LiDO-DAG: A Framework for Verifying Safety and Liveness of DAG-Based Consensus Protocols

LONGFEI QIU, Yale University, USA
JINGQI XIAO, Yale University, USA
JI-YONG SHIN, Northeastern University, USA
ZHONG SHAO, Yale University, USA

Blockchains operating at the global scale demand high-performance byzantine fault-tolerant (BFT) consensus protocols. Most classic PBFT-like protocols suffer from an issue known as the leader bottleneck, which severely limits their throughput and resource utilization. Recently, Directed Acyclic Graph, or DAG-based protocols, have emerged as a promising approach for eliminating the leader bottleneck and achieving better performance. They attain higher throughput by separating data dissemination and block ordering. However, their safety and liveness logic is also significantly more elaborate. So far, most DAG-based protocols have only enjoyed on-paper security proofs, and it is not clear how to construct formal proofs of these protocols efficiently.

We introduce LiDO-DAG, a concurrent object model that abstracts the common logic of these protocols. LiDO-DAG is constructed by combining a DAG abstraction and LiDO, a recently proposed abstraction for leader-based consensus. To demonstrate that our framework enables rapid validation of new DAG-based protocol designs, we implemented LiDO-DAG in Coq and applied it to three recent DAG-based protocols, including Narwhal, Bullshark, and Sailfish. Our framework readily yields mechanized safety and liveness proofs for all three protocols, which are also the first mechanized liveness proofs of any DAG-based protocol. Our framework has also revealed an optimization for Sailfish that improves its worst-case latency.

CCS Concepts: • **Networks** → **Protocol testing and verification**; • **Software and its engineering** → **Formal software verification**; • **Theory of computation** → **Distributed algorithms**.

Additional Key Words and Phrases: DAG-based consensus, safety, liveness, verification, Coq proof assistant

ACM Reference Format:
Longfei Qiu, Jingqi Xiao, Ji-Yong Shin, and Zhong Shao. 2025. LiDO-DAG: A Framework for Verifying Safety and Liveness of DAG-Based Consensus Protocols. *Proc. ACM Program. Lang.* 9, PLDI, Article 203 (June 2025), 25 pages. https://doi.org/10.1145/3729306

1 Introduction

Since their inception in 2008, blockchains such as Bitcoin [Nakamoto 2008] and Ethereum [Buterin 2014] have grown into major alternative financial platforms, with billions of dollars traded on them daily [CoinGecko 2024]. However, a major factor limiting the adoption of blockchains is their low throughput. As of writing, Ethereum only supports a theoretical maximum of ~600 transactions per second [Buterin 2024], the actual value being still lower, whereas the Visa payment system processes more than 8,000 transactions per second on average [Visa Inc. 2023]. Therefore, there is significant interest in developing blockchains with higher transaction throughput.

At the core of many blockchains is a Byzantine Fault-Tolerant (BFT) consensus protocol. Blocks of transactions collected by each participating process are submitted to the consensus protocol, which

Authors' Contact Information: Longfei Qiu, Yale University, New Haven, USA, longfei.qiu@yale.edu; Jingqi Xiao, Yale University, New Haven, USA, jingqi.xiao@yale.edu; Ji-Yong Shin, Northeastern University, Boston, USA, j.shin@northeastern.edu; Zhong Shao, Yale University, New Haven, USA, zhong.shao@yale.edu.

ACM 2475-1421/2025/6-ART203
https://doi.org/10.1145/3729306

produces a single linear transaction log. Most BFT algorithms execute in *views*, and within each view a single process is elected as the *leader*. Only the leader is allowed to propose and commit blocks within each view. The other processes produce *votes* which prevent the leader from equivocating. This architecture of BFT protocols is called *leader-based* and is by far the most well-understood kind of consensus protocol. Protocol designs following this pattern include Buchman et al. [2019]; Castro [2001]; Gelashvili et al. [2022]; Lewis-Pye et al. [2024]; Naor et al. [2021]; Yin et al. [2019]. Their formal security properties have also been investigated in many works including Bravo et al. [2022]; Carr et al. [2022]; Cirisci et al. [2023]; Qiu et al. [2024b]; Rahli et al. [2018]; Vukotic et al. [2019]. However, they also exhibit several undesirable characteristics, collectively known as the *leader bottleneck* [Danezis et al. 2022; Neiheiser et al. 2021]:

- As only leaders may propose blocks, network throughput is limited by that of the leaders;
- The other processes do almost nothing, leading to significant resource under-utilization;
- Moreover, if the leader is faulty then network may not progress for long periods of time.

Recently, there emerged an alternative approach to BFT protocols, based on Directed Acyclic Graph (DAG) [Arun et al. 2024; Babel et al. 2024; Baird 2016; Danezis et al. 2022; Gągol et al. 2019; Keidar et al. 2021, 2023; Shrestha et al. 2025; Spiegelman et al. 2023, 2022a]. DAG-based protocols work around the leader bottleneck by an elegant separation between a data dissemination phase and a block ordering phase. In the *dissemination* phase, all processes collaborate to build a DAG of data blocks, representing a partial order on these blocks (Fig. 1). Data dissemination is leaderless: all processes can submit blocks to the DAG even when they are not leaders. The *ordering* phase then extends this partial order into a total order of blocks, representing the consensus log. This phase is still leader-based, and can be implemented with any leader-based BFT protocol.

Up to this point DAG-based protocols seem to be just simple compositions of DAG and traditional BFT algorithms. Indeed, some simple DAG protocols like Narwhal [Danezis et al. 2022] are structured exactly this way. The new twist in the story is that many protocols implement the ordering phase directly upon the DAG structure itself (e.g. Spiegelman et al. [2022a]). From a practical perspective, this simplifes the implementation by removing the need for a separate BFT component, and reduces commit latency. However, it comes at the cost of a significantly more complex DAG-building algorithm. Specifically, whenever each process receives a new DAG vertex, it needs to execute an *ordering algorithm* that interprets the local view of the DAG and outputs a linear log of committed blocks. The protocol must carefully arrange the DAG-building process and the ordering algorithm so that they together satisfy three correctness criteria:

- **Safety**: the consensus logs from all non-faulty processes are consistent with each other;
- **Liveness**: every block from non-faulty processes is eventually committed;
- **Fairness**: every non-faulty process can eventually submit new blocks into the DAG.

It is highly non-trivial to achieve all three goals simultaneously. As we will see in Section 2.2, the ordering algorithm can exhibit quite subtle and counterintuitive behaviors, which are necessary to ensure safety. Existing works indicate it is already challenging to verify only the safety property [Bertrand et al. 2024]. To prove liveness would require further analysis of how the ordering algorithm interacts with the DAG-building process. To our knowledge there has been no attempt to formally verify any liveness proof of DAG-based protocols, either using theorem provers or model checkers. On the other hand, new DAG-based protocols with better theoretical performance are kept being proposed in the literature, with safety and liveness proven only on paper. These proofs have grown increasingly intricate, to the point that their correctness has occasionally become controversial (see, for example, the appendix of Shrestha et al. [2025] criticizing the liveness proof of Mysticeti [Babel et al. 2024]). To resolve such disputes, we need a clean conceptual framework for understanding the behavior of these protocols and verifying their correctness proofs.

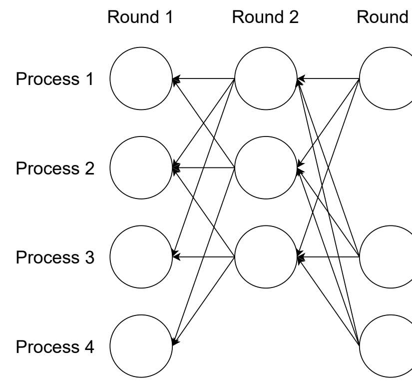

Fig. 1. DAG-based data dissemination. Each vertex in the graph represents a block of data. Vertices are stratified into rounds. Each process may only create one vertex per round. Each vertex must embed at least $2f + 1$ pointers to vertices of the immediate previous round. Each column in this figure represents one round, and each row represents vertices from a single process. Processes may skip rounds to catch up with network progress, leaving holes in the figure. The DAG graph represents only a partial order on the blocks. Additional mechanisms are required to reach a total order.

In this work, we introduce **LiDO-DAG**, a concurrent object model for DAG-based consensus that enables refinement-based safety and liveness proofs for these protocols. Our starting point is the key observation that most partially synchronous DAG-based protocols, though seemingly complex, can still be logically interpreted as a composition of DAG and a leader-based BFT protocol, a point we will explain in Section 2.3. This suggests that approaches for verifying leader-based BFT can also be applied to DAG-based BFT. In particular, Qiu et al. [2024b] described a theory called **LiDO** for verifying leader-based consensus and applied it to several BFT protocols. However, as we will see in Section 2.4, liveness of DAG-based consensus involves a number of subtle issues. In particular, it is possible that a system satisfying liveness of both leader-based consensus and DAG-based block dissemination still does not satisfy liveness as defined above. To patch this gap it is necessary to make LiDO aware of the DAG layer, and introduce new safety requirements on how consensus interacts with DAG, resulting in the LiDO-DAG model.

Our model enables efficient validation of new DAG-based protocol designs. To demonstrate this, we implemented in Coq [The Coq Development Team 2024] three state-of-the-art DAG-based protocols, namely Narwhal [Danezis et al. 2022], Bullshark [Spiegelman et al. 2022b], and Sailfish [Shrestha et al. 2025], and constructed mechanized correctness proofs for all three protocols by refinement to LiDO-DAG. Surprisingly, our model has also helped us discover an optimization for Sailfish that improves its worst-case latency, also formally verified.

To summarize, our contributions are:

- **LiDO-DAG**, a concurrent object abstraction for partially synchronous DAG-based protocols that enables refinement-based safety, liveness, and fairness proofs for these protocols;
- **Coq implementations** of three state-of-the-art DAG-based protocols, namely Narwhal [Danezis et al. 2022], Bullshark [Spiegelman et al. 2022b], and Sailfish [Shrestha et al. 2025], including mechanized safety and liveness proofs.
- **An optimization for Sailfish** with lower worst-case latency, formalized under LiDO-DAG.

All results claimed in this paper have been mechanized in Coq and available as an artifact [Qiu et al. 2025a]. More details about our work can be found in the technical report [Qiu et al. 2025b].

2 Overview

2.1 The Landscape of DAG-Based Protocols

To set the stage, we begin with a survey of the current ideas on DAG-based protocol design.

In leader-based consensus, when a non-leader process receives a client request, it either delays processing the request, or forwards it to the current leader. In contrast, the common theme of all DAG-based protocols is that each process immediately packages requests it has received into data blocks that are then disseminated within the network. Clearly if block creation is unconstrained then the network could be easily flooded by byzantine processes. To prevent such attacks, the blocks are stratified into *rounds*. Each process may only create one block per round, and blocks within each round (except the first round) must contain pointers to at least $n - f$ vertices in the previous round, where n is the total number of processes and f is the fault-tolerance threshold. In the standard setting $n = 3f + 1$, this ensures at least half of the blocks in the network are from non-faulty processes, and prevents flooding attacks.

If we interpret data blocks as graph vertices, and pointers as directed edges, then the valid blocks always form a directed acyclic graph (Fig. 1), which is why these protocols are called DAG-based.

Authenticated vs. Unauthenticated Protocols. The above description immediately raises the question of how to prevent byzantine processes from creating multiple blocks within a single round. The easy way is to use a *reliable broadcast* (RBC) protocol to deliver each vertex. Protocols based on RBC are called **authenticated** and include DAG-Rider [Keidar et al. 2021], Narwhal [Danezis et al. 2022], etc. Informally, RBC provides each party with an infinite sequence of *message slots*. Each party p_k may call $r\_bcast_k(m, r)$ to fill its slot r with message m. Each slot may only be filled once, even if p_k is byzantine. Each party may receive signals $r\_deliver(m, r, p_k)$ which state that p_k has filled its slot r with message m. RBC additionally requires that, after one non-faulty party receives a message, all other non-faulty parties will receive the same message within bounded time, so that participants do not need to worry about delivery omissions.

