usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Shoal++: High Throughput DAG BFT Can Be Fast and Robust!

Balaji Arun and Zekun Li, *Aptos Labs;* Florian Suri-Payer, *Cornell University;* Sourav Das, *UIUC;* Alexander Spiegelman, *Aptos Labs*

https://www.usenix.org/conference/nsdi25/presentation/arun

This paper is included in the Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of Science and Technology

Shoal++: High Throughput DAG BFT Can Be Fast and Robust!

Balaji Arun
Aptos Labs

Zekun Li
Aptos Labs

Florian Suri-Payer*
Cornell University

Sourav Das*
UIUC

Alexander Spiegelman
Aptos Labs

Abstract

Today's practical partially synchronous Byzantine Fault Tolerant consensus protocols trade off low latency and high throughput. On the one end, traditional BFT protocols such as PBFT and its derivatives optimize for latency. They require, in fault-free executions, only 3 message delays to commit, the optimum for BFT consensus. However, this class of protocols typically relies on a single leader, hampering throughput scalability. On the other end, a new class of so-called DAG-BFT protocols demonstrates how to achieve highly scalable throughput by separating data dissemination from consensus, and using every replica as proposer. Unfortunately, existing DAG-BFT protocols pay a steep latency premium, requiring on average 10.5 message delays to commit transactions.

This work aims to soften this tension, and proposes Shoal++, a novel DAG-based BFT consensus system that offers the throughput of DAGs while reducing end-to-end consensus commit latency to an average of 4.5 message delays. Our empirical findings are encouraging, showing that Shoal++ achieves throughput comparable to state-of-the-art DAG BFT solutions while reducing latency by up to 60%, even under less favorable network and failure conditions.

1 Introduction

This paper presents Shoal++, a novel partially synchronous DAG-based Byzantine Fault Tolerant (BFT) consensus protocol that matches the throughput of contemporary DAG-BFT protocols while significantly reducing latency, narrowing the gap with the theoretical optimum.

BFT consensus offers the attractive abstraction of a single, always available server, that remains correct even as a subset of replicas may fail arbitrarily. Fueled by recent advances to simplicity and practicality [15, 19, 35, 47], interest and adoption of BFT protocols is skyrocketing: it is at the core of aspiring multi-national projects such as the digital Euro [2], confidential data sharing frameworks [6] and the growing Web3 industry [1, 5]. Already today, numerous blockchain companies [40–42, 44] are actively deploying BFT protocols [16, 22, 31, 34, 35], supporting millions of daily worldwide users.

Following the seminal PBFT [15] protocol, most traditional BFT consensus protocols [24, 28, 47] operate in a primary-backup like regime in which a single, dedicated leader replica proposes an ordering of commands, and follower replicas vote to ensure consistency and durability.

As demonstrated by PBFT, this approach can enjoy excellent latency in fault-free executions, requiring only three message delays (md) to order a leader's proposal, the optimum achievable for BFT protocols. A dedicated leader, however, poses not only a single point of failure, but constrains the achievable throughput to the processing and networking bandwidth available to a single server. To improve scalability, and meet the high throughput demands of modern applications, recent proposals emphasize maximizing resource utilization and balancing load across *all* replicas.

Initial approaches propose multi-log frameworks [37] that partition the request space, and operate multiple consensus instances in parallel (each led by a different replica) whose logs are carefully intertwined. More recent designs take scalability a notch further by entirely separating data dissemination and consensus. Pioneered by Aleph [21] and DAG-Rider [26], and made practical by Narwahl [19] a new class of so-called DAG-BFT protocols have risen to popularity.

DAG-BFT protocols employ a structured, highly parallel, asynchronously responsive data dissemination layer as a backbone, and layer consensus atop. Data dissemination proceeds through a series of *rounds* in which each replica proposes new transaction batches via reliable broadcast (RB); each proposal (DAG *node*) references a subset of preceding proposals (DAG *edges*), resulting in a Directed Acyclic Graph (DAG) of temporally (or *causally*) ordered proposals. The consensus layer reaches agreement on prefixes of the DAG, and derives an ordering for all causally referenced proposals.

State-of-the-art DAG protocols [34, 35] avoid running an explicit consensus protocol, and instead embed the consensus process onto the DAG directly: they ensure that all replicas converge on the same view of the DAG, and interpret it's structure locally to implicitly determine commitment. Consensus simulates leaders by assigning pre-determined "fixpoint" nodes in the DAG (henceforth *anchors*) and confirms agreement of an anchor by waiting until it has gathered sufficient references in future DAG rounds that guarantee its durability.

Upon committing an anchor node, all of its causally referenced proposals are implicitly committed and ordered. Such design is appealingly simple – it requires no additional communication for consensus and avoids the notoriously complex view change logic inherent to traditional BFT protocols. It is also very resource efficient – each replica acts as a proposer, thus allowing high throughput with a large number of replicas. As a result, DAG-BFT has prompted swift adoption by several large blockchain systems [42, 44].

However, today's DAG protocols fall short in terms

*The work was done while the authors were interns at Aptos Labs.

of latency. They require several rounds, each consisting of reliable broadcasts to *certify* DAG nodes, to commit a transaction. Bullshark [35] for instance requires, even in fault-free cases, up to 4 rounds to commit a proposal (12md in expectation, §3.2); recent advances made by Shoal [34] reduce it to 3 rounds (10.5md expected, §3.2). A far cry from the 3md possible by traditional BFT protocols!

This work proposes Shoal++: a novel DAG-based BFT system that achieves near-optimal latency while maintaining the high throughput and robustness of the state-of-the-art certified DAG construction.

We begin by analyzing sources of end-to-end (e2e) latency in DAG protocols (§3.2); at a high level, it can be broken down into three stages. (i) DAG proposals happen only at regular intervals (rounds), and transactions that narrowly miss inclusion (e.g. arrive just after a node is formed) must wait for the next round. We refer to the time a transaction must wait to be included in a DAG proposal as **Queuing Latency**. (ii) In order to be committed, a proposed DAG node must be referenced by a committed anchor. Unfortunately, in existing DAG protocols [34, 35], at most one anchor is scheduled per round[1]. This results in additional **Anchoring latency**, the time until a node is referenced by the next committed anchor. (iii) Finally, in existing DAG protocols, an anchor requires *at least* two more DAG rounds to be committed; we refer to this as **Anchor Commit Latency.**

Key ideas. To reduce end-to-end consensus latency, Shoal++ employs three techniques, respectively addressing each latency stage. First, Shoal++ identifies and optimizes an inefficiency in the existing Bullshark commit rule to reduce common case Anchor Commit Latency to only 4md. Next, we configure Shoal++ to dynamically attempt to make *every* node an anchor, thus eliminating Anchoring Latency. Shoal++ builds on Shoal's idea to dynamically re-interpret the anchor assignment at each commit, and introduces small timeouts to help replicas advance rounds in lock-step to ensure that future rounds sufficiently reference anchor candidates. Finally, to minimize Queuing latencies, Shoal++ orchestrates *multiple* staggered DAG instantiations in parallel, and intertwines their resulting logs; this allows transactions that miss inclusion to simply board the new round of the next, staggered DAG.

Put together, these techniques allow Shoal++ to reduce expected end-to-end consenus latency to only 4.5md, shaving off 6md compared to Shoal, the state of the art DAG-BFT.

We've implemented a prototype of Shoal++ and evaluated it empirically against two popular DAG-BFT protocols (Bullshark [35] and Shoal [34]), a concurrent DAG-BFT protocol (Mysticeti [12]), and a traditional BFT protocols (Jolteon [22]). Our results are promising, demonstrating matching throughput of existing DAG BFT solutions while reducing latency by up to 60% over Bullshark in the common case and by up to 10x over Mysticeti in the failure case.

[1]Bullshark [35] schedules an anchor every second round; Shoal [34] improves it to every round.

2 Model

We adopt standard BFT assumptions [15, 47]. We refer to participants that follow the protocol as correct, and those that deviate as faulty (or *Byzantine*). There are a total of $n=3f+1$ replicas, of which at most f may be faulty at any time. We make no assumptions on the number of faulty clients. We assume the presence of a strong, yet static adversary that may arbitrarily delay messages and coordinate all faulty particpiants, but may not break standard cryptographic primitives such as signatures. We denote signed messages using $\langle m \rangle_\sigma$. We assume that all signatures and Quorums are validated, and henceforth omit explicit mention.

