

# On Orchestrating Parallel Broadcasts for Distributed Ledgers

Peiyao Sheng<sup>1</sup> ✉

University of Illinois Urbana-Champaign

Chenyuan Wu<sup>1</sup> ✉

University of Pennsylvania

Dahlia Malkhi ✉

University of California, Santa Barbara

Chainlink Labs

Michael K. Reiter ✉

Duke University

Chainlink Labs

Chrysoula Stathakopoulou ✉

Chainlink Labs

Michael Wei ✉

VMware

Maofan Yin ✉

University of California, Santa Barbara

---

## Abstract

This paper introduces and develops the concept of “ticketing”, through which atomic broadcasts are orchestrated by nodes in a distributed system. The paper studies different ticketing regimes that allow parallelism, yet prevent slow nodes from hampering overall progress. It introduces a hybrid scheme which combines managed and unmanaged ticketing regimes, striking a balance between adaptivity and resilience. The performance evaluation demonstrates how managed and unmanaged ticketing regimes benefit throughput in systems with heterogeneous resources both in static and dynamic scenarios, with the managed ticketing regime performing better among the two as it adapts better. Finally, it demonstrates how using the hybrid ticketing regime performance can enjoy both the adaptivity of the managed regime and the liveness guarantees of the unmanaged regime.

**2012 ACM Subject Classification** Computer systems organization → Reliability

**Keywords and phrases** replication, consensus, fault tolerance, blockchain

**Digital Object Identifier** 10.4230/LIPIcs...

**Acknowledgements** This work was conducted when all authors were working at Chainlink Labs. We thank Gregory Neven for his help with the discussions and proofreading.

## 1 Introduction

In state machine replication, operations are organized into a totally ordered sequence through an atomic broadcast protocol [26]. In this paper, we are interested primarily in Byzantine fault-tolerant (BFT) atomic broadcast protocols in partial-synchrony, for which solutions are myriad, but share certain ingredients. In sequential leader-based protocols, one process at a time is designated the leader, and the leader proposes (blocks of) operations/transactions for the next available (i.e., not yet occupied) slot in the sequence. After decades of advances

---

<sup>1</sup> Both authors contributed equally to the paper.



in scaling-**up** leader-based solutions, recent advances increase throughput by scaling-**out** and allowing parallel proposing to form a *block-DAG*, e.g., SwirlDS [4], Blockmania [13], Aleph [19], and Narwhal/Tusk [14]. In block-DAG protocols, all nodes propose in parallel for the next group of slots, and then the entire group simultaneously commits. In both the sequential-leader and block-DAG paradigms, the slots in which proposed transactions might settle are implicitly left to be the next available slots in the sequence.

In this paper, we introduce the concept of “*ticketing*” to explicitly manage the slots in which proposed transactions might settle. We stress that ticketing is separate from the mechanics by which consensus is reached on the operation in a slot. Rather, we leverage the protocol by which slot finalization (and commitment) is performed as a black box.

The goal of ticketing is to orchestrate atomic broadcasts by nodes in a distributed system with parallelism, yet prevent slow nodes from hampering overall progress. Tickets capture the right to propose transactions to be committed to the totally ordered sequence, and ticketing refers to the method of orchestrating the assignment of privileges for slots in the sequence. Several properties factor into the success of a ticketing scheme. For example, we want to allow parallel proposing but throttle fast proposers from depleting system resources. We want to prevent bad (or even malicious) proposers from slowing down progress. And we want all of this to dynamically adapt to changing system conditions. These properties are summarized in our problem definition in Section 2.3.

In the BFT literature, when viewed as a ticketing regime, the prevailing approach for orchestrating proposals is through a sequential leader replacement regime. The vast majority of protocols employ a uniform regime that rotates leaders in a round-robin manner or via a randomized lottery among all nodes. This requires all honest nodes to participate uniformly.

Adaptive sequential leader replacement methods based on reputation were introduced first in Carousel [12] for sequential BFT protocols and later, in the context of block-DAG, in Shoal [28], Hammerhead [32], and Mysticeti [3]. All of the approaches above do not allow more than one-third of the participants to opt-out of participating, and (in the case of block-DAG protocols) do not orchestrate parallel proposing for nodes with varying speeds. These approaches may be categorized as *unmanaged* as they are governed by a distributed protocol.

On the other side, in the crash fault-tolerant (CFT) atomic broadcast literature, distributed shared logs like CORFU [5] demonstrated excellent performance with a *managed* ticketing approach. In such systems, there is a designated “*ticketing-server*” that is responsible for assigning proposers to slots. The ticketing-server orchestrates proposing but is decoupled from the consensus protocol that finalizes slots. Thus, the ticketing-server primarily serves as a performance enhancement tool, whereas replication via the consensus protocol guarantees safety. A similar approach was recently explored for BFT settings in BBCA-Ledger [31].

The benefit of this approach is that it can drive latency down to the limit by allowing parallel broadcasts, yet commits can happen as soon as they are delivered. More concretely, it allows broadcasts to become *finalized* out-of-order, because they have pre-designated slots. A finalized slot becomes *committed* when the slot preceding it is committed. Under good conditions, this can happen instantaneously. In order to address “*holes*,” which might be left in the sequence by bad ticket-holders and prevent higher slots from committing, each slot may be finalized by consensus with a special  $\perp$  value after an expiration period.

Managed ticketing embodies several desirable properties, including (i) automatically adapting to faulty/slow nodes, (ii) supporting parallel proposing, and (iii) permitting participants to opt-out from proposing.

One seeming drawback of managed ticketing would be introducing a centralized bottleneck, namely the ticketing-server. Surprisingly, we demonstrate in Section 4 that despite this, managed ticketing has excellent performance because it prevents other sources of slowness. Additionally, we prove in Section 3 (Theorem 4) that under crash-failures only, after a bounded “warm-up” segment, managed ticketing prevents any holes from forming in the sequence.

The second drawback, unique to the Byzantine setting, is the threat of a bad ticketing-server. To tackle this problem, we introduce a dual managed/unmanaged regime called Hybrid Ticketing Regime (HTR), a flexible ticketing regime that transitions between the two without sacrificing consistency. Hybrid Ticketing Regime thus strikes a balance between adaptiveness and resilience.

