Sailfish: Towards Improving Latency of DAG-based BFT

Nibesh Shrestha[1], Aniket Kate[2], and Kartik Nayak[3]

[1]Supra Research, n.shrestha@supraoracles.com
[2]Supra Research/Purdue University, aniket@purdue.edu
[3]Duke University, kartik@cs.duke.edu

Abstract

Existing DAG-based BFT protocols exhibit long latency to commit decisions. The primary reason for such a long latency is having a *leader* every 2 or more "rounds". Even under honest leaders, these protocols require two or more reliable broadcast (RBC) instances to commit the proposal submitted by the leader (leader vertex), and additional RBCs to commit other proposals (non-leader vertices). In this work, we present Sailfish, the first DAG-based BFT that supports a leader vertex in each round. Under honest leaders, Sailfish maintains a commit latency of one RBC round plus 1δ to commit the leader vertex (where δ is the actual transmission latency of a message) and only an additional RBC round to commit non-leader vertices.

1 Introduction

Byzantine Fault Tolerant state machine replication (BFT SMR) protocols form the core underpinning for blockchains. At a high level, these protocols enable a group of parties to agree on a sequence of values, even if some of these parties are Byzantine (arbitrarily malicious). Owing to the need for efficient blockchains, there has been a lot of progress in improving the key efficiency metrics namely, latency, communication complexity, and throughput under various network conditions. Under the commonly assumed partial synchrony network, we know of protocols that can commit with a latency of 3δ (where δ represents the actual network delay) [10, 9, 16] and also achieve linear communication complexity [35, 25] under optimistic conditions (such as an honest leader).

Most of these protocol designs rely on a designated leader who is the party responsible for proposing transactions and driving the protocol forward while other parties agree on the proposed values and ensure that the leader keeps making progress. From an efficiency standpoint, this approach results in two key drawbacks. First, there is an unequal distribution of work among the parties. The leader is burdened with sending large amounts of data in its proposal, while others are only responsible for acknowledging/voting on these proposals. Second, and more importantly, there is an uneven scheduling of work among the parties. While the leader is sending a proposal, the other parties' processors and their network is not used. Thus, even if the former problem can be addressed by amortizing the work among parties over time, the problem with uneven usage of the resources still remains.

Several techniques proposed in the literature can potentially mitigate these concerns. These include the use of erasure coding techniques [28, 4] or the data availability committees [18, 19, 34] to disseminate the data more efficiently. Recently, a novel approach known as DAG-based BFT has emerged [6, 20, 23, 24, 31, 32, 13]. These protocols enable all participating parties to propose in parallel, maximizing bandwidth utilization and ensuring equitable distribution of workload. Consequently, these protocols have demonstrated improved throughput compared to the leader-based counterparts under moderate network sizes [31, 14]. However, all existing DAG-based protocols incur a high latency compared to their "leader-heavy" counterparts [10, 35, 21, 16, 25]. Is high latency inherent for such DAG-based protocols? Addressing this question is the key goal of this paper.

Table 1: **Comparison of DAG-based BFT protocols, after GST**

		RBC Used	LV Commit Latency	NLV Commit[(1)] Latency	Communication Complexity	Leader Frequency	NLV Latency [(2)] Under Failure
Bullshark	[31, 32]	Das et al. [15]	8δ	$+8\delta$	$O(n^3)$	1/2	$8\Delta + 8\delta$
Cordial Miners	[24]	None	6Δ	$+6\Delta$	$O(n^4)$	1/3	6Δ
Shoal	[30]	Das et al. [15]	8δ	$+4\delta$	$O(n^3)$	1	$8\Delta + 4\delta$
Sailfish		Das et al. [15]	5δ	$+4\delta$	$O(n^3)$	1	$8\Delta + 2\delta$
Sailfish		Abraham et al. [2]	3δ	$+2\delta$	$O(n^4)$	1	$4\Delta + 2\delta$

LV implies leader vertex. **NLV** implies non-leader vertex. We use the erasure-coded reliable broadcast from Das et al. [15] which incurs 4 communication steps and $O(n^2)$ communication complexity to propagate $O(n)$-sized message. Bullshark (and Shoal) can also use RBC protocol of Abraham et al. [2] to achieve a commit latency of 4δ for leader vertices and 4δ (2δ for Shoal) for non-leader vertices. (1) This column lists the additional latency to commit non-leader vertices that share a round with the leader vertex; the commit latency of these vertices is the maximum among non-leader vertices. (2) The column lists the increase in latency to commit non-leader vertices when a single Byzantine failure occurs between honest leaders.

In the following, we first discuss the core structure involved in a DAG-based protocol, then describe the latency of the state-of-the-art protocols compared to ours, and then explain the key challenges and our contributions.

Typical structure of DAG-based BFT. A DAG-based BFT progresses through a series of *rounds*. In each round r, each party makes a proposal, represented as a DAG vertex. The vertex includes references to at least $2f + 1$ vertices proposed in round $r - 1$ (where f is the maximum number of Byzantine faults). These references form the edges of the DAG. The edges and paths formed from these edges are used for committing vertices in the DAG. Many DAG-based protocols rely on a reliable broadcast protocol (RBC) [8] to disseminate the vertices; this ensures non-equivocation and guaranteed delivery [31, 30, 23]. Depending on whether a communication-optimal [15] or latency-optimal [2] RBC protocols are used, the RBC would incur a latency of 4δ and 2δ respectively.

Partially synchronous DAG-based protocols rely on designated parties called leaders to commit vertices. In these protocols, the vertices proposed by the leaders (leader vertices) are committed whereas non-leader vertices are ordered as part of the causal history of leader vertices.

Latency in state-of-the-art partially synchronous DAG-based BFT protocols. The state-of-the-art partially synchronous DAG-based protocols are Bullshark [31, 32], Shoal [30], and Cordial miners [24]. We elaborate on the results obtained by these protocols in Table 1.

In Bullshark, each round employs an RBC to disseminate the proposal, and a leader is assigned every 2 rounds. The round after the leader round serves to "vote" the leader vertex; hence called the voting round. Thus, committing the leader vertex requires two RBC rounds. On the other hand, non-leader vertices require a minimum of 4 RBCs.

A recent work, Shoal [30] proposed a "pseudo-pipelining" technique to support leaders in each round; aiming to reduce the commit latency of non-leader vertices. Their technique relies on executing multiple instances of the Bullshark-based protocol sequentially, ensuring that a leader vertex is present in every round. This approach relies on the observation that all parties agree on the first-ordered leader vertex, enabling the system to deterministically start a new protocol instance in the subsequent round. By doing so, Shoal achieves a pipelining effect as each instance starts with a leader vertex in its first round.

However, Shoal relies on Bullshark to commit some vertex before initiating a new instance. When Bullshark fails to commit, Shoal requires an additional two rounds to commit some vertex and start a new Bullshark instance. Furthermore, with an adversarial leader schedule alternating between Byzantine and honest leaders, Bullshark (and Shoal) fails to make progress. Consequently, Shoal's ability to ensure a leader vertex in each round is compromised to some extent. Moreover, Shoal inherits the latency of 2 RBCs to commit the leader vertex.

Similarly, Cordial Miners [24] aimed to improve the latency of DAG-based BFT protocols by using best-effort broadcast instead of RBC. However, their protocol necessitates parties to explicitly wait for a timeout (of at least 2Δ, where Δ is the known bound on message delay after *global stabilization time* (GST)) before advancing to the next round, indicating a lack of responsiveness [29]. Consequently, this results in a commit latency of at least 6Δ for the leader vertex (leader vertex requires 3 rounds to commit) and an additional 6Δ for non-leader vertices that coincide with the leader vertex (leader round repeats every 3 rounds). Furthermore, the communication complexity of their protocol blows to $O(n^4)$ per round in the presence of Byzantine faults (where n is the number of parties in the system).