In Alg. 1 we provide an formal *ideal* model of RBC. The finite map rb_map keeps track of the messages filled in each slot. Initially, all slots are empty (line 4). Each party p_k may call $r\_bcast_k(m, r)$ to fill one of its slots, unless the slot is already filled (line 6). A virtual agent known as the adversary $\mathcal{A}$ controls message delivery (line 8-10), subject to liveness constraints (line 12 and 13).

Algorithm 1 Ideal Model of Reliable Broadcast

1: Agents: parties $p_1, \cdots, p_n$, adversary $\mathcal{A}$.
2: State variable: finite map $rb\_map : \{1, \cdots, n\} \times \mathbb{N} \mapsto$ `option val`.
3: **initialize**:
4: Assume $\forall k, \forall r, rb\_map(k, r) =$ None.
5: **upon** $r\_bcast_k(m, r)$ from party p_k:
6: **if** $rb\_map(k, r) =$ None **then**
7: $rb\_map(k, r) \leftarrow$ Some m
8: **upon** $deliver(k, r, i)$ from $\mathcal{A}$:
9: **if** $rb\_map(k, r) =$ Some m **then**
10: Send signal $r\_deliver(m, r, p_k)$ to party p_i
11: Liveness requirements after GST:
12: If p_k is non-faulty, after p_k calls $r\_bcast_k(m, r)$, $\mathcal{A}$ calls $deliver(k, r, i)$ for each non-faulty p_i within time T_{RBC}.
13: Regardless of whether p_k is non-faulty, after $rb\_map(k, r) \neq$ None and $\mathcal{A}$ has called $deliver(k, r, i)$ for any non-faulty p_i, it calls $deliver(k, r, i')$ for every non-faulty $p_{i'}$ within Δ.

Another line of work attempts to reduce block creation latency by avoiding RBC. These protocols are known as **unauthenticated** and include Cordial Miners [Keidar et al. 2023] and Mysticeti [Babel et al. 2024]. They rely instead on *failure detection*. If a non-faulty process observes equivocating blocks from the same process within a single round, it forwards the evidence to other processes and they expel the faulty process from the system.

Establishing Total Order over the Blocks. The DAG graph only provides a partial order over the blocks. To extend this partial order into a linear order, the easiest way is to use an **external BFT** algorithm to order the blocks. A naive implementation would commit the hash of every single block into the BFT log. A better way is to make each entry in the log implicitly include its entire closure, the set of all (indirectly) reachable vertices in the DAG. Thus each BFT leader proposes only the latest blocks it has received, and they will transitively include all previous blocks.

More recently, people have realized that it is in fact possible to implement BFT directly upon the DAG structure. Two approaches known as **physical DAG** and **logical DAG** have been proposed. In logical DAG, consensus information such as votes and timeouts are packaged into the data blocks. In physical DAG, they are encoded onto the topological structure of the DAG.

Table 1. Classification of recent DAG-based protocols. Some protocols such as Bullshark and Cordial Miners have multiple versions for different settings. A indicates authenticated, U indicates unauthenticated protocol. Bold text indicates protocols we have formally verified in this work.

	Asynchronous	Partially Synchronous
External BFT		**Narwhal**^{A} [Danezis et al. 2022]
Physical DAG	DAG-Rider^A [Keidar et al. 2021] Bullshark^A [Spiegelman et al. 2022a] Cordial Miners^U [Keidar et al. 2023] Tusk^A [Danezis et al. 2022]	**Bullshark**^{A} [Spiegelman et al. 2022b] Mysticeti^U [Babel et al. 2024] **Sailfish**^{A} [Shrestha et al. 2025] Cordial Miners^U [Keidar et al. 2023] Shoal^A [Spiegelman et al. 2023] Shoal++^{A} [Arun et al. 2024]
Logical DAG		Fino^A [Malkhi and Szalachowski 2023]

As we will see in Section 2.3, physical DAGs like Bullshark must carefully control the timing of DAG vertex creation to ensure liveness. This makes them more difficult to implement than logical DAGs, where vertex creation and consensus progress independently. However, experiments suggest that logical DAGs do not perform as well as physical DAGs [Spiegelman et al. 2022a]. Therefore, the majority of DAG-based protocols in the literature are physical DAGs rather than logical DAGs.

Communication Models. Like other consensus protocols, liveness of DAG-based protocols depend on the communication model. The distributed system literature distinguishes between **asynchronous**, **synchronous**, and **partially synchronous** protocols [Dwork et al. 1988]. In asynchronous protocols, messages may arrive arbitrarily late. In synchronous protocols, they must arrive within a known bound Δ. Partial synchrony represents a middleground: messages must arrive within Δ, but only so after an unknown timepoint known as the Global Synchronization Time (GST).

Most works on DAG-based protocols use either the asynchronous or the partially synchronous model. It is known that consensus under asynchrony cannot be deterministically solved [Fischer et al. 1985]. Hence all asynchronous protocols rely on probabilistic primitives such as public coins which are difficult to model. Furthermore, asynchronous protocols suffer from a dilemma between fairness and garbage collection [Spiegelman et al. 2022a]. Therefore most practical deployments of DAG-based protocols use partially synchronous versions [Arun et al. 2024; Babel et al. 2024]. We thus exclusively focus on partially synchronous DAG-based protocols in this work.

Summary. In Table 1 we present a classification of recently-proposed DAG-based protocols along the three aspects we discussed above. The figure clearly shows that partially synchronous protocols, especially physical DAGs, have attracted the most attention in current literature.

2.2 The Subtleties of DAG-Based Protocols

In terms of performance, currently the best DAG-based protocols are physical DAGs. We now analyze what makes this class of protocols difficult to understand.

The general operation of physical DAG is as follows. First, within the DAG graph a subset of vertices are designated as the *anchors*, also known as *leader vertices*. Second, a *commit rule* is defined for each anchor. If a process observes that the commit rule is satisfied, it considers the corresponding anchor committed. Finally, an *ordering algorithm* is defined upon the local state of each process. The ordering algorithm inspects the set of received vertices and returns an ordered list of anchors. This list is guaranteed to contain all committed anchors, and may contain additional anchors. Each entry in the list is then expanded into its closure. After deduplication, the result is the currently observed consensus log.

As a concrete example, we consider the Bullshark [Spiegelman et al. 2022b] protocol. Fig. 2 shows a possible DAG under Bullshark, running with $n = 3f+1 = 4$ processes. In Bullshark, the DAG is divided into *waves*, each wave consisting of two rounds. Within each wave, one of the processes is designated as the *leader*. The anchor of each wave is the vertex created by the leader in the first round of that wave. Thus each wave has at most one anchor. For each anchor, the commit rule is at least $f + 1$ blocks in the second round of the same wave embed a pointer to that anchor. The ordering algorithm is specified in Alg. 8.

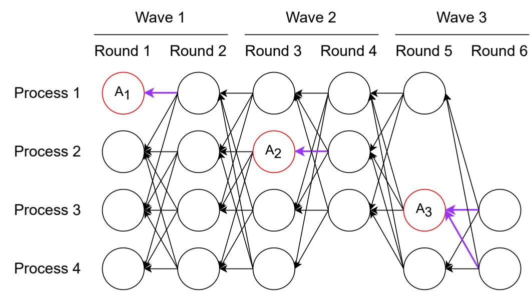

Fig. 2. An example DAG graph under Bullshark. Rounds are divided into groups of two, called waves. Anchors are colored red and labeled as A_k. Thick purple arrows indicate edges to anchors. If there exists $f + 1$ purple edges within a single wave, the anchor of that wave is considered committed. Hence A_3 is committed but A_1, A_2 are not.

Within this section, let us use A_k to denote the anchor of wave k. In Fig. 2, we observe the commit rule is satisfied for A_3, so the ordering algorithm is guaranteed to return A_3. Although the commit rule is not satisfied for A_1 and A_2, they are both reachable from A_3. Unless the reader is already familiar with Bullshark, it seems natural to conjecture the ordering algorithm should return these two anchors as well. The reader would then be very surprised to learn the actual list returned is $[A_2, A_3]$, which includes A_2 but omits A_1.

Before discussing why this is the case, we point out this paradoxical behavior shows that in physical DAG protocols, merely creating an anchor and disseminating it does not guarantee it will be committed. Thus the protocol must include additional rules that constrain DAG vertex creation, in order to achieve liveness. In Bullshark, for example, each process is equipped with a local timer that controls the timing of vertex creation (see line 24 of Alg. 8). The interaction between timers and DAG-building is a major factor that complicates liveness proofs of physical DAG protocols.

2.3 Understanding DAG-Based Consensus with LiDO

Why is it that in Fig. 2, the ordering logic omits A_1, despite it being reachable from A_3? One way to understand the problem is to follow Alg. 8 and the proofs in Spiegelman et al. [2022a,b] line-by-line. However they are long and difficult to read. Our key insight is that things can be understood much more easily, by realizing Bullshark is in fact simulating a leader-based BFT protocol.

Verifying Protocols via Abstract Model Refinement. From the informal description of Bullshark one can already see many conceptual analogies between Bullshark and leader-based BFTs like HotStuff [Yin et al. 2019] and Jolteon [Gelashvili et al. 2022]: 1) all these protocols execute in a succession of *views* (waves in Bullshark); 2) there is a single predetermined leader in each view, whose goal is to commit new data blocks; 3) all these protocols use timers to ensure liveness for non-faulty leaders.

It is tempting to ask how this similarity can be exploited to simplify verification of Bullshark. Although Bullshark and HotStuff share some similarity, it is not possible to prove that one is simulating the other, as their network-level details are very different. Recently, Qiu et al. [2024b] advanced the idea that we can prove all these protocols are simulating an abstract model of consensus. They introduced a formal model called **LiDO** for verifying leader-based consensus.

There are several advantages in introducing abstract models for verifying complex protocols like Bullshark. First, abstract models supply convenient guides in formulating network-level invariants. The LiDO model postulates three linearization points [Herlihy and Wing 1990] within each view of consensus, called *pull*, *invoke*, and *push* (explained on the next page). To verify a protocol one

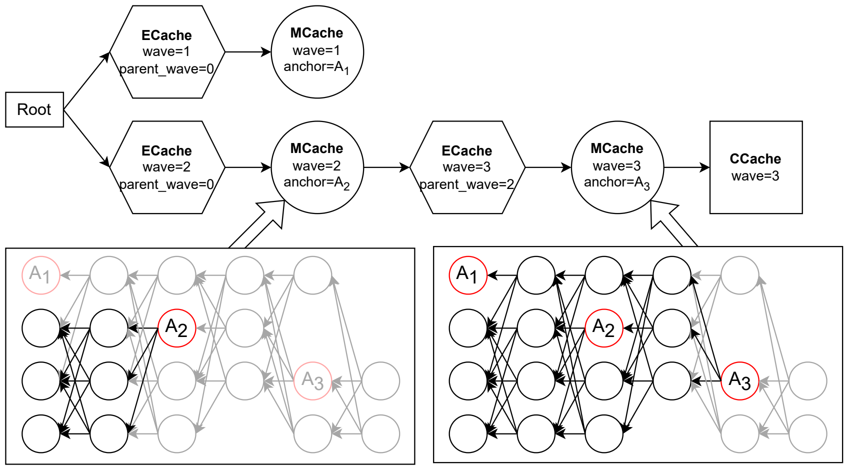

Fig. 3. The state of consensus in Fig. 2, represented under the LiDO-DAG framework. The closure of anchors A_2, A_3 are shown below. The closure of anchor A_1 contains only A_1 itself. When an anchor is committed, the entire closure is implicitly committed. See Section 2.3 for explanation.

first defines what these linearization points correspond to in the network model. The abstract model defines a number of invariants necessary to prove safety of consensus. After defining the refinement relation for the linearization points, these invariants can be translated to properties of the network model, simplifying the error-prone task of formulating safety invariants.