The goal of BFT consensus is to establish a common totally ordered log across all correct replicas. We operate under the partial synchrony model [20]. The system is safe at all times – no two correct replicas ever disagree on a committed prefix of the log. Liveness is not guaranteed during periods of asynchrony, but eventually there arrives a Global Stabilization Time (GST) after which the network (temporarily) behaves synchronously, and progress is guaranteed.

3 Dissecting DAG-BFT

Before we present Shoal++, we provide a brief background on the core mechanisms of DAG-BFT, and analyze the latency breakdown of existing protocols. We implicitly focus on *certified* DAG protocols derived from Narwahl[19]. Although some recent works propose uncertified DAGs [12, 27] in an effort to save latency, this comes at the cost of reduced robustness and introduces new sources of unwanted latency (§3.2).

3.1 DAG core

At the core of DAG-BFT is the round-based certified DAG structure proposed in Narwahl [19]. Replicas proceed through a series of structured rounds, in which replicas continuously propose new nodes containing batches of transactions. Each replica maintains their own local view of a DAG.

A replica adds another replica's proposal as node in its local DAG once the proposal is *certified*. Proposals in round r must reference $n-f$ certified proposals from round $r-1$ to be considered *valid* candidates, and each replica may propose and certify at most one candidate per round. To certify a proposal a replica follows a simple Reliable Broadcast procedure.

1. It broadcasts a signed proposal $P := \langle r, B, edges \rangle_\sigma$ for round r containing (i) a batch B of transactions, and (ii) $n-f$ unique proposal certificates from round $r-1$ (*edges* for short).
2. Upon receiving a valid proposal, a replica checks whether it has already received a different proposal from the same author for round r. If it has, it ignores

the proposal; if it has not, it casts a signed vote $S := \langle r, d = hash(P) \rangle_\sigma$ back to the proposing replica.

3. A replica waits for $n - f$ matching signatures for its proposal P and aggregates them into a certificate $C := \langle r, d, \{S\} \rangle$ that it broadcasts to all replicas.
4. Upon receiving a proposal certificate a replica adds a node to its local DAG.

Figure 1 illustrates a DAG of certified nodes as it appears in a local view of one of the validators. We note that because proposals are certified, no replicas can produce two conflicting nodes for the same (round, replica) position. Thus, all replicas eventually converge on the **same** local view of the DAG.

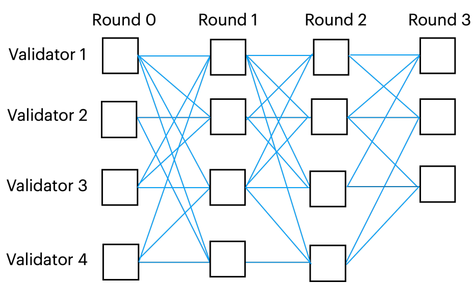

Figure 1: Narwhal's round-based DAG from a validator's local point of view.

3.1.1 Consensus core.

By itself, the DAG is no more than a scalable and robust mempool. It continues to grow at the pace of the network (i.e. *responsively*), and, by virtue of being certified, ensures that all disseminated transaction batches are reliably available. A consensus layer is needed to establish agreement. It can be modularly layered by running an **external** consensus blackbox that commits views of the DAG (e.g., Narwhal+Hotstuff [19]), or logically **embedded** into the existing DAG messages without the need for additional messages (e.g., Tusk [19], Bullshark [35], or Shoal [34]). The rest of the paper focuses on embedded DAG-BFT protocols as they are more efficient.

Embedded consensus. At a high level, consensus is projected onto the DAG structure by designating specific DAG nodes as fixpoints (henceforth *anchors*) that simulate a leader, and interpreting DAG edges as votes for a leader. Upon committing an anchor, all of its causal history can be implicitly committed as well; an ordering can be derived using any deterministic function, e.g. using a topological sort.

Anchor nodes are chosen deterministically in advance (much like leaders in traditional BFT protocols) and placed at regular intervals. Bullshark [35], for instance, uses a round-robin scheme to select anchor candidates and places an anchor candidate per every other round of the DAG. Figure 2 illustrates the anchor commit logic in Bullshark.

An anchor in round r becomes committed as soon as $f+1$ nodes in a round $r' > r$ link to it (**Direct Commit Rule**). The

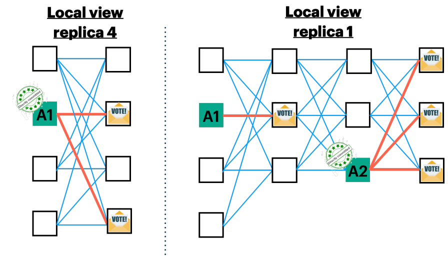

Figure 2: Bullshark Commit example. Replica 4 observes $f+1=2$ votes for anchor A1, and directly commits it. Replica 1 observes only one vote ($< f+1$), and cannot directly commit A1; it commits A1 indirectly upon committing A2 and its causal history, ordering A1 before A2.

safety intuition is straightforward: (i) all nodes are certified, and thus every anchor is equivocation-free, and (ii) all nodes in rounds $r'' > r'$ must include an edge to at least one of the nodes that link to the anchor, and thus the anchor is durable.

In particular, every future committed anchor must link to all previously committed anchors, thereby establishing a consistent total order amongst anchors. Upon committing an anchor, a replica recursively traverses the anchor's causal history, and first commits all preceding anchors (**Indirect Commit Rule**). Conversely, if an anchor candidate is *not* in a committed anchors causal history, then it could never have committed, and it is safe to skip it. Intuitively, the Indirect Commit Rule ensures that, even if a replica does not invoke the Direct Commit rule on some anchors (e.g. because its local DAG view is temporarily out of sync), all correct replicas will agree on the same sequence of committed anchors.

We note that while this design is always safe, it relies on conservative timeouts to ensure liveness (similarly to traditional BFT protocols). Since nodes include only $2f+1$ edges, anchor candidates are not guaranteed to receive the $f+1$ links required to commit. In the worst-case, candidates are repeatedly skipped, thus stalling commit progress.

To reduce the expected latency, Shoal [34] optimizes Bullshark in two ways. First, it leverages the deterministic anchor commit sequence to dynamically re-interpret the anchor schedule upon every commit. Rather than placing anchors every other round, this approach allows for an anchor to be placed every round. Second, it introduces an anchor reputation scheme to attempt to select as candidates only the fastest and well-connected replicas; such candidates are more likely to be included as edges. This allows Shoal to avoid timeouts in all but extremely unlikely scenarios.

3.2 Latency breakdown

We breakdown different sources of latencies present in DAG-protocols, and analyze their respective costs in Bullshark and Shoal. We denote as the e2e consensus latency of a transaction the time between a replica first receives the transaction, and the time that the transaction is ordered. We separate the e2e latency of certified DAG-BFT protocols into three components:

1. **Queuing latency:** the time it takes for a transaction to be proposed in a DAG node.
2. **Anchoring latency:** the time it takes for a non-anchor node's proposal to be "picked up" by an eventually committing anchor.
3. **Anchor Commit latency:** the time it takes to commit an anchor proposal.

Queuing Latency. The DAG advances in a series of synchronized rounds, and each replica may issue at most one proposal (containing a batch of transactions) per round. Each round requires a Reliable Broadcast to certify a proposal, for a total of 3md. Consequently, a transaction that narrowly misses inclusion in the current round proposal must, in the worst-case wait up to 3md before being broadcast; assuming transactions arrive at a uniform rate, the average Queuing Latency is 1.5md.

Anchoring latency. A proposal becomes "anchored" once it is referenced by an eventually committing anchor. Anchoring latency depends on two factors: (i) the frequency of anchor candidates, and (ii) the reliability with which said anchor candidates are committed (i.e. not skipped).[2] Bullshark has anchor candidates in every odd numbered round (i.e., only every second round). Consequently, the best-case inclusion latency of non-anchor proposals in even rounds is 1 round (3md), and 2 rounds (6md) in odd rounds; a best-case average of 4.5md. Shoal increases candidate frequency to every round, and thus improves the inclusion latency of non-anchor proposals to one round for all (3md). The practical reliability of commitment cannot be easily quantified by pen-and-paper, and is best left to empiric analysis as it depends on network configuration and message delivery patterns. Shoal demonstrates that the use of a reputation can aid significantly in ensuring that anchor candidates are reliably included [34].

