Mechanized Safety and Liveness Proofs for the Mysticeti Consensus Protocol under the LiDO-DAG Framework

Longfei Qiu
longfei.qiu@yale.edu
Yale University

Jingqi Xiao
xjq_xjq@sjtu.edu.cn
Shanghai Jiao Tong University

Zhong Shao
zhong.shao@yale.edu
Yale University

***Abstract*—Directed acyclic graphs (DAG) have recently become a popular building block for high-throughput consensus protocols used in blockchains. Mysticeti is a state-of-the-art DAG-based consensus protocol that is currently deployed in the Sui blockchain and the IOTA blockchain. Compared to previous protocols, Mysticeti achieves lower commit latency by eliminating reliable broadcast and increasing leader vertex frequency. However, this comes at the cost of significantly more complex security proofs than previous protocols. In fact, shortly after Mysticeti was published, flaws were found in its liveness proof, leaving the correctness of the protocol uncertain.**

In this work, we resolve the controversy around correctness of Mysticeti by presenting the first complete analysis of the safety and liveness properties of Mysticeti. Our key finding is that, unlike previous DAG-based protocols like Narwhal and Bullshark, liveness of Mysticeti is highly sensitive to the round-jumping behavior of honest participants. If honest processes are allowed to jump over rounds arbitrarily, then we present an explicit counterexample to the liveness of Mysticeti: an infinite trace where no data blocks are ever committed. We then introduce a simple restriction on the round-jumping behavior, and show that our modification is sufficient to restore liveness of Mysticeti. We mechanized proofs of safety and liveness of Mysticeti under the LiDO-DAG framework, an abstract model of DAG-based consensus protocols proposed by Qiu et al., confirming that our modified protocol is fully correct. We also audited the current implementation of Mysticeti in the Sui blockchain and found it is susceptible to the described liveness bug. We have contacted Mysten Labs and are working with them to fix the liveness issues.

1. Introduction

Blockchains are public, distributed ledgers that enable secure financial transactions without the need of a centralized bank. Early blockchains like Bitcoin [1] are based on *proof-of-work*, where each participant must solve a cryptographic puzzle locally before proposing a block. This approach has the drawback of being energy-intensive [2], and its security properties are difficult to analyze [3]. Instead, many newer blockchains such as Ethereum 2.0 [4] and Aptos [5] are based on *proof-of-stake*, where the validators participate in a byzantine fault-tolerant (BFT) consensus protocol to propose and agree on a sequence of transactions. As public blockchains need to support both a large number of validators and a large volume of transactions, there has been a long line of research over improving the latency, throughput, and scalability of BFT protocols [6], [7], [8], [9], [10], [11].

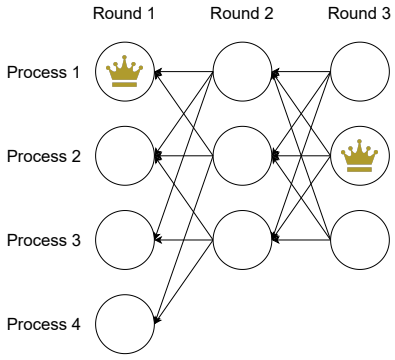

Figure 1. Example of DAG-based mempool. Each vertex in the figure is a block of transactions. Each edge is a reference embedded in the block. Some vertices are considered leader vertices and typically marked with a crown. The location and frequency of leader vertices are protocol-specific.

DAG-Based Consensus. Many classic BFT protocols like PBFT [6] and HotStuff [8], are *leader-based*. Their execution is structured as an infinite sequence of *views*, each view having a single predesignated *leader* who is responsible for proposing and commiting data blocks. The other participants merely replicate the blocks and confirm the leader is not equivocating. Although this architecture is well-understood and easy to implement, it has the obvious drawback of placing unbalanced burden on the resources of the leader, an issue known as the *leader bottleneck* [12], [13].

To overcome the leader bottleneck, the latest trend in BFT protocol design is to combine consensus with a *mempool* component [12], [14], [15], [16], [17], [18], [19]. All participants can add blocks to the mempool, even when they are not the consensus leader. Each view of consensus selects a list of blocks in the mempool to commit. This allows a large number of blocks from different participants to be committed in each view of consensus, improving the throughput of the protocol. The mempool is typically structured as a directed acyclic graph (DAG) (Fig. 1). Each vertex of the graph represents a block of transactions. The edges correspond to references to previous blocks that are

embedded in each block. As such these protocols are called *DAG-based consensus*.

How DAG-Based Consensus Works. At a high level, how DAG-based protocols work can be described as follows. The mempool DAG graph is stratified into layers called *rounds*. We assume that rounds are numbered from 1. In each round, each participant is supposed to create at most one vertex. Also, each vertex in round $r > 1$ should contain references to $n - f$ vertices from different participants in round $r - 1$, where n is the total number of participants and f is the byzantine fault threshold. These rules ensure *chain quality*: at least half of the committed blocks are from honest parties, and byzantine parties cannot flood the network with blocks.

A combination of a round number and a participant ID is called a *slot*. Thus each vertex belongs to a single slot. Honest participants never create two vertices in a single slot, but byzantine participants may attempt to do so. Each protocol designates a subset of slots as *leader vertex slots*. Vertices created in these slots are called *leader vertices*.

Each participant of consensus maintains a local consensus log, which is a linear list of vertices in the mempool. The basic goals are safety and liveness, which we define as:

- **Safety**: the consensus logs of honest parties are consistent with each other;
- **Liveness**: All vertices created by honest parties get included in the consensus log within bounded time.

To obtain the consensus log, each participant runs an *ordering algorithm* on its locally-observed DAG, which returns a *leader vertex consensus log*. Each entry v in the leader vertex consensus log is expanded into the list of all vertices that are directly or indirectly referenced by that leader vertex. As long as the leader vertex consensus log seen by each participant is consistent with each other, the full consensus logs remain consistent. Thus leader vertices play a key role in the safety and liveness of DAG-based protocols, and the latency of consensus is mainly determined by:

- Leader vertex frequency: how often leader vertex slots appear in the mempool DAG;
- Leader vertex commit latency: how long it takes to commit a leader vertex after creation.

Authenticated and Unauthenticated DAG. As explained above, chain quality of DAG-based protocols relies on each participant creating no more than one vertex in each round. There are two possible ways to enforce this rule. The first way is to require each party to distribute their vertices through a *reliable broadcast* (RBC) protocol [20], [21], [22]. RBC ensures that even if a byzantine party attempts to create multiple vertices in a single slot, only one of the vertices will ever be delivered to any honest party. Protocols based on RBC are called *authenticated* or *certified* DAG.

While RBC proactively prevents equivocation, it also takes up a significant portion of the commit latency, as the leader vertex cannot be committed before it is received by the peers through RBC. This has led to works exploring DAG-based protocols without RBC [17], [18]. They rely on honest parties detecting equivocations from byzantine parties, and punishing the violators by mechanisms like stake-slashing. Protocols not using RBC are called *unauthenticated* DAG.

Mysticeti. Within this context, Babel et al. [18] recently proposed a new DAG-based protocol called **Mysticeti**. Compared to previous protocols like Narwhal [12] and Bullshark [15], Mysticeti achieves lower commit latency by incorporating several optimizations. First, inspired by Cordial Miners [17], Mysticeti uses an unauthenticated DAG rather than an authenticated DAG. Second, inspired by the pipelining optimization in HotStuff [8] and Jolteon [9], Mysticeti supports having a leader vertex in every single round, rather than having a leader vertex in every second round (Bullshark [15]) or third round (Cordial Miners [17]). In July 2024, Mysticeti was adopted by the Sui blockchain, and the developers reported a 75% drop in actual commit latency [18]. In May 2025, the IOTA blockchain also rebased itself to Mysticeti [23].

Correctness of Mysticeti. While Mysticeti seems to be performing well so far, its security under byzantine attacks is currently disputed. Shortly after Mysticeti was published, [16, Appendix D] pointed out flaws in the liveness proof of Mysticeti. It claims that due to these errors, the honest parties may not always commit leader vertices within bounded time. However, it does not provide an explicit counterexample for liveness of Mysticeti. As such there are several possibilities regarding correctness of Mysticeti:

- The protocol is correct, and the flaws are merely gaps in the proofs that can be patched;
- The protocol satisfies liveness but does not achieve its claimed commit latency in the worst case;
- The protocol does not satisfy liveness: it does not commit blocks within bounded time in the worst case.

Currently, it is unclear which of the above situations is the actual case. If the protocol is incorrect, it is also unclear how to restore its liveness and latency properties.

Our Analysis. In this work, we critically analyze the flaws in the liveness proof of Mysticeti. We show that the original presentation of Mysticeti [18] underspecified the round-jumping behavior of honest parties, which is the action that a parties takes to catchup when it finds itself lagging too far behind the global network progress. While previous DAG-based protocols like Narwhal [12] and Bullshark [15] are mostly insensitive to the details of round-jumping, liveness of Mysticeti requires specific actions to be taken by honest parties when they jump over a round.

The interpretation taken in Shrestha et al. [16] is that round-jumping should be implemented in the same way as Bullshark [15]: honest parties do not need to take specific actions upon jumping. Under the same assumption, we construct an explicit counterexample for liveness of Mysticeti: an infinite trace in which no leader vertices are ever committed. Thus there is a real liveness bug in Mysticeti, and to patch the bug one must include restrictions on round-jumping.

We then introduce a simple rule regarding round-jumping. We show that, if all honest parties follow our specified actions when jumping over rounds, then we can recover the liveness and latency properties of Mysticeti claimed in

Babel et al. [18]. Our modified protocol remains compatible with the original Mysticeti (i.e. implementations which do not follow our rule). Moreover, we do not require the honest parties to follow our rule from the beginning of execution. We define a timepoint called the *global catchup time* (GCT), which is analogous to the *global synchronization time* (GST) used in the partial synchrony assumption [24]. We assume that honest parties do not need to follow our rule before GCT, but must follow it after GCT. We provide full liveness guarantees after $\max\{\mathsf{GST}, \mathsf{GCT}\}$. As such we expect existing implementations of Mysticeti can be easily upgraded to incorporate our proposed fix.

Formal Verification of Mysticeti. Although we claim that our modified protocol avoids the flaws in the original Mysticeti protocol, there are still chances that we introduce new unintentional bugs in our specification. The only way to fully ascertain correctness of the protocol is to formally model it in a specification language, and construct machine-checkable proofs of its security properties.

Verifying safety and liveness of consensus protocols has traditionally been a daunting task. For example, the Verdi project [25], [26] verified only safety properties of Raft, a consensus protocol that only tolerates benign faults, which took more than 50,000 lines of formal proof.

Recently, Qiu et al. [27] introduced a refinement-based formal framework called **LiDO** for verifying both safety and liveness of leader-based consensus, which has been successfully applied to Jolteon [9], a pipelined leader-based consensus protocol. In Qiu et al. [28], the framework was extended to **LiDO-DAG** which was used to verify several authenticated DAG-based protocols, but not unauthenticated protocols like Mysticeti. LiDO and LiDO-DAG are formalized in the Rocq proof assistant (formerly called Coq). The key innovations of LiDO that simply proof engineering of consensus protocols are 1) the LiDO cache tree, an abstract representation of the global consensus log; 2) the abstract pacemaker, which allows decomposing liveness of consensus into safety properties that are easy to prove by refinement.

In this work, we construct a formal model of the modified Mysticeti protocol in Rocq and prove its safety and liveness via refinement to LiDO-DAG. This eliminates all ambiguities in the specification of Mysticeti and confirms the modified protocol is fully correct. Our proofs are available as an artifact at [29].

Real-world Implementations of Mysticeti. Since Mysticeti has been adopted by multiple real-world blockchains, one naturally asks whether the liveness bug described here also affects existing implementations. To this end, we audited the source code of the Sui blockchain which uses Mysticeti at its core. We found that current versions of Sui indeed implemented round-jumping incorrectly, making them susceptible to liveness attacks. We have contacted Mysten Labs and they have acknowledged the issue. We are currently working with them to resolve the liveness issues.

The implementation used by the IOTA blockchain is a fork of Sui. We also discussed the issue with its developers. The IOTA developers are working on a different fix [30], which we discuss in Sec. 7.

Summary of Our Contributions. In this work, we 1) construct an explicit counterexample to liveness of the Mysticeti protocol; 2) describe a modification to Mysticeti that fixes the liveness issue and maintains compatibility with existing implementations; 3) provide a formal model of Mysticeti with our modification and machine-checked proofs its safety and liveness properties; 4) identify vulnerabilities in deployed implementations of Mysticeti that make them susceptible to the liveness bug we found.