To the best of our knowledge, no DAG-based protocol supports a leader vertex in each round in a true sense. Furthermore, all DAG-based BFT protocols require at least 2 RBCs to commit the leader vertex. In this work, we address these concerns and introduce Sailfish, the first DAG-based BFT protocol that achieves support for a leader vertex in each round while achieving a latency of 1RBC plus 1δ time to commit the leader vertex, along with an additional RBC to commit the non-leader vertices. When employing the optimal latency RBC [2], Sailfish incurs only 3δ to commit the leader vertex, effectively matching the best latency achieved by classical approaches [10]. When using a communication-optimal RBC [15], our protocol incurs 5δ latency to commit the leader vertex. Compared to the state-of-the-art DAG-based BFT, Sailfish improves the latency for committing leader vertices by at least 1δ (when using [2]) and 3δ time (when using [15]).

Challenges and Key Contributions

The key technical challenge. In DAG-based protocols, a crucial safety invariant that needs to be maintained is: when a round r leader vertex v_k is committed by an honest party, the leader vertex of any round $r' > r$ should have a path to v_k. In earlier protocols, v_k is committed when a sufficient ($f+1$ or more) round $r+1$ vertices have a path to v_k and the safety invariant is achieved by having a leader vertex in every two or more rounds. As a round $r+2$ vertex has paths to $2f+1$ round $r+1$ vertices, a round $r+2$ leader vertex will trivially have a path to v_k. Similarly, the leader vertex of round $r' > r+2$ will have a path to v_k. However, the round $r+1$ leader vertex lacks paths to other round $r+1$ vertices. Consequently, even if v_k is committed, the round $r+1$ leader vertex cannot establish a path to it via other round $r+1$ vertices. The only way it can form a path to v_k is by awaiting its delivery. However, waiting for v_k to be delivered poses liveness concerns. Alternatively, if the round $r+1$ leader vertex is proposed (after a timeout), it can lack a path to v_k even when other parties have committed v_k, violating the safety requirement. This is the key challenge when supporting a leader vertex in each round.

Towards having a leader vertex in each round. Our solution to the above challenge is simple. In our protocol, we mandate the round $r+1$ leader vertex to have a path to v_k or contain a proof that shows a sufficient number of honest parties did not vote for v_k. When such a proof exists, we can guarantee v_k cannot be committed; it is thus safe for the round $r+1$ leader vertex to lack a path to v_k.

The requirement for the round $r+1$ leader vertex to wait for v_k or the proof marginally increases the timeout duration a party has to wait in a round compared to existing protocols, potentially impacting latency under failures. To address this concern, we conduct a thorough analysis of the latency. Our analysis indicates that despite the increased timeout, our latencies outperform the state-of-the-art in the presence of a single Byzantine failure between honest leaders (see Table 1).

Towards reducing the commit latency to 1RBC plus 1δ for leader vertices. In a typical RBC protocol [8, 27, 15], the sender first sends its value to all other parties, followed by multiple rounds of message exchanges among the parties. When the sender is honest, the first value received from the sender is the value that is eventually delivered. We rely on this observation and decide based on the first received values of the round $r+1$ vertices, i.e., we do not require the RBC of round $r+1$ vertices to be delivered to commit the round r leader vertex. However, when the sender is faulty, the first value received from the sender can be different from the final delivered value. In order to account for such Byzantine behavior, our protocol commits the round r leader vertex only when $2f+1$ round-$(r+1)$ vertices have paths to the round r leader vertex. Out of the $2f+1$ first messages for the round $r+1$ vertices, at least $f+1$ are sent by honest parties which will be delivered by all honest parties.

This approach ensures the safety invariant while enabling our protocol to commit the leader vertex with a latency of 1 RBC plus 1δ, and an additional RBC to commit the non-leader vertices. We further note that this optimization is unique to our protocol and does not apply to the other protocols as it can cause liveness concerns. We provide the intuition behind this reasoning in detail in Section 3.

2 Preliminaries

We consider a system $\mathcal{P} := P_1, \ldots, P_n$ consisting of n parties out of which up to $f = \lfloor \frac{n-1}{3} \rfloor$ parties can be Byzantine. The Byzantine parties may behave arbitrarily. A party that is not faulty throughout the execution is considered to be *honest* and executes the protocol as specified.

We consider the partial synchrony model of Dwork et al. [17]. Under this model, the network starts in an initial state of asynchrony during which the adversary may arbitrarily delay messages sent by honest nodes. However, after an unknown time called the *Global Stabilization Time* (GST), the adversary must ensure that all messages sent by honest nodes are delivered to their intended recipients within Δ time of being sent. We use δ to characterize the actual (variable) transmission latencies of messages and observe that $\delta \leq \Delta$ after GST. Additionally, we assume the local clocks of the parties have *no clock drift* and *arbitrary clock skew.*

We make use of digital signatures and a public-key infrastructure (PKI) to prevent spoofing and replays and validate messages. We use $\langle x \rangle_i$ to denote a message x digitally signed by party P_i using its private key. We use $H(x)$ to denote the invocation of the hash function H on input x.

2.1 Building Blocks

Byzantine reliable broadcast. In a Byzantine reliable broadcast protocol (RBC), a designated sender P_k invokes $\text{r\_bcast}_k(m, r)$ to propagate its input m in some round $r \in \mathbb{N}$. Each party P_i then outputs the message m via $\text{r\_deliver}_i(m, r, P_k)$ where P_k is the designated sender and r is the round number in which sender P_k sent the message m. The reliable broadcast primitive satisfies the following properties:

- **Agreement.** If an honest party P_i outputs $\text{r\_deliver}_i(m, r, P_k)$, then every other honest party P_j eventually outputs $\text{r\_deliver}_j(m, r, P_k)$.
- **Integrity.** For every round $r \in \mathbb{N}$ and party $P_k \in \mathcal{P}$, an honest party P_i outputs r_deliver_i at most once regardless of m.
- **Validity.** If an honest party P_k calls $\text{r\_bcast}_k(m, r)$ then every honest party eventually outputs $\text{r\_deliver}(m, r, P_k)$.

2.2 Problem Definition

Following Bullshark [31], we focus on the Byzantine Atomic Broadcast (BAB) problem as defined below:

Definition 1 (Byzantine atomic broadcast [23, 31])**.** *Each honest party $P_i \in \mathcal{P}$ can call $a\_bcast_i(m, r)$ and output $a\_deliver_i(m, r, P_k)$, $P_k \in \mathcal{P}$. A Byzantine atomic broadcast protocol satisfies reliable broadcast properties (agreement, integrity, and validity) as well as:*

- ***Total order.*** *If an honest party P_i outputs $a\_deliver_i(m, r, P_k)$ before $a\_deliver_i(m', r', P_\ell)$, then no honest party P_j outputs $a\_deliver_j(m', r', P_\ell)$ before $a\_deliver_j(m, r, P_k)$.*

3 The Sailfish Protocol

In this section, we present Sailfish, a protocol that supports a leader vertex in each round and improves the latency to commit both leader and non-leader vertices. Specifically, Sailfish incurs one RBC, plus 1δ to commit the leader vertex and an additional RBC to commit the non-leader vertex. We first provide some basic preliminaries to ease the protocol description.