Anchor Commit latency. In both Bullshark and Shoal, an anchor becomes committed upon either (i) being references by $f+1$ nodes in the next round (Direct Commit Rule), (ii) or being subsumed in the causal history of future committed anchor (Indirect Commit Rule). The Direct Commit Rule requires at least 2 rounds: one to certify the anchor proposal, and one to certify $f+1$ proposals that reference it, for a total of at least 6md. The latency of indirectly committed anchors can be broken down into their own Anchoring Latency to a future anchor, plus the future anchors Commit Latency.

Putting all latency stages together results in a best case average latency of 12md (1.5md + 4.5md + 6md) for Bullshark, and 10.5 for Shoal (1.5md + 3md + 6md).

3.3 A note on uncertified DAGs

A class of recent DAG-based designs [12, 27, 29] proposes to replace DAG nodes with uncertified proposals that are disseminated with best-effort. At first glance, such designs holds promise by reducing latencies across the stack. However, removing certificates makes the DAG brittle: it can introduce a significant amount of unwanted synchronization on the critical path and increase susceptibility to timeout violations [23]. Edges to uncertified, yet locally unavailable proposals must be validated by fetching missing data (at least 2 additional md per round) *before* advancing to avoid losing liveness in the face of Byzantine fabrications or unstable networks. In practice, we find that even occasional message drops can significantly degrade overall latency (§8.3). In contrast, Shoal++ demonstrates that certification is *not* the root-cause of high latencies, and that latency can be significantly reduced without sacrificing robustness.

[2]The proposals of non-anchor candidates are subject to both factors; the proposals of anchors candidates themselves are subject only to the latter.

4 Shoal++ Overview

Shoal++ strives to reduce latency across each of the three latency stages; it employs three different ideas that respectively address Queuing, Anchoring, and Anchor Commit Latency. Together, Shoal++ is able to reduce total e2e expected latency close to an average of 4.5md. We briefly outline each idea in sequence.

Faster anchors. The first observation Shoal++ makes is that the Bullshark Direct Commit rule can be optimized when anchors are referenced by a supermajority of uncertified nodes. Intuitively, a replica may eagerly commit an anchor as soon as $2f+1$ *proposals* that link to an anchor node are observed, as this implicitly guarantees that eventually the Direct Commit Rule is satisfied. Since this is an even stricter condition that may not be fulfilled, Shoal++ (i) retains the existing Direct Commit Rule as backup, and (ii) leverages Shoal's reputation mechanism to reliably trigger the Fast Direct Commit Rule. This allows Shoal++, in practice, to reduce Anchor Commit latency to 4md.

More anchors. Next, Shoal++ takes Shoal's idea of increased anchor frequency one step further, and tries to turn as many nodes into anchors as possible. Doing so requires striking balance. On the one hand, increasing the number of anchors increases the opportunities to avoid incurring the anchoring latency. On the other hand, it introduces also opportunity for more uncommitted anchors, which can stall progress. Adopting the reputation scheme proposed by Shoal can help identify a reliable subset of candidates, but is not enough to streamline progress in practice. Shoal++ (i) slows down round advancement to put the DAG into lockstep and avoid missed anchors, and (ii) dynamically re-interprets the tentative anchor schedule to skip anchors that are no longer needed.

More DAGs. Finally, Shoal++ minimizes Queuing latency by operating not one, but multiple DAGs in parallel. DAGs are staggered using a small offset, allowing waiting transactions that miss a round to quickly be included in the *next* DAG.

This simultaneously improves throughput via better resource utilization: proposals are sent more frequently, batches are smaller and can be sent and processed more efficiently (in a streaming fashion) than less frequent, large batches.

5 Shoal++ In-Depth

Shoal++ uses as starting points the DAG-core of Narwahl [19], and Consensus-core of Bullshark [35] outlined in §3. For the sake of brevity, we do not duplicate the pseudocode that is available in previous works and, in particular, omit a detailed description of the DAG construction (i.e., node broadcasts, transaction submission, node fetching, garbage collection, and so on). We defer to the original papers for an in-depth discussion and correctness proofs.

Algorithm 1 summarizes the DAG API and node structure. We abstract away the node certification process, and the commit logic. PROCESS_NODE and PROCESS_CERTIFIED_NODE respectively involve checking that a node proposal is valid and voting for it, and adding a new confirmed node to the DAG, triggering commit rules. RUN_BULLSHARK performs an instance of the Bullshark commit logic to determine (given a starting point) the first anchor to be ordered (§3.1.1).

Upon committing an anchor, all of its (yet-to-be-ordered) causal history is deterministically ordered as well. Helper functions CAUSAL_HISTORY and GET_ANCHORS aid, respectively, in identifying the causal history of a node (necessary for both commit and ordering logic), and the anchor candidate schedule. Shoal++ adopts and extends Shoal's [34] reputation logic; GET_ANCHORS returns, for each round, a vector of eligible anchors.

Algorithm 1 DAG interface

```
DAG API:
    PROCESS_NODE(v: Node)
    PROCESS_CERTIFIED_NODE(v: Node)
    CAUSAL_HISTORY(v: Node) → Vec<Node>
    RUN_BULLSHARK(anchor: Node) → Node
    GET_ANCHORS(round: int) -> Vec<Node>

struct Node n:
    n.round                    ▷ the associated round in the DAG
    n.source                   ▷ the replica proposing n
    n.parents                  ▷ n−f nodes from round v.round−1
```

Algorithm 2 summarizes the core protocol additions made by Shoal++ in an effort to reduce Anchor Commit, and Anchoring latency. We now discuss both in detail in turn.

5.1 A faster Direct Commit rule

As described in §3.2 it currently takes, in the best case, 6 message delays to commit an anchor using the Bullshark Direct Commit Rule. One DAG round is required to certify the anchor node itself, and a second DAG round is required to collect sufficient votes. A replica can directly commit an anchor upon observing $f+1$ certified nodes that link to the anchor (i.e. include the anchor in their parent set).

Algorithm 2 Core Shoal++ utilities

```
Local variables:
    dag: DAG                              ▷ Initially empty
    weak_votes: Vec< Vec<int> >           ▷ Initially empty
    round: int                            ▷ Initially 0
    anchors: Vec<Node>                    ▷ Initially empty

procedure NEXT_ORDERED_NODES()
    while true do
        if anchors.IS_EMPTY() then
            round++
            anchors ← dag.GET_ANCHORS(round)
        anchor ← anchors.POP()
        anchor_to_order ← select
                              FAST_COMMIT(anchor)
                              dag.RUN_BULLSHARK(anchor)
        if anchor_to_order ≠ anchor then
            SKIP_TO(anchor_to_order)
        return dag.CAUSAL_HISTORY(anchor_to_order)

procedure SKIP_TO(anchor: Node)
    round ← anchor.round
    anchors ← dag.GET_ANCHORS(round)
    anchors.REMOVE(anchor)

procedure FAST_COMMIT(n: Node)
    while true do
        if weak_votes[n.round][n.source] ≥ 2f+1 then
            return n

upon receiving node n do
    r ← n.round
    s ← n.source
    if n is the first node revived from s in round r then
        for each parent ∈ n.parents do
            weak_votes[r][parent]++
        dag.PROCESS_NODE(n)

upon receiving certified node cn do
    dag.PROCESS_CERTIFIED_NODE(cn)
```

Shoal++ makes the following simple observation: if sufficiently many DAG *proposals* link to the anchor, the anchors fate is already set in stone, and we need not wait for round certification to complete. Recall that the certification process consists of three message exchanges, a proposal step, a voting step, and a certificate forwarding step (§3). Shoal++ allows replicas to FAST_COMMIT (Alg. 2) as soon as it observes $2f+1$ proposals that link to an anchor (Fig. 3). To do so, replicas simply additionally keep track of *weak votes* for each round that correspond to the proposals observed. Weak votes are not guaranteed to survive certification. For instance, a faulty proposer may equivocate, and produce a node certificate containing a different proposal that does not link to the anchor. However, out of any $2f+1$ weak-votes, at least $f+1$ must be cast by correct proposers that will never equivocate. Consequently, the presence of $2f+1$ weak-votes guarantees that eventually

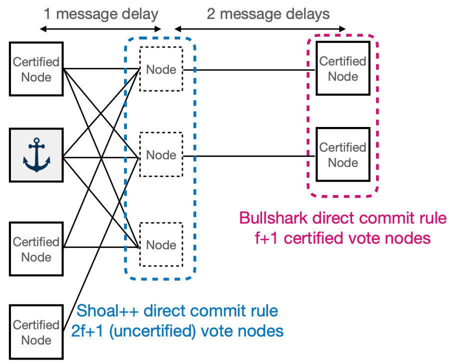

Figure 3: Bullsharks's direct commit rule requires f+1 certified vote nodes, while Shoal++'s direct commit rule requires $2f+1$ (uncertified) node proposals.