2. Background: The Mysticeti Protocol and its Liveness Flaw

In this section, we first provide an introduction to the Mysticeti protocol. We then focus on the round-jumping behavior of honest participants in Mysticeti, which lies at the heart of the liveness dispute. We take a close look at the argument that the round-jumping behavior in current implementations is flawed. The argument implies there are errors in the current liveness proof of Mysticeti, but it is unclear whether it implies a genuine liveness bug in Mysticeti, or merely a gap in the proof that can be patched.

Notations. We use $[x_1; \cdots; x_n]$ to represent a finite list whose entries are $x_1, \cdots, x_n$. If L_1, L_2 are two finite lists, we use $L_1 \mathbin{+\!\!+} L_2$ to denote their concatenation. If L is a finite list of items with decidable equality (e.g. integers), we use $|L|$ to denote the number of non-duplicate items in L.

System Model. We consider a network system consisting of $3f + 1$ processes, of which $2f + 1$ are honest and f are byzantine. This is a standard setting in BFT protocols, since [31] has shown that it is not possible to tolerate byzantine faults in more than $1/3$ of the processes without sacrificing safety or liveness. Within this setting, a set S of at least $2f + 1$ processes is called a *quorum*. By the pigeonhole principle, two quorum sets must intersect on at least $f + 1$ processes. Since there are only f byzantine processes, they intersect on at least one honest process. This is a key property exploited by most BFT protocols, including Mysticeti.

We assume each honest process is equipped with a local timer that can be reset to a fixed duration. Each honest process is an atomic state machine that receives three kinds of signals: 1) a client submits a transaction to the process; 2) a network message is delivered to the process; 3) the local timer expires. Upon each event the process faithfully executes the corresponding event handler. For each process only one event handler may execute at a time. The byzantine processes have no internal state and can inject arbitrary messages into the network, subject to conditions that will be explained later.

Communication between the processes follows the partial synchrony model [24]. There is a timepoint called the *global synchronization time* (GST) that is unknown to the honest parties. Before GST, message delivery can be arbitrarily delayed, but each message must be delivered within a known bound Δ after GST, provided both the sender and

Algorithm 1 Building the Global DAG

```
1:  State variables: verts : FinMap (ID ↦ Vertex),
2:  State variables: usedID : list ℕ, invalidID : list ℕ.
3:  initialize:
4:      Assume usedID = [ ], invalidID = [ ].
5:      Assume ∀ id, verts[id] = ⊥.
6:  upon process p_i creates record (p_i, r, d, preds):
7:      Choose id s.t. id ∉ usedID ∧ id ∉ invalidID.
8:      Add id to usedID list.
9:      if ∃ p ∈ preds, verts[p] = ⊥ then
10:         Add every p ∈ preds s.t. verts[p] = ⊥ to invalidID list.
11:         Discard record and return.
12:     if ∃ p ∈ preds, verts[p].round ≥ r then
13:         Discard record and return.
14:     if r = 1 ∧ preds = [ ] then
15:         verts[id] ← (p_i, r, d, preds) and return.
16:     strongEdges ← filter (p ↦ verts[p].round = r − 1) preds.
17:     if |map (p ↦ verts[p].builder) strongEdges| ≥ 2f + 1 then
18:         verts[id] ← (p_i, r, d, preds) and return.
19:     else
20:         Discard record and return.
```

the recipient are honest. Formally, if a message is sent at timepoint t, then we assume it is delivered at least once before $\max\{t, \mathsf{GST}\} + \Delta$.

Also, before GST we do not assume the local timers of honest processes provide reliable timing, but they must timeout reliably after GST. This assumption is more difficult to encode formally. In our formal proofs, we follow the technique suggested in [27] and encode it using segmented traces. See Sec. 5 for details.

Global and Local DAGs. The only kind of message that is exchanged between the processes is the *vertex record*, which we model as follows:

$$Vertex \triangleq \mathbb{N}_{builder} * \mathbb{N}_{round} * Data * (\mathsf{list}\ \mathbb{N}_{preds}).$$

The $builder$ field records the ID of the process that created the record. We assume each vertex record is cryptographically signed by its builder, so that byzantine processes can only put their own IDs into this field. The $round$ field records the round this vertex record belongs to, and we assume $round \geq 1$. The $data$ field is used to embed transaction requests, but we do not model its content. Finally, each vertex record contains a list of *predecessor references*, which correspond to edges in the DAG. Each reference is represented by the ID of the target record, which is uniquely assigned upon creation of the record and explained later.

At any given moment, the set V_G of all vertex records ever sent into the network forms a *global* DAG, which is formally defined in the next paragraph. However, the set V_i of vertex records seen by any particular process p_i is a subset of V_G. Thus, each process needs to reconstruct a *local* DAG from the vertex records it has received. For most DAG-based protocols, the local DAG is always a subgraph of the global DAG, and this is also true for Mysticeti (see Proposition 1). The idea of DAG-based consensus is to define a global consensus log on the global DAG. Each process runs an ordering algorithm on its local DAG to obtain a local consensus log, which in general must be a prefix of the global consensus log.

We now look at how the global DAG is constructed from the vertex records. We assume there is a virtual system agent $\mathcal{S}$ that can monitor all vertex record creations in the system. $\mathcal{S}$ maintains a map from record IDs to vertex records, a list of *used* IDs, and a list of *invalid* IDs.

When process p_i creates a record $(p_i, r, d, preds)$, $\mathcal{S}$ follows Alg. 1 to update the global DAG. It first assigns an ID to this record. In actual implementations, the ID is computed with a cryptographic hash function. If we model this hash function as a random oracle [32], then the ID is uniformly drawn from the set of all possible IDs, and the probability of choosing any used or invalid ID is negligible. We model this step as non-deterministically choosing an ID that is not in the used or invalid ID list.

After adding the chosen ID to the list of used IDs, $\mathcal{S}$ performs several semantic checks on the record. If $r = 1$ then the record is a *genesis* vertex, and the list of predecessors must be empty. If $r > 1$ then $\mathcal{S}$ looks at the IDs in the predecessor list. If any $id \in preds$ does not refer to an existing vertex in the global DAG, then such IDs are added to the list of invalid IDs, and the whole record is discarded. Otherwise, $\mathcal{S}$ checks that each $id \in preds$ refers to a vertex in some round $r' < r$. Also, $preds$ must contain references to $2f + 1$ vertices in round $r - 1$ from different builders. If all these semantic conditions are satisfied, the record is considered valid and added to the global DAG.

When any honest process p creates or receives a vertex record, it executes a procedure similar to Alg. 1 to update its local DAG. There only differences are:

- At lines 7–8, instead of assigning a new ID, p calls the random oracle to retrieve the ID that $\mathcal{S}$ has assigned;
- At lines 9–11, if $preds$ contains any id that p has not seen, then it cannot tell whether id is invalid, or refers to a record it has not yet received. In this case, instead of discarding the record, p stores it in a "staging area" and revisits it when all IDs in $preds$ have been resolved.

If a byzantine process creates a record that is discarded by $\mathcal{S}$, then we can easily see it cannot be added to the local DAG of any honest process either. In the worst case the record stays permanently in the staging area of honest processes. Therefore, Mysticeti maintains the following property:

Proposition 1. *The local DAG of any honest process p is a subgraph of the global DAG constructed by $\mathcal{S}$.* □

This property is currently implicitly assumed in our formal model as it does not yet capture the staging area of each process. We are developing more realistic refinement layers that model these details, but as we explain in Sec. 5, the current high-level model is sufficient to capture the safety and liveness proofs.

The Ordering Algorithm. As explained in the introduction, DAG-based consensus centers around the leader vertex ordering algorithm, which interprets the local DAG and returns a list of committed leader vertices. The ordering

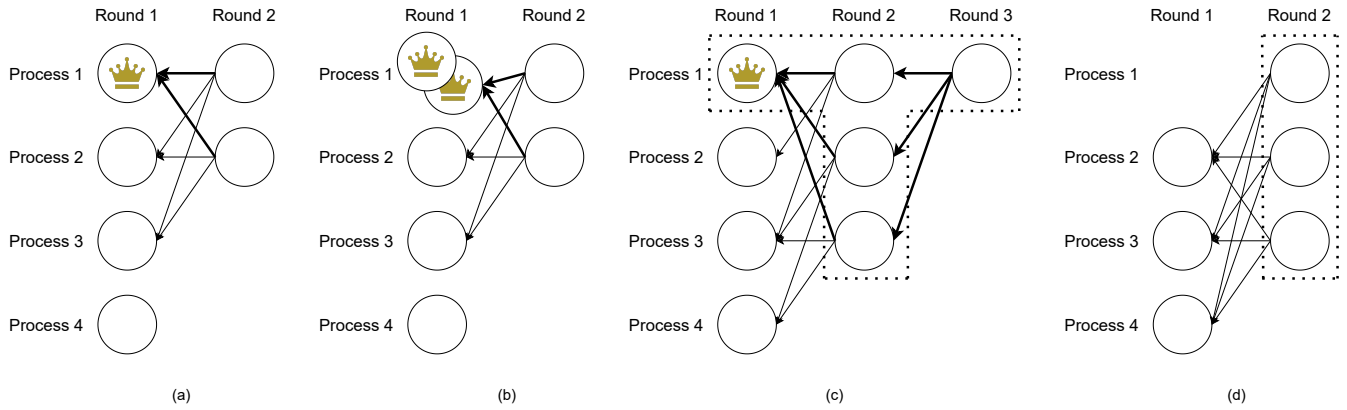

Figure 2. Supporter, certificate, and skip patterns in Mysticeti. Reproduced from similar figures in [18]. (a) The two vertices in round 2 are supporters of the leader vertex in round 1. The bold arrows are the references to the leader vertex. (b) Even if the leader in round 1 equivocates, each vertex in round 2 can still support only one leader vertex in round 1. This is guaranteed by the definition of the supporter relation. (c) The vertices shown in the dashed box form a certificate pattern for the leader vertex in round 1. The bold arrows are the references to the leader vertex, and references to the supporters. (d) The vertices shown in the dashed box form a skip pattern for round 1. In this case no certificate for any leader vertex of round 1 may ever be created.

algorithm of Mysticeti is shown in Alg. 2, and we now explain how it works.

We assume every round r has a predefined leader $leader\_at(r)$, which is one of the $3f+1$ processes. The leader vertex slots are $(r, leader\_at(r))$ with $r \geq 1$. Since byzantine leaders may equivocate, there may be more than one leader vertex in a round. As such, the process of adding a leader vertex into the consensus log requires two steps:

1) From the possibly multiple vertices created by the leader, the consensus participants choose a single vertex to be potentially committed.
2) The consensus participants decide on whether to commit the leader vertex chosen in the first step.

Many leader-based BFT protocols such as PBFT [6] and Jolteon [9] rely on *voting* to complete each step. For example, in the first step each process sends out a vote for one of the leader vertices it has observed. Of course, byzantine processes may equivocate and send out multiple votes, but each honest processes can only make one vote. A leader vertex is chosen when it gets votes from $2f+1$ processes, which is a quorum set. By the quorum overlapping property, no two leader vertices in a single round can both get $2f+1$ votes. This guarantees only a single leader vertex can be chosen in the first step. In the second step, the leader vertex also needs $2f+1$ votes to become committed.

In DAG-based protocols like Mysticeti there are no explicit vote messages. Instead, voting information is implicitly encoded into the topological structure of the global DAG. In Mysticeti they are embodied as *supporter*, *certificate*, and *skip* patterns, which we define as follows:

Definition 1 (Supporters)**.** *Within the DAG, if v is a leader vertex of round r, v' is a vertex in round $r+1$, then v' is a* ***supporter*** *of v if v is the first vertex in $v'.preds$ that is in the leader vertex slot of round r. We also say v' supports v.*

Definition 2 (Certificates)**.** *Within the DAG, if v is a leader vertex of round r, v' is in round $r+2$, then v' is a* ***certificate*** *of v if $v'.preds$ contains $2f+1$ vertices in round $r+1$ from different processes that are supporters of v. We also say v' certifies v.*

Algorithm 2 The Ordering Algorithm

1: State variable: $decisions$: FinMap $(\mathbb{N} \mapsto Decision)$.
2: **initialize**: Assume $\forall\, r, decisions[r] = Undecided$.
3: **upon** observing $2f+1$ certificates for some leader vertex v:
4: $\quad decisions[v.round] \leftarrow Committed(v)$.
5: **upon** observing a skip pattern for round r:
6: $\quad decisions[r] \leftarrow Skipped$.
7: **upon** observing two rounds r, r', such that $r' \geq r+3$, $decisions[r] = Undecided$, $decisions[r'] = Committed(v)$, and $\forall\, r'', r+3 \leq r'' < r' \Rightarrow decisions[r] = Skipped$:
8: $\quad$ **if** v directly or indirectly references a certificate for some leader vertex v' of round r **then**
9: $\quad\quad decisions[r] \leftarrow Committed(v')$
10: $\quad$ **else**
11: $\quad\quad decisions[r] \leftarrow Skipped$

Definition 3 (Skip patterns)**.** *Within the DAG, if S is a set of at least $2f+1$ vertices from different processes in round $r+1$, then S is a* ***skip pattern*** *for round r if no vertex in S references any leader vertex of round r.*

Fig. 2 shows some examples of these patterns. Each supporter can be understood as a vote in the first step of committing a leader vertex, in the two-step process described earlier. Notice that Definition 1 guarantees each vertex can only support a single leader vertex. Since each honest process creates at most one vertex in each round, it can also only cast a single vote in the first step.