Local variables:

struct vertex v: ▷ The struct of a vertex in the DAG

$v.round$ - the round of v in the DAG

$v.source$ - the party that broadcast v

$v.block$ - a block of transactions

$v.strongEdges$ - a set of vertices in $v.round$ that represent strong edges

$v.weakEdges$ - a set of vertices in rounds $< v.round-1$ that represent weak edges

$v.nvc$ - a no-vote certificate for $v.round - 1$(if any)

$v.tc$ - a timeout certificate for $v.round - 1$ (if any)

$DAG_i[]-$ An array of sets of vertices (indexed by rounds)

$blocksToPropose$ - A queue, initially empty, P_i enqueues valid blocks of transactions from clients

$leaderStack \leftarrow$ initialize empty stack

procedure path(v, u) ▷ Check if exists a path consisting of strong and weak edges in the DAG
  return exists a sequence of $k \in \mathbb{N}$, vertices $v_1, \ldots, v_k$ s.t.
    $v_1 = v,\ v_k = u$, and $\forall j \in [2, .., k] : v_j \in \bigcup_{r\geq 1} DAG_i[r] \wedge (v_j \in v_{j-1}.weakEdges \cup v_{j-1}.strongEdges)$

procedure strong_path(v, u) ▷ Check if exists a path consisting of only strong edges from v to u in the DAG
  return exists a sequence of $k \in \mathbb{N}$, vertices $v_1, \ldots, v_k$ s.t.
    $v_1 = v,\ v_k = u$, and $\forall j \in [2, .., k] : v_j \in \bigcup_{r\geq 1} DAG_i[r] \wedge v_j \in v_{j-1}.strongEdges$

procedure set_weak_edges(v, r) ▷ Add edges to orphan vertices
  $v.weakEdges \leftarrow \{\}$
  for $r' = r - 2$ down to 1 **do**
    for every $u \in DAG_i[r']$ s.t. $\neg$path(v, u) **do**
      $v.weakEdges \leftarrow v.weakEdges \cup \{u\}$

procedure get_vertex(p, r)
  if $\exists v \in DAG_i[r]$ s.t. $v.source = p$ **then**
    return v
  return $\bot$

procedure get_vertex_leader(r)
  return get_vertex(L_r, r)

procedure broadcast_vertex(r)
  $v \leftarrow$ create_vertex(r)
  try_add_to_dag(v)
  r_bcast$_i(v, r)$

procedure a_bcast$_i(b, r)$
  $blocksToPropose$.enqueue(b)

procedure order_vertices()
  while $\neg leaderStack$.isEmpty() **do**
    $v \leftarrow leaderStack$.pop()
    $verticesToDeliver \leftarrow \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v, v') \wedge deliveredVertices\}$
    for every $v' \in verticesToDeliver$ in some deterministic order **do**
      output a_deliver$_i(v'.block, v'.round, v'.source)$
      $deliveredVertices \leftarrow deliveredVertices \cup \{v'\}$

Figure 1: Basic data structures for Sailfish. The utility functions are adapted from [23, 31].

Round based execution. Our protocol progresses through a sequence of numbered *rounds*. Rounds are numbered by non-negative integers starting with 1. Each round r consists of a designated leader, denoted by L_r, which is selected via a deterministic method based on the round number.

Basic data structures. We adopt the DAG construction protocol from Bullshark and modify it appropriately to fit our need. At a high level, the communication among parties is represented in the form of DAG. In each round, each party proposes a single vertex containing a (possibly empty) block of transactions along with references to at least $2f + 1$ vertices proposed in an earlier round. Those references serve as the edges in the DAG. The proposed vertices are propagated using reliable broadcast to ensure non-equivocation and guarantee all honest parties eventually deliver the proposed vertex.

The basic data structures and utilities for DAG construction are presented in Figure 1. Each party maintains a local copy of the DAG and different honest parties may observe different views of the DAG. However, due to the reliable broadcast of the vertices, each party will eventually converge on the same view of the DAG. The local view of DAG for party P_i is represented as DAG_i. Each vertex is associated with a unique round number and a unique sender (source). When P_i delivers a round r vertex, it is added to $DAG_i[r]$. $DAG_i[r]$ contains up to n vertices.

Each vertex consists of two sets of outgoing edges — strong edges and weak edges. The strong edges of round r vertex v consist of at least $2f + 1$ vertices from round $r - 1$ while the weak edges of the vertex consist of up to f vertices from rounds $< r - 1$ such that there is no path from v to these vertices. A path from vertex v_k to v_ℓ following the strong edges is called a strong path. Compared to Bullshark [31], we add two additional fields in the structure of the vertex – (i) $v.nvc$, which stores a no-vote certificate (consisting of a quorum of no-vote messages in a round), and (ii) $v.tc$, which store timeout certificate (consisting of a quorum of timeout messages in a round). We explain the purpose of these fields shortly.

DAG construction protocol. The DAG construction protocol is presented in Figure 2. In each round r, each party P_i proposes one vertex v. A round r vertex proposed by leader L_r is referred to as the round r vertex leader while the other round r vertices are non-leader vertices. In order to propose a vertex in a round r, P_i waits to receive at least $2f+1$ round $r-1$ vertices along with the round $r-1$ leader vertex until a timeout occurs in round $r - 1$. In the event that P_i receives $2f + 1$ round $r - 1$ along with round $r - 1$ leader vertex, P_i can immediately enter round r and propose a round r vertex. We note that including a reference to the round $r - 1$ leader vertex serves as the "vote" towards the round $r - 1$ leader vertex. These votes are later used to commit the leader vertex. Thus, waiting for the leader vertex until a timeout helps honest parties to vote for the leader vertex and helps commit the leader vertex with a small latency when the leader is honest (after GST).

If P_i did not receive the round $r-1$ leader vertex before the timeout, it multicasts $\langle \mathsf{timeout}, r-1\rangle_i$ to all other parties. In addition, an honest party P_j in round $r' \le r-1$ also multicasts $\langle \mathsf{timeout}, r-1\rangle_j$ messages if it receives $f+1$ distinct round $r-1$ timeout messages. Upon receiving $2f+1$ round $r-1$ timeout messages (denoted by $\mathcal{TC}_{r-1}$), P_i can enter round r and propose a round r vertex as long as it has received at least $2f + 1$ round $r - 1$ vertices. In our protocol, we require a round r vertex to either have a strong path to the round $r - 1$ leader vertex or include $\mathcal{TC}_{r-1}$ in $v.tc$. This is a constraint that we place on all vertices. We will clarify the purpose of this constraint shortly.

When P_i proposes a round r vertex without a strong path to the round $r-1$ leader vertex, it also sends a no-vote message to L_r indicating that P_i did not vote for round $r-1$ leader vertex. Upon entering round r, P_i starts a timer which is set to some τ time. We will shortly provide more details on the value of τ.

We place an additional constraint on the leader vertex. A round r leader vertex needs to either have a strong path to the round $r - 1$ leader vertex or contain $2f + 1$ round $r - 1$ no-vote messages (denoted by $\mathcal{NVC}_{r-1}$). The $\mathcal{NVC}_{r-1}$ serves as a proof that a quorum of parties did not "vote" for the round $r-1$ leader vertex. Hence, the round $r-1$ leader vertex cannot be committed and it is safe to lack a strong path to the round $r - 1$ leader vertex.

Upon delivering a round r vertex v, each party P_i checks if these constraints are met via is_valid(v) function. In particular, is_valid(v) checks whether v consists of either a strong path to round $r - 1$ leader vertex or $\mathcal{TC}_{r-1}$ (and $\mathcal{NVC}_{r-1}$ for the round r leader vertex). In addition, P_i also checks if vertex v consists of at least $2f + 1$ strong edges to round $r - 1$ vertices. Once these checks are satisfied, vertex v is added to $DAG_i[r]$ via try_add_to_dag(v) which succeeds when P_i has delivered all the vertices that have a path from vertex v in the DAG. If try_add_to_dag(v) fails, the vertex is added to the *buffer* for a later retry. In addition, when try_add_to_dag(v) succeeds, the vertices in the *buffer* are re-attempted to be added to the DAG_i.

