duplication and defeating the natural network-based consensus on "oldest pending request" that Rabia depends on. Since f replicas may fail, clients or front-ends must be prepared to submit a given request to at least $f + 1$ replicas – eventually if not immediately. It therefore appears that Rabia makes an essential assumption that *some* node apart from the replicas or replicas themselves (*e.g.*, client or front-end proxy) assigns each request a *single* timestamp consistently.

Correct request timestamping is critical to Rabia's liveness, at least in principle. Suppose, for example, an adversarial network scheduler could conspire with a single compromised client capable of injecting a series of requests with *decreasing* (back-dated) timestamps. In this case, each time the replicas nearly reach agreement on some oldest pending request, the adversarial scheduler could simply "discover" (inject) a new, still "older" pending request with which to disrupt consensus, and thereby block Rabia's progress indefinitely. Thus, Rabia's assumption of (correct) request timestamping is essential to Rabia's liveness, not just to its performance.

Since fail-stop consensus algorithms assume the replicas are uncompromised (non-Byzantine) anyway, this effective "trusted timestamping" assumption may not appear to represent a major expansion of the trust base, especially in a tightly-controlled data center environment. Nevertheless, it is normally preferable for consensus algorithms to rely only on the trustworthiness of the (typically few) replicas themselves, and not on all of the clients or front-ends that may submit requests to them, which are typically far more numerous and more likely to be spread across multiple trust domains. Thus, Rabia's reliance on correct timestamps assigned by clients or proxies represents a significant expansion of the consensus algorithm's trust base, even if this expansion may be deemed reasonable in Rabia's target data center environment. QuePaxa, in contrast, does not assume that the clients or front-end proxies submitting requests are trustworthy.

Even if Rabia's timestamping tier is uncompromised, its failure semantics yields further subtle challenges. If a client first submits a request through one timestamping proxy, but does not receive a response after a timeout period, does the client re-submit its request through another timestamper? If so, and the original request was merely delayed but not lost, then the Rabia replicas may sometimes receive the same client request multiple times with distinct timestamps attached, potentially affecting Rabia's correctness or performance. If Rabia expects client requests to arrive with *exactly one* timestamp, however, then it appears that a fault-tolerant group of timestamping servers would have to implement consensus in order to agree on a unique timestamp for each request. But there would then be no need for Rabia to implement consensus redundantly. It is therefore important, but non-trivial, to define and implement a timestamping tier with failure semantics strong enough to satisfy Rabia's requirements and support its reliability and performance, but weak enough that the timestamping tier need not already implement consensus. This hidden complexity of designing the timestamping tier, and avoiding subtle failure hazards from its interaction with Rabia, thus somewhat undermines Rabia's apparent simplicity. By avoiding any dependency on request timestamping, in contrast, QuePaxa avoids these subtleties and the hidden complexity they may create in practical implementations.

Implications of Rabia's fast-path delay assumption Another subtlety arises from Rabia's fast-path delay assumption, namely that "message delay is small compared to the interval between two consecutive requests" [66, §3.2].

We can safely expect this assumption to hold on a low-delay data center network, *provided* the actual requests ultimately arrive from *outside* the data center, *e.g.*, over a higher-delay, more bandwidth-constrained Internet link. This is presumably the case when the consensus group is directly serving Internet-based clients external to the relevant data center.

This assumption might *not* hold, however, if Rabia is used for applications in which sudden "requests storms" or other heavy loads might be generated by other servers *within* the same data center, over the same internal high-capacity internconnect. If numerous request submittors *within* the same data center may generate bursts of requests under heavy load or other unusual conditions, then the performance differential between the data center's network and long-haul WAN links no longer guarantees Rabia's fast-path delay assumption. Thus, even without any deliberate malicious behavior, request storms originating within the same data center could potentially defeat Rabia's assumption that replica usually observe a natural network-based consensus on oldest-timestamped pending request, and thereby cause Rabia to fall off its past path for extended periods (perhaps during the very high-load periods when maximum performance is most needed).

In summary, Rabia's key fast-path delay assumption is not just an assumption about the network itself, but is also an assemption about the nature and *origin* of application-imposed traffic loads: *e.g.*, whether load originates from within or from outside the data center. Thus, while Rabia might consistently perform well when serving application clients that are always across the Internet external to the data center, its performance may be less reliable in multi-tenant cloud environments where appplication clients may often be other servers that happen to be hosted within the same data center.

While our experiments with Rabia broadly confirmed its dependence for performance on the network conditions it expects, exploring in detail Rabia's sensitivity to external versus internal application loads under varying conditions is a task out of this paper's scope that we leave to future work. In contrast, since QuePaxa makes no delay assumptions (even for its fast path) that depend on the nature and origin of application-imposed load, we can expect QuePaxa to maintain good performance under more diverse conditions, as our experimental results appear to support.