The performance evaluation demonstrates how managed and unmanaged ticketing regimes benefit throughput in systems with heterogeneous resources both in static and dynamic scenarios, with the managed ticketing regime performing better of the two as it adapts more effectively. Finally it demonstrates how using the hybrid ticketing regime performance can enjoy both the adaptivity of the managed regime and the liveness guarantees of the unmanaged regime.

## 2 System Overview

At a high level, this work tackles the classic problem of *log replication* in permissioned settings; for completeness, the fault model and problem definition are provided in Section 2.1.

We consider standard BFT atomic broadcast, but with the additional flexibility of allowing proposers to inject values (blocks) into log slots in parallel, while allowing individual proposals to become committed immediately rather than delaying to commit proposals in batches. To accomplish this, first, the protocol for individual slots should exhibit a property referred to as *out-of-order finality*, on which we elaborate in Section 2.2. Second, and the principal focus of this paper, injecting proposals into slots should be orchestrated wisely: the goal is to allow parallelism while preventing contention and while adapting to varying workloads and dynamic conditions. This leads us to introduce in Section 2.3 the notion of a *ticketing* regime and formulate a set of desirable ticketing properties.

### 2.1 Model

Our system involves a network of  $n$  nodes  $P = \{0, 1, \dots, n - 1\}$ . Within this network, up to  $f$  nodes may be faulty. Crash faults stall the node permanently while Byzantine nodes act in arbitrarily malicious ways. Nodes that are neither crashed nor Byzantine are correct.

We assume partially synchronous communication [18], indicating that there exists an unknown Global Synchronization Time (GST), after which the communication delays within the network are bounded by  $\Delta$ . Each communication channel is authenticated, and each node has a public identity established by Public Key Infrastructure (PKI). The notation  $\langle m \rangle_p$  denotes a message  $m$  signed using the public key of node  $p \in P$ .

The system’s nodes implement atomic broadcast via a replicated log. We denote the data structure maintained by each node as *log*, and  $log[sn]$  represents the *slot* in the log with slot number  $sn$ . We assume that the replicated log exposes a basic interface with a  $broadcast(sn, b)$  method to propose a value (block in the context of building a blockchain)  $b$  for slot  $log[sn]$ .

We assume that each slot in the log can be in one of three states: *unwritten*, *finalized*, or *committed*. If a slot is finalized, then its contents will not be altered in the future, and

## XX:4 On Orchestrating Parallel Broadcasts for Distributed Ledgers

content in the slot, if any, is reliably replicated and permanently stored. Slot commitment is defined inductively: if a prefix is finalized, then each slot in the prefix is considered committed. A slot that is neither finalized nor committed is *unwritten*. We assume that the log interface further exposes a notification event  $\text{LOG-COMMIT}(sn, b)$  which is triggered at all nodes once a slot  $\text{log}[sn]$  gets committed with value  $b$ .

A replicated log must maintain the following guarantees:

- **Consistency** If two correct nodes attain  $\text{LOG-COMMIT}(sn, b)$  and  $\text{LOG-COMMIT}(sn, b')$  for the same slot number  $sn$ , then  $b = b'$ .
- **Liveness** Eventually every slot  $\text{log}[sn]$  attains  $\text{LOG-COMMIT}(sn, b)$  at all correct nodes, where  $b$  is a block or a special  $\perp$  value.

### 2.2 Out-of-Order Finality

To benefit from the ability to inject slot proposals in parallel, slots should be able to finalize *out-of-order*. That is, each proposal explicitly carries with it a ticket for a slot; the log replication protocol then tries to finalize each slot with a proposal ticketed for it. Out-of-order finality allows “holes” to be left in the log by bad ticket-holders. Holes prevent the commit progress, but not the finality progress, of subsequent slots. That is, higher slots may become finalized without indirectly finalizing all lower slots, unlike many log-replication protocols (e.g., Raft [25], HotStuff [34]) that finalize slots in monotonically increasing order. In order to prevent holes from preventing higher slots from committing, each slot may be finalized by consensus with a special  $\perp$  value after an expiration period.

Whereas any consensus algorithm could be used as a per-slot protocol, our evaluation focuses on BBKA-Ledger [31]. For completeness, briefly BBKA-Ledger implements a single-view regime of PBFT per slot, driven by a leader designated as a “ticket holder” for that slot. If there is no observed decision for a certain period, nodes “eject” and trigger a fallback consensus, which is akin to a view-change mechanism but can determine only one of two possible outcomes: either the ticket-holder’s original proposal or  $\perp$ .

To guarantee that ejecting does not disrupt liveness, it needs to be synchronized across nodes. We assume that there exists a view-synchronization module called *Pacemaker* [6, 24, 12]. *Pacemaker* manages the starting time and the length of the slot timer for each slot and guarantees a minimum time frame for making progress after GST. More specifically, nodes can access a local pacemaker module  $\Gamma[sn]$  maintaining the following guarantee:

► **Definition 1** (Synchronized slot). *With Pacemaker, for all slots  $\text{log}[sn]$  starting after GST,  $\Gamma[sn]$  of all correct nodes are active for at least  $\Delta_p$  time. We say a slot  $\text{log}[sn]$  starts after GST if the earliest slot timer for  $sn$  of a correct node starts after GST.*

Here the parameter  $\Delta_p$  represents the overlap duration in slots sufficient for correct participants to finalize a block after GST. It is determined by the replicated log protocol and the fallback consensus module.

### 2.3 Ticketing: Problem Statement

The goal of this work is to develop an efficient ticketing mechanism for *orchestrating* broadcasts; i.e., assigning to nodes the right to propose to slots in the log, referred to as tickets. The ticketing module specifies a local interface  $\text{verifyTicket}(\text{log}, sn, p) \in \{\text{valid}, \text{invalid}, \text{undefined}\}$ , which allows nodes to verify locally whether node  $p$  is eligible for proposing in slot  $\text{log}[sn]$ , given the current view of  $\text{log}$ . If the return value is *invalid*, the message will be ignored;

when `undefined` is returned, the message will be buffered and checked again when log gets updated; otherwise, nodes will further process the proposal in the replicated log protocol. One key property of the interface is that if `verifyTicket(log, sn, p) = valid` at some correct node, then `verifyTicket(log, sn, p) ≠ invalid` at any correct node. This ensures consistency in the eligibility of proposals across correct nodes.