Apart from advancing the rounds sequentially, our protocol supports honest parties in round $r' < r$ to "jump" to a higher round r when they observe $2f + 1$ round $r - 1$ vertices along with round $r - 1$ leader vertex or receive a $\mathcal{TC}_{r-1}$. If L_r is the lagging party, it additionally needs to wait to receive either $\mathcal{NVC}_{r-1}$ or round $r - 1$ leader vertex in order to propose round r leader vertex. When jumping rounds from r' to r, parties do not propose vertices between those rounds.

Committing and ordering the DAG. In our protocol, only the leader vertices are committed. The

Local variables:
$round \leftarrow 1$; $buffer \leftarrow \{\}$
29: **upon** r_deliver$_i(v, r, p)$ **do**
30: **if** $v.source = p \wedge v.round = r \wedge |v.StrongEdges| \geq 2f + 1 \wedge$ is_valid(v) **then**
31: **if** $\neg$try_to_add_to_dag(v) **then**
32: $buffer \leftarrow buffer \cup \{v\}$
33: **else**
34: **for** $v' \in buffer : v'.round \leq r$ **do**
35: try_to_add_to_dag(v')

36: **upon** timeout **do**
37: multicast $\langle \mathsf{timeout}, round \rangle_i$

38: **upon** receiving $f + 1$ distinct $\langle \mathsf{timeout}, r \rangle_*$ such that $r \geq round$ **do**
39: multicast $\langle \mathsf{timeout}, r \rangle_i$

40: **upon** receiving $\mathcal{TC}_r$ such that $r \geq round$ **do**
41: multicast $\mathcal{TC}_r$

42: **upon** $|DAG_i[r]| \geq 2f + 1 \wedge (\exists v' \in DAG_i[r] : v'.source = L_r \vee \mathcal{TC}_r$ is received) for $r \geq round$ **do**
43: advance_round$(r + 1)$

44: **procedure** create_new_vertex(r)
45: $v.round \leftarrow r$
46: $v.source \leftarrow P_i$
47: $v.block \leftarrow blocksToPropose$.dequeue()
48: $v.strongEdges \leftarrow DAG_i[r - 1]$
49: set_weak_edges(v, r)
50: **if** $\nexists v' \in DAG_i[r - 1] : v'.source = L_{r-1}$ **then**
51: $n.tc \leftarrow \mathcal{TC}_{r-1}$
52: **if** $P_i = L_r$ **then**
53: $v.nvc \leftarrow \mathcal{NVC}_{r-1}$
54: **return** v

55: **procedure** try_to_add_to_dag(v)
56: **if** $\forall v' \in v.strongEdges \cup v.weakEdges : v' \in \bigcup_{k \geq 1} DAG_i[k]$ **then**
57: $DAG_i[v.round] \leftarrow DAG_i[v.round] \cup \{v\}$
58: **if** $|DAG_i[v.round]| \geq 2f + 1$ **then**
59: try_commit$(v.round - 1, DAG_i[v.round])$
60: $buffer \leftarrow buffer \setminus \{v\}$
61: **return** true
62: **return** false

63: **procedure** advance_round(r)
64: **if** $\nexists v' \in DAG_i[r - 1]| : v'.source = L_{r-1}$ **then**
65: send $\langle \mathsf{no\text{-}vote}, r - 1 \rangle_i$ to L_r
66: **if** $P_i = L_r$ **then**
67: **wait until** $\exists v' \in DAG_i[r - 1] : v'.source = L_{r-1}$ or $\mathcal{NVC}_{r-1}$ is received
68: $round \leftarrow r$; *start timer*
69: broadcast_vertex$(round)$

Figure 2: Sailfish: DAG construction protocol for party P_i

non-leader vertices are ordered (in some deterministic order) as part of the causal history of a leader vertex when the leader vertex is (directly or indirectly) committed as shown in order_vertices function (see Line 22).

An honest party P_i directly commits a round r leader vertex v_k when it observes $2f + 1$ "first messages" (of the RBC) for round $r + 1$ vertices with strong paths to the round r leader vertex, i.e., P_i does not need to wait for the RBC of round $r + 1$ vertices to terminate. This is because when the sender of the RBC is honest, the first observed value (i.e., the first message of the RBC) is the value that will eventually be delivered. Among the $2f + 1$ round $r + 1$ vertices, at least $f + 1$ vertices are sent by honest parties which will eventually be delivered such that the delivered value is equal to the first received value (in the first message of RBC). This is sufficient to ensure $\mathcal{NVC}_r$ will not exist and any round $r' > r$ leader vertex (if it exists)

```
Local variables:
    committedRound ← 0
upon receiving a set S of ≥ 2f + 1 first messages for round r + 1 vertices do
    try_commit(r, S)

procedure try_commit(r, S)
    p ← get_vertex_leader(r)
    votes ← {v′ ∈ S | strong_path(v′, p)}
    if votes ≥ 2f + 1 then
        commit_leader(p)

procedure commit_leader(v)
    leaderStack.push(v)
    r ← v.round − 1
    v′ ← v
    while r > committedRound do
        v_s ← get_vertex_leader(r)
        if strong_path(v′, v_s) then
            leaderStack.push(v_s)
            v′ ← v_s
        r ← r − 1
    committedRound ← v.round
    order_vertices()
```

Figure 3: Sailfish: The commit rule for party P_i

will have strong paths to the round r leader vertex; thus ensuring the safety of a commit.

In addition to the above commit rule, our protocol also allows party P_i to directly commit a round r leader vertex v_k if it delivers (via RBC) $2f+1$ round $r+1$ vertices that have strong paths to v_k (see Line 59). This commit rule is helpful in scenarios when the RBC delivers a vertex without having received the first message of the RBC. Such scenarios arise when the sender of the RBC is faulty or during an asynchronous period.

Upon directly committing v_k in round r, P_i first indirectly commits leader vertices v_m in smaller rounds such that there exists a path from v_k to v_m (based on its local copy of the DAG) until it reaches a round $r' < r$ in which it previously directly committed a leader vertex. In this protocol, we ensure that when a round r leader vertex v_k is directly committed by some honest party, leader vertices for any round $r' > r$ have a strong path to v_k. This ensures v_k will be (directly or indirectly) committed by all honest parties.

Remark on timeout parameter τ. The value of timeout parameter τ depends on two factors (i) the underlying RBC primitive used to propagate the vertices, and (ii) how an honest party P_i entered round r.

Several RBC primitives [8, 2, 3, 27] have been proposed in the literature with various tradeoffs in communication complexity, number of steps required, setup assumptions, etc. For a comprehensive list of RBC primitives, we refer readers to the recent work by Alhaddad et al. [3]. The value of parameter τ should be long enough to ensure that when an honest party enters round r, it can deliver the round r leader vertex broadcast by an honest leader along with $2f+1$ round r vertices before its timeout occurs. In particular, when P_i enters round r, the parameter τ should accommodate the time it takes for other honest parties to enter the common round r, including L_r (if honest) and deliver their round r vertices before the timeout occurs for P_i.

The timeout parameter τ also depends on whether party P_i entered round r via $\mathcal{TC}_{r-1}$ or not. When $\mathcal{TC}_{r-1}$ exists and L_r does not deliver round $r-1$ leader vertex, L_r has to collect $\mathcal{NVC}_{r-1}$ before proposing a round r leader vertex which may require up to 2Δ time. Accordingly, party P_i has to wait for 2Δ additional time in round r when entering round r via $\mathcal{TC}_{r-1}$ compared to when it enters round r via receiving round $r-1$ leader vertex.

The RBC primitive of Das et al. [15] has 4 communication steps and delivers a value within 4Δ time (see Property 1). In addition, it also ensures that when an honest party delivers a value at time t, all honest parties deliver the value by $t+2\Delta$ (see Property 2). Accordingly, party P_i sets its parameter τ to 6Δ when it enters round r after delivering round $r-1$ leader vertex and to 8Δ when it enters round r via $\mathcal{TC}_{r-1}$. We note that different honest parties may set different values for τ depending on how they entered a round.