$f+1$ certificates linking to the anchor must be formed, thus ensuring safety. Using the Direct Fast Commit rule allows replicas to commit an anchor in only 4md: 3md to certify the anchor, and 1md to broadcast and receive proposals.

We note, however, that this rule places a *stronger* requirement for liveness; f additional proposals are (tentatively) required to link to an anchor. We thus retain the existing Direct Commit rule as backup, and allow replicas to commit using whichever rule is satisfied first. In some cases, it may be faster (and more reliable) to commit using $f+1$ certified nodes. For instance, when the network is unstable or point latencies are asymmetric, the fastest $f+1$ replicas might advance much more quickly than the fastest $2f+1$.

In our empirical evaluation, we find that this case is rare (§8) and that replicas are, most of the time, able take advantage of the Fast Direct Commit rule. Shoal's reputation scheme ensures that eligible anchor candidates are typically well connected, making it exceedingly commonplace for future round proposals to link to an anchor.

5.2 Increasing anchor frequency

In order to reduce Anchoring latency, Shoal++ tries to dynamically designate as many nodes as possible as anchors. Intuitively, if all nodes were anchors that become committed, no node would experience any Anchoring latency.[3] Safely operating more than one anchor per round requires additional orchestration. To ensure a consistently ordered log across all correct replicas, parallel anchors must be committed in a predefined order. One easy way to visualize parallel orchestration is by mapping anchors to fixed slots; much like parallel proposals in traditional BFT protocols such as PBFT [15]. Figure 4 shows a simple example in which there are four anchors in one round (all n nodes are anchors). Each anchor employs its *own* consensus instance that may proceed in parallel, and out of order, but anchors can be committed only in sequence.

[3] We note that such an approach is briefly discussed in the closing remarks of Shoal [34], but is neither implemented nor carefully evaluated for practical considerations.

Each consensus instance operates a one-shot Bullshark instance, augmented with the Fast Direct Commit Rule. We remark that each Bullshark instance operates its own tentative anchor schedule (e.g. the yellow anchor in Fig. 4 is considered the next anchor in the instance started by the red anchor), but terminates as soon as the initial anchor is resolved.

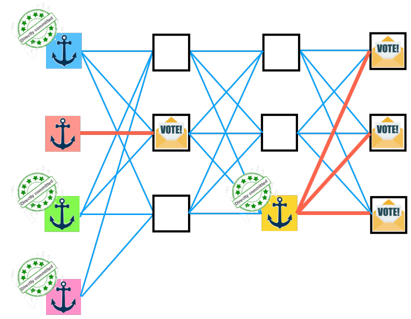

Figure 4: An illustration with four anchors in round 1, using the pre-assigned order: blue, red, green, pink. The blue anchor commits directly; it has $\geq f+1$ references from nodes in round 2. The red anchor is only indirectly committed upon directly committing the yellow anchor in round 4. The green and pink anchors must wait to be ordered until the red anchor is resolved.

While simple, a straightforward instantiation of this design is not robust, and may, in practice negate (or even hurt) any latency benefit. By treating every node as an anchor, progress is, in particular, bottlenecked on the *slowest* anchor. We illustrate such an unwelcome scenario in Figure 4: although the green and pink anchors have sufficient support to commit, they must wait until the preceding red anchor becomes committed or skipped. Unfortunately, the red anchor is slow and does not have any immediate links; it is indirectly committed only two rounds later via the next anchor in its consensus instance (yellow anchor). In worse circumstances, an anchor might never commit. In this case, we must wait for confirmation that the anchor is skipped, and fill its assigned slot with a no-op. Such scenarios are not fringe outliers, but expected by design: since DAG nodes include only $2f+1$ edges, rounds are privy to advancing based on the fastest $2f+1$ nodes, and leaving the remainder behind. Thus, with high probability, *at least* one anchor per round will not receive sufficient direct votes.

Adopting the reputation mechanism proposed by Shoal can help ameliorate, but not remedy the concern. We leverage reputation and modify GET_ANCHORS(r) to limit the set of eligible anchors per round only to those associated with historically fast and well-connected replicas. Unfortunately, we find that in practice leveraging reputation alone is insufficient, and does not reliably enough exclude slow replicas. While past performance is a helpful indicator, network instabilities in geo-distributed systems make it hard to accurately predict *future* performance, and we observe that the set of "slow" replicas changes very dynamically. To improve robustness further, we add two additional techniques that, respectively, try to (i) avoid unnecessarily classifying nodes as slow, and (ii) dynamically skip likely obsolete anchors.

Round Timeouts. Reputation alone is overly harsh in categorizing nodes as slow. Since DAGs advance with the fastest $2f+1$ replicas, to maximize the likelihood of anchors that commit directly, reputation *pessimistically* caps the eligible anchors per round to include only the (estimated) fastest $2f + 1$ nodes. (i.e. $\frac{2}{3}$ of total nodes). In reality, however, ineligible nodes may only be fractionally slower, and arrive imminently. To avoid discriminating against such nodes, Shoal++ makes the conscious choice to, in each round, additionally wait for a small timeout beyond the first $2f+1$ nodes observed, and include any momentarily received nodes as additional edges. While this slightly slows down individual rounds, it encourages replicas to operate in lockstep, resulting in a more densely connected DAG. This allows us, in turn, to optimistically increase the set of eligible anchors back to include all n nodes, thus reducing *overall* latency.

We note that timeouts are not necessary for liveness, and are purely optimizing performance. They can thus be configured very aggressively, and are, in practice, negligible compared to the 3 network delays required to certify a node.

Skipping Anchor Candidates. Although round timeouts increase the reliability of the Direct Commit Rule, they cannot guarantee its success. Occasionally, replicas may fail or experience high networking delays, resulting in missing, or unconnected nodes. In such cases, all subsequently pre-ordered anchors must wait until the current anchor is resolved. Recall that an anchor $A1$ in round r is only resolved only upon committing, in its local consensus instance, an anchor $A2$ in a future round $r' > r$: $A1$ is committed if it is present in the causal history of $A2$ (Indirect Commit), or skipped if it is not. In practice, we expect indirectly committed anchors to be rare. Anchors that are merely slow, are likely to be committed thanks to the round timeout. Crashes, in turn, predominantly result in skips. Byzantine abuse can unfortunately not be avoided, and we must rely on the reputation scheme to avoid such anchors; fortunately, such faults are exceedingly rare in most practical deployments.[4]

We note that, even though $A2$ has been committed, it cannot be ordered before all preceding anchors in the pre-defined schedule have resolved their respective consensus instances (i.e. commit or skip). This is undesirable: such anchors are (i) likely already included in the causal history of $A2$, and (ii) may be subject to further latency delays. Unfortunately we cannot avoid this scenario if $A1$ is committed; another replica may have committed it directly, and resolved (and applied) the output of the next scheduled anchor $B1$. We can, however, sidestep the resolution of preceding anchors' Bullshark instances if $A1$ is skipped. For any given consensus instance, all replicas necessarily agree on the order of *all* committed and skipped anchors. Thus, in particular, they agree on the *first* committed anchor (in this case $A2$, since $A1$ is skipped). Let the round of this anchor be r'. Shoal++ opts to *skip* all tentative anchor candidates in rounds $< r'$.

[4]Blockchain systems [42, 44] report observing zero Byzantine faults in over a year of deployment. Reported faults are typically benign crashes.