Second, abstract models provide reusable decomposition of liveness properties into safety properties. Most consensus algorithm designs come with liveness claims about the commit latency of the protocol. Actually formalizing this claim turns out to be tricky. For example, if the commit latency is 8Δ, it does not mean after GST a new log entry will be committed every 8Δ: it cannot hold if the current leader of the system is faulty. Thus liveness of the protocol must be stated relative to the "current leader," which leads to the question of who the "current leader" is. LiDO resolves the issue by providing a pacemaker abstraction, which contains two variables *current view* and *remaining time*. Hence the commit latency can be stated as a safety property: if leader of current view is non-faulty, and remaining time is at least 8Δ, then a new log entry will be committed within 8Δ. We can then prove on the abstract model that new entries will be committed infinitely often.

How Bullshark Refines LiDO. Under the LiDO framework, the actions of each leader within a wave can be summarized as three steps:

(1) **Pull**: the leader updates its local consensus log;
(2) **Invoke**: the leader proposes new entries to be appended to the log;
(3) **Push**: the leader commits the proposed entries.

We now look at what these actions correspond to in Bullshark. For the moment, we ignore the implicitly included closure, and focus only on the anchors. Thus the consensus log corresponds to the list returned by the ordering algorithm, and in each wave the leader proposes a single anchor.

Fig. 3 shows how we interpret the state of consensus in Fig. 2 under the LiDO framework. The LiDO model uses a tree of *cache nodes* to represent the result of each action by the leader. The Root node represents the empty consensus log at the beginning of execution. The results of each pull, invoke, and push action are represented by **ECache**, **MCache**, and **CCache** respectively. They stand for leader Election, Method proposal, and Commit.

When execution begins, process 1 performs pull. Nothing has been proposed yet, so process 1 obtains the empty consensus log. This is represented by an ECache in wave 1 that is attached to

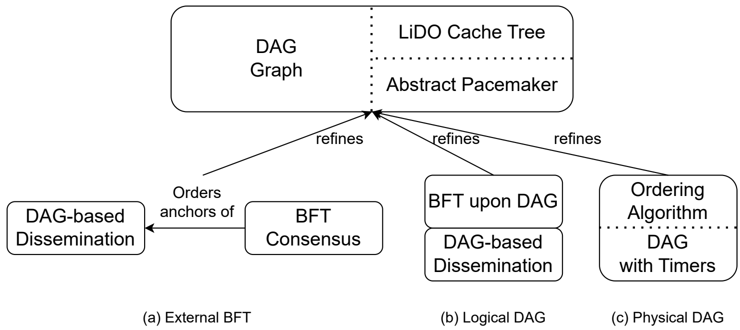

Fig. 4. The LiDO-DAG model, with three implementation strategies. (a) In external BFT implementations, DAG is solely used to disseminate data blocks, and an external BFT algorithm orders the blocks. (b) In logical DAG implementations, BFT is implemented by packaging votes and timeouts into data blocks that are disseminated by DAG. (c) In physical DAG implementations, DAG is modified to cooperate with timers so that votes and timeouts are encoded into its topological structure.

the Root node. Process 1 then proposes the anchor A_1 but fails to commit it (the commit rule is not satisfied), so there is an MCache but not CCache in wave 1.

Now wave 2 begins and process 2 performs pull. The result of this operation can be learned from the pointers embedded in the anchor A_2. Here we notice that in Fig. 2, A_1 *is not reachable from* A_2. We thus infer that when process 2 proposed its anchor A_2, it was not aware of the anchor A_1, and its local consensus log was still the empty log. We represent this by having the ECache of wave 2 also attached to Root. It then goes on to propose the anchor A_2 but again fails to commit.

When process 3 performs pull, it observes both anchors A_1 and A_2, but they represent conflicting consensus logs, represented as a fork in Fig. 3. It can infer that A_1 is not committed by any process, because otherwise process 2 must have observed A_1. However, it cannot be certain whether any process has committed A_2 or not. Thus it is forced to use the consensus log represented by A_2. This follows from the general pattern that leaders must use the latest consensus log they see, and ignore all other conflicting logs. Process 3 then proposes the anchor A_3 and successfully commits it, represented by the CCache of wave 3. The final consensus log is thus $[A_2, A_3]$ which omits A_1.

In summary, there is a direct correspondence between the behavior of Bullshark and the LiDO model of leader-based consensus: each anchor corresponds to an MCache; the ECache of each wave is determined by the latest previous anchor reachable from an anchor; and each anchor that is observed to be committed corresponds to a CCache.

The key safety invariant of LiDO is that whenever a CCache exists in wave w, and an ECache exists in wave $w' > w$, the ECache must be attached to an MCache whose wave number is at least w. By the above correspondence, this means whenever the anchor A_w is committed, and an anchor $A_{w'}$ ($w' > w$) exists, then A_w is reachable from $A_{w'}$. This rule was previously described in Shrestha et al. [2025]; we now see it simply follows from the general safety rule for leader-based BFT.

2.4 Challenges of Adapting LiDO to DAG-Based Protocols

Although network-level descriptions of DAG-based protocols seem complicated, the above analysis shows there is strong resemblance between the behavior of these protocols and classic leader-based BFT. Thus it is natural to think that safety and liveness of DAG-based consensus can be verified by proving a refinement relation to the LiDO model.

The notion that DAG-based protocols are simulating leader-based consensus is informally discussed in several papers including Arun et al. [2024] and Shrestha et al. [2025]. In these works, the simulation relation is used to provide intuition that informs the design of new protocols. In this work we aim to show this relation can be made formally precise, and interpreting these protocols this way substantially simplifies the correctness verification of these protocols.

However, when we actually attempted to prove refinement between these protocols and LiDO, we immediately noticed several gaps between the semantics of LiDO and the correctness criteria of DAG-based protocols. We show two examples here:

External Validity. All BFT protocols are required to satisfy *external validity*, meaning every committed entry in the consensus log must have been submitted by an external client. In most cases external validity is enforced by checking the signature embedded in the block, which is a local action. This is also the assumption made in the Jolteon implementation provided by Qiu et al. [2024b]. However, when a BFT protocol is composed with a data dissemination protocol, as in Narwhal, then external validity takes on a new definition: each BFT log entry must reference an existing vertex in the DAG. This introduces unexpected complications. For example, if by the time the leader's BFT request message arrives, the voter has not yet received the proposed block from RBC, then it will reject the proposal, which will break liveness. In Section 4.2, we explain how we worked around this issue by modifying the network model of Jolteon, but it shows the trickiness in adapting existing verified BFT protocols to a DAG-based setting.

Stagnant Anchors. Liveness of DAG-based consensus requires that all blocks are eventually committed. This seems easy to prove if we assume that 1) each block is eventually included in the closure of some anchor; and 2) each leader will always eventually commit new anchors in the BFT log. However, there is a subtle gap: when a leader proposes a new anchor, it might not be the latest block created by this leader. If a leader keeps proposing the same anchor in the BFT protocol, then the newer blocks would not be committed, even though liveness of neither block dissemination nor consensus is violated. This does not occur in Bullshark, but it is another formal factor to consider in protocols like Narwhal where leaders can choose which anchor to propose.

Thus merely proving refinement to LiDO does not yield the expected liveness properties of DAG-based protocols. To bridge these gaps it is necessary to extend the LiDO theory with abstractions for DAG, and formalize new requirements necessary for liveness on the extended model. This leads to the LiDO-DAG model, which we formally define in the next section. As shown in Fig. 4, the LiDO part of LiDO-DAG is kept largely similar to Qiu et al. [2024b], which is intentional to allow reusing their proofs, while we added abstractions for DAG-based block dissemination. All three styles of DAG-based consensus can be refined to our model.

3 The LiDO-DAG Model

In this section we formally define the LiDO-DAG model. We first define a concurrent object that represents the DAG of data blocks. Then we combine the DAG object with the LiDO model [Qiu et al. 2024b] which represents the consensus log of DAG anchors. Finally we explain how to reduce liveness and fairness of DAG-based protocols into safety properties with our model. Here we describe LiDO-DAG mostly in pseudocode. For details of Coq formalization see Appendix D.

3.1 The DAG Object

The internal state of a DAG is a set of *vertices*. The vertex record is defined in Fig. 5. Each vertex contains a piece of client-submitted data, a list of pointers to other vertices, and some other metadata. We also give each vertex a unique ID. Each pointer is represented by the ID of the target.

The data embedded within each vertex is required to satisfy *external validity*: the external client should have ultimate control over what gets recorded in the DAG. To enforce this we follow Qiu et al. [2024b] and assume the existence of a *data pool* object (Alg. 3). The data pool interacts with

Algorithm 2 The DAG Object

1: Agents: DAG-builder threads $p_1, p_2, \cdots, p_{3f+1}$.
2: State variable: $verts$: `list Vertex`.
3: Notation: $verts[id] = v$ means $v \in verts$ and $v.id = id$; $verts[id] = \bot$ if no vertex with $v.id = id$ exists.
4: Notation: $isQuorum(P) = true$ if P is a set of at least $2f + 1$ threads.
5: Notation: $\mathrm{cl}(v)$ is the closure of vertex v, defined in Section 3.1.
6: **initialize**: Assume $verts = \{\}$.
7: **upon** $DAG\text{-}Pull(r)$ with $r \geq 1$ from p_i:
8:     **if** $r = 1$ **then**
9:         **return** $\{\}$
10:     **if** p_i is honest but has not called $DAG\text{-}Invoke(r-1, \_, \_)$ before **then**
11:         **return** $InvalidCall$
12: **upon** $DAG\text{-}Invoke(r, val, preds)$ with $r \geq 1$ from p_i:
13:     **if** $CheckRegister(val) = false$, or $verts[id] = \bot$ for some $id \in preds$ **then**
14:         **return** $InvalidCall$
15:     **if** $verts[id].round \geq r$ for some $id \in preds$ **then**
16:         **return** $InvalidCall$
17:     $strongEdges \leftarrow$ `filter` $(id \mapsto verts[id].round = r - 1)$ $preds$
18:     $strongEdgeBuilders \leftarrow$ `map` $(id \mapsto verts[id].builder)$ $strongEdges$
19:     **if** $r > 1 \land isQuorum(strongEdgeBuilders) = false$ **then**
20:         **return** $InvalidCall$
21:     **if** p_i is honest and there exists vertex $v' \in verts$ with $v'.builder = p_i$, $v'.round \geq r$ **then**
22:         **return** $InvalidCall$
23:     **if** p_i is honest, there exists vertex $v' \in verts$ with $v'.builder = p_i$ but $\forall id \in preds, v' \notin \mathrm{cl}(verts[id])$ **then**
24:         **return** $InvalidCall$
25: **upon** Respond to call $DAG\text{-}Pull(r)$:
26:     Nondeterministically choose set $S \subseteq verts$
27:     $strongEdges \leftarrow$ `filter` $(id \mapsto verts[id].round = r - 1)$ $preds$
28:     $strongEdgeBuilders \leftarrow$ `map` $(id \mapsto verts[id].builder)$ $strongEdges$
29:     Check $isQuorum(strongEdgeBuilders) = true$, otherwise go back to line 26 and choose another set S.
30:     If no such set S exists, respond to the call later.
31:     **return** S to p_i
32: **upon** Respond to call $DAG\text{-}Invoke(r, val, preds)$ with $Success$:
33:     Nondeterministically choose id such that $verts[id] = \bot$.
34:     $v \leftarrow \{id := id; round := r; builder := p_i; data := val; preds := preds\}$
35:     $verts \leftarrow verts \cup \{v\}$
36:     **return** v to p_i
37: **upon** Respond to call $DAG\text{-}Invoke(r, val, preds)$ with $Timeout$:
38:     Precondition: exists set P with $isQuorum(P) = true$ and $\forall p_i \in P, \exists v \in verts, v.builder = p_i \land v.round \geq r$
39:     **return** $Timeout$ to p_i

both external clients and the DAG object. An external client may call $Register(v)$ to add some value v to the pool. The DAG object may call $CheckRegister(v)$ to check if some value has been registered or not. Both operations return atomically.