Intuition behind including a timeout certificate on the vertex. As mentioned above, we place a constraint on all the vertices: a valid round $r+1$ vertex should either have a strong path to round r leader vertex or include a $\mathcal{TC}_r$. This is to prevent Byzantine parties from driving the protocol too fast and prevent an honest leader vertex from getting directly committed (even after GST). Note that our protocol requires $2f+1$ round $r+1$ vertices with strong paths to round r leader vertex for the round r leader vertex to be directly committed. In addition, our protocol also supports parties to "jump" to a higher round $r' > r$ when they observe $2f+1$ round $r'-1$ vertices including the round $r'-1$ leader vertex or $\mathcal{TC}_{r'-1}$. If $\mathcal{TC}_r$ were not included in the vertex, the f Byzantine parties can propose round $r+1$ vertices without strong paths to the round r leader vertex. And, as soon as $f+1$ honest parties propose round r vertices (with strong paths to the round r leader vertex), the protocol can move to round $r+1$ while f honest parties are lagging behind in some lower round $r'' \leq r$. Relying on the same technique, the protocol can proceed to round $r' > r$. The adversary can then deliver $2f+1$ round r' vertices along with round r' leader vertex to the f lagging honest parties; causing them to enter round $r'+1$ such that these f lagging honest parties do not propose a round $r+1$ vertex. This prevents the round r leader vertex from being committed.

After GST, when L_r is honest, honest parties do not timeout in round r. Thus, Byzantine parties cannot propose round $r+1$ vertex without voting for the round r leader vertex. This ensures round r leader vertex gets directly committed.

Explicit round-synchronization. Our protocol consists of an explicit round-synchronization via multicasting of timeout messages and $\mathcal{TC}_r$ when L_r is faulty. This is to ensure all honest parties can receive $\mathcal{TC}_r$ and $2f+1$ round r vertices within 2Δ time and send $\langle \textsf{no-vote}, r\rangle$ to L_{r+1}. This allows L_{r+1} to collect a $\mathcal{NVC}_r$ in a timely manner and allows all honest parties to receive the round $r+1$ leader vertex before they timeout in round $r+1$.

3.1 Efficiency Analysis

Commit latencies. The commit latency of the leader vertex is the time taken to propagate round r vertices (via RBC), and one communication step required to receive the first messages for $2f+1$ round $r+1$ vertices i.e., one RBC, plus 1δ. When employing the RBC protocol due to Das et al. [15], the commit latency of the leader vertex is 5δ. The non-leader vertices require an additional RBC (i.e. 4δ) to be committed.

We note that the Bullshark (and Shoal) cannot support a commit with a latency with one RBC, plus 1δ. This is due to the following reasons. First, Bullshark waits for only $f+1$ round $r+1$ vertices with strong paths to round r leader vertex to commit the round r leader vertex. Out of these round $r+1$ vertices, up to f could be sent by Byzantine parties. If we rely only on the first received value of the RBC (based on the first message), the final delivered value could be different when its sender is faulty. In this case, the final delivered vertices may not have strong path to the round r leader vertex for up to f vertices. A single round $r+1$ vertex from an honest party with a strong path to the round r leader vertex is insufficient to ensure the safety of a commit. On the other hand, if Bullshark were to be modified to commit upon receiving $2f+1$ round $r+1$ vertices with strong paths to round r leader vertex, it may fail to commit any leader vertices. This is because Bullshark does not require a round $r+1$ vertex to include $\mathcal{TC}_r$ when it does not have a strong path to round r vertex leader. As explained above, this allows Byzantine parties to drive the protocol fast and prevent a commit (even after GST).

Latency analysis under failures. Note that τ of our protocol is 6Δ in the round following an honest leader and 8Δ in the round following a Byzantine leader. The additional timeout is required because the round r leader vertex needs to wait for $\mathcal{NVC}_{r-1}$ when L_{r-1} is faulty. In contrast, Bullshark (and Shoal) requires τ of 6Δ in all scenarios (when using the RBC primitive of Das et al. [15]).

Despite our protocol having a slightly larger τ compared to Bullshark (and Shoal), the commit latency does not worsen when a single Byzantine failure occurs between two honest leaders. This is because both our protocol and Bullshark (and Shoal) require honest parties to wait for 6Δ in the round corresponding to the Byzantine leader. In the subsequent round, the honest leader can obtain $\mathcal{NVC}$ and propose responsively, meaning the increased value of τ doesn't increase latency in practice (when messages arrive in $\delta < \Delta$ time). In fact, our protocol incurs less latency despite the need to wait for $\mathcal{TC}$ and $\mathcal{NVC}$, primarily due to having a leader every round and smaller commit latency.

As a concrete example, we consider the commit latency of the non-leader vertices of round $r-1$ when L_r is Byzantine and both L_{r-1} and L_{r+1} are honest. For both our protocol and Bullshark (and Shoal), honest parties need to wait for 6Δ time in round r. Let t be the time when the first honest party enters round r. Since honest parties may enter round r within 2Δ of each other, all honest parties receive $\mathcal{TC}_r$ by time $t + 8\Delta + \delta$ and L_{r+1} receives $\mathcal{NVC}_r$ by $t + 8\Delta + 2\delta$. As L_{r+1} is honest, its leader vertex can be committed in the next 5δ time; committing round $r-1$ non-leader vertices in $8\Delta + 11\delta$ time (compared to 9δ when L_r is honest.)

In the case of Bullshark (and Shoal), apart from 6Δ wait in round r, honest parties would need to wait for round $r+1$ vertices from some honest parties that entered round r late (since honest parties enter a round within 2Δ of each other). Moreover, in their case, the round $r+2$ leader vertex is the next vertex to be committed in round $r+3$. In total, the latency to commit round $r-1$ non-leader vertices is $8\Delta + 16\delta$ (compared to 12δ when L_r is honest, in the case of Shoal). Thus, under a single Byzantine failure between honest leaders, our protocol still performs better compared to both Bullshark and Shoal.

However, when there is a sequence of two or more bad leaders in between honest leaders, honest parties need to wait for τ of 8Δ time, and hence our protocol would slightly underperform compared to Bullshark (and Shoal) in terms of latency.

Communication complexity. The size of each vertex is $O(n)$ since it consists of references to up to n vertices and, may contain timeout certificate and no-vote certificate. The size of these certificates is $O(1)$ assuming threshold signatures [7] ($O(n)$ without threshold signatures). In each round, each party propagates a single vertex via RBC. The RBC protocol of Das et al. [15] incurs an optimal $O(n^2)$ communication to propagate $O(n)$-sized messages. Thus, the total communication complexity is $O(n^3)$ per round. Similarly, all-to-all multicast of timeout certificates incurs $O(n^2)$ communication assuming threshold signatures (or $O(n^3)$ without threshold signatures). Thus, the overall communication complexity is $O(n^3)$ per round.

We note that a single vertex can contain $O(n)$ transactions without increasing its size. This results in amortized linear communication complexity per round.

3.2 Security Analysis

We say that a *leader vertex v_i is committed directly* by party P_i if P_i invokes commit_leader(v_i). Similarly, we say that a *leader vertex v_j is committed indirectly* if it is added to *leaderStack* in Line 84. In addition, we say party P_i consecutively directly commit leader vertices v_k and $v_{k'}$ if P_i directly commits v_k and $v_{k'}$ in rounds r and r' respectively and does not directly commit any leader vertex between r and r'.

The following fact is immediate from using reliable broadcast to propagate a vertex v and waiting for the entire causal history of v to be added to the DAG before adding v.