Instead of pre-assigning anchors to fixed slots, Shoal++ considers all but the first anchor in each round *virtual*, and dynamically materializes them as it sees fit. At any given time, there is only a single materialized consensus instance (with an anchor every other round); upon resolving the current instance, Shoal++ re-evaluates where to sensibly place the next anchor. Only then it begins evaluating the associated consensus instance. Since all replicas resolve each instance identically, each locally re-interprets and materializes the same anchor schedule. Consider a slight modification of the example in Figure 4, in which the yellow anchor does **not** extend the red anchor. The blue anchor commits directly, and Shoal++ dynamically materialized the red node as the next anchor to evaluate. The red anchor, however, will be resolved as skipped upon committing the yellow anchor. Shoal++ thus skips evaluating the virtual green and pink anchors, and instead tries to materialize as anchor the first node *after* the yellow anchor.

5.3 Operating multiple DAGs in parallel

Finally, in an effort to minimize Queuing latency, Shoal++ chooses to operate k concurrent DAG instances, and interleave their outputs to construct a single, totally ordered log. Intuitively, staggering multiple DAGs, in a pipline-like manner, allows transactions to be proposed quicker. Instead of waiting (up to 3md) for the next DAG round to begin, a transaction simply joins the next round of the *next DAG*. This reduces the expected Queuing latency from 1.5md to $1.5/k$ md. In practice, we deploy Shoal++ with three DAGs, each offset by one message delay. Since each DAG round requires 3md to certify a node proposal, this ensures that a some DAG's proposal is available every 1md. We find that 3 DAGs offer a good sweetspot between theoretical Queuing latency and batch sizes; additional DAGs offer diminishing returns as proposals carry increasingly smaller batches.

Algorithm 3 summarizes the algorithm. Each DAG instance runs as a black-box, isolated from the others. In order to create a consistent total order, we must interleave their outputs. Each time a DAG commits an anchor, it outputs a new log segment.Shoal++ rotates across DAGs in round-robin fashion, and appends, in each round, exactly one available segment to the log. If, for whatever reason, one DAG commits more frequently, then its excess available segments must wait to be ordered until the other DAGs to make progress. Importantly, the individual DAGs themselves never block, and progress unimpeded. We note that segments also may not (and need not) be of equal size; each segment corresponds to a newly committed anchor, each of which may order varying sizes of causal histories.

Operating additional DAGs incurs additional messages and associated processing overhead. However, we find that in practice, this does not negatively affect performance.

Algorithm 3 Shoal++

```
Local variables:
    d_1,d_2,d_3: DAG instances                 ▷ Initially empty
loop
    for i=1 to 3 do
        output d_i.NEXT_ORDERED_NODES()
```

In fact, throughput increases because smaller batches are proposed more frequently which helps amortize bandwidth and processing, resulting in better resource utilization. In theory, messages from different DAGs can be overlayed in order to share signatures. For instance, a node-proposal of DAG $i \in \{0,1,2\}$ can easily be combined with the certified-node-broadcast message of DAG $(i+1)\%3$.[5] This too, we find not to be worth it in practice as (i) signatures are not the bottleneck, and (ii) message stages have assymetric latencies, and artificially aligning them increases Queuing latency.

5.4 Discussion

The combination of the above techniques allows Shoal++ to, in the common, fault-free case, reduce average e2e latency to only 4.5md: 4md to commit anchors, and 0.5md queuing latency. In the presence of faults or bad network behaviors, additional anchoring and commit latency will be incurred to resolve anchors via the Indirect Commit/Skip Rule.

We note that Shoal++ uses a *linear* star-based communication pattern to ceritfy nodes, while protocols such as PBFT [15] leverages *quadratic* all-to-all communication. If desired, Shoal++ can adopt all-to-all communication, reducing latency by 1md (to 3.5md total) at the cost of increased message complexity. We also remark that although Shoal++ incurs a small additional queuing latency, every replica acts as proposer, and clients need only contact their *local* replica (in absence of faults). Single leader designs, in contrast, require clients to contact possibly remote leaders, introducing additional "queuing" latency. Finally, we note that Shoal++ consumes more resources—such as CPU, memory, and disk space—than Shoal due to the multiple DAGs technique. However, this approach enables Shoal++ to reduce queuing latency and makes better use of the available network bandwidth.

6 Correctness

We now prove the Safety and Liveness of Shoal++. We separately prove each augmentation Shoal++ introduces through a reduction to the Safety and Liveness proofs of Bullshark [35] and Shoal [34]. To formalize the arguments, our proof relies on the following three properties:

Property 1. *For any node n, the call* CAUSAL_HISTORY*(n) returns the same vector of nodes to all replicas.*

Property 2. *All replicas commit the same anchors, and in the same order.*

Property 3. *For any given round r the leader reputation mechanism ensures that* GET_ANCHORS*(r) return the same vector of nodes at all replicas.*

Property 1 holds true for all Narwhal-based DAGs, and follows directly from the fact that all nodes in the DAG are certified; Properties 2 and 3 follow, respectively, from the safety proofs of Bullshark and Shoal.

Fast Direct Commit Rule. Since Shoal++ can commit using both the Fast Direct Commit Rule and the existing (Bullshark) Direct Commit Rule, Liveness follows directly from Bullshark. We prove that the Fast Direct Commit rule is safe:

Lemma 1. *For any anchor a, the procedures* FAST_COMMIT*(a) and* RUN_BULLSHARK*(a) never return contradictory values.*

Proof. By the code, FAST_COMMIT(a) can only return a. Thus we need to prove that if FAST_COMMIT(a) triggers, then RUN_BULLSHARK(a) also returns a. FAST_COMMIT(a) implies that the replica received $2f+1$ (uncertified) node proposals from different replicas that reference a. Since there are at most f faulty replicas, at least $f+1$ of these proposals were broadcasted by correct replicas and will eventually be certified. Since $f+1$ such certified nodes exist, some replicas may directly commit the anchor a via the Bullshark logic. It follows from P.2 that in this case, the RUN_BULLSHARK(a) call returns a to all replicas. □

Multiple Anchors per Round. Liveness follows directly from Bullshark. We show that Shoal++'s dynamic anchor mechanism retains Safety.

Lemma 2. *We assume all correct replicas are initialized with the same state (*round *and* anchors*). Then, all correct replicas' invocations of* NEXT_ORDERED_NODES *(Alg 2) return the same result.*

Proof. By the lemma assumption and Property 3, anchors.POP() returns the same anchor in the first iteration of the while loop to all replicas (Line 6). By Lemma 1, all correct replicas agree on which anchor to order (Line 7). It follows, from the determinism of NEXT_ORDERED_NODES and P.1 that all replicas return the same vector of nodes to be ordered in their first NEXT_ORDERED_NODES call (line 12). In addition, by the determinism of NEXT_ORDERED_NODES, all replicas end up with the same state (*round* and *anchors*) after the calls return. Therefore, by applying the argument above inductively, it is easy to show that all replicas also return the same vector of nodes in all future calls. □

Theorem 1. *The Shoal++ protocol Algorithm 3 satisfies Safety and Liveness.*

[5] In theory, votes can also be overlayed, though typically it is favorable to send them as fast as possible, rather than synchronizing.

The Liveness and Safety proof of each DAG is given above. Since DAG outputs are interleaved deterministically, Safety and Liveness of the composed system follow immediately.

7 Practical Considerations

We briefly discuss some implementation considerations.

Inline data streaming. Narwahl [19] proposes, in addition to the DAG construction, to further decouple, and horizontally scale data dissemination using an additional "worker"-layer. Workers disseminate batches optimistically a-priori, and DAG proposals contain only hash references. This construction is extremely throughput-friendly in the best case, as data is streamed continuously, and consensus carries only metadata. In practice, however, much like with uncertified DAG-BFT protocols [12, 27], other replicas might not have the data behind the hashes (or the references in the case of uncertified DAGs) when the proposal is received and thus additional 2md latency is required for fetching. Shoal++ opts to forgo the worker layer in order to avoid this latency, and chooses to disseminate transaction batches inline with DAG proposals. Operating staggered DAGs allows Shoal++ to nonetheless achieve throughput competitive to that of Narwahl (with a single worker), as smaller batches are sent more frequently, thus amortizing bandwidth close to a streaming design.[6] Finally, we note that inline data streaming simplifies data fetching as it avoids an additional indirection needed to dereference digests.