Similarly, each certificate can be seen as a vote in the second step. Recall that to proceed to the second step, a leader vertex must get at least $2f+1$ votes in the first step. This corresponds to the requirement in Definition 2 that a certificate for a leader vertex must reference at least $2f+1$ supporters of it. Thus all certificates in a single round must be for a single leader vertex.

A skip pattern is the opposite of a certificate. If we

observe a $2f+1$ vertices in round $r+1$ not supporting any leader vertex in round r, then no certificate for any leader vertex in round r may ever be created. In this case we can skip waiting for certificates of round r. This is an optimization intended for reducing failover latency when a leader has experienced crash fault. However, not implementing this feature does not affect overall safety and liveness of the protocol.

Given a valid DAG G, the ordering algorithm assigns to each round r a *decision*, whose possible values are:

- $Committed(v)$ where v is a leader vertex of round r, meaning v has been committed;
- $Skipped$, meaning no leader vertex of this round will ever be committed;
- $Undecided$, meaning a decision for this round cannot yet be made.

Initially, all rounds are *undecided*. If an honest process p_i assigns the decision $Committed(v)$ to round r, we say p_i has *committed* vertex v. We may also say it has committed round r. If it assigns the decision $Skipped$ to round r, we say p_i has *skipped* round r. In either case, we say it has *decided* round r.

When $2f+1$ certificate patterns (from different processes) are observed for a leader vertex v, the status of $v.round$ is set to $Committed(v)$. When a skip pattern is observed for round r, the status of round r is set to $Skipped$. In [18], these two rules are known as the *direct* decision rules.

In case of network failures or byzantine leaders, direct decision rules are not sufficient to decide all rounds. Mysticeti additionally uses an *indirect* decision rule: to decide round r, we look for the earliest round $r' \geq r+3$ whose current status is not $Skipped$. If round r' is still undecided, then round r remains undecided. Otherwise, round r' has a committed leader vertex v. If v directly or indirectly references any certificate for a leader vertex v' of round r, then the status of round r is set to $Committed(v')$; if v does not reference any such certificate, then round r is skipped.

Important Note: In this paper, "skipping" a round and "jumping over" a round (introduced later) are two different notions. To skip round r is the decision not to commit any leader vertex from round r. To jump over round r is to not create any vertex in round r. The terminology of "jumping" comes from [16].

Theorem 1 (Safety of Mysticeti)**.** *If two honest processes both decided round r, their decisions must be the same.*

Safety of Mysticeti is mostly uncontroversial. We give an outline of the proof of Theorem 1 in Sec. 5. This theorem corresponds to the Rocq lemmas `mcommit_safety` and `mcommit_commit_eq` in the artifact.

Once all rounds $r \leq R$ have been decided, the leader vertex consensus log up to round R can be found by taking the list of all committed leader vertices in rounds $r \leq R$ and ordering them by round number. To compute the full consensus log, we define the *closure* of each vertex v as follows:

$$\mathrm{cl}\,(v) = \mathrm{cl}\,(pred_1) \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \mathrm{cl}\,(pred_n) \mathbin{+\!\!+} [v]$$

Algorithm 3 Vertex Creation Rules in Original Mysticeti

State variables: $curr\_round : \mathbb{N}$ (short for $current\_round$).
initialize:
    Assume timer is disabled.
    Create vertex in round 1.
    $curr\_round \leftarrow 1$.
upon observing $2f+1$ vertices in round $curr\_round$, and the process has already created a vertex in $curr\_round$:
    Enable and reset local timer to 2Δ.
    $curr\_round \leftarrow curr\_round + 1$.
upon observing both a leader vertex of round $curr\_round - 1$ and $2f+1$ supporters of some leader vertex of round $curr\_round - 2$:
    Disable local timer.
    Create vertex in round $curr\_round$.
upon local timer expires:
    Disable local timer.
    Create vertex in round $curr\_round$.
upon observing $2f+1$ vertices in some round $r > curr\_round$:
    Enable and reset local timer to 2Δ.
    Create vertex in round r.
    $curr\_round \leftarrow r + 1$.
Predecessor Rule: When creating a vertex in round r, if the process has observed at least one leader vertex of round $r-1$, the new vertex should be a supporter for some leader vertex of round $r-1$; if the process has observed $2f+1$ supporters for a single leader vertex in round $r-2$, the new vertex should be a certificate for that leader vertex.

where $pred_1, \cdots, pred_n$ is the list of predecessors of v. Since each predecessor of v must have a smaller round number than v, the closure is always well-defined. Vertices in round 1 have no predecessor, so their closure is a list containing only themselves.

If the leader vertex consensus log is $[v_1; \cdots; v_n]$, then the full consensus log is $\mathrm{dedup}\,(\mathrm{cl}\,(v_1) \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \mathrm{cl}\,(v_n))$ where dedup is list deduplication. Note that to compute the consensus log up to round R, we need decisions for all rounds before R, not just the decision for round R.

Theorem 2. *If an honest process has committed all three rounds $R, R+1, R+2$, then it can make a decision for all rounds $r \leq R+2$.*

Proof: Since the rounds $R, R+1, R+2$ are committed, the indirect decision rule immediately provides a decision for the rounds $R-3, R-2, R-1$. If all rounds r with $R-k < r \leq R+2$ have been decided, then the round $R-k$ can also be indirectly decided through the earliest committed leader vertex after round $R-k$. Such a leader vertex always exists since round R is committed. □

This theorem corresponds to the Rocq lemma `mcommit_can_decide` in the artifact.

Vertex Creation Rules. To finish defining the Mysticeti protocol we have to specify when and how honest processes create new vertices. Unfortunately this is where the original presentation [18] gets vague and the liveness bug creeps in.

The creation rules in [18] are summarized in Alg. 3. At the beginning of execution, each honest process should create a genesis vertex in round 1. After creating a vertex in round r, the minimal condition for creating a vertex in round $r+1$ is to receive $2f$ other vertices in round r, apart

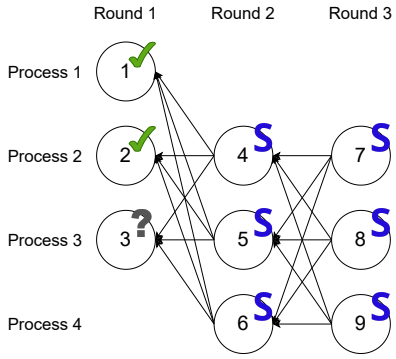

Figure 3. Example of round-jumping behavior. Vertices 1 and 2 are in the local DAG of process 1 and marked with green check. Vertices 4,5,6,7,8,9 are currently in the staging area of process 1 and marked with S. Vertex 3 has not yet been delivered to process 1. At this point, if we deliver vertex 3 to process 1, then process 1 should jump to round 3.

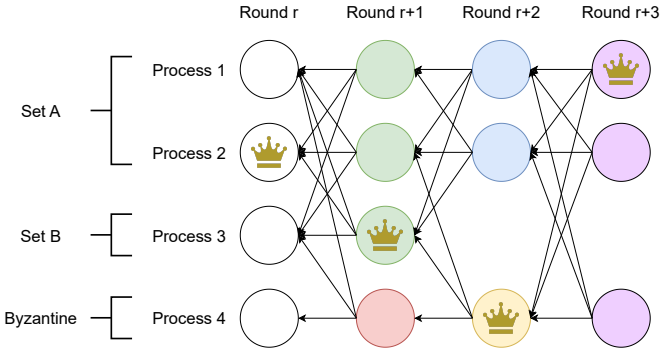

Figure 4. A counterexample to a key lemma in the liveness proof of Mysticeti. Refer to the text below on the steps that lead to the shown scenario. Assume that the leader of round r is process 2, and subsequent leaders are assigned with round-robin. Green: vertices from honest processes in step 2, which support L_r. Red: vertices from the byzantine process in step 2, which do not support L_r. Blue: vertices from processes in A in step 4, which certify L_r and support L_{r+1}. Yellow: vertices from the byzantine process in step 5, which support L_{r+1} but do not certify L_r. Purple: vertices created in steps 6 and 7, which certify L_{r+1}. At this point, if we deliver the purple vertices to processes in B, then they jump over round $r+2$, and L_r will never get $2f+1$ certificates.

from the one created by itself. This is because any vertex the process creates in round $r+1$ must reference at least $2f+1$ vertices in round r.

However, this does not mean honest processes should create a vertex in round $r+1$ immediately upon observing $2f+1$ vertices in round r. Since a major goal of the protocol is to commit leader vertices, honest processes should try their best to create supporters and certificates. To achieve this, each honest process resets its local timer to 2Δ upon observing $2f+1$ vertices in round r. It creates a vertex in round $r+1$ when it receives both the leader vertex of round r and $2f+1$ supporters for a leader vertex in round $r-1$, or when the timer expires, whichever comes first.

When creating the vertex, the process is required to try its best to make the new vertex a supporter and/or a certificate, as formalized in line 19 of Alg. 3. Thus, for example, if an honest process executes line 11 to create a vertex, then the new vertex should be both a supporter and a certificate.

By case analysis on the vertex creation rules in Alg. 3, we also see that Mysticeti maintains the following invariant:

Proposition 2. *For each honest process, if* $curr\_round > 1$*, then it has both 1) observed* $2f+1$ *vertices in round* $curr\_round - 1$*, and 2) created a vertex in round* $curr\_round - 1$*.* □

This property is currently encoded as an explicit rule on updating the $curr\_round$ variable in the formal model.

Round-jumping. We now consider a situation shown in Fig. 3. Suppose that process 1 has created a vertex in round 1, and is waiting for $2f$ other vertices in round 1. However, due to network failure, so far it has only received vertex 2. Meanwhile, the processes 2, 3, 4 have progressed to round 3. The six vertices 4,5,6,7,8,9 have been delivered to process 1. They are still in the staging area of process 1 because it has not yet received vertex 3. At this point, we deliver vertex 3 to process 1. Now process 1 suddenly observes $2f+1$ vertices in round 3, even though it has not yet created a vertex in round 2. How should process 1 behave in this case?

Since there are already $2f+1$ vertices in round 3, it is likely that some honest processes are waiting for supporters and certificates in round 3 (lines 7–10 of Alg. 3). If process 1 continues waiting for timer expiration, it may lag further behind global network progress. This will jeopardize liveness of the whole protocol, as other honest processes will not be able to collect supporter and certificate vertices built by process 1. Thus it is natural to require that process 1 immediately "jumps" to round 3 and creates a vertex there (lines 15–18 of Alg. 3). This behavior is called *round-jumping*. It is analogous to the *view synchronization* behavior in leader-based consensus protocols [8], [33].

But what about round 2? Should process 1 also create a vertex in round 2, or should it just ignore round 2 altogether? [18] provides no explicit answer to this question. In previous DAG-based protocols, the liveness proof is mostly insensitive to this particular detail. In Bullshark [15], for example, when an honest process jumps to a higher round, it does not matter if it creates vertices in the intermediate rounds. Therefore, the intuitive answer seems to be that honest processes in Mysticeti can jump over these intermediate rounds as well. This is the interpretation taken in [16] and Alg. 3. As we will see in Sec. 6, it is also the how current Mysticeti implementations behave.

The Liveness Flaw. Sadly, the round-jumping behavior specified on lines 15–18 of Alg. 3 is *wrong*. It was first noted in [16] that the current behavior leads to gaps in the liveness proof of Mysticeti.

One of the key lemmas (Lemma 10 in [18]) in the liveness proof of Mysticeti is that, after GST, when any honest process creates a leader vertex v, every honest process will eventually create a certificate for v. Hence every honest process will eventually observe $2f+1$ certificates for v and commit v directly. The counterargument in [16] is that under our current interpretation, honest processes do not always create certificates for leader vertices, so the key lemma is incorrect.

For a more explicit example, consider the following

sequence of events, shown in Fig. 4:

1) The leader of round r creates a leader vertex L_r;
2) All honest processes create vertices in round $r+1$ that support L_r, but byzantine processes create vertices that do not support L_r; the vertex created by the leader of round $r+1$ is denoted by L_{r+1};
3) We segregate the honest processes into two sets $A = \{p_1, \cdots, p_{f+1}\}$ and $B = \{p_{f+2}, \cdots, p_{2f+1}\}$;
4) We deliver the vertices in round $r+1$ to processes in A, and they create vertices in round $r+2$ that certify L_r and support L_{r+1};
5) The byzantine processes create vertices in round $r+2$ that support L_{r+1} but do not certify L_r;
6) We deliver the vertices in round $r+2$ to processes in A, and they create certificates in round $r+3$ for L_{r+1};
7) The byzantine processes also create vertices in round $r+3$, so that there are $2f+1$ vertices in round $r+3$;
8) Finally, we deliver the vertices in round $r+3$ to processes in B; these processes immediately jump to round $r+3$ without creating a vertex in round $r+2$, thus L_r will never get $2f+1$ certificates.