Designing a ticketing scheme that enables good performance under varying conditions and workloads surfaces several desiderata. For instance, the ticketing scheme that assigns slots to all nodes uniformly should perform well in a symmetric network.<sup>2</sup> However, in situations where many nodes lack data to propose, such an even allocation could unintentionally waste bandwidth by finalizing empty slots. A more refined strategy would wisely allocate more slots, for example, to nodes with a larger pool of payloads. Beyond just the volume of data to propose, nodes equipped with other resources such as better network capabilities and more advanced computational power should be given a greater number of slots, proportional to their contribution to the log. This property is captured by the principle known as *meritocracy*, emphasizing the importance of efficiency and smart resource utilization.

Furthermore, a good ticketing scheme should be adaptable, ready for both ideal and worst-case scenarios. In an ideal setting where the network is well-connected and all nodes are fault-free, it is best to assign each slot to a single node, eliminating any potential conflicts (referred to as *contention-free allocation*). In the face of unstable network conditions or instances of node crashes, however, the design should be resilient enough to minimize the waste of resources caused by failures. To help measure this, we introduce a property called *slot utilization*, inspired by the leader utilization proposed by Carousel [12], which aims to restrict the number of skipped slots after GST in a crash-only execution.

Additionally, we take into account the *chain quality* [20] of the entire log. This aspect ensures that the portion of log slots proposed by Byzantine nodes after GST remains bounded. The ticketing process specifies a sliding window of pending proposals, capturing the system's inherent parallelism. The design of the window size should aim to enable seamless system operation while preventing potential bottlenecks. These essential properties for a desired ticketing regime are summarized as follows:

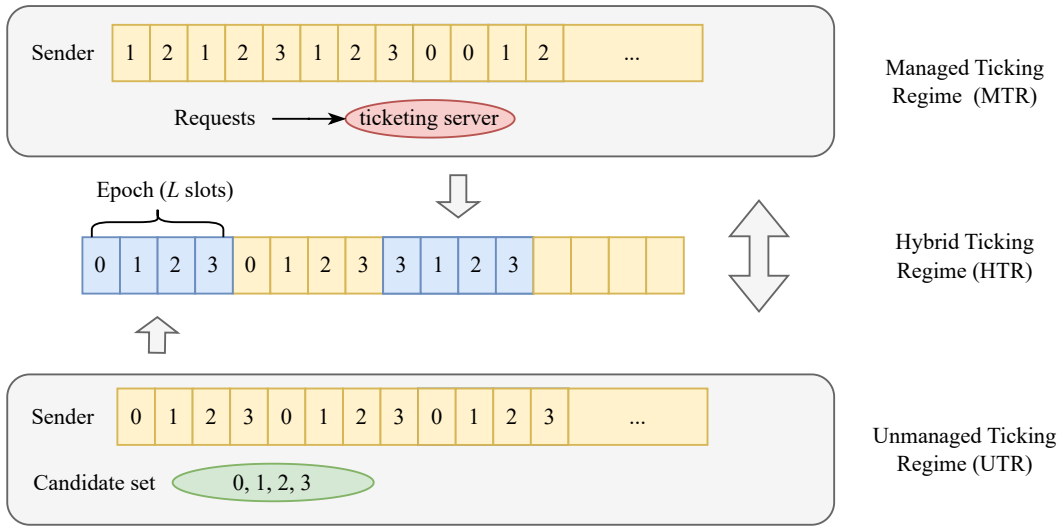
- **Meritocracy** Nodes that are active and equipped with better resources (e.g., larger payload, advanced network, and computational resources) are favored with a higher number of slots.
- **Contention-free allocation** A unique node is assigned to every individual slot.
- **Slot utilization** In crash-only executions, after GST, the number of skipped slots is bounded.
- **Chain quality** After GST, the proportion of blocks contributed by Byzantine nodes is bounded in the committed chain of correct nodes.

## 2.4 Technical Approach

Exploring the dynamics of ticketing in distributed systems, we distinguish between two main patterns: managed and unmanaged. Managed ticketing utilizes a centralized coordinator who listens to ticket requests from potential proposers and assigns slots. Unmanaged ticketing operates in a decentralized manner.

---

<sup>2</sup> Our subsequent experiments reveal that, even in a statistically symmetrical network, there are nuanced discrepancies in each node's progress due to system bootstrapping, necessitating a more adaptive design.



■ **Figure 1** A hybrid managed/unmanaged ticketing regime.

Our analysis compares three specific ticketing regimes, two basic regimes implementing solely managed or unmanaged types, and a dual-mode ticketing regime incorporating both types. On the one hand, the evaluations reveal that the managed ticketing regime adapts well to dynamic network conditions but is vulnerable to single-node failures. On the other hand, the unmanaged ticketing regime offers simplicity and stability in face of faulty environments, yet lacks responsiveness to network changes. Recognizing the limitations inherent in both models, we identify the need for a hybrid paradigm, which integrates the adaptability of the managed approach with the stability of the unmanaged approach, addressing the complexities in system environments and participant behaviors.

The proposed algorithm, called hybrid ticketing regime, defines a switching mechanism between a managed ticketing regime (MTR) and a unmanaged ticketing regime (UTR), based on assessments of network stability and performance metrics (Figure 1). Specifically, when the network is good and the log is growing without skipped slots, the managed scheme will be adopted to optimize resource allocation. When the network is unstable or a Byzantine ticketing-server is in place, the log might be stalled, or chain quality is harmed. In this case, the unmanaged scheme will be adopted to resynchronize nodes and bring the system back to a normal pace. This adaptive mechanism ensures efficient resource utilization and good system performance across a range of conditions.

### 3 The Hybrid Ticketing Regime

#### 3.1 The Protocol

When designing the ticketing scheme, there are two possible basic paradigms, *managed* and *unmanaged*. In a managed ticketing approach, a special role is given to one node at a time to actively manage the assignment of the slots. In an unmanaged ticketing approach, there is a deterministic rule allowing nodes to independently find slot assignments based on their local copy of the log. As outlined in Section 2, we combine both approaches into a hybrid scheme, thus enjoying the agility of a managed approach, coupled with dynamic switching to an unmanaged one for Byzantine resilience.