Efficient fetching. DAG edges in Shoal++ contain only certified nodes, allowing Shoal++ to efficiently fetch missing nodes and resolve inconsistencies across replicas. Replicas whose DAG views are out of sync, may validate and vote on proposals *without* locally observing the proposed node's causal history. This streamlines the certification process, and ensures that the DAG operates at predictable rates. Any missing causal histories can be inferred asynchronously, off the critical path. We remark that this is not possible in uncertified DAGs; to ensure liveness, replicas must fetch all missing data *before* processing a new node. This results in unsteady processing times, and may significantly increase e2e latency. Certified edges, additionally allow us to balance data fetching load: since at least $f+1$ correct replicas must have observed any certified node, replicas missing the node may request it from different replicas to balance load.

Distance-based priority broadcast. We observe that in systems with large n, message broadcasting consumes considerable CPU times. When sending messages in a fixed order (e.g. first to $R1$, then to $R2$, ...) can unintentionally cause replicas to consistently receive messages with delay, making their nodes subject to falling behind in the DAG. To circumvent this issue, Shoal++, periodically records point latencies between replicas, and adjust broadcast send-orders to prioritize farther away replicas. This results in a more balanced message distribution, increasing the homogeneity of node certification pace.

[6] Scaling throughput by employing more workers is typically not worth it in existing BFT deployments, as single-worker ordering throughput in existing protocols already far exceeds execution bandwidth.

8 Evaluation

Our evaluation seeks to answer three questions:

1. **Latency reduction:** How much does Shoal++ reduce Latency compared to state-of-the-art alternatives?
2. **Latency breakdown:** How much does each of the augmentations introduced in Shoal++ contribute to the latency reduction?
3. **Robustness:** How well does Shoal++ tolerate faults?

We implemented Shoal++ into an open-source code base of Aptos, one of the major blockchain projects. Our prototype is written in Rust and utilizes the Tokio [10] asynchronous runtime. It uses BLS [14] implemented over BLS12-381 curves for signatures, RocksDB [9] for persistent storage of consensus data, and Noise [8] for authentication. We disable the execution and ledger storage components of the original blockchain in order to isolate consensus performance. Each data point represents the 50 percentile (median) with error bars representing the 25, and 75th percentiles.

Baselines. We compare Shoal++ against three popular high throughput DAG-BFT consensus protocols, Bullshark [36], Shoal [34], and Mysticeti [12], as well as one traditional low-latency BFT protocol, Jolteon [22]. Bullshark and Shoal represent state-of-the-art *certified* DAG-based protocols, while Mysticeti [12] is a concurrent work to Shoal++ that proposes an *uncertified* design to reduce latency. Jolteon [22] is a state-of-the-art leader-based BFT protocol, which improves Hotstuff [47] latency by 50%. A variant of Jolteon is currently deployed on Aptos [42]. Sui [44] recently replaced Bullshark with a Mysticeti deployment. For an apples-to-apples comparison, we re-implemented Bullshark, Shoal, and Jolteon according to the papers' description in the same codebase as Shoal++. For Mysticeti we run the publicly available source code [7] referred to in the latest version of the paper; its prototype too is written in Rust, using Tokio, but forgoes writing consensus data to persistent storage, making it less production-realistic. To minimize latency, we opt to disable the Narwhal [19] worker layer for both Bullshark and Shoal. However, for a fair throughput comparison with Shoal++, we additionally augment both systems with Shoal++'s proposed parallel-DAG strategy ("Bullshark/Shoal More DAGs", Fig. 5).

Experimental setup. We use the Google Cloud Platform [4] to mimic the deployment of a globally decentralized network. Our testbed consists of 100 replicas spread evenly across 10 regions around the world: we choose two regions in the US (us-west1 and us-east1), two in Europe (europe-west4

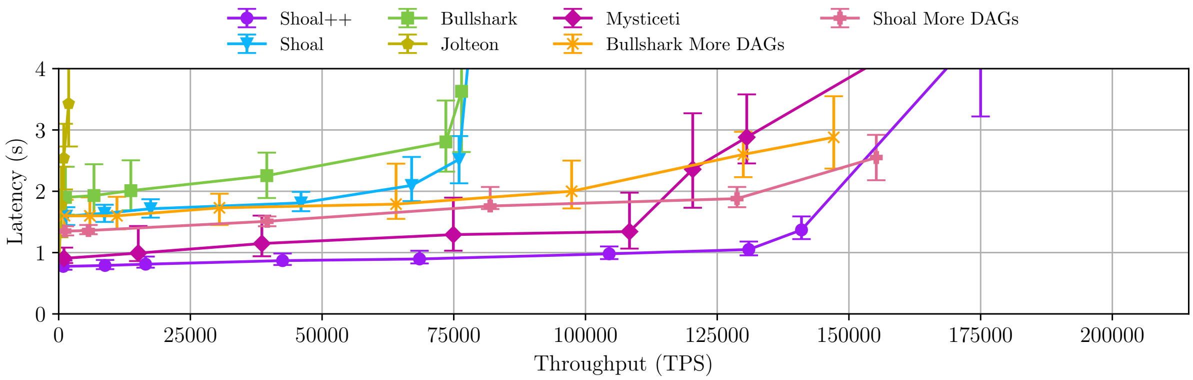

Figure 5: No failures with 100 replicas.

and europe-southwest1), three in Asia (asia-northeast3, asia-southeast1, and asia-south1), and, respectively, one each in South America (southamerica-east1), South Africa (africa-south1) and Australia (australia-southeast1). The round-trip times between these regions range between 25ms and 317ms. We use `n2d-standard-64` virtual machines, containing 64 vCPUs and 256 GB of memory [3]. We provisioned a 2TB network attached disk to each machine to guarantee enough IOPS for persistence. This spec is similar to those used by production blockchains and qualifies as commodity grade.

Clients connect to a single (local) replica and issue a continuous stream of dummy transactions (310 random bytes). As is standard in blockchain deployments, we measure consensus latency as the time between when a transaction first arrives at a replica, and when the transaction is ordered and ready to be executed. We use a batch size of 500 transactions across all systems. In all DAG-based systems, each node proposal contains one batch; Jolteon proposals may contain up to 100 batches. We configure Jolteon to use a 1.5s timeout for view changes (the standard in existing production systems [42]), and use a round timeout of 600ms for Bullshark, Shoal and Shoal++; Mysticeti by default uses a 1s round timeout. We remark that timeouts in Bullshark, Shoal and Shoal++ serve different functions. Bullshark requires them for liveness, while Shoal manages to sidestep using timeouts for liveness all but edge cases; Shoal++ follows Shoal, but re-introduces a timeout only for improved performance. We note that the timeout is started at the *beginning* of a round, and thus typically introduces only a small additional wait past observing the first $2f+1$ certified nodes.

8.1 Failure-free case

Figure 5 illustrates throughput and latency under failure-free conditions. Shoal++ is the only system to sustain sub-second latency for throughput of 100k transactions per second (tps).

Jolteon offers low latency (900ms) at low load, but is unable to scale throughput beyond 2100 tps as it is bottlenecked by the leader's network bandwidth. Bullshark and Shoal support significantly higher throughput (up to 75k tps) but incur high latency: 1.9s and 1.45s, respectively, at low load, and 2.4s and 1.7s, respectively at 50k tps. This follows directly from their high average number of required message delays (§3.2). Throughput scalability in both systems is bottlenecked by network utilization.

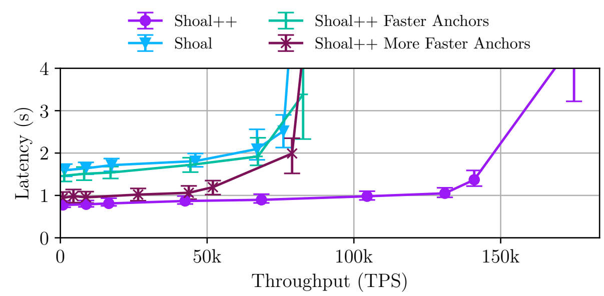

Figure 6: Shoal++ Breakdown. No failures with 100 geo-distributed replicas.

Shoal++ significantly reduces latency (775ms at low load), while simultaneously improving throughput scalability (up to ca. 140k tps). The parallel DAG construction allows Shoal++ to better utilize network and storage resources: rather than broadcasting and processing a large batch once per round, it broadcasts and processes three smaller batches at the same time. Applying the same technique to Bullshark and Shoal ("More DAGs") allows both systems to match the throughput of Shoal++ (while simultaneously improving queuing latency), without incurring the latency penalty of a Narwahl-style worker layer.