The agents interacting with the DAG object are a fixed set of DAG-builder threads, one from each participating process. Each process is classified as *synchronous*, *asynchronous*, or *byzantine*. Both synchronous and asynchronous processes are *honest*. Their difference is that asynchronous processes may undergo omission failure even after GST. To simplify presentation, we assume the standard setting with

```
Record Vertex := {
  (* Unique ID for each vertex *)
    vertex_id : nat;
  (* Round number of vertex *)
    vertex_round : nat;
  (* Builder thread ID *)
    vertex_builder : nat;
  (* Client data *)
    vertex_data : val;
  (* Target ID of embedded ptrs *)
    vertex_preds : list nat;
}.
```

Fig. 5. The vertex record.

$2f + 1$ synchronous processes, f byzantine processes, and no asynchronous processes. A *quorum* is any set of $2f + 1$ processes, and a *weak quorum* is any set of $f + 1$ processes. See Appendix A for how to tolerate omission faults in addition to byzantine faults.

The object exposes two operations $\textit{DAG-Pull}(r)$ and $\textit{DAG-Invoke}(r, val, preds)$. When an agent invokes one of these operations, the object first makes a number of validity checks. If one of these checks fails, the object returns $InvalidCall$ atomically. Otherwise, the operation is valid, and the object may respond to the call at any later timepoint, or not respond at all (to model network failure). No change to the object state occurs until the object responds to the call. After an honest thread invokes a valid operation, it waits until the object responds to the call. If the thread is byzantine, it may voluntarily withdraw the operation, with no change in object state. We represent this with the response $Withdrawn$. The validity conditions and the possible responses to each operation are specified in Alg. 2.

Algorithm 3 Data Pool

```
State variable: finite set S of data values.
initialize:
    Assume S = {}.
upon Register(v):
    S ← S ∪ {v}
upon CheckRegister(v):
    return v ∈ S
```

The purpose of $\textit{DAG-Invoke}(r, val, preds)$ is to add a vertex of round r to the DAG, with val as its contained data, and $preds$ as its embedded pointers. Round numbers start from 1. As discussed in Section 2, to prevent byzantine flooding, $preds$ must contain at least $2f + 1$ vertices of round $r - 1$ created by different threads. This in turn implies the thread must know the existence of these vertices. The process of learning vertices in round $r - 1$ is abstracted by the operation $\textit{DAG-Pull}(r)$, which each honest thread should perform before invoking $\textit{DAG-Invoke}(r, \_, \_)$.

The $\textit{DAG-Invoke}(r, \_, \_)$ operation may either return $Success$ or $Timeout$. When it returns $Success$, a new vertex is added and returned to the caller. Upon $Timeout$ no change to the DAG occurs. $Timeout$ may occur only when there are already $2f + 1$ vertices in some round $r' \geq r$. The $Timeout$ outcome is used to model the round-skipping behavior in some protocols, where honest processes catch up network progress by abandoning proposing vertices in some rounds.

When $\textit{DAG-Pull}(r)$ with $r > 1$ returns, it always returns a set of vertices containing at least $2f + 1$ vertices in round $r - 1$. To prevent the system from getting stuck, when an honest thread calls $\textit{DAG-Pull}(r)$, we must ensure either there already exists $2f + 1$ vertices in round $r - 1$, or these vertices will eventually be created. We enforce this by requiring that before any honest thread calls $\textit{DAG-Pull}(r)$ with $r > 1$, it must have already performed $\textit{DAG-Invoke}(r - 1, \_, \_)$ (line 10 of Alg. 2). Thus each honest thread is forced to follow the interaction pattern below:

$$\textit{DAG-Pull}(1) \cdot \textit{DAG-Invoke}(1, \_, \_) \cdot \textit{DAG-Pull}(2) \cdot \textit{DAG-Invoke}(2, \_, \_) \cdot \textit{DAG-Pull}(3) \cdots$$

It can be shown that if all honest threads follow this pattern, then it is not possible for *DAG-Pull* to get stuck. We will return to this point in Section 3.4, where we discuss liveness of LiDO-DAG.

The Closure of a Vertex. For each vertex v in the DAG object, we define its *closure* $\mathrm{cl}(v)$ by induction on $v.round$. If $v.round = 1$ then $\mathrm{cl}(v) = \{v\}$. If $v.round > 1$, then $\mathrm{cl}(v')$ is already defined for every $v' \in v.preds$. If $v.preds = \{id_1, id_2, \cdots, id_n\}$ then let $u_i = verts[id_i]$ and we define

$$\mathrm{cl}(v) = \mathrm{cl}(u_1) \cup \mathrm{cl}(u_2) \cup \cdots \cup \mathrm{cl}(u_n) \cup \{v\}.$$

3.2 Combining DAG and LiDO

The LiDO-DAG object augments the DAG object with two additional concurrent components: a cache tree and an abstract pacemaker. The agents interacting with these components are $3f + 1$ LiDO-proposer threads (one from each participating process), and an adversary $\mathcal{A}$.

The Cache Tree. The internal state of the LiDO cache tree is a set Σ of *cache nodes*, defined in Fig. 6. Five kinds of cache nodes exist: Root, ECache, MCache, CCache, and TCache. Each cache

$$
\begin{aligned}
CacheNode \triangleq\, & Root \\
& \mid ECache(\mathbb{N}_{wave} * \mathbb{N}_{parent\_wave}) \\
& \mid MCache(\mathbb{N}_{wave} * \mathbb{N}_{anchor\_ID}) \\
& \mid CCache(\mathbb{N}_{wave}) \\
& \mid TCache(\mathbb{N}_{wave})
\end{aligned}
$$

(a) Cache Nodes

$$
\begin{aligned}
parent(ECache(w, p)) &\equiv \begin{cases} Root & (p = 0) \\ \Sigma[p].mcache & (p > 0) \end{cases} \\
parent(MCache(w, m)) &\equiv \Sigma[w].ecache \\
parent(CCache(w)) &\equiv \Sigma[w].mcache
\end{aligned}
$$

(b) Cache Node Parent Relation

Fig. 6. Definition of LiDO cache nodes and node parents [Qiu et al. 2024b].

Algorithm 4 The LiDO-DAG Cache Tree

Agents: LiDO-proposer threads $p_1, p_2, \cdots, p_{3f+1}$.
State variable: Σ : `list CacheNode`.
initialize: Assume $\Sigma = \{Root\}$.
upon $Pull(w)$ from p_i:
  if $p_i \neq leader\_of(w)$, or $\Sigma[w].ecache \neq \bot$ **then**
    return $InvalidCall$
upon $Invoke(w, id)$ from p_i:
  if $p_i \neq leader\_of(w)$, or $verts[id] = \bot$, or $\Sigma[w].ecache = \bot$, or $\Sigma[w].mcache \neq \bot$ **then**
    return $InvalidCall$
  if p_i is honest, and $verts[id]$ is not the latest vertex built by p_i when $Pull(w)$ was called (or a later vertex) **then**
    return $InvalidCall$
upon $Push(w)$ from p_i:
  if $p_i \neq leader\_of(w)$ or $\Sigma[w].mcache = \bot$ **then**
    return $InvalidCall$
upon Respond $Timeout$ to $Pull(w)$, $Invoke(w, \_)$, or $Push(w)$:
  $\Sigma \leftarrow \Sigma \cup \{TCache(w)\}$
  return $TCache(w)$ to p_i
upon Respond $Success$ to $Pull(w)$:
  Choose p s.t. $p < w \land (p = 0 \lor \Sigma[p].mcache \neq \bot) \land \forall w', w' < w \Rightarrow \Sigma[w'].ccache \neq \bot \Rightarrow p \geq w'$.
  Such p always exists.
  $c \leftarrow \{ECache(w, p)\};\ \Sigma \leftarrow \Sigma \cup \{c\}$
  return c to p_i
upon Respond $Success$ to $Invoke(w, id)$:
  $c \leftarrow \{MCache(w, id)\};\ \Sigma \leftarrow \Sigma \cup \{c\}$
  return c to p_i
upon Respond $Success$ to $Push(w, id)$:
  Precondition: $\forall w', w' > w \Rightarrow \Sigma[w'].ecache = \bot \lor \Sigma[w'].ecache.parent\_wave \geq w$.
  If precondition is not satisfied, respond $Timeout$ instead.
  $c \leftarrow \{CCache(w)\};\ \Sigma \leftarrow \Sigma \cup \{c\}$
  return c to p_i

node except Root has a *wave* argument indicating the wave it belongs to. Wave numbers begin from 1. The intended meaning of these cache nodes is that: each ECache represents the log that the leader of wave w received when it entered wave w; each MCache represents one new anchor entry appended to the log; each CCache is a mark that a particular branch of the log has been committed; each TCache is a record that some network failure has occured in wave w. In the following paragraphs these meanings will be made more precise.

The cache tree maintains that within each wave, there is at most one ECache, one MCache, and one CCache. We thus use $\Sigma[w].ecache$ to represent the ECache of wave w if it exists. If it does not exist, we write $\Sigma[w].ecache = \bot$. The notations $\Sigma[w].mcache$, $\Sigma[w].ccache$ are defined similarly.

Algorithm 5 The Abstract Pacemaker

```
Agents:
    LiDO-proposer threads p_1, p_2, ··· p_{3f+1};
    Adversary 𝒜.
Constant: timer_reset_val : nat.
State variables:
    current_wave : nat
    remaining_time : nat
    can_start_next : bool
initialize:
    Assume current_wave = 1
    Assume remaining_time = timer_reset_val
    Assume can_start_next = false
upon StartNext(w) from p_i:
    if p_i ≠ leader_of(w) then
        return InvalidCall
    if current_wave = w then
        can_start_next ← true
    return Success
upon Elapse() from 𝒜:
    if remaining_time > 0 then
        remaining_time ← remaining_time − 1
    else
        can_start_next ← true
    return Success
upon TimeoutStartNext() from 𝒜:
    if can_start_next = true then
        current_wave ← current_wave + 1
        remaining_time ← timer_reset_val
        can_start_next ← false
        return Success
    else
        return InvalidCall
```

Each cache node in Σ except Root and TCache has a parent cache node, defined in Fig. 6. The cache nodes are chained by this parent relation into a tree (Fig. 3). Each cache node c except TCache is also associated with a *consensus log*, denoted by $\log(c)$. It is defined by induction on the tree structure as follows: $\log(Root) = []$; if c is an ECache or CCache, then $\log(c) = \log(parent(c))$; and if c is an MCache, then $\log(c) = \log(parent(c)) \mathbin{++} [c.anchor\_id]$.

Three operations are exposed for manipulating the cache tree, which are $Pull(w)$, $Invoke(w, id)$, and $Push(w)$. The cache tree responds to each operation with either $Success$ or $Timeout$, adding one new cache node to Σ. Each $Pull(w)$, $Invoke(w, id)$, $Push(w)$ operation only creates cache nodes in wave w. Each wave has a unique, predetermined *leader*, denoted by $leader\_of(w)$ and known to all participants. Only the $leader\_of(w)$ may call $Pull(w)$, $Invoke(w, id)$, or $Push(w)$.