In Algorithm 1 we introduce the hybrid ticketing regime (HTR). The protocol groups

every  $L \geq 2f + 1$  slots into an *epoch*, such that all slots in each epoch are assigned by the same ticketing scheme. For each epoch  $i$ , each node maintains a local candidate set  $C[i]$ , and a local ticketing scheme  $TR[i]$ . When  $TR[i] = -1$ , the epoch employs an unmanaged, round-robin scheme, rotating through the nodes in  $C[i]$  as eligible proposers. In this case, the `verifyTicket` function simply verifies whether a proposer for slot  $sn$  in epoch  $i$  has index  $sn \bmod L$  in  $C[i]$ . When  $0 \leq TR[i] \leq n - 1$ ,  $TR[i]$  represents the elected ticketing-server. The ticketing-server accepts requests from nodes and sends signed certificates allocating slots to nodes, which can be verified by the `verifyTicket` function. For completeness, we define that the `verifyTicket(log, sn, *)` function will return `undefined` for all checks to slot  $sn$  in epoch  $i$  when  $TR[i]$  has not been decided. However, correct nodes will never check an unentered epoch and thus will never receive `undefined` as the return value.

To enable high parallelism for data dissemination, our protocol allows  $K$  epochs to proceed concurrently. Initially, the first  $K$  epochs start simultaneously using an unmanaged round-robin scheme rotating through all nodes (line 8-10). For each subsequent epoch  $i > K$ , the scheme for the epoch is determined by the outcome of all the slots of epoch  $i - K$ . As soon as  $TR[i]$  is known, designated proposers can start initiating broadcasts for the epoch. As a result, our protocol allows a node to propose slots at least  $(K - 1)L$  ahead of the highest committed slot it knows, this bound is denoted as *Maximum Sliding-Window (MSW)*.

Each node subscribes to the LOG-COMMIT event from the replicated log protocol. Upon the commitment of all slots in epoch  $i$  (interchangeably, we say epoch  $i$  gets committed) (line 12), the protocol reverts the ticketing regime to UTR (or keeps it, if already using it) if either one of the following two conditions holds: (1) there exists at least one skipped slot in epoch  $i$  (i.e., slots committed with a special  $\perp$  value), (2) the the number of distinct ‘active’ senders in epoch  $i$  is less than  $2f + 1$ . An active sender is a node with at least one proposed slot committed in epoch  $i$ . If either (1) or (2) hold, epoch  $i + K$  is set to use an unmanaged round-robin scheme (i.e.,  $TR[i + K] = -1$ ). Otherwise, a ticketing-server is selected among the candidate set  $C[i + K]$  as explained below (line 22-25) through function `getTicketingServer`.

The candidate set  $C[i + K]$  is updated after every unmanaged epoch  $i$  (line 14). Usually, the candidate set for epoch  $i + K$  is updated to the set of ‘active’ senders from epoch  $i$ , containing all senders with at least one proposed slot committed in epoch  $i$ . However, to ensure chain quality, it is imperative to maintain a minimum of  $2f + 1$  senders. In situations where there are not enough active senders,  $C[i + K]$  is reset to the group of all nodes. If the committed epoch  $i$  operates with MTR, the protocol keeps the same candidate set to counter potential manipulations by a Byzantine ticketing-server, who could possibly sideline correct senders to gain unfair election privilege. Figure 2 illustrates an example of how HTR updates different parameters based on execution results.

## 3.2 Analysis

In Section 2.3, we identified the desired properties of an efficient ticketing regime. In this section, we prove that our design satisfies these properties, except for meritocracy, which is demonstrated in Section 4.

We first state the most straightforward property. In epochs with the unmanaged ticketing regime and correct ticketing-servers, only one sender will be assigned to each slot, therefore our design is a *contention-free allocation* in these good cases.

**Algorithm 1** The hybrid ticketing regime implementation

---

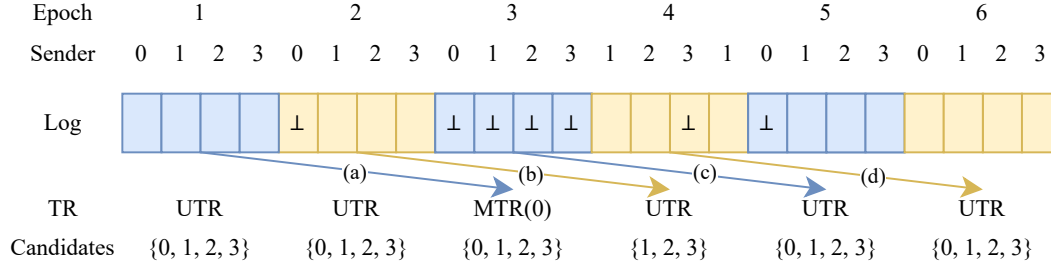
```

1: Init:
2:    $n$  ▷ Number of nodes
3:    $L$  ▷ Number of slots per epoch
4:    $K$  ▷ Number of concurrent epochs
5:    $seed$  ▷ Common seed
6:    $C \leftarrow \{\}$  ▷ Map of candidate set per epoch
7:    $TR \leftarrow \{\}$  ▷ Map of ticketing regime per epoch
8: upon event  $INIT()$  do
9:    $C[1] = C[2] = \dots = C[K] = [0, 1, \dots, n - 1]$ 
10:   $TR[1] = TR[2] = \dots = TR[K] = -1$  ▷ First  $K$  epochs are round-robin epochs
11: upon event  $LOG-COMMIT(sn, b)$  do
12:  if  $sn \bmod L = 0$  then ▷ All slots in an epoch are committed
13:     $i \leftarrow sn/L$ 
14:    ▷ Update candidate set
15:    if  $TR[i] = -1$  then
16:       $S \leftarrow \{log[sn].sender \mid \forall sn \in [(i - 1)L + 1, iL], log[sn] \neq \perp\}$ 
17:      if  $|S| < 2f + 1$  then
18:         $C[i + K] = [0, 1, \dots, n - 1]$ 
19:      else
20:         $C[i + K] = S$ 
21:      else
22:         $C[i + K] = C[i]$ 
23:    ▷ Switch ticketing regime
24:    if  $\exists sn \in [(i - 1)L + 1, iL], log[sn] = \perp$  or  $|S| < 2f + 1 \wedge TR[i] \neq -1$  then
25:       $TR[i + K] \leftarrow -1$ 
26:    else
27:       $TR[i + K] \leftarrow getTicketingServer(i + K)$ 
28: function  $getTicketingServer(epoch)$ 
29:    $k \leftarrow Hash(seed, epoch) \bmod |C[epoch]|$ 
30:   sort  $C[epoch]$  by nodes' public keys in ascendant order
31:   return  $C[epoch][k]$ 

```

---





■ **Figure 2** An example with six epochs,  $n = 4$ ,  $L = 4$ ,  $K = 2$ . Log slots with  $\perp$  are skipped slots and others are committed with non empty values. The example shows four possible updating rules: (a) epoch 3 uses a ticketing-server since no slots are skipped in epoch 1; (b) epoch 4 keeps using round-robin since one slot is skipped in epoch 2 and the candidate set is updated to exclude node 0; (c) though all slots in epoch 3 are skipped, the candidate set remains the same since epoch 3 uses the managed ticketing regime; (d) the candidate set is reset to the full group of nodes in epoch 5 since epoch 4 adopts the unmanaged ticketing regime and has only 2 active senders.