Fact 1. *For every two honest parties P_i and P_j (i) for every round r, $\bigcup_{r' \leq r} DAG_i[r']$ is eventually equal to $\bigcup_{r' \leq r} DAG_j[r']$, (ii) at any given time t and round r, if $v \in DAG_i[r] \wedge v' \in DAG_j[r]$ s.t. $v.source = v'.source$, then $v = v'$. Moreover, for every round $r' < r$, if $v'' \in DAG_i[r']$ and there is a path from v to v'', then $v'' \in DAG_j[r']$ and there is a path from v' to v''.*

Claim 1. *If an honest party P_i directly commits a leader vertex v_k in round r, then for every leader vertex v_ℓ in round r' such that $r' > r$, there exists a strong path from v_ℓ to v_k.*

Proof. Since P_i directly committed v_k in round r, there exists a set $\mathcal{Q}$ of $2f+1$ vertices in $DAG_i[r+1]$ that included a reference to vertex v_k. Let $\mathcal{H} \subset \mathcal{Q}$ be the set of vertices proposed by honest parties in $\mathcal{Q}$. We complete the proof by showing the statement holds for any $r' > r$.

Case $r' = r+1$: If $v_\ell \in \mathcal{H}$, we are trivially done. Otherwise, the vertices in $\mathcal{H}$ are from round $r+1$ honest non-leader parties. When a round $r+1$ honest non-leader party P_i includes a reference to vertex leader v_k, it does not send a round r no-vote message. Since $|\mathcal{H}| \geq f+1$, by standard quorum intersection argument, $\mathcal{NVC}_r$ does not exist. Moreover, parties in $\mathcal{H}$ have delivered v_k. By Fact 1, L_{r+1} will eventually deliver v_k. Thus, if v_ℓ exists, it must include a reference to v_k and there exists a strong path from v_ℓ to v_k.

Case $r' > r+1$: Observe that a round $r+2$ vertex has a strong path to $2f+1$ round $r+1$ vertices. By standard quorum intersection, this includes at least $f+1$ vertices in $\mathcal{Q}$ which has a strong path to v_k. Thus, all-round $r+2$ vertices (including round $r+2$ leader vertex) have a strong path to v_k. Moreover, each round $r'' > r+2$ vertex has strong paths to at least $2f+1$ vertices in round $r''-1$. By transitivity, each vertex at round r'' has strong paths to at least $2f+1$ vertices in round $r+2$. This implies v_ℓ must have a strong path to v_k. □

Claim 2. *If an honest party P_i directly commits a leader vertex v_k in round r and an honest party P_j directly commits a leader vertex v_ℓ in round $r' \geq r$, then P_j (directly or indirectly) commits v_k in round r.*

Proof. If $r' = r$, by Fact 1, $v_k = v_\ell$ and we are trivially done. When $r' > r$, by Fact 1 and Claim 1, there exists a strong path from v_ℓ to v_k in DAG_j. By the code of commit_leader, after directly committing a leader vertex v_ℓ in round r', P_i tries to indirectly commit leader vertices v_m in smaller rounds such that there exists a path from v_ℓ to v_m until it reaches a round $r'' < r'$ in which it previously directly committed a leader vertex. If $r'' < r < r'$, party P_j will indirectly commit v_k in round r. Otherwise, by inductive argument and Claim 1, party P_j must have indirectly committed v_k when directly committing round r'' leader vertex. □

Claim 3. *Let v_k and v'_k be two leader vertices consecutively directly committed by a party P_i in rounds r_i and $r'_i > r_i$ respectively. Let v_ℓ and v'_ℓ be two leader vertices consecutively directly committed by party P_j in rounds r_j and $r'_j > r_j$ respectively. Then, P_i and P_j commits the same leaders between rounds $\max(r_i, r_j)$ and $\min(r'_i, r'_j)$ and in the same order.*

Proof. If $r'_i < r_j$ or $r'_j < r_i$, then there are no rounds between $\max(r_i, r_j)$ and $\min(r'_i, r'_j)$ and we are trivially done. Otherwise, assume wlog that $r_i \leq r_j \leq r'_i$. By Claim 2, both P_i and P_j will (directly or indirectly) commit the same leader in the round $\min(r'_i, r'_j)$. By the code of commit_leader, after (directly or indirectly) committing a leader vertex, parties try to indirectly commit leaders in smaller round numbers until they reach a round in which they previously directly committed a leader. Therefore, both P_i and P_j will indirectly commit all leaders from $\min(r'_i, r'_j)$ to $\max(r_i, r_j)$. Assume $\min(r'_i, r'_j) = r'_i$. By Fact 1, both DAG_i and DAG_j will contain v'_i and all vertices that have a path from v'_i in DAG_i. Due to deterministic code of commit_leader, both parties will commit the same leaders between rounds $\min(r'_i, r'_j)$ to $\max(r_i, r_j)$. □

By inductively applying Claim 3 between any two pairs of honest parties we obtain the following corollary.

Corollary 1. *Honest parties commit the same leaders in the same order.*

Lemma 1 (Total order)**.** *The protocol in Figures 1 to 3 satisfies Total order.*

Proof. By Corollary 1, honest parties commit the same leaders in the same order. By the code of order_vertices, parties iterate on the committed leaders according to their order and a_deliver all vertices in their causal history by a predefined deterministic rule. By Fact 1, all honest parties have the same causal history in their DAG for every committed leader. Thus, the lemma follows. □

Lemma 2 (Agreement)**.** *The protocol in Figures 1 to 3 satisfies Agreement.*

Proof. If an honest party P_i outputs a_deliver$_i(v_i.block, v_i.round, v_i.source)$, v_i must be in the causal history of some leader vertex v_k.

When party P_j eventually directly commits a leader vertex v_ℓ for round higher than $v_k.round$, by Lemma 1, P_j also commits v_k. By Fact 1, the causal histories of v_k in DAG_i and DAG_j are the same. Thus, when P_j orders the causal histories of v_k, it outputs a_deliver$_j(v_i.block, v_i.round, v_i.source)$. □

Lemma 3 (Integrity)**.** *The protocol in Figures 1 to 3 satisfies Integrity.*

Proof. An honest party P_i calls a_deliver$_i(v.block, v.round, v.source)$ only when vertex v is in DAG_i. The vertices in DAG_i are added with event r_deliver$_i(v, v.round, v.source)$. Therefore, the proof follows from the Integrity property of reliable broadcast. □

Validity. We rely on GST to prove validity. For RBC, we use the protocol from Das et al. [15] for its (nearly) optimal communication complexity. Their protocol requires 4 communication steps and satisfies the RBC properties at all times. After GST, it provides the following stronger guarantees:

Property 1. *Let t be a time after GST. If an honest party reliably broadcasts a message M at time t, all honest parties deliver M by time $t + 4\Delta$.*

Property 2. *Let t_g denote the GST. If an honest party delivers message M at time t, then all honest parties deliver M by time* $\max(t_g, t) + 2\Delta$*.*

Claim 4. *Let t_g denote the GST and P_i be the first honest party to enter round r. If P_i enters round r at time t via receiving round $r-1$ leader vertex, then all honest parties enter round r or higher by* $\max(t_g, t) + 2\Delta$*.*

Proof. Observe that P_i must have delivered $2f + 1$ round $r - 1$ vertices along with round $r - 1$ leader vertex by time t. By Property 2, all honest parties must have delivered $2f+1$ round $r-1$ vertices along with round $r-1$ leader vertex by $\max(t_g, t)+2\Delta$. Thus, all honest parties will enter round r by $\max(t_g, t)+2\Delta$ if they have not already entered a higher round. □

Claim 5. *Let t_g denote the GST and P_i be the first honest party to enter round r. If P_i enters round r at time t via $\mathcal{TC}_{r-1}$, then (i) all honest parties (except L_r when $P_i \neq L_r$) enter round r or higher by* $\max(t_g, t) + 2\Delta$*, and (ii) L_r (if honest and $P_i \neq L_r$) enters round r or higher by* $\max(t_g, t) + 4\Delta$*.*

Proof. Observe that P_i must have delivered $2f + 1$ round $r - 1$ vertices and received $\mathcal{TC}_{r-1}$ by time t. By Property 2, all honest parties must have delivered $2f + 1$ round $r - 1$ vertices by $\max(t_g, t) + 2\Delta$. In addition, P_i must have multicasted $\mathcal{TC}_{r-1}$ which arrives all honest parties by $\max(t_g, t)+\Delta$. Thus, all honest parties (except L_r when $P_i \neq L_r$) will enter round r by $\max(t_g, t) + 2\Delta$ if they have not already entered a higher round. This proves part (i) of the claim.