Mysticeti matches the throughput of Shoal++, but fails to match its latency, even at moderate loads. At high loads (upwards of 100k tps) Mysticeti notably incurs a signficant latency penalty as replicas may be temporarily out of sync, and must fetch missing data on the critical path of consensus.

8.2 Latency improvement breakdown

Figure 6 isolates the respective latency improvement of each Shoal++ augmentation compared to the baseline, Shoal.

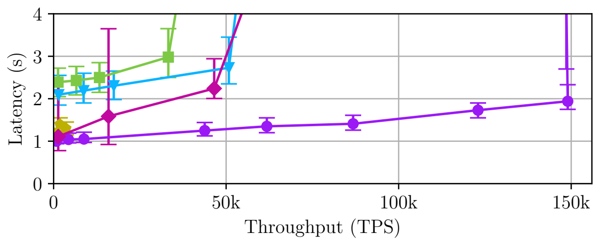

Figure 7: Latency/Throughput graph with 33 out of 100 crashed replicas.

Shoal++ *Faster Anchors* shows Shoal augmented only with the Fast Direct Commit Rule while Shoal++ *More Faster Anchors* additionally applies the multiple anchor augmentation. Finally, Shoal++ includes also the parallel DAG optimization.

The Fast Commit rule, in theory, improves anchor commit latency by up to 2md. In real, asymmetric network settings, however, it is not always 2md faster than the Direct Commit rule, and thus the practical latency improvement is smaller. The multiple anchor augmentation, in turn, improves latency significantly: (i) it saves, on average, 3md by eliminating the anchoring latency for most nodes (those that become anchors), and (ii) ensures that the anchoring latency for non-anchor nodes is typically only one DAG round because (with overwhelming probability) nodes are referenced by at least one of the (many) anchors in the next round. The parallel DAG framework improves latency further, by reducing queuing latency, but more importantly improves throughput scalability by approaching a data "streaming" effect; crucially, and unlike Narwhal-based worker designs, it does so *without* incurring additional latency.

8.3 Failure case

To quantify the robustness of Shoal++, relative to the baseline systems, we evaluate performance under two types of disruptions. First, we measure the performance of all systems while simulating crash failures of 33 (out of 100) replicas (Fig. 7). The performance of Jolteon remains largely unaffected, as its leader reputation mechanism quickly detects failures and subsequently elects only alive replicas. Shoal and Shoal++, likewise, swiftly adjust and elect only live anchors; latency, however, increases (up to ca. 2x at high load) as quorums must span more regions, increasing tail latency. Bullshark and Mysticeti, in contrast, suffer drastically increased latency since they do not employ a leader reputation mechanism to elect suitable anchor candidates, and thus must "wait" to commit until a non-faulty replica is elected as anchor.

Second, we measure the impact of sporadic message drops for Shoal++ and Mysticeti (Fig. 8). We inject network layer message drops for 1% of egress traffic in 5 nodes (out of 100 total), beginning at 60 seconds (red line). Under normal network conditions and a low-moderate load (18k tps) both Shoal++ and Mysticeti offer 700ms median consensus latency. Upon failure injection the latency of Mysticeti rises

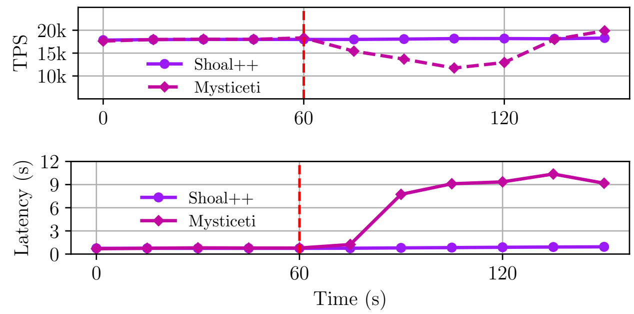

Figure 8: Impact of message drops in Certified vs Uncertified DAGs.

sharply (by a factor of 10x) as replicas scramble to perform critical-path synchronization on missing data (resulting, at times, in timeouts). As latency spikes, throughput initially drops as well, but recovers once the backlog of disseminated DAG proposals commit. Shoal++, in contrast, remains largely unaffected by message drops (latency rises to at most 1.3x). Because all nodes are *certified*, the DAG construction process in Shoal++ proceeds smoothly; any required synchronization is asynchronous and off the critical path.

9 Related work

Latency optimal BFT. Spurred by the seminal PBFT [15] protocol, a long line of systems [22, 24, 28, 30, 39] optimize for consensus latency. PBFT-based protocols designate a single leader to act as sequencer, and replace it only sporadically (e.g upon failure, or eagerly for fairness). This approach enjoys excellent latency. PBFT requires, when led by a correct replica, only 3 message delays (md); the optimum achievable latency in presence of faults, and optimal resilience of $n = 3f + 1$. Several followup works suggest optimistic Fast Paths [24, 28, 30, 39] that improve latency to only 2md in absence of faults, or by weakening resilience to $n = 5f + 1$. Unfortunately, single leader-based designs are fundamentally bottlenecked in throughput by the bandwidth and processing capacity of a single replica.

High throughput BFT. The key idea to scale beyond this bottleneck was realizing that networking resources must be fully utilized. Some works propose to decouple data dissemination from the consensus logic [11, 19, 46], while others [25, 37, 38] propose multi-log frameworks: these systems partition the request space and operate multiple parallel black-box instances of consensus protocols (e.g. PBFT) – each led by a different replica –, and carefully intertwine their outputs to construct a single, totally ordered log. This approach can achieve high throughput but requires complex coordination to deal with failures and re-configuration.

DAG BFT. More recently, a new class of so-called DAG-BFT protocols have emerged as the popular choice for high throughput BFT. At a high level, deployed DAG-BFT protocols propose to explicitly separate data dissemination from consensus, and utilizing only a single common DAG data structure to implement consensus (§3).

HashGraph [13] was the first to propose a DAG-based consensus structure, and Danezis et al. [32] showed that any deterministic BFT protocol can, in theory, be embedded on a DAG. Aleph [21] and DAG-Rider [26] were the first to introduce additional rigidity to the DAG construction, leveraging a round-based structure to design asynchronous BFT protocols. Blockmania [18] was the first attempt to implement a DAG-BFT system, but fell short of adoption as it inherited a complex view change procedure, notorious to traditional BFT protocols such as PBFT.

Narwhal and Tusk [19] catapulted DAG-BFT to popularity by demonstrating that (i) (asynchronous) consensus can be embedded onto DAG-structures without the need for explicit view changes (Tusk), and (ii) throughput can be scaled nearly linearly through the addition of a horizontally scalable worker layer (Narwahl Mempool). Bullshark [35] improved upon Tusk by adopting partial synchrony and introducing a common case fast path during synchronous intervals; in a future iteration [36] Bullshark removed the asynchronous fallback, and established itself as the de facto partially synchronous DAG-BFT protocol.

Bullshark designates, at regular intervals (every two rounds), DAG-nodes to simulate a leader (anchor) akin to traditional BFT protocols, and commits them upon observing DAG patterns that ensure durable agreement of the leader. We refer to §3 for an overview of the Narwahl core DAG, and the partially synchronous Bullshark consensus protocol. Shoal improves the latency of Bullshark by (i) increasing anchor frequency to every round, (ii) utilizing a deterministic reputation scheme to select as anchors only the fastest, and best-connected replicas, and (iii) eliminating leader/anchor timeouts in all but extremely unlikely scenarios. Shoal, in a closing remark, briefly discusses the idea of multiple anchor per round; however, it does not provide detailed consideration.

Uncertified DAGs. A recent group of protocols propose the design of uncertified DAGs. Such constructions can, in theory, reduce latency, but are increasingly brittle as they are prone to data fetching on the critical path (§3.2). Cordial Miners [27] replaces the Reliable Broadcast (RB) in each round with best-effort broadcast (BEB), allowing anchors to (in the best case) commit within 3md; anchors are placed every other round, for an average latency of 4.5md. Similarly, BBCA-chain [29] replaces non-anchor nodes with BEB, and strengthens anchor nodes to single-shot pbft instances. Unfortunately, neither proposal has an implementation, and thus it is impossible to empirically evaluate practical performance.