### Slot utilization.

To prove slot utilization, we first prove that the ticketing regime in each epoch is consistent across correct nodes.

► **Lemma 2** (Epoch consistency). *For any epoch  $i$  and any two correct nodes  $p$  and  $q$ , let  $C_p, C_q$  and  $TR_p, TR_q$  denote their local candidate sets and ticketing regimes, then  $C_p[i] = C_q[i]$  and  $TR_p[i] = TR_q[i]$ .*

**Proof.** We first demonstrate that for a series of epochs executed sequentially, such as epochs  $1, K + 1, 2K + 1, \dots$ , if correct nodes begin with consistent views of their candidate sets and ticketing schemes, they will maintain these views consistently in all subsequent epochs.

Base case: initially, any two correct nodes  $p$  and  $q$  have  $C_p[1] = C_q[1]$  and start with  $TR_p[1] = TR_q[1] = -1$ .

Inductive Step: for any  $i = tK + 1 (t \geq 1)$ , we will show that  $C_p[i] = C_q[i]$  and  $TR_p[i] = TR_q[i]$  holds if  $TR_p[i - K] = TR_q[i - K]$  and  $C_p[i - K] = C_q[i - K]$  holds.

First we prove the candidate sets are consistent. If  $TR_p[i - K] = TR_q[i - K] = -1$ , by consistency of the distributed log, all correct nodes recognize the identical set of active senders  $S$ . As a result, regardless of the number of senders in this set,  $C_p[i] = C_q[i]$ . If  $TR_p[i - K] = TR_q[i - K] \geq 0$ , following the given protocol, we have  $C_p[i] = C_p[i - K] = C_q[i - K] = C_q[i]$ .

Then we prove the ticketing schemes are consistent. Again by consistency of the distributed log,  $TR_p[i] = TR_q[i] = -1$  if there are any skipped slots. Otherwise, correct nodes unanimously elect a ticketing-server by invoking the function `getTicketingServer(i)`. Given the same seed and the previously demonstrated consistency  $C_p[i] = C_q[i]$ , it follows that  $TR_p[i] = TR_q[i]$ .

By induction, we conclude that the assertion holds for all  $iK + 1$ .

An identical inductive proof holds for  $iK + j$  for every  $j = 1, 2, \dots, K$  hence the Lemma holds for all epochs  $> 0$ . ◀

To uphold slot utilization, after GST, for any given slot, we must ensure that if there exists a single correct sender, the slot will not be skipped.

## XX:10 On Orchestrating Parallel Broadcasts for Distributed Ledgers

► **Lemma 3** (Non-skipping epoch). *In a crash-only execution, if no nodes have crashed in an epoch  $i$  starting after GST, nor are any nodes in  $C[i]$  currently crashed, then no slots will be skipped in epoch  $i$ .*

**Proof.** By Definition 1, slots in epoch  $i$  are synchronized slots, so correct nodes have enough time to participate. By Lemma 2, all correct nodes have the same candidate set. If epoch  $i$  is a round-robin epoch, then since all candidates are alive and no nodes have crashed in this epoch, all slots will be finalized. If epoch  $i$  uses a ticketing-server selected from the candidate set, it is alive, and since there is no crash in the current epoch, every slot will be finalized. ◀

► **Theorem 4** (Slot utilization). *In a crash-only execution after GST, the number of slots  $s$  committed with  $\perp$  (called skipped slots) is bounded by  $O(fKL)$ .*

**Proof.** To simplify notation, we will fix  $1 \leq k \leq K$  and bound the number of skipped slots after GST in the “ $k$ -subsequence” of epochs  $i$  of the form  $k + x \cdot K$ , for  $x = 1, 2, \dots$ . Each such sub-sequence operates independently from others with respect to determining the ticketing regime. We can then multiply the bound we obtain in the end by  $K$ .

We argue that after GST, within each  $k$ -subsequence there are most  $f + 1$  candidate-set resets to a full set. If a candidate-set reset happens at, say, epoch  $i$ , then for all epochs  $j = i + xK$ ,  $x = 1, 2, \dots$ , there are two cases to consider. Case 1 is that epoch  $j$  is unmanaged or it is managed by a nonfaulty ticketing-server. Then there are at least  $2f + 1$  non-crashed active senders and the candidate-set will not be reset. Case 2 is that epoch  $j$  is managed by a crashed ticketing-server  $p$ . Then  $j + K$  will be unmanaged. From here on, any higher epoch  $j + K + xK$  can switch to a managed regime only if every node in  $C[j + K + xK]$  is an active sender and there are no crashed nodes. Hence, each node who has crashed by epoch  $i$  cannot become a ticketing-server (and a fortiori,  $p$  cannot become a ticketing-server again). Hence, this case can contribute at most  $f$  resets.

We now put together the number of skipped slots both cases may contribute. The second case, managed epochs whose ticketing-servers are crashed, may contribute  $fL$  skipped slots. Additionally, in both bases put together, each crashed node  $p$  can contribute at most  $2(f + 1)(L/n)$  skipped slots, because after an epoch with (at most  $L/n$ ) skipped slots by  $p$ , it is removed from the candidate-set. As we showed above, nodes can return to the candidate-set at most  $(f + 1)$  times.