While the above argument correctly points out gaps in the liveness proof, it does not immediately imply Mysticeti is incorrect. For a complete counterexample we have to show that L_r will definitely never be committed. However, although L_r will never get $2f+1$ certificates, it at least gets $f+1$ certificates from *honest* processes. This is sufficient to show that every vertex in round $r' \geq r+3$ will indirectly reference at least one certificate of L_r. Perhaps this implies L_r can still be indirectly committed within bounded time?

For an analogy with leader-based consensus, we notice the liveness proof of Jolteon [9] also incorrectly states that after GST every honest process will eventually vote for an honest leader's proposal, for the same reason that the network adversary may manipulate a subset of the honest processes to jump over the view. This does not imply there are liveness bugs in Jolteon. In Jolteon, even if some honest processes jump over view v, it can be shown that any proposal from the leader of view $v+1$ must still reference the proposal from view v, which is sufficient to commit the proposal of view v. Can we fix the proof of Mysticeti in the same way?

In Sec. 3, we answer this question negatively by presenting a full counterexample of Mysticeti: an infinite trace where no leader vertex can be directly committed. Since indirect commit relies on direct commit, this means no leader vertex will ever be committed. Thus there is a genuine liveness bug in Mysticeti, and the round-jumping behavior (lines 15–18 of Alg. 3) must be modified. In Sec. 4, we present a simple modification to Mysticeti, which is sufficient to restore liveness of Mysticeti.

3. A Counterexample to Liveness of Mysticeti

In this section, we describe an explicit counterexample to liveness of Mysticeti. We present an infinite trace of Mysticeti having the property that in each round r, at most $2f$ certificates will ever be created. Since a direct commit requires $2f+1$ certificates, this is sufficient to ensure no leader vertex will be committed.

3.1. Some Early False Hopes on Salvaging the Liveness Proof

The definition of liveness is that every new leader vertex created by an honest process after GST will eventually be committed. The negation of this statement is that some leader vertex created by an honest process after GST will never be committed. As introduced in Sec. 2, in Mysticeti there are two different ways to commit a leader vertex, known as direct and indirect commit. In the scenario shown in Fig. 4, it is true that L_r will never be directly committed. However, it is still possible to commit L_r indirectly, since $f+1$ honest processes have created certificates for it. This is actually a general phenomenon. As we will see in Sec. 4, after GST each new leader vertex from an honest process will eventually get $f+1$ certificates that are also from honest processes, a result which we call "weak liveness."

To extend Fig. 4 into a full counterexample of Mysticeti, a simple idea is to repeat the steps that lead to Fig. 4, using round $r+3$ as the new round r. Notice that in Fig. 4, although L_r is not committed, L_{r+1} is committed since every vertex in round $r+3$ is a certificate for L_{r+1}. Therefore, if we repeat the process of building Fig. 4, then L_{r+4} would also be committed. Since L_{r+4} references at least one certificate of L_r, this is very close to committing L_r indirectly, the only obstacle being we cannot decide round $r+3$.

In an early stage of this investigation, the above observation has led to some suggestions that it is possible to salvage the original liveness proof, perhaps through some extensions to the ordering algorithm. After some deliberation, we found this is a dead end, as we came up with a trace in which no leader vertices are directly committed, which is what we describe below. This dispels all hopes of recovering liveness by extending the ordering algorithm, and so the round-jumping behavior must be modified.

3.2. The Counterexample Construction

Assume we have $2f+1$ honest processes and f byzantine processes. We require $f \geq 3$. The basic idea is to make each process behave as follows in each round $r \geq 3$:

- At least $f+2$ and at most $f+3$ honest processes create vertices in round r, and these vertices are both supporters for a leader vertex in round $r-1$ and certificates for a leader vertex in round $r-2$;
- The other honest processes jump over round r without creating any vertex;
- Each byzantine process creates two vertices v_1, v_2, such that v_1 is a supporter of a leader vertex in round $r-1$ but not a certificate, while v_2 is neither a supporter nor a certificate.

The rationale for the conditions listed above is as follows:

- First, to make an honest process create a vertex in round r, we have to let it observe a leader vertex of round

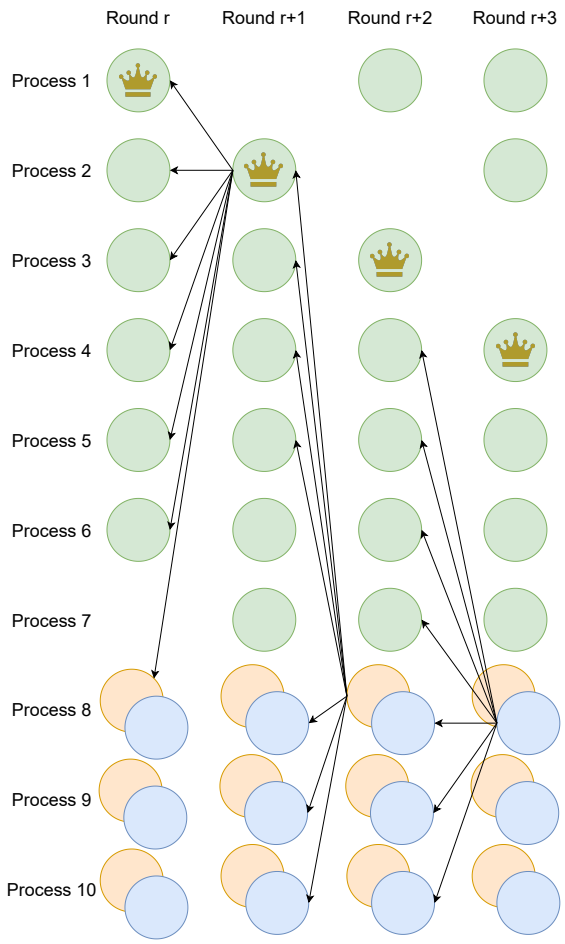

Figure 5. A counterexample to Mysticeti with $f = 3$ where no leader vertex gets directly committed. Refer to Sec. 3.2 for explanation. In each round, vertices of the same color have the same predecessor list. To reduce visual clutter, for each color we show the predecessor list of only one vertex. A gap indicates that the process has jumped over this round. Green: vertices created by honest processes, which are both supporters and certificates. Orange: the v_1 vertices created by byzantine processes, which are supporters but not certificates. Blue: the v_2 vertices created by byzantine processes, which are neither supporters nor certificates.

$r-1$, as well as $2f+1$ supporters for a leader vertex of round $r-2$. This means the leader of round $r-1$ must create at least one vertex in round $r-1$. Also, the byzantine processes must create supporter vertices so we get a total of $2f+1$ supporters. These are the v_1 vertices from byzantine processes.

- To allow byzantine processes to create vertices in round r which are not certificates, we have to ensure there is at least one vertex in round $r-1$ that is not a supporter of any leader vertex. We fulfill this requirement by letting the byzantine processes create the v_2 vertices.
- Now each of the v_2 vertices in round r must reference at least $2f+1$ vertices in round $r-1$, and they cannot come from the leader of round $r-1$. Therefore, at least $2f+2$ processes (the leader plus $2f+1$ other processes) must create vertices in round $r-1$. Among them, at least $f+2$ must be honest.
- Finally, we cannot delay delivering vertices to the honest processes indefinitely, due to the partial synchrony assumption. If an honest process p does not create any vertex in round r, then eventually it has to create a vertex in some round $r' > r$. Therefore, we have to periodically refresh the set of honest processes that create vertices in each round.

To fulfill the last bullet above, we associate each round $r \geq 3$ with a set $S(r)$ of honest processes, such that the processes creating certificates in round r are exactly the processes in $S(r) \cup S(r+1)$. Formally, we require the sets $S(r)$ to satisfy the following properties:

- If $leader\_at(r)$ is honest, then $leader\_at(r) \in S(r)$ (because $leader\_at(r)$ must create at least one vertex in round r);
- $|S(r)| \geq f+2$;
- $|S(r) \cup S(r+1)| \leq f+3$.

Thus the number of certificates in each round is upper-bounded by $f+3$. If we assume $f \geq 3$, we have $f+3 \leq 2f$.

One way to construct the $S(r)$ sets is to first pick any set S with $|S| = f+2$ and $leader\_at(3) \in S$ as $S(3)$. If $S(r)$ includes $leader\_at(r+1)$ then set $S(r+1) = S(r)$. Otherwise, remove one element from $S(r)$ and insert $leader\_at(r+1)$ to get $S(r+1)$.

Fig. 5 shows how the global DAG with $f = 3$ would look like when all the conditions described above are satisfied. One can check that in each round there are indeed at most $2f$ certificates in each round. Thus no leader vertex can be directly committed.

In what follows we describe an infinite trace that constructs Fig. 5 in a round-by-round fashion. We begin from the state where each process has created a genesis vertex in round 1. In round 2, we make every honest process create a supporter vertex by including L_1 in its predecessor list. These are not certificates since there is no round 0. Each byzantine process creates two vertices v_1, v_2. We let v_1 be a supporter of L_1 and v_2 be a non-supporter, by not including L_1 in its predecessor list.

In each round $r \geq 3$, we perform the following actions:

1) At the beginning, all processes in $S(r)$ should have created a vertex in round $r-1$, and are waiting for other vertices;
2) We deliver $2f+1$ supporter vertices in round $r-1$ to the processes in set $S(r)$, so they enter round r and create vertices that both support a leader vertex of round $r-1$ and certify a leader vertex of round $r-2$;
3) Each byzantine process creates a supporter for a leader vertex of round $r-1$; these vertices do not certify any leader vertex of round $r-2$; this is possible as there are byzantine vertices in round $r-1$ that are not supporters;
4) Each byzantine process additionally creates a vertex that neither supports any leader vertex of round $r-1$ nor certifies any leader vertex of round $r-2$;
5) Now that there are $2f+1$ vertices in round r, we deliver these vertices to processes that are in $S(r+1)$ but not in $S(r)$, so they jump to round r and create vertices that both support a leader vertex of round $r-1$ and

Algorithm 4 Modified Round-jumping Behavior

```
1: GCT: the timepoint where every honest process has been updated to
   follow the new round-jumping behavior.
2: upon observing 2f + 1 vertices in some round r > curr_round:
3:     Enable and reset local timer to 2Δ.
4:     if after GCT then
5:         Update commit decisions based on current DAG.
6:         for each round r' with curr_round < r' < r do
7:             if r' ≥ 3 and decisions[r' − 2] = Undecided then
8:                 Create vertex in round r'.
9:     else
10:        Whether to create vertices in rounds
11:        curr_round < r' < r is implementation-defined.
12:    Create vertex in round r.
13:    curr_round ← r + 1.
```

certify a leader vertex of round $r-2$; this restores the loop invariant at the beginning of this procedure.

Since every honest process gets periodically updated about the latest vertices, we are not violating the partial synchrony assumption. As long as we deliver these vertices fast enough, we will not trigger any timeouts either. Thus an adversary that controls both byzantine processes and message delivery order can break liveness of Mysticeti.

4. Restoring Liveness of Mysticeti

As shown in Sec. 3, Mysticeti as specified by Algs. 1 to 3 does not satisfy liveness. In this section we describe a modification to Mysticeti that restores its liveness property.

Our modification is to replace lines 15–18 of Alg. 3 with Alg. 4. We assume there is a timepoint called the *global catchup time* (GCT) that is known to the honest parties. Before GCT, when an honest process jumps to round r, it is implementation-defined whether it creates vertices in rounds $r' < r$. In particular, existing implementations that follow Alg. 3 are compatible with Alg. 4 before GCT.

After GCT, we require that if an honest party jumps to round r, it must create a vertex in every round $r' < r$, unless it has already made a decision for the round $r' - 2$. The intuition behind this rule is as follows. After GCT, suppose that an honest leader creates a vertex L_r in round r. Before any honest process times out in round $r+2$, it should have received $2f+1$ supporters for L_r. Now if some honest process chooses to jump over round $r+2$, it must have already made a decision for round r, which it should disseminate to other processes. We will prove later that the decision can only be to commit L_r. If none of the honest processes jump over round $r+2$, then they should all create certificates for L_r. In that case L_r is also committed.