The semantics of these operations are defined in Alg. 4. When $Pull(w)$ succeeds, the object creates an ECache c, with $\log(c)$ representing the consensus log that $leader\_of(w)$ receives. The wave number of the last entry in the log is recorded in $c.parent\_wave$, and the previous entries are defined via the parent relation. When $Invoke(w, id)$ succeeds, a new anchor entry (represented by an MCache) is created but not yet committed. It gets committed when $Push(w)$ succeeds.

The LiDO cache tree maintains a key invariant (line 19 and 27 of Alg. 4): *if* $\Sigma[w].ccache \neq \bot$, $w' > w$, *and* $\Sigma[w'].ecache \neq \bot$, *then* $\Sigma[w'].parent\_wave \geq w$. Then it is easy to show that if $\Sigma[w].ccache \neq \bot$, then the log of every cache node c in wave $w' > w$ extends from $\log(\Sigma[w].ccache)$. In this sense we say, each CCache marks a committed branch of consensus log.

The Abstract Pacemaker. The LiDO cache tree presents an elegant abstraction for the consensus log, but is in itself insufficient to reduce liveness of consensus to safety properties. Qiu et al. [2024b] points out that this is because we lack information about the local timers running at each process, without which we cannot infer whether leader actions will succeed before timer expiration. They introduced an abstract pacemaker that can be simulated by a group of local timers.

The pacemaker is defined in Alg. 5. It has two variables $current\_wave$ and $remaining\_time$ ($curr\_wave$ and rem_time for short), representing a logical timer. The idea is that $curr\_wave$ indicates the current wave for which liveness is guaranteed (until pacemaker intervention), and rem_time is the minimum period of time the pacemaker guarantees not to intervene. rem_time should decrease by 1 within each period of Δ. If rem_time reaches 0, the pacemaker shall move the system to the next wave. The pacemaker also allows $leader\_of(w)$ to call $StartNext(w)$ to start the next wave before timer expiration, if it completes its tasks early.

The LiDO-DAG concurrent object is thus defined as a transition system specified by Alg. 2, Alg. 4, and Alg. 5, with each **upon** clause representing an atomic transition event.

3.3 The Consensus Log

We now define the DAG consensus log, providing a formal safety criterion for DAG-based protocols.

Definition 3.1. The *anchor log* is defined as $\log(c)$ where c is the CCache with the highest wave number in Σ. The anchor log is empty in case no CCache exists in Σ.

The following theorem easily follows from the key invariant of LiDO cache tree:

THEOREM 3.2 (SAFETY OF LIDO-DAG). *If z, z' are states of LiDO-DAG and z' is reachable from z, then $anchor\_log(z')$ extends from $anchor\_log(z)$.*

The DAG consensus log is defined by expanding each anchor into its closure. To convert the closure set to a linear list, we assume a parameter δ, a deterministic topological sort algorithm.

Definition 3.3. If the anchor log is $[id_1, \cdots, id_n]$, and $verts[id_i] = u_i$, then the *DAG consensus log* is $\mathsf{dedup}(\delta(\mathrm{cl}(u_1)) \mathbin{++} \cdots \mathbin{++} \delta(\mathrm{cl}(u_n)))$, where dedup is the deduplication function.

Definition 3.4 (Safety of DAG-based protocols). A DAG-based protocol D is safe, if 1) it is possible to construct a refinement-mapping ϕ from a network model of D to the LiDO-DAG model, and 2) whenever a participating process outputs a log, it is a prefix of the current DAG consensus log.

3.4 Liveness and Fairness of LiDO-DAG

The abstract pacemaker enables refinement-based liveness proofs, which has been successfully applied to BFT protocols like Jolteon [Gelashvili et al. 2022] in Qiu et al. [2024b]. We now apply their ideas to DAG-based protocols.

Liveness refinement is based on an abstraction called *trace segmentation.* Assume that an infinite timed-trace τ is non-Zeno, i.e. only a finite number of events occur in every finite period. Let T be any timepoint after GST, and Δ the network latency after GST. The idea is that τ can be represented as the limit of $\tau_0, \tau_1, \cdots$, where τ_k is the prefix of τ containing only events with $time < T + k\Delta$. Since the trace is non-Zeno, each of $\tau_0, \tau_1, \cdots$ is finite. Each τ_i is also a prefix of τ_{i+1}. We will use $(\tau, \tau']$ to denote the trace τ' with prefix τ removed. Liveness assumptions such as partial-synchrony can be stated as safety properties over finite and contiguous subsequences of the segmentation.

If ϕ is a refinement mapping from a protocol D to LiDO-DAG, and $\tau_0, \tau_1, \cdots$ is the segmentation of an infinite network trace, then $\phi(\tau_0), \phi(\tau_1), \cdots$ is the segmentation of the corresponding LiDO-DAG trace. The definition of refinement mapping guarantees that ϕ is prefix-preserving. A protocol D is live if all live traces of its network model refine live traces of LiDO-DAG (Definition 3.5).

Definition 3.5. An infinite segmented trace $\tau_0, \tau_1, \cdots$ of LiDO-DAG is live, if each of its finite contiguous subsequence $\tau_i, \cdots, \tau_{i+k}$ satisfies the following conditions:

(1) (Liveness of *DAG-Invoke*) There exists constant C, s.t. if a synchronous thread has called *DAG-Invoke*$(r, \_, \_)$ before the end of τ_i, then it receives response before the end of τ_{i+C};
(2) (Liveness of *DAG-Pull*) There exists constant C, s.t. if a synchronous thread has called *DAG-Pull*(r) before the end of τ_i, then it receives response before the end of τ_{i+C}, provided that either 1) at least one other synchronous thread has called *DAG-Invoke*$(r, \_, \_)$, or 2) all synchronous have called *DAG-Invoke*$(r-1, \_, \_)$ and received response;
(3) (Liveness of DAG-building) When execution begins, each synchronous DAG-builder thread immediately calls *DAG-Pull*(1); they immediately call *DAG-Invoke*$(r, \_, \_)$ after *DAG-Pull*(r) returns, and immediately call *DAG-Pull*$(r+1)$ after *DAG-Invoke*$(r, \_, \_)$ returns;

(4) (Liveness of LiDO) There exists constant C, s.t. if $curr\_wave(\tau_i) < w$, $curr\_wave(\tau_{i+1}) \geq w$, and $leader\_of(w)$ is synchronous, then there exists a CCache of wave w at the end of τ_{i+C};
(5) (Liveness of abstract pacemaker) 1) Within $(\tau_i, \tau_{i+1}]$, $Elapse()$ is called at most once, and it cannot be called after a valid $TimeoutStartNext()$ call; 2) If $rem\_time(\tau_i) > 0$, then within $(\tau_i, \tau_{i+1}]$ either $Elapse()$ or $TimeoutStartNext()$ is called at least once; 3) There exists constant C, s.t. if $rem\_time(\tau_i) = 0$ then $curr\_wave(\tau_{i+C}) > curr\_wave(\tau_i)$.

Conditions (4) and (5) are consistent with Qiu et al. [2024b] and ensure that every wave started after GST will eventually have a CCache. Conditions (1)-(3) are new in LiDO-DAG and ensure every round of DAG will eventually have at least $2f + 1$ vertices.

Liveness of Consensus. We proved the following theorem for all live traces of LiDO-DAG:

Theorem 3.6 (Liveness of LiDO-DAG). *For each vertex v built by a synchronous process, eventually there exists an anchor v' in the anchor log, s.t. $v \in \mathrm{cl}(v')$.*

The outline of the proof is as follows. First, the DAG part enforces that (line 21 and 23 of Alg. 2), when an honest thread p_i attempts to create a new vertex v, the closure of v must include all vertices ever created by p_i. This means if one of the vertices created by p_i is committed, then all previous vertices are also committed.

Second, the LiDO part enforces that (line 10 of Alg. 4), when an honest LiDO-proposer becomes the leader, it must propose the latest vertex created by the same process as the new anchor. This requirement avoids the stagnant anchor problem mentioned in Section 2.4. It ensures new vertices are eventually included in the closures of new anchor proposals.

Finally, requirements (4)-(5) of Definition 3.5 guarantee the newly proposed anchor will be committed. Assuming that the leader schedule is fair (all processes become leader infinitely often), this ensures all vertices from synchronous threads will eventually be committed.

Progress of DAG-Pull. The rationale of the liveness requirement around $\textit{DAG-Pull}(r)$ is as follows. If at least one synchronous thread has called $\textit{DAG-Invoke}(r, \_, \_)$, then it has already learned $2f + 1$ vertices in round $r - 1$, and it should forward this knowledge to threads still waiting upon $\textit{DAG-Pull}(r)$, so they also learn them within Δ.

If all synchronous threads have returned from $\textit{DAG-Invoke}(r - 1, \_, \_)$, then either all of them succeeded, or at least one of them received $Timeout$. In either case, there exists at least $2f + 1$ vertices in round $r - 1$. Since vertices are gossiped among synchronous threads, they should all learn them within Δ. Based on these considerations, we proved that:

Theorem 3.7 (Progress of DAG). *Each DAG round r will eventually contain at least $2f + 1$ vertices from different DAG-builder threads.*

Fairness of DAG. Theorem 3.7 says nothing about individual DAG-builders. It is possible that $2f + 1$ vertices are created in each round, but some synchronous DAG-builder threads are starved.

The simplest way to formalize the fairness requirement is that after GST, all new calls to $\textit{DAG-Invoke}(r, \_, \_)$ return *Success*. However this requirement is impractical: even after GST it is occasionally necessary to skip rounds and catch up with network progress, for example if network speed is faster for some honest builder than others. Instead we use the following definition:

Definition 3.8 (Fairness of DAG-based protocols). There exists constant C, s.t. each synchronous DAG-builder creates at least one new vertex within $(\tau_i, \tau_{i+C}]$.

This does not guarantee absolute fairness: some threads can still create vertices more frequently than others. However it gives a minimum vertex creation frequency for all synchronous threads.

4 Implementations of LiDO-DAG

4.1 System Model

We produced several different implementations of the LiDO-DAG object, showing good reusability of our model. In this section, we describe the general structure of our network models. Features of individual implementations will be described in subsequent sections following this skeleton.

We specify each honest process as a state machine that interacts with three ideal objects (Fig. 7), namely the network object (providing message sending and delivery), the reliable broadcast object (RBC, specified in Alg. 1), and a local timer object. Byzantine processes do not have internal state. Instead, they are allowed to interact with the network object and the RBC object arbitrarily.

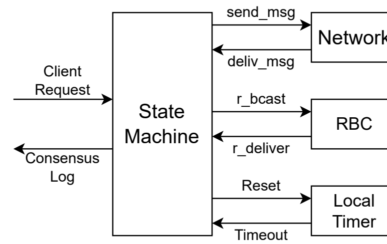

Fig. 7. Model of each honest process.

The Network Object. The internal state of the network object is a set of *messages*. Each message must come from a predefined set $\mathcal{M}$ called the *message space*. The message space is protocol-specific. Each message is signed by a *sender*. We assume byzantine processes cannot forge signatures of honest processes. For example, in consensus protocols, byzantine processes cannot send votes on behalf of other processes. On the other hand, we follow Dolev and Yao [1983] and allow the adversary to see every sent message and send any message signed by byzantine processes. Message delivery is UDP-like: sent messages may be delivered in any possible order. Although each message has a set of *intended recipients*, the message may be delivered to any process in the system.

The Local Timer. Each honest process is equipped with a local timer object, specified in Alg. 6. It has a single variable $local\_rem\_time$. The state machine may call $Reset()$ to reset $local\_rem\_time$ to a fixed value $timer\_reset\_val$. The adversary may call $Elapse()$ to decrease $local\_rem\_time$. If $Elapse()$ is called when $local\_rem\_time = 0$, a timeout signal is delivered.