The total number of skipped slots per  $k$ -subsequence is therefore bounded by  $fL + 2(f + 1)(L/n)$ , and the total skipped slots by  $K \times (fL + 2(f + 1)(L/n))$ . ◀

### Chain quality.

Last, to prove *chain quality* after GST, we consider the worst case when Byzantine ticketing-servers are consecutive. We have the following theorem:

► **Theorem 5** (Chain Quality). *Algorithm 1 satisfies chain quality: at least  $(f + 1)K$  blocks are proposed by correct nodes in every  $2KL$  blocks after GST.*

**Proof.** Since in epochs with the unmanaged ticketing regime or with correct ticketing-servers there are at least  $2f + 1$  senders, by Definition 1 and Lemma 2, these senders will successfully commit their slots. Hence, the number of blocks contributed by correct nodes in each such epoch is at least  $f + 1$ .

In epochs with Byzantine ticketing-servers, it's possible that all blocks are proposed by Byzantine nodes, but this can be detected by counting the number of distinct active senders

■ **Table 1** Comparison of different ticketing regimes.

	Properties	Managed	Unmanaged	Dual-regime
Good-case	Meritocracy	✓		✓
	Contention-free allocation	✓	✓	✓
Worst-case	Slot utilization	unbounded	unbounded	$O(fKL)$
	Chain quality	unbounded	$O((f+1)/L)$	$O((f+1)/2L)$

after the epochs are committed. Once such an epoch  $i$  is detected, the upcoming epoch  $i + K$  will use UTR, and it will have at least  $f + 1$  blocks contributed by correct nodes. As a result, in every  $2K$  epochs, at most  $K$  epochs have Byzantine ticketing-servers, which means at least  $(f + 1)K$  blocks are contributed by correct nodes in every  $2KL$  blocks. ◀

### Remark

An analysis of the two basic ticketing regimes (managed and unmanaged) is summarized in Table 1. The managed ticketing regime naturally supports meritocracy but has an unbounded number of skipped slots in face of a malicious ticketing-server. As such, it does not ensure slot utilization or maintain chain quality. The unmanaged scheme distributes tickets evenly across all nodes, which does not support meritocracy but is robust against Byzantine behavior, thereby providing good chain quality. Our approach integrates the advantages of both regimes and improves slot utilization by adaptively updating active sender set. To address the chain quality issues associated with managed epochs, we transition them to unmanaged epochs, thereby maintaining overall high chain quality. Additionally, by introducing parallel epochs, our scheme prevents the epochs with undecided ticketing scheme from hindering the progress of consensus.

## 4 Evaluation

### 4.1 Setup

This paper focuses on ticketing methods for orchestrating parallel proposing of BFT atomic broadcast. Our experiments vary the number of simultaneously pending broadcasts and who is permitted to propose. For a practical system, the total number of outstanding broadcasts (that haven't finished) is bounded due to the limited resources, because until they are finalized, pending broadcasts cannot be compacted or checkpointed to secondary storage. We denote this tuning knob by *Global Sliding-Window (GSW)*. The Maximum Sliding-Window (*MSW*) defined in Section 3 captures the maximum *GSW* the experiments can set. The Global Sliding-Window mechanism allows nodes to participate in broadcasts up to a bounded window of size *GSW* beyond their last locally known committed log-slot. This self-throttling sliding window allows all nodes to catch up, limits buffering needs, and prevents faster nodes from proceeding too far ahead.

The assignment of slot numbers to nodes is orthogonal to the *GSW* mechanism. Indeed, we explore and evaluate three approaches to manage slot allocation. In the unmanaged ticketing regime (UTR), slots are allocated to permitted nodes in a round-robin rotation. In the managed ticketing regime (MTR), slots are allocated via an active ticketing-server. In the hybrid ticketing regime (HTR), slot allocation is operated by a hybrid protocol as

■ **Table 2** Performance of different ticketing regimes under a heterogeneous setup

Tickets Assignment	Finality	Commit	Throughput	Proposed by ... (bps)			
	latency (ms)	latency (ms)	(bps)	0	1	2	3
UTR (node 0)	6.3	9.3	10 770	10770	0	0	0
UTR (node 3)	9.0	14.2	4719	0	0	0	4719
UTR (all nodes)	6.1	22.8	4406	1102	1102	1102	1102
MTR (batch=1)	3.9	3.9	2686	853	661	651	356
MTR (batch=10)	1.6	2.0	8830	3167	2863	2611	2
MTR (batch=100)	18.2	27.3	10 755	3546	3790	3383	10

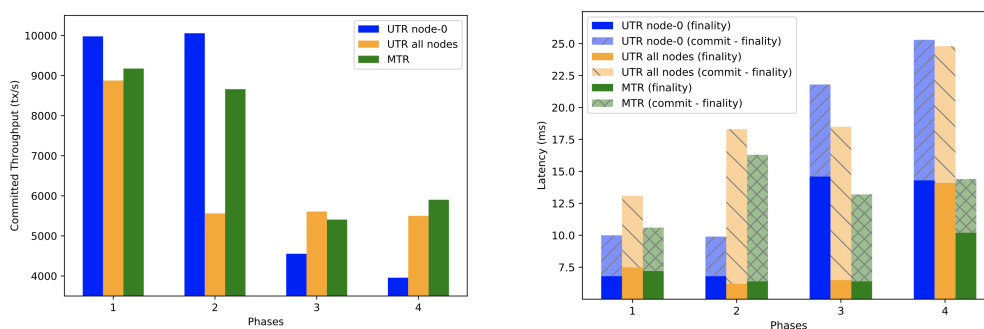
specified in Algorithm 1. We evaluate latency against throughput of these regimes under varying network and failure conditions.

Our experiments are conducted with a distributed setup on CloudLab. Since this paper studies ticketing rather than transaction dissemination, we use a small transaction size with 2 bytes payload throughout our evaluation to show the base performance of the broadcasts, and we do not batch multiple transactions per broadcast. In a production system, however, each proposal could batch hundreds of actual transactions and thus scale up throughput (comparable to published results in the literature). We intentionally isolate this away to emphasize the fundamental performance difference between distinct ticketing regimes.

## 4.2 Static Heterogeneity

We first compare different ticketing regimes over a set of nodes that vary in their processing and network speeds. Specifically, nodes 0–2 are `c6525-25g` instances (16-core 3.0GHz CPU, 25GB NIC), while the remaining node 3 is a slower `m510` instance (8-core 2.0GHz CPU, 10GB NIC). We compare four ticket assignment strategies: unmanaged ticketing regime (UTR) with only node 0 (a fast node) included in the candidate set and permitted to propose, UTR with only node 3 (a slow node) permitted, UTR with all nodes permitted in a round-robin rotation, and managed ticketing regime (MTR). For the first three strategies,  $GSW$  is set to be 100, which is where the throughput saturates without latency impact. For the last strategy, the  $GSW$  bound is not applied, since parallelism is implicitly managed by the active ticketing server and the way each node asks for tickets (i.e., how many and when). Specifically, the active ticketing server batches tickets and distributes a batch to each proposer upon request, where the batch size is a tunable parameter. Each proposer requests for the next batch of tickets only when all slots in the previous batch have been finalized.