Observe that if no honest party delivered round $r-1$ leader vertex by $\max(t_g, t)+2\Delta$, all honest parties (including L_r) will send $\langle \textsf{no-vote}, r-1\rangle$ to L_r. Thus, L_r will receive $\mathcal{NVC}_{r-1}$ by time $\max(t_g, t)+3\Delta$. On the other hand, if at least one honest party delivered round $r-1$ leader vertex by $\max(t_g, t)+2\Delta$, by Property 2, L_r will deliver round $r-1$ leader vertex by $\max(t_g, t)+4\Delta$. Thus, L_r will enter round r by $\max(t_g, t)+4\Delta$ if it has not already entered a higher round. This proves part (ii) of the claim. □

Claim 6. *All honest parties keep entering increasing rounds.*

Proof. Suppose all honest parties are in round r or above. Let party P_i be in round r. If there exists an honest party P_j in round $r' > r$ at any time, then by Claim 4 and Claim 5, all honest parties will enter round r' or higher. Otherwise, all honest parties are in round r. Observe that all honest parties will r_broadcast round r vertex when entering round r. Thus, all honest parties will deliver $2f + 1$ round r vertices.

Observe that if no honest party delivered round r leader vertex, due to the timeout rule, all honest parties will multicast $\langle \textsf{timeout}, r\rangle$ and receive $\mathcal{TC}_r$. In addition, all honest parties will also send $\langle \textsf{no-vote}, r\rangle$ to L_{r+1} and L_{r+1} will receive $\mathcal{NVC}_{r-1}$. Thus, all honest parties will move to round $r + 1$. On the other hand, if at least one honest party has delivered round r leader vertex, by Fact 1, all honest parties will deliver the round r leader vertex. Having delivered $2f+1$ round r vertices and round r leader vertex, all honest parties will move to round $r + 1$. □

Claim 7. *If an honest party enters round r then at least $f + 1$ honest parties must have already entered $r - 1$.*

Proof. For an honest party to enter round r, it must have delivered $2f + 1$ round $r - 1$ vertices. At least $f + 1$ of those vertices are sent by honest parties while they were in round $r - 1$. Thus, $f + 1$ honest parties must have already entered $r - 1$. □

Claim 8. *If the first honest party to enter round r does so after GST and L_r is honest, then there exists at least $2f+1$ round $r+1$ vertices with strong paths to round r leader vertex.*

Proof. Let t be the time when the first honest party (say P_i) entered round r. Observe that no honest party sends $\langle \mathsf{timeout}, r\rangle$ before $t+8\Delta$ due to its round timer expiring. Accordingly, no honest party sends $\langle \mathsf{timeout}, r\rangle$ due to receiving $f+1$ $\langle \mathsf{timeout}, r\rangle$ before $t+8\Delta$. Thus, $\mathcal{TC}_r$ does not exist before $t+8\Delta$. In addition, by Claim 7, no honest party can enter a round greater than r until at least $f+1$ honest parties have entered r. Thus, no honest party sends a timeout message for a round greater than r before $t+8\Delta$ and no honest party enters a round greater than r via a timeout certificate before $t+8\Delta$.

Since, P_i entered round r at time t, by Claim 5, all honest parties (except L_r) enter round r or higher by $t+2\Delta$ and L_r enters round r or higher by $t+4\Delta$. Observe that if some honest party enters a round higher than $r+1$ before $t+8\Delta$, there exists at least $2f+1$ round $r+1$ vertices with strong paths to round r leader vertex (say v_k). This is because for an honest party to enter round r', it must have delivered $2f+1$ round $r'-1$ vertices. By transitive argument, it must be that there exists $2f+1$ round $r+1$ vertices. Since $\mathcal{TC}_r$ does not exist before $t+8\Delta$, the round $r+1$ vertices must have a strong path to round r to v_k.

Also, note that if an honest party enters round $r+1$ before $t+8\Delta$, it must have delivered $2f+1$ round r vertices and vertex v_k (since $\mathcal{TC}_r$ does not exist before $t+8\Delta$). Thus, its round $r+1$ vertex must have a strong path to v_k.

In the rest of the proof, we consider the case when no honest party entered a round higher than r before $t+8\Delta$. Thus, by Claim 5, all honest parties (except L_r) enter round r by $t+2\Delta$ and L_r enters round r by $t+4\Delta$. Note that an honest party r_broadcasts round r vertex when it enters round r. By Property 1, round r vertices from all honest parties (except L_r) will be delivered by $t+6\Delta$. In addition, by Property 1, v_k will be delivered by $t+8\Delta$. Thus, all honest parties will receive $2f+1$ round r vertices by $t+8\Delta$ along with v_k and send round $r+1$ vertex with a strong path to v_k. □

The above claim uses $\tau = 8\Delta$. When an honest party enters round r via receiving round $r-1$ leader vertex, by using Claim 4 (instead of Claim 5), we can show the above claim holds with $\tau = 6\Delta$. By the commit rule and Claim 8, the following corollary follows.

Corollary 2. *If the first honest party to enter round r does so after GST and L_r is honest, all honest parties will directly commit round r leader vertex.*

Lemma 4 (Validity)**.** *The protocol in Figures 1 to 3 satisfies Validity.*

Proof. Let party P_i be an honest party that invokes a_bcast(b, r). We show that all honest parties eventually output a_deliver(b, r, p_i). Observe that P_i pushes b into the *blocksToPropose* queue. By Claim 6, P_i keeps increasing rounds and creating new vertices in those new rounds. Thus, P_i will eventually create a vertex v_i with b at some round r and reliably broadcast it. By the Validity property of reliably broadcast, all honest parties will eventually add it to their DAG i.e., $v_i \in DAG[r]$ for every honest party. By the code of create_new_vertex, every vertex that P_j creates after v_i is added to $DAG_j[r]$ has a path to v_i.

By Corollary 2, the leader vertex proposed by an honest leader is directly committed after GST. With a leader-election function that elects all parties with equal probability, there will be an honest leader who will propose a vertex with a path to v_i and the leader vertex will be committed. By the code of order_vertices, P_j will eventually invoke a_deliver(b, r, p_i). By Lemma 2, all honest parties will eventually invoke a_deliver(b, r, p_i). □

4 Related Work

There has been an extensive body of research aimed at enhancing the performance of BFT consensus protocols. Recently, DAG-based BFT protocols have emerged as a means to enhance the throughput of BFT consensus protocols. We review the most recent and closely related works below. Compared to all these protocols, our protocols require one RBC, plus 1δ to commit the leader vertex and an additional RBC to commit the non-leader vertices. When employing the RBC protocol by Das et al. [15], our protocol requires

5δ to commit the leader vertex and an additional 4δ to commit the non-leader vertices. Moreover, our protocol maintains a communication complexity of $O(n^3)$ per round.