Concurrent Latency reduction efforts. In concurrent efforts to Shoal++, several recent (non-peer reviewed) protocols suggest latency improvements, underlining the demand for high throughput systems with lower latency.

Sailfish [33] introduces an anchor commit rule that resembles the Fast Direct Commit Rule proposed in Shoal++. However, unlike Shoal++, it does not leverage the Bullshark Direct Commit rule as a potentially faster alternative. Furthermore, Sailfish does not employ the "more anchors" and "more DAGs" techniques, which enable Shoal++ to further reduce latency.

Mysticeti improves upon Bullshark's best case latency by transitioning to an uncertified DAG and implementing the Cordial Miners [27] consensus protocol. Mysticeti, like Shoal++, adopts Shoals high-level proposal to optimistically employ multiple anchors per round. Our evaluation demonstrates that Mysticeti almost matches Shoal++'s latency in the common case. However, as shown in §8, and independently corroborated by Autobahn [23], uncertified DAG constructions are prone to data fetching on the critical path, which negates the suggested latency benefits. A single Byzantine (or slow) replica, for instance, can impose at least 2 additional message delays *per* round in Mysticeti by not fully (or timely) disseminating its node proposals. In §8 we observed 10x latency degradation when 5 out of 100 nodes experienced 1% message drop.

Autobahn [23] introduces a promising DAG-*free* consensus approach in which replicas efficiently and reliably disseminate data independently in parallel "data lanes", and consensus commits only snapshot cuts of the lane states.

Leader reputation in BFT. To the best of our knowledge, leader reputation for BFT systems was first implemented in DiemBFT [43], and formalized by Carousel [17]. Shoal [34] and later Hammerhead [45] adopt leader reputation to DAG-BFT in an effort to exclude underperforming leaders from anchor candidacy. Shoal++ extends the approach to provide, in each round, a set of anchor candidates.

10 Conclusion

This work introduces Shoal++, a DAG-based BFT consensus protocol that offers high throughput, while significantly lowering e2e latency compared to current state of the art DAG-BFT protocols. Shoal++ takes as its starting point existing state of the art DAG-BFT protocols, and augments them in three key ways: (i) it reduces, in the common case, the commit latency of leaders (anchors), (ii) dynamically reinterprets anchor schedules to increase anchor frequency, and (iii) operates multiple parallel DAG instances to increase proposal frequency. This, in sum, allows Shoal++ to reduce the fault-free average e2e latency from 10.5 message exchanges (Shoal [34]) to only 4.5 message exchanges, within reach of traditional latency optimal (but throughput-inefficient) BFT protocols such as PBFT.

We note that the parallel DAG instances technique might be of independent interest, as this idea can be applied to any BFT algorithm to reduce its queuing latency.

References

[1] [n. d.]. Accolade Gathers $202 Million for New Blockchain Fund. https://arc.net/l/quote/srjmeosq Accessed: March 2025.

[2] [n. d.]. The Digital Euro. https://www.ecb.europa.eu/euro/digital_euro/html/index.en.html Accessed: May 2024.

[3] [n. d.]. GCP n2d machines. https://cloud.google.com/compute/docs/general-purpose-machines Accessed: May 2024.

[4] [n. d.]. Google Cloud Platform. https://cloud.google.com/ Accessed: May 2024.

[5] [n. d.]. Janus Henderson to follow BlackRock and Fidelity into tokenisation. https://arc.net/l/quote/gvxzbiuv Accessed: March 2025.

[6] [n. d.]. Microsoft Confidential Consortium Framework. https://www.microsoft.com/en-us/research/project/confidential-consortium-framework/ Accessed: May 2024.

[7] [n. d.]. Mysticeti codebase. https://github.com/asonnino/mysticeti/tree/paper Commit 96fd831.

[8] [n. d.]. Noise Cyrpto Framework. https://github.com/noiseprotocol/noise_spec Accessed: May 2024.

[9] [n. d.]. RocksDB. https://rocksdb.org Accessed: May 2024.

[10] [n. d.]. Tokio: Rust Asynchronous Runtime. Accessed: May 2024.

[11] Balaji Arun and Binoy Ravindran. 2022. Scalable Byzantine Fault Tolerance via Partial Decentralization. *Proc. VLDB Endow.* 15, 9 (May 2022), 1739–1752. https://doi.org/10.14778/3538598.3538599

[12] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. 2023. Mysticeti: Low-Latency DAG Consensus with Fast Commit Path. *arXiv preprint arXiv:2310.14821* (2023).

[13] Leemon Baird. 2016. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep* (2016).

[14] Dan Boneh, Benjamin Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *Advances in Cryptology - CRYPTO 2001 (Lecture Notes in Computer Science, Vol. 2139)*. Springer, 514–532. https://doi.org/10.1007/3-540-44647-8_32

[15] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)* 20, 4 (2002), 398–461.

[16] Jing Chen and Silvio Micali. 2019. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science* 777 (2019), 155–183.

[17] Shir Cohen, Rati Gelashvili, Lefteris Kokoris Kogias, Zekun Li, Dahlia Malkhi, Alberto Sonnino, and Alexander Spiegelman. 2022. Be aware of your leaders. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*. Springer, 279–295.

[18] George Danezis and David Hrycyszyn. 2018. Blockmania: from block dags to consensus. *arXiv preprint arXiv:1809.01620* (2018).

[19] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 34–50.

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

[21] Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. 2019. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 214–228.

[22] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*. Springer, 296–315.

[23] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. 2024. Autobahn: Seamless high speed BFT. *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (2024).

[24] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 568–580.

[25] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. Rcc: Resilient concurrent consensus for high-throughput secure transaction processing. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1392–1403.

[26] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All You Need is DAG. In *Proceedings of the 40th Symposium on Principles of Distributed Computing (PODC '21)*.

[27] Idit Keidar, Oded Naor, and Ehud Shapiro. 2022. Cordial Miners: A Family of Simple, Efficient and Self-Contained Consensus Protocols for Every Eventuality. *arXiv preprint arXiv:2205.09174* (2022).

[28] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 45–58.

[29] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. 2023. BBCA-CHAIN: One-Message, Low Latency BFT Consensus on a DAG. *arXiv preprint arXiv:2310.06335* (2023).

[30] J-P Martin and Lorenzo Alvisi. 2006. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (2006), 202–215.

[31] Team Rocket. [n. d.]. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. ([n. d.]).

[32] Maria A Schett and George Danezis. 2021. Embedding a deterministic BFT protocol in a block DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 177–186.

[33] Nibesh Shrestha, Aniket Kate, and Kartik Nayak. 2024. Sailfish: Towards Improving Latency of DAG-based BFT. Cryptology ePrint Archive, Paper 2024/472. https://eprint.iacr.org/2024/472.

[34] Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. 2024. Shoal: Improving DAG-BFT latency and robustness. In *International Conference on Financial Cryptography and Data Security*, Jeremy Clark and Elaine Shi (Eds.). Springer.

[35] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2705–2718.

[36] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: The Partially Synchronous Version. *arXiv preprint arXiv:2209.05633* (2022).

[37] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2019. Mir-BFT: High-Throughput BFT for Blockchains. *CoRR* abs/1906.05552 (2019). arXiv:1906.05552 http://arxiv.org/abs/1906.05552

[38] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. 2022. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 17–33.

[39] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 1–17.

[40] The Algorand Team. 2019. Algorand Networks. https://www.algorand.foundation/ Accessed: April 2024.

[41] The Avalanche Team. 2020. Avalanche Networks. https://www.avax.network/ Accessed: May 2024.

[42] The Aptos Team. 2022. Aptos Networks. https://aptosfoundation.org/ Accessed: May 2024.

[43] The Diem Team. 2021. DiemBFTv4. "https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf" Accessed: May 2023.

[44] The Sui Team. 2023. Sui Networks. https://sui.io/ Accessed: May 2023.

[45] Giorgos Tsimos, Anastasios Kichidis, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2023. Hammerhead: Leader reputation for dynamic scheduling. *arXiv preprint arXiv:2309.12713* (2023).

[46] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. 2022. {DispersedLedger}:{High-Throughput} Byzantine Consensus on Variable Bandwidth Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*.

[47] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, Peter Robinson and Faith Ellen (Eds.). ACM, 347–356.