Table 2 summarizes the results, where the last four columns present the number of slots each node has proposed. When the fastest node is known *a priori*, designating it as the single proposer renders best throughput. Having a slower node as the single proposer or allowing all nodes to propose in UTR regime have similar performance: the round robin assignment is bottlenecked by the slowest node in the system. Although the slots assigned to fast proposers are finalized rapidly, the slots assigned to the slow proposer progress slowly and result in transient holes everywhere in the ledger given round robin’s uniform allocation. This further limits the rate at which fast nodes can propose, due to the parallelism constraints imposed by  $GSW$ . The MTR regime, on the other hand, demonstrates meritocracy by assigning fast nodes more tickets. With ticket batches of 10, it strikes the sweet spot between latency and throughput. We repeated the same experiment on larger scale networks and observed similar trends.



■ **Figure 3** Throughput and latency of different ticketing regimes under dynamic slowness. Each phase in the experiment lasts 30 seconds where certain nodes are slowed down. MTR achieves best optimal performance in all phases, demonstrating meritocracy and adaptivity to dynamic conditions.

The results indicate that when nodes have stable and predictable capabilities, by appointing the most capable proposer, the system reaches the best performance. However, without such prior knowledge, we can still approximate the best case with a managed ticketing regime since it adapts automatically. This seems to leave the system designer with a choice of either sticking to a single capable proposer or paying a little in performance for adaptivity. Our next set of experiments will explore a dynamic setup of the system where it is infeasible to stick to a fast proposer.

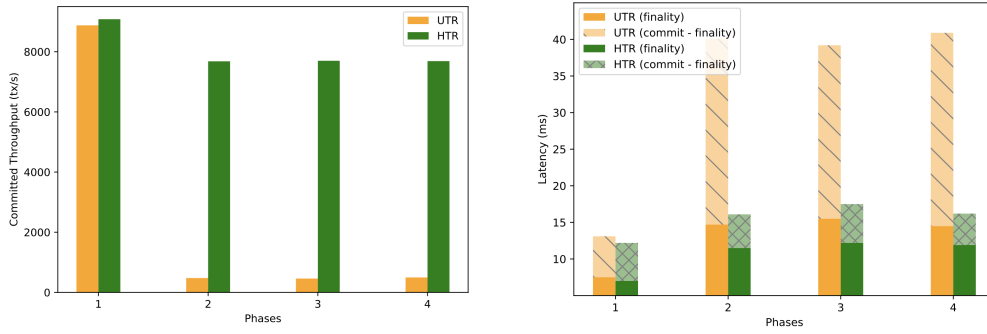
### 4.3 Dynamic Heterogeneity

We repeat the comparison of different ticketing regimes with heterogeneity, but vary the capabilities of nodes over time. We run four consecutive phases on four `c6525-25g` instances, where each phase lasts for 30 seconds. In each phase, we slow down certain nodes by idling a half of available CPU cores: in phase 1, no nodes are slow; in phase 2, only node 3 is slow; in phase 3, only node 0 is slow; in phase 4, only node 1 and node 2 are slow. We compare UTR with only node 0 permitted to propose, UTR with all nodes permitted in a round-robin rotation, and MTR with ticket batches of 10. We choose this batch size for tickets since it strikes the sweet spot between latency and throughput.

Figure 3 summarizes the performance averaged during each phase. In the latency graph, the solid bar at bottom represents the latency for finality, while the entire bar represents the commit latency. MTR achieves nearly optimal performance in all phases, demonstrating meritocracy and adaptivity to dynamic conditions. Conversely, assigning a single fixed proposer results in lower performance as the capability of the node is not static and thus it does not capture the “fastest” node of all time (because there is no such a node). The round robin scheme suffers from poor performance as well. In the practical deployment of a system, nodes could run fast and slow at times due to the uneven load imposed by the clients and the handling of different tasks (voting, verification, transaction execution, storage, etc.). With such dynamic heterogeneity, MTR can still adapt much better and mitigates the unnecessary performance loss compared to other approaches.

### 4.4 Dual-Mode Regime

Compared to the UTR regime with all nodes permitted in round-robin, the main possible drawback for MTR could come from a faulty centralized ticketing server. To address this,



■ **Figure 4** Throughput and latency of different ticketing regimes under dynamic faults. Each phase in the experiment lasts 30 seconds where a certain node is faulty and creates skipped slots. HTR achieves superior performance in all phases, demonstrating fault resilience.

we proposed in Section 3 a dual-mode ticketing regime, and our next experiment evaluates both single and dual-mode regimes with dynamic faults.

We run four consecutive phases on four `c6525-25g` instances, where each phase lasts for 30 seconds. In each phase, we vary which node is faulty: in phase 1, no nodes are faulty; in phase 2, only node 3 is faulty; in phase 3, only node 0 is faulty; in phase 4, only node 1 is faulty. The faulty node will not propose slots even when it is assigned with tickets, thus creating skipped slots in the ledger. In all experiments, we use a simulated fallback consensus for simplicity (that is applied to all ticketing designs) and a 10ms timeout to trigger the fallback consensus. We set the epoch length  $L$  to be 50 and allow  $K = 2$  concurrent epochs, which effectively sets  $GSW$  to its maximum value 50.

Figure 4 summarizes the performance averaged during each phase, where we compare HTR versus UTR with all nodes permitted in a round-robin rotation. Other ticketing regimes suffer from single point failures and are hence not presented in the figure. In the latency graph, the solid bar at bottom represents the latency for finality, while the entire bar represents the commit latency. HTR exhibits superior performance in all phases, since the protocol is designed to bound the number of skipped slots. On the contrary, UTR has unbounded skipped slots, and thus suffers from major performance loss. This means with a dual-mode design, the performance can remain resilient in the case of a faulty ticketing server. Therefore, it is worthwhile to introduce a centralized role to ticketing, given that the faulty server scenario can be mitigated by switching back to a round-round regime and the faulty server is excluded from candidates.