What about leader vertices created before GCT? Can we still provide some liveness guarantee to them? Due to the counterexample in Sec. 3 we cannot ensure they will be committed within bounded time. However, it turns out the adversary cannot manipulate honest processes to skip them either. The idea is that we can still prove at least $f+1$ honest processes will create certificates for a leader vertex L_r that is created before GCT but after GST. Thus every vertex in round $r' \geq r+3$ will indirectly reference at least one certificate of L_r, so it will not be indirectly skipped.

To summarize, our modified Mysticeti satisfies the following liveness properties:

Theorem 3 (Weak Liveness of Mysticeti)**.** *After* $\mathsf{GST}+\Delta$*, every honest process will always eventually create new leader vertices; all new leader vertices created by honest processes will get at least* $f+1$ *certificates from honest processes.*

Theorem 4 (Strong Liveness of Mysticeti)**.** *After* $\max\{\mathsf{GST}+\Delta, \mathsf{GCT}\}$*, every new leader vertex created by an honest process will be committed within bounded time.*

In Sec. 5, we show how to formally prove these results. They correspond to the Rocq lemmas `mys_liveness_weak` and `mys_liveness_strong` in the artifact.

Performance Impact of our Fix. The notion that honest processes should create vertices in every round they jump over might sound alarming. An honest process might be manipulated into creating a large number of vertices in a short period of time, which would congest the network. Nevertheless, in the normal case where all participating parties are honest, and the adversary is only capable of reordering network messages, we can show this performance concern can be dismissed.

Theorem 5. *If all processes are honest, then after* $\mathsf{GST}+6\Delta$ *every new vertex is a certificate.*

This lemma is a side-result of the liveness proof. The corresponding Rocq lemma is called `every_vert_certificate` in the artifact. See Appendix B for the proof. Thus after $\mathsf{GST}+6\Delta$, if an honest process jumps to round r, it must have observed $2f+1$ certificates for every round $r' \leq r-2$. Following Alg. 4, it only needs to create one vertex in round r.

In the case where there is active byzantine attack, honest processes can still revert back to the behavior before GCT, to limit the number of vertices created during jumping. As shown in Theorem 3, this will not break liveness of the protocol, but may delay committing leader vertices.

5. Formalizing Safety and Liveness of Mysticeti under LiDO-DAG

The LiDO-DAG Model. Readers will notice that the "safety" and "liveness" theorems of Mysticeti (Theorems 1, 3 and 4) are not quite the same as what we define as safety and liveness in the introduction. For example, Theorem 1 concerns only the decisions made by honest processes for each round, but says nothing about the global consensus log. A proper safety proof following the definition given in the introduction usually takes the following form:

- For every reachable network state s, we define a corresponding global consensus log $log(s)$;
- For every network state s' that is reachable from s, we prove that $log(s)$ is a prefix of $log(s')$;

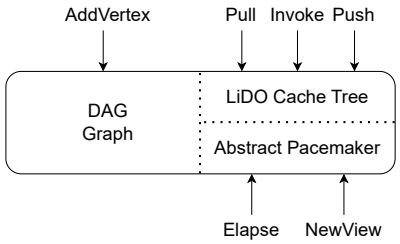

Figure 6. The LiDO-DAG model of DAG-based consensus.

- For every honest process p_i, we prove the local consensus log of p_i under state s must be a prefix of $log(s)$.

These conditions are sufficient to ensure the local consensus logs of honest processes are always consistent with each other, as they are all prefixes of the global log. However, extracting the global consensus log from a given network state is often a complex process that is inconvenient to reason about formally. This is especially true for DAG-based consensus as its consensus log is defined in multiple steps: first ordering the leader vertices and then expanding each entry to its graph closure. This makes proof engineering rather tedious, and the proofs are difficult to reuse.

The general philosophy of LiDO [27] and LiDO-DAG [28] is to provide resuable abstractions that factor out the common logic of consensus protocols, so that it is easier to connect the network-level safety and liveness theorems (Theorems 1, 3 and 4) to the abstract-level statements. To reuse the proofs of LiDO-DAG, one defines a refinement mapping from the network model to LiDO-DAG, and then translate the theorems of LiDO-DAG back to network-level properties. Here we give an overview of LiDO-DAG.

As shown in Fig. 6, LiDO-DAG is a transition system whose state consists of three parts: 1) the global DAG graph; 2) the LiDO cache tree; 3) the abstract pacemaker. Only the first two parts are related to the safety proof and will be introduced first. The abstrace pacemaker will be described as we introduce the liveness proof.

The global DAG graph of LiDO-DAG corresponds directly to the global DAG in the network model. Its state consists of a finite set of *vertices*, which are defined as:

$$Vertex \triangleq \mathbb{N}_{id} * \mathbb{N}_{builder} * \mathbb{N}_{round} * Data * (\mathsf{list}\ \mathbb{N}_{preds}).$$

The only allowed operation on the global DAG is $AddVertex()$ which adds a single vertex to the graph.

In authenticated DAG protocols like Bullshark, each vertex can be uniquely identified by its builder and round. Therefore in [28] the id field is always implicitly defined by builder and round. For unauthenticated DAG protocols like Mysticeti, we instead use the cryptographic hash of vertex records to identify vertices. Refinement from the global DAG of Mysticeti to the global DAG of LiDO-DAG is straightforward: whenever a new vertex record is accepted by the system agent $\mathcal{S}$ (see Alg. 1), the corresponding vertex is added to the global DAG of LiDO-DAG.

The LiDO cache tree is a representation of the leader vertex consensus log that abstracts away details like leader vertex slot location and frequency. It postulates that DAG-based consensus protocols can be interpreted as simulating leader-based consensus, and progresses through logical *views*. In each view, a predefined leader attempts to commit a new leader vertex. For example, in the Bullshark protocol [15] there is one leader vertex slot in every second round, thus a logical view of Bullshark corresponds to two DAG rounds (called a *wave* in [15]). For Mysticeti, DAG rounds and logical views are in one-to-one correspondence: DAG round r corresponds to logical view v with $v = r$.

The task of committing a leader vertex is further broken into three steps:

- **Pull**: the *parent* of the current view is uniquely determined, which is the branch of the global consensus log the current view will append to;
- **Invoke**: the leader vertex itself is uniquely determined;
- **Push**: the leader vertex is committed.

When each step succeeds, a corresponding *cache node* is added to the cache tree. The state of the LiDO cache tree is thus a finite set of cache nodes. The cache nodes corresponding to pull, invoke, and push are called ECache (leader Election), MCache (Method invocation), and CCache (Commit), respectively. There is also a root cache node corresponding to the initial empty consensus log. Formally, they are defined as:

$$\begin{aligned} CacheNode \triangleq & Root \\ & \mid ECache(\mathbb{N}_{view} * \mathbb{N}_{parent}) \\ & \mid MCache(\mathbb{N}_{view} * \mathbb{N}_{leader\_vert\_ID}) \\ & \mid CCache(\mathbb{N}_{view}). \end{aligned}$$

If Σ is a cache tree, we use $\Sigma[v].ecache$ to represent the ECache of view v. If such a cache node does not exist, we write $\Sigma[v].ecache = \bot$. Similarly, we use $\Sigma[v].mcache$ to represent the MCache of view v, and use $\Sigma[v].ccache$ to represent the CCache of view v. The cache creation rules of LiDO guarantees the uniqueness of these nodes.

Each cache node except root has a *parent node* defined, which chains the nodes into a tree structure (Fig. 7). The parent relation is defined as:

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

The consensus log up to an MCache is the list of all MCaches along the path from root to that MCache.

The cache node creation rules of LiDO-DAG maintain the following invariant: if view v has a CCache, and view $v' > v$ has an ECache, then $v'.ecache.parent \geq v$. Intuitively this means if the leader vertex of view v is committed, then all future views must append after the leader vertex of view v. It is shown in [27], [28] that this invariant is sufficient to ensure all committed leader vertices are on a single branch.

The *highest committed view* is the view with the highest number that contains a CCache. The global leader vertex

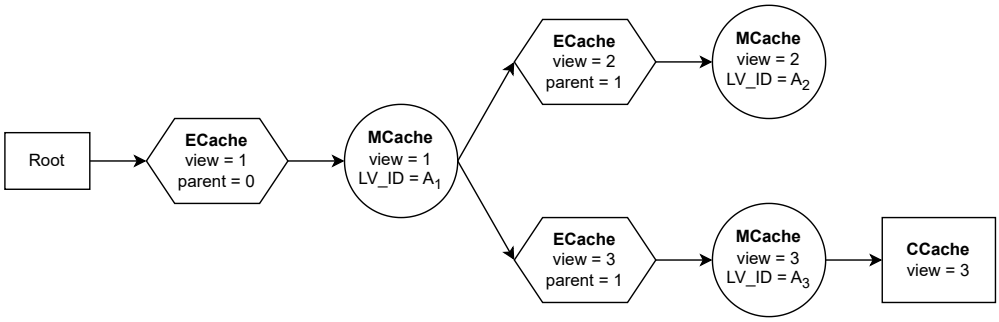

Figure 7. Example of LiDO-DAG cache tree. Reproduced from [27]. `LV_ID` stands for `leader_vertex_ID`.

```
Record MDAG_State : Type := {
  (* The global DAG *)
  mdag_udag : UDAG_State;
  (* History variable (p, r) that indicates
     process p has received leader vertex of
     round r before timeout in round r+1 *)
  mdag_recv_leader_verts : list (nat * nat);
  (* History variable (p, r) that indicates
     process p has received 2f+1 supporters for
     leader vertex of round r before timeout
     in round r+2 *)
  mdag_recv_certs : list (nat * nat);
  (* History variable (p, r) that indicates
     process p has timed-out in round r *)
  mdag_timeouts : list (nat * nat);
  (* History variable (p, r) that indicates
     process p has decided to jump over round
     r without creating a vertex *)
  mdag_jumps : list (nat * nat);
}.
```

Figure 8. The MysticetiDAG Layer.

consensus log is the consensus log up to the MCache of the highest committed view. In [28], it is proved that if Σ, Σ' are two LiDO cache trees and Σ' is reachable from Σ, then the leader vertex consensus log of Σ is a prefix of the leader vertex consensus log of Σ'.

The full consensus log is defined to be the leader vertex consensus log with each entry replaced by its graph closure. [28] similarly proved that if Σ' is reachable from Σ, then the full consensus log of Σ is a prefix of the full consensus log of Σ'. Thus the two remaining steps of proving safety of a protocol are:

- Construct a refinement mapping from network states to LiDO-DAG states;
- Prove that the local leader vertex consensus logs are prefixes of the global log.

The Network Model of Mysticeti. To describe details of the safety proof as well as the liveness proof we need to introduce how the network system is formally modeled.

The network model is defined in two layers. In the first layer (`MysticetiDAG.v`) we only consider the vertex creation rules. In this layer we prove the weak liveness theorem (Theorem 3). In the second layer (`MysticetiCommit.v`) we add in the ordering algorithm and prove safety and the strong liveness theorem (Theorem 4).

The state variables of the `MysticetiDAG` layer are shown in Fig. 8. To keep things simple, the model does not capture the local DAGs of individual processes. Instead, we record only the global DAG, and use history variables to keep track of the parts each honest process has received.

A complete modeling of Algs. 3 and 4 would also need to capture the $curr\_round$ variable and the remaining time of the local timer. As these variables are unrelated to the safety proof, they are defined later in the liveness proof.

As shown in Alg. 3, there are three conditions under which an honest process may create a vertex in round r: 1) it has received both a leader vertex of round $r-1$ and $2f+1$ supporters of a leader vertex of round $r-2$; 2) it has timed-out in round r; 3) it has received $2f+1$ vertices of round r. The formal encodings of these conditions are straightforward, except one important detail. When an honest process p_i creates a vertex v in round r, we require it to "try its best" to make the vertex a supporter and a certificate. Formally: if p_i has previously received a leader vertex of round $r-1$, i.e. $(p_i, r-1) \in$ `mdag_recv_leader_verts`, then we require v to reference at least one leader vertex of round $r-1$. Similarly, if p_i has previously received $2f+1$ supporters for a leader vertex v' of round $r-2$, then we require v to be a certificate for v'. Fig. 9 shows the formal precondition of process p_i adding vertex v upon timeout in round r, which shows how the above conditions are formalized.

The MysticetiCommit layer is shown in Fig. 10. In this layer we add history variables about the commit decisions the honest processes have made. We also record whether we are before or after GCT, as it controls the round-jumping rules of honest processes.

The Safety Proof. To prove safety of Mysticeti, we construct a refinement mapping from traces of the network model to traces of LiDO-DAG. Since the MysticetiCommit layer records all valid vertices and all commit decisions, we are essentially acting as the virtual system agent $\mathcal{S}$ which monitors all network activities and builds the global DAG and consensus log.