Asynchronous DAG-based BFT. Hashgraph [6] builds an unstructured DAG, with each vertex containing two references to previous vertices, and on top of the DAG, the parties run an inefficient binary agreement protocol. This leads to expected exponential time complexity. Aleph [20] is an asynchronous DAG-based BFT that builds a structured round-based DAG, where parties proceed to the next round once they receive $2f + 1$ DAG vertices from other parties in the same round. On top of the DAG construction protocol, an asynchronous binary agreement protocol decides on the order of vertices to commit; resulting in a higher commit latency.

DAG-Rider [23] is an asynchronous DAG-based BFT protocol. DAG-Rider progresses through waves where each wave consists of 4 rounds. There is a single leader in each wave and it requires an expected 6 rounds (i.e., 6 sequential RBCs) to commit the leader vertex. Since the non-leader vertices are ordered when the leader vertex is committed, they require an additional 4 rounds to commit the non-leader vertices that share a round with the leader vertex. Tusk [14] is an implementation based on DAG-Rider.

Very recently, GradedDAG [12] and LightDAG [11] improve the latency of asynchronous DAG-based BFT protocols by using weaker primitives such as consistent broadcast [33] instead of RBC. While the use of weaker primitives improves the latency in fault-free cases, they require parties to download missing vertices at a later point when failures occur, leading to an increase in latency.

Partially synchronous DAG-based BFT. Bullshark [31, 32] builds upon DAG-Rider to improve the commit latency during the synchronous period. It follows the same wave structure consisting of 4 rounds. The partially synchronous version of Bullshark has one leader every two rounds. It requires 2 RBCs to commit a leader vertex and an additional 2 RBCs to commit the non-leader vertices that share a round with the leader vertex. Furthermore, Bullshark relies on an honest leader to synchronize all parties post the GST, committing a vertex only after such synchronization. Consequently, it demands two honest leaders to successfully commit a vertex after GST, leading to latency issues in case of frequent transitions between synchrony and asynchrony in the network. In contrast, our protocol has explicit round synchronization and supports commit with a single honest leader after GST.

Shoal [30] proposed a pseudo-pipelining approach to reduce the latency of non-leader vertices in Bullshark-based consensus protocols. In their protocol, they execute multiple instances of the Bullshark-based protocol sequentially, ensuring that a leader vertex is present in every round. However, their protocol relies on an instance of Bullshark to commit some vertex before initiating a new instance with a leader in the next round. With an adversarial leader schedule alternating between Byzantine and honest leaders, Bullshark (and Shoal) fails to make progress. Consequently, Shoal's ability to ensure a leader vertex in each round is compromised. Furthermore, Shoal inherits the latency of 2 RBCs to commit leader vertices.

In a recent work, Cordial Miners [24] proposed a DAG-based BFT protocol by using best-effort-broadcast instead of RBC to propagate the vertices in order to improve the latency. In their protocol, each party P_i sends references to at least $2f+1$ round $r-1$ vertices in the round r vertex. In the next round, each honest party P_j sends vertices "not seen" by party P_i to party P_i. Thus, honest parties will need to send $O(n)$ blocks in each round when the Byzantine parties "selectively" send their blocks to only some honest parties. Moreover, the Byzantine parties can always send their round r vertices without strong paths to up to f round $r-1$ vertices sent by honest parties. This causes the honest parties to send the missing vertices to the Byzantine parties. Thus, their communication complexity is always $O(n^4)$ per round with f Byzantine failures. Moreover, their protocol requires honest parties to "wait" for a timeout before moving to higher round in order to receive honest block proposals in each round. This is to ensure honest parties do not need to forward honest vertices at a later round. In order to ensure timely delivery of the messages, each round has to be at least 2Δ [1]; this results in a commit latency of at least 6Δ for leader vertices and an additional 6Δ to commit non-leader vertices that share a round with the leader vertices.

Mysticeti [5] introduces fast path commit for account based transactions and adds support to commit multiple leaders from a single round. Their protocol requires 3 RBCs to commit a leader vertex. Moreover, leader rounds occur every four rounds. This further increases the commit latency for non-leader vertices.

We also note two recent works, BBCA-chain [26] and Motorway [22] that focus on improving the throughput of chain-based BFT protocols by enabling all parties to propose in each round. In BBCA-chain, non-leader parties propose their blocks using best-effort-broadcast, and the leader incorporates these non-leader blocks in its proposal. In Motorway, all parties additionally acquire data availability certificates (acknowledgments from $f+1$ parties) for their proposed blocks with the leader including n data availability certificates in its block proposal. In both schemes, the leader is responsible for propagating $O(n)$ proposals when the Byzantine parties "selectively" send their proposals only to the leader. When the size of each proposal is $O(n)$ bits (which is typically the case with DAG-based BFT), the leader is responsible to disseminate $O(n^2)$ bits; imposing an heavier burden on the leader i.e., these protocol do not have equal distribution of work. In comparison, in our protocol (and DAG-based BFT protocols in general), each party is responsible for performing the same amount of work.

References

[1] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected o (1) rounds, expected communication, and optimal resilience. In *International Conference on Financial Cryptography and Data Security*, pages 320–334. Springer, 2019.

[2] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 331–341, 2021.

[3] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Balanced byzantine reliable broadcast with near-optimal communication and improved computation. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 399–417, 2022.

[4] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Brief announcement: Asynchronous verifiable information dispersal with near-optimal communication. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 418–420, 2022.

[5] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. Mysticeti: Low-latency dag consensus with fast commit path. *arXiv preprint arXiv:2310.14821*, 2023.

[6] Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep*, 34:9–11, 2016.

[7] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *Journal of cryptology*, 17:297–319, 2004.

[8] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[9] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet computer consensus. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 81–91, 2022.

[10] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, number 1999 in 99, pages 173–186, 1999.

[11] Xiaohai Dai, Guanxiong Wang, Jiang Xiao, Zhengxuan Guo, Rui Hao, Xia Xie, and Hai Jin. Lightdag: A low-latency dag-based bft consensus through lightweight broadcast. *Cryptology ePrint Archive*, 2024.

[12] Xiaohai Dai, Zhaonan Zhang, Jiang Xiao, Jingtao Yue, Xia Xie, and Hai Jin. Gradeddag: An asynchronous dag-based bft consensus with lower latency. *Cryptology ePrint Archive*, 2024.

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

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

[15] Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2721, 2021.

[16] Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. Moonshot: Optimizing chain-based rotating leader bft via optimistic proposals. *arXiv preprint arXiv:2401.01791*, 2024.

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

[18] EigenLabs. Intro to eigenda: Hyperscale data availability for rollups, 2023. Accessed on March 20, 2024.

[19] Ethereum. Data availability — ethereum.org, 2024. Accessed on March 20, 2024.

[20] Adam Gκagol, Damian Leśniak, Damian Straszak, and Michał Świκetek. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 214–228, 2019.

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

[22] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Motorway: Seamless high speed bft. *arXiv preprint arXiv:2401.10369*, 2024.

[23] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021.

[24] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. Cordial miners: Fast and efficient consensus for every eventuality. In *37th International Symposium on Distributed Computing (DISC 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

[25] Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

[26] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. Bbca-chain: One-message, low latency bft consensus on a dag. *arXiv preprint arXiv:2310.06335*, 2023.

[27] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

[28] Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. *arXiv preprint arXiv:2002.11321*, 2020.

[29] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *31 International Symposium on Distributed Computing*, page 6, 2017.

[30] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness. *arXiv preprint arXiv:2306.03058*, 2023.

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

[32] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: The partially synchronous version. *arXiv preprint arXiv:2209.05633*, 2022.

[33] TK Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94, 1987.

[34] Espresso Systems. Designing the espresso sequencer: Combining hotshot consensus with tiramisu da - hackmd, 2023. Accessed on March 20, 2024.

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