Algorithm 6 Local Timer

Constant: $timer\_reset\_val$: nat
State variable: $local\_rem\_time$: nat
initialize:
  $local\_rem\_time \leftarrow timer\_reset\_val$
upon $Reset()$ from state machine:
  $local\_rem\_time \leftarrow timer\_reset\_val$
upon $Elapse()$ from $\mathcal{A}$:
  if $local\_rem\_time > 0$ **then**
    $t \leftarrow local\_rem\_time$
    $local\_rem\_time \leftarrow t - 1$
  else
    Send signal $timeout()$

Liveness Assumptions. Infinite timed-traces of the network model are segmented by Δ in the same way described in Section 3.4. To prove liveness, we consider all segmented traces that satisfy Definition 4.1 and show that, under the refinement mapping ϕ from the safety proof, they refine live traces of LiDO-DAG.

Definition 4.1. An infinite segmented trace $\tau_0, \tau_1, \cdots$ of the network model is live, if each of its finite contiguous subsequence $\tau_i, \cdots, \tau_{i+k}$ satisfies:

(1) (Liveness of network) If a synchronous process p_i has sent a message m before the end of τ_i, then it is delivered to every synchronous recipient at least once before the end of τ_{i+1};
(2) (Liveness of timer) For each local timer, within $(\tau_i, \tau_{i+1}]$, 1) either $Elapse()$ or $Reset()$ is called at least once; 2) $Elapse()$ is called at most once; 3) $Elapse()$ cannot be called after $Reset()$;
(3) (Liveness of RBC) The liveness requirements in Alg. 1 are satisfied.

Implementing DAG using Reliable Broadcast. The protocols we considered in this work are all authenticated DAG-based protocols, which are easier to model than unauthenticated ones. Although they implement consensus differently, the way they implement the DAG graph is largely the same.

Algorithm 7 Building DAG From RBC

State variable: $local\_verts$: `list Vertex`; $buffer$: `list (nat * nat * val * list nat)`.
initialize:
  $local\_verts \leftarrow \{\}$; $buffer \leftarrow \{\}$
procedure TRYADDVERT($r, k, v, preds$)
  $id \leftarrow r(3f+1)+k$
  if $local\_verts[id] \neq \bot$, or $CheckRegister(v) = false$, or $local\_verts[p\_id] = \bot$ for some $p\_id \in preds$ **then**
    return
  if $local\_verts[p\_id].round \geq r$ for some $p\_id \in preds$ **then**
    return
  $strongEdges \leftarrow$ `filter` $(p\_id \mapsto verts[p\_id].round = r-1)$ $preds$
  $strongEdgeBuilders \leftarrow$ `map` $(p\_id \mapsto verts[p\_id].builder)$ $strongEdges$
  if $r > 1 \land isQuorum(strongEdgeBuilders) = false$ **then**
    return
  $v \leftarrow \{id := id; round := r; builder := p_k; data := v; preds := preds\}$
  $local\_verts \leftarrow local\_verts \cup \{v\}$
upon $r\_deliver(m, r, p_k)$ from RBC:
  Interpret m as $(v, preds)$.
  $buffer \leftarrow buffer \cup \{(r, k, v, preds)\}$
  Call TRYADDVERT($r, k, v, preds$) for each $(r, k, v, preds) \in buffer$ until no entry can be added to $local\_verts$.

Each process is given access to an infinite number of instances of reliable broadcast (RBC), numbered from 1. The instance number is specifed with parameter r in Alg. 1.

Each honest process maintains a local view of the DAG graph (Alg. 7). When RBC sends $r\_deliver(m, r, p_k)$, it builds a vertex record (Fig. 5) with $builder = p_k$ and $round = r$. It interprets m as a tuple of $data$ and $preds$, and id is implicitly defined from $builder$ and $round$. It then checks whether the pointers in $preds$ lead to known vertices in the local view. If so, the new vertex is added to the local view. Otherwise, it buffers the record until these pointers can be resolved.

To prove that the procedure outlined above refines the DAG object, we can imagine there is a global observer of all RBC instances. Whenever an honest or byzantine process calls $r\_bcast(m, r)$, the value is immediately delivered to the observer. The observer runs the same procedure to build its local view of the graph. The global DAG graph is the local graph built by this observer. It is easy to see that a vertex must be added to this global graph before it can be added to the local graph of any honest process. Thus each local view is a subgraph of this global graph.

4.2 Narwhal

Narwhal implements consensus using an external BFT algorithm. The original combination described by Danezis et al. [2022] is called **Narwhal-HS** where HS is the HotStuff [Yin et al. 2019] protocol. Here we reused the Jolteon [Gelashvili et al. 2022] implementation provided by Qiu et al. [2024b]. In each wave (*view* in Jolteon), we make the leader propose a single vertex to be committed as the new anchor. The main challenge is to ensure all entries proposed by leaders are valid vertices.

Our implementation buffers the BFT request messages until the voter has added the proposed vertex to its local view. This introduces some delay between receiving a BFT request and processing it. However when a synchronous process proposes a vertex v in Jolteon, it must have already received v through RBC, and RBC will deliver v to all other synchronous processes within Δ (with suitably large Δ). Hence the request message is still processed by every synchronous process within Δ, and the liveness proof still works. Overall, we were able to integrate the Jolteon implementation into DAG with minimal changes to its safety and liveness proofs.

Algorithm 8 Summary of Bullshark

1: State variable: $local\_curr\_round$: `nat`.
2: Notation: $isWeakQuorum(P) = true$ if P is a set of at least $f + 1$ processes.
3: Notation: $isAnchor(v) = true$ if $v.round = 2w - 1$ and $v.builder = leader\_of(w)$ for some w.
4: Notation: $local\_curr\_wave = \lfloor(local\_curr\_round + 1)/2\rfloor$.
5: **initialize**:
6: $\quad$ $local\_curr\_round \leftarrow 1$
7: $\quad$ Propose vertex in round 1.
8: **procedure** COMMITCONDITION(w)
9: $\quad$ $commitVotes \leftarrow$ `filter` $(v \mapsto v.round = 2w \wedge (2w - 1, leader\_of(w)) \in v.preds)$ $local\_verts$
10: $\quad$ $commitVoteBuilders \leftarrow$ `map` $(v \mapsto v.builder)$ $commitVotes$
11: $\quad$ **return** $isWeakQuorum(commitVoteBuilders)$
12: **procedure** GETANCHORLOG(w)
13: $\quad$ $v \leftarrow$ anchor of wave w; $anchorLog \leftarrow [\,]$
14: $\quad$ **while** $v \neq \bot$ **do**
15: $\quad\quad$ $anchorLog \leftarrow v :: anchorLog$
16: $\quad\quad$ $v \leftarrow \arg\max_{v' \in \mathrm{cl}(v),\, v' \neq v,\, isAnchor(v') = true} \lfloor(v'.round + 1)/2\rfloor$
17: $\quad$ **return** $anchorLog$
18: **procedure** ADVANCEROUNDCONDITION(r)
19: $\quad$ $prevRoundVerts \leftarrow$ `filter` $(v \mapsto v.round = r - 1)$ $local\_verts$
20: $\quad$ $prevRoundBuilders \leftarrow$ `map` $(v \mapsto v.builder)$ $prevRoundVerts$
21: $\quad$ **if** $r \leq local\_curr\_round$, or $isQuorum(prevRoundBuilders) = false$ **then**
22: $\quad\quad$ **return** $false$
23: $\quad$ **if** $r = 2w$ for some w **then**
24: $\quad\quad$ **return** $true$ if anchor of wave w received, or $local\_curr\_round = 2w - 1$ and local timer has expired.
25: $\quad\quad$ **return** $false$ otherwise.
26: $\quad$ **else**
27: $\quad\quad$ **return** $true$
28: **upon** ADVANCEROUNDCONDITION(r) $= true$ for some r:
29: $\quad$ $local\_curr\_round \leftarrow r$
30: $\quad$ Propose vertex in round r.
31: $\quad$ Reset timer to $T_{RBC} + 1$ if $local\_curr\_wave$ has increased.
32: **upon** COMMITCONDITION(w) $= true$ for some w:
33: $\quad$ Output GETANCHORLOG(w) as the new anchor log.

4.3 Bullshark

Bullshark [Spiegelman et al. 2022b] implements consensus by utilizing edges in the DAG graph as commit votes, avoiding the need for a separate BFT protocol. The basic concepts of Bullshark were introduced in Section 2.2, and Alg. 8 shows a summary of the protocol. In this section, we focus on showing how Bullshark refines the safety and liveness requirements of LiDO-DAG.

Safety. To prove safety, we construct a refinement mapping ϕ, which defines a LiDO cache tree from the global DAG graph. We show that whenever an honest process outputs an anchor log (line 12-17 of Alg. 8), it is the consensus log of a CCache, which must be a prefix of the global anchor log.

In Bullshark, each wave w corresponds to two DAG rounds ($2w - 1$ and $2w$). The anchor of wave w is the vertex created by $leader\_of(w)$ in round $2w - 1$. Our refinement mapping is as follows. We create an ECache and an MCache of wave w simultaneously, when the anchor v of wave w is created. To compute $ecache.parent\_wave$, we look at the closure $\mathrm{cl}(v)$. If $\mathrm{cl}(v)$ contains no anchors, then $ecache.parent\_wave = 0$. Otherwise, we find the anchor in $\mathrm{cl}(v)$ with the highest wave number, and take that wave number as $ecache.parent\_wave$. Thus in Fig. 2, since $\mathrm{cl}(A_2)$ contains no anchors, $\Sigma[2].ecache$ is attached directly to $Root$ (Fig. 3).

CCache of wave w is created when there exists $f + 1$ vertices (a weak quorum) in round $2w$ that embed pointers to the anchor of wave w. To see how this definition maintains the key invariant of LiDO, notice that a quorum and a weak quorum must intersect on at least one process. If $f + 1$ vertices in the DAG round $2w$ contain pointers to the anchor v, and every vertex in round $2w + 1$ embeds pointers to $2f + 1$ vertices in round $2w$, then every vertex in round $2w + 1$ must contain v in its closure. By induction, this applies to every round $r \geq 2w + 1$. Hence anchors of every wave $w' > w$ also contain v in their closures, and $\Sigma[w'].ecache.parent\_wave \geq w$.

Now observe that in Alg. 8, if commitCondition(w) returns *true*, then a CCache of round w exists. The procedure getAnchorLog(w) begins from the anchor of wave w, and every iteration of line 16 corresponds to moving to the previous entry of the consensus log. Hence getAnchorLog(w) always returns the consensus log of a CCache, which completes the safety proof.

Liveness. An intuitive way to describe liveness of Bullshark is as follows. Each process enters wave w when it observes $2f + 1$ vertices of round $2w - 2$. It first proposes a vertex in round $2w - 1$, then resets the timer to $T_{RBC} + 1$ and waits for the anchor of wave w (line 28-31 of Alg. 8). If it receives the anchor before the timer expires, it will create a vertex in round $2w$ with a pointer to the anchor; otherwise it creates a vertex without that pointer. In any case, eventually there will be $2f + 1$ vertices in round $2w$, and every process will enter wave $w + 1$.

To formalize this liveness argument, we use the LiDO-DAG model to decompose it into a number of safety invariants (Definition 3.5). To begin, we define how the local timers implement the abstract pacemaker (Alg. 5). Let P be the set of all synchronous processes. For each $p \in P$, we use $local\_curr\_wave(p)$ to denote the highest wave p has ever entered (line 4 of Alg. 8). Then the state variables of the abstract pacemaker are computed as: $curr\_wave = \max_{p \in P} local\_curr\_wave(p)$, and $rem\_time = \min_{p \in P,\, local\_curr\_wave(p) = curr\_wave} local\_rem\_time(p)$. That is, $curr\_wave$ is the highest wave any synchronous process has ever entered, and rem_time is the least local remaining time among those synchronous processes currently in $curr\_wave$.