Let D be the set of all decisions made by honest processes. The difficult part of the proof is to show that two processes cannot make conflicting decisions for the same round (Theorem 1). Once this is established, constructing the cache tree is relatively straightforward. We find the largest R such that every round $r \leq R$ is decided in D. For each round that is skipped, we do not create any cache node. For each round r that has a committed leader vertex v, we create

```
Definition mdag_add_vert_timeout_pre
  (r : nat) (nid : nat) (id : nat)
  (vert : UDAG_Vert) (mdag : MDAG_State) :=
  (* r = 1 or the process has created vertex in
     round r - 1 or the process has decided to
     jump over round r - 1 *)
  (r = 1 \/
   (exists id,
      match NatMap_find id
        mdag.(mdag_udag).(udag_verts) with
      | None => False
      | Some v => v.(udag_vert_round) = r - 1 /\
        v.(udag_vert_builder) = nid
      end) \/
   In (nid, r - 1) mdag.(mdag_jumps)) /\
  (* The new vertex is valid, and honest
     processes do not create two vertices
     in a single round *)
  udag_add_vert_pre id vert mdag.(mdag_udag) /\
  (* If the process has previous received a
     leader vertex of round r-1, then it
     references at least one leader vertex
     of round r-1 *)
  (In (vert.(udag_vert_builder),
       vert.(udag_vert_round) - 1)
      mdag.(mdag_recv_leader_verts) ->
   mdag_vert_is_supporter vert mdag) /\
  (* If the process has previous received
     2f+1 supporters for some leader vertex
     of round r-2, then the new vertex is
     a certificate *)
  (In (vert.(udag_vert_builder),
       vert.(udag_vert_round) - 2)
      mdag.(mdag_recv_certs) ->
   mdag_vert_is_certificate vert mdag).
```

Figure 9. Formal requirements of adding vertices upon timeout.

```
Record MDAG_Commit_State : Type := {
  (* The MysticetiDAG state *)
  mcommit_mdag : MDAG_State;
  (* History variable (r, v) that indicates
     leader vertex v of round r has been
     committed by an honest process *)
  mcommit_commit : list (nat * nat);
  (* History variable r that indicates an honest
     process has decided to skip round r *)
  mcommit_nack : list nat;
  (* Whether we are before or after GCT *)
  mcommit_gct : bool;
}.
```

Figure 10. The MysticetiCommit layer.

cache nodes as follows:

- Pull: the parent of round r is set to the largest round $r' < r$ that is committed. If such a round does not exist, the parent is set to the root node.
- Invoke: the $leader\_vert\_ID$ is set to the ID of v.
- Push: we create a CCache to indicate this round has been committed.

Now because the decisions made by any honest process is a subset of D, it is easy to see the local leader vertex consensus logs must be prefixes of the global log. Then we invoke lemmas in the LiDO-DAG model to show the local full consensus logs are also prefixes of the global log.

Proof of Theorem 1: First we notice the following lemmas. In the artifact they are contained in the proofs of the safety theorems.

Lemma 1. *If there exists a certificate for a leader vertex v in round r, then there cannot exist a skip pattern for round r. Also, there cannot exist certificates for two different leader vertices v, w in the same round.*

Proof: If v' is a certificate for v then we have $2f+1$ supporters for v in round $r+1$. If a skip pattern for round r exists then we also have $2f+1$ vertices which do not support v in round $r+1$. By quorum overlapping, at least one honest process must have created two vertices, one of which supports v, the other does not, which is impossible.

If v' is a certificate for v and w' is a certificate for w, then we have $2f+1$ supporters for v, as well as $2f+1$ supporters for w in round $r+1$. By quorum overlapping, at least one honest process must have created two vertices that support v and w respectively, which is also impossible. □

Lemma 2. *If two honest processes decide to commit v, v' in round r respectively, then $v = v'$.*

Proof: Regardless of whether we use the direct or indirect decision rule to commit v, there must exist at least one certificate of v. By Lemma 1, we cannot commit two different vertices in round r. □

Lemma 3. *If there exists $2f+1$ certificates for a leader vertex v in round r, then every vertex in round $r' \geq r+3$ directly or indirectly references at least one certificate for v, and no honest process will skip round r. This lemma is also true if we replace "$2f+1$ certificates" with "$f+1$ certificates from honest processes."*

Proof: Every vertex in round $r' \geq r+3$ will reference at least $2f+1$ vertices of round $r+2$. By quorum overlapping, it should reference at least one certificate of v. □

Now there are four different ways to decide a round (direct/indirect commit/skip). If two processes both make a decision for round r, there are 16 combinations, of which 14 can be directly closed with Lemmas 1 to 3 above.

The remaining cases are where one process indirectly commits vertex v of round r, and the other process indirectly skips round r. This case requires induction over the network trace. Suppose process p_i has decided to skip every round r' with $r+3 \leq r' < R$ and commit vertex v' of round R. Meanwhile, process p_j has decided to skip every round r' with $r+3 \leq r' < R'$ and commit vertex v'' of round R'. Notice that we must have $R = R'$, otherwise the decisions on round $\min\{R, R'\}$ are conflicting which violates the induction hypothesis. Then we have $v' = v''$, which proves p_i, p_j cannot make conflicting decision for round r. □

The Liveness Proof: Segmented Traces. To prove liveness of Mysticeti we have to model the local timers of honest processes. In general, to model the dynamics of time-dependent systems we have to introduce timed-traces [34]. However, continuous time variables can be difficult to work with. Especially in partially synchronous protocols like Mysticeti, we have to deal with multiple time variables, one at each

```
Class MysticetiWeakOracle : Type := {
  (* k |-> MysticetiDAG state at end of tau_k *)
  mys_weak_oracle_dag : nat -> MDAG_State;
  (* (k, p) |-> current_round of process p
     at end of tau_k *)
  mys_weak_oracle_local_ar : nat -> nat -> nat;
  (* (k, p) |-> remaining time of local timer of
     process p at end of tau_k *)
  mys_weak_oracle_local_rt : nat -> nat -> nat;

  (* Reachability of traces *)
  mys_weak_oracle_init_valid :
    mdag_valid (mys_weak_oracle_dag 0);
  mys_weak_oracle_reachable :
    forall n,
    mdag_reachable
      (mys_weak_oracle_dag n)
      (mys_weak_oracle_dag (S n));
}.
```

Figure 11. Encoding of segmented traces of Mysticeti.

```
mys_weak_oracle_timeout :
  (* For any given k and process ID nid ... *)
  forall k nid,
  (* if nid is an honest participant ... *)
  In nid participant ->
  node_assump nid = Synchronous ->
  (* local timer remaining time has reached 0
     at the end of tau_k ... *)
  mys_weak_oracle_local_rt k nid = 0 ->
  (* the process does not enter a new round
     within delta ... *)
  mys_weak_oracle_local_ar (S k) nid =
    mys_weak_oracle_local_ar k nid ->
  (* then it times out before the
     end of tau_{k+1} *)
  In (nid, mys_weak_oracle_local_ar k nid)
     (mys_weak_oracle_dag (S k)).(mdag_timeouts);
```

Figure 12. Example of encoding temporal assumptions on segmented traces.

local timer. Adding to the difficulty is the fact that the model is parametric in the number of participants and the network latency Δ. To handle these difficulties, [27] introduced a formalism called *segmented traces* for encoding the temporal dynamics of honest processes.

The core idea of segmented traces is to take snapshots of the system at regular intervals of Δ after GST. Formally, if τ is an infinite timed-trace, then its (GST, Δ)-segmentation is $\tau_0, \tau_1, \cdots$ where τ_k is the prefix of τ consisting of all events occurred before the timepoint $GST + k\Delta$. Thus each τ_k is a prefix of τ_{k+1}, as τ_{k+1} extends τ_k with the events occurred within the interval $[GST + k\Delta, GST + (k+1)\Delta)$.

The reason segmented traces are useful is it allows encoding temporal assumptions of network systems without explicit reference to the latency parameter Δ. Fig. 11 shows how we formally represent segmented traces of Mysticeti. We assume a function `mys_weak_oracle_dag` which maps k to the system state at the end of the partial trace τ_k. We also introduce two functions that represent the $curr\_round$ variables and the remaining time of local timers in Alg. 3.

Fig. 12 shows how the timeout assumption is encoded using segmented traces. We follow [27] and represent the remaining time t of the local timer by its discrete approximate value $\lfloor t/\Delta \rfloor$. This makes it an integer that decreases by exactly 1 over each period of Δ, unless the timer is reset. We then state that, if at the end of τ_k the remaining time is 0 (which means $0 \leq t < \Delta$), and the process does not enter a new round between τ_k and τ_{k+1}, then by the end of τ_{k+1} there should be at least one timeout event delivered to that process. See Appendix B.

The Liveness Proof: The Abstract Pacemaker. Our goal is to prove that after GST, every honest process can always eventually create and commit new leader vertices. It is tempting to think that, if an honest process p is in round r when GST commences, then it can commit a leader vertex in every round $r' > r$ with $leader\_at(r') = p$. This is not strictly true, for when GST commences p might be lagging behind global network progress, so that even if it creates a leader vertex in some round $r' > r$ it might not get enough certificates to be committed. The gist is that to prove liveness of consensus protocols it is not sufficient to look at the state of any single process. We have to consider the timer status of all honest processes together.

LiDO and LiDO-DAG simplifies the issue by introducing a pacemaker abstraction. The pacemaker consists of just two variables called *global active view* (GAV) and *global remaining time* (GRT). They provide a useful summary of the local timers of honest processes. For Mysticeti, they are related to the local states as follows:

- GAV $= \max_{p \in honest} p.curr\_round$, i.e. the highest round any honest process has ever entered.
- GRT $= \min_{\substack{p \in honest \\ p.curr\_round = \text{GAV}}} p.remaining\_time$, i.e. the least amount of remaining time among processes that are in the GAV round.

By a slight abuse of notation, for any finite trace τ we shall write $\text{GAV}(\tau)$ to denote the value of GAV in the final system state after replaying the trace τ. Similarly, $\text{GRT}(\tau)$ is the value of GRT after replaying the trace τ. With the pacemaker abstraction, liveness of Mysticeti is decomposed into the following safety properties:

- There exists constant C, s.t. for any k we have $\text{GAV}(\tau_{k+C}) > \text{GAV}(\tau_k)$;
- After GCT, there exists constant C', s.t. for any r with $leader\_at(r)$ being honest, if $\text{GAV}(\tau_k) < r$ and $\text{GAV}(\tau_{k+1}) \geq r$, then a leader vertex of round r is committed by the end of $\tau_{k+C'}$;
- The leader schedule periodically admits three consecutive honest leaders, so that all previous rounds can be decided via Theorem 2.

The first property can be easily proved from liveness assumptions. The third property is true for the special case of round-robin leader schedule [9], but we simply state it as an assumption. The second property is the tricky part.

The Liveness Proof: Liveness Checkpoints. To prove that a leader vertex of round r will be eventually committed, we need to analyze the dynamics of the system starting from the point where the first honest process enters round r.

An informal outline of the proof is as follows. We first introduce an important lemma, which can be easily proved

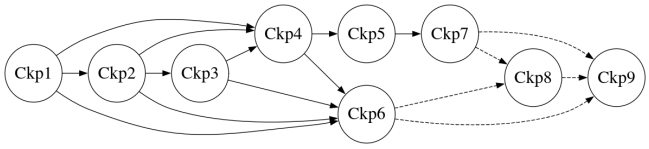

Figure 13. Round checkpoint transition guarantees. Once the network state has reached a certain checkpoint for round r, it should hop to the next checkpoint within bounded time. Dashed lines represent transitions only guaranteed after GCT.

by induction over the network trace.

Lemma 4. *If no honest process has ever timed out in some round R, then every vertex in round R built by an honest process must support a leader vertex in round $R-1$, and also certify a leader vertex in round $R-2$.*

Proof: If no honest process has ever timed out in round R, then honest processes can create vertices in round R under only two conditions: 1) it has observed a leader vertex of round $R-1$ and $2f+1$ supporters for a leader vertex of round $R-2$; 2) it has received $2f+1$ other vertices in round R.

In the first case, the lemma immediately follows from the predecessor rule (line 19 of Alg. 3. In the second case, among the $2f+1$ vertices that already exist in round R, at least $f+1$ must come from honest processes. By the induction hypothesis, they must all support a leader vertex in round $R-1$, and certify a leader vertex in round $R-2$. The lemma follows again from the predecessor rule. □

Now suppose that $\text{GAV}(\tau_k) < r$. We wait until the the first honest process enters round $r+1$. Within Δ, the leader of round r will have either created a vertex in round r, or jumped over round r. The second case is impossible, because to jump over round r there must be $2f+1$ vertices in round $r+1$, of which at least $f+1$ must be supporters for some leader vertex of round r, which contradicts the fact that $leader\_at(r)$ has jumped over round r.