## 5 Related Work

### Ordering layer in shared log.

A shared log is an abstraction that addresses challenges in data consistency and fault tolerance. CORFU [5] pioneered the design by separating ordering from replication, introducing a centralized sequencer to manage ordering as a separate layer. The subsequent advance, Scalog [16], replaces the centralized sequencer with a replicated counter service using Paxos to improve robustness, and aggregates requests through a tree structure to reduce communication. Recently, FlexLog [21] combines the tree structure of sequencer nodes and the single sequencer design in the normal path to further enhance efficiency, and allows for multi-record appends to concurrently append logs.

### Leader election in consensus.

Leader-based consensus algorithms (e.g., [22, 25, 8, 34, 7]) have been widely used to address issues of coordination and agreement among distributed nodes. These protocols usually contain a leader election phase to ensure consistent decision-making on leader rotations and tolerate leader failures in both benign and malicious settings. The simplest leader election scheme, round-robin rotation [33, 9, 34], inherently guarantees fairness but may continuously elect faulty leaders despite evidence of their misbehavior.

To address this issue, Ardvaark and follow up works [11, 33, 1, 2, 30] temporarily eliminate from the leader candidate set nodes that are suspected to be faulty with a blacklist mechanism. Another line of work [23, 17, 10, 15] leverages randomized leader election to prevent a succession of faulty (corrupted) leaders.

Recent work [12] introduces a property called *leader utilization* to bound the number of faulty leaders in crash-only executions after the global stabilization time (GST). This property has been incorporated into some blockchain systems [28, 32, 3] to build a reputation system for leaders, enhancing election quality.


Our paper introduces a more flexible scheme which goes one step further than avoiding potentially faulty leaders. It allows for faster nodes to broadcast more often, such that networking imbalances have less impact on throughput.

### Orchestrating broadcasts in a DAG.

Aleph [19] and follow-up work [14, 29, 28, 27] separate the consensus logic from broadcasting. While they adopt a consensus leader election mechanism which falls in one of the categories discussed above, broadcasts are structured in a layered direct acyclic graph (DAG) such that each broadcast block has causal links (edges) to  $2f + 1$  blocks of the previous layer. This broadcast layering is incompatible with the out-of-order finality which we examine in this work, as blocks in the DAG are finalized every (few) layers by the consensus leader, therefore, comparison with this line of work is out of scope. Our paper presents an alternative, more agile approach to the layered ticketing which does not require all nodes to participate in the protocol as block broadcasters. Whether the insights of the flexible ticketing studied here can be applied to layered DAG broadcasts is an open problem.

---

### References

- 1 Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE transactions on dependable and secure computing*, 8(4):564–577, 2010.
  - 2 Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. Rbft: Redundant byzantine fault tolerance. In *2013 IEEE 33rd international conference on distributed computing systems*, pages 297–306. IEEE, 2013.
  - 3 Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. Mysticeti: Low-latency dag consensus with fast commit path. *arXiv preprint arXiv:2310.14821*, 2023.
  - 4 Leemon Baird. The swirls hashgraph consensus algorithm: Fair, fast, Byzantine fault tolerance. *Swirls Tech Reports SWIRLDS-TR-2016-01, Tech. Rep.*, 34:9–11, 2016.
  - 5 Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. CORFU: A shared log design for flash clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 1–14, 2012.
  - 6 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. *Distributed Computing*, 35(6):503–532, 2022.
- 

## XX:16 On Orchestrating Parallel Broadcasts for Distributed Ledgers

- 7 Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.
- 8 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
- 9 Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. *IACR Cryptol. ePrint Arch.*, 2020:88, 2020.
- 10 Jing Chen and Silvio Micali. Algorand. *arXiv preprint arXiv:1607.01341*, 2016.
- 11 Allen Clement, Mirco Marchetti, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. *Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults (Superseded by UT TR-08-44)*. Computer Science Department, University of Texas at Austin, 2008.
- 12 Shir Cohen, Rati Gelashvili, Lefteris Kokoris Kogias, Zekun Li, Dahlia Malkhi, Alberto Sonnino, and Alexander Spiegelman. Be aware of your leaders. In *International Conference on Financial Cryptography and Data Security*, pages 279–295. Springer, 2022.
- 13 George Danezis and David Hrycyszyn. Blockmania: from block dags to consensus. *arXiv preprint arXiv:1809.01620*, 2018.
- 14 George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.
- 15 Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Advances in Cryptology—EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part II 37*, pages 66–98. Springer, 2018.
- 16 Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 325–338, 2020.
- 17 Sisi Duan, Michael K Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2028–2041, 2018.
- 18 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
- 19 Adam Gągól, Damian Leśniak, Damian Straszak, and Michał Świątek. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 214–228, 2019.
- 20 Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 281–310. Springer, 2015.
- 21 Dimitra Giantsidi, Emmanouil Giortamis, Nathaniel Tornow, Florin Dinu, and Pramod Bhatotia. Flexlog: A shared log for stateful serverless computing. 2023.
- 22 Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001), pages 51–58, 2001.
- 23 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.
- 24 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. 2021.
- 25 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.
- 26 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. URL: <http://doi.acm.org/10.1145/98163.98167>, doi:10.1145/98163.98167.



- 27 Nibesh Shrestha, Aniket Kate, and Kartik Nayak. Sailfish: Towards improving latency of dag-based bft. *Cryptology ePrint Archive*, 2024.
- 28 Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness. *arXiv preprint arXiv:2306.03058*, 2023.
- 29 Alexander Spiegelman, Neil Girdharan, Alberto Sommino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022.
- 30 Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State-machine replication scalability made simple (extended version). *arXiv preprint arXiv:2203.05681*, 2022.
- 31 Chrysoula Stathakopoulou, Michael Wei, Maofan Yin, Hongbo Zhang, and Dahlia Malkhi. Bbca-ledger: High throughput consensus meets low latency. *arXiv preprint arXiv:2306.14757*, 2023.
- 32 Giorgos Tsimos, Anastasios Kichidis, Alberto Sonnino, and Lefteris Kokoris-Kogias. Hammerhead: Leader reputation for dynamic scheduling. *arXiv preprint arXiv:2309.12713*, 2023.
- 33 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one’s wheels? byzantine fault tolerance with a spinning primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 135–144. IEEE, 2009.
- 34 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, 2019.