It remains to show that all invariants in Definition 3.5 are satisfied. The complete proof is described in Appendix B. Here we present a key part of it, namely liveness of consensus. It states if $curr\_wave(\tau_i) < w$, $curr\_wave(\tau_{i+1}) \geq w$, and $leader\_of(w)$ is synchronous, then a CCache of wave w will be created. We first observe that if $curr\_wave(\tau_i) < w$, then no process has timed-out in wave w (since no process has entered that wave), nor will anyone timeout in wave w before the end of $\tau_{i+T_{RBC}+2}$. Now if a synchronous process creates a vertex v in round $2w$, then either it has timed-out in wave w, or v contains a pointer to anchor of wave w. Together they imply:

Lemma 4.2. *If $curr\_wave(\tau_{i+k}) > w$ for some $1 \leq k \leq T_{RBC} + 2$, then a CCache of wave w exists by the end of τ_{i+k}.*

The reason being, if $curr\_wave(\tau_{i+k}) > w$ then at least $2f + 1$ vertices exist in round $2w$, of which at least $f + 1$ come from synchronous processes. They must contain pointers to the anchor of wave w, hence the anchor is committed.

Hence we only consider the case where $curr\_wave(\tau_{i+k}) = w$ for every $1 \leq k \leq T_{RBC} + 2$. Since at least one synchronous process has received $2f + 1$ vertices in round $2w - 2$ by the end of τ_{i+1}, all synchronous processes including $leader\_of(w)$ will enter wave w by the end of τ_{i+2}. Thus by the end of $\tau_{i+T_{RBC}+2}$, all synchronous processes will have received at least $2f + 1$ vertices in round $2w - 1$, including the anchor. They will all create vertices in round $2w$ that contain pointer to the anchor. Hence the anchor is committed.

4.4 Sailfish

One of the drawbacks of the Bullshark protocol is its long commit latency and low anchor frequency. This leads to a number of works [Arun et al. 2024; Shrestha et al. 2025; Spiegelman et al. 2023] proposing ideas for admitting more anchors. We now look at the single-leader Sailfish protocol [Shrestha et al. 2025] which is easier to implement.

Sailfish optimizes Bullshark in a way analogous to how pipelining optimizes HotStuff [Yin et al. 2019]. It reduces each wave to just one DAG round, but each round now simultaneously serves to introduce a new anchor and commit the anchor in the previous round. Although this sounds simple, the actual differences from Bullshark are complex, and the proofs very subtle. Fig. 8 shows what could go wrong if we only reduce the wave length without changing other components of Bullshark. In this figure, there are 2 edges from vertices in round 2 to the anchor A_1, so it seems A_1 is committed. However the anchor A_2 is also committed, but A_1 is not in the closure of A_2, so we have committed two conflicting logs, a safety violation.

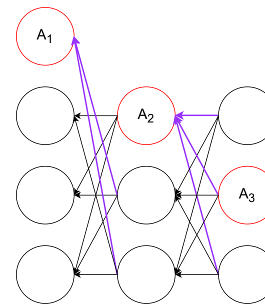

Fig. 8. A counterexample to a “naive” version of Sailfish.

To avoid the kind of paradox shown above, the commit rule needs to be carefully designed, so that if the anchor A_{r-1} is committed, then the anchor A_r (if it exists) must embed a pointer to it. Sailfish resolves this issue by introducing $Commit$, $Timeout$, and $NoVote$ messages. Specifically: when a process p_k enters round $r+1$, it either sends $\langle Commit, r, p_k \rangle$ to $leader\_of(r)$ if it has received the anchor A_r, or it sends $\langle NoVote, r+1, p_k \rangle$ to $leader\_of(r+1)$, if it has not seen A_r. When $leader\_of(r+1)$ proposes the new anchor A_{r+1}, it must embed either a pointer to the anchor of round r, or $2f+1$ $NoVote$ messages of round $r+1$.

The $\langle Commit, \_, \_ \rangle$ message does not appear in the original presentation of Sailfish. Instead it was called “the first message of RBC.” In Appendix C, we describe the full details of Sailfish and its proofs. Here, we merely notice that correctness of the scheme hinges on that each honest process p will send out either $\langle Commit, r, p \rangle$ or $\langle NoVote, r+1, p \rangle$, but never both. Thus if we have $2f+1$ Commit messages, it is not possible to collect $2f+1$ NoVote messages, so the anchor of round $r+1$ must embed a pointer to the anchor of round r.

The Extra Latency Problem of Sailfish. Although Sailfish resolves the paradox around consecutive anchors, it introduces a new latency problem: if $leader\_of(r)$ is byzantine and refuses to create an anchor, then $leader\_of(r+1)$ will have to wait until everyone else enters round $r+1$ and sends it $NoVote$ messages, before it may propose its vertex. The timer duration also has to be lengthened to accommodate the worst-case extra latency. Shrestha et al. [2025] summarizes the consequence as follows: *when there is a sequence of two or more faulty leaders in between honest leaders, ... our protocol would slightly underperform compared to Bullshark in terms of latency.*

Table 2. Statistics of proof effort

Component		Lines
LiDO-DAG model	specs	747 (586 imported)
	proofs	1500 (397 imported)
Narwhal	specs	2307 (1706 imported)
	proofs	9697 (6813 imported)
Bullshark	specs	603
	proofs	4343
Sailfish (original)	specs	270
	proofs	3479
Sailfish (with modification)	specs	339
	proofs	3040

It is natural to ask whether this extra latency can be eliminated. During our attempt to formalize Sailfish, we realized this is in fact possible. Note that the purpose of $NoVote$ messages is to ensure that, whenever we see $2f+1$ vertices of round $r+1$ embedding pointers to the anchor A_r, we can be sure the anchor A_{r+1} (if it exists) must also contain a pointer to A_r. Thus if we simply remove $NoVote$ messages, then we cannot commit A_r until we receive A_{r+1} and check that A_{r+1} contains a pointer to A_r. This is analogous to the 2-chain rule of Jolteon, and implies we need $leader\_of(r)$

and $leader\_of(r+1)$ to be both synchronous to ensure A_r can be committed. However, with a simple modification, we were able to remove this consecutive leader requirement.

Our modification is to add a field in timeout messages that indicates the latest anchor the sender has received. When the local timer of party p_k expires, p_k now sends $\langle Timeout, r, qc_{high}, p_k \rangle$ where qc_{high} is the highest round $r' \leq r$ whose anchor $A_{r'}$ has already been received by p_k. We require that if a vertex of round r embeds $2f+1$ timeouts, then its closure must include the highest anchor referenced by these timeouts. Hence if there exists $2f+1$ commit votes of round r, all future vertices must include the anchor A_r in their closures. See Appendix C for details of liveness.

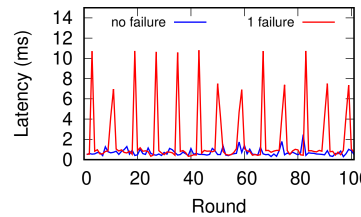

Fig. 9. Bullshark latency to advance rounds with and without failures.

4.5 Proof Effort

Table 2 shows the statistics of our Coq proofs. Proofs for Narwhal take up the largest size, but a significant portion of it is imported from Qiu et al. [2024b]. The additional effort to adapt it to the DAG setting is comparatively small. The proofs were written by two persons over three months.

5 Experimental Evaluation

To demonstrate that our verified Coq specification is detailed enough and realistic, we extracted the lowest layers of Narwhal, Bullshark, and Sailfish into executable OCaml code. The extracted OCaml code follows the process model of Fig. 7 and implements RBC with quorum signing, but lacks network and timer facility. We linked the code to OCaml's Unix libraries, but the core logic remains unchanged. The code was tested on a local cluster with four nodes, each with Intel Xeon Gold 6338 CPU, 128 GB memory, and 10 GigE NIC. We compared our code against verified Jolteon code from [Qiu et al. 2024a,b].

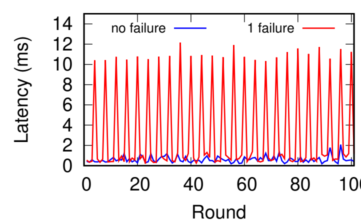

Fig. 10. Narwhal latency to advance rounds with and without failures.

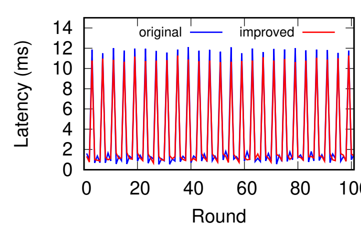

Fig. 11. Comparison of latencies of original and modified Sailfish with 1 failed node.

We ran Bullshark and Narwhal without any failures and with one crashed node while the timeout is set to 10 ms. We measured the latency to advance a round and throughput for committing blocks. Fig. 9 and Fig. 10 measure the latency to advance the DAG-round from 1 to 100. On average, it takes 688 and 594 μs in the absence of failure and 3,166 and 1,998 μs with one failed node to advance a round for Bullshark and Narwhal, respectively. The spikes in the figures capture the cases when it is the failed node's turn to act as a leader and the timeout kicks in to advance the round. In contrast, it took on average 3,861 μs and 5,690 μs to advance a view for Jolteon (not in the figure) with and without one failure, respectively. Note that a view of Jolteon contains two phases and corresponds to roughly two DAG-rounds.

Table 3. Throughput (Commits/s)

Protocols	without failures	with 1 failure
Bullshark	2,917	1,444
Narwhal	5,857	4,851
Jolteon	267	173

The throughput measurement (Table 3) more clearly shows the benefit of DAG-based protocols against leader-based protocols: the throughput of Bullshark and Narwhal are over 10 to 21 times higher without failures and 8 to 28 times higher with one node failure than Jolteon. The result clearly shows that DAG-based protocols which process new blocks in all nodes in parallel outperform the leader-based protocol which processes new blocks only in the leader node.

We also measured the latencies of the two versions of Sailfish we verified. In the original version of Sailfish, if the leader of round r is faulty, then everyone except the leader of round $r + 1$ enters round $r + 1$ within 8Δ. However, the leader of round $r + 1$ needs one extra Δ to collect the $NoVote$ messages. Our modified Sailfish removes the $NoVote$ messages. As shown in Fig. 11, in rounds where the leader has crashed, the original Sailfish consistently requires ~1 ms longer failover time than the modified Sailfish, which demonstrates the effectiveness of our optimization.

6 Related Works and Limitations

The literature on the designs of DAG-based protocols has been surveyed in Section 2. Further survey can be found in [Raikwar et al. 2024]. Here we focus on works on verifying DAG-based consensus and consensus in general.

Two existing works [Bertrand et al. 2024; Crary 2021] attempted to verify DAG-based protocols. Crary [2021] used Coq to model Hashgraph [Baird 2016], an asynchronous protocol based on random coins. They studied both safety and liveness, but liveness is based on an ad-hoc axiom with on-paper justification. Their approach is specialized to a single protocol. Bertrand et al. [2024] provided a generic safety model of DAG-based protocols, and applied it to Bullshark [Spiegelman et al. 2022a] and Cordial Miners [Keidar et al. 2023], but without liveness proof. Their model is based on the local views of individual processes and their mutual consistency. By contrast, our approach is based on introducing a virtual global state of DAG and consensus, and proving every local view is consistent with this global state. We believe our approach is conceptually cleaner and scales better with complex protocols, as it enables proving invariants at an abstract level. Thomsen and Spitters [2021] verified Nakamoto-style proof-of-stake, which is a kind of unstructured DAG. Similar to Crary [2021], the probabilistic portion of the liveness proof is not verified.