After the leader vertex v of round r is created, within Δ every honest process will receive v. They could not have timed out in round $r+1$ at this point. Therefore, within another Δ each process will either create a supporter for v, or jump over round $r+1$. In the second case, we can show that there must be $2f+1$ vertices in round $r+2$, of which at least $f+1$ are from honest processes and are certificates for v. This is the weak liveness result. If no honest process jumps over round $r+1$, then every honest process will receive $2f+1$ supporters for v, so eventually there should be at least $f+1$ certificates for v in round $r+2$ as well.

To get the strong liveness result, notice that every honest process will eventually either create a vertex in round $r+2$, or jump over round $r+2$. In the second case it must have already made a decision for round r. Since there exists at least $f+1$ certificates from honest processes in round $r+2$, by Lemma 3, the decision must be to commit v. If no honest process jumps over round $r+2$, then there will be $2f+1$ certificates for v, and we can directly commit v. □

To translate the above outline into a rigorous proof, we defined a list of "checkpoints" for round r that the network state must progress through. There are a total of 9 checkpoints. Appendix B lists the formal definition of every checkpoint. The first checkpoint states that at least one honest process has entered round $r+1$. The final checkpoint is that a leader vertex of round r is committed. However, before GCT we can only guarantee each round will reach checkpoint 6 or 7.

Fig. 13 summarizes the checkpoint progress guarantees we have proved. Once the network state reaches a certain checkpoint for round r, it should advance by one hop in the figure within bounded time (Δ or 2Δ depending on the checkpoint). For example, if the network state at the end of τ_k satisfies checkpoint 2, then we should reach checkount 3, 4, or 6 by the end of τ_{k+1}. By chaining these guarantees together, we formally derived liveness guarantees for Mysticeti.

An Example of Liveness Reasoning. We explain the proof for the following statement, as an example of how we perform liveness reasoning.

Lemma 5. *If the network state at the end of τ_k satisfies checkpoint 2 of round r, then at the end of τ_{k+1} it must satisfy checkpoint 3, 4, or 6 of round r. Relevant definitions:*

- *Checkpoint 2: GAV $= r+1$, GRT ≥ 1, and the leader of round r has created a vertex in round r.*
- *Checkpoint 3: GAV $= r+1$ and every honest process has received a leader vertex of round r.*
- *Checkpoint 4: GAV $= r+2$, GRT ≥ 2; each honest process either has received a leader vertex of round r, or is still in some round $r' \leq r+1$ and will not time out in round $r+1$ within Δ; at least $f+1$ honest processes have created supporter vertices in round $r+1$ for a leader vertex of round r.*
- *Checkpoint 6: GAV $\geq r+3$; each honest process either has received $2f+1$ supporters for a leader vertex of round r, or is still in some round $r' \leq r+2$ and will not time out in round $r+2$ within Δ; at least $f+1$ honest processes have created certificates in round $r+2$ for a leader vertex of round r.*

Proof: Checkpoint 2 states that GAV $= r+1$ and GRT ≥ 1 at the end of τ_k, and the leader of round r has created a vertex in round r. Let $R = \text{GAV}(\tau_{k+1})$. Since $curr\_round$ can only increase but not decrease, we have $R \geq r+1$. We have three cases: $R = r+1, R = r+2$, or $R \geq r+3$.

Case 1: $R = r+1$: In this case, every honest process must have received the leader vertex of round r, so checkpoint 3 is satisfied.

Case 2: $R = r+2$: Since GRT ≥ 1 at the end of τ_k, no honest process could have timed out in round $r+1$ by the end of τ_{k+1}. Since $R = r+2$, there must be $2f+1$ vertices in round $r+1$, of which at least $f+1$ come from honest processes. By Lemma 4, these $f+1$ vertices must be supporters for the leader vertex of round r. Hence checkpoint 4 is satisfied.

Case 3: $R \geq r+3$: Since GAV $= r+1$ by the end of τ_k, no honest process could have timed out in round $r+2$ by the end of τ_{k+1}. Since $R = r+3$, there must be $2f+1$ vertices in round $r+2$, of which at least $f+1$ come from honest processes. By Lemma 4, these $f+1$ vertices

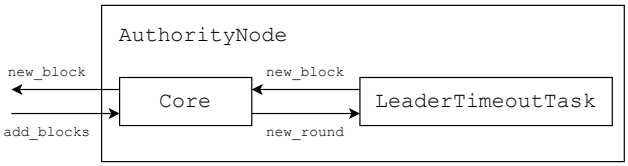

Figure 14. Simplified architecture of the Sui implementation of Mysticeti. The `Core` and `LeaderTimeoutTask` structures roughly correspond to the atomic state machine and local timer in our system model. `Core` communicates with other peers via gRPC over TLS. It accepts vertex records via `add_blocks()`. Byzantine peers can exploit `add_blocks()` to inject vertex records into the staging area in a controlled order.

must be certificates for the leader vertex of round r. Hence checkpoint 6 is satisfied. □

Proof Effort. Beyond the files imported from the LiDO-DAG project, the proofs specific to Mysticeti span 6271 lines and was developed by two persons in two months.

6. Real-world Implementations of Mysticeti

In the previous sections we have fully resolved the theoretical correctness issues of Mysticeti. However, there remains the question of how Mysticeti behaves in practice. In particular we want to know whether the liveness attack described in Sec. 3 will work on existing implementations. To this end we audited the source code of the Sui blockchain [35, commit 2f52a72] which is based on Mysticeti. Here we describe our findings.

Fig. 14 shows a simplified architecture of the consensus component of Sui. The two most important structures are `Core` and `LeaderTimeoutTask`, which roughly corresponds to the atomic state machine and local timer in our system model. `Core` structures communicate with each other through gRPC over TLS. `LeaderTimeoutTask` only serves to deliver timeout signals to the local `Core`.

There are two different ways to deliver vertex records to a process. First, the builder of a vertex may perform gRPC calls to push the record to its peers. This code-path is difficult to manipulate because TLS already protects the order of messages. In Fig. 3, for example, we cannot deliver vertex 5 before vertex 3 through this method because process 3 is honest and it must send vertex 3 before vertex 5.

However, the implementation has a second way of receiving vertices via *synchronizers*. When a process suspects it is lagging behind global network progress, it executes one of the synchronizers to fetch missing blocks from its peers. The simplest kind of synchronizer is triggered when the process receives a vertex record with missing predecessors. In this case the synchronizer makes a gRPC call to the author of the record to fetch the missing vertices. If the author is byzantine, this allows the adversary to inject an arbitrary list of vertices into the staging area of the honest process.

We constructed a testcase that simulates delivering vertices to an honest process in the order shown in Fig. 3. We confirmed that the current implementation follows Alg. 3 and jumps over round 2 without creating a vertex. We also confirmed that a malicious peer can trigger the synchronizers of honest processes by sending vertex records. Together, this provides an actual path to launch the attack shown in Sec. 3 and delay committing leader vertices.

7. Discussions and Related Works

Formal verification of security properties of distributed systems is a longstanding topic that has been attacked from many angles, including proof checking (e.g. [25], [27], [36]), model checking (e.g. [37], [38], [39]), and specialized languages that enable correctness-by-construction (e.g. [40], [41]). However, since DAG-based consensus is a relatively new paradigm, its formal security has received comparatively little attention.

[42] introduced a formal model for DAG-based consensus in TLA+ and applied it to several protocols including DAG-Rider [14], Bullshark [15], and Cordial Miners [17]. Similarly, [43] presented a model of DAG-based consensus with dynamic stakes in the ACL2 proof checker. However, both works only considered the safety aspect of consensus. [44] verified both safety and liveness of Hashgraph [45], an early DAG-based consensus algorithm. Another work [46] verified safety and liveness of "Nakamoto-style proof-of-stake," a consensus protocol that also uses a DAG of blocks, but quite distinct from what is called DAG-based protocols today. In these works, liveness of consensus depends on probability, but the probabilistic part of the proof is not verified. In contrast, the Mysticeti protocol does not use probability, and liveness is deterministic.

The liveness flaw of Mysticeti was first noted in [16], but [16] does not provide suggestions on how to fix the protocol. In [30], it is suggested that each honest process must create a vertex in every single round. This rule is more stringent than our proposed fix. Our rule still allows honest processes to jump over rounds, provided they have made commit decisions for their leader vertices. As argued in Sec. 4, in the normal case where there is no active byzantine attack, our rule allows much more efficient round-jumping. Also, [30] requires their rule to be observed from the beginning of protocol execution. Our analysis allows adopting the rule during the middle of execution, making our model more compatible with existing implementations.

One aspect of round-jumping we have not yet considered in this work is making the intermediate vertices logical rather than physical. That is, when jumping to a higher round the honest processes do not actually need to create and distribute the intermediate vertices. They can simply embed commit votes for earlier vertices in their latest vertices. However, this will likely complicate a protocol that is already extremely complex and subtle, as evidenced by the counterexamples in this work. It might also increase the computational overhead of the ordering algorithm. Therefore, we leave this possbility of improving the Mysticeti protocol to future work.

Acknowledgement

We would like to thank Alberto Sonnino and other authors of [18], the anonymous reviewers, and the shepherd for their

helpful feedback. This work is supported in part by a Sui Academic Research Award, by NSF grants 2019285 and 2313433, and by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590130. 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.

References

[1] S. Nakamoto. (2008, Oct.) Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] H. Vranken, "Sustainability of bitcoin and blockchains," *Current Opinion in Environmental Sustainability*, vol. 28, pp. 1–9, 2017, sustainability governance. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877343517300015

[3] M. Saad, A. Anwar, S. Ravi, and D. Mohaisen, "Revisiting nakamoto consensus in asynchronous networks: A comprehensive analysis of bitcoin safety and chain quality," *IEEE/ACM Transactions on Networking*, vol. 32, no. 1, pp. 844–858, 2024.

[4] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, and Y. X. Zhang, "Combining ghost and casper," 2020. [Online]. Available: https://arxiv.org/abs/2003.03052

[5] Aptos Foundation, "The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure," 2022. [Online]. Available: https://aptosfoundation.org/whitepaper/aptos-whitepaper_en.pdf

[6] M. Castro, "Practical byzantine fault tolerance," Ph.D. dissertation, Massachusetts Institute of Technology, 2001. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/thesis-mcastro.pdf

[7] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on bft consensus," 2019. [Online]. Available: https://arxiv.org/abs/1807.04938

[8] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–356. [Online]. Available: https://doi.org/10.1145/3293611.3331591

[9] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback," in *Financial Cryptography and Data Security*, I. Eyal and J. Garay, Eds. Cham: Springer International Publishing, 2022, pp. 296–315.

[10] O. Naor, M. Baudet, D. Malkhi, and A. Spiegelman, "Cogsworth: Byzantine View Synchronization," *Cryptoeconomic Systems*, vol. 1, no. 2, oct 22 2021. [Online]. Available: https://cryptoeconomicsystems.pubpub.org/pub/naor-cogsworth-synchronization

[11] A. Lewis-Pye, D. Malkhi, O. Naor, and K. Nayak, "Lumiere: Making optimal bft for partial synchrony practical," 2024. [Online]. Available: https://arxiv.org/abs/2311.08091

[12] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: a dag-based mempool and efficient bft consensus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 34–50. [Online]. Available: https://doi.org/10.1145/3492321.3519594

[13] R. Neiheiser, M. Matos, and L. Rodrigues, "Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 35–48. [Online]. Available: https://doi.org/10.1145/3477132.3483584

[14] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is dag," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, ser. PODC'21. New York, NY, USA: Association for Computing Machinery, 2021, p. 165–175. [Online]. Available: https://doi.org/10.1145/3465084.3467905

[15] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: Dag bft protocols made practical," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2705–2718. [Online]. Available: https://doi.org/10.1145/3548606.3559361

[16] N. Shrestha, R. Shrothrium, A. Kate, and K. Nayak, " Sailfish: Towards Improving the Latency of DAG-based BFT ," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 21–21. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00021

[17] I. Keidar, O. Naor, O. Poupko, and E. Shapiro, "Cordial miners: Fast and efficient consensus for every eventuality." Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2023.26

[18] K. Babel, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, A. Koshy, A. Sonnino, and M. Tian, "Mysticeti: Reaching the limits of latency with uncertified dags," 2024. [Online]. Available: https://arxiv.org/abs/2310.14821

[19] A. Spiegelman, B. Arun, R. Gelashvili, and Z. Li, "Shoal: Improving dag-bft latency and robustness," 2023. [Online]. Available: https://arxiv.org/abs/2306.03058

[20] G. Bracha, "Asynchronous byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/089054018790054X

[21] C. Cachin and S. Tessaro, "Asynchronous verifiable information dispersal," in *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, 2005, pp. 191–201.

[22] S. Das, Z. Xiang, and L. Ren, "Asynchronous data dissemination and its applications," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2705–2721. [Online]. Available: https://doi.org/10.1145/3460120.3484808

[23] IOTA Foundation, "Iota rebased: Fast forward," 2024. [Online]. Available: https://blog.iota.org/iota-rebased-fast-forward/

[24] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, p. 288–323, Apr. 1988.

[25] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," *SIGPLAN Not.*, vol. 50, no. 6, p. 357–368, jun 2015. [Online]. Available: https://doi.org/10.1145/2813885.2737958

[26] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson, "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*, ser. CPP 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 154–165. [Online]. Available: https://doi.org/10.1145/2854065.2854081

[27] L. Qiu, Y. Kim, J.-Y. Shin, J. Kim, W. Honoré, and Z. Shao, "Lido: Linearizable byzantine distributed objects with refinement-based liveness proofs," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3656423

[28] L. Qiu, J. Xiao, J.-Y. Shin, and Z. Shao, "LiDO-DAG: A framework for verifying safety and liveness of dag-based consensus protocols," Yale Univ., Tech. Rep. TR1574, Apr. 2025. [Online]. Available: https://flint.cs.yale.edu/publications/lido-dag.html

[29] L. Qiu, J. Xiao, and Z. Shao, "Artifact for s&p 2026 paper #131 mechanized safety and liveness proofs for the mysticeti consensus protocol under the lido-dag framework," Oct. 2025. [Online]. Available: https://zenodo.org/records/17345693

[30] N. Polyanskii, S. Mueller, and I. Vorobyev, "Starfish: A high throughput BFT protocol on uncertified DAG with linear amortized communication complexity," Cryptology ePrint Archive, Paper 2025/567, 2025. [Online]. Available: https://eprint.iacr.org/2025/567

[31] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 382–401, Jul. 1982. [Online]. Available: https://doi.org/10.1145/357172.357176

[32] M. Bellare and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, ser. CCS '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 62–73. [Online]. Available: https://doi.org/10.1145/168588.168596

[33] M. Bravo, G. V. Chockler, and A. Gotsman, "Liveness and latency of byzantine state-machine replication," in *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*, ser. LIPIcs, C. Scheideler, Ed., vol. 246. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 12:1–12:19. [Online]. Available: https://doi.org/10.4230/LIPIcs.DISC.2022.12

[34] L. Lamport, "Real time is really simple," Tech. Rep. MSR-TR-2005-30, March 2005. [Online]. Available: https://www.microsoft.com/en-us/research/publication/real-time-is-really-simple/

[35] Mysten Labs, "Sui." [Online]. Available: https://github.com/MystenLabs/sui

[36] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "Ironfleet: Proving practical distributed systems correct," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–17. [Online]. Available: https://doi.org/10.1145/2815400.2815428

[37] O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham, "Reducing liveness to safety in first-order logic," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, dec 2017. [Online]. Available: https://doi.org/10.1145/3158114

[38] I. Berkovits, M. Lazić, G. Losa, O. Padon, and S. Shoham, "Verification of threshold-based distributed algorithms by decomposition to decidable logics," in *Computer Aided Verification*, I. Dillig and S. Tasiran, Eds. Cham: Springer International Publishing, 2019, pp. 245–266.

[39] C. Enea, D. Giannakopoulou, M. Kokologiannakis, and R. Majumdar, "Model checking distributed protocols in must," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3689778

[40] A. K. Hirsch and D. Garg, "Pirouette: higher-order typed functional choreographies," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, Jan. 2022. [Online]. Available: https://doi.org/10.1145/3498684

[41] M. A. Le Brun and O. Dardha, "Magπ: Types for failure-prone communication," in *Programming Languages and Systems: 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2023, p. 363–391. [Online]. Available: https://doi.org/10.1007/978-3-031-30044-8_14

[42] N. Bertrand, P. Ghorpade, S. Rubin, B. Scholz, and P. Subotic, "Reusable formal verification of dag-based consensus protocols," 2024. [Online]. Available: https://arxiv.org/abs/2407.02167

[43] A. Coglio and E. McCarthy, "Formal verification of blockchain nonforking in dag-based bft consensus with dynamic stake," 2025. [Online]. Available: https://arxiv.org/abs/2504.16853

[44] K. Crary, "Verifying the hashgraph consensus algorithm," 2021. [Online]. Available: https://arxiv.org/abs/2102.01167

```
Record Client_ViewDesc : Type := {
  client_view_pull_src : option (nat * nat);
  client_view_mcaches : NatMap nat;
  client_view_max_ccache : option nat;
}.
```

Figure 15. View descriptors of LiDO cache tree.

```
Record UDAG_Vert : Type := {
  udag_vert_round : nat;
  udag_vert_builder : nat;
  udag_vert_data : dag_t;
  udag_vert_preds : list nat;
}.

Record UDAG_State : Type := {
  udag_verts : NatMap UDAG_Vert;
  udag_closure : NatMap (list nat);
}.
```

Figure 16. The unauthenticated DAG

[45] L. Baird, "The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance," Swirlds, Tech. Rep. SWIRLDS-TR-2016-01, 2016. [Online]. Available: https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf

[46] S. E. Thomsen and B. Spitters, "Formalizing nakamoto-style proof of stake," in *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, 2021, pp. 1–15.

Appendix A. Some Formal Details of the LiDO-DAG Model

Here we provide some formal details of the LiDO-DAG encoding in the artifact. See also the artifact README file which contains more extensive documentation.

In the Rocq formalization, the LiDO cache tree is encoded as a finite map from view numbers to *view descriptors* (Fig. 15). Each view descriptor is a succint representation of the cache nodes within a single view. The `client_view_pull_src` field is the *parent* field of ECache. It is a pair of numbers because we might have multiple MCaches in a single view (supported by LiDO formalization but not currently used). The first number is the parent view number, and the second number is the *version number* that is used to distinguish between multiple MCaches within that view. `client_view_mcaches` is a finite map from version numbers to method entries. In the LiDO-DAG model, each method entry must be an existing vertex ID in the DAG. Finally, `client_view_max_ccache` is the highest MCache version that has a corresponding CCache, i.e. the latest method entry that is successfully committed.

The unauthenticated DAG is formalized in `UDAG.v`. The basic definitions are in Fig. 16. Each vertex is represented by a record of type `UDAG_Vert`. The `udag_vert_round` field and the `udag_vert_builder` field specify the round number and the builder ID of the vertex, respectively. The `udag_vert_data` field contains the data block of this vertex. Its type is a user-specified parameter `dag_t`. The only assumption we make on `dag_t` is that its equality is decidable (so we can perform list deduplication). Finally, the `udag_vert_preds` field contains the list of predecessor references embedded in this vertex. Each reference is represented using the ID of the target vertex.

The DAG state consists of two finite maps. The `udag_verts` map is from vertex IDs to vertex records. The `udag_closure` map is from vertex IDs to the closure of each vertex (represented as a list of vertex IDs). The closure is defined on page 6. It is automatically computed when a vertex is added to the graph.

Appendix B. Further Details of the Liveness Proof

B.1. The Liveness Assumptions

Here we describe the assumptions we make on segmented traces. They can be classified into three groups. The first group concerns how the $curr\_round$ variable and the remaining time variable change over time:

- Each honest process has $curr\_round > 0$.
- Each local timer has remaining time $\leq 2\Delta$.
- Let r and t be the value of $curr\_round$ and local timer remaining time of process p at the end of τ_k. Let r' and t' be the corresponding values at the end of τ_{k+1}. Then either $r' = r$ and $t' = t - 1$, or $r' > r$ and $t' = 2$.
- If process p has created a vertex in round r, then $curr\_round \geq r$ for process p.
- If $(p, r) \in$ `mdag_timeouts`, then $r \leq curr\_round$ of process p. If $r = curr\_round$ then the remaining time of the local timer of process p must be 0.
- If $(p, r) \in$ `mdag_timeouts` at the end of τ_{k+1}, then either the timeout already occurred before the end of τ_k, or $r = curr\_round$ of process p at the end of τ_k, and the remaining time of the local timer of process p is 0 at the end of τ_k.
- If the remaining time of the local timer of process p is 0 at the end of τ_k, and $curr\_round$ does not increase between τ_k and τ_{k+1}, then $(p, r) \in$ `mdag_timeouts` by the end of τ_{k+1}, where r is the $curr\_round$ value of p at the end of τ_k.

The second group of assumptions model Proposition 2.

- If $curr\_round > 1$ for process p, then there exists a vertex of round $curr\_round - 1$ created by p.
- If $curr\_round > 1$ for process p, then there exists $2f + 1$ vertices in round $curr\_round - 1$.

The third group of assumptions are consequences of the partial synchrony assumption. Informally, we assume that after an honest process adds a vertex into the global DAG, within Δ all honest processes will receive that vertex, as well as any direct or indirect predecessors of it.

- If an honest process p is in round r at the end of τ_k, then by the end of τ_{k+1} every honest process p' is in some round $r' \geq r$. (Rationale: because they must have received $2f + 1$ vertices in round $r - 1$.)
- If every honest process p has created a vertex in round r by the end of τ_k, then by the end of τ_{k+1} every honest process is in some round $r' > r$. (Same rationale.)
- If the leader of round r is honest and has created a vertex v in round r by the end of τ_k, then by the end of τ_{k+1} every honest process must have received v, as indicated by the history variable `mdag_recv_leader_verts`.
- If the leader of round r is honest, and every honest process has created a supporter for a leader vertex of round r by the end of τ_k, then by the end of τ_{k+1} every honest process must have received $2f + 1$ supporters for that leader vertex, as indicated by the history variable `mdag_recv_cert`.
- If every honest process has created a certificate for some leader vertex v of round r by the end of τ_k, then $(r, v) \in$ `mcommit_commit` by the end of τ_{k+1}.

B.2. The Liveness Checkpoints

The 9 checkpoints for round r are:

1) GAV $= r + 1$ and GRT ≥ 2.
2) GAV $= r + 1$, GRT ≥ 1, and the leader of round r has created a vertex in round r.
3) GAV $= r + 1$ and every honest process has received a leader vertex of round r.
4) GAV $= r + 2$, GRT ≥ 2; each honest process either has received a leader vertex of round r, or is still in some round $r' \leq r + 1$ and will not time out in round $r + 1$ within Δ; at least $f + 1$ honest processes have created supporters for a leader vertex of round r.
5) GAV $= r + 2$, GRT ≥ 1, and every honest process has created a supporter vertex in round $r + 1$ for a leader vertex of round r.
6) GAV $\geq r + 3$; each honest process either has received $2f + 1$ supporters for a leader vertex of round r, or is still in some round $r' \leq r + 2$ and will not time out in round $r + 2$ within Δ; at least $f + 1$ honest processes have created certificates for a leader vertex of round r.
7) GAV $\geq r + 2$ and every honest process has received $2f + 1$ supporters for a leader vertex of round r.
8) Every honest process has created a certificate in round $r + 2$ for a leader vertex of round r.
9) A leader vertex of round r is committed.

B.3. Proof of Theorem 5

Let R be the GAV when GST commences. The liveness proof establishes that, every round $r \geq R$ with an honest leader will eventually reach checkpoint 6 or 7. Theorem 5 assumes that every process is honest, thus every round $r \geq R$ will reach checkpoint 6 or 7.

For each round r that reaches checkpoint 6 or 7, we can see that every honest process will eventually either jump over round $r + 2$, or create a certificate for a leader vertex in round r. The argument is similar to the proof of Lemma 4. Since every process is honest, this means every vertex in round $r + 2$ must be a certificate.

It remains to prove that after $GST + 6\Delta$, every new vertex is in some round $r \geq R + 2$. Note that since the timer is always reset to 2Δ, GAV should increase at least once every 3Δ. Thus after 6Δ no honest process should continue to create vertices in round $r < R + 2$.

Appendix C.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

The paper presents a counterexample to the liveness of the consensus protocol Mysticeti, which also affects existing protocol deployments. It then presents a mechanized safety and liveness proof for a fixed version of Mysticeti in the proof assistant Rocq.

C.2. Scientific Contributions

- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

C.3. Reasons for Acceptance

1) The paper shows that a previously found issue in Mysticeti's manual soundness proof indeed translates to a concrete liveness issue. With this, the paper identifies an impactful vulnerability since, to this point, it was unknown whether the identified issue in the proof indeed pointed to a liveness bug or if it could have been fixed by enhancing the proof. As Mysticeti is implemented in existing projects (e.g., the Sui and the IOTA blockchain), the identified liveness bug could affect the guarantees of such real-world deployments. This was acknowledged by the developers of the affected projects.
2) It is shown how the liveness of Mysticeti can be reestablished with a careful protocol adjustment. It is confirmed that the adjusted protocol satisfies both liveness and safety guarantees. The corresponding proof is mechanized in the proof assistant Rocq. With this, the paper provides a valuable step forward in the field, since it provides the first mechanized safety and liveness proof for the Mysticeti protocol. It is generally the first such proof for an unauthenticated DAG-based consensus protocol.