There are extensive studies [Berkovits et al. 2019; Bertrand et al. 2022; Bravo et al. 2022; Carr et al. 2022; Cirisci et al. 2023; Drăgoi et al. 2016; Hawblitzel et al. 2015; Konnov et al. 2023; Losa and Dodds 2020; Padon et al. 2017; Qiu et al. 2024b; Rahli et al. 2018; Taube et al. 2018; Vukotic et al. 2019; Wilcox et al. 2015; Woos et al. 2016; Zhao et al. 2024] on verification of either benign or byzantine leader-based consensus. They can be classified into model-checking approaches (e.g. Berkovits et al. [2019]; Bertrand et al. [2022]; Losa and Dodds [2020]) and proof-checking approaches (e.g. Carr et al. [2022]; Rahli et al. [2018]; Zhao et al. [2024]). While model-checking provides greater automation, it is challenging to apply it to partially-synchronous liveness, and existing works mostly focus on safety. The tricky part of liveness reasoning is to keep track of local timers and ordering between message delivery and timeout events. Sun et al. [2024] verified liveness of a cluster controller, but their liveness reasoning does not need to deal with local timers. Hawblitzel et al. [2015] verified liveness of Multi-Paxos. Their reasoning is completely at network level, and one can expect that they relied on a large number of complex invariants. In our engineering experience, formulating these invariants is extremely error-prone. It is easier to first formulate them on simpler models like LiDO-DAG, and then translate them down to network-level properties.

The way LiDO and LiDO-DAG currently models external validity is not completely ideal. It postulates that as soon as a value gets registered, all voters can immediately check its validity. This notion becomes problematic as we compose consensus with RBC: there is a gap between broadcasting a value and learning the value. Thus although we were able to adapt the Jolteon implementation from Qiu et al. [2024b], it was not as effortless as we expected. Making consensus aware of this timing-gap is future work.

Our work is currently limited to partially synchronous protocols. To model liveness of asynchronous protocols would require a theory of probabilistic refinement. Bertrand et al. [2021] has introduced a model-checking technique for verifying randomized distributed algorithms. Introducing their techniques into asynchronous DAG-based protocols is future work.

Acknowledgments

We would like to thank our anonymous reviewers for their helpful feedback. This material is based upon work supported in part by NSF grants 2019285 and 2313433, and by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4018. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

Artifact-Availability Statement

The artifact accompanying this paper is available on Zenodo [Qiu et al. 2025a].

References

Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. 2024. Shoal++: High Throughput DAG BFT Can Be Fast! arXiv:2405.20488 [cs.DC] https://arxiv.org/abs/2405.20488

Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. 2024. Mysticeti: Reaching the Limits of Latency with Uncertified DAGs. arXiv:2310.14821 [cs.DC] https://arxiv.org/abs/2310.14821

Leemon Baird. 2016. *The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance.* Technical Report SWIRLDS-TR-2016-01. Swirlds. https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf

Idan Berkovits, Marijana Lazić, Giuliano Losa, Oded Padon, and Sharon Shoham. 2019. Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 245–266. https://doi.org/10.1007/978-3-030-25543-5_15

Nathalie Bertrand, Pranav Ghorpade, Sasha Rubin, Bernhard Scholz, and Pavle Subotic. 2024. Reusable Formal Verification of DAG-based Consensus Protocols. arXiv:2407.02167 [cs.LO] https://arxiv.org/abs/2407.02167

Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazić, Pierre Tholoniat, and Josef Widder. 2022. Holistic Verification of Blockchain Consensus. In *36th International Symposium on Distributed Computing (DISC 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:24. https://doi.org/10.4230/LIPIcs.DISC.2022.10

Nathalie Bertrand, Igor Konnov, Marijana Lazić, and Josef Widder. 2021. Verification of randomized consensus algorithms under round-rigid adversaries. *International Journal on Software Tools for Technology Transfer* 23, 5 (2021), 797–821. https://doi.org/10.1007/s10009-020-00603-x

Manuel Bravo, Gregory V. Chockler, and Alexey Gotsman. 2022. Liveness and Latency of Byzantine State-Machine Replication. In *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA (LIPIcs, Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 12:1–12:19. https://doi.org/10.4230/LIPIcs.DISC.2022.12

Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2019. The latest gossip on BFT consensus. arXiv:1807.04938 [cs.DC] https://arxiv.org/abs/1807.04938

Vitalik Buterin. 2014. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf

Vitalik Buterin. 2024. Possible futures of the Ethereum protocol, part 2: The Surge. https://vitalik.eth.limo/general/2024/10/17/futures2.html

Harold Carr, Christa Jenkins, Mark Moir, Victor Cacciari Miraldo, and Lisandra Silva. 2022. Towards Formal Verification of HotStuff-Based Byzantine Fault Tolerant Consensus in Agda. In *NASA Formal Methods*, Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer International Publishing, Cham, 616–635. https://doi.org/10.1007/978-3-031-06773-0_33

Miguel Castro. 2001. *Practical Byzantine Fault Tolerance.* Ph. D. Dissertation. Massachusetts Institute of Technology. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/thesis-mcastro.pdf

Berk Cirisci, Constantin Enea, and Suha Orhun Mutluergil. 2023. Quorum Tree Abstractions of Consensus Protocols. In *Programming Languages and Systems*, Thomas Wies (Ed.). Springer Nature Switzerland, Cham, 337–362. https://doi.org/10.1007/978-3-031-30044-8_13

CoinGecko. 2024. *Top Blockchains by Total Value Locked (TVL).* https://www.coingecko.com/en/chains

Karl Crary. 2021. Verifying the Hashgraph Consensus Algorithm. arXiv:2102.01167 [cs.LO] https://arxiv.org/abs/2102.01167

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*

(Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 34–50. https://doi.org/10.1145/3492321.3519594

D. Dolev and A. Yao. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory* 29, 2 (1983), 198–208. https://doi.org/10.1109/TIT.1983.1056650

Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: A Partially Synchronous Language for Fault-Tolerant Distributed Algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 400–415. https://doi.org/10.1145/2837614.2837650

Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *Journal of the ACM* 35, 2 (April 1988), 288–323. https://doi.org/10.1145/42282.42283

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

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*, Ittay Eyal and Juan Garay (Eds.). Springer International Publishing, Cham, 296–315. https://doi.org/10.1007/978-3-031-18283-9_14

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* (Zurich, Switzerland) *(AFT '19)*. Association for Computing Machinery, New York, NY, USA, 214–228. https://doi.org/10.1145/3318041.3355467

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972

Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All You Need is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (Virtual Event, Italy) *(PODC'21)*. Association for Computing Machinery, New York, NY, USA, 165–175. https://doi.org/10.1145/3465084.3467905

Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. 2023. Cordial Miners: Fast and Efficient Consensus for Every Eventuality. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPICS.DISC.2023.26

Igor Konnov, Marijana Lazić, Ilina Stoilkovska, and Josef Widder. 2023. Survey on Parameterized Verification with Threshold Automata and the Byzantine Model Checker. *Logical Methods in Computer Science* Volume 19, Issue 1 (Jan. 2023). https://doi.org/10.46298/lmcs-19(1:5)2023

Andrew Lewis-Pye, Dahlia Malkhi, Oded Naor, and Kartik Nayak. 2024. Lumiere: Making Optimal BFT for Partial Synchrony Practical. arXiv:2311.08091 [cs.DC] https://arxiv.org/abs/2311.08091

Giuliano Losa and Mike Dodds. 2020. On the Formal Verification of the Stellar Consensus Protocol. In *2nd Workshop on Formal Methods for Blockchains (FMBC 2020) (OpenAccess Series in Informatics (OASIcs), Vol. 84)*, Bruno Bernardo and Diego Marmsoler (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:9. https://doi.org/10.4230/OASIcs.FMBC.2020.9

Dahlia Malkhi and Pawel Szalachowski. 2023. Maximal Extractable Value (MEV) Protection on a DAG. In *4th International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2022) (Open Access Series in Informatics (OASIcs), Vol. 110)*, Yackolley Amoussou-Guenou, Aggelos Kiayias, and Marianne Verdier (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:17. https://doi.org/10.4230/OASIcs.Tokenomics.2022.6

Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf

Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. 2021. Cogsworth: Byzantine View Synchronization. *Cryptoeconomic Systems* 1, 2 (oct 22 2021). https://doi.org/10.21428/58320208.08912a03

Ray Neiheiser, Miguel Matos, and Luís Rodrigues. 2021. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 35–48. https://doi.org/10.1145/3477132.3483584

Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2017. Reducing Liveness to Safety in First-Order Logic. *Proc. ACM Program. Lang.* 2, POPL, Article 26 (dec 2017), 33 pages. https://doi.org/10.1145/3158114

Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, Wolf Honoré, and Zhong Shao. 2024a. *Artifact for PLDI 2024 paper # 290: LiDO: Linearizable Byzantine Distributed Objects with Refinement-Based Liveness Proofs.* Yale University, New Haven, USA. https://doi.org/10.5281/zenodo.10909272

Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, Wolf Honoré, and Zhong Shao. 2024b. LiDO: Linearizable Byzantine Distributed Objects with Refinement-Based Liveness Proofs. *Proc. ACM Program. Lang.* 8, PLDI, Article 193 (June 2024), 25 pages. https://doi.org/10.1145/3656423

Longfei Qiu, Jingqi Xiao, Ji Yong Shin, and Zhong Shao. 2025a. *Artifact for PLDI 2025 Paper #316 LiDO-DAG: A Framework for Verifying Safety and Liveness of DAG-Based Consensus Protocols.* https://doi.org/10.5281/zenodo.15223659

Longfei Qiu, Jingqi Xiao, Ji-Yong Shin, and Zhong Shao. 2025b. *LiDO-DAG: A Framework for Verifying Safety and Liveness of DAG-Based Consensus Protocols.* Technical Report TR1574. Yale Univ. https://flint.cs.yale.edu/publications/lido-dag.html

Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 619–650. https://doi.org/10.1007/978-3-319-89884-1_22

Mayank Raikwar, Nikita Polyanskii, and Sebastian Müller. 2024. SoK: DAG-based Consensus Protocols. In *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–18. https://doi.org/10.1109/icbc59979.2024.10634358

Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. 2025. Sailfish: Towards Improving the Latency of DAG-based BFT . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 21–21.

Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. 2023. Shoal: Improving DAG-BFT Latency And Robustness. arXiv:2306.03058 [cs.DC] https://arxiv.org/abs/2306.03058

Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022a. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2705–2718. https://doi.org/10.1145/3548606.3559361

Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022b. Bullshark: The Partially Synchronous Version. arXiv:2209.05633 [cs.DC] https://arxiv.org/abs/2209.05633

Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. 2024. Anvil: verifying liveness of cluster management controllers. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 35, 18 pages.

Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for Decidability of Deductive Verification with Applications to Distributed Systems. *SIGPLAN Not.* 53, 4 (jun 2018), 662–677. https://doi.org/10.1145/3296979.3192414

The Coq Development Team. 2024. The Coq Reference Manual – Release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman.

Søren Eller Thomsen and Bas Spitters. 2021. Formalizing Nakamoto-Style Proof of Stake. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–15. https://doi.org/10.1109/CSF51468.2021.00042

Visa Inc. 2023. Annual Report 2023. https://s29.q4cdn.com/385744025/files/doc_downloads/2023/Visa-Inc-Fiscal-2023-Annual-Report.pdf

Ivana Vukotic, Vincent Rahli, and Paulo Esteves-Veríssimo. 2019. Asphalion: trustworthy shielding against Byzantine faults. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 138 (Oct. 2019), 32 pages. https://doi.org/10.1145/3360564

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. *SIGPLAN Not.* 50, 6 (jun 2015), 357–368. https://doi.org/10.1145/2813885.2737958

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) *(CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 154–165. https://doi.org/10.1145/2854065.2854081

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* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591

Qiyuan Zhao, George Pîrlea, Karolina Grzeszkiewicz, Seth Gilbert, and Ilya Sergey. 2024. Compositional Verification of Composite Byzantine Protocols. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 34–48. https://doi.org/10.1145/3658644.3690355

Received 2024-11-15; accepted 2